УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Стефан Пијетловић

Предлог за уштеду меморијских ресурса у алгоритму коначних разлика у временском домену коришћењем аритметике у блоковском покретном зарезу

– ДОКТОРСКА ДИСЕРТАЦИЈА –

Ментор:

проф. др Иван Каштелан

Нови Сад, 2023

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5а**
НАВЕСТИ НАЗИВ ФАКУЛТЕТА ИЛИ ЦЕНТРА

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА[1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Стефан Пијетловић
Ментор (титула, име, презиме, звање, институција)	Др Иван Каштелан, ванредни професор, Факултет техничких наука, Универзитет у Новом Саду
Наслов рада:	Предлог за уштеду меморијских ресурса у алгоритму коначних разлика у временском домену коришћењем аритметике у блоковском покретном зарезу
Језик публикације (писмо):	Српски (ћирилица)
Физички опис рада:	Унети број: Страница 92 Поглавља 5 Референци 73 Табела 8 Слика 22 Графикона 0 Прилога 0
Научна област:	Електротехничко и рачунарско инжењерство
Ужа научна област (научна дисциплина):	Рачунарска техника и рачунарске комуникације
Кључне речи / предметна одредница:	Меморијски интензивни проблеми, аритметика у блоковском покретном зарезу, алгоритам коначних разлика у временском домену (FDTD), FPGA
Резиме на језику рада:	Докторска теза анализира предлог за уштеду меморијских ресурса у меморијски интензвином алгоритму. Предмет истраживања је проналажење решења које би услед ефикаснијег руковања меморијом имало и мању потрошњу исте што би резултовало ефикаснијим системом и решењем. Резултат истраживања је симулација која потрвђује хипотезу, као и физичка имплементација која проверава исправност концепта.
Датум прихватања теме од стране надлежног већа:	29.09.2022.
Датум одбране: (Попуњава одговарајућа служба)	
Чланови комисије: (титула, име, презиме, звање, институција)	Председник: Др Никола Теслић, редовни професор Члан: Др Мирослав Поповић, редовни професор Члан: Др Небојша Пјевалица, редовни професор Члан: Др Марио Врањеш, ванредни професор Ментор: Др Иван Каштелан, ванредни професор
Напомена:	

[1] Аутор докторске дисертације потписао је и приложио следеће Обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије и о личним подацима;
5г – Изјава о коришћењу.
Ове Изјаве се чувају на факултету у штампаном и електронском облику и не кориче се са тезом.

UNIVERSITY OF NOVI SAD
FACULTY OR CENTER

KEY WORD DOCUMENTATION[2]

Document type:	Doctoral dissertation
Author:	Stefan Pijetlović
Supervisor (title, first name, last name, position, institution)	PhD Ivan Kaštelan, associate professor, Faculty of Technical Sciences, University of Novi Sad
Thesis title:	Improving finite-difference time-domain memory bandwidth by using block floating-point arithmetic
Language of text (script):	Serbian language (cyrilic)
Physical description:	Number of: Pages 92 Chapters 5 References 73 Tables 8 Illustrations 22 Graphs 0 Appendices 0
Scientific field:	Electrical and Computer Engineering
Scientific subfield (scientific discipline):	Computer engineering and communications
Subject, Key words:	Memory intense problems, Block floating-point arithmetic, Finite-Difference Time-Domain (FDTD), FPGA
Abstract in English language:	PhD thesis analyzes a proposal for improving memory bandwidth in a memory intense algorithm. The purpose of the study is to find a more effective and efficient solution which would use less memory for the same algorithm and thus have better performance. The result of the study is a simulation model which proves the hypothesis as well as a hardware implementation on an FPGA development board which acts as a proof of concept.
Accepted on Scientific Board on:	29.09.2022.
Defended: (Filled by the faculty service)	
Thesis Defend Board: (title, first name, last name, position, institution)	President: PhD Nikola Teslić, full professor, Faculty of Technical Sciences, University of Novi Sad Member: PhD Miroslav Popović, full professor, Faculty of Technical Sciences, University of Novi Sad Member: PhD Nebojša Pjevalica, full professor, Faculty of Technical Sciences, University of Novi Sad Member: PhD Mario Vranješ, associate professor, Faculty of Electrical Engineering, Computer Science and Information Technology Osiek Mentor: PhD Ivan Kaštelan, associate professor, Faculty of Technical Sciences, University of Novi Sad
Note:	

[2] The author of doctoral dissertation has signed the following Statements:
5б – Statement on the authority,
5в – Statement that the printed and e-version of doctoral dissertation are identical and about personal data,
5г – Statement on copyright licenses.
The paper and e-versions of Statements are held at he faculty and are not included into the printed thesis.

Захваљујем се институту и одсеку за Рачунарску технику и рачунарске комуникације, због пружене прилике да радим на овом истраживању, као и свим запосленим који су на било који начин помогли да се ово истраживање спроведе до краја. Овде бих напоменуо и инжењере Trenz-а и Windbond-а чија је техничка подршка помогла оживљавању система.

Захваљујем се ментору др Ивану Каштелану, као и др Небојши Пјевалици и др Миодрагу Ђукићу, који су својим саветима навигирали овај дуготрајан пут.

Посебно се захваљујем колеги др Милошу Суботићу без чије безрезервне техничке подршке и помоћи, ово истраживање можда не би било ни започето, а сигурно не би било и завршено.

На крају, највише се захваљујем породици, пријатељима и девојци Адријани, који су били уз мене и веровали у успешан исход чак и када ја нисам.

САЖЕТАК

Алгоритам коначних разлика у временском домену (eng. Finite-Difference Time-Domain, скраћено FDTD) је алгоритам који се користи за опис понашања електромагнетног поља. Његова главна карактеристика јесте да даје решење Максвелових једначина у временском домену дискретизовањем и просторне и временске компоненте.

Максвелове једначине описују интеракцију између електричног и магнетног поља односно начин на који промена јачине магнетног поља кроз време индукује промену електричног поља и обрнуто – како промена електричног поља утиче на магнетно. Анализом ових једначина могуће је направити моделе простирања таласа кроз одређени простор. FDTD алгоритам је једна од метода која се користи за ову сврху и као такав је применљив у областима попут оптоелектронике, откривања мина и подземних објеката, микроталасне томографије, испитивања и емулације електромагнетне компатибилности као и многих других.

Један од недостатака овог алгоритма јесте да спада у групу меморијски захтевних алгоритама. Главна одлика ове групе јесте да је однос рачунских операција и приступа меморији неповољан за чисто рачунарску имплементацију. За сваку рачунску операцију потребно је неколико приступа меморији што је проблематично будући да су рачунари направљени првенствено с намером да би се бавили рачунарски интензивним проблемима, док су приступи меморији скупи са становишта утрошеног времена. Постоје бројна решења која су се ухватила у коштац са оптимизацијом овог алгоритма и представљена су касније у раду.

Основна хипотеза јесте да би се другачијом представом података, у виду блоковског покретног зареза, могао добити ефикаснији систем који троши мање меморије, чак и за системе који нису прилагођени меморијски интензивним проблемима попут стандардних рачунара. Као провера концепта је направљена симулација система као и провера на физичкој платформи.

ABSTRACT

FDTD (Finite-Difference Time-Domain) is an algorithm used for modelling of the electromagnetic field. Its main benefit, stemming from the name of the algorithm itself, is that it provides a solution to Maxwell's equations in the time domain while discretizing both the space and time component.

Maxwell’s equations describe the interaction between the electric and magnetic field or in other words they describe the way a shifting magnetic field impacts the electric field and vice versa - how the change in the electric field induces change in the magnetic field. By analyzing this behavior, we can model the transient movement of the electromagnetic waves through space, making this algorithm usable in areas such as optoelectronics, mine and underground objects detection, microwave tomography, electromagnetic compatibility and many more.

One downside of this algorithm is that it belongs to the group of memory intense algorithms. The main trait of this group is that the ratio of computing operations and accesses to memory is unfavorable. Several accesses to memory are required per each computing operation which is problematic for most PC implementations because they were designed to handle compute heavy problems. There are many existing solutions which try to handle optimization of this algorithm through various techniques which will be presented further in the paper.

The main hypothesis is that an alternative way of presenting data, such as block floating point, could lead to a more efficient system in terms of memory usage. This is also true even for systems which are not designed to tackle memory intense problems, such as the standard PC. As a proof of concept, a simulation was made as well as a real time implementation on the development board.

САДРЖАЈ

Списак слика

Списак табела

Скраћенице

ASIC	Application Specific Integrated Circuit – интегрисано коло специфичне намене
BFP	Block Floating Point – блоковски покретан зарез
BRAM	Block Random-Access Memory – унутрашња меморија FPGA чипова
CPU	Central Processing Unit – централни процесор
DSP	Digital Signal Processor – процесор за дигиталну обраду сигнала
FDTD	Finite-Difference Time-Domain – метода коначних разлика у временском домену
FFT	Fast Fourier Transform – брза Фуријеова трансформација
FPGA	Field Programmable Gate Array – програмабилне логичке мреже
GPU	Graphical Processing Unit – графички процесор
OpenCL	Open Computing Language – оквир за писање преносивих програма за хетерогене системе
SDRAM	Synchronous Dynamic Random-Access Memory – синхрона динамичка меморија која се налази ван FPGA чипа
SIMD	Single Instruction Multiple Data – једна инструкција, више података
USB	Universal Serial Bus – универзална серијска магистрала

ПОГЛАВЉЕ 1.

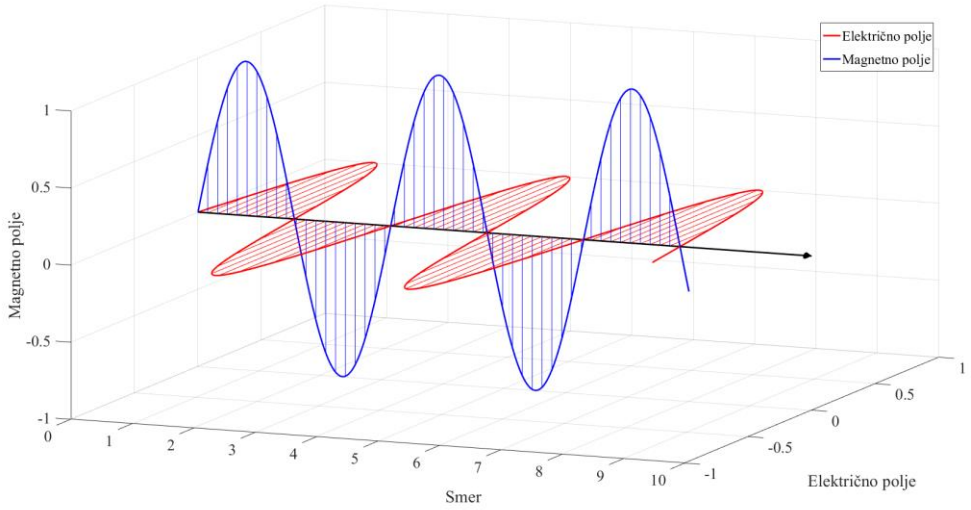
Увод

Електромагнетика се као област физике формира крајем 19. века откирићма научника попут Мајкла Фарадеја, Џејмса Клерка Максвела, Хајнриха Херца и других. Иако су и електрична и магнетна сила биле познате човечанству вековима (поготово магнетна), обе су сматране засебним силама и као такве су и проучаване. Неизоставна веза између њих је откривена тек након серије успешних експеримената и објављених научних радова горе поменутих људи, тако да се може рећи да је област као таква настала и кренула да се развија тек пре око век и по. Упркос релативно кратком времену постојања, утицај електромагнетике на развој човечанства је немерљив. Само откриће струје као енергије је увело човечанство у другу индустријску револуцију, док су електроника и електротехника, као дисциплине произашле из електромагнетике, потпомогле прелазак човечанства у трећу и четврту индустријску револуцију и из корена променили начин живота просечног човека и произбодњу добара [1].

Са појавом рачунара у другој половини 20. века, многи проблеми који су до тада били нерешиви због потребе за дуготрајним и прецизним рачунањем су постали решиви. Међутим, нова техничка достигнућа су довела и до открића потпуно нових проблема који су били везани за недостатке тада актуелне технологије. Један од таквих проблема је, између осталих, и моделовање простирања електромагнетних таласа, који изискује много меморијских ресурса.

У мноштву метода које се баве датом тематиком издваја се FDTD алгоритам као основа ове тезе, који је развио кинеско-амерички математичар Кејн Ји [2]. Алгоритам даје решење Максвелових једначина и одговарајућих конститутивних једначина у временском домену дискретизацијом и просторне и временске компоненте.

Максвелове једначине представљају скуп једначина које су темељ електромагнетизма и описују како промена једног поља у времену утиче на промену другог и обрнуто. Из ових једначина се могу извести једначине за ажурирање поља, који уз конститутивне једначине представљају основу алгоритма (где је главна улога конститутивних једначина да моделују интеракцију са простором који није вакуум).



Слика 1 - Илустрација везе између електричног и магнетног поља

Сам алгоритам ради на следећи начин: најпре се простор који се посматра подели на мрежу која се у част изумитељу назива Јиова мрежа или Јиова решетка. У случају да се не посматра простирање таласа у равни него у тродимензионалном простору, онда се мрежа претвара у тензор (тродимензионалну мрежу). Потом се у простор убацује извор електромагнетног поља. У главној петљи алгоритма се затим из итерације у итерацију, користећи конститутивне једначине, ажурирају вредности и електричног и магнетног поља за сваки елемент у Јиовој мрежи. Било који објекат који постоји у посматраном простору ће у зависности од својих физичких особина у мањој или већој мери одбијати, упијати и преламати

електромагнетне таласе који се шире у концентричним круговима од извора. Након сваке итерације је по потреби могуће графички представити вредности електричног и магнетног поља у сваком елементу мреже и повезивањем низа слика у филм се може ефикасно визуелизовати простирање електромагнетних таласа у посматраном простору.

Постоје разни критеријуми који се могу одредити за крај симулације од којих су најчешћи или неки коначан унапред познат број итерација главне петље или чекање док таласни фронт не прође у потпуности кроз систем. У реалном свету, таласи би прошли кроз неки простор и потом наставили даље да путују у бесконачност са извесним слабљењем. Међутим у условима у којима се користи FDTD, Јиова мрежа је најчешће доста мала и унутар ње би се таласи одбијали од ивице мреже као од савршених зидова, што је у реалном свету немогуће. Како ово не би био случај, на саму ивицу се додаје посебан слој који упија граничне услове што даје реалнију симулацију да је таласни фронт који је ударио у ивицу заправо напустио посматрани простор [3], [4].

FDTD алгоритам као метода за моделовање електромагнетних таласа има своје предности од којих је најбитније издвојити да са повећањем симулације и модела, сам алгоритам скалира скоро линеарно, као и да је добар за описивање нелинеарног понашања. Неке од основних мана јесу лоша представа закривљених површина због квадратног облика Јиове мреже, као и неефикасна представа веома резонантних објеката [5]. Узимајући све ово у обзир, алгоритам је пронашао своју примену у областима као што су дизајн антена [6], [7], [8] оптоелектоника [9], откривање мина [10], микроталасна томографија [11], [12], [13] испитивање електромагнетне компатибилности [14] и многе друге.

Остатак дисертације је организован у следеће целине:

У поглављу 2 је дат преглед стања у области. Објашњена је кратка историја развоја FDTD алгоритма и наведени неки од битних резултата као и њихова примена. Такође су продискутовани и постојећи проблеми као и методологије које се користе да би оне биле превазиђене као и позиција овог истраживања у целом систему.

Поглавље 3 садржи неопходне теоријске основе. Детаљније су описани електромагнетни таласи и пропратна физика, сам FDTD алгоритам и направљен је

кратак осврт о аритметици у рачунарима са посебним акцентом на аритметику с блоковским покретним зарезом чија употреба у овом новом контексту представља главни допринос истраживања.

Поглавље 4 представља теоријско решење проблема и имплементацију решења. У њему су описани сви спроведени кораци у истраживању, еволуција FDTD алгоритма како би могао да ради са аритметиком с блоковским покретним зарезом, као и транзиција у решење на платформи која користи програмабилну логику односно FPGA као потврду концепта.

Последње, 5. поглавље садржи резултате симулације и резултате добијене након извршавања алгоритма на наменској платформи. У овом поглављу је дата и дискусија као и предлози за будући рад.

На самом крају рада се налази преглед литературе као и кратка биографија аутора.

ПОГЛАВЉЕ 2.

Стање у области

FDTD алгоритам је објављен још давне 1966. године од стране кинеско-америчког математичара Кејн Јиа [2]. У овом раду, Ји описује свој скуп једначина са којим замењује изворне Максвелове једначине и илуструје их на примеру расејавања електромагнетног пулса о савршено проводни цилиндар. У раду су објашњене све апроксимације које су основ за сву даљу еволуцију алгоритма кроз деценије које ће доћи.

Упркос томе што сам алгоритам постоји већ дуже од пола века, није пронашао ширу примену све до скоријих времена. Главни разлог за то су пре свега били ограничени рачунарски ресурси који нису имали довољно велику моћ рачунања да реше Максвелове једначине у догледно време како би њихов резултат био применљив на терену [15]. Овако нешто је и сам Ји предвиђао у свом раду и окарактерисао као једно од највећих препрека саме методе у тренутку настанка.

Један од основних узрока за релативно слабу примену алгоритма до скоријих времена јесте што је FDTD у својој основи меморијски интензиван проблем. Проблеми из ове групе за свако појединачно рачунање захтевају неколико приступа меморији који су, са становишта процесорског времена, веома скупи. Самим тим FDTD је у последњих пар децинија заинтересовао многе ауторе, превасходно оне активне у пољу оптимизације. Како би се перформансе овог алгоритма побољшале, он је успешно имплементиран на вишејезгарним

архитектурама [16]. У овом раду, аутори су користили тада и даље релативно нове а данас стандардне технике паралелног програмирања као што су екстензије за низове и употреба оквира за вишејезгарно програмирање као што је OpenMP [17]. Напредна проширења за низове (*енг*. Advanced Vector Extensions или скраћено AVX) су заправо проширења у инструкцијском скупу процесора који су се појавили крајем прве деценије 21. века. Њихова главна предност јесте што омогућавају да се иста инструкција (математичка операција) изврши над више података одједном (такозвана прагма SIMD – Single Instruction Multiple Data). У зависности од генерације процесора, разликује се и број наменских регистара за овај тип инструкција па самим тим и степен паралелизма који је могуће постићи. OpenMP (скраћено од Open Multi-Processing) с друге стране представља један оквир (*енг*. framework) који са својим директивама и рутинама омогућује упошљавање свих језгара унутар савремених вишејезгарних процесора. Употребом ових техника као и многих других наведених у раду, аутори су добили убрзања од три до неколико десетина пута у односу на почетна референтна решења.

Даља побољшања су постигнута од тренутка када су се у обраду укључиле графичке процесне јединице (GPU) [18], [19], [20], [21]. У овим радовима аутори су најинтензивнији рачун изместили у графичке картице које имају најчешће велики број процесних језгара у односу на модерне процесоре који имају мали број, али зато знатно моћнијих. Уз додатне оптимизације, добијена су убрзања реда величине неколико десетина пута у зависности од комбинације параметара и сложености почетног проблема. Највећа убзрања су добијена за највеће димензије што је и природно за очекивати јер је тада могуће у паралели обавити највећи део посла.

Наредна серија убрзања су постигнута укључивањем у обраду и програмибилне логике (FPGA) [22], [23], [24], [25], [26]. FPGA представља компромис између флексибилности и брзине у поређењу са чисто софтверском имплементацином на процесору и интегрисаних кола специфичне намене (*енг*. Application Specific Integratet Circuit) тј. ASIC-а. Они се састоје од мноштва логичких блокова које је могуће конфигурисати по потреби. Такође, с обзиром на то да пројектанти хардвера имају потпуну контролу, могуће је директно у самом

хардверу имплементирати технике за оптимизацију као што је нпр. проточна обрада (*енг*. pipeline). У овим радовима су убрзања у односу на референтну имплементацију са покретним зарезом у софтверу већи за пар редова величине. Разлози за ово су разни. За почетак, иако у нумерици већина метода захтева двоструку прецизност покретног зареза, FDTD је могуће реализовати у једнострукој прецизности без већег губитка прецизности нити нумеричке нестабилности. Због природе FDTD-а која је таква да барата увек са коначним величинама поља, алгоритам је могуће реализовати и у непокретном зарезу. Оваква представа је знатно једноставнија, бржа и погоднија за FPGA имплементацију јер захтева мање логичких елемената. Даље, објашњено је да је главна препрека у обради спрега процесора са меморијом, такозвано Von Neumann-ово уско грло [27], [28] – проблем који постоји од настанка рачунара и прихватање ове архитектуре као доминантне. Самим тим, знатно већа радна фреквенција процесора није у потпуности искоришћена због чекања на податке из меморије. То је и за очекивати јер су процесори пројектовани да решавају изузетно широк дијапазон проблема од којих меморијски интензивни нису примарно узети у обзир.

Са друге стране спектра у односу на софтверска решења би било пројектовање у потпуности наменских интегрисаних кола ASIC-а само за потребе FDTD-а. ASIC-и имају убедљиво најбоље перформансе и најефикасније раде у поређењу са свим осталим решењима. Њихова главна мана јесте да су непроменљиви, поред изузетно дугог и скупог процеса развоја и верификације који се тешко може ублажити. Једном када је направљен, ASIC може да обавља само функционалност за коју је пројектован. Чак и у случају нових напредака у еволуцији самог алгоритма, те иновације је немогуће убацити уколико захтевају другачију структуру већ траже нову верзију самог чипа. У литератури се могу пронаћи покушаји ASIC реализације FDTD-а који нису узели маха [29], [30], [31]. Главни разлог за то је вероватно то што FDTD није толико распрострањен алгоритам да би оправдао изузетно скуп развој ових наменских интегрисаних кола.

У последње време, развојем програмских оквира као што су OpenCL (Open Computing Language) је постало могуће упослити још већи број рачунарских

компоненти на једној машини [32], [33]. OpenCL је отишао још један корак даље по питању брзине развоја и флексибилности у односу на претходна решења. То је омогућено уз помоћ такозваних функција језгра (*енг.* Kernel funciton) које се могу извршавати и на процесорима, и на графичким картицама и на FPGA чиповима. Аутори говоре о додатним оптимизацијама и напрецима у развоју како физичких компоненти тако и самих алата како би се овај проблем што ефикасније решио, што иде у прилог потребом за даљом еволуцијом методе. Такође напомињу да је поводом неких критичних тачака инжењерска експертиза и даље незаменљива модерним алатима.

Све у свему, решења базирана на графичким картицама су се до сада показала као најбоља, с тим да треба напоменути да није искоришћен њихов максимални потенцијал.

Упркос својим неоспорним доприносима, један недостатак већине поменутих решења која су базирана на Von Neumann-овој архитектури јесте што су она и даље базирана на оптимизацији рачунарског дела алгоритма а недовољно пажње је посвећено главном проблему а то је огромна количина меморије која се користи. Von Neumann-ова архитектура се састоји од следећих елемената: аритметичко-логичке јединице и одговарајућих регистара, управљачке јединице која садржи регистар за тренутну инструкцију и програмски бројач, заједничку меморију за податке и инструкције, улазно-излазне подсистеме и спољашњу масовну меморију. У својој сржи, оваква архитектура се увек ослања на процесор као централну тачку обраде. Самим тим је подразумевано рачунски оријентисана, и није оптимална за меморијски интензивне проблеме код којих највећи део посла не обавља процесор – највише времена одлази на читање и упис података. Иако је развој технологије учинио данашње меморије све бржим и бржим, и даље је за време једног приступа меморији могуће урадити неколико (некада и неколико стотина или хиљада) рачунских операција, у зависности од тога где се налазе подаци (да ли је то у скривеној или радној меморији или на масовној меморији односно диску). Последица је када имамо пуно приступа меморији, добар део процесорског времена се троши у празно, чекајући да се подаци добаве што чини његов изузетно висок радни такт узалудним.

Као покушај да се овај недостатак отклони или барем ублажи, предложено је решење у потпуности базирано на FPGA имплементацији [34]. Након анализе тржишта и доступних FPGA чипова, направљен је закључак да је, спрам становишта приступа меморији, ефикасније имати неколико малих FPGA чипова који раде у паралели на истом задатку (самим тим остварујући већи паралелни приступ меморијама) него један велики и моћнији. Главни циљ овог истраживања је био пронаћи алтернативу решењима базираним на употреби графичких картица, пре свега због њихове велике цене и потрошње енергије. Ипак, није било могуће достићи перформансе постојећих решења са оваквим приступом. Главни разлог за то је изванредан проток података које обезбеђују наменске меморије у графичким картицама пете генерације (GDDR5 SDRAM) у поређењу са меморијама компатибилим са одабраним FPGA чиповима, које су старије генерације. Овако добре перформансе графичких картица је наметнуло тржиште због тога што су оне постале готово неизоставни део данашњих личних рачунара. Велика потражња је проузроковала изузетно јаку конкуренцију међу произвођачима и самим тим и пад цене (који додуше јесте бележио и нагле растове у одређеним периодима модерне историје када су графичке картице постале изузетно популарне у одређеним апликацијама као што је рударење криптовалута). Употреба FPGA је и даље веома скромна у поређењу са GPU. До истог закључка су дошли и остали аутори који су упоређивали ефикасност и перформансе графичких картица и FPGA. Заједничко за сва досадашња истраживања јесте да графичке картице и даље дају најбоље перформансе просто због чињенице да су оптимизоване за изузетно велики проток података, упрткос томе што су решења која укључују или су у потпуности базирана на FPGA технологији ефикаснија.

Као следећи корак у истраживању се потом наметнуо правац ефикаснијег руковања меморијом. С обзиром да је иницијална идеја и даље остала иста а то је алтернатива која би била јефтинија и енергетски ефикаснија, као решење је дошла компресија података. Идеја је била користити мање бита за представу података без значајних губитака у виду прецизности, и тада се као могуће решење јавља блоковски покретни зарез.

Блоковски покретни зарез представља нестандарни формат аритметике и нуди добар однос између смањене комплексности аритметике у непокретном

зарезу и проширеног динамичког опсега аритметике са покретним зарезом који је погодан за имплементацију оваквог типа нумеричких решења [34]. Он се типично среће у апликацијама и алгоритмима који су погодни за процесоре са непокретним зарезом као што су процесори за обраду сигнала - DSP процесори. Неки од најпознатијих примера где се користи блоковски покретни зарез јесу алгоритми базирани на брзој Фуријеовој трансформацији (FFT - Fast Fourier transform) [35], [36], [37] неуронске мреже [38], множење матрица [39], дигитални филтри [40], [41], комуникациони системи [42] и други. Употреба блоковског покретног зареза у контексту FDTD алгоритма јесте (према сазнањима аутора у тренутку писања тезе) новина и представља један од главних доприноса ове тезе.

Представа података у виду блоковског покретног зареза би требала да омогући прибилжно исту тачност алгоритма али уз мањи утрошак, и самим тим и мање приступа меморији. Пошто је приступ меморији критичан за све архитектуре базиране на Von Neumann-овој архитектури, хипотеза је да би оваква измена побољшала перформансе и отворила прилику за нова истраживања на ову тему.

ПОГЛАВЉЕ 3.

ТЕОРИЈСКЕ ОСНОВЕ

У овом поглављу су описане теоријске основе на којима је заснована докторска дисертација. Изложени су основни принципи који се тичу физике електромагнетних таласа и дат опис самог FDTD алгоритма. Потом следи преглед аритметике у рачунарима и на крају детаљнији увид у аритметику блоковског покретног зареза и његову примену у контексту овог алгоритма.

3.1 FDTD алгоритам

У електромагнетици постоји мноштво метода које описују понашање електромагнетних таласа у датом простору. Оно што је заједничко јесте да важе Максвелове једначине било да раде у временском или у фреквентном домену [44], [45]. FDTD представља скраћеницу од енглеских речи *Finite-Difference Time-Domain* што у слободном преводу значи коначна разлика (коначни корак) у временском домену. Већ из самог имена се види у коју групу FDTD алгоритам спада и пошто је резултат модела симулација, он такође спада и у групу директних метода. FDTD проналази апроксимације одговарајућих система диференцијалних једначина на прегледан начин, у широком спектру фреквенција и у разним окружењима. Због својих предности, алгоритам је нашао примену у многим областима, почевши од чисто електротехничких попут дизајнирања антена и

електромагнетне компатибилности па све до медицине. FDTD алгоритам се може користити самостално [46], [47], [48] или у комбинацији / спрегнуто са још неким методама [45], [49].

Као што је већ напоменуто, FDTD полази од Максвелових једначина у диференцијалном облику, и то су следеће четири:

- Гаусов закон:

$$\nabla \cdot \mathbf{E} = 0 \tag{1}$$

- Гаусов закон за магнетизам:

$$\nabla \cdot \mathbf{H} = 0 \tag{2}$$

- Максвел-Фарадејев закон индукције:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{3}$$

- Амперов закон са Максвеловим додатком:

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{4}$$

У овим једначинама фигуришу следеће величине: **E** представља јачину електричног поља, **D** је електрична индукција односно вектор диелектричног помераја, **H** је јачина магнетног поља док је **B** густина магнетног флукса, а **J** густина струје. Треба напоменути да је дат облик Гаусовог закона без статичког наелектрисања које није од интереса за посматрани случај и примену алгоритма. Од интереса су и две конститутивне једначине које повезују једначине 3 и 4:

$$\mathbf{D} = \varepsilon \mathbf{E} \tag{5}$$

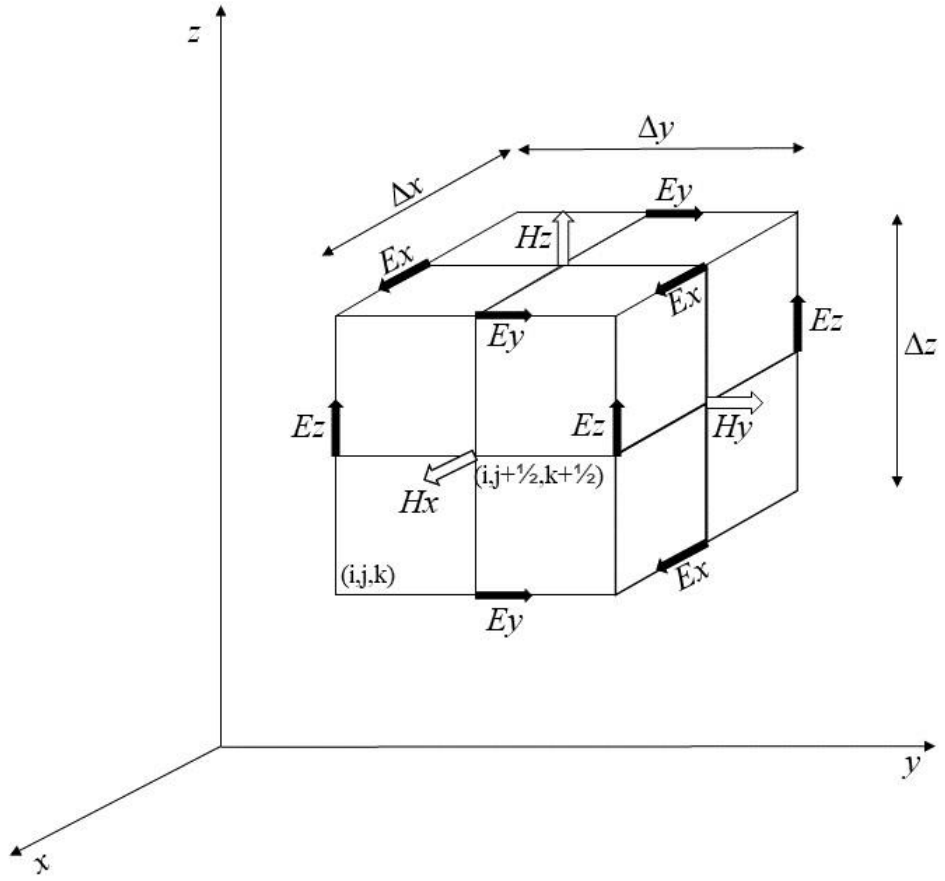
$$\mathbf{B} = \mu \mathbf{H} \tag{6}$$

Од нових величина које се појављују у овим једначинама имамо **ε** који представља пермитивност материјала и описује његово понашање у електричном пољу и њему аналоган **μ** која представља пермеабилност материјала и описује понашање материјала у магнетном пољу. Ове вредности су резултат производа две константе – пермитивности/пермеабилности вакуума и релативне пермитивности/пермеабилности материјала.

$$\varepsilon = \varepsilon_r \varepsilon_0 \tag{7}$$

$$\mu = \mu_r \mu_0 \tag{8}$$

Посматрани простор се дели на мрежасту структуру чије су ћелије дужи, квадрати или коцке, у зависности од димензионалности простора. Ова структура се зове Јиова мрежа или Јиова решетка.



Слика 2 - Изглед једне ћелије Јиове мреже

Δx, **Δy** и **Δz** представљају димензије ћелије, док **i**, **j** и **k** представљају индексе односно координате сваке ћелије унутар Јиове мреже. Јачина електричног и магнетног поља су растављене на компоненте тако да свака електрична компонента има 4 магнетне које је оркужују, исто као што и свака магнетна има 4 електричне које је окружују (у случају тродимензионалног проблема). Као што се може приметити са слике, поља су „смакнута“ за пола димензије ћелије. Потребно је још напоменути да електрично поље око себе индукује одговарајуће ротирајуће магнетно поље за које важи правило десне руке. С обзиром на разлике у вредностима електричног и магнетног поља (када се користе стандардне јединице из SI система) које могу бити пар редова величине, у свим једначинама које ће

уследити су вредности нормализоване. Главни разлог за то јесте већа прецизност приликом рачунања на рачунару.

Нормализоване Максвелове једначине се сада могу написати у скаларном облику на следећи начин:

$$\frac{\partial \mathbf{E}_x}{\partial \mathbf{x}} + \frac{\partial \mathbf{E}_y}{\partial \mathbf{y}} + \frac{\partial \mathbf{E}_z}{\partial \mathbf{z}} = \mathbf{0} \tag{9}$$

$$\frac{\partial \mathbf{H}_x}{\partial \mathbf{x}} + \frac{\partial \mathbf{H}_y}{\partial \mathbf{y}} + \frac{\partial \mathbf{H}_z}{\partial \mathbf{z}} = \mathbf{0} \tag{10}$$

$$\left(\frac{\partial \mathbf{E}_z}{\partial \mathbf{y}} - \frac{\partial \mathbf{E}_y}{\partial \mathbf{z}}\right)\boldsymbol{i} + \left(\frac{\partial \mathbf{E}_x}{\partial \mathbf{z}} - \frac{\partial \mathbf{E}_z}{\partial \mathbf{x}}\right)\boldsymbol{j} + \left(\frac{\partial \mathbf{E}_y}{\partial \mathbf{x}} - \frac{\partial \mathbf{E}_x}{\partial \mathbf{y}}\right)\boldsymbol{k} = -\frac{\mu_r}{\boldsymbol{C_0}}\left(\frac{\partial \mathbf{H}_x}{\partial \mathbf{t}}\boldsymbol{i} + \frac{\partial \mathbf{H}_y}{\partial \mathbf{t}}\boldsymbol{j} + \frac{\partial \mathbf{H}_z}{\partial \mathbf{t}}\boldsymbol{k}\right) \tag{11}$$

$$\left(\frac{\partial \mathbf{H}_z}{\partial \mathbf{y}} - \frac{\partial \mathbf{H}_y}{\partial \mathbf{z}}\right)\boldsymbol{i} + \left(\frac{\partial \mathbf{H}_x}{\partial \mathbf{z}} - \frac{\partial \mathbf{H}_z}{\partial \mathbf{x}}\right)\boldsymbol{j} + \left(\frac{\partial \mathbf{H}_y}{\partial \mathbf{x}} - \frac{\partial \mathbf{H}_x}{\partial \mathbf{y}}\right)\boldsymbol{k} = -\frac{\boldsymbol{\varepsilon_r}}{\boldsymbol{C_0}}\left(\frac{\partial \mathbf{E}_x}{\partial \mathbf{t}}\boldsymbol{i} + \frac{\partial \mathbf{E}_y}{\partial \mathbf{t}}\boldsymbol{j} + \frac{\partial \mathbf{E}_z}{\partial \mathbf{t}}\boldsymbol{k}\right) \tag{12}$$

Ове једначине се могу поделити по координатама без утицаја на остале једначине. На пример уколико посматрамо само $\boldsymbol{i}$ по $\boldsymbol{x}$ оси добили бисмо следећи систем једначина:

$$\left(\frac{\partial \mathbf{E}_z}{\partial \mathbf{y}} - \frac{\partial \mathbf{E}_y}{\partial \mathbf{z}}\right) = -\frac{\mu_r}{\boldsymbol{C_0}}\left(\frac{\partial \mathbf{H}_x}{\partial \mathbf{t}}\right) \tag{13}$$

$$\left(\frac{\partial \mathbf{H}_z}{\partial \mathbf{y}} - \frac{\partial \mathbf{H}_y}{\partial \mathbf{z}}\right) = -\frac{\boldsymbol{\varepsilon_r}}{\boldsymbol{C_0}}\left(\frac{\partial \mathbf{E}_x}{\partial \mathbf{t}}\right) \tag{14}$$

Применом истог принципа се добијају и једначине за остале координате. Ово је важно јер се у случају дводимензионалног FDTD (који је коришћен у овој тези)

могу занемарити сви изводи по **z** оси. Следећи корак јесу саме апроксимације које се опет могу применити на сваког од чланова, овде су дати примери за **z** компоненте. Потребно је извршити дискретизацију по просторној компоненти:

$$\frac{\partial \mathbf{E_z}}{\partial \mathbf{y}} \Rightarrow \frac{\mathbf{E_z}\Big|_{t}^{i,j+1,k} - \mathbf{E_z}\Big|_{t}^{i,j,k}}{\Delta \mathbf{y}} \tag{15}$$

$$\frac{\partial \mathbf{E_z}}{\partial \mathbf{x}} \Rightarrow \frac{\mathbf{E_z}\Big|_{t}^{i+1,j,k} - \mathbf{E_z}\Big|_{t}^{i,j,k}}{\Delta \mathbf{x}} \tag{16}$$

$$\frac{\partial \mathbf{H_z}}{\partial \mathbf{y}} \Rightarrow \frac{\mathbf{H_z}\Big|_{t+\frac{\Delta t}{2}}^{i,j,k} - \mathbf{H_z}\Big|_{t+\frac{\Delta t}{2}}^{i,j-1,k}}{\Delta \mathbf{y}} \tag{17}$$

$$\frac{\partial \mathbf{H_z}}{\partial \mathbf{x}} \Rightarrow \frac{\mathbf{H_z}\Big|_{t+\frac{\Delta t}{2}}^{i,j,k} - \mathbf{H_z}\Big|_{t+\frac{\Delta t}{2}}^{i-1,j,k}}{\Delta \mathbf{x}} \tag{18}$$

И по временској компоненти:

$$\frac{\partial \mathbf{H_z}}{\partial \mathbf{t}} \Rightarrow \frac{\mathbf{H_z}\Big|_{t+\frac{\Delta t}{2}}^{i,j,k} - \mathbf{H_z}\Big|_{t-\frac{\Delta t}{2}}^{i,j,k}}{\Delta \mathbf{t}} \tag{19}$$

$$\frac{\partial \boldsymbol{D_z}}{\partial \mathbf{t}} \Rightarrow \frac{\mathbf{D_z}\Big|_{t+\Delta t}^{i,j,k} - \mathbf{D_z}\Big|_{t}^{i,j,k}}{\Delta \mathbf{t}} \tag{20}$$

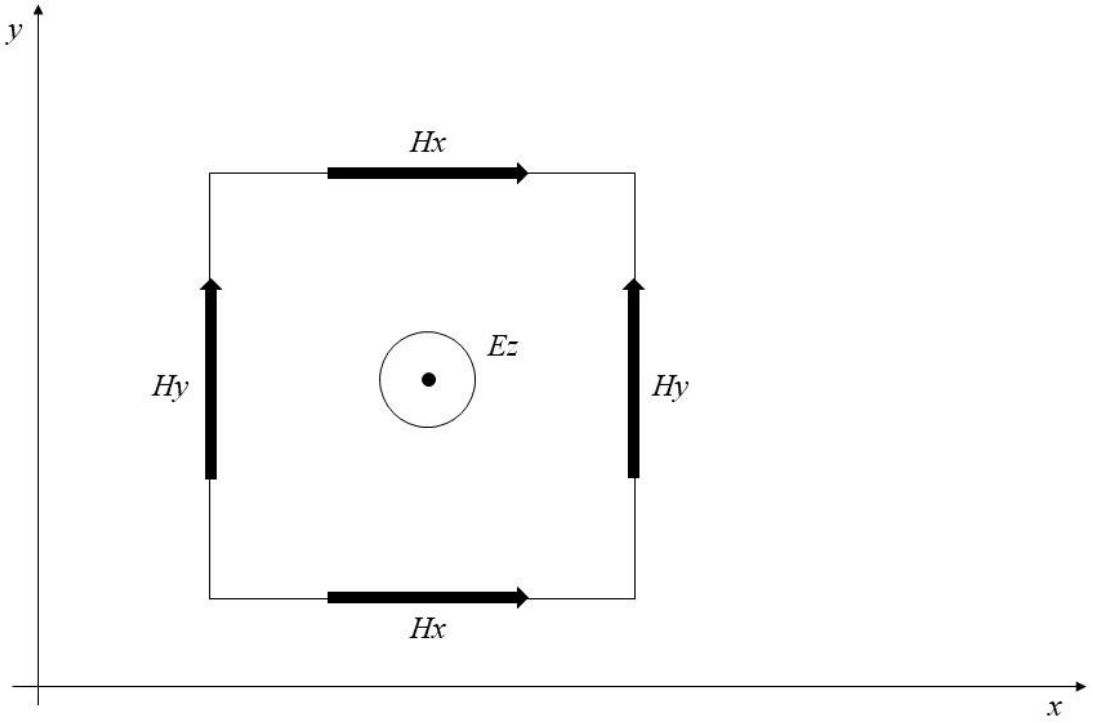
Како је H смакнут у односу на D и у времену и у простору, зато и постоји разлика у формулама дискретизације. Разлог за „смакнуће“ јесте то што се на основу вредности H у две суседне ћелије рачуна једно D и обрнуто – на основу вредности D две суседне ћелије се освежава вредност H.

По истом принципу се могу извести и једначине за све чланове и преостале компоненте. У случају дводимензионалног FDTD-а, сви парцијални изводи по **z** оси су једнаки 0. Последица тога јесте да се сам рачун поједностављује јер се $\mathbf{E}_{\boldsymbol{x}}$, $\mathbf{E}_{\boldsymbol{y}}$ и $\mathbf{H}_{\boldsymbol{z}}$ могу изоставити док је све преостале величине могуће добити користећи само $\mathbf{E}_{\boldsymbol{z}}$, $\mathbf{H}_{\boldsymbol{x}}$ и $\mathbf{H}_{\boldsymbol{y}}$.

Након апроксимација остаје још формирање корачних једначина што је илустровано на примеру јачине магнетног поља по **x** оси:

$$\mathbf{H}_x\Big|_{t+\frac{\Delta t}{2}}^{x+\frac{\Delta x}{2},\,y+\frac{\Delta y}{2}} = \mathbf{H}_x\Big|_{t-\frac{\Delta t}{2}}^{x+\frac{\Delta x}{2},\,y+\frac{\Delta y}{2}} - \frac{\Delta t}{\mu}\frac{\boldsymbol{E}_z\Big|_{t}^{x,\;y+\Delta y} - \mathbf{E}_z\Big|_{t}^{x,y}}{\mathbf{\Delta y}} \tag{21}$$

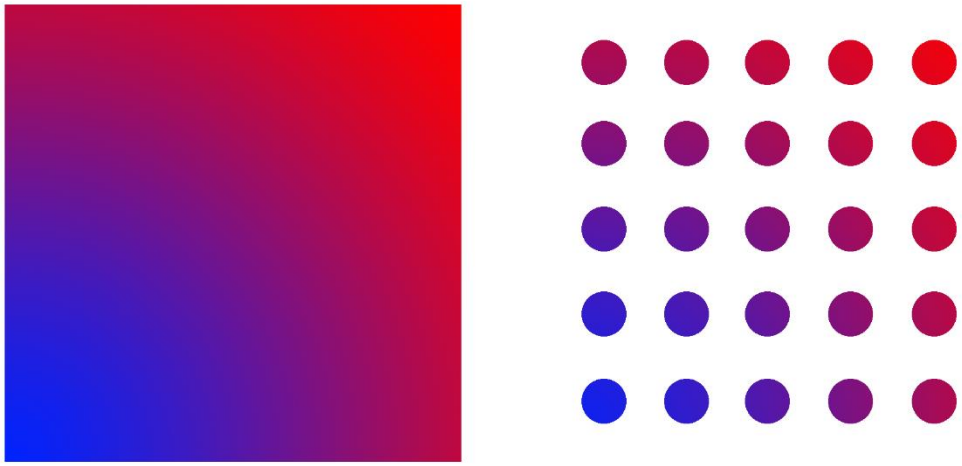
Δt представља временски корак док **Δx** и **Δy** представљају просторне кораке по одговарајућим осама. $\mathbf{H}_{\boldsymbol{x}}$ је **x** компонента вектора магнетног поља **H** док је $\mathbf{E}_{\boldsymbol{z}}$ **z** компонента вектора електричног поља **E**. Из ове формуле се може видети да се на основу вредности E поља у тренутку **t** (садашњост) и **H** поља у тренутку $\mathbf{t} - \frac{\mathbf{\Delta t}}{\mathbf{2}}$ (прошлост у односу на **t**) рачуна **H** поље у тренутку $\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}$ (будућност). Такође се може приметити да се користе **E** са позиција **y** + **Δy** и **y** које су испред и иза позиције $\mathbf{y} + \frac{\mathbf{\Delta y}}{\mathbf{2}}$ у ком је **H**. Зарад стабилности алгоритма, просторни кораци морају бити мањи од најкраће таласне дужине која се појављује у симулацији, као што и временски корак мора бити довољно мали како не би долазило до великих промена између две итерације. Треба споменути да и даље важи Никвистова фреквенција односно фреквенција одабирања мора бити барем 2 пута већа од највеће фреквенцијске компоненте која се појављује у проблему. У практичним имплементацијама FDTD-а то типично иде и знатно више.



Слика 3 - Изглед једне ћелије Јиове мреже за дводимензионални случај

На алгоритамском нивоу, FDTD се састоји из главне петље и визуелизације. У оквиру главне петље се током сваке итерације прво ажурирају вредности електричног поља спрам магнетног а затим у истој итерацији вредности магнетног поља ажурирају спрам промена електричног. Све док не буде промена код једног поља, неће их бити ни код другог. Овај процес је илустрован на Слика 5.

Први корак јесте убацивање извора. Тај извор може бити један електромагнетни импулс или може бити и неки извор који ће константно одашиљати електромагнетне таласе. Извор ће, наравно, изменити вредност поља у континуалном простору и то прво у својој непосредној околини, а затим ће се таласи простирати у свим правцима. Међутим, како радимо у свету рачунара, не постоји континуални простор и он се мора на неки начин адекватно дискретизовати. Ти новонастали дискретизовани елементи су просторни елементи Јиове мреже. Уколико је дискретизација обављена исправно, применом апроксимација ће бити могуће направити реалан модел простора. Принцип је исти без обзира да ли је реч о две или три димензије.



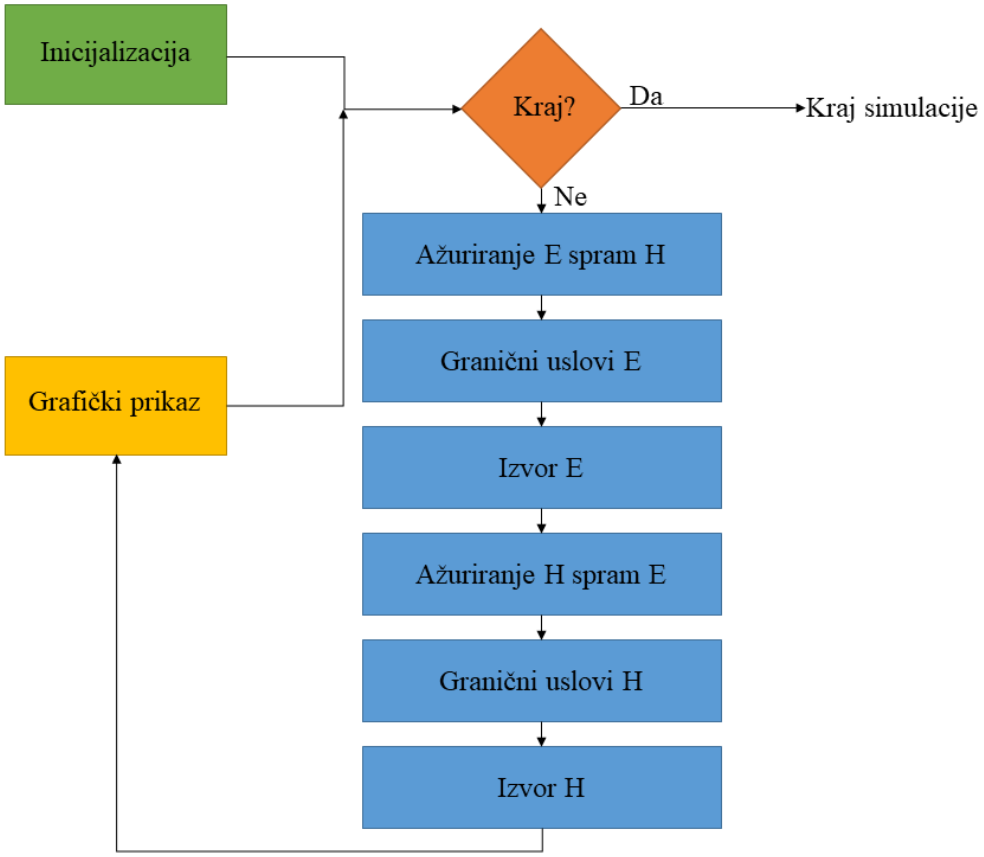
Слика 4 – Илустрација дискретизације дводимензионалног простора

Када је завршено рачунање, пре преласка у наредну итерацију је могуће забележити и приказати тренутно стање. Ово, наравно, значајно успорава само извршење програмског кода због додатних инструкција али је погодно у фазама док се алгоритам развија (као један од механизама провере) као и на крају када је потребно резултате и негде графички представити крајњем кориснику. У пракси се визуелизација типично не обавља после сваке итерације већ после неколико десетина, стотина или хиљада уколико се ради о великој симулацији.

Без додатне интервенције, пратећи Максвелове једначине и апроксимације алгоритма, таласи би се пропагирали до ивица Јионе мреже након чега би се одбиле од њих као од савршеног зида без икаквог губитка енергије. У реалном свету ово није случај, те се у близини ивица мреже уводи посебан слој чија је намена да у потпуности упије надолазеће таласне фронтове. Ово доста реалније моделује стваран свет где би таласи наставили да путују у бесконачност.

У пракси, ретко када је интересантно простирање таласа кроз условно речено празан простор. Најчешћи сценарији где се овакве методе користе јесте када хоћемо да откријемо унутрашњу структуру неког простора и направимо његов дводимензионални или тродимензионални модел (с тим да постоје и случајеви где се занима шта се дешава само и по једној димензији). Самим тим, у Јионој мрежи ће се пронаћи одређени објекти са својом површином/запремином и специфичностима материјала од кога су направљени. У зависности од особина тих материјала, таласи ће се у већој или мањој мери одбијати, упијати и преламати. За

ове комплексне интеракције је најбоље користити конститутивне једначине јер су све особине материјала садржане у параметрима **ε** и μ.



Слика 5 - Илустрација алгоритма

Узевши у обзир све до сада споменуто о FDTD алгоритму, могуће је извући и неке његове позитивне и негативне особине. Пре свега, погодан је за опис великих система и дугачких симулација. Разлог за то јесте што сам алгоритам скалира скоро линеарно са повећањем комплексности проблема док друге методе типично скалирају експоненцијално [5]. С обзиром да FDTD даје решења у временском домену, могуће је добити одзив система кроз широки спектар фреквенција унутар исте симулације. Такође, с обзиром да се особине материјала могу једноставније узети у обзир, FDTD се добро носи и са нелинеарним понашањима. Поред свега наведеног, алгоритам је већ одавно у зрелој фази и има доказаних примена и доступних решења на тржишту, тако да постоји и довољно доступне литературе и готових примера кода алгоритма [11].

FDTD наравно има и одређене мане које могу ограничити његову употребу. За почетак, сама чињеница да је мрежа четвртаста структура чини да је представа закривљених површина лоша односно доста апроксимирана. Веома резонантни објекти могу значајно продужити трајање симулације која иначе траје дуго и за мале проблеме. Свакако да је највеће ограничење употребе FDTD алгоритма чињеница да је то меморијски интензиван проблем. Највећи допринос овим проблемима јесте што је код реалних материјала њихова пермитивност функција фреквенције који захтева моделе као што су Лоренцов, Друдов или, у случају примена у медицини, Дебијев [50], [51]. Могуће је све то моделовати и у оквиру самог FDTD-а али то значајно компликује све једначине и није подржано у многим до сада објављеним имплементацијама.

3.2 Аритметика у рачунарству

Једна од главних намена рачунара произилази из њиховог имена а то је да рачунају ствари које су људима тешке или за које би људима требало јако пуно времена. Иако је током историје, а поготово у самом зачетку рачунарства, било различитих покушаја представе бројних система, на крају се као најефикаснији показао бинарни систем, пре свега због појаве транзистора који су лако и што је најбитније поуздано могли да представе два дискретна стања. Самим тим, као основа аритметике је послужила Булова алгебра чији закони важе над скупом од две вредности - логичке јединице и логичке нуле које се лако транслирају на електричне представе физичке јединице и нуле које карактеришу висок односно низак ниво напона.

Најједноставнија аритметичка операција са којом се сви први сусрећемо представља сабирање природних бројева. Постоје јасна правила како се цифре сабирају и како се резултат преноси даље на „више“ цифре. Сличан поступак важи и код рачунара уз једно велико ограничење, а то је да за разлику од теоријске математике где бројеви иду у бесконачност, у рачунарима имамо коначно много ресурса па самим тим и коначно велике бројеве које можемо представити. Стога имамо и дефинисан опсег над којим обављамо све операције. Једном када изађемо

изван опсега, резултат више није валидан и самим тим и неке законитости које постоје у теоријској математици престају да важе.

Што се тиче сабирања у Буловој алгебри, треба напоменути да сама операција сабирања заправо и није дефинисана директно јер је Булова алгебра првенствено применљива у области логике. Ипак, користећи логичке операције је могуће доћи до истог резултата као код сабирања. Описаћемо прво жељено понашање са једном табелом. У Табела 1 – Сабирање уз помоћ Булове алгебре, **A**, **B** и **C** представљају вредности на улазу – два сабирка и пренос са бита ниже тежине. **Sum** представља суму добијену сабирањем вредности са улаза док **Carry** представља пренос на бит веће тежине.

A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

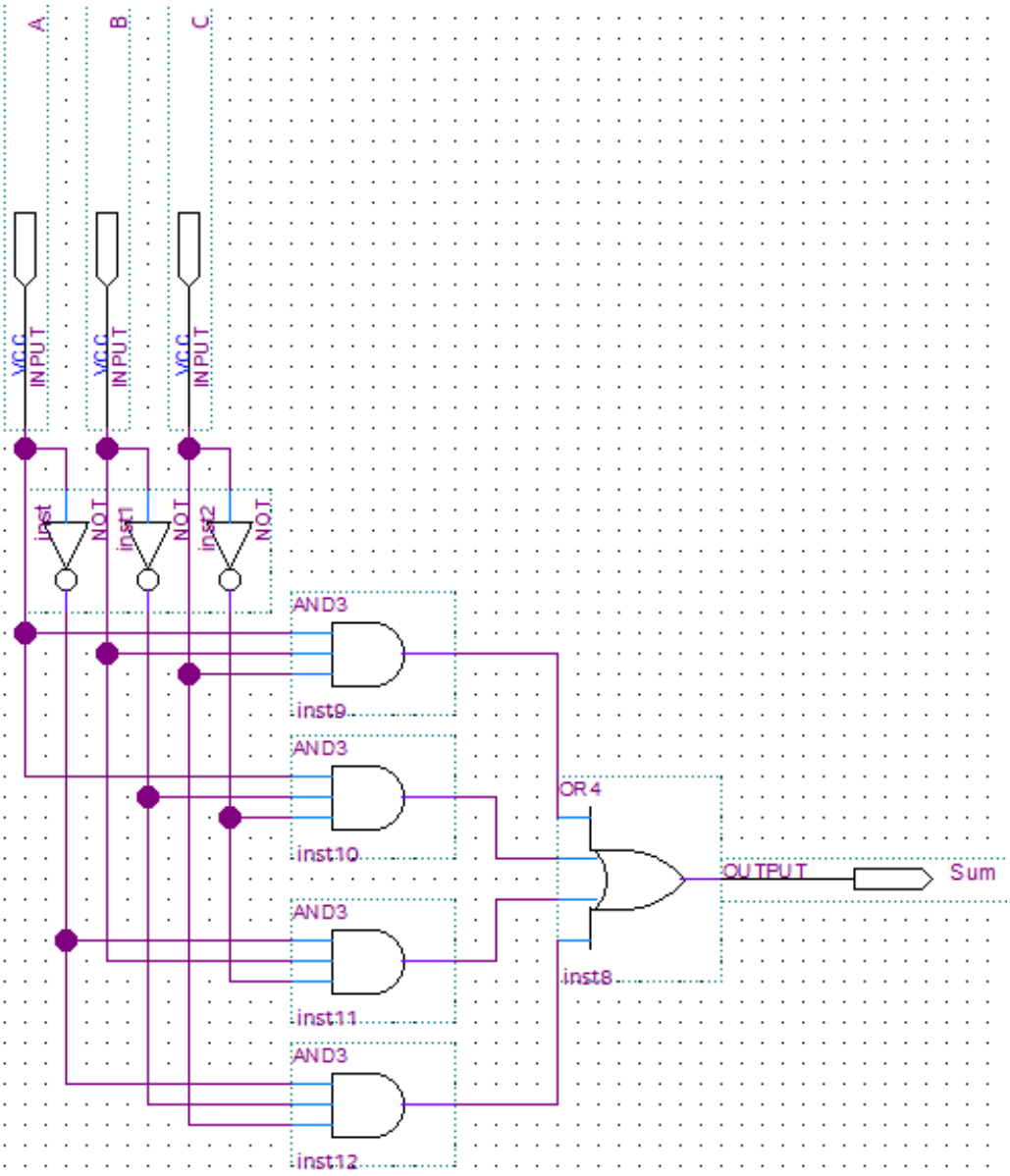
Табела 1 – Сабирање уз помоћ Булове алгебре

Над овом табелом је могуће применити законе Булове алгебре како бисмо пронашли минимални број једначина који је потребан да бисмо добили еквивалентно понашање. Користећи само операције конјункције, дисјункције и негације илустроване са следеће две једначине, можемо добити збир два бинарна броја са преносом.

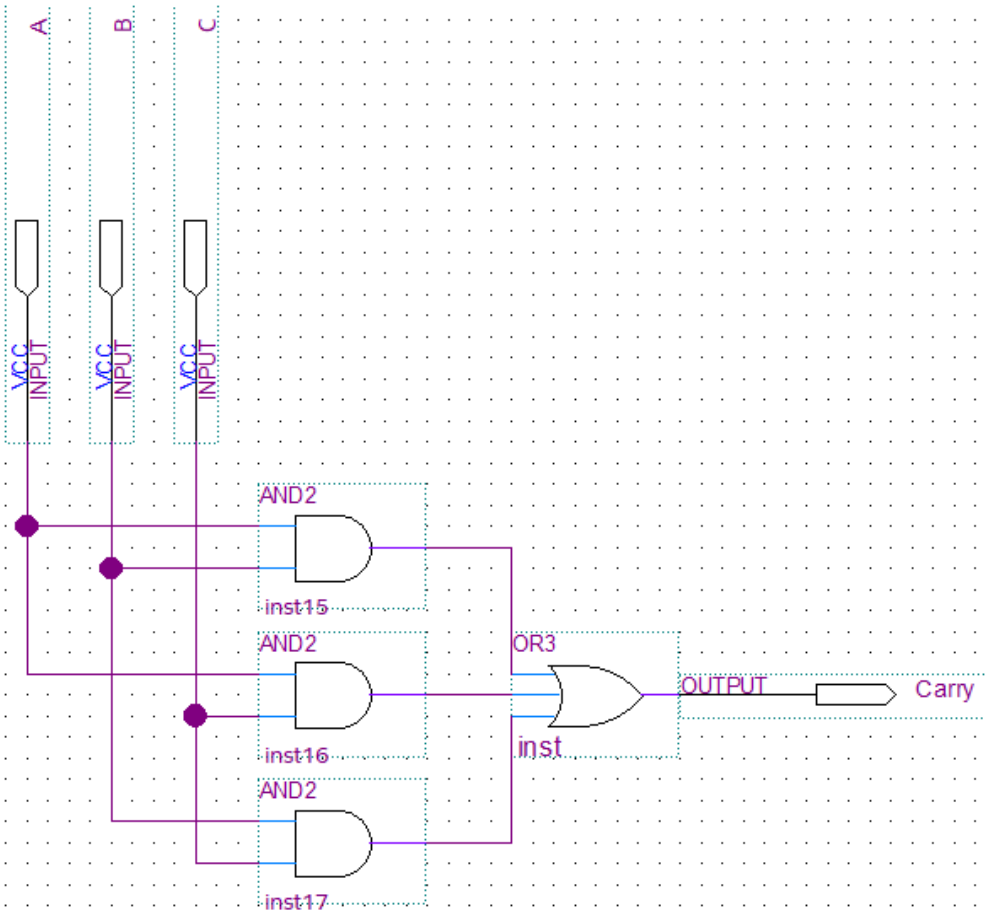
$$\text{Sum} = \overline{A}*\overline{B}*C + \overline{A}*B*\overline{C} + A*\overline{B}*\overline{C} + A*B*C$$

$$\text{Carry} = A*\overline{B}*\overline{C} + \overline{A}*B*\overline{C} + \overline{A}*\overline{B}*C + \overline{A}*\overline{B}*\overline{C} = A*B + A*C + B*C$$

Исте ове формуле је некада лакше представити уз помоћ еквивалентне електричне шеме користећи логичка кола која директно одговарају логичким операцијама Булове алгебре.



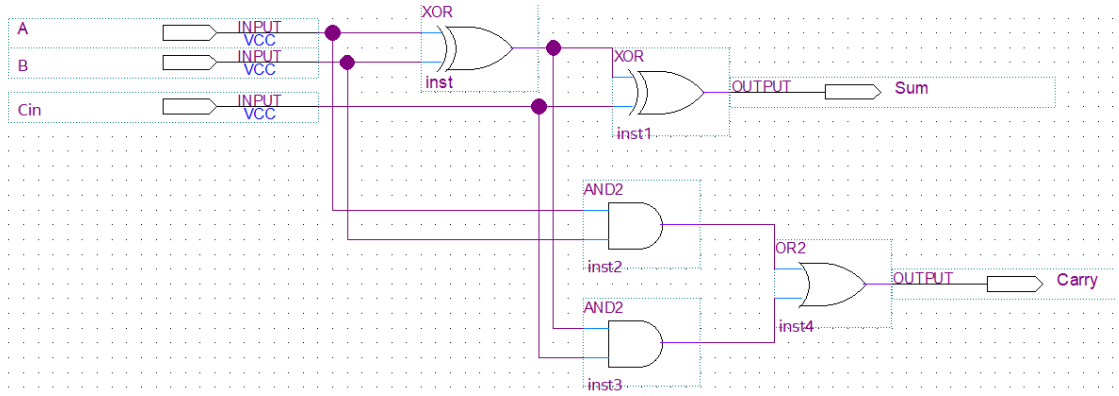
Слика 6 – Електрична шема пуног сабирача 1/2 (рачунање суме)



Слика 7 - Електрична шема пуног сабирача 2/2 (рачунање преноса)

У електричној шеми на сликама 6 и 7, струја путује у смеру од лева ка десно и од горе ка доле. Проласком кроз одговарајућа дигитална логичка кола, над физичким јединицама и нулама се примењују правила и закони Булове алгебре и на излазу добијамо одговарајуће вредности опет физичких нула или јединица.

Користећи још неке од закона и теорема Булове алгебре, могуће је конструисати еквивалентне једначине/електричне шеме које ће користити мањи број логичких операција/логичких кола заменом неких од основних дигиталних логичких кола другим. Пример сабирача које користи оваква кола је приказан на слици 8. У зависности од тога у којој технологији ће бити израђено коначно решење, некада је исплативије имати комплекснији систем који се састоји увек од истих кола него мањи број различитих врста дигиталних логичких кола.



Слика 8 – Електрична шема сабирача са ексклузивним или

Новина јесте што се овде појављује нова операција а то је ексклузивна дисјункција. Овакав компактнији запис односно електрична шема је погодан за боље људско разумевање и тумачење. Истовремено, захтева мањи број транзистора за реализацију у физичком свету па је с тог становишта ефикасније решење.

Прве компликације настају са увођењем негативних бројева у причу. У математици која се учи у школи се позивитни и негативни бројеви разликују са предзнаком (код позитивних је увек подразумеван плус, док се код негативних минус експлицитно наводи). Сличан приступ би се могао урадити и у имплементацији аритметике целих бројева тако што би један бит определили да буде бит знака, а остатак бита да представља апсолутну вредност броја. Оваква представа је позната под називом знак плус модуо. Иако је ова представа интуитивна људима чисто због изложености овог концепта кроз готово читаво школовање, уколико би пратили исти принцип као и код ручног сабирања целих бројева који су различитог предзнака, то би у многоме закомпликовало пројектовање аритметичко-логичких јединица рачунара. Разлог за то је што у случају различитих знакова за утврђивање решења је прво неопходно установити који је број по апсолутној вредности већи, затим преписати његов знак и потом урадити операцију одузимања која захтева потпуно нове Булове једначине. Поред свега наведеног, представа знак плус модуо уноси и проблем двоструке нуле, односно позитивне и негативне нуле (бројеви 1000 0000 и 0000 0000 су исти по апсолутној вредности али имају различит предзнак).

Међутим, инжењери су установили да са становишта рачунара постоји много једноставнији механизам који на први поглед није интуитиван људима. Како би се искористила аритметика која већ ради над скупом целих бројева уводи се појам комплемента двојке. Комплемент двојке је начин представе бинарних бројева такав да се и позитивни и негативни бројеви третирају као неозначени и самим тим све операције раде истоветно. Да би се направила комплемент два представа броја, прво се позитиван број од кога хоћемо да направимо негативан допуни са водећим нулама. Потом се над свим цифрама примени операција логичке негације, тако да се све нуле замене јединицама а све јединице нулама, и као последњи корак се сабира са још једном јединицом на позицији најнижег бита. Тако би нпр. број 2 са осам бита представили као 0000 0010. Уколико желимо да добијемо број -2, први корак јесте да применимо негацију што нам даје 1111 1101 и на крају сабирање са јединицом што као коначан резултат даје 1111 1110. [52]. Сабирањем бројева 3 и -2 бисмо очекивали да резултат буде 1. Уколико то урадимо над оваквом представом бројева имамо следеће понашање.

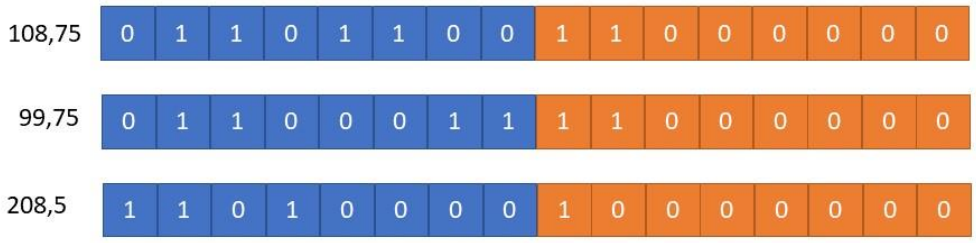
	0	0	0	0	0	0	1	1	(3)
	1	1	1	1	1	1	1	0	(-2)
[1]	0	0	0	0	0	0	0	1	(1)

Оно што је битно напоменути јесте да се приликом сабирања бројева у овом случају занемарује пренос на највишој позицији јер је он резултат другачије представе бројева а не „стварна“ апсолутна вредност другог операнда.

Следеће питање које се поставља је шта чинити када нам целобројна аритметика није довољно прецизна? Како представити реалне бројеве? Као одговор на то се јављају два приступа - један који је оптимизован по питању брзине, и други који омогућава већу прецизност по цену сложенијег математичког апарата.

Једноставнији од ова два приступа јесте представа реалних бројева уз помоћ непокретног зареза. У том случају, аритметика ради готово непромењено у односу на целе бројеве. Децимална тачка се поставља увек на истом месту и одговарајући бити се сабирају са кореспондентним битима са друге стране. Када збир фракција

пређе један, преноси се у цели део резултата, као што би био поступак и са децималним бројевима.



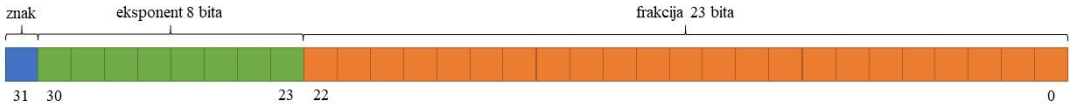
Слика 9 – Приказ сабирања бројева са непокретним зарезом

Као што у целом делу броја, сваки бит представља неки од степенова двојке, исто је и са фракцијом, само што представља 2^{-N} где N зависи колико смо се места померили десно од децималне тачке.

Оваква представа се често среће код наменских рачунара и микроконтролера. Један од типичних примера за то су и DSP процесори. Предности овакве представе јесу пре свега брзина извршавања операција и једноставност - принцип остаје апсолутно исти као и код сабирања целих бројева. Такође, једна особина коју не треба занемарити јесте униформна расподела вредности на бројевној оси. Другим речима растојање између две суседне вредности које се могу представити је исто над целим опсегом. Главна мана овакве представе бројева јесте што је динамички опсег мали у односу на други приступ и уколико се зарез не налази на добром месту, лако може доћи до прекорачења опсега или до неодговарајуће прецизности и самим тим до невалидних односно неповољних резултата.

Алтернатива оваквој представи је такозвани плутајући одосно покретни зарез (*енг*. floating point). Као што му име каже, код њега не постоји унапред предефинисано место где се увек налази граница између целог дела броја и фракције. Уместо тога, зарез се помера спрам потреба тако да што више корисних информација буде представљено а да се при том избегне непотребно понављање нула. Ако погледамо број 0,000000127 користи се 6 нула за одређивање положаја значајних цифара иза децималног зареза. Ефикаснији запис би био нешто попут $1{,}27 * 10^{-7}$. У оваквом запису 1,27 представља значајни део броја или мантису, 10

је база док је -7 експонент [52]. У случају бинарних бројева, база је увек једнака 2, а величине експонената и фракција зависе од тога да ли се ради о једнострукој или двострукој прецизности.



Слика 10 – Представа бројева у покретном зарезу у једнострукој прецизности

Покретни зарез по ревидираном IEEE стандарду [53] има 32 бита за једноструку прецизност, са распоредом бита као на Слика 10 – Представа бројева у покретном зарезу. Главна предност овакве представе јесте знатно већи динамички опсег него што би то било могуће постићи са непокретним зарезом исте битске ширине. То важи са обе стране спектра - могуће је представити и много мање и много веће вредности него што је то могуће са непокретним зарезом. Међутим, то има цену и то не занемарљиву у виду комплексности аритметичко-логичке јединице и времена извршавања операција. Када имамо посла са сабирањем два реална броја код којих се зарез не налази на истој позицији, потребно је један од два операнда „померити“ тако да им се зарези поравнају и тек онда приступити са операцијом сабирања кореспондентних цифара. Овај процес се зове нормализација. Поређење је ипак нешто лакше јер је често довољно упоредити само експоненте.

Нормализација уопште није тривијалан поступак и врло често постоје посебни наменски копроцесори чији је једини задатак да обављају аритметику са покретним зарезом. Такође, за разлику од представе са непокретним зарезом, код покретног зареза имамо нагомилавање вредности блиских нули, док како идемо ка све већим бројевима, имамо све мању резолуцију. Последица овакве расподеле вредности јесте да када сабирамо јако велике и јако мале вредности, резултат неће моћи да стане у опсег бита који је дефинисан архитектуром и зато ће се неке вредности напросто одбацити. Самим тим, сабирање престаје да буде у потпуности асоцијативна операција. То можемо илустровати примером низа са великим бројем елемената чији је сваки парни елемент изузетно велики број а сваки непарни изузетно мали (иако је упитно колико је овакав случај реално

сусрести у пракси). Приликом сабирања суседних бројева, изузетно мали бројеви ће бити занемарени јер неће моћи бити представљени ваљано ако између значајних цифара имамо пуно нула. Али уколико тај низ прво сортирамо у растућем редоследу и онда сабирамо најпре мале бројеве једне са другима, довољно малих бројева када се сабере ће дати вредност која се неће у потпуности одбацити приликом сабирања са већим бројем.

Углавном је спрам типа проблема који се решава јасно која од две представе је боље решење – када је потребна брзина и мали утрошак ресурса, користи се фиксни зарез, док када нам је потребна велика прецизност и динамички опсег је природније користити представу са покретним зарезом. Некад просто не постоји избор због ограничења платформи са којима се ради уколико подржавају само једну представу.

3.3 Блоковски покретни зарез

Остаје нерешено питање, шта чинити у ситуацијама када нам је потребна и брзина али и већи динамички опсег? Да ли је могуће сачувати брзину блиску непокретном зарезу а при том имати повећан динамички опсег покретног зареза? Одговор на ово питање је потврдан и крије се у блоковском покретном зарезу.

Блоковски покретан зарез (BFP) није нов концепт, мада је његова примена ограничена на неколико алгоритама, од којих је типичан пример алгоритам брзе Фуријеове трансформације [36], [37]. Он представља један вид емулације покретног зареза на архитектури која користи непокретни зарез. За разлику од покретног зареза, уместо да је сваки елемент представљен са својом мантисом и експонентом, код BFP се сви елементи једног блока скалирају са заједничким експонентом који се одређује спрам највећег елемента.

BFP можемо описати на следећи начин – скуп бројева у покретном зарезу ћемо означити са **X**, где $\boldsymbol{x_i}$ представља *i*-ти елемент скупа **X**, док $\boldsymbol{m_i}$ и $\boldsymbol{e_i}$ представљају одговарајуће мантисе и експоненте за *i*-ти елемент. Математички запис би био следећи:

$$\mathbf{X} = (\boldsymbol{x_1}, \boldsymbol{x_2}, \ \dots \ , \boldsymbol{x_i}, \ \dots \ , \boldsymbol{x_N}) =$$

$(\boldsymbol{m_1} \times \boldsymbol{2^{e_1}}, \boldsymbol{m_2} \times \boldsymbol{2^{e_2}}, \ldots, \boldsymbol{m_i} \times \boldsymbol{2^{e_i}}, \ldots, \boldsymbol{m_N} \times \boldsymbol{2^{e_N}})$

Скуп **X'** је BFP репрезентација скупа **X**, док су M'_X и E'_X скупови који садрже мантисе и експоненте придружене скупу **X'**. Математички запис би био следећи:

$X' = (x'_1, x'_2, \ldots, x'_i, \ldots, x'_N) = M'_X \times 2^{E'_X}$, такви где важи:

$M'_X = (m'_1, m'_2, \ldots, m'_i, \ldots, m'_N)$ и

$E'_X = \max(e_i)$, где **i** узима вредности између 1 и N.

Начин на који се m'_i формира је следећи:

$m'_i = m_i >> (E'_X - e_i)$. Оператор a >> b значи померање броја a, b пута у десно [28].

Да бисмо илустровали мало боље како ради овај формат, погледаћемо пример како би се један број могао представити у покретном зарезу, а затим видети како можемо уопштити случај за блоковски покретан зарез. Узмимо случај где имамо формат од 16 бита (1 за знак, 5 за експонент и 10 за мантису). Да бисмо децимални број 10.75 представили у покретном зарезу, први корак јесте раставити број на цео део (10) и на фракцију (0.75). Број 10 у бинарном запису износи 1010. Аналогно томе, фракцију такође записујемо у степенима двојке где нам свака следећа цифра десно од децималне тачке представља 2 на минус наредни степен. Тако да бинарна вредност 0.1 у децималном запису износи 2^{-1} односно 0.5, а 0.01 износи 2^{-2} односно 0.25. Самим тим, фракција 0.75 би у бинарном запису износила 0.11.

У принципу, слично као у запису са експонентом у децималном систему, желимо да имамо само једну „корисну“ вредност лево од децималне тачке, а све остале са десне стране. Овој новоформираној вредности придодајемо множење са одговарајућим експонентом који је једнак бази на одговарајућ степен који је једнак броју цифара за колико смо померили децималну тачку. Следи и математички запис.

$$
\begin{aligned}
(10.75)_{10} &= (1010.11)_2 \\
&= (1.0101100000)_2 * 2^3 \\
&= (1.0101100000)_2 * 2^{(00011)_2} \\
&= 0.00011.0101100000
\end{aligned}
$$

На крајњем запису, плава 0 представља позитиван број, зелени 00011 представљају множење са 2^3 а преостале наранџасте вредности мантису.

Замислимо да треба да представимо бројеве 7.25, 8.5, 10.75 и 16.8. Фракције за прва три броја су погодне за представу у бинарном систему и износе 0.01, 0.1, 0.11 респективно. Фракција 0.8 је незгодна за бинарну представу и износи 0.110011001100...Дакле, вредност 0.8 је немогуће представити прецизно као што бисмо то могли урадити у децималном систему те ће бити заокружена на најближу могућу, слично као што је немогуће у децималном запису представити количник 1/3 са коначним бројем цифара. Примењујући исти алгоритам као и у претходном кораку, добијамо да би запис ових вредности у бинарном запису износио:

$(7.25)_{10}$ = 0.00010.1101000000

$(8.5)_{10}$ = 0.00011.0001000000

$(10.75)_{10}$ = 0.00011.0101100000

$(16.8)_{10}$ = 0.00100.0000110011

У пракси, запис који је дефинисан стандардом IEEE уводи додатне претпоставке (као што су подразумеване вредности појединих цифара и мало другачији распоред ради оптимизација) тако да би ове вредности биле мало другачије. Разлог за то је такозвана пристрасна (*енг*. biased) представа која додаје константу 127 на експонент (односно вредност 0 је представљена као број 127, и све остале вредности су онда померене такође). У суштини фракција је иста, али су бити експонента другачије распоређени.

$(7.25)_{10}$ = 0.10001.1101000000 = 1 * 2^2 * 1.05

$(8.5)_{10}$ = 0.10010.0001000000 = 1 * 2^3 * 1.032

$(10.75)_{10}$ = 0.10010.0101100000 = 1 * 2^3 * 1.344

$(16.8)_{10}$ = 0.10011.0000110011 = 1 * 2^4 * 1.027

Корак који фали да имамо блоковски покретан зарез јесте да нормализујемо све ове вредности тако да имају заједнички експонент, и то типично онај са највећом вредношћу (2^4). То значи да би број 16.8 као такав остао непромењен, а преостале мантисе би се трансформисале по формули $\boldsymbol{m'_i = m_i >> (E'_X - e_i)}$. Тако да би се мантиса од 7.25 померила за 2 места у десно, док би се мантисе од 8.5 и 10.75 помериле за једно место.

Међутим, блоковски покретни зарез је емулација покретног зареза на архитектури која подржава само непокретни зарез. Те прво морамо знати на ком месту се налази децимална тачка и онда у складу са тим распоредити бите. Узмимо илустрације ради да је то на средини, после осмог бита. Тада ће редослед бита пре увођења заједничког експонента бити организован као на слици 10 и за ове вредности ће изгледати овако:

$(7.25)_{10}$ = 0000111.01000000

$(8.5)_{10}$ = 0001000.10000000

$(10.75)_{10}$ = 0001010.11000000

$(16.8)_{10}$ = 0010000.11001100

На крају, потребно је још све вредности померити у десно у зависности од разлике експонената, те би крајњи изглед изгледао овако:

$(7.25)'_{10}$ = 000.01110100~~0000~~

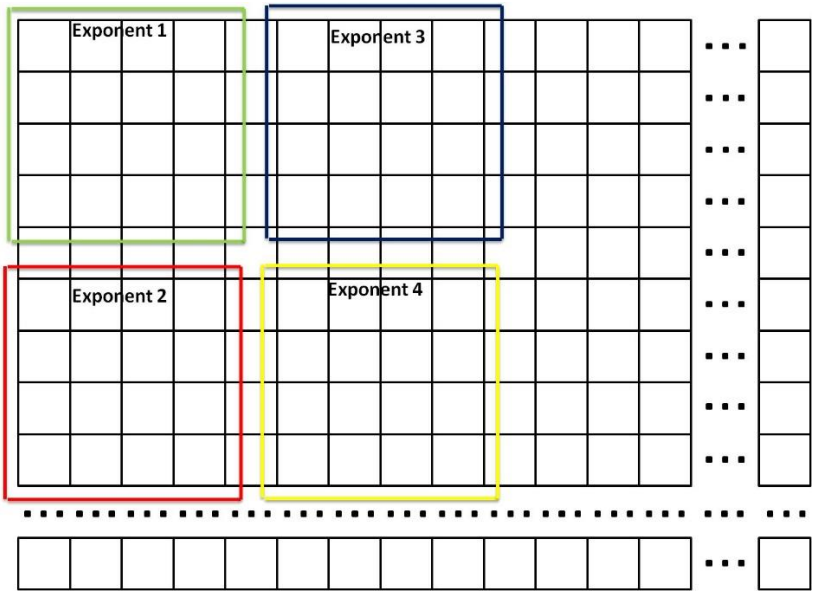
$(8.5)'_{10}$ = 000.10001000~~0000~~

$(10.75)'_{10}$ = 000.10101100~~0000~~

$(16.8)'_{10}$ = 001.00001100~~1100~~

Из овог примера видимо да бисмо неке вредности изгубили и зато је веома важно определити довољан број бита за фракције (већи део) како би се што мање губило на прецизности. Опет, с друге стране потребно је оставити довољно бита лево од децималне тачке како приликом рачунања не би дошло до испадања изван опсега. Зато се типично пре почетка рачуна све вредности скалирају са одговарајућим константама како би се уклопиле у опсег [-1, 1].

Приликом повратка тј. ишчитавања ових вредности из меморије је потребно урадити математички инверзну операцију, односно померање бита у лево за онолико бита колико стоји у експоненту. Уколико су вредности елемената које се скалирају истим експонентом довољно блиске, BFP представља погодан формат записа који омогућује агресивније скалирање и самим тим и већи динамички опсег у односу на непокретни зарез. Истовремено, захтева и мање операција нормализације у односу на покретни зарез, тако да представља фин компромис између ове две крајности.



Слика 11 – Графичка представа BFP

Наравно, BFP није без мана и самим тим што је резултат компромиса, значи да је применљив на доста ужем скупу проблема. С обзиром на то да суседне вредности деле исти експонент, уколико се вредности међу њима драстично разликују, то ће веома да угрози прецизност и тачност решења. Разлог за то јесте што када се експонент формира, све вредности се скалирају спрам најекстремније у датом скупу. Самим тим, проблем код кога се користи треба бити такве природе да су суседне вредности блиске. Како се у FDTD-у моделује кретање таласних фронтова, за очекивати је да ће суседне вредности бити сличне и тако је и настала идеја за његову примену. Пре радова [34], [54] није уочено да је овај приступ коришћен раније и то представља један од главних доприноса саме дисертације, заједно са уштедом меморије која ће бити објашњена у поглављима која следе.

Предности коришћења BFP у овом истраживању има неколико. Првенствено се иста количина података може представити са мање бита уз минимално жртвовање прецизности. Мањи утрошак меморије резултује и са мањим протоком података који је кључан за убрзавање обраде у FDTD алгоритму с обзиром на то да је у питању меморијски интензиван проблем. Друго, операције множења и акумулирања резултата (*енг*. Multiply And Accumulate - MAC) су веома брзе, ефикасне и добро прилагођене аритметичко логичким јединицама које се могу

наћи у типичним наменским (ASIC) и FPGA чиповима у поређењу са оним који користе покретни зарез [39].

ПОГЛАВЉЕ 4.

ТЕОРИЈСКО РЕШЕЊЕ И ИМПЛЕМЕНТАЦИЈА

Како би проверили хипотезу да употреба BFP-а може позитивно утицати на перформансе FDTD алгоритма, прво је било потребно имплементирати основну верзију. Програмски језик који је изабран за ову сврху је Julia [55]. Julia је језик дизајниран за алгоритме високих перформанси пре свега у области науке и као такав је већ коришћен у бројним истраживањима [56], [57], [58]. Циљ језика је да представи оквир за целокупан развој који има релативну благу криву учења у контексту једноставности употребе а истовремено задовољавајућу брзину извршавања. Самим тим, аутори тврде да за коначну примену није неопходно цео код преписати у неком другом ефикаснијем језику као што је Це или Це++ (C/C++). У духу истраживања, и Julia представља компромисно решење између MATLAB-а као традиционалног језика за научна истраживања и Python-а као модерног скрипт језика предвиђеног за брз развој апликација. Поред свих својих погодности, Julia нуди и могућност дефинисања сопствених типова што је веома погодно за потребе BFP, иако детаљи импелементације на машинском нивоу нису познати.

Првобитна верзија програма је за све своје потребе користила покретни зарез у једнострукој прецизности (32 бита). Главни програм започиње као што је илустровано на Слика 5 - Илустрација алгоритма. Почетни корак јесте дефинисање свих неопходних константи (брзина светлости, пермитивност и

пермбеабилност вакума, фреквенција одабирања, величина корака и друге). Потом се креира условно речено „празан“ простор у виду матрица величине 64*64, у коме ће се простирати таласи, и попуни са почетним вредностима (нула) за све величине које се користе у једначинама. Величине које се мере су оне које се појављују у једначинама – јачина електричног поља, јачина магнетног поља поља, електрична и магнетна индукција. Неке величине имају компоненте у све 3 осе, а неке само у 2 осе (оне које се губе преласком из тродимензионалног у дводимензионални проблем). У овом кораку се убацује и поменути посебан слој [3] дебљине неколико ћелија који изузетно брзо упија таласе, како се не би одбијали од зидова матрице. Као последњи корак иницијализације остаје дефинисање извора електромагнетног поља.

Једном када су сви почетни услови подешени, алгоритам може да крене са рачунањем у петљи унапред дефинисан број итерација. У првој итерацији ажурирање електричног поља спрам вредности магнетног неће имати никаквог ефетка јер је магнетно поље у том тренутку непостојеће. Иста ситуација је и са граничним условима електричног поља као резултат претходног корака. Затим се јавља електрични импулс који емитује извор електричног поља и талас креће да се шири у концентричним круговима ка ивицама матрице. Наравно, чим се појави електрични импулс тј. промени вредност електричног поља, то значи да ће се индуковати одговарајуће магнетно поље и самим тим ће се преостала 3 корака одвити према дефинисаним једначинама.

На крају прве итерације се мере вредности од интереса (у нашем случају јачина електричног поља у правцу Z осе) и потом се, у зависности од параметра колико често то желимо, прави одговарајућа графичка представа тренутног стања или не. Потом креће нови циклус и петља се понавља све док се не достигне крајњи корак – унапред дефинисани број итерација.

Резултати ове верзије су узети као референтна тачка односно такозвани златни вектор спрам које су све наредне еволуције упоређиване. Када је установљено да решење базирано искључиво на FPGA чиповима неће бити у стању да се пореди са постојећим решењима по питању перформанси [34], следећа акција је била пребацити сав рачун у непокретни зарез.

Први корак је био све вредности сместити у опсег [-1, 1] скалирањем са одговарајућим константама. Оваква смена са покретног на фиксни зарез није драматично утицала ни на брзину извршавања нити на прецизност, што је било и за очекивати. Следећи корак је био одредити колико је минимално бита неопходно за рад алгоритма. Експериментално је утврђено да је теоретски минимум био 14 бита за дату симулацију, али то је за последицу имало драстичан губитак прецизности, као и проблем да иако су крајњи резултати могли да стану у дате вредности, извесни међурезултати су били проблематични. На крају је узета компромисна вредност од 24 бита која је у старту представљала уштеду од 25% меморијског протока уз губитак прецизности од око 5%.

Питање које се поставља након овог корака јесте да ли је могуће смањити грешку испод прага од 5%? Приде, да ли је могуће направити још већу компресију података и самим тим направити још већу уштеду? Уколико би 24 бита у непокретном зарезу искористили за емулацију покретног зареза, начелно би то значило већи динамички опсег, самим тим и мању грешку.

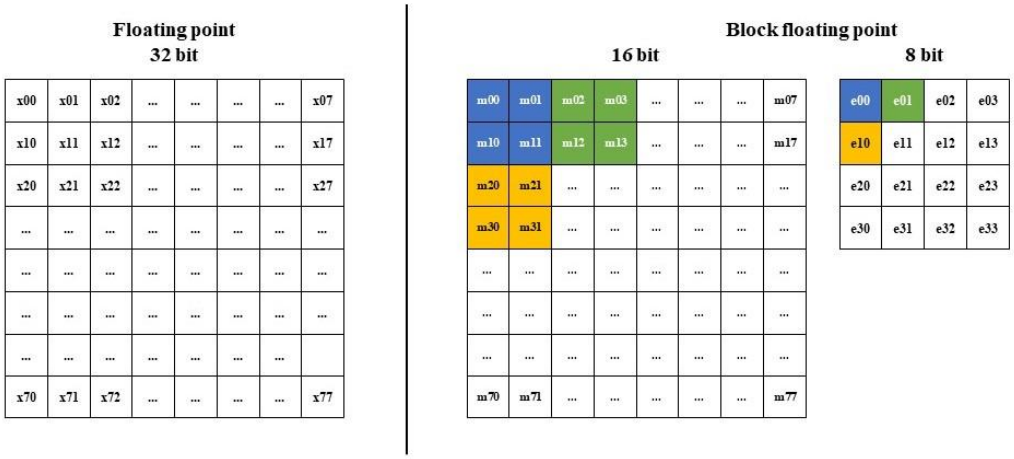
4.1 Одступања од досадашњих решења

Следећи корак је био направити сопствени формат за BFP. То је у суштини опет један оптимизациони проблем где је потребно пронаћи најбољи однос уштеде меморије и што мањег губитка прецизности. Сви параметри полазе директно од величине самих блокова. Што је блок већи, тиме је и уштеда меморије већа, али то лошије утиче на прецизност. С друге стране што су блокови мањи, то је ситуација ближа почетном решењу које користи покретни зарез где се у најекстремнијем случају за величину блока 1 они поклапају.

Свакако да је највећи изазов руковање подацима. За разлику од ранијих верзија апликације када су се у једној матрици налазили сви елементи, сада је за њихово складиштење било потребно имати две матрице. Илустроваћемо то на примеру Јиове мреже димензија 8*8.

У случају првобитног решења, постојала је једна структура чији је сваки елемент био величине 32-бита (представа у покретном зарезу једноструке прецизности) и представљао је јачину електричног поља у правцу Z осе у

одговарајућем делу простора. Сва рачунања су полазила из ове структуре и након рачунања су сви резултати били похрањени у овој структури, како би се могли искористити у наредној итерацији.



Слика 12 – Илустрација представе матрице уз помоћ блоковског покретног зареза

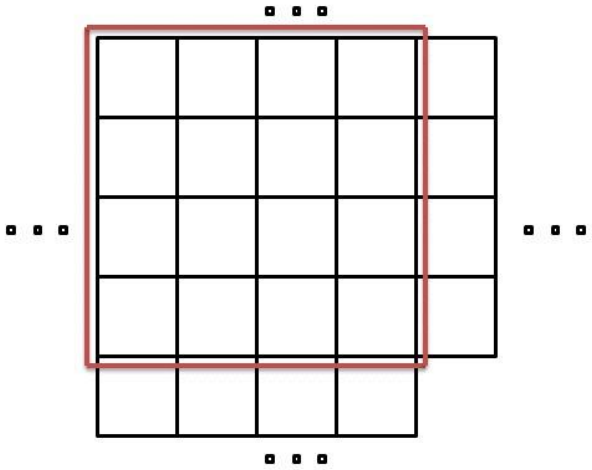
Преласком на BFP, та иста Јиова мрежа би сада била подељена на две структуре – две матрице. Прва би била такође димензија 8*8 али овог пута би сваки елемент био величине свега 16 бита. Ових 16 бита представљају значајне цифре односно фракцију (као што би било и у покретном зарезу). У зависности од величине блокова, разликује и величина друге структуре у којој се налазе 8-битни експоненти. Ако посматрамо случај величине блока 2*2, то значи да би цела иницијална структура могла да се поплоча са 4*4 тј. 16 блокова, те знамо колика ће бити величина матрице са експонентима. Већ у овом стадијуму видимо колика је потенцијална уштеда по питању потрошње меморије – готово да је преполовљена. Ипак, оваква уштеда долази са ценом нешто компликованијег руковања самом Јиовом мрежом, пре свега са стране броја операција. Ова последица је прихватљива јер количина рачунарских операција није проблем са FDTD-ом него број приступа меморије.

За пример у овој тези је узета баш оваква подела где су вредности у првој матрици 16-битне док су експонентни 8-битни како би радили са целим бројем бајта. Коначна потрошња и прецизност су описани у поглављу са резултатима.

Што се тиче измена у самом алгоритму, њих има неколико. За почетак, прво су све матрице које су претходно садржале 32-битне вредности поља замењене са

2 матрице – једна димензија 64*64 која сада садржи 16-битне мантисе и друга чија величина зависи од величине тј. броја блокова. Ако узмемо величину блокова димензија 4*4, то би значило да имамо 256 таквих блокова и наша матрица експонената ће бити димензија 16*16. Наравно, пошто радимо са непокретним зарезом, не смемо изоставити, као и код случаја са обичним непокретним зарезом, скалирање свих вредности како би стале у опсег [-1, 1].

За исправно функционисање алгоритма је било неопходно увести додатне структуре података уз сваки блок. С обзиром на имплементацију, приликом освежавања вредности било које ћелије су потребне и вредности суседних ћелија (конкретно оних суседа који се налазе десно и испод). Преласком на BFP, вредности суседа на границама блокова нису познате јер су оне скалиране са другим експонентом. Због тога су уведене додатне структуре које представљају једнодимензионалне низове чија димензија одговара димензији самог блока што је илустровано на Слика 13 – Додатне структуре за потребе BFP-а.



Слика 13 – Додатне структуре за потребе BFP-а

Уз ове додатне структуре убачене су и додатне функције које су служиле за процесе отпакивања података, како би сва рачунања и даље радила исто (да не би утицали на време извршавања у односу на референтно решење), и процес паковања у тренутку када су све операције извршене. Приликом паковања долази

и до рачунања новог експонента за сваки од блокова. Поједностављен псеудо-код који следи у прилогу објашњава навигацију кроз ове нове структуре.

```
...
for tj in 1:num_of_tiles_y_axis
    for ti in 1: num_of_tiles_x_axis
        # unpacking
        for jj in 1:tile_size
            for ii in 1:tile_size
                j = ((tj - 1) << tile_shift_x) + jj
                i = ((ti - 1) << tile_shift _y) + ii
                X_tile[ii,jj] = from_block(X_bf[i,j], block_exp)
            end
        end
        do_the_calculations() #calculations
        #packing
        for jj in 1: tile_size
            for ii in 1: tile_size
                j = ((tj - 1) << tile_shift _x) + jj
                i = ((ti - 1) << tile_shift _y) + ii
                X_bf[i,j] = to_block(X_tile[ii,jj], block_exp)
            end
        end
    end
end
...
```

У представљеном алгоритму **ti**, **tj**, **ii** и **ij** су бројачи који се користе за навигацију. Бројачи у спољашњим петљама (**ti** и **tj**) се крећу кроз целу мрежу док се унутрашње петље (**ii** и **ij**) крећу кроз појединачне блокове. Вредности **i** и **j** се користе за индексирање читаве мреже. **X_tile** је привремена помоћна матрица у

коју ће бити учитани сви неопходни подаци, док је **X_bf** матрица на којој се тренутно ради. **Block_exp** је нови експонент који одговара тренутно обрађиваној матрици. Функције **to_block** и **from_block** раде на исти начин као што је описано и у поглављу 3.3 – на основу израчунатог експонента скалирају све елементе померајући их у одговарајућу страну за вредност експонента. Рачунање експонента обавља помоћна функција **calculate_block_exponent** која се позива непосредно пре угњеждене for петље и она проналази највећи елемент по апсолутној вредности (онај који ће имати највећи експонент), те његов експонент проглашава за онај с којим ће сви остали елементи тог блока бити скалирани. Због саме природе таласа је за очекивати да ће вредности суседних ћелија бити сличне, и самим тим грешка услед скалирања не би требала да буде велика.

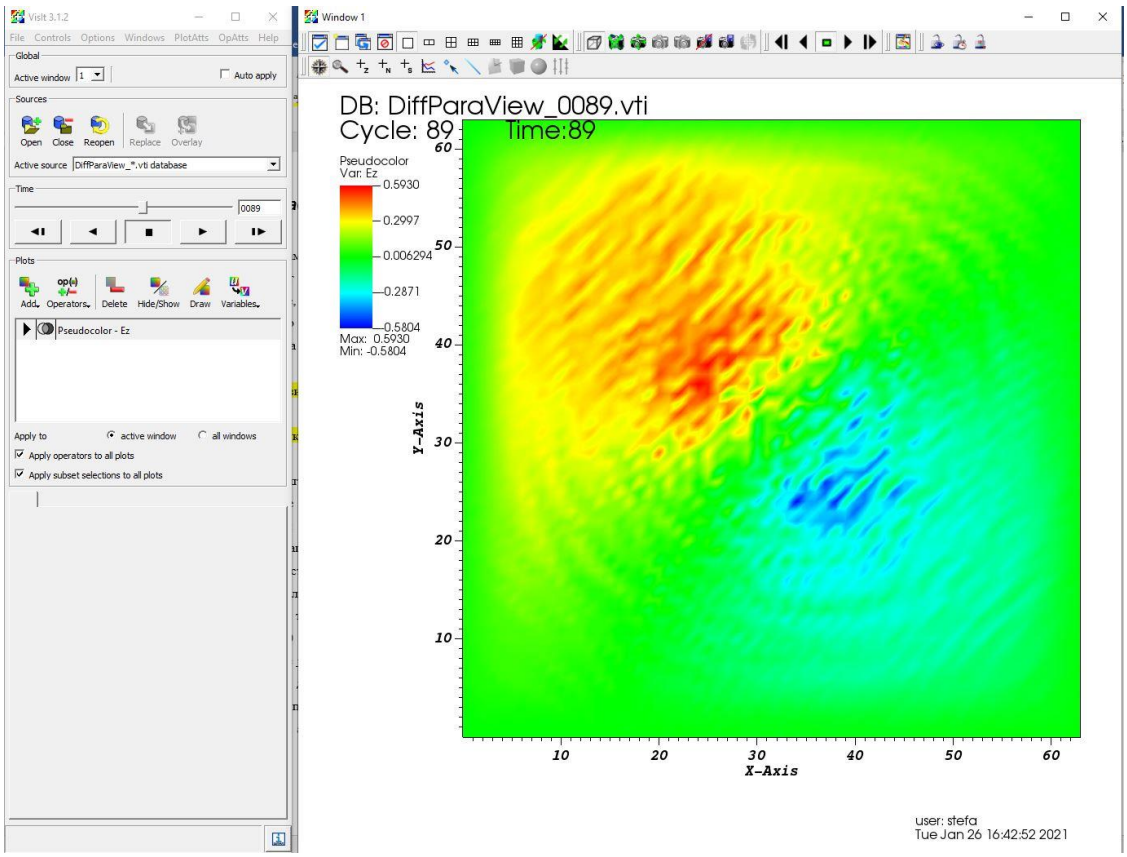
Неким једначинама су потребне вредности елемената који још увек нису израчунати у оваквој представи Јиове мреже и због тога се укључују додатне структуре са Слика 13 – Додатне структуре за потребе BFP-а. Посебни гранични услови се примењују за елементе који би се нашли ван мреже – њихове вредности су увек постављене на 0.

Једна од погодности јесте што приликом оваквог рада алгоритма долази до појаве просторне локалности. Следећи блок који ће бити обрађиван се налази негде близу у меморији. Овако предвидиво понашање омогућава имплементацију и додатних техника за оптимизацију као што су прибављање елемената унапред (енгл. *pre-fetching*), проточна структура (енгл. *pipeline*) и читање у већим количинама (енгл. *burst memory reading*).

Пре паковања по жељи је могуће визуелизовати тренутно стање. За ову сврху је коришћена библиотека WriteVTK. Она омогућава да се у свакој итерацији FDTD алгоритма релевантне вредности сачувају као структура података која се касније може исцртати и приказати кориснику путем одговарајућих алата за визуелизацију. Свака структура представља један исечак симулације који је забалежен у виду слике. Слагањем ових слика у филм добијамо врло интуитивну симулацију која визуализује простирање таласа у жељеном простору. У пракси се за велике симулације не визуализује након сваке итерације већ након сваке десете, стоте или чак и неке веће вредности. Главни разлог је свакако што овај процес додатно успорава овај већ спор и интензиван алгоритам.

4.2 Графички приказ симулације

За саму представу је коришћен алат по имену VisIt [59]. VisIt (skraćeno od Visualize It) је алат отвореног кода развијен у америчкој националној лабораторији Lawrence Livermore, чија је једна од намена графичка представа и анализа података независно од њихове величине. Користи се свакодневно у многим научним областима почевши од нуклеарне физике до медицине.



Слика 14 – Изглед VisIt окружења

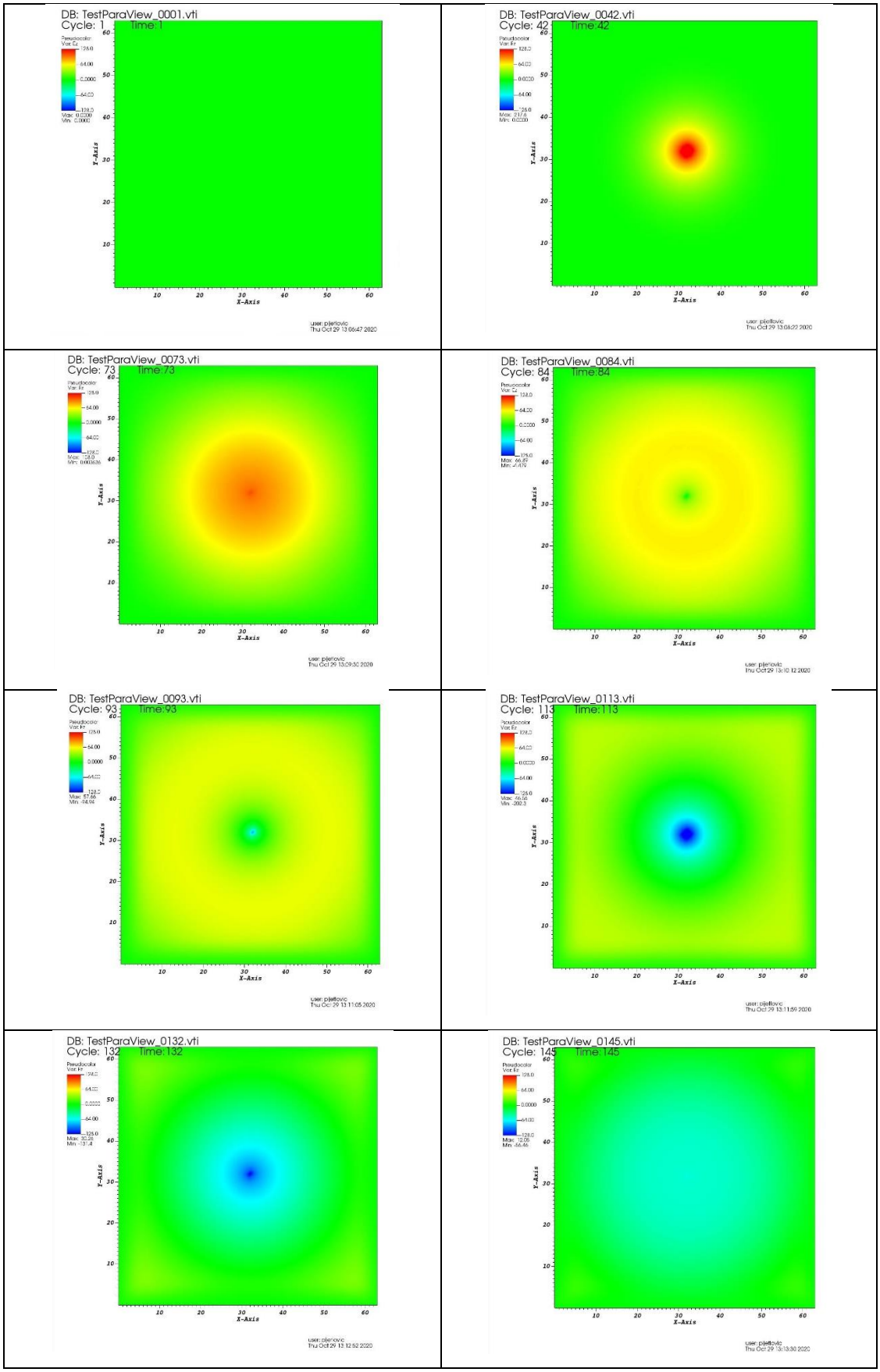
VisIt се састоји из два прозора који се приказују кориснику одмах при покретању алата. Са леве стране је главни прозор који служи за навигацију до базе података, за временску контролу симулација (покретање, заустављање, пуштање уназад) као и за одабир различитих начина представе података (дводимензионални, тродимензионални, са или без мреже као и многи други). Са десне стране је прозор за саму графичку представу. За потребе ове симулације је

коришћено подешавање *pseudocolor* које даје спектар боја које одговарају различитим вредностима величине која се представља.

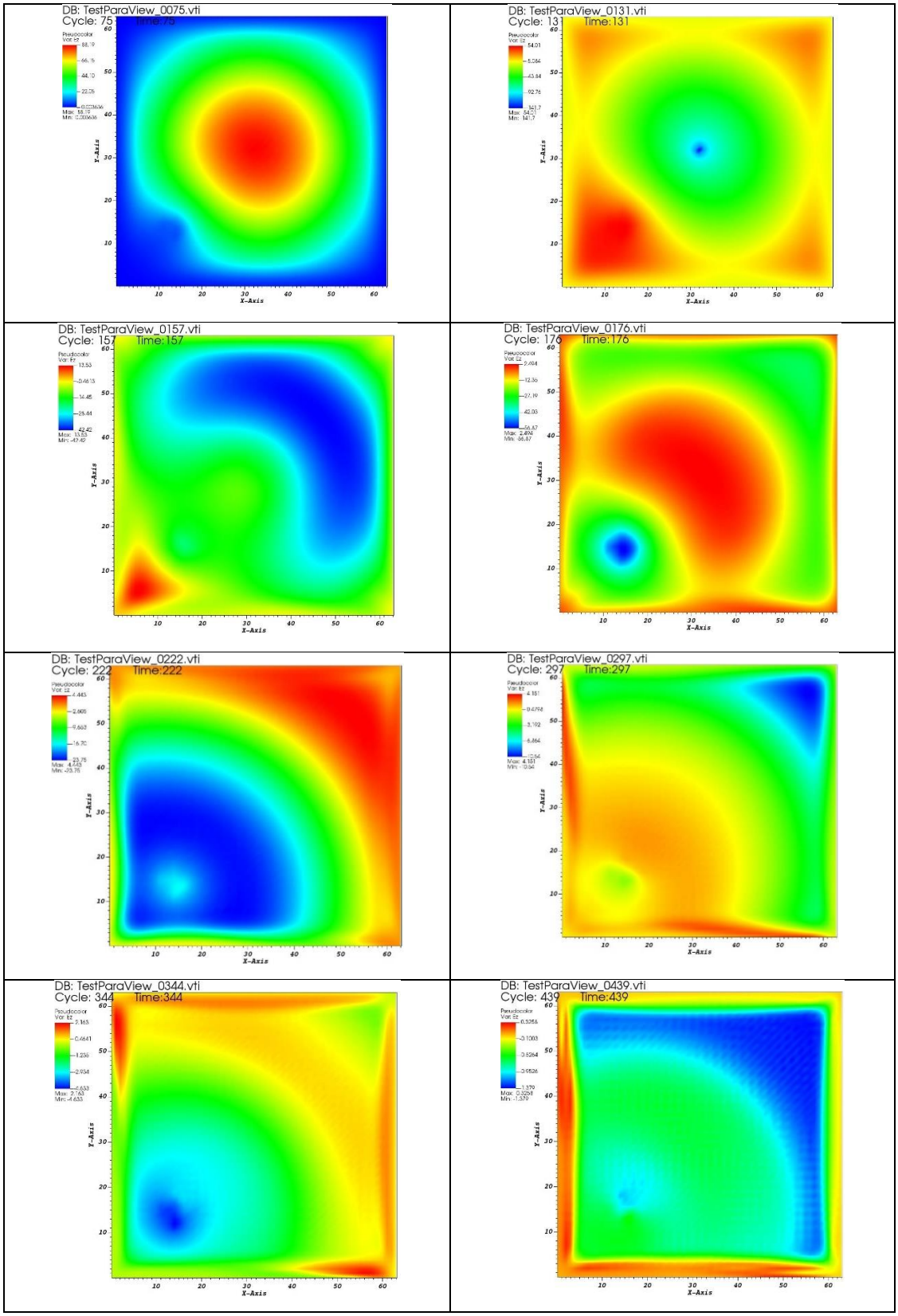
VisIt је адекватан алат за потребе FDTD-а пошто омогућује да се разне информације извуку приликом саме визуелизације. Спајањем великог броја слика се добија филм који моделује понашање електромагнетног таласа у времену. Следе две табеле са неколико изолованих оквира који представљају стање система у појединим тренуцима (слике посматрати с лева на десно и од горе ка доле).

У Табела 2 – Графички приказ симулације система је на сликама представљена „празна“ Јиова мрежа димензија 64*64 у чијим ћелијама је приказана вредност односно јачина електричног поља. У почетном тренутку не постоји електромагнетни извор, тако да је вредност поља једнака 0 у свим ћелијама. После одређеног времена се у средини мреже појављује један електромагнетни импулс. Вредност електричног поља у правцу Z-осе нагло расте док се таласни фронт шири у свим правцима. Како време одмиче, вредност поља достиже свој максимум након чега креће да опада у средини мреже и амплитуда односно таласни фронт се помера све даље од центра. После довољног броја итерација вредност у средини постаје негативна и иде ка минималној док је истовремено максимална вредност (која је сада доста мања од јачине почетног пулса) тада већ негде близу ивица мреже или ју је „напустила“.

На петој слици се виде и недостаци FDTD-а када је у питању представљање закривљених површина. Због увођења слоја који упија све вредности како не би долазило до одбијања од ивица мреже, видимо да сферична структура постаје четвртаста када се приближи ивицама, што се наравно не дешава у природи. Како време одмиче даље тако вредност електричног поља постепено тежи нули.



Табела 2 – Графички приказ симулације система



Табела 3 - Графички приказ симулације система са објектом у мрежи

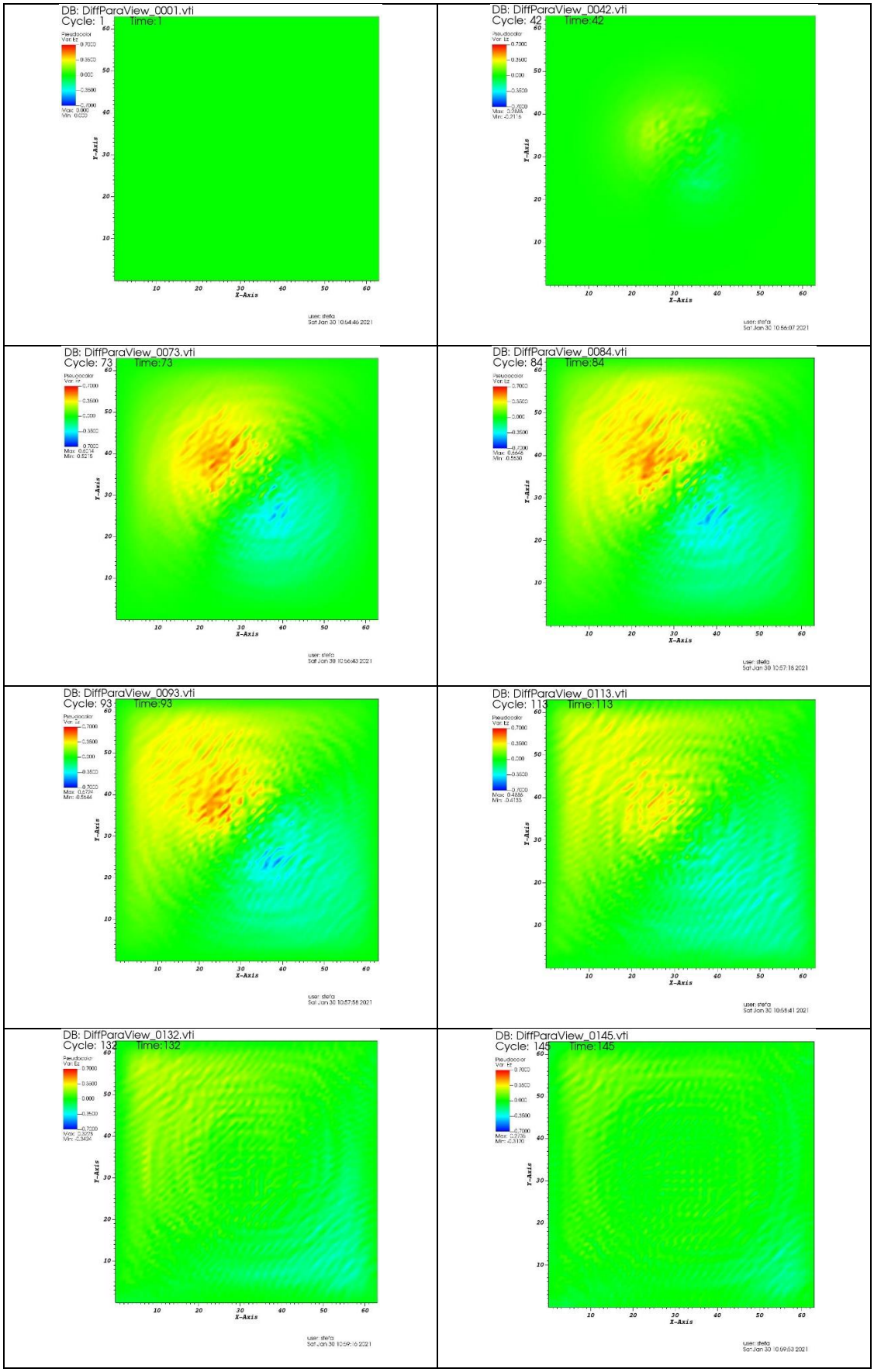
У Табела 3 - Графички приказ симулације система са објектом у мрежисе види знатно другачији исход симулације када се у доњи леви угао Јиове мреже убаци мали објекат налик на авион чија је релативна пермитивност већа 15 пута од окружења. При том, треба нагласити да вредности за релативну пермитивност могу ићи и до неколико стотина хиљада пута веће од оне за вакуум. Прво можемо приметити да се таласни фронт „криви“ тј. прелама око препреке и да рефлектује нови таласни фронт ка извору.

Исцравање јачине електричног поља је само једна од могућности. Оно што такође може бити интересантно као тема овог рада јесте представа разлике између почетног решења базираног на 32-битном покретном зарезу и оптимизованог решења које користи 24-битни блоковски покретни зарез.

У Табела 4 – Графички приказ разлике између референтног и решења које користи BFPсе налазе упоредне слике на којима је представљена разлика између почетног референтног решења и коначног решења. Свака слика одговара истом временском тренутку као у Табела 2 – Графички приказ симулације системакако би резултати били што веродостојније представљени.

Оно што бисмо идеално желели да се види на другој симулацији јесте да је боја увек зелена тј. да је разлика између референтног и оптимизованог решења једнака нули. То је, наравно, математички немогуће с обзиром да се у оптимизованом случају користи знатно мање меморије за представу података па самим тим долази до грешака приликом заокруживања. На местима где је боја црвена је израчуната мања вредност у оптимизованом решењу, док на местима где је боја плава је израчуната већа. Битна ствар коју је овде важно напоменути јесте да су скале за *pseudocolor* поприлично различите за табелу са јачином електричног поља и табелу са разликом.

Конкретно, у табели 2 је забележена максимална вредност од 217,6 волта по метру [V/m] за јачину електричног поља. У табели 4, максимална забележена разлика је 0,67 [V/m]. Уколико посматрамо просечне вредности, разлика је реда величине око 200 пута мање. Како би ишта од наведеног било видљиво, палета боја је остала иста, само су скале прилагођене за вредности које су исто тако око 200 пута мање, како би се разлике лакше уочиле. Више детаља је дато у поглављу са резултатима.



Табела 4 – Графички приказ разлике између референтног и решења које користи BFP

4.3 Имплементација

4.3.1 Архитектура система и имплементација

Почетно, референтно, решење је у потпуности имплементирано у програмском језику Julia. Овај софтверски део решења је подељен у више датотека, из разлога што се неке користе и у другим истраживањима, како би принцип модуларности био испоштован а и одржавање кода је лакше када је величина датотека мања него када она прелази неколико стотина линија кода. Даље у тексту ће бити наведене неке од најбитнијих.

Utils.jl је датотека у којој се налазе помоћне функције и формуле као што су руковање изузецима, подршка за инсталацију одговарајућих јавно доступних пакета, формуле за рачунање апсолутне и релативне средње квадратне грешке, исцртавање спектра и многе друге. *Src_Func.jl* садржи различите моделе простирања таласа који се могу по жељи одабрати као што су различити Гаусови модели. *FDTD_2D.jl* садржи различите имплементације дводимензионалног FDTD алгоритма. У овој датотеци се налази полазни 32-битни алгоритам који користи покретни зарез. Потом следи имплементација са непокретним зарезом и затим неколико различитих верзија алгоритма који користи блоковски покретни зарез. *Compression_FDTD_2D.jl* игра улогу главног програма из кога се позивају сви неопходни модули и функције, као и модул у коме се рачунају сви параметри који илуструју квалитет решења као што су потрошња меморије, време извршавања, грешке између решења као и део који је задужен за прављење „слика“ које се потом анимирају и визуелизују уз помоћ VisIt-a.

Почетно решење када је промењено на BFP није имало боље перформансе у односу на почетно из простог разлога што су данашњи процесори опремљени са наменским копроцесорима за аритметику у покретном зарезу. Други разлог за то јесте што број приступа меморији није смањен јер је за рачунар свеједно да ли учитава из меморије 32-битни или 16-битни или 8-битни податак са становишта временског извршавања. Зато је потребно имплементацију изместити у програмабилну логику као што је FPGA која даје већу слободу за дефинисање сопствених протокола и организације и руковања меморијом.

Од целокупног програмског кода написаног у Julia-и, део који највише има смисла изместити у FPGA је заправо петља за само рачунање. Прерачунавање константи није неопходно обрађивати у програмибилној логици јер има доста операција дељења које није идеално за хардверску имплементацију јер би одузело пуно ресурса. Направљена је посебна скрипта која узима све неопходне податке, рачуне, константе и остале информације од значаја и потом их похрањује у посебну датотеку која се користи као златни вектор података. Разлог за овакву расподелу посла јесте како бисмо могли извршити бит-егзактно поређење између чисто софтверског рачунања и оног извршеног на реалној плочи.

Након имплементације на рачунару, следеће питање која се отвара јесте дилема како спровести имплементацију у програмибилној логици? Намећу се два приступа.

Прво решење јесте у потпуности дефинисати свој сопствени аутохтони систем и реализовати га у неком од језика за опис физичке архитектуре као што је VHDL [60] или Verilog [61]. По правилу, развој система у језицима за опис физичке архитектуре је спор, скуп и подложан већој количини грешака у поређењу са софтверским решењима. У овом случају је неопходно пројектовати систем до најситнијих детаља и дефинисати множачке и проточне структуре како би се максимално искористиле могућности FPGA чипа. Предност оваквог решења јесу најбоље перформансе и максимална искоришћеност ресурса. Оно што је очигледна мана јесте време и енергија утрошени у развој. Поред тога што је овакав процес изузетно подложан грешкама, оваква имплементација би захтевала и потпуно рефакторисање самог изворног кода алгоритма да би се операције прилагодиле хардверским структурама. Приликом рефакторисања би сигурно дошло и до другачијих резултата и поређење не би било адекватно. Решења у потпуности базирана на FPGA чиповима већ постоје [15] и истраживање би у том случају била само још једна имплементација без посебних иновација.

Алтенративни приступ јесте искористити неке од постојећих готових компоненти (процесорских језгара) које би имплементацијом на платформи која садржи FPGA чип била, идеално, боље решење него почетно. Уколико бисмо ова језгра спојили са аритметиком у блоковском покретном зарезу, увећали бисмо

њихову ефикасност јер би количина приступа меморији била смањена, самим тим и време извршавања би било мање.

Једна идеја је користити језгро као што је Nios II иза које стоји компанија Intel [62]. Ово решење је познато, тестирано и за очекивати да ће његова имплементација ићи без великих проблема. Ипак, једно од незанемарљивих недостатака овог решења јесте да је оно затворено, те да није могуће унети измене за своје потребе. Не само да није могуће правити измене, него није могуће ни имати увид у изворни код решења. Такође, за рад са готовим компонентама су често неопходне лиценце чији буџет може бити ограничавајућ фактор.

Уместо таквог приступа, могуће је користити и неко језгро отвореног кода. У том случају је могуће избацити све сувишне делове као и по потреби убацити своје у већ постојећи екосистем. Пример једног овако широко доступног језгра јесте RISC-V [63] развијено на универзитету Berkley California. RISC-V је језгро базирано на редукованом скупу инструкција (самим тим и ознака RISC у имену, скраћено од енг. *reduced instruction set architecture*) створено са идејом да се направи практични инструкцијски скуп који би био доступан свима и који би се користио како у академији, тако и у индустрији. Једна од идеја водиља пројекта јесте да је инструкцијски скуп спрега између физичке архитектуре и програмске подршке. Када би постојао широко доступан скуп, то би смањило цене развоја и подстакло конкуренцију између произвођача физичких архитектура те би више ресурса могло да се преусмери на саме архитектуре а мање на програмску подршку. Другим речима „Инструкцијски скупови требају бити бесплатни“ [64]. На интернету постоји неколицина доступних решења од којих је за потребе овог истраживања искоришћено picorv32, имплементација језгра за микроконтролере, написана у Verilog-у од аутора Клер Вулфа [65].

Главна предност приликом оваквог приступа јесте што је могуће петљу за рачунање написати у неком вишем програмском језику, као што је C++, и потом превести дати код на жељену архитектуру која има широку подршку, што представља велику уштеду у времену и елминише неконзистенстости које би настале евентуалном „директном“ имплементацијом алгоритма у хардверу и преко потребног рефакторисања. Приде, архитектура је таква да се делови сувишни за потребе истраживања могу искључити и тиме остварити уштеду на

ресурсе приликом FPGA имплементације. Тако је могуће направити ефикасан систем који заузима минимално ресурса и евентуално повезати више оваквих система да у паралели рачунају алгоритам као што је предложено у раду [34].

Иако и RISC-V језгро није изузето од проблема са уским грлом, оно је релативно мало и може се имплементирати на малим, повољним FPGA чиповима које троше мало струје у поређењу са графичким картицама. Штавише, могуће је на платформи имати неколико FPGA чипова, сваки спрегнут са својом меморијом, који би радили у паралели и опет трошили мање струје од GPU као тренутног предводника за најбоља решења.

Треба скренути пажњу још једном да брзина извршавања није примарни мотиватор у овом решењу него ефикасније искоришћење меморије, уз мању потрошњу. Самим тим, сматрамо да мане другог приступа не нарушавају иницијалну намеру а то је провера концепта и потврда да је могуће добити резултате из симулација и на реалном систему. Стога је опредељена друга опција.

4.3.2 Рачунска петља у језику C++

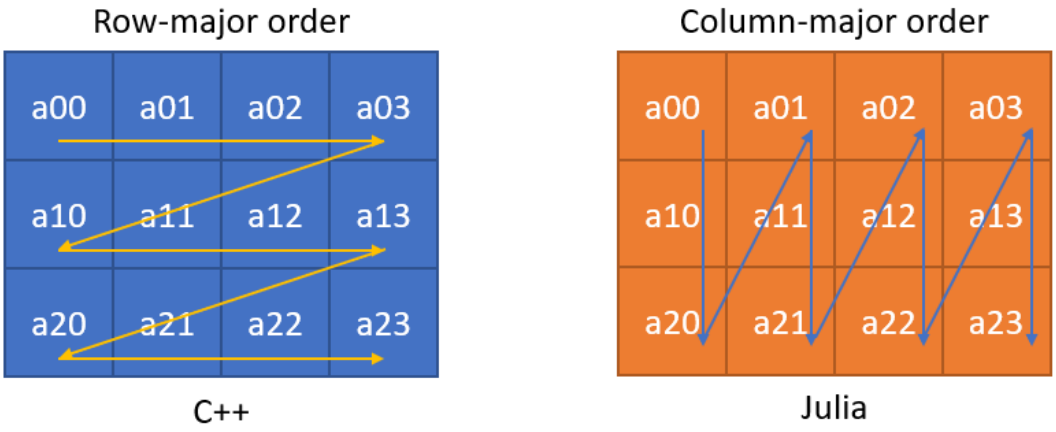
Нулти корак који је био неопходан пре почетка развоја кода у C++-у јесте урадити превођење свих неопходних библиотека из Julia-е у C++. Ту се превасходно мисли на посебну репрезентацију и библиотеку за рад са блоковски покретним зарезом. За ове потребе су искоришћени шаблони (енг. *Templates*) које нуди C++. Шаблони су веома моћан концепт чија је суштина релативно једноставна. Идеја је писати програмски код на начин тако да је независан од типова података над којима ће се радити. Приликом превођења кода, шаблони ће се развити и бити замењени типовима који су прослеђени као параметар шаблону, слично као код макроа, само са додатним проверама које омогућавају лакше откривање грешака.

Један од примера где су коришћени шаблони је за потребе проширивања података са мањег типа на већи (нпр. са целобројног 16-битног на 32-битни). Ово је било неопходно како би могла да се имплементира аритметика BFP-а. Други пример јесте баш класа за прављење BFP-а, која је такође реализована преко шаблона где се приликом креирања прослеђује величина података. Још један

пример јесте класа која моделује матрице које могу да прихватају и обичне просте типове, али могу да прихватају и наш BFP који може имати различите ширине, те се ослања на шаблоне.

Једном када постоји одговарајућа имплементација BFP-а, следећи корак јесте направити сопствене класе које моделују рад са матрицама и које поседују само оно што је неопходно. То укључује преклопљене операторе за сабирање, одузимање и множење матрица, као и функције које би замениле оператор .* (скаларно множење одговарајућих елемената различитих матрица) у Julia-и, будући да такав не постоји у језику C++.

Посебан извор проблема представља интерна интерпретација и руковање дводимензионалним низовима које се разликује међу језицима. Наиме, у језику C++ је заступљен такозвани *row-major* приступ где када се итерира кроз матрицу се прво пролази кроз све колоне једног реда пре него што се пређе на наредни ред. Насупрот њега, у Julia-и се користи *column-major* приступ где се прво пролази кроз све редове једне колоне, пре него што се пређе на наредну.



Слика 15 - Итерирање кроз матрицу

Када се погледају предности и мане ових различитих приступа - у математичком погледу нема разлике. Обилажење матрице у оба случаја има комплексност O(n^2). Оно што је различито јесте поравнање са скривеном (*cache*) меморијом и оно зависи од реализације „испод хаубе“. У C++-у је ситуација иста као и у програмском језику C, приступ по редовима је бржи јер на тај начин постоји већа шанса за погодак у скривеној меморији и самим тим резултује мањим утрошком времена за добављање података, те имамо брже време извршавања.

С обзиром да брзина извршавања није главна ставка истраживања (а и решење у C++-у је само међукорак), како би се минимизовале грешке приликом превођења с једног језика на други, донета је одлука да се класе у C++-у моделују тако да се руковање дводимензионалним низовима понаша идентично као и у Julia-и. На тај начин се највећи део већ готовог и верификованог кода може директно прекопирати уз примену одговарајуће синтаксе језика. Треба споменути још и разлике у виду индексирања, а то је да у C++-у оно почиње од нуле, док у Julia-и почиње од јединице. Самим тим, биле су неопходне додатне измене и провере приликом приступања елементима како не би било приступања погрешним сегментима меморије или пробијања опсега.

Један од проблема који се јавио одмах у старту јесте проблем типизираности. Julia је нетипизиран језик што значи да приликом дефиниције промењиве није потребно назначити њен тип већ ће се он имплицитно одредити. Исто тако ће и све конверзије из једног типа у други (уколико су могуће) бити урађене имплицитно. Насупрот томе, у C++-у је неопходно навести типове промењивих приликом дефиниције као и приликом конверзије. Самим тим, додела типова у C++-у приликом превођења програмског кода из Julia-е понекад није био тако очигледан процес.

Још један од проблема који је настао услед разлике у типизираности језика јесте промоција типова. У језику C++ целобројни тип од 128 бита (int128_t) по стандарду не мора бити подржан од стране свих преводилаца, док он постоји у Julia-и. Приликом конверзија које настају за време множења одређених вредности се у Julia-и појављивао такав тип који онда приликом бинарног записа у датотеку није могао добро да се прочита у C++-у. Иако су све коришћене вредности без проблема могле да стану у 64-битни тип, промоција се дешавала аутоматски и проузроковало је незанемарљив временски губитак. C++ шаблони су били неопходни да би се ови проблеми решили.

Како би се подаци из златног вектора могли учитати у новодефинисане структуре података, направљене су и сопствене функције за рад са серијализованим подацима (енг. *stream*). Једном када су учитани, сам део кода који се бави рачунањем је идентичан као у Julia-и.

Када је установљено да рачунање у C++у даје идентичне (бит-егзактне) резултате као и оно у Julia-и, решење је било спремно за превођење за RISC-V. Остаје још последњи корак – комуникација. Потребно је развити неки вид протокола који би слао податке са рачунара преко спреге на FPGA плочу. Пакете са информацијама је потом потребно изделити у одговарајућем редоследу и проследити петљи на рачунање. У сличном маниру је онда потребно резултате вратити са FPGA на рачунар ради последње провере да су резултати идентични.

4.3.3 Протокол за слање података

Систем је са рачунаром повезан преко UART [66] протокола, уз помоћ посебног модула. UART представља један од стандардних протокола за серијску комуникацију између рачунара и периферије и као такав се веома често налази на микроконтролерима. Сам модул се састоји од дела за генерисање такта, улазних и излазних померачких регистара, управљачког дела као и параметра за брзину преноса података.

На рачунару домаћину (на коме се развија како дизајн, тако и програмска подршка) се налази Linux оперативни систем, због његовог лакшег рада са периферијама. Најпре се програм који ће обављати FDTD рачун на плочи преведе на рачунару домаћину за циљану архитектуру која је у нашем случају RISC-V. Када се програм преведе, направи се извршна датотека коју је потом потребно пребацити на плочу. За то је задужен посебан програм написан у језику C++ који уз помоћ UART протокола уписује садржај извршне датотеке директно у меморију за програм.

Проблем који се овде јавља јесте и проблем односно карактеристика FDTD алгоритма, а то је велика употреба меморије. Иако саме инструкције програма могу без проблема стати у унутрашњу меморију која се налази у FPGA чипу (такозвани *block RAM*, односно BRAM), све матрице које су неопходне за исправан рад напросто превазилазе капацитет ове меморије. Алтернатива јесте пребацити их у следећи ниво меморије, који је у случају доступне развојне плоче синхрона динамичка RAM меморија, односно скраћено SDRAM. Спрега са овим модулом је доста компликованија и захтева посебну компоненту за комуникацију у виду

контролера. Осим тога, овде се појављују проблеми којих нема када целокупан систем почиње и завршава унутар FPGA чипа, а то су временске зависности сигнала који морају у тачно одређено време бити постављени на одређене нивое. У овом тренутку више није могуће у потпуности занемарити време које је потребно да сигнал дође с једног до другог краја система.

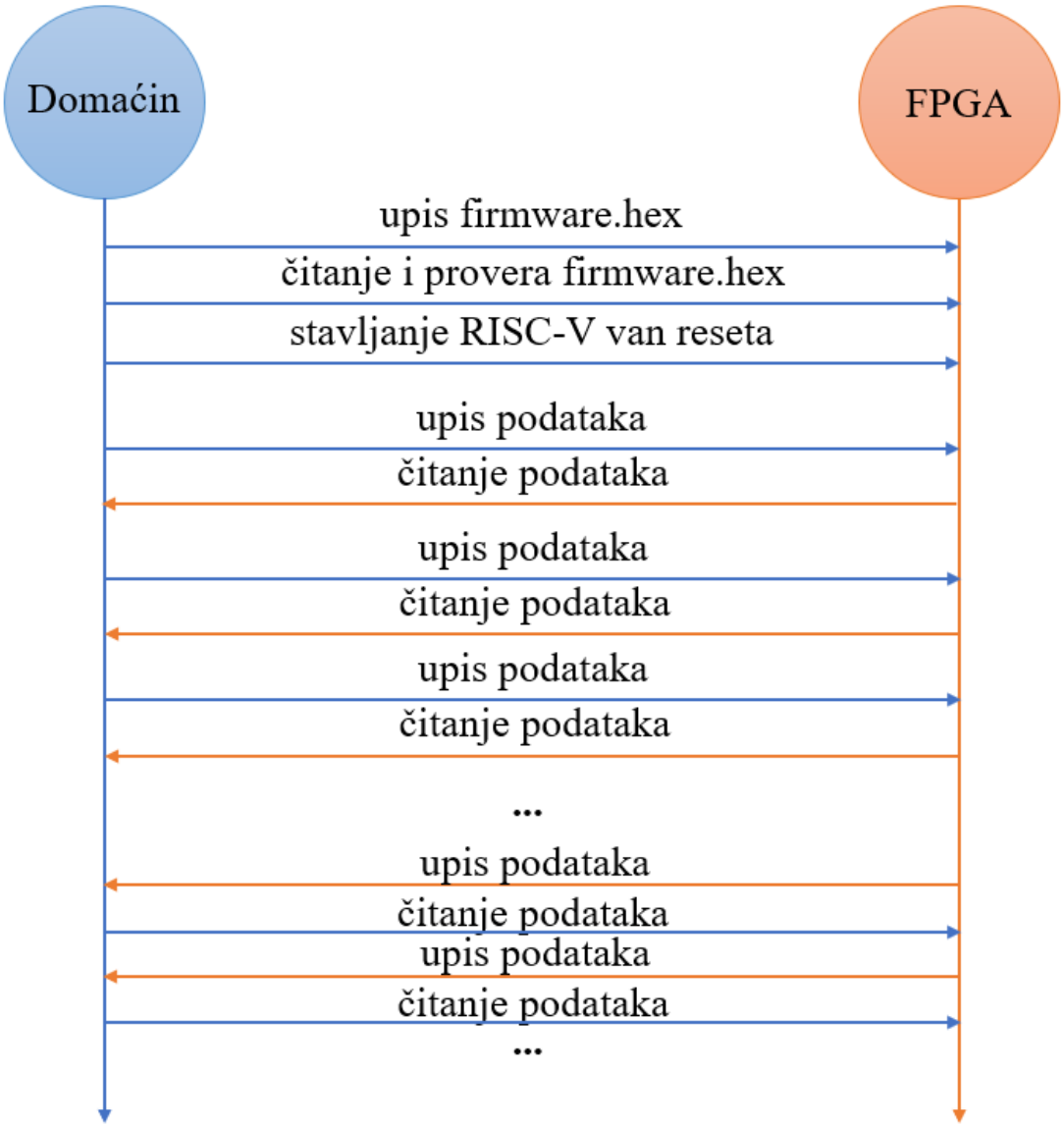
Програм најпре уписује целу извршну датотеку, као и матрице потребне за рад алгоритма, на унапред дефинисане адресе у меморији и потом проверава да ли је заиста уписано све по спецификацији. Када се утврди да је извршна датотека успешно уписана, шаље се посебна команда у меморијски мапирани регистар да се процесор избаци из стања ресета. Овај корак је неопходан јер би у супротном одмах по доласку напајања процесор кренуо да учитава насумичне податке из меморије и понаша се непредвидиво. Зато се процесор држи под константним ресетом све док се сви остали делови система не иницијализују на жељене вредности. Након команде која процесор ставља у оперативно стање, он креће да извршава програм као што би то чинио и да се извршава на персоналном рачунару.

С обзиром да је интерна меморија на развојној плочи исувише мала за све потребе програма, следећи задатак програмске подршке јесте да преко протокола за комуникацију упише све константе статичке матрице у спољашњу меморију. Овде се поново јавља својеврстан проблем – матрице у којима су смештене мантисе су веће и од расположивог бафера, те их је потребно изделити. Опет, с друге стране, пренос података би требало бити што је више могуће агностичан према подацима – није битно кога су типа, за UART протокол су то све бити.

Помоћни протокол функционише на принципу једноставног поштанског сандучета. Програм на развојној плочи очитава стање бафера и све док је празан чека. Истовремено с друге стране, са становишта рачунара домаћина се шаље први део података. Ову операцију је могуће извршити само у случају да је бафер празан. Једном када рачунар домаћин пошаље све податке, обавештава програм на FPGA да је бафер попуњен и он може да крене да учитава низ бајтова и премешта их у спољашњу меморију у одговарајуће променљиве, константе, низове и матрице.

Поједностављен временски дијаграм размене порука је приказан на следећој слици. Дакле, пре извршења рачунарске петље, подаци иду из смера од рачунара

домаћина ка развојној плочи, а када се сви подаци учитају и петља изврши, смер се мења.



Слика 16 – Временски дијаграм размене порука и података

4.3.4 ФПГА имплементација

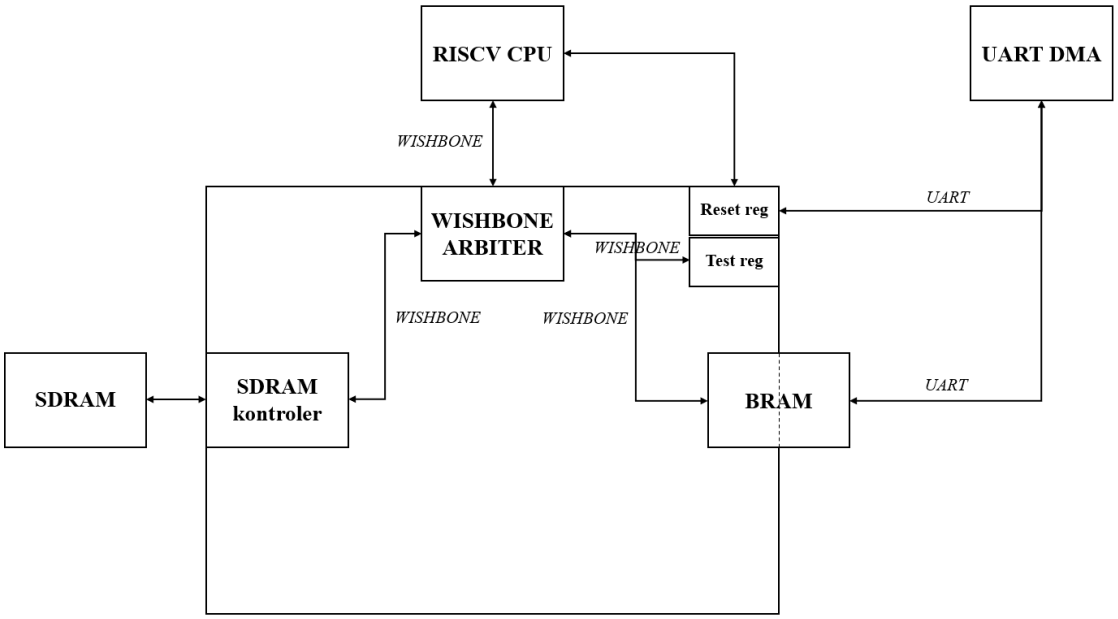
За реализацију потребних елемената у програмибилној логици коришћени су језици за опис физичке архитектуре VHDL и Verilog. Разлог за употребу оба језика уместо само једног јесте постојање већ готових модула од којих су неки реализовани у Verilog-у (цео picorv32, SDRAM контролер), а неки у VHDL-у

(модули за комуникацију преко UART протокола), као и већа упознатост и боље владање VHDL-ом. Оно што је олакшавајућа околност јесте што оба језика могу без проблема укључити и инстанцирати модуле написане у оном другом. Оно што је отежавајућа околност је слична проблематика која постоји у проблему Julia-е и C++-а, а то је да је VHDL изричито типизиран језик где поклапање типова и ширина (број бита) сигнала се мора у потпуности поклапати, док Verilog није ни близу толико ригорозан и преводилац ће у случају да нема поклапања се потрудити да сам разреши ове недоумице. Једна од грешака која је дуго правила проблеме јесте да ширина меморијске магистрале није била добра у два модула и Verilog је ово пријавио само као упозорење.

Први корак у развоју био је имплементација RISC-V процесора на циљној платформи. С обзиром да је у питању дизајн отвореног кода који се може слободно преузети и користити у академске сврхе, то је и урађено. RISC-V процесор подржава мноштво различитих инструкцијских скупова и приликом превођења је могуће одабрати само жељене. У случају овог истраживања, узет је минимални скуп инструкција (стандардне аритметичко-логичке операције) и проширење за множење које је неопходно у рачуну, док су остала проширења искључена.

Осим RISC-V процесора, систем чине још и два меморијска модула – један, мањи унутрашњи, за смештање програма (BRAM – *Block Random Access Memory*) и други, већи спољашњи, који служи за смештање свих података неопходних за рачун (SDRAM – *Syncrhonous Dynamic Random Access Memory*). Са становишта процесора, обе меморије комуницирају с процесором преко Wishbone [67] протокола. То је протокол отвореног кода намењен превасходно да интегрисана логичка кола комуницирају међусобно (као што је у нашем случају где меморијски модули комуницирају са процесором). С обзиром да је идентична спрега према обе меморије, за исправно функционисање система је неопходан и модул (арбитер) који би преусмеравао комуникацију према једном, односно према другом модулу. За комуникацију са спољашњим светом, направљен је и наменски меморијски контролер који с једне стране прихвата податке са рачунара преко UART-а, а са друге стране те податке уписује директно у интерну меморију. Део система је и пар меморијски мапираних регистара са посебном наменом као што су ресетовање процесора или контролни регистри који се користе као показатељи

да програм ради успешно. Архитектура система је приказана на Слика 17 – Архитектура система.



Слика 17 – Архитектура система

UART DMA је модул који је повезан са програмском подршком и са рачунаром домаћином на коме се налазе резултати симулације. Састоји се од аутомата с коначним бројем стања, неколицине бројача и помоћних логичких компоненти које служе за откривање специфичних догађаја. С обзиром да је комуникација врло поједностављена, постоје само две врсте порука – читање и писање. Када се установи која је команда у питању, следећа ставка јесте прочитати колико података ће бити прочитано / уписано и најзад од које адресе креће читање / писање, које се потом обавља сукцесивно једну по једну локацију. Бројачи су ту да изброје 3 бајта унутар којих се налази информација о количини података, као и 4 бајта који представљају адресу. У UART DMA бајти стижу већ формирани од стране спреге која је задужена за UART протокол на нивоу бита.

BRAM је унутрашња меморија присутна у самом FPGA чипу. Ово је стандардна компонента која има две спреге – једну са UART DMA модулом и другу са процесором преко арбитера. Две спреге омогућују да се у меморију истовремено може уписивати и из ње читати. Са „спољашње“ стране, овај модул комуницира, уз помоћ посебног аутомата са коначним стањем, директно са UART DMA модулом. Са „унутрашње“ стране према процесору, односно арбитеру,

постоји горе поменута Wishbone спрега. Суштински, унутрашња меморија је заправо низ регистара укупног капацитета од 16 kB, који се састоји од 4096 речи. Овај капацитет је довољан за цео преведен програм, као и бафер за комуникацију рачунара домаћина и RISC-V. Иако су спреге различите, обе раде на истом принципу – на растућој ивици такта, уколико постоји дозвола, вредност на линији за податке ће бити уписана на адресу која се налази на линијама за адресу. Читање је комбинационо и самим тим не зависи од такта.

SDRAM је спољашња меморија [68] која је са FPGA чипом повезана преко излазних пинова. Њен капацитет је знатно већи од унутрашње меморије, али је и руковање компликованије. Зато постоји посебан контролер који с једне стране прихвата Wishbone команде и претвара их у одговарајући скуп наредби које препознаје SDRAM. Генерално гледано, SDRAM модул има неколико сигнала за контролу рада, неколико за издавање наредби и остале који служе за адресирање, односно позиционирање на одговарајућу меморијску локацију унутар меморије. Ако изузмемо сигнал такта који је подразумеван у свакој секвенцијалној мрежи, главни сигнал јесте сигнал дозволе рада односно **CKE**. Када је овај сигнал на ниском напонском нивоу, меморија се понаша као да је извор такта стао, односно нема интерпретација команди. Следећи сигнал од интереса је сигнал одабира **CS#** који је инвертован (ознака # у неким документацијама означава да је сигнал активан на ниском напонском нивоу, а неактиван на високом). Када је овај сигнал на високом нивоу, меморија игнорише све управљачке сигнале. Од управљачких сигнала су интересантни **RAS#**, **CAS#** и **WE#**, сва три у инверзној логици. Заједно, ова 3 сигнала различитим комбинацијама дају команде као што су – без операције (када није потребно ништа радити с меморијом), читање, писање, активација, пуњење, освежавање и уписивање режима рада. Читање и писање су интуитивно јасни шта раде, и овде је битно споменути да се приликом обављања ових операција обично не ради о једном податку, него о више података. Пре читања и писања је неопходно активирати одговарајућу банку у меморији, а након што је операција успешно затворена, банка се може затворити или оставити отвореном. Пуњење и освежавање су операције које су неопходне због технологије израде. SDRAM као и меморије које су наследнице DDR2, DDR3, DDR4 су реализоване уз помоћ кондензатора који чувају наелектрисање и самим тим и вредности

логичких нула и јединица. Међутим, кондензатори временом губе наелектрисање и да не би дошло до „цурења“ и самим тим губитка информација, периодично се ради освежавање док пре уписа и читања, меморијске локације (које су дефинисане редом и колоном) морају бити активне. Поред ових контролних сигнала, ту су још и сигнали за адресирање као и за податке. Оно што је посебно код SDRAM меморије, јесте да је пре прве употребе потребно исправно конфигурисати чип. То се ради низом веома прецизних и специфичних команди где се временска ограничења морају строго поштовати. Овај део истраживања је одузео доста времена и била је неопходна и спољашња помоћ од произвођача.

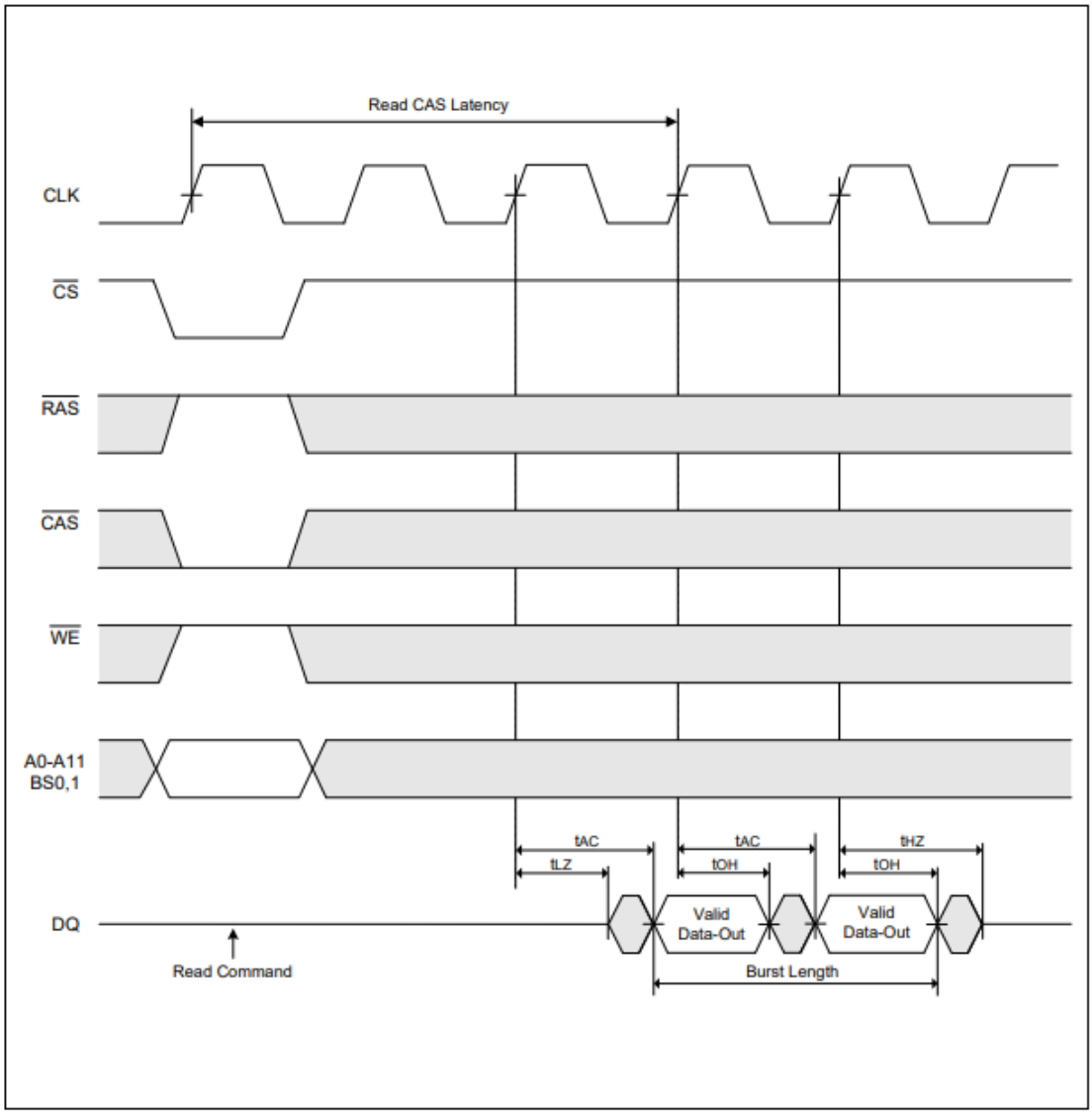
Арбитер је посебна компонента која служи за преусмеравање комуникације од процесора као водећег елемента ка меморијама и регистрима као пратећим. С обзиром да процесор на исти начин комуницира са било којим од својих пратећих модула, потребна је посебна компонента која ће се постарати да подаци заврше на правом месту или буду прочитани са праве локације. Главна информација која је потребна за рад арбитера јесте адреса. У зависности од тога ком подскупу адреса припада, подаци ће завршити или у унутрашњој, или спољашњој меморији или у неким од меморијски мапираних регистара.

Компонента која недостаје на слици због прегледности а која је свакако битна јесте PLL односно фазно закључана петља. На улазу ове компоненте се налази такт сигнал, а на излазу се налази (неколико) такт сигнал(а) чија је фреквенција и фаза у унапред предефинисаној релацији са почетним сигналом. Тако да се фазно закључана петља може користити за повећање или смањење фреквенције али и за померање фазе. Када се цео систем налази унутар FPGA чипа, све компоненте типично раде на истом такту и на истој фази – растућа ивица такта долази у истом тренутку до свих компоненти које од ње зависе. Сам процес спровођења такта је комплексна ствар коју за инжењере обављају окружења за развој. Међутим, када цео систем не може стати у чип, као што је ситуација у овом истраживању будући да се спољашња меморија физички налази на другом чипу, ту минимална кашњења могу да доведу до ситуације да систем не ради исправно.

Узмимо за пример операцију читања од стране процесора ка SDRAM-у, приказану на Слика 18 – Временски дијаграм операције читања SDRAM-а. Спецификација меморије налаже да на растућу ивицу такта када желимо да

извршимо операцију читања, **CS#** мора имати ниску вредност, **RAS#** мора имати високу вредност, **CAS#** мора имати ниску вредност, док **WE#** мора имати високу вредност. Уколико је ово испоштовано, у зависности од параметра колико је кашњење читања, податак ће се појавити 2 или 3 такта касније тј. незнатно пре тога. Када се деси растућа ивица, сви горе поменути сигнали морају имати стабилне вредности. Уколико би такт SDRAM-а био у потпуности поравнат са остатком система, могуће је да се деси да неки од сигнала некад закасни за коју нано секунду и резултује скроз другом операцијом. Довољно је да се овај догађај деси једном за време извршавања програма и то ће резултовати неисправним решењем или процесор неће добити податке које очекује и замрзнуће свој рад. Једини начин да ово ограничење буде испуњено јесте да такт који улази у SDRAM буде фазно померен. У нашем случају, фазни померај је $\pi/2$, односно 90 степени. Другим речима, ако је периода такта 20 наносекунди, растућа ивица за SDRAM ће каснити 5 наносекунди за растућом ивицом остатка система. Тај период је сасвим довољан да се сви сигнали који иду од процесора ка меморији стабилизују и да нема неконзистентних стања.

Још треба напоменути да је радни такт плоче скромних 12.5 MHz али да уз плочу иде наменска фазно закључана петља која може подићи радни такт за ред величине. У принципу, што је већи радни такт, то је систем бржи, али и ту постоје ограничења. Наиме, што је радни такт већи, то је теже конфигурисати FPGA чип тако да тај такт подједнако брзо дође до свих компоненти истовремено. Ову анализу такође обавља окружење за развој. Уколико је радни такт исувише велики, то ограничење неће бити испоштовано и самим тим постоји велика опасност да ће систем доћи у неконзистентно стање, будући да неће сви регистри истовремено мењати своје вредности. Исто треба споменути и да је SDRAM меморија такође ограничена с горње стране максималним радним тактом којим може без грешке да испоручује перформансе које се од ње очекују. Узимајући све ово у обзир, фазно закључана петља је конфигурисана да даје такт од 50 MHz, а утицај ове фреквенције на крајње резултате је дат у дискусији.



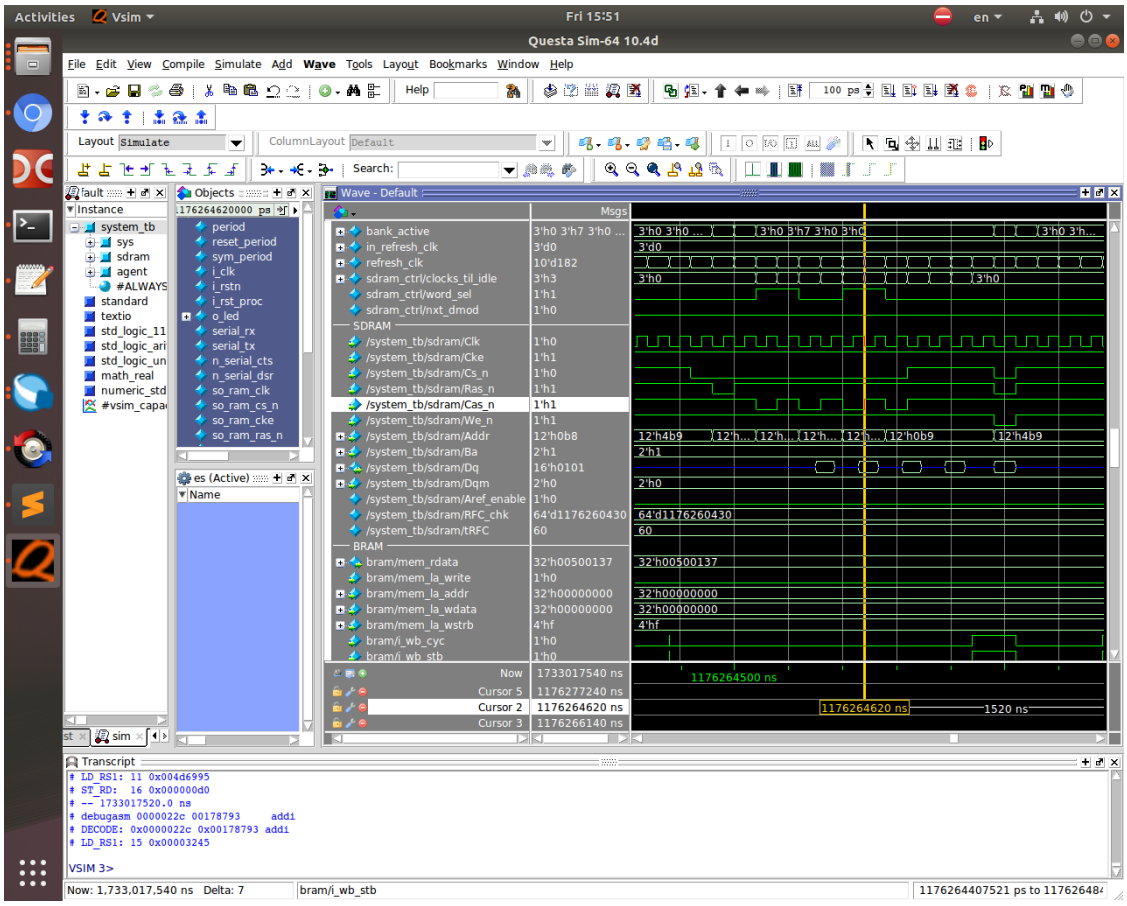
Слика 18 – Временски дијаграм операције читања SDRAM-а [68]

4.3.5 Отклањање грешака и Verilog-ова програмска језична спрега

Једна од главних мана пројектовања физичке архитектуре спрам програмског решења неког проблема јесте време развоја и верификације. Рад на тако ниском нивоу где проблем долази у додир са реалним физичким светом са

собом вуче изазове који не постоје у свету који је заштићен оперативним системом и готовим компонентама које су преживеле тест тржишта.

Генерално гледано, постоје два најистакнутија метода провере исправности физичке архитектуре – софтверски (користећи неки вид симулатора или емулатора) и хардверски тј. употребом развојне плоче. Оба начина су неизоставна јер имају различите предности и мане. Софтверски вид који се ослања на симулацију даје најбољи увид у све компоненте система. У сваком тренутку је могуће видети вредност сваког сигнала и упоредити га са осталима, као што је приказано на Слика 19 – Изглед симулације система.

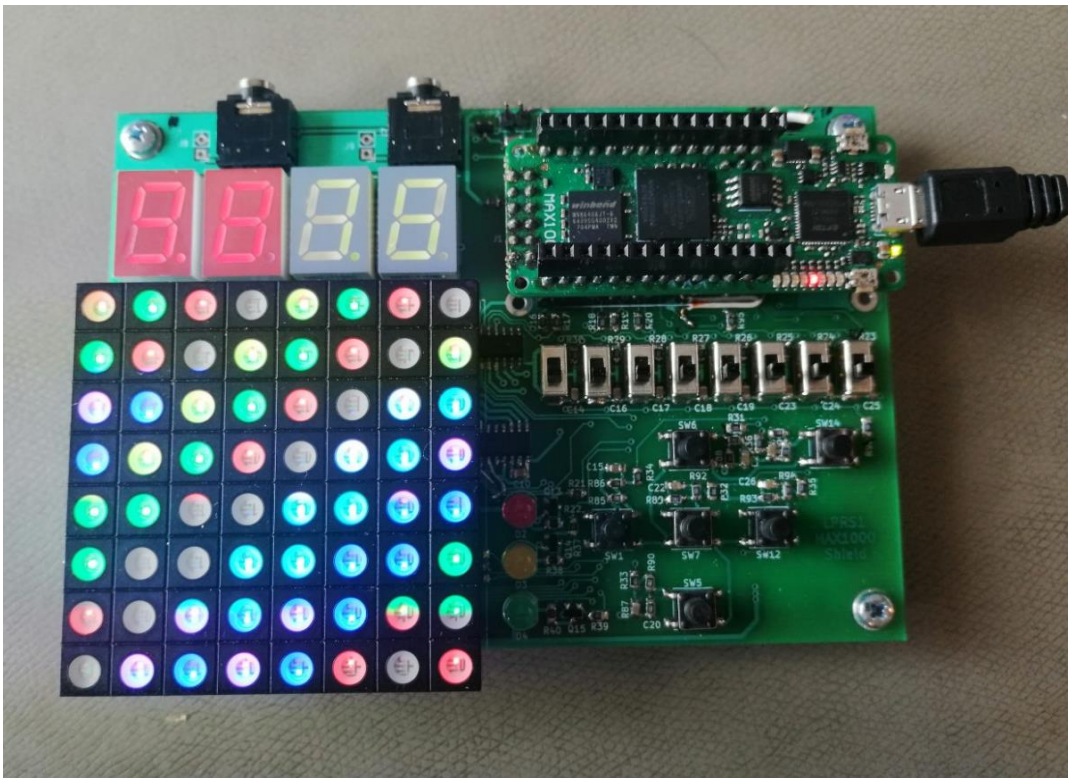


Слика 19 – Изглед симулације система

Од свих мана овог приступа истичу се две. Прва је да је симулација неупоредиво спорија од провере на реалној плочи. Симулирање једне секунде иоле комплекснијег система као што је овај, траје десетине минута, што је посебно незгодно када је потребно симулирати мало дужи рад система јер проблеми

настају касније у његовом раду. Овај проблем је до те мере изражен да раунар на коме је спроведен највећи део истраживања једноставно није могао да поднесе дуже од 20 минута симулације. Друга мана која се никако не сме занемарити јесте да је симулација начин како рачунар тумачи понашање другог дигиталног система. Због секвенцијалне природе рачунара који извшава програме линију по линију (јер на крају крајева и симулатор је врста програма), симулатори наилазе на проблеме када је потребно да се 2 догађаја десе истовремено. Овај проблем није у потпуности решен ни дан данас. Самим тим, оно што се десило небројени број пута и на овом истраживању јесте да систем у симулацији ради исправно али да у реалном свету то није ситуација. Тако да, иако систем који ради у симулацији јесте показатељ његове ваљаности, то није истовремено и гаранција.

Други начин јесте провера исправности система у „реалном свету“ користећи развојну плочу. За те потребе, коришћена је плоча MAX1000 [69], која на себи има жељени FPGA чип, раније поменути SDRAM и периферије као што су UART, прекидачи, тастери, диоде и седмосегментни дисплеји. Изглед платформе је приказан на следећој слици.



Слика 20 – Развојна платформа

Овај начин провере исправности даје резултате у реалном времену – једна секунда је заиста једна секунда рада система. Проблем се огледа у спрези плоче са човеком – са свега 8 диода и 4 седмосегментна дисплеја, не може се велика количина информација пренети кориснику. Оно што се може донекле закључити јесте да ли систем ради или не ради. Уколико систем даје исправне резултате, може се сматрати да ради. Међутим, шта кад не ради? Уз помоћ диода је могуће установити која је последња инструкција која је успешно извршена, али се не може установити зашто наредна није. С обзиром да систем неће радити док се не уклоне све логичке и техничке грешке, неопходно је користити и развојну плочу и симулатор.

Симулација система који је у потпуности садржан у FPGA чипу углавном није проблем, осим уколико систем није изузетно велики па рачунар не може да обави све прорачуне због недостатка ресурса. Чак и у таквим ситуацијама је могуће посматрати само неколицину сигнала од интереса што драстично смањује потребе рачунара за ресурсима. Ситуација се знатно компликује ако није цео систем садржан у чипу. У том случају је неопходно некако емулирати све спољашње сигнале. Притисак тастера и мењање вредности прекидача је релативно тривијално емулирати, SDRAM чип је у овом случају имао свој модел који је доставио произвођач јер без тога не би било могуће установити било какав проблем са меморијом. Чак и са тим моделом, рад и проблеми са SDRAM-ом су одузели убедљиво највише времена. Унос података у BRAM је већ за нијансу компликованији проблем. Могуће је директно у самом опису модула као почетне вредности ставити бинарне машинске инструкције, али овај процес је изузетно спор и склон грешкама. Додатне компликације настају уколико је изворни код програма потребно мењати јер је време од промене кода до резултата симулације у том случају изузетно дугачко, али и даље у домену изводивог.

Домен напорног али изводивог напуштамо оног тренутка када је потребно емулирати податке које домаћин рачунар (на коме се налазе златни вектори података) уписује у бафер. Не само да је немогуће израчунати временске периоде свих сигнала, слања пакета и слично него у суштини немогуће је предвидети проблеме који могу настатати у комуникацији између рачунара и платформе. Зато

је потребан неки објективан и робуснији механизам. Један од механизама за решење овог проблема јесте Verilog-ова прогамска језична спрега (PLI) [70].

Једна од главних снага језика Verilog управо јесте његова програмска спрега која је донекле и заслужна за његову широку примену [70]. PLI омогућује проширења могућности како самог језика тако и симулатора, и самим тим превазилази оно што је иницијално замишљено да окружење језика и симулатора може постићи - развој и симулација дигиталних логичких система. Од свих елемената које се могу уградити, за потребе истраживања је најзначајнија спрега односно интеграција са функцијама написаним у C/C++-у. C/C++ се релативно лако може спрегнути са другим програмским језицима тако да се могу симулирати и промене и дешавања у неким другим програмским језицима. Уз помоћ PLI-а је могуће, на пример, пратити вредност променљивих у које UART протокол смешта своје вредности, те је могуће приказати тачне временске тренутке када је (и како) до трансфера дошло. Променљива дефинисана у неком вишем програмском језику се може приказати у симулатору и упоредно се анализирати колико је кашњење у односу на друге сигнале и да ли су све транизиције временски поравнате као што је предвиђено. За успешан рад је потребно дефинисати такозване агенте.

Агент је један посебан модул написан у Verilog-у који у себи садржи цео систем од интереса. У случају ове дисертације, систем се састојао од модела SDRAM-а, процесора и његових саставних делова, унутрашње меморије и UART модула као спреге са „спољашњим светом“. Дакле, агент ће бити свестан вредности свих сигнала који би у реалном свету напустили платформу. Са друге стране, унутар програма се дефинише одговарајућа C/C++ функција која ће бити позвана када се одређени догађај деси. Спрезањем ова два света је могуће програмски обрађивати жељене догађаје унутар „хардверског“ света, као и унутар симулатора приказати како се хардвер понаша када се појави неки догађај из „софтверског“ света. Тако да ће агент бити свестан и свих вредности сигнала које би требале да уђу на плочу у реалном свету. Другим речима, уз помоћ PLI-а је сада могуће симулирати цео систем а не само делове који се налазе унутар FPGA чипа.

Да би агенти могли да раде, потребно је и да физички буду повезани са реалним светом. За потребе тога су коришћена два UART прилагођивача (eng. *dongle*) који су међусобно повезани.

Слика 21 – UART прилагођивачи

Прилагођивач с једне стране преузима податке са рачунара преко USB магистрале и обликује их и шаље другом прилагођивачу поштујући UART протокол. Од три жице које се виде на Слика 21 – UART прилагођивачи, једна

представља уземљење, а друге две служе за пренос података. Унутар UART протокола постоји предајни (TX) и пријемни (RX) канал. Конфигурација је таква да је предајни канал једног прилагођивача повезан на пријемни канал другог и обрнуто, а уземљења су међусобно повезана. Оба прилагођивача морају бити подешена на исту брзину сигнализације. За наш конкретан случај је вредност била 2 милиона бита у секунди. Када подаци дођу до другог прилагођивача, они се опет преко USB магистрале читају од стране агента и обрађују. Ти подаци се сада могу приказати у симулатору и испратити цео њихов пут од уласка на плочу, све до смештања у SDRAM, на другом крају система, и обрнуто од SDRAM-а ка излазу. Тек у овој фази су констатовани многи проблеми који су се испољили тек након дужег рада програма и већег преноса података.

Једна од првих грешака која је установљена јесте да због логичке грешке у софтверу, преведени програм када се упише у унутрашњу меморију система није радио уколико је био довољно велик. Тачније, свака 256. интрукција није била добро уписана. Овај проблем се није могао приметити у ранијим фазама јер програм који се уписивао је био мање величине и никада није долазило до овог проблема.

Још једна велика грешка је била услед губитка података од SDRAM-а ка остатку система због грешке у куцању. Овде се испољио и проблем коришћења различитих језика за опис архитектуре у реализацији система. У суштини је једноставно укључити VHDL или Verilog модуле један у други. Међутим, због њихове различите типизираности неки проблеми које би у VHDL-у резултовали грешком, у Verilog-у преводилац третира као упозорење и грешке као такве се „провукле“. Грешка је била у ширини магистрале која је на уласку у арбитер имала 32 бита а на излазу само 1. Тако да су први једноставни тестови да ли нешто ради или не пролазили тј. давали илузију да је све у реду јер су се ослањали само на вредност последње диоде.

Проблем који је одузео убедљиво највише времена јесте био освежавање SDRAM меморије. Према спецификацији, меморију је неопходно освежити једном сваке 64 милисекунде. У реализацији контролера је операција освежавања била најприоритетнија, будући да ако временски рок није испоштован меморија би просто изгубила неке вредности и цео систем више не би радио. Оно што се

дешавало, међутим, јесте да се операција освежавања поклапала са читањем или писањем. Једном када је читање или писање започето, није могуће прекинути га операцијом освежавања и ово је резултовало изгубљеним подацима. Да ствар буде гора, због стохастике система (у ком тренутку ће почети пренос података у односу на тренутак када меморија постане оперативна) су се подаци губили на другим местима и увек у другачије време, тако да је систем на моменте деловао да ради а на моменте да не ради и то је уносило додатну конфузију у систем. Овај проблем је решен тако што је удуплана фреквенција освежавања, а сама операција стављена на најнижи приоритет. Будући да освежавање траје свега један такт, утицај дуплирања учестаности освежавања на перформансе система је занемарив. А с обзиром да је алгоритам такав да између суседних читања и писања SDRAM-а има по неколико тактова где процесор рачуна, то је сасвим довољан временски период да се меморија освежи без утицаја на перформансе система.

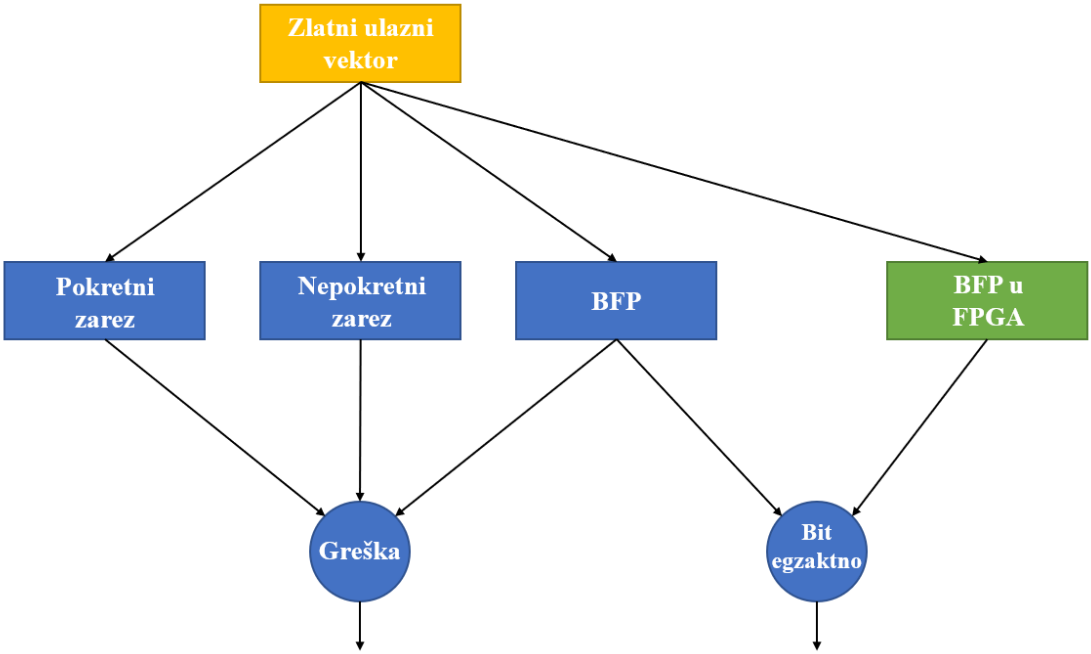
Ни један од ових проблема, као ни сијасет ситнијих није било могуће открити а самим тим ни решити без употребе Verilog PLI-а. Све ово додатно илуструје потешкоће које могу настати и самим мењањем и прилагођавањем једног дизајна новој намени, а камо ли имплементација свега од почетка.

ПОГЛАВЉЕ 5.

РЕЗУЛТАТИ

5.1 Симулација

Као прва потврда хипотезе је коришћена симулација са релативно малом величином Јиове матрице. Ипак, корисни закључци су изведени и из ове мале симулације који су касније пробани и у пракси, користећи имплементацију у програмибилној логици. Принцип који је коришћен за верификацију је следећи: иницијално решење је узето као референтна тачка и за њу је генерисан такозвани златни вектор података. Овај исти вектор података се касније користио у свим наредним унапређењима самог алгоритма, док су се излази поредили како би се установило које решење је боље по жељењном критеријуму. За прва три облика су коришћене методе рачунања грешке док је између последњег облика (који користи блоковски покретни зарез) и решења на плочи урађена бит-егзактна провера.



Слика 22 – Верификација система

Јиова матрица је димензија 64 × 64 елемента, док је број итерација самог FDTD алгоритма био 150. У потрази за оптималним решењем, величина блокова је варирана од димензија 2 × 2 до 32 × 32, при том је сваки пут величина блокова била таква да није било остатака при дељењу димензије матрице и димензије блока. Сами елементи Јиове матрице су се формирали из две матрице - једне која садржи експонент за дати блок, и друге која представља мантису. За представу експонента је коришћено 8 бита, док је за мантису 16. Од тих 16, број бита опредељен за фракцију је 14, а преостала 2 за цео део броја. Тако да све укупно, је кориштено 24 бита по елементу, с тим да су се неке вредности користиле на више места. За мерење протока података, број бита по матрици је дефинисан следећом формулом:

$$\mathbf{Sum = Size*16 + Num*8} \qquad (22)$$

Sum је укупан број бита, *Size* је величина једног блока (нпр. 4 × 4) и множи се са 16 јер је толики број бита опредељен за непокретни зарез. *Num* је укупан број блокова потребан да се поплоча цела Јиова матрица и множи се са 8 јер је за сваки блок опредељен тачно један осмобитни експонент.

Друга метрика која је коришћена за мерење перформанси јесте прецизност. Прецизност као мера „тачности“ решења се може дефинисати на разне начине. Ако узмемо почетно решење као „тачно“ (што не мора бити случај), било које одступање би представљало грешку. Грешка се може изразити као апсолутна – разлика између почетне и добијене вредности по апсолутној вредности. Међутим ради илустрације је боље користити релативну апсолутну грешку која нам говори процентуално колико смо одступили. С тим да грешке могу бити велике у једном сегменту мерења а мале у другом, тако да још лепшу слику добијамо када усреднимо вредности. Самим тим, прецизност је рачуната користећи средњу релативну апсолутну грешку електричног поља у правцу Z осе, изражену у процентима. Грешку дефиншемо по следећој формули:

$$\mathbf{D(x,y)} = \begin{cases} \mathbf{0}, & \boldsymbol{x} = \mathbf{0} \text{ и } \boldsymbol{y} = \mathbf{0} \\ \frac{\mathbf{2}(\boldsymbol{x}-\boldsymbol{y})}{|\boldsymbol{x}|+|\boldsymbol{y}|}, & \text{у осталим случајевима} \end{cases} \tag{23}$$

X представља вредност из решења са покретним зарезом док **Y** представља вредност у решењу која користи BFP. Овакав вид рачунања грешке је релативно једноставан али свакако не без мана, поготово када су у питању ситуације када је јачина поља 0 или када су грешке јако велике. С обзиром да је циљ био добити грешку која је испод 5%, други проблем није увео потешкоће. Ипак, пошто је вредност поља неретко једнака 0, како би се избегло дељење с нулом, у коду је постојао посебан случај који је покривао ове ситуације где се грешка једноставно дефинише да је 0. Ово је и логично јер ако не постоји разлика у почетној и измереној вредности, грешка треба да буде 0 без обзира на стварне вредности поља. У супротном, грешка је дефинисана као у формули.

Сама прецизност је рачуната на начин на који се пролазило кроз низ матрица од којих свака садржи вредност електричног поља у правцу Z осе за сваку итерацију и њихове вредности су сабиране. На крају се резултат дели са бројем итерација (*iters* у формули) и величином матрица (*Nx* и *Ny* представљају димензије) како би се добио медијан апсолутне вредности грешке која је објашњена у формули (23). *L1* је вредност почетног решења базираног на покретном зарезу а *L2* коначног решења са BFP.

$$\boldsymbol{Err} = \frac{\sum_{i=i}^{i=iters} \sum_{x=1}^{x=Nx} \sum_{y=1}^{y=Ny} |\boldsymbol{D}(\boldsymbol{L1}[\boldsymbol{i}][\boldsymbol{x}][\boldsymbol{y}], \boldsymbol{L2}[\boldsymbol{i}][\boldsymbol{x}][\boldsymbol{y}])|}{\boldsymbol{iters} * \boldsymbol{Nx} * \boldsymbol{Ny}} \quad (24)$$

Овакав тип релативне грешке (24) узима у обзир колика је разлика између почетног и крајњег решења у поређењу са апсолутним вредностима. Помножен са 100, постаје релативна грешка у процентима. Примера ради, уколико је измерена разлика у вредности од 0.1 где су при том апсолутне вредности 0.5 и 0.6, ова грешка се сматра већом него у случају да је разлика 10, а забележене апсолутне вредности 550 и 560. Главни разлог за овакав вид мерења грешке јесте што пружа бољу перспективу перформанси решења. Следи табеларни приказ презизности и уштеде меморије.

Величина блока	Број блокова	Релативна грешка [%]	Потрошња меморије [%]
2 × 2	1024	3.59	56.25
4 × 4	256	3.79	51.56
8 × 8	64	5.84	50.39
16 × 16	16	10.10	50.10
32 × 32	4	12.92	50.02

Табела 5 – Измерене вредности спрам симулације

Вратимо се на пример авиончића који је убачен у простор зарад илустрације како алгоритам ради. Грешка између симулације када се авиончић налази у мрежи и када се не налази је око 5.2%. То значи да уколико преласком на BFP добијамо грешку која је већа од ове, то значи да промена уноси превише шума да би се могла користити у практичним апликацијама.

Из табеле се може закључити да за наш тражени случај (релативна грешка испод 5%) највећа величина блокова (и самим тим и највећа уштеда) која задовољава критеријум износи 4*4. Разлике у времену извршавања за различите блокове су биле безначајно мале и већу разлику је правила стохастика услед распоређивања процеса од стране оперативног система.

Иницијално 32-битно решење се узима као полазна тачка, њена релативна грешка је 0 (јер се пореди само са собом) и захтева највећу количину меморије која је илустрације ради означена са 100%. Са првом еволуцијом алгоритма и преласком на 24-битни формат са непокретним зарезом се остварује 25% уштеде. Из табеле 6 се да приметити да је утрошак меморије користећи BFP тек нешто већи од 50% полазног. У поређењу са решењем које је користило само 16-битни фиксни зарез које даје још незнатно већу уштеду у меморији, ово решење има грешку која је, у завиности од величине блокова, мања за скоро 6 пута и тиме представља видан искорак.

Бројке из симулације представљају обећавајући резултат али колико је стварно убрзање у реалном свету остаје да се види. Ограничења Von Neuman-ове архитектуре остају а то је да је приступ меморији скуп у односу на рачунске операције. Иако је мање приступа меморији свакако добра ствар, није за очекивати да ће дупло мањи утрошак меморије повећати перформансе два пута.

5.2 Резултати са рачунара и платформе

Иницијално референтно решење је написано без оптимизација и паралелизација и имплементирано у покретном зарезу у програмском језику Julia. Програм је извршаван под Linux оперативним системом и процесором Intel i5 радне фреквенције 3.6 GHz. Просечно време извршавања програма је износило 9.43 секунде. Када направимо количник укупне количине података и времена извршавања, добијамо фактор од 0.022 мега ћелија по секунди. Уколико упоредимо ово решење са почетним решењем имплементираним у [15] чији је просек 0.59 мега ћелија по секунди за процесор сличних карактеристика, може се установити да има доста простора за унапређивање оригиналног алгоритма, као и да програмски језик Fortran и Julia вероватно дају знато другачији извршни машински код, самим тим и различита времена извршавања.

С обзиром да је решење на FPGA такође базирано на процесорском језгру, и да је у питању FPGA а не наменско интегрисано коло, очекује се да време извршања буде знатно спорије него што би то теоретски било могуће. Програм

преписан у C++ и измењен да користи блоковски покретни зарез на циљној платформи која ради на скромних 50 MHz је трајао у просеку 40.89 секунди, што резултује са скромних 0.005 мега ћелија по секунди. Чак и у случају када би се решење догурало до максималне фреквенције од 100 MHz коју дозвољава платформа (и посматрајући идеалне услове, без промене временских зависности и карактеристика), резултат не би теоретски могао прећи 0.01 мега ћелија по секунди.

Платформа	Почено решење на рачунару	BFP решење на FPGA
Радни такт [MHz]	3600	50
Време извршавања [s]	9.43	40.89
Количник [MĆ/s]	0.022	0.005

Табела 6 – Поређење почетних и крајњих резултата

Другим речима, за 36 пута већом фреквенцијом се остварује 22 пута бољи резултат. Ово указује да би употреба блоковски покретног зареза требала позитивно да се одрази на перформансе али је учинак мањи него што се приликом постављања хипотезе учинио.

Слика је мало јаснија када се заправо преброји број приступа меморија који се не мапира 1 на 1 у односу на укупно утрошену меморију. Иницијално решење има фактор 1 а број приступа меморији за за различите величине блока је представљен у следећој табели.

Величина блока	Релативан број приступа у односу на почетно решење
2 × 2	0,787
4 × 4	0,770
8 × 8	0,750
16 × 16	0,745
32 × 32	0,744

Табела 7 – Број приступа меморији у односу на почетно решење

Разлог зашто овај однос није близак укупној потрошњи меморије лежи у помоћним структурама које су неопходне за сваки блок. Будући да су за рачунање електричног и магнетног поља за сваки елемент Јиове мреже потребне ћелије које се налазе десно и испод, то није могуће за ћелије које се налазе на рубу блока илустровано на Слика 13. Из тог разлога је потребно дефинисати посебне структуре које иако саме по себи не заузимају пуно меморије, повећавају број приступа меморији.

5.3 Дискусија

И на самом крају се још једном подсећамо који је био почетни и основни мотив истраживања а то је другачији односно ефикаснији приступ организације података са циљем што мањег утрошка меморије. Мања потрошња меморије у Von Neuman-им архитектурама ће сигурно позитивно утицати на перформансе, будући да је приступ меморији знатно скупљи са временског аспекта него аритметичко-логичке операције процесора. Имајући то у виду, из симулација и резултата који су добијени са физичком платформом, може се установити да је хипотеза потврђена.

Поента овог истраживања није била да решење буде директно упоредиво и конкуретно доступним решењима на тржишту јер је још у раној фази истраживања установљено да то у овом тренутку није могуће. Главни разлог за то јесте пре свега широка распрострањеност графичких картица које су постале практично неизоставан део сваког десктоп рачунара, а често и преносивих. Изузетно јака конкуренција међу произвођачима је резултовала да графички процесори имају велику процесну моћ за релативно малу цену. Све док распрострањеност и доступност FPGA чипова не порасте, неће бити могуће направити упоредиво решење које ће моћи да прати графичке процесоре а да при том не буде неколико пута скупље. Самим тим, ово истраживање више служи као прототип и „одскочна даска“ за будућа истраживања на ову тему.

Што се самих резултата тиче, треба споменути неколико ствари. Прво и основно јесте да је однос уштеде меморије и прецизности задовољавајући. С

обзиром да је FDTD меморијски интензиван проблем, било какав напредак у смањењу потрошње меморије је пожељан. Одступање у прецизности од 5% на нивоу целе симулације је постављено као довољно добро (грешка је мања него приликом убацивања објекта) и постигнуто је за тражене параметре. Симулација је била релативно кратка и величина матрице је такође била мала, пре свега како би могла да се испроба и у FPGA чипу скромних могућности. Повећавањем броја итерација и димензије Јиове мреже се повећава и уштеда меморије. С обзиром да се рачуна грешка на нивоу целе матрице, ситне грешке у заокруживању се акумулирају. Оно што охрабрује јесте приказ у Табела 4 – Графички приказ разлике између референтног и решења које користи BFPгде се види да је на појединачним ћелијама разлика у најнеповољнијем случају око 0.5% што сматрамо да је задовољавајуће с обзиром на уштеду.

Када упоредимо почетно решење засновано на покретном зарезу и коначно решење засновано на блоковском покретном зарезу видимо мању укупну потрошњу меморије за скоро 50%. Када се ово провери и у самом броју приступа меморији, за посматрани случај димензија матрица 4*4, решење базирано на BFP има 23% мање приступа меморији од почетног решења.

Од тренутно доступних решења, најуспешнија и најзаступљенија су она базирана на графичким картицама. GPU-ови су изузетно моћне архитектуре које имају много ресурса и неприкосновен проток меморије. Самим тим имају и релативно велику потрошњу енергије, која може бити битан фактор у временима која долазе, с обзиром на тренутну ситуацију са енергетиком у свету.

Узевши то у обзир, чињеница да је цео FDTD алгоритам (изузев визуелизације) стао на једну изузетно скромну платформу као што је MAX1000 [71] је успех сам за себе. На платформи се налази MAX10 FPGA серије 10M16SAU169C8G, који спада у ситне и приступачне. У наредној табели су представљене његове карактеристике као и удео коришћених елемената.

/	Максимум	Искоришћено
Укупни логички елементи	15.840	3495 [22%]
Укупан број пинова	130	55 [42%]
Укупно меморијских бита	562.176	133.120 [24%]
Уграђених множача	90	0
Укупно PLL-ова	1	1
Укупно UFM блокова	1	0
Укупно ADC блокова	1	0

Табела 8 – Искоришћеност ресурса

Ова табела илуструје још једном предности FPGA технологије у њеној флексибилности и конфигурабилности, као и релативно великој количини меморије. У случају система који би непрекидно радио, разлика у потрошњи енергије може бити фактор који иако не прави суштинску разлику, може превагнути за случај евентуалних будућих бежичних решења која би се ослањала на батерије. Не треба у потпуности и занемарити чињеницу да графичке картице имају потрошњу која се мери стотинама вати [W] док је за мале FPGA чипове она реда величине око 20-30 вати. При том, из табеле 5 се може видети да је искоришћено мање од половине расположивих ресурса, па је са очекивати да је самим тим и потрошња још додатно мања. Ако причамо о апликацијама које би радиле непрестано или на преносивим уређајима, мања потрошња је сигурно нешто што је аргумент за овакав вид решења.

Интересантна опажања се могу направити на тему ефикасности FPGA у односу на GPU у овом конкретном примеру. Ако анализирамо однос броја уписа и читања из меморије у поређењу са бројем аритметичких операција (сабирања и множења) долазимо до закључка да добар проценат рада графичких процесора одлази у празно и не може се искористити у случају FDTD алгоритма.

Ако за нашу матрицу посматрамо величину блока 8 × 8, број уписа по итерацији је 7224, док је број читања из меморије 20152 то јест скоро три пута више. Оваква разлика је добар показатељ огромне количине података коју је неопходно прочитати за потребе рачунања. Када се у рачуницу убаце и саме

операције долази се до односа да је за сваки бајт потадака потребно 0.58 операција. Уколико променимо димензије блокова, оно што се мења јесте број читања и писања који је незнатно другачији, док број операција остаје исти, будући да се и даље мора проћи кроз сваки елемент матрице. Ако апроксимирамо количину операција GPU-а са покретним зарезом по секунди (FLOPS-ова - floating-point operations per second) са меморијским протоком, види се јаснија слика колико мало потенцијала графичких процесора се заправо користи. Илустроваћемо то на следећа два примера.

У случају сада већ застареле графичке картице AMD Radeon R7 265, произвођач тврди да остварује 1843.2 GFLOPS-а (милијарди операција у аритметици са покретним зарезом) што резултује са максималним теоријским протоком од 179.2 GB/s (гига бајта у секунди). Ако поделимо једну величину са другом, долазимо до броја од 10.28 FLOPS-а по бајту података што је 17 пута више него што је неопходно за алгоритам, пошто је установљено да алгоритам (барем у овој изведби) има око 0.6 операција по једном бајту података. Ситуација се додатно погоршава како идемо ка скупљим и моћнијим моделима. Ако узмемо за пример графички процесор GeForce GTX 1070, он постиже импресивних 7816.4 GFLOPS-ова и резулује протоком који превазилази 256 GB/s. Дељењем долазимо до односа од 30.52 FLOPS-ова по бајту података, што је преко 50 пута више него што је потребно за рад алгоритма. Ово иде у прилог хипотези да је ефикаснији начин потребан као и да је FDTD првенствено меморијски интензиван проблем.

Када на све то додамо да решење које користи BFP има готово упола мањи број приступа меморији уз нешто више рачунарских операција, видимо да се ефикасност додатно побољшава. Оно што остаје за будући рад јесте имплементација на неком јачем FPGA чипу у спрези са бржим меморијама.

Главна предност FPGA у овом случају јесте што је погодан за бит егзактне операције што резултује са врло мало „празног хода“. Ширина магистрала се може подесити да тачно одговара потребама алгоритма. Такође, подаци се могу читати у блоковима уместо појединачно како би све потребне ћелије биле учитане у једном приступу меморије. У случају озбиљнијег истраживања и прототипа би требало укључити меморије које имају још већи проток као што су DDR5 или HBM меморије.

Закључно, хипотеза је прошла потрврду концепта и отвара простор за скроз нова истраживања на ову тему. Сама измена представе података у BFP иако помаже, није довољна да самостално драстично побољша перформансе алгоритма да би решења базирана на процесорским језгрима у спрези са меморијом могла поново постати најбоља у одређеним апликацијама.

5.4 Будући рад

Што се тиче будућег рада и унапређивања решења, постоји неколико путања. Нулта тачка би била оптимизација самог алгоритма. Када је установљено да почетно решење на рачунару боље конфигурације даје слабија решења од оних већ описаних у научној заједници, јасно је да сама имплементација алгоритма може бити боља. Најједноставнији корак без много измена на самом алгоритму би била измена обраде матрице да буде онаква каква је погодна у имплементацији за C/C++ а то је по редовима, уместо по колонама.

Даље оптимизације на раду су могуће како би се извукао максимум перформанси из плоче. Ни у овом случају процесор није максимално оптерећен због непостојања кеш (енг. *cache*) меморије, те опет део времена проводи чекајући меморију да подаци пристигну. Првенствено се мисли на већи радни такт на коме би радио цео систем и који би се приближио максималном такту који може да поднесе SDRAM меморија. Нажалост, FPGA чип коришћен у овом истраживању има радни такт од 12 MHz који се са фазно закључаном петљом може подићи највише до 166 MHz, јер то представља горњу границу такта радне меморије. Уз одређене интервенције на меморијском контролеру, то би требало да резултује угрубо дупло мањем времену рачунања и самим тим максимално дупло бољим перформансама. Већ је установљено да би бољи однос цене и перформанси могао да се постигне са незнатно јачим XC7A15T-1FGG484C који има више ресурса и самим тим би могао истовремено радити у спрези са више других чипова.

Следећи корак који се природно намеће јесте пробати урадити додатну паралелизацију која би укључивала више развојних плоча које истовремено

обрађују различите сегменте Јиове мреже. Све ово би у паралели пратио и развој додатне програмске подршке која би обављала синхронизацију свега наведеног.

Алтернативни већи искорак би био уместо процесорског језгра, направити наменске множаче и аритметичко-логичке јединице које би биле специјализоване искључиво за рад на FDTD алгоритму. Ове измене би биле изузетно велике и захтевале би практично у потпуности нови систем, и вероватно и доста моћнији FPGA чип како би онда сва неопходна меморија могла да постоји унутар чипа. Ово решење сигурно не би било исплативије од доступних, али је извесно да би се боље перформансе могле постићи.

Литература

[1] A. N. Azmi, Y. Kamin, M. K. Noordin и A. N. M. Nasir, „Towards Industrial Revolution 4.0: Employers' Expectations,“ *International Journal of Engineering & Technology,* т. 7, pp. 267-272, 2018.

[2] K. Yee, „Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media,“ *IEEE Transactions on antennas and propagation,* pp. 302-307, 1966.

[3] Z. S. Sacks, D. M. Kingsland, R. Lee и J.-F. Lee, „A perfectly matched anisotropic absorber for use as an absorbing boundary condition,“ *IEEE transactions on Antennas and Propagation,* т. 43, бр. 12, pp. 1460-1463, 1995.

[4] S. Gedney, „An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices,“ *IEEE transactions on Antennas and Propagation,* т. 44, бр. 12, pp. 1630-1639, 1996.

[5] A. Taflove и S. C. Hagness, Computational Electrodynamics: The Finite-difference Time-domain Method, Artech House, 2000.

[6] M. Subotic, N. Pjevalica и L. Palfi, „Design and Modelling of an Enclosed Array of Square Spiral Antennas for Microwave Tomography,“ *Elektronika ir Elektrotechnika,* т. 23, бр. 2, pp. 47-53, 2017.

[7] S. C. Hagness, A. Taflove и J. E. Bridges, „Three-dimensional FDTD analysis of a pulsed microwave confocal system for breast cancer detection: Design of an antenna-array element,“ *IEEE Transactions on Antennas and Propagation,* т. 47, бр. 5, pp. 783-791, 1999.

[8] S. Aguilar, M. Al-Joumayly, M. Burfeindt, N. Behdad и S. Hagness, „Multiband miniaturized patch antennas for a compact, shielded microwave breast imaging array,“ *IEEE transactions on antennas and propagation,* т. 62, бр. 3, pp. 1221-1231, 2014.

[9] F. Zepparelli, P. Mezzanotte, F. Alimenti, L. Roselli, R. Sorrentino, G. Tartarini и P. Bassi, „Rigorous analysis of 3D optical and optoelectronic devices by the

compact-2D-FDTD method,“ *Optical and quantum electronics,* т. 31, бр. 9, pp. 827-841, 1999.

[10] P. Kosmas, Y. Wang и C. Rappaport, „Three-dimensional FDTD model for GPR detection of objects buried in realistic dispersive soil,“ *In Proc. SPIE,* т. 4742, pp. 330-338, 2002.

[11] S. Noghanian, A. Sabouni, T. Desell и A. Ashtari, Microwave Tomography: Global Optimization, Parallelization and Performance Evaluation, Springer, 2014.

[12] A. Ashtari, S. Noghanian, A. Sabouni, J. Aronsson, G. Thomas и S. Pistorius, „Using a priori information for regularization in breast microwave image reconstruction,“ *IEEE Transactions on Biomedical Engineering,* т. 57, бр. 9, p. 2197–2208, 2010.

[13] A. Sabouni, S. Noghanian и S. Pistorius, „A global optimization technique for microwave imaging of the inhomogeneous and dispersive breast,“ *Canadian Journal of Electrical and Computer Engineering,* т. 35, бр. 1, pp. 15-24, 2010.

[14] J. Mix, G. Haussmann, M. Piket-May и K. Thomas, „EMC/EMI design and analysis using FDTD. In Electromagnetic Compatibility,“ *1998 IEEE International Symposium,* т. 1, pp. 177-181, 1998.

[15] W. Chen, P. Kosmas, M. Leeser и C. Rappaport, „An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm,“ *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays,* pp. 213-222, 2004.

[16] J. Francés, S. Bleda, A. Márquez, C. Neipp, S. Gallego, B. Otero и A. Beléndez, „Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computing on FDTD scheme for solid and fluid vibration problems,“ *The Journal of Supercomputing,* т. 70, бр. 2, pp. 514-526, 2014.

[17] „OpenMP,“ [На мрежи]. Available: https://www.openmp.org/. [Последњи приступ 7 September 2020].

[18] J. Chi, F. Liu, E. Weber, Y. Li и S. Crozier, „GPU-accelerated FDTD modeling of radio-frequency field–tissue interactions in high-field MRI,“ *IEEE Transactions on Biomedical Engineering,* т. 58, бр. 6, p. 1789–1796, 2011.

[19] E. L. Tan, „Acceleration of lod-fdtd method using fundamental scheme on graphics processor units,“ *IEEE Microwave and Wireless Components Letters,* т. 20, бр. 12, p. 648–650, 2010.

[20] H. M. Waidyasooriya, M. H. Y. Takei и M. Kameyama, „Hybrid single/double precision floating-point computation on GPU accelerators for 2-D FDTD,“ *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA),* 2012.

[21] H. M. Waidyasooriya, Y. Takei, M. Hariyama и M. Kameyama, „Low-power heterogeneous platform for high performance computing and its application to 2D-FDTD computation: ERSA’12 short paper,“ *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA),* 2012.

[22] W. Chen, M. Leeser и C. Rappaport, Acceleration of the 3D FDTD Algorithm in Fixed-point Arithmetic using Reconfigurable Hardware, Northeastern University, 2007.

[23] R. N. Schneider, L. E. Turner и M. M. Okoniewski, „Application of FPGA technology to accelerate the finite-difference time-domain (FDTD) method,“ *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays,* pp. 97-105, 2002.

[24] H. Suzuki, R. Yamaguchi и S. Uebayashi, „A study on accuracy of FDTD method using fixed-point arithmetic,“ *IEICE Trans,* p. 37–41, 2005.

[25] J. Durbano, J. Humphrey, F. Ortiz, P. Curt, D. Prather и M. Mirotznik, „Hardware acceleration of the 3D finite-difference time-domain method,“ *IEEE Antennas and Propagation Society Symposium,* 2004.

[26] R. Schneider, M. Okoniewski и L. Turner, „A software-coupled 2D FDTD hardware accelerator [electromagnetic simulation],“ *IEEE Antennas and Propagation Society Symposium,* 2004.

[27] J. Backus, „Can Programming Be Liberated From the Von Neumann Style?: a Functional Style and Its Algebra of Programs.,“ *Communications of the ACM,* pp. 613-641, 1978.

[28] L. Chisvin и R. J. Duckworth, „Content-Addressable and Associative Memory: Alternatives to Theubiquitous RAM.," *Computer,* pp. 51-64, 1989.

[29] P. Placidi, L. Verducci, G. Matrella, L. Roselli и P. Ciampolini, „A custom VLSI architecture for the solution of FDTD equations," *IEICE Transactions on Electronics,* т. 85, бр. 3, p. 572–577, 2002.

[30] L. Verducci, P. Placidi, G. Matrella, L. Roselli, F. Alimenti, P. Ciampolini и A. Scorzoni, „A feasibility study about a custom hardware implementation of the FDTD algorithm," *Proc. of The 27th General Assembly of the URSI,* 2002.

[31] L. Verducci, P. Placidi, P. Ciampolini, A. Scorzoni и L. Roselli, „A standard cell hardware implementation for finite-difference time domain (FDTD) calculation," *IEEE MTT-S International Microwave Symposium Digest,* 2003.

[32] H. M. Waidyasooriya, T. Endo, M. Hariyama и Y. Ohtera, „OpenCL-Based FPGA Accelerator for 3D FDTD with Periodic and Absorbing Boundary Conditions," *International Journal of Reconfigurable Computing,* 2017.

[33] T. Kenter, J. Förstner и C. Plessl, „Flexible FPGA design for FDTD using OpenCL," *In Field Programmable Logic and Applications,* pp. 1-7, 2017.

[34] S. Pijetlović, M. Subotić и N. Pjevalica, „An approach to finding an optimal FPGA for memory intensive problems," у *4th IcETRAN international conference,* 2017.

[35] K. Kalliojarvi и J. Astola, „Roundoff errors in block-floating-point systems," *IEEE Transactions on Signal Processing,* т. 44, бр. 4, p. 783–790, 1996.

[36] D. Elam и C. Lovescu, „A block floating point implementation for an N-point FFT on the TMS320C55X DSP," Texas Instruments Application Report, 2003.

[37] E. Bidet, D. Castelain, C. Joanblanq и P. Senn, „A fast single-chip implementation of 8192 complex point fft," *IEEE Journal of Solid-State Circuits,* т. 30, бр. 3, p. 300–305, 1995.

[38] H. K. Boyapati, R. K. Elubudi, S. Ungati, S. Y. Chaudhari и M. Jain, „Transmit evm improvement of ofdm based wireless lan through rescaling and block floating point ifft implementation," *Procedia Computer Science,* т. 115, p. 635–642, 2017.

[39] Z. Song, Z. Liu, C. Wang и D. Wang, Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design, arXiv preprint, 2017.

[40] E. H. D'Hollander, „High-level synthesis optimization for blocked floating-point matrix multiplication," *ACM SIGARCH Computer Architecture News,* т. 44, бр. 4, p. 74–79, 2017.

[41] A. Oppenheim, „Realization of digital filters using block-floating-point arithmetic," *IEEE Transactions on Audio and Electroacoustics,* т. 18, бр. 2, p. 130–136, 1970.

[42] K. Bauer и P. Ralev, „Realization of block floating-point digital filters and application to block implementations," *IEEE Transactions on Signal Processing,* т. 47, бр. 4, p. 1076–1086, 1999.

[43] Y. F. Choo, B. L. Evans и A. Gatherer, Complex block floating-point format with box encoding for wordlength reduction in communication systems, arXiv preprint, 2017.

[44] K. W. v. Dongen и C. Brennan, Solution methods for electric field integral equations, 2012.

[45] Q. Fang, P. Meaney и K. Paulsen, „Viable three-dimensional medical microwave," *IEEE Transactions on Antennas,* т. 58, бр. 2, pp. 449-458, 2010.

[46] A. Fhager, S. Padhi и J. Howard, „3D image reconstruction in microwave tomography using an efficient FDTD model," *IEEE Antennas and Wireless Propagation Letters,* т. 8, p. 1353–1356, 2009.

[47] D. R. Gibbins, T. Henriksson, I. Craddock и M. Sarafianou, „Time-domain inverse scattering with a coarse reconstruction mesh," *IEEE Asia-Pacific Microwave Conference 2011,* p. 485–488 , 2011.

[48] T. Takenaka, H. Zhou и a. T. Tanaka, „Inverse scattering for a three-dimensional object in the time domain," *Journal of the Optical Society of America,* т. 20, бр. 10, p. 1867, 2003.

[49] A. Sabouni и S. Noghanian, „Experimental results for microwave tomography imaging based on FDTD and GA,“ *Progress In Electromagnetics Research M,* т. 33, pp. 69-82, 2013.

[50] A. Fhager, M. Gustafsson и S. Nordebo, „Image reconstruction in microwave tomography using a dielectric Debye model,“ *IEEE Transactions on Biomedical Engineering,* т. 59, бр. 1, p. 156–166, 2012.

[51] T. G. Papadopoulos и I. T. Rekanos, „Time-domain microwave imaging of inhomogeneous Debye dispersive scatterers,“ *IEEE Transactions on Antennas and Propagation,* т. 60, бр. 2, p. 1197–1202, 2012.

[52] M. Hajduković и Ž. Živanov, Arhitektura računara (pregled principa i evolucije), Novi Sad, 2011.

[53] IEEE, „754-2019 - IEEE Standard for Floating-Point Arithmetic,“ 2019.

[54] S. Pijetlović, M. Subotić и N. Pjevalica, „Optimizing FDTD Memory Bandwidth by Using Block Float-Point Arithmetic,“ *Elektronika Ir Elektrotechnika,* т. 24, бр. 8, pp. 32-37, 2018.

[55] „The Julia Programming Language,“ [На мрежи]. Available: https://julialang.org/. [Последњи приступ 7 November 2020].

[56] M. Lubin и I. Dunning, „Computing in operations research using Julia,“ *INFORMS Journal on Computing,* т. 27, бр. 2, pp. 238-248, 2015.

[57] J. Bezanson, A. Edelman, S. Karpinski и V. Shah, „Julia: A fresh approach to numerical computing. SIAM review,“ *Society for Industrial and Applied Mathematics,* pp. 65-98, 2017.

[58] C. Fieker, W. Hart, T. Hofmann и F.Johansson, „Nemo/Hecke: computer algebra and number theory packages for the Julia programming language,“ *Proceedings of the 2017 acm on international symposium on symbolic and algebraic computation,* pp. 157-164, 2017.

[59] „VisIt,“ Lawrence Livermore National Laboratory, [На мрежи]. Available: https://wci.llnl.gov/simulation/computer-codes/visit/. [Последњи приступ 7 November 2020].

[60] „Military Standard, Standard general requirements for electronic equipment,“ US Department of Defense, 1992. [На мрежи]. Available: http://everyspec.com/MIL-STD/MIL-STD-0300-0499/MIL-STD-454N_9160/. [Последњи приступ 7 November 2020].

[61] „IEEE Standard for Verilog Hardware Description Language,“ 7 April 2006. [На мрежи]. Available: https://ieeexplore.ieee.org/document/1620780. [Последњи приступ 2021].

[62] „Nios® II Processor Reference Guide,“ Intel, [На мрежи]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683836/current/introduction.html. [Последњи приступ 2023].

[63] „RISC V,“ [На мрежи]. Available: https://riscv.org/. [Последњи приступ 2021].

[64] K. Asanović и D. A. Patterson, „Instruction Sets Should Be Free: The Case For RISC-V,“ Electrical Engineering and Computer Sciences, University of California at Berkeley, 2014.

[65] C. Wolf, „picorv32,“ 2 March 2019. [На мрежи]. Available: https://github.com/cliffordwolf/picorv32. [Последњи приступ 2021].

[66] „Universal asynchronous receiver/transmitter,“ [На мрежи]. Available: https://www.nxp.com/docs/en/data-sheet/SCC2691.pdf#page=14. [Последњи приступ 2022].

[67] „Wishbone protocol,“ [На мрежи]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf. [Последњи приступ 2022].

[68] „Winbond W9864G6JT datasheet,“ Winbond, [На мрежи]. Available: https://www.winbond.com/hq/support/documentation/levelOne.jsp?__locale=en&DocNo=DA00-W9864G6JT. [Последњи приступ 2023].

[69] „MAX1000 User Guide,“ Trenz-electronic, [На мрежи]. Available: https://shop.trenz-electronic.de/trenzdownloads/Trenz_Electronic/Modules_and_Module_Carriers/2.5x6.15/TEI0001/User_Guide/MAX1000%20User%20Guide.pdf. [Последњи приступ 2023].

[70] S. Sutherland, The Verilog PLI Handbook, Springer, 2002.

[71] „MAX1000 User Guide,“ Trenz Electronic, [На мрежи]. Available: https://shop.trenz-electronic.de/trenzdownloads/Trenz_Electronic/Modules_and_Module_Carriers/2.5x6.15/TEI0001/User_Guide/MAX1000%20User%20Guide.pdf. [Последњи приступ 2023.].

[72] S. Pijetlovic, M. Subotic и N. Pjevalica, „Improving FDTD algorithm performance using block floating-point,“ *25th Telecommunication Forum (TELFOR)*, 2017.

[73] „Nios® II Processor Reference Guide,“ Intel, [На мрежи]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683836/current/introduction.html. [Последњи приступ 2023].

Биографија

Стефан Б. Пијетловић је рођен 21. јуна 1991. године у Новом Саду, где је завршио основну и средњу школу. Факултет техничких наука у Новом Саду је уписао 2010. године, студијски програм Рачунарство и аутоматика. Основне студије је завршио 2014. године на смеру Рачунарска техника и рачунарске комникације са просечном оценом 9.14 (девет и 14/100), одбранивши бечелор рад под темом „Једно решење онтологије за опис података у систему за добављање разнородног садржаја са интернета“. Исте године је уписао и мастер студије на Факултету техничких наука, смер Рачунарска техника и рачунарске комуникације. Мастер студије је завршио 2015. године са проечном оценом 10, одбранивши мастер рад под темом „Један приступ обједињавању алата за развој програмске подршке“. Докторске студије је уписао школске 2015/2016. године на истом факултету.

Истраживачки интерес Стефана Пијетловића је усмерен ка пројектовању и верификацији рачунарских система, дигиталном обрадом звука као и унапређењу процеса наставе.

Био је стипендиста Истраживачко-развојног института РТ-РК од 2012. до 2014. године, на коме се и запослио након завршетка основних студија. Шкослке 2015/2016. године је изабран у звање Асистент мастер на Факултету техничких наука и од тада активно учествује у извођењу наставе из предмета: Логичко пројектовање рачунарских система 1 и 2, Системска програмска подршка у реалном времену 1, Алгоритми дигиталне обраде звука и Паралелно програмирање.

Аутор је једног рада у међународним часописима, аутор или коаутор 9 радова на међународним конференцијама, 2 рада на националним конференцијама.

Од страних језика говори енглески и служи се немачким језиком.

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Предлог за уштеду меморијских ресурса у алгоритму коначних разлика у временском домену коришћењем аритметике у блоковском покретном зарезу
Назив институције/институција у оквиру којих се спроводи истраживање
а) **Факултет техничких наука, Универзитет у Новом Саду** б) **Институт РТ-РК, Нови Сад**
Назив програма у оквиру ког се реализује истраживање
Истраживање се реализује у оквиру израде докторске дисертације на студијском програму Рачунарство и аутоматика.
1. Опис података
1.1 Врста студије *Укратко описати тип студије у оквиру које се подаци прикупљају* **Докторска дисертација** 1.2 Врсте података а) **квантитативни** б) **квалитативни**

1.3. Начин прикупљања података

а) анкете, упитници, тестови

б) клиничке процене, медицински записи, електронски здравствени записи

в) генотипови: навести врсту _____

г) административни подаци: навести врсту _____

д) узорци ткива: навести врсту__ _____

ђ) снимци, фотографије: навести врсту_____

е) текст, навести врсту _____

ж) мапа, навести врсту _____

з) остало: описати **Рачунарске симулације**_____

1.3 Формат података, употребљене скале, количина података

1.3.1 Употребљени софтвер и формат датотеке:

а) Excel фајл, датотека _____

b) SPSS фајл, датотека _____

c) PDF фајл, датотека _____

d) Текст фајл, датотека _____

e) JPG фајл, датотека _____

f) Остало, датотека _____

1.3.2. Број записа (код квантитативних података)

а) број варијабли **Велик број**_____

б) број мерења (испитаника, процена, снимака и сл.) **Велик број** _____

1.3.3. Поновљена мерења

а) да

б) не

Уколико је одговор да, одговорити на следећа питања:

а) временски размак између поновљених мера је _____

б) варијабле које се више пута мере односе се на _____

в) нове верзије фајлова који садрже поновљена мерења су именоване као _____

Напомене: _____

Да ли формати и софтвер омогућавају дељење и дугорочну валидност података?

а) ***Да***

б) Не

Ако је одговор не, образложити _____

_____

2. Прикупљање података

2.1 Методологија за прикупљање/генерисање података

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) **експеримент**, навести тип **Рачунарске симулације** _____

б) корелационо истраживање, навести тип _____

ц) **анализа текста**, навести тип **Анализа доступне литературе**_____

д) остало, навести шта _____

2.1.2 Навести врсте мерних инструмената или стандарде података специфичних за одређену научну дисциплину (ако постоје).

_____

_____

2.2 Квалитет података и стандарди

2.2.1. Третман недостајућих података

а) Да ли матрица садржи недостајуће податке? Да **Не**

Ако је одговор да, одговорити на следећа питања:

а) Колики је број недостајућих података? _____

б) Да ли се кориснику матрице препоручује замена недостајућих података? Да Не

в) Ако је одговор да, навести сугестије за третман замене недостајућих података

_____

2.2.2. На који начин је контролисан квалитет података? Описати

Квалитет података је контролисан поређењем експерименталних и теоријских података_____

2.2.3. На који начин је извршена контрола уноса података у матрицу?

Контрола уноса података је изведена на основу експертског знања_____

_____

3. Третман података и пратећа документација

3.1. Третман и чување података

3.1.1. Подаци ће бити депоновани у **Репозиторијуму докторских дисертација на Универзитету у Новом Саду**.

3.1.2. URL адреса

_____

3.1.3. DOI

_____

3.1.4. Да ли ће подаци бити у отвореном приступу?

а) ***Да***

б) *Да, али после ембарга који ће трајати до _____*

в) *Не*

Ако је одговор не, навести разлог _____

3.1.5. Подаци неће бити депоновани у репозиторијум, али ће бити чувани.

Образложење

_____

_____

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? **Стандард који примењује Репозиторијум докторских дисертација Универзитета у Новом Саду**_____

3.2.1. Навести метаподатке на основу којих су подаци депоновани у репозиторијум.

_____

_____

Ако је потребно, навести методе које се користе за преузимање података, аналитичке и процедуралне информације, њихово кодирање, детаљне описе варијабли, записа итд.

_____

_____

_____

_____

3.3 Стратегија и стандарди за чување података

3.3.1. До ког периода ће подаци бити чувани у репозиторијуму?

_____

3.3.2. Да ли ће подаци бити депоновани под шифром? **Да** Не

3.3.3. Да ли ће шифра бити доступна одређеном кругу истраживача? **Да** Не

3.3.4. Да ли се подаци морају уклонити из отвореног приступа после извесног времена?

Да **Не**

Образложити

_____

_____

4. Безбедност података и заштита поверљивих информација

Овај одељак МОРА бити попуњен ако ваши подаци укључују личне податке који се односе на учеснике у истраживању. За друга истраживања треба такође размотрити заштиту и сигурност података.

4.1 Формални стандарди за сигурност информација/података

Истраживачи који спроводе испитивања с људима морају да се придржавају Закона о заштити података о личности (*https://www.paragraf.rs/propisi/zakon_o_zastiti_podataka_o_licnosti.html*) и одговарајућег институционалног кодекса о академском интегритету.

4.1.2. Да ли је истраживање одобрено од стране етичке комисије? Да **Не**

Ако је одговор Да, навести датум и назив етичке комисије која је одобрила истраживање

_____

4.1.2. Да ли подаци укључују личне податке учесника у истраживању? Да **Не**

Ако је одговор да, наведите на који начин сте осигурали поверљивост и сигурност информација везаних за испитанике:

а) Подаци нису у отвореном приступу

б) Подаци су анонимизирани

ц) Остало, навести шта

_____

_____

5. Доступност података

5.1. Подаци ће бити

*а) **јавно доступни***

б) доступни само уском кругу истраживача у одређеној научној области

ц) затворени

Ако су подаци доступни само уском кругу истраживача, навести под којим условима могу да их користе:

_____

_____

Ако су подаци доступни само уском кругу истраживача, навести на који начин могу приступити подацима:

_____

_____

5.4. Навести лиценцу под којом ће прикупљени подаци бити архивирани.

Ауторство - некомерцијално_____

6. Улоге и одговорност

6.1. Навести име и презиме и мејл адресу власника (аутора) података

Стефан Пијетловић
stefan.pijetlovic@live.com_____

6.2. Навести име и презиме и мејл адресу особе која одржава матрицу с подацима

Стефан Пијетловић
stefan.pijetlovic@live.com

6.3. Навести име и презиме и мејл адресу особе која омогућује приступ подацима другим истраживачима

Стефан Пијетловић stefan.pijetlovic@live.com