УНИВЕРЗИТЕТ У БЕОГРАДУ

ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Ненад Королија

УБРЗАВАЊЕ ИЗВРШАВАЊА ВРЕМЕНСКИ ЗАХТЕВНИХ СОФТВЕРСКИХ АПЛИКАЦИЈА КОНФИГУРИСАЊЕМ НАМЕНСКОГ ХАРДВЕРА У ВРЕМЕ ИЗВРШАВАЊА ПРОГРАМА НА ВИШЕПРОЦЕСОРСКИМ РАЧУНАРИМА

ДОКТОРСКА ДИСЕРТАЦИЈА

Београд, 2016.

UNIVERSITY OF BELGRADE

SCHOOL OF ELECTRICAL ENGINEERING

Nenad Korolija

ACCELERATING THE EXECUTION OF TIME CONSUMING SOFTWARE APPLICATIONS BY CONFIGURING SPECIAL HARDWARE DURING THE PROGRAM EXECUTION ON MULTIPROCESSOR COMPUTERS

Doctoral Dissertation

Belgrade, 2016

ПОДАЦИ О МЕНТОРУ И ЧЛАНОВИМА КОМИСИЈЕ

Ментор:

Др Вељко Милутиновић, редовни професор у пензији, Универзитет у Београду – Електротехнички факултет

Чланови комисије:

Др Милош Цветановић, доцент, Универзитет у Београду – Електротехнички факултет
Др Душан Старчевић, редовни професор, Универзитет у Београду – Факултет организационих наука
Др Захарије Радивојевић, доцент, Универзитет у Београду – Електротехнички факултет
Др Милан Поњавић, ванредни професор, Универзитет у Београду – Електротехнички факултет

Датум одбране: _____.

ЗАХВАЛОСТ

Желео бих да се захвалим ментору и комисији на залагању, спремности да ми посвете време и помогну у сваком тренутку и великом доприносу у изради ове дисертације из релативно нове области хардверско софтверског инжењерства. С обзиром да је тешко пронаћи експерте у овој области како у индустрији тако и у академским круговима, менторски рад на оваквом истраживању је представљао велики изазов. Посебну захвалност бих упутио др Јовану Поповићу који је активно учествовао у истраживањима, анализи идеја, али и давању нових идејама. Захвалио бих се и свим колегама на разумевању и подршци у писању докторске тезе. Коначно, захвалио бих се својој породици на подршци и времену којим су ми омогућили да завршим ово истраживање.

УБРЗАВАЊЕ ИЗВРШАВАЊА ВРЕМЕНСКИ ЗАХТЕВНИХ СОФТВЕРСКИХ АПЛИКАЦИЈА КОНФИГУРИСАЊЕМ НАМЕНСКОГ ХАРДВЕРА У ВРЕМЕ ИЗВРШАВАЊА ПРОГРАМА НА ВИШЕПРОЦЕСОРСКИМ РАЧУНАРИМА

Резиме

За разлику од рачунара који се заснивају на контроли тока (енг. *control-flow*), чији су процесори способни за обављање свих инструкција дефинисаних архитектуром рачунара, а од којих сваки у једном тренутку обавља највише неколико инструкција, код рачунара заснованих на протоку података се хардвер конфигурише тако да се просторно распореде компоненте од којих је свака у стању да изврши само инструкцију за коју је предвиђена. Извршавање се своди на проток података кроз такав хардвер. Главне одлике овакве архитектуре рачунара су већа проточност података и смањена потрошња електричне енергије. Иако хардверске архитектуре рачунара засноване на протоку података постоје деценијама, технологија је тек недавно омогућила њихово равноправно коришћење са рачунарима заснованим на контроли тока, чиме проблем распоређивања послова између хардвера заснованог на протоку података и конвенционалних процесора све више добија на значају. Неке од временски захтевних апликација већи део времена извршавања проводе у цикличном понављању истих операција. Уколико су те итерације међусобно независне, или се могу довести у такав облик, онда је њихово извршавање погодно обавити употребом реконфигурабилног хардвера и парадигме засноване на протоку података.

Ова теза описује постојеће метеде и предлаже нове за прављање распореда извршавања послова на оваквим архитектурама рачунара у циљу побољшања перформанси, при чему су само неке од апликација погодне за убрзавање коришћењем реконфигурабилног хардвера и парадигме засноване на протоку података. Предлажу се и временско и просторно дељење реконфигурабилног

хардвера од стране конвенционалних процесора. Прављење распореда извршавања нити спада у NP2 класу сложености, а време прављења распореда се сматра режијским временом, па су описани алгоритми који користе хеуристику и траже најбоље распореде само до одређене дубине. Резултати потврђују могућност убрзавања извршавања апликација коришћењем овакве архитектуре рачунара и указују на услове под којима се апликације могу убрзати.

Кључне речи: Хардвер заснован на протоку података, убрзавање извршавања апликација, прављење распореда извршавања послова

Научна област: Техничке науке – Електротехника и рачунарство

Ужа научна област: Софтверско инжењерство

УДК: 621.3

ACCELERATING THE EXECUTION OF TIME CONSUMING SOFTWARE APPLICATIONS BY CONFIGURING SPECIAL HARDWARE DURING THE PROGRAM EXECUTION ON MULTIPROCESSOR COMPUTERS

Abstract

In contrast to control-flow computer architectures, whose processors are capable of executing all instructions defined by the architecture, while each processor executes only up to few instructions simultaneously, hardware dataflow architectures are based on configuring hardware by spreading components capable of executing one instruction each over the surface. Computation is based on dataflow through the hardware. Main characteristics of this architecture are higher data throughput and reduced power consumption. Some of the computation demanding applications spend most of the execution time in iterating over the same set of instructions. Although hardware dataflow architectures exist for decades, due to the technology limitations, they have became valuable for executing such applications only recently. Therefore, the problem of scheduling jobs on dataflow hardware and conventional processors becomes increasingly important. Some of the computation demanding applications spend most of the execution time in executing for loops. If iterations are mutually independent, or if they can be transformed in such a form, then these applications are suitable for executing on dataflow hardware.

This thesis presents available methods for creating schedules for this kind of architectures in order to reduce total execution times, and proposes new ones. Sharing the dataflow hardware in both time and space is proposed. Scheduling jobs on this architecture belongs to the NP problem class and scheduling time is considered as an overhead, so the algorithms use heuristics and search possible combinations of jobs only up to appropriate depth. Results confirm that this architecture can reduce total execution time and reveal the conditions under which the acceleration is possible.

Keywords: Dataflow hardware, accelerating application execution, creating schedules

Scientific field: Technical science – Electrical engineering and computer science

Specific scientific field: Software engineering

UDK: 621.3

САДРЖАЈ

ИНДЕКС СЛИКА

ИНДЕКС ТАБЕЛА

СПИСАК СКРАЋЕНИЦА

ALU	Arithmetic Logic Unit
CFD	Computational Fluid Dynamic
CLB	Configurable Logic Block
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random-Access Memory
DFEs	DataFlow Engines
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FPU	Floating Point Unit
FPGA	Field-Programmable Gate Array
GFLOPs	Giga-FLoating-point Operations Per Second
ILP	Instruction Level Parallelism
I/O	Input/Output
IOBs	Input-Output Blocks
LBM	Lattice-Boltzmann Method
NS	Navier-Stokes
PCIe	Peripheral Component Interconnect express
PE	Processing Element
RAM	Random-Access Memory
RAW	Read-After-Write
RC	Reconfigurable Computing
ROCCC	Riverside Optimizing Compiler for Configurable Computing
SYS	The Lam and Mostow's method
VHDL	VHSIC language - Very high speed integrated circuit Hardware Description Language

VLSI	Very-Large-Scale Integration
XST	Xilinx Synthesis Technology
WAW	Write-After-Write

1. Увод

Број транзистора процесора се годинама повећавао у складу са Муровим законом, дуплирајући се на по две године. Значајан удео у томе имало је смањивање растојања између компоненти. Транзистори су достигли величину која не може значајно да се смањи без нарушавања њихових карактеристика. Имајући у виду од колико транзистора су данашњи процесори направљени, свака промена излаза из кола за задати улаз на лошије би могла да изазове битно повећање вероватноће да произведен чип неће функционисати као што је предвиђено. Самим тим се лимитира густина паковања транзистора на чипу. Са друге стране, потрошња електричне енергије, па самим тим и загревање су приближно сразмерни квадрату фреквенције, чиме се лимитира повећање фреквенције процесора. Таласна дужина брзине светлости при фрефвенцији од 3GHz је приближно 10cm. Димензије данашњих процесора се приближавају тој величини.

Услед свега тога, даље напредовање процесора се претежно заснива на повећању броја језгара процесора. На тај начин се омогућава паралелно извршавање већег броја инструкција, чиме се у општем случају постиже већа брзина извршавања апликација. Међутим, овим се намећу нови проблеми. Један од главних, који је одавно присутан, је програмирање алгоритама који ће се извршавати на више процесорских јединица, било да су то језгра процесора, чворови кластера или рачунари повезани мрежом. Иако овај проблем некад захтева веће ангажовање програмера него што је то био случај раније, преводиоци помажу у тзв. паралелизацији програма, чиме је за програмера често сакривена логика потребна за обезбеђивање паралелног извршавања инструкција.

Проблем за који није познато да може у општем слчају да се реши приликом извршавања програма на више процесорских јединица је потреба за комуникацијом процесорских јединица. Када се алгоритам извршава на више процесорских јединица скоро онолико пута брже него на једној процесорској јединици колико има процесорских јединица, онда се тај алгоритам сматра скалабилним. За остале алгоритме, потребна је честа комуникација између

процесорских јединица приликом извршавања алгоритма, па се након повећавања броја процесорских јединица изнад одређене вредности за тај алгоритам брзина извршавања не повећава, већ се смањује. Ово се дешава услед тога што је потребно више времена да се пренесу подаци од једне процесорске јединице до друге и прикупе резултати, него да се процесирају на процесорској јединици.

Један од могућих начина решавања овог проблема је промена парадигме. У поређењу са конвенцијалним, рачунарима заснованим на контроли тока (енг. *control-flow*), тзв. рачунари засновани на протоку података су у стању да извршавању на једном чипу знатно већи број операција паралелно. Постоје хардверска и софтверска парадигма заснована на протоку података. Код софтверске се прави ток података за алгоритам, након чега се инструкције шаљу на извршавање или у редове за чекање. Свака аритметичко логичка јединица процесора на бази контроле тока извршава инструкције, докле год постоје у реду за ту јединицу. Код хардверске парадигме, конструисани ток података се трансформише у хардвер, тако што се хардвер конфигурише за тај алгоритам. Овај хардвер се најчешће јавља у оквиру реконфигурабилног хардвера који се обично састоји од FPGA (енг. *Field Programmable Gate Array*). Уместо постојања релативно велике површине чипа потребне за подршку извршавању свих инструкција предвиђених архитектуром рачунара, затим кеш меморије и разне друге механизме, за сваку инструкцију се генерише површина чипа потребна да прими потребне улазне податке, изврши само ту инструкцију и проследи излаз до дела хардвера предвиђеног за наредну инструкцију која зависи од резултата ове.

Смањивањем површине чипа неопходне за извршавање сваке од инструкција, смањује се време потребно за комуникацију између инструкција, али и потрошња електричне енергије. Процењује се да цена електричне енергије потрошене током животног века процесора може лако да надмаши цену самог процесора. Сходно томе, може се очекивати сличан однос и код FPGA. Многи од временски захтевних алгоритама се могу убрзати коришћењем парадигме засноване на протоку података [1-9], уз истовремено смањење укупне потрошње електричне енергије.

Још један важан аспект савремених рачунарских система је хлађење процесора, што је од недавно постало један од главних проблема приликом дизајнирања

организација рачунара намењених за извршавање апликација које захтевају велику количину процесирања. Приближно пропорционално смањењу потрошње електричне енергије, смањује се и загревање процесора.

Иако је фреквенција оваквог хардвера обично за ред величине мања него код конвенционалних процесора, паралелним извршавањем далеко већег броја инструкција се добијају боље перформансе. Услед тога, као и чињенице да је употреба хардверске парадигме засноване на протоку података (енг. *hardware dataflow*) тек од недавно постала економски оправдана, ова парадигма представља значајан допринос савременој рачунарској техници.

На жалост, описана парадигма је погодна само за одређене алгоритме. Зато се реконфигурабилни хардвер јавља у рачунарима или серверима уз конвенционални процесор или процесоре. Не само да апликације погодне за убрзавање коришћењем реконфигурабилног хардвера садрже делове који нису погодни за овакав хардвер, већ је и за саму иницијлизацију потребан процесор.

Да би конфигурисање хардвера за извршавање одређеног скупа инструкција било исплативо, потребно је да се извршавање инструкција понавља довољно велик број пута. Алгоритми који су погодни за хардвер заснован на протоку података се обично састоје од петљи које се понављају релативно велик број пута, а код којих су итерације независне. Код неких алгоритама ово није случај, али се могу трансформисати у такав облик.

1.1. Проблеми у прављењу распореда извршавања послова

Главни проблем описаних реконфигурабилних архитектура рачунара је што су у стању да извршавају релативно мали број алгоритама конкурентно са конвенционалним процесорима, па самим тим углавном могу да извршавају само део апликације, услед чега се овакав хардвер обично користи заједно са конвенционалним процесором или процесорима. Као што је већ речено, чак и ако ће се оваква архитектура рачунара користити само за извршавање алгоритма који

се могу убрзати употребом реконфигурабилног хардвера, за иницијализацију је потребан конвентионални процесор. У општем случају, оваква реконфигурабилна архитектура рачунара која поседује хардвер заснован на протоку података, поред апликација погодних за реконфигурабилни хардвер може да извршава све програме које и конвенционалне архитектуре рачунара могу да извршавају. Дакле, могуће је извршавати део програма на конвенционалном процесору, а затим део програма који се своди на извршавање алгоритма погодног за реконфигурабилни хардвер извршити на оваквом хардверу. Исто тако, могуће је паралелно извршавати више алгоритама, од којих се неки извршавају употребом реконфигурабилног хардвера, а неки употребом конвенционалног процесора.

Упоредо са напретком рачунара, потребе за извршавањем све већег броја инструкција у јединици времена су се повећавале. Претпоставља се да ће овај тренд и даље постојати и да ће захтевни алгоритми, који се данас користе готово искључиво за дуготрајна извршавања ради симулације динамике флуида, временске прогнозе и слично, бити коришћени и у персоналним рачунарима.

1.2. Предмет и циљ истраживања

Предмет ове тезе је ефикасно прављење распореда извршавања послова на реконфигурабилном хардверу који се састоји од конвенционалних процесора и реконфигурабилног хардвера заснованог на протоку података. Циљ је убрзавање извршавања програма на реконфигурабилном хардверу заснованом на протоку података који поседује своју меморију и конвенционалне процесоре. У скуп програма ће бити укључене и апликације које имају релативно много извршавања међусобно независних итерација. Биће приказан поступак прављења распореда извршавања послова помоћу две предложене методе. Оно што разликује предложене алгоритме распоређивања извршавања послова (у овом случају програма, односно апликација) од већине доступних из отворене литературе је чињеница да су послови који се извршавају коришћењем хардвера заснованог на протоку података дуготрајнији од већине послова који се извршавају на

конвенционалним процесорима, услед чега се толерише релативно велико време прављења распореда. Још једна од специфичности је релативно мали број послова који се истовремено може извршавати на хардверу заснованом на протоку података. Резултати указују на могућност убрзавања извршавања апликација коришћењем предложене архитектуре и организације рачунара и откривају неопходне услове да би се убрзање постигло.

Очекивани допринос се односи на побољшање перформанси целокупног система, а према унапред дефинисаним критеријумима.

Разматра се реконфигурабилни хардвер заснован на FPGA (енг. *Field Programmable Gate Array*) који поседује своју меморију. Количина меморије је, сходно намени, пожељно да буде знатно већа него код конвенционалних процесора. Упоредо са развојем конвенционалних процесора и меморија, напредује и реконфигурабилни хардвер, па није неуобичајено да количина меморије коју реконфигурабилни хардвер поседује буде и за ред величине већа од оне којом процесор директно располаже.

За време превођења апликација ће бити потребно припремити (превести) и конфигурационе фајлове за реконфигурабилни хардвер, конфигурациони фајл операција које ће бити потребно вршити употребом реконфигурабилног хардвера, распоред компоненти, као и њихове везе на процесорским јединицама, тј. DFEs (енг. *DataFlow Engines*). Овакви фајлови се добијају превођењем VHDL (енг. *VHSIC Hardware Description Language*) фајлова којима се описује хардвер.

Једном припремљен реконфигурабилни хардвер се обично користити на два начина. Први подразумева да се упоредо са слањем улазних података (нпр. великих низова) из главне меморије којом процесор располаже, на самом хардверу врши рачунање и шаље излаз назад у главну меморију. Други начин подразумева да се пре почетка рачунања на сам реконфигурабилни хадвер копирају потребни подаци (нпр. велики низови и матрице), а затим да се покрене извршавање, да би се након завршене обраде резултати вратили назад у главну меморију. Први начин је погоднији код алгоритама који захтевају релативно много рачунања за сваку *n*-торку низова, при чему сваки елемент фигурише само једном у рачунању (нпр.

множење вектора скаларом). Пример алгоритма погодног за други начин коришћења је симулација у којој се више пута итерира кроз једну или више матрица. У том случају, брже је подацима током понављања итерација приступати из меморије која је, не само ближа процесорским јединицама реконфигурабилног хардвера, већ и оптимизована тако да могу упоредо по 6 или више меморијских локација да се читају и прослеђују процесорским јединицама истовремено.

Уколико се улазни подаци прослеђују реконфигурабилном хардверу упоредо са њиховом обрадом на самом хардверу и слању резултата назад, процесорска моћ реконфигурабиног хардвера најчешће није уско грло система, већ комуникација између процесора и реконфигурабилног хардвера.

Уколико се улазни подаци копирају у меморију реконфигурабиног хардвера пре почетка извршавања, а затим се покрене обрада података, да би се на крају вратио резултат, уско грло система је најчешће само процесирање. За то време, PCIe (Pheripheral component interconnect express) магистрала остаје неискоришћена.

У овој тези се предлаже и просторно и временско дељење реконфигурабилног хардвера ради свеукупног повећања перформанси. Да би истовремено извршавање више послова било могуће, потребно је припремити конфигурационе фајлове (VHDL) за реконфигурабилни хардвер за одговарајуће комбинације послова.

Како је реконфигурабилни хардвер намењен извршавању временски захтевних апликација, број послова који ће се на њему истовремено извршавати ће најчешће бити релативно мали (један или два). Имајући ово у виду, као и чињеницу да реконфигурабилни хардвер није погодан за извршавање већине алгоритама које данашњи рачунари извршавају, очекивано је да број ових комбинација неће бити превише велик да би један савремен рачунар могао да направи распоред. Један од начина да се број комбинација послова лимитира је дефинисање унапред задатих алгоритама који се на реконфигурабилном хардверу могу извршити комбиновањем са другим алгоритмима из овог скупа. За све остале алгоритме, било би неопходно извршавати их независно од других алгоритама, баш као што је то и сада случај, када је извршавање послова на реконфигурабилном хардверу у питању. Други начин је да се једноставно корисницима или односно онима који апликације праве,

омогући да бирају који послови се са којима могу комбиновати. У овој тези се претпоставља постојање спремних конфигурационих фајлова за реконфигурабилни хардвер пре почетка извршавања послова.

У овој тези неће бити разматрани послови за реконфигурабилни хардвер који се периодично понављају. Дакле, овакви послови ће бити распоређивани у тренутку када се за време прављења распореда јаве као спремни за извршавање.

1.3. Основне хипотезе

Истраживање описано у овој тези се заснива на следећим хипотезама:

- Брзина процесора, мерена у броју инструкција у секунди које процесори могу да обављају се деценијама повећавала у складу са Муровим законом. Имајући у виду могућност да се процесирање убрза тако што се уместо повећања фреквенција процесора повећава број језгара, као и могућности да се употребом хардвера намењеног искључиво за одређене апликације убрза њихово извршење, за очекивати је да ће се овај тренд наставити. Самим тим, за очекивати је и да ће захтеви за процесирањем бити већи.

- Како је реконфигурабилни хардвер погодан за убрзавање извршавања одређених алгоритама, могуће је убрзати извршавање скупа програма који укључују извршавање оваквих алгоритама коришћењем конвенционалних процесора и реконфигурабилног хардвера. Ово истраживање се односи на убрзавање извршавања програма када је процесору доступан реконфигурабилни хардвер заснован на FPGA, који поседује сопствену меморију.

- Реконфигурабилни хардвер има мању фреквенцију од конвенционалних процесора. Самим тим, извршавање једне инструкције процесора на бази контроле тока употребом реконфигурабилног хардвера је спорије. Иако је ово лимитирајући фактор, када је употреба реконфигурабилног хардвера у питању, треба имати у виду да реконфигурабилни хардвер омогућава обраду већег броја инструкција паралелно, услед чега је ипак могуће постићи већу проточност података.

- Рачунар може да извршава различите алгоритме, од којих су само неки погодни за убрзавање коришћењем реконфигурационог хардвера.

- Посао (програм односно апликација) погодан за реконфигурабилни хардвер се може извршавати било на конвенционалном процесору или на реконфигурабилном хардверу. Остали послови се могу извршавати само на конвенционалним процесорима.

- Време извршавања послова за реконфигурабилни хардвер је унапред познато и то и у случају употребе конвенционалног процесора и у случају употребе реконфигурабилног хардвера. Реконфигурабилни хардвер често служи за извршавање временски захтевних алгоритама код којих је количина рачунања функција количине улазних података. Уколико време извршавања алгоритма који се може убрзати коришћењем реконфигурабилног хардвера није познато, али се извршавање алгоритма може поделити на више независних позива алгоритма познатог времена извршавања, могуће је да се извршавање оваквог алгоритма третира као циклично извршавање алгоритма. Тиме се, у зависности од тога да ли је извршавање целог алгоритма завршено или не, може поново позвати алгоритам за прављење распореда, како би се алгоритму који се може убрзати коришћењем реконфигурабилног хардвера омогућило да се и даље извршава на реконфигурабилног хардверу или не, у зависности од уског грла система у тренутку доношења одлуке.

- Програми су написани тако да се делови програма који се може убрзати могу извршити или на конвенцоналном (control-flow) процесору, без употребе реконфигурабилног хардвера, или позовом реконфигурабилног хардвера уз чекање резултата извршавања. На овај начин, за програме написане у програмским језицима за које је направљена библиотека за коришћење реконфигурабилног хардвера, могуће је да се део програма који није могуће убрзати коришћењем реконфигурабилног хардвера изврши на самом процесору, док се преостали део програма може извршити било на реконфигурабилном хардверу или на процесору. Избор ће бити направљен у зависности од тога шта је у тренутку доношења одлуке уско грло система, а што ће се утврдити на основу одговора који програм добије од алгоритма за прављење распореда извршавања послова.

- Постоје унапред припремљени (компајлирани) конфигурациони фајлови, не само за послове који се на реконфигурационом хардверу могу извршити, већ и за одговарајуће комбинације послова. Тиме се омогућава да више процесора који извршавају програме који се могу убрзати употребом реконфигурабилног хардвера могу да деле реконфигурабилни хардвер, уколико се алгоритмом за прављење распореда извршавања послова утврди да је овако нешто оправдано. Припремање оваквих реконфигурационих фајлова је могуће аутоматизовати, па ће у овој тези бити дат само кратак осврт на овај проблем.

- Укупно време извршавања свих програма се може смањити комбиновањем извршавања послова на конвенционалним процесорима и реконфигурабилном хардверу, и то толико да постојање реконфигурабилног хардвера буде економски оправдано.

- Извршавање послова уз просторно дељење реконфигурабилног хардвера може убрзати извршавање у односу на извршавање послова уз искључиво временско дељење тог хардвера.

- Алгоритам за прављење распореда извршавања послова је довољно брз да би се смањило укупно трајање извршавања послова и алгоритма за прављење распореда толико да постојање реконфигурабилног хардвера буде оправдано.

Претпоставља се да је могућа уштеда електричне енергије, па самим тим и свеукупног загревања, употребом предложеног начина распоређивања послова.

1.4. Методе истраживања

Први корак овог истраживања представља сагледавање могућих организација рачунара које би се могле употребити за убрзавање извршавања алгоритама. Један од основних захтева који се поставља је да реконфигурабилни хардвер поседује барем онолико меморије колико обично имају рачунари истих година производње. Потребно је да конфигурисање хардвера може да се обави уз помоћ алата, тако да инжењер не мора да познаје детаље имплементације реконфигурабилног хардвера.

Пожељно је да одговарајући алат (преводилац или окружење) нуди интерфејс у виду рутина које се позивају из неког од доступних програмских језика.

Да би се сагледале могућности имплементације апликација предвиђених за архитектуре рачунара засноване на контроли тока на архитектурама рачунара заснованих на протоку података, биће описане методе трансформације израза и алгоритама из разних области, применљиве на задат проблем. Након тога, биће описани алати који олакшавају или елиминишу потребу за програмером у процесу превођења апликација. Након избора одговарајућег алата који из претходно описаних метода покрива што већи скуп решења проблема који се пред програмера постављају, биће дат пример имплементације Lattice-Boltzmann алгоритма, добијеног превођењем из парадигме засноване на контроли тока у парадигму засновану на протоку података, уз анализу убрзања и уштеде електричне енергије.

Затим би се анализирали програми који се користе за тестирање перформанси вишепроцесорских система, уз разматрање могућности убрзавања програма коришћењем претходно дефинисаних архитектура и организација рачунара.

У оквиру истраживања ће бити извршена евалуација постојећих метода прављења распореда извршавања послова. Утврдиће се и потенцијални узроци лоших особина појединих метода тако што ће се идентификовати основни недостаци које методе генерално имају, као и које од ових недостатака која од метода има.

Кључни део истраживања представља дефинисање алгоритама прављења распореда извршавања послова који се заснивају на временско-просторном дељењу реконфигурабилног хардвера.

У реалном времену ће се вршити процена исплативости (могућности убрзања) хардверске реализације појединих послова које је могуће извршити и на конвенционалном процесору и на реконфигурабилном хардверу, при чему би у разматрање био узет само унапред задат број послова. У случају позитивне процене, анализираће се колике су могућности свеукупног убрзавања извршавања послова уколико би се тај посао убрзао коришћењем реконфигурабилног хардвера. Прављење распореда извршавања послова спада у НП класу сложености. Време

прављења распореда извршавања послова се сматра режијским временом. Услед овога није увек могуће испитати све могуће комбинације послова које се на реконфигурабилни хардвер могу истовремено сместити. Зато ће бити коришћена хеуристика која омогућава да се знатно мањим бројем покушаја смештања послова на реконфигурабилни хардвер добију приближно исте перформансе. То се планира тако што ће се тражење распореда извршавања послова обављати само до одређене дубине, при чему ће у обзир бити узети послови за које је највише за очекивати да ће допринети смањењу укупног времена извршавања свих послова.

Биће имплементиран одговарајући симулатор у програмском језику C++, помоћу којег ће бити експериментално проверена времена извршавања послова у зависности од алгоритма за прављење распореда.

1.5. Очекивани научни допринос

У овој тези ће бити представљене архитектуре рачунара које користе реконфигурабилни хардвер ради убрзавања извршавања алгоритама. Затим ће бити дата класификација и поређење алгоритама распоређивања послова доступних у литератури.

Очекивани научни допринос представљају два нова алгоритма за временско просторну расподелу послова на реконфигурабилном хардверу.

Поред овога, биће дати и услови под којима се укупно време извршавања послова може смањити довољно да се употреба реконфигурабилног хардвера може сматрати исплативом.

Један од доприноса ће бити и симулатор извршавања послова на претпостављеној архитектури рачунара имплементиран у програмском језику C++, који се релативно лако може прилагодити и искористити у другим областима примене.

1.6. Структура рада

У наредној глави ће бити дат преглед постојећих решења проблема из области превођења алгоритама и програма, односно апликација из парадигме базиране на контроли тока у парадигму базирану на протоку података. Биће дат кратак преглед коришћења реконфигурабилног хардвера. Након тога ће бити представљени неки од програма који се обично извршавају на оваквим архитектурама рачунара.

Затим ће бити описана доступна решења у области прављења распореда послова на посматраним архитектурама и организацијаа рачунара. Биће дат опис предложеног система и начина прављања распореда, а затим предложена два алгоритма за просторну и временску расподелу послова на хардвер заснован на протоку података.

У следећој глави ће бити дати резултати извршавања програма на реконфигурабилном хардверу. Најпре ће бити описан пример на којем се виде разлике у направљеном распореду извршавања унапред задатог скупа послова у зависности од изабраног алгоритма, а затим и времена извршавања синтетички генерисаних послова на основу послова који се обично извршавају коришћењем посматране архитектуре и организације рачунара.

Након тога ће бити упоређени резултати извршавања програма на реконфигурабилном хардверу са резултатима извршавања програма на другим рачунарским архитектурама, као и анализа и поређење предложеног алгоритма за прављење распореда извршавања послова са претходно описаним алгоритмима из отворене литературе. Биће процењена исплативост употребе реконфигурабилног хардвера у зависности од врста послова и односа количина послова који се могу убрзати коришћењем реконфигурабилног хардвера и оних који се обављају коришћењем конвенционалних процесора.

На крају ће бити дат закључак и даљи правци истраживања.

2. Употреба реконфигурабилног хардвера

У овој глави ће најпре бити дат кратак осврт на употребу реконфигурабилног хардвера. Затим ће бити дат преглед доступних технологија из ове области, имајући у виду циљ овог истраживања. Као резултат ће бити изабран одговарајући реконфигурабилни хардвер који се може употребити у десктоп рачунару, који поседује сопствену меморију и који је програмабилан употребом одговарајуће екстензије неког од најчешће коришћених програмских језика.

2.1. Увод

Рачунарство се практично од свог оснивања заснива на Von Neumann парадигми. Сама архитектура рачунара функционише тако што процесор извршава унапред задат скуп инструкција који се назива програм или софтверска апликација. До појаве механизама којима се омогућава извршавање више инструкција паралелно, извршавање инструкција је текло на следећи начин. Прво би се са адресе тренутне инструкције читала, а затим и декодовала инструкција. Након дохватања података неопходних за извршење инструкције, аритметичко логичка јединица би по потреби извршила тражену операцију, након чега би се резултат извршавања сместио на одговарајуће место. Овакав приступ се сматра веома флексибилним, јер је практично после било које инструкције могуће извршити било коју другу инструкцију.

Инструкције су се у почетку дефинисале директно помоћу нула и јединица. То је био машински језик, који су први рачунари прихватали као улаз. Усавршавањем рачунара, јавила се већа потреба за рачунањем, па самим тим и за програмским језиком који би програмирање учинио једноставнијим. Коришћењем асемблера, програмер је могао симболима да зада инстукцију коју је потребно извршити, као и операнде инструкције. Појавом програмских језика као што су Pascal и C, омогућено је да се програмер сконцентрише на решавање проблема на вишем

нивоу апстракције, пишући комплетне изразе које је било потребно извршити, док су се преводиоци оваквих програмских језика бринули о претварању таквих израза у машински код. Даљим усавршавањем рачунара и програмских језика, дошло је до развоја објектно оријентисаног приступа, где се по први пут уводи концепт објекта. Од тада је програмер могао да пише функције на нивоу објеката, чиме се додатно подигао ниво апстракције при решавању проблема употребом рачунара. Примери оваквих језика су C++, Java, C#, Python, али и многобројни функционални језици.

Како је могуће да се једна апликација састоји из више целина које је могуће независно превести, компајлиран изворни код се повезује тзв. линкером, чиме се добија апликација спремна за покретање.

Оперативни системи су у стању да извршавају више покренутих апликација мењајући контекст довољно често да се ствара привид да се оне истовремено извршавају. У позадини, свака од апликација се извршава у одређеним интервалима које је оперативни систем одредио. За време извршавања, захваљујући механизмима за извршавање по неколико инструкција паралелно, у сваком тренутку је могуће да се, у зависности од ограничења, извршава између једне и неколико инструкција.

Брзина извршавања инструкција неког процесора се мери у броју инсткрукција које је процесор у стању да изврши у трајању од једне секунде. До недавно се бризина процесора повећавала повећавањем фреквенције процесора и смањивањем величине компонената на чипу, чиме се омогућава паковање већег броја транзистора на исту површину силицијума.

Услед приближавања фреквенције процесора граници изнад које прегревање постаје превелико, као и смањења величине транзистора близу границе код које би карактеристике транзистора могле бити угрожене, развијени су мултипроцесорски системи. Код оваквих система, могуће је да сваки од процесора извршава посебну апликацију. Међутим, могуће је и да се више процесора користи за извршавање једног јединог алгоритма. Као проблем се намеће комуникација између процесора.

Да би се решио проблем комуникације између јединица које извршавају инструкције, потребно је смањити растојање између њих. Један од начина на који је ово могуће решити је смањивање површине чипа неопходног за извршење инструкција. Данашњи процесори се заснивају на раду многобројних стручњака који траје већ неколико деценија. Самим тим, може се рећи да су данашњи процесори у великој мери оптимизовани по питању величина које поједине функционалности заузимају на чипу.

Са друге стране, површина која је потребна за извршавање једне инструкције је далеко мања. Дакле, могуће је направити хардвер ког кога ће ток извршавања програма бити такав да свака инструкција свој излаз прослеђује директно инструкцијама чије извршавање од тог излаза зависи. Овакви чипови се називају ASICs (енг. *Application-Specific Integrated Circuits*) и најчешће се праве употребом специјалних језика за опис хардвера, као што су Verilog и VHDL (VHSIC Hardware Description Language), који се преведу у FPGA (енг. *Field Programmable Gate Array*).

Иако овакав приступ обезбеђује већу брзину извршавања, конфигурисање оваквог хардвера би трајало далеко више него само извршавање употребом конвенционалних процесора. Зато је овакав приступ погодан једино уколико је одређени скуп инструкција потребно извршавати циклички много пута.

Још један од проблема који се јавља је могућност реконфигурације оваквог хардвера, како би био употребљив за извршавање више од једног програма. Употребом FPGA, могуће је реконфигурисати овакав хардвер. Цена употребе FPGA је мања фреквенција хардвера.

Међутим, главна предност оваквог реконфигурабилног хардвера је што је независне итерације петље програма, које највише доприносе укупном трајању извршавања програма, могуће обављати паралелно. Ово се постиже тако што ће се улаз за прву итерацију проследити делу хардвера задуженом за извршавање прве инструкције, а затим, чим се извршавање прве инструкције заврши и резултат проследи делу хардвера задуженом за следећу инструкцију у низу, проследити улаз за другу итерацију делу хардвера задуженом за извршавање прве

инструкције. На тај начин, могуће је да се истовремено користе делови хардвера задужени за сваку од инструкција итерације. Тиме је могуће извршавати истовремено онолико инструкција колико их има у итерацији петље. Ова парадигма се назива парадигма заснована на протоку података.

Како је у данашње време могуће извршавати алгоритам и на хиљаду и више процесора са једне графичке картице, приступ ових процесора меморији постаје уско грло система. Убрзавање магистрале података само помаже, али главни проблем остаје. Што више процесорских јединица приступа заједничким подацима, то је већа шанса да меморија постане уско грло система приликом извршавања програма. Самим тим, и поред свих унапређења конвенционалних процесора, изгледа да се оптимизација оваквих архитектура рачунара приближава својим границама.

Иако је од давнина било могуће направити наменски хардвер предвиђен за извршавање унапред задатог програма, услед технолошких ограничења, примена оваквих решења дуго није била исплатива.

Овакви системи се обично користе за извршавање временски захтевних апликација, код којих се одређени број инструкција циклички понавља. При томе део инструкција који се не понавља, или бар не довољно пута да би га било исплативо имплементирати на хардверу, извршава конвенционални процесор.

Већина програмера није обучена за програмирање апликација које користе реконфигурабилни хардвер да би се убрзало њихово извршвање. Зато су произвођачи оваквог хардвера направили програмске језике који овај процес чине једноставнијим. Тако се на пример, употребом специјалних програмских језика може дефинисати шта је то што ће реконфигурабилни хардвер извршавати, док се употребом библиотека намењених за коришћење тог хардвера, уместо извршавања програма у једном од програмских језика једноставно позива реконфигурабилни хардвер да изврши одређени део програма.

Реконфигурабилни хардвер се може јавити у виду картице која се повезује са матичном плочом рачунара употребом PCIe магистрале. Како се овакав хардвер обично користи за убрзавање извршавања временски захтевних апликација, јавила

се потреба за коришћење више оваквих картица за извршавање једног програма. Услед тога, направљени су кластери чији чворови поседују одређен број конвенционалних процесора и реконфигурабилни хардвер.

За распоређивање послова на оваквом кластеру задужен је одговарајући софтвер за прављење распореда извршавања послова. Уобичајено, сваки корисник добија одговарајућу процесорску моћ у трајању одговарајућег временског периода за који је платио коришћење ресурса. На нивоу једног корисника, апликација корисника се брине о распоређивању послова на чворове који су му додељени.

У овој тези се разматра проблем распоређивања послова више апликација које деле заједнички реконфигурабилни хардвер.

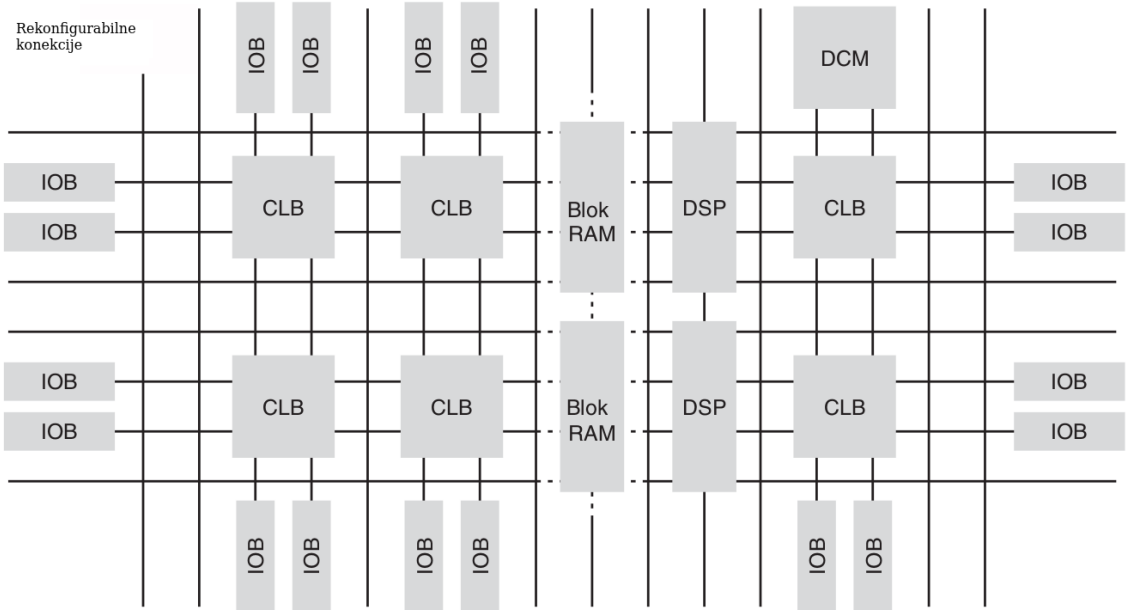
2.1. Преглед реконфигурабилног хардвера

У овој глави ће бити представљена решења из области реконфигурабилног хардвера која би се могла делити од стране више процесора ради убрзавања извршавања алгоритама. За свако од решења ће бити узете у обзир основне функционалности као што су: основне карактеристике реконфигурабилног хардвера које укључују фреквенције на којима ради, програмабилност и количина меморије коју хардвер поседује.

2.1.1. Xilinx

Један од најзаступљенијих произвођача реконфигурабилног хардвера је Xilinx. Преко четвртине века је Xilinx FPGA платформа била главни избор за дизајнирање реконфигурабилних система. Захваљујући флексибилности, коришћена је као веома флексибилна алтернатива за интегрална кола специфична за одређене апликације (ASIC).

Поред флексибилности, Xilinx је нудио могућност велике паралелне обраде коришћењем FPGA архитектура. Њихов FPGA се састоји од великог низа подесивих логичких блокова (CLBs), блокова за дигиталну обраду сигнала (DSP), RAM блокова и улазно / излазних блокова (IOBs), као што је приказано на слици 1.



Слика 1: Блок дијаграм FPGA високог нивоа.

CLBs и DSP, слично као и процесорска аритметичка логичка јединица (ALU), могу се програмирати да обављају аритметичке и логичке операције као што су додавање, множење, одузимање и слично.

За разлику од процесора, у којем је ALU архитектура фиксна и дизајнирана за опште намене, CLBs се могу програмирати тако да поседује само операције које су потребне апликацији, чиме се побољшавају перформансе.

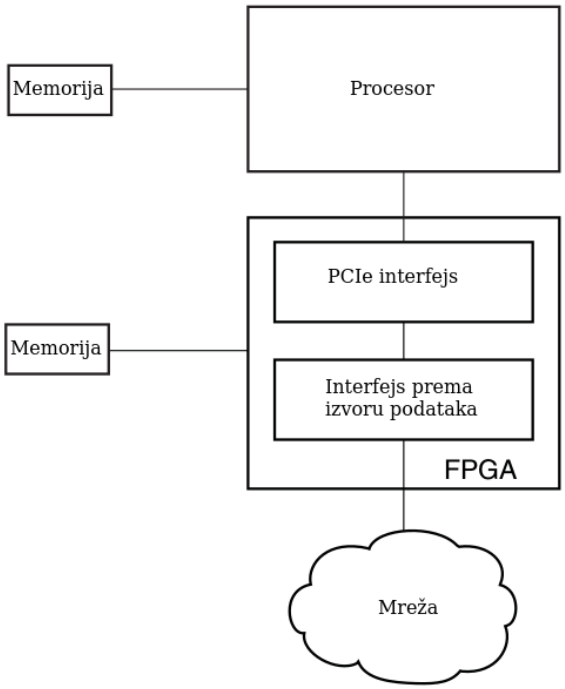
Стандардан процес израде подразумева дизајнирање система, верификовање коректности извршавања, а затим синтетизовање коришћењем Xilinx алата (XST).

Уграђени рачунари високих перформанси су рачунари који су укључени у опрему или апарате за обављање специфичних задатака интензивне обраде података и интензивног рачунања. У таблели 1 је дан сажетак различитих намена уграђених рачунара у индустрији за које се Xilinx решења користе.

Таβела 1: Апликације уграђених уређаја високих перформанси.

Индустрија	Апликација или опрема
Одβрана	Формирање снопа радара
Мултимедија	Компресија слике
Комуникације	Енкрипција података
Медицина	Рендеровање слика скенера
Финансијски сервиси	Процесирање података великом βрзином ради аутоматског трговања на βерзама

На слици 2 је приказан пример повезивања процесора са FPGA и спољњим системима. У овом случају је изаβрана мрежа. Треβа имати у виду да је могуће да мрежа βуде сенсорска мрежа или βило који други извор података. Као што се са слике види, процесор и FPGA имају засеβне меморије, а комуникација се оβавља преко PCIe интерфејса. Ово је једна од могућих имплементација, која се примењује у десктоп рачунарима.



Слика 2: Блок повезивања процесора и FPGA са спољњим системима.

2.1.2. Maxeler

Maxeler је релативно нова фирма, која се бави дизајном, имплементирањем и програмирањем реконфигурабилног хардвера. Она производи реконфигурабилни хардвер у виду PCIe картица за десктоп рачунаре, али и за кластере.

Као основу овог истраживања, одабрана је Maxeler технологија из следећих разлога:
1) она омогућава коришћење језика MaxJ, који је високи програмски језик налик на Јаву, намењен за програмирање језгара (енг. *kernels*) који ће бити извршаван на хардверу заснованом на протоку података.

2) она организује комуникацију између процесора и хардвера заснованог на протоку података, уз једноставан кориснички интерфејс.

DFEs су већ комерцијално реализовани од стране разничитих произвођача. У овој тези, биће употребљен MaxCompiler преводилац, који омогућава програмирање хардвера заснованог на протоку података и MaxelerOS оперативни систем, који је задужен за организовање ресурса на картици заснованој на протоку података. Дакле, на основу језгра написаног у програмском језику MaxJ, могуће је употребом Maxeler технологије конфигурисати одговарајућу FPGA слику и на одговарајући начин користити ресурсе. Maxeler технологија такође омогућава прављење менаџера, који се користи за повезивање више кернела, као и повезивање кернела са процесором и самим хардвером заснованом на протоку податакаом. Ово се остварује једноставним дефинисањем конекција између именованих конектора сваке од компонената које укључују: кернел, процесор и хардвер заснован на протоку података. Очигледно, у случају комбиновања кернела две или више апликација, менаџер би једноставно могао да повеже одговарајуће компоненте сваке од апликација независно, док би конекције од процесора биле опционално вођене кроз демултиплексер, а конекције ка процесору биле вођене преко мултиплексера.

Процесорска моћ, како конвенционалних процесора, тако и хардвер заснован на протоку података расте великом брзином. Услед тога, приказане имплементације

алгоритама употребом Maxeler технологије ће бити представљене на различитим генерацијама Maxeler рачунара. У табели 2 ће бити приказане различите имплементације Maxeler хардвера заснованог на протоку података.

Табела 2: Поређење различитих генерација Maxeler картица заснованих на протоку података.

Критеријум	2009: MaxBox+MAX2C	Лето 2010: MAX3	Крај 2011: MAX4
Технологија израде силицијумске плоче	65nm	40nm	32nm
MaxCard процесорска снага по процесорској јединици	1,920 LC	3,800 LC	7,600 LC
MaxCard меморија по процесорској јединици	96GB	192GB	384GB
DRAM пропусна моћ	15GB/s	39GB/s	51GB/s
Локална, тзв. On-chip меморија	2.25MB	4.67MB	9.4MB
Потрошња електричне енергије	417W	550W	680W

Једна од важних карактеристика окружења које је развио Maxeler је доступност веб стране на којој се могу програмирати Maxeler апликације [10].

2.2. Пример употребе реконфигурабилног хардвера

Као што је већ речено, апликација која понавља независне инструкције релативно велики број пута, може се убрзати тако што се такве инструкције имплементирају употребом FPGA. За остале инструкције, конвенционалан процесор је погоднији, јер остварује већу брзину извршавања, уз мању потрошњу електричне енергије. Самим тим, погодно је имплементирати апликацију тако што ће се инструкције које се могу убрзати употребом реконфигурабилног хардвера имплементирати директно на FPGA, док ће се остале извршавати на процесору. Блок дијаграм Maxeler хардвера може бити идентичан ономе са слике 2. Самим тим, подаци са процесора до FPGA се могу преносити употребом PCIe магистрале. Иако ова магистрала има релативно велику пропусну моћ, она и даље није добрих перформанси као меморија. Зато је пожељно пренети податке у меморију реконфигурабилног хардвера, како би се минимизовао проток кроз PCIe магистралу, па самим тим и убрзало извршавање апликације.

На слици 3 је дат један пример кернела програма који рачуна усредњену суму елемената низа. Резултујући низ на позицији *i* треба да има средњу вредност елемената на позицијама *i-1*, *i* и *i+1*. Као што се са слике види, најпре је потребно иницијализовати параметре и дефинисати тзв. хардверске променљиве (HWVar) које репрезентују улаз. Кернел је задужен за израчунавање једне вредности. Подаци са процесора се као ток прослеђују кернелу, како би сви подаци били обрађени. Функција *offset* враћа претходну или следећу вредност елемента низа, у зависности од вредности задатог параметра. У овом примеру, може се видети и начин на који се дефинише бројач. Условно извршавање је реализовано употребом тернарног оператора. Коначно, потребно је резултат повезати са излазом кернела, који ће бити видљив са стране процесора, односно главног програма.

```
package chap01_overview.ex2_movingaverage;

import com.maxeler.maxcompiler.v1.kernelcompiler.Kernel;

import com.maxeler.maxcompiler.v1.kernelcompiler.KernelParameters;
```

```
import com.maxeler.maxcompiler.v1.kernelcompiler.types.base.HWVar;

public class MovingAverageOverviewKernel extends Kernel {

	public MovingAverageOverviewKernel(KernelParameters parameters, int N) {

		super(parameters);

		HWVar x = io.input("x", hwFloat(8, 24)); // Input

		HWVar x_prev = stream.offset(x, -1); // Data

		HWVar x_next = stream.offset(x, 1);

		HWVar cnt = control.count.simpleCounter(32, N); // Control

		HWVar sel_nl = cnt > 0;

		HWVar sel_nu = cnt < N-1;

		HWVar sel_m = sel_nl & sel_nu;

		HWVar prev = sel_nl ? x_prev : 0;

		HWVar next = sel_nu ? x_next : 0;

		HWVar divisor = sel_m ? constant.var(hwFloat(8, 24), 3) : 2;

		HWVar sum = prev+x+next;

		HWVar result = sum/divisor;

		io.output("y", result, hwFloat(8, 24));

	}

}
```

Слика 3: Кернел примера усредњене суме

На слици 4 је дат изворни код програма написаног у језику C++, који позива претходно описани кернел за рачунање усредњене суме елемената низа. Као што се из примера види, најпре се иницијализују менаџер и кернел, након чега се

постављају подаци за кернел, а затим се стартује кернел који рачуна усредњене вредности. По потреби се може направити функција *checkStream* која би проверавала коректност изврачунатих вредности.

```
package chap01_overview.ex2_movingaverage;

import com.maxeler.maxcompiler.v1.kernelcompiler.Kernel;

import com.maxeler.maxcompiler.v1.managers.standard.SimulationManager;

public class MovingAverageOverviewSimRunner {

	public static void main(String[] args) {

		SimulationManager m = new

			SimulationManager("MovingAverageOverviewSim");

		Kernel k = new

			MovingAverageOverviewKernel(m.makeKernelParameters(), 6);

		m.setKernel(k);

		m.setInputData("x", 1, 5, 6, 7, 2, 0);

		m.setKernelCycles(6);

		m.runTest(); m.dumpOutput();

		double[] expected = { 3, 4, 6, 5, 3, 1 };

		double[] got = m.getOutputDataArray("y");

		//checkStream(expected, got);

		m.logMsg("stream is correct!");

	}

}
```

Слика 4: Главни програм примера усредњене суме

На овом примеру могуће је видети употребу реконфигурабилног хардвера за униформну обраду података, при чему се конвенционални процесор употребљава за иницијализацију података. Наравно, да би употреба реконфигурабилног хардвера убрзала извршавање програма, потребна је већа количина обраде, како би слање податка и дохватање резултата краће трајало од саме обраде на процесору. Поред овога, потребно је да се обрађује довољно велика количина података, како би реконфигурација била оправдана.

3. Паралелизација извршавања апликација употребом парадигме засноване на протоку података

У поређењу са архитектурама рачунара заснованим на контроли тока, архитектуре засноване на протоку података обично нуде боље перформансе за апликације које се употребљавају на рачунарима високих перформанси (енг. *High performance computing*). Осим тога, архитектуре засноване на протоку података троше мање електричне енергије. Међутим, тек од недавно, технологија омогућава развој архитектура рачунара заснованих на протоку података које су конкурентне са архитектурама заснованим на контроли тока. Са тачке гледишта програмирања, постоји релативно мали број искусних програмера за архитектуре засноване на протоку података. Као резултат, постоји потреба за превођењем алгоритама написаних за архитектуре засноване на контроли тока у одговарајуће архитектуре засноване на протоку података [11].

Фокус овог поглавља је на проналажењу и употреби метода из различитих области, које би се могле применити на превођење алгоритама из парадигме засноване на контроли тока у парадигму засновану на протоку података, затим поређење постојећих алата за превођење програма имајући у виду претходно описане методе, и коначно, оцењивање убрзања и смањења потрошње електричне енергије на примеру имплементације Lattice-Boltzmann метода за архитектуру засновану на протоку података. Резултати показују релативно велико убрзање (један до два реда величине), а у исто време, значајно смањење потрошње електричне енергије.

3.1. Увод

У протеклим деценијама, велика већина рачунара била је заснована на тзв. control-flow парадигми. Сходно томе, програмери су учили како да реше проблеме на основу пројектовања алгоритама којима се контролише у ком тренутку се ком

податку приступа, шта се са њиме ради и где се смешта резултат операције. Како су рачунари напредовали, постајали су у стању да реше све више и више сложених проблема. Главни проблем у програмирању је постало пројектовање алгоритама који ефикасно користе рачунарске ресурсе, нпр. ефикасним коришћењем кеш меморије, као и коришћењем главне оперативне меморије уместо чврстих дискова, кад год је то могуће.

Уброзо, паралелизација на различитим нивоима је остварена, што је резултовало у појави нових парадигми, као што су: векторски процесори, pipelining, super-pipelining, супер-скалар, вишејезгарни процесори (енг. *multicore*) и рачунари са великим бројем процесора. У међувремену, фокус програмера се променио од оптимизације алгоритама за једно-језгарне процесоре, преко оптимизације на нивоу нити, а затим и на нивоу језгра, на оптимизацију извршавања програма на нивоу захтева. Иако је тренд је да се постепено оптимизују алгоритми на све вишим нивоима апстракције, оптимизовање комуникације у дистрибуираном рачунарском систему и даље представља главни проблем.

Не само да овај проблем не нестаје, већ и расте током времена. Повећање такта рачунара је ограничено технологијом. Због тога, хиљаде процесора могу бити доступни у једном данашњем рачунару. Упркос повећању протока ка меморији, пренос података из главне меморије процесора и процесора, као и између више процесора и даље представља уско грло система. Још један технолошки проблем које расте у важности је хлађење процесора. Недавно, потрошња електричне енергије рачунара је постао један од главних проблема у рачунарским системима високих перформанси. Процењује се да цена електричне енергије коју процесор једног кластера потроши годишње, може лако прећи цену самог процесора [12]. Стога, чини се да је оптимизација архитектура предвиђених за алгоритме засноване на контроли тока достиже своје границе. Једно од могућих решења за описане проблеме је прелазак на парадигму засновану на протоку података [13, 14]. Треба напоменути да мењање парадигме није редак случај. Од појаве рачунара, то се десило много пута. Сведоци смо преласка са директног уношења машинских инструкција на програмирање у програмском језику асемблер, а затим и у вишим програмским језицима, од процедуралних до објектно оријентисаних

језика и функционалних програмских језика. Чини се да програмирање архитектура заснованих на протоку података, нудећи супериорне перформансе са смањеном потрошњом енергије, постаје све чешће.

Постоје софтверска и хардвера реализација парадигме засноване на протоку података. Софтверска подразумева низ процесних елемената, који су процесори засновани на контроли тока, где у датом тренутку, сваки од њих или извршава инструкцију, или чека на извршење. Главни циљ је извршавање што више независних инструкција истовремено. Међутим, недостатак софтверске парадигме засноване на протоку података је исти као код процесора заснованих на контроли тока, па самим тим није у фокусу ове тезе. Стога, хардверска ће бити подразумевана приликом објашњавања концепата архитектура и организација рачунара заснованих на протоку података.

Процесор заснован на протоку података, односно хардвер, ради на принципима парадигме засноване на протоку података. У основи би се могао посматрати као збир велике количине релативно малих површина чипа, од којих је свака у стању да извршава једну инструкцију, прима улаз и прослеђује излаз на оне делове процесора који извршавају зависне инструкције. Док је процесор заснован на контроли тока способан да извршава све врсте инструкција дефинисане архитектуром (и стога има релативно велику површину чипа), иако извршава само неколико инструкција истовремено, процесор заснован на протоку података се може конфигурисати да изврши одређени алгоритам на такав начин да се оствари проток података кроз њега.

Једном конфигурисан, хардвер заснован на протоку података је у стању да почне са извршавањем алгоритма чим се за тиме јави потреба. Међутим, када се јави потреба да хардвер извршава други алгоритам, потребно је да се реконфигурише. Због потребе за реконфигурацијом процесора заснованог на протоку података и ограничења технологије, процесор заснован на протоку података треба да ради на нижој фреквенцији од данашњих процесора заснованих на контроли тока. Ипак, процесор заснован на протоку података може да има већи проток инструкција, јер не укључује сложене механизме као што су кеш меморије и бафере за претварање адреса (eng. translation lookaside buffer, TLB), остављајући више простора на

процесору за извршавање. Конфигурацијом процесора заснованог на протоку података може да се извршава и на хиљаде инструкција истовремено [15], чиме се постижу супериорне перформансе.

Поред високог протока инструкција, процесори засновани на протоку података троше мање енергије. Потрошња струје је приближно пропорцијална квадрату фреквенције, а процесори засновани на протоку података често имају за ред величине ниже фреквенције. Исто тако, процесори засновани на протоку података не садрже претходно поменуте сложене механизме који захтевају снагу, већ се дисипација концентрише на извршавање инструкција.

Важна особина процесора заснованих на протоку података имплементираних помоћу FPGA је да могу да прилагоде хардвер извршавању других алгоритама у току извршавања програма [16]. Међутим, пошто је промена процесора сама по себи много спорија од извршења инструкција, процесори засновани на протоку података су ефикасни само за апликације које захтевају извршење великог скупа истих инструкција. Чак и такве апликације обично укључују инструкције за иницијализацију података које се не могу ефикасно обрадити помоћу хардвера заснованог на протоку података. Стога, процесори засновани на протоку података се обично комбинују са процесорима заснованим на контроли тока. Ова врста архитектуре позната је као хардвер заснован на протоку података [17].

Овај концепт није нов. Током 1970-тих и 1980-их година, ограничења у технологији резултовала су у развоју архитектура заснованих на протоку података ближих традиционалним рачунарима [18]. Парадигма позната као Reconfigurable computing (RC) комбинује хардвер који може да се реструктурира да спроведе специфичне функције за апликације и конвенционални процесор или процесоре. Као резултат тога, апликација високих перформанси се може убрзати у односу на извршење користећи процесоре засноване на контроли тока [19, 20]. Динамички реконфигурабилни системи прилагођавају своју унутрашњу структуру у току рада. Иако овај концепт постоји од 1960-их, постао је комерцијално доступан тек недавно. Према неким ауторима [1], тренд је да рачунари треба да имају и компоненте засноване на контроли тока и компоненте засноване на протоку података.

Упркос чињеници да би процесори засновани на контроли тока могли да побољшају перформансе у рачунарству високих перформанси (енг. *High performance computing*) [2, 3], они још увек нису уобичајени у пракси. Ово нас доводи до истраживања главних фактора трошкова рачунарских кластера, као и самог њиховог рада. Цена процесора у кластерима може се упоредити са ценом потрошене електричне енергије за својих неколико година. Због тога смањење потрошње електричне енергије може бити валидан разлог за промену инфраструктуре.

Решавање технолошких проблема није довољно. Већина апликација је писана за процесоре засноване на контроли тока, који се концептуално разликују од процесора заснованих на протоку података. Пре него што би такве апликације могле бити извршаване на процесорима заснованим на протоку података, морале би да се прилагоде. Иако је извршавање инструкција коришћењем парадигме засноване на протоку података брже него помоћу парадигме засноване на контроли тока још одавно, проблем трансформације изворног кода програма је игнорисан у време док су се фреквенције процесора заснованих на контроли тока скоро удвостручивале на сваке две године, што је процесоре засноване на контроли тока чинило константно у предности у односу на процесоре засноване на протоку података. Са повећањем популарности архитектура заснованих на протоку података јавила се и потреба за истраживање доступних начина за трансформисање графова контроле тока или изворног кода на одговарајућу репрезентацију у виду протока података.

Ретко који алгоритам се састоји само од инструкција које се понављају изнова и изнова. За друге инструкције (које обично представљају највећи део изворног кода, али учествују у веома малом делу укупног времена извршења), конвенционални процесори су погоднији, због виших фреквенција извршавања у поређењу са процесорима заснованим на протоку података. Због тога, већина инструкција се може и даље извршити на конвенционалном процесору, док би само делови апликације (оне инструкције које се извршавају у току великог дела укупног времена извршавања) морали да се адаптирају.

Апликације које се извршавају у сличним окружењима деценијама су добро оптимизоване и тестиране. Неке од њих су одговорне за важне прорачуне, где би појава грешака довела до великих финансијских губитака. Због тога, превођење апликација је јако скупо, па је самим тим од кључног значаја дизајнирање алата који ће помоћи у превођењу кода за парадигму засновану на протоку података.

У оквиру ове тезе, биће дат преглед расположивих метода из области које се односе на рачунарство засновано на протоку података (област systolic arrays, област анализе протока података, и област преводиоца), који се могу користити за трансформацију алгоритама заснованих на контроли тока у алгоритме засноване на протоку података. Све методе ће бити представљене на јединствен начин, како би њихово директно поређење било могуће. Сваки метод ће прво бити описан, а затим приказан користећи слику са истом структуром као и за све друге методе. Доступни алати за компајлирање изворних кодова за хардвер заснован на протоку података биће разматрани у односу на претходно описане методе. Даље, резултати поређења једне апликације користећи парадигму засновану на протоку података и одговарајуће апликације засноване на контроли тока ће бити представљени. У следећем одељку разматра се проблем преласка из парадигме засноване на контроли тока на парадигму засновану на протоку података. Затим ће бити представљене основне разлике између парадигме засноване на контроли тока и парадигме засноване на протоку података. Након тога ће бити представљене изабране методе за трансформацију алгоритама заснованих на контроли тока у алгоритме засноване на протоку података, а у истом делу, разматране методе трансформисања наслеђене из сродних области. Коришћење ових метода у трансформацији Lattice-Boltzmann алгоритма заснованих на контроли тока у парадигму засновану на протоку података ће затим бити представљено. Након поређења перформанси на примеру Lattice-Boltzmann алгоритма биће дати и резултати других истраживача који су имплементирали различите алгоритме коришћењем сличних технологија. Коначно, биће дати закључци и прогнозе о будућности система заснованих на протоку података, као алтернатива или допуна за системе засноване на контроли тока.

3.2. Опис проблема

Као што је већ речено, прелазак са парадигме засноване на контроли тока на парадигму засновану на протоку података може да побољша перформансе апликација које обављају интензивна израчунавања [1, 2, 3, 15, 20]. Да би се то постигло, преводиоци за архитектуре засноване на протоку података би требало да буду у стању да преводе апликације писане за архитектуре засноване на контроли тока. О неким од принципа овог приступа ће бити речи у овом одељку. Други приступ је да програмери прилагоде своје апликације. Већи део изворног кода постојећих програма је посвећен рачунарима заснованим на контроли тока. Такође, програмери се углавном уче како да решавају проблеме помоћу рачунара заснованих на контроли тока. Као резултат тога, преписивање свих апликација високих перформанси би било скупо, склоно грешкама, и дуготрајно. Ово истраживање се фокусира на представљање алата и метода за програмере да прилагоде своје апликације.

Једно од могућих решења за проблем се састоји у развију преводиоца који би били у стању да покрену апликације писане за парадигму засновану на контроли тока на хардверу заснованом на протоку података. То би захтевало да преводилац утврди који делови апликације су погодни за хардвер заснован на протоку података. Чини се да у најгорем случају преводилац не би био у стању да пронађе било који део апликације погодан за хардвер заснован на протоку података. Претпоставимо да желимо да дизајнирамо преводилац. Као што је већ поменуто, делови кода који се више пута узастопно извршавају су потенцијално погодни за хардвер заснован на протоку података. На пример, петља која извршава међусобно независне итерације може бити трансформисана у хардвер заснован на протоку података. Ова трансформација се може постићи користећи исте принципе који се могу наћи код преводиоца за архитектуре засноване на протоку података. Преводилац за архитектуре засноване на протоку података може да трансформише језгро програма написаног у програмском језику вишег нивоа у одговарајућу репрезентацију у виду тока података. Кернел би се састојао од тих инструкција које формирају петљу. Ове инструкције би се извршавале помоћу хардвера

заснованог на протоку података. Осим дефинисања улаза и излаза кернела, сама логика можда не би требала уопште да се мења. Главни проблем је што би извршавање *for* петљи са свега неколико итерација било много спорије користећи хардвер заснован на протоку података, јер би хардвер заснован на протоку података био само делимично искоришћен (само неколико инструкција би могле да се изврше паралелно).

Могуће решење проблема недовољног искоришћења хардвера заснованог на протоку података због малих димензија проблема је складиштење броја итерација у датотеку дневника (енг. *log*) у току извршавања програма, као и коришћење ове информације за рекомпајлирање апликације. Преводилац може да ради скоро као типични преводиоци, осим што би могао ставити инструкције за прављење дневника на почетку сваке петље. Исто тако, на самом почетку, он би морао да провери да ли је дати број извршавања апликација за потребе обуке већ урађен, и у том случају поново превести изворну апликацију. Током рекомпилације, петље за које би се утврдило да нису погодне за хардвер заснован на протоку података могле би се обрадити превођењем на конвенционалан начин, без инструкција за евидентирање. Друге петље, које имају међусобно независне итерације, могле би да се трансформишу у одговарајуће репрезентације у виду тока података. Међутим, чување броја итерација у време извршавања намеће бројне проблеме. На пример, инструкције за прављење дневника могу утицати на понашање апликације. Исто тако, исти преводилац би требало да постоји и у случају иницијалног компајлирања и у случају рекомпајлирања. Дакле, морало би да се обезбеди да компајлер буде и даље на располагању у време рекомпајлирања (вреди напоменути да рачунар који извршава захтев можда има другачију архитектуру од оне на којој се апликација компајлирала, као што је некад случај са специјализованим хардвером). Поред ових техничких питања, преводилац би рекомпајлирање извршио на основу претходног понашања извршавања апликације. Замислимо сценарио где је апликација за рад у реалном времену написана на такав начин да ни најгори сценарио *for* петљи не доводи до пропуштања рокова. Оптимизација просечног времена извршења помоћу хардвера заснованог на протоку података може да доведе до неиспуњених рокова, који би

могло довести до катастрофе, на пример у случају да се апликација у реалном времену користи за код возила која сама собом управљају или код ракета.

Чак и ако оставимо технолошка ограничења и ивичне случајеве на страну, овакав преводилац можда не би могао у да преведе петљу која нема међусобно независне итерације, али се може трансформисати у такву петљу.

Ово није први пут да се софтверски инжењери сусрећу са проблемом промене парадигме извршавања апликација. Када је било пројектовати систем са много језгара као што су тзв. *Compute Unified Device* (CUDA) архитектуре [4], преводиоци су морали да буду у стању да преводе и апликације писане за такву архитектуру, али су и програмери морали да промене своје програме. Ако апликација мора бити преписана за концептуално другачије архитектуре, изворни код који би требало да се паралелно извршава би требало да се експлицитно дефинише. Због тога, програмер треба да утврди који сегменти кода садрже најинтензивније рачунање и самим тим који од њих треба да буду извршени паралелно.

Стога, циљ овог истраживања није да се омогући да рачунар разуме апликацију која је покренута, већ да се обезбеде методе и средства која ће учинити трансформацију апликација лакшим. Да би се то урадило, извршен је преглед расположивих метода погодних за рачунарство засновано на протоку података доступну из сродних области. Друго, алат мора да буде изабран за проблем који треба да реши. За сваки доступан алат, разматраће се које методе за трансформацију алгоритма заснованог на протоку података су укључене, као и шта су алгоритамска и хардверска ограничења коришћења тог алата. Са тачке гледишта алгоритама, размотриће се која ограничења морају бити елиминисана како би се извршио алгоритам помоћу одабраног алата, а које методе треба директно програмер да примени.

3.3. Парадигма заснована на протоку података и Фејнман парадигма

Иако Нобеловац Ричард Фејнман није радио на рачунару заснованом на протоку података, његова запажања су подстакла многе истраживаче (универзитета и индустрије) за рад на рачунарима заснованим на протоку података [5]. Ова чињеница оправдава да се његово име користи као референца за парадигму која се све више и више користи у супер-рачунарству. Овде ће бити описани општи принципи приступа заснованог на протоку података који прати Фајнман парадигму.

Максимизовање перформанси извршења апликација заснованих на протоку података подразумева активности у два поља. Један је сам алгоритам, а други је хардвер. Крајњи циљ модификације алгоритма је да може да се изврши у мање циклуса такта. Ако алгоритам треба извршити на једном језгру, то може да се уради једноставним алгоритамским трансформација. У супротном, могла би да се оптимизује комуникација између процесора при извршењу алгоритма. Оптимизација хардвера би могла да се уради на два начина. Један је прављење процесора који ће бити што је могуће бржи. Други је коришћење што је више могуће језгара да се изврши алгоритам погодан за такву архитектуру рачунара. Следе основни принципи и њихова ограничења:

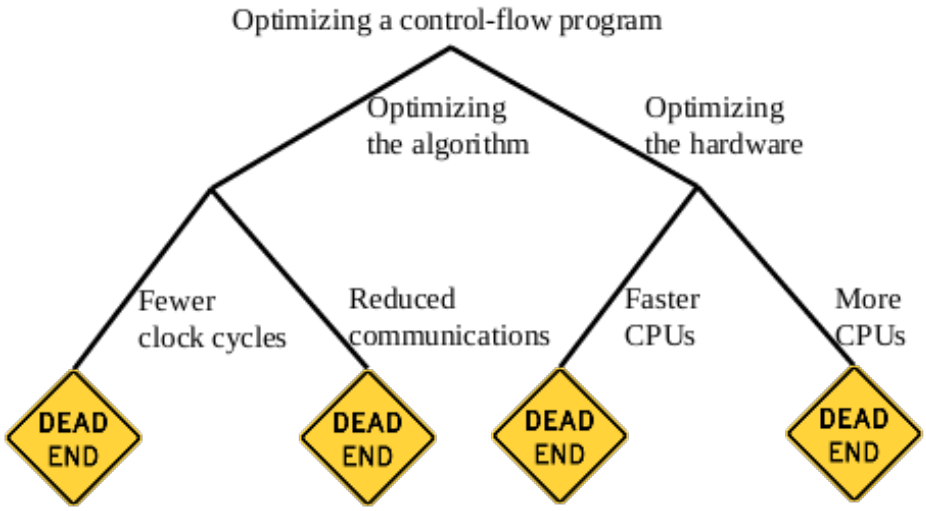
1) Смањење броја циклуса такта: када је алгоритам оптималан,
није могућ даљи напредак коришћењем овог приступа.

2) Оптимизација комуникација: када је алгоритам оптималан,
није могућ даљи напредак коришћењем овог приступа.

3) Повећање брзине такта процесора: границе технологије су готово досегнуте,
што нас ограничава од даљег скалирања перформанси у том правцу.

4) Додавање више процесора: иако ово може да убрза извршавање делова
алгоритма који нису зависни један од другог,
на крају је потребно прикупити све податке,
што нас ограничава од дељења скупа података за обраду на мање целине.

Слика 5 приказује претходно дефинисана четири сценарија.



Слика 5: Оптимизовање програма заснованих на контроли тока.

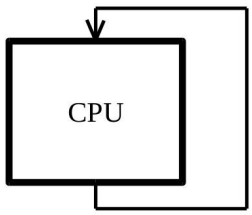
С друге стране, оптимизација алгоритама заснованих на протоку података се фокусира на брзини којом сигнал протиче кроз жицу. Људски мозак обрађује више информација у секунди од данашњих рачунара, иако није познато да људи могу рачунати ни изблиза брзо као модерни рачунари. Стога, да би се заиста убрзало извршење алгоритма, можда ћемо морати да се вратимо на почетак.

Од како су се рачунари први пут појавили, побољшаване су им перформансе релативно брзом стопом раста, приближно удвостручавањем броја извршених инструкција у секунди сваке двије године. Ниво паралелизма на нивоу инструкција (енг: Instruction level parallelism - ILP) датира из 1940-их и 1950-их, када је паралелизам по први пут експлоатисан у виду хоризонталног микрокода. Овај тренд је настављен све до данас. Постало је тешко да се одржи исти степен повећања брзине, док је обично ограничена могућност експлоатације паралелизма на нивоу инструкција у апликацијама. Касније, у 1960-им годинама, више ресурса је било могуће учинити доступним него што је било потребно за процесоре опште намене, што је довело до разних побољшања. Једна од најпознатијих до данашњег дана су кеш меморије које су имале пресудан утицај на перформансе извршавања апликација, посебно од 1980-их, када се јаз између брзине процесора и меморије повећавао драстично. Раст у брзини процесора је наставио да се повећава сличном стопом. Када су границе технологије скоро достигнуте, није било могуће да се

повећава брзина процесора истим темпом, чак и по цену експоненцијално веће потрошње електричне енергије. Као што је већ речено, таласна дужина брзине светлости једнака је брзини светлости подељене са фреквенцијом, а данашњи рачунари имају фреквенције од око 3GHz, што чини таласну дужину око 10cm, па величина транзистора не може бити много смањена, јер би у великој мери утицала на њихову функцију. То не оставља пуно простора за дугорочне побољшања у појединачним чиповима користећи постојеће технологије.

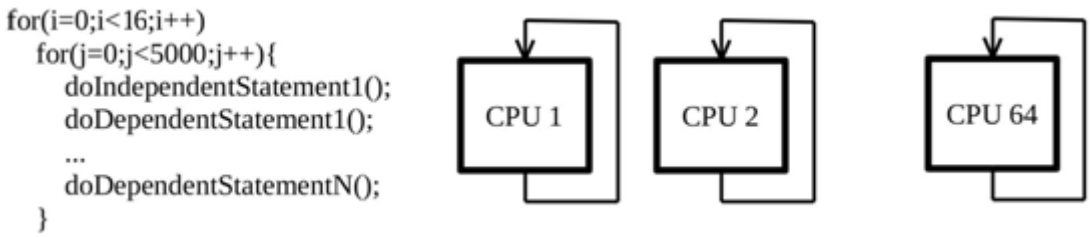
Узмимо једноставан пример алгоритма како би истражили могућности за убрзавање извршења користећи вишејезгарну парадигму и парадигму засновану на протоку података. Пример изворног кода и рачунарске парадигме једнојезгарног процесора су приказани на слици 6, где су петље итерација међусобно независне, а свака наредба у једној итерацији осим прве зависи од резутата претходне.

```
for(i=0;i<1024;i++)
   for(j=0;j<5000;j++){
      doIndependentStatement1();
      doDependentStatement1();
      ...
      doDependentStatementN();
   }
```



Слика 6: Једнојезгарна парадигма извршавања програма.

Мултијезгарна и мултипроцесорска парадигма решиле су овај проблем увођењем много језгара односно процесора уместо једног. Главни проблем у извршавању апликација природно се померио од пројектовања најбрже апликације за једнојезгарне рачунаре на паралелизацију извршавања апликација. Ипак, повећање брзине је ограничено комуникационим кашњењима, док потрошња електричне енергије језгара расте, као и величине језгара. Побољшања у мултијезгарним процесорима често више нису праћена бржим извршавањем апликација [21, 22]. Извршење истог кода коришћењем више језгара или процесора је приказано на слици 3, где сваки од 64 процесора извршава само мали део спољне петље. Овакво процесирање је често реализовало коришћењем MPI и OpenMP, а данас се обично спроводи коришћењем рачунара са графичким картицама које подржавају CUDA.



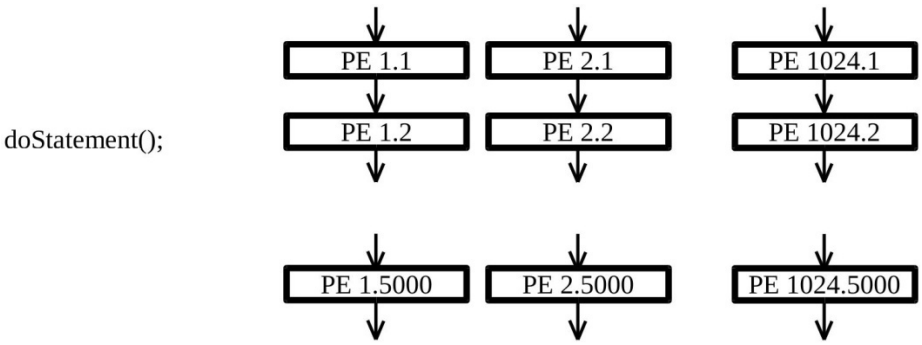
Слика 7: Мултипроцесорска парадигма извршавања програма.

Парадигма заснована на протоку података нуди могућност тзв. просторног рачунарства, што је природно решење за претходно наведене проблеме, третирањем компјутерске обраде као фабрике производних линија уместо једног или више специјализованих радника. Основна разлика између просторног рачунарства и тзв. временског рачунарства лежи у чињеници да просторно рачунарство претпоставља да се операције извршавају паралелно, са дистрибуираном меморијом широм кола, као и распоређивање извршавања дириговано протоком података. За посебне рачунарски интензивне апликације и апликације које користе велике количине података, може се дизајнирати ток података, а према њему се може конфигурисати рачунар заснован на протоку података. Апликације из различитих области су забележиле повећање перформанси, постизањем бољих односа цене и перформанси, као и снаге и перформанси у односу на одговарајуће Von Neumann имплементације [23].

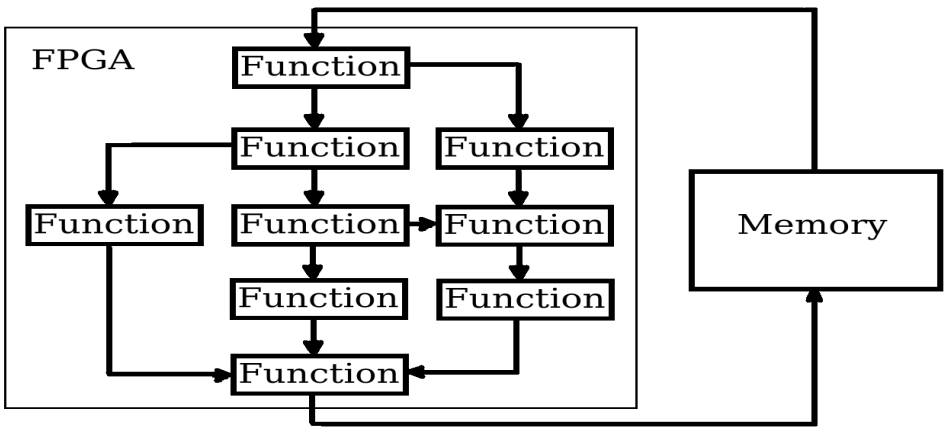
Већина архитектура заснованих на протоку података се састоји од међусобно повезаних елемената за обраду. Многи алгоритми се могу имплементирати коришћењем парадигме засноване на протоку података, укључујући и вештачку интелигенцију, симулацију динамике флуида, алгоритми за обраду сигнала, обраду слике, препознавање облика, динамичко програмирање, алгоритми за рад са графиком, итд. Једна од многих предности овог приступа је да је подаци теку од једних процесних елемента ка другима, без потребе да се приступи главној меморији. На овај начин, велики проток података је могуће постићи са релативно малим мрежним протоком. Ово елиминише потребу за постојањем тзв. Broadcast busses магистрала. Повезивањем елемената за процесирање, може се направити дијаграм тока података. Многи подаци могу тећи паралелно, спајајући се и раздвајајући у елементима за процесирање. Парадигма заснована на протоку

података је приказана на слици 8, где сваки елемент обрађује само једну наредбу, а сваки елемент осим елемената из првог и последњег реда прима потребан улаз из претходног елемента за обраду и шаље наредном у низу.

У општем случају, апликација се не састоји само од петље или две угнеждене петље. Међутим, све што је претходно поменуто и даље важи. Слика 9 приказује извршење алгоритма парадигме засноване на протоку података у општем случају. Може се видети да је за исти обим рачунања, потребан много мањи број приступа меморији. Исто тако, функције које су имплементиране у FPGA су једноставније од процесора који су у стању да извршавају било коју инструкцију дефинисану одговарајућом архитектуром рачунара. Према томе, може се размишљати о парадигми заснованој на протоку података са слике као о процесору заснованом на контроли тока који има тзв. pipeline дубине 10, без застоја и без икакве потребе за синхронизацијом.



Слика 8: Извршавања програма парадигмом заснованом на протоку података.



Слика 9: Поређење парадигми заснованих на контроли тока и протоку података.

Пошто парадигма заснована на протоку података није тако широко распрострањена као парадигма заснована на контроли тока, архитектуре засноване на протоку података су и даље релативно скупе и често прилагођене за одређене апликације. Други проблем је да постоји ограничен број апликација и алгоритама за њих. Примећивањем предности проточне обраде за петље, индустрије су развиле разне алатке који олакшавају процес трансформације апликације за Von Neumann архитектуре у апликације за архитектуре засноване на протоку података. Главни изазов у претварању је у превазилажењу суштинских разлика између архитектура. Један од најважнијих проблема за решавање је развој методологија за трансформацију алгоритама представљених у неким од програмских језика на високом нивоу у алгоритме који могу бити ефикасно извршени на архитектурама заснованоим на протоку података. Ово намеће неопходност вођења рачуна о времену када се подаци прослеђују између елемената, као и међусобним везама између елемената на такав начин да се проток података максимизира.

Поставља се питање зашто рачунари засновани на протоку података нису толико распрострањени као рачунари засновани на контроли тока, или чак и више? Наравно, то је зато што ова парадигма има своја ограничења. Као што је речено, убрзавање извршавања алгоритама помоћу парадигме засноване на протоку података подразумева активности у две области: хардвера и алгоритама. Од самог почетка, ограничења технологије у стварању ефикасног рачунара заснованих на протоку података су била присутна. Данас технологија омогућава креирање брзих рачунара заснованих на протоку података. Међутим, већина алата и људских вештина су оријентисани ка стварању бржих апликазија за рачунаре засноване на контроли тока. У раној историји рачунара, алгоритми су укључивали релативно мало рачунања у поређењу са данашњим алгоритмима. Обично су дизајнирани како би се ослободио људски мозак, и тиме омогућило људима да раде на компликованијим проблемима од обрачуна и једноставних обрада информација. Данас, није тако лако наћи програмера за рачунар заснован на протоку података као што је пронаћи програмера за рачунар заснован на контроли тока. Због тога, ограничења у убрзавању извршења алгоритама за архитектуре засноване на протоку података се могу грубо поделити на:

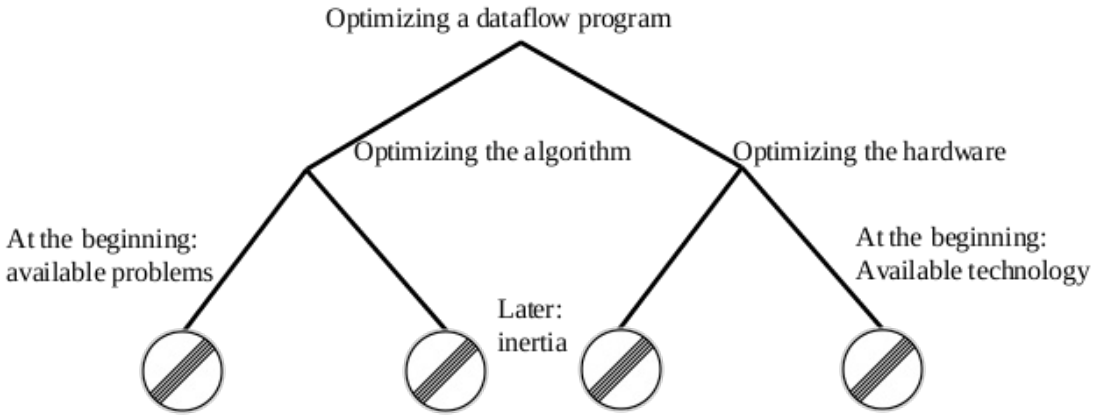
1) Постојећи проблеми које треба решити алгоритамски коришћењем рачунара заснованих на протоку података, на почетку

2) Недостатак алгоритама писаних за парадигме засноване на протоку података због инерције, касније

3) Доступне технологије за имплементацију ефикасних рачунара заснованих на протоку података, на почетку

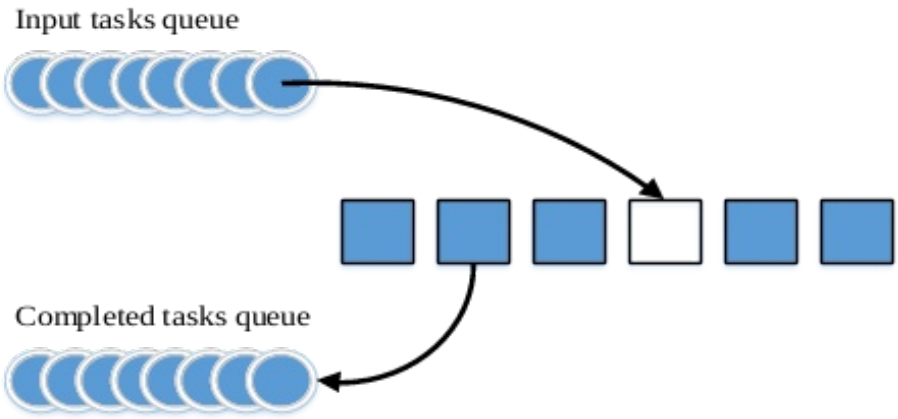
4) Потреба за променом технологије, касније.

Срећом, живимо у свету са све бржим темпом развоја. Стога је за очекивати да ће се расположиви потенцијал за убрзавање извршавања алгоритама помоћу парадигме засноване на протоку података ускоро постићи. Слика 10 приказује горе наведене сценарије.



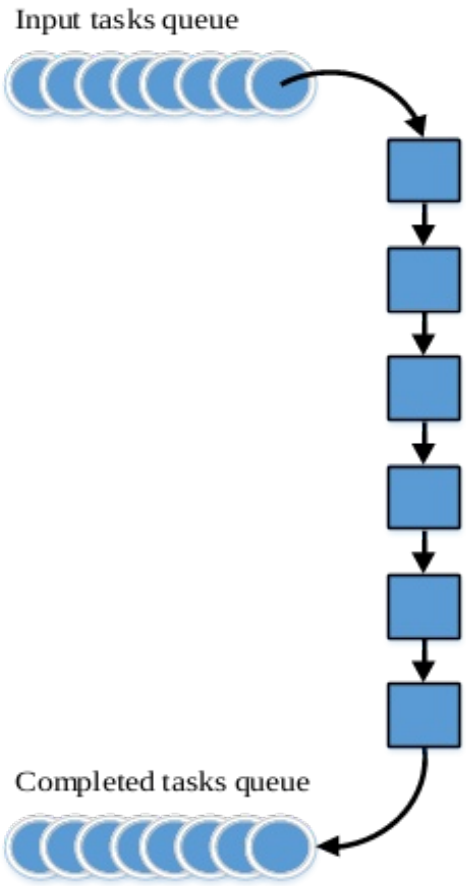
Слика 10: Оптимизовање апликација заснованих на протоку података.

Слика 11 показује типичну обраду пакета задатака (енг. *bag of tasks*). Ред задатака садржи задатке које треба процесуирати. Ако једнојезгарни или вишејезгарни процесор мора да ради ове задатке користећи тзв. multithreading, задаци ће се непрестано извршавати на следећи начин: биће узет први задатак из реда улазних задатака, затим ће он бити процесиран, а након тога излаз послат у излазни ред. Очигледно, додатни напор је потребан за распоређивање ових догађаја. Такође, приступ сваком реду би морао да буде атомски, што захтева коришћење семафора или неког другог механизма.



Слика 11: Оптимизовање апликација заснованих на контроли тока.

Претпоставимо да постоји 6 процесора који се користе за обраду скупа задатака који захтевају исту врсту обраде и да је сваки процесор одговоран за један део сваког задатка. Очигледно, они би морали да комуницирају, али синхронизација може да буде прилично једноставна. У свакој таквој итерацији, први процесор ће добити задатак из реда за унос задатака, процесирати њен део, и проследити резултат на следећи процесор. Следећи процесори ће добити улазе од претходних, а излазе довести до следећих, осим последњег који ће резултат проследити у ред завршених задатака. На тај начин, уместо релативно сложене архитектуре рачунара, релативно једноставни елементи за прераду би били довољни. Исто тако, уместо магистрале за комуницију процесора, елементи за обраду могу да буду директно повезани међусобно. Ако сваки елемент треба да процесира само око 1/6 од једног целог задатка, уместо сваког од ових елемената обраде, може се увести једноставнији елемент за обраду, који је способан за извршавање само подскупа инструкција. На овај начин, сви потребни елементи за прераду могу да стану на један чип. Као резултат тога, задаци се могу обрађивати са релативно малом количином рачунарских ресурса (транзистора, магистрала, итд.), уз уштеду енергије и простора истовремено, али и повећања брзине, као резултат паралелне обраде много елемената без потребе да се подаци дохватају и складиште у меморију више него што је потребно. Овај приступ је примењен код парадигме засноване на току података. У идеалном случају, дијаграм је приказан на слици 12.



Слика 12: Оптимизовање извршавања апликација заснованих на контроли тока.

Међутим, постоје одређена ограничења када се користе архитектуре засноване на протоку података. Прво, архитектура заснована на протоку података мора да буде подешена за одређени посао. Као што је већ речено, потребно је написати апликацију која ће велики број пута извршавати исти скуп инструкција. Исто тако, обрада једне итерације *for* петље не би смела да зависи од резултата претходне итерације. У супротном, процесор заснован на протоку података би требало да буде заустављен, чекајући да улаз буде спреман. Други проблем је фактор између броја бајтова који треба да се шаље на обраду и броја операција коју обрада укључује. Да би се ефикасно користила архитектура заснована на протоку података, време проведено у слању података са процесора заснованог на протоку података мора да буде мање од времена за обраду података користећи процесор. Чак и ако се подаци унапред сместе на меморијски чип хардвера заснованог на протоку података, редослед обраде података може утицати на брзину извршења у великој мери. На пример, ако апликација треба да анализира податке који моделују

саобраћајну мрежу, обрада узастопних ивица за редом може донети много зависности између података који се узастопно обрађују [24]. Међутим, ако алгоритам може да се изврши на такав начин да се подаци из релативно далеких ивица обрађују истовремено, може се постићи да подаци који су потребни за обраду буду увек спремни. Према томе, постоји потреба за методама и алатима који ће ефикасно претварати апликације засноване на контроли тока у апликације за архитектуре засноване на протоку података.

3.4. Постојећа решења

Решења за проблем трансформације алгоритама из парадигме засноване на контроли тока на парадигму засновану на протоку података могу се наћи у разним научним областима. Приказана решења могу се посматрати као уградње метода које су првобитно измислили у друге сврхе, посебно у области VLSI дизајна, као на пример, коришћење више повезаних елемената за обраду (енг. *Processing Elements - PEs*) познате као систолички низови (енг. s*ystolic arrays*). Трансформација алгоритама из контроле тока на архитектуре рачунара засноване на протоку података је концептуално аналогна трансформацији алгоритама за систоличке низове. Баш као што се систолички низови састоје од повезаних процесних елемената, FPGA има логичке елементе и тзв. *look-up* табеле, повезане ресурсе за рутирање, и прилагодљив улаз/излаз (I/O) [25]. Анализираћемо методе које се користе у области преводилаца, и коначно, у вези са претходним методама доступне алате (на пример преводиоце) за трансформацију алгоритама заснованих на контроли тока у алгоритме за архитектуре рачунара засноване на протоку података.

Табела 3 сумира методе које ће бити представљене, при чему ће за сваку од њих бити представљени улази, резултат који производе, области њихове применљивости, ограничења, и да ли они захтевају интеракцију корисника.

Таβела 3: Поређење метода за трансформацију алгоритама заснованих на контроли тока у алгоритме засноване на протоку података.

Метод	**Улаз**	**Излаз**	**Οβласт применљивости**	**Ограничења**	**Да ли захтева интеракцију корисника**
The Cohen, Johnson, Weiser, and Davis [26-28]	Математички израз	Систолички низ	Може бити примењен у оквиру алата, нпр. за препознавање подстрингова	Транслација математичких израза	Не
The Lam and Mostow's method (SYS) [29]	Алгоритам, опционално добијен трансформацијом софтвера	Систолички низ	Може бити примењен у оквиру алата	Једноставне *for* петље	Само ради оптимизације
The Gannon's method [30]	Алгоритам	Алгоритам	Извођење функционалне спецификације помоћу векторских оператора који представљају паралелизам	Трансформисање алгоритама који укључују векторске операције	Да
The H. T. Kung and Lin's method [31]	Каноничка математичка репрезентација алгоритма	Алгоритам	Помаже у дефинисању функционалности процесорских	Транслација математичких израза	Не

			елемената и рачунању временских ограничења		
The Kuhn's method [32, 33] & The Miranker and Winkler's method [34]	Алгоритам дат у виду математичког израза или као апликација са *for* петљом	Алгоритам	Користан за трансформисање одређених алгоритама и математичких израза, ослобађајући инжењера од рачунања временских ограничења	Систематски дизајн изгледа могућ само за апликације које имају *for* петље	Не
The Moldovan and Fortes' method [35-40]	Апликација или сет једначина	Алгебарски модел алгоритма	Погодан за апликације високих рачунарских система које се састоје од петљи	Апликације високих рачунарских система које се састоје од петљи	Не
The Lerner's method [41]	Код	Оптимизован код	Оптимизује код и елиминише недостижан код	Оптимизује код	Не
The ROCCC system [42]	Апликација написана у С програмском језику за RTL VHDL	Генерише високо оптимизована кола за делове кода у програмском	Апликација мора бити написана у програмском језику С за RTL VHDL	Корисник нема довољно контроле. Апликације морају бити написане у	Неопходно је барем основно разумевање начина на који оптимизација

		језику C		програмском језику C	ради, као и процес мапирања. Такође, корисник мора да учествује у процесу трансформи-сања апликације засноване на контроли тока
The Maxeler framework [43, 44]	Апликација написана у програмском језику C или Java или Python и кернел написан у програмском језику MaxJ	Генерише високо оптимизо-вана кола за дати кернел	Погодан за апликације за рачунаре високих перформанси које се састоје од петљи. Оптимизује код и елиминише недостижан код	Корисник мора да узме учешће у процесу трансформи-сања апликације засноване на контроли тока	Да

Следећа окружења ће бити укратко описана имајући у виду претходно описане методе:

- Mitrionics-C [45],

- StreamsC [46],

- ImpulseC [47],

- SystemC [48],

- SPARK [49],

- Метод за аутоматско партиционисање [50] и

- Синтеза софтвера за преглед мреже [51].

3.4.1 Методи наслеђени из теорије систоличких низова

Да би се истражила доступна решења за проблеме на које се наилази код парадигме засноване на протоку података, могу се посматрати најбоља решења сродних научних области која се могу применити на парадигму засновану на протоку података. Аутори [52] називају ово Мендељевизацијом, јер проблем трансформације изворног кода рачунара заснованог на протоку података очигледно постоји, а не постоји општи консензус о томе шта је решење. Према њима, коришћење парадигме засноване на протоку података се може посматрати као ревитализација, јер је решење проблема брже него што је могуће коришћењем парадигме контроле тока постојало одавно, али је игнорисано у време док су се фреквенције процесора заснованих на контроли тока скоро удвостручавале на сваке две године. Представљене методе могу се посматрати као имплантација метода које су првобитно измишљене у друге сврхе (на пример за потребе систоличких низова).

Концепт систоличних процесора су увели Kung, Leiserson и Lehman [53, 54]. Систолички низови ефикасно користе масивни паралелизам у апликацијама које захтевају интензивна израчунавања [55]. Процесор који се састоји из систоличких низова има међусобно повезане процесне елементе (PEs), где сваки процесни елемент користи резултате произвољног броја процесних елемената и даје резултат процесним елементима којима је потребан. Сваки процесни елемент може бити посебна ћелија са предефинисаним функцијама, ћелија са једноставним сетом инструкција за прераду векторских елемената, или ћелија са контролном јединицом и процесорском јединицом. Процесори засновани на систоличким низовима су први пут употребљени за сложене проблеме попут множења матрица

релативно великих димензија, који се састоји од потпуно истих израчунавања као и рачунање које графичке картице данас обављају. Систолички низови се могу класификовати као полу-систолички низови, који поседују глобалне комуникационе линије података до сваког процесног елемента и чисте систоличне низове који су аналогни компјутерским архитектурама заснованим на протоку података. Ми ћемо се фокусирати само на чисте систоличне низове. Сви процесни елементи раде истовремено и неопходни операнди морају бити доступни у одговарајућем циклусу како би се процесирање података могло обављати.

Као што је случај са парадигмом заснованом на протоку података, један од најважнијих проблема систоличних низова је дизајнирање методологије за трансформацију алгоритама контрола тока у алгоритме систоличних низова. Многе методе су пројектовани и могу се грубо поделити у следеће групе:

1) Методе које омогућавају директно мапирање алгоритама на систоличне низове.

2) Методе за трансформацију алгоритама у погодне за систоличне низове; додатни рад је неопходан да би трансформисани алгоритами могли да се извршавају на процесорима заснованим на систоличким низовима.

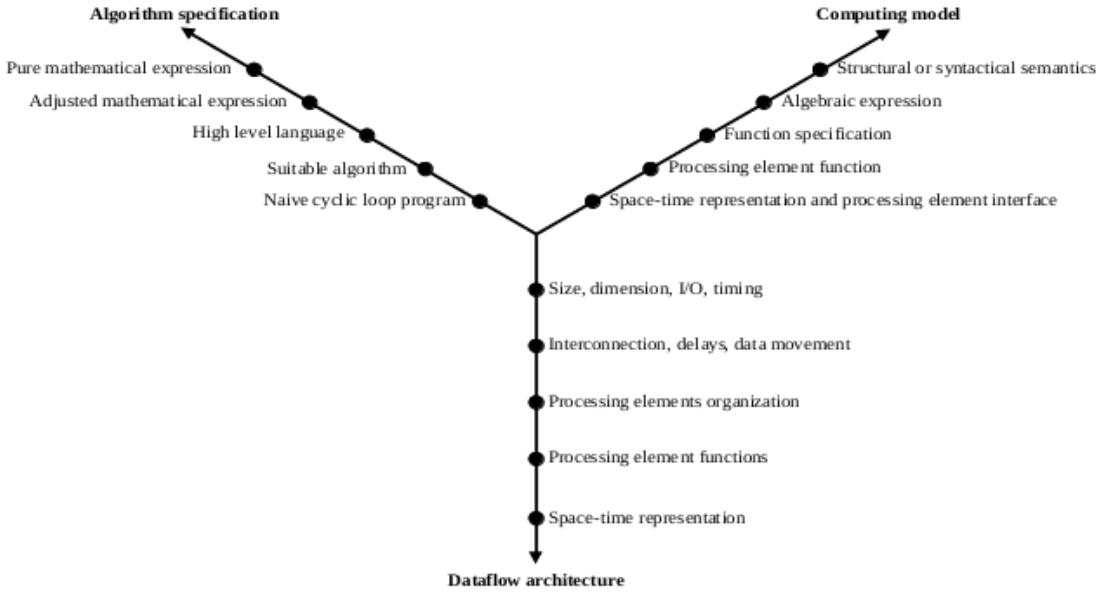
3) Методе за трансформацију постојећих архитектура у нове архитектуре.

4) Методе за доказивање исправности већ имплементираних алгоритама за систоличке низове.

Последње две групе метода су ван области ове тезе. Од интереса је трансформација апликација заснованих на контроли тока у апликације за архитектуре засноване на протоку података, а не проверавама исправности имплементације за архитектуре засноване на протоку података.

Методе наслеђене из теорије систоличних низова и теорије анализе токова података у преводиоцима ће бити описане коришћењем модела приказаног на слици 13. Методе наслеђене из теорије алата за програмирање архитектура заснованих на протоку података садрже много више детаља, па неће бити представљене истим моделом. Три праве представљају три осе: спецификацију алгоритма, рачунарски модел и архитектуру засновану на протоку података. Осе

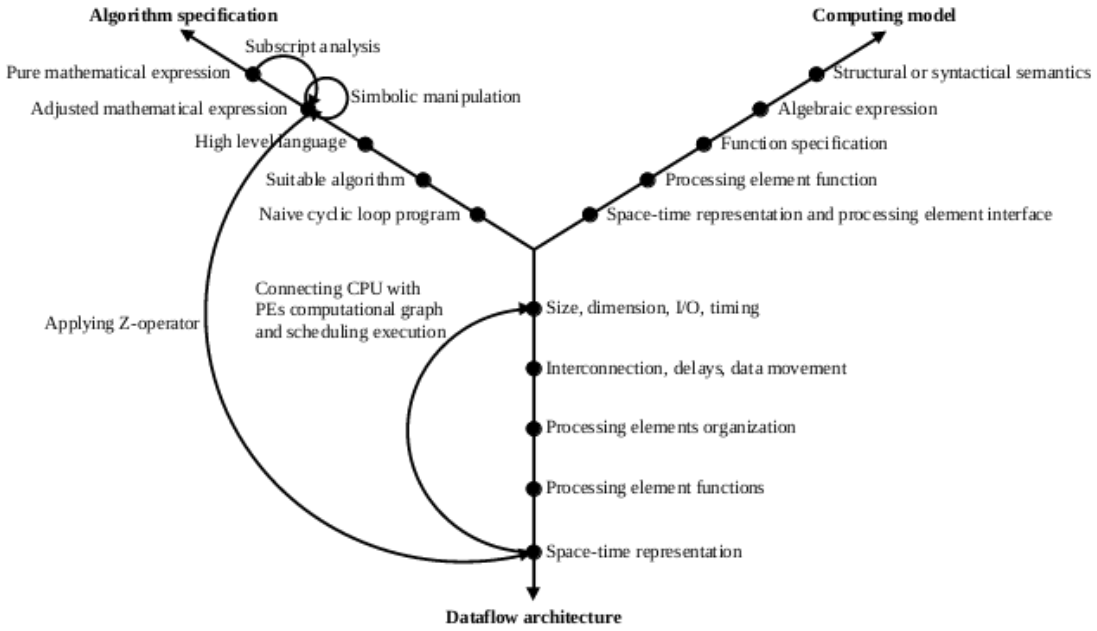
се користе за означавање нивоа апстракције. Сваки метод је описан као скуп трансформација које воде од почетне тачке до крајње тачке или тачака. Свака тачка мора припадати једној од оса. Што је тачка даља од центра, виши је ниво апстракције алгоритма, модела, односно архитектуре засноване на протоку података. На пример, почетна тачка може бити алгоритам у облику који би један математичар или један програмер могао написати. Користећи трансформације, може се добити тешко читљива репрезентација, што би се манифестовало стрелицом која води ближе центру оваквог система са три осе.



Слика 13: Генерализован модел за презентацију метода.

Cohen, Johnson, Weiser, and Davis метод [26-28] омогућава мапирање математичких израза на систоличне низове. Он полази од трансформације добро дефинисаних математичких израза који укључују променљиве са индексима користећи тзв. *subscript* анализу и симболичке манипулације. Затим користи Z оператор за моделовање растојања у времену и простору. Редослед извршавања операција трансформисаног израза се добија помоћу правила предности (енг. *precedence rules*). Временска ограничења се одређују на основу спецификација процесних елемената, а коришћење меморије се прилагођава расположивој у зависности од архитектуре процесних елемената.

Овај метод је се може применити на рачунаре засноване на протоку података, али само на превођење математичких израза, а не целе апликације написане у неком од високих програмских језика. Овај метод је успешно коришћен у претраживању низа знакова (енг. *string*), која је заједничка функционалност коју користе данашњи облаци рачунара (*cloud computing*) који користе Hadoop. Слика 14 показује метод примењен на проблем програмирања архитектура заснованих на протоку података.

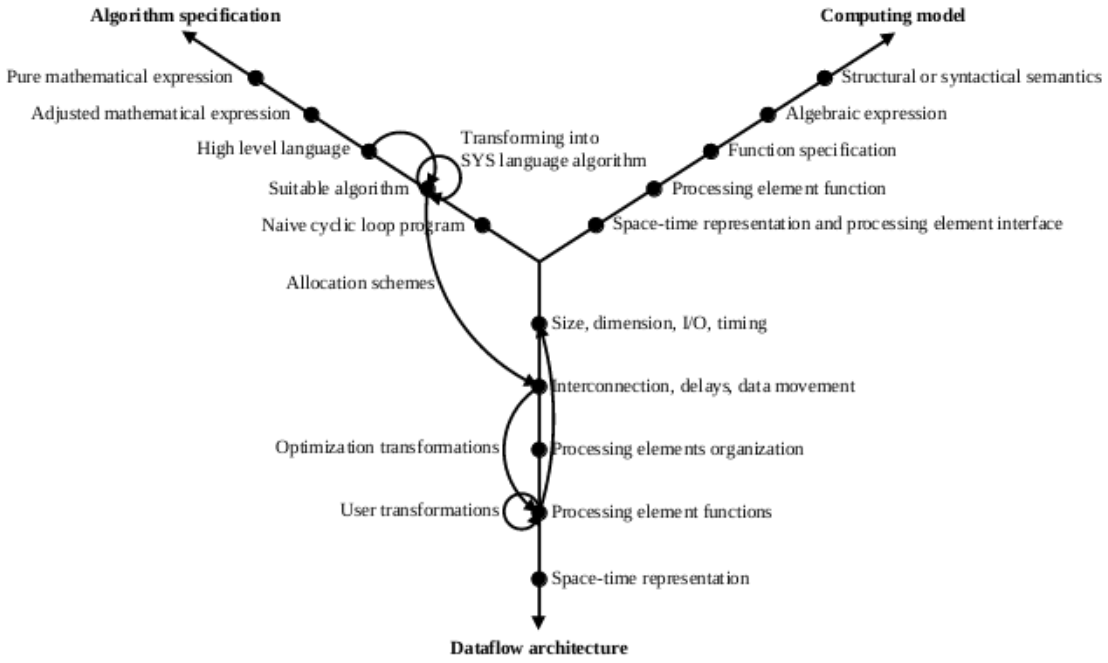


Слика 14: Cohen, Johnson, Weiser and Davis метод.

Ограничење ове методе је то што је у стању да преводи само математичке изразе. Стога, она може да послужи само као метод који могао бити укључен у алат који помаже превођењу апликација за архитектуре засноване на контроли тока у апликације за архитектуре засноване на протоку података.

Lam and Mostow метод *SYS* [29] такође мапира алгоритам на систолички низ. Међутим, он полази од већ направљеног софтверског кода у програмском језику сличном програмском језику Pascal, који обезбеђује анотације које чине програм погодним за аутоматско превођење. Метод укључује *for* петље са распонима познатим у време компајлирања и не зависи од променљивих које ће бити познате тек у време извршавања апликације. Условни изрази нису дозвољени. Петље могу да обухвате само једноставне тзв. *begin-end* блокове. *SYS* прихвата улаз добијен

софтверском трансформацијом алгоритма из спецификације писане у програмском језику високог нивоа који се састоји од кода који се периодично понавља. Алгоритам се пресликава на систолички низ на основу структуре која одређује архитектуру процесних елемената и драјвера који одређује временска ограничења и мапира токове на генерисан хардвер. Овај приступ је коришћен у реализацији систоличког низа за проналажење највећег заједничког делиоца два полинома. Слика 15 показује овај метод примењен на проблем програмирања архитектура заснованих на протоку података.



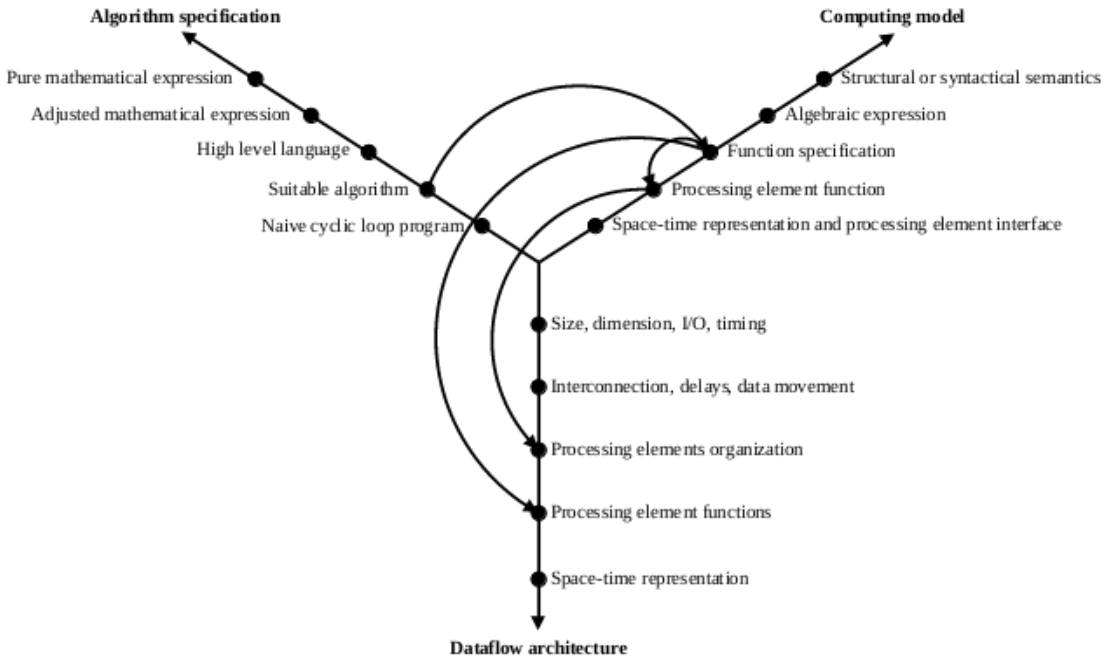
Слика 15: Lam and Mostow метод (SYS).

Ограничења овог метода су: (а) фокусиран је само на алгоритме са једноставним *for* петљама и (б) захтева од корисника трансформације у циљу оптимизације излаза. Делови овог метода могу бити укључени у алат који за циљ има превођење алгоритама са једноставним *for* петљама и сложену обраду.

Следеће методе се користе за трансформацију алгоритама у друге алгоритме, који су погоднији за систоличке низове.

Gannon метод [30] као улаз прима алгоритам. Он дефинише векторске операторе као логичке јединице које обухватају функције које могу бити реализоване

коришћењем процесних елемената. На пример, могло би се дефинисати множење два цела броја које ће се даље користити за множење два вектора. Други пример је пермутација података, који се могу користити за множење великих бинарних бројева. Такође укључују оператор ланца (енг. *chain operator*) који представља композицију других оператора. Функционална спецификација је изведена помоћу векторских оператора који представљају паралелизам. Функционална спецификација алгоритма може да се трансформише у график тока података, који се даље може трансформисати у систолички низ коришћењем овог метода. Као што је очекивано, овај метод је погодан за трансформацију алгоритама који укључују векторске операције. Слика 16 показује овај метод примењен на проблем програмирања архитектура заснованих на протоку података.



Слика 16: Gannon метод.

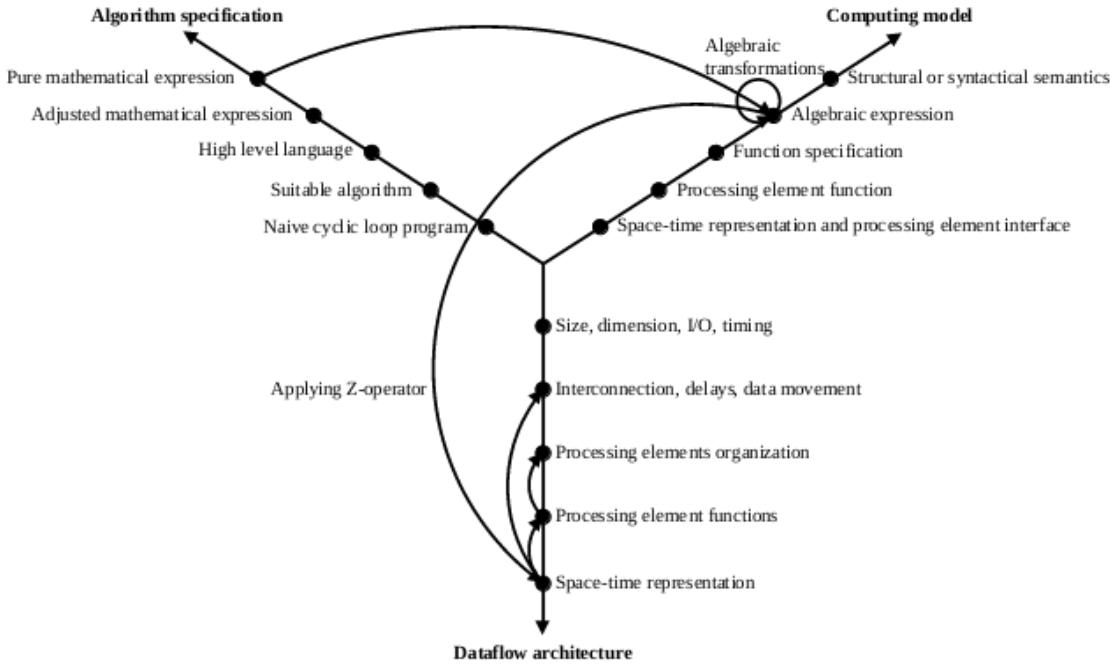
На жалост, људска интеракција може бити потребна, што чини овај метод неодговарајућим за аутоматско трансформисање великих апликација, али парадигма овог метода се може применити на сличним проблемима у програмирању рачунара заснованих на протоку података. За разлику од претходна два метода који се фокусирају на превођење одређених делова алгоритама за архитектуре засноване на протоку података, овај је ближи програмирању високих програмских језика. То омогућава дефинисање функционалности које ће се

извршити за сваки елемент низа. Инжењер је одговоран за паралелизам и ток података, али метод помаже у примени функционалности на елементе низа.

H. T. Kung and Lin метод [31] генерише алгоритамску репрезентацију од каноничке математичке репрезентације алгоритма. Аутори објашњавају принцип на примеру FIR филтера, где једноставна функција која сумира пондерисане вредности улаза мора да се израчуна. Почевши од C дизајна у Z-граф репрезентације, где сви производи одговарајућих улаза и тежинских фактора морају да се додају у једном циклусу, они прво генеришу алгебарску репрезентацију алгоритма. С обзиром на чињеницу да праве алгебарску репрезентацију основних блокова, аутори овог рада могу да тврде да би се овај процес могао аутоматски обављати за сложеније дизајне. Они даље примењују алгоритамске трансформације, стварајући на тај начин Z-граф репрезентацију, а затим, систолички дизајн који извршава само једну операцију у циклусу. Од Z-графова, могу да се добију временска ограничења и спецификација процесних елемената. Слика 17 показује овај метод примењен на проблем програмирања рачунара заснованих на протоку података.

Као што се може видети са слике, овај метод преводи само математичке изразе, применом алгебарских трансформација и првенствено је концентрисан на архитектуре засноване на протоку података, помажући у дефинисању функционалности процесних елемената и рачунању временских ограничења.

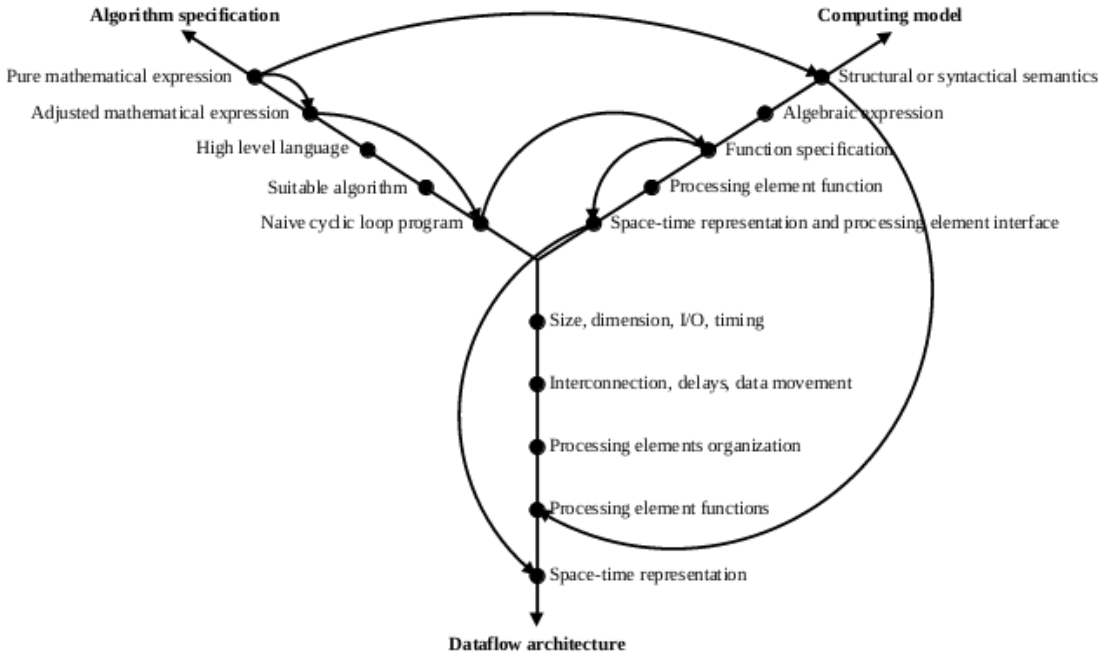
Kuhn метод [32, 33] је најпогоднији за алгоритме који се састоје од петљи, уз константно време извршења и зависности података међу итерацијама. Због тога, он је у стању да обради множења матрица и вектора, као и матрица и матрица, сортирање, итд.



Слика 17: H. T. Kung and Lin метод.

Miranker and Winkler метод [34] је наставак Kuhn методе. На основу алгоритма који се може дати у форми математичког израза или алгоритма са *for* петљом, овај метод подешава израз или петљу и претвара у наивни циклични програм који се састоји од петљи. Од тела петље, добија се функционална спецификација, а затим, добијају се просторно временска представа и интерфејси за процесирање елемената. Са тачке гледишта програма за архитектуре засноване на протоку података, оне би могле дефинисати функције процесних елемената и просторно временску репрезентацију, као што је показано на слици 18.

Овај метод подржава и друге алгоритме, али изгледа да је систематски дизајн једино могућ за апликације са петљама. Због тога је овај метод користан за трансформисање одређених алгоритама и математичких израза, ослобађајући инжењере од рачунања временских ограничења.



Слика 18: Miranker and Winkler метод.

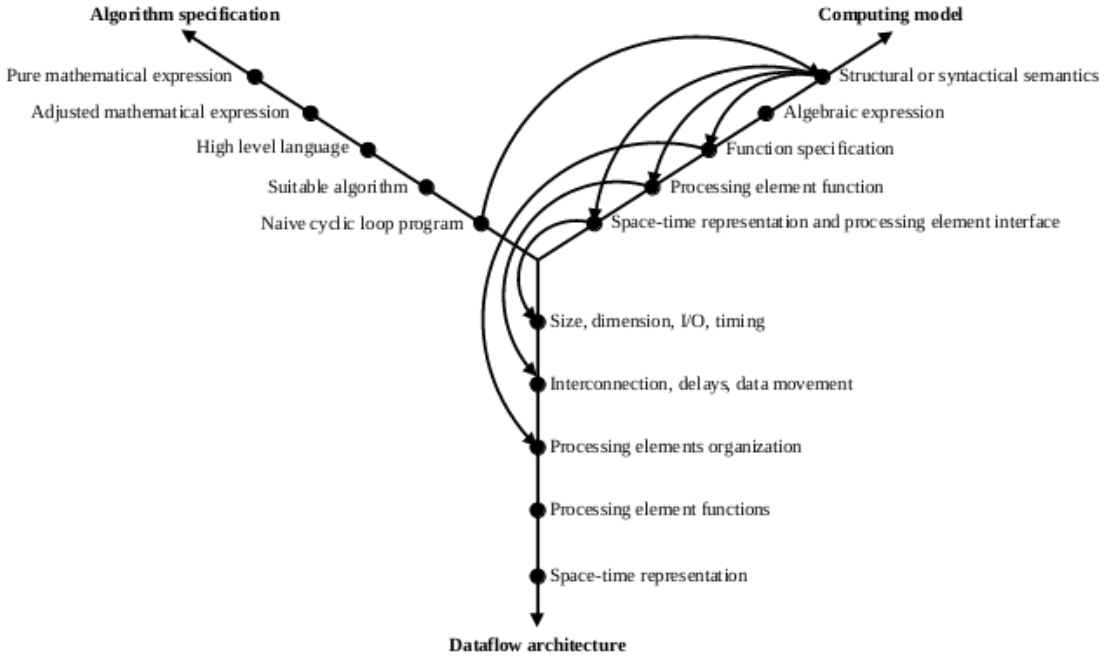
3.4.2 Методе из преводилаца и теорије анализе тока података

Методе оптимизације у конвенционалним преводиоцима су напредовале у великој мери захваљујући неколико деценија опсежних истраживања. Исто важи и за алате за аутоматизацију електронског дизајна (енг. *Electronic Design Automation - EDA*). Као резултат тога, имамо моћне компајлере и моћне алате који могу да преводе код из језика за опис хардвера VHSIC (енг. *VHSIC Hardware Description Language* - VHDL) и Verilog кода, као и SistemC кода. Међутим, мало истраживања је урађено у комбиновању тих приступа [56]. Неколико истраживача фокусирало се на транслирању кода написаног у високом програмском језику на код у језику за опис хардвера [57-60], углавном преводећи C или C++ код у језик за опис хардвера са оптимизацијама које су подржавале једноставне петље и векторе.

Moldovan and Fortes метод [35-40] користи технике сличне онима које се користе у преводиоцима за извођење алгебарских модела алгоритма из апликација или скупа рекурентних једначина. Метод се састоји од израчунавања индексираних

променљивих, чиме се добијају резултати задатих формула или апликација. Овај метод врши додатне трансформације користећи локалне трансформације и мапирање у функционалне и структурне спецификације за процесне елементе, као и глобалне трансформације које реструктурирају алгоритам тако да се нови сет зависности уклапа боље у постојећи хардвер, који је имплементиран користећи VLSI технологију. Трансформације везане за временска ограничења су одговорне за утврђивање тренутака долазака података до и од процесних елемената, док просторне трансформације утврђују међусобне везе између процесних елемената и премештања података. Слика 19 приказује овај метод примењен на проблем програмирања архитектура заснованих на протоку података.

Овај метод је погодан за апликације писане за рачунаре високих перформанси које се састоје од петљи са рекурентним једначинама. Међутим, како би се обезбедила већа флексибилност у подршци превођења других алгоритама у алгоритме за архитектуре засноване на протоку података, потребно је уложити додатан напор.

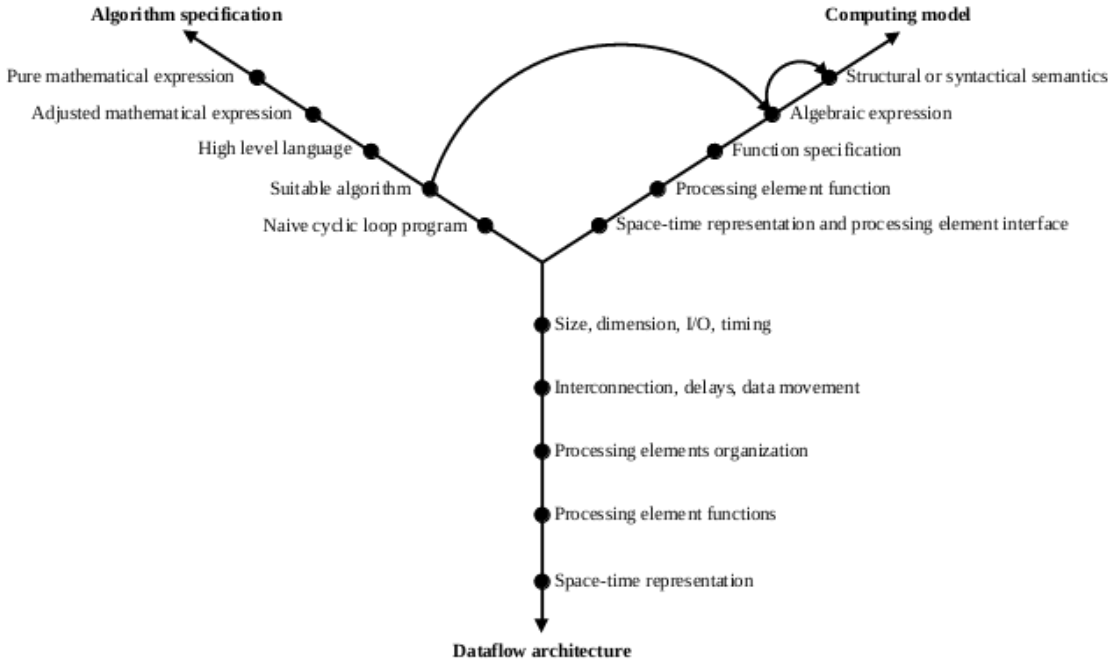


Слика 19: Moldovan and Fortes метод.

Многи истраживачи који су активни у области преводилаца за Von Neumann архитектуре рачунара су радили на анализи тока података и трансформацијама, у циљу оптимизације кода. Lerner [41] је увео Propagate и Replace акције које су у

стању да оптимизују код и да елиминишу недоступан код. Слика 20 показује овај метод, код кога су само осе за спецификацију алгоритма и рачунарски модел у употреби.

Ово је очигледан пример коришћења анализе тока података у оптимизацији кода, али такође може да помогне у програмирању архитектура заснованих на протоку података. Иако се Lerner није фокусирао на припрему алгоритма за архитектуру засновану на протоку података, његов рад се може применити у парадигми заснованој на протоку података.



Слика 20: Lerner метод.

На жалост, за разлику од специјализованих преводилаца за архитектуре засноване на протоку података и алате, конвенционални преводиоци не преводе апликације за архитектуре засноване на протоку података. Стога, они могу да послуже само као смерница у трансформацији алгоритама заснованих на контроли тока у алгоритме за архитектуре засноване на протоку података.

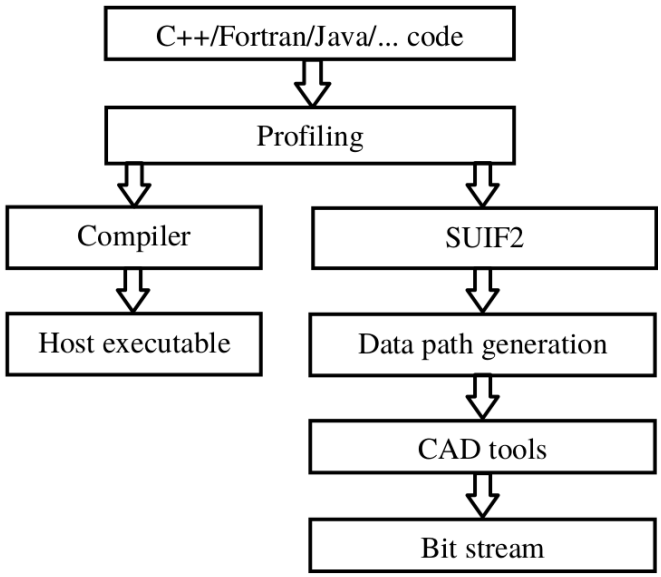
3.4.3 Методе наслеђене из теорије програмских алата за архитектуре засноване на протоку података

Решење проблема програмирања за мултипроцесорске системе [61] који се заснивају на примени архитектура заснованих на протоку података је започето од стране Adams [62], Chemberlin [63], и Rodriguez [64]. Од тада, многи алати су уведени, помажући у програмирању архитектура заснованих на протоку података, као и програмирању апстракција (нпр. [65]).

Поред самог процеса претварања апликација из програмских језика високог нивоа (на пример, C или Java) или дијаграма тока података у оптимални еквивалент ниског нивоа посвећен циљној платформи, развој софтвера побољшава процес развоја алата, обезбеђујући одговарајућу апстракцију циљних платформи, сакривајући детаље, како програмери не би морали да их знају. Међутим, у пракси, свако програмирање апликација за рачунаре високих перформанси захтеве најмање рудиментарно разумевање начина на који оптимизације и мапирање процеса раде, и морају учествовати у процесу трансформације апликација заснованих на контроли тока (на пример, подешавањем протока, или оптимизацијом оригиналних апликација) [25].

Као што је случај са већином доступних алата, *Riverside* оптимизациони преводилац за конфигурабилне рачунаре (енг. *Riverside Optimizing Compiler for Configurable Computing - ROCCC*) [42] мапира само подскуп апликација писаних у програмском језику C (нарочито за петље за обраду низова) на RTL VHDL. За већину апликација које захтевају рачунски интензивно извршавање, довољно је да се генеришу високо оптимизована кола за делове апликација написаних у програмском језику C, који се изнова и изнова извршавају. У поређењу са пружањем подршке за извршење целе апликације на FPGA, то омогућава ефикасно коришћење FPGA ресурса, остављајући простор за паралелизацију рачунања делова апликација који се највише извршавају. Аутори су развили сопствени ниво SUIF2 оптимизације. Следеће трансформације се примењују код ROCCC система: нормализација (енг. *normalization*), кретање инваријантног кода (енг. *invariant code motion*), piling, размотавање (енг. *unrolling*), фузија (енг. *fusion*),

блокирање (енг. *tiling/blocking*), размена (енг. *strip mining*), *interchange*, *unswitching*, *skewing*, индукције променљивих (енг. *induction variable*), и замена унапред (енг. *forward substitution*). Следеће процедуралне трансформације се примењују: *code hoisting*, *code sinking*, простирање константи (енг. *constant propagation*), поједностављивање алгебарских идентитета (енг. *algebraic identities simplification*), уклањање константи / уклањање поделу по један и множење са нулом (енг. *constant folding / removing division by one and multiplication by zero*), копирање пропагирање (енг. *copy propagation*), елиминација мртвог кода (енг. *dead code elimination*), елиминација недоступног кода (енг. *unreachable code elimination*), преименовање скалара (енг. *scalar renaming*), паралелизација редукција (енг. *reduction parallelization*), оптимизација дељења/множења константом (енг. d*ivision/multiplication by constant optimization*), као и конверзије засноване на предикатском извршењу (енг. *if conversion and predicated execution*). Трансформације низова које се примењују у ROCCS систему су: замена скалара (енг. *scalar replacement*), читање-после-писања (енг. *array read-after-write/RAW elimination*) и писање-после-писања (енг. *write-after-write/WAW elimination*), преименовање низа (енг. *array renaming*), елиминација повратних референци и простирање константних вредности [49, 51, 56]. Слика 21 приказује ROCCC систем.



Слика 21: ROCCC систем.

Постоје и додатна два комерцијална алата која су слична ROCCC систему, а то су Mitronics-C и ImpulseC.

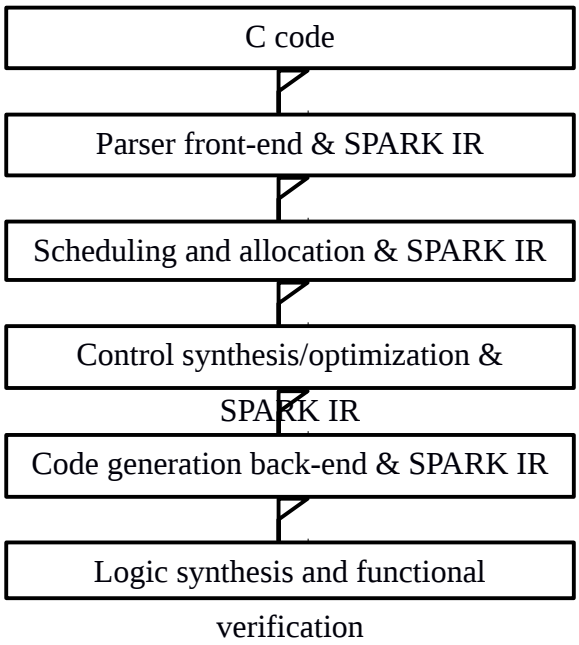
Mitrionics-C [45] окружење нуди програмски језик Mitrion-C који је заснован на програмском језику C, али има посебну наредбу петље која користи *foreach* кључну реч. Кернел, односно језгро, се програмира помоћу Mitrion-C програмског језика. Корисник мора да дефинише меморијски интерфејс помоћу кључних речи (на пример, *memread* и *memwrite*) и информације за регулисање временских ограничења користећи кључу реч *wait*. Када се језгро направи, Mitronics-C инстанцира паралелно језгро Mitrion виртуелног процесора на FPGA.

StreamsC [46] и ImpulseC [47] (комерцијални верзија StreamC) преводиоци паралелизују извршавање кода написаног у програмском језику C користећи VHDL. Њихов фокус је на петљама које би могле бити мапиране на хардвер. Међутим, корисник је одговоран за поделе кода на хардверске и софтверске процесе, и успостављање комуникације користећи канале за комуникацију између њих. За разлику од ROCCC, StreamC и ImpulseC се ослањају на секвенцијалну комуникацију процеса. Због тога, StreamsC и ImpulseC су намењени за структуре података засноване на току података и самим тим нису погодни за руковање мултидимензионалним низовима, јер би било потребно озбиљно реструктурирање кода. Међутим, Stream-C програмери понекад морају ручно да пишу код за поновно коришћење истих података коришћењем C програмског језика, како би се осигурало довлачење података из спољне меморије само једном.

SistemC [48] је дизајниран за комбинован систем који се састоји од софтверског и хардверског дела који раде синхронизовано, пружајући функционалност VHDL или Verilog језика. Handle-C [63] је језик ниског нивоа за генерисање хардвера на основу софтвера са синтаксом сличном програмском језику C.

SPARK [49] такође трансформише C у VHDL. Он подржава одмотавање петље (енг. l*oop unrolling*), елиминацију заједничких под-израза (енг. c*ommon sub-expression elimination*), пропагирање копирања (енг. c*opy propagation*), елиминацију мртвог кода (енг. d*ead code elimination*), копирање и стално пропагирање (енг. c*opy and constant propagation*), итд. Не подржава приступ више

димензионалним низовима. SPARK полази од дефинисаног понашања у програмском језику C, са ограничењем да изворни код не дефинише показиваче и рекурзивне функције. Трансформација почиње екстракцијом података зависности (енг. *data dependency extraction*), преименовањем променљивих, проточном обрадом инструкција петљи (енг. *pipelining loop instructions*). Након распоређивања, врши се контрола синтезе и оптимизације, контролором попут оне код коначних аутомата. Компонента *code generation back-end* генерише RTL VHDL коло које се додатно синтетизује и верификује. Синтеза коју SPARK систем обавља је на високом нивоу апстракције приказана на слици 22.



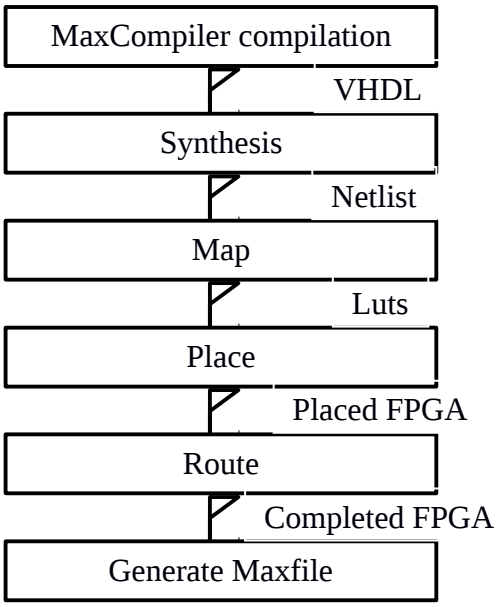
Слика 22: Процес синтезе SPARK система.

Међусобни позиви између процесора домаћина и FPGA су један од главних препрека за подршку аутоматском превођењу језика високог нивоа у VHDL. Истраживачи су радили на систему за комуникацију између процесора и FPGA, који подржава неограничене међусобне и хардверске рекурзивне позиве [66].

Једна од многих предности FPGA је да приступи меморији могу да се обаве паралелно. Ben-Ashem и Rotem су развили метод аутоматског партиционисања меморије за аутоматску синтезу најчешће коришћених структура података (низова и показивача) у више меморијских банака ради повећања паралелизма и, самим тим, побољшања перформанси [50].

За разлику од већине преводиоца за синтезу који се фокусирају на убрзању извршавања петљи, неки истраживачи су се фокусирали на употребу специјализованог хардвера. На пример, Jin Kim је развио софтверску синтезу за проналажење на мрежи (енг. *network Lookup*) која генерише читаве ланце за проналажење, тако да обављају агресивну проточну обраду (енг. *aggressive pipelining*) ради великог протока [51].

Maxeler окружење [43, 44] такође мапира делове кода на хардвер. Ово окружење такође пружа могућност програмирања кернела који се могу покренути на FPGA користећи MaxJ програмски језик високог нивоа (налик на програмски језик Java), који подржава функције програмског језика Java потребне за рачунање, али такође обезбеђује хардверске типове променљивих. MaxCompiler компајлер генерише VHDL спреман за пребацивање на FPGA. Синтеза претвара VHDL у логички „*netlist*", који обухвата основне логичке изразе. *Map* функција је задужена за паковање основне логике у N-улазне *look-up* табеле. *Place* је задужен за *look-up* табеле, дигиталне процесоре сигнала (енг. *Digital Signal Processors* - DSP), тзв. *Random Access* меморије (RAM) и флип флопове на одређене локације на чипу. *Put* поставља конекције између блокова. Овај систем је приказан на слици 23.



Слика 23: Фазе превођења употребом MaxCompiler преводилаца.

Алати чак могу да послуже као модел за програмирање који користи визуелни програмски интерфејс где програмер може да повеже блокове кода заједно, како би формирала представа кола високог нивоа, док је алат задужен за бригу о осталом. Листа алата са одговарајућим веб страницама дата је у табели 4.

Табела 4: Доступни алати за програмирање реконфигурабилних хардвера заснованих на протоку података.

Назив алата	Место настанка алата	Веб адреса
SystemC	Open SystemC Initiative (OSCI)	http://www.systemc.org/
Catapult C	Mentor Graphics	http://www.mentor.com/products/c-based_design/
Impulse C	Impulse Accelerated Technologies	http://www.impulsec.com/
Carte	SRC Computers	http://www.srccomp.com/CarteProgEnv.htm
Dime C	Nallatech	http://www.nallatech.com/?node_id=1.2.2&id=19
Streams C	Los Alamos National Laboratory	http://www.streams-c.lanl.gov/
AccelChip	MATLAB DSP Synthesis	http://www.accelchip.com/
Starbridge	VIVA	http://www.starbridgesystems.com/
NAPA-C	National Semiconductor	http://portal.acm.org/citation.cfm?id=795813
SA-C	Colorado State University	http://www.cs.colostate.edu/cameron/compiler.html
CoreFire	Annapolis Micro Systems	http://www.annapmicro.com/
Maxeler	Maxeler Technologies	http://www.maxeler.com/

3.5. Истраживање могућности парадигме засноване на протоку података

Симулирање динамике флуида употребом рачунара (енг. *Computational Fluid Dynamic - CFD*) је познато као компјутерски интензивно, односно рачунарски захтевно. Потребе за рачунањем сложених задатака симулације често могу бити задовољене само симулацијама на савременим суперрачунарима. И високе перформансе извршавања операција са покретним зарезом и висок проток меморије су неопходни [67]. Деценијама, научна заједница се борила да развије и оптимизује одговарајуће алгоритме као што су метода коначних запремина (енг. *finite volume*), коначних елемената (енг. *finite element*), односно метода коначних разлика (енг. *finite difference*) како би решиле Navije-Stokes једначине за високе Reynolds бројеве [68]. Алтернатива Navije-Stokes (NS) једначинама за израчунавање протока флуида представља Lattice-Boltzmann метод (LBM) [69]. LBM се заснива на Болцмановој једначини и важи у ширем опсегу физичких проблема тока него NS једначина [70]. LBM је еволуирао током последње две деценије и данас је широко прихваћен метод на универзитетима и у индустрији за решавање симулације токова нестишљивих течности [70, 71].

Као пример имплементације алгоритма који је прикладан извршавању на рачунарима високих перформанси, у овој тези ће бити описана једна имплементација овог алгоритма. Биће дана имплементација Lattice-Boltzmann метода и у програмском језику C и коришћењем парадигме засноване на протоку података и Maxeler окружења. Биће упоређено не само убрзање верзије алгоритма за архитектуре засноване на протоку података над одговарајућом верзијом за процесоре засноване на контроли тока, већ и смањење потрошње електричне енергије.

3.5.1 Lattice-Boltzmann метод

LBM је релативно нови приступ симулирању CFD. Он је у стању да нумерички решава NS једначине, али служи и за симулацију сложених физичких феномена

као што су хемијске интеракције између елементарних запремина течности и околине. Он је у широкој употреби због ефикасности и једноставности, као алтернативни алат за традиционалне методе решавања проблема симулације динамике флуида. Метод Lattice-Boltzmann је такође нумерички решавач Болцманове једначине која је аналогна NS једначинама на молекуларном нивоу. Она има способност да опише течности у не-хидродинамичним режимима са великим степенима слобода молекуларних путева (енг. *molecular mean free paths*). Lattice-Boltzmann метод је еволуирао од такозваних модела коначних стања, где стање представља положај и брзину интеракције псеудо-молекуларне запремине течности. Овај метод је дизајниран од самог почетка да ради на рачунарима високих перформанси. Једна од важних предности LBM у односу на неке друге методе, јесте да пред-процесирање траје само мали део укупног времена симулације. Иако је метод у неком смислу хибридан, јер је заснован на мрежи (енг. *mesh based*), а такође наслеђује и неке аспекте метода заснованих на честицама.

За разлику од класичних CFD метода, Lattice-Boltzmann метод може изгледати као да конзумира више ресурса него што је неопходно. Функције за дискретну расподелу вероватноће које се користе у овом моделу захтевају више меморије за складиштење него хидродинамичке променљиве у NS једначинама. Међутим, у пракси, посебно на савременим рачунарима, то се у великој мери компензује њеном рачунарском ефикасношћу. LBM укључује само врло ограничену количину операција са покретним зарезом у сваком рачунарском чвору, па је погодан за израчунавање на паралелним архитектурама рачунара, чак и са спорим везама између чворова односно процесора.

Тестирани су и време извршавања на процесору и на Maxeler картици за имплементације са матрицом са 112 редова и 320 колона. Број итерација је вариран између 1 и 100000.

3.5.2 Имплементација Lattice-Boltzmann метода у програмском језику C

Слика 24 представља компактан облик главне петље методе Lattice-Boltzmann, која је имплементирана у програмском језику C за архитектуру рачунара засноване на контроли тока. Можемо видети да се одређене функције позивају изнова и изнова, *maxIter* пута укупно, где *maxIter* потребан број итерација за симулацију.

```
for (iter=0;iter<maxIter;iter++){
   // NOTE: this part of the code should be parallelized,
   // more precisely the functions stream and collide
   stream();
   apply_BCs();
   collide();
   saveData();
}
```

Слика 24: Критични код Lattice-Boltzmann методе.

Функција *stream* захтева довлачење свих матрица. Међутим, она не захтева много процесирања. Функција *apply_BCs* је одговорна само за ажурирање ивичних елемената. Функција c*ollide* захтева довлачење свих матрица, али за разлику од функције *stream*, ова функција има значајну количину обраде за сваку координату матрице.

Функције *stream* и *collide* су део изворног кода Lattice-Boltzmann реализације доступне на интернету [72]. Анализом времена извршавања сваке од функција алгоритма, примећује се да се скоро све време извршавања троши на непрестано извршавање функције *stream* и *collide*. Укупно време извршења проведено на функцију *collide* је око 50% веће од укупног времена проведеног на извршење функције *stream*. Имајући у виду да функција *collide* готово све прорачунава,

могло би се очекивати да се времена извршавања ових функција више разликују. Међутим, преузимање података чак и са *L3* кеш меморије траје доста, у поређењу са множењем броја у покретном зарезу. На пример, Intel Core i7 Xeon 5500 серија чекања на довлачење податка траје отприлике 4 циклуса такта за L1 кеш, 10 за L2, и 40-65 за L3. Приступ удаљеном *L3* ће трајати између 100 и 300 циклуса такта. У случају да велике матрице које се морају обрадити методом Lattice-Boltzmann, могло би се очекивати кашњење од око 60ns, потребно за дохватање елемента из динамичке RAM меморије (DRAM).

3.5.3 Анализа потенцијала за убрзавање Lattice-Boltzmann методе

За почетак, биће дана аналитичка анализа потенцијала за убрзање LBM метода коришћењем принципа заснованих на протоку података. Пошто се технологије мењају, како у случају процесора засновних на контроли тока, тако и процесора заснованих на протоку података, подразумевани радни такт процесора је тачно 10 пута мањи за процесор заснован на протоку података. Као репрезентативна, изабрана је следећа брзина магистрале. Треба имати у виду да је ово такође одлука и да ће се и ова брзина сматрати застарелом, али се сама анализа неће променити докле год се одговарајуће две хардверске имплементације које се налазе у основи не промене. Брзина преноса између процесора и DFE је једнака брзини PCIe v4.0, а то је 31,508 MB/s.

Амдалов закон каже да је горња граница убрзања за било коју апликацију дана са *1 / (1 - p)*, где *p* представља део апликације који се може извршавати паралелно [73]. У нашем случају, то значи да би само инструкције која се извршавају на DFE могле да се убрзају. Срећом, готово сво време извршавања LBM алгоритма се троши на главној петљи, што резултује високим потенцијалима за убрзање.

Сложеност извршавања језгра архитектуре засноване на протоку података је *O (1)*, јер се по један улаз шаље на DFE у сваком такту. С друге стране, процесор заснован на протоку података извршава језгро онолико циклуса колико је потребно за извршење свих инструкција које производе исти резултат као језгра. Назовимо

овај број n. Имајмо у виду да нису све инструкције исте у реализацији алгоритма за процесор заснован на контроли тока и процесор заснован на протоку података. Сложеност одговарајућег кода на процесор заснован на контроли тока (CPU) је $O(n)$, где је n број инструкција. Предности приступа заснованог на протоку података су већ очигледна. Овде ће бити дана процена максималног убрзања које може да се оствари. Анализом кода процесора заснованог на контроли тока, може се проценити да је број циклуса процесора утрошен на израчунавање резултата које језгро у једном циклусу производи 190, на основу претпоставке да је број машинских инструкција 140, укључујући *add*, *sub*, *mul* и *div* и да је један циклус такта довољан за извршење сваке од поменутих инструкција, а да просечна дужина *sqrt* наредбе која израчунава вредност корена броја износи 50 циклуса такта. Није претпостављено постојање технике pipeline, и паралелних језгара у случају архитектуре засноване на протоку података. Очигледно је да је максимално убрзање које бисмо могли да добијемо под овим претпоставкама 190 подељено са 10, пошто процесор заснован на протоку података има тачно 10 пута нижу фреквенцију такта. Међутим, видећемо да ће убрзање бити мање у случају имплементације Lattice-Boltzmann алгоритма за архитектуре засноване на протоку података у поређењу са одговарајућом имплементацијом за процесор заснован на контроли тока. Треба имати у виду да данашњи процесори подржавају технику multithreading, и да на картицу засновану на протоку података може стати више од једног језгра, што чини паралелизацију на обе врсте процесора могућом.

Будући да је потрошња електричне енергије скоро пропорционална квадрату фреквенције, потрошња архитектуре засноване на протоку података има потенцијал да буде 100 пута мања у истим околностима. Ово подразумева само процесорску снагу у VLSI. Као што ћемо видети касније, рачунар са Maxeler картицом троши скоро дупло више електричне енергије, што је за очекивати, јер је потребно напајати и процесор и Maxeler картицу, али је парадигма заснована на протоку података претпоставља истовремно вршење више инструкција. Стога, укупно умањење електричне енергије се и даље може постићи, јер укупно време извршења може да се смањи. У општем случају, постоје кластери рачунара који садрже Maxeler картице, што у великој мери смањује количину електричне

енергије која се потроши на процесоре у поређењу са електричном енергијом потребном за Maxeler картице.

Поредећи са фреквенцијама данашњих процесора, можемо видети да се око 2,5 бајта могу пренети по сваком циклусу такта процесора, или 25 бајтова по једном циклусу Maxeler картице. То значи да око 6 бројева у покретном зарезу могу бити пренети на Maxeler у сваком циклусу. То би довело до застоја на Maxeler картици. Срећом, све матрице могу бити пренете на Maxeler картицу пре него што почне рачунање. Дакле, време потребно за пренос се троши само једном, што је упоредиво са временом извршења само неколико итерација LBM алгоритма коришћењем процесора.

Имплементација сваке инструкције алгоритма директно на хардверу уместо извршавања на конвенционалном процесору води ка смањењу броја транзистора који се могу употребити за убрзавање алгоритма. Међутим, не може се тврдити да би ово довело до смањења потрошње електричне енергије, јер поређење транзистора који су на процесорима заснованим на контроли тока и оних који су у употреби код процесора заснованих на протоку података није фер.

3.5.4 Имплементација Lattice-Boltzmann методе за Maxeler архитектуру рачунара

У циљу реализације LBM помоћу парадигме засноване на протоку података, могао би се изабрати одговарајући алат који ће обавити што је могуће више од процеса трансформације из алгоритма прављеног за процесор заснован на контроли тока на алгоритам за процесор заснован на протоку података. Следи кратак преглед метода доступних у области систоличких низова, овај пут разматраних са становишта LBM.

Cohen, Johnson, Weiser, and Davis метод [4, 5, 21, 22] мапира математичке изразе на систоличке низове. Ово је неопходно за превођење скоро сваке апликације из алгоритма за процесор заснован на контроли тока на алгоритам за архитектуру

рачунара засновану на протоку података, али је такође укључено у већини алата који могу помоћи у процесу превођења. LBM укључује повољан однос израчунавања по сваком податаку који је потребно преместити. Lam and Mostow метод (SYS) [23] такође мапира алгоритам за систолички низ, али се у случају овог метода полази од алгоритма погодног за превођење на парадигму засновану на протоку података. Као су алату потребне одговарајуће ознаке у циљу превођења делова апликација на хардвер заснован на протоку података, већина алата укључује одговарајуће библиотеке и наредбе које су потребне у процесу писања алгоритма, како би се назначили делови које је потребно прилагодити хардверу заснованом на протоку података или укључује методе којим је могуће алату обезбедити одговарајућу репрезентацију алгоритма за архитектуру засновану на протоку података. Gannon метод [25] производи функционалну спецификацију на основу алгоритма. У случају LBM, неопходно је да програмер дефинише функционалности које треба да се извршаве на хардверу заснованом на протоку података. H. T. Kung and Lin метод [26] трансформише математички израз у алгебарски израз, а затим прави спецификацију процесних елемената и дефинише временска ограничења. У случају LBM, оваква метода се не може користити без већих измена, јер би превођење само математичких израза у хардвер довело до претеране комуникације између главне меморије и хардвера заснованог на протоку података. Kuhn метод [27] и Miranker and Winkler метод [28] су најпогоднији за алгоритме који се састоје од петљи које имају константно време извршења и зависности података у итерацијама. Ово је користно у општем случају, али LBM захтева много комплексније методе како би се покрили ивични случајеви (енг. *corner cases*). Прецизније, LBM захтева другачију обраду елемената који припадају ивицама матрице. Moldovan and Fortes метод [29-34] производи алгебарски модел алгоритма из апликације или скупа рекурентних једначина. Овај приступ није примењен у овој тези због тога што не би био нарочито користан у процесу превођењу LBM метода. Lerner метод оптимизује код за пропагирање резултата претходних наредби следећим.

Као алат, изабрано је Maxeler окружење због следећих карактеристика, имајући у виду претходно описане методе:

- Аутоматско трансформисање математичких израза,

- Аутоматска трансформација *for* петљи са фиксним опсезима; Такође подржава одмотавање петљи,

- Аутоматско руковање комуникацијом између процесора заснованог на контроли тока и процесора заснованог на протоку података, и

- Аутоматско избацивање недостижног кода.

Поред ових, Maxeler има и могућност програмирања у програмском језику високог нивоа, подржава кретање инваријантног кода (енг. *invariant code motion*), piling, фузију (енг. *fusion*), блокирање (енг. *tiling/blocking*), размену (енг. *strip mining*), скаларне варијабле, замену унапред (енг. *forward substitution*), читање-после-писања (енг. *array read-after-write / RAW elimination*) и писање-после-писања (енг. *write-after-write / WAW elimination*) и простирање константних вредности.

Maxeler захтева да програмер имплементира језгро које ће бити одговорно за обраду елемената низа користећи хардвер заснован на протоку података, где ће подаци бити слати као улаз језгру, а други ток ће бити враћан као резултат обраде језгра. Језгро мора да има улазне и излазне параметре дефинисане од стране програмера као хардверске променљиве. Све између је скоро исти као у Java апликацији. Све математичке операције су дозвољене. Једино ограничење је да једном сачувана променљива као хардверска променљива не може поново да се преведе у Java променљиву. Можемо то посматрати на следећи начин: *floating-point* променљива постављена на нулу, а затим сачувана као хардверска променљива се мења због чињенице да жице нису идеалне. Стога напон на тој жици неће бити нула, већ близу нуле, због постојања шума на жици. Ако покушамо да сачувамо напон као Java променљиву, пошто напон није баш нула, вредност би се разликовала од почетне. Како програмери не могу лако да се прилагоде на неконзистентност извршавања написане апликације, као и из

чињенице да сама стартовања такве апликације не би увек давала исти резултат, то није дозвољено. Подаци могу да се шаљу у виду тока или из главне меморије, или из меморије хардвера заснованог на протоку података.

Специфичности Maxeler окружења ће бити описане на примеру LBM. Највећи део времена извршења проведеног у извршавању LBM на процесору се троши на извршавање функције *collide*. Због тога, прво је имплементирана ова функција као језгро које би процесуирало све елементе матрице, при чему би елементи били слати у виду тока са процесора ка картици и, а резултат враћан током од картице ка главној меморији. И у случају имплементације LBM алгоритма за процесор заснован на контроли тока и у случају имплементације на процесору заснованом на протоку података, матрице се чувају као вектори, при чему се први ред матрице чува на самом почетку вектора, а затим други ред матрице, итд. Слика 25 приказује код језгра које је одговорно за ажурирање елемената матрице, при чему су неке линије које су одговорне за рад са векторима од 1 до 9 изостављене ради прегледности и замењене тачкама.

```
public class Collidekernel extends kernel{

   public Collidekernel (kernelParameters parameters){

      super(parameters);

      HWVar rtau = io.scalarInput("rtau" ,hwFloat(8, 24));

      ...

      HWVar f0  = io.input("f0i",  hwFloat(8, 24));

      ...

      // Do the summations needed to evaluate the density and components of velocity

      HWVar ro = f0 + ...;

      HWVar rovx = f1 - f3 + f5 - f6 - f7 + f8;

      HWVar rovy = f2 - f4 + f5 + f6 - f7 - f8;
```

```
HWVar vx = rovx/ro;

HWVar vy = rovy/ro;

// Also load the velocity magnitude into plotvar;
// this is what we will display using OpenGL later

HWVar v2x = vx * vx;

HWVar v2y = vy * vy;

HWVar plotvar = kernelMath.sqrt(v2x + v2y);

HWVar v_sq_term = 1.5f*(v2x + v2y);

// Evaluate the local equilibrium f values in all directions

HWVar vxmvy = vx - vy;

HWVar vxpvy = vx + vy;

HWVar rortau = ro * rtau;

HWVar rortaufaceq2 = rortau * faceq2;

HWVar rortaufaceq3 = rortau * faceq3;

HWVar vxpvyp3 = 3.f*vxpvy;

HWVar vxmvyp3 = 3.f*vxmvy;

HWVar vxp3 = 3.f*vx;

HWVar vyp3 = 3.f*vy;

HWVar v2xp45 = 4.5f*v2x;

HWVar v2yp45 = 4.5f*v2y;

HWVar mv_sq_term = 1.f - v_sq_term;

HWVar mv_sq_termpv2xp45 = mv_sq_term + v2xp45;
```

```
        HWVar mv_sq_termpv2yp45 = mv_sq_term + v2yp45;

        HWVar vxpvyp45vxpvy = 4.5f*vxpvy*vxpvy;

        HWVar vxmvyp45vxmvy = 4.5f*vxmvy*vxmvy;

        HWVar mv_sq_termpvxpvyp45vxpvy = mv_sq_term + vxpvyp45vxpvy;

        HWVar mv_sq_termpvxmvyp45vxmvy = mv_sq_term - vxmvyp45vxmvy;

        HWVar f0eq = rortau * faceq1 * mv_sq_term;

        ...

        f0 = rtau1 * f0 + f0eq;

        …

        io.output("f0o", f0, hwFloat(8, 24));

        …

    }

}
```

Слика 25: Lattice-Boltzmann *collide* језгро.

У поређењу са имплементацијом LBM *collide* методе за процесор заснован на контроли тока, примењене су трансформације да би се код брже извршавао на архитектури заснованој на протоку података [74]. Наиме, готово све математичке операције се разликују од оних у случају имплементације алгоритма за процесор заснован на контроли тока. Из тог разлога, овај алгоритам је један од најповољнијих за демонстрацију примене представљених метода у трансформацији алгоритма из алгоритма за процесор заснован на контроли тока на алгоритам за процесор заснован на протоку података.

Пошто LBM има *for* петље са константним временом извршења, могуће је постићи оно што је подразумевано у методи Kuhn и методи Winkler помоћу Maxeler окружења и имплементирати функционалности за DFE. Maxeler

окружење даље изводи функционалну спецификацију на основу овог језгра, као у Gannon методи. Док Cohen, Johnson, Weiser, and Davis метод трансформише математичке изразе у функционалност процесних елемената и спецификацију временских ограничења, Lam and Mostow метод бliже описује шта је постигнуто коришћењем Maxeler окружења, јер се у случају Maxeler окружења морао написати код у MaxJ програмском језику, како би тако дефинисано језгро било мапирано на FPGA. На жалост, док је Moldovan and Fortes метод способан да аутоматски изводи алгебарски модел алгоритма из апликације или скупа рекурентних једначина, у случају Maxeler окружења, неопходно је размишљати о токовима података и индекса елемената. Maxeler окружење је одговорно за бригу о функционалностима дефинисаних у H. T. Kung and Lin методи. Коришћењем технике које су доступне у преводиоцима, оптимизовано је простирање сигнала, као што је предложено у Lerner методи.

LBM има *for* петље са константним временом извршења, па је могуће постићи оно што је подразумевано у методи Kuhn и методи Winkler помоћу Maxeler окружења и имплементирати функционалности на DFE. Maxeler окружење даље изводи функционалну спецификацију на основу овог језгра, као у Gannon методи. Док Cohen, Johnson, Weiser, and Davis метод трансформише математичке изразе у функционалност процесних елемената и спецификацију временских ограничења, Lam and Mostow метод бliже описује шта је постигнуто коришћењем Maxeler окружења, јер се у случају Maxeler окружења морао код написати у MaxJ програмском језику, како би тако дефинисано језгро било мапирано на FPGA. На жалост, док је Moldovan and Fortes метод способан да аутоматски изводи алгебарски модел алгоритма из апликације или скупа рекурентних једначина, у случају Maxeler окружења, неопходно је размишљати о токовима података и индекса елемената. Maxeler окружење је одговорно за руковање функционалностима дефинисаних у H. T. Kung and Lin методи. Коришћењем техника које су доступне у преводиоцима, оптимизовано је простирање сигнала, као што је предложено у Lerner методи.

Као што се види на слици, постоји значајна количина обраде која мора да се уради за сваки елемент сваке од матрица. Имајући у виду предности парадигме

засноване на протоку података, укључујући извршавање стотине или хиљада инструкција паралелно, а да при томе није неопходно да се сваки улазни податак дохвати из меморије, већ се размена улаза и излаза суседних процесних елемената које су одговорне за обраду обавља међусобно, за очекивати је да ће овако дефинисано језгро брже извршавати алгоритам на Maxeler картици него што се извршава на процесору. Међутим, да би се убрзало извршење алгоритма, преостале функције морају такође да се изврше на Maxeler картици.

Језгро које одговара функцији *stream* је дато на слици 26. Као и у случају функције *collide*, неки редундантни делови су замењени са тачкама. Главни разлог за дуготрајно извршење функције *stream* је што елементи морају бити учитани. Са друге стране, процесирање које мора да се уради за сваки елемент је веома једноставно. Треба приметити да ово језгро подразумева и извршење инструкција из *apply_BCs()* функције.

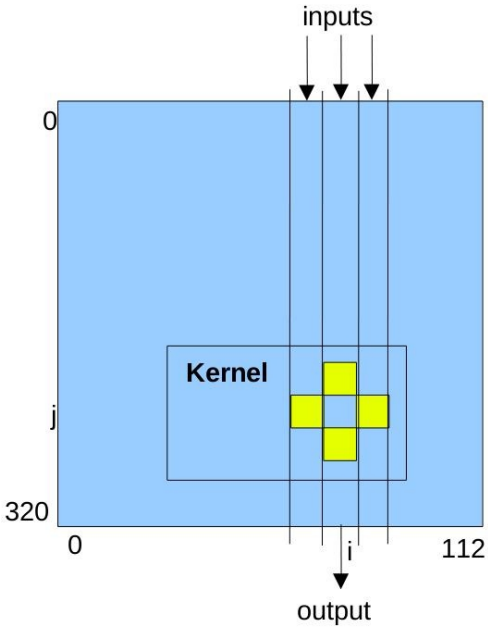
```
public class Streamkernel extends kernel {

   public Streamkernel (kernelParameters parameters) {

      super(parameters);

      HWVar f1new = io.scalarInput("f1new" ,hwFloat(8, 24));

      HWVar f5new = io.scalarInput("f5new" ,hwFloat(8, 24));

      HWVar f8new = io.scalarInput("f8new" ,hwFloat(8, 24));

      ...

      HWVar f1  = io.input("f1",  hwFloat(8, 24)); // j

      HWVar f2m = io.input("f2m", hwFloat(8, 24)); // j-1

      HWVar f3  = io.input("f3",  hwFloat(8, 24)); // j

      HWVar f4p = io.input("f4p", hwFloat(8, 24)); // j+1

      ...
```

```
//Boundary conditions

//HWVar cnt = control.count.simpleCounter(32);//.cast(hwFloat(8, 24));

HWVar cntMod = control.count.simpleCounter(32, 320);

// 32 bit counter with module 320

HWVar cond318temp = cntMod > 317;

HWVar cond318 = cntMod < 319 ? cond318temp : 0; // k5

HWVar sel_l = cntMod > 0;

HWVar sel_h = cntMod < 320;

HWVar pom1tmpf1 = cond318 ? f1 : stream.offset(f1, -1);

// tmpf1[i0] = f1[I2D(ni,im1,j)];

HWVar tmpf1 = sel_l ? pom1tmpf1 : f1new;  // tmpf1[i0] = f1[I2D(ni,im1,j)];

HWVar tmpf2 = cond318 ? stream.offset(f2m, 1) : f2m;
// tmpf2[i0] = f2[I2D(ni,i,jm1)];

HWVar tmpf3pom = cond318 ? stream.offset(f3, +2) : stream.offset(f3, +2);

HWVar tmpf3 = sel_h ? tmpf3pom : f3;  // tmpf3[i0] = f3[I2D(ni,ip1,j)];

...

HWVar tmpf8pom = cond318 ? f8p : stream.offset(f8p,-1);

// tmpf8[i0] = f8[I2D(ni,im1,jp1)];

HWVar tmpf8 = sel_l ? tmpf8pom : f8new; // tmpf8[i0] = f8[I2D(ni,im1,jp1)];

...

io.output("tmpf1", tmpf1, hwFloat(8, 24));

io.output("tmpf2", tmpf2, hwFloat(8, 24));
```

```
    ...

} }
```

Слика 26: Имплементација Lattice-Boltzmann *stream* функције за архитектуру засновану на протоку података.

Морамо признати да поред велике функционалности које обезбеђују алати и методе, оптимизација извршења алгоритма на архитектури заснованој на протоку података и даље зависи од програмера. Слика 27 показује главни принцип имплементације LBM метода на изабраној архитектури заснованој на протоку података. Да би се израчунали параметри течности у једној елементарно малој запремини простора (2D или 3D), језгро мора да зна параметаре околних елементарних запремина. Стога, резултат извршења језгра зависи не само од тока елемената једне колоне која садржи параметре одговарајућих елементарних запремина, већ и од два тока суседних колона. Осим тога, елемент тока ће зависити од свог претходника и наследника, који се могу добити применом Maxeler окружења, уводећи индексе у токове (у нашем случају -1 и +1, означавајући претходни и следећи елемент тока респективно).



Слика 27: Процесирање елемената заснованог на току података.

Слика 28 приказује сврху LBM менаџера. Менаџер је одговоран за међусобно повезивање језгара, али и процесора и језгара. Иако Maxeler окружење аутоматски направи менаџера апликације, свако прилагођавање захтева да програмер мења аутоматски генерисаног менаџера. Алгоритам за LBM метод итерира кроз све елементарне запремине много пута (*maxIter* пута). Само крајњи резултат је оно за шта је корисник заинтересован. Према томе, сва обрада се може урадити на хардверу заснованом на протоку података, пошто хардвер заснован на протоку података има довољно меморије за смештање свих матрица потребних за извршење алгоритма. У овом случају, важна одлука је била да се сви међурезултату чувају у меморији на Maxeler картици, како би се уштедело на времену које би било потребно за пренос ових података између процесора и DFE напред и назад. То значи да треба дохватити податке из главне меморије на самом почетку, а улаз језгра треба да буде преузет током из меморије хардвера заснованог на протоку података за време извођења самих итерација. Слично томе, резултати итерација треба да се чувају у меморији хардвера заснованог на протоку података, док је само на самом крају потребно пренети податке који представљају резултат извршења алгоритма у главну меморију. Стога, поред дефинисања токова између језгара и процесора, мултиплексери су морали бити уведени, којима би се дириговало када је потребно извршити учитавање података са процесора преко тзв. *Peripheral Component Interconnect Express* (PCIe) картице, затим кад је потребно учитавати податке из DRAM меморије која је на картици заснованој на протоку података, као и кад је потребно вратити резултат извршења алгоритма у главну меморију.

```
public class LBMManager extends CustomManager {

  LBMManager(MAXBoardModel board_model, String name, Target is_simulation) {

    super(board_model, name, is_simulation);

    kernelBlock k1 = addkernel(new

      Streamkernel(makekernelParameters("Streamkernel")));

    kernelBlock k6 = addkernel(new
```

```
    Collidekernel(makekernelParameters("Collidekernel")));

Demux split1 = demux("split1");

...

split1.getInput() <== addStreamFromHost("u1");

    // Stream k1_f1  = split1.addOutput("f1_u1");

...

Mux join1 = mux("join1");

...

addStreamToHost("i1") <== join1.getOutput();

...

k1.getInput("f1")  <== split1.addOutput("k1_u1");

...

join1.addInput("k1_i1") <== k1.getOutput("tmpf1");

...

k6.getInput("f1i") <== split1.addOutput("k6_u1");

...

join1.addInput("k6_i1") <== k6.getOutput("f1o");

...

Stream tread = addStreamFromOnCardMemory("tread", kt.getOutput("rdTest1"));

kf.getInput("inData") <== tread;

Stream twrite = addStreamToOnCardMemory("twrite", kt.getOutput("wrTest1"));
```

```
        twrite <== kf.getOutput("outData");

        //kernel to host

        Stream toHost = addStreamToHost("outData");

        toHost <== k2.getOutput("outData");

        //DRAM to host

        Stream fromDRAM = addStreamFromOnCardMemory("fromDRAM",

              k2.getOutput("rdInMyDRAM"));

        k2.getInput("inData") <== fromDRAM;

        //host to kernel

        Stream fromHost = addStreamFromHost("inData");

        k1.getInput("inData") <==fromHost;

        //kernel to DRAM

        Stream toDRAM =

             addStreamToOnCardMemory("toDRAM",k1.getOutput("wrOutMyDRAM"));

        toDRAM <== k1.getOutput("outData");

    }

}
```

Слика 28: Lattice-Boltzmann менаџер.

3.6. Евалуација перформанси

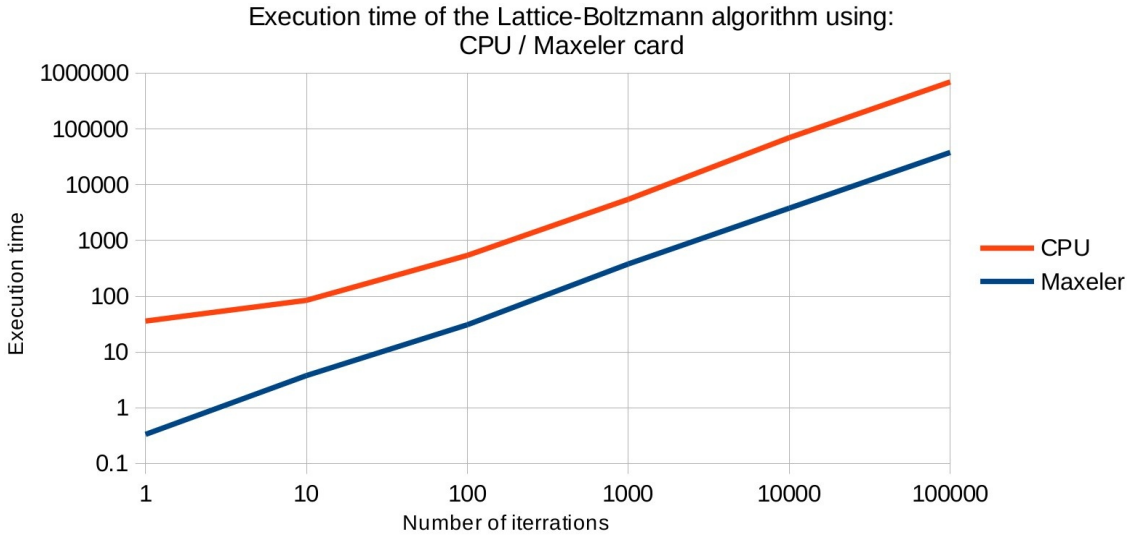
Резултати поређења имплементације LBM метода за процесор заснован на контроли тока и имплементације за архитектуру засновану на протоку података ће бити дате у три области: брзина, сложеност и моћ.

3.6.1 Студија случаја: имплементације LBM метода за процесор заснован на контроли тока и за процесор заснован на протоку података

Као што се види из кратког прегледа имплементације Lattice-Boltzmann методе за архитектуру процесора заснован на контроли тока у C програмском језику датог у претходном поглављу, главна петља се састоји од две функције које се извршавају највећи део укупног времена извршавања алгоритма. Функција *stream* захтева дохватање свих елемената матрице, па самим тим дохватање елемената представља уско грло система. Међутим, она не захтева много рачунања. Функција *collide* захтева и дохватање свих елемената и одређену количину обраде за сваки од елемената. Укупно време извршења апликације се може грубо поделити у суму износа укупног времена извршења ове две функције. Време извршења функције *collide* је око 1,5 пута веће него време извршења функције *stream*. Поређењем времена извршења ове две функције, можемо видети да време обраде елемената у функцији *collide* траје око 50% од времена потребног за дохватање елемената.

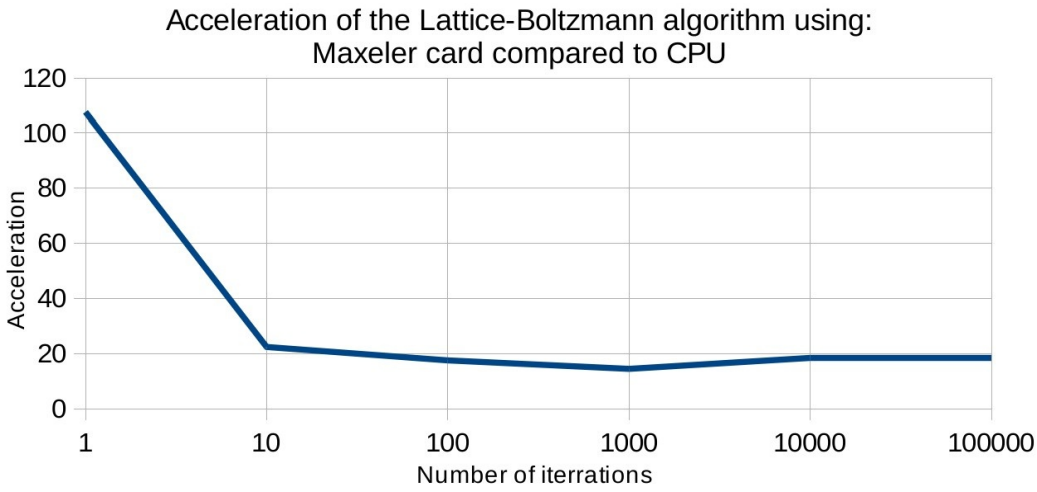
Упоређено је време извршења одговарајућих имплементација LBM метода користећи MAX2 картицу са 6GB RAM меморије и Intel i5 650 процесора са брзином такта од 3.2GHz. Рачунар поседује 4GB RAM меморије која ради брзином од 1333MHz. Резултати поређења времена извршења показују да је убрзање коришћењем хардвера заснованог на протоку података у односу на примену искључиво процесора заснованог на контроли тока у случају LBM метода у реалним случајевима око 17. Слика 29 представља поређење времена извршења

одговарајућих имплементација LBM алгоритма за различите бројеве итерација петље.



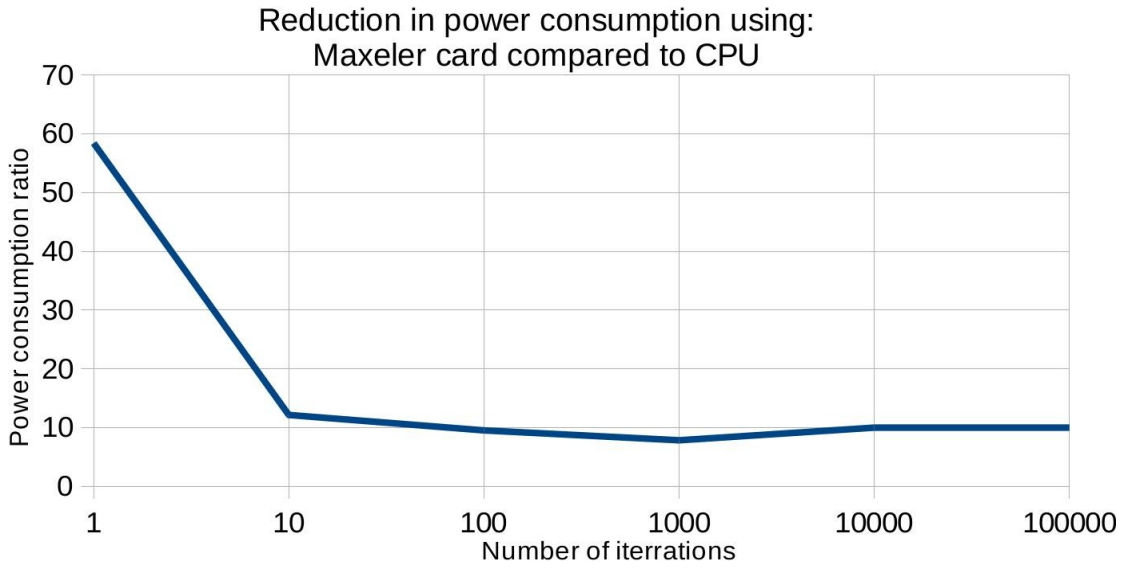
Слика 29: Поређење времена извршења Lattice-Boltzmann алгоритма коришћењем процесора заснованог на контроли тока и коришћењем Maxeler картице.

Слика 30 показује одговарајуће убрзање. У поређењу са временом извршења алгоритма само помоћу процесора заснованог на контроли тока, резултати показују фактор убрзања од око 17 у реалном сценарију. Без узимања у обзир времена потребног за слање података на Maxeler картицу и враћања резултата, у случају да се врши само једна итерација петље, фактор убрзања је већи од 100. Време извршења на Maxeler картици је скоро пропорционално броју итерација, док процесор после прве итерације извршава итерације знатно брже. Ово се може објаснити временом потребним да процесор дохвати податке из главне меморије у кеш меморију. Треба имати у виду да за довољно велики проблем, смештање свих података у кеш меморију не би било могуће. У том случају, очекивани фактор убрзања би био око 100. Ипак, треба имати у виду и то да је проблем величине кеш меморије такође могуће решити применом парадигме засноване на контроли тока, коришћењем кластера или тзв. CUDA програмирања nVidia графичких картица. Међутим, то би захтевало додатно време потребно за пренос података између процесора и главне меморије.



Слика 30: Постигнуто убрзање извршавања Lattice-Boltzmann алгоритма на FPGA, у односу на извршење на процесору.

Потрошња електричне енергије је процењена применом следеће методе. На основу [75], можемо претпоставити да је потрошња електричне енергије десктоп рачунара без MAX2 картице 61W при мировању, при чему рачунар привидно ништа не ради осим опслуживања оперативног система, а 76W под оптерећењем. Потрошња са MAX2 картицое је 86W при мировању, а 118W под оптерећењем. Сам процесор троши око 51W при мировању, а 91W када је потпуно оптерећен. Поредећи време извршења функција *stream* и *collide*, може се израчунати да је процесор у стању мировања око 20% времена, што доводи до процењене просечне потрошње од 64 вата. Десктоп рачунар са Maxeler MAX2 картицом троши 118W. Стога, прорачунат фактор снаге је око 1.84. На овај начин, добија се график са слике 31.



Слика 31: Поређење потрошње електричне енергије Lattice-Boltzmann алгоритма на FPGA, у односу на потрошњу на процесору.

Остварен фактор смањења потрошње електричне енергије је око 8.5. Иако смањење потрошње електричне енергије није тако велико као потенцијално смањење при преласку са архитектуре процесора заснованог на контроли тока на архитектуру засновану на протоку података, треба имати у виду да се пуно снаге и даље троши на процесор. Међутим, ако би процесни елементи архитектуре засноване на протоку података били оптимално окупирани од стране алгоритма (на пример тако што би се ажурирао менаџер тако да се извршава много процеса паралелно), могли би да очекујемо веће уштеде у потрошњи електричне енергије. На пример, суперкомпјутер Maxeler MPC-X заснован на протоку података је способан да извршава инструкције брзином од 8.97 GFLOPs/W, што представља перформансе по вату упоредиве са рачунарима на самом врху Green500 листе данас.

3.6.2 Убрзање других алгоритама употребом архитектура заснованих на протоку података

Истраживачи су постигли значајна убрзања користећи парадигму засновану на протоку података. Овде ћемо представити неке од њих које се обављене такође употребом Maxeler окружења.

Стојановић је решио једначину Gross-Pitaevskii користећи архитектуру засновану на протоку података, прецизније користеће Maxeler окружење и одговарајући хардвер, постижући убрзање од 8.2, у односу на одговарајућу имплементацију алгоритма за процесор заснован на контроли тока [75]. Да би се постигло ово убрзање, преуређивање података је било потребно. Као што је већ описано, овај метод узастопно рачуна вредности елемената који не зависе један од другог. Ако је проблем довољно велик, елементи чија калкулација зависи од раније израчунатих вредности ће имати спремне податке у тренутку када они буду потребни, под условом да је било довољно обраде независних елемената у међувремену. На пример, ако сваки елемент, осим елемената прве колоне, зависи од претходног елемента у истој врсти, при чему су редови међусобно независни, уместо рачунања вредности елемената ред по ред, вредности елемената се могу рачунати колону по колону.

Кос је постизао убрзања између од 100 до 150 за сортирање више од 1000 низова узастопно коришћењем имплементације Odd-even merge network алгоритма за архитектуру засновану на протоку података [76, 77]. Низови се састојала од 16 до 128 елемената; убрзање се израчунава као однос између времена за извршење сортирања на процесору и времена извршења сортирања на DFE. Odd-even merge sort алгоритам за сортирање сортира елементе низа у 10 до 28 корака цикличном поделом низа у два дела, затим сортирањем сваке половине независно, а затим спајањем резултата сортирања одговарајућих половина, почевши од најмањих. Убрзање је постигнуто паралелизовањем свих калкулација у сваком кораку Odd-even merge network sort алгоритма за сортирање.

Станојевић [78] је остварио убрзање од 18 до 24 пута у односу на импелентацију Spherical code design based on the Variable repulsion force метода за процесор заснован на контроли тока. Већина операција се обавља у једној петљи и довлачење података се може обавити паралелно са обрадом. У представљеној реализацији, аутори су модификовали сам алгоритам чинећи га скалабилним, тако да метод није идентичан оригиналном методу, али такође функционише на принципу итерирања ка решењу. Да би поређење било валидно, извршен је одговарајући број итерација, како би резултат извршавања алгоритма употребом хардвера заснованог на протоку података био барем исте прецизности као резултат извршавања алгоритма употребом процесора заснованог на контроли тока.

Бежанић [79] је имплементирао RSA алгоритам користећи Maxeler окружење. Забележио је убрзање од 25 до 30% за датотека веће од 40MB. Иако убрзање коришћењем DFE није толико добро као у претходним случајевима, ово је и даље добар резултат, јер према амдаловом закону, само део времена извршења који се троши на инструкције која се могу паралелизовати се може убрзати.

Табела 5 даје преглед постигнутих убрзања ових алгоритама користећи хардвер заснован на протоку података који поседује меморију на себи.

Табела 5: Фактори убрзања алгоритама имплементираних за хардвер заснован на протоку података у поређењу са одговарајућим имплементацијама за процесоре засноване на контроли тока.

Научни рад	Назив алгоритма	Фактор убрзања	Неопходни услови који морају бити задовољени
Станојевић [75]	Gross-Pitaevskii	8.2x	-
Кос [76]	Odd-even merge network sort	100-150x	Више од 1000 низова
Королија [74]	Lattice-Boltzmann	17x	Димензије матрице 320x112
Станојевић [78]	Spherical code design	18-24x	3<=D<=6

Бежанић [79]	RSA	1.25-1.30	Величина фајла мора бити већа од 40MB

Трифуновић је у свом раду „Paradigm Shift in Big Data SuperComputing: DataFlow vs. ControlFlow" забележио убрзања различитих апликација применом хардвера заснованог на протоку података у односу на одговарајуће имплементације за процесоре засноване на контроли тока, укључујући и CUDA програмирање [80].

Поредећи факторе убрзања приказаних апликација, можемо видети да је време извршавања коришћењем парадигме засноване на протоку података углавном један до два реда величине мање у односу на одоговарајуће имплементације за процесоре засноване на контроли тока. Алгоритми који често размењују податке између елемената за обраду (нпр. *Odd-even merge network sort*) имају тенденцију да поседују веће могућности за убрзање, јер се сви ови преноси података могу урадити паралелно, док са друге стране процесор заснован на контроли тока мора дохватити сваку променљиву из меморије и сачувати резултат у меморију.

3.6.3 Претње валидности

Овде је представљена разлика између парадигми процесора заснованих на контроли тока и парадигме засноване на протоку података, откривајући очигледне могућности за убрзавање одређене врсте апликација које користе парадигму засновану на протоку података. Међутим, представљени резултати подлежу поређењу различитих архитектура и организација рачунара, користећи различите врсте меморија, итд. Аутори представљених апликација су покушали да упореде време извршења алгоритама на Maxeler картицама и процесорима заснованин на контроли тока који су слични по датуму производње, обично извршавајући одговарајуће алгоритме за процесоре засноване на контроли тока и верзије алгоритама засноване на протоку података на истом рачунару, са и без употребе хардвера заснованог на протоку података.

У овој тези, коришћен је као пример LBM метод који ради са матрицама које се могу сместити у кеш меморије данашњих рачунара. Порастом величине проблема изнад величине кеш меморије, фактор убрзања реализације Lattice-Boltzmann алгоритма за архитектуру засновану на протоку података би био много већи.

Циљ је био да се представе потенцијали парадигме засноване на протоку података. Пошто је постигнуто убрзање много веће од фактора убрзања две узастопне генерације процесора, верујемо да читалац може да прихвати изнете резултате са опрезом, и једноставно прихвати концептуалну разлику у овим рачунарским парадигмама.

3.7. Закључак

Као резултат поређења имплементација претходно описаних алгоритама, могло би се рећи да би архитектуре засноване на контроли тока ускоро могле бити застареле у употреби у рачунарству високих перформанси. Много језгара процесора заснованих на контроли тока нису увек довољан услов за убрзање апликација због удаљености између процесора, док је потрошња електричне енергије знатно већа него што је теоријски потребно за функционалност које овакве архитектуре и организације рачунара обезбеђују. Парадигма заснована на протоку података решава ове проблеме, али њихова примена није изводљива за потребе извршавања већине програма кућних рачунара, као ни неких од проблема алгоритама који се извршавају на данашњим рачунарима високих перформанси. Такође, алгоритми обично поседују део који се односи на иницијализацију података, за чије извршавање парадигма заснована на протоку података није повољна. Стога, оптимално решење за већину рачунарских проблема високих перформанси је хибридно решење.

Метод Lattice-Boltzmann је један од најчешће коришћених метода за симулирање динамике флуида. Може се лако трансформисати у циљу паралелизације извршавања. Полазећи од C++ кода, анализирањем потенцијала за убрзање, може се наћи понављање итерација главне петље која чини готово сво време

извршавања алгоритма. У току извођења сваке итерације, одређеним матрицама приступају функције *stream* и *collide*. Овде је представљена имплементација метода Lattice-Boltzmann за архитектуру засновану на протоку података која је развијена коришћењем Maxeler окружења као средства које комбинује многе расположиве методе из теорије систоличких низова. Као следећи корак, за извршавање итерација које се понављају употребом Maxeler картице дизајнирана су одговарајућа језгра, као и пренос података од и до процесора заснованих на контроли тока после сваке од итерација. Међутим, резултати су држани у главној меморији, па су стално требали да се шаљу од процесора ка хардверу заснованом на протоку података и назад. Следећа промена алгоритма се односила на употребу меморије Maxeler картице уместо слања података између процесора и Maxeler картице после сваког понављања.

Користећи парадигму засновану на протоку података, симулација протока флуида може да се уради брже са мањом потрошњом електричне енергије, иако FPGA раде на мањим фреквенцијама и део чипа одговрног за извршење једне инструкције није у стању да уради више него саму ту инструкцију. Поред веће брзине извршења на FPGA, остварује се и мања потрошња електричне енергије. Такође је показано да се многи други алгоритми могу убрзати помоћу парадигме засноване на протоку података.

Додатан рад је потребан да би се програмирање хардвера заснованог на протоку података учинило још лакшим, ослобађајући програмера потребе да знањем детаља хардвера на ком се алгоритам извршава.

Понекад радна снага колоније мрава више одговара послу него један слон. И у рачунарству, обично се примењује опште правило: што ближе – то брже.

4. Алгоритми за прављење распореда извршавања послова

У овој глави ће најпре бити дата подела алгоритама за прављење распореда, након чега ће бити представљени неки од постојећих алгоритама за прављење распореда извршавања послова. Затим ће бити описани предложени алгоритми уз резултате поређења времена извршавања.

4.1. Постојећи алгоритми за прављење распореда извршавања послова

Распоређивање послова на FPGA је већ истраживан проблем у литератури [81-85]. Начини распоређивања могу се поделити у:

- Статичко прављење распореда, или прављење распореда у време компајлирања програма: брзо, али без могућности прављења распореда на основу параметара који ће бити познати тек у време извршавања програма и

- Динамичко, или прављење распореда у време извршавања програма: захтева додатну обраду у време извршавања програма и суб-оптимална је у генералном случају услед немогућности анализирања свих могућих распореда.

Abdessamad је у свом раду [81] моделирао математички проблем распоређивања у хетерогеној рачунарској архитектури која укључује FPGA, како би се у реалном времену постигла способност прављења распореда неопходна за симулације у индустрији авиона. Он представља алгоритам за прављење распореда заснован на тзв. *Mixed integer* принципу за решавање проблема распоређивања послова под претпоставком да су познати пре заказивања сваког од послова граф топологије, времена извршења и параметри комуникације. Његов алгоритам је у стању да распореди до 50 послова у року од само неколико секунди.

Fei је предложио алгоритам за прављење распореда заснован на решавању тзв. s*hop scheduling problema*. Сам његов интелигентан алгоритам оптимизације је имплементиран помоћу FPGA.

Распоређивање послова на процесор и хардвер заснован на протоку података може се третирати као тзв. *resource constrained project scheduling problem* (RCPSP) [83]. Zheng и Wang су предложили више-агентни оптимизациони алгоритам за RCPSP [86], где више агената који представљају потенцијална решења раде као група агената.

Hamilton [84] је представио *multitasking* систем за рад у реалном времену који се извршава на FPGA-CPU платформи, где се корисничке апликације извршавају као процеси за мешовите архитектуре. Њихов систем такође подржава промену контекста (енг. c*ontext switching*), као и механизме блокирања и прекидања послова и преузимања процесора од стране послова. Hamilton је за циљ имао подршку паралелног извршења процеса мешовите архитектуре.

Khuat [85] је увео просторно-временско прављење распореда за зависне задатке који се извршавају на хетерогеној FPGA архитектури како би се смањило време реконфигурације тако што би се послови унапред учитавали. Предложено решење за прављење распореда ради у реалном времену, односно у току извршавања послова.

Многи алгоритми за прављење распореда послова су дизајнирани да распоређују задатке на хардвер заснован на протоку података. Међутим, већина од њих су дизајнирани за статичко распоређивање послова на FPGA површину у две димензије. Cilardo [87] је предложио методологију за оптимизовање вишејезгарних архитектура заснованих на FPGA. Тренутни распоред који Maxeler окружење пружа је оријентисан према ефикасном коришћењу хардвера за извршавање једног алгоритма. Овде предложени алгоритам за прављење распореда није дизајниран за оптимизовање прављења распореда за рачунаре високих перформанси, већ за ефикасно коришћење процесора заснованих на контроли тока и хардвера заснованог на протоку података за покретање више апликација истовремено.

Iturbe [88] разматра распоређивање задатака у реалном времену помоћу FPGA. Он представља осам једноставних алгоритама, погодних за релативно велик број послова чија су трајања релативно мала. Претпоставка је да ће у будућности бити потребна сложенија обрада, што ће захтевати боље распоређивање, а да ће истовремено веће време за распоређивање бити на располагању.

У литератури се могу наћи два најчешће коришћена приступа за распоређивање послова са непредвидљивим трајањем. Један од њих се заснива на израчунавању иницијалног распореда пре самог извршавања послова и мењања (ако је потребно) у току извршавања послова [89]. Други приступ је да се дизајнирају правила за прављење распореда у циљу пружања могућности да се релатвино брзо у време извршавања послова одлучи који послови треба да буду започети и који ресурси требају да им буду додељени [90]. У генералном случају распоређивања послова није могуће направити распоред у време превођења због непознатих трајања послова, као и непознатих тренутака када ће се послови јавити. Други приступ боље одговара прављењу распореда у генералном случају. Стога, пре него што се покрене извршавање послова и прављење распореда, могуће је утврдити који послови могу стати заједно на један FPGA. Надаље ћемо послове који су смештени на један FPGA називати FPGA сликом. Касније, у време извршавања послова, у зависности од величине проблема који сваки од послова мора обрадити, програмер може одабрати најприкладнију FPGA слику и распоред послова који ће бити извршени на хардверу заснованом на протоку података у складу с тим.

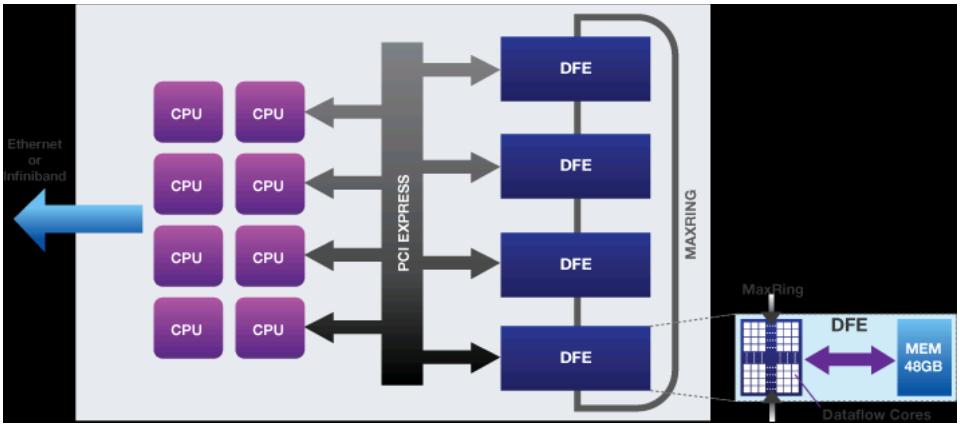
4.2. Предложени алгоритми за прављење распореда извршавања послова

Овде ће бити описане хеуристички алгоритам за прављење распореда заснован на тзв. *branch-and-bound* алгоритму који има за циљ да распореди послове на FPGA слике, тако да се минимизује укупно време извршења свих апликација.

Неки од данашњих најнапреднијих рачунара који су тренутно у употреби само у областима као што су финансијске услуге и сектору енергије у индустрији могли

би да постану свакодневни рачунари у блиској будућности. Услед тога, циљна архитектура и организација рачунара ове тезе је рачунар који поседује хардвер заснован на протоку података који обрађује релативно велике низове података. У рачунару заснованом на протоку података, сваки део чипа је у суштини изграђен за посебне функционалности. На тај начин многе инструкције могу да раде паралелно. У овој тези се разматра реконфигурабилна архитектура рачунара, где се задаци постављају на хардвер и то на скуп DFE јединица, од којих свака представља реконфигурабилни чип са релативно високом количином меморије (као на слици 32). Процесор је одговоран за слање спецификације задатака, а алгоритам за прављење распореда одређује да ли задатак треба да буде извршен на процесору или некој од DFE јединица.

Најважнија разлика у прављењу распореда за DFE и регуларне реконфигурабилне рачунаре засноване на FPGA је у чињеници да се у овом случају не додељује једно-димензионална или дво-димензионална област FPGA послу који треба да се изврши на чипу. У овом случају, додељује се одређен број DFE јединица који ће обављати обраду података. Циљ је да се истражи на који начин је најбоље да се подели реконфигурабилни хардвер као дељени ресурс који ће бити коришћен од стране различитих процесора.



Слика 32. Повезивање DFE јединица и процесора.

Као полазна тачка, развијена су два рекурзивна алгоритма за проналажење најбољег распореда, чији су принципи слични онима предложеним у постојећем раду [91]. Први алгоритам има за циљ да минимизује укупно време извршења

свих апликација. Другом је циљ да повећа проток колико год је то могуће. Оба су заснована на истим принципима и разлика између њих је само у процени вредности другачијих сценарија. Стога, приликом објашњавања основних принципа, говориће се о алгоритму за прављење распореда, јер ће се само оцењивање (бодовање) поменута два алгоритма разликовати. Полазећи од празног скупа послова укључених у потенцијални распоред, покушава да се укључи сваки посао за архитектуру засновану на протоку података у сет. За сваки потенцијални распоред, рачуна се резултат, на основу односа укупног времена извршења свих послова у случају да се сви раде на процесору и у случају да се раде у складу са распоредом који је алгоритам за прављење распореда направио. За сваки потенцијални распоред, исти поступак се понавља. Затим се додају нови послови за потенцијалне распореде, вреднују резултати, итд. Распоред са највећом оценом се сматра најбољим распоредом. Треба имати у виду да потенцијални распоред може укључивати више од једне FPGA слике. Стога, резултат за потенцијални распоред првог планера се израчунава као уштеда у времену извршења између случаја када се сви послови извршавају користећи само процесор и случаја када се користи потенцијални распоред, где су послови оптимално распоређени на произвољни број FPGA слика, тако да се максимизује укупна уштеда времена извршавања свих послова. За други планер, укупан проток мора имати максималну могућу вредност. Пропусна моћ се дефинише као укупно време извршења свих послова планираних за хардвер заснован на протоку података у случају да их ради процесор подељеним са укупним временом извршења свих FPGA слика. Ови алгоритми нису корисни у прављењу распореда извршавања послова у време извршавања послова услед чињенице да је само ограничен број послова могуће заказати, пре него што време потребно за прављење распореда постане превелико и самим тим прављење распореда неисплативо. Међутим, они су и даље корисни за оцењивање других распореда. Још један показатељ који ће се користити за процену распореда је поређење са временом које је потребно за процесор да обради све послове које су били предвиђени за извршавање на хардверу заснованом на протоку података.

На основу рекурзивног алгоритма за прављење распореда за проналажење најбољег решења, дефинисан је алгоритам који користи хеуристику ради

временски ефикаснијег прављења распореда који се заснива на прављењу распореда заснованом на поенима. За сваки посао који може бити распоређен и на процесор и на хардвер заснован на протоку података, додељује се одређена оцена изражена у поенима. Број поена зависи од:

- Времена које је потребно за процесор заснован на контроли тока да обради посао,

- Времена потребног за хардвер заснован на протоку података да обради исти посао,

- Времена потребног за иницијализацију посла, како би могао да се извршава на хардверу заснованом на протоку података,

- Количине ресурса које би посао конзумирао у случају извршења свих послова користећи процесор заснован на контроли тока, и

- Количине ресурса које би посао конзумирао у случају извршења свих послова користећи и хардвер заснован на протоку података.

Пошто FPGA може да се конфигурише да извршава многе послове истовремено, а времена потребна за послове који се обрађују се разликују једно од другог, додељивање броја поена мора да се врши у време извршавања.

Главно ограничење рекурзивног алгоритма за прављење распореда је број послова који се могу узети у обзир за прављење распореда послова за хардвер заснован на протоку података. Због тога је алгоритам модификован тако да бира само k послова из n слободних послова, где $k \leq n$, као што је приказано на слици 33. Након што се изврши m послова, где $m \leq k$, алгоритам за прављење распореда извршавања послова се поново позива да направни распоред преосталих k послова из реда послова на располагању за распоређивање. Бројеви k и m су експериментално одређени. Треба имати у виду да за $k = m$ алгоритам враћа најбољи распоред. Иако постоје итеративни алгоритми за проналажење k подскупа од n елемената, алгоритам заснован на рекурзији више одговара у случају када већина грана у претрази по дубини не могу да произведу бољи резултат од оног који је већ нађен. Још једна важна модификација у алгоритму за прављење

најбољег распореда је да за сваки посао који треба додати потенцијалом распореду, два различита сценарија треба узети у обзир. Први подразумева додавање посла на тренутно посматрану FPGA слику. Ограничење које мора бити испуњено је да је укупан проценат сваког ресурса који посао заузима расположив на посматраној FPGA слици, тј. да је за сваки ресурс сума процената тог ресурса која је потребна свим пословима укључујући и посао који се тренутно разматра мања 100%. Ресурси укључују количину односно број DFE јединица, и PCIe магистралу, као и количину главне меморије и количину меморије на картици заснованој на протоку података. Ако се посао може додати на тренутно посматрану FPGA слику без утицаја на време извршавања FPGA слике, односно најдужег посла који је предвиђен за извршавање на тој FPGA слици, разматра се други сценарио који се састоји у додавању новог посла на нову FPGA слику.

Сет задатака који је на располагању за прављење распореда у датом тренутку *t* се дефинише као $T_t = \{T_{t1}, T_{t2}, \ldots T_{tntasks}\}$, где је *ntasks* број послова доступних у тренутку *t*.

Најбољи распоред се дефинише као:

$B = \{T_{tb1}, T_{tb2}, \ldots T_{tbm}\}$, где за сваки тренутак *t*, T_{tbi} представља *i*-ти посао у најбољем распореду који садржи *m* послова.

Функција *DFE* враћа проценат ресурса DFE (%DFE) који је потребан послу или скупу послова прослеђеним као аргумент функције. Функција *PCI* враћа проценат PCIe (%PCIe) који је потребан послу или скупу послова прослеђеним као аргумент функције. Функција *mem* проверава да ли има довољно главне меморије и меморије на картиці заснованој на протоку података да бе се извршио посао и сет послова послатих као аргумент функцији.

```
T0 = {T0₁, T0₂,...T0ₙ} // available tasks for scheduling
FPGAimages = {} // candidate FPGA images set
                // whose elements are set of jobs
nFPGAimages = 0 // number of FPGA images
                // used by jobs in the candidate schedule
bestFPGAimages = {} // the best schedule found so far
```

```
find_best(k) {
  if k=0
    then{
      if score(C) > score(B)
        then B = C
    }else{
      for all tasks t in T0 {
        T0 = T0 - t
        if( (PCIe(t) + PCIe(FPGAimages[nFPGAimages]) < 100) &&
          (DFE(t) + DFE(FPGAimages[nFPGAimages]) < 100)  ) &&
          (mem(t, FPGAimages[nFPGAimages]))  ) {
          FPGAimages[nFPGAimages] = FPGAimages[nFPGAimages] + t
          find_best(k-1)
          FPGAimages[nFPGAimages] = FPGAimages[nFPGAimages] - t
        }
        nFPGAimages = nFPGAimages + 1
        FPGAimages[nFPGAimages] = FPGAimages[nFPGAimages] + t
        find_best(k-1)
        FPGAimages[nFPGAimages] = FPGAimages[nFPGAimages] - t
        nFPGAimages = nFPGAimages - 1
        T0 = T0 + t
      }
    }
}
```

Слика 33: Хеуристички алгоритам за налажење најбољег распореда.

Функције за бодовање алгоритмама заснованих на хеуристици су модификоване тако да се време потребно за извршење свих послова укључених у једну FPGA слику множи са фактором скалирања, за који је подељен са емпиријски добијеним бројем 1.1. Исти поступак се понавља за сваки FPGA. На тај начин, укупно смањење у времену извршења коришћењем FPGA слике се вреднује више него за исто смањење времена приликом извршења следеће FPGA слике.

Алгоритам за прављење распореда на самом почетку разматра све задатке који нису укључени у распоред. За дату дубину *k*, алгоритам рекурзивно тражи све могуће комбинације са понављањем послова у распореду, придружујући резултат

свакој од њих. Онај са највећим бројем бодова се третира као најбоље могуће решење. Првих *m* послова из распореда (уколико постоји) бива заказан за извршавање. На тај начин, послови који се налазе у најбољем распореду можда неће постојати у следећем најбољем распореду.

Замислимо сценарио да се већина послова може извршити помоћу хардвера заснованог на протоку података, где алгоритам за прављење распореда може да изабере било којих *k* из релативно великог броја доступних послова за распоређивање *n*, што би водило ка релативно великом времену прављења распореда. Како би се време прављења распореда што више смањило, сваком послу би могао да буде додељен почетни број поена. Овај резултат представља колико добро би било да се закаже посао на хардверу заснованом на протоку података без узимања у обзир послова који су већ додељени хардверу заснованом на протоку података. Ако се неки послови могу много више убрзати од других који користе хардвер заснован на протоку података уместо процесора, постоји већа вероватноћа да би управо такве послове требало извршавати коришћењем хардвера заснованог на протоку података. Једноставна модификација полазног рекурзивног планера је да се приликом прављења распореда послови прво сортирају према иницијалном броју бодова. Тако би планер прво разматрао извршавање на хардверу заснованом на протоку података послове са највишим оценама. На овај начин, приликом анализирања распореда који укључује послове за које није вероватно би довели до најбољег распореда, за очекивати је да ће релативно брзо доћи до заустављања рекурзивног тражења најбољег распореда пре достизања дубине *k*. Почетна оцена, односно број бодова, је пре стартовања извршавања послова израчуната као вредност смањења времена извршења посла користећи хардвер заснован на протоку података у односу на време извршења посла користећи процесор. Затим је ова вредност подељена фактором који представља однос ресурса које би посао конзумирао уколико би се извршавао на хардверу заснованом на протоку података, након чега се множи са разликом у количини ресурса који би се трошили на процесору у случају извршавања посла на хардверу заснованом на протоку података и у случају извршавања на процесору.

Претходне измене се могу користити и за заказивање послова у време превођења, и складиштење најбољег сценарија за сваку комбинацију послова у табели индексираној по хеш вредностима које се утврђују у зависности од сета послова који су били на располагању за заказивање. Иако би то решење било релативно брзо у неким случајевима, овај приступ није увек довољно добар да би било исплативо користити га. На пример, ако процесор има само малу количину RAM меморије, послови који троше превише меморије треба да се радије ставе на хардвер заснован на протоку података. Због тога се почетно додељен број бодова модификује у време извршавања послова. У зависности од тога шта је уско грло система у тренутку прављења распореда извршавања послова, треба повећати почетни број поена за послове који захтевају мање ресурса који представљају уско грло система.

Понекад, уколико је потребно направити распоред за два посла са различитим трајањем тако да деле хардвер заснован на протоку података, чак и ако оба им доступна количина ресурса то омогућава, може бити боље да се закаже сваки од њих појединачно, односно да се евентуално комбинују други послови са ова два. Због тога је важно не само да се изабере подскуп од *k* послова за заказивање, већ и да се одреди који ће послови требати да се пакују и извршавају заједно. Међутим, уколико само одређени посао може да стане на хардвер заснован на протоку података заједно са другим пословима предвиђеним за извршавање на хардверу заснованом на протоку података, а да при томе тај посао траје мање од најдужег времена извршавања послова предвиђених за хардвер заснован на протоку података, онда се тај посао одмах заказује за извршавање заједно са другим пословима. Ово смањује рачунање потребно за прављење распореда послова.

Главни део алгоритма за прављење распореда је имплементиран у програмском језику C++ и дат је на слици 34.

```
if (currentScore > bestScore) {
  bestScore = currentScore;
  bestSchedule = currentSchedule;
  bestScheduleFPGAs.clear();
  for (auto &it : currentSchedule)
```

```
    bestScheduleFPGAs.push_back(it->_FPGAimage);
}
if(currentSchedule.size() == MAX_SCHED_JOBS) return;
// Calculate occupation of the nFPGAimage
for (auto &it : currentSchedule)
  if( it->_FPGAimage==nFPGAimage )
    it->sum(sumPercentageDFEs, sumPercentagePCIe, sumTInitDFEs,
            sumMemDFEs, sumTDFE, sumTCPU, tDFEmax);

// Adding each job separately to the currentSchedule
for (auto &it : jobs) { // for each job
  if( it->add(&currentSchedule,
              sumPercentageDFEs, sumPercentagePCIe,
              sumMemDFEs, sumTDFE, sumTCPU,
              sumTInitDFEs, tDFEmax, nFPGAimage) ){
    score1 += it->score(); // increasing by the it job score
    double scoreForMode;
    if(mode == THROUGHPUT)
      scoreForMode = 1.0*sumTCPU/tDFEmax;
    else // (mode == TIMEGAIN) || (mode == FCFS)
      scoreForMode = sumTCPU-tDFEmax;
    // If no other job can fit on nFPGAimage
    if(cantFit(jobs, sumPercentageDFEs, sumPercentagePCIe,
       sumMemDFEs, tDFEmax))
      // Schedule the rest of jobs on nFPGAimage+1
      schedJobs(mode, jobs, currentSchedule,
                currentScore + factor*(score1 + scoreForMode),
                bestScore, nFPGAimage+1, factor/1.1);
    else
      // Schedule on the same FPGA image
      schedJobs(mode, jobs, currentSchedule,
                currentScore + factor*score1,
                bestScore, nFPGAimage, factor);

    if(mode!=FCFS)
      it->remove(&currentSchedule, sumPercentageDFEs,
                 sumPercentagePCIe,
```

```
                    sumMemDFEs, sumTDFE, sumTCPU, sumTInitDFEs,
                    tDFEmax, nFPGAimage);
    }else{
      if(mode==FCFS)
        // Schedule the rest of jobs on nFPGAimage+1
        schedJobs(mode, jobs, currentSchedule, currentScore + 1,
                  bestScore, nFPGAimage+1, factor);
    }
  }
```

Слика 34: Прављење распореда извршавања послова.

Наредна поглавља ће представити симулационо окружење са параметрима који утичу на распоред, затим различите сценарије распореда, резултате и анализу.

4.3. Опис алгоритама погодиних за хардвер заснован на протоку података и њихова убрзања коришћењем Maxeler окружења

Да би тестирали могућност убрзавања извршавања апликација користећи процесор заснован на контроли тока, хардвер заснован на протоку података и одговарајући планер, најпре је дато поређење времена извршавања различитих алгоритама на процесорима и на Maxeler хардверу заснованом на протоку података.

Истраживачи су радили на убрзавању различитих алгоритама користећи парадигму засновану на протоку података. Овде ћемо се фокусирати на оне радове који укључују примену алгоритама имплементираних за Maxeler окружење, како би било могуће применити предложени алгоритам за прављење распореда извршавања послова, тестирањем комбинација апликација које могу да се покрећу само на процесору и оних за које се може остварити убрзање помоћу Maxeler окружења.

Тзв. *Computational Fluid Dinamics* (CFD) методе [92] се користе за симулацију токова флуида. Традиционалне методе су засноване на решавању Navier-Stokes

једначина. Lattice-Boltzmann метод (LBM) моделира флуид као мрежу која се састоји од честица које обављају узастопна ширења и сударе изнова и изнова [93]. Израчунавање утицаја у свакој честици дате запремине у сваком од тренутака захтева много рачунања. Метод LBM је дизајниран да буде прилагодљив и способан за ефикасно извршавање на паралелним рачунарским архитектурама. У раду [74], Bhatnagar-Gross-Krook модел (који користи тзв. *single relaxation time approximation*) [94] се користи за симулацију протока крви. Постигнути су фактори убрзања од 15 до 100, у зависности од проблема. Међутим, није узето у обзир време потребно за припрему хардвера заснованог на протоку података за извршење алгоритма. Због тога, фактор убрзања од 100 који је постигнут само за веома мали број понављања може да се објасни као спорије време обраде процесора у случају да подаци треба да се преузму из главне меморије, уместо да се дохватају из кеш меморије. Пошто је време потребно за припрему хардвера заснованог на протоку података неколико редова величина мање од времена извршења алгоритма за прорачунавање протока крви, можемо да посматрамо фактор убрзања од 15 до 17 као релевантан, на основу графика убрзања датог на слици 30.

Gross-Pitaevskii једначина је широко коришћена у контексту Bose-Einstein кондензата. Заснована је на локалним интеракцијама између честица. Стојановић је решио нумерички једначину Gross-Pitaevskii користећи Maxeler хардвер заснован на протоку података, и постигао фактор убрзања од 8.2 у свом раду [75]. У односу на имплементацију алгоритма за процесор заснован на контроли тока, имплементација за архитектуру засновану на протоку података је захтевала преуређење улазних података, како би се искористио хардвер заснован на протоку података тако што би се у сваком циклусу читања доставио нови улазни податак. За довољно велики проблем, уместо довођења елемената једне по једне врсте, при чему би сваки елемент зависио од претходно обрађеног, успео је да обради прво само прве елементе независних врста, затим друге сваке врсте, итд. На овај начин, сви елементи су имали спремне податке када је обрада заказана.

Алгоритам за сортирање *Odd-even merge network sort* укључује релативно висок број поређења два броја која се могу радити паралелно. Дизајниро га је Batcher за

сортирање мреже величине O (*n* * (лог *n*) 2), али је стекао популарност са појавом графичких процесора (GPU). Кос је имплементирао алгоритам за сортирање *Odd-even merge network sort algorithm* помоћу Maxeler хардвера заснованог на протоку података, и постигао факторе убрзање од 100 до 150 за сортирање више од 1000 низова узастопно [76, 95]. Сваки од низова састојао се од 16 до 128 елемената. Њихова имплементација *Odd-even merge network sort* алгоритма за сортирање паралелно сортира елементе низа у 10 до 28 корака.

Сферични код је скуп *n*-димензионалних вектора на јединици сфере, чије су координате реални бројеви. Станојевић [78] се бавио проблемом паковања, који је један од два стандардна проблема оптимизације у вези са сферним кодовима. На основу великог броја вектора и димензија, њихов циљ је био да се пронађу вектори који имају максималну вредност минималне Еуклидијанове удаљености између два вектора. Достигнут је фактор убрзања од 18 до 24 пута, коришћењем Maxeler хардвера заснованог на протоку података на коме је имплементиран одговарајући код написан за процесор заснован на контроли тока. Иако је првобитни алгоритам промењен, главни принцип је остао исти, а који се заснива на итерирању ка тачном решењу.

Криптовање RSA јавним кључем (енг. *RSA public-key cryptosystems*) је засновано на производу два велика проста броја, као и чињенице да не постоји познати алгоритам који може ефикасно пронаћи просте факторе производа. Јавни кључ се слободно шири и служи за шифровање, док је дешифровање тајни процес. На овај начин, једна компанија може дати свој јавни кључ корисницима, тако да они могу да шифрују своје податке и пошаљу их натраг у компанији која је једина у стању да податке дешифрује. Бежанић [79] је имплементирао RSA алгоритам помоћу Maxeler хардвера заснованог на протоку података, постизањем убрзања од 25% до 30% за шифровање података већих од 40MB. Иако хардвер заснован на протоку података има добре потенцијале, фактор убрзања је у овом случају ограничен, јер се само део времена извршења троши на инструкцијама које се могу извршавати ефикасно коришћењем хардвера заснованог на протоку података.

Табела 5 даје преглед фактора убрзања претходно наведених апликација које користе Maxeler хардвер заснован на протоку података. Услови који морају да буду

испуњени да би резултати били важећи нису посебни случајеви, већ пре типични услови, осим за RSA алгоритам који може бити убрзан тек када се користи за шифровање релативно великих фајлова.

5. Резултати поређења алгоритама

У овом поглављу ће бити представљен начин моделовања система, а затим ће бити дат пример прављења распореда извршавања послова. Важност примера се огледа у чињеници да није лако израчунати најбољи распоред извршавања послова у произвољном случају. Зато је дизајниран пример који садржи релативно мали број послова, тако да је могуће рекурзивним алгоритмом испитати све комбинације послова, а затим тако добијене резултате упоредити са резултатима који се добијају коришћењем предложених алгоритама за прављење распореда извршавања послова.

Даље ће бити описан синтетички пакет послова, који се састоји од алгоритама које су истраживачи имплементирали коришћењем Maxeler окружења. За овакве послове, биће дати графици убрзања извршавања.

5.1. Моделовање система

Улаз за алгоритам за прављење распореда извршавања послова представља скуп послова који треба да буду извршени било на процесору или хардверу заснованом на протоку података. Сваки посао који се може извршити и на FPGA и процесору је дефинисан са следећим атрибутима:

-% DFE: Проценат DFE неопходан за извршавање задатка помоћу хардвера заснованог на протоку података

- tDFE: Време потребно за хардвер заснован на протоку података да изврши посао

- tCPU: Време потребно процесору за извршење посла

- tInitDFE: Време потребно за иницијализацију хардвера заснованог на протоку података одговарајућим VHDL фајлом

- MemDFE: Потребна количина меморије на хардверу заснованом на протоку података

- MemCPU: RAM меморија потребна за извршавање посла на процесору

- MemRest: Меморија потребна за извршавање посла на хардверу заснованом на протоку података.

Осим тога, додатни атрибут *tRest* дефинише време потребно за процесор да покрене апликацију која треба да се извршава помоћу хардвера заснованог на протоку података, као и за прикупљање резултата извршавања са хардвера заснованог на протоку података. Пошто су алгоритми нађени у отвореној литератури захтевне апликације по питању коришћења процесорског времена, при чему су алгоритми поново имплементирани користећи хардвер заснован на протоку података тако да хардвер заснован на протоку података извршава захтевни део алгоритма, док је процесор задужен само за иницијализацију и прикупљање резултата, овај параметар можемо искључити из даљег разматрања.

5.2. Пример прављења распореда

У циљу тестирања предложених алгоритама за прављење распореда, изабране су апликације које би могле бити убрзане употребом хардвера заснованог на протоку података. Табела 2 даје преглед алгоритама имплементираних помоћу Maxeler окружења нађених у отвореној литератури и њихових фактора убрзања употребом хардвера заснованог на протоку података. За сваки од ових алгоритама, формирана су по два проблема који се могу разликовати у потреби за рачунањем. На пример, у случају Lattice-Boltzmann алгоритма, времена извршења за два различита проблема могу да се разликују 10 пута. На овај начин, формиран је скуп примера послова који се могу заказати за извршавање на хардверу заснованом на протоку података. Овај скуп је дат у табели 3, при чему се време увек мери у

секундама. Неопходне количине главне меморије (*memCPU*, *memRest*) и меморије хардвера заснованог на протоку података (*memDFE*) су изостављени из даљег разматрања, без губљења општости, пошто је провера сваког од ограничења униформа. Ни главна меморија, ни меморија хардвера заснованог на протоку података нису ограничавајући фактор у случају прављења распореда извршавања ових алгоритама. Време потребно за иницирање послова за хардвер заснован на протоку података (*tInitDFE*) је обично занемарљиво у односу на време које је потребно за обраду, тако да ће и оно бити изостављено из разматрања.

Табела 6: Пример послова за алгоритам за прављење распореда.

#	Алгоритам	%DFEs (%)	%PCIe (%)	tCPU (s)	tDFE (s)
1	Gross-Pitaevskii	50	0	820	100
2	Odd-even merge network sort	50	100	1000	10
3	Lattice-Boltzmann	20	0	170	10
4	Spherical code design	40	0	1800	100
5	RSA	50	100	250	200
6	Gross-Pitaevskii	50	0	820	100
7	Odd-even merge network sort	50	100	1000	10
8	Lattice-Boltzmann	20	0	1700	100
9	Spherical code design	40	0	180	10
10	RSA	50	100	25	20

За демонстрацију примера прављења распореда, можемо изабрати да планер треба да распореди шест послова на хардвер заснован на протоку података, узимајући у разматрање скуп послова формиран на основу послова из табеле 6. Табела 7 приказује најбољи распоред у смислу укупног времена извршења свих апликација.

Таβела 7: Најβољи распоред за случај максимизовања уштеде времена.

#	Алгоритам	%DFEs (%)	%PCIe (%)	tCPU (s)	tDFE (s)	FPGA image
4	Spherical code design	40	0	1800	100	1
8	Lattice-Boltzmann	20	0	1700	100	1
1	Gross-Pitaevskii	50	0	820	100	2
2	Odd-even merge network sort	50	100	1000	10	2
6	Gross-Pitaevskii	50	0	820	100	3
7	Odd-even merge network sort	50	100	1000	10	3

5.3. Прављење распореда извршавања послова употребом предложених алгоритама

Таβела 8 приказује распоред направљен предложеним алгоритмом за прављење распореда извршавања послова који као критеријум за распоређивање користи максималну уштеду у времену извршавања свих апликација. За случај када се распоред прави техником први-дошао-први-опслужен (енг. *First-Come-First-Serve - FCFS*), имали βисмо списак послова који су на располагању за заказивање на хардверу заснованом на протоку података. Планер ће заказати прве послове за извршење на првој FPGA слици, пакујући онолико послова колико може стати истовремено на хардвер заснован на протоку података. Распоред ће βити исти као и првих 6 послова из таβеле 3, где ће по два посла да се пакују на сваку FPGA слику. Анализа овако доβијених резултата ће βити описана накнадно. На жалост, Maxeler окружење нема планер за заказивање више од једног алгоритма на једној FPGA слици.

Табела 8: Распоред извршавања послова направљен предложеним алгоритмом за максималну уштеду укупног времена извршавања.

#	Алгоритам	%DFEs (%)	%PCIe (%)	tCPU (s)	tDFE (s)	FPGA slika
4	Spherical code design	40	0	1800	100	1
8	Lattice-Boltzmann	20	0	1700	100	1
9	Spherical code design	40	0	180	10	1
1	Gross-Pitaevskii	50	0	820	100	2
2	Odd-even merge network sort	50	100	1000	10	2
7	Odd-even merge network sort	50	100	1000	10	3

У табели 9 је представљен распоред направљен алгоритмом за максимизовање протока. Најбољи распоред који максимизује проточност би изгледао исто.

Табела 9: Распоред предложеног алгоритма за прављење распореда који максимизује проток.

#	Алгоритам	%DFEs (%)	%PCIe (%)	tCPU (s)	tDFE (s)	FPGA slika
2	Odd-even merge network sort	50	100	1000	10	1
9	Spherical code design	40	0	180	10	1
3	Lattice-Boltzmann	20	0	170	10	2
7	Odd-even merge network sort	50	100	1000	10	2
4	Spherical code design	40	0	1800	100	3
8	Lattice-Boltzmann	20	0	1700	100	3

Таβела 10 представља параметре који се могу користити за поређење алгоритама уа прављење распореда: укупно време извршења у случају да се сви послови извршавају на процесору (*tCPUtotal*), укупно време извршења послова на хардверу заснованом на протоку података (*tDFEtotal*), разлика између ове две вредности, као и уβрзање мерено као време *tCPUtotal* подељено са временом *tDFEtotal*.

Таβела 10: Поређење алгоритама за прављење распореда на датом примеру послова.

Алгоритам за прављење распореда	tCPUtotal (s)	tDFEtotal (s)	Razlika (s)	Ubrzanje
FCFS	4860	520	4340	9.3461538462
Најβољи алгоритам за прављење распореда за уштеду времена	7140	420	6720	17
Предложени алгоритам за прављење распореда за уштеду времена	6500	330	6170	19.696969697
Предложени алгоритам за прављење распореда за максимизовање протока	5850	240	5610	24.375

Да βи алгоритам са слике 33, заснован на хеуристици, βио разумно мали, изоставили смо ограничење βроја послова који се узимају у οβзир за заказивање, узрокујући да, ако постоји више од 20 послова доступних за заказивање, само првих 20 сортираних по почетном βроју поена βуду узети у οβзир. Трајање прављења распореда у случају узимања у οβзир само пет послова у нашем случају

је око 0.06 секунди извршењем на Intel i5 650 процесору. У случају заказивања шест послова, време прављења распореда је око 0.45 секунди.

5.4. Прављење распореда извршавања послова из синтетичког пакета послова

Можемо да дефинишемо укупан фактор убрзања као однос између времена потребног само за процесор да изврши све послове и времена потребног процесору и хардверу заснованом на протоку података за извршење истог скупа послова. Ако се измени однос броја послова за процесор и послова за хардвер заснован на протоку података, укупан фактор убрзања ће бити различит. Што је већи број послова за процесор, односно послова који нису погодни за извршавање на хардверу заснованом на протоку података, фактор убрзања ће бити мањи. Како није могуће унапред знати какве послове ће будући рачунари извршавати, биће најпре приказано убрзање коришћењем одређених врста послова, а затим и у случају комбиновања тих врста послова. Вредности на хоризонталној оси ће бити увек линеарне и представљаће однос количине послова за процесор и хардвер заснован на протоку података. Вредност један представља однос између количине послова процесора и послова за хардвер заснован на протоку података за које су хардвер заснован на протоку података и процесор у просеку једнако окупирани.

На сваком од графика, биће приказана три мода прављења распореда. Сваки од њих користи исти скуп послова, како би поређење било фер. Први планер оптимизује уштеду времена (*Time gain*). Други оптимизује проток (*Throughput*). Трећи прави распоред послова тако што хардверу заснованом на протоку података додељује послове у истом редоследу у коме су дошли, пакујући на сваку FPGA слику онолико узастопних послова колико је могуће (*FCFS*).

Поред ова три графика, биће приказани и графици смањења потрошње електричне енергије када се користи хардвер заснован на протоку података у односу на потрошњу када се користи једино процесор заснован на контроли тока. Прорачуната потрошња се заснива на претпоставци да је потрошња рачунара само са процесором заснованом на контроли тока 500W, док је додатна потрошња хардвера заснованог на протоку података 550W. Треба напоменути да различите

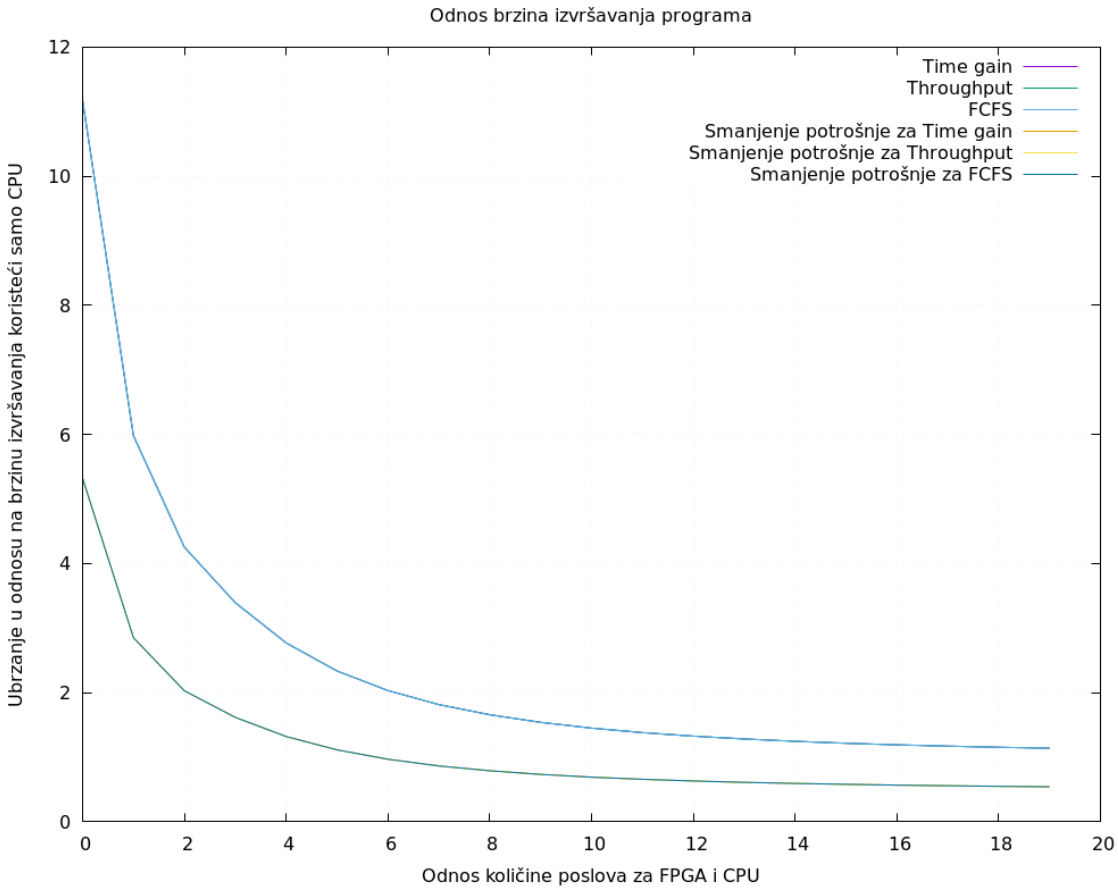
верзије хардвера заснованог на протоку података имају различите карактеристике, али и потрошњу енергије, па је тачан износ потрошње у зависности од типа хардвера 417W, 550W, или 680W.

За вредности фактора смањења потрошње електричне енергије мање од један, укупна потрошња ће бити већа уколико се користи хардвер заснован на протоку података. На тај начин се може проценити до ког односа послова, односно програма, за хардвер заснован на протоку података у односу на оне за процесор заснован на контроли тока је оправдано користити хардвер заснован на протоку података.

У случају да се времена извршавања скупа програма за одређене типове распореда незнатно разликују, неки од графика ће се делом или у потпуности преклапати.

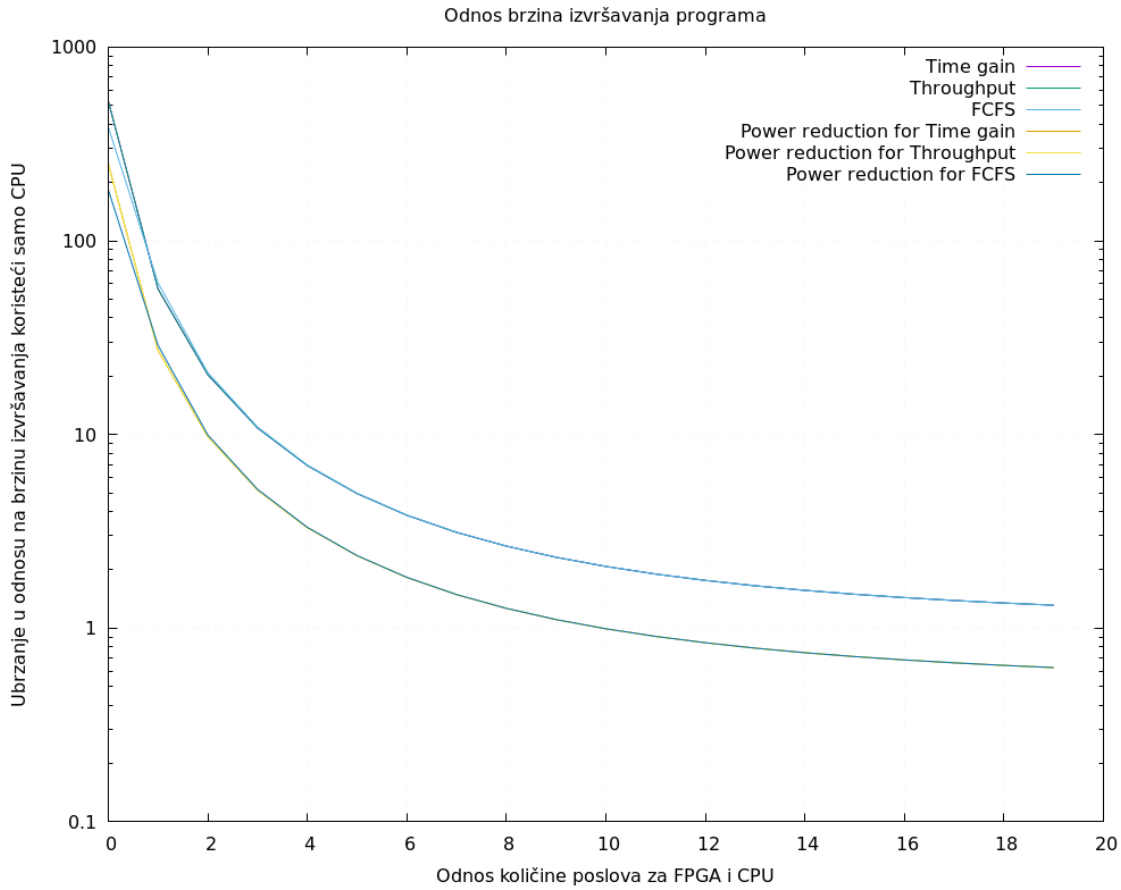
Сваки скуп програма (енг. *benchmark*) се састоји од алгоритама из табеле 6. Трајања послова су постављана у складу са величинама скупова података датих у апликацијама. За сваку симулацију сценарија, извршено је 100 итерација. Тестиран је сваки од алгоритама за прављење распореда користећи 20 односа послова за процесор и послова за хардвер заснован на протоку података, и то са вредностима од 0 до 9,5 са кораком од 0,5.

График на слици 35 приказује укупан фактор убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су из скупа могућих послова за хардвер заснован на протоку података коришћени само послови који решавају проблем *Gross-Pitaevskii*.



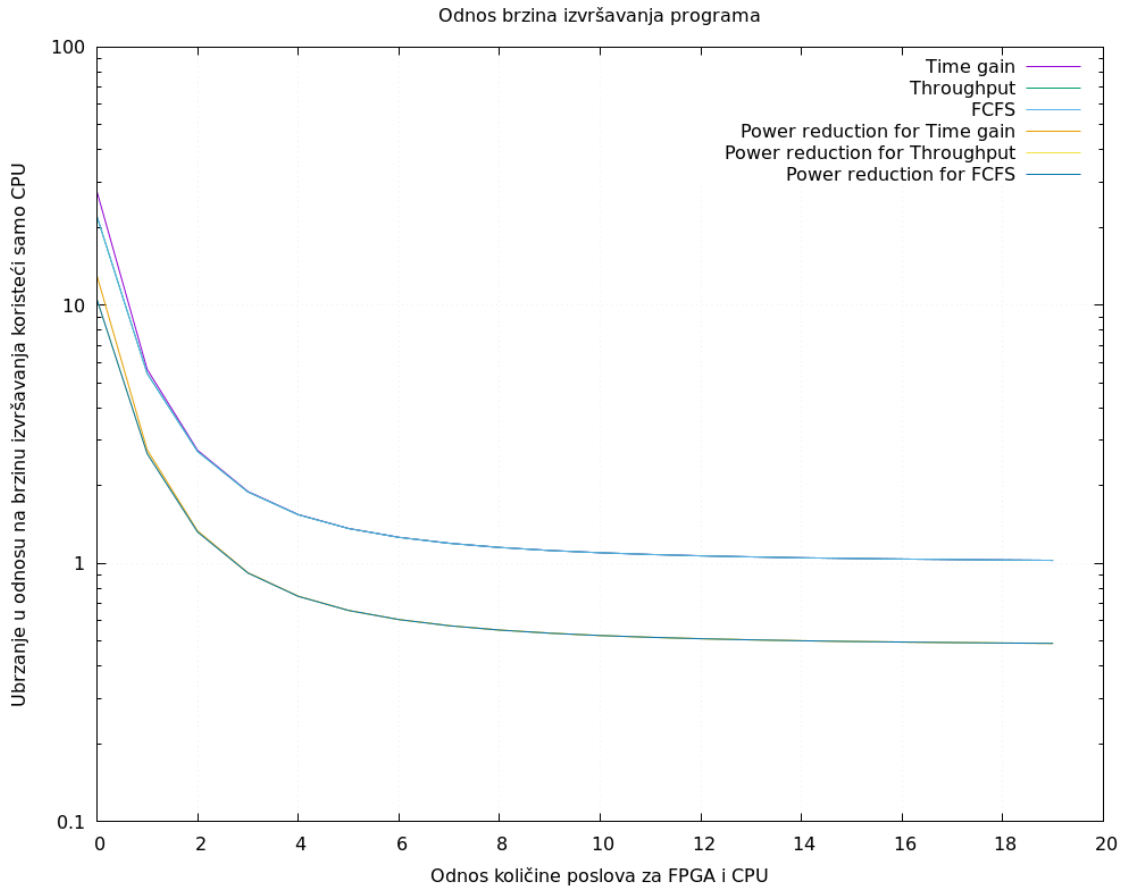
Слика 35: Поређење извршавања *Gross-Pitaevskii* алгоритама на FPGA и алгоритама за процесор заснован на контроли тока.

График на слици 36 приказује укупан фактор убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су из скупа могућих послова за хардвер заснован на протоку података коришћени само послови који решавају проблем *Odd-even merge network sort*. Доњи график представља смањење потрошње електричне енергије.



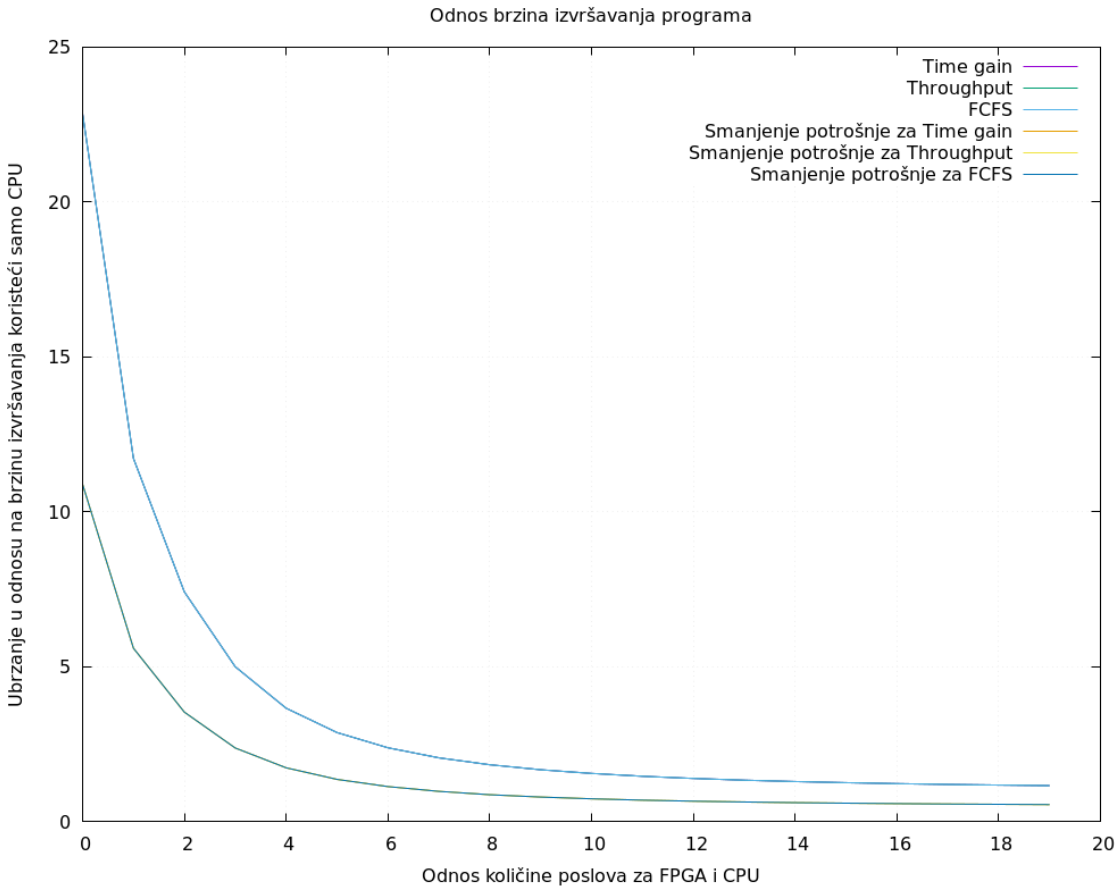
Слика 36: Поређење извршавања *Odd-even merge network sort* алгоритама на FPGA и алгоритама за процесор заснован на контроли тока.

График на слици 37 приказује укупан фактор убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су из скупа могућих послова за хардвер заснован на протоку података коришћени само послови који решавају проблем Lattice-Boltzmann. На овом графику је вертикална оса логаритамска, јер је максимално убрзање у случају извршавања оваквих послова веће између 20 и 30 пута.



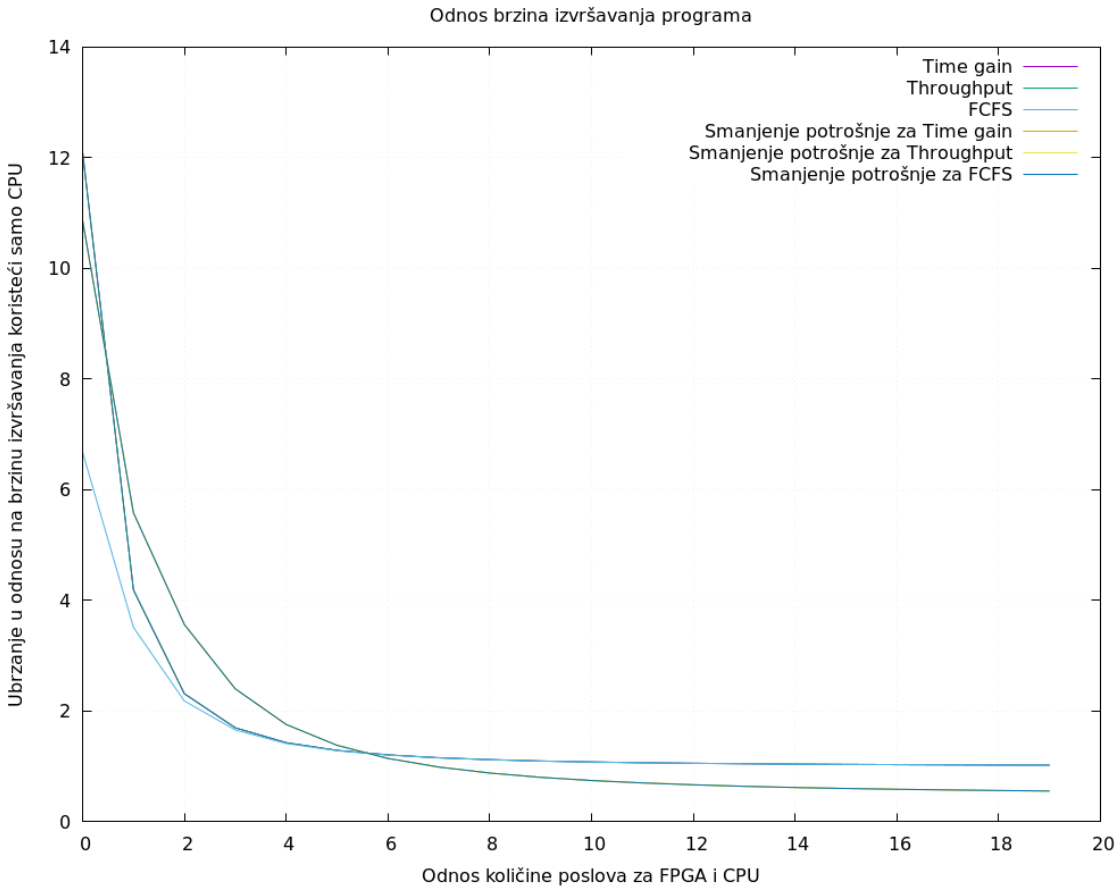
Слика 37: Поређење извршавања *Lattice-Boltzmann* алгоритама на FPGA и алгоритама за процесор заснован на контроли тока.

График на слици 38 приказује укупан фактор убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су из скупа могућих послова за хардвер заснован на протоку података коришћени само послови који решавају проблем *Spherical code design*.



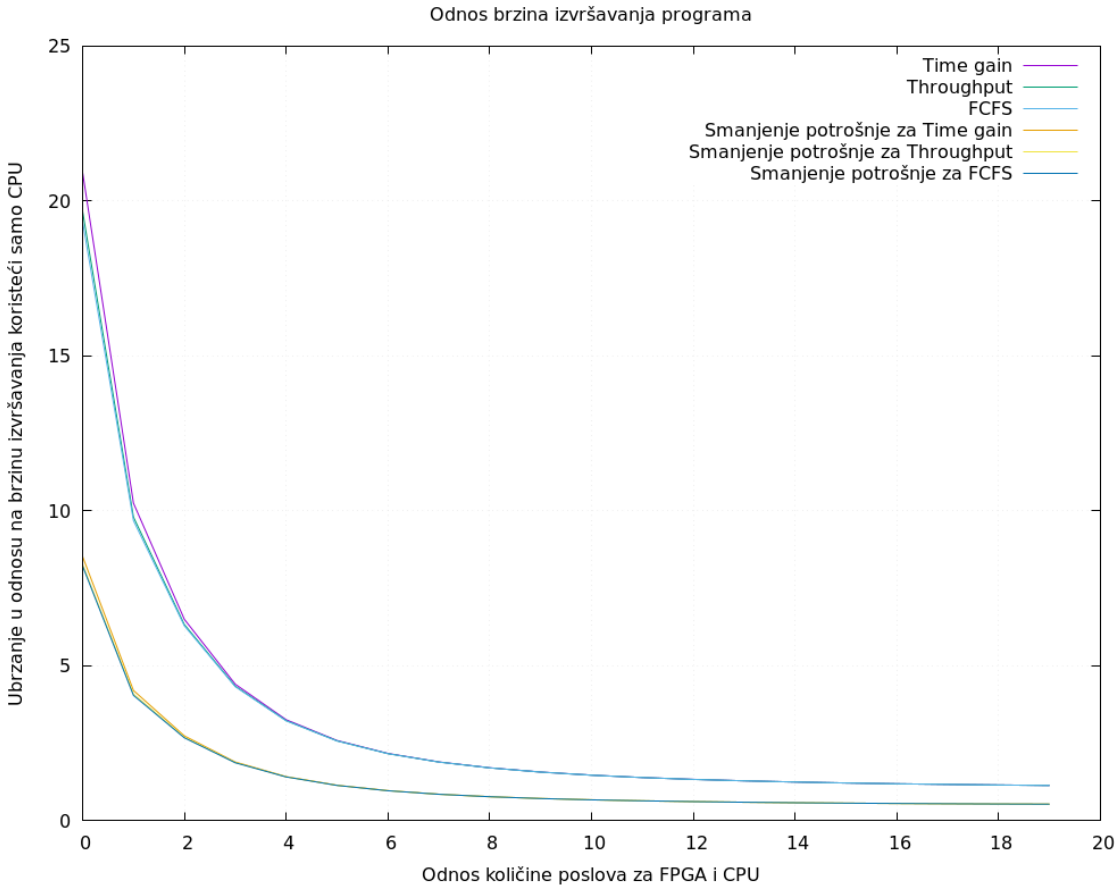
Слика 38: Поређење извршавања *Spherical code design* алгоритама на FPGA и алгоритама за процесор заснован на контроли тока.

График на слици 39 приказује укупан фактор убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су из скупа могућих послова за хардвер заснован на протоку података коришћени само послови који решавају проблем *RSA*.



Слика 39: Поређење извршавања *RSA* алгоритама на FPGA и алгоритама за процесор заснован на контроли тока.

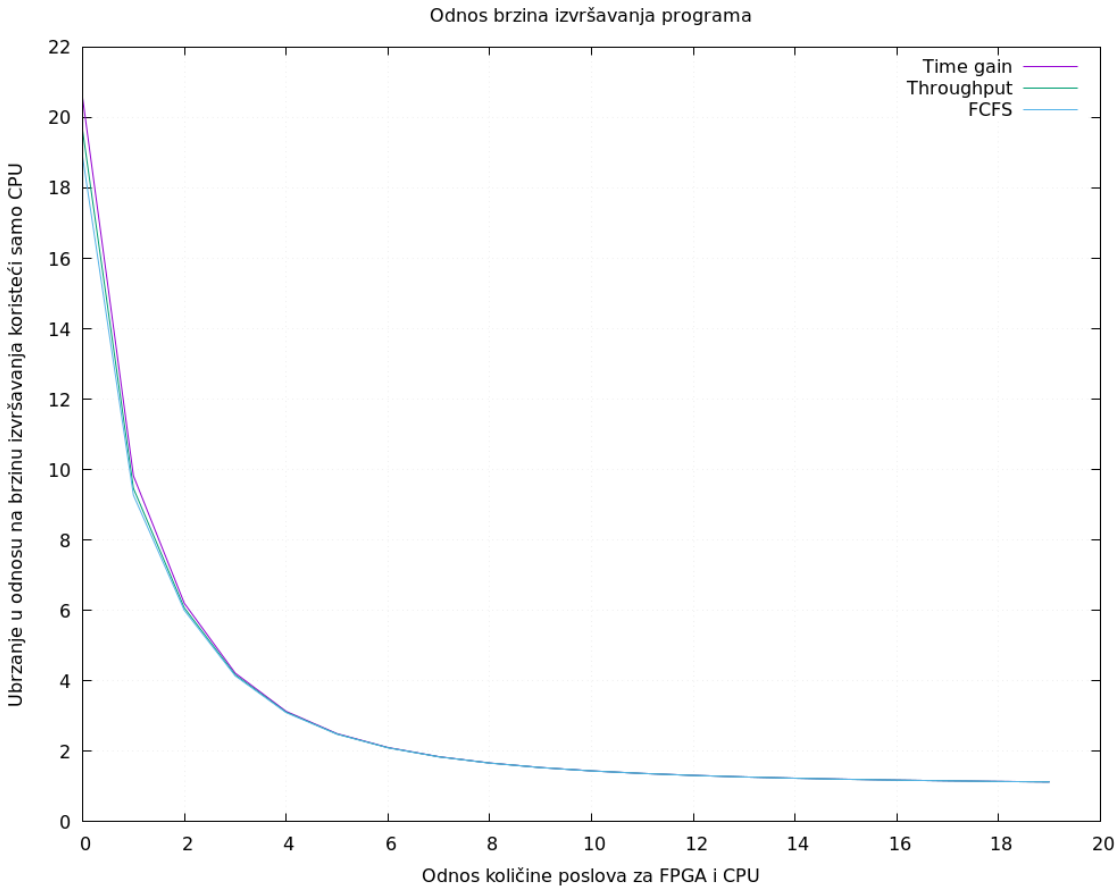
График на слици 40 приказује укупан фактор убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су као послови бирани случајно изабрани послови из скупа могућих послова за хардвер заснован на протоку података. Како је код *RSA* алгоритма убрзање мање него код осталих поменутих алгоритама за хардвер заснован на протоку података, дешава се да је у неком тренутку мање убрзање него што је смањење потрошње електричне енергије. У пракси, ова тачка не представља важну тачку, јер је у зависности од важности убрзања или потрошње електричне енергије исплативо користити хардвер заснован на протоку података до односа количине послова који је било већи или мањи од вредности у тој тачки.



Слика 40: Поређење извршавања алгоритама за процесор заснован на протоку података и алгоритама за процесор заснован на контроли тока.

Прецизне вредности фактора убрзања који су коришћени при прављењу графика са слике 40 се могу наћи у табели 1 у прилогу.

Поред овога, тестиран је и утицај доступности оперативне меморије за извршавање послова. На графику са слике 41 приказани су фактори убрзања у зависности од односа послова за процесор и послова за хардвер заснован на протоку података, при чему су као послови бирани случајно изабрани послови из скупа могућих послова за хардвер заснован на протоку података, а меморија представља још један лимитирајући фактор.



Слика 41: Убрзање извршавања програма у зависности од односа укупног трајања програма за процесор заснован на контроли тока и хардвер заснован на протоку података, када је оперативна меморија лимитирајући ресурс.

6. Анализа резултата

Поређењем резултата прављења распореда датих у табели 10, можемо видети да распоред за уштеду времена производи највећи успех у смањењу укупног времена извршења. Иако овај алгоритам остварује највећи пад у укупном времену извршења, овај распоред оставља довољно ресурса на трећој FPGA слици, тако да послови који пристижу за време извршења прве две FPGA слике се могу упаковати поред посла број 7 из табеле 8. Треба имати на уму да то не значи да је предложени распоред бољи од најбољег распореда, већ да најбољи распоред укључује само послове који су доступни у тренутку прављења распореда. Ограничење најбољег распореда је да је NP-тешко (енг. *NP-hard*). С друге стране, FCFS планер је више налик на обичан ред послова него на прави алгоритам за распоређивање послова.

Пример из табеле 8 јасно показује потенцијалну грешку предложеног планера за максималну уштеду времена. Последња два посла у распореду имају факторе убрзања 100. Ипак, они нису заказани за прву FPGA слику. Постоје две тачке гледишта у овом случају. Прва је да треба задња два посла заказати на самом почетку, чак и сваки засебно на прве две FPGA слике. На тај начин би се очигледно завршили послови који највише одговарају хардверу заснованом на протоку података у најкраћем могућем року, постизањем релативно велике разлике у времену извршења ових послова на процесору и на хардверу заснованом на протоку података. Друга тачка гледишта је да би те послове могли оставити за касније, тако да бар неки од њих може да се уклопи са другим пословима који не захтевају PCIe магистралу током извршења. Предложени алгоритам прави распоред за максимализовање протока дат у табели 6.

Фактори убрзања из табеле 7 показују да је добитак перформанси, у случају да хардвер заснован на протоку података није више од 50% времена у стању мировања, послова за хардвер заснован на протоку података је довољан да се плати додатна величина чипа за хардвер заснован на протоку података, док се укупна потрошња електричне енергије смањује.

Слика 2 приказује укупан фактор убрзања система у зависности од алгоритма за прављење распореда послова. Уколико нема послова за процесор заснован на контроли тока, фактор убрзања је приближно исти као однос између времена које је потребно за процесор да изврши све послове и времена потребног за хардвер заснован на протоку података да их изврши. Како се количина послова за процесор повећава, укупан фактор убрзања се смањује. Када се количина послова за процесор повећа толико да сви послови за хардвер заснован на протоку података могу да се изврше за време док процесор обавља остале послове, нема више разлике између два алгоритма за прављење распореда извршавања послова. Даљим повећањем количине послова за процесор, фактор убрзања се смањује и приближава вредности један, која указује да не постоји убрзање.

Планер који оптимизује уштеду времена се показао као ефикаснији од оног који оптимизује проток и то за све односе количине послова за процесор и послова који се могу убрзати употребом хардвера заснованог на протоку података. Позивајући оба алгоритма за прављење распореда за односе послова за процесор и послова који се могу убрзати коришћењем хардвера заснованог на протоку података од 0 до 40 пута, израчунате су корелације 0.952 и коваријансе од 12.4. Ова корелација се могла очекивати, јер су оба планера убрзала извршавање истог скупа послова.

Важан налаз је да након што однос постане једнак или већи од један, разлика између укупног времена извршења користећи два алгоритма за прављење распореда постаје мања од 1% укупног времена извршења. Ово наглашава значај односа између послова. У случају FCFS распореда, разлика у укупном времену извршења постаје мања од 1% само када однос достигне три или више.

Мењањем процената главне меморије која је потребна за послове који се могу убрзати коришћењем хардвера заснованог на протоку података, можемо симулирати услове у којима главна меморија постаје уско грло система. Као резултат тога, додатна ограничења ће се појавити приликом прављења распореда, што ће у потрази за наредним послом који се може ставити на хардвер заснован на протоку података, тј. на FPGA слику, сузити избор послова. Као што се очекивало, резултати показују ниже факторе убрзања, као и мању разлику између укупног времена извршења оба алгоритма за прављење распореда.

У односу на алгоритам за прављење распореда застнованог на мешовитом целобројном линеарном програмирању (енг. *mixed integer linear programming*) које је предложио Abdessamad [81], алгоритми за прављење распореда предложени у овој тези се заснивају на *branch-and-bound* принципу, при чему су узети у обзир не само трошкови заказивања послова на FPGA, већ и ограничења, што је омогућило послу за који није вероватно да произведе најбољи резултат да ипак буде постављен на исту FPGA слику поред других послова, уколико не постоји бољи посао који може да се смести уместо њега. Ово чини извршавање алгоритма за прављење распореда споријим, али дозвољава смањење укупног времена извршења послова, као што можемо видети из табеле 5, где се посао број 9 пакује на прву FPGA слику, иако извршавање овог посла на хардверу заснованом на протоку података најмање смањује укупно време извршења свих послова. На овај начин, смањује се укупно време израчунавања далеко више него што је време потребно за прављење распореда извршавања послова. Међутим, овакав планер не заказује чак ни близу 50 послова, већ само послове који могу да стану на неколико FPGA слика, остављајући заказивање других расположивих послова у том тренутку ближе њиховом извршењу, јер нови послови могу доћи у међувремену. Треба имати у виду да Abdessamad [81] прави распоред извршавања послова према задатом графику зависности, остављајући само део послова на располагању за планирање у сваком тренутку. Главни разлог из ког можемо направити план извршавања мање послова је то што је хардвер заснован на протоку података обично у стању да уради само до неколико алгоритама симултано, без утицаја на ефикасност извршавања алгоритама. У случају истраживања које су обављали истраживачи који су радили на Maxeler хардверу заснованом на протоку података, није познато да су више независних алгоритама извршавани на хардверу заснованом на протоку података у исто време користећи једну картицу засновану на протоку података. Смањење ефикасности у циљу повећања броја послова који се могу извршавати истовремено би донело више ограничења за алгоритам за прављење распореда, што би опет довело до споријег прављења распореда.

Алгоритам за прављење распореда извршавања послова који је предложио Fej [82] је у неким својим деловима сличнан предложеном решењу. Међутим, предложено решење је дизајнирно да буде брже и адаптабилно у време извршавања послова,

што га чини једним од бржих алгоритама за прављење распореда извршавања послова. Као и у случају решења које предлаже Fej, и други аутори [96-105] су предлагали решења са циљем да побољшају прављење распореда под одговарајућим претпоставкама, али се таква решења не могу ефикасно применити на проблем који се овде разматра. Стога би или њихово време извршавања било веће од времена извршавања предложених алгоритама, или би добијени распореди били субоптимални.

Оптимизациони алгоритам заснован на више агената који су предложили Zheng и Wang [86] води ка оптималном решењу тако што итерира према најбољем решењу избором најбољег кандидата у сваком кораку. Овај алгоритам је захтеван у погледу рачунања у случају да постоји довољно агената неопходних да се покрију све могућности у случају постојања различитих ограничења (RAM меморије, хардвер заснован на протоку података меморије, процесор време прорачун за сваки посао, времена извршавања послова употребом хардвера заснованог на протоку података, ограничења саме FPGA технологије, ограничења PCIe магистрале, итд). Уместо тога, предложени приступ употребом хеуристике уклапа посао на прву FPGA слику релативно брзо, а у исто време и релативно ефикасно у односу на најбољи могући распоред и сценарио распоређивања користећи FCFS технику.

Hamilton [84] је развио систем за процесе мешовитих архитектура који подржавају промену контекста и блокирање. Међутим, овакав модел извршавања не разматра ограничења хардвера заснованог на протоку података.

Предложени алгоритми за прављење распореда извршавања послова се могу подесити да подржавају постојање више од једног хардвера заснованог на протоку података и више од једног процесора. На овај начин, дохватање конфигурације унапред (енг. *prefetching*) може да се користи као ефикасан механизам за скривање кашњења припреме FPGA слике. У случају дохватања унапред, посао за хардвер заснован на протоку података може да се учита чим сам хардвер заснован на протоку података постане слободан (под условом да је познато да ће послови бити обрађени), чиме се сакрива одлагање извршења ради конфигурације. Khuat [85] је објавио свеукупно смањење у времену извршења од 22% користећи свој механизам за дохватање унапред.

7. Претње валидности резултата

Најважнија претња валидности презентованих резултата је чињеница да није могуће знати које врсте апликација ће будући рачунари претежно извршавати. Ова теза се заснива на претпоставци да ће алгоритми који су познати као алгоритми који захтевају много процесорског времена за своје извршавање све чешће бити извршавани, јер ће хардвер омогућавати извршавање више инструкција у секунди.

Време извршења захтева је тешко знати пре него што се крене са извршавањем програма. На срећу, хардвер заснован на протоку података је често програмиран за посао прераде одређене количине података која је позната у време покретања посла, а самим тим, трајање посла је познато планеру. Међутим, овде приказани резултати су засновани на претпостављеним познатим трајањима извршења алгоритама, иако није могуће знати трајања извршења алгоритама за величине улазних података који ће се користити у будућности.

Иако хардвер заснован на протоку података може да има више меморије него процесор заснован на контроли тока, скуп послова који је коришћен за тестирање преложених алгоритама за прављење распореда извршавања послова се састојао од апликација које не захтевају много меморије. Могућа претња валидности је чињеница да би послови који се извршавају на хардверу заснованом на протоку података могли да захтевају релативно велике количине главне меморије приликом извршавања на хардверу заснованом на протоку података, спречавајући њихово заказивање у сваком тренутку, што би утицало на укупан фактор убрзања.

8. Закључак

Парадигма заснована на протоку података постоји већ релативно дуго, али изгледа да јој се популарност повећава од како је технологија израде FPGA кола постала довољно ефикасна. Релативно велики број алгоритама за хардверске архитектуре засноване на протоку података се појавио у последњих неколико година. У овој тези су представљене неки од њих и предложен нови за прављење распореда извршавања више послова на дељеном FPGA у време самог извршавања заснован на тзв. *branch-and-bound* принципу. За разлику од већине алгоритама за прављење распореда доступних у отвореној литератури, овај алгоритам примењује хеуристику такву да се у разматрање за заказивање узима само релативно мали број послова, при чему се шанса даје прво пословима за које је за очекивати да ће највише допринети смањењу укупног времена извршавања послова у зависности од тога да ли се извршавање врши помоћу процесора или хардвера заснованог на протоку података. Из тог разлога, предложени алгоритам је више од реда величине бржи од генетских алгоритама и алгориитама заснованих на техникама линеарног програмирања, док је способан за прављење распореда који су релативно близу најбољим могућим распоредима.

Главно ограничење коришћења предложеног алгоритма за прављење распореда је доступност хардвера заснованог на протоку података у данашњим рачунарима. Није могуће знати да ли ће се тренд повећавања коришћења хардвера заснованог на протоку података за решавање проблема извршавања захтевних алгоритама наставити, а па самим тим није познато ни још колико дуго. Стога, овај рад може да послужи као смерница за креирање будућих архитектура и организација рачунара, али и за повећање продуктивности расположивих суперкомпјутера употребом хардвера заснованог на протоку података на тај начин да алгоритми који захтевају у изразитој мери различите ресурсе хардвера заснованог на протоку података могу да се аутоматски комбинују и тако извршавају на хардверу заснованом на протоку података истовремено.

Будући рад ће укључивати прављење распореда поштујући приоритет послова. За важне послове, могуће је обезбедити ресурсе хардвера заснованог на протоку података пре прављења распореда извршавања преосталих алгоритама. Комбиновањем различитих алгоритама за прављење распореда доступних у отвореној литератури, укупно време извршења у случају прављења распореда већег броја послова могло би се додатно смањити.

9. Литература

[1] A. Hurson, V. Milutinovic, Special Issue on DataFlow SuperComputing, Advances in Computers, Vol. 96, 2015.

[2] V. Milutinovic, J. Salom, N. Trifunovic, R. Giorgi, Guide to DataFlow Supercomputing, Springer International Publishing, 2015, pp. 1-129.

[3] V. Milutinovic, A. Hurson, Dataflow Processing, Academic Press, 1st edition, 2015, pp. 1-266.

[4] D. Kirk, NVIDIA CUDA software and GPU parallel computing architecture, ISMM, Vol. 7, 2007, pp. 103-104.

[5] R. P. Feynman, Lectures on Computation, The ACM Digital Library, 1998.

[6] V. Milutinovic, J. Salom, N. Trifunovic, R. Giorgi, “Guide to DataFlow Supercomputing,” Springer, 2015, ISBN 978-3-319-16229-4.

[7] S. Stojanovic, D. Bojić, M. Bojović, M. Valero, V. Milutinović, An overview of selected hybrid and reconfigurable architectures, 2012 IEEE International Conference on Industrial Technology (ICIT), IEEE, 2012, DOI: 10.1109/ICIT.2012.6209978.

[8] Z. Jovanovic, V. Milutinovic, V., FPGA Accelerator for Floating-Point Matrix Multiplication, The IET Computers and Digital Techniques Premium Award for 2014, Volume 6, Issue 4, 2012, pp. 249-256.

[9] R. Trobec, R. Vasiljevic, M. Tomasevic, V. Milutinovic, et al, Interconnection Networks for PetaComputing, ACM Computing Surveys, September 2016.

[10] N. Trifunovic, V. Milutinovic, et al, The Appgallery.Maxeler.com for BigData SuperComputing, Journal of Big Data, Springer, 2016.

[11] N. Korolija, J. Popović, M. Cvetanović, M. Bojović, Dataflow Based Parallelization of Control-Flow Algorithms, Creativity in Computing and Dataflow

Supercomputing, Advances in Computers, Vol. 104, DOI: 10.1016/bs.adcom.2016.09.003, 2017, Print ISBN 9780128119556.

[12] J. G. Koomey, C. Belady, M. Patterson, A. Santos, K.-D. Lange, Assessing Trends Over Time in Performance, Costs, and Energy Use for Servers, 2009, Analytics Press.

[13] M. Flynn, O. Mencer, V., Milutinovic, G., Rakocevic, P., Stenstrom, M., Valero, R., Trobec, Moving from PetaFlops to PetaData, Communications of the ACM, May 2013, pp. 39-43.

[14] A. Kos, S. Tomazic, J. Salom, N. Trifunovic, M. Valero, V. Milutinovic, New Benchmarking Methodology and Programming Model for Big Data Processing, International Journal of Distributed Sensor Networks, ISSN 1550-1477, 2015, vol. 2015, pp. 1-7.

[15] T. Nowatzki, V. Gangadhar, K. Sankaralingam: Exploring the potential of heterogeneous von neumann/dataflow execution models, Proceedings of the 42nd Annual International Symposium on Computer Architecture, June 13, 2015, ACM, pp. 298-310.

[16] A. DeHon, J. Wawrzynek, Reconfigurable Computing: What, Why, and Implications for Design Automation, Proceedings of the 36th annual ACM/IEEE Design Automation Conference, ACM, 1999, pp. 610-615.

[17] A. H. Veen, Dataflow Machine Architecture, ACM Computing Surveys (CSUR). ACM. Vol 18 Num 4, 1986, pp. 365–396.

[18] B. K. Essink, Using FPGAs as Fine-Grained Static Dataflow Machines, 21st Twente Student Conference on IT, June 23rd, 2014, Enschede, The Netherlands, 2014.

[19] C. Schryver, H. Marxen, S. Weithoffer, N. Wehn, High-Performance Hardware Acceleration of Asset Simulations, High-Performance Computing Using FPGAs, Springer, New York, 2013, pp. 3–32.

[20] S. Stojanovic, D. Bojic, M. Bojovic, An Overview of Selected Heterogeneous and Reconfigurable Architectures, Advances in Computers, Vol. 96, Burlington: Academic Press, 2015, pp. 1-45.

[21] K. Olukotun, L. Hammond, The Future of Microprocessors, ACM Queue 3 (7) (2005) , 26–29.

[22] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, D. Burger, Power Challenges May End the Multicore Era, Communications of the ACM, Vol. 56, No. 2, February 2013, pp. 93-102.

[23] C. Yang, H. Wu, Q. Huang, Z. Li, J. Li, Using Spatial Principles to Optimize Distributed Computing for Enabling the Physical Science Discoveries, PNAS, vol. 108, no. 1., April 5, 2011.

[24] L. Heendaliya, M. Wisely, D. Lin, S. S. Sarvestani, A. Hurson, Influence-Aware Predictive Density Queries Under Road-Network Constraints, Advances in Spatial and Temporal Databases, Springer International Publishing, 2015, pp. 80-97.

[25] D. Pellerin, S. Tibault, Practical FPGA Programming in C, Prentice Hall Press, Upper Saddle River, NJ, USA, 2005.

[26] D. Cohen, Mathematical Approach to Iterative Computation Networks, Proceedings of Fourth Symposium on Computer Arithmetic, IEEE, 1978, pp. 226-238.

[27] L. Johnsson, D. Cohen, Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks, VLSI Systems and Computations, Computer Science Press, Rockville, USA, 1981, pp. 213-225.

[28] U. Weiser, A. Davis, A Wavefront Notion Tool for VLSI Array Design, VLSI Systems and Computations, Computer Science Press, Rockville, USA, 1981.

[29] M. Lam, J. Mostow, A Transformational Model of VLSI Systolic Design, IFIP Sixth International Symposium on Computer Hardware Descriptive Languages and their Applications, Carnegie-Mellon University, Pittsburg, USA, May 1983.

[30] D. Gannon, Pipelining Array Computations for MIMD Parallelism: A Functional Specification, Procedings of International Conference on Parallel Processing, 1982, pp. 284-286.

[31] H. T. Kung, W. T. Lin, An Algebra for VLSI Algorithm Design, Proceedings of Conference on Elliptic Problem Solvers, Monterey, CA, 1983.

[32] R. H. Kuhn, Transforming Algorithms for Single-stage and VLSI Architectures, Proceedings of Workshop on Interconnection Networks for Parallel and Distributed Processing, April 1980, pp. 11-19.

[33] R. H. Kuhn, Optimization and Interconnection Complexity for Parallel Processors, Single Stage Networks and Decision Trees, Ph.D. thesis, Technical Report 80-1009, University of Illinois, Urbana-Champaign, USA, 1980.

[34] W. L. Miranker, A. Winkler, Space-time Representations of Computational Structures, Computing, Vol. 32, 1984, pp. 93-114.

[35] J. A. B. Fortes, F. Parisi-Presicce, Optimal Linear Schedules for the Parallel Execution of Algorithms, Proceedings of International Conference on Parallel Processing, IEEE, August 1984.

[36] J. A. B. Fortes, C. S. Raghavendra, Dynamically Reconfigurable Fault-tolerant Array Processors, Proceedings of 14th International Conference on Fault-tolerant Computing, IEEE, 1984.

[37] J. A. B. Fortes, D. I. Moldovan, Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms, Journal of Parallel and Distributed Computing, Vol. 2, Academic Press, 1985.

[38] J. A. B. Fortes, D. I. Moldovan, Data Broadcasting in Linearly Scheduled Array Processors, Proceedings of 11th Annual Symposium on Computer Architecture, ACM/IEEE, 1984, pp. 224-231.

[39] D. I. Moldovan, On the Design of Algorithms for VLSI Systolic Arrays, Proceedings of IEEE, Vol. 71, No. 1, January 1983, pp. 113-120.

[40] D. I. Moldovan, A. Varma, Design of Algorithmically Specialized VLSI Devices, Proceedings of International Conference on Computer Design: VLSI in Computers, 1983, pp. 88-91.

[41] S. Lerner, D. Grove, C. Chambers, Composing Dataflow Analyses and Transformations, The 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2002, January 16-18, 2002 Portland Oregon, USA, ACM, ISBN 1-58113-450-9/02/01.

[42] J. Villarreal, A. Park, W. Najjar, R. Halstead, Designing Modular Hardware Accelerators in C with ROCCC 2.0, 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2010.

[43] M.J. Flynn, O. Pell, O. Mencer, Dataflow Supercomputing, 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 1-3.

[44] R. Dimond, S. Racanière, O. Pell, Accelerating Large-Scale HPC Applications Using FPGAs, 20th IEEE Symposium on Computer Arithmetic (ARITH), 2011, pp. 191-192.

[45] A. Dellson, G. Sandberg, S. Mohl, Turning FPGAs into Supercomputers - Debunking the Myths about FPGA-based Software Acceleration, In Proceedings of the 48th Cray User Group meeting, 2006.

[46] D. J. William, The End of Denial Architecture and The Rise of Throughput Computing, Keynote speech at Desgin Automation Conference, 2010.

[47] K. Georgia, N. S. Voros, K. Masselos, System Level Design of Complex Hardware Applications using ImpulseC, 2010 IEEE Annual Symposium on VLSI. IEEE, 2010.

[48] T. Grtker, S. Liao, G. Martin, S. Swan, System Design with SystemC (1st ed.), Springer Publishing Company, Incorporated, 2010.

[49] S. Gupta, R. K. Gupta, N. Dutt, A. Nicolau, Coordinated Parallelizing Compiler Optimizations and High-level Synthesis, ACM Transactions on Design Automation of Electronic Systems, 9(4):441-470, October 2004.

[50] Y. Ben-Asher, N. Rotem, Using Memory Profile Analysis for Automatic Synthesis of Pointers Code, ACM Transactions on Embedded Computing Systems (TECS), Volume 12 Issue 3, March 2013.

[51] S. J. Kim, L. De Carli, K. Sankaralingam, C. Estan, SWSL: Software Synthesis for Network Lookup, ANCS '13 Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 191-202.

[52] Blagojevic, et al, A Systematic Approach to Generation of New Ideas for PhD Research in Computing, The Advances in Computers, Vol. 102, 2015.

[53] H. T. Kung, C. E. Leiserson, Systolic Arrays (for VLSI), Proceedings of Sparse Matrix, 1978, pp. 256-282.

[54] H. T. Kung, P. L. Lehman, Systolic (VLSI) Arrays for Relational Database Pperations, Proceedings of International Conference Management of Data, ACM SIGMOD, May 1980, pp. 105-116.

[55] K. T. Johnson, A. R. Hurson, B. Shirazi, General-Purpose Systolic Arrays, IEEE Computer, November 1993, pp. 20-31.

[56] W. A. Najjar, B. A. Buyukkurt, Z. Guo, J. Villareal, J. Cortes, A. Mitra, Compiled Code Acceleration on FPGAs, Department of Computer Science and Engineering University of California-Riverside Riverside CA 92507, USA. Available at: http://samos-conference.com/Resources_
Samos_Websites/Symposium/09_Najjar_Buyukkurt_Guo_Villareal_Cortes_Mitra.pdf

[57] T. J. Callahan, J. R. Hauser, J. Wawrzynek, The Garp Architecture and C Compiler, IEEE Computer, 2000.

[58] M. B. Gokhale, J. M. Stone, J. Arnold, M. Lalinowski, Stream-oriented FPGA Computing in the Streams-c high Level Language, In IEEE Symp. on FPGAs for Custom Computing Machines (FCCM 2000), 2000.

[59] M. Hall, P. Diniz, K. Bondalapati, H. Zeigler, P. Duncan, R. Jain, J. Granacki, DEFACTO: A Design Environment for Adaptive Computing Technology, In Proc. 6th Reconfigurable Architectures Workshop (RAW’99), 1999.

[60] W. A. Najjar, A. Bohm, B. Draper, J. Hammes, R. Rinker, R. Beveridge, M. Chawathe, C. Ross, From Algorithms to Hardware - A High-Level Language Abstraction for Reconfigurable Computing, IEEE Computer, 36(8):63–69, August 2003.

[61] R. A. Arvind, R. A. Iannucci, Two Fundamental Issues in Multiprocessing: the Dataflow Solutions, MIT Laboratory for Computerr Science, MIT/LCS/MT-241, Seprember 1983.

[62] D. A. Adams, A Computation Model with Data Flow Sequencing, Tehnical Report CS117, Computer Science Department, Stanford University, Stanford, California, December 1968.

[63] D. D. Chamberlin, Parallel Implementation of a Single-Assignment Language, Stanford University, Computeer Science, Ph.D. thesis, 1971.

[64] J. E. Rodriguez, A Graph Model for Parallel Computation, Technical Report ESL1-R-398, MAC-TR-64, Laboratory for Computer Science, Massachusetts Institute of Technology, USA, 1969.

[65] K. Stavrou, D. Pavlou, M. Nikolaides, P. Petrides, Z. Popovic, R. Giorgi, Programming Abstractions and Toolchain for Dataflow Multithreading Architectures, Eighth International Symposium on Parallel and Distributed Computing, 2009. ISPDC '09. pp. 107-114.

[66] G. N. T. Huong, Y. Na, S. W. Kim, Applying Frame Layout to Hardware Design in FPGA for Seamless Support of Cross Calls in CPU-FPGA Coupling Architecture, Microprocessors & Microsystems, Volume 35 Issue 5, July, 2011, pp.462-472.

[67] C. Feichtinger, J. Habich, H. Köstler, U. Rüde, T. Aoki, Performance Modeling and Analysis of Heterogeneous Lattice Boltzmann Simulations on CPU-GPU Clusters, Parallel Computing, 2014.

[68] M. Krafczyk, J. Tölke, E. Rank, M. Schulz, Two-dimensional Simulation of Fluid–structure Interaction using Lattice-Boltzmann Methods, Computers & Structures, Volume 79, Issues 22–25, September 2001, pp. 2031–2037.

[69] G.R. McNamara, G. Zanetti, Use of the Boltzmann Equation to Simulate Lattice-gas Automata, Physical Review Letter, 61 (1988), pp. 2332–2335.

[70] H. Yua, S. S. Girimajia, L. Luob, DNS and LES of Decaying Isotropic Turbulence with and without Frame Rotation using Lattice Boltzmann Method, Journal of Computational Physics, Volume 209, Issue 2, 1 November 2005, pp. 599–616.

[71] C. Feichtinger, Design and Performance Evaluation of a Software Framework for Multi-Physics Simulations on Heterogeneous Supercomputers, Doctoral thesis, Erlangen, 2012.

[72] Lattice-Boltzmann Source Code, Web Page Visited in June, 2015. http://www.pbx-brasil.com/Pesquisa/Ferramentas/cuda/dia02/aulasCuda/exemplos/programas/latticeBoltzman/LB_Demo/2dLB_C.c

[73] G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), pp. 483-485, ACM, New York, NY, USA, 1967.

[74] N. Korolija, T. Djukic, V. Milutinovic, and N. Filipovic, Accelerating Lattice-Boltzman Method Using the Maxeler DataFlow Approach, Transactions on Internet Research, Vol. 9, No. 2, July 2013, pp. 5-10.

[75] S. Stojanovic, D. Bojic, and V. Milutinovic, Solving Gross Pitaevskii Equation Using Dataflow Paradigm, Transactions on Internet Research, Vol. 9, No. 2, July 2013.

[76] A. Kos, V. Rankovic, and S. Tomazic, Sorting networks on Maxeler dataflow supercomputing systems, Advances in computers, Vol. 96. Amsterdam, Elsevier: Academic Press, cop, 2015, pp. 139-186.

[77] V. Rankovic, A. Kos, V. Milutinovic, Bitonic Merge Sort Implementation on the Maxeler Dataflow Supercomputing System, Transactions on Internet Research, Vol. 9, No. 2, July 2013, pp. 34-42.

[78] I. Stanojevic, V. Senk, and V. Milutinovic, Application of Maxeler Dataflow Supercomputing to Spherical Code Design, Transactions on Internet Research, Vol. 9, No. 2, July 2013, pp. 1-4.

[79] N. Bezanic, J. Popovic-Bozovic, V. Milutinovic, and I. Popovic, Implementation of the RSA Algorithm on a DataFlow Architecture, Transactions on Internet Research, Vol. 9, No. 2, July 2013, pp. 11-16.

[80] N. Trifunovic, V. Milutinovic, J. Salom, A. Kos, Paradigm Shift in Big Data SuperComputing: DataFlow vs. ControlFlow, Journal of Big Data, 2015.

[81] A.E.C. Abdessamad, S. Omar, B.A. Rabie, B. Nicolas, and A. Abdelhakim, Mathematical programming models for scheduling in a CPU/FPGA architecture with heterogeneous communication delays, Journal of Intelligent Manufacturing, Springer US, 2015. pp. 1-12.

[82] T. Fei, Z. Lin, and L. Yuanjun, Job Shop Scheduling with FPGA-Based F4SA, Configurable Intelligent Optimization Algorithm, Part of the series Springer Series in Advanced Manufacturing, Springer International Publishing, 2015, pp. 333-347.

[83] W. Herroelen, B. De Reyck, and E. Demeulemeester, Resource-constrained project scheduling: a survey of recent developments, Computers & Operations Research, 1998, Vol. 25, No. 4, pp. 279-302.

[84] X. L. Zheng and L. Wang, A multi-agent optimization algorithm for resource constrained project scheduling problem, Expert Systems with Applications, Elsevier, Vol. 42, No. 15-16, September 2015, pp. 6039–6049.

[85] B.K. Hamilton, M. Inggs, and H.K.-H. So, Scheduling Mixed-Architecture Processes in Tightly Coupled FPGA-CPU Reconfigurable Computers, IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014, pp. 240.

[86] Q.-H. Khuat, D. Chillet, and M. Hubner, Considering reconfiguration overhead in scheduling of dependent tasks on 2D reconfigurable FPGA, NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, pp. 1-8.

[87] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo, Joint communication scheduling and interconnect synthesis for FPGA-based many-core systems, Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, pp. 1-4.

[88] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, T. Arslan, and I. Martinez, Runtime Scheduling, Allocation, and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence, International Journal of Reconfigurable Computing, Vol. 2013 (2013), Article ID 905057, pp. 1-32.

[89] S. Van de Vonder, E. Demeulemeester, and W. Herroelen, A classification of predictive-reactive project scheduling procedures, Journal of scheduling, Vol. 10 No. 3, 2007, pp. 195-207.

[90] R. H. Möhring and F. Stork, Linear preselective policies for stochastic project scheduling, Mathematical Methods of Operations Research, Vol. 52, No. 3, 2000, pp. 501-515.

[91] R. Roma, M. Sidia, and H. P. Tan, Design and analysis of a class-aware recursive loop scheduler for class-based scheduling, Elsevier, Vol. 63, No. 9-10, October 2006, pp. 839-863.

[92] J. D. Anderson and J. Wendt, Computational fluid dynamics, New York, McGraw-Hill, 1995.

[93] S. Chen, G. D. Doolen, Lattice Boltzmann method for fluid flows, Annual review of fluid mechanics, 1998, pp. 329-64.

[94] A. Sequeira, A. M. Artoli, A. S. Silva-Herdade, and C. Saldanha, Leukocytes dynamics in microcirculation under shear-thinning blood flow, Computers and Mathematics with Applications, 58(5), pp. 1035-1044, 2009.

[95] V. Rankovic, A. Kos, and V. Milutinovic, Bitonic Merge Sort Implementation on the Maxeler Dataflow Supercomputing System, Transactions on Internet Research, Vol. 9, No. 2, July 2013, pp. 34-42.

[96] K. Danne and M. Platzner, A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware, International Conference on Field Programmable Logic and Applications, 2005; 24-26 Aug. 2005 Page(s):568-573.

[97] K. Compton, L. Zhiyuan, J. Cooley, S. Knol, and S. Hauck, Configuration relocation and defragmentation for run-time reconfigurable computing, IEEE

Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 10, No. 3, June 2002.

[98] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, Dynamic scheduling of tasks on partially reconfigurable FPGAs, IEEE Proceedings-Computers and Digital Techniques, Vol. 147, No. 3, May 2000 Page(s):181-188.

[99] S. Banerjee, E. Bozorgzadeh, and N. Dutt, Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration, Proceedings - Design Automation Conference, 2005. 42nd, 13-17 June 2005 Page(s):335-340.

[100] E. El-Araby, M. Taher, T. El-Ghazawi, and J.Le Moigne, Prototyping Automatic Cloud Cover Assessment (ACCA) Algorithm for Remote Sensing On-Board Processing on a Reconfigurable Computer, Proceedings of the IEEE Conference on Field Programmable Technology, FPT'05, Singapore, December 11-14, 2005.

[101] J.L. Tripp, H.S. Mortveit, A. A. Hansson, and M. Gokhale, Metropolitan road traffic simulation on FPGAs, Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on, 18-20 April 2005 Page(s):117-126.

[102] A. Michalski, D. Buell, and K. Gaj, High-throughput reconfigurable computing: design and implementation of an IDEA encryption cryptosystem on the SRC-6E reconfigurable computer, Field Programmable Logic and Applications, 2005. International Conference on 24-26 August 2005, Page(s): 687- 690.

[103] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, Dynamic scheduling of tasks on partially reconfigurable FPGAs, IEE Proceedings - Computers and Digital Techniques, Vol. 147, No. 3, May 2000, pp. 181-188.

[104] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, Maximum performance computing with dataflow engines, In High-Performance Computing Using FPGAs, W. Vanderbauwhede and K. Benkrid, Eds. Springer-Verlag, 2013.

[105] D’ariano, A., Pacciarelli, D. and Pranzo, M., 2007. A branch and bound algorithm for scheduling trains in a railway network. European Journal of Operational Research, 183(2), pp.643-657.

Прилог

Табела 1: Укупно време извршавања у зависности од односа послова за хардвер заснован на протоку података и послова за процесор заснован на контроли тока.

Однос послова	Уштеда времена (*Time gain*)	Проточност (*Throughput*)	FCFS
0	20.934677	19.677945	19.347436
1	10.251790	9.802322	9.687217
2	6.045375	5.982859	5.604964
3	4.031196	4.008519	3.866492
4	2.978221	2.968537	2.906722
5	2.379296	2.374582	2.344153
6	2.011726	2.009187	1.992688
7	1.771775	1.770297	1.760648
8	1.607195	1.606279	1.600286
9	1.489711	1.489115	1.485207
10	1.403059	1.402655	1.400003
11	1.33739	1.337107	1.335246
12	1.286472	1.286268	1.284925
13	1.246218	1.246067	1.245074
14	1.213856	1.213742	1.212992
15	1.187457	1.18737	1.186793
16	1.165646	1.165578	1.165128

17	1.147421	1.147367	1.147011
18	1.13204	1.131996	1.13171
19	1.11894	1.118905	1.118673

Подаци о аутору

Ненад Королија је рођен 16.02.1978. у Београду, репуб̀лика Срб̀ија, где је завршио основну и средњу школу са одличним успехом. Уписао је основне студије на Електротехничком факултету, Универзитета Београду, 1997. године. Дипломирао је 2002. године на смеру за рачунарску технику и информатику.

Од 2003. године ради као стручни сарадник на катедри за Рачунарску технику и информатику Електротехничког факултета у Београду, а такође је учествовао и у реализацији више интернационалних пројеката, међу којима је и пројекат „Cache implications of non-blocking thread execution in a multithreaded architecture“, у трајању од 12 месеци, финансиран од стране FP7 пројекта HiPEAC (High-Performance Embedded Architectures and Compilers). Као стручни сарадник, б̀ио је ангажован на предметима из об̀ласти рачунарства. Последипломске студије на Електротехничком факултету у Београду, смер Софтверски системи, уписао је 2002. године и положио све испите предвиђене Наставним планом и програмом магистарских студија, са просечном оценом 10 (десет). Магистарски рад под насловим „Прављење распореда извршавања нити DTA архитектуре“ је одбранио 2009. код проф. др Вељка Милутиновића.

Аутор је више радова у часописима, од чега су три часописа на SCI листи, као и више радова об̀јављених на домаћим и интернационалним конференцијама у об̀ластима архитектуре рачунара, софтвера и дистриб̀уираних рачунарских система.

Радови об̀јављени у часописима међународног значаја – М20

Радови у међународним часописима са СЦИ листе (М22)

1. Huang, K., Liu, Y., Korolija, N., Carulli, J., Makris, Y.: Recycled IC Detection based on Statistical Methods, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 34, No. 6, pp. 947-960, Jun, 2015, DOI: 10.1109/TCAD.2015.2409267, Print ISSN 0278-0070.

Радови у међународним часописима са СЦИ листе (М23)

1. Popović, J., Bojić, D., Korolija, N.: Analysis of task effort estimation accuracy based on use case point size, IET Software, Vol. 9, Issue 6, December 2015, p. 166 – 173, DOI: 10.1049/iet-sen.2014.0254 , Print ISSN 1751-8806.

2. Korolija, N., Popović, J., Cvetanović, M., Bojović, M.: Dataflow Based Parallelization of Control-Flow Algorithms, Creativity in Computing and Dataflow Supercomputing, Advances in Computers, Vol. 104, DOI: 10.1016/bs.adcom.2016.09.003, 2017, Print ISBN 9780128119556.

Зборници међународних скупова – М30

Саопштења са међународног скупа штампана у целини (М33)

1. S. Marković, N. Korolija, S. Manasijević, Simulation of crack formation in aluminum billets resulting from direct chill electromagnetic casting, 13th International Foundrymen Conference, Innovative Foundry Processes and Materials, PROCEEDINGS BOOK, ISBN 978-953-7082-15-4, 16–17th May, Opatija, Croatia, pp. 239−250, 2013.

2. S. Marković, N. Korolija, E. Romhanji, S. Manasijević, M. Stakić, Simulating formation of cracks during cooling aluminium alloy, First Metallurgical & Materials Engineering Congress of South-East Europe (MME SEE 2013), PROCEEDINGS & BOOK OF ABSTRACTS ISBN 978-86-87183-24-7, 23−25. May, Belgrade, Serbia, pp. 121−128, 2013, (http://www.mme-see.org/scientific-information/program).

Магистарске и докторске тезе – М70

Одбрањена магистарска теза (М72):

1. Королија, Н.: Прављење распореда извршавања нити DTA архитектуре, Електротехнички факултет, Универзитет у Београду, 2009, М72.

Овај рад је делимично финансиран пројектима Министарства просвете, науке и техничког развоја Републике Србије (ТР32047 анд ИИИ44006).

PRILOG 1

Izjava o autorstvu

Potpisani Nenad Korolija

Izjavljujem

da je doktorska disertacija pod naslovom "Ubrzavanje izvršavanja vremenski zahtevnih softverskih aplikacija konfigurisanjem namenskog hardvera u vreme izvršavanja programa na višeprocesorskim računarima"

- rezultat sopstvenog istraživačkog rada,
- da predložena disertacija u celini ni u delovima nije bila predložena za dobijanje bilo koje diplome prema studijskim programima drugih visokoškolskih ustanova,
- da su rezultati korektno navedeni i
- da nisam kršio autorska prava i koristio intelektualnu svojinu drugih lica.

U Beogradu, 29.11.2016.

Nenad K.

Potpis doktoranta

PRILOG 2

Izjava o istovetnosti štampane i elektronske verzije doktorskog rada

Ime i prezime autora: Nenad Korolija

Naslov rada: Ubrzavanje izvršavanja vremenski zahtevnih softverskih aplikacija konfigurisanjem namenskog hardvera u vreme izvršavanja programa na višeprocesorskim računarima

Mentor: dr Veljko Milutinović, redovni profesor

Potpisani Nenad Korolija

Izjavljujem da je štampana verzija mog doktorskog rada istovetna elektronskoj verziji koju sam predao za objavljivanje na portalu Digitalnog repozitorijuma Univerziteta u Beogradu.

Dozvoljavam da se objave moji lični podaci vezani za dobijanje akademskog zvanja doktora nauka, kao što su ime i prezime, godina i mesto rođenja i datum odbrane rada.

Ovi lični podaci mogu se objaviti na mrežnim stranicama digitalne biblioteke, u elektronskom katalogu i u publikacijama Univerziteta u Beogradu.

U Beogradu, 29.11.2016.

Nenad K.

Potpis doktoranta

PRILOG 3

Izjava o korišćenju

Ovlašćujem Univerzitetsku biblioteku „Svetozar Marković“ da u Digitalni repozitorijum Univerziteta u Beogradu unese moju doktorsku disertaciju pod naslovom „Ubrzavanje izvršavanja vremenski zahtevnih softverskih aplikacija konfigurisanjem namenskog hardvera u vreme izvršavanja programa na višeprocesorskim računarima“, koja je moje autorsko delo. Disertaciju sa svim prilozima predao/la sam u elektronskom formatu pogodnom za trajno arhiviranje. Moju doktorsku disertaciju pohranjenu u Digitalnom repozitorijumu Univerziteta u Beogradu i dostupnu u otvorenom pristupu mogu da koriste svi koji poštuju odredbe sadržane u odabranom tipu licence Kreativne zajednice (Creative Commons) za koju sam se odlučio/la.

1. Autorstvo (CC BY)

2. Autorstvo – nekomercijalno (CC BY-NC)

3. Autorstvo – nekomercijalno – bez prerada (CC BY-NC-ND)

4. Autorstvo – nekomercijalno – deliti pod istim uslovima (CC BY-NC-SA)

5. Autorstvo – bez prerada (CC BY-ND)

6. Autorstvo – deliti pod istim uslovima (CC BY-SA)

(Molimo da zaokružite samo jednu od šest ponuđenih licenci.

Kratak opis licenci je sastavni deo ove izjave).

U Beogradu, 29.11.2016.

Potpis doktoranta

1. Autorstvo. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence, čak i u komercijalne svrhe. Ovo je najslobodnija od svih licenci.

2. Autorstvo – nekomercijalno. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca ne dozvoljava komercijalnu upotrebu dela.

3. Autorstvo – nekomercijalno – bez prerada. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, bez promena, preoblikovanja ili upotrebe dela u svom delu, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca ne dozvoljava komercijalnu upotrebu dela. U odnosu na sve ostale licence, ovom licencom se ograničava najveći obim prava korišćenja dela.

4. Autorstvo – nekomercijalno – deliti pod istim uslovima. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence i ako se prerada distribuira pod istom ili sličnom licencom. Ova licenca ne dozvoljava komercijalnu upotrebu dela i prerada.

5. Autorstvo – bez prerada. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, bez promena, preoblikovanja ili upotrebe dela u svom delu, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca dozvoljava komercijalnu upotrebu dela.

6. Autorstvo – deliti pod istim uslovima. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence i ako se prerada distribuira pod istom ili sličnom licencom. Ova licenca dozvoljava komercijalnu upotrebu dela i prerada. Slična je softverskim licencama, odnosno licencama otvorenog koda.