УНИВЕРЗИТЕТ У НОВОМ САДУ
ПРИРОДНО-МАТЕМАТИЧКИ ФАКУЛТЕТ
ДЕПАРТМАН ЗА МАТЕМАТИКУ И ИНФОРМАТИКУ

Јелена Текић

ОПТИМИЗАЦИЈА CFD СИМУЛАЦИЈЕ НА ГРУПАМА ВИШЕЈЕЗГАРНИХ ХЕТЕРОГЕНИХ АРХИТЕКТУРА

- докторска дисертација -

Нови Сад, 2019. године

Предговор

Предмет истраживања дисертације је примена паралелног програмирања за побољшање перформанси имплементација нумеричких метода које се користе у научне сврхе.

Под конкурентним (паралелним) извршавањем подразумева се извршавање више различитих операција истовремено, односно једне исте операције истовремено над различитим подацима. Изменама у дизајну архитектуре прво централних процесорских јединица, а касније и графичких картица, паралелно програмирање постало је широко доступно. *CFD* (*Computational Fluid Dynamic*) симулације описују механику флуида користећи нумеричке методе које захтевају извршавање сложених прорачуна. Паралелизација *CFD* симулација доводи до ефикаснијег решавања проблема из области динамике флуида. Једна од метода која се користи за решавање *CFD* проблема, а веома је погодна за паралелизацију, је *Lattice Boltzmann* метода.

Основни циљ истраживања приказаног у докторској дисертацији је оптимизација имплементације *Lattice Boltzmann* методе (*LBM*) у складу са *OpenCL* стандардом, развојем алгоритама који ће омогућити истовремено извршавање методе на свим доступним вишејезгарним хетерогеним уређајима. Алгоритми развијени у оквиру истраживања пружају могућност да се перформансе доступног хардвера максимално искористе. Поред оптимизације циљ дисертације је и развој микросервис архитектуре која ће бити платформски независна и поједноставити приступ симулацији (решењу), извршавање симулације и приказ резултата симулације.

Дисертација се састоји из следећих поглавља:

1. Увод
2. Основни концепти паралелног програмирања
3. Преглед технологија и алата коришћених за развој апликације за симулацију динамике флуида
4. Алгоритми за паралелизацију *Lattice Boltzmann* методе
5. Приказ и анализа резултата извршавања
6. Имплементација солвера у микросервис архитектури
7. Закључак

У првом поглављу укратко су представљени паралелно програмирање и динамика флуида. Описана је мотивација за рад на тези, дефинисан је циљ докторске дисертације и описан допринос који је резултат изведених истраживања. Такође, дат је преглед досадашњих истраживања везаних за проблем који се обрађује у оквиру дисертације.

Друго поглавље доноси кратак преглед историје развоја паралелног програмирања и опис основних концепата паралелног програмирања на вишејезгарним архитектурама и кластерима.

У трећем поглављу описане су технологије и алати који су коришћени за имплементацију решења. Описан је *OpenCL* стандард који је искоришћен за паралелизацију симулације на више хетреогених вишејезгарних уређаја. Затим је дат кратак преглед микросервиса, *SpringBoot*-а и *ReactJS*-а који су искоришћени да се постигне флексибилност и једноставнија употреба апликације.

Четврто поглавље описује *Lattice Boltzmann* методу и детаљно описује све алгоритме који могу бити коришћени за имплементацију проблема на више вишејезгарних уређаја. Уз сваки алгоритам дато је и образложење у којим случајевима је његова примена најефикаснија.

У петом поглављу дата је детаљна анализа извршавања различитих имплементација на различитим конфигурацијама и указано је на предности одређених алгоритама за сваку конфигурацију. Приказана је и верификација резултата у односу на апсолутне резултате наведене у литератури.

Шесто поглавље описује имплементацију апликације креиране у микросервис архитектури. Коришћење микросервис архитектуре треба

да омогући лаку употребу, лако проширивање апликације за друге врсте проблема и могућност лаке употребе апликације од стране трећих апликација.

На крају, у седмом поглављу дат је закључак дисертације и дати су предлози за могућа даља истраживања.

Захваљујем се Институ за Нумеричку Математику Марчука Руске Академије Наука који ми је омогућио приступ и рад на њиховој опреми, кластеру Тесла графичких картица.

Захваљујем се Природно-математичком факултету у Новом Саду који ми је омогућио приступ AXIOM кластеру и тестирање имплементираних алгоритама на NVIDIA графичким картицама AXIOM кластера.

Захваљујем се Комисији чије су сугестије, дате након детаљног прегледа рада, утицале на квалитет и коначни садржај дисертације.

Посебно се захваљујем ментору професору Милошу Рацковићу на уложеном труду, времену, подршци и конструктивним саветима који су имали кључни утицај на композицију и квалитет рада.

Захваљујем се својој породици на подршци и разумевању.

Садржај

1. Увод

Паралелно програмирање омогућава решавање проблема коришћењем паралелизма података или паралелизма процеса. Паралелизам података подразумева извршавање истих инструкција истовремено над различитим подацима. Паралелизам процеса представља растављање проблема на више независних делова који се решавају симултано, коначно решење проблема код овог типа паралелизације добија се комбиновањем резултата који су добијени извршавањем независних делова проблема. Паралелно програмирање користи се за решавање многобројних комплексних проблема у астрономији, метеорологији, авио индустрији, аутомобилској индустрији, *blockchain* технологијама и слично. Интересовање за паралелно програмирање појављује се касних педесетих година двадесетог века и уско је повезано са развојем првих супер рачунара. Осамдесетих година прошлог века са појавом масовно паралелних процесора долази до великог побољшања перформанси рачунара који се користе за паралелно програмирање. Развојем кластера долази до смањења цене рачунара који се могу користити за паралелно рачунање и њихове шире употребе (Parallel Computing: Background; Supercomputer, 2019).

Од 2000. године под утицајем убрзаног развоја индустрије видео игара долази до промена у архитектури графичких картица и појаве *many-core* архитектура. Видео игре захтевају нумеричку обраду великог броја података да би се формирао графички излаз који представљаја сложене тродимензионалне мреже. Пошто истраживачки прорачуни захтевају сличне операције *many-core* архитектуре почињу да се користе у истраживачке сврхе. Након појаве *many-core* архитектура сложене нумеричке методе могу се решавати и помоћу персоналних рачунара, што за последицу има да перформансе сличне онима које имају супер рачунари буду лакше доступне истраживачима који се баве нумеричким проблемима који захтевају велики број прорачуна у секунди. Нумеричке методе и њихове компјутерске симулације постају све популарније за проучавање и појављује се велики број публикованих резулата из ових тема. Због специфичне архитектуре *many-core* уређаја, код (алгоритма) који ће бити извршаван на њима мора бити паралелизован. Да би произвођачи омогућили

максималну искористивост својих уређаја праве своје моделе за програмирање уређаја које производе (уређај сваког произвођача има специфичну архитектуру) (Akhter et al., 2006; Central processing unit, 2019; Multi-core processor, 2019; Persson Mattsson, 2014; Vajda, 2011). У научним истраживањима до сада су за извршавање прорачуна најчешће коришћени *NVIDIA*-ини уређаји и *NVIDIA*-ин програмски модел *CUDA*. Појава *OpenCL* стандарада омогућила је развој кода за научне прорачуне који се може извршавати на хетерогеним уређајима различитих произвођача (Cheng et al., 2014; David et al., 2015; Sanders et al., 2011).

Динамика флуида је једна од области научних истаживања која је веома погодна за решавање помоћу рачунарских симулација. Динамика флуида бави се протоком флуида и има широку примену. Неке од примена су: прорачунавање сила и момената приликом кретања авиона, израчунавање брзине протока нафте унутар цевовода, у метеорологији за предвиђање временских прилика и слично. Симулације из области динамике флуида су веома захтевне па се за проучавање ове области најчешће користе супер рачунари. Развојем нових архитектура омогућено је да се симулације извршавају на персоналним рачунарима, чиме је проучавање области динамике флуида лакше доступно све већем броју истраживача.

Постоји више нумеричких метода које се могу користити за решавање проблема из области динамике флуида. Једна од метода која се данас све више користи је *Lattice Boltzmann* метода. *Lattice Boltzmann* метода (*LBM*) користи функције расподеле честица за представљање заједничког понашања молекула флуида. Карактеристике *LBM* методе су: једноставно формулисање граничних услова, погодна за програмирање и погодна за паралелизацију (Mohamad, 2007).

Архитектура микросервиса омогућава креирање апликација које су лако проширive и независне, а за комуникацију користе *HTPP* (*Hypertext transfer protocol*) и *REST* (*Representational State Transfer*) (Dragoni et al. 2017; Mazzara et al. 2016; Shadija et al., 2017; Sharma, 2017). *Java Script* пружа могућност брзог креирања динамичких и интерактивних *web* страница које корисницима треба да омогуће једноставно коришћење апликација, а *java script framework* библиотеке

као што је *ReactJS* омогућавају убрзан развој и додатне функционалности апликације. *JAVA* програмски језик омогућава креирање платформски независних апликација. *OpenCL* стандард је платформски независан и пружа могућност максималног коришћења свог доступног hardvera. *OpenCL* може се користити на супер рачунарима, кластерима, персоналним рачунарима, *CPU* и *GPU* уређајима. Комбиновање наведених технологија и алата омогућава креирање апликација које су: платформски независне, омогућавају потпуно коришћење доступног хардвера, лако су проширive и једноставне за коришћење.

1.1. Мотивација, циљ истраживања и допринос истраживања

Област динамике флуида захтева компликоване прорачуне за решавање нумеричких метода. Једна од метода која се у литератури врло често користи за решавање проблема из области динамике флуида је *Lattice Boltzmann* метода. Ова метода веома је погодна за паралелизацију, али су у литератури до сад пручаване само имплементације које за паралелизацију на више уређаја истовремено, користе *CUDA* стандард. Коришћење *CUDA* стандарада везује истраживаче за рад на *NVIDIA* уређајима. Појава *OpenCL* стандарда пружа могућност да се имплементирана решења користе на уређајима различитих произвођача.

Мотивација истраживања представљеног у тези је креирање имплементације коју би пре свега могли да користе истраживачи који раде на персоналним рачунарима различитих произвођача и да извршавањем апликације на опреми која им је доступна постигну максималне перформансе за ту опрему. Коришћењем свих ресурса доступног хардвера *CPU* и *GPU* уређаја истовремено истраживачи могу убрзати и олакшати своја истраживања у области динамике флуида.

Циљ истраживања дисертације је израда портабилне и флексибилне *benchmark* симулације за кретање флуида у шупљини коришћењем *Lattice Boltzmann* методе. Симулација ће се извршавати истовремено на свим уређајима који су доступни у оквиру једног хардвера и пружити могућност да се оптимално искористи постојећи

хардвер. Портабилност и флексибилност апликације биће обезбеђена коришћењем *OpenCL* стандарда, *JAVA* програмског језика и микросервис архитектуром.

Научни допринос тезе је убрзавање извршавања *Lattice Boltzmann* методе проучавањем и применом различитих алгоритама за паралелизацију методе истовременим коришћењем више хетерогених вишејезгарних уређаја и проучавање коришћења различитих структура података за пренос података на уређаје за паралелно рачунање. Анализом више алгоритама на различитим конфигурацијама показано је како се симулација *LBM* оптимално може убрзати када се истовремено користи више уређаја и различите специфичности *OpenCL* стандарда.

Детаљна анализа алгоритама и резултата извршавања на различитим конфигурацијама дата је у поглављима 4 и 5. Анализирање предности различитих имплементација уз коришћење специфичности *OpenCL* стандарда допринело је да се уређаји на којима се извршава симулација максимално искористе, уз очување могућности примене на разним хардвер и софтвер архитектурама. Резултати су верификовани на различитим конфигурацијама, различитих проивођача што је приказано у поглављу 5.

Креирање микросервиса описано је у поглављу 6. Микросервис архитектура пружа могућност лаког проширивања апликације за рад са другим примерима из динамике флуида (на пример: додавање разних препрека, укључивање темепратурних параметара и слично). Израдом *web* апликације постигнута је интерактивност и лака проширивост апликације. Омогућено је једноставније прикупљање и једноставнија анализа резултата извршавања симулације.

Треба посебно истаћи да циљ тезе није имплементација која ће омогућити максимално апсолутно убрзање у односу на постојеће симулације, већ имплементација која ће пружити могућност убрзања симулација на персоналним рачунарима са хардвером различитих произвођача. Анализирани резултати и имплементирана апликација омогућиће бржи рад и лакшу примену *Lattice Boltzmann* методе за проучавање области динамике флуида. Такође може се очекивати да

описани резултати помогну у побољшању перформанси паралелизције других нумеричких метода на персоналним рачунарима.

1.2. Преглед досадашњих истраживања

У наставку је дат кратак преглед досадашњих истраживања везаних за имплементацију *LBM* на *multy core, many core* архитектурама и кластерима.

LBM алгоритам за имплементацију динамике флуида веома је погодан за паралелизацију са релативно једноставном структуром *kernel* делова кода. Паралелизација овог алгоритма може да постигне значајан пораст перформанси у односу на секвенцијалну имплементацију, али пружа могућност додатног значајног побољшања перформанси паралелизацијом на више вишејезгарних уређаја (Pohl et al., 2004; Vidal et al., 2010; Williams et al., 2008).

Li и сарадници (Li et al., 2003) су 2003. године имплементирали LB методу на графичкој картици користећи растеризацију и frame buffer операције. 2008. године Tolke (Tölke, 2008) имплементира *LB* методу за ток флуида кроз цев са препрекама помоћу *CUDA* програмског модела користећи графичке картице. Проблем пропагације у имплементацији решен је коришћењем блокова дељене меморије величине *16 x float*. Резултати извршавања показују да имеплементација на *GPU* уређају постиже за ред величине боље резултате у односу на имплементацију на *CPU* уређају. Исте године Tolke у сарадњи са Krafczyk-ом (Tolke and Krafczyk, 2008) имплементира алгоритам помоћу *CUDA*-е за тродимензионални *LB* проблем користећи *D3Q19* модел.

Peng и сарадници (Peng et al., 2008) имплементирали су алгоритам за решавање *LB* методе користећи *CUDA* програмски модел на кластеру *Sony PlayStation3* и показали да су перформансе *PlayStation3* боље од *Xeon* кластера и да се разлика у перформансама повећава са порастом величине тестиране мреже, па за највећу мрежу *PlayStation3* има једанаест пута боље перформансе у односу на *Xeon* кластер. Побољшање перформанси за ред величине на графичким картицама у односу на процесоре показали су и Arie и сарадници (Arie and et al., 2009). Bailey и сарадници (Bailey et al., 2009) су

имплементирали алгоритам за *LBM* на *GPU* користећи модел *D3Q19* и показали да се перформансе могу побољшати за 20% повећањем заузетости мулти процесора графичке картице и коришћењем *space* методе складиштења која доводи до смањења захтева за *RAM* меморијом графичке картице. Bernaschi и сарадници (Bernaschi et al., 2009) показали су значајно повећање перформанси за имплементације на *GPU* за више различитих *LB* модела вишекомпонентних комплексних флуида у односу на имплементације за *CPU*. Aksnes и Elster (Aksnes and Elster, 2009) су показали да се паралелизацијом алгоритма на *GPU* време извршавања симулације за пермеабилност порозних стена може свести на пар минута у односу на пар сати, колико за извршавање симулације захтева серијски код извршаван на *CPU*. Kuznik и сарадници (Kuznik et al., 2010) су поредили имлементације *LB* метода које користе *single-float* и *double-float* прецизност и закључили да имплементација која користи *single-float* прецизност даје задовољавајуће резултате. Док су McClure и сарадници (McClure et al., 2010) поредили имплементације *LB* методе које користе *BGK* (*Bhatnagar–Gross–Krook*) схему и *MRT* (*Multiple-Relaxation-Time*) схему и показали да за извршавање симулације на графичким картицама *MRT* схема доноси бројне предности. Massimo и сарадници (Massimo et al., 2010) су имплементирали једно-фазну, мулти-фазну и мулти-компонентну *LBM* на кластеру који се састоји од више *GPU* уређаја користећи *CUDA* програмски модел и *OpenMP*. Xian and Takayuki (Xian and Takayuki, 2011) описују *CUDA* имплементацију тока око сфере користећи *MRT* схему и модел *D3Q19*. Паралелизам кода заснива се на *MPI* библиотеци. Постигнуто је смањење времена које је потребно за комуникацију помоћу расподеле домена или коришћењем више различитих процесних токова за рачунање и комуникацију. За тестирање коришћен је супер рачунар који се састоји од 170 чворова и 680 Тесла графичких картица.

Obrecht и сарадници (Obrecht et al., 2013) су имплементирали *LB* методу помоћу *CUDA*-е користећи *MRT* схему и модел *D3Q19* на шест Тесла графичких картица. Показали су да се на две Тесла графичке картице повезане у оквиру једног чвора кластера постиже убрзање од 86% у односу на имплементацију која користи само једну Тесла графичку картицу. Решење је засновано на коришћењу *POSIX* нити за сваки *CUDA* уређај (*CUDA context*) и додели дела мреже сваком

појединачном уређају. Имплементација пружа могућност коришћења великих мрежа, симулацију нестабилних токова флуида и коришћење великих Рејнолдсови бројева.

Chang и сарадници (Chang et al., 2013; Chang et al., 2013) су у два рада описали *cavity flow* (ток у шупљини) за различите промере дубине и различите *Reynolds*-ове бројеве користећи *D3Q19* модел и *MRT* схему. Симулација је паралелизована помоћу *OpenMP* методе. Тестирање алгоритма извршено је на једном чвору кластера који се састоји од 3 *NVIDIA* M2070 графичке картице или три *NVIDIA* GTX560 графичке картице. Показано је да брзина извршавања симулације зависи од величине мреже, односно да се симулација ефикасније извршава на већим мрежама. Највећа тестирана мрежа (2403 тачаке на једној оси) постигла је највеће убрзање, за ову мрежу симулација се извршава 159 пута брже када се користе три уређаја у односу на извршавање симулације на једном уређају. Такође, упоређене су брзине извршавања за single и double прецизности израчунавања и показано да коришћење double прецизности доводи до малог пада брзине извршавања.

Huang и сарадници (Huang et al., 2014) су имлементирали симулацију тока флуида кроз порозну средину на више *GPU* уређаја користећи *CUDA* и *MPI*. Предложили су и стратегије за оптимизацију кода које се заснивају на изменама структуре података и распореда. Имплементација је тестирана на једном ноду кластера који садржи четири Tesla GPU уређаја и показано је да долази до готово линеарног пораста перформанси са повећањем броја *GPU* уређаја. Hong и сарадници (Hong et al., 2015) описали су имплементацију која користи (*MPI*) технику за управљање *GPU* уређајима у кластеру и повећали брзину извршавања имплементације преклапањем операција комуникације и рачунања.

У литератури је до сада описан мали број имплементација *LB* методе коришћењем *OpenCL* спецификације. Текић и сарадници (Tekić et al., 2012) су показали да код написан помоћу *OpenCL* спецификације на *GPU* уређајима показује значајно боље перформансе од секвенцијалне имплементације на *CPU* уређајима. Поређењем *CUDA* и *OpenCL LBM* имплементације на једном уређају показано је да *OpenCL* може да достигне перформансе кода писаног помоћу *CUDA* архитектуре (Calore et al., 2014; Tekić et al., 2014). Calore и сарадници

(Calore et al., 2014) имплементирали су *2D* модел *Lattice Boltzmann* методе на једном хетерогеном уређају. Показали су да имплементација алгоритма у *OpenCL*-у постиже сличне перформансе као алгоритам имплементиран у *CUDA* коду и алгоритам имплементиран у *C* коду. Резултати су тестирани на *NVIDIA* графичким картицама и *Intel Xeon-Phi* вишејезгарним акцелераторима. У раду (Tekić et al., 2014) такође је упоређена имплементација *LBM* алгоритма у *OpenCL* коду и имплементација у *CUDA* коду на *NVIDIA* графичкој картици и показано је да обе имеплементације постижу сличне перформансе на једном *GPU* уређају (*GeForce GT 220*). Имплементација *LBM* алгоритма у *OpenCL* коду приказана у радовима (Tekić et al., 2012; Tekić et al., 2014) извршава симулацију на једном уређају и коришћена је као референтна имплементација за поређење резултата у овој тези, за различите варијације алгоритма у *OpenCL* коду, за истовремено извршавање на више уређаја. Simon и сарадници (Simon et al. 2014) имплементирали су *D3Q19* модел нестишљиве течности у *OpenCL* коду који се извршава на једном уређају демонстрирајући портабилност кода. Такође, у раду је показано да *OpenCL* код и *CUDA* код постижу сличне перформансе за *D3Q19* модел нестишљиве течности.

До сада је у доступној литератури углавном приказивано имплементирање *LB* методе коришћењем *CUDA* кода који истраживаче везује за коришћење *NVIDIA*-а уређаја. Прва имплементација коришћењем *CUDA* програмског модела урађена је 2008 године (Tölke, 2008), додатно убрзање симулације постигнуто је истовременим извршавањем на више уређаја за различите моделе: ток флуида у шупљини (Obrecht et al., 2013), ток флуида кроз порозну средину (Huang et al., 2014). Коришћењем модела тока у шупљини показано је да симулације имплементиране на више *GPU* уређаја истовремено помоћу *CUDA* програмског модела имају пораст перформанси са порастом величине мреже у односу на имплементацију на једном *GPU* уређају (Chang et al., 2013; Chang et al., 2013). У *OpenCL* коду до сада је имплементирано извршавање *LB* методе на једном уређају. У овој тези биће приказана имплементација извршавања *LBM* (за модел *D2Q9*) на више уређаја истовремено што је и њен основни допринос. Такође, у постојећој литератури је показано да *OpenCL* код који се извршава на једном уређају може да постигне сличне перформансе као имплементација *LB* методе у *CUDA* коду. (Calore et al., 2014; Simon et

al. 2014; Tekić et al., 2012; Tekić et al., 2014). Ова теза искоришћена је да се код поређења ефикасности симулације на више уређаја истовремено са симулацијом на једном уређају, као референтни модел за симулацију на једном уређају користи такође *OpenCL* имплементација (Tekić et al., 2012; Tekić et al., 2014).

Архитектура микросервиса је сервисно оријентисана архитектура која пружа могућност креирања дистрибуираних апликација које су лако прошириве и независне (Dragoni et al., 2017; Mazzara et al., 2016; Shadija et al., 2017).

2. Основни концепти паралелног програмирања

У овом поглављу дат је преглед основних концепата везаних за паралелно програмирање. У првом делу описан је развој хардвера за паралелно извршавање програма. Прво је дат историјат развоја супер рачунара за чију појаву се везује и појава паралелног рачунања. Затим је дат кратак преглед кластера који су омогућили прву ширу употребу паралелизације. Представљен је и кратак историјат персоналних рачунара који се са развојем вишејезгарних архитектура такође могу користити за извршавање паралелних симулација. Дат је преглед вишејезгарних архитектура: *multy core* и *many core* архитектура. У другом делу описан је развој софтвера за паралелно програмирање.

2.1. Развој хардвера за паралелно рачунање

2.1.1. *HPC* рачунари

Развој првих рачунара који се могу сматрати супер (*HPC*) рачунарима почиње крајем педесетих година прошлог века. *HPC* рачунари имају перформансе које су једнаке или су близу највећим брзинама операција у секунди које рачунари могу да постигну. *HPC* рачунари користе се за извршавање сложених симулација помоћу којих се проучавају сложени научни проблеми. У последњих 20 година перформансе супер рачунара појачале су се за 500000 пута. Пошто су преформансе *HPC* рачунара значајно боље од перформанси персоналних рачунара уместо *MIPS* (милиона инструкција у секунди), перформансе *HPC* рачунара изражавају се помоћу *FLOPS* (*floating-point* операција у секунди) (Parallel Computing: Background; Supercomputer, 2019).

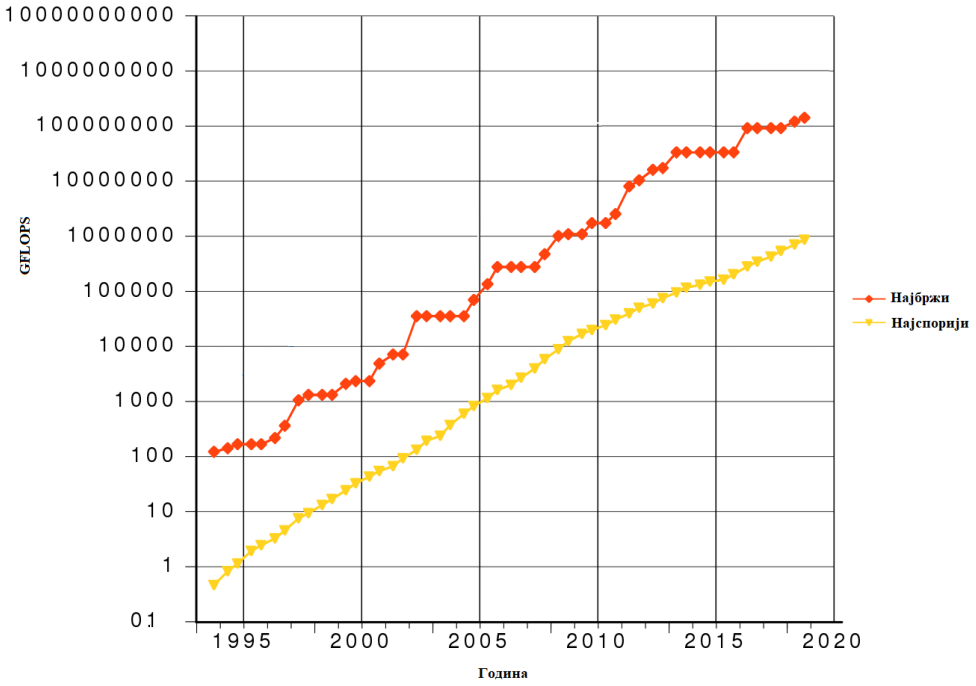
Првим супер рачунаром сматра се *CDC 6600* из 1964. године који је дизајнирао Seymour Cray. Код *CDC 6600* долази до преласка са германиумских на силиконске транзисторе који су омогућили да *CDC* буде око 10 пута бржи од осталих рачунара из тог периода и да достиже брзину од 3 *MFLOPS*-а. 1976. године Cray је представио *Cray-1* који је радио на 80*Mhz* и постизао брзину од 160 *MFLOPS*-а. *Cray-1* је први рачунар који је користио векторску обраду података и меморију транзистора уместо меморије од магнетних језгара али је и даље имао

само један главни процесор. 1982. године представљен је *Cray-X-MP* који је у суштини био *Cray-1* на који су додати процесори, имао је 4 процесора и могао да достигне брзину до 800 *MFLOPS*-а. 1984. године представљени су *Cray-2* и *M-13*. *Cray-2* је достизао брзину од 1.9 *GFLOPS*-а и имао 8 процесора и течно хлађење, док је *M-13* који је представљен у Москви достизао брзину од 2.4 *GFLOPS*-а. Производњом супер рачунара векторске архитектуре наставили су да се баве углавном произвођачи из Јапана, па је од 1993. до 1996. године најбржи супер рачунар био *Numerical Wind Tunnel* који је произвела компанија *Fujitsu* (History of supercomputing, 2019; Supercomputer, 2019, Vaughan-Nichols, 2017).

Поред векторске архитектуре седамдесетих година прошлог века представљен је и први масовно паралелан супер рачунар *ILIAC IV*. Овај супер рачунар требао је да има 256 процесора и постиже брзину од 1 *GFLOPS*. Међутим због проблема у развоју *ILIAC IV* опремљен је само са 64 процесора и достизао је брзину од 200 *MFLOPS*-а. Иако *ILIAC IV* није реализован у потпуности рад на њему био је значајан за даљи развој супер рачунара. Први у потпуности реализован масовно паралелан супер рачунар је *CM-1* (*Connection Machine*) који је развијен на *MIT* универзитету почетком осамдесетих, овај рачунар настао је умрежавањем 65536 микропроцесора. 1991. године представљен је *CM-5* који је могао да изврши билионе аритметичких операција у секунди. 1982. године универзитет у Осаки развио је *LINKS-1* који се састојао од 514 микропроцесора и достизао брзину од 1.7 *GFLOPS*-а. 1996. године *Hitachi* је представио *Hitachi SR2201* који је постизао брзину од 600 *GFLOPS*, а састојао се од 2048 процесора који су били повезани са брзом тро-димезионом *crossbar* мрежом (History of supercomputing, 2019; Supercomputer, 2019).

Intel Paragon представљен је 1993. године, а могао је да има од 1000 до 4000 *Intel i860* процесора различите конфигурације. *Paragon* је подржавао *MIMD* (*multiple instruction, multiple data*) и извршавање процеса на различитим чворовима, док је за комуникацију коришћен *MPI (Message Passing Interface).* Развој *Paragon*-а довео је до побољшања перформанси *CPU* јединица средином деведесетих година. Омогућено је коришћење *CPU* уређаја опште намене за супер рачунаре уместо посебно прилагођених чипова. Следећи развијен супер рачунар *ASCI Red*, био је најбољи супер рачунар до краја двадесетог века,

користио је *Pentium Pro* процесор опште намене и састојао се од 9000 чворова. У двдесет првом веку прелази се на архитектуре са више десетина хиљада процесора опште намене. Развој графичких картица доводи до развоја архитектура које комбинују графичке картице и процесоре опште намене. 2004. године *NEC* је представио *Earth Simulator* који је достизао брзину од 35.9 *teraflops*-а, а састојао се из 640 чворова. 27 рачунара који су били на листи 500 најбржих супер рачунара користили су *IBM Blue Gene* архитектуру која може да користи преко 60000 процесора. Кинески рачунар *Tianhe-I* био је најбржи супер рачунар 2010. године и постизао је брзину од 2.5 *petaflops*-а. Јапан је 2011. године представио *Fujitsu K* рачунар који је радио на 8.1 *petaflops*-а. У 2016. години најбржи рачунар је Кинески *Sunway TaihuLight* који ради на 93.01 *PFLOPS*, а у 2018. години то је Амерички *IBM Summit* који достиже брзину од 122.3 *PFLOPS*. На Слици 1.1. приказане су перформансе у *GFLOPS* првог и последњег рачунара на листи 500 најбржих рачунара од 1994. године до данас (History of supercomputing, 2019; Supercomputer, 2019; Vaughan-Nichols, 2017).



Слика 1.1. Перформансе најбржег и најспоријег супер рачунара на листи 500 најбољих супер рачунара од 1995. године до данас (History of supercomputing, 2019).

Супер рачунар може да се формира и од кластера рачунара. *Beowulf* принцип пружа могућност да се од персоналних рачунара повезаних у локалну мрежу инсталирањем одговарајућих библиотека и програма омогући дељење процеса и да кластер функционише као супер рачунар (Vaughan-Nichols, 2017).

2.1.2. Кластери

Кластер рачунара подразумева више рачунара повезаних (слабо или чврсто) у локалној мрежи. Сви повезани рачунари извршавају исте инструкције и понашају се као један рачунар са много бољим перформансама. Кластером управља "*clustering middleware*" софтверски слој, који омогућава корисницима да користе кластер као један рачунар. Кластер рачунара може се направити од два пресонална рачунара, али се може састојати и од великог броја рачунара који су

повезани у један систем и формирају један супер рачунар. На овај начин кластери се користе као замена за традиционалне супер рачунаре. У 2011. години најбржи супер рачунар био је *K* рачунар који је користио дистрибуирану меморију и кластер архитектуру (Computer cluster, 2019).

Сматра се да је формалну основу за кластер архитектуру рачунара поставио Gene Amdahl из *IBM*-а 1967. године представљањем *Amdahl*-овог закона. Овај закон описује математичко убрзање које се може очекивати када се на паралелној архитектури изврши паралелизација серијских задатака. Први кластер који је развијан за ширу употребу био је *ARC (Attached Resource Computer)* систем, развила га је *Datapoint* корпорација 1977. године. Међутим тек са појавом *VAX* кластера из 1984. године заиста долази до шире употребе кластера. Данас кластер архитектура има велику применљивост, системи који користе ову архитктуру налазе се на листи најбржих светских супер рачунара (Computer cluster, 2019).

2.1.3. Персонални рачунари

Развојем персоналних рачунара односно појавом вишејезгарних архитектура, компликовани прорачуни научних симулација могу да се извршавају на персоналним рачунарима и постају доступни за истраживање великом броју истраживача. Пероснални рачунари (*PC*-рачунари) имају ниску цену, мале димензије (у односу на супер компјутере) и намењени су индивидуалним корисницима. Први корисници рачунара (институције и корпорације) морали су да пишу сопствене програме да би могли да користе рачунаре. Први оперативни системи персоналних рачунара подржавали су само једног корисника и на њима је било могуће покренути само један програм, који се извршавао серијски. Развојем персоналних рачунара почетком овога века корисницима персоналних рачунара постаје лако доступно мноштво комерцијалних софтвера, бесплатних софтвера и бесплатних *open-source* софтвера. Ове софтвере могуће је веома лако инсталирати и одмах користити. Софтвер се данас развија независно од хардвера (Akhter et al., 2006; Gandon, 1999; Personal computer, 2019).

Персонални рачунари први пут су се појавили крајем седамдесетих година прошлог века и првобитно су извршавали само

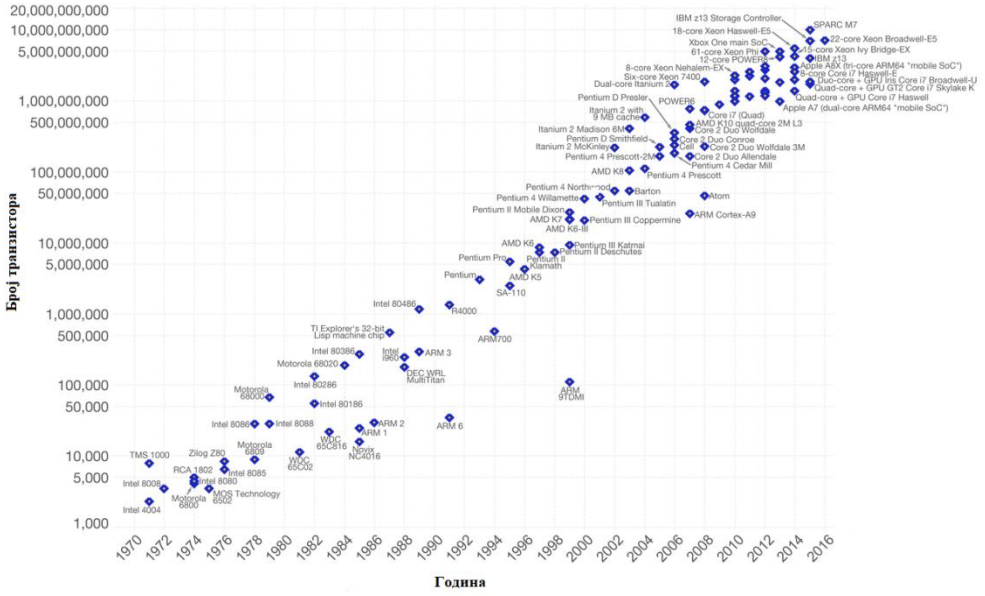
серијски код. 1977. године појавио се *APPLE II* који је био један од најпопуларнијих представника прве генерације персоналних рачунара. Међутим врло брзо примат у производњи и продаји рачунара преузео је *IBM* (*PC* је скраћеница за *IBM* персонални рачунар или *IBM PC*). Једна од ретких компанија које је наставила да производи и продаје рачунаре под својим именом био је *APPLE*. Већина осталих произвођача правила је копије *IBM*-ових рачунара по много нижим ценама. Копије *IBM*-ових рачунара користиле су исте софтвере као и *IBM*-ови рачунари пошто су произвођачи користили исте микропроцесоре као и *IBM*. Данас се под појмом *PC* рачунар подразумева сваки рачунар за индивидуалног корисника који има *Intel* микропорцесор или микропроцесор компатибилан Intel-овом микропроцесору. Појава архитектура са више језгара омогућила је извршавање сложених симулација на персоналним рачунарима (Akhter et al., 2006; Gandon, 1999; Personal computer, 2019).

2.1.4. Архитектуре са више језгара

Централна процесорска јединица (*CPU*) користи се за извршавање програмских инструкција. Инструкције представљају аритметичке, логичке и улазно/излазне операције. Првобитно су развијани процесори са једним језгром који су инструкције извршавали секвенцијално. У другој половини двадесетог века долази до употребе транзистора (микропроцесора) и интегралних кола у производњи процесора што доноси значајно повећање перформанси процесора. Гордон Мур, оснивач *Intel*-а, 1965. године предвиђа да ће се број транзистора на чипу удвостручавати сваке године. Ово предвиђање 1970. године добија назив "Муров закон" (*Moore's Law*). 1975. године Мур је продужио период на две године, а касније је у оквиру *Intel*-а прорарачунато да ће се број транзистора на интегралном колу удвостручавати на 18 месеци. 2015. године *Intel* је објавио да је дошло до успоравања "Муровог закона" и да период удвостручавања сада износи две и по године. Слика 1.2. приказује повећање броја транзистора на интегралном колу (Akhter et al., 2006; Central processing unit, 2019).

Процесори са једним језгром достигли су свој максимум пре десет година када се долази до фреквенције од око 3 *GHz*-а. Због повећане потрошње струје и великог загревања процесора прављење

јачих порцесора са једним језгром постаје неисплативо. Тада долази до промена у дизајну архитектуре централних порцесорских јединица и појаве вишејезгарних процесорских архитектура (Persson Mattsson, 2014).



Слика 1.2. Повећање броја транзистора на интегрисаном колу (Central processing unit, 2019, Moore's Law, 2019)

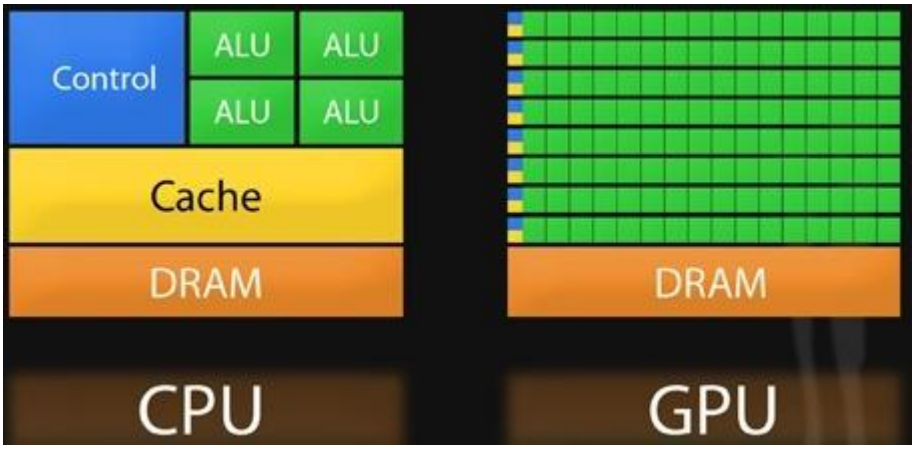
2.1.4.1. Multi-core архитектуре

Од 2005. године јављају се *multi-core* архитектуре, чија се централна процесорска јединица састоји из две или више независних процесорских јединица (језгара). *Multi-core* архитектуру чине језгра која су интегрисана на једном интегралном колу (*chip multiprocessor)* или на више повезаних интегралних кола (Multi-core processor, 2019).

2.1.4.2. Many-core архитектуре

Many-core архитектуре су посебно дизајниране *multi-core* архитектуре предвиђене пре свега за извршавање паралелних прорачуна. Графичке картице су представници *many-core* архитектура

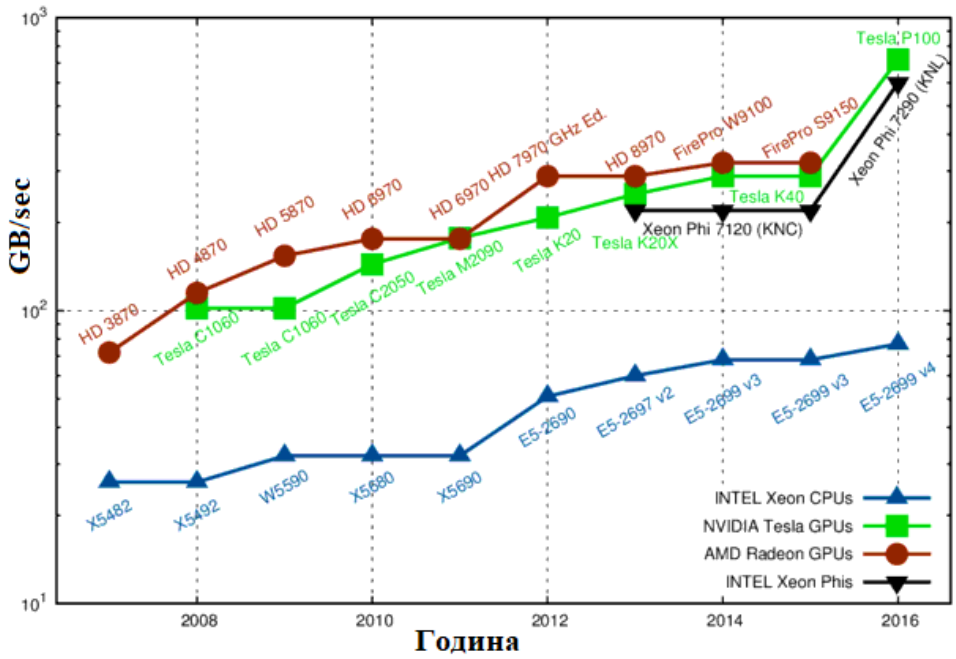
али постоје и други уређаји који припадају овој групи архитектура, на пример *IBM Cell BE (CBEA)* процесор који су заједно развили *IBM*, *Toshiba* и *Sony* (користи се за конзоле за видео игрице и *HPC* рачунаре), *Sony Playstation 4*. Број процесора на *many-core* архитектурама је значајно већи него на обичним *multi-core* архитектурама, али су процесори на њима значајно слабији и једноставнији (на пример *TITAN V* има 5120 *CUDA* језгара). Такође код процесора *many-core* архитектура број транзистора процесора који се користе за обраду података много је већи него број транзистора који се користи за кеширање (*cashing*) и управљање токовима процеса (*flow control*). На Слици 1.3. приказано је поређење броја чипова/транзистора на *CPU* и *GPU* уређајима. Већи број транзистора за обраду података доноси добре перформансе *many-core* архитектура у броју операција са покретним зарезом у секунди (*Floating-point operations per second*, *FLOPS*) и добре перформансе за решавање проблема који користе *data-parallel* технику (извршавања захтевних артиметичких прорачуна истовремено над различитим подацима) (Vajda, 2011).



Слика 1.3. Поређење *CPU-GPU* чипова/транзистори посвећених обради података (Chen et al, 2013)

Данашње графичке картице поседују своју меморију. Поређење протока меморије графичке картице и протока системске меморије показује да је проток (*bandwidth*) меморије *GPU* уређаја за ред величине већи. На Слици 1.4. приказано је поређење меморијског протока између *CPU* и *GPU* уређаја. Приликом рачунања графичке картице користе stream processing модел, који омогућава "ток"

података кроз језгра, помоћу ког се сакрива кашњењење (*latency*) приступу меморији (Vajda, 2011).



Слика 1.4. Поређења меморијског протока (*CPU-GPU*) (Rupp, 2013)

Код *many-core* архитектура могуће је истовремено израчунавање пар стотина инструкција па се ове архитектуре могу користити као алтернатива за *HPC* (*High Performance Computing*) рачунаре који су због висине цене доступни малом броју истраживача.

2.2. Историја развоја софтвера за паралелно рачунање

Развој софтвера за паралелно програмирање је веома битан део развоја области паралелног програмирања. Да би се у потпуности искористиле предности паралелне архитектуре непоходно је написати паралелне алгоритме. Писање паралелног кода компликованије је него писање секвенцијалног, јер се мора вршити синхронизација и комуникација између више уређаја. Седамдесетих година *Hoare* логика коришћена је доста успешно за верификацију основних елемената парелелног програмирања. У касним седамдесетим и касним

осамдесетим најчешће су коришћене *CCS* (*calculus of communicating systems*) и *CSP* (*communicating sequential processes*) теорије, које се и данас користе у разним варијацијама. *Petri*-јеве мреже су представљене касних шездесетих и коришћене су за контролисање конкурентности. Међутим ниједан од ових приступа није шире коришћен код *HPC* рачунара. Програми који изгледају као секвенцијални лакши су за верификацију, а овакав изглед програма могућ је код паралелизма података. Секвенцијални изглед паралелног програма лако се остварује коришћењем функционалних језика. Sisal развија функционални језик који на крају ипак губи битку са *Fortranom*. Међутим Sisal у оквиру овог језика представља полиморфизам и функције вишег реда које ће и даље наставити да се развијају. Почетком деведесетих у оквиру *Fortran*-а појављују се коментари који указују да се делови кода извршавају паралелно, а компајлер може да користи или игнорише коментаре зависно од перформанси рачунара. Средином деведесетих долази до преласка на *MPI* (*Message Passing Interface*) стандард који преузима водећу позицију између великог броја различитих интерфејса за програмирање масовно паралелних рачунара (*MPP*) и кластера. Такође крајем деведесетих за контролу дељене меморије примат преузимају два стандарда *pthreads* и *OpenMP* (Lengauer, 2000).

Са развојем *many-core* архитектура, сваки од произвођача развија свој модел за паралелно програмирање уређаја које производи. *NVIDIA* развија *CUDA* програмски модел. *ATI* такође почиње да развија свој програмски пакет *Brooke,* али са појавом *OpenCL (Open Computing Language)* стандарда прикључује се његовом развоју и напушта развој сопственог софтверског пакета. *OpenCL* стандард пружа могућност паралелног програмирања хетерогених уређаја различитих произвођача.

2.2.1. MPI

Message Passing Interface је спецификација која се користи за комуникацију код паралелног програмирања. *MPI* је првобитно развијен за писање апликација и билиотека за окружења која користе дистрибуирану меморију, циљ је био да се за ова окружења омогући портабилност и једноставно коришћење програма. *MPI* је прва стандардизована и платформски независна спецификација за размену порука у чији развој је било укључено 40 организација, односно

велики броја универзитетских истраживача и већина највећих произвођача рачунара. Почетак рада на стандардизацији спецификације започет је на радионици коју је организовао Центар за истраживање паралелног програмирања у *Williamsburg*-у (Virginia) у априлу 1992. године. У оквиру ове радионице постављене су основне смернице везане за развој *MPI* спецификације и основана *MPI* група. *MPI* дефинише скуп рутина које произвођачи могу ефикасно имплементирати. Скуп рутина намењен је да обезбеди синхронизацију и комуникацију између скупа процеса (који су мапирани на чворове, сервере или на инстанце рачунара) независно од коришћеног програмског језика. *MPI* спцификација првобитно је развијена, а и данас се најчешће користи за паралелизацију на нивоу процеса, мада може да подржи и паралелизацију на нивоу података (Blaise, 2019; Message Passing Interface, 2019).

2.2.2. OpenMP

OpenMP је апликативни програмски интерфејс (АПИ) који пружа подршку за рад са паралелним рачунарским системима са дељеном меморијом. Конзорцијум *OpenMP Architecture Review Board*, састављен од представника водећих произвођача хардвера и софтвера управља развојем *OpenMP* интерфејса. *OpenMP* омогућава лако развијање паралелних апликација на различитим платформама како персоналних рачунара тако и супер рачунара. Креирањем хибридних модела који користе *OpenMP* и *MPI* паралелни програми се могу извршавати на кластерима рачунара. *OpenMP* се користи да би се обезбедио паралелизам у оквиру једног чвора, док се *MPI* користи за паралелизам између чворова кластера. Прва верзија *OpenMP*-ја представљена је 1997. године за *Fortran* 1.0, а најновија верзија која пружа подршку и за рад са графичким картицама представљена је у новембру 2018. године (OpenMP, 2019).

2.2.3. CUDA

CUDA (*Compute Unified Device Architecture*) софтверски пакет развила је компанија *NVIDIA* за програмирање својих *GPU* уређаја. Софтверски пакет *CUDA* представљен је у новембру 2006. године, представљен је заједно са првом *DirectX 10* графичком картицом-*GeForce 8800 GTX*. Појава *CUDA* пакета омогућила је значајно

повећање перформанси рачунара коришћењем *GPU* уређаја. *CUDA* је паралелни програмски модел и скуп наредби који пружају могућност лакшег и ефикаснијег решавања великог броја сложених проблема који захтевају велики број израчунавања у јединици времена. Програмски пакет *CUDA* има ниску криву учења за програмере који користе језике сличне *C* програмском језику. Данас се *CUDA* користи у хиљадама апликација и коришћена је за решавање научних проблема из којих су објављене хиљаде радова (Cheng et al., 2014; Sanders et al., 2011).

2.2.4. OpenCL

OpenCL спецификација представљена је 2008. године, сви водећи произвођачи софтвера имају своје имплементације *OpenCL* стандарда. *OpenCL* је прва спецификација која је омогућила извршавање паралелних програма на процесорима и графичким картицама, односно хетерогеним уређајима (David et al., 2015). *OpenCL* спецификација биће детаљније описана у поглављу 3.

3. Преглед технологија и алата коришћених за развој апликације за симулацију динамике флуида

У тези су коришћене две основне групе технологија и алата, за паралелизацију и за развој веб апликација. Централни део апликације/софтвера представља солвер који је имплементиран у *OpenCL-у* (*krenel*-и) помоћу којег је могуће симулацију извршавати паралелно на различитим врстама вишејезгарних уређаја. За развој апликације за симулацију динамике флуида коришћене су технологије и алати који пружају могућност платформске независности, лаке проширивости и једноставног коришћења. Платформска независност постигнута је коришћењем *OpenCL* спецификације и *JAVA* програмског језика, док је лака проширивост омогућена коришћењем микросервис архитектуре и *Spring framework*-а. Одабиром микросервис архитектуре постигнута је лака проширивост и размена података на API нивоу између софтвера писаних у различитим програмским језицима. Динамичне и интерактивне веб странице креиране помоћу *ReactJS* framework-а пружају могућност једноставног коришћења апликације и искоришћене су за демонстрацију једне од могућих примена софтвера.

3.1. OpenCL

Вишејезгарне архитектуре имају специфичан дизајн и за њихово програмирање неопходни су посебни програмски модели. Уређај једног произвођача има свој скуп инструкција за рад са тим уређајем, зато сваки произвођач прави свој софтверски пакет за програмирање уређаја које производи. *OpenCL* стандард настао је из потребе да се развије један стандард који би омогућио програмирање *many core* уређаја различитих произвођача али и програмирање *multy core* уређаја, односно програмирање веишејезгарних хетерогених архитектура (David et al., 2015).

OpenCL (*Open Computing Language*) је бесплатан и отворен стандард. Произвођачи могу бесплатно да имплементирају *OpenCL* спецификацију за програмирање уређаја које производе (процесоре, графичке картице, играчке конзоле ...).

OpenCL стандард је првобитно почела да развија компанија *Apple*. *Apple* је у својим рачунарима користио графичке картице различитих произвођача и желео је да буде независан од постојећих програмских модела које су развијали произвођачи графичких картица. У сарадњи са водећим произвођачима графичких картица *Apple* је започео развој *OpenCL* стандарда. Врло брзо даљи наставак рада на његовом развоју препустио је *Khronos Group*-и, која је у то време већ управљала *OpenGL* стандардом за 3D рендеринг.

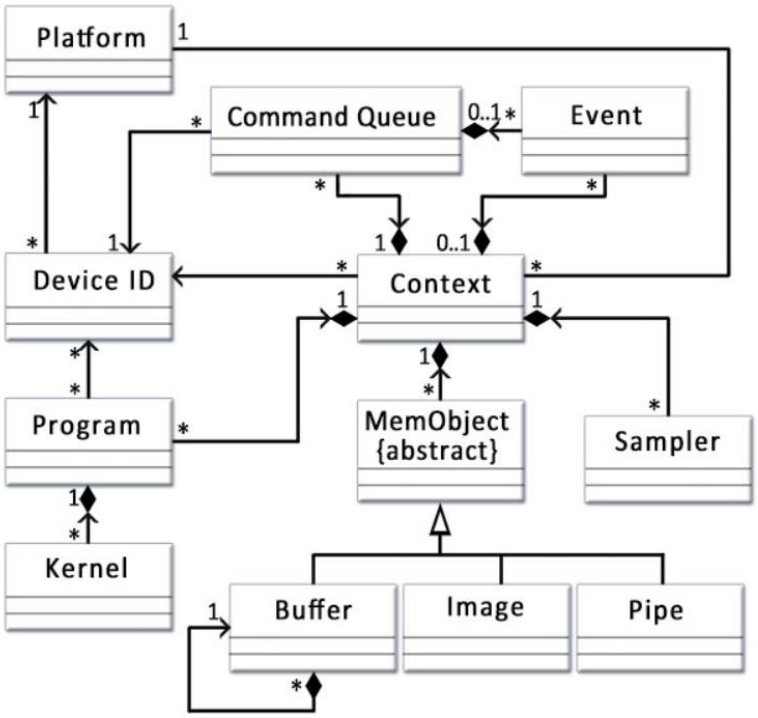
Након што је *Khronos* група преузела управљање *OpenCL* стандардом, у јуну 2008. године формирана је *OpenCL* радна група. Групу су чинили представници произвођача процесора и графичких картица (*AMD*, *Intel*, *IBM*, *NVIDIA*...) чији је циљ био формирање стандарда који ће омогућити уопштавање програмирања хетерогених архитектура. Након само шест месеци радна група је представила прву верзију спецификације за *OpenCL* стандард који је пружао могућност програмирања хетерогених архитектура различитих произвођача. *OpenCL 1.0* - прва верзија стандарда званично је представљена 8. децембра 2008. године. Широку подршку за *OpenCL* гарантовао је велики број произвођача који је имао удела у његовом развоју и који је подршку за *OpenCL* стандард затим и уградио у своје производе. *OpenCL* је омогућио да програм може да се извршава на различитим врстама уређаја, али и могућност истовременог извршавања кода на више различитих уређаја који такође могу бити произведени од стране различитих произвођача (David et al., 2015).

Још једна предност *OpenCL* стандарда, поред тога што пружа могућност програмирања хетерогених архитектура, је подршка паралелизације на нивоу процеса и паралелизације на нивоу података. Паралелизација на нивоу процеса подразумева истовремено извршавање више процеса, а паралелизација на нивоу података истовремено извршавање једне инструкције над различитим подацима (*Single Instruction Multiple Data*- SIMD).

3.1.1. OpenCL дијаграм класа

Дијаграмом класа *UML*-а (*Unified Modelling Language*) на Слици 3.1. су приказане класе *OpenCL* спецификације и везе међу њима.

Платформа представља једну имплементацију *OpenCL* спецификације која се користи за приступ једном или више уређаја који су подржани том имплементацијом. Приликом паралелног програмирања долази до истовременог извршавања задатака, ти задаци су код *OpenCL*-а представљени помоћу *krenel*-а који се генеришу на основу програма и прослеђују уређајима. Основни објекат који управља свим осталим објектима је контекст. Он је контејнер који управља повезаним уређајима и који повезује *kernel* са *command queue-ом* и меморијским објектима. *Kernel* је написан у *OpenCL C* језику и приликом креирања *program*-а компајлира се помоћу *OpenCL* компајлера. Меморијски објекти могу бити *buffer*, слика и *pipe*.



Слика 3.1. Дијаграм класа *OpenCL* стандарда(Class diagram, 2015)

3.1.2. Архитектура OpenCL-а

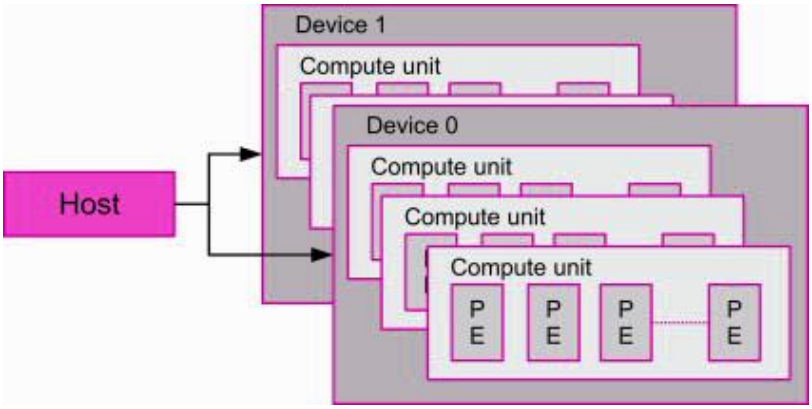
Прилком рада са хетерогеним архитектурама за постизање најбољих перформанси неопходно је максимално искорисити све постојеће ресурсе. *OpenCL* стандард направљен је као програмски модел који пружа могућност хардверске апстракције ниског нивоа која треба да омогући максималну искористивост постојећих ресурса и креирање портабилних програма који се могу извршаватти на хетерогеним архитектурама (David et al., 2015; Ravishekhar, 2013).

OpenCL подржава следеће моделе:

- Модел платформе
- Модел извршавања
- Меморијски модел
- Програмски модел

3.1.2.1. Модел платформе

Модел платформе састоји се од *hosta*-а и *OpenCL* уређаја (*OpenCL devices)* који су повезани са њим и извршавају *kernel*-е. Уколико host уређај подржава *OpenCL* стандард он може истовремено бити *host* уређај и *OpenCL* уређај. Сваки *OpenCL* уређај састоји се из једне или више рачунарских јединица (*Compute Units - CU*), а оне се могу поделити на елементе процесирања (*Processing Element – PE*). Једна инстанца *kernel*-а извршава се на једном елементу процесирања. Модел платформе уређаја који подржавају *OpenCL* је приказан на Слици 3.2. (David et al., 2015; Ravishekhar, 2013).



Слика 3.2. Модел платформе *OpenCL* уређаја (David et al., 2015)

3.1.2.2. Модел извршавања

OpenCL програм састоји се из два дела: *host* програма који је написан у неком од стандардних језика и извршава се на *host* уређају и *kernel*-а који су написани у складу са *OpenCL* спецификацијом, а извршавају се паралелно на једном или више *OpenCL* уређаја. *Host* уређај иницира креирање инстанци *kernel*-а, и тада долази до дефинисања индексног простора (*index* space). За сваку тачку индексног простора извршава се једна инстанца *kernel*-а и она представља једну радну јединицу (*work-item***)**. Све радне јединице извршавају исти код али над различитим подацима. Радне јединице (*work-item***)** груписане су у радне групе (*work-group*), радна група има свој идентификатор, а свака радна јединица има у оквиру радне групе којој припада свој локални идентификатор. Радна јединица може да се идентификује својом глобалном идентификацијом (*global ID*) или паром идентификатора - идентификатор локалне групе и локални идентификатор радне јединице (David et al., 2015; Ravishekhar, 2013).

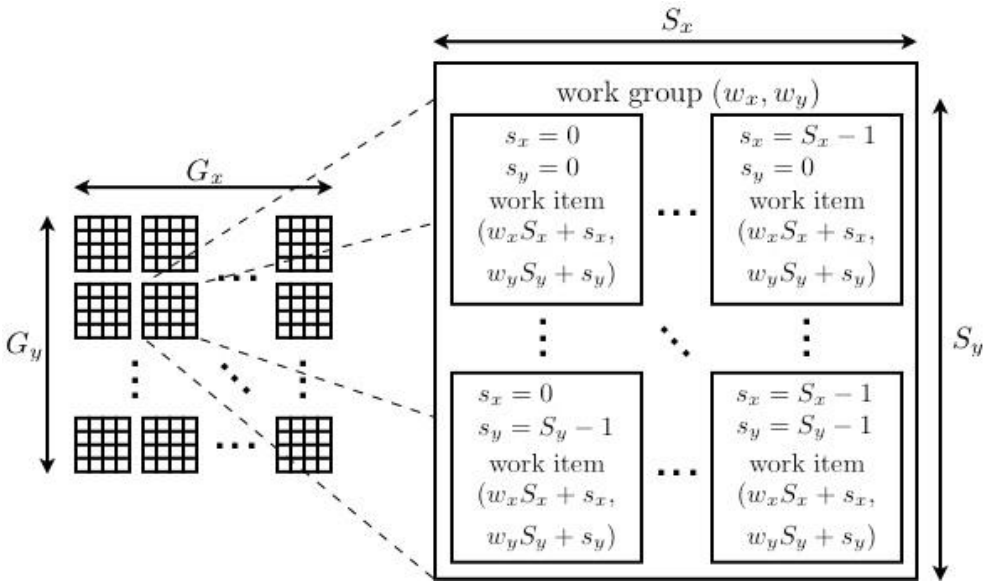
Индексни простор може да подржи до 3 димензије као *N* димензионе торке глобалних и локалних идентификатора. Индексни простор назива се и *NDRange* при чему се *N* замењује бројем који представља димензију индексног простора. Дводимензиони индексни простор приказан је на Слици 3.3. Торка која одређује број радних јединица (*work-items*) је (G_x, G_y). Радне групе индексног простора имају

величину одређену торком (S_x, S_y), а глобални офсет индикатори су представљени са (F_x, F_y). Глобални идентификатор (g_x, g_y) се рачуна помоћу формуле (Ravishekhar, 2013):

$$(g_x, g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + s_y + F_y)$$

Док се број радних група може израчунати на следећи начин:

$$(W_x, W_y) = (G_x / S_x, G_y / S_y)$$



Слика 3.3. Пример *NDRange* индексног простора (Ahmed, 2010)

3.1.2.3. Меморијски модел

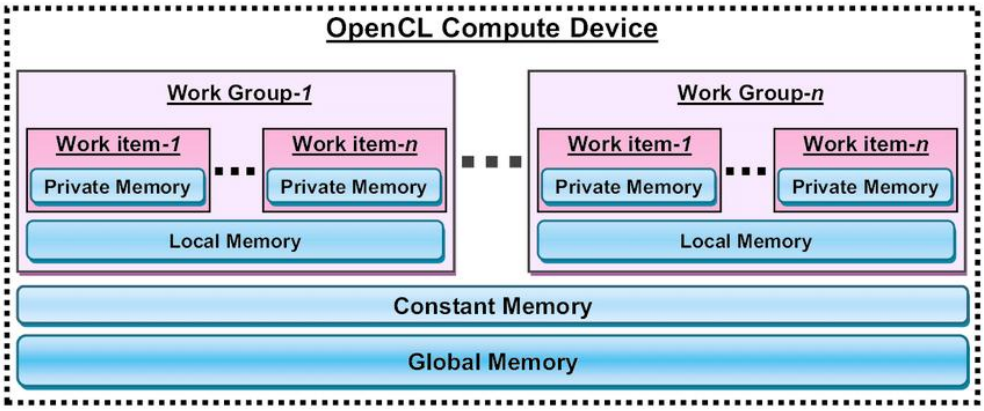
Код меморијског модела разликују се четири врсте меморије: глобална меморија, константна меморија, локалана меморија и приватна меморија.

Глобална меморија (*Global memory*) је највећа али и најспорија врста меморије. У глобалну меморију могу да пишу и да из ње читају све радне јединице (*work-item*) било које радне групе (*work-group*). Уколико уређај (*device*) подржава кеширање, могуће је кеширати податке приликом читања и писања у глобалну меморију.

Константна меморија (*Constant memory*) је део глобалне меморије у коју се смештају константе. У току извршавања *Kernel*-а могуће је вршити само читање из константне меморије.

Локална меморија (*Local memory*) је мања, али бржа од глобалне меморије. Свака радна група има своју локалну меморију којој могу приступити само радне јединице те радне групе. Ова врста меморије подржава читање и писање података.

Приватна меморија (*Private memory*) је најмања али и најбржа врста меморије и припада само једној радној јединици. Подаци који се налазе у приватној меморији једне радне јединице нису видљиви другој радној јединици.



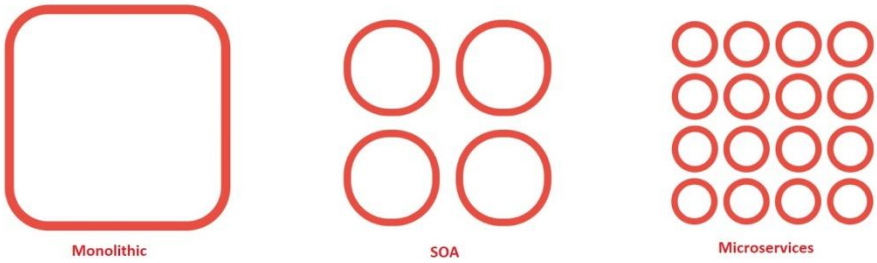
Слика 3.4.Меморијски модел (Aleem. 2015)

3.1.2.4. Програмски модел

OpenCL модел подржава паралелизацију на нивоу података и паралелизацију на нивоу процеса, а могуће је правити комбинацију паралелизације на нивоу података и паралелизације на нивоу процеса. Паралелизација на нивоу података (*Data parallel* програмски модел) подразумева извршавање низа инструкција над елементима меморијског објекта. Програмски модел паралелизације задатака односи се на конкурентно извршавање задатака у оквиру једног или више *command-queue*-ова (Howes et al., 2015).

3.2. Микросервиси

Микросервис архитектура је у последње време постала популарна у развоју софтвера, па тако и за развој *Java web* апликација, као варијанта сервисно оријентисане архитектуре (*SOA*). Апликације/софтвери се креирају/дизајнирају као скуп међусобно слабо повезаних сервиса. Сервиси су подељени по појединачним функционалностима, а за комуникацију (размену података) између сервиса се користе једноставни "лаки" протоколи (нпр. *HTTP* протокол). На Слици 3.5. је графички приказана разлика (еволуција) између наведених архитектура (Dragoni et al. 2017; Mazzara et al. 2016; Shadija et al., 2017; Sharma, 2017).



Слика 3.5. Монолитна, *COA* и микросервис архитектура (Ramees, 2018)

Предности оваквог приступа, дељења апликације у више различитих малих сервиса (микросервиса), су побољшање модуларности система. Оваквим приступом апликација коју развијамо постаје лакша за разумевање, једноставнија за развој/кодирање, лакша за тестирање и једноставнија за измене током животног циклуса. Предности су и у могућности паралелног развоја апликације од више мањих аутономних тимова задужених за развој, одржавање, скалирање сервиса за које је тим задужен. Микросервиси су такође погодни за континуални развој (*continuous delivery*) и рефакторисање кода који су стандард у савременом развоју софтвера.

Приликом развоја *Java web* апликација до скоро је била устаљена пракса развоја монолитних апликација које код комплекснијих система врло брзо постају сложене за одржавање и развој. Потребан је велики број програмера од којих сваки треба да има познавање функционисања читавог (или већег дела) система да би могао да буде део тима који ради на одржавању и имплементацији нових функционалности. Модуларним приступом, какав је случај код микросервис архитектуре, пословна логика система који се имплементира се дели на мање логички независне целине, које се независно имплементирају у појединачним микросервисима. Предност оваквог приступа је и у томе што су појединачне апликације (микросервиси) мање сложене, брже се стартују у развојном окружењу и самим тим убрзавају развој. Такође, програмерима је потребно мање времена и мање доменског знања проблема који се решава, јер раде само на ограниченом делу система (Dragoni et al. 2017; Mazzara et al. 2016; Shadija et al., 2017; Sharma, 2017).

Микросервиси немају јасну дефиницију, али током времена су се издвојиле одређене карактеристике које се подразумевају да микросервис задовољава (Characteristics of Microservices. 2019; Sharma, 2017).

Сервиси у микросервис архитектури су процеси који комуницирају преко мреже (интернет/интранет) користећи програмски/технолошки независан протокол (*HTTP*) у циљу извршавања одређеног задатка.

Сервиси у микросервис архитектури би требали да буду релативно мали, децентрализовани, међусобно независни и да се могу независно поставити у продукцију неким од аутоматизованих процеса/алата.

Сервиси су организовани према функционалностима (контексту) пословне логике одређеног софтвера и развијају се тако да буду грануларни.

Сервиси могу бити написани у различитим програмским језицима, користећи различите системе за управљање базама података и различито софтверско и хардверско окружење.

Предност микросервис архитектуре огледа се и у изоловању грешака система, где се систем моделира тако да грешка у једном од модула микро сервиса нема велик утицај на остале микросервисе. Грешка је изолована у једном делу система, који независно може бити исправљен, тестиран а затим и лако замењен без потребе за поновном имплементацијом и тестирањем комплетног система.

3.3. Spring Boot

У складу са тенденцијама у развоју софтвера, развијен је *Spring Boot framework* са циљем да убрза развој *Spring web* апликација. Микросервис архитектура у *Java* програмском језику се најчешће реализује помоћу *Spring Boot framework-a* (постоји још неколико конкурената од којих је најпознатији *Dropwizard* који је развијен пре *Spring Boot*-а).

Помоћу *Spring Boot*-а врло брзо можемо доћи до *Standalone* апликације која је спремна за продукцију, избегавајући иницијално конфигурасиње веб сервера. На овај начин, програмери су оспособљени да самостално развијају мини апликације без потребе за знањем додатних послова (конфигурисање сервера, поставка апликације на сервер, итд.) који су раније углавном били везани за администраторе мрежних система.

У себи садржи неки од *servlet container*-а, *Tomcat* или *Jetty*, као и стартер пакете за *maven* који садрже иницијалне депенденције које су потребне за веб апликацију. Такође, у инцијалну конфигурацију су укључене и функционалности које су потребне за продукциони сервер као што су метрика, провера доступности сервера (*healthcheck*) и слично (Walls, 2016).

3.4. ReactJS

ReactJS је *Javascript* библиотека за креирање корисничког интерфејса. Библиотека је иницијално развијена од инжењера компаније *Facebook*, која је тренутно задужена за одржавање и развој заједно са групом програмера из *ReactJS* заједнице (Banks et al., 2017).

ReactJS се користи као основа за развијање “*single-page*” апликација или апликација за мобилне уређаје. У оквиру ове тезе коришћен је за развој корисничког интерфејса за покретање симулација.

Карактеристике које су битне да се издвоје у *ReactJS* библиотеци су следеће:

Једносмерно повезивање података помоћу “*props*” (скраћено од речи *Properties*). Компоненте добијају податке од надређене компоненте преко јединственог и непроменљивог (*immutable*) “*props*” скупа вредности.

Компоненте чувају податке о стању (*state*) и могуће је проследити стање подређеним компонентама преко *props*-а.

Битна карактеристика је и коришћење "*virtual DOM*" (*Document Object Model*). *React* креира структуру података у меморији (*cache*) упоређује разлике и затим ажурира само измењене вредности *DOM* објекта који претраживач (*browser*) приказује. На овај начин програмер пише код као да се читава страница рендерује приликом сваке измене, а *React* библиотека рендерује само оне под компоненте које се стварно измене.

Методе животног циклуса компоненти су следеће: *shouldComponentUpdate, componentDidMount, componentWillUnmount* и *render*. Наведене методе се користе респективно за: експлицитна превенција рендеровања компоненте од стране програмера уколико сматра да компонента не треба да се ажурира у одређеном случају, извршавање кода одмах након учитавања (*mount*) компоненте, извршавање кода непосредно пре уништења (*unmount*) компоненте и извршавање кода након сваке промене компоненте уједно и најважнија метода животног циклуса компоненти помоћу које се одсликавају промене на корисничком интерфејсу.

ReactJS библиотека користи *JSX* (*JavaScript XML*) који представља проширење синтаксе *JavaScript* језика. На овај начин се може структурирано приступити рендеровању компоненти користећи синтаксу која је блиска програмерима. Компоненте су углавном

написане помоћу *JSX*-а, али могу бити написане и само користећи *JavaScript* језик.

Негативне стране *ReactJS* библиотеке које се најчешће истичу су велика количина *RAM* меморије која је потребна с обзиром на горе описани "*virtual DOM*" концепт (Banks et al., 2017).

4. Алгоритми за паралелизацију Lattice Boltzmann методе

У овом поглављу прво је укратко представљена *Lattice Boltzmann* метода. Затим су описани алгоритам за паралелизацију *Lattice Boltzmann* методе и варијације алгоритма. Примери алгоритама илустровани су кроз имплементацију симулације струјања/проток флуида у "шупљинама са покретним поклопцем" (*lid-driven cavity*). Овај проблем је одабран јер се у литератури најчешће користи као референти модел и решаван је помоћу различитих нумеричких метода.

4.1. Lattice Boltzmann метода

Кретање флуида се помоћу *Lattice Boltzmann* метода описује као кретање и међусобно сударање честица у оквиру униформне мреже (Mohamad, 2007). У наставку је дата *Boltzmann*-ова једначина:

$$f_i\left(x+e_i\Delta t, t+\Delta t\right)- f_i\left(x,t\right)=\Omega_i\left[f\left(x,t\right)\right] \qquad (4.1)$$

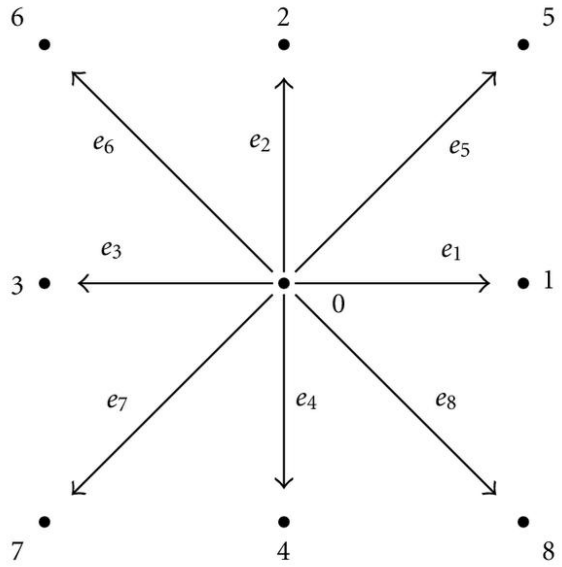
$f_i(x,t)$ је функција расподеле честица које се крећу у смеру i брзином e_i (приказано на Слици 4.1.), тачки x и времену t, Ω је оператор судара (колизије). Функција f_i која се односи на суседни чвор у наредном временском кораку једнак је збиру тренутне функције расподеле и колизионог оператора. Колизиони оператор описује количину промене функције f до које долази прилком судара две честице.

Решавање *Boltzmann*-ове (4.1) једначине је компликовано због израчунавања колизионог оператора Ω. *Bhatnagar, Gross и Krook* (*BGK*) (Körner et al., 2006, Thürey, 2003) предложили су апроксимацију колизионог оператора коришћењем колизионог модела који је у складу са *Navier-Stokes*-овом једначином (He and Luo, 1997). Након апроксимације једначина има следећи облик:

$$f_i\left(x+e_i\Delta t, t+\Delta t\right)- f_i\left(x,t\right)=-\frac{1}{\tau}\left[f_i\left(x,t\right)- f_i^{eq}\left(x,t\right)\right] \qquad (4.2)$$

τ је релаксациони параметар повезан са релаксацијом судара ка локалној равнотежи, f_i је функција дстрибуције, f_i^{eq} је равнотежна функција дистрибуције, e_i је вектор дискретне брзине. Такође део једначина (4.2) може се написати као $\omega=\frac{1}{\tau}$. При чему ће ω бити коефицијент који представља учесталост судара.

Приликом решавања *LB* једначине посматрана област струјања подели се решетком. Један чвор решетке састоји се од више честица (функција расподеле). Честице се крећу до суседног чвора одређеним правцима. Број веза између честица и правци по којима се честице крећу зависи од врсте мреже која се користи. Врста мреже означава се као *DnQm*, *n* је ознака димензије (2 за *2D* и 3 за *3D*), а *m* представља број праваца струјања. У тези ће бити коришћен *D2Q9* модел, на Слици 4.1. приказани су правци микроскопских брзина за модел *D2Q9*.



Слика 4.1. *D2Q9* модел (Mohamad, 2007)

Равнотежна функција расподеле дата је помоћу следеће формуле (He et al., 1998):

$$f_i^{eq} = w_i \rho \left[1 + \frac{3}{c^2} e_i \cdot u + \frac{9}{2c^4} (e_i \cdot u)^2 - \frac{3u^2}{2c^2} \right] \tag{4.3}$$

u и ρ су макроскопска брзина и макроскопска густина, w је фактор тежине, *c* је брзина звука.

Макроскопске величине густина и брзина могу се израчунати помоћу следећих формула:

$$\rho = \sum_{i=0}^{8} f_i \tag{4.4}$$

$$\rho u = \sum_{i=0}^{8} f_i e_i \tag{4.5}$$

c је једнако $\frac{\Delta x}{\delta t}$ и има магнитуду један у овом моделу.

Вредности брзина се приказују као

$$e_0 = (0,0)$$

$$e_i = c\left[\cos\left((i-1)\frac{\pi}{2} \right), \sin\left((i-1)\frac{\pi}{2} \right) \right] \qquad i = 1,2,3,4$$

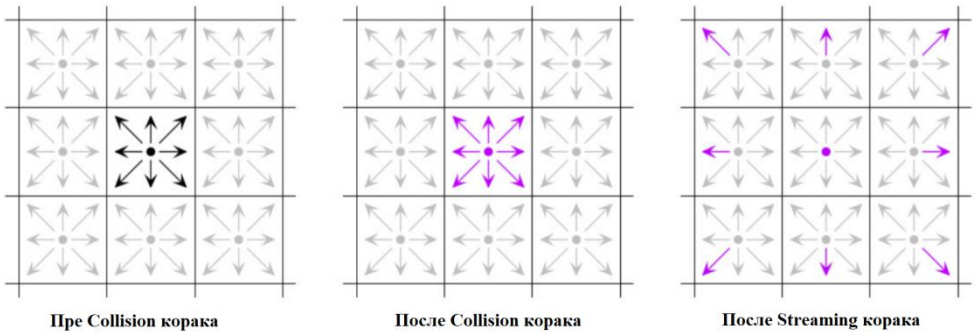
$$e_i = \sqrt{2}c\left[\cos\left((i-5)\frac{\pi}{2} + \frac{\pi}{4} \right), \sin\left((i-5)\frac{\pi}{2} + \frac{\pi}{4} \right) \right] \quad i = 5,6,7,8$$

Макроскопски кинематски вискозитет ν је дат са

$$\nu = \frac{1}{3}\left(\tau - \frac{1}{2} \right) \tag{4.6}$$

Основне операције у *Lattice Boltzmann*-овој методи су: "*streaming*" (струјање) , "*collision*" (сударање) и рачунања граничних услова који морају бити задовољени.

Код “*collision*”-а долази до сударања са другим честицама у суседним ћелијама. “S*treaming*” описује кретање по одговарајућем вектору брзине у одговарајуће суседне ћелије за сваки дискретан интервал. На основу редоследа извршавања операција разликујемо *Push* и *Pull* схему. За имплементацију алгоритма на више хетерогених вишејезгарних уређаја коришћена је *Push* схема. На Слици 4.2. приказани су *Collision* и *Streaming* корак за *Push* схему.



Слика 4.2. *Push* схема (Pananilath, 2016)

Једначина за “*collision*”:

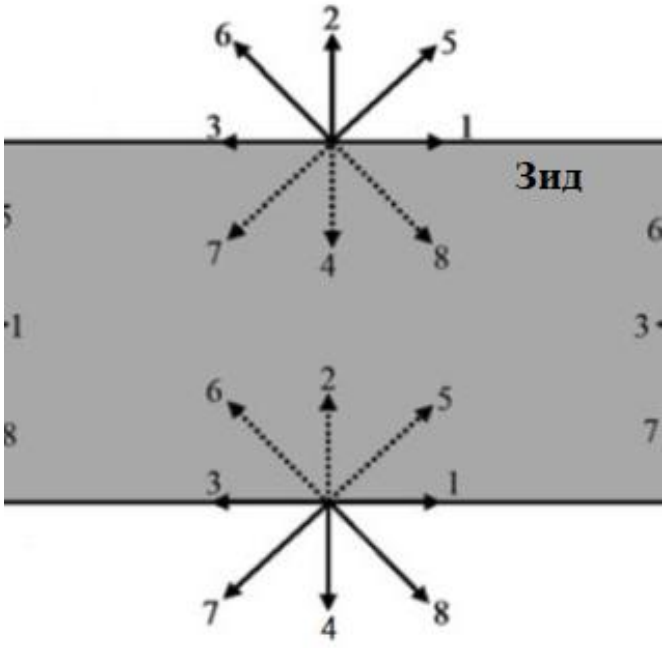
$$f_i^{out}(x,t) = f_i^{in}(x,t) - \frac{1}{\tau}\left[f_i(x,t) - f_i^{eq}(x,t)\right] \tag{4.7}$$

Једначина за “*streaming*”:

$$f_i^{in}(x+e_i,t+1) = f_i^{out}(x,t) \tag{4.8}$$

f_i^{out} представља функцију расподеле након судара, док је f_i^{in} вредност функције расподеле након извршавања “*collision*” и “*streaming*” операција.

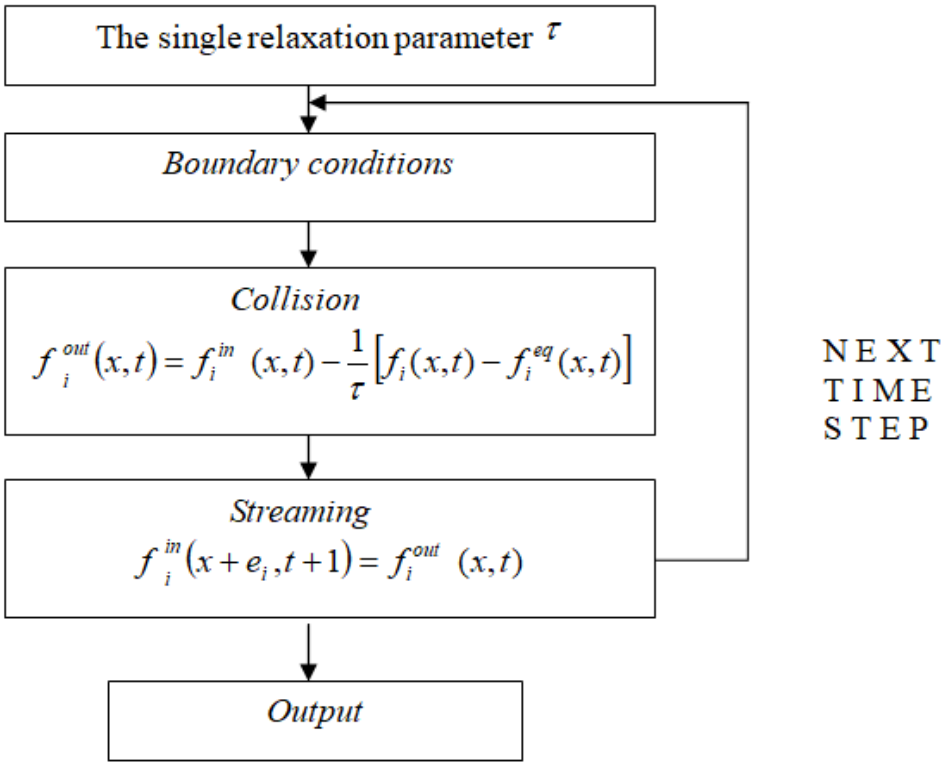
За израчунавање граничних услова непокретних зидова користи се “*bounce back*” схема, а за покретни зид схема *Equilibrium*-а. Коришћењем “*bounce back*” схеме обезбеђено је важење закона о одржању масе и испуњен је услов да је на зиду брзина флуида нула (“*no-slip*” гранични услов), само у случају да је зид гладак и да се струјањем флуида појављује занемарљиво трење примењује се услов проклизавања (“*slip*” гранични услов). На Слици 4.3. приказана је “*bounce bac*k” схема.



Слика 4.3. “*Bounce back*” схема (Mohamad, 2007)

4.2. Основни алгоритам за имплементацију LBM

На Слици 4.4. приказана је основа алгоритма који се користи за решавање *Lattice Boltzmann* методе.



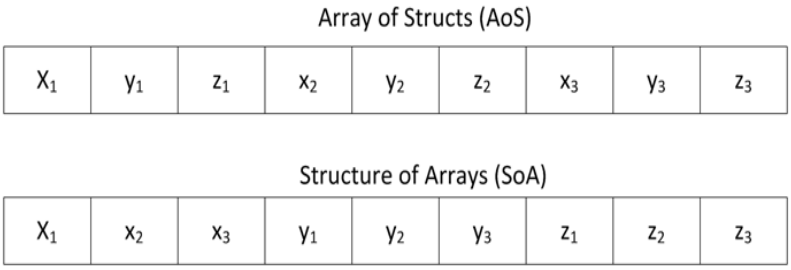
Слика 4.4. Кораци који морају бити извршени приликом симулирања струјања флуида помоћу *LatticeBoltzmann* методе

Прва два корака алгоритма су извршавање граничних услова и *Collision* корак. Имплементација ових корака је једноставна пошто не постоји зависност података између две итерације (временска корака). *Streaming* корак описује пренос информација од једног до другог чвора. У оквиру *streaming* корака јавља се зависност података између итерација. Функција расподеле у *streaming* корак-у итерације (временски корак) *t+1* добија вредност функције расподеле суседног чвора из претходне итерације *t*, односно подаци који се рачунају у тренутној итерацији зависе од података из претходне итерације. За

решавање ове зависности постоји неколико алгоритама који се примењују: алгоритам две мреже и два корака, алгоритам две мреже и једног корака, *Compressed* grid алгоритам, *Swap* алгоритам, *A-A* патерн алгоритам и *Esoteric twist* алгоритам. У овој тези коришћен је алгоритам две мреже и једног корака који решава зависност података коришћењем додатне мреже.

Приликом имплементације алгоритма за индексирање података могу бити коришћени дводимензионални и једнодимензионални низови. За паралелелизацију помоћу *OpenCL* стандарда користе се једнодимезионални низови, пошто *OpenCL* стандард не подржава дводимензионалне низове.

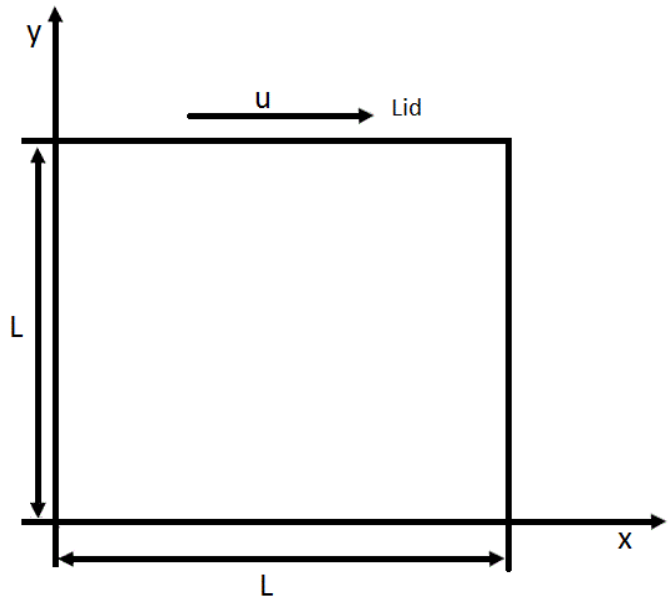
За сваки чвор мреже постоји 9 функција расподеле које одговарају векторима брзина, па се према њиховом груписању разликују две врсте индексирања: *ArrayOfStructure* (*AOS*) и *StructureOfArray* (*SOA*). Код *StructureOfArray* индексирања сваком вектору брзине одговара један низ, ова схема врши оптимизацију *streaming* корака. *ArrayOfStructure* схема оптимизована је за *collision* корак, све вредности вектора брзина груписане су за један чвор и сви подаци се налазе у једном низу. У описаним алгоритмима у овој тези биће коришћена схема *ArrayOfStructure*.



Слика 4.5. *ArrayOfStructure* и *StructureOfArray* индексирање података (Thyholdt, 2012)

Струјање флуида у шупљини са покретним поклопцем (*lid driven cavity flow*) решаван је у алгоритмима описаним у тези. Овај проблем одабран је за референтни пошто је решаван различитим

методама, како емпиријски тако и у области компјутерске симулације. Такође овај проблем се у литератури користи за валидацију нових алгоритама коришћених за решавање *LBM*. На Слици 4.6. приказан је модел шупљине с покретним поклопцем (*lid driven cavity flow*). Шупљина има три непокретна зида и један покретни зид на врху. Зидови имају исте пропорције (*L*), покретни део (*lid*) креће се хоризонтално, паралелно са *x*-осом.



Слика 4.6. Геометрија модела *"lid driven cavity"* (Mohamad, 2007)

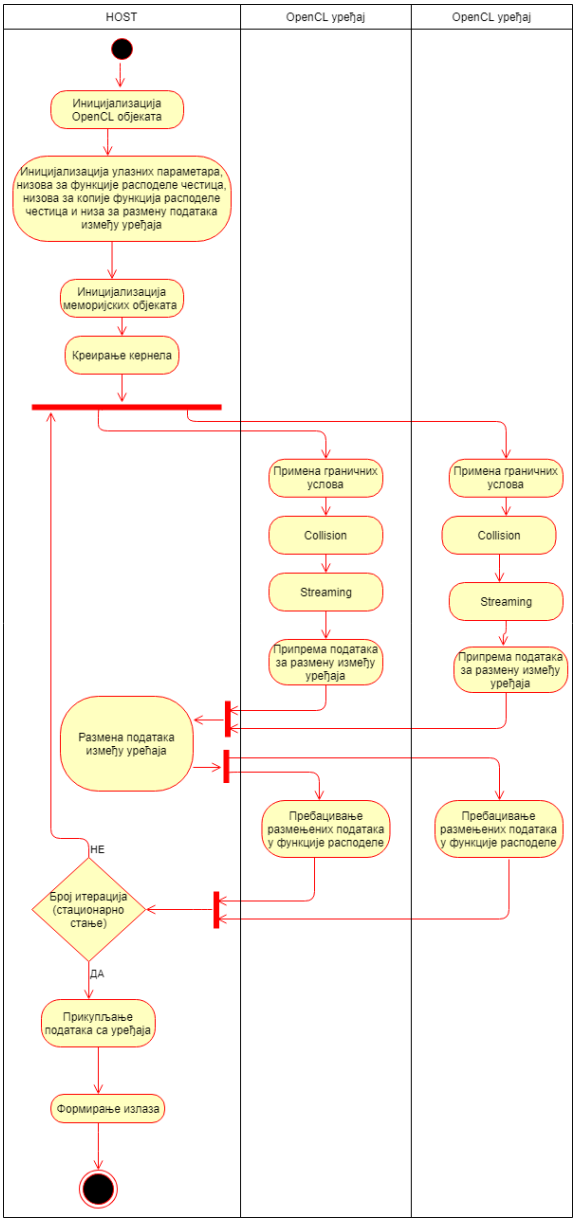
У тези је приказана имплементација LB алгоритма за претходно описани модел *"lid driven cavity"*. Без измене алгоритма могуће је симулирати моделе *deep cavity* и *shallow cavity*. Такође овај модел могуће је проширити додавањем препрека у *cavity*, али би оваква измена утицала на измену алгоритма у делу који се односи на граничне услове.

4.3. Паралелизација LBM на више хетерогених уређаја

Паралелизација *LBM* на више хетерогених вишејезгарних уређаја урађена је на више различитих начина користећи различите карактеристике *OpenCL* стандарда и различите структуре података за пренос података са HOST-а на уређаје за паралелно рачунање. На тај

начин испитане су предности и мане коришћења *OpenCL* стандарада на различитим уређајима. До сада је у литератури приказана имплементација *LBM* на неколико вишејезгарних *NVIDIA* уређаја помоћу *CUDA* програмског модела. У дисертацији први пут је приказана имплементација *LBM* методе помоћу *OpenCL* спецификације на више хетерогених вишејезгарних уређаја. У литератури је до сада нагласак увек био на *HPC* уређајима, у дисертацији нагласак је на персоналним рачунарима. Циљ дисертације је да се искористе сви уређаји различитих произвођача који се налазе на једном персоналном рачунару и постигне максимално убрзање симулације. Различите карактеристике уређаја који се користе на персоналним рачунарима довели су до развоја неколико алгоритама у оквиру ове дисертације. Избором одговарајућег алгоритма на основу карактеристика уређаја долази се до максималног убрзања рада симулације на персоналном рачунару који се користи. Основни алгоритам на основу ког су настали остали алгоритми приказан је у раду (Tekić et al. 2018). Након публикације рада настављен је рад на побољшању алгоритма (на пример за размену података се користи један низ уместо шест, како је било у првој верзији алгоритма) па су резултати презентовани у дисертацији бољи у односу на раније публиковане резултате.

На Слици 4.8. дат је уопштен алгоритам за паралелизацију *LBM* на више хетерогених уређаја. Прво се иницијализују објекти за рад са *OpenCL* уређајима. Након тога се изврши иницијализација података и структура које се користе за пренос података на *OpenCL* уређаје. Креирају се *kernel*-и и поставе аргументи *kernel*-а, зависно од алгоритма разликује се број и имплементација *kernel*-а. За сваки уређај који се користи креира се по једна инстанца *kernel*-а. Након што су *kernel*-и постављени контрола се предаје *OpenCL* уређајима и креће извршавање итерација. Када се заврши *Streaming* корак долази до размене података између уређаја. У наставку се размењени подаци поставе на одговарјућа места у оквиру мреже за сваки уређај и почиње извршавање следеће итерације. Када су све итерације завршене на *HOST*-у се прикупљају подаци и формира се излаз.



Слика 4.8. Уопштен алгоритам имплементације *LBM*

4.4. Дефинисање објеката за рад са OpenCL уређајима

Први корак имплементације је дефинисање *OpenCL* објеката који су неопходни за рад са *OpenCL* уређајима. На самом почетку креира се платформа или платформе уколико постоји више уређаја различитих произвођача. Платформа представља једну *OpenCL* имплементацију и најчешће је везана за једног произвођача, односно уређаји једног произвођача чине једну платформу. Постоје произвођачи који подржавају уређаје других произвођача, али ове имплементације најчешће не могу у потпуности да искористе све предности архитектура других произвођача. На Листингу 4.1. приказан је пример креирања једне платформе.

```
// Одређује се број platform-и
int numPlatformsArray[] = new int[1];
clGetPlatformIDs(0, null, numPlatformsArray);
int numPlatforms = numPlatformsArray[0];
// Након избора platform -е, прави се одговарајућа platform-а platformIndex садржи id одабране platform-е
cl_platform_id platforms[] = new cl_platform_id[numPlatforms];
clGetPlatformIDs(platforms.length, platforms, null);
cl_platform_id platform = platforms[platformIndex];
```

Листинг 4.1. Креирање платформе

Следећи корак је провера доступних уређаја за одабрану платформу/платформе и креирање низа који садржи све уређаје одабране платформе. Када сви уређаји припадају једној платформи у наредном кораку формира се само један контекст, ако постоји више платформи за сваку платформу креира се по један контекст. На Листингу 4.2. приказано је креирање објеката за уређаје и креирање једног контекст објекта.

```
// Obtain the number of devices for the platform
int numDevicesArray[] = new int[1];
clGetDeviceIDs(platform, deviceType, 0, null, numDevicesArray);
int numDevices = numDevicesArray[0];
// Obtain a device IDs
cl_device_id devices[] = new cl_device_id[numDevices];
clGetDeviceIDs(platform, deviceType, numDevices, devices, null);
context = clCreateContext(contextProperties, devices.length, devices, null, null, null);
```

Листинг 4.2. Креирање објеката за *OpenCL* уређаје и креирање једног контекст објекта

За сваки уређај креира се по један *commandQueue* који се користи за слање података (меморијских објеката) и наредби на уређај.

```
commandQueues = new cl_command_queue[numDevices];
long properties = 0;
properties |= CL.CL_QUEUE_PROFILING_ENABLE;
for (int i = 0; i < numDevices; i++) {
    commandQueues[i] = clCreateCommandQueue(context, devices[i], properties, null);
}
```

Листинг 4.3. Креирање *commandQueue*-а

Учита се *".cl*" фајл који садржи *kernel*-е написане у складу са *OpenCL* спецификацијом и на основу њега формира се *Source* стринг. За контекст се прави програм објекат у који се учитава код свих *kernel*-а симулације.

```
program = clCreateProgramWithSource(context, 1, new String[] { KernelSource }, null, null);

clBuildProgram(program, 0, null, null, null, null);
```

Листинг 4.4. Креирање програм објекта

4.5. Иницијализација података и структуре података коришћене за расподелу мреже на OpenCL уређаје

Након креирања *OpenCL* објеката извршавају се следећи кораци: иницијализују се подаци потребни за извршавање симулације и креирају се меморијски објекти. Меморијски објекти представљају омотач око иницијализованих података, они врше слање и расподелу података на уређаје за паралено извршавање.

За извршавање симулације потребни су нам следећи подаци и низови: улазни параметри, функције раподеле, низови за копије функција расподеле (за привремено преузимање вредности функција расподеле), низ за размену података између уређаја, подаци о величини и подаци о почетној тачци дела мреже, који се обрађује на посматраном уређају. Разликујемо две врсте улазних параметара, за сваки формирамо посебан низ: целобројни, који се односе на величину мреже; подаци типа *float*, који се односе на брзину кретања горње плоче и релаксациони параметар. За сваку функцију расподеле формира се један низ, односно прави се укупно 9 низова за функције

расподеле и зависно од типа алгоритма прави се 8, 6 или 4 низа који се користе за привремено смештање вредности функција расподеле. Прави се један низ који служи за размену података и који садржи вредности из више функција расподеле. Размена података врши се између функција расподеле које представљају кретање честица по вертикалном вектору брзине. Коришћење једног већег низа за све функције пружа могућност брже размене података у односу на коришћење више мањих низова (по један за сваку функцију). У низ за размену ставља се само првих *nx* или последњих *nx* чланова сваке функције расподеле која описује кретање честица по вертикалном вектору брзине.

Након што су формирани сви низови за функције расподеле и низови за копије функција расподеле извршена је њихова иницијализација, свим елементима додељена је вредност нула. Затим се за сваки направљен низ формира показивач - *org.jocl.Pointer*.

```
float f1[] = new float[nx*ny];
float f2[] = new float[nx*ny];

….
Pointer ptr_f1 =  Pointer.to(f1);
Pointer ptr_f2 =  Pointer.to(f2);
...
```

Листинг 4.5. Креирање *buffer*-а (низа) и показивача за функцију расподеле *f1*

Након креирања *buffer*-а (низа) и показивача за све променљиве креирају се меморијски објекти. Размена података између *OpenCL* уређаја и *HOST*-а врши се помоћу меморијских објеката. Меморијски објекти према *OpenCL* спецификацији представљају се *cl_mem* структуром података. Разликују се три врсте меморијских објеката *buffer*, *image* и *pipe*. У овој дисертацији је коришћен *buffer* тип меморијских објеката. Сви подаци који се шаљу на уређаје смештени су у један низ меморијских објеката. Меморијски објекат је омотач око показивача (*org.jocl.Pointer*) који показује на одговарајуће податке. Пре него што се меморијски објекти пошаљу на *OpenCL* уређаје мора се извршити подела података на под домене - број под домена једнак је броју коришћених *OpenCL* уређаја.

Подела података може се направити пре креирања *OpenCL* меморијских објеката и након њиховог креирања. Када се подела мреже врши на под домене након креирања објеката разликујемо: поделу података на парцијалне делове помоћу *Subbufer* објеката (*Subbufer* структура) и поделу података на парцијалне делове креирањем додатне структуре за парцијалне податке (*Copy* структура). Подела мреже на парцијалне делове пре креирања *OpenCL* меморијских објеката врши се помоћу показивача (*Pointer* структура).

4.5.1. Подела мреже на под домене након креирања OpenCL меморијских објеката

За слање података на *OpenCL* уређаје направљен је један низ меморијских објеката који садржи све податке који се шаљу на уређаје. Сваки од елемената низа иницијализован је методом *clCreateBuffer*. Параметри методе су контекст, *flag* (у Табели 4.1. приказане су могуће вредности), величина *buffer*-а у бајтовим која треба да буде алоцирана, показивач на низ који садржи податке и код за грешку. За константне вредности користи се *flag CL_MEM_READ_ONLY*. За *buffer*-е функција расподеле и buffer за размену података између уређаја користи се *flag CL_MEM_READ_WRITE*.

cl_mem_flags	Опис
CL_MEM_READ_WRITE	*Flag* одређује да ће *kernel* моћи да чита и пише у меморијски објекат
CL_MEM_WRITE_ONLY	*Flag* одређује да ће *kernel* моћи само да пише у меморијски објекат
CL_MEM_READ_ONLY	*Flag* одређује да ће *kernel* моћи само да чита меморијски објекат
CL_MEM_USE_HOST_PTR	Меморијски објекат ће бити креиран коришћењем меморије *HOST* уређаја
CL_MEM_COPY_HOST_PTR	Меморијски објекат ће бити креиран коришћењем меморије уређаја копирањем са *HOST* меморије на меморију уређаја

Табела 4.1. *Flag* вредности.

```
memObjectsGlobal[0] = clCreateBuffer(context, CL_MEM_READ_ONLY|
CL_MEM_COPY_HOST_PTR, Sizeof.cl_int * 3,     src_host_dim, null);

memObjectsGlobal[1] = clCreateBuffer(context, CL_MEM_READ_ONLY|
CL_MEM_COPY_HOST_PTR, Sizeof.cl_float * 3, src_host_input, null);

memObjectsGlobal [2] = clCreateBuffer(context, CL_MEM_READ_WRITE|
CL_MEM_COPY_HOST_PTR,     Sizeof.cl_float * nx * ny, src_host_f0, null);
...
memObjectsGlobal [16] = clCreateBuffer(context, CL_MEM_READ_WRITE|
CL_MEM_COPY_HOST_PTR, Sizeof.cl_float * nx * ny,
                src_host_tf8, null);

memObjectsGlobal [17] = clCreateBuffer(context, CL_MEM_READ_ONLY|
CL_MEM_COPY_HOST_PTR,
                2 * numDevices * Sizeof.cl_int, src_host_offsetArray, null);
memObjectsGlobal [18] = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR,
                2 * nx * 6 * Sizeof.cl_float, src_host_excahngeData, null);
```

Листинг 4.6, Креирање меморијских објеката

Подаци се након креирања меморијских објеката могу поделити на уређаје на два начина: коришћење *Subbufer*–а (*Subbufer* структура) без заузимања додатног меморијског простора и креирањем додатне структуре (додатних меморијских објеката) за парцијалне податке (*Copy* структура).

4.5.1.1. Подела података на парцијалне делове помоћу Subbufer објеката

Сви глобални меморијски објекти подељени су на парцијалне меморијске објекте помоћу методе *clCreateSubbufer*. За сваки глобални меморијски објекат направљен је један меморијски објекат који се састоји од низа парцијалних меморијских објеката. Број парцијалних меморијских објеката који настану од једног глобалног објекта једнак је броју уређаја на којима ће се извршавати симулација. Подаци првог парцијалног објекта се обрађују на првом уређају, подаци другог парцијалног објекта на другом уређају и тако све до последњег парцијалног објекта и уређаја за сваки од глобалних меморијских објеката. Помоћу методе *createInfo* креиран је показивач на структуру која дефинише подскуп за посматрани *sub-buffer*, сви парцијални

објекти на овај начин показују на глобални меморијски објекат и не долази до алоцирања новог меморијског простора. На Листингу 4.7. дат је пример креирања парцијалног објекта за функције расподеле: *memObjectsGlobal[j+2]* одређује глобални објекат који се дели на парцијалне, *workOffset[i]* означава од које позиције глобалног објекта почињу подаци за тај парцијални објекат, *workSize[i]* представља величину парцијалног објекта.

```
memObjectsPart[i][j] = clCreateSubBuffer(memObjectsGlobal[j+2], (int)
CL_MEM_READ_WRITE, CL_BUFFER_CREATE_TYPE_REGION,
createInfo(workOffset[i], workSize[i], Sizeof.cl_float), null);
```

Листинг 4.7. Креирање парцијалног објекта

Глобални објекти који се односе на улазне параметре нису подељени на парцијалне објекте и шаљу се на сваки од уређаја. На сваки уређај шаље се комплетна направљена структура али одређени елементи структуре имају вредност 0, односно на први уређај шаљу се сви парцијални објекти али само елементи свих првих парцијалних објеката имају вредности различите од нуле, на другом уређају сви парцијални објекти осим другог парцијалног објекта за сваки глобални меморијски објекат су једнаки нули. Последњи уређај садржи информације о комплетној структури и ниједан парцијални објекат на овом уређају није једнак нули.

Када се користи *Subbufer* структура требало би водити рачуна о редоследу уређаја у *CommandQueue*-у. Последњи уређај који се ставља у *CommandQueue*-у треба бити уређај који има најбоље перформансе. Такође треба избегавати постављање процесора на последње место у *CommandQueue*-у, пошто се процесор користи као *HOST* уређај и као уређај за паралелно рачунање. Постављање уређаја који имају слабије перформансе на последње место у *CommandQueue*-у успориће извршавање симулације, а уколико је последњи уређај за паралелно израчунавање процесор старије генерације (*Intel Core i5* и ранији модели) неће бити могуће извршити симулацију. Ови проблеми јављају се јер последњи уређај у *CommandQueue*-у има две функције: обрађује податке из последње групе парцијалних објеката и садржи податке о осталим парцијалним објектима, односно о целокупној мрежи.

4.5.1.2. Подела података на парцијалне делове креирањем додатне структуре за парцијалне податке

Пошто је направљен низ меморијских објеката који показује на глобалне *buffer*-е (низове), за сваки глобални меморијски објекат направљен је нови низ парцијалних објеката (*Copy* структура). За низове парцијалних објеката алоциран је додатни меморијски простор. Затим су из глобалног меморијског објекта у нови низ меморијских објеката помоћу методе *clEnqueueCopyBuffer* копирани одговарајући делови података, што је приказано на Листингу 4.8.

```
for (int j = 0; j < 9; j++) {
    memObjectsPart[i][j] = clCreateBuffer(platform.context, particalDistrFlag, workSize[i]
* Sizeof.cl_float, null, null);
}
            ...
for (int j = 2; j < 13; j++) {
    clEnqueueCopyBuffer(platform.commandQueues[i], memObjectsGlobal[j],
memObjectsPart[i][j - 2], workOffset[i] * Sizeof.cl_float, 0, workSize[i] * Sizeof.cl_float,
0, null, null);
}
```

Листинг 4.8. Креирање структуре за парцијалне податке

На уређај су послати оригинални објекат и његова копија подељена на парцијалне објекте. Над подацима оригиналних меморијских објеката не врше се никакве операције и он остаје исти до краја симулације. Почетни глобални објекти направљени су само да би се омогућила подела глобалних *buffer*-а на парцијалне објекте. Пошто су на почетку симулације подаци иницијализовани само нулама почетна структура заузима малу количину меморијског простора. Обрада података врши се над парцијалним меморијским објектима и њихове вредности се враћају на *HOST* назад у глобалне *buffer*-е на основу којих се формира излаз. Оваква меморијска структура је показала боље перформансе на персоналним рачунарима, када се не користи локална меморија уређаја, од структуре направљене помоћу *clCreateSubbufer* методе јер су уређаји који се користе на персоналним рачунарима слабијих перформанси и чување комплетне мреже на последњем уређају успорава рад уређаја више него чување иницијалне структуре на свим доступним уређајима.

4.5.2. Расподела података помоћу показивача пре креирања OpenCL меморијских објеката

Подела података пре креирања *OpenCL* меморијских објеката врши се помоћу методе *org.jocl.Pointer.withByteOffset*. Сваки глобални показивач помоћу методе *org.jocl.Pointer.withByteOffset* дели се на низ парцијалних показивача. Величина низа показивача који представља један глобални buffer једнака је броју *OpenCL* уређаја на којима се извршава симулација. За сваки парцијални показивач креира се један *OpenCL* објекат помоћу методе *clCreateBuffer*. На Листингу 4.9. приказана је подела једне функције расподеле на под домене помоћу методе *org.jocl.Pointer*. Flag *flagPtr* је *CL_MEM_COPY_HOST_PTR*, осим уколико је уређај истовремено *HOST* и уређај за паралелно рачунање тада се користити flag *CL_MEM_USE_HOST_PTR* да би се избегло додатно копирање и повећала брзина симулације.

```
float f0[] = new float[nx*ny];
…..
Pointer ptr_f0 =  Pointer.to(f0);
……..
Pointer ptr_f0Part [] = new Pointer[numDevices];
……..
ptr_f0Part [i] = src_f0.withByteOffset(workOffset[i]*Sizeof.cl_float);
memObjectsPart[i][0] = clCreateBuffer(context, CL_MEM_READ_WRITE | flagPTR,
workSize[i] * Sizeof.cl_float, ptr_f0Part[i], null);
Pointer (org.jocl.Pointer.withByteOffset);
```

Листинг 4.9. Подела једне функције расподеле на под домене помоћу методе org.jocl.Pointer

4.6. Алгоритми

Пошто су направљене структуре које ће омогућити расподелу података на уређаје креирају се *kernel*-и и постављају параметри *kernel*-а. За сваки уређај направи се једна инстанца кернела и почиње извршавање симулације: извршавање граничних услова, *Streaming*-а и *Collision*-а. Разликујемо четири начина реализације (алгоритма) главних корака симулације.

4.6.1. Основни алгоритам

Код основног алгоритма разликујемо три *kernel-a*: *BSC* (*BounceStreamingCollision*), *Exchange* и *Order*. Да би се *kernel*-и покренули потребно их је додати у *commandQueues* и поставити глобалне бројаче на *HOST* уређају, Листинг 4.10. Први глобални бројач иде од *0* до *nx* (број тачака на *x* оси). Уколико се у оквиру *kernel*-а не користи бројач по *y* оси, global[1] има вредност 1, у супротном пошто се мрежа дели између уређаја по *y* оси други бројач узима вредности од 0 до *sizePerGPU[i] /nx* (одсечак *y* осе на посматраном уређају).

```
global[0] = nx;
for (int i = 0; i < numDevices; i++) {
     global[1] = sizePerGPU[i] / nx;
     clEnqueueNDRangeKernel(commandQueues[i], kernelBSC[i], 2, null,  global, local, 0,
null, null);
    global[1] = 1;
     clEnqueueNDRangeKernel(commandQueues[i], kernelExchange[i], 2, null, global,
local, 0, null, null);
      int tempi;
    if (i== numDevices-1)
      tempi=0;
   else
       tempi=i-1;
     clSetKernelArg(kernelOrder[i], 7, Sizeof.cl_mem, Pointer.to( memObjects[tempi][18]));
    clSetKernelArg(kernelOrder[tempi], 7, Sizeof.cl_mem, Pointer.to( memObjects[i][18]));
}
for (int i = 0; i < numDevices; i++) {
     clEnqueueNDRangeKernel(commandQueues[i], kernelOrder[i], 2, null, global, local, 0,
     null,    null);
     clFinish((cl_command_queue) commandQueues[i]);
}
```

Листинг 4.10. Постављање *kernel*-а у *commandQueue*

Сви главни кораци обухваћени су у оквиру једног *kernel*-а *BSC*. За сваку функцију расподеле направљен је један помоћни низ који служи за решавање проблема зависности података између корака који се јавља у *Streaming*-у. Помоћни низ, након завршетка *kernel*-а *BSC*, садржи податке о мрежи. Први корак који се решава у оквиру *BSC kernel*-а су гранични услови, овај корак дат је на Листингу 4.11. Подаци су по *y* оси подељени на уређаје на којима се извршавају, *i* и *j* су глобални бројачи. Оба глобална бројача крећу од нуле и односе се на локалну мрежу у оквиру посматраног уређаја. Да би се утврдила

позиција чвора у оквиру глобалне мреже неопходно је увести додатни бројач *global_j*.

```
int i=get_global_id(0);

int j=get_global_id(1);

int global_j = get_global_id(1)+offsetIndex[0]/nx;

if (i==0){
      f1[k]=f3[k];
      f5[k]=f7[k];
      f8[k]=f6[k];
}
if (global_j ==0 ){
      f2[k]=f4[k];
      f5[k]=f7[k];
      f6[k]=f8[k];
}
if(i==nx-1){
      f3[k]=f1[k];
      f6[k]=f8[k];
      f2[k]=f4[k];
}
if (global_j ==ny-1 && i>0 && i<nx-1){
float rhon=f0[k]+f1[k]+f3[k]+2.*(f2[k]+f6[k]+f5[k]);
f4[k]=f2[k];
f8[k]=f6[k]+rhon*u0/6.0f;
f7[k]=f5[k]-rhon*u0/6.0f;
}
```

Листинг 4.11. Решавање граничних услова

Следећи корак који се израчунава у оквиру *kernel*-а *BSC* је *Collision* и приказан је на Листингу 4.12.

```
rho = f0[k]+f1[k]+f2[k]+f3[k]+f4[k]+f5[k]+f6[k]+f7[k]+f8[k];
if (global_j ==ny-1 && i>0 && i<nx-1){
    rho = f0[k]+f1[k]+f3[k]+2*(f2[k]+f5[k]+f6[k]);
}
if(i>0 && i<nx-1 && global_j >0 && global_j <ny-1){
          u = (f0[k]*0+f1[k]*1+f2[k]*0+f3[k]*(-1)+f4[k]*  0 +f5[k]*1+f6[k]*(-1)+f7[k]*(-
1)+f8[k]*  1 )/rho;
          v = (f0[k]*0+f1[k]*0+f2[k]*1+f3[k]*  0 +f4[k]*(-1)+f5[k]*1+f6[k]*  1 +f7[k]*(-
1)+f8[k]*(-1))/rho;
}else if(tempj==ny-1 && i>0 && i<nx-1){
    u = u0;
```

```
    v = 0;
}else{
    u=0;
    v=0;
}
float t1 = u*u+v*v;
float feq0 = rho*4.f/9.f *(1.0+3.0*(u*  0 +v*  0 )+4.5*(u*  0 +v*  0 )*(u*  0 +v*  0 )-
1.5*t1);
float feq1 = rho*1.f/9.f *(1.0+3.0*(u*  1 +v*  0 )+4.5*(u*  1 +v*  0 )*(u*  1 +v*  0 )-
1.5*t1);
float feq2 = rho*1.f/9.f *(1.0+3.0*(u*  0 +v*  1 )+4.5*(u*  0 +v*  1 )*(u*  0 +v*  1 )-
1.5*t1);
float feq3 = rho*1.f/9.f *(1.0+3.0*(u*(-1)+v*  0 )+4.5*(u*(-1)+v*  0 )*(u*(-1)+v*  0 )-
1.5*t1);
float feq4 = rho*1.f/9.f *(1.0+3.0*(u*  0 +v*(-1))+4.5*(u*  0 +v*(-1))*(u*  0 +v*(-1))-
1.5*t1);
float feq5 = rho*1.f/36.f*(1.0+3.0*(u*  1 +v*  1 )+4.5*(u*  1 +v*  1 )*(u*  1 +v*  1 )-
1.5*t1);
float feq6 = rho*1.f/36.f*(1.0+3.0*(u*(-1)+v*  1 )+4.5*(u*(-1)+v*  1 )*(u*(-1)+v*  1 )-
1.5*t1);
float feq7 = rho*1.f/36.f*(1.0+3.0*(u*(-1)+v*(-1))+4.5*(u*(-1)+v*(-1))*(u*(-1)+v*(-1))-
1.5*t1);
float feq8 = rho*1.f/36.f*(1.0+3.0*(u*  1 +v*(-1))+4.5*(u*  1 +v*(-1))*(u*  1 +v*(-1))-
1.5*t1);
f0[k] = tau*feq0+(1.-tau)*f0[k];
f1[k] = tau*feq1+(1.-tau)*f1[k];
f2[k] = tau*feq2+(1.-tau)*f2[k];
f3[k] = tau*feq3+(1.-tau)*f3[k];
f4[k] = tau*feq4+(1.-tau)*f4[k];
f5[k] = tau*feq5+(1.-tau)*f5[k];
f6[k] = tau*feq6+(1.-tau)*f6[k];
f7[k] = tau*feq7+(1.-tau)*f7[k];
f8[k] = tau*feq8+(1.-tau)*f8[k];
```

Листинг 4.12. Решавање *Collision* корака

Наредни корак је израчунавање *Streaming*-а, приказан је на Листингу 4.13. , за одговарајуће функције расподеле увећава се бројач по *x* и/или *y* за један (нпр. *ip1* је *i+1*).

```
tf1[ip1+j*nx] = f1[k];
tf2[i+jp1*nx] = f2[k];
tf3[im1+j*nx] = f3[k];
tf4[i+jm1*nx] = f4[k];
tf5[ip1+jp1*nx] = f5[k];
tf6[im1+jp1*nx] = f6[k];
tf7[im1+jm1*nx] = f7[k];
tf8[ip1+jm1*nx] = f8[k];
```

Листинг 4.13. Решавање *Streaming* корака

Након извршавања *kernel*-а *BSC*, извршава се *kernel Exchange* у оквиру ког се постављају подаци које треба разменити између уређаја. Уколико постоје два контекста у оквиру овог *kernel*-а извршава се пребацивање података из помоћних низова назад у оригиналне низове за функције расподеле, приказано на Листингу 4.14.

```
__kernel void Exchange(__constant int *dim,  __global float* f1,
                       __global float* f2,
                       __global float* f3,
                       __global float* f4,
                       __global float* f5,
                       __global float* f6,
                       __global float* f7,
                       __global float* f8,
                       __global float* tf1,
                       __global float* tf2,
                       __global float* tf3,
                       __global float* tf4,
                       __global float* tf5,
                       __global float* tf6,
                       __global float* tf7,
                       __global float* tf8,
                       __global float* exchangeArray,
                       __constant int*  offsetIndex)
{
int i=get_global_id(0);
int j=get_global_id(1);
int nx = dim[0];
int k=i+j*nx;
int sizePerGpu = offsetIndex [1];
f1[k] = tf1[k];
f2[k] = tf2[k];
f3[k] = tf3[k];
f4[k] = tf4[k];
f5[k] = tf5[k];
f6[k] = tf6[k];
f7[k] = tf7[k];
f8[k] = tf8[k];
if (j==0){
    exchangeArray[i] = tf2[i+nx];
    exchangeArray [i+2*nx] = tf5[i+nx];
    exchangeArray [i+3*nx] = tf6[i+nx];
}
if (j==sizePerGpu/nx-1){
    exchangeArray [i+nx] = tf4[i+(sizePerGpu/nx-1)*nx];
    exchangeArray [i+4*nx] = tf7[i+(sizePerGpu/nx-1)*nx];
    exchangeArray [i+5*nx] = tf8[i+(sizePerGpu/nx-1)*nx];
}
}
```

Листинг 4.14. *Kernel Exchange*

Након извршавања *kernel*-а *Exchange* контрола се враћа *HOST* уређају. На *HOST* уређају подаци ће бити ишчитани са уређаја помоћу методе *clEnqueueReadBuffer* и потом поново уписани у *buffer* који се шаље на наредни уређај помоћу методе *clEnqueueWriteBuffer*.

```
if (i== numDevices-1)
    tempi=0;
else
    tempi=i+1;
float tempObject[] = new float[6*nx];
float tempObjectNext[] = new float[6*nx];
clEnqueueReadBuffer(.commandQueues[i], mk.memObjects[i][18], true, 0, 6*nx *
Sizeof.cl_float,Pointer.to(tempObject),0,null, null);
clEnqueueReadBuffer(commandQueues[tempi], mk.memObjects[tempi][18], true, 0, 6*nx
* Sizeof.cl_float,Pointer.to(tempObjectNext),0,null, null);
clEnqueueWriteBuffer(platform.commandQueues[tempi],mk.memObjects[tempi][18],true,
0,6* nx * Sizeof.cl_float,Pointer.to(tempObject),0,null, null);
clEnqueueWriteBuffer(platform.commandQueues[i],mk.memObjects[i][18],true, 0,6* nx *
Sizeof.cl_float,Pointer.to(tempObjectNext),0,null, null);
```

Листинг 4.15. Замена параметара када постоји више контекста

Ако постоји само један контекст, у *kernel*-у *Exchange* постоји само додела вредности за *exchangeArray* (Листинг 4.16.) док се размена података између низова врши на *HOST*-у променом параметара *kernel*-а (Листинг 4.17.). Помоћни низ се проследи као оригинални, а оригинални као помоћни у непарним корацима, док је у парним корацима обрнуто.

```
__kernel void Exchange(__constant int *dim, __global float* f2,
                       __global float* f4, __global float* f5, __global float* f6,
                       __global float* f7, _global float* f8, __global float* exchangeArray,
                       __constant int*  offsetIndex)
{
    int i=get_global_id(0);
    int nx = dim[0];
    exchangeArray [i] = f2[i];
    exchangeArray [i+2*nx] = f5[i];
    exchangeArray [i+3*nx] = f6[i];
    exchangeArray [i+nx] = f4[i+(offsetIndex[1]/nx-1)*nx];
    exchangeArray [i+4*nx] = f7[i+(offsetIndex[1]/nx-1)*nx];
    exchangeArray [i+5*nx] = f8[i+(offsetIndex[1]/nx-1)*nx];
}
```

Листинг 4.16 *Exchange kernel* за један контекст

```
if (timecounter % 2 == 1) {
   clSetKernelArg(kernelBSC[i],3,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][9]));
   clSetKernelArg(kernelBSC[i],4,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][10]));
                    ...
   clSetKernelArg(kernelBSC[i],10,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][16]));
   clSetKernelArg(kernelBSC[i],11,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][1]));
   clSetKernelArg(kernelBSC[i],12,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][2]));
                    ...
   clSetKernelArg(kernelBSC[i],18,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][8]));
   clSetKernelArg(kernelExchange[i],1,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][2]));
   clSetKernelArg(kernelExchange[i],2,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][4]));
   clSetKernelArg(kernelExchange[i],3,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][5]));
   clSetKernelArg(kernelExchange[i],4,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][6]));
   clSetKernelArg(kernelExchange[i],5,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][7]));
   clSetKernelArg(kernelExchange[i],6,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][8]));
   clSetKernelArg(kernelOrder[i],1,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][2]));
   clSetKernelArg(kernelOrder[i],2,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][4]));
   clSetKernelArg(kernelOrder[i],3,Sizeof.cl_mem,Pointer.to(.memObjectsPart[i][5]));
   clSetKernelArg(kernelOrder[i],4,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][6]));
   clSetKernelArg(kernelOrder[i],5,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][7]));
   clSetKernelArg(kernelOrder[i],6,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][8]));
} else {
   clSetKernelArg(kernelBSC[i],3,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][1]));
   clSetKernelArg(kernelBSC[i],4,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][2]));
   clSetKernelArg(kernelBSC[i],5,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][3]));
     ...
   clSetKernelArg(kernelBSC[i],17,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][15]));
   clSetKernelArg(kernelBSC[i],18,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][16]));
   clSetKernelArg(kernelExchange[i],1,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][10]));
   clSetKernelArg(kernelExchange[i],2,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][12]));
   clSetKernelArg(kernelExchange[i],3,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][13]));
   clSetKernelArg(kernelExchange[i],4,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][14]));
   clSetKernelArg(kernelExchange[i],5,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][15]));
   clSetKernelArg(kernelExchange[i],6,Sizeof.cl_mem,Pointer.to(memObjectsPart[i][16]));
   clSetKernelArg(kernelOrder[i], 1, Sizeof.cl_mem, Pointer.to(memObjectsPart[i][10]));
   clSetKernelArg(kernelOrder[i], 2, Sizeof.cl_mem, Pointer.to(memObjectsPart[i][12]));
   clSetKernelArg(kernelOrder[i], 3, Sizeof.cl_mem, Pointer.to(memObjectsPart[i][13]));
   clSetKernelArg(kernelOrder[i], 4, Sizeof.cl_mem, Pointer.to(memObjectsPart[i][14]));
   clSetKernelArg(kernelOrder[i], 5, Sizeof.cl_mem, Pointer.to(memObjectsPart[i][15]));
   clSetKernelArg(kernelOrder[i], 6, Sizeof.cl_mem, Pointer.to(memObjectsPart[i][16]));
}
```

Листинг 4.17. Размена вредности низова променом аргумената *kernel*-а

Размена података између уређаја се у случају једног контекста врши помоћу замене параметара.

```
int tempi;
if (i== numDevices-1)
    tempi=0;
else
    tempi=i-1;
clSetKernelArg(kernelOrder[i], 7, Sizeof.cl_mem, Pointer.to( memObjects[tempi][18]));
clSetKernelArg(kernelOrder[tempi], 7, Sizeof.cl_mem, Pointer.to( memObjects[i][18]));
```

Листинг 4.18. Размена података имеђу уређаја када постоји један контекст

Након размене података контрола се поново враћа уређајима. Затим се на уређајима паралелно врши постављање размењених података на одговарајућа места у низове помоћу *kernel*-а *Order* (Листинг 4.19.) и почиње извршавање нове итерације. Након што су све итерације завршене на *HOST*-у се прикупљају подаци са свих уређаја и формира се излаз.

```
__kernel void Order(__global int *dim,
                    __global float* f2,
                    __global float* f4,
                    __global float* f5,
                    __global float* f6,
                    __global float* f7,
                    __global float* f8,
                    __global float* exchangeArray,
                    __constant int*  offsetIndex)
{
  int i=get_global_id(0);
    int nx = dim[0];
    int sizePerGPU = offsetIndex [1];

    f2[i+0*nx] = exchangeArray[i];
    f5[i+0*nx] = exchangeArray[i+2*nx];
    f6[i+0*nx] = exchangeArray[i+3*nx];

    f4[i+(sizePerGPU/nx-1)*nx]= exchangeArray[i+nx];
    f7[i+(sizePerGPU/nx-1)*nx]= exchangeArray[i+4*nx];
    f8[i+(sizePerGPU/nx-1)*nx]= exchangeArray[i+5*nx];
}
```

Листинг 4.19. Постављање параметара на одговарајућа места у функцијама расподеле након размене података између уређаја

4.6.2. Модификација алгоритма коришћењем локалне меморије у оквиру Streaming-а

Код уређаја који имају *GLOBAL MEMORY CASH TYP*E типа *ReadWriteCache* могуће је премештање вредности функција расподеле од једног до другог чвора мреже коришћењем локалне меморије у оквиру једне радне групе. Разликујемо два начина: први где се *local_work_size* састоји од једног елемента чија величина одговара величини локалне групе и други где *local_work_size* чине два елемената чији производ одговара величини локалне групе. У првом случају постоји један локални *ID* (бројач), а у другом два локална *ID*-а (бројача).

4.6.2.1 Модификација алгоритма коришћењем локалне меморије коришћењем лоаклног бројача по x оси

Када постоји један локални бројач *Streaming* корак се по *x* оси ради у локалу и последица тога је смањењење броја променљивих за резервне копије. Пошто се за функције расподеле *f1* и *f3* вредности пропагирају само хоризонтално локална дељена променљива користи се за решавање зависности података и није потребно формиртати помоћне глобалне низове *tf1* и *tf2*. Симулација почиње извршавањем *kernel*-а *BSC*, део *kernela* који се разликује односи се на *Streaming* корак и дат је на Листингу 4.20. Да би се искористила локална меморија дефинише се 6 локалних променљивих и уводи локални бројач *ix*, *tempny* је величина *y* осе која се обрађује на тренутно посматраном уређају. Величина локалних низова једнака је *local_work_size[0]* а у коду је означена као *BLOCK_SIZE*. Бројачи *upi* и *downi* користе се за померање вредности по *x* оси. Након што су вредности за све функције расподеле чије се вредности пропагирају по *x* оси додељене локално дељеним променљивама позива се метода *barrier(CLK_LOCAL_MEM_FENCE)* која треба да омогући да се извршавање *kernel*-а настави тек када све локално дељене променљиве добију одговарајуће вредности. Овим је завршен *Streaming* по *x* оси, за функције *f1* и *f3* завршен је комплетан *Streaming* корак. У наставку се ради *Streaming* по *y* оси и вредности функција расподеле се затим пребацују у резервне копије.

```
int sizePerGpu = offsetIndex[1];
int tempny =   sizePerGpu/nx;
int i=get_global_id(0);
int j=get_global_id(1);
int k=i+j*nx;
int ix=get_local_id(0);
...
__local float shared_f1[BLOCK_SIZE];
__local float shared_f3[BLOCK_SIZE];

__local float shared_f5[BLOCK_SIZE];
__local float shared_f6[BLOCK_SIZE];
__local float shared_f7[BLOCK_SIZE];
__local float shared_f8[BLOCK_SIZE];

int upi=(ix+1+BLOCK_SIZE)%BLOCK_SIZE;
int downi=(ix-1+BLOCK_SIZE)%BLOCK_SIZE;

int upj=nx*((j+1+tempny)%tempny)+i;
int downj=nx*((j-1+tempny)%tempny)+i;


shared_f1[upi] =  f1[k];
shared_f3[downi] = f3[k];
shared_f5[upi] = f5[k];

shared_f6[downi] = f6[k];
shared_f7[downi] = f7[k];
shared_f8[upi] = f8[k];

barrier(CLK_LOCAL_MEM_FENCE);

f1[k] = shared_f1[ix];
f3[k] = shared_f3[ix];

tf2[upj] = f2[k];
tf4[downj] = f4[k];

tf5[upj] = shared_f5[ix];
tf6[upj] = shared_f6[ix];
tf7[downj] = shared_f7[ix];
tf8[downj] = shared_f8[ix];
```

Листинг 4.20. *Streaming* корак извршен помоћу локалне меморије у оквиру радних група

Пошто је *Streaming* извршен у оквиру радних група коришћењем локалне меморије, глобални елементи мреже нису у исправном редоследу па их је потребно поставити у одговарајући редослед. Због тога је за овај алгоритам направљен још један додатни кернел *OrderByX* приказан на Листингу 4.21. који треба да исправи редослед елемената по *x* оси.

```
__kernel void OrderByX (__global int* dim, __global float* f1, __global float* f3,
                        __global float* f5, __global float* f6, __global float* f7, __global float* f8)
{
 int nx = dim[0];
 int bx=nx/BLOCK_SIZE;
 int ystart = get_global_id(1);
 float temp1=f1[nx-1-BLOCK_SIZE+1 + nx*ystart];
 float temp5=f5[nx-1-BLOCK_SIZE+1 + nx*ystart];
 float temp8=f8[nx-1-BLOCK_SIZE+1 + nx*ystart];
 float temp3=f3[BLOCK_SIZE-1 + nx*ystart];
 float temp6=f6[BLOCK_SIZE-1 + nx*ystart];
 float temp7=f7[BLOCK_SIZE-1 + nx*ystart];
 int ktarget,kstart;
 for(int i=1;i<bx;i++)
 {
     ktarget=(i*BLOCK_SIZE-1)+nx*ystart;
     kstart=((i+1)*BLOCK_SIZE-1)+nx*ystart;
     f3[ktarget]=f3[kstart];
     f6[ktarget]=f6[kstart];
     f7[ktarget]=f7[kstart];
 }
 for(int i=bx-1;i>0;i--)
 {
     ktarget=(i*BLOCK_SIZE)+nx*ystart;
     kstart=((i-1)*BLOCK_SIZE)+nx*ystart;
     f1[ktarget]=f1[kstart];
     f5[ktarget]=f5[kstart];
     f8[ktarget]=f8[kstart];
 }
 f1[0 + nx*ystart]=temp1;
 f5[0 + nx*ystart]=temp5;
 f8[0 + nx*ystart]=temp8;
 f3[nx-1 + nx*ystart]=temp3;
 f6[nx-1 + nx*ystart]=temp6;
 f7[nx-1 + nx*ystart]=temp7;
}
```

Листинг 4.21. Kernel *OrderByX*

Пошто се подаци поставе у исправан редослед позива се *kernel Exchange* (у овом *kernel*-у нема измена у односу на Листинг 4.16.). Затим се на *HOST*-у изврши размена података између уређаја, уколико постоји један контекст врши се замена параметара *kernel*-а. На крају се позива *kernel Order* (Листинг 4.19.) који је без измена у односу на основни алгоритам.

4.6.2.2 Модификација алгоритма коришћењем локалне меморије и коришћењем лоаклног ID по x и y оси

У случају постојања два локална бројача нису потребне глобалне копије за функције расподеле. Зависност података која се јавља у *Streaming* кораку решена је помоћу локалних дељених променљивих. На Листингу 4.22. дат је део *BSC kernel*-а који се односи на *Streaming* корак. Направљено је 7 локалних променљивих чија је величина низа *BLOCK_SIZE*BLOCK_SIZE*, величина низа једнака је величини локалне групе. Сваки елеменат локалне дељене променљиве представља један чвор мреже и он добија вредност функције расподеле одговарајућег суседног чвора. Помоћу функције *barrier* чека се да свака дељена променљива добије одговарајуће вредности за све чворове мреже. Затим се вредности из локалних дељених променљивих враћају у функције расподеле и тиме је у потпуности завршен *Streaming* корак.

```
__local float shared_f1[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f3[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f2[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f4[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f5[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f6[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f7[BLOCK_SIZE*BLOCK_SIZE];
__local float shared_f8[BLOCK_SIZE*BLOCK_SIZE];
int upi=(ix+1+BLOCK_SIZE)%BLOCK_SIZE;
int downi=(ix-1+BLOCK_SIZE)%BLOCK_SIZE;
int upj=(iy+1+BLOCK_SIZE)%BLOCK_SIZE;
int downj=(iy-1+BLOCK_SIZE)%BLOCK_SIZE;
int im1, jm1, jp1, ip1;
if (ix==0) {im1= BLOCK_SIZE-1;}
else{im1= ix-1;}
if (ix==BLOCK_SIZE-1){ip1= 0;}
else{ip1= ix+1;}
if (iy==0) {jm1= BLOCK_SIZE-1;}
else{jm1= iy-1;}
if (iy==BLOCK_SIZE-1) {jp1= 0;}
else{jp1= iy+1;}
shared_f1[ip1+iy*BLOCK_SIZE] = f1[k];
shared_f2[ix+jp1*BLOCK_SIZE] = f2[k];
shared_f3[im1+iy*BLOCK_SIZE] = f3[k];
shared_f4[ix+jm1*BLOCK_SIZE] = f4[k];
shared_f5[ip1+jp1*BLOCK_SIZE] = f5[k];
shared_f6[im1+jp1*BLOCK_SIZE] = f6[k];
shared_f7[im1+jm1*BLOCK_SIZE] = f7[k];
shared_f8[ip1+jm1*BLOCK_SIZE] = f8[k];
barrier(CLK_LOCAL_MEM_FENCE);
f1[k] = shared_f1[ix+iy*BLOCK_SIZE];
f2[k] = shared_f2[ix+iy*BLOCK_SIZE];
f3[k] = shared_f3[ix+iy*BLOCK_SIZE];
f4[k] = shared_f4[ix+iy*BLOCK_SIZE];

f5[k]=shared_f5[ix+iy*BLOCK_SIZE];
f6[k]=shared_f6[ix+iy*BLOCK_SIZE];
f7[k]=shared_f7[ix+iy*BLOCK_SIZE];
f8[k]=shared_f8[ix+iy*BLOCK_SIZE];
```

Листинг 4.22. *Streaming* рађен помоћу локалне меморије у оквиру радних група

Пошто је *Streaming* корак по *x* и *y* оси рађен на локалним уређајима уз употребу локалне меморије након *kernel*-а *BSC* подаци

глобалне мреже нису у исправном редоследу па је потребно извршити *kernel*-е *OrderByX* (Листинг 4.21.) и *OrderByY*(Листинг 4.23.). Такође у оквиру *kernel*-а *OrderByY* постављају се вредности низа за размену података. Након завршетка *kernel*-а *OrderByX* и *OrderByY* контрола се враћа *HOST*-у и изврши се размена података између уређаја. Након што су подаци размењени позива се *kernel Order* (Листинг 4.19.) и завршава се итерација.

```
__kernel void OrderByY(__global int* dim, __global float* input,
                       __global float* f1, __global float* f2, __global float* f3, __global float* f4,
                       __global float* f5, __global float* f6,   __global float* f7, __global float* f8,
                       __global float* exchangeArray, __constant int*  offsetIndex)
{
 int nx = dim[0];
 int sizePerGpu = offsetIndex[1];
 int tempny =   sizePerGpu/nx;
 int bx=tempny/BLOCK_SIZE;
 int xstart = get_global_id(0);
 float temp1=f2[xstart +(sizePerGpu/nx-1-(BLOCK_SIZE-1)) *nx];
 float temp5=f5[xstart +(sizePerGpu/nx-1-(BLOCK_SIZE-1)) *nx];
 float temp6=f6[xstart +(sizePerGpu/nx-1-(BLOCK_SIZE-1)) *nx];
 float temp2=f4[xstart +((BLOCK_SIZE-1)) *nx];
 float temp7=f7[xstart +((BLOCK_SIZE-1)) *nx];
 float temp8=f8[xstart +((BLOCK_SIZE-1)) *nx];
 int ktarget,kstart;
 for(int i=1;i<bx;i++){
      ktarget=xstart+nx*(i*BLOCK_SIZE-1);
      kstart=xstart+nx*((i+1)*BLOCK_SIZE-1);
      f4[ktarget]=f4[kstart];
      f7[ktarget]=f7[kstart];
      f8[ktarget]=f8[kstart];
 }
 for(int i=bx-1;i>0;i--){
      ktarget=xstart+(i*BLOCK_SIZE)*nx;
      kstart=xstart+nx*((i-1)*BLOCK_SIZE);
      f2[ktarget]=f2[kstart];
      f5[ktarget]=f5[kstart];
      f6[ktarget]=f6[kstart];
 }
 f2[xstart]=temp1;
 f5[xstart]=temp5;
 f6[xstart]=temp6;
 f4[xstart+(sizePerGpu/nx-1)*nx]=temp2;
 f7[xstart+(sizePerGpu/nx-1)*nx]=temp7;
 f8[xstart+(sizePerGpu/nx-1)*nx]=temp8;
      tempAll[xstart] = f2[xstart];
      tempAll[xstart+2*nx] = f5[xstart];
      tempAll[xstart+3*nx] = f6[xstart];
      tempAll[xstart+nx] = f4[xstart+(sizePerGpu/nx-1)*nx];
      tempAll[xstart+4*nx] = f7[xstart+(sizePerGpu/nx-1)*nx];
      tempAll[xstart+5*nx] = f8[xstart+(sizePerGpu/nx-1)*nx];

 }
```

Листинг 4.23. *Kernel OrderByY*

4.6.3. Модификација основног алгоритма смањењем броја помоћних низова за копирање вредности функција расподеле

Алгоритам је модификација основног алгоритма уз смањење броја низова за копије функција расподеле и поделе *Streaming* корака на два дела. Први део *Streaming* корака као и до сада се израчунава у *BSC kernel*-у. На Листингу 4.24. приказан је део *BSC kernel*-а који се односи на *Streaming* корак. *Streaming* корак се израчунава само за функције *f1, f2, f3* и *f4*.

```
int im1 = i-1;
int ip1 = i+1;
int jm1=j-1;
int jp1=j+1;
if (j==0) jm1=sizePerGpu/nx-1;
if (j==(sizePerGpu/nx-1) ) jp1=0;
if (i==0) im1=nx-1;
if (i==(nx-1)) ip1=0;

tf1[ip1+j*nx] = f1[k];
tf2[i+jp1*nx] = f2[k];
tf3[im1+j*nx] = f3[k];
tf4[i+jm1*nx] =f4[k];
```

Лиситнг 4.24. *Streaming* корак

4.6.3.1. Модификација основног алгоритма смањењем броја помоћних низова за копирање вредности функција расподеле више context-а

У оквиру *kernel*-а *StreamingEnd* (Листнг 4.25.) ради се *Streaming* корак за преостале четири функције расподеле *f5, f6, f7* и *f8*. Када постоје два контекста, онда се у оквиру *kernel*-а *StreamingEnd* прво пребаце вредности из привремених низова (*tf1, tf2, tf3, tf4*) назад у низове за функције расподеле (*f1, f2, f3, f4*) и тиме се заврши *Streaming* корак за прве четири функције расподеле. Вредности које се размењују између уређаја за функције *f2* и *f4* додају се у *exchangeArray*. Затим се ради *Streaming* корак за последње четири функције расподеле и вредности функција се пребаце у привремене низове. Затим се позива *kernel Exchange* (Листнг 4.26.) у оквиру ког се вредности из привремених низова враћају назад у оригиналне функције расподеле *f5, f6, f7* и *f8*. Затим се вредности ових функција које ће бити размењене додају у *exchangeArray*. По завршетку *kernel*-а *Exchange* контрола се

враћа *HOST*-у, размењују се подаци између уређаја, позива *kernel Order* (Листинг 4.19.) и завршава се итерација.

```
__kernel void StreamEnd(__constant int *dim,      __global float* f1,
                        __global float* f2, __global float* f3,
                        __global float* f4, __global float* f5,
                        __global float* f6, __global float* f7,
                        __global float* f8, __global float* tf1,
                        __global float* tf2, __global float* tf3,
                        __global float* tf4, __global float* exchangeArray,
                        __constant int*  offsetIndex)

{
int i=get_global_id(0);
int j=get_global_id(1);
int nx = dim[0];
int sizePerGpu = offsetIndex [1];
int k=i+j*nx;
int tempj = get_global_id(1)+startIndex[0]/nx;
f1[k]=tf1[k];
f2[k]=tf2[k];
f3[k]=tf3[k];
f4[k]=tf4[k];
if (j==(sizePerGpu/nx-1)){
    exchangeArray[i]=f2[i];
    exchangeArray[i+nx]=f4[k];
}
int im1 = i-1;
int ip1 = i+1;
int jm1=j-1;
int jp1=j+1;
if (j==0) jm1=sizePerGpu/nx-1;
if (j==(sizePerGpu/nx-1) ) jp1=0;
if (i==0) im1=nx-1;
if (i==(nx-1)) ip1=0;
tf1[k] = f5[im1+jm1*nx];
tf2[k] = f6[ip1+jm1*nx];
tf3[k] = f7[ip1+jp1*nx];
tf4[k] = f8[im1+jp1*nx];
}
```

Листинг 4.25. *Kernel StreamingEnd*

```
__kernel void Exchange(__constant int *dim,
                       __global float* f5,
                       __global float* f6,
                       __global float* f7,
                       __global float* f8,
                       __global float* tf1,
                       __global float* tf2,
                       __global float* tf3,
                       __global float* tf4,
                       __global float* exchangeArray,
                       __constant int*  offsetIndex)
{
int nx = dim [0];
int i=get_global_id(0);
int j=get_global_id(1);
int k=i+j*nx;
int sizePerGpu = offsetIndex[1];

f5[k]=tf1[k];
f6[k]=tf2[k];
f7[k]=tf3[k];
f8[k]=tf4[k];

if ( j==(sizePerGpu/nx-1)){
    exchangeArray[i+2*nx]=f5[i];
    exchangeArray[i+3*nx]=f6[i];
    exchangeArray[i+4*nx]=f7[k];
    exchangeArray[i+5*nx]=f8[k];
}
}
```

Листинг 4.26. *Kernel Exchange*

4.6.3.2. Модификација основног алгоритма смањењем броја помоћних низова за копирање вредности функција расподеле за један context

Када постоји само један контекст није потребно експлицитно пребацивати промењиве из привремених низова у оригиналне низове за функције расподеле, већ се као што је раније описано врши замена параметара *kernel*-а. У случају смањења броја привремених низова, након што се вредности функција *f1*, *f2*, *f3* и *f4* доделе привременим низовима, ове функције се користе као привремени низови за функције *f5*, *f6*, *f7* и *f8*. Пре почетка наредне итерације врши се замена параметара *kernel*-а па се функције *tf1*, *tf2*, *tf3* и *tf4* постављају на место

функција *f1*, *f2*, *f3* и *f4*, док се уместо функција *f5*, *f6*, *f7* и *f8* користе *f1*, *f2*, *f3* и *f4* пошто су у претходној итерацији преузеле њихове вредности. Функције *f5*, *f6*, *f7* и *f8* у овој итерацији користе се као привремени низови. Након три итерације све вредности су враћене у првобитне низове, Листинг 4.27.

```
if(mstep%1==0) {
    clSetKernelArg(kernelBSC[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][9]));
    clSetKernelArg(kernelBSC[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelBSC[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][11]));
    clSetKernelArg(kernelBSC[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][12]));
    clSetKernelArg(kernelBSC[i], 7, Sizeof.cl_mem, Pointer.to(memObjects[i][1]));
    clSetKernelArg(kernelBSC[i], 8, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelBSC[i], 9, Sizeof.cl_mem, Pointer.to(memObjects[i][3]));
    clSetKernelArg(kernelBSC[i], 10,Sizeof.cl_mem, Pointer.to(memObjects[i][4]));
    clSetKernelArg(kernelBSC [i], 11, Sizeof.cl_mem, Pointer.to(memObjects[i][5]));
    clSetKernelArg(kernelBSC [i], 12, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelBSC [i], 13, Sizeof.cl_mem, Pointer.to(memObjects[i][7]));
    clSetKernelArg(kernelBSC [i], 14,Sizeof.cl_mem, Pointer.to(memObjects[i][8]));

    clSetKernelArg(kernelStreamEnd[i], 1, Sizeof.cl_mem, Pointer.to(memObjects[i][1]));
    clSetKernelArg(kernelStreamEnd[i], 2, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelStreamEnd[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][3]));
    clSetKernelArg(kernelStreamEnd[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][4]));
    clSetKernelArg(kernelStreamEnd[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][9]));
    clSetKernelArg(kernelStreamEnd[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelStreamEnd[i], 7, Sizeof.cl_mem, Pointer.to(memObjects[i][11]));
    clSetKernelArg(kernelStreamEnd[i], 8, Sizeof.cl_mem, Pointer.to(memObjects[i][12]));

    clSetKernelArg(kernelExchange[i], 1, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelExchange[i], 2, Sizeof.cl_mem, Pointer.to(memObjects[i][8]));
    clSetKernelArg(kernelExchange[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][9]));
    clSetKernelArg(kernelExchange[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelExchange[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][11]));
    clSetKernelArg(kernelExchange[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][12]));
}
```

Листинг 4.27. Размена параметара

```
else if(mstep%1==1) {
    clSetKernelArg(kernelBSC[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][5]));
    clSetKernelArg(kernelBSC[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelBSC[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][7]));
    clSetKernelArg(kernelBSC[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][8]));
    clSetKernelArg(kernelBSC[i], 7, Sizeof.cl_mem, Pointer.to(memObjects[i][9]));
    clSetKernelArg(kernelBSC[i], 8, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelBSC[i], 9, Sizeof.cl_mem, Pointer.to(memObjects[i][11]));
    clSetKernelArg(kernelBSC[i], 10,Sizeof.cl_mem, Pointer.to(memObjects[i][12]));

    clSetKernelArg(kernelBSC[i], 11, Sizeof.cl_mem, Pointer.to(memObjects[i][1]));
    clSetKernelArg(kernelBSC[i], 12, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelBSC[i], 13, Sizeof.cl_mem, Pointer.to(memObjects[i][3]));
    clSetKernelArg(kernelBSC[i], 14,Sizeof.cl_mem, Pointer.to(memObjects[i][4]));

    clSetKernelArg(kernelStreamEnd[i], 1, Sizeof.cl_mem, Pointer.to(memObjects[i][9]));
    clSetKernelArg(kernelStreamEnd[i], 2, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelStreamEnd[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][11]));
    clSetKernelArg(kernelStreamEnd[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][12]));
    clSetKernelArg(kernelStreamEnd[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][5]));
    clSetKernelArg(kernelStreamEnd[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelStreamEnd[i], 7, Sizeof.cl_mem, Pointer.to(memObjects[i][7]));
    clSetKernelArg(kernelStreamEnd [i], 8, Sizeof.cl_mem, Pointer.to(memObjects[i][8]));

    clSetKernelArg(kernelExchange[i], 1, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelExchange[i], 2, Sizeof.cl_mem, Pointer.to(memObjects[i][4]));
    clSetKernelArg(kernelExchange[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][5]));
    clSetKernelArg(kernelExchange[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelExchange[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][7]));
    clSetKernelArg(kernelExchange[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][8]));
}
```

Листинг 4.27. Размена параметара (наставак)

```
else {
    clSetKernelArg(kernelBSC[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][1]));
    clSetKernelArg(kernelBSC[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelBSC[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][3]));
    clSetKernelArg(kernelBSC[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][4]));
    clSetKernelArg(kernelBSC[i], 7, Sizeof.cl_mem, Pointer.to(memObjects[i][5]));
    clSetKernelArg(kernelBSC[i], 8, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelBSC[i], 9, Sizeof.cl_mem, Pointer.to(memObjects[i][7]));
    clSetKernelArg(kernelBSC[i], 10,Sizeof.cl_mem, Pointer.to(memObjects[i][8]));
    clSetKernelArg(kernelBSC[i], 11, Sizeof.cl_mem, Pointer.to(memObjects[i][9]));
    clSetKernelArg(kernelBSC[i], 12, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelBSC[i], 13, Sizeof.cl_mem, Pointer.to(memObjects[i][11]));
    clSetKernelArg(kernelBSC[i], 14,Sizeof.cl_mem, Pointer.to(memObjects[i][12]));


    clSetKernelArg(kernelStreamEnd[i], 1, Sizeof.cl_mem, Pointer.to(memObjects[i][5]));
    clSetKernelArg(kernelStreamEnd[i], 2, Sizeof.cl_mem, Pointer.to(memObjects[i][6]));
    clSetKernelArg(kernelStreamEnd[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][7]));
    clSetKernelArg(kernelStreamEnd[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][8]));
    clSetKernelArg(kernelStreamEnd[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][1]));
    clSetKernelArg(kernelStreamEnd[i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelStreamEnd[i], 7, Sizeof.cl_mem, Pointer.to(memObjects[i][3]));
    clSetKernelArg(kernelStreamEnd[i], 8, Sizeof.cl_mem, Pointer.to(memObjects[i][4]));

    clSetKernelArg(kernelExchange[i], 1, Sizeof.cl_mem, Pointer.to(memObjects[i][10]));
    clSetKernelArg(kernelExchange[i], 2, Sizeof.cl_mem, Pointer.to(memObjects[i][12]));
    clSetKernelArg(kernelExchange[i], 3, Sizeof.cl_mem, Pointer.to(memObjects[i][1]));
    clSetKernelArg(kernelExchange[i], 4, Sizeof.cl_mem, Pointer.to(memObjects[i][2]));
    clSetKernelArg(kernelExchange[i], 5, Sizeof.cl_mem, Pointer.to(memObjects[i][3]));
    clSetKernelArg(kernelExchange [i], 6, Sizeof.cl_mem, Pointer.to(memObjects[i][4]));
}
```

Листинг 4.27. Размена параметара (наставак)

У оквиру *kernel*-а *StreamEnd* (Листинг 4.28.) завршава се *Streaming* корак за функције расподеле *f5*, *f6*, *f7* и *f8*. Унутар *kernel*-а *Exchange* (Листинг 4.29.) постављају се одговарајуће вредности у низ за размену података. Затим се контрола враћа *HOST*-у, врши се размена података. Размењени подаци постављају се на на одговарајућа места у *kernel*-у *Order* и на овај начин се завршава итерација.

```
__kernel void StreamEnd (__constant int *dimArray,
                         __global float* f5,
                         __global float* f6,
                         __global float* f7,
                         __global float* f8,
                         __global float* tf1,
                         __global float* tf2,
                         __global float* tf3,
                         __global float* tf4,
                         __constant int*  offsetIndex)

{

     int i=get_global_id(0);
     int j=get_global_id(1);
     int nx = dimArray[0];
     int sizePerGpu = offsetIndex [1];
     int k=i+j*nx;

     int im1 = i-1;
    int ip1 = i+1;
    int jm1=j-1;
    int jp1=j+1;
     if (j==0) jm1=sizePerGpu/nx-1;
    if (j==(sizePerGpu/nx-1) ) jp1=0;
     if (i==0) im1=nx-1;
    if (i==(nx-1)) ip1=0;

     tf1[k] = f5[im1mix+jm1mix*nx];
    tf2[k] = f6[ip1mix+jm1mix*nx];
    tf3[k] = f7[ip1mix+jp1mix*nx];
    tf4[k] = f8[im1mix+jp1mix*nx];
}
```

Листинг 4.28. *Kernel StreamEnd*

```
__kernel void Exchange(__constant int *dim,
                       __global float* f2,
                       __global float* f4,
                       __global float* f5,
                       __global float* f6,
                       __global float* f7,
                       __global float* f8,
                       __global float* exchangeArray,
                       __constant int*  offsetIndex)
{
    int nx = dim [0];
    int i=get_global_id(0);

    int sizePerGpu = offsetIndex[1];
    tempAll[i]=f2[i];
    tempAll[i+nx]=f4[i+(sizePerGpu/nx-1)*nx];

    tempAll[i+2*nx]=f5[i];
    tempAll[i+3*nx]=f6[i];
    tempAll[i+4*nx]=f7[i+(sizePerGpu/nx-1)*nx];
    tempAll[i+5*nx]=f8[i+(sizePerGpu/nx-1)*nx];
}
```

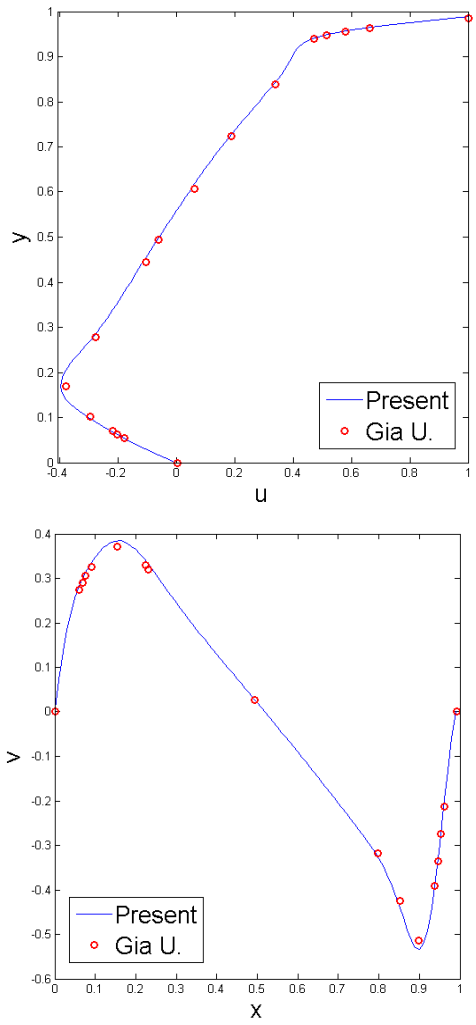
Листинг 4.29. *Kernel Exchange*

5. Приказ и анализа резултата извршавања

У овом поглављу дати су резултати извршавања свих претходно описаних имплементација симулације на различитим платформама и извршена је анализа добијених резултата.

Раније је урађена имплементација *LBM* на једном хетерогеном уређају у *OpenCL* коду, добијени резултати публиковани су и описани у (Tekić et al. 2012). Имплементација *LBM* на једном уређају у *OpenCL* коду у раду (Tekić et al., 2014) упоређена је са имплементацијом у *CUDA* коду на *NVIDIA* графичкој картици и показано је да обе имеплементације постижу сличне перформансе на једном *GPU* уређају. Имплементација приказана у радовима (Tekić et al., 2012; Tekić et al., 2014) извршава симулацију на једном уређају и коришћена је као референтна имплементација за поређење са постигнутим резултатима у овој тези, за различите варијације алгоритма у *OpenCL* коду, за истовремено извршавање на више уређаја. Уређаји који су коришћени у наведеним радовима нису више доступни па је исти алгоритам у тези тестиран на тренутно доступним уређајима. Детаљнија анализа за све тестиране конфигурације дата је у секцији 5.10.

У тези је имплементиран *LBM* алгоритам у *OpenCL* коду који се истовремено извршава на више хетерогених уређаја, што је публиковано и описано у (Tekić et al. 2018). Алгоритам приказан у (Tekić et al. 2018) у тези је додатно побољшан и на основу овог алгоритма развијене су још три варијације које пружају могућност додатног убрзања методе зависно од врсте хардвера који се користи. Резултати решавања *LBM* симулације за Рејнолдсов број (*Re*) 1000 добијени помоћу алгоритама имплементираних у дисертацији упоређени су са резултатима из литературе и добијено је задовољавајуће поклапање. На Слици 5.1. приказани су пресеци брзина по *x* и *y* оси. Резултати приказани на Слици 5.1. израчунати су на Конфигурацији 2. За остале конфигурације и све варијације алгоритма исправност резултата валидирана је упоређивањем графичких приказа пресека брзина као и графичких резултата линија струјања. Резултати добијени за пресеке брзина идентични су онима приказаним на Слици 5.1. па нису додатно приказивани на посебним сликама.



Слика 5.1. Вертикалне брзине за *y*=0.5 за *Re*=1000 (горе) и Хоризонталне брзине за *x*=0.5 за *Re*=1000 (доле) – кругови представљају податке Ghia et al., 1982. године, а линија представља резултат паралелне имплементације у *OpenCL*-у на више хетерогених уређаја

За анализу резултата коришћене су различите величине мреже, а симулације су извршаване за претходно описан референтни проблем кретања флуида у шупљини са покретним поклопцем. За Рејнолдсов

број узета је вредност 1000. Верификација и поређење перформанси симулације извршено је на основу броја измењених чворова мреже по секунди (*MLUPS*), Формула 5.1.

$$MLUPS = \frac{n \cdot m}{t \cdot 10^6} \tag{5.1}$$

n представља број чворова мреже, *m* је број итерација, *t* је време извршавања. У литератури се број измењених чворова мреже по секунди (*MLUPS*) најчешће користи за проверавање, верификацију и поређење перформанси имплементација *Lattice Boltzmann* методе.

За верификацију резултата поређене су постигнуте *MLUPS* вредности за различите имплементације на више различитих конфигурација.

Део мреже који се обрађује на једном уређају одређен је емпиријским путем.

Свака конфигурација тестирана је за сваки описан алгоритам. На сваком алгоритму примењена су сва три типа структуре података која су претходно описана: подела мреже на парцијалне делове помоћу *Pointer*-а пре креирања *OpenCL* објеката (у наставку *Pointer* структура), подела мреже на парцијалне делове креирањем *Subbuffer*-а над претходно направљеним меморијским објектима (у наставку *Subbuffer* структура) и подела мреже на парцијалне делове креирањем меморијских објеката који су парцијалне копије претходно креираних меморијских објеката (у наставку *Copy* структура).

5.1. Конфигурација 1

Конфигурација 1 је чвор кластера Института за Нумеричку Математику Марчука Руске Академије Наука. Чвор се састоји из две Тесла графичке картице, каркатеристике конфигурације дате су у Табели 5.1.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	Tesla C2070	Tesla C2070
CL_DEVICE_TYPE	GPU	GPU
CL_PLATFORM_VENDOR	NVIDIA Corporation	NVIDIA Corporation
CL_PLATFORM_VERSION	OpenCL 1.1 CUDA 6.5.45	OpenCL 1.1 CUDA 6.5.45
CL_DEVICE_VERSION	OpenCL 1.1 CUDA	OpenCL 1.1 CUDA
CL_DEVICE_VENDOR	NVIDIA Corporation	NVIDIA Corporation
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.1	OpenCL C 1.1
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.1. *Конфигурација 1*

Тесла графичке картице користе *OpenCL 1.1* и имају само једну имплементацију платформе - *NVIDIA* Corporation. Из карактеристика уређаја види се да је могуће извршти тестирање имплементације која користи локалну меморију пошто графичке кратице имају вредност *ReadWriteCache* за особину *GLOBAL_MEM_CACHE_TYPE*.

1. *Алгоритам 1*

   У Табели 5.2. приказане су *MLUPS* вредности за извршавање основног алгоритма. Основни алгоритам најбоље перформансе постиже када се за пренос података направе копије за парцијалне податке - *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	521	613	710	730	735	799	807	811
Subbuffer	522	615	715	745	737	801	808	815
Copy	529	635	738	755	770	810	835	840

Табела 5.2. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

У Табели 5.3 приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. Из табеле се види да не постоји велика разлика у преформансама за различите структуре података, *Subbuffer* структура има незнатно боље перформансе од преостале две структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	493	705	776	866	872	937	942	965
Subbuffer	495	707	778	867	873	938	942	966
Copy	490	706	773	866	972	925	941	961

Табела 5.3. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

У Табели 5.4. приказани су резултати извршавања алгоритма који користи локалну меморију и локалне бројаче по *x* и *y* оси. Најбоље резултате алгоритам постиже када се подаци преносе коришћењем *Subbuffer* структуре. Перформансе које се постижу знатно су лошије него за претходно обрађене алгоритме.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	303	386	390	421	425	437	429	441
Subbuffer	304	387	390	421	426	437	430	442
Copy	304	386	389	421	424	436	429	440

Табела 5.4. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и локалних бројача по *x* и *y* оси

4. *Алгоритам 4*

У Табели 5.5. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних

променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	434	535	584	605	620	636	652	661
Subbuffer	434	536	585	605	621	637	653	665
Copy	436	540	588	610	624	640	661	670

Табела 5.5. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

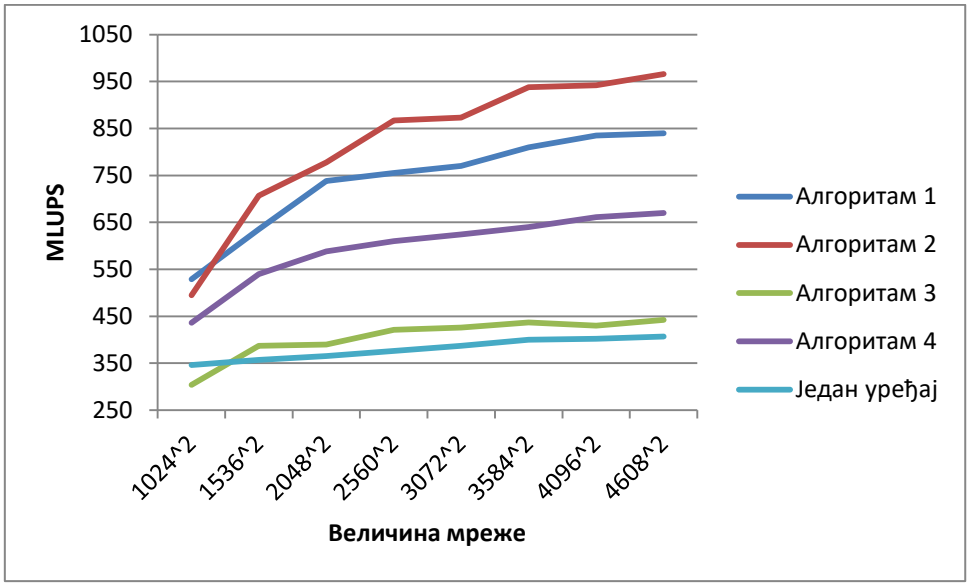
На основу приказаних резултата може се закључити да алгоритми који не користе локалну меморију (*Алгоритам 1* и *Алгоритам 4*) постижу највеће брзине када се за пренос података на уређаје користи *Copy* структура, док се у случају алгоритама који користе локалну меморију (*Алгоритам 2* и *Алгоритам 3*) најбоље перформансе постижу коришћењем *Subbuffer* структуре за пренос података на уређаје.

У Табели 5.6. упоређени су најбољи резултати за сваки од алгоритама и резултати извршавања алгоритма на једном уређају представљеног у (Tekić et al. 2012).

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	529	635	738	755	770	810	835	840
Алгоритам 2	495	707	778	867	873	938	942	966
Алгоритам 3	304	387	390	421	426	437	430	442
Алгоритам 4	436	540	588	610	624	640	661	670
Један уређај	346	357	365	366	367	369	373	375

Табела 5.6. Поређење различитих алгоритама за имеплементцију симулације тока флуида за *Конфигурацију 1*

На Слици 5.2. дат је графички приказ Табеле 5.6.

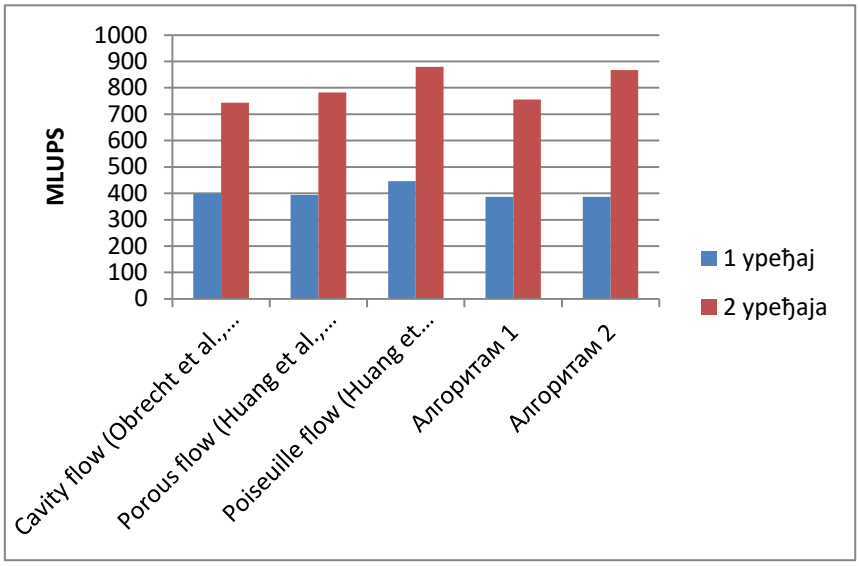


Слика 5.2. Графички приказ Табеле 5.6.

На основу графичког приказа резултата може се закључити да најбоље перформансе на Тесла графичким картицама постиже *Алгоритам 2* који користи локалну меморију и један локални бројач по *x* оси, док најлошије резултате постиже *Алгоритам 3* који користи локалну меморију и бројаче по *x* и *y* оси. Оновни алгоритам (*Алгоритам 1*) има нешто слабије резултате у односу на модификацију која користи локалну меморију, али и он постиже значајно убрзање у односу на извршавање симулације на једном уређају.

На Слици 5.3. дато је поређење резултата добијених коришћењем Тесла графичких картица из литературе (*Cavity flow* (Obrecht et al., 2013), *Porous flow* (Huang et al., 2014), *Poiseuille flow* (Huang et al., 2014)) са резултатима презентованим у тези (*Алгоритам 1*, *Алгоритам 2*). У литератури, за истовремено извршавање LBM алгоритма на више Тесла графичких картица, доступни су резултати само за један модел и једну величину мреже (*D3Q19* модел 192^3, са приближно 7000000 тачака) као и једну врсту Тесла графичких картица. Приликом израде ове дисертације коришћена је друга врста Тесла графичких картица (једино доступних у том тренутку) и други тип модела мреже али на приближно истој величини мреже, а у циљу

поређења перформанси са резултатима доступним у литератури. У тези је имплементиран 2D модел (*D2Q9*) па је одабрана одговарајућа величина мреже (2560^2) са којом се добија нешто мањи број тачака. За *Алгоритам 1* и *Алгоритам 2* приказани су резултати само за наведену величину мреже за један и два уређаја. Са графика се види да *Алгоритам 1* и *Алгоритам 2* представљени у тези постижу задовољавајуће перформансе када се постигнути резултати пореде са резултатима из литературе.



Слика 5.3. Поређење резултата добијених у тези са резултатима из литературе

5.2. Конфигурација 2

Конфигурацију 2 чине графичка картица *AMD Radeon (TM) R5 M430* и процесор *Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz*. Процесор истовремено игра улогу *HOST* уређаја и уређаја на коме се врши паралелно израчунавање. *Конфигурацију 2* могуће је тестирати на два начина. Пошто *AMD OpenCL* платформа подржава *Intel*-ове процесоре могуће је извршити тестирање уређаја користећи *AMD* платформу за оба уређаја, креирањем само једног контекста, или тестирати уређаје тако да сваки од уређаја користи своју оригиналну имплементацију за *OpenCL* (платформу) уз креирање два контекста.

Карактеристике уређаја за *AMD* платформу дате су у Табели 5.7. Карактеристике уређаја за оригиналне платформе дате су у Табели 5.8. Оригинална *Intel* платформа за процесор је за 50% бржа од *AMD* платформе, а графичка картица је за 300% бржа у односу на процесор уколико се користи *AMD* платформа, односно око 170% бржа када се за процесор користи *Intel*-ова платформа.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	AMD Radeon (TM) R5 M430	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
CL_DEVICE_TYPE	GPU	CPU
CL_PLATFORM_VENDOR	Advanced Micro Devices, Inc.	Advanced Micro Devices, Inc.
CL_PLATFORM_VERSION	OpenCL 2.0 AMD-APP (2117.13)	OpenCL 2.0 AMD-APP (2117.13)
CL_DEVICE_VERSION	OpenCL 1.2 AMD-APP (2117.13)	OpenCL 1.2 AMD-APP (2117.13)
CL_DEVICE_VENDOR	Advanced Micro Devices, Inc.	GenuineIntel
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.2	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.7. *Палтформа 2 AMD* имплементација платформе

	Уређај 1	Уређај 2
CL_DEVICE_NAME	AMD Radeon (TM) R5 M430	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
CL_DEVICE_TYPE	GPU	CPU
CL_PLATFORM_VENDOR	Advanced Micro Devices, Inc.	Intel(R) Corporation
CL_PLATFORM_VERSION	OpenCL 2.0 AMD-APP (2117.13)	OpenCL 2.0
CL_DEVICE_VERSION	OpenCL 1.2 AMD-APP (2117.13)	OpenCL 2.0 (Build 359)
CL_DEVICE_VENDOR	Advanced Micro Devices, Inc.	Intel(R) Corporation
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.2	OpenCL C 2.0
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.8. *Платформа 2* комбинација *AMD* имплементације платформе за *GPU* и *Intel* имплементације платформе за *CPU*

5.2.1. AMD имплементација платформе

1. *Алгоритам 1*

   У Табели 5.9. приказане су *MLUPS* вредности извршавања основног алгоритма. Из табеле се види да су перформансе сличне за различите структуре података али да се ипак најбољи резултати постижу када се за пренос података користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	152	161	164	165	166	166	167	167
Subbuffer	153	164	165	167	167	168	169	169
Copy	152	163	166	167	167	169	170	170

Табела 5.9. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.10. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. Из табеле се види да симулација достиже највећу брзину када се користи *Subbuffer* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	52	80	110	129	129	131	131	131
Subbuffer	106	113	117	129	129	133	133	135
Copy	91	113	113	129	129	131	131	131

Табела 5.10. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

   Комбинација *AMD* графичке картице и *Intel*-овог процесора за *Алгоритам 3* који користи локалну меморију и два локална бројача даје веома лоше резултате. За мање мреже постигнуте брзине су спорије од брзина извршавања на само једном уређају, а за велике мреже једнаке или незнатно веће од брзине које се постижу коришћењем само једног уређаја.

4. *Алгоритам 4*

   У Табели 5.11. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Pointer* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	100	146	155	148	157	157	157	157
Subbuffer	101	100	100	101	101	101	102	102
Copy	102	102	102	102	102	102	103	103

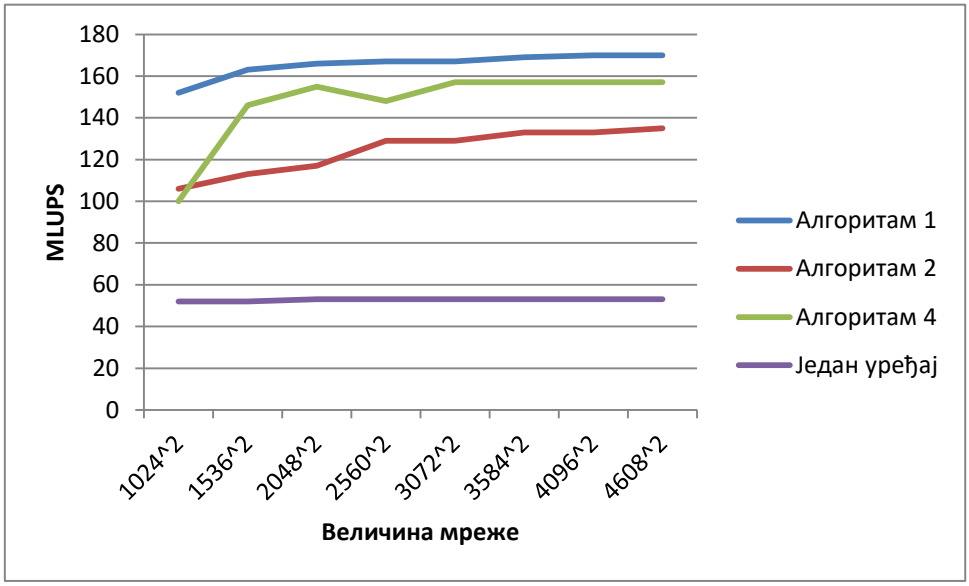
Табела 5.11. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

У Табели 5.12. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај AMD Radeon (TM) R5.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	152	163	166	167	167	169	170	170
Алгоритам 2	106	113	117	129	129	133	133	135
Алгоритам 4	102	102	102	102	102	102	103	103
Један уређај	52	52	53	53	53	53	53	53

Табела 5.12. Поређење различитих алгоритама који користе једну платформу за импеплементацију симулације тока флуида за *Конфигурацију 2*

На Слици 5.4. дат је графички приказ Табеле 5.12.



Слика 5.4. Графички приказ Табеле 5.12.

На основу графичког приказа резултата види се да најбоље перформансе за *Конфигурацију 2* (на *AMD* платформи) постиже основни алгоритам - *Алгоритам 1*, док *Алгоритам 4* постиже нешто мање, али ипак значајно убрзање у односу на брзину симулације на једном уређају.

5.2.2. Комбинација AMD имплементације за GPU и Intel имплементације платформе за CPU

1. *Алгоритам 1*

   У Табели 5.13. приказане су брзине извршавања основног алгоритма за два контекста. Из табеле се види да су постигнуте брзине сличне за различите структуре података, највећу брзину алгоритам постиже када се за пренос података користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	53	53	53	54	56	56	56	56
Subbuffer	52	53	53	54	55	55	55	55
Copy	55	56	56	56	57	57	57	57

Табела 5.13. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.14. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси уз коришћење два контекста. Перформансе које постиже *Алгоритам 2* за различите структуре података су сличне, *Subbuffer* структура постиже нешто боље перформансе.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	61	62	63	63	64	66	66	68
Subbuffer	69	70	71	73	74	74	77	79
Copy	70	70	70	70	71	71	72	72

Табела 5.14. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

   Комбинација *AMD* графичке картице и *Intel*-овог процесора за *Алгоритам 3*, који користи локалну меморију, два локална бројача и два контекста даје веома лоше резултате. За мање мреже постигнуте брзине су спорије од брзина извршавања на само једном уређају, а за велике мреже незнатно веће од брзине која се постиже коришћењем само једног уређаја.

4. *Алгоритам 4*

   У Табели 5.15. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних

променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	72	72	73	74	74	77	80	80
Subbuffer	71	71	72	72	74	74	77	77
Copy	75	75	77	77	80	80	85	85

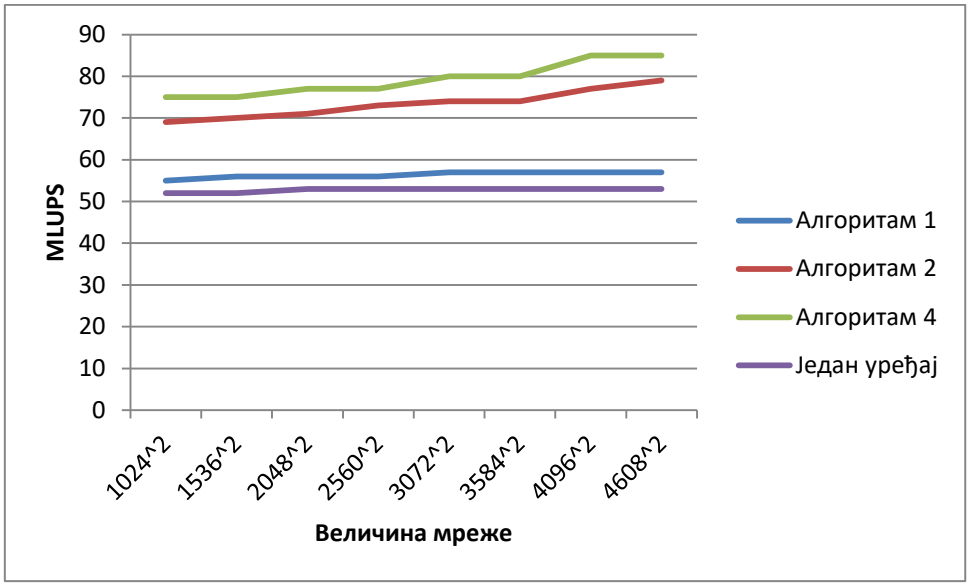
Табела 5.15. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

У Табели 5.16. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај AMD Radeon (TM) R5 M430.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	55	56	56	56	57	57	57	57
Алгоритам 2	69	70	71	73	74	74	77	79
Алгоритам 4	75	75	77	77	80	80	85	85
Један уређај	52	52	53	53	53	53	53	53

Табела 5.16. Поређење различитих алгоритама који користе две платформе за имеплементцију симулације тока флуида за *Конфигурацију 2*

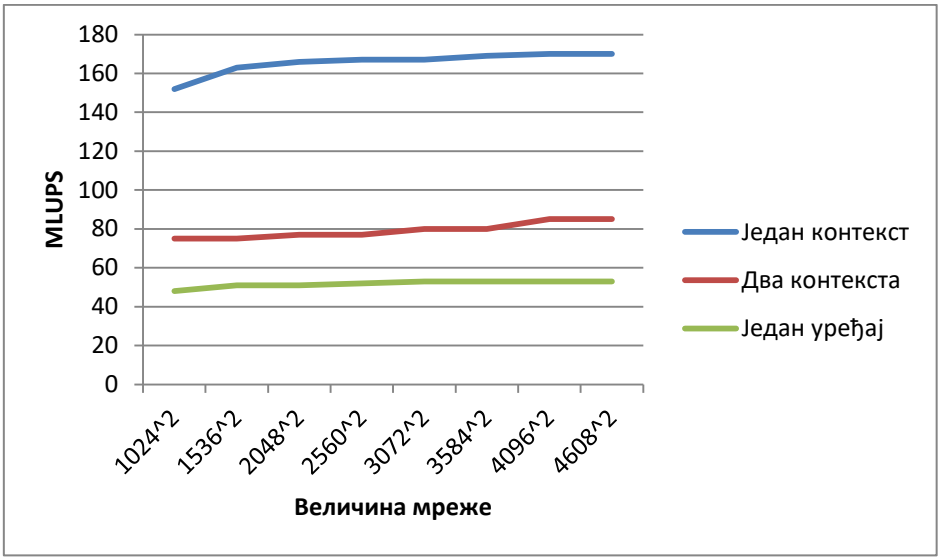
На Слици 5.5. дат је графички приказ Табеле 5.16.



Слика 5.5. Графички приказ Табеле 5.16.

На основу графичког приказа резултата види се да најбоље перформансе за *Конфигурацију 2* која користи оригиналне платформе за оба уређаја постиже *Алгоритам 4* који има смањен број помоћних променљивих. Убрзање које се постиже је значајно и износи око 50%.

На Слици 5.6. дато је поређење брзина за један уређај, два уређаја тестирана на *AMD* платформи и два уређаја уз коришћење оригиналних платформи за оба уређаја постигнутих за *Конфигурацију 2*. Са графика се види да се најбољи резултати постижу када се користи један контекст. Перформансе алгоритма који користи један контекст су 100% бољи у односу на перформансе алгоритма које се постижу када постоје два контекста. Иако *AMD* платформа не може да омогући постизање најбољих перформанси за *Intel*-ов процесор, комбинација процесора са графичком картицом и једним контекстом постиже знатно већу брзину у односу на коришћење два контекста јер је размена података између уређаја код једног контекста знатно бржа и једноставнија у односу на размену података када постоји више инстанци контекста. Такође са слике се види да и алгоритам који користи два контекста ипак постиже значајно убрзање у односу на брзину извршавања симулације на једном уређају.



Слика 5.6. Поређење брзина за исту конфигурацију и различит број контекста.

5.3. Конфигурација 3

Конфигурација 3 састоји се из интегрисане графичке картице *Intel(R) HD Graphics 520* и графичке картице *AMD Radeon (TM) R5 M430*. Ова два уређаја могу бити тестирана само коришћењем два контекста и две платформе, у Табели 5.17. дате су карактеристике уређаја.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	Intel(R) HD Graphics 520	AMD Radeon (TM) R5 M430
CL_DEVICE_TYPE	GPU	GPU
CL_PLATFORM_VENDOR	Intel(R) Corporation	Advanced Micro Devices, Inc.
CL_PLATFORM_VERSION	OpenCL 2.0	OpenCL 2.0 AMD-APP (2117.13)
CL_DEVICE_VERSION	OpenCL 2.0	OpenCL 1.2 AMD-APP (2117.13)
CL_DEVICE_VENDOR	Intel(R) Corporation	Advanced Micro Devices, Inc.
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 2.0	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.17. Карактеристике *Конфигурације 3*

1. *Алгоритам 1*

   У Табели 5.18. приказане су *MLUPS* вредности извршавања основног алгоритма. Структуре података које се користе за пренос података на уређаје не утичу на побољшање перформанси, из табеле може да се закључи да незнатно боље резулате бележи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	42	51	53	54	56	58	58	58
Subbuffer	47	51	53	54	55	57	58	58
Copy	50	52	54	56	57	58	59	59

Табела 5.18. MLUPS вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.19. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. Из табеле се види да су перформансе за различите структуре података готово идентичне.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	63	64	64	66	66	69	69	69
Subbuffer	59	64	64	67	68	69	69	69
Copy	63	64	64	66	66	69	69	69

Табела 5.19. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

   Алгоритам 3 који користи локалну меморију и два локална бројача даје веома лоше резултате за *Конфигурацију 3*. За мање мреже постигнуте брзине су спорије од брзина извршавања на само једном уређају, а за велике мреже једнаке или незнатно веће од брзине која се постиже коришћењем само једног уређаја.

4. *Алгоритам 4*

   У Табели 5.19. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	63	64	66	67	67	68	71	75
Subbuffer	68	70	71	71	71	73	73	75
Copy	70	72	73	73	74	76	81	81

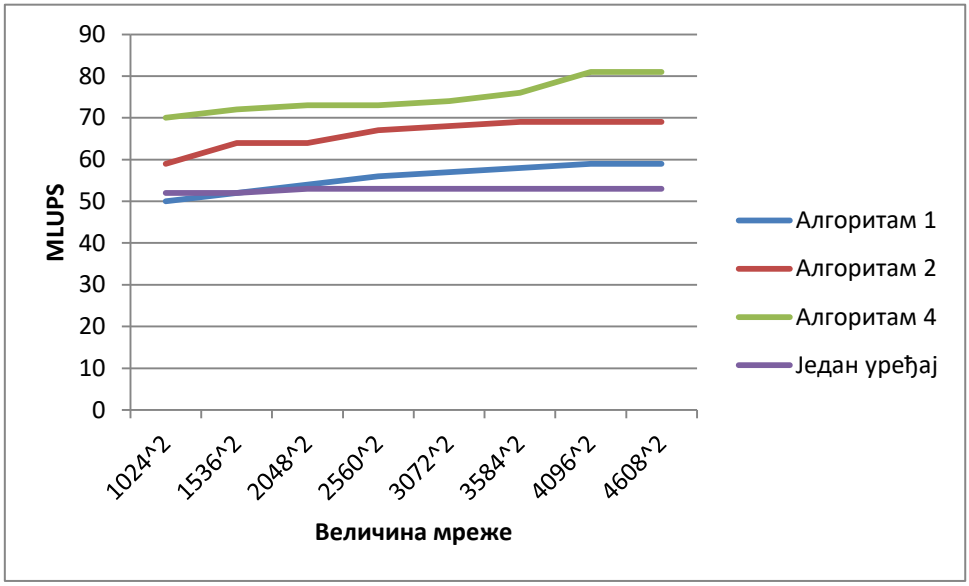
Тебела 5.19. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

У Табели 5.20. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај AMD Radeon (TM) R5 M430.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	50	52	54	56	57	58	59	59
Алгоритам 2	59	64	64	67	68	69	69	69
Алгоритам 4	70	72	73	73	74	76	81	81
Један уређај	52	52	53	53	53	53	53	53

Табела 5.20. Поређење различитих алгоритама за *Конфигурацију 3*

На Слици 5.7. дат је графички приказ Табеле 5.20.



Слика 5.7. Графички приказ Табеле 5.20.

На основу графичког приказа резултата види се да најбоље перформансе на *Конфигурацији 3* постиже *Алгоритам 4* - модификација основног алгоритма смањењем броја помоћних променљивих.

5.4. Кофигурација 4

Конфигурацију 4 чине графичка картица *AMD Radeon (TM) R5 M420* и процесор *Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz*. Процесор *Конфигурације 4* истовремено је *HOST* уређај и уређај на коме се врши паралелно израчунавање. Конфигурацију је могуће тестирати на два начина: у првом случају користи се *AMD* платформа за оба уређаја и креира само један контекст, у другом случају користе се посебне платформе за оба уређаја уз креирање два контекста. Карактеристике уређаја за *AMD* платформу дате су у Табели 5.21., а карактеристике уређаја у случају коришћења оригиналних платформи дате су у Табели 5.22.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	AMD Radeon (TM) R5 M420	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
CL_DEVICE_TYPE	GPU	CPU
CL_PLATFORM_VENDOR	Advanced Micro Devices, Inc.	Advanced Micro Devices, Inc.
CL_PLATFORM_VERSION	OpenCL 2.0 AMD-APP (2348.3)	OpenCL 2.0 AMD-APP (2348.3)
CL_DEVICE_VERSION	OpenCL 1.2 AMD-APP (2348.3)	OpenCL 1.2 AMD-APP (2348.3)
CL_DEVICE_VENDOR	Advanced Micro Devices, Inc.	GenuineIntel
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.2	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.21. *Платформа 2 AMD* имплементација платформе

	Уређај 1	Уређај 2
CL_DEVICE_NAME	AMD Radeon (TM) R5 M420	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
CL_DEVICE_TYPE	GPU	CPU
CL_PLATFORM_VENDOR	Advanced Micro Devices, Inc.	Intel(R) Corporation
CL_PLATFORM_VERSION	OpenCL 2.0 AMD-APP (2348.3)	OpenCL 2.1
CL_DEVICE_VERSION	OpenCL 1.2 AMD-APP (2348.3)	OpenCL 2.1 (Build 2)
CL_DEVICE_VENDOR	Advanced Micro Devices, Inc.	Intel(R) Corporation
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.2	OpenCL C 2.0
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.22. *Платформа 2* комбинација *AMD* имплементације за *GPU* и *Intel* имплементације платформе за *CPU*

5.4.1. AMD имплементација платформе

1. *Алгоритам 1*

   У Табели 5.23. приказане су *MLUPS* вредности извршавања основног алгоритма. На основу вредности добијених тестирањем симулације за различите структуре података може се закључити да се постижу слични резултати за све структуре, а да *Copy* структура постиже мало боље резултате од друге две структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	127	128	130	132	132	132	132	132
Subbuffer	129	129	131	131	132	132	132	132
Copy	131	132	132	132	134	134	136	136

Табела 5.23. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.24. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. Из табеле се види да се највећа брзина симулације постиже када се користи *Pointer* структура података.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	97	100	98	107	110	112	114	114
Subbuffer	84	98	98	102	102	102	102	102
Copy	84	97	98	101	101	106	106	107

Табела 5.24. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

   Комбинација *AMD* графичке картице и Intelovog процесора за *Алгоритам 3* који користи локалну меморију и два локална бројача даје веома лоше резултате. За мање мреже постигнуте брзине су спорије од брзина извршавања на само једном уређају, а за велике мреже једнаке или незнатно веће од брзине која се постиже коришћењем само једног уређаја.

4. *Алгоритам 4*

   У Табели 5.25. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Pointer* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	102	103	105	106	107	109	109	112
Subbuffer	85	87	89	92	92	95	96	100
Copy	84	87	90	91	92	92	94	99

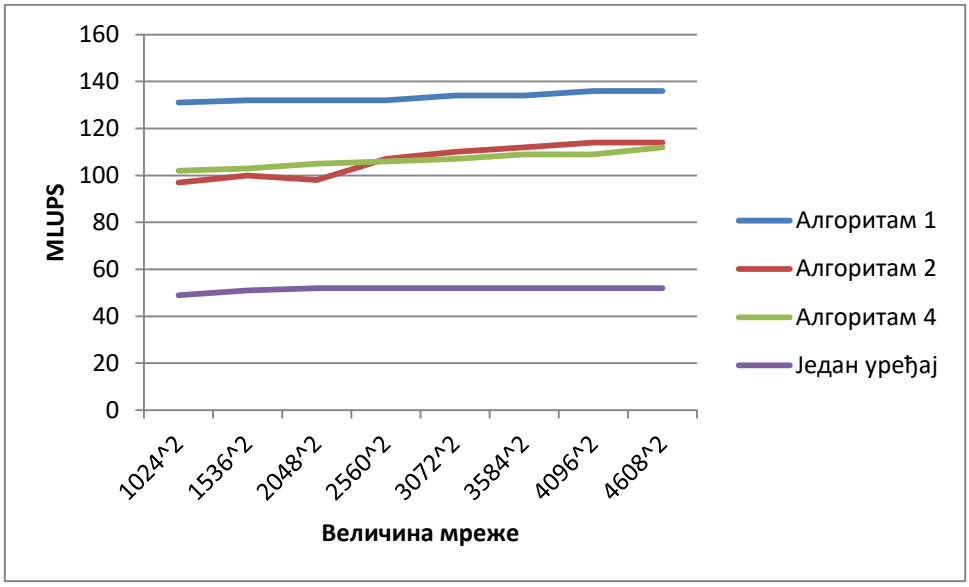
Табела 5.25. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

У Табели 5.26. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај AMD Radeon (TM) R5 M420.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	131	132	132	132	134	134	136	136
Алгоритам 2	97	100	98	107	110	112	114	114
Алгоритам 4	102	103	105	106	107	109	109	112
Један уређај	49	51	52	52	52	52	52	52

Табела 5.26. Поређење различитих алгоритама који користе *AMD* платформу за *Конфигурацију 4*

На Слици 5.8. дат је графички приказ Табеле 5.26.



Слика 5.8. Графички приказ Табеле 5.26.

На основу графичког приказа резултата види се да најбоље перформансе на *Платформи 4* када се користи *AMD* платформа постиже основни алгоритам - *Алгоритам 1*.

5.4.2. Комбинација AMD имплементације за GPU и Intel имплементације платформе за CPU

1. *Алгоритам 1*

   У Табели 5.27. приказане су брзине извршавања основног алгоритма за два контекста. Из табеле се види да су постигнуте брзине сличне за различите структуре података, за нијансу већу брзину алгоритам постиже када се користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	51	56	55	57	56	58	56	57
Subbuffer	50	55	56	58	56	58	56	57
Copy	51	56	55	57	57	58	57	58

Табела 5.27. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.28. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси уз коришћење два контекста. Из табеле се види да *Subbuffer* структура података постиже најбоље перформансе.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	63	73	78	80	80	80	80	81
Subbuffer	75	79	79	82	82	82	82	84
Copy	65	72	78	78	79	79	79	80

Табела 5.28. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

   Комбинација *AMD* графичке картице и *Intelovog* процесора за *Алгоритам 3* који користи локалну меморију, два локална бројача и два контекста даје веома лоше резултате. За мање мреже постигнуте брзине су спорије од брзина извршавања на само једном уређају, а за велике мреже незнатно веће од брзине која се постиже коришћењем само једног уређаја.

4. *Алгоритам 4*

   У Табели 5.29. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користе копије за парцијалне податке.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	59	60	61	62	63	63	64	64
Subbuffer	58	59	59	61	63	63	63	63
Copy	61	63	64	67	67	67	68	68

Табела 5.29. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

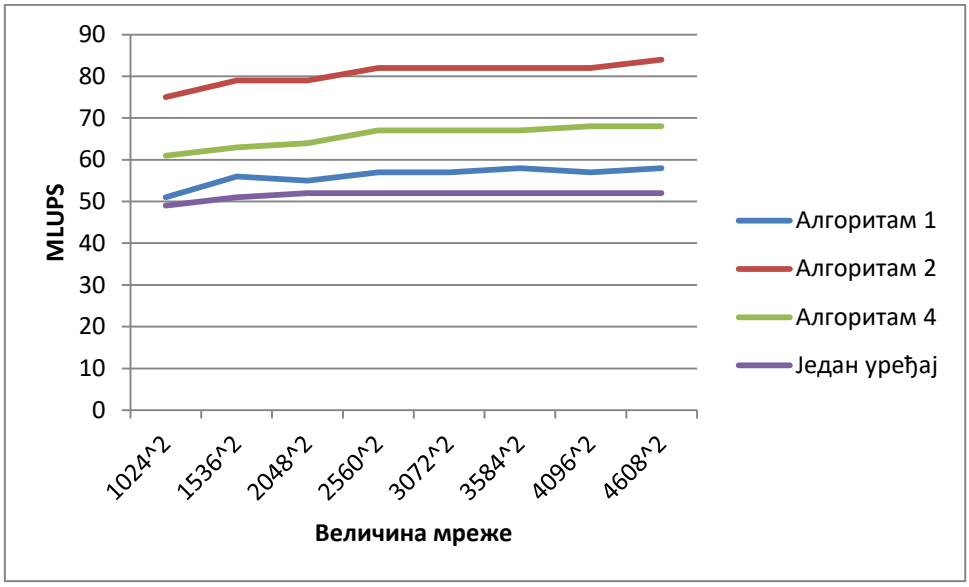
На основу приказаних резултата може се закључити да уколико се не користи локална меморија (*Алгоритам 1* и *Алгоритам 4*) највећа брзина алгоритма постиже се када се за пренос података користи *Copy* структура, док се у случају коришћења локалне меморије (*Алгоритам 2*) најбоље перформансе постижу коришћењем *Subbbuffer*-а за пренос података на уређаје. *Алгоритам 3* није погодан за *AMD* графичке картице.

У Табели 5.30. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај AMD Radeon (TM) R5 M420.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	51	56	55	57	57	58	57	58
Алгоритам 2	75	79	79	82	82	82	82	84
Алгоритам 4	61	63	64	67	67	67	68	68
Један уређај	49	51	52	52	52	52	52	52

Табела 5.30. Поређење различитих алгоритама који користе две платформе за имплементцију симулације тока флуида за *Конфигурацију 4*

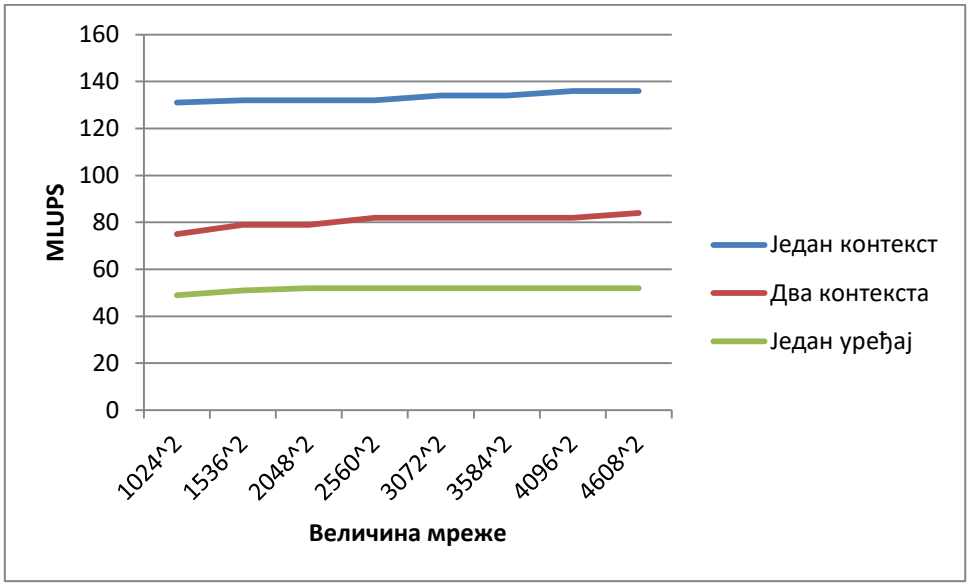
На Слици 5.9. дат је графички приказ Табеле 5.30.



Слика 5.9. Графички приказ Табеле 5.30.

На основу графичког приказа резултата види се да најбоље перформансе за *Конфигурацију 4* која користи оригиналне платформе за оба уређаја постиже *Алгоритам 2* који користи локалну меморију са једним локалним бројачем.

На Слици 5.10. дато је поређење брзина: за један уређај, два уређаја тестирана на *AMD* платформи и два уређаја уз коришћење оригиналних платформи. Са графика се види да се знатно бољи резултати постижу када се користи један контекст, у односу на коришћење два контекста јер је размена података између уређаја код једног контекста знатно бржа и једноставнија него уколико постоји више инстанци контекста. Такође из података са графика може се закључити да и алгоритам који користи два контекста постиже заначајно убрзање у односу на брзину извршавања симулације на једном уређају.



Слика 5.10. Поређење брзина за исту конфигурацију и различит број контекста.

5.5. Конфигурација 5

Конфигурација 5 састоји се из интегрисане графичке картице *Intel(R) HD Graphics 620* и графичке картице *AMD Radeon (TM) R5 M420*. Ова два уређаја могу бити тестирана само коришћењем два контекста и две платформе, у Табели 5.31. дате су карактеристике уређаја.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	Intel(R) HD Graphics 620	AMD Radeon (TM) R5 M420
CL_DEVICE_TYPE	GPU	GPU
CL_PLATFORM_VENDOR	Intel(R) Corporation	Advanced Micro Devices, Inc.
CL_PLATFORM_VERSION	OpenCL 2.1	OpenCL 2.0 AMD-APP (2348.3)
CL_DEVICE_VERSION	OpenCL 2.1	OpenCL 1.2 AMD-APP (2348.3)
CL_DEVICE_VENDOR	Intel(R) Corporation	Advanced Micro Devices, Inc.
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 2.0	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.31. Карактеристике Конфигурације 5.

Алгоритам 1

У Табели 5.32. приказане су *MLUPS* вредности извршавања основног алгоритма. Из табеле се види да симулација постиже нешто бољу брзину када се за структуру података користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	57	69	69	69	70	71	71	71
Subbuffer	59	68	68	72	73	74	75	75
Copy	59	69	69	73	74	74	78	78

Табела 5.32. *MLUPS* вредности извршавања основног алгоритма

1. *Алгоритам 2*

   У Табели 5.33. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. Из табеле се види да симулација постиже највећу брзину када се за пренос података на уређаје за паралелно извршавање користи Pointer структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	55	62	64	64	68	69	72	72
Subbuffer	55	62	63	63	65	66	66	66
Copy	55	61	62	62	63	63	66	66

Табела 5.33. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по x оси

2. *Алгоритам 3*

   Алгоритам 3 који користи локалну меморију и два локална бројача даје веома лоше резултате за *Конфигурацију 5*. За мање мреже постигнуте брзине су спорије од брзина извршавања на само једном уређају, а за велике мреже једнаке или незнатно веће од брзине које се постижу коришћењем само једног уређаја.

3. *Алгоритам 4*

   У Табели 5.34. приказани су резултати извршавања симулације за алгоритам са смањеним

бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Pointer* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	85	85	86	87	87	89	92	95
Subbuffer	74	77	79	80	83	84	85	86
Copy	74	78	80	81	84	84	86	87

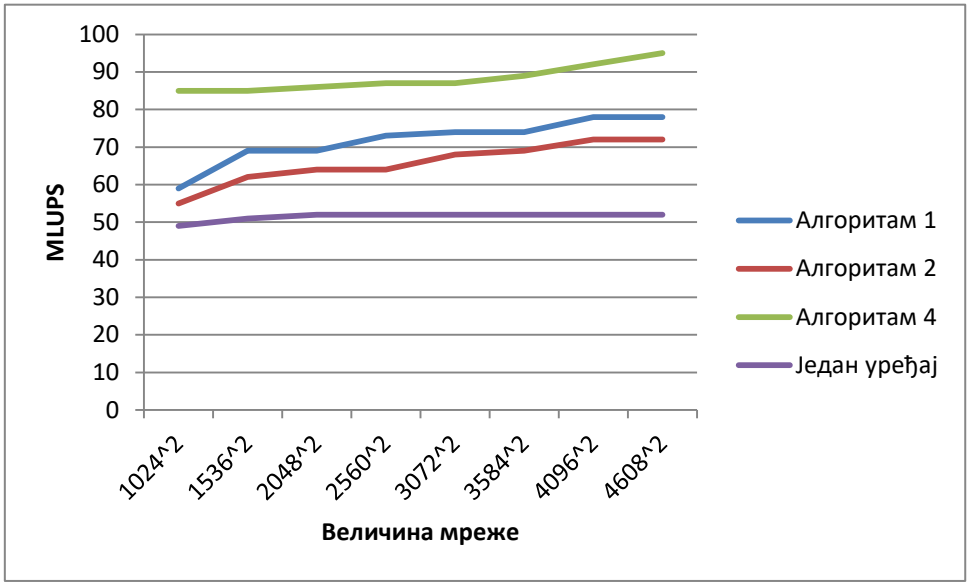
Тебела 5.34. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

У Табели 5.35. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај AMD Radeon (TM) R5 M420.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	59	69	69	73	74	74	78	78
Алгоритам 2	55	62	64	64	68	69	72	72
Алгоритам 4	85	85	86	87	87	89	92	95
Један уређај	49	51	52	52	52	52	52	52

Табела 5.35. Поређење различитих алгоритама за *Конфигурацију 5*

На Слици 5.11. дат је графички приказ Табеле 5.35.



Слика 5.11. Графички приказ Табеле 5.35.

На основу графичког приказа резултата види се да најбоље перформансе на *Конфигурацији 5* постиже *Алгоритам* 4 - модификација основног алгоритма смањењем броја помоћних променљивих.

5.6. Конфигурација 6

Конфигурација 6 састоји се из интегрисане графичке картице *Intel(R) HD Graphics 530* и графичке картице *AMD Radeon Pro 455 Compute Engine*. Уређаји се налазе на Apple платформи и из тог разлога без обзира на различите произвођаче могу бити тестирани у оквиру једног контекста, у Табели 5.36. дате су карактеристике уређаја. Пошто уређаји немају глобални *cash* на овим уређајима није могуће користити локалну меморију.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	Intel(R) HD Graphics 530	AMD Radeon Pro 455 Compute Engine
CL_DEVICE_TYPE	GPU	GPU
CL_PLATFORM_VENDOR	Apple	Apple
CL_PLATFORM_VERSION	OpenCL 1.2 (May 24 2018 20:07:03)	OpenCL 1.2 (May 24 2018 20:07:03)
CL_DEVICE_VERSION	OpenCL 1.2	OpenCL 1.2
CL_DEVICE_VENDOR	Intel Inc.	AMD
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.2	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	None	None

Табела 5.36. Карактеристике *Конфигурације 6*

1. *Алгоритам 1*

   У Табели 5.37. приказане су *MLUPS* вредности извршавања основног алгоритма. Симулација постиже највећу брзину када се користи *Subbuffer* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	429	432	435	438	440	443	443	448
Subbuffer	447	450	450	452	453	457	460	465
Copy	431	436	440	445	445	447	449	450

Табела 5.37. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 4*

У Табели 5.38. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Subbuffer* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	329	332	337	340	345	348	350	352
Subbuffer	365	370	374	375	379	385	387	392
Copy	361	364	370	373	376	380	382	386

Тебела 5.38. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

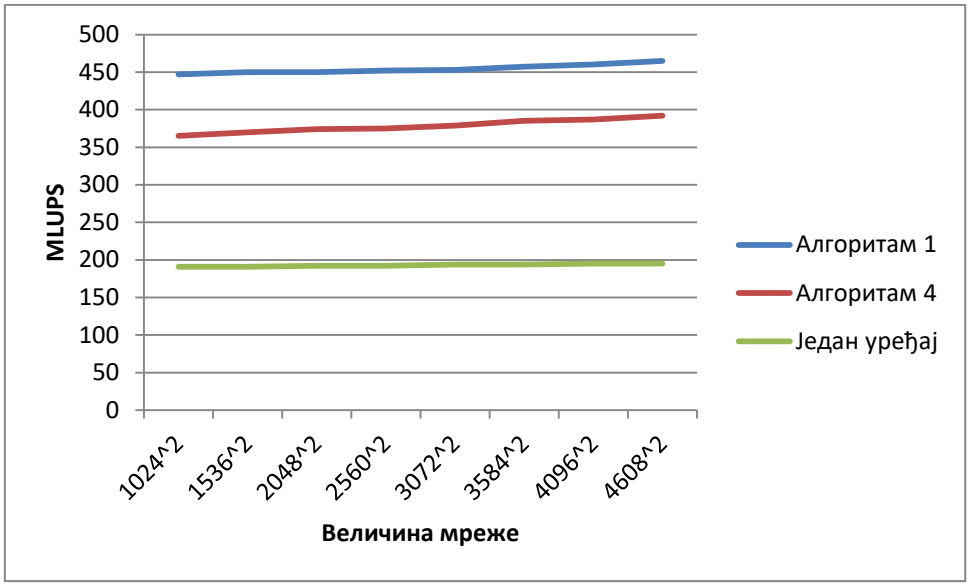
За оба посматрана алгоритма највеће брзине се постижу коришћењем *Subbuffer* структуре података за пренос података на уређаје.

У Табели 5.39. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је *GPU* уређај AMD Radeon Pro 455 Compute Engine.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	447	450	450	452	453	457	460	465
Алгоритам 2	365	370	374	375	379	385	387	392
Један уређај	191	191	192	192	194	194	195	195

Табела 5.39. Поређење различитих алгоритама за *Конфигурацију 6*

На Слици 5.12. дат је графички приказ Табеле 5.39.



Слика 5.12. Графички приказ Табеле 5.39.

На основу графичког приказа резултата види се да у оба случаја извршавање симулације на више уређаја постиже значајно убрзање у односу на извршавање симулације на једном уређају. Најбоље перформансе постижу се помоћу *Алгоритма 1*.

5.7. Конфигурација 7

Конфигурација 7 састоји се из интегрисане графичке картице *Intel(R) HD Graphics 5500* и графичке картице *GeForce 940M@ 2.40GHz*. Ова два уређаја могу бити тестирана само коришћењем два контекста и две платформе, у Табели 5.40. дате су карактеристике уређаја.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	Intel(R) HD Graphics 5500	GeForce 940M@ 2.40GHz
CL_DEVICE_TYPE	GPU	GPU
CL_PLATFORM_VENDOR	Intel(R) Corporation	NVIDIA Corporation
CL_PLATFORM_VERSION	OpenCL 2.0	OpenCL 1.2 CUDA 8.0.0
CL_DEVICE_VERSION	OpenCL 2.0	OpenCL 1.2 CUDA
CL_DEVICE_VENDOR	Intel(R) Corporation	NVIDIA Corporation
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 2.0	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.40. Карактеристике *Конфигурације 7*

1. *Алгоритам 1*

   У Табели 5.41. приказане су *MLUPS* вредности извршавања основног алгоритма. На основу података из табеле види се да се највећа брзина извршавања симулације постиже коришћењем *Copy* структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	54	57	60	64	68	70	75	79
Subbuffer	50	53	58	62	66	68	72	78
Copy	57	62	64	68	71	74	79	84

Табела 5.41. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

У Табели 5.42. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. На основу података из табеле види се да се највеће брзине извршавања симулације постижу коришћењем *Copy* структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	63	65	68	71	74	77	82	85
Subbuffer	61	62	65	69	72	76	81	85
Copy	68	71	72	74	76	80	84	90

Табела 5.42 *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. *Алгоритам 3*

У Табели 5.43. Приказани су резултати извршавања алгоритма који користи локалну меморију и два локална бројача, симулација постиже највећу брзину када се користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	71	73	76	82	84	86	89	91
Subbuffer	80	81	85	86	88	89	92	94
Copy	80	82	85	87	90	93	97	101

Табела 5.43. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и два локална бројача

4. *Алгоритам 4*

У Табели 5.44. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	57	60	61	63	64	67	70	71
Subbuffer	57	57	59	60	62	65	66	68
Copy	57	58	60	64	67	70	72	75

Тебела 5.44. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

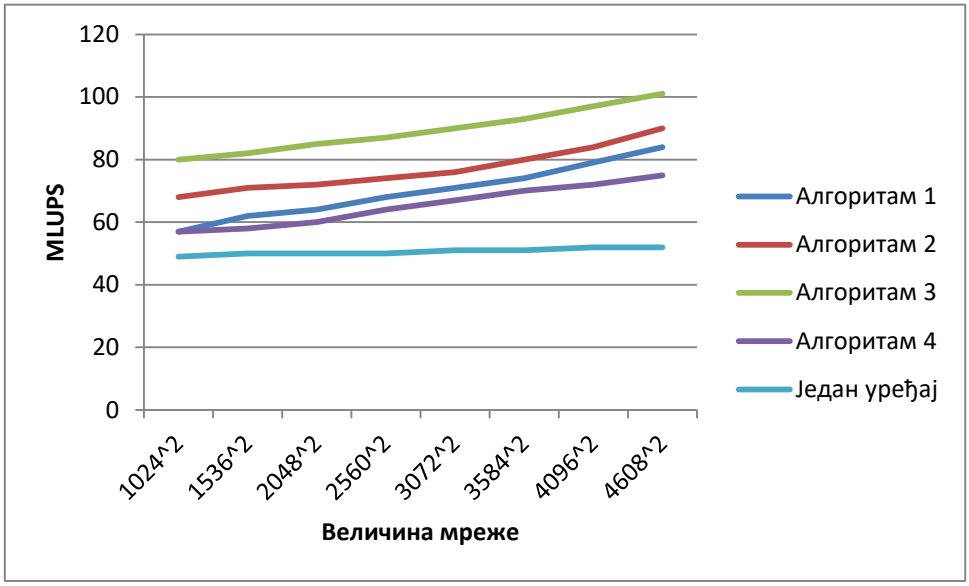
На основу приказаних резултата може се закључити да је за посматрану конфигурацију најбољи избор *Copy* структура.

У Табели 5.45. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је GPU уређај GeForce 940M@ 2.40GHz.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	57	62	64	68	71	74	79	84
Алгоритам 2	68	71	72	74	76	80	84	90
Алгоритам 3	80	82	85	87	90	93	97	101
Алгоритам 4	57	58	60	64	67	70	72	75
Један уређај	49	50	50	50	51	51	52	52

Табела 5.45. Поређење различитих алгоритама за *Конфигурацију 7*

На Слици 5.13. дат је графички приказ Табеле 5.45.



Слика 5.13. Графички приказ Табеле 5.45.

На основу графичког приказа резултата види се да најбоље перформансе на *Конфигурацији 7* постиже *Алгоритам 3* који користи локалну меморију и два локална бројача.

5.8. Конфигурација 8

Конфигурација 8 састоји се из процесора *Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz*, који је истовремено *HOST* уређај и уређај за паралелно рачунање и графичке картице *GeForce 940M@ 2.40GHz*. Ова два уређаја могу бити тестирана само коришћењем два контекста и две платформе, у Табели 5.46 дате су карактеристике уређаја.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz	GeForce 940M@ 2.40GHz
CL_DEVICE_TYPE	CPU	GPU
CL_PLATFORM_VENDOR	Intel(R) Corporation	NVIDIA Corporation
CL_PLATFORM_VERSION	OpenCL 2.0	OpenCL 1.2 CUDA 8.0.0
CL_DEVICE_VERSION	OpenCL 2.0 (Build 10094)	OpenCL 1.2 CUDA
CL_DEVICE_VENDOR	Intel(R) Corporation	NVIDIA Corporation
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 2.0	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.46. Карактеристике *Конфигурације 8*

1. *Алгоритам 1*

   У Табели 5.47. приказане су *MLUPS* вредности извршавања основног алгоритма. Структура података која даје највећу брзину извршавања симулације је *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	57	59	61	65	67	69	73	75
Subbuffer	59	60	63	66	69	71	73	77
Copy	61	62	65	67	70	74	77	81

Табела 5.47. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.48. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. На основу података из табеле види се да се највеће брзине извршавања

симулације постижу коришћењем *Copy* структуре података.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	62	63	66	72	73	77	80	82
Subbuffer	61	63	65	69	71	76	78	80
Copy	62	64	67	72	74	81	87	89

Табела 5.48. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. Алгоритам 3

У табели 5.49. приказани су резултати извршавања алгоритма који користи локалну меморију и два локална бројача, најбоље перформансе симулација постиже када се користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	73	75	77	78	80	83	86	87
Subbuffer	74	75	77	80	83	85	88	90
Copy	75	76	81	85	87	90	92	97

Табела 5.49. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и два локална бројача

4. Алгоритам 4

У Табели 5.50. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	57	58	60	61	63	65	67	68
Subbuffer	55	56	58	60	62	64	65	67
Copy	59	60	60	61	66	68	71	73

Тебела 5.50. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

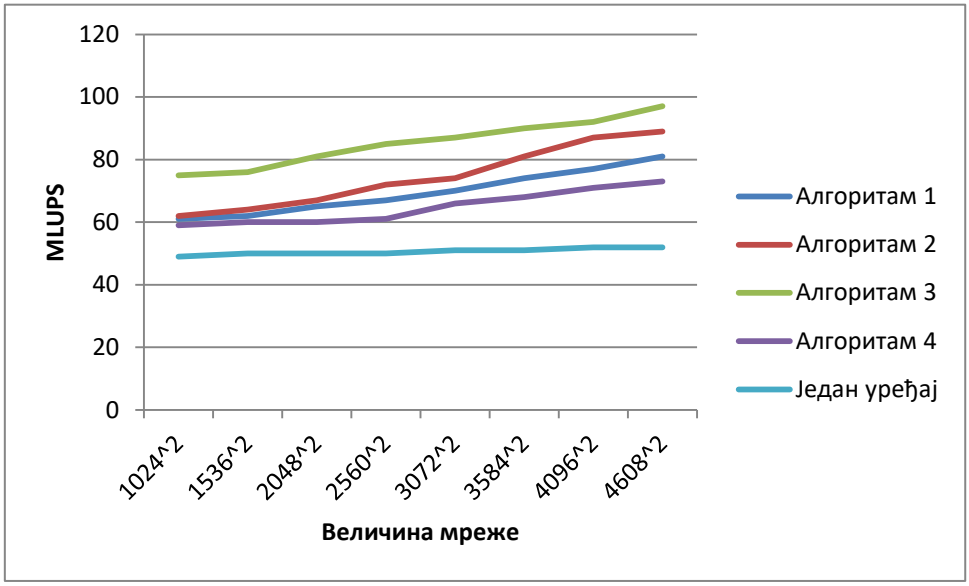
У сва четири размотрена случаја највеће брзине симулације постижу се уколико се користи *Copy* структура података.

У Табели 5.51. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају. За извршавање симулације на једном уређају узима се уређај који постиже веће убрзање симулације, у овом случају то је *GPU* уређај GeForce 940M@ 2.40GHz.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	61	62	65	67	70	74	77	81
Алгоритам 2	62	64	67	72	74	81	87	89
Алгоритам 3	75	76	81	85	87	90	92	97
Алгоритам 4	59	60	60	61	66	68	71	73
Један уређај	49	50	50	50	51	51	52	52

Табела 5.51. Поређење различитих алгоритама за *Конфигурацију 8*

На Слици 5.14. дат је графички приказ Табеле 5.51.



Слика 5.14. Графички приказ Табеле 5.51.

На основу графичког приказа резултата види се да најбоље перформансе за *Конфигурацију 8* постиже *Алгоритам 3* који користи локалну меморију и два локална бројача.

5.9. Конфигурација 9

Конфигурација 9 је чвор *AXIOM* кластера Природно-математичког факултета у Новом Саду, састоји се из две *NVIDIA* графичке картице *GeForce GTX 960*, у Табели 5.52 дате су карактеристике уређаја.

	Уређај 1	Уређај 2
CL_DEVICE_NAME	GeForce GTX 960	GeForce GTX 960
CL_DEVICE_TYPE	GPU	GPU
CL_PLATFORM_VENDOR	NVIDIA Corporation	NVIDIA Corporation
CL_PLATFORM_VERSION	OpenCL 1.2 CUDA 10.1.105	OpenCL 1.2 CUDA 10.1.105
CL_DEVICE_VERSION	OpenCL 1.2 CUDA	OpenCL 1.2 CUDA
CL_DEVICE_VENDOR	NVIDIA Corporation	NVIDIA Corporation
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.2	OpenCL C 1.2
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	ReadWriteCache	ReadWriteCache

Табела 5.52 Карактеристике *Конфигурације 9*

1. *Алгоритам 1*

   У Табели 5.53. приказане су *MLUPS* вредности извршавања основног алгоритма. На основу података из табеле види се да се највећа брзина извршавања симулације постиже коришћењем *Pointer* структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	402	428	440	444	445	449	451	452
Subbuffer	400	426	438	442	444	447	449	450
Copy	401	427	439	443	446	448	450	451

Табела 5.53. *MLUPS* вредности извршавања основног алгоритма

2. *Алгоритам 2*

   У Табели 5.54. приказани су резултати извршавања алгоритма који користи локалну меморију и један локални бројач по *x* оси. На основу података из табеле види се да се највеће брзине извршавања симулације постижу коришћењем *Copy* структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	380	404	416	420	423	425	427	427
Subbuffer	378	404	416	420	423	425	427	427
Copy	381	405	417	421	424	426	428	428

Табела 5.54. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и једног локалног бројача по *x* оси

3. Алгоритам 3

У Табели 5.55. Приказани су резултати извршавања алгоритма који користи локалну меморију и два локална бројача, симулација постиже исте брзине за све структуре.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	285	289	290	295	300	305	312	322
Subbuffer	285	289	290	295	300	305	312	322
Copy	285	289	290	295	300	305	312	322

Табела 5.55. *MLUPS* вредности извршавања модификације основног алгоритма коришћењем локалне меморије и два локална бројача

4. Алгоритам 4

У Табели 5.56. приказани су резултати извршавања симулације за алгоритам са смањеним бројем помоћних променљивих. Најбоље перформансе постижу се уколико се за пренос података на уређаје користи *Copy* структура.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Pointer	298	325	331	337	340	341	342	343
Subbuffer	302	324	332	338	341	342	343	344
Copy	303	327	336	341	344	345	346	347

Тебела 5.56. *MLUPS* вредности извршавања модификације основног алгоритма смањењем броја помоћних променљивих

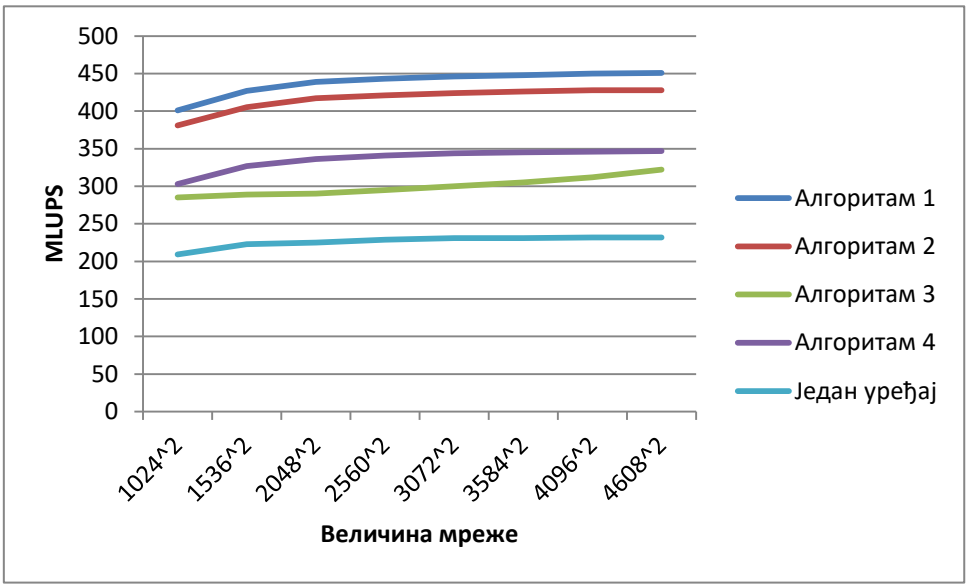
На основу приказаних резултата може се закључити да је за посматрану конфигурацију најбољи избор *Copy* структура.

У Табели 5.57. дати су најбољи резултати за сваки од алгоритама и резултат извршавања симулације на једном уређају.

	1024^2	1536^2	2048^2	2560^2	3072^2	3584^2	4096^2	4608^2
Алгоритам 1	401	427	439	443	446	448	450	451
Алгоритам 2	381	405	417	421	424	426	428	428
Алгоритам 3	285	289	290	295	300	305	312	322
Алгоритам 4	303	327	336	341	344	345	346	347
Један уређај	209	223	225	229	231	231	232	232

Табела 5.57. Поређење различитих алгоритама за *Конфигурацију 9*

На Слици 5.15. дат је графички приказ Табеле 5.57.



Слика 5.15. Графички приказ Табеле 5.57.

На основу графичког приказа резултата види се да најбоље перформансе на *Конфигурацији 9* постиже *Алгоритам 1*.

5.10. Анализа резултата

У претходно наведеним табелама и на графиконима у оквиру петог поглавља приказани су резултати извршавања различитих алгоритама на девет различитих конфигурација. За конфигурације су узети модели уређаја свих тренутно актулених произвођача. Конфигурација 1 користи *NVIDIA* графичке картице а добијени резултати су упоређени са резултатима доступним у литератури. Није било могуће извршити поређење на основу потпуно истих хардверских параметара, али сви параметри су довољно слични па је поређење ипак адекватно. У Конфигурацијама 2 до 6 коришћен је хардвер који не може да извршава *CUDA* код. Самим тим једино могуће поређење је било такође са *OpenCL* имплементацијом на једном уређају, као што је и приказано у тези. У Конфигурацијама 1 и 7 до 9 коришћене су *NVIDIA* графичке картице па је теоретски било могуће поређење са *CUDA* имплементацијом на једном уређају. Пошто је слично поређење већ вршено у (Tekić et al. 2014) на различитом хардверу, овде је одлучено да се као референтна такође користи *OpenCL* имплементација јер се и у њој, као што је показано, постижу слични резултати као са *CUDA* имплементацијом. Пошто перформансе симулације зависе од перформанси самих уређаја које се битно разликују - перформансе уређаја не зависе само од произвођача и модела уређаја, него и од осталих елемената у конфигурацији (врста и количина меморије, матична плоча, процесор, верзија *OpenCL* имплементације, ...) - нису вршена унакрсна графичка поређења између различитих произвођача. Уместо тога за сваку групу уређаја дата је кратка анализа на основу уочених карактеристика за посматрану групу.

Из презентованих резултата може се закључити да су за *NVIDIA* графичке картице боља решења алгоритми који користе локалну меморију, у случају да се за *HOST* програм користи процесор новије генерације (*Intel i7*) најбољи избор је алгоритам који користи два локална бројача. У случају да се за *HOST* уређај користи процесор старије генерације боље је користити алгоритам са једним локалним бројачем.

Платформе *AMD* графичких картица не могу добро да искористе локалну меморију па је за *AMD* платформе боље користити основни алгоритам или основни алгоритам са смањеним бројем променљивих.

Алгоритам са смањеним бројем променљивих показао се као добар избор уколико постоји *AMD* платформа у комбинацији са другом платформом, па је потребно формирати два контекста.

Коришћење различитих структура података не утиче превише значајно на брзину извршавања симулације, али је ипак може у малој мери повећати код неких конфигурација. Из резултата је закључено да код персоналних рачунара који садрже *AMD* графичке картице постоји минимална разлика у брзини за коришћење различитих структура података. За персоналне рачунаре и *HPC* уређаје који користе *NVIDIA* графичке картице и *Apple* платформу избор одговарајуће структуре података може побољшати брзину извршавања за око 10%. За Тесла графичке картице и *Apple* платформу најбољи избор је *Subbufer* структура, док је за *GeForce* графичке картице најбољи избор *Copy* структура.

Када на рачунару постоје уређаји различитих проивођача уколико постоји могућност да се симулација покрене на једној платформи, постигнуте брзине ће бити веће него ако се користе две платформе, чак и ако се перформансе једног од уређаја смање услед примене имплементације платформе другог произвођача. Пренос података између две платформе је компликован и утиче на брзину више од избора платформе другог произвођача. Уколико је неопходно користити два контекста треба одабрати алгоритам са смањеним бројем помоћних променљивих.

Перформансе извршавања симулације на више хетерогених вишејзгарних уређаја не зависе само од перформанси уређаја који се користе за паралелно извршавање него и од перформанси *HOST* уређаја. Након извршених тестирања на различитим врстама *HOST* уређаја, може се извући закључак да се преласком на процесоре новије генерације (*Intel i7* и више) очекује да ће најбоље перформансе постизати алгоритам који користи локалну меморију и две локалне променљиве. Такође, очекује се да ће и други произвођачи унапредити рад са локалном меморијом и да ће и на њиховим уређајима у будућности алгоритам који користи локалну меморију и две локалне променљиве имати најбоље перформансе. Већина актуелних произвођача још увек није имплементирала *OpenCL* 2.0. Имплеметација *OpenCL* 2.0, а затим *OpenCL* 2.1 и *OpenCL* 2.2 такође

би требала да допринесе побољшању перформанси извршавања симулације.

6. Имплементација солвера у микросервис архитектури

У овом поглављу описана је апликација за симулацију тока флуида. Апликација је направљена у архитектури микросервиса. Формирана су два микросервиса: микросервис за кориснички интерфејс и микросервис за извршавање симулације тока флуида у шупљини.

6.1. Микросервис за кориснички интерфејс

Микросервис за кориснички интерфејс направљен је коришћењем *ReactJS framework*-а. Када се стартује апликација, прво је потребно приказати који уређаји и платформе су доступни за извршавање апликације. Контролеру *HardwareController* у оквиру микросервиса за извршавање апликације биће послат *HTTP* захтев за излиставање платформи и уређаја пре него што дође до учитавања *React* компоненте, што је приказано на листингу 6.1.

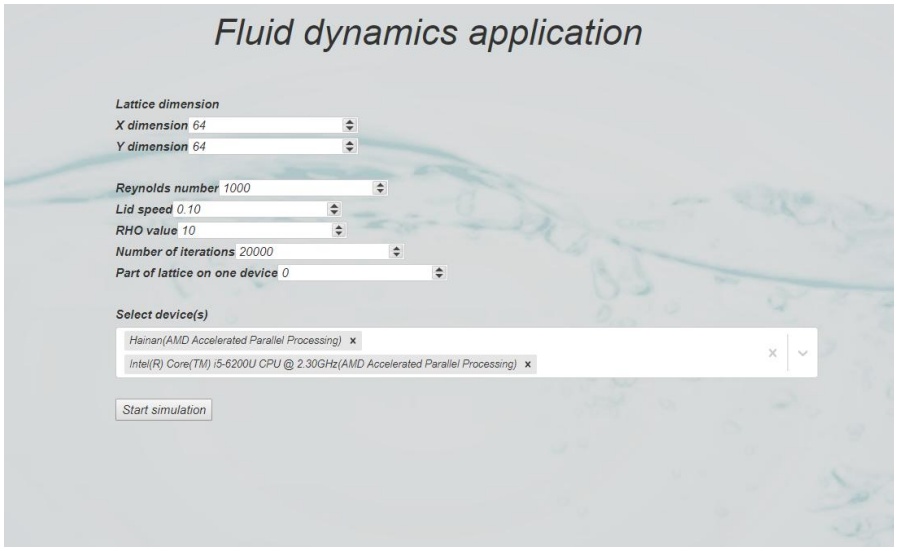
```
componentDidMount() {
  fetch('http://localhost:8080/hardware')
.then(response => response.json())
. then(json => {
     this.setState(json.map((item,i) => {
       console.log("jsonItem", item.platformName);
       item.platformDevices.map((device,i)=>{

       let joined = this.state.filterOptions.concat({ value: device.deviceId+'-
'+item.platformId, label: device.deviceName+'('+item.platformName+')' });
       this.setState({ filterOptions: joined })})

   }));
   })
 }
```

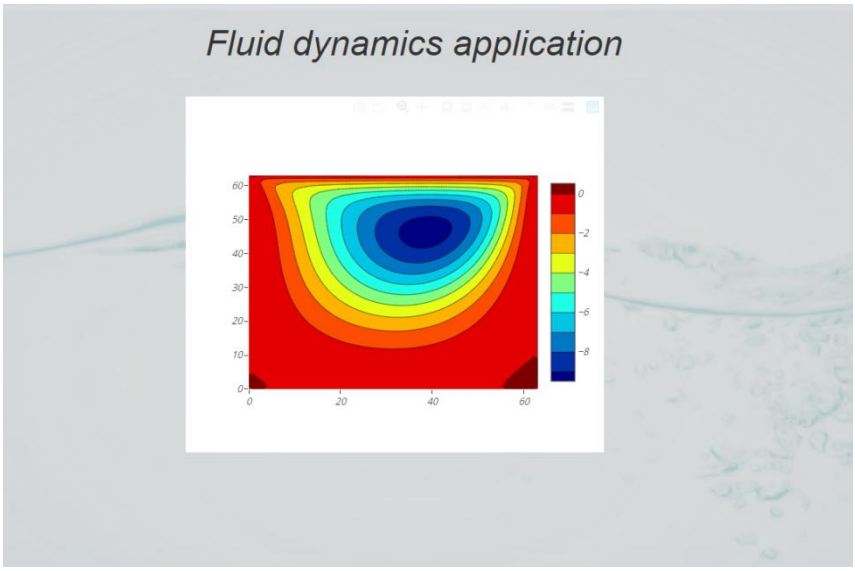
Листинг 6.1 Захтев за излиставање доступних платформи и уређаја

Након што су учитани подаци о доступним *OpenCL* уређајима, кориснику се приказују поља за унос улазних параметара. Приказано на Слици 6.1.

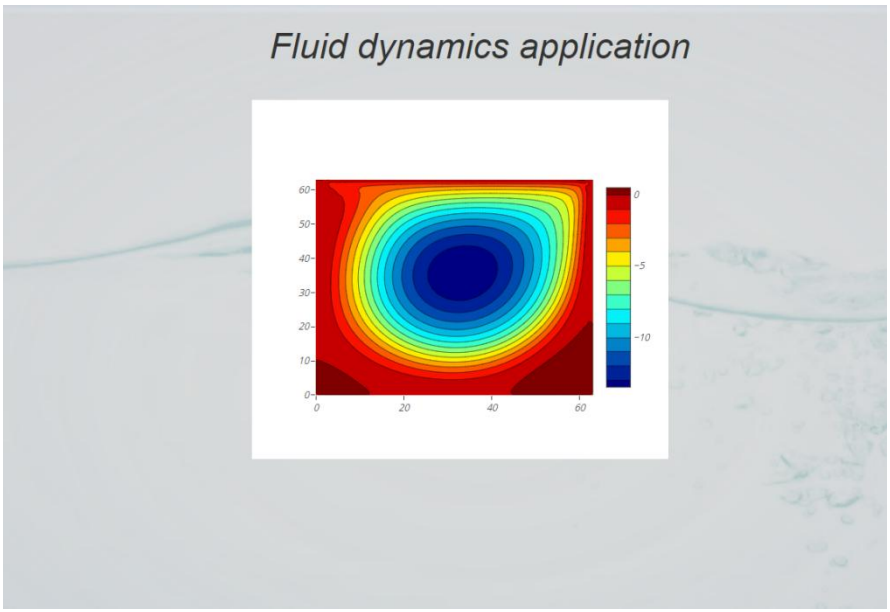


Слика 6.1. Унос улазних параметара

Након што се унесу улазни параметри исти се путем *HTTP REST* позива шаљу микросервису за извршавање апликације и долази до покретања симулације. Након што се извршавање симулације заврши контролер шаље резултат извршавања симулације. На основу враћених података помоћу *Plotly java script* библиотеке (приказано на Листингу 6.2.) прикаже се резултат извршавања симулације. На слици Слици 6.2. приказана је визуализација резултата симулације за Рејнолдсов број (Re) 100, а на Слици 6.3. визуализација добијених резултата за Рејнолдсов број (Re) 1000.



Слика 6.2. Резултат извршавања *CFD* симулације за Re=100



Слика 6.3. Резултат извршавања *CFD* симулације за Re=1000

```
            handleSubmit (event) {
              var parent = this;
              var form = document.getElementById("formId");
              form.style.display = "none";

              let bodyData=
'multiValue='+JSON.stringify(this.state.multiValue)+'&nx='+this.state.nx+'&ny='+this.state.ny+'&u='
+this.state.u+'&re='+this.state.re+'&rho='+this.state.rho+'&steps='+this.state.steps+'&part='+this.state
.part;
              event.preventDefault();
              fetch('http://localhost:8080/simulation', {
                method: 'POST',
                headers: new Headers({

'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
                  'Content-Type': 'application/x-www-form-urlencoded',

                }),
                body: bodyData,

              }).then((dataResponse) => {

                dataResponse.json().then(function(text) {

                var data = [{z: text, type:'contour', colorscale: 'Jet'}];
                var layout = [{  title: 'Contour Plot' }];
                 var   datatr = [{
                              z: text,
                              type: 'contour',
                              colorscale: 'Jet',
                             }];

                    Plotly.newPlot('pDiv', datatr, layout,{showSendToCloud: true});


                  })
                   })
              .catch(err => console.error('Caught error: ', err));

              }
```

Листинг 6.2. Обрада *response*-а

6.2. Микросервис за извршавање симулације

Микросервис за извршавање симулације има два *HTTP endpoint*-а. Први *HTTP endpoint* је контролер *HardwareController*, чији је задатак да врати листу доступних *OpenCL* уређаја на којима је могуће покренути извршавање симулације.

```
@RestController
public class HardwareController {
private Logger log = LoggerFactory.getLogger(this.getClass());
@Autowired
public HardwareService hws;

@RequestMapping(value = "/hardware")
public List<PlatformDTO> hardwareInfo() {
    return hws.getPlatforms();
    }

}
```

Листинг 6.3. *HardwareController*

HardwareService (Листинг 6.4.) излистава доступне уређаје и враћа контролеру *HardwareController* листу *Data Transfer* објеката са подацима о уређајима.

```
@Service
public class HardwareService {
public List<PlatformDTO> getPlatforms(){

     List<PlatformDTO> platforms = new ArrayList<PlatformDTO>();

     int numPlatforms[] = new int[1];
    clGetPlatformIDs(0, null, numPlatforms);

    cl_platform_id platformsCL[] = new cl_platform_id[numPlatforms[0]];
    clGetPlatformIDs(platformsCL.length, platformsCL, null);
    int platformNo = platformsCL.length;
    // Collect all devices of all platforms

    for (int i=0; i < platformNo ; i++) {

     List<cl_device_id> devicesCL = new ArrayList<cl_device_id>();
       String platformName = getString(platformsCL[i], CL_PLATFORM_NAME);

       PlatformDTO platform = new PlatformDTO(i,platformName);

       // Obtain the number of devices for the current platform
       int numDevices[] = new int[1];
       clGetDeviceIDs(platformsCL[i], CL_DEVICE_TYPE_ALL, 0, null, numDevices);

       cl_device_id devicesArray[] = new cl_device_id[numDevices[0]];
       clGetDeviceIDs(platformsCL[i], CL_DEVICE_TYPE_ALL, numDevices[0],
devicesArray, null);

       devicesCL.addAll(Arrays.asList(devicesArray));

      List<DeviceDTO> devices = new ArrayList<>();
          int j = 0;
          for (cl_device_id deviceCL : devicesCL)
          {
              String deviceName = getString(deviceCL, CL_DEVICE_NAME);
              String casheType = getString(deviceCL,
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE);
              DeviceDTO device = new DeviceDTO(j,deviceName, casheType);
              devices.add(device);
              j++;
          }
       platform.setPlatformDevices(devices);
       platforms.add(platform);
    }
     return platforms;
  }
```

Листинг 6.4. *HardwareService*

Извршавање саме симулације иницира се контактирањем контролера *SimulationController* (Листинг 6.5.) који преузима улазне параметре и позива сервис *SimulationService* који покреће извршавање симулације са одговарајућим алгоритмом за дату конфигурацију.

```
@RestController
public class SimulationController {
  private Logger log = LoggerFactory.getLogger(this.getClass());

  @Autowired
  SimulationService simulationServices;

  @RequestMapping(value = "/simulation", method = RequestMethod.POST)
  public String startSimulation(@RequestParam(value = "nx") int nx, @RequestParam(value = "ny") int ny,
      @RequestParam(value = "u") float u, @RequestParam(value = "re") int re,
      @RequestParam(value = "rho") int rho, @RequestParam(value = "steps") int steps, , @RequestParam(value = "part") int part,
      @RequestParam(value = "multiValue") String multiValue) {

    log.info("PARAMETARS: " + "nx=" + nx + "ny=" + ny + "u=" + u + "re=" + re + "rho=" + rho);
    log.info("Devices selected: " + multiValue);

    String result = simulationServices.startSimulation(nx, ny, u, re, rho, steps, part, multiValue);
    log.info(result);
    return result;

  }

}
```

Листинг 6.5. *SimulationController*

Након извршавања симулације на екрану се приказује резултат иршавања (Слика 6.2.) и формирају фајлови који могу бити искоришћени за визуализацију у *Matlab*-у или неком другом програму за визуализацију.

7. Закључак

Резултати истраживања приказани у дисертацији показују да се решавање нумеричких модела симулације може унапредити истовременим коришћењем неколико хетерогених вишејезгарних уређаја. Показано је да се убрзање коришћењем *OpenCL* стандарда на више уређаја истовремено може постићи на *HPC* уређајима (Тесла картицама) али и на персоналним рачунарима у односу на извршавање симулације само на једном уређају. Проучено је неколико различитих имплементација и показано да се у односу на специфичности доступног хардвера и избором одговарајуће имплементације може значајно убрзати извршавање симулација.

У првом поглављу представљен је проблем који је обрађен у тези, мотиви који су довели до рада на истраживању, дефинисани су циљеви истраживања, описан научни допринос изведених истраживања и дат преглед досадашњих истраживања из доступне литературе.

Друго поглавље је осврт на развој паралелног програмирања. Паралелно програмирање је у почетку било резервисано само за супер рачунаре и доступно уском кругу људи, али је развојем нових архитектура рачунара постало доступно широком кругу истраживача. Упоредо са развојем хрдвера развија се и софтвер који пружа могућност рада са новим архитектурама.

У трећем поглављу дат је опис технологија и алата који су коришћени за развој апликације. Употреба описаних технологија и алата омогућила је креирање ефикасног, портабилног и лако проширивог решења које корисници могу једноставно покренути и користити на персоналним рачунарима.

У четвртом поглављу дате су основе *Lattice Boltzmann* методе, укратко описани могући начини решавања методе и дат је основни алгоритам. Затим су детаљно описани различити алгоритми којима се проблем имплементира на више хетерогених уређаја, уз навођење предности сваког алгоритма.

У петом поглављу су тестирани и детаљно анализирани сви понуђени алгоритми, на тај начин верификоване су предности одређеног алгоритма за одговарајући хардвер.

У овој тези показано је да је могуће искористити *OpenCL* спецификацију за креирање решења које ће се извршавати на више хетерогених вишејезгарних уређаја и омогућити значајно убрзање извршавања симулације динамике флуида. У овакву групу уређаја спадају пре свега персонални рачунари, циљ тезе је пре свега да омогући постизање максималних перформанси симулације на оваквим уређајима. Решење приказано у тези постиже брзине на *HPC* уређајима које су сличне брзинама решења из доступне литературе. Такође креирана је апликација која је портабилна и флексибилна и може бити коришћена од стране трећих апликација.

Даље истраживање могло би се одвијати у неколико праваца. Рад на побољшању алгоритма могао би укључити измене у структури података, коришћење *AoS* (оптимизоване за *collision* корак) уместо *SoA* структуре података или примену неког другог алгоритма којим би се решавала зависност која се јавља у *streaming* кораку. Такође би се могли проучавати слични модели који би имали препреке, уз измену дела алгоритма који се односи на граничне услове.

Други правац за даљи рад могло би бити проучавање архитектуре уређаја и прављење алгоритма који би аутоматски одређивао како извршити расподелу података на уређаје, уместо емпиријског одређивања расподеле који је коришћен у тези.

Трећи правац за даљи рад је проширење функционалности апликације за симулацију динамике флуида додавањем микросервиса који би омогућили рад на додатним проблемима везаним за симулацију динамике флуида, на пример рад са флуидима који се греју (Tekić et al. 2014), ток флуида кроз цев и ток флуида кроз порозну средину.

Четврти правац даљег истраживања је прављење хибридне имплементације која би могла да ради на кластерима који су састављени од хетерогених уређаја.

Литература

ABOUHAMZA, A. & PIERRE, R. (2003) A neutral stability curve for incompressible flows in a rectangular driven cavity. *Mathematical and Computer Modelling,* 38**,** 141-157

AHMED, M. F. (2010) OpenCL.

AIDUN, C. K., TRIANTAFILLOPOULOS, N. G. & BENSON, J. D. (1991) Global stability of a lid-driven cavity with throughflow: Flow visualization studies. *Physics of Fluids A,* 3**,** 2081-2091.

AKHTER, S. & ROBERTS, J. (2006) *Multi-core Programming: Increasing Performance Through Software Multi-threading,* Intel Press.

AKSNES, E. O. & ELSTER, A. C. (2009) Porous Rock Simulations and Lattice Boltzmann on GPUs.

ALBENSOEDER, S. & KUHLMANN, H. C. (2002) Linear stability of rectangular cavity flows driven by anti-parallel motion of two facing walls. *Journal of Fluid Mechanics,* 458**,** 153-180.

ALBENSOEDER, S., KUHLMANN, H. C. & RATH, H. J. (2001) Multiplicity of Steady Two-Dimensional Flows in Two-Sided Lid-Driven Cavities. *Theoretical and Computational Fluid Dynamics,* 14**,** 223-241.

ALEEM, M. (2015) A Java-based Programming andExecution Environment for Many-coreParallel Computers. *Institute of Computer Science.* University of Innsbruck.

ALLEBORN, N., RASZILLIER, H. & DURST, F. (1999) Lid-driven cavity with heat and mass transport. *International Journal of Heat and Mass Transfer,* 42**,** 833-853.

ANDERSON, J. A., LORENZ, C. D. & TRAVESSET, A. (2008) General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics,* 227**,** 5342-5359.

ARIE, K. & ET AL. (2009) Implementing the lattice Boltzmann model on commodity graphics hardware. *Journal of Statistical Mechanics: Theory and Experiment,* 2009**,** P06016.

ARIS, R. (1990) *Vectors, Tensors and the Basic Equations of Fluid Mechanics,* Dover Publications.

ARUMUGA PERUMAL, D. & DASS, A. K. (2008) Simulation of flow in two-sided lid-driven square cavities by the lattice Boltzmann method. IN RAHMAN, M. & BREBBIA, C. A. (Eds.) *Advances in Fluid Mechanics VII.* WIT Press.

ARUMUGA PERUMAL, D. & DASS, A. K. (2010) Multiplicity of steady solutions in two-dimensional lid-driven cavity flows by Lattice Boltzmann Method. *Computers & Mathematics with Applications,* doi:10.1016/j.camwa.2010.03.053.

ASLAN, E., NAHAVANDI, A., TAYMAZ, I. & BENIM, A. (2012) A note on modelling non–rectangular boundaries by the Lattice Boltzmann Method. *Progress in Computational Fluid Dynamics, an International Journal,* 12**,** 433-438.

BAILEY, P., MYRE, J., WALSH, S. D. C., LILJA, D. J. & SAAR, M. O. (2009) Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. *International Conference on Parallel Processing, 2009. ICPP '09.*

BANGER RAVISHEKHAR, B. K. (2013) *OpenCL Programming by Example*, Packt Publishing.

BANKS, A. & PORCELLO, E. (2017) *Learning React: Functional Web Development with React and Redux*, O'Reilly Media, Inc.

BENIM, A., ASLAN, E., NAHAVANDI, A. & TAYMAZ, I. (2011) LBM investigation of laminar forced convection in a channel with triangular prism. *7th International Conference on Computational Heat and Mass Trasnfer.*

BENIM, A. C., ASLAN, E. & TAYMAZ, I. (2011) Lattice Boltzmann method for laminar forced convection in a channel with a triangular prism. *Heat Transfer Research,* 42.

BENZI, R., SUCCI, S. & VERGASSOLA, M. (1992) The lattice Boltzmann equation: theory and applications. *Physics Reports,* 222**,** 145-197.

BERNASCHI, M., FATICA, M., MELCHIONNA, S., SUCCI, S. & KAXIRAS, E. (2010) A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurr. Comput. : Pract. Exper.,* 22**,** 1-14.

BERNASCHI, M., ROSSI, L., BENZI, R., SBRAGAGLIA, M. & SUCCI, S. (2009) Graphics processing unit implementation of lattice Boltzmann models for flowing soft systems. *Physical Review E,* 80**,** 066707.

BHATNAGAR, P. L., GROSS, E. P. & KROOK, M. (1954) A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *PHYSICAL REVIEW,* 94.

BLAIR, S. R. (2012) LATTICE BOLTZMANN METHODS FOR FLUID STRUCTURE INTERACTION. NAVAL POSTGRADUATE SCHOOL.
BLAISE, B. (2019) Message Passing Interface (MPI). Lawrence Livermore National Laboratory.
BLOHM, C. & KUHLMANN, H. C. (2002) The two-sided lid-driven cavity: Experiments on stationary and time-dependent flows. *Journal of Fluid Mechanics,* 450**,** 67-95.
BOTELLA, O. & PEYRET, R. (1998) Benchmark spectral results on the lid-driven cavity flow. *Computers & Fluids,* 27**,** 421-433.
BRUNEAU, C.-H. & SAAD, M. (2006) The 2D lid-driven cavity problem revisited. *Computers & Fluids,* 35**,** 326-348.
CALORE, E., SCHIFANO, S. F. & TRIPICCIONE, R. (2014) A Portable OpenCL Lattice Boltzmann Code for Multi- and Many-core Processor Architectures. *Procedia Computer Science,* 29**,** 40-49.
CERCIGNANI, C. (1975) *Theory and application of the Boltzmann equation*, Scottish Academic Press, Edinburgh.
CHANG, C., LIU, C.-H. & LIN, C.-A. (2009) Boundary conditions for lattice Boltzmann simulations with complex geometry flows. *Computers & Mathematics with Applications,* 58**,** 940-949.
CHANG, H.-W., HONG, P.-Y., LIN, L.-S. & LIN, C.-A. (2013) Simulations of flow instability in three dimensional deep cavities with multi relaxation time lattice Boltzmann method on graphic processing units. *Computers & Fluids,* 88**,** 866-871.
CHANG, H.-W., HONG, P.-Y., LIN, L.-S. & LIN, C.-A. (2013) Simulations of Three-dimensional Cavity Flows with Multi Relaxation Time Lattice Boltzmann Method and Graphic Processing Units. *Procedia Engineering,* 61**,** 94-99.
CHE SIDIK, N. A., OSMAN, K., KHUDZAIRI, A. Z. & NGALI, Z. (2008) Numerical investigation of lid-driven cavity flow based on two different methods: lattice Boltzmann and splitting method. *Jurnal Mekanikal,* 25**,** 1-8.
CHEN, G. K. & GUO, Y. (2013) Discovering epistasis in large scale genetic association studies by exploiting graphics cards. *Frontiers in genetics,* 4**,** 266-266.
CHEN, J. (2008) Topological chaos and mixing in lid-driven cavities and rectangular channels. Blacksburg, Virginia, Faculty of the Virginia Polytechnic Institute.

CHEN, K., LIN, C., ZHONG, S., GUO, L (2015) A Parallel SRM Feature Extraction Algorithm for Steganalysis Based on GPU Architecture. *Computer Science and Information Systems,* 12.

CHEN, S. (2009) A large-eddy-based lattice Boltzmann model for turbulent flow simulation. *Applied Mathematics and Computation,* 215**,** 591-598.

CHEN, S. & DOOLEN, G. D. (1998) Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics,* 30**,** 329-364.

CHEN, S., TÖLKE, J. & KRAFCZYK, M. (2008) A new method for the numerical solution of vorticity-streamfunction formulations. *Computer Methods in Applied Mechanics and Engineering,* 198**,** 367-376.

CHENG, J., GROSSMAN, M. & MCKERCHER, T. (2014) *Professional CUDA C Programming,* Wiley.

CHENG, M. & HUNG, K. C. (2006) Vortex structure of steady flow in a rectangular cavity. *Computers & Fluids,* 35**,** 1046-1062.

CHOPARD, B. & LUTHI, P. O. (1999) Lattice Boltzmann computations and applications to physics. *Theoretical Computer Science,* 217**,** 115-130.

CRESTETTO, A., HELLUY, P. & JUNG, J. (2013) Numerical resolution of conservation laws with OpenCL. *ESAIM: Proc.,* 40**,** 51-62.

DAVID, R. K., PERHAAD, M., DANA, S. & DONG PING, Z. (2015) *Heterogeneous Computing with OpenCL 2.0,* Morgan Kaufmann Publishers Inc.

DE, S., NAGENDRA, K. & LAKSHMISHA, K. N. (2009) Simulation of laminar flow in a three-dimensional lid-driven cavity by lattice Boltzmann method. *International Journal of Numerical Methods for Heat & Fluid Flow,* 19**,** 790-815.

DING, L., SHI, W., LUO, H. & ZHENG, H. (2009) Investigation of incompressible flow within 1/2 circular cavity using lattice Boltzmann method. *International Journal for Numerical Methods in Fluids,* 60**,** 919-936.

DIXIT, H. N. & BABU, V. (2006) Simulation of high Rayleigh number natural convection in a square cavity using the lattice Boltzmann method. *International Journal of Heat and Mass Transfer,* 49**,** 727-739.

D'ORAZIO, A., CORCIONE, M. & CELATA, G. P. (2004) Application to natural convection enclosed flows of a lattice Boltzmann BGK

model coupled with a general purpose thermal boundary condition. *International Journal of Thermal Sciences,* 43**,** 575-586.
DRAGONI, N., GIALLORENZO, S., LAFUENTE, A. L., MAZZARA, M., MONTESI, F., MUSTAFIN, R. & SAFINA, L. (2017) *Microservices: Yesterday, Today, and Tomorrow*, Springer, Cham.
DU, R. & LIU, W. (2013) A new multiple-relaxation-time lattice Boltzmann method for natural convection. *Journal of Scientific Computing,* 56**,** 122-130.
ERTURK, E. (2009) Discussions on driven cavity flow. *International Journal for Numerical Methods in Fluids,* 60**,** 275-294.
ERTURK, E., CORKE, T. C. & GÖKÇÖL, C. (2005) Numerical solutions of 2-D steady incompressible driven cavity flow at high Reynolds numbers. *International Journal for Numerical Methods in Fluids,* 48**,** 747-774.
ERTURK, E. & DURSUN, B. (2007) Numerical solutions of 2-D steady incompressible flow in a driven skewed cavity. *Journal of Applied Mathematics and Mechanics,* 87**,** 377-392.
ERTURK, E. & GÖKÇÖL, C. (2006) Fourth-order compact formulation of Navier–Stokes equations and driven cavity flow at high Reynolds numbers. *International Journal for Numerical Methods in Fluids,* 50**,** 421-436.
FAN, Z., QIU, F., KAUFMAN, A. & YOAKUM-STOVER, S. (2004) GPU Cluster for High Performance Computing. *ACM / IEEE Supercomputing Conference 2004.* Pittsburgh.
FEICHTINGER, C., HABICH, J., KÖSTLER, H., HAGER, G., RÜDE, U. & WELLEIN, G. (2011) A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters. *Parallel Computing,* 37**,** 536-549.
FILIPPOVA, O. & HANEL, D. (2000) A novel BGK approach for low mach number combustion. *J. Comput. Phys.,* 158**,** 139-160.
GALLIVAN, M. A., NOBLE, D. R., GEORGIADIS, J. G. & BUCKIUS, R. O. (1997) AN EVALUATION OF THE BOUNCE-BACK BOUNDARY CONDITION FOR LATTICE BOLTZMANN SIMULATIONS. *International Journal for Numerical Methods in Fluids,* 25**,** 249-263.
GANDON, F. (1999) Types of Computers.
GASKELL, P. H., SAVAGE, M. D., SUMMERS, J. L. & THOMPSON, H. M. (1998) Stokes flow in closed, rectangular domains. *Applied Mathematical Modelling,* 22**,** 727-743.

GEVELER, M., RIBBROCK, D., MALLACH, S. & GÖDDEKE, D. (2011) A simulation suite for Lattice-Boltzmann based real-time CFD applications exploiting multi-level parallelism on modern multi- and many-core architectures. *Journal of Computational Science,* 2**,** 113-123.

GHIA, U., GHIA, K. N. & SHIN, C. T. (1982) High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics,* 48**,** 387-411.

GINZBURG, I., VERHAEGHE, F. & D'HUMIERES, D. (2008) Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions. *Communications in computational physics,* 3**,** 427-478.

GUO, Z., SHI, B. & WANG, N. (2000) Lattice BGK Model for Incompressible Navier-Stokes Equation. *Journal of Computational Physics,* 165**,** 288-306.

GUO, Z., SHI, B. & WANG, N. (2000) Lattice BGK Model for Incompressible Navier–Stokes Equation. *Journal of Computational Physics,* 165**,** 288-306.

GUO, Z., SHI, B. & ZHENG, C. (2002) A coupled lattice BGK model for the Boussinesq equations. *International Journal for Numerical Methods in Fluids,* 39**,** 325-342.

GUO, Z. & ZAO, T. S. (2005) A lattice Boltzmann model for convective heat transfer in porous media. *Numerical Heat Transfer, Part B Fundamentals***,** 155-177.

GUO, Z. & ZHAO, T. S. (2002) Lattice Boltzmann model for incompressible flows through porous media. *Physical Review E,* 66.

GUO, Z. & ZHAO, T. S. (2003) Discrete velocity and lattice Boltzmann models for binary mixtures of nonideal fluids. *Physical Review E,* 68.

GÜRCAN, F. (2003) Streamline Topologies in Stokes Flow Within Lid-Driven Cavities. *Theoretical and Computational Fluid Dynamics,* 17**,** 19-30.

GÜRCAN, F., DELICEOGLU, A. & BAKKER, P. G. (2005) Streamline topologies near a non-simple degenerate critical point close to a stationary wall using normal forms. *Journal of Fluid Mechanics,* 539**,** 299-311.

GÜRCAN, F., GASKELL, P. H., SAVAGE, M. D. & WILSON, M. C. T. (2003) Eddy genesis and transformation of Stokes flow in a double-lid driven cavity. *Proceedings of the Institution of Mechanical*

Engineers, Part C: Journal of Mechanical Engineering Science, 217**,** 353-364.
GÜRCAN, F., WILSON, M. C. T. & SAVAGE, M. D. (2006) Eddy genesis and transformation of Stokes flow in a double-lid-driven cavity. Part 2: deep cavities. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science,* 220**,** 1765-1774.
HABICH, J., ZEISER, T., HAGER, G. & WELLEIN, G. (2011) Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software,* 42**,** 266-272.
HARADA, T., KOSHIZUKA, S. & KAWAGUCHI, Y. (2007) Smoothed Particle Hydrodynamics on GPUs.
HARVEY, M. J. & FABRITIIS, G. D. (2011) Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*.
HE, X., CHEN, S. & DOOLEN, G. D. (1998) A Novel Thermal Model for the Lattice Boltzmann Method in Incompressible Limit. *Journal of Computational Physics,* 146**,** 282-300.
HE, X. & LUO, L.-S. (1997) Lattice Boltzmann Model for the Incompressible Navier–Stokes Equation. *Journal of Statistical Physics,* 88**,** 927-944.
HE, X. & LUO, L.-S. (1997) Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E,* 56**,** 6811-6817.
HO, C.-F., CHANG, C., LIN, K.-H. & LIN, C.-A. (2009) Consistent Boundary Conditions for 2D and 3D Lattice Boltzmann Simulations. *Computer Modeling in Engineering & Sciences,* Vol. 44**,** pp. 137-156.
HONG, P.-Y., HUANG, L.-M., LIN, L.-S. & LIN, C.-A. (2015) Scalable multi-relaxation-time lattice Boltzmann simulations on multi-GPU cluster. *Computers & Fluids,* 110**,** 1-8.
HOU, S., ZOU, Q., CHEN, S., DOOLEN, G. & COGLEY, A. C. (1995) Simulation of cavity flow by the lattice Boltzmann method. *Journal of Computational Physics,* 118**,** 329-347.
HOU, S., ZOU, Q., CHEN, S., DOOLEN, G. D. & COGLEY, A. C. (1995) Simulation of Cavity Flow by the Lattice Boltzmann Method. *Journal of Computational Physics,* 118**,** 329.
HOWES, L. & MUNSHI, A. (2015) The OpenCL Specification. Khronos OpenCL Working Group.

HUANG, C., SHI, B., HE, N. & CHAI, Z. (2015) Implementation of Multi-GPU Based Lattice Boltzmann Method for Flow Through Porous Media. *Advances in Applied Mathematics and Mechanics,* 7**,** 1-12.
INAMURO, T., YOSHINO, M. & OGINO, F. (1995) A non-slip boundary condition for lattice Boltzmann simulations. *Physics of Fluids,* 7**,** 2928-2930.
IRWAN, M. A. M., FUDHAIL, A. M. & C. S. NOR AZWADI, G. M. (2010) Numerical Investigation of Incompressible Fluid Flow through Porous Media in a Lid-Driven Square Cavity. *American Journal of Applied Sciences,* 7**,** 1341-1344.
JAMI, M. & MEZRHAB, A. (2008) Numerical study of natural convection in a square cavity containing a cylinder using the lattice Boltzmann method. *Engineering Computations,* 25**,** 480-489.
KARIMI, K. (2015) The Feasibility of Using OpenCL Instead of OpenMP for Parallel CPU Programming. *CoRR,* abs/1503.06532.
KELMANSON, M. A. & LONSDALE, B. (1996) Eddy genesis in the double-lid-driven cavity. *Quarterly Journal of Mechanics and Applied Mathematics,* 49**,** 635-656.
KHANNA, G. & MCKENNON, J. (2010) Numerical modeling of gravitational wave sources accelerated by OpenCL. *Computer Physics Communications,* 181 1605–1611.
KÖRNER, C., POHL, T., RÜDE, U., THÜREY, N. & ZEISER, T. (2006) Parallel Lattice Boltzmann Methods for CFD Applications
Numerical Solution of Partial Differential Equations on Parallel Computers. IN BRUASET, A. M. & TVEITO, A. (Eds.), Springer Berlin Heidelberg.
KRISHNA, D. J., BASAK, T. & DAS, S. K. (2008) Numerical study of lid-driven flow in orthogonal and skewed porous cavity. *Communications in Numerical Methods in Engineering,* 24**,** 815-831.
KRÜGER, T., VARNIK, F. & RAABE, D. (2009) Shear stress in lattice Boltzmann simulations. *Physical Review E,* 79.
KUHLMANN, H. C., ALBENSOEDER, S. & BLOHM, C. (2001) Flow Instabilities in the Two-Sided Lid-Driven Cavity. *12th International Couette-Taylor Workshop.* Evanston, IL USA.
KUHLMANN, H. C., WANSCHURA, M. & RATH, H. J. (1997) Flow in two-sided lid-driven cavities: non-uniqueness, instabilities, and cellular structures. *Journal of Fluid Mechanics,* 336**,** 267-299.

KUHLMANN, H. C., WANSCHURA, M. & RATH, H. J. (1998) Elliptic instability in two-sided lid-driven cavity flow. *European Journal of Mechanics - B/Fluids,* 17**,** 561-569.
KUTAY, M. E., AYDILEK, A. H. & HARMAN, T. (2006) *Dynamic Hydraulic Conductivity (Permeability) of Asphalt Pavements*, ASCE.
KUZNIK, F., OBRECHT, C., RUSAOUEN, G. & ROUX, J.-J. (2010) LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications,* 59**,** 2380-2392.
LATT, J., CHOPARD, B., MALASPINAS, O., DEVILLE, M. & MICHLER, A. (2008) Straight velocity boundaries in the lattice Boltzmann method. *Physical Review E,* 77**,** 056703.
LENGAUER, C. (2000) A Personal, Historical Perspective of Parallel Programming for High Performance. UNIVERSITY OF PASSAU.
LI, W., WEI, X. & KAUFMAN, A. (2003) Implementing Lattice Boltzmann Computation on Graphics Hardware. *Visual Computer,* 19**,** 444-456.
LI, X., ZHANG, Y., WANG, X. & GE, W. (2013) GPU-based numerical simulation of multi-phase flow in porous media using multiple-relaxation-time lattice Boltzmann method. *Chemical Engineering Science,* 102**,** 209-219.
LIN, L.-S., CHEN, Y.-C. & LIN, C.-A. (2011) Multi relaxation time lattice Boltzmann simulations of deep lid driven cavity flows at different aspect ratios. *Computers & Fluids,* 45**,** 233-240.
LOUAKED, M., HANICH, L. & NGUYEN, K. D. (1997) An efficient finite difference technique for computing incompressible viscous flows. *International Journal for Numerical Methods in Fluids,* 25**,** 1057-1082.
LÜ, X.-Y., ZHANG, C.-Y., LIU, M.-R., KONG, L.-J. & LI, H.-B. (2005) Thermal lattice Boltzmann simulation of viscous flow in a square cavity. *International Journal of Modern Physics C: Computational Physics & Physical Computation,* 16**,** 867-877.
LUO, L.-S. (2000) The lattice-gas and lattice Boltzmann methods: Past, present, and future. IN WU, J. & ZHU, Z. (Eds.) *Proc Int Conf Appl Comput Fluid Dyn.* Beijing.
LUO, W.-J. & YANG, R.-J. (2007) Multiple fluid flow and heat transfer solutions in a two-sided lid-driven cavity. *International Journal of Heat and Mass Transfer,* 50**,** 2394-2405.
MAIER, R. S., BERNARD, R. S. & GRUNAU, D. W. (1996) Boundary conditions for the lattice Boltzmann method. *Physics of Fluids,* 8**,** 1788-1801.

MARTYS, N. S. & HAGEDORN, J. G. (2002) Multiscale modeling of fluid transport in heterogeneous materials using discrete Boltzmann methods. *Materials and Structures,* 35**,** 650-649.
MASSIMO, B., MASSIMILIANO, F., SIMONE, M., SAURO, S. & EFTHIMIOS, K. (2010) A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency and Computation: Practice and Experience,* 22**,** 1-14.
MATTHEW, S. (2011) *OpenCL in Action,* Manning Publications.
MATTILA, K., HYVALUOMA, J., TIMONEN, J. & ROSSI, T. (2008) Comparison of implementation of the lattice-Boltzmann method. *Comput. Math. Appl.***,** 1514_1524.
MAZZARA M., K. K., MUSTAFIN R., RIVERA V., SAFINA L., SILLITTI A. (2016) Microservices Science and Engineering. *Proceedings of 5th International Conference in Software Engineering for Defence Applications.*
MCCLURE, J. E., PRINSY, J. F. & MILLER, C. T. (2010) COMPARISON OF CPU AND GPU IMPLEMENTATIONS OF THE LATTICE BOLTZMANN METHOD. IN CARRERA, J. (Ed.) *XVIII International Conference on Water Resources.* Barcelona.
MCNAMARA, G. & ALDER, B. (1993) Analysis of the lattice Boltzmann treatment of hydrodynamics. *Physica A: Statistical Mechanics and its Applications,* 194**,** 218-228.
MEI, R., SHYY, W., YU, D. & LUO, L. S. (2000) Lattice Boltzmann method for 3-D flows with curved boundary. *J. Comput. Phys. ,* 161**,** 680-699.
MERCAN, H. & ATALıK, K. (2009) Vortex formation in lid-driven arc-shape cavity flows at high Reynolds numbers. *European Journal of Mechanics B/Fluids,* 28**,** 61-71.
MEZRHAB, A., AMINE MOUSSAOUI, M., JAMI, M. & NAJI, H. (2010) Double MRT thermal lattice Boltzmann method for simulating convective flows. *Physics Letters A,* 374**,** 3499-3507.
MILLER, W. (1995) Flow in the driven cavity calculated by the lattice Boltzmann method. *Physical Review E,* 51**,** 3659-3669.
MOHAMAD, A. A. (2007) *Applied Lattice Boltzmann Method for Transport Phenomena,* Calgary, Sure.
MOHAMAD, A. A. & KUZMIN, A. (2010) A critical evaluation of force term in lattice Boltzmann method, natural convection problem. *International Journal of Heat and Mass Transfer,* 53**,** 990-996.

MOUFEKKIR, F., MOUSSAOUI, M., MEZRHAB, A., BOUZIDI, M. & LARAQI, N. (2013) Study of double-diffusive natural convection and radiation in an inclined cavity using lattice Boltzmann method. *International Journal of Thermal Sciences,* 63**,** 65-86.

MOUFEKKIR, F., MOUSSAOUI, M. A., MEZRHAB, A., BOUZIDI, M. H. & LEMONNIER, D. (2012) Combined double-diffusive convection and radiation in a square enclosure filled with semitransparent fluid. *Computers & Fluids,* 69**,** 172-178.

MOUSSAOUI, M. A., MEZRHAB, A. & NAJI, H. (2011) A computation of flow and heat transfer past three heated cylinders in a vee shape by a double distribution MRT thermal lattice Boltzmann model. *International Journal of Thermal Sciences,* 50**,** 1532-1542.

MYRE, J., WALSH, S. D. C., LILJA, D. & SAAR, M. O. (2010) Performance analysis of single-phase, multiphase, and multicomponent lattice-Boltzmann fluid flow simulations on GPU clusters. *Concurrency and Computation: Practice and Experience,* 23**,** 332-350.

NAFFOUTI, T. & MAAD, R. B. (2013) Lattice Boltzmann Analysis of 2-D Natural Convection Flow and Heat Transfer within Square Enclosure including an Isothermal Hot Block.

NI, J., ZHANG, Y., LIN, C.-L. & WANG, S. (2003) Parallelization of a Lattice Boltzmann Method for Lid-driven Cavity Flow *upercomputing.*

NITHIARASU, P. & LIU, C.-B. (2005) Steady and unsteady incompressible flow in a double driven cavity using the artificial compressibility (AC)-based characteristic-based split (CBS) scheme. *International Journal for Numerical Methods in Engineering,* 63**,** 380-397.

NOOR, D. Z., KANNA, P. R. & CHERN, M.-J. (2009) Flow and heat transfer in a driven square cavity with double-sided oscillating lids in anti-phase. . *International Journal of Heat and Mass Transfer,* 52**,** 3009-3023.

NOR AZWADI, C. & TANAHASHI, T. (2007) Three-dimensional thermal lattice Boltzmann simulation of natural convection in a cubic cavity. *International Journal of Modern Physics B,* 21**,** 87-96.

NOR AZWADI, C. S. & TANAHASHI, T. (2007) THREE-DIMENSIONAL THERMAL LATTICE BOLTZMANN SIMULATION OF NATURAL CONVECTION IN A CUBIC CAVITY. *International Journal of Modern Physics B,* 21**,** 87-96.

NOURGALIEV, R. R., DINH, T. N., THEOFANOUS, T. G. & JOSEPH, D. (2003) The lattice Boltzmann equation method: theoretical interpretation, numerics and implications. *International Journal of Multiphase Flow,* 29**,** 117-169.

OBRECHT, C., KUZNIK, F., TOURANCHEAU, B. & ROUX, J.-J. (2011) A new approach to the lattice Boltzmann method for graphics processing units. *Computers & Mathematics with Applications,* 61**,** 3628-3638.

OBRECHT, C., KUZNIK, F. D. R., TOURANCHEAU, B. & ROUX, J.-J. (2013) Multi-GPU implementation of the lattice Boltzmann method. *Computers & Mathematics with Applications,* 65**,** 252-261.

ONISHI, J., CHEN, Y. & OHASHI, H. (2001) Lattice Boltzmann simulation of natural convection in a square cavity. *JSME International Journal Series B,* 44**,** 53-62.

OSTASZEWSKI KATHARINA, H. P., RANOCHA HENDRIK (2018) Advantages and pitfalls of OpenCL in computational physics. *Proceedings of the International Workshop on OpenCL (IWOCL '18).* Oxford, United Kingdom.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E. & PURCELL, T. J. (2007) A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum,* 26**,** 80-113.

ÖZSOY, E., RAMBAUD, P., STITOU, A. & RIETHMULLER, M. L. (2005) Vortex characteristics in laminar cavity flow at very low Mach number. *Experiments in Fluids,* 38**,** 133-145.

PACHECO, J., PACHECO-VEGA, A., RODIĆ, T. & PECK, R. (2005) Numerical Simulations of Heat Transfer and Fluid Flow Problems Using an Immersed-Boundary Finite-Volume Method on NonStaggered Grids. *Numerical Heat Transfer: Part B Fundamentals,* 48**,** 267-291.

PACK, D. C. (1977) Theory and Application of the Boltzmann Equation. By CARLO CERCIGNANI. Elsevier, 1975. 415 pp. $35.00. *Journal of Fluid Mechanics,* 81**,** 793-794.

PANANILATH, I., ACHARYA, A., VASISTA, V. & BONDHUGULA, U. (2016) An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations. Indian Institute of Science.

PATIL, D. V., LAKSHMISHA, K. N. & ROGG, B. (2006) Lattice Boltzmann simulation of lid-driven flow in deep cavities. *Computers & Fluids,* 35**,** 1116-1125.

PENG, L., NOMURA, K., T. OYAKAWA, R. K., NAKANO, A. & VASHISHTA, P. (2008) Parallel Lattice Boltzmann Flow Simulation on Emerging Multi-core Platforms. *Springer-Verlag, Berlin, Heidelberg*.

PENG, Y.-F., SHIAU, Y.-H. & HWANG, R. R. (2003) Transition in a 2-D lid-driven cavity flow. *Computers & Fluids,* 32**,** 337-352.

PERSSON MATTSSON, P. (2014) Why Haven't CPU Clock Speeds Increased in the Last Few Years?

POHL, T., DESERNO, F., THUREY, N., RUDE, U., LAMMERS, P., WELLEIN, G. & ZEISER, T. (2004) Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing.* IEEE Computer Society.

PRASAD, A. K. & KOSEFF, J. R. (1996) Combined Forced and Natural Convection Heat Transfer in a Deep Lid-Driven Cavity Flow. *International Journal of Heat and Fluid Flow,* 17**,** 460-467.

QIAN, Y. H., D'HUMIÈRES, D. & LALLEMAND, P. (1992) Lattice BGK Models for Navier-Stokes Equation. *Europhysics Letters,* 17**,** 479-484.

RAMEES, M. P. (2018) Understanding Microservices Architecture. Dot Net Tricks Innovation Pvt. Ltd.

RAVISHEKHAR, B. & KOUSHIK, B. (2013) *OpenCL Programming by Example*, Packt Publishing.

RIBBROCK, D., GEVELER, M., GÖDDEKE, D. & TUREK, S. (2010) Performance and accuracy of Lattice-Boltzmann kernels on multi- and manycore architectures. *Procedia Computer Science,* 1**,** 239-247.

ROSSINELLI, D., BERGDORF, M., COTTET, G.-H. & KOUMOUTSAKOS, P. (2010) GPU accelerated simulations of bluff body flows using vortex particle methods. *Journal of Computational Physics,* 229**,** 3316-3333.

RUPP, K. (2013) CPU, GPU and MIC Hardware Characteristics over Time.

SAHIN, M. & OWENS, R. G. (2003) A novel fully implicit finite volume method applied to the lid-driven cavity problem - Part I: High Reynolds number flow calculations. *International Journal for Numerical Methods in Fluids,* 42**,** 57-77.

SANDERS, J. & KANDROT, E. (2011) *CUDA by Example: An Introduction to General-purpose GPU Programming*, Addison-Wesley.

SETA, T. (2009) Lattice Boltzmann Method for Fluid Flows in Anisotropic Porous Media with Brinkman Equation. *Journal of Fluid Science and Technology,* 4**,** 116-127.
SHADIJA, D., REZAI, M. & HILL, R. (2017) Towards an understanding of microservices. *23rd International Conference on Automation and Computing (ICAC).*
SHAH, P., ROVAGNATI, B., MASHAYEK, F. & JACOBS, G. B. (2007) Subsonic Compressible Flow in Two-Sided Lid-Driven Cavity. Part I: Equal Wall Temperatures. *International Journal of Heat and Mass Transfer,* 50**,** 4206-4219.
SHAH, P., ROVAGNATI, B., MASHAYEK, F. & JACOBS, G. B. (2007) Subsonic compressible flow in two-sided lid-driven cavity. Part II: Un equal walls temperatures. *International Journal of Heat and Mass Transfer,* 50**,** 4219-4228.
SHAN, X. (1997) Simulation of Rayleigh-Bénard convection using a lattice Boltzmann method. *Physical Review E,* 55**,** 2780-2788.
SHAN, X. & CHEN, H. (1993) Lattice Boltzmann model for simulating flows with multiple phases and components. *Physical Review E,* 47**,** 1815-1819.
SHANKAR, P. N. & DESHPANDE, M. D. (2000) Fluid Mechanics in the Driven Cavity. *Annual Review of Fluid Mechanics,* 32**,** 93-136.
SHARIF, M. A. R. (2007) Laminar mixed convection in shallow inclined driven cavities with hot moving lid on top and cooled from bottom. *Applied Thermal Engineering,* 27**,** 1036-1042.
SHARMA, U. R. (2017) *Practical Microservices,* PACKT PUBLISHING.
SHEN, J., FANG, J., SIPS, H. & VARBANESCU, A. L. (2012) Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. *41st International Conference on Parallel Processing Workshops.*
SHI, W., SHYY, W. & MEI, R. (2001) Finite-difference-based lattice Boltzmann method for inviscid compressible flows. *Numerical Heat Transfer, Part B Fundamentals***,** 1-21.
SHI, Y., ZHAO, T. S. & GUO, Z. L. (2004) Thermal lattice Bhatnagar-Gross-Krook model for flows with viscous heat dissipation in the incompressible limit. *Physical Review E,* 70.
SHKLYAR, A. & ARBEL, A. (2003) Numerical method for calculation of the incompressible flow in general curvilinear co-ordinates with double staggered grid. *International Journal for Numerical Methods in Fluids,* 41**,** 1273-1294.

SHKLYAR, A. & ARBEL, A. (2008) Accelerated convergence of the numerical simulation of incompressible flow in general curvilinear co-ordinates by discretizations on the double-staggered grids. *International Journal for Numerical Methods in Fluids,* 57**,** 205-236.
SIEGMANN-HEGERFELD, T., ALBENSOEDER, S. & KUHLMANN, H. C. (2008) Two- and three-dimensional flows in nearly rectangular cavities driven by collinear motion of two facing walls. *Experiments in Fluids,* 45**,** 781-796.
SIMON, M.-S. & DAN, C. (2014) Evaluation of a performance portable lattice Boltzmann code using OpenCL. Proceedings of the International Workshop on OpenCL 2013; 2014. Bristol, United Kingdom, ACM.
SKORDOS, P. A. (1993) Initial and boundary conditions for the lattice Boltzmann method. *Physical Review E,* 48**,** 4823-4842.
STÜRMER, M., GÖTZ, J., RICHTER, G. & RÜDE, U. (2007) Blood flow simulation on the Cell Broadband Engine using the Lattice Boltzmann Method. *Tech. Rep. 07-9, Lehrstuhl f¨ur Systemsimulation, Universit¨at Erlangen-N¨urnberg*.
SUCCI, S. (2001) *The Lattice Boltzman Equation for Fluid Dynamics and Beyond,* Oxford, Oxford University Press.
SUCCI, S. (2007) Applied Lattice Boltzmann Method for Transport Phenomena, Momentum, Heat and Mass Transfer. A. A. Mohamad Sure Printing, Calgary, AB April 2007. *The Canadian Journal of Chemical Engineering,* 85**,** 946-947.
SUKOP, M. C. & THORNE, D. T. J. (2007) *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers,* Berlin, Springer.
TAHER, M., SAHA, S., LEE, Y. & KIM, H. (2013) Numerical Study of Lid-Driven Square Cavity with Heat Generation Using LBM. *American Journal of Fluid Dynamics,* 3**,** 40-47.
TANAHASHI, T. & NOR AZWADI, C. S. (2007) THREE-DIMENSIONAL THERMAL LATTICE BOLTZMANN SIMULATION OF NATURAL CONVECTION IN A CUBIC CAVITY. *International Journal of Modern Physics B,* 21**,** 87-96.
TEIGLAND, R. & ELIASSEN, I. K. (2001) A multiblock/multilevel mesh refinement procedure for CFD computations. *International Journal for Numerical Methods in Fluids,* 36**,** 519-538.
TEKIC, J. B., TEKIC, P. M., RACKOVIC, M. (2014) VISUALISATION OF FLOW AND TEMPERATURE FIELD CALCULATED BY LB

METHOD IN POST-PROCESSING SOFTWARE PARAVIEW. *YU INFO 2014.* Kopaonik.
TEKIC, J. B., TEKIC, P. M., RACKOVIC, M. (2018) Lattice boltzmann method implementation on multiple devices using opencl. *Advances in Electrical and Computer Engineering,* 3**,** 3-8.
TEKIĆ, P. M., RAĐENOVIĆ, J. B., LUKIĆ, N. L. & POPOVIĆ, S. S. (2010) Lattice Boltzmann simulation of two-sided lid-driven flow in a staggered cavity. *International Journal of Computational Fluid Dynamics,* 24**,** 383-390.
TEKIĆ, P. M., RAĐENOVIĆ, J. B. & RACKOVIĆ, M. (2012) Implementation of the Lattice Boltzmann method on heterogeneous hardware and platforms using OpenCL. *Advances in Electrical and Computer Engineering*.
THÜREY, N. (2003.) *A single-phase free-surface Lattice-Boltzmann Method,* FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG.
THYHOLDT, K. C. (2012) Lattice Boltzmann Simulations on a GPU : An optimization approach using C++ AMP.
TOLKE, J. & KRAFCZYK, M. (2008) TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics,* 22**,** 443-456.
TÖLKE, O. (2008) Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*.
TOMOV, S., MCGUIGAN, M., BENNETT, R., SMITH, G. & SPILETIC, J. (2005) Benchmarking and implementation of probability-based simulations on programmable graphics cards. *Computers & Graphics,* 29**,** 71-80.
TUBBS, K. R. & TSAI, F. T. C. (2011) GPU accelerated lattice Boltzmann model for shallow water flow and mass transport. *International Journal for Numerical Methods in Engineering,* 86**,** 316-334.
TUCKER, P. G. & PAN, Z. (2000) A Cartesian cut cell method for incompressible viscous flow. *Applied Mathematical Modelling,* 24**,** 591-606.
VAJDA, A. (2011) *Programming Many-Core Chips*, Springer US.
VALDERHAUG, T. K. (2011) The Lattice Boltzmann Simulation on Multi-GPU Systems. *Department of Computer and Information Science.* Norwegian University of Science and Technology.

VAUGHAN-NICHOLS, S. (2017) A super-fast history of supercomputers: From the CDC 6600 to the Sunway TaihuLight. Hewlett Packard Enterprise Development LP.
VIDAL, D., ROY, R. & BERTRAND, F. (2010) A parallel workload balanced and memory efficient lattice-Boltzmann algorithm. *Computers & Fluids,* 39**,** 1411–1423.
VISHNUVARDHANARAO, E. & DAS, M. K. (2009) Mixed convection in a buoyancy-assisted two-sided lid-driven cavity filled with a porous medium. *International Journal of Numerical Methods for Heat & Fluid Flow,* 19**,** 329-351.
WAGNER, A. J. A Practical Introduction to the Lattice Boltzmann Method.
WAHBA, E. M. (2009) Multiplicity of states for two-sided and four-sided lid driven cavity flows. *Computers & Fluids,* 38**,** 247-253.
WALLS, C. (2016) *Spring Boot in Action*, Manning Publications Co.
WALSH, S. D. C., SAAR, M. O., BAILEY, P. & LILJA, D. J. (2009) Accelerating geoscience and engineering system simulations on graphics hardware. *Computers & Geosciences,* 35**,** 2353-2364.
WANG, C. Y. (2009) The recirculating flow due to a moving lid on a cavity containing a Darcy-Brinkman medium. *Applied Mathematical Modelling,* 33**,** 2054-2061.
WANG YUE, A. M. A., A YI YUN KYU, A CHAN THEODORE C. (2011) T IMPLEMENTING CFD (COMPUTATIONAL FLUID DYNAMICS) IN OPENCL FOR BUILDING SIMULATION. *12th Conference of International Building Performance Simulation Association.* Sydney.
WELLEIN, G., ZEISER, T., HAGER, G. & DONATH, S. (2006) On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids,* 35**,** 910-919.
WILLIAMS, S., CARTER, J., OLIKER, L., SHALF, J. & YELICK, K. A. (2008) Lattice Boltzmann simulation optimization on leading multicore platforms. *IEEE International Symposium on Parallel and Distributed Processing.* IEEE.
WOLF-GLADROW, D. A. (2000) *Lattice-gas cellular automata and lattice Boltzmann Models-an introduction*.
WU, J. S. & SHAO, Y. L. (2004) Simulation of lid-driven cavity flows by parallel lattice Boltzmann method using multi-relaxation-time scheme.
WU, J.-S. & SHAO, Y.-L. (2003) Assessment of SRT and MRT Scheme in Parallel Lattice Boltzmann Method for Lid-Driven Cavity Flows.

The 10th National Computational Fluid Dynamics Conference. Tainan, Taiwan.
XIAN, W. & TAKAYUKI, A. (2011) Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing,* 37**,** 521-535.
YANG, Y., STRAATMAN, A. G., MARTINUZZI, R. J. & YANFUL, E. K. (2002) A study of laminar flow in low aspect ratio lid-driven cavities. *Canadian Journal of Civil Engineering,* 29**,** 436-447.
YANG, Y.-T. & LAI, F.-H. (2011) Numerical study of flow and heat transfer characteristics of alumina-water nanofluids in a microchannel using the lattice Boltzmann method. *International Communications in Heat and Mass Transfer,* 38**,** 607-614.
YOU-SHENG, X., YANG, L. & GUO-XIANG, H. (2004) Using Digital Imaging to Characterize Threshold Dynamic Parameters in Porous Media Based on Lattice Boltzmann Method. *Chinese Physics Letters,* 21.
YU, D., MEI, R., LUO, L.-S. & SHYY, W. (2003) Viscous flow computations with the method of lattice Boltzmann equation. *Progress in Aerospace Sciences,* 39**,** 329-367.
ZDANSKI, P. S. B., ORTEGA, M. A. & FICO, N. G. C. R. (2003) Numerical study of the flow over shallow cavities. *Computers & Fluids,* 32**,** 953-974.
ZHANG, T., SHI, B. & CHAI, Z. (2010) Lattice Boltzmann simulation of lid-driven flow in trapezoidal cavities. *Computers & Fluids,* 39**,** 1977-1989.
ZHEN-HUA, C., BAO-CHANG, S. & LIN, Z. (2006) Simulating high Reynolds number flow in two-dimensional lid-driven cavity by multi-relaxation-time lattice Boltzmann method. *Chinese Physics,* 15**,** 1855-1863.
ZHOU, Y. C., PATNAIK, B. S. V., WAN, D. C. & WEI, G. W. (2003) DSC solution for flow in a staggered double lid driven cavity. *International Journal for Numerical Methods in Engineering,* 57**,** 211-234.
ZOU, Q. & HE, X. (1997) On pressure and velocity boundary conditions for the lattice Boltzmann BGK model *Physics of Fluids,* 9**,** 1591-1598.

Parallel Computing: Background Intel.
(2015) Class diagram.
(2019) Central processing unit. Wikipedia.

(2019) Characteristics of Microservices. Amazon Web Services.
(2019) Computer cluster. Wikipedia.
(2019) History of supercomputing.
(2019) Message Passing Interface. Wikipedia.
(2019) Moore's law. Wikipedia.
(2019) Multi-core processor. Wikipedia.
(2019) OpenMP. Wikipedia.
(2019) Personal computer. Wikipedia.
(2019) Supercomputer. Wikipedia.

Биографија

мр Јелена Текић рођена је 21.03.1980. године у Бачкој Паланци. Школске 1999/2000. године уписује Природно-математички факултет Универзитета у Новом Саду, смер дипломирани информатичар. Дипломирала је 2004. године са просеком 9,1. Дипломски рад „XML и DB2 TexExtender" одбранила је са оценом 10.

На последипломске студије, смер рачунарске науке уписала се школске 2004/2005. године. Све предвиђене испите положила је са оценом десет и у мају 2006. године одбранила је магистарску тезу „Моделирање и имплементација библиотечких каталошких листића помоћу софтверског пакета FreeMarker".

У октобру 2013. године уписала је докторске студије, све предвиђене испите положила је са оценом десет.

У периоду од маја 2005. до фебруара 2006. године била је стипендиста Националне службе за запошљавање. За истаживача приправника изабрана је у јулу 2005. године. Од фебруара 2006. године до маја 2007. године била је запослена на Природно-математичком факултету у Новом Саду на радном месту истраживач-приправник. Држала је вежбе из предмета: Информатика и рачунарска техника (за студенте физике). Од јуна 2007. године до фебруара 2008. године била је запослена у програмерској фирми Интенс као јуниор програмер, а од марта до септембра 2008. године у програмерској фирми Прозон као сениор програмер. Од септембра 2009. године до децембра 2017. године радила је у Служби за управљање људским ресурсима Покрајинске владе као Саветник за информатичке послове. Од јануара 2018. Године запослена је у програмерској фирми Jiway као сениор програмер.

Од фебруара 2006. године до маја 2007. године била је учесник на пројекту Министарства заштите и животне средине – „Апстрактни методи и примене у рачунарским наукама“. Била је учесник пројекта QinR у оквиру европског програма ТЕМПУС. Као учесник овог пројекта у периоду од 12. до 23. Маја 2014. године била је у студијској посети универзитету Париз 8 у Паризу.

Коаутор је 14 нучних радова публикованих у међународним и домаћим часописима и зборницима међународних и домаћих конференција.

Нови Сад, 19.04.2019. Јелена Текић

УНИВЕРЗИТЕТ У НОВОМ САДУ
ПРИРОДНО-МАТЕМАТИЧКИ ФАКУЛТЕТ

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број: **РБР**	
Идентификациони број: **ИБР**	
Тип документације: **ТД**	Монографска документација
Тип записа: **ТЗ**	Текстуални штампани материјал
Врста рада: **ВР**	Докторска дисертација
Аутор: **АУ**	Јелена Текић
Ментор: **МН**	др Милош Рацковић, редовни професор, ПМФ, Нови Сад
Наслов рада: **НР**	Оптимизација CFD симулације на групама вишејезгарних хетерогених архитектура
Језик публикације: **ЈП**	српски (ћирилица)
Језик извода: **ЈИ**	српски/енглески
Земља публиковања: **ЗП**	Република Србија
Уже географско подручје: **УГП**	Војводина
Година: **ГО**	2019
Издавач: **ИЗ**	Ауторски репринт

Место и адреса: **МА**	Природно-математички факултет, Трг Доситеја Обрадовића 4, Нови Сад
Физички опис рада: **ФО**	7/169/231/58/31/0/0
Научна област: **НО**	Информатика
Научна дисциплина: **НД**	Паралелно програмирање
Предметна одредница/ кључне речи: **ПО**	OpenCL, Lattice Boltzman, GPU, many-core, multi-core
УДК	
Чува се: **ЧУ**	Библиотека Департмана за математику и информатику ПМФ-а у Новом Саду
Важна напомена: **ВН**	Нема
Извод: **ИЗ**	Предмет истраживања тезе је из области паралелног програмирања, имплементација CFD (**C**omputational **F**luid **D**ynamics) методе на више хетерогених вишејезгарних уређаја истовремено. У раду је приказано неколико алгоритама чији је циљ убрзање CFD симулације на персоналним рачунарима. Показано је да описано решење постиже задовољавајуће перформансе и на HPC уређајима (Тесла графичким картицама). Направљена је симулација у микросервис архитектури која је портабилна и флексибилна и додатно олакшава рад на персоналним рачунарима.
Датум прихватања теме од НН већа: **ДП**	15.11.2018.

Датум одбране:
ДО
Чланови комисије:
КО

Председник:	Др Срђан Шкрбић, ред. проф., ПМФ, Нови Сад
члан:	Др Милош Рацковић, ред. проф., ПМФ, Нови Сад, ментор
члан:	Др Милош Савић, доцент, ПМФ, Нови Сад
члан:	Др Наташа Лукић, доцент, Технолошки факултет, Нови Сад

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCE

KEY WORDS DOCUMENTATION

Accession number: **ANO**	
Identification number: **INO**	
Document type: **DT**	Monograph publication
Type of record: **TR**	Textual printed material
Content code: **CC**	Doctoral dissertation
Author: **AU**	Jelena Tekić
Mentor/comentor: **MN**	Miloš Racković, Ph. D., full professor
Title: **TI**	Optimization of CFD simulations on groups of many-core heterogeneous architectures
Language of text: **LT**	Serbian (Cyrilic)
Language of abstract: **LA**	English
Country of publication: **CP**	Serbia
Locality of publication: **LP**	Vojvodina
Publication year: **PY**	2019
Publisher: **PU**	Author's reprint
Publication place: **PP**	Faculty of Science, Trg Dositeja Obradovića 4, Novi Sad

Physical description: ***PD***	7/169/231/58/31/0/0
Scientific field: **SF**	Informatics
Scientific discipline: **SD**	Parallel Computation
Subject/ Key words: **SKW** **UC**	OpenCL, Lattice Boltzman, GPU, many-core, multi-core
Holding data: **HD**	Library of Department of Mathematics and Informatics, Trg Dositeja Obradovića 4
Note: **N**	None
Abstract: **AB**	The case study of this dissertation belongs to the field of parallel programming, the implementation of CFD (Computational Fluid Dynamics) method on several heterogeneous multiple core devices simultaneously. The paper presents several algorithms aimed at accelerating CFD simulation on common computers. Also it has been shown that the described solution achieves satisfactory performance on HPC devices (Tesla graphic cards). Simulation is created in micro-service architecture that is portable and flexible and makes it easy to test CFD simulations on common computers.
Accepted by the Scientific Board: **ASB**	November 15, 2018.
Defended on: **DE**	
Thesis defend board: **DB**	

President:	Srđan Škrbić, Ph. D., full prof., Faculty of Science, Novi Sad
Member:	Miloš Racković, Ph. D., full prof., Faculty of Science, Novi Sad - mentor
Member:	Miloš Savić, Ph. D., assistant prof., Faculty of Science, Novi Sad
Member:	Nataša Lukić, Ph. D., assistant prof., Faculty of Technology, Novi Sad