УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Радован Бирдић

Примена векторског проширења процесора MIPS за убрзање извршења програма на примеру неколико карактеристичних алгоритама

МАСТЕР РАД

Нови Сад, 2016

УНИВЕРЗИТЕТ У НОВОМ САДУ ●**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски – мастер рад
Аутор, **АУ**:		**Радован Бирдић**
Ментор, **МН**:		**Др Миодраг Темеринац**
Наслов рада, **НР**:		**Примена векторског проширења процесора MIPS за убрзање извршења програма на примеру неколико карактеристичних алгоритама**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2016
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кqучне речи, **ПО**:		**MIPS, оптимизације, векторизација**
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		**Циљ овог рада је да опише процеспримене векторског проширења MIPSпроцесора(MSA)на примеру неколико карактеристичних алгоритама.Дати су описи алгоритама и предложена су решења за сваки од њих. MSAпроширење је проверено на системима са процесорским језгром MIPS.**
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	**Др Жељен Трповски**
	Члан:	**Др Иван Каштелан** — Потпис ментора
	Члан, ментор:	**Др Миодраг Темеринац**

UNIVERSITY OF NOVI SAD●**FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		Radovan Birdić	
Mentor, **MN**:		**Miodrag Temerinac, PhD**	
Title, **TI**:		**Application of MIPS Vector Extensions for Acceleration of Program Execution in the Case of Several Characteristic Algorithms**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2016	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)			
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		MIPS, Optimizations, Vectorization	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		The purpose of this paper is to describe the process of applying the vector extensions MIPS processors (MSA) on several typical algorithms.There are given descriptions of algorithms and solutions for each of them. MSA extension is tested on systems with MIPS processor core.	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**Željen Trpovski, PhD**	
	Member:	**Ivan Kaštelan, PhD**	Menthor's sign
	Member, Mentor:	**Miodrag Temerinac, PhD**	

SADRŽAJ

1. Uvod....1
2. Teorijske osnove....3
2.1 Vektorizacija....3
2.2 SIMD....6
2.3 MIPS I6400 arhitektura i MSA proširenje....7
2.4 Karakteristični algoritmi....9
2.4.1 FFT....9
2.4.1.1 Radix-2 FFT....10
2.4.1.2 Split-Radix FFT....11
2.4.2 Chacha....11
2.4.3 Eigen....13
3. Analiza problema....15
4. Realizacija....16
4.1 Optimizacija FFT algoritma....17
4.2 Realizacija ChaCha....20
4.3 Realizacija Eigen biblioteke....22
5. Rezultati....25
5.1 Rezultati merenja FFT....25
5.2 Rezultati merenja Eigen....26
5.3 Rezultati merenja Chacha....29
6. Zaključak....31
7. Literatura....32

SPISAK SLIKA

Slika 2.1 RAW zavisnost 4
Slika 2.2 WAR zavisnost 5
Slika 2.3 RAR zavisnost 5
Slika 2.4 WAW zavisnost 6
Slika 2.5 SIMD instrukcija (A), klasično množenje (B) 6
Slika 2.6 Različite operacije nije moguće vektorizovati 7
Slika 2.7 Blok dijagram arhitekture MIPS I6400 8
Slika 2.8 Struktura DIT Radix - 2 algoritmaza N veličine 16 10
Slika 2.9 Struktura DIT Split Radix algoritmaza N veličine 16 11
Slika 2.10 Početno stanje ChaCha matrice 12
Slika 2.11 Row-round funkcija 12
Slika 2.12 Column-round funkcija 13
Slika 2.13 ChaCha algoritam “Quarter-Round” 13
Slika 4.1 Raspetljavanje petlje 17
Slika 4.2 Raspored Twiddle factor koeficijenata (A – pre, B – posle) 18

SPISAK TABELA

Tabela 4.1 Prikaz realizacije *madd* funkcije 16
Tabela 4.2 C kod (*mips_FFTFwd_RToCCS_F32_complex.c*) 19
Tabela 4.3 MSA kod (*msa_FFTFwd_RToCCS_F32_complex.c*) 20
Tabela 4.4 C kod chacha_generic.c.....21
Tabela 4.5 MSA kod chacha_vec_mips.c 21
Tabela 4.6 Izgled objektne datoteke (chacha_vec_mips.c.o).....22
Tabela 4.7 Primer realizacije funkcija padd.....22
Tabela 4.8 Primer realizacije složenije funkcije preduxp 24
Tabela 5.1 Vreme izvršavanja funkcija i ubrazanja u odnosu na referentni kod 25
Tabela 5.2 Rezultati merenje za testove bench_matrix_vector_eigen3 i bench_atv_eigen3 27
Tabela 5.3 Rezultati merenja za testove bench_symv_eigen3 i bench_syr2_eigen3.....28
Tabela 5.4 Rezultati merenja za test bench_ger_eigen3 29
Tabela 5.5 Rezultati merenja za Chacha algoritam.....30

SKRAĆENICE

FFT - *Fast Fourier transform*, Brza Furijeova transformacija

IFFT - *InverseFast Fourier transform*, Inverzna brza Furijeova transformacija

DFT - *Discrete Fourier transform,* Diskretna Furijeova transformacija

SIMD - *Single Instruction Multiple Data*

MSA - *MIPS SIMD Architecture*

SSL - *Secure Sockets Layer,*

TSL - *Transport Layer Security*

GCC - *GNU Compiler Collection*

1. Uvod

Na tržištu potrošačke elektronike sve se češće koriste platforme sa ograničenim resursima, tako da je bitno imati što kvalitetniju programsku podršku koja će omogućiti izvršavanje naprednih algoritama za obradu podataka. Kako bi se postigla što efikasnija obrada digitalnog signala u što kraćem vremenskom periodu, neophodno je maksimalno iskoristiti dostupne resurse. To se postiže paralelizacijom obrade (na visokom nivou) ili upotrebom vektorskih instrukcija (na niskom nivou), a vrlo često i kombinovanjem ovih tehnika. U današnje vreme se prilikom razvoja programske podrške uglavnom koristi pisanje sekvencijalnog koda, tako da se za ubrzanje programskog koda najčešće koriste SIMD optimizacije. Cilj ovog rada je da se na primeru nekoliko karakterističnih algoritama objasni primena vektorskog proširenja MIPS[1] procesora za ubrzanje izvršenja programa. Algoritmi koji su opisani u ovom radu i za koje su urađene optimizacije su:

- Brza Furijeova transformacija FFT
- Chacha algoritam za kriptografiju
- Eigen biblioteka za matematičke operacije

Ovi algoritmi su optimizovani uz pomoć MIPS vektorskog SIMD instrukcijskog seta sa vektorskim proširenjem MIPS MSA, koji je namenjen digitalnoj obradi signala i radi obradu 128-bitnih podataka. Kao ciljna platforma u ovom radu, izabrana je procesorska arhitektura MIPS I6400 i operativni sistem Linux. MIPS je izabran jer predstavlja dominantnu arhitekturu na tržištu ugrađenih (eng. *embedded*), namenskih uređaja za domaćinstvo, prvenstveno digitalnih televizora, mobilnih telefona itd. Poslednjih godina, procesorska jezgra MIPS uvode paralelizaciju i ubrzanja kroz implementaciju vektorskih proširenja postojećeg instrukcijskog seta, koje nude mogućnost rešavanja kompleksnih programskih rešenja.

Rad je podeljen u nekoliko delova. U poglavlju 2 opisani su vektorizacija kao jedna metoda optimizacije, SIMD programiranje i dat je detaljan opis MIPS arhitekture, sa posebnim naglaskom na osobine instrukcijskog skupa MIPS MSA. Dati su pregledi glavnih karakteristika algoritama koji su optimizovani (FFT, Eigen biblioteka i Chacha algoritam za enkripciju). U poglavlju 3, urađena je analiza problema, dok je u poglavlju 4 opisana realizacija predloženog rešenja problema.U poglavlju 5 je opisano okruženje za ispitivanje i priloženi su postignuti rezultati, dok su u poglavlju 6 rezimirani postignuti rezultati i dati mogući pravci daljeg razvoja.

2. Teorijske osnove

Program može biti optimizovan tako da se brže izvršava, da je u stanju da radi sa manjim trošenjem memorijskog prostora, drugih resursa ili manje trošenje energije. Optimizacija je način programiranja ili izmene programa, gde se postojeći program prilagođava za upotrebu uz određenu platformu, ili saradnju sa drugim programskim okruženjima u cilju povećanja brzine izvršavanja, smanjenja zazuzeća resursa(memorija, procesora, itd.). Optimizacija se može postići traženjem boljeg algoritma za određenu obradu, kao i korišćenjem specifičnih karakteristika hardvera. Metoda optimizacije koja se veoma često koristi jeste vektorizacija i najčešće se koristi za optimizovanje petlji kod višeprocesorskih sistema. Pored vektorizacije postoje i sledeće metode optimizacije: paralelizacija, optimizacija petlje (deljenje, kombinovanje, inverzija, odmotavanje, itd.), optimizacije generatora koda (alokacija registara, izbor instrukcija, planiranje instrukcija).

2.1 Vektorizacija

Vektorizacija je proces pretvaranja algoritma iz skalarnog u vektorski, gde jedna instrukcija može da vrši operaciju nad celim vektorom [1]. Ona dodaje obrazac paralelizma u programsku podršku gde je jedna instrukcija ili operacija primenjena na višestruke delove podataka. Danas mnogi mikroprocesori opšte namene imaju podršku za SIMD paralelizam (Intel, ARM, MIPS).

Od strukture koda zavisi kolika će biti dobit pri njegovoj vektorizaciji i najbolji rezultati se dobijaju kod sekvencijalno pisanog koda. Vektorizacija podrazumeva izmene u redosledu obavljanja operacija u okviru petlje, jer svaka SIMD instrukcija radi nad nekoliko elemenata

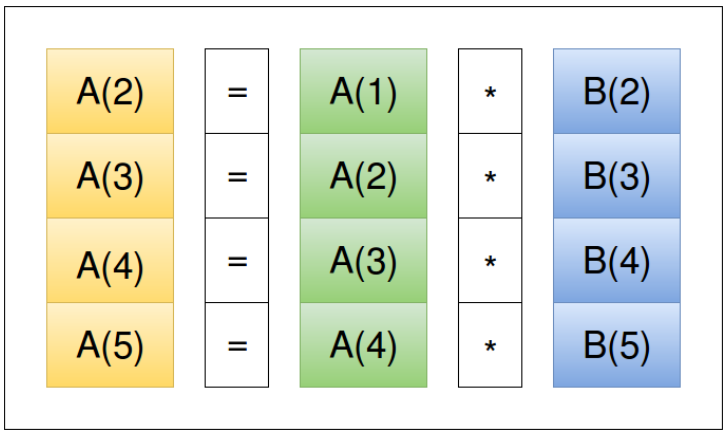
podataka odjednom. Vektorizacija je moguća samo ukoliko ta promena ne menja rezultate računanja.

Postoje brojne prednosti ove metode optimizacije, a najvažnije su sledeće:

- nezavisnost trenutnog rezultata od prethodnog rezultata
- smanjen broj instrukcija
- petlje se zamenjuju vektorskim instrukcijama → jednostavnije računanje
- manji broj pristupa memoriji
- brže vreme izvršenja
- niži troškovi zbog manjeg broja operacija u odnosu na skalarnu obradu

U odnosu na redosled izvršavanja operacija nad vektorima, postoje različite vrste zavisnosti u okviru vektorizacije:

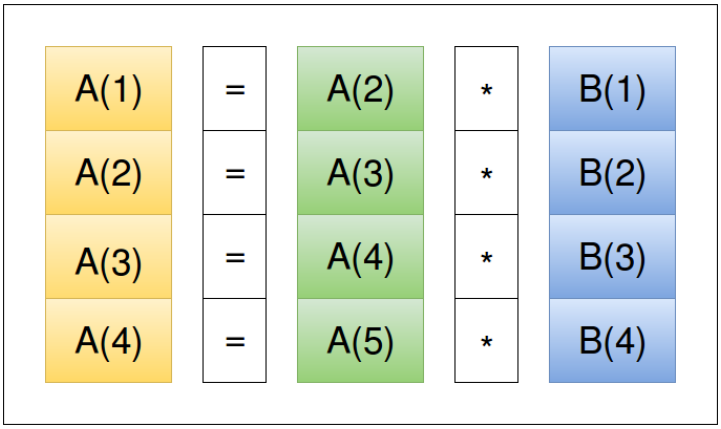
1. Čitanje posle pisanja tj. RAW (eng. *Read after write*) predstavlja zavisnost u okviru vektorizacije, gde je promenljiva upisana u jednoj iteraciji, a čita se u nekoj od sledećih. Ova zavisnost je poznata i kao protočna zavisnost. Takva petlja ne može biti bezbedno vektorizovana sa SIMD instrukcijama. Problem nastaje ukoliko se zahteva čitanje vrednosti A(1) u nekoj drugoj iteraciji, a da prethodna iteracija nije ni završena, tako da će biti učitana pogrešna vrednost.



Slika 2.1 RAW zavisnost

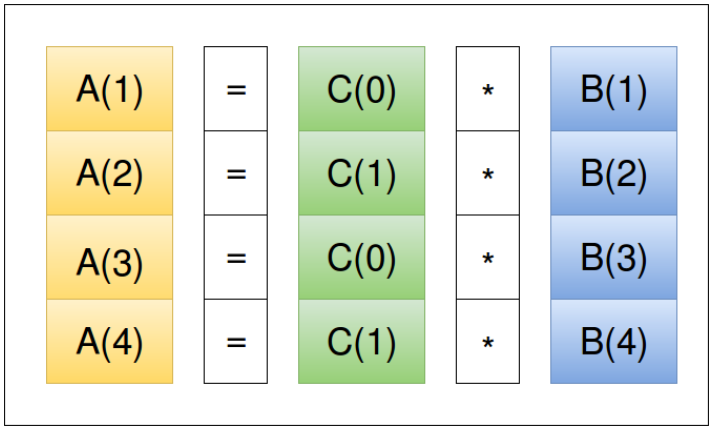
2. Pisanje posle čitanja tj. WAR (eng. *Write after read, anti-dependency*) predstavlja zavisnost gde je promenljiva u jednoj iteraciji pročitana, a upisana u nekoj narednoj iteraciji. Na Slici 2.2 je prikazan slučaj kada se iteracije za pisanje mogu izvršiti pre iteracija za čitanje. Vektorizacija je u ovom slučaju bezbedna, jer se iteracije sa većim indeksom neće izvršiti pre iteracija sa manjim indeksom, tako da ne može doći do

učitavanja pogrešne vrednosti. Dobija se identičan rezultat kao i kod nevektorizovane verzije koda.



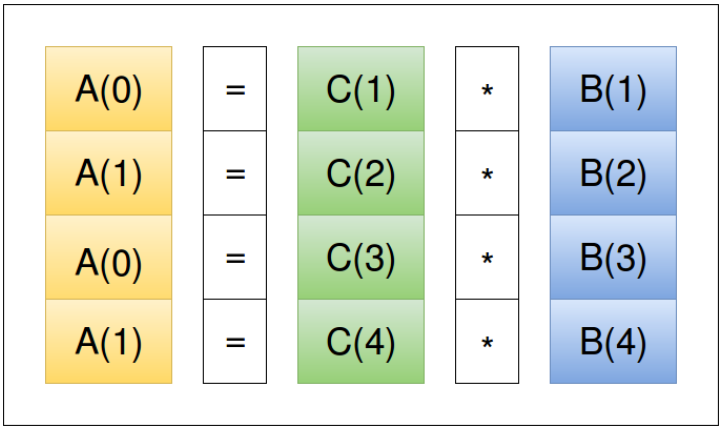
Slika 2.2 WAR zavisnost

3. Čitanje posle čitanja tj. RAR (eng. *Read after read*) predstavlja zavisnost u okviru vektorizacije, gde zapravo nema zavisnosti.Operacije čitanja ne zavise jedna od druge, jer nema operacije upisa niti ograničenja za vektorizaciju. Ova zavisnost je potpuno bezbedna.



Slika 2.3 RAR zavisnost

4. Pisanje posle pisanja tj. WAW (eng. *Write after write*) predstavlja zavisnost operacija gde može doći do upisivanja u istu promenljivu, tako da se prilikom čitanja može lako dobiti pogrešna vrednost. Ova vrsta zavisnosti nije pogodna za vektorizaciju.

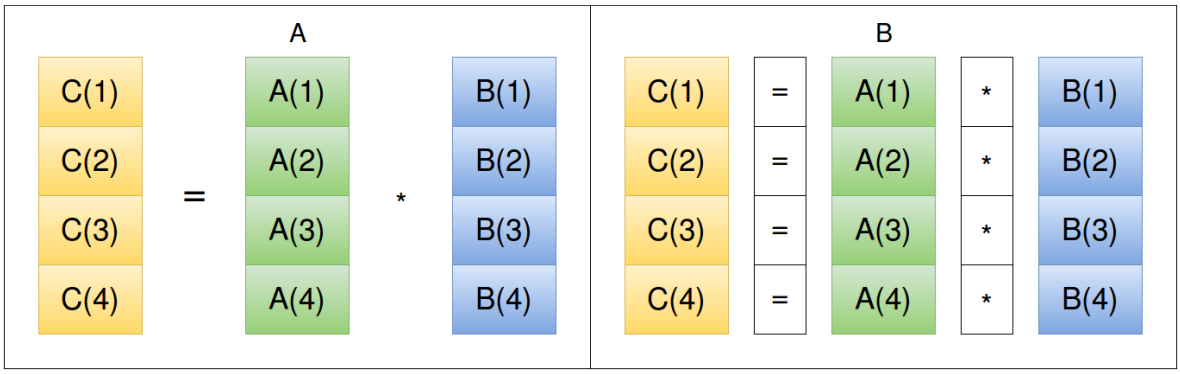


Slika 2.4 WAW zavisnost

2.2 SIMD

SIMD predstavlja korišćenje jedne instrukcije za obradu više podataka istovremeno, ali samo za istu operaciju. Nasuprot tome, sekvencijalni pristup omogućava da se sa jednom instrukcijom može obraditi samo jedan podatak [2].

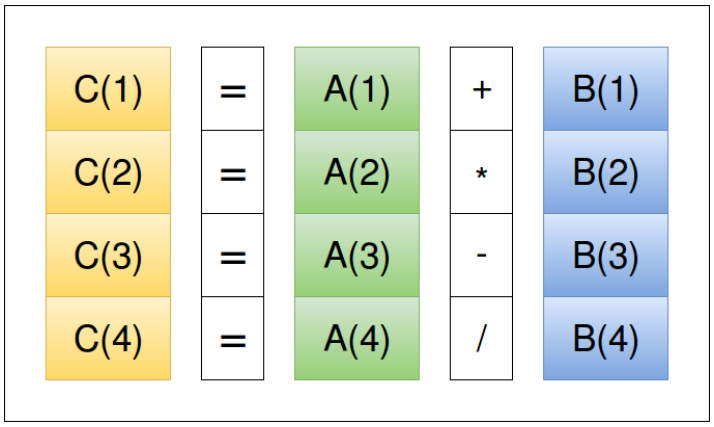
Sa SIMD procesorom se dobija mogućnost da se umesto niza istih naredbi npr. “pročitaj ovaj podatak”, zameni jednom naredbom “pročitaj n podataka”. Ovo oduzima daleko manje vremena u odnosu na učitavanje svakog podatka posebno, kao što je slučaj kod sekvencijalnog koda. Dodatna prednost je ta što SIMD arhitekture uključuju samo one instrukcije koje se mogu primeniti na sve podatke u jednoj operaciji. To znači da će se operacija koja treba da se izvrši nad 4 podatka, izvršiti u istom vremenskom periodu za sve. Ovaj tip arhitekture je savršeno pogodan za postizanje ubrzanja kod obrade velike količine podataka npr. kod obrade slike, zvuka i videa. Podaci se mogu podeliti na više različitih nezavisnih delova i mogu se vršiti različite obrade nad njima u isto vreme. Razlika između skalarnih i SIMD instrukcija je prikazana na Slici 2.5. gde se vidi da se 4 klasična množenja mogu zameniti jednom SIMD instrukcijom za množenje.



Slika 2.5 SIMD instrukcija (A), klasično množenje (B)

Problem koji se često javlja je što većina prevodioca ne generiše SIMD instrukcije prilikom prevođenja tipičnog C koda, tako da implementacija obično zahteva pomoć programera. Instrukcijski skupovi su različiti za svaku arhitekturu (Intel, Mips, Arm, itd.). Neki procesori nemaju u potpunosti realizovane SIMD instrukcije, tako da programer mora da obezbedi nevektorizovane realizacije istih ili da ih vektorizuje na drugačiji način.

SIMD instrukcije se ne mogu koristiti za obradu podataka za više različitih operacija. Primer je dat na Slici 2.6 gde su prikazane 4 različite operacije koje nije moguće vektorizovati korišćenjem SIMD instrukcija.



Slika 2.6 Različite operacije nije moguće vektorizovati

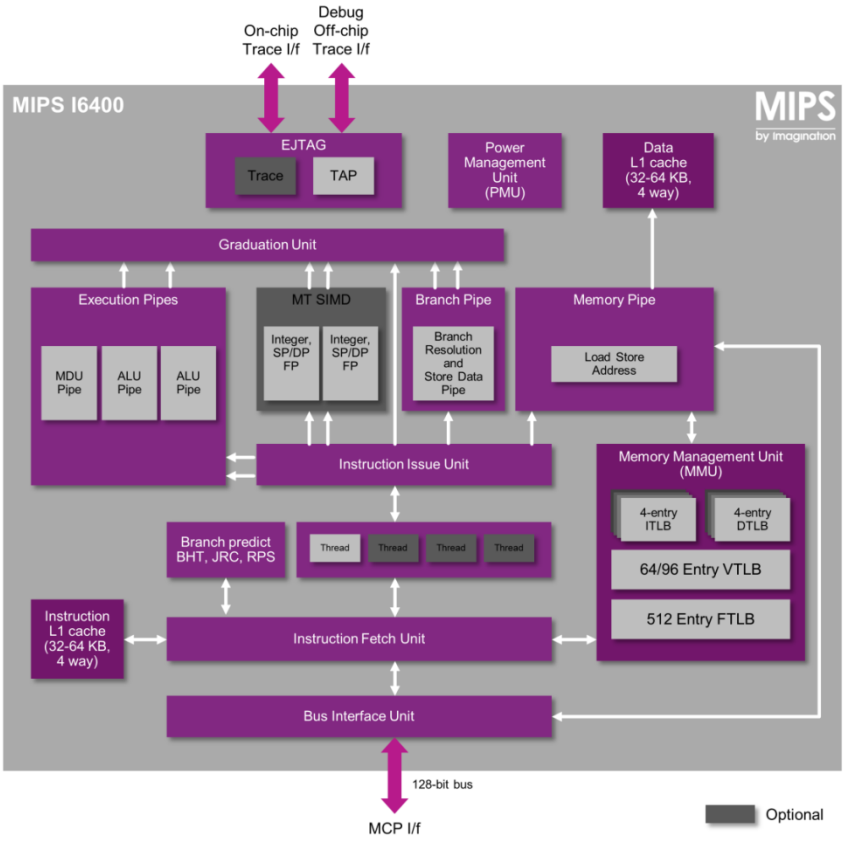
SIMD se našao u širokoj upotrebi kod audio i video obrade, u komunikaciji i kriptografiji, jer daje mnogo bolje rezultate nego tehnologije koje nemaju SIMD podršku. U video obradi se dobijaju dobri razultati kod 3D grafike, tako da su moderne grafičke kartice sa ugrađenim SIMD sistemom preuzele ovaj posao od procesora.

2.3 MIPS I6400 arhitektura i MSA proširenje

Na tržištu ugrađenih (engl. *embedded*) uređaja, jedna od dominatnih arhitektura je MIPS, koja je predstavnik RISC (eng. *Reduced instruction set computing*) arhitekture sa procesorom smanjenog skupa naredbi. Zbog njene niske potrošnja energije, slabe disipacije toplote, dostupnosti alata za razvoj programske podrške i cene integrisanih kola zasnovanih na ovoj arhitekturi, čine je pogodnom za različite namene. Pored standardnog instrukcijskog skupa, pojedini čipovi bazirani na arhitekturi MIPS nude i njegovo proširenje namenjeno digitalnoj obradi signala. Ovo proširenje specijalne namene (MSA) nudi vektorske instrukcije koje olakšavaju i ubrzavaju obradu velike količine podataka istog tipa, što je najčešće slučaj u algoritmima za obradu digitalnog signala. Na Slici 2.7. prikazan je blok dijagram MIPS arhitekture I6400, na kome se vide osnovne jedinice za izvršavanje celobrojnih operacija ALU (*eng. Arithmetic Logic Unit*), jedinice za množenje i deljenje MDU (*eng. Multiply and Divide*

Unit), memorijske jedinice MMU (*eng. Memory Management Unit*), jedinice za dekodovanje instrukcija ID (*eng. Instruction Decoder*), SIMD jedinica, itd.

Korišćena MIPS I6400 platforma, predstavlja MIPS64 arhitekturu i omogućava simultani multi-threading (SMT), 128-bitni SIMD, napredno upravljanje napajanjem i proširenja koherentnim operacijama multi-klastera. Ovaj procesor takođe ima i višenitne FPU / SIMD jedinice, pogodne za širok spektar obrade kao što su audio i video obrada.Skalarna i SIMD FP jedinica podržava 8- / 16- / 32- / 64-bitne celobrojne i fiksne tipove podataka, kao i 16- / 32- / 64-bitne tipove s pokretnim zarezom.[3]



Slika 2.7 Blok dijagram arhitekture MIPS I6400

Dodatno proširenje MIPS arhitekture, koje je korišćeno u ovom radu je MSA modul i predstavlja proširenje pete revizije standardnog MIPS-ovog instrukcijskog seta. Ovo proširenje je dizajnirano tako da odgovara velikom broju uobičajenih DSP algoritama, kao i drugim algoritmima koji obrađuju podatke na sličan način (prvenstveno vektorski).[4]

U odnosu na druga proširenja, MSA uvodi nove instrukcije na postojeći set koje omogućavaju efikasnu paralalnu obradu nad vektorskim oparacijama sa 128-bitnim registrima.

Operacije koje se najčešće koriste u multimedijalnoj obradi, a koje se mogu vektorizovati pomoću MSA uključuju:

- Sabiranje / oduzimanje

- Množenje i akumuliranje (Dot proizvod i jednostavna množenja)
- Logičko i aritmetičko pomeranje (sa opcionim zaokruživanjem)
- Ostale logičke operacije (I, ILI, XOR, itd)
- Uslovna selekcija / maskiranje
- Čitanje i upis podataka
- Pakovanje / umetanje

MSA sadrži vektorske registre širine 128 bita i izvršava operacije nad:

- celobrojnim podacima od 8, 16, 32 i 64 bita
- podacima sa nepokretnim zarezom od 16 i 32 bita
- podacima sa pokretnim zarezom od 32 i 64 bita

Tipovi podataka koji se koriste u MSA arhitekturi su:

- v16i8, vektor od 16 označena 8-bitna podataka celobrojnog tipa
- v16u8, vektor od 16 neoznačena 8-bitna podataka celobrojnog tipa
- v8i16, vektor od 8 označena 16-bitna podataka celobrojnog tipa
- v8u16, vektor od 8 neoznačena 16-bitna podataka celobrojnog tipa
- v4i32, vektor od 4 označena 32-bitna podataka celobrojnog tipa
- v4u32, vektor od 4 neoznačena 32-bitna podataka celobrojnog tipa
- v2i64, , vektor od 2 označena 64-bitna podataka celobrojnog tipa
- v2u64, vektor od 2 neoznačena 64-bitna podataka celobrojnog tipa
- v4f32, vektor od 4 32-bitna podataka tipa sa pokretnim zarezom
- v2f64, vektor od 2 64-bitna podataka tipa sa pokretnim zarezom

2.4 Karakteristični algoritmi

2.4.1 FFT

Poznato je da se svaki periodičan ili aperiodičan signal može razložiti na sumu sinusoidalnih komponenti, čije se učestanosti razlikuju ili za konačan iznos kao što je to slučaj kod periodičnih signala ili za beskonačno mali iznos kao kod aperiodičnih signala. Postupak izdvajanja harmonijskih komponenti iz složenog signala se naziva spektralna tj. harmonijska analiza signala.

Jedna od najčešće korišćenih transformacija u digitalnoj obradi signala je Diskretna Furijeova transformacija DFT(eng. *Discrete Fourier transform*) pomoću koje se obavlja prebacivanje

signala iz vremenskog u spektralni domen. Ova transformacija se koristi kao neophodan alat u spektralnoj analizi signala. Takođe, DFT omogućava efikasno izvođenje filtriranja u frekvencijskom domenu.

Brza Furijeova transformacija FFT (eng. *Fast Fourier transformation*) je algoritam za "brzo" izračunavanje vrednosti diskretne Furijeove transformacije. Ubrzanje u odnosu na uobičajen postupak izračunavanja diskretne Furijeove transformacije postiže se izbegavanjem ponovnog izračunavanja izraza koji se međusobno negiraju. [5]

2.4.1.1 Radix-2 FFT

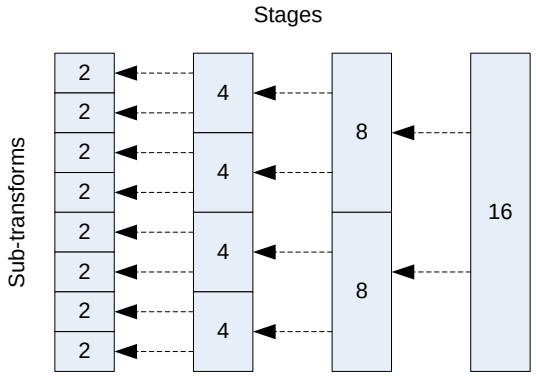
Prvi računski ekonomičan algoritam za izračunavanje Diskretne Furijeove transformacije predložili su Cooley i Turkey 1965. godine. Algoritam vrši računanje Brze Furijeove transformacije deljenjem transformacije veličine N na R jednakih pod-transformacija. Postupak se rekurzivno ponavlja dok veličina pod-transformacije ne bude veličine 1. Faktor deljenja R je poznat kao Radix, zbog toga je algoritam poznat kao Radix - R FFT. Postoje dve varijante Radix- R algoritma:

- Decimacija u vremenu DIT (eng. *Decimation in time*)
- Decimacija po frekvenciji DIF (eng. *Decimation in frequency*)

Kako su dve varijante samo slike u ogledalu, ovde će se razmatrati samo Decimacija u vremenu. Radix-2 DIT algoritam je izveden grupisanjem parnih i neparnih odbiraka ulaznog signala, što omogućava da se transformacije veličine N mogu podeliti u dve grupe pod-transformacije veličine N/2.

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} = \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{nk}$$

Celokupna procedura računanja za slučaj kada je N=16 je prikazana na Slici 2.8.



Slika 2.8 Struktura DIT Radix - 2 algoritmaza N veličine 16

Uprkos rekurzivnom definisanju procedure proračuna, njena visoka regularna struktura dozvoljava iterativnu implementaciju u kojoj se 2L-K pod-transformacija veličine 2K obavljaju u svakoj fazi blokova 2K odbiraka. Svaka pod-transformacija veličine 2K sastoji se od 2K-1

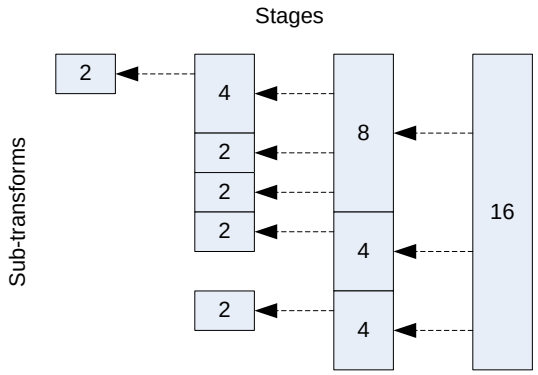
“butterfly” operacija koje uključuju kompleksno množenje sa N-tim korenom jedinice W_N^k (u FFT terminologiji poznatim kao “twiddle factor”) i po jedno kompleksno sabiranje i oduzimanje.

2.4.1.2 Split-Radix FFT

Pored fiksnih (radix) algoritama koji su opisani u 2.4.1.1, postoje i algoritmi kod kojih se radix menja u različitim fazama.Oni su poznati kao miks-radix algoritmi. S druge strane Split-Radix FFT algoritam je prvi put predložen 1978. godine od strane Yavne-a, a 1984. je popularizovan od strane Duhamel-a and Hollman-a. Predstavlja kombinaciju radix-2 i radix-4 u jednoj fazi računanja.Može se izvesti iz radix-2 FFT algoritma daljom decimacijom skupa neparnih odbiraka za faktor 2, Slika 2.9. [6]

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/4-1} x_{4n+1} W_{N/4}^{nk} + W_N^{3k} \sum_{n=0}^{N/4-1} x_{4n+3} W_{N/4}^{nk}$$

Split-Radix FFT algoritam zahteva 4Nlog2N - 6N + 8aritmetičkih operacija, što je za oko 20% manje u odnosu naRadix-2 FFT algoritam, koji zahteva 5Nlog2N aritmetičkih operacija.



Slika 2.9 Struktura DIT Split Radix algoritmaza N veličine 16

2.4.2 Chacha

Sigurna komunikacija na internetu zahteva da krajnje tačke komunikacije koriste različite kriptografske algoritme da bi se uspostavio bezbedan komunikacioni kanal.

Sigurnosni protokol komunikacije na internetu koji se najčešće koristi je SSL (eng. *Secure Sockets Layer*), a korisnici ga najčešće sreću u oblasti elektronske trgovine i elektronskog bankarstva. Nekada nije bio podržan od strane svih internet pretraživača, ali danas jeste. Prvu specifikaciju SSL sertifikata napravila je kompanija Netscape 1994.godine, tada tvorci najpopularnijeg pretraživača na svetu. Nakon nekoliko iteracija, SSL je evoluirao u tzv.*Transport Layer Security* – TLS, standard koji i danas koristimo u aktuelnoj verziji 1.2, a uskoro će početi upotreba i 1.3 standarda. Ipak, ime SSL se zadržalo i danas. [7]

Ovaj protokol funkcioniše kao klijent-server model. Klijent je strana koja inicira sigurnu komunikaciju, dok server odgovara na zahtev klijenta. Svrha je da omogući šifrovanu

komunikaciju direktno između korisnika i verifikovane lokacije, što prenos osetljivih informacija čini bezbednim, bez prisluškivanja, ometanja ili falsifikovanja.

Jedna od najčešće korišćenih kriptografskih biblioteka je OpenSSL u kojoj su realizovani SSL i TLS protokoli. U okviru OpenSSL biblioteke su podržani sledeći kriptografski algoritmi:

- Algoritmi za šifrovanje (eng. *Cipher*):
  - AES, RC2, RC4, RC5
- Algoritmi sa kriptografskim funkcijama (eng. *Cryptographic hash functions*)
  - MD5, MD4, MD2, SHA-1, SHA-2, RIPEMD-160, MDC-2
- Kriptografija sa javnim ključevima (eng. *Public-key cryptography*)
  - RSA, DSA, Diffie–Hellman key exchange, Elliptic curve

Kompanija Google je zamenila RC4 algoritam iz OpenSSL biblioteke sa ChaCha algoritmom za autentifikaciju poruka, koji se koristi za internet sigurnost. Uz pomoć ovog algoritma, Google osigurava HTTPS (TLS / SSL) saobraćaj između Chrome aplikacije na Android telefonima i Google web stranica.

Ulaz u funkciju ChaCha je 256-bitni ključ (eng. *key*), 64-bitni brojač (eng. *counter*), 64-bitni privremeni podaci i četiri 32-bitne konstante, raspoređeni u matricu veličine 4x4. Na Slici 2.10 je prikazano početno stanje matrice nad kojom se vrše kružne operacije (eng. *round function*).

$$\begin{pmatrix} 0x61707865 & 0x3320646E & 0x79622D32 & 0x6B206574 \\ key[0] & key[1] & key[2] & key[3] \\ key[4] & key[5] & key[6] & key[7] \\ counter[0] & counter[1] & nonce[0] & nonce[1] \end{pmatrix}$$

Slika 2.10 Početno stanje ChaCha matrice

ChaCha transformiše početnu matricu kroz 20 iteracija i dobijeni rezultat dodaje početnoj matrici pri čemu se dobija izlaz od 64 bita. Kružne operacije su podeljene na dve jednake operacije:

- Kružne operacije nad redovima (eng. *row-round function*), koje vrše izračunavanja nad matricom u neparnim iteracijama prikazane su na Slici 2.11. Kod ove operacije se u prvom prolazu rotiraju redovi matrice u desno, a u drugom se kolone rotiraju na gore.

$$\begin{pmatrix} x0 & x1 & x2 & x3 \\ x4 & x5 & x6 & x7 \\ x8 & x9 & xA & xB \\ xC & xD & xE & xF \end{pmatrix} \xrightarrow{row\ rot.} \begin{pmatrix} x0 & x1 & x2 & x3 \\ x7 & x4 & x5 & x6 \\ xA & xB & x8 & x9 \\ xD & xE & xF & xC \end{pmatrix} \xrightarrow{col.\ rot.} \begin{pmatrix} x0 & x4 & x8 & xC \\ x1 & x5 & x9 & xD \\ x2 & x6 & xA & xE \\ x3 & x7 & xB & xF \end{pmatrix}$$

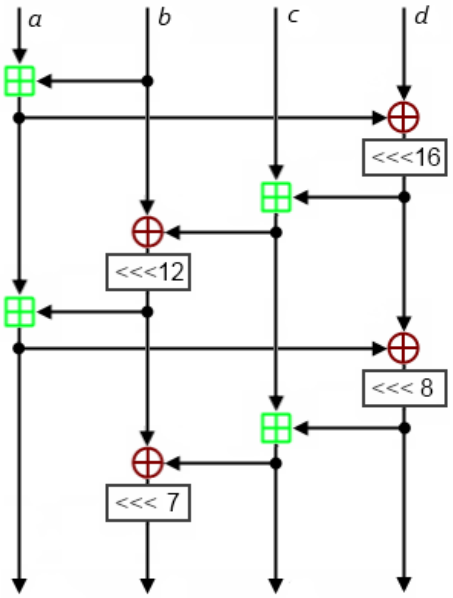
Slika 2.11 Row-round funkcija

- Kružne operacije nad kolonama (eng. *column-round function*) koje vrše izračunavanja nad matricom u parnim iteracijama su prikazane na Slici 2.12. Ova kružna operacija rotira vertikalne kolone matrice u odnosu na broj pozicije kolone u matrici. Nakon rotacije, dobijene vrste u matrici predstavljaju vektorski ulaz za ChaCha algoritam.

$$\begin{pmatrix} x0 & x4 & x8 & xC \\ x1 & x5 & x9 & xD \\ x2 & x6 & xA & xE \\ x3 & x7 & xB & xF \end{pmatrix} \xrightarrow{column\ rotation} \begin{pmatrix} x0 & x5 & xA & xF \\ x1 & x6 & xB & xC \\ x2 & x7 & x8 & xD \\ x3 & x4 & x9 & xE \end{pmatrix}$$

Slika 2.12 Column-round funkcija

U prvoj fazi ChaCha algoritma, ključevi se sabiraju sa konstantama. Nakon toga obavlja se operacija "ekskluzivno ili" tj. XOR između dobijenih rezultata i rotacija za određen broj bita. Na kraju se rezultat prethodno opisanih operacija smešta u ključeve. Ovaj algoritam je poznat kao "Quarter-Round" algoritam. Opisani algoritam je prikazan na Slici 2.13.




```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<=  8;
c += d; b ^= c; b <<<=  7;
```

Slika 2.13 ChaCha algoritam "Quarter-Round"

2.4.3 Eigen

Eigen je C++ biblioteka namenjena za izvršavanje matematičkih operacija nad matricama i vektorima. Biblioteka podržava operacije nad matricama svih veličina, od malih matrica fiksne veličine do proizvoljno velikih matrica. Eigen obezbeđuje podršku za sve standardne numeričke tipove, kompleksne, celobrojne, realne.[8]

Okruženje Renderscript [9] je napravljeno za pokretanje računskih intenzivnih operacija pri visokim performansama na operativnom sistemu Android. Prvenstveno okruženje Renderscript orijentisano je na paralelno izračunavanje podataka, ali takođe se dobijaju dobre performanse prilikom intezivnog opterećenja koje se dobijaju usled velikog broja operacija koje se ne mogu paralelizovati. Prilikom pokretanja programa na uređaju, okruženje Renderscript će raspoređivati posao na sve procesore koji se nalaze na uređaju kao što su procesori sa više jezgara, grafički procesori i procesori za digitalnu obradu signala, omogućavajući programerima da se fokusiraju na pisanje algoritama, a ne na to kako da resurse optimalno iskoriste ili kako da ubrzaju izvršavanje programa koristeći paralelizaciju. Okruženje Renderscript je posebno korisno za aplikacije koje obavljaju obradu slike, za računarsku fotografiju ili za prepoznavanje dvodimenzionalnih i trodimenzionalnih predmeta. Ukoliko neka platforma nema programsku podršku za osnovne linearne operacije nad matricama i vektorima u okviru Renderscript okruženja, automatski će se koristiti Eigen biblioteka, koju je u ovom slučaju bilo neophodno optimizovati za MSA arhitekturu.

3. Analiza problema

U prethodnim poglavljima su opisani algoritmi koji su predmet ovog rada i koje je potrebno optimizovati za MSA arhitekturu. Oni su izabrani iz razloga što na MSA arhitekturi mogu da pokažu rezultate vektorizacije korišćenjem SIMD instrukcijskog seta za potpuno različite vrste algoritama i operacija. Takođe, svaki od ovih algoritama je ubrzan za neku drugu arhitekturu (Intel, ARM, itd.) korišćenjem SIMD instrukcija, pa sa na osnovu toga može zaključiti da je moguće dobiti solidna ubrzanja i za MSA arhitekturu.

Algoritam FFT je već bio ubrzan korišćenjem C programskog jezika, gde je postojeći Radix-2 algoritam zamenjen sa Split-Radix FFT algoritmom. Dobijeno je ubrzanje od oko 20% u odnosu na početni FFT kod, koji je takođe napisan u C programskom jeziku. Dodatno ubrzanje se može postići iskorišćenjem osobine da je FFT transformacija relanog signala dužine N ekvivalentna FFT transformaciji kompleksnog signala dužine N/2 [10].

Ostali algoritmi i biblioteke koji su tema ovog rada (Eigen i ChaCha) nisu imali dodatne C optimizacije, pa su MSA optimizacije rađene u odnosu na referentni C kod.

Pre same realizacije, bilo je neophodno uraditi analizu koda i svake biblioteke, pronaći kritične tačke i uska grla, kao i funkcije koje troše najviše resursa.

4. Realizacija

Ovo poglavlje opisuje korake koje je bilo potrebno ispratiti kako bi se dobilo SIMD proširenje tj. vektorizacija opisanih biblioteka za MSA arhitekturu. Osnovni cilj je bio da se primenom vektorizacije instrukcijskog seta postigne manji broj potrebnih instrukcija, samim tim i manji broj pristupa memoriji, što na kraju rezultira smanjenjem procesorskog vremena koje se utroši na njihovo izvršavanje, (konkretno instrukcija Madd i Msub), a da se pri tome dobiju identični rezultati kao i kod izvršavanja tih instrukcija bez optimizacije, odnosno da rezultati budu nezavisni. Na primeru jednostavne računske operacije iz izraza (1), pokazano je kako funkcioniše instrukcija madd, što je prikazano u Tabeli 4.1.

$$r = a * b + c * d \qquad (1)$$

Programski prevodilac	**MIPS32r2 optimizacija**
e = mul(a, b) f = mul(c, d) r = add(e, f)	e = mul(a, b) r = madd(e, c, d)

Tabela 4.1 Prikaz realizacije *madd* funkcije

Programski prevodilac , "kompajler", (eng. *compiler*) je računarski program (ili skup programa) koji se koristi za prevođenje izvornog koda iz programskog jezika visokog nivoa (C, C++) na jezik nižeg nivoa (asemblerski jezik ili mašinski kod). GCC je vrsta programskog prevodioca koji podržava mnoge programske jezike, kao što su C, C++, Fortran, Java , itd. Takođe, GCC prevodilac obezbeđuje ugrađene funkcije za pristup MSA instrukcijskom setu [10]. Dodavanjam <msa.h> biblioteke u izvornu datoteku i korišćenjem odgovarajućih predefinisanih oznaka (-

mmsa -mhard-float -mfp64 -mnan=2008) prilikom prevođenja izvorne datoteke, uključuje se podrška za MSA. Ugrađene funkcije (eng. *intrinsics*) su dostupne u određenom programskom jeziku i za njihovu implementaciju je zadužen iskjučivo programski prevodilac (kompajler). Slično kao "*inline"* funkcije, programski prevodilac pri pozivu ugrađenih funkcija ubacuje odgovarajuću sekvencu automatski generisanih instrukcija, dok sa druge strane, kompajler je bliži ugrađenim funkcijama i može bolje da ih integriše i optimizuje. Kompajler omogućava korišćenje ugrađenih funkcija isključivo kada programer zahteva optimizaciju. Ove funkcije su takođe poznate i kao "*builtin"* funkcije. Svaka ugrađena funkcija ima prefiks u imenu "*__builtin_msa_*"*.

Kako bi se što efikasnije optimizovao neki kod, koriste se razne tehnike optimizacije. Jedna od najčešće korišćenih tehnika je raspetljavanje petlji. Ova tehnika je korišćena prilikom optimizacije FFT i ChaCha algoritma. Postupak je takav da se glavni deo petlje raspiše kako bi se više iteracija odradilo u jednoj, a ostatak do konačnog broja iteracija se odradi pojedinačno. Prednosti ove metode ogledaju se u smanjenom broju instrukcija petlje koje nisu deo same obrade, i u činjenici da se dobija više instrukcija u iteraciji petlje koje je moguće bolje rasporediti kako bi se više iskoristila dubina protočne strukture. Primer raspetljavanja petlje je dat na Slici 4.1.

```
/* petlja pre raspetljavanja */

for (i = 0; i < 16; ++i)
{
    A[i] = B[i] * C[i];
}
```

```
/* petlja posle raspetljavanja */

for (i = 0; i < 16; i += 4)
{
    A[i] = B[i] * C[i];
    A[i + 1] = B[i + 1] * C[i + 1];
    A[i + 2] = B[i + 2] * C[i + 2];
    A[i + 3] = B[i + 3] * C[i + 3];
}
```

Slika 4.1 Raspetljavanje petlje

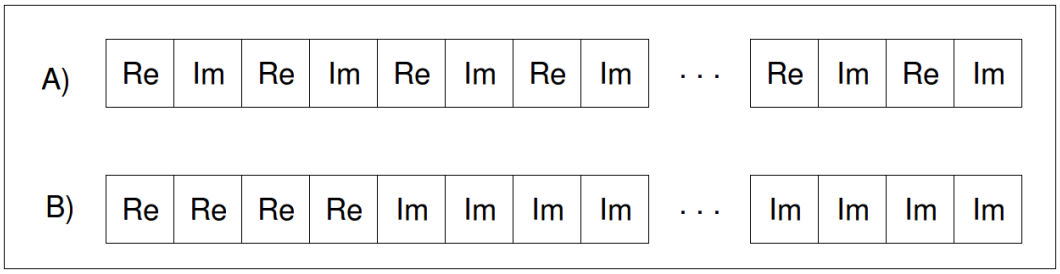
U nastavku rada će biti opisane ostale korišćene metode za svaki algoritam posebno.

4.1 Optimizacija FFT algoritma

Kompleksni signal se formira tako što parni odbirci realnog signala predstavljaju realan deo, a neparni odbirci signala predstavljaju imaginarni deo kompleksnog signala. Kada se nad novim, kompleksnim, signalom dužine N/2 primeni FFT, dobije se rezultat identičan kao i kada se primeni FFT nad realnim signalnom dužine N [11]. Obrada se ovim postupkom može dodatno ubrzati za oko 5 do 10%. Analizom asemblerskog koda dobijenog pomoću prevodioca,

potvrđeno je da ima prostora za dodatnu optimizaciju. Prevodilac nije uspevao da rasporedi instrukcije dovoljno dobro i postojali su delovi gde je procesor trošio vreme na čekanje rezultata. Ovaj problem se može rešiti ručnim pisanjem asemblerskog koda, kako bi se maksimalno iskoristilo procesorsko vreme. Usled složenosti koda, prevodilac nije uspevao da prepozna kombinacije više instrukcija koje bi mogao da zameni sa jednom [12]. Pomenuti algoritmi su takođe optimizovani za druge platforme (ARM, Intel), koje takođe sadrže vektorski SIMD instrukcijski set, što je bio jedan od glavnih razloga zbog koje je bilo potrebno optimizovati ih i za MIPS arhitekturu.

Pre početka izvršavanja FFT algoritma, urađena je inicijalizacija tabele sa twiddle faktorima, koji se koriste u računanju. Prilikom vektorizacije je pre svakog izračunavanja FFT-a, bilo potrebno učitati u registar pomenute koeficijente. Inicijalno, koeficijenti nisu bili raspoređeni u redosledu koji je bio najpovoljniji za vektorsko učitavanje i njihovu obradu. Koeficijenti su bili upisani u tabelu u izmešanom redosledu tj. realni i imaginari delovi su bili upisivani naizmenično jedan za drugim. Kod ovakvog redosleda je bilo potrebno da se nakon učitavanja koeficijenata, izvrši i preraspodela istih, jer je bilo potrebno da se u svakom registru nalazi 4 realna ili 4 imaginarna dela. Takvo premeštanje koeficijenata je oduzimalo dodatno vreme procesoru, a samim tim je i obrada trajala duže. Ovaj problem je rešen izmenom inicijalne tabele sa koeficijentima. Koeficijenti su upisivani u redosledu od 4 realna, pa zatim 4 imaginarna. Izgled prvobitne i izmenjene tabele, prikazane su na Slici 4.2.



Slika 4.2 Raspored Twiddle factor koeficijenata (A – pre, B – posle)

U tabelama 4.2 i 4.3 prikazan je izgled C kod-a i optimizovanog MSA koda. Petlja je optimizovana tako da su vektorizovane operacije sabiranja, oduzimanja i množenja, tako da je odjednom rađena obrada tj. određena operacija nad 4 podatka istovremeno.

```
float w_re, w_im;
float tmp1, tmp2, tmp3, tmp4, tmp5, tmp6;
float* w_re_ptr;
float* w_im_ptr;
unsigned int step;
for (i = 1; i < n1_4; ++i) {
```

```
      w_re = *w_re_ptr;
      w_im = *w_im_ptr;
      tmp1 = w_re * p_tmp[n1_2 + i].Re + w_im * p_tmp[n1_2 + i].Im;
      tmp2 = w_re * p_tmp[n1_2 + i].Im - w_im * p_tmp[n1_2 + i].Re;
      tmp3 = w_re * p_tmp[n3_4 + i].Re - w_im * p_tmp[n3_4 + i].Im;
      tmp4 = w_re * p_tmp[n3_4 + i].Im + w_im * p_tmp[n3_4 + i].Re;
      tmp5 = tmp1 + tmp3;
      tmp1 = tmp1 - tmp3;
      tmp6 = tmp2 + tmp4;
      tmp2 = tmp2 - tmp4;
      p_tmp[n1_2 + i].Re = p_tmp[i].Re - tmp5;
      p_tmp[n1_2 + i].Im = p_tmp[i].Im - tmp6;
      p_tmp[i].Re = p_tmp[i].Re + tmp5;
      p_tmp[i].Im = p_tmp[i].Im + tmp6;
      p_tmp[n3_4 + i].Re = p_tmp[n1_4 + i].Re - tmp2;
      p_tmp[n3_4 + i].Im = p_tmp[n1_4 + i].Im + tmp1;
      p_tmp[n1_4 + i].Re = p_tmp[n1_4 + i].Re + tmp2;
      p_tmp[n1_4 + i].Im = p_tmp[n1_4 + i].Im - tmp1;
      w_re_ptr += step;
      w_im_ptr -= step;
}
```

Tabela 4.2 C kod (*mips_FFTFwd_RToCCS_F32_complex.c*)

```
for (uint32_t i = 0; i < n1_4 >> 2; ++i) {
      v4f32 tmp_f0, tmp_f1, tmp_f2, tmp_f3, tmp_f4, tmp_f5, tmp_f6, tmp_f7;
      tmp_f0 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_re_12, 0);
      tmp_f1 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_im_12, 0);
      tmp_f2 = (v4f32)__builtin_msa_ld_w((void*)w_ptr_, 0);
      tmp_f3 = (v4f32)__builtin_msa_ld_w((void*)w_ptr_, 16);
      tmp_f4 = __builtin_msa_fmul_w(tmp_f2, tmp_f0);
      tmp_f4 = __builtin_msa_fmadd_w(tmp_f4, tmp_f3, tmp_f1);
      tmp_f5 = __builtin_msa_fmul_w(tmp_f2, tmp_f1);
      tmp_f5 = __builtin_msa_fmsub_w(tmp_f5, tmp_f3, tmp_f0);
      tmp_f0 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_re_34, 0);
      tmp_f1 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_im_34, 0);
      tmp_f6 = __builtin_msa_fmul_w(tmp_f2, tmp_f0);
      tmp_f6 = __builtin_msa_fmsub_w(tmp_f6, tmp_f3, tmp_f1);
      tmp_f7 = __builtin_msa_fmul_w(tmp_f2, tmp_f1);
      tmp_f7 = __builtin_msa_fmadd_w(tmp_f7, tmp_f3, tmp_f0);
      tmp_f2 = __builtin_msa_fadd_w(tmp_f4, tmp_f6);
      tmp_f3 = __builtin_msa_fsub_w(tmp_f4, tmp_f6);
      tmp_f4 = __builtin_msa_fadd_w(tmp_f5, tmp_f7);
      tmp_f5 = __builtin_msa_fsub_w(tmp_f5, tmp_f7);
      tmp_f0 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_re_0, 0);
```

```
    tmp_f1 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_im_0, 0);
    tmp_f6 = __builtin_msa_fsub_w(tmp_f0, tmp_f2);
    tmp_f7 = __builtin_msa_fsub_w(tmp_f1, tmp_f4);
    tmp_f2 = __builtin_msa_fadd_w(tmp_f0, tmp_f2);
    tmp_f4 = __builtin_msa_fadd_w(tmp_f1, tmp_f4);
    __builtin_msa_st_w((v4i32)tmp_f6, (void*)p_tmp_re_12, 0);
    __builtin_msa_st_w((v4i32)tmp_f7, (void*)p_tmp_im_12, 0);
    __builtin_msa_st_w((v4i32)tmp_f2, (void*)p_tmp_re_0, 0);
    __builtin_msa_st_w((v4i32)tmp_f4, (void*)p_tmp_im_0, 0);
    tmp_f0 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_re_14, 0);
    tmp_f1 = (v4f32)__builtin_msa_ld_w((void*)p_tmp_im_14, 0);
    tmp_f6 = __builtin_msa_fsub_w(tmp_f0, tmp_f5);
    tmp_f7 = __builtin_msa_fadd_w(tmp_f1, tmp_f3);
    tmp_f2 = __builtin_msa_fadd_w(tmp_f0, tmp_f5);
    tmp_f4 = __builtin_msa_fsub_w(tmp_f1, tmp_f3);
    __builtin_msa_st_w((v4i32)tmp_f6, (void*)p_tmp_re_34, 0);
    __builtin_msa_st_w((v4i32)tmp_f7, (void*)p_tmp_im_34, 0);
    __builtin_msa_st_w((v4i32)tmp_f2, (void*)p_tmp_re_14, 0);
    __builtin_msa_st_w((v4i32)tmp_f4, (void*)p_tmp_im_14, 0);
    p_tmp_re_12 += 4;
    p_tmp_im_12 += 4;
    p_tmp_re_34 += 4;
    p_tmp_im_34 += 4;
    p_tmp_re_14 += 4;
    p_tmp_im_14 += 4;
    p_tmp_re_0 += 4;
    p_tmp_im_0 += 4;
    w_ptr_ += 8;
```

Tabela 4.3 MSA kod (*msa_FFTFwd_RToCCS_F32_complex.c*)

4.2 Realizacija ChaCha

Referentni kod ChaCha algoritma je napisan u C programskom jeziku i na osnovu njega je bilo potrebno napisati optimizovanu verziju korišćenjem MSA instrukcijskog seta. Deo koda napisan u C programskom jeziku koji služi za generisanje početne matrice prikazan je u Tabeli 4.4. Optimizovan MSA kod prikazan je u Tabeli 4.5. Obzirom da je ChaCha enkripcijski algoritam, nije bilo moguće u potpunosti vektorizovati algoritam, jer su operacije koje se sprovode nad podacima i sami podaci u velikoj međuzavisnosti jedni od drugih.

```
void chacha_prepare(uint32_t *input, const uint8_t *sigma, const uint8_t *key,const
uint8_t *nonce, size_t counter) {
input[0] = U8TO32_LITTLE(sigma + 0);
  input[1] = U8TO32_LITTLE(sigma + 4);
  input[2] = U8TO32_LITTLE(sigma + 8);
```

```
  input[3] = U8TO32_LITTLE(sigma + 12);

  input[4] = U8TO32_LITTLE(key + 0);
  input[5] = U8TO32_LITTLE(key + 4);
  input[6] = U8TO32_LITTLE(key + 8);
  input[7] = U8TO32_LITTLE(key + 12);

  input[8] = U8TO32_LITTLE(key + 16);
  input[9] = U8TO32_LITTLE(key + 20);
  input[10] = U8TO32_LITTLE(key + 24);
  input[11] = U8TO32_LITTLE(key + 28);

  input[12] = counter;
  input[13] = U8TO32_LITTLE(nonce + 0);
  input[14] = U8TO32_LITTLE(nonce + 4);
  input[15] = U8TO32_LITTLE(nonce + 8);
}
```

Tabela 4.4 C kod chacha_generic.c

```
void chacha_prepare_mips_msa(uint32_t *input, const uint8_t *sigma, const uint8_t
*key,const uint8_t *nonce, size_t counter) {
v4i32 tmp0, tmp1, tmp2, tmp3, tmp4;
  tmp0 = __builtin_msa_ld_w((void*)sigma, 0);
  tmp1 = __builtin_msa_ld_w((void*)key, 0);
  tmp2 = __builtin_msa_ld_w((void*)key, 16);
  tmp4 = __builtin_msa_ld_w((void*)nonce, 0);
  tmp3 = __builtin_msa_fill_w(*(uint32_t *)&counter);
  tmp3 = __builtin_msa_sldi_b((v16i8)tmp4, (v16i8)tmp3, 12);
  __builtin_msa_st_w(tmp0, (void*)input, 0);
  __builtin_msa_st_w(tmp1, (void*)input, 16);
  __builtin_msa_st_w(tmp2, (void*)input, 32);
  __builtin_msa_st_w(tmp3, (void*)input, 48);
}
```

Tabela 4.5 MSA kod chacha_vec_mips.c

Analizom objektne datoteke, koja je dobijena korišćenjem alata “objdump” u okviru programskog prevodioca, utvrđeno je da je programski prevodilac sam menjao redosled izvršavanja nekih operacija. U delovima u kojima je bilo potrebno čekati na izvršenja neke operacije, programski prevodilac je ubacio izvršavanje neke druge operacije da bi se procesorsko vreme iskoristilo na što bolji način. Ovaj problem je prikazan u Tabeli 4.6.

```
00000100 <chacha_prepare_mips_msa>:
 100:  8fa20010     lw     v0,16(sp)
```

```
104:  78003822      ld.w   $w0,0(a3)
108:  780030e2      ld.w   $w3,0(a2)
10c:  7b02105e      fill.w $w1,v0
110:  780430a2      ld.w   $w2,16(a2)
114:  780c0819      sldi.b $w0,$w1[12]
118:  78002862      ld.w   $w1,0(a1)
11c:  780420e6      st.w   $w3,16(a0)
120:  780820a6      st.w   $w2,32(a0)
124:  78002066      st.w   $w1,0(a0)
128:  03e00009      jr     ra
12c:  780c2026      st.w   $w0,48(a0)
```

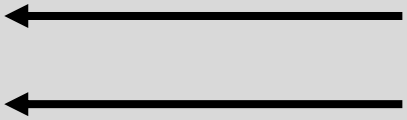


Tabela 4.6 Izgled objektne datoteke (chacha_vec_mips.c.o)

4.3 Realizacija Eigen biblioteke

Za razliku od prethodna dva opisana algoritma, funkcije iz Eigen biblioteke su već bile optimizovane za druge arhitekture, na osnovu kojih je urađen i MSA kod. Postojeće optimizacije su bile urađene za sledeće arhitekture:

- Intel (instrukcijski set SSE)
- ARM (instrukcijski set NEON).

Obzirom da ove dve arhitekture imaju veoma slične instrukcijske setove kao i MSA, osnovne operacije i deo složenih je bilo moguće uraditi sa ugrađenim MSA funkcijama koje obavljaju istu obradu. Ukoliko neka složenija operacija nije postojala u MSA instrukcijskom setu, bilo je neophodno koristiti kombinaciju više instrukcija, dok su za složenije operacije korišćene kombinacije funkcija iz MSA instrukcijskog seta. U Tabeli 4.7 se može videti da sve tri arhitekture imaju mogućnost obavljanja jednostavnih operacija sa jednom instrukcijom. U ovom slučaju je prikazana operacija za sabiranje.

```
typedef __m128  Packet4f;  --->SSE
typedef float32x4_t Packet4f; --->NEON
typedef v4f32 Packet4f; --->MSA


/* sabiranje */
template<> EIGEN_STRONG_INLINE Packet4f padd<Packet4f>(const Packet4f& a, const
Packet4f& b) { return _mm_add_ps(a,b); } --->SSE
template<> EIGEN_STRONG_INLINE Packet4f padd<Packet4f>(const Packet4f& a, const
Packet4f& b) { return vaddq_f32(a,b); } --->NEON
template<> EIGEN_STRONG_INLINE Packet4f padd<Packet4f>(const Packet4f& a, const
Packet4f& b) { return __builtin_msa_fadd_w(a, b); }--->MSA
```

Tabela 4.7 Primer realizacije funkcija padd

U Tabeli 4.8 se može videti slučaj kada nije moguće zameniti određenu obradu sa istim operacijama. Obzirom da je neophodno dobiti tačne rezultate određene obrade, bilo je potrebno koristiti druge postojeće MSA instrukcije.

```
template<> EIGEN_STRONG_INLINE Packet4f preduxp<Packet4f>(const Packet4f* vecs)
{
  Packet4f tmp0, tmp1, tmp2;
  tmp0 = _mm_unpacklo_ps(vecs[0], vecs[1]);
  tmp1 = _mm_unpackhi_ps(vecs[0], vecs[1]);
  tmp2 = _mm_unpackhi_ps(vecs[2], vecs[3]);
  tmp0 = _mm_add_ps(tmp0, tmp1);
  tmp1 = _mm_unpacklo_ps(vecs[2], vecs[3]);
  tmp1 = _mm_add_ps(tmp1, tmp2);
  tmp2 = _mm_movehl_ps(tmp1, tmp0);
  tmp0 = _mm_movelh_ps(tmp0, tmp1);
  return _mm_add_ps(tmp0, tmp2);
}  --->SSE

template<> EIGEN_STRONG_INLINE Packet4f preduxp<Packet4f>(const Packet4f* vecs)
{
  float32x4x2_t vtrn1, vtrn2, res1, res2;
  Packet4f sum1, sum2, sum;
  // NEON zip performs interleaving of the supplied vectors.
  // We perform two interleaves in a row to acquire the transposed vector
  vtrn1 = vzipq_f32(vecs[0], vecs[2]);
  vtrn2 = vzipq_f32(vecs[1], vecs[3]);
  res1 = vzipq_f32(vtrn1.val[0], vtrn2.val[0]);
  res2 = vzipq_f32(vtrn1.val[1], vtrn2.val[1]);
  sum1 = vaddq_f32(res1.val[0], res1.val[1]);
  sum2 = vaddq_f32(res2.val[0], res2.val[1]);
  sum = vaddq_f32(sum1, sum2);
  return sum;
} --->NEON

template<> EIGEN_STRONG_INLINE Packet4f preduxp<Packet4f>(const Packet4f* vecs)
{
  Packet4i temp0, temp1, temp2;
  temp0 = __builtin_msa_ilvr_w((Packet4i)vecs[1], (Packet4i)vecs[0]);
  temp1 = __builtin_msa_ilvl_w((Packet4i)vecs[1], (Packet4i)vecs[0]);
  temp0 = (Packet4i)__builtin_msa_fadd_w((Packet4f)temp0, (Packet4f)temp1);
  temp1 = __builtin_msa_ilvr_w((Packet4i)vecs[3], (Packet4i)vecs[2]);
  temp2 = __builtin_msa_ilvl_w((Packet4i)vecs[3], (Packet4i)vecs[2]);
  temp1 = (Packet4i)__builtin_msa_fadd_w((Packet4f)temp1, (Packet4f)temp2);
```

```
  temp2 = __builtin_msa_ilvr_d(temp1, temp0);
  temp1 = __builtin_msa_ilvl_d(temp1, temp0);
  return __builtin_msa_fadd_w((Packet4f)temp2, (Packet4f)temp1);
} --->MSA
```

Tabela 4.8 Primer realizacije složenije funkcije preduxp

5. Rezultati

Tokom rada, ispravnost koda je ispitivana pomoću emulatora Qemu. Qemu može pokrenuti operativne sisteme i programe namenjene namenskim arhitekturama (npr. ploča bazirana na MIPS arhitekturi) na drugom računaru (npr. X86_64 arhitektura). Emulator je bio dovoljan da bismo utvrdili ispravnost koda. Za merenje brzine, tačnije, ubrazanja optimizovanog koda u odnosu na referentni, koristili smo MIPS ploču sa MIPS I6400 procesorom.

5.1 Rezultati merenja FFT

U tabeli 5.1 su prikazani rezlutati merenja ubrazanja za FFTFwd i IFFT algoritam. Prva kolona u tabeli predstavlja veličinu reda računanja FFT-a. U narednim kolonama se mogu videti dobijena vremena merenja za originalan (C kod) i optimizovan (MSA) kod. Poslednja kolona predstavlja dobijena ubrzanja u procentima.

Veličina reda	Vreme izvršenja [μs]				Ubrzanje [%]	
	C kod		MSA			
	IFFT	FFTFwd	IFFT	FFTFwd	IFFT	FFTFwd
5	11.5417	**10.7400**	8.3958	**6.9300**	27.26	35.47
6	10.5625	**9.8800**	6.5208	**5.6700**	38.26	42.61
7	10.3958	**9.8700**	5.4792	**4.9300**	47.29	50.05
8	10.2500	**9.7500**	4.8958	**4.4700**	52.24	54.15
9	10.0625	**9.6700**	4.5000	**4.1500**	55.28	57.08
10	9.8958	**9.5800**	4.1875	**3.8700**	57.68	59.60
11	9.8542	**9.5400**	4.0000	**3.6900**	59.41	61.32
12	9.8958	**9.7500**	4.6250	**4.1500**	53.26	57.44
13	10.6667	**10.2900**	4.8125	**4.5700**	54.88	55.59
14	13.4792	**14.1700**	5.2917	**5.8900**	60.74	58.43
15	14.0833	**14.3300**	5.8125	**6.1700**	58.73	56.94
				MAX	60.74	61.32
				MIN	27.26	27.26

Tabela 5.1 Vreme izvršavanja funkcija i ubrazanja u odnosu na referentni kod

Iz priloženih rezultata se može videti da maksimalno ubrzanje iznosi 60,74% za IFFT i 61,32% za FTTFwd, a minimalno 27,26% u oba slučaja. To je očekivano ubrzanje, obzirom da smo radili obradu nad četiri podatka istovremeno.

5.2 Rezultati merenja Eigen

U tabelama 5.2, 5.3 i 5.4 su prikazani dobijeni rezultati merenja za Eigen biblioteku. U kolonama "original" su dati rezultati merenja referentnog C koda, a u kolonama "optimizovan" su dati rezultati merenja vektorizovanih Eigen funkcija pomoću MSA instrukcijskog seta. Rezultati merenja su dati u FLOPS (broj operacija sa pokretnim zarezom u sekundi, eng. *floating point operations per second)*. Korišćeno je 5 različitih testova i dobijeno je da je najveće poboljšanje primećeno kod testa bench_matrix_vector_eigen3 i iznosi 4,8 puta više MFLOPS-a u odnosu na referentni kod, najlošiji rezultat je dobijen u testu bench_ger_eigen3 gde je dobijeno duplo manje (0,5 puta) MFLOPS-a u odnosu na referenti kod.

Veličina testa	bench_matrix_vector_eigen3			bench_atv_eigen3		
	original [MFLOPS]	**optimizovan [MFLOPS]**	**poboljšanje [x]**	**original [MFLOPS]**	**optimizovan [MFLOPS]**	**poboljšanje [x]**
1500	6.75	24.00	3.56	9.82	28.80	2.93
1323	5.79	19.77	3.41	9.88	24.00	2.43
1167	6.23	18.68	3.00	9.34	21.79	2.33
1030	5.99	20.37	3.40	9.26	29.10	3.14
908	6.60	22.61	3.43	8.79	28.78	3.27
801	6.16	20.53	3.33	10.27	22.40	2.18
707	6.00	19.19	3.20	9.60	21.33	2.22
624	6.80	24.92	3.67	12.46	29.90	2.40
550	5.81	23.23	4.00	9.68	29.04	3.00
485	6.45	20.07	3.11	9.03	22.58	2.50
428	6.39	23.45	3.67	10.05	28.14	2.80
378	6.86	21.95	3.20	9.98	27.43	2.75
333	6.55	18.93	2.89	9.46	22.71	2.40
294	6.03	22.13	3.67	10.21	26.55	2.60
259	5.72	18.73	3.27	9.37	22.90	2.44
229	6.20	17.90	2.89	10.07	23.01	2.29
202	6.27	20.89	3.33	9.64	27.86	2.89
178	6.08	21.63	3.56	10.81	25.96	2.40
157	6.88	17.82	2.59	9.47	21.63	2.29
138	6.50	21.27	3.27	10.64	26.00	2.44
122	8.31	30.48	3.67	15.24	43.03	2.82
108	7.96	38.22	4.80	15.93	44.10	2.77
95	8.53	29.57	3.47	15.84	34.12	2.15
84	7.88	34.68	4.40	17.34	43.35	2.50
74	8.41	29.91	3.56	14.95	35.89	2.40
65	8.65	25.96	3.00	15.97	29.67	1.86
57	7.98	24.57	3.08	14.52	29.04	2.00
50	8.78	24.58	2.80	13.65	30.72	2.25
44	7.93	27.19	3.43	15.86	31.72	2.00
39	8.31	19.94	2.40	12.46	24.92	2.00
34	7.58	25.25	3.33	12.63	28.41	2.25
30	7.37	22.12	3.00	12.64	23.59	1.87
27	7.17	17.92	2.50	11.03	19.11	1.73
24	8.09	22.65	2.80	12.58	25.17	2.00
21	7.23	17.34	2.40	11.56	16.52	1.43
18	7.08	14.16	2.00	9.80	14.99	1.53

16	6.71	20.13	3.00	11.18	16.78	1.50
14	6.42	15.41	2.40	7.34	12.85	1.75
12	5.66	14.16	2.50	8.09	12.58	1.56
11	4.76	7.32	1.54	6.80	7.32	1.08
10	4.92	8.28	1.68	6.05	8.28	1.37
9	4.90	7.96	1.62	7.08	7.49	1.06
8	6.29	8.39	1.33	7.19	7.74	1.08
7	4.82	6.42	1.33	5.51	5.93	1.08
6	4.72	4.72	1.00	5.66	4.36	0.77
5	4.37	3.57	0.82	4.92	3.28	0.67
4	3.36	2.80	0.83	4.19	2.80	0.67
3	1.89	1.05	0.56	2.36	1.35	0.57
2	0.90	0.63	0.70	1.26	0.84	0.67
1	0.45	0.24	0.54	0.35	0.24	0.69
min	0.45	0.24	0.54	0.35	0.24	0.57
max	8.78	38.22	4.80	17.34	44.10	3.27
average	6.27	18.47	2.74	9.96	21.52	2.00

Tabela 5.2 Rezultati merenje za testove bench_matrix_vector_eigen3 i bench_atv_eigen3

	bench_symv_eigen3			**bench_syr2_eigen3**		
Veličina testa	**original [MFLOPS]**	**optimizovan[MFLOPS]**	**poboljšanje [x]**	**original [MFLOPS]**	**optimizovan [MFLOPS]**	**poboljšanje [x]**
1500	9.00	30.86	3.43	7.20	21.60	3.00
1323	9.34	30.55	3.27	7.31	21.00	2.87
1167	10.90	29.05	2.67	7.26	20.11	2.77
1030	10.18	31.34	3.08	6.79	18.52	2.73
908	8.79	31.66	3.60	7.20	19.79	2.75
801	11.20	30.80	2.75	6.84	17.60	2.57
707	8.72	29.53	3.38	7.38	19.19	2.60
624	10.68	33.23	3.11	7.48	18.69	2.50
550	10.56	30.98	2.93	7.26	19.36	2.67
485	10.04	30.11	3.00	7.53	18.07	2.40
428	9.38	28.14	3.00	7.82	20.10	2.57
378	10.97	27.43	2.50	7.32	18.29	2.50
333	9.46	28.39	3.00	7.74	17.03	2.20
294	10.21	29.50	2.89	7.38	16.60	2.25
259	9.37	24.24	2.59	7.36	18.73	2.55
229	10.07	26.85	2.67	7.32	16.11	2.20
202	10.45	25.07	2.40	7.83	15.67	2.00
178	10.81	24.33	2.25	7.49	17.70	2.36
157	10.10	27.54	2.73	7.57	16.83	2.22
138	13.00	29.25	2.25	8.36	19.50	2.33
122	12.19	28.14	2.31	8.31	18.29	2.20
108	11.94	31.85	2.67	8.96	20.48	2.29
95	12.32	23.35	1.89	8.53	20.16	2.36
84	11.56	26.68	2.31	7.88	17.34	2.20
74	11.21	22.43	2.00	8.41	14.95	1.78
65	11.54	20.77	1.80	7.99	14.83	1.86
57	12.28	19.96	1.63	7.98	13.31	1.67
50	11.17	17.55	1.57	7.68	12.29	1.60
44	10.57	17.30	1.64	7.32	13.59	1.86
39	10.68	13.59	1.27	7.48	12.46	1.67
34	10.33	14.20	1.37	7.10	10.33	1.45
30	9.83	12.64	1.29	7.37	11.06	1.50
27	8.96	11.03	1.23	7.17	11.94	1.67
24	8.71	12.58	1.44	7.08	10.30	1.45
21	7.88	10.20	1.29	6.67	9.63	1.44
18	7.96	8.49	1.07	6.37	9.10	1.43
16	7.19	8.39	1.17	6.29	9.15	1.45
14	7.01	7.01	1.00	5.93	8.56	1.44

12	6.29	6.29	1.00	5.66	8.09	1.43
11	5.60	5.29	0.94	5.95	7.93	1.33
10	5.62	5.24	0.93	5.24	6.55	1.25
9	4.55	4.55	1.00	5.31	5.31	1.00
8	4.19	4.19	1.00	6.29	5.59	0.89
7	3.50	3.85	1.10	5.93	5.14	0.87
6	3.54	3.54	1.00	6.29	3.77	0.60
5	3.28	3.02	0.92	5.62	3.57	0.64
4	2.29	2.10	0.92	5.03	3.15	0.63
3	1.42	1.29	0.91	4.04	2.36	0.58
2	0.90	0.79	0.88	2.65	1.40	0.53
1	0.35	0.29	0.82	1.14	0.74	0.65

min	0.35	0.29	0.82	1.14	0.74	0.53
max	13.00	33.23	3.60	8.96	21.60	3.00
average	8.56	18.51	1.96	6.84	13.24	1.84

Tabela 5.3Rezultati merenja za testove bench_symv_eigen3 i bench_syr2_eigen3

	bench_ger_eigen3		
Veličina testa	**original [MFLOPS]**	**optimizovan [MFLOPS]**	**poboljšanje [x]**
1500	5.14	14.40	2.80
1323	5.09	14.00	2.75
1167	5.03	14.53	2.89
1030	4.85	14.55	3.00
908	5.28	14.39	2.73
801	5.13	15.40	3.00
707	5.33	14.76	2.77
624	5.34	14.95	2.80
550	5.28	14.52	2.75
485	5.02	13.90	2.77
428	5.02	14.07	2.80
378	4.99	13.72	2.75
333	5.32	14.19	2.67
294	5.11	13.28	2.60
259	4.68	12.88	2.75
229	5.03	13.42	2.67
202	4.82	12.54	2.60
178	5.41	12.17	2.25
157	5.05	12.62	2.50
138	5.32	13.00	2.44
122	7.62	18.29	2.40
108	8.43	20.48	2.43
95	7.92	15.84	2.00
84	8.67	21.68	2.50
74	8.41	14.95	1.78
65	7.99	13.84	1.73
57	7.98	13.31	1.67
50	7.68	11.17	1.45
44	7.93	13.59	1.71
39	7.48	10.68	1.43
34	7.10	12.63	1.78
30	7.37	13.61	1.85
27	7.17	11.03	1.54
24	7.08	16.18	2.29
21	7.23	12.39	1.71
18	6.37	11.58	1.82
16	6.29	14.38	2.29
14	4.82	10.28	2.13

12	4.36	11.32	2.60
11	4.33	9.52	2.20
10	7.86	7.86	1.00
9	7.49	7.08	0.94
8	6.71	8.39	1.25
7	7.71	6.42	0.83
6	6.29	5.15	0.82
5	6.05	4.37	0.72
4	5.03	5.30	1.05
3	5.66	2.98	0.53
2	4.19	2.10	0.50
1	1.57	1.05	0.67

min	1.57	1.05	0.50
max	8.67	21.68	3.00
average	6.06	12.09	2.04

Tabela 5.4 Rezultati merenja za test bench_ger_eigen3

5.3 Rezultati merenja Chacha

U tabeli 5.5 su prikazani dobijeni rezultati merenja za Chacha algoritam. Prikazana vremena su data u mikrosekundama i predstavljaju vreme potrebno za generisanje koda za enkripciju, tačnije vreme ulaska i izlazka iz funkcije za generisanje šifre.

Može se primetiti da maksimalno ubrzanje koje je dobijeno u ovom slučaju, korišćenjem MSA koda, iznosi svega 5,18% u odnosu na početni C kod. To je zbog toga što sama priroda algoritma ne dozvoljava da kod bude bolje optimizovan, zbog same bezbednosti algoritma. Podaci i računanja koje se vrše nad njima su u velikoj zavisnosti jedni od drugih, tako da algoritam nije pogodan za vektorizaciju kako se bezbednost ne bi dovela u pitanje.

Vreme izvršenja [μs]		Ubrzanje[%]
C kod	MSA	
92352	91200	1.25
94753	93217	1.62
92064	89856	2.40
94945	90625	4.55
92160	88896	3.54
94945	90529	4.65
92064	88896	3.44
94465	90624	4.07
92160	88608	3.85
94464	91584	3.05
92064	88704	3.65
94464	89568	5.18
92928	88608	4.65
93408	91584	1.95
92064	88704	3.65
94464	90624	4.07
93504	88704	5.13
94464	90720	3.96

92640	88608	4.35
94464	90816	3.86
93504	88704	5.13
93408	91776	1.75
92160	88704	3.75
94464	89568	5.18
92928	88608	4.65
94464	91584	3.05
92161	88704	3.75
94464	90624	4.07
92929	88608	4.65
93408	90528	3.08
92161	88704	3.75
94464	90624	4.07
92929	88609	4.65
	MAX	5.18
	MIN	1.25

Tabela 5.5 Rezultati merenja za Chacha algoritam

6. Zaključak

U ovom radu je opisana optimizacija pomenutih algoritama uz pomoć MSA instrukcijskog seta. Na osnovu dobijenih rezultata može se zaključiti da je optimizacija kod nekih algoritama dala bolje, a kod nekih lošije rezultate. Rezultati zavise od same prirode algoritma, što je pokazano na primeru ChaCha, gde nije bilo moguće dobiti veće ubrzanje od svega nekoliko procenata. Nasuprot tome, FFT algoritam je dao odlične rezultate jer u njegovoj obradi ima dosta operacija koje je bilo moguće vektorizovati. Dalji pravci razvoja bi mogli biti usmereni na primenu ovih postojećih MIPS optimizacija, na nove buduće MIPS platforme. Takođe, moguće je uraditi dodatne analize alatima za profajliranje biblioteke, kojima bi se utvrdila nova uska grla, kritične tačke i problemi koji bi eventualno mogli biti rešeni sa dodatnim optimizacijama.

7. Literatura

[1] A Guide to Vectorization with Intel® C++ Compilers, https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

[2] SIMD, http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

[3] MIPS I6400, https://imgtec.com/mips/warrior/i-class-i6400-multiprocessor-core/

[4] MSA, https://gcc.gnu.org/onlinedocs/gcc/MIPS-SIMD-Architecture-_0028MSA_0029-Support.html

[5] Miodrag Temerinac, Stevan Berber, Željko Lukač : *Osnovi algoritma i struktura DSP 1*, Univerzitet u Novom Sadu, Fakultet Tehničkih Nauka, 2014

[6] P. Duhamel and H. Hollmann, “Split-radix FFT algorithm,” Electron. Lett., vol. 20, no. 1, pp. 14–16, 1984

[7] Vectorization of ChaCha Stream Cipher, Martin Goll, Shay Gueron, Ruhr-University Bochum, Germany Department of Mathematics, University of Haifa, Israel Intel Corporation, Israel Development Center, Haifa, Israel

[8] Eigen, http://eigen.tuxfamily.org/

[9] Renderscript, http://developer.android.com/guide/topics/renderscript/index.html

[10] GCC, https://gcc.gnu.org/

[11] Stanislav Očovaj and Željko Lukač “Optimization of Conjugate-Pair Split-Radix FFT Algorithm for SIMD Platforms”, IEEE International Conference on Consumer Electronics, Las Vegas, USA, 2014

[12] Kevin R. Wadleigh,Isom L. Crawford, „Software Optimization for High-performance Computing“, 2014.