Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Iztok Ramovš

Dualna hitra gradientna metoda na grafičnih procesnih enotah

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2018

Copyright. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Optimizirajte paralelno izvedbo iterativnega optimizacijskega algoritma v nizkolatenčnem okolju Linux za uporabo s prediktivnim regulatorjem. Za izhodišče vzemite algoritem z dualno hitro gradientno metodo (angl. Dual Fast Gradient Method), izvedeno z generatorjem kode QPgen, in njegovo že delno optimizirano paralelno izvedbo za večjedrni procesor. Rešitev iščite v naprednejši optimizaciji matrično-vektorskih operacij z učinkovitejšimi knjižnicami, z boljšo uporabo medpomnilnika ali pa z uporabo grafičnega procesorja v okolju CUDA ali OpenCL.

Na tem mestu bi se rad zahvalil mentorju izr. prof. dr. Urošu Lotriču, ki mi je bil vedno na voljo pri pisanju diplomske naloge. Zahvalil bi se rad tudi dr. Samu Gerkšiču, ki mi je omogočil delo na projektu, iz katerega je nastala diplomska naloga.

Zahvaljujem se tudi vsem tistim, ki jih nisem poimensko omenil in so kakorkoli posredno ali neposredno pomagali pri izdelavi diplomske naloge.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Ozadje in namen algoritma prediktivnega regulatorja **3**

2.1 Tokamak 3

2.2 Prediktivni regulator toka in oblike plazme v tokamaku 6

2.3 Metoda prediktivnega vodenja 6

2.4 Metoda za reševanje optimizacijskega problema 8

2.5 Metoda DHGM in orodje QPgen 9

3 Obstoječa izvedba **11**

3.1 Struktura projekta 11

3.2 Funkcija qp 15

3.3 Obstoječe prilagoditve 18

4 Prilagojena zaporedna implementacija **21**

4.1 Razvojno okolje 21

4.2 Novo ogrodje in struktura projekta 22

4.3 Preverjanje pravilnosti in natančnosti 23

4.4 Funkcija qp 24

5 Ogrodje OpenCL 27
5.1 Arhitektura in abstraktni prikaz strojne opreme 27
5.2 Model izvajanja 28
5.3 Pomnilniška struktura 30
5.4 Programski jezik 31

6 Izvedba metode DHGM z OpenCL 33
6.1 Paralelizacija pomožnih funkcij 33
6.2 Premik celotnega poteka v en ščepec 35
6.3 Prilagoditev dostopov do podatkov 39
6.4 Spremembe računanja spremenljivke theta 40

7 Meritve in rezultati 41
7.1 Meritve natančnosti 42
7.2 Meritve časa izvajanja 44

8 Zaključek 49

Literatura 51

Seznam uporabljenih kratic

kratica	angleško	slovensko
DHGM	Dual Fast Gradient Method	dualna hitra gradientna metoda
FMPCFMPC	Fast Model Predictive Control for Magnetic Plasma Control	projekt Hitro prediktivno vodenje za regulacijo plazme
MPC	model predictive control	prediktivno vodenje
CPE	central processing unit	centralna procesna enota
GPE	graphics processing unit	grafična procesna enota
Intel MKL	Intel Math Kernel Library	matematična jedrska knjižnica Intel
OpenMP	Open Multi-Processing	jezik OpenMP
OpenCL	Open Computing Language	jezik OpenCL

Povzetek

Naslov: Dualna hitra gradientna metoda na grafičnih procesnih enotah

Avtor: Iztok Ramovš

V napravah tipa tokamak je eden izmed večjih izzivov proces vodenja oblike in toka plazme, ki je pod konstantnim vplivom motenj. Tega izziva se je bilo možno lotiti z uporabo metode prediktivnega vodenja. Reševanje optimizacijskega problema prediktivnega vodenja je bilo narejeno z algoritmom z dualno hitro gradientno metodo. Vendar pa se je izkazalo, da se narejena implementacija izvaja prepočasi za hitro dinamiko v napravah tokamak. Cilj te diplomske naloge je ponovna implementacija te metode z uporabo paralelnih tehnologij z namenom hitrejšega izvajanja. Pri tem moramo paziti, da paralelna implementacija probleme rešuje pravilno in z enako natančnostjo kot njena sekvenčna izvedba. Pohitritev poizkušamo doseči s paralelnim izvajanjem algoritma na splošno namenskih grafičnih procesnih enotah z izbrano tehnologijo OpenCL.

Ključne besede: Tokamak, MPC, OpenCL, dualna hitra gradientna metoda, paralelna implementacija, GPE.

Abstract

Title: Dual Fast Gradient Method on Graphical Processing Units

Author: Iztok Ramovš

One of the biggest challenges in tokamak devices is to maintain plasma shape which is affected by constant disturbances. One of the possible solutions for this challenge is usage of model predictive control methodology. Optimization problem which represents used methodology was solved with dual fast gradient method. However, described implementation was too slow. The goal of this thesis is to re-implement the existing implementation using parallel technologies. New implementation must be able to provide solutions with the same level of accuracy as current serial implementation. We attempt to achieve faster execution with parallel program running on general purpose graphics processing units. Chosen technology is OpenCL framework.

Keywords: Tokamak, MPC, OpenCL, dual fast gradient method, parallel implementation, GPU.

Poglavje 1

Uvod

Elektrarne po celem svetu pretvarjajo velike količine mehanske energije v električno energijo. Za ta proces potrebujejo fosilna goriva, jedrsko fisijo ali obnovljive vire, na primer vodo ali veter. Zaradi vedno večje uporabe električne energije se išče nove alternativne vire. Ena od alternativ naštetim virom bi lahko bila tudi jedrska fuzija. Trenutno je raziskovanje za izrabo energije, ki se sprošča pri jedrski fuziji, najbolj usmerjeno v tako imenovani tokamak.

Tokamak je eksperimentalna naprava za magnetno omejevanje plazme, katere cilj je proizvajanje električne energije s pomočjo jedrske fuzije [15, 17]. V tokamaku je vakuumska komora v obliki torusa, v kateri pri visoki temperaturi in pritisku nastane plazma. Nabite delce v plazmi se s pomočjo elektromagnetnih tuljav v stenah naprave pospeši in omejuje, dokler ne nastanejo dovolj ugodni pogoji za fuzijsko reakcijo. Zaradi motenj je potrebno konstantno sprotno popravljanje in omejevanje plazme, kar se posebno pri manjših tokamakih izkaže za časovno zahteven problem.

Rešitve za ta problem iščejo v okviru projetka Hitro prediktivno vodenje magnetne plazme v tokamaku (FMPCFMPC, angl. Fast Model Predictive Control for Magnetic Plasma Control) [21]. Koordinator projekta je Odsek za sisteme in vodenje iz Inštituta Jožef Stefan pod vodstvom dr. Sama Gerkšiča. Cilj projekta je na podlagi prediktivnega vodenja razviti hiter regulator za

vodenje toka in oblike plazme.

Izhodišče za diplomsko delo je obstoječ algoritem prediktivnega regulatorja, ki je narejen na osnovi dualne hitre gradientne metode (DHGM, angl. dual fast gradient method). Naš cilj je pohitritev te metode z uporabo paralelnega izvajanja. Odločili smo se za tehnologijo OpenCL. Z ogrodjem OpenCL se lahko tako nekateri deli metode paralelno izvajajo na grafični kartici, izrabi pa se lahko tudi njeno posebno arhitekturo pomnilnikov.

Diplomsko delo je razdeljeno na sedem poglavij. V poglavju 2 bomo predstavili vsebinsko in teoretično ozadje ter namen algoritma prediktivnega regulatorja plazme. V poglavju 3 bomo predstavili obstoječo izvedbo in njene dele. V poglavju 4 bomo opisali ponovno napisano zaporedno implementacijo in njene najpomembnejše prilagoditve. V poglavju 5 bomo opisali ogrodje OpenCL. V poglavju 6 bomo opisali OpenCL izvedbo metode DHGM in njen razvoj. Meritve in rezultate bomo predstavili v poglavju 7. Sledile bodo sklepne ugotovitve v poglavju 8.

Poglavje 2

Ozadje in namen algoritma prediktivnega regulatorja

V tem poglavju predstavimo vsebinsko ozadje algoritma prediktivnega regulatorja. Na kratko opišemo delovanje tokamaka in nekatere probleme, ki so vzrok za omejevanje plazme. S tem spoznamo kakšen je namen in uporaba algoritma pri delovanju tokamaka. Razložimo splošni koncept prediktivnega vodenja, iz katerega lahko razviden problem, ki ga rešuje metoda DHGM.

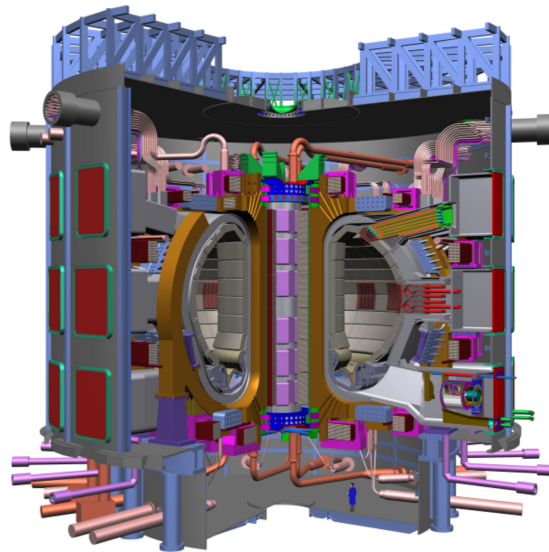
V nadaljevanju na kratko opišemo metodo DHGM in orodje QPGen. V matematično ozadje metode DHGM se ne spuščamo, saj smo jo le uporabili in nam je zato bolj pomembna njena izvedba.

2.1 Tokamak

Tokamak je naprava za magnetno omejevanje plazme v obliki torusa. Trenutno je tokamak eden od načinov, s katerim bi se lahko proizvajalo električno energijo s pomočjo jedrske fuzije. Potrebno je omeniti, da je v času pisanja še v eksperimentalni fazi. Zato se trenutno zgrajeni tokamaki uporabljajo zgolj v raziskovalne namene.

Primer projekta, v okviru katerega se v Franciji gradi velik eksperimentalni reaktor tipa tokamak, je ITER [18]. Namen gradnje reaktorja ITER

je popolnoma testne in eksperimentalne narave – reaktor ne bo proizvajal električne energije. Reaktor ITER naj bi proizvedel 500 MW fuzijske energije iz 24 MW potrebne vhodne energije. To bi bilo veliko več od trenutnega rekorda, ki ga je leta 1997 postavil tokamak JET. Ta je proizvedel 16 MW energije iz 24 MW vhodne energije [5].



Slika 2.1: Model tokamaka.

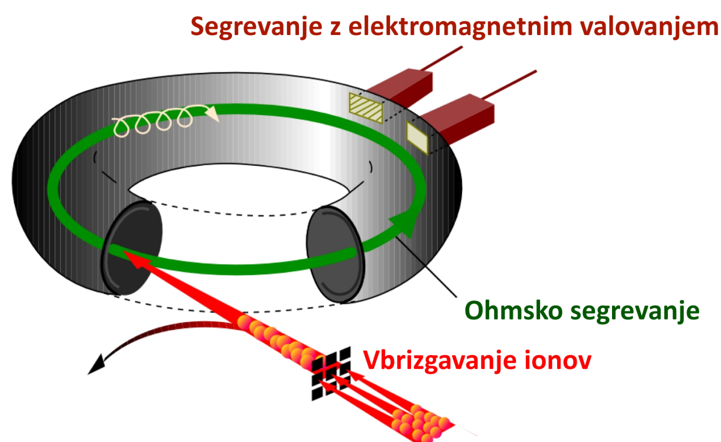
V tokamaku je vakuumska komora v obliki torusa. Vakuum v komori se vzdržuje s pomočjo zunanjih črpalk. V komori se nahaja plazma, katere delci so zelo hitri in vroči. Plazma je sestavljena iz pozitivno nabitih ionov in negativno nabitih elektronov. Da lahko v plazmi nastane fuzija, so potrebni trije pogoji [17]:

- visoka temperatura, ki sproža trk delcev,
- zadostna gostota delcev v plazmi, da se poveča verjetnost trkov, in
- dovolj časa v katerem se omejuje plazmo, ki se zelo nagiba k širjenju.

Za potek fuzije je potrebno doseči približno 150.000.000 °C. Za to je potrebno več različnih metod segrevanja. Pri projektu ITER so zasnovali več notranjih in zunanjih sistemov segrevanja plazme [16].

Prvi sistem segrevanja je notranji. Magnetne tuljave, ki služijo tudi pri omejevanju in pospeševanju plazme, preko indukcije segrevajo plazmo. Takšnemu načinu pravimo ohmsko segrevanje (angl. ohmic heating). Vendar se z višanjem temperature učinek indukcije zmanjšuje in zato so potrebni alternativni sistemi segrevanja.

Dva sistema zunanjega segrevanja plazme sta dodana kot dopolnilo ohmskemu segrevanju. Prvi deluje na osnovi obstreljevanja plazme z energijsko zelo nabitimi delci, ki pri trku s plazmo prenesejo veliko količino energije. Temu se reče vbrizgavanje ionov (angl. neutral beam injection). Vir druge metode zunanjega segrevanja plazme pa je visoko frekvenčno elektromagnetno valovanje.



Slika 2.2: Metode segrevanja plazme. (Povzeto po [6])

2.2 Prediktivni regulator toka in oblike plazme v tokamaku

Algoritem prediktivnega regulatorja služi za omejevanje in spreminjanje oblike plazme v tokamaku. Omejevanje plazme se zlasti pri manjših tokamakih izkaže za zapleten proces. Zelo vročo plazmo se od sten komore vodi in omejuje s pomočjo magnetnega polja. Okoli komore sta postavljena dva sistema močnih magnetnih tuljav, ki lahko vertikalno in horizontalno nadzorujeta magnetno polje v komori. S tem magnetnim poljem se lahko pospešuje in omejuje plazmo in tako skrbi, da je pravilne oblike ter se ne dotika sten komore. Za napajanje teh magnetnih tuljav je navadno potrebno tudi več ločenih napajalnih sistemov.

Zaradi motenj, ki se pojavijo v plazmi, je potrebno konstantno sprotno omejevanje in vodenje oblike plazme. Algoritem prediktivnega regulatorja glede na prejšnja stanja predvidi najboljše napetosti magnetnih tuljav, da se motnje popravijo. Ker se mora algoritem konstantno izvajati v dinamičnemu sistemu tokamaka, je potreben čim krajši čas izvajanja.

2.3 Metoda prediktivnega vodenja

Reševanje problema vodenja plazme v tokamaku, ki je opisan v prejšnjem poglavju, lahko predstavimo tudi s prediktivnim vodenjem (angl. model predictive control).

Prediktivno vodenje izvira iz poznih sedemdesetih let prejšnjega stoletja. Uporablja se v zapletenih dinamičnih sistemih z več spremenljivkami za sprotno računanje kontrolnih rešitev. Danes se precej uporablja v procesni industriji, robotiki in kemični industriji.

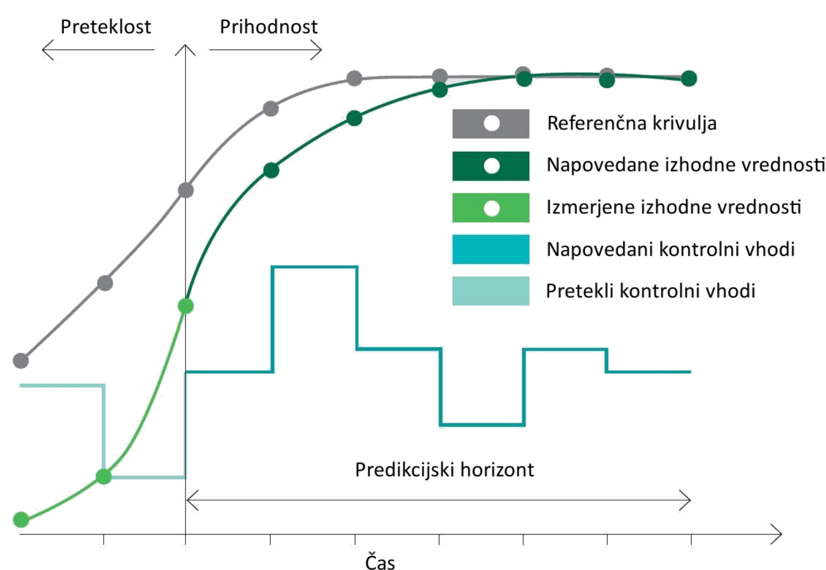
Pod ta termin sodi mnočica metod za vodenje, ki s pomočjo modela ciljnega procesa pridobi signale za vodenje z minimiziranjem kriterijske funkcije. Glavne ideje teh metod prediktivnega vodenja so:

- eksplicitna uporaba modela za napovedovanje izhodnih vrednosti pro-

cesa v določenem trenutku v prihodnosti (skupina teh vrednosti se imenuje predikcijski horizont),

- izračun vrste kontrolnih signalov z minimiziranjem kriterijske funkcije,
- upoštevanje odstopanj dejanske vrednosti kontrolnega signala od izračunane v vsakem trenutku, ko se premaknemo naprej po predikcijskem horizontu.

Različne metode, ki se uporabljajo za prediktivno vodenje, se med sabo razlikujejo v modelu, ki predstavlja proces, in kriterijsko funkcijo z različnimi omejitvami, odvisnimi od danega problema [2].



Slika 2.3: Potek preditktivnega vodenja. (Povzeto po [1])

Na sliki 2.3 lahko vidimo shemo poteka prediktivnega vodenja, ki ga lahko opišemo s tremi koraki [2]:

1. Napovedane izhodne vrednosti so s pomočjo procesnega modela izračunane iz znanih preteklih vrednosti (izmerjenih izhodnih vrednosti in preteklih kontrolnih vhodnih signalov) in napovedanih kontrolnih vhodov.

2. Množica napovedanih kontrolnih vhodov se izračuna s funkcijo, ki je navadno sestavljena iz kvadratov razlik med napovedanimi izhodnimi vrednostmi in referenčno krivuljo. Če so vsebovane tudi omejitve, je potrebno rešitev izračunati v obliki optimizacijskega problema – iterativno računanje maksimalne oziroma minimalne vrednosti.

3. Trenutna napovedana kontrolna vhodna vrednost se uporabi kot rešitev. Pri uporabi te vhodne vrednosti lahko izmerimo novo izhodno vrednost. Z novo znano izhodno vrednostjo se ponovi prvi korak.

2.4 Metoda za reševanje optimizacijskega problema

Konkretno računanje rešitev prediktivnega vodenja je optimiziranje kriterijske funkcije pri upoštevanju omejitev. Tako na primer z minimiziranjem take funkcije iščemo globalni minimum, ki predstavlja želeno najboljšo rešitev našega problema.

V našem primeru je kriterijska funkcija kvadratna. Poteku reševanja take kriterijske funkcije pravimo kvadratično programiranje (angl. quadratic programming).

Kvadratično programiranje je poseben primer nelinearnega programiranja, kjer minimiziramo ali maksimiziramo kvadratno funkcijo več spremenljivk z linearnimi omejitvami. Pogosta uporaba je reševanje optimizacijskih problemov na področju financ, proizvodnje električne energije in inženirskem načrtovanju [19].

Cilj kvadratičnega programiranja je najti rešitev naslednje funkcije z n spremenljivkami in m omejitvami:

$$\begin{aligned} \text{minimiziraj} \quad & \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ \text{pri pogoju} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b}, \end{aligned} \tag{2.1}$$

kjer je

- **Q** simetrična matrika realnih števil velikosti $n \times n$,
- **c** vektor realnih števil velikosti n,
- **A** matrika realnih števil velikosti $m \times n$,
- **b** vektor realnih števil velikosti m in
- **x** iskani vektor realnih števil velikosti n.

Za reševanje takšnega problema je bilo razvitih veliko različnih metod. Med bolj pogostimi so metoda notranjih točk, metoda omejenega koraka in metoda aktivne množice [19].

2.5 Metoda DHGM in orodje QPgen

Pri izbiri algoritma za reševanje zgornje funkcije je zaradi narave problema potrebno paziti, da je čas reševanja čim krajši, hkrati pa morajo biti rešitve še vedno dovolj natančne. Potrebna je tudi konsistentnost in zanesljivost, saj si ne smemo privoščiti, da bi metoda pri določenih vhodnih podatkih porabila preveč časa ali pa preveč odstopala od dejanske rešitve.

Za reševanje problema je bila v okviru projekta FMPCFMPC izbrana metoda DHGM. Metoda DHGM je razširitev metode gradientnega spusta (angl. gradient descent). Metoda gradientnega spusta preko gradienta išče minimum in je v osnovi iterativni algoritem prvega reda.

Metoda DHGM je bila implementirana s pomočjo orodja QPgen [7], ki pa je bil dopolnjen za potrebe projekta FMPCFMPC. QPgen je orodje v programskem paketu Matlab, ki iz podanih parametrov in opisa kvadratnega problema ustvari kodo v programskem jeziku C. Orodje QPgen rešuje zgoraj opisano funkcijo 2.1 z uporabo izbranega algoritma. Izbiramo lahko med:

- metodo izmenične usmeritve multiplikatorjev (angl. alternating direction method of multipliers),

- primarno hitro gradientno metodo (angl. fast dual proximal gradient method) in
- metodo DHGM.

Orodje QPgen deluje tako, da naredi okviren potek algoritma v glavni funkciji za reševanje kvadratnega problema, imenovani `qp`. Funkcija iterativno išče najboljšo rešitev za podane omejitve z izbrano metodo. Za vsako računsko vrstico tega poteka pokliče pomožno funkcijo, ki navadno izračuna matrično operacijo. Vsaka vrstica, ki zahteva množenje matrike z vektorjem, na primer pokliče pomožno funkcijo za računanje matričnega množenja. Tako nastane programska koda lahko berljive funkcije `qp` in njenih pomožnih funkcij, ki se lahko uredijo v orodju QPgen.

Poglavje 3

Obstoječa izvedba

V okviru projekta FMPCFMPC je bila implementirana izvedba algoritma prediktivnega regulatorja plazme na podlagi metode DHGM.

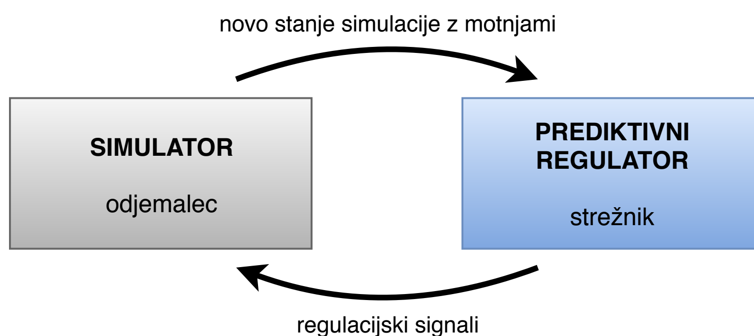
Programska koda obstoječe izvedbe je narejena z orodjem QPgen, opisanem v poglavju 2.5. Za ustvarjanje te kode je potrebno imeti nameščen paket Matlab in ogrodje QPgen, predelano za projekt FMPCFMPC [21], ter vnaprej pripravljene parametre in podatke o samem problemu. Vsi podatki in parametri so zapisani v podatkovnih strukturah paketa Matlab.

Obstoječi poskusi pohitritve so prav tako vključeni v samo orodje QPgen, ki ustvarja želene nadgradnje v programsko kodo jezika C. Ustvarjanje različic programske kode teh pohitritev je možno s spreminjanjem zgoraj omenjenih parametrov v paketu Matlab.

3.1 Struktura projekta

Obstoječi projekt je v osnovi sestavljen iz prediktivnega regulatorja, ki predstavlja strežniški del, in simulatorja, ki predstavlja vlogo odjemalca. Simulator toka in oblike plazme pošilja podatke o novem stanju z možnimi motnjami. Prediktivni regulator podatke sprejme in pošlje nazaj regulacijske signale za vodenje plazme. Iz teh signalov lahko simulator pripravi novo stanje. Celotna struktura je zasnovana tako, da omogoča nastavljivo število

izmenjav omenjenih podatkov.



Slika 3.1: Izmenjava podatkov simulatorja in obstoječega regulatorja.

3.1.1 Simulator

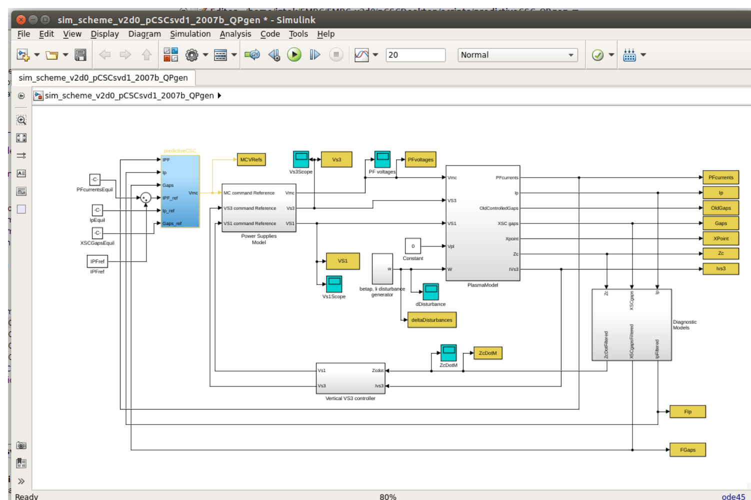
Simulator je program, napisan v okolju Simulink, ki simulira proces premikanja plazme. Njegov namen je preverjanje pravilnosti regulacijskih signalov prediktivnega regulatorja.

Zasnovan je tako, da na začetku iz ravnovesnega stanja preide v stanje z določeno motnjo. Tekom delovanja sistema, prikazanega na sliki 3.1, lahko opazujemo odziv regulatorja na motnjo ter čas vzpostavitve ravnovesnega stanja. S tem lahko ugotavljamo pravilnost in učinkovitost regulatorja glede na simulirane motnje. Podrobnejši opis delovanja simulacijske sheme in odzivi regulatorja so na voljo v [12].

Simulator pošilja in sprejema podatke preko protokola TCP, kar omogoča izvajanje simulatorja na ločenem sistemu. To je pomembno predvsem za natančno merjenje časa izvajanja algoritma regulatorja, saj je na izoliranem sistemu manj motenj zaradi prekinitev.

Za zagon simulacije je potrebno okolje Simulink, v katerem je bil simulator tudi razvit. Simulink je okolje za večdomenske simulacije in razvoj na podlagi modela (angl. model-based design). Vmesnik Simulink vsebuje grafični urejevalnik, s katerim podpira razvoj in modeliranje na podlagi diagramov in blokov. Integriran je v paket Matlab. Podpira uporabo algoritmov, ki so na-

pisani v paketu Matlab, in izvoz rezultatov simulacije, potrebnih za analizo, v orodje Matlab [20].



Slika 3.2: Shema simulatorja plazme v okolju Simulink.

3.1.2 Pregled obstoječega regulatorja

Program obstoječega prediktivnega regulatorja je sestavljen iz datotek z izvorno kodo programskega jezika C, podatkovnih datotek in manjših skript. Najpomembnejše so štiri datoteke z izvorno kodo, ki jih ustvari prilagojeno orodje QPgen:

- `server.c`,
- `QPgen.c`,
- `init_data.c` in
- `init_work_space.c`.

Kljub temu da je programska koda ustvarjena preko orodja QPgen, je koda berljiva in prostorsko dokaj optimizirana. To je posledica omenjenih obstoječih prilagoditev kode v QPgenu.

Glavne značilnosti zgoraj omenjenih datotek z izvorno kodo so naslednje:

1. Namen glavne datoteke `server.c` je, da pokliče funkcije, ki inicializirajo začetne statične podatke, in nato naredi preprost strežnik, ki v iteracijah najprej sprejme vhodne podatke, jih preko argumenta pošlje globalni funkciji `qp` za reševanje kvadratnega problema z metodo DHGM. Rešitve nato pošlje nazaj simulatorju.

   Strežnik posluša na vratih protokola TCP s funkcijami privzete Linux knjižnice `sys/socket`. Za dodeljevanje vtičnice (angl. socket) je uporabljena funkcija `bind`, za poslušanje povezav na vtičnici `listen`, za sprejemanje podatkov `recv` in za posredovanje podatkov nazaj v simulator je uporabljena funkcija `send`.

   Po sprejemu vhodnih podatkov in pred pošiljanjem rešitev simulatorju, se meri čas izvajanja glavne računske funkcije. Merjenje časa je implementirano s funkcijo `clock_gettime` iz standardne Linux knjižnice `time`.

2. V izvorni datoteki `QPgen.c` se nahaja funkcija `qp`, ki je zadolžena za računski del regulatorja. Funkcija `qp` se za vsake prejete vhodne podatke pokliče iz zgoraj omenjenega strežniškega ogrodja. Poleg funkcije `qp` so v tej datoteki zapisane tudi pomožne funkcije. Te pomožne funkcije so tipa `static`, ker so narejene samo za računske operacije znotraj funkcije `qp`. Klic pomožne funkcije je prikazan v izseku programske kode na sliki 3.3.

   Parametri funkcije `qp` so kazalci na statične, na vhodne in na izhodne podatke, ki jih funkcija potrebuje za izvajanje. Statične matrike in vektorji so označeni s spremenljivko `d`. Delovno okolje, ki služi shranjevanju vmesnih rezultatov, je označeno s spremenljivko `ws`. Število notranjih iteracij, ki jih naredi funkcija `qp`, predstavlja spremenljivka `j`. Kazalca `in_ref` in `in_x` kažeta na medpomnilnik vektorjev vhodnih podatkov, kazalec `out_sol` pa na izhodni.

3. V datoteki `init_data.c` je del programske kode, ki iz ločenih binarnih datotek prebere statične matrike in vektorje ter jih zapiše v strukturo

```
// Pomožna funkcija za matrično množenje
static void mat_vec_mult_full(struct FULL_MAT *M, double *x,
        double *y) { ... }

// Glavna računska funkcija
void qp(struct DATA *d, struct WORK_SPACE *ws, double *out_sol,
        int *iter, double *in_ref, double *in_x)
{
        mat_vec_mult_full(d->L2,in_x,ws->tmp_var_p2);
        ...
}
```

Slika 3.3: Izsek programske kode, ki prikazuje uporabo pomožne funkcije.

tipa `DATA`. Prebrani podatki se tekom programa ne spreminjajo in zgolj opisujejo omejitve in lastnosti modela problema. Prilagoditi bi jih morali v primeru sprememb v velikosti ali obliki ciljnega tokamaka. Zato se tej nespremenjeni statični podatki preberejo na začetku programa in se nato ponovno uporabljajo pri vsakem nadaljnjem izračunu vhodnih podatkov iz simulatorja.

4. Funkcije v datoteki `init_work_space.c` inicializirajo strukturo delovnega okolja. V to strukturo poimenovano `WORK_SPACE` spadajo spremenljivke, ki se uporabljajo kot vmesni rezultati v funkciji `qp`. To so podatki, ki se tekom programa spreminjajo.

3.2 Funkcija qp

Funkcija `qp` rešuje kvadratni problem na podlagi metode DHGM. Predstavlja najpomembnejši del regulatorja. Njen potek lahko okvirno razdelimo na tri dele: **začetni del**, **zanka** in **končni del**. Časovno najzahtevnejši je osrednji del – potek v zanki, ki predstavlja iterativno iskanje najboljše rešitve na

podlagi parametrov. Število iteracij je v našem primeru vnaprej določeno in je tesno povezano s kvaliteto rešitve. Med testiranjem smo število iteracij nastavljali na vrednosti od 500 do 10.000.

V času izvajanja funkcije `qp` se poleg vhodnih podatkov pogosto uporabljajo tudi statične matrike in vektorji, ki opisujejo model in omejitve. Zapisani so v strukturi `DATA` v formatu plavajoče vejice z dvojno natančnostjo – tipa `double` v jeziku C. Velikosti in mesta branja so predstavljeni v tabeli 3.1.

ime vektorja/matrike	velikost	del funkcije, kjer se bere
q1	33×1	začetni
Q2	33×81	začetni
cstr_smpls	$N \times 1$	začetni
ymin, **ymax**	$M \times 1$	začetni
umin, **umax**	$P \times 1$	začetni
l1	99×1	začetni
L2	99×81	začetni
R2	33×81	začetni
M	33×99	zanka
C	99×33	zanka
soft	99×1	zanka
E	99×99 (diag.)	začetni, zanka
qwsoft	99×1	zanka
R	33×33	končni

Tabela 3.1: Tabela dimenzij in mesta uporabe v funkciji `qp` statičnih vektorjev in matrik. Matrika **E** je diagonalna in je zapisana v obliki vektorja.

Velikosti vektorjev **cstr_smpls**, **ymin**, **ymax**, **umin** in **umax** so glede na simulacijo lahko različne. Definirane so s parametri, ki so, podobno kot statični podatki, zapisani v strukturi `DATA`. Velikosti teh vektorjev se med samo simulacijo ne spreminjajo. Našteti vektorji, skupaj z referenčnim vek-

torjem **in_ref**, prilagajajo zgornje in spodnje omejitve optimizacijskega problema.

N	Vrstica algoritma	Velikosti matrik
Začetni del:		
1	q1 = *q1*	[33] = [33]
2	q2 = *Q2* * in_x	[33] = [33×81] x [81]
3	q = q1 + q2	[33] = [33] + [33]
4	ref_adjust(*lb*, *ub*)	[99]
5	l = diag(E) * *lb*	[99] = [99] .* [99]
6	u = diag(E) * *ub*	[99] = [99] .* [99]
7	tmp_var_p = *l1*	[99] = [99]
8	tmp_var_p2 = *L2* * in_x	[99] = [99×81] x [81]
9	shift_arg = tmp_var_p + tmp_var_p2	[99] = [99] + [99]
10	eshift_arg = diag(*E*) * shift_arg	[99] = [99] .* [99]
11	tmp_var_n2_o = *R2* * in_x	[33] = [33×81] x [81]
12	r = tmp_var_n_o + tmp_var_n2_o	[33] = [33] + [33]
Zanka:		
13	tmp_var_n = *M* * v	[33] = [33×99] x [99]
14	x = tmp_var_n + q	[33] = [33] + [33]
15	tmp_var_p = *C* * x	[99] = [99×33] x [33]
16	tmp_var_p = v + tmp_var_p	[99] = [99] + [99]
17	arg_prox_h = tmp_var_p	[99] = [99]
18	tmp_var_p = eshift_arg + tmp_var_p	[99] = [99] + [99]
19	clip_soft(tmp_var_p)	[99]
20	y = diag(*Einv*) * tmp_var_p	[99] = [99] .* [99]
21	y = y – shift_arg	[99] = [99] - [99]
22	tmp_var_p = diag(*E*) * y	[99] = [99] .* [99]
23	lambda_old = lambda	[99] = [99]
24	lambda = arg_prox – tmp_var_p	[99] = [99] - [99]
25	tmp_var_p = lambda – lambda_old	[99] = [99] - [99]
26	tmp_var_p = theta[k] * tmp_var_p	[99] = [1] * [99]
27	v_old = v	[99] = [99]
28	v = tmp_var_p + lambda	[99] = [99] + [99]
29	restart(lambda, lambda_old, v, v_old)	[99]
Končni del:		
30	tmp_var_n_o = *R* * x	[33] = [33×33] x [33]
31	out_sol = tmp_var_n_o + r	[33] = [33] + [33]

Slika 3.4: Obstoječa funkcija qp. Imena statičnih matrik in vektorjev iz strukture DATA so označena s poševnim tiskom.

Vektor **in_ref**, ki je velikosti 21×1, je eden izmed dveh vhodnih vektorjev. Drugi vhodni vektor je poimenovan **in_x** in je velikosti 81×1. Vhodna vektorja se v funkcijo `qp` pošiljata preko parametrov.

Obstoječa funkcija `qp` je sestavljena tako, da se pri vsaki matrični ali vektorski računski operaciji pokliče pomožna funkcija. Klic funkcije matričnega množenja statične matrike **Q2** z vhodnim vektorjem **in_x** v začasno spremenljivko **Q2** je na primer `mat_vec_mult_full(d->Q2, in_x, ws->Q2)`.

Na sliki 3.4 lahko vidimo nekatere poimenovane funkcije, ki predstavljajo bolj zapletene operacije. Tudi te, poleg zgoraj opisanih matričnih in vektorskih operacij, spadajo med pomožne funkcije.

1. Funkcija `ref_adjust` prilagaja dva vektorja, ki vsebujeta zgornje in spodnje omejitve optimizacijskega problema. To naredi glede na vhodni referenčni vektor. Za ta potek je potrebno primerjanje (vejitev `if`) in morebitno odštevanje ali kopiranje elementov vektorja. Opisana operacija se izvede samo enkrat v začetnem delu, v 4. vrstici algoritma na sliki 3.4.

2. Funkcija `clip_soft` vsak element danega vektorja primerja in prilagodi z vektorjema omejitev. Ta operacija se uporablja v zanki, kar je prikazano v 19. vrstici na sliki 3.4.

3. Funkcija `restart` v določenih primerih ponastavi dva vektorja na vrednost iz prejšnje iteracije. Za to operacijo potrebuje štiri različne vektorje, s katerimi naredi dve vektorski odštevanji, skalarni produkt, primerjavo vrednosti in dva morebitna kopiranja vektorjev. Tudi ta operacija se uporablja v delu zanke, kar je vidno v 29. vrstici na sliki 3.4.

3.3 Obstoječe prilagoditve

Zaradi potrebe po hitrejšem času izvajanja algoritma, so bile narejene in testirane nekatere prilagoditve obstoječe programske kode.

Za časovno optimizacijo sta najzanimivejši izvedba z vmesnikom OpenMP [3] in izvedba z uporabo Intelove knjižnice MKL [14]. Pri obeh izvedbah se vse optimizacijske spremembe nahajajo v pomožnih funkcijah, ki so zadolžene za posamezne vektorske in matrične operacije. Računske operacije nekaterih pomožnih funkcij se tako lahko izvajajo paralelno.

Izvedba z vmesnikom OpenMP je v večini primerov počasnejša od obstoječe zaporedne izvedbe. Razlog za to je, da se deljenje dela zgodi približno enkrat v vsaki pomožni funkciji. Ker je klicev pomožnih funkcij zelo veliko, se ogromno časa izgubi za organizacijo dela in sinhronizacijo niti.

Izvedba s knjižnico MKL dosega krajši čas izvajanja od zaporedne izvedbe, vendar pa je čas izvajanja še vedno predolg. V večini pogosto uporabljenih pomožnih funkcij se kličejo zelo dobro optimizirane matrične funkcije knjižnice MKL. Na večjedrnih procesnih enotah se nekatere funkcije lahko izvajajo tudi paralelno, odvisno od velikosti problema. Tudi ta izvedba na nekaterih mestih, kjer se to izplača, uporablja paralelizacijo z vmesnikom OpenMP.

Poglavje 4

Prilagojena zaporedna implementacija

Na začetku projekta smo na podlagi obstoječe izvedbe ponovno napisali programsko kodo prediktivnega regulatorja. To nam je pozneje pomagalo pri paralelizaciji regulatorja, prenosljivosti na druge operacijske sisteme ter izvajanju vnaprej pripravljenih testov pravilnosti.

Pripravili smo si tudi ogrodje s skripto za preverjanje pravilnosti in natančnosti rezultatov ter analiziranje časa izvajanja. Tak sistem smo kasneje lahko uporabili tudi za paralelne različice izvedb.

4.1 Razvojno okolje

Programsko kodo smo v začetku prilagajali v operacijskem sistemu Linux, za katerega je napisana tudi obstoječa izvedba. Kodo smo urejali v odprtokodnem urejevalniku *Visual Studio Code*, ki ga je razvil Microsoft. Kasneje smo nadaljevali razvoj v operacijskem sistemu Microsoft Windows v urejevalniku *Microsoft Visual Studio 15*.

4.2 Novo ogrodje in struktura projekta

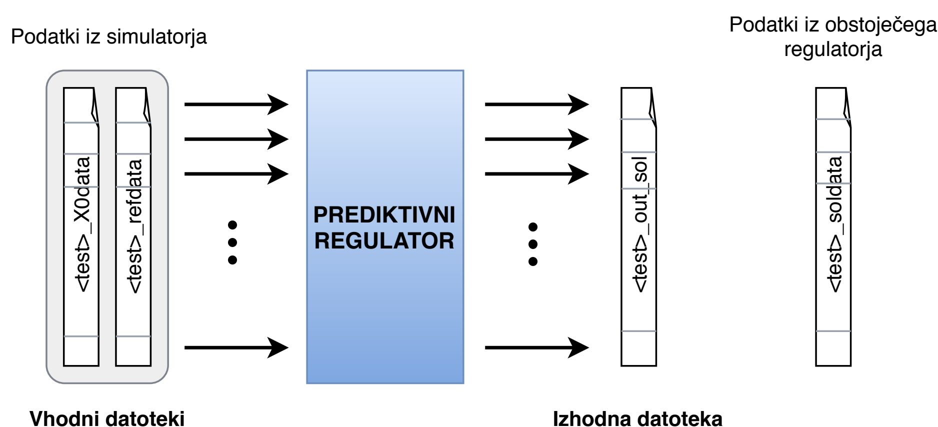
Struktura projekta prilagojene izvedbe je iz oblike strežnika in odjemalca poenostavljena v en sam samostojen program. Za to smo morali ponovno napisati programsko kodo ogrodja. Imena statičnih matrik in vektorjev ter načina klicanja glavne funkcije `qp` niso spremenjena zaradi morebitne kasnejše integracije funkcije `qp` v obstoječo strukturo.

Pri novi zaporedni in kasneje tudi pri paralelni izvedbi nas je najbolj zanimalo, ali novo nastale implementacije vračajo enako rešitev kot obstoječa izvedba. V ta namen smo si pripravili ogrodje, ki testira funkcijo `qp` na podlagi vnaprej pripravljenih testnih primerov.

Testne primere smo pridobili tako, da smo za vsako izmenjavo podatkov (slika 3.1) obstoječega simulatorja in regulatorja posneli vhodno-izhodne podatke. Vhodne podatke iz simulatorja smo shranili v dve ločeni datoteki; v prvo smo shranili vhodne vektorje **in_x**, v drugo pa vhodne vektorje **in_ref**. Tudi izhodne podatke obstoječe izvedbe smo shranili v ločeno datoteko, katero kasneje primerjamo z izhodnimi datotekami novih izvedb. Vhodni in izhodni vektorji iz vsake iteracije simulatorja in regulatorja so shranjeni v ločeni vrstici datotek.

Za vsak testni primer, ki predstavlja različen profil motnje, smo posneli po 300 izmenjav. Testni primeri zajemajo štiri tipe motenj (MD, ELM, H-L/L-H, VDE) v treh različnih ravnovesnih modelih (t80, t90, t520) [12]. Vseh testnih primerov je 12.

Program prilagojenega regulatorja (slika 4.1) se lahko izvaja neodvisno od simulatorja. Vhodni podatki so vedno enaki, kar olajša tudi preverjanje pravilnosti programov pri razvoju drugih izvedb. Program vsebuje tudi manjše število vrstic bolj kompaktne programske kode v primerjavi z obstoječo izvedbo. Pazili smo, da je programska koda prenosljiva iz okolja Linux v okolje Windows.



Slika 4.1: Tok podatkov v projektu.

4.3 Preverjanje pravilnosti in natančnosti

Rezultati prilagojene izvedbe se shranjujejo v izhodne datoteke, poimenovane `<ime testnega primera>_out_sol.dat`, za vsak testni primer ločeno. V vsaki vrstici te izhodne datoteke prvih 33 vrednosti predstavlja izhodni vektor **out_sol**, zadnja vrednost pa čas izvajanja. Enako so sestavljene tudi datoteke `<ime testnega primera>_soldata.dat`, v katere smo shranili rešitve obstoječe izvedbe, narejene z orodjem QPgen.

Prvih enajst vrednosti izhodnega vektorja **out_sol** predstavlja kontrolne signale za trenutni vzorec. Nato sledita še dve skupini po enajst predvidenih vrednosti za kontrolne signale naslednjih vzorcev. Poznavanje sestave izhodnih datotek je pomembno pri računanju relativne napake.

Algoritem regulatorja je v osnovi iterativno iskanje najboljše rešitve. Tako so lahko rezultati pri drugi izvedbi regulatorja z enakim potekom različni. Vzrok so ponavadi numerične napake in s tem povezano malo drugačno konvergiranje proti rešitvi. Zaradi tega je potrebno preverjanje pravilnosti programa s pomočjo relativne napake.

Relativno napako RE smo računali po enačbi

$$RE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\frac{u_i - u_i'}{r_i})^2}, \tag{4.1}$$

kjer je n število kontrolnih signalov v izhodnem vektorju (11 za trenutni vzorec, 33 za vse), u so vrednosti shranjenih testnih primerov obstoječe izvedbe, u' so vrednosti izračunov prilagojene izvedbe in r so območja (angl. range) signalov. Območje vsakega izmed signalov poznamo vnaprej.

Za hitrejše računanje RE smo napisali skripto v orodju Octave, ki računa relativno napako za vse izhodne vektorje trenutne izvedbe glede na obstoječo. Vrednosti smo nato lahko obdelali (na primer izračunali povprečje in standardni odklon) in tako ugotavljali pravilnost in natančnost želene izvedbe. Podobno smo lahko obdelali čas izvajanja klicev funkcije `qp` in pri tem opazovali morebitne pohitritve.

4.4 Funkcija qp

Poleg ogrodja, branja podatkov in nekaterih podatkovnih tipov smo ponovno napisali tudi funkcijo `qp`. Zaradi morebitnega poznejšega prenosa programske kode smo pazili, da so imena funkcij, parametrov in podatkov podobna kot v obstoječi izvedbi.

Pri podrobni analizi algoritma smo ugotovili, da je pri obstoječi izvedbi uporabljenih precej nepotrebnih vmesnih spremenljivk. Razlog za to je uporaba pomožnih funkcij za posamezne računske operacije. Nekatere vmesne spremenljivke v novi funkciji `qp` (slika 4.2) so vseeno potrebne, ker se ponovno uporabljajo v kasnejših delih funkcije. Na novo zapisana funkcija `qp` ima v primerjavi z obstoječo rešitvijo (slika 3.4) 12 vrstic kode manj.

N	Vrstica algoritma	Velikosti matrik
Začetni del:		
1	ref_adjust(lb, ub)	[99]
2	l = diag(E) * lb	[99] = [99] .* [99]
3	u = diag(E) * ub	[99] = [99] .* [99]
4	shift_arg = L1 + (L2 * in_x)	[99] = [99] + ([99×81] × [81])
5	eshift_arg = diag(E) * shift_arg	[99] = [99] .* [99]
6	q = (Q2 * in_x) + q1	[33] = ([33×81] × [81]) + [33]
7	r = (R2 * in_x)	[33] = [33×81] × [81]
Zanka:		
8	x = (M * v) + q	[33] = ([33×99] × [99]) + [33]
9	h = v + (C * x)	[99] = [99] + ([99×33] × [33])
10	p = h + eshift_arg	[99] = [99] + [99]
11	clip_soft(p)	[99]
12	p = diag(E) * ((diag(Einv)*p)-shift_arg)	[99] = [99].*(([99].*[99])-[99])
13	lambda_old = lambda	[99] = [99]
14	lambda = h - p	[99] = [99] - [99]
15	p = (lambda - lambda_old) * theta[k]	[99] = ([99] - [99]) * [1]
16	v_old = v	[99] = [99]
17	v = tmp_var_p + lambda	[99] = [99] + [99]
18	restart(lambda, lambda_old, v, v_old)	[99]
Končni del:		
19	out_sol = (R * x) + r	[33] = ([33×33] × [33]) + [33]

Slika 4.2: Prilagojena funkcija qp. Imena statičnih matrik in vektorjev iz strukture DATA so označena s poševnim tiskom.

Poglavje 5

Ogrodje OpenCL

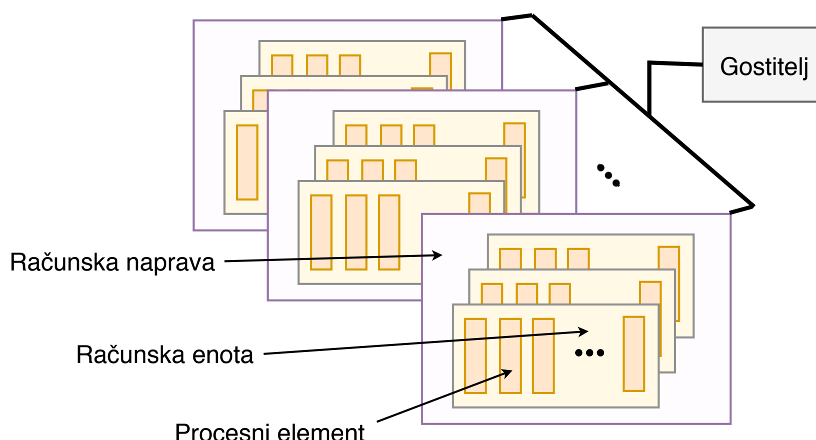
OpenCL je odprt brezplačni standard za splošno namensko paralelno programiranje na heterogenih sistemih, sestavljenih iz CPE, GPE in drugih procesnih enot [9]. Za standard OpenCL skrbi konzorcij Khronos Group [8]. Ogrodje OpenCL razvijalcem programske opreme nudi možnost prenosljivega in učinkovitega dostopa do računske moči takih heterogenih sistemov. Vključuje svoj programski jezik, programski vmesnik (angl. application programming interface, API), knjižnico in sistem izvajanja za razvoj programov [11].

Pogosto se uporablja za splošno namensko računanje na grafičnih procesnih enotah. Z ogrodjem OpenCL se lahko doseže visoko stopnjo paralelizma brez spuščanja v nizkonivojske rutine.

5.1 Arhitektura in abstraktni prikaz strojne opreme

Program napisan v jeziku OpenCL se lahko paralelno izvaja na strojni opremi z različnimi tipi procesnih enot različnih proizvajalcev. Ogrodje OpenCL nudi visokonivojsko abstrakcijo strojne opreme. To na primer razvijalcu omogoča pisanje splošne programske kode, ki je enaka za paralelno izvajanje na CPE in za paralelno izvajanje na GPE.

Abstraktni prikaz strojne opreme (platformni model) je prikazan na sliki 5.1. Platformni model ogrodja OpenCL je zgrajen iz enega gostitelja, na katerega je priključenih več računskih naprav, na primer koprocesor ali GPE. Računske naprave so zgrajene iz računskih enot, te pa so sestavljene iz procesnih elementov.



Slika 5.1: Abstrakcija strojne opreme pri ogrodju OpenCL. (Povzeto po [11])

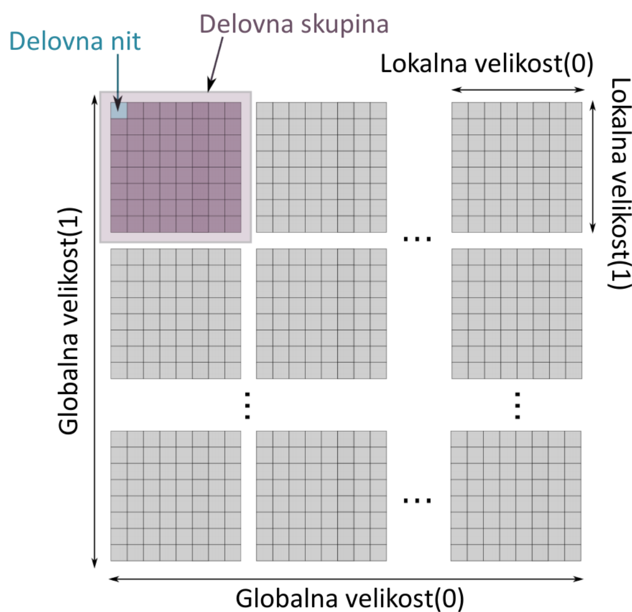
5.2 Model izvajanja

Izvajanje OpenCL programa poteka v dveh ločenih izvajalnih enotah: v ščepcu (angl. kernel), ki se izvaja na računskih napravah, in programu gostitelja, ki se izvaja na napravi gostitelja. Delo ščepca je organizirano v strukture imenovane delovne niti (angl. work-item). Delovne niti pa so skupaj povezane v delovne skupine (angl. work-group). Vse delovne niti iz delovne skupine se izvajajo na procesnih elementih v okviru iste računske enote.

Ščepec se izvaja znotraj konteksta. Kontekst opisuje okolje izvajanja ščepca in je narejen v delu programa, ki ga izvaja gostitelj. Gostiteljski program naredi in upravlja kontekst s pomočjo programskega vmesnika ogrodja OpenCL. Funkcije omenjenega programskega vmesnika omogočajo komunikacijo z računsko napravo prek ukazne vrste (angl. command-queue). Vsaka

ukazna vrsta je povezana s točno eno računsko napravo. Poleg programa gostitelja lahko tudi ščepec uporablja ukazne vrste in tako potencialno kliče nov ščepec.

Ukazi se lahko v ukazni vrsti izvajajo urejeno, s čimer je zagotovljen vrstni red izvajanja, ali neurejeno, kjer je vrstni red izvajanja ukazov odvisen od prostih virov na napravi.



Slika 5.2: Organizacija dela v 2-dimenzionalnem razponu. (Povzeto po [4]).

Organizacija dela v delovnih nitih je izvedena z N-dimenzionalnim razponom (angl. NDRange), kjer je število dimenzij N lahko ena, dva ali tri. N-dimenzionalni razpon je razdeljen v delovne skupine, ki morajo zajemati vse delovne niti. Definiran je z naslednjimi vrednostmi:

- Število vseh delovnih niti, s katerimi se bo izvajal dani ščepec, imenovano globalna velikost (angl. global size), podano za vsako dimenzijo posebej.

- Odmiki, ki bodo enaki vrednostim prvih indeksov v vsaki dimenziji.

- Število delovnih niti v eni delovni skupini, ali lokalno velikost (angl. local size), spet za vsako dimenzijo posebej.

Vsako delovno nit se lahko naslovi z indeksi globalnega prostora in z indeksi lokalnega prostora. Primer organizacije delovnih niti je prikazan na sliki 5.2.

5.3 Pomnilniška struktura

Časovna optimizacija programa je pogosto povezana s pomnilniškimi dostopi. Strukturo OpenCL pomnilniškega modela lahko razdelimo na dva dela: **pomnilnik gostitelja** in **pomnilnik naprave**. Podatke med njima prenašamo preko ukazov programskega vmesnika ali pa uporabe navideznega skupnega pomnilnika.

Pomnilnik naprave je sestavljen iz naslednjih naslovljivih pomnilniških delov [11]:

- **Globalni pomnilnik** (angl. global memory) je za branje in pisanje dostopen vsem delovnim nitim znotraj vseh delovnih skupin. Dostopi so lahko predpomnjeni, odvisno od zmožnosti naprave.

- **Pomnilnik konstant** (angl. constant memory) je del globalnega pomnilnika in je na namenjen nespreminjajočim podatkom.

- **Lokalni pomnilnik** (angl. local memory) je pomnilnik delovne skupine. Lahko ga uporabljajo vse lokalne niti znotraj te delovne skupine.

- **Privatni pomnilnik** (angl. private memory) je na voljo samo trenutni delovni niti. Ostalim nitim ni viden. Navadno je ta pomnilnik najhitrejši.

Slika 5.3: Prikaz OpenCL pomnilniške strukture. (Povzeto po [11])

5.4 Programski jezik

Programska jezika, s katerima se lahko napiše ščepec, sta OpenCL C in OpenCL C++. Programski jezik OpenCL C je zasnovan na podlagi standarda ISO/IEC 9899:1999 C oziroma specifikacije C99. V primerjavi z omenjeno specifikacijo ima jezik OpenCL C 2.0 naslednje pomembnejše omejitve [10]:

- omejena raba kazalcev,
- tabele in strukture s spremenljivo ali nedefinirano velikostjo niso podprte,
- rekurzija ni podprta,
- implicitna deklaracija funkcije ni dovoljena.

Jezik je razširjen z vgrajenimi funkcijami za upravljanje z delovnimi niti in skupinami, pomnilnikom, sinhronizacijo, ipd. Primer takšne funkcije za določitev delovne niti znotraj skupine glede na dimenzijo je `get_local_id(i)`. Vgrajeni so tudi posebni vektorski in skalarni podatkovni tipi. Pomembni so tudi posebni kazalci `__global`, `__local`, `__constant` in `__private`, ki označujejo v katerem delu pomnilnika se objekt ustvari.

Poglavje 6

Izvedba metode DHGM z OpenCL

Paralelno izvedbo z ogrodjem OpenCL smo razvijali postopno in sproti spremljali pravilnost delovanja algoritma s skripto, ki je opisana v poglavju 4.3. Ukvarjali smo se predvsem s prilagajanjem programske kode in potekom funkcije `qp`, saj ta predstavlja računsko najintenzivnejši del programa. Del programa regulatorja, ki vsebuje ogrodje za merjenje časa, branje, pripravo in pisanje podatkov, je enak kakor pri prilagojeni zaporedni izvedbi.

6.1 Paralelizacija pomožnih funkcij

Paralelizacije smo se lotili podobno, kot so naredili optimizacije z orodjem QPgen. Osnovna funkcija `qp` pri izvedbi s paralelizacijo pomožnih funkcij je podobna kot pri obstoječi izvedbi. To pomeni, da v vsaki vrstici kliče pomožne funkcije. Naša osnovna ideja je bila spreminjanje omenjenih pomožnih funkcij v ščepce. Tako spreminjanje programske kode omogoča postopno in ločeno paralelizacijo različnih vektorskih operacij, ki so v pomožnih funkcijah.

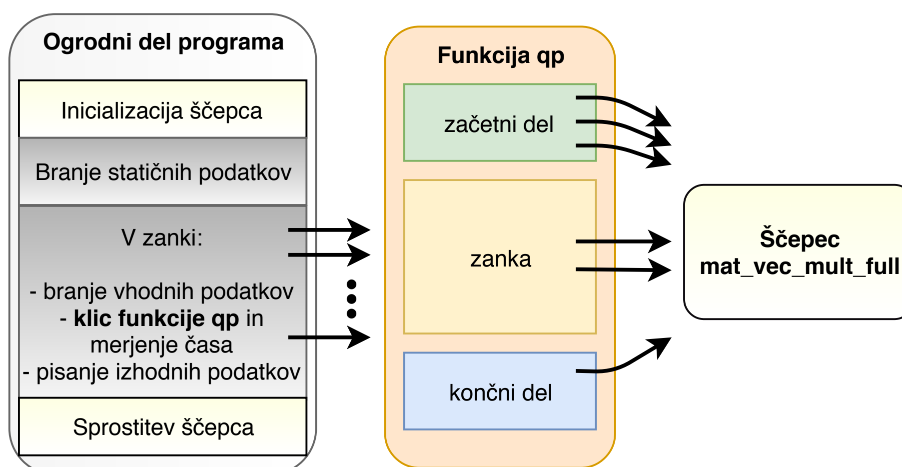
Začeli smo s pomožno funkcijo za množenje matrike z vektorjem, saj se v metodi precej uporablja in je računsko dokaj zahtevna. Napisali smo ščepec

mat_vec_mult_full, ki iz matrike in vektorja paralelno izračuna rešitev.

Na začetek programa, v ogrodni del, smo dodali inicializacijo ščepca. Med inicializacijo spadajo naslednji koraki:

- izbira platforme,
- izbira naprave tipa GPE na platformi,
- definiranje konteksta in ukazne vrste za napravo,
- alokacija pomnilnika na napravi,
- priprava OpenCL programa iz ločene datoteke in njegovo prevajanje,
- priprava objekta ščepca mat_vec_mult_full in definiranje njegovih argumentov.

Inicializacija ščepca se izvede samo enkrat tekom izvajanja programa. Tako se isti ščepec pokliče za vsako množenje matrike z vektorjem. Pred vsakim klicem ščepca se morata matrika in vektor prenesti preko ukazne vrste v pomnilnik GPE, nato pa se mora še rezultat prenesti iz pomnilnika GPE nazaj v pomnilnik gostitelja. Inicializacija in klici ščepca mat_vec_mult_full so prikazani na sliki 6.1.



Slika 6.1: Izvajanje ščepca mat_vec_mult_full.

Množenje matrike **A** velikosti $n \times m$ z vektorjem **x** velikosti m lahko ponazorimo z naslednjo enačbo.

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & ... & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{6.1}$$

Ščepec je narejen tako, da vsaka delovna nit računa svojo vrednost vektorja **y**. Prva delovna nit na primer računa y_1, druga pa y_2. Vsaka nit bere vektor **x** in svojo vrstico matrike **A**. Če je število vrstic matrike **A** večje od števila delovnih niti *global_size*, vsaka nit dodatno računa še tiste vrednosti vektorja **y**, katere indeksi so enaki seštevku globalnega indeksa te niti in večkratnikov števila *global_size*. Če bi bil na primer *global_size* enak 32, bi prva delovna nit računala vrednosti y_1, y_{33}, y_{65}, y_{97}, itd.

Opazili smo, da spreminjanje pomožnih funkcij v ščepce privede do zelo počasnega izvajanja celotnega programa. Ugotovili smo, da se taka paralelizacija ne izplača, saj mora program pri vsakem klicu ščepca čakati, da se podatki prenesejo v pomnilnik GPE in nazaj. Število klicev zelo narašča s številom notranjih zank funkcije `qp`. Število klicev ščepca `mat_vec_mult_full`, ki se iz zanke pokliče dvakrat, pri 500 iteracijah naraste na 1.004. Paralelizacijo pomožnih funkcij smo zato opustili, razvoja smo se znova lotili od začetka.

6.2 Premik celotnega poteka v en ščepec

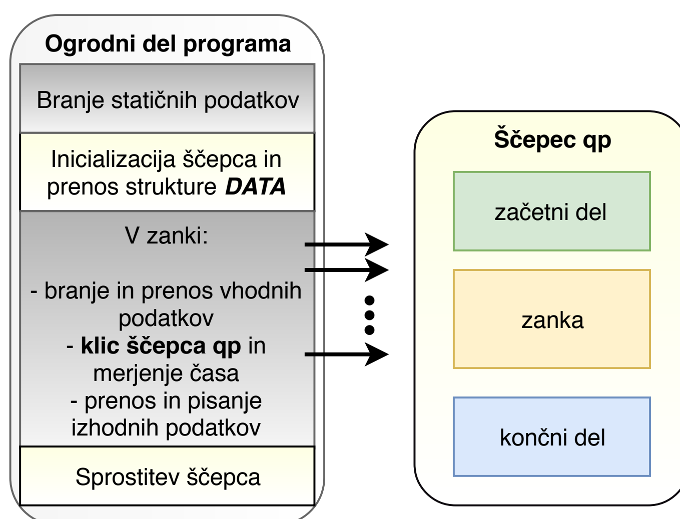
Tokrat smo izhajali iz poteka algoritma z odstranjenimi vmesnimi spremenljivkami (slika 4.2). Cilj je bil metodo DHGM v celoti implementirati v enem ščepcu. Tako se izognemo velikemu številu klicev ščepcev in prenosov podatkov, ki so problematični pri izvedbi s paralelizacijo pomožnih funkcij, opisani v poglavju 6.1.

Oblika statičnih podatkov, ki se uporabljajo v obstoječi izvedbi, ni pri-

merna za uporabo v ščepcu. Podatki so zapisani v strukturi `DATA`, ki za shranjevanje matrik vsebuje strukture tipa `FULL_MAT`, za vektorje pa kazalce tipa `double*`. Struktura `DATA` ima spremenljivo velikost in je zato ni možno definirati ali prenesti v ščepec. Spremenili smo jo tako, da smo za shranjevanje statičnih vektorjev in matrik napisali novi strukturi `vec` in `mat`, ki imata vnaprej inicializirani tabeli vrednosti. Struktura `vec` vsebuje 99 vrednosti, kar je velikost največjega vektorja, ki se uporablja v funkciji `qp`. Struktura `mat` pa ima tabelo z 9.801 elementi, kar je dovolj za matrike do velikosti 99×99.

Poleg sprememb strukture `DATA` se vse vrednosti tipa `double` shranjuje in računana s tipom `float`. Razlog za to je, da grafična kartica, na kateri smo razvijali in testirali program, ne podpira računanja z dvojno natančnostjo.

V ogrodnem delu programa se ščepec inicializira podobno kakor je opisano v poglavju 6.1. Prilagojena struktura `DATA` se v GPE prenese že pri inicializaciji ščepca, v koraku alokacije pomnilnika na napravi. Tako se statične podatke v GPE prenese samo enkrat. V zanki se v vsaki iteraciji vhodni podatki prenesejo v pomnilnik GPE, izvede se ščepec in nato se izhodni podatki prenesejo nazaj v pomnilnik gostitelja.



Slika 6.2: Izvajanje ščepca `qp`.

Po ponovni analizi poteka algoritma smo ugotovili, da je računsko najbolj potraten sredinski del funkcije `qp`, ki je napisan v zanki. Te zanke ni možno paralelizirati, saj predstavlja iterativno iskanje in je zato vsaka iteracija te zanke odvisna od prejšnje.

Opazili smo, da lahko v poteku algoritma, ki smo si ga pripravili v zaporedni izvedbi, veliko zaporednih operacij paraleliziramo po elementih vektorjev in vrsticah matrik. Če vzamemo na primer diagonalno matriko $\mathbf{A}$, vektor $\mathbf{b}$ in vektor $\mathbf{c}$: pri izračunu $\mathbf{y} = \mathbf{Ab} + \mathbf{c}$ lahko prvi element vektorja $\mathbf{y}$ računamo neodvisno od drugih elementov. Vsaka delovna nit bi lahko paralelno računala svoj element. Po tem zgledu smo tako preoblikovali vrstice, da imajo čim več takšnih zaporednih operacij, ki se lahko paralelizirajo. V ščepcu `qp` se zato izvaja celotna metoda, brez klicanja pomožnih funkcij.

Tekom celotnega izvajanja algoritma računamo z vektorji velikosti 99 ali 33 in s celimi matrikami velikosti 99×81 ali manj. Zato je pri inicializaciji ščepca `qp` nastavljenih 128 delovnih niti na delovno skupino, število delovnih skupin pa je nastavljeno na 1. Število 128 smo izbrali zato, ker je prvi večkratnik števila 32, ki je večji od omenjenega največjega števila elementov 99. Število delovnih niti mora biti večkratnik števila 32, ker je to na naši GPE vrednost zaželenega množitelja velikosti delovne skupine (OpenCL parameter GPE z imenom `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`. Tako večji del programa 99 od 128 delovnih niti računa s svojimi elementi vektorjev in svojimi vrsticami matrik. Število aktivnih delovnih niti po vrsticah algoritma lahko vidimo na sliki 6.3.

Ponekod v ščepcu `qp` moramo pred naslednjo operacijo zagotoviti, da so vse vrednosti nekega vektorja, ki ga potrebujemo, že izračunane in shranjene v pomnilniku. Če tega ne zagotovimo, lahko določena delovna nit prebere vrednost, ki je druga delovna nit še ni izračunala ali zapisala. Ta pojav se imenuje tvegano stanje (angl. race condition). Problem smo rešili tako, da pred takšne operacije postavimo prepreko (angl. barrier). Prepreka sinhronizira delovne niti tako, da zagotovi čakanje delovnih niti, dokler vse ne dosežejo prepreke. Prepreke so v ščepcu implementirane z ukazom

N	Vrstica algoritma	Paralelno izvajanje
Začetni del:		
1	ref_adjust(*Lb*, *ub*)	1
2	**l** = diag(*E*) * *Lb*	99
3	**u** = diag(*E*) * *ub*	
4	**shift_arg** = *L1* + (*L2* * **in_x**)	
5	**eshift_arg** = diag(*E*) * **shift_arg**	
6	**q** = (*Q2* * **in_x**) + *q1*	33
7	**r** = (*R2* * **in_x**)	
Zanka:		
8	**x** = (*M* * **v**) + **q**	33
9	**h** = **v** + (*C* * **x**)	99
10	**p** = **h** + **eshift_arg**	
11	clip_soft(**p**)	
12	**p** = diag(*E*) * ((diag(*Einv*)*p)-**shift_arg**)	
13	**lambda_old** = **lambda**	
14	**lambda** = **h** - **p**	
15	**p** = (**lambda** - **lambda_old**) * theta[k]	
16	**v_old** = **v**	
17	**v** = **tmp_var_p** + **lambda**	
18	restart(**lambda**, **lambda_old**, **v**, **v_old**)	99
Končni del:		
19	**out_sol** = (*R* * **x**) + **r**	99

Slika 6.3: Prikaz paralelnega izvajanja ščepca `qp`. Mesta uporabe preprek so označena z vodoravnimi rdečimi črtami. Imena statičnih matrik in vektorjev iz strukture `DATA` so označena s poševnim tiskom.

`barrier(CLK_LOCAL_MEM_FENCE)`, ki prisili niti, da izračunane vrednosti iz lokalnega pomnilnika dokončno zapišejo nazaj vanj. Splakovanje v globalni pomnilnik ni potrebno, saj ga uporabljamo samo za konstante. Mesta uporabe opisanih preprek v poteku algoritma so z rdečo črto označena v prikazu paralelnega izvajanja na sliki 6.3.

Največji izziv pri sinhronizaciji s pregradami nam je predstavljalo pisanje učinkovite kode za funkcijo `restart`, ki vsebuje tudi skalarni produkt. Skalarni produkt se, tako kakor funkcija `restart`, nahaja v srednjem delu ščepca `qp`, v zanki. V tem delu je potrebno še posebej paziti na učinkovito izvajanje kode. Od rezultata skalarnega produkta sta odvisna dva vektorja, ki se takoj

spet uporabita za nadaljnjo računanje. Vsaka delovna nit izračuna svoj delni rezultat, delne rezultate pa je potem potrebno sešteti. Seštevanje smo preizkusili narediti z redukcijo, ki v 7 korakih sešteje vseh 99 delnih rezultatov. Za to je potrebno dodati 6 novih preprek. Ugotovili smo, da se pri tako malih velikostih vektorjev redukcija časovno ne izplača, saj se porabi preveč časa za sinhronizacijo. Seštevanje delnih rezultatov skalarnega produkta je zato v ščepcu `qp` implementirano zaporedno.

6.3 Prilagoditev dostopov do podatkov

Po obširnem testiranju in popravljanju manjših napak smo nadaljevali razvoj z optimizacijo dostopov do pomnilnika. Vsi statični podatki v strukturi `DATA` se pri inicializaciji prenesejo v globalni pomnilnik konstant. Dostopi do globalnega pomnilnika pri GPE pa so relativno potratni. Predvidevali smo, da bi program pohitrili, če bi pogosto uporabljene podatke prestavili v lokalni pomnilnik. S tem predvsem mislimo na matrike, ki so potrebne v sredinskem delu – delu, kjer je glavna zanka.

Ker je lokalni pomnilnik na GPE precej manjši od globalnega, smo se odločili zmanjšati prostor potreben za nekatere matrike. Večina matrik namreč vsebuje 3.267 elementov, kar ustreza matrikam velikosti 99×33 in 33×99. Tabelo v strukturi `mat` smo zato spremenili na velikost 3.267 elementov tipa `float`. Naredili smo tudi strukturo `mat_big`, ki lahko hrani do 8.019 elementov, kar ustreza večjim matrikam velikosti 99×81. Tako je velikost statičnih podatkov zmanjšana iz 358.900 B na 168.820 B. Optimizirana velikost podatkov, ki se uporabljajo tekom dela zanke, je 27.752 B.

Najbolj uporabljene matrike in vektorje se na začetku izvajanja ščepca vsakega ločeno asinhrono prenese iz globalnega pomnilnika v lokalni. Ukazi za asinhrone prenose so med sabo povezali v seznam dogodkov tipa `event_t`. Pred nadaljevanjem ščepca je z ukazom `wait_group_events` zagotovljeno, da se vsi prenosi končajo. Potek prenosa je ponazorjen z odsekom kode na sliki 6.4.

```
event_t copy_event_list[6]; // seznam dogodkov za prenašanje

__local mat C;     // asinhrono kopiranje globalne matrike
C.n = d->C.n;      // d->C v lokalno matriko C
C.m = d->C.m;
copy_event_list[0] = async_work_group_copy(C.data,
   d->C.data, d->C.n * d->C.m, 0);

...
wait_group_events(6, copy_event_list); // čakanje na dogodke
```

Slika 6.4: Izsek programske kode, ki prikazuje asinhrono kopiranje podatkov v ščepcu `qp`.

6.4 Spremembe računanja spremenljivke theta

V 15. vrstici poteka na sliki 6.3 se v vsaki iteraciji izračuna vrednost spremenljivke `theta`. V prvi iteraciji je vrednost spremenljivke `theta` 1, nato pa se vsako iteracijo sproti izračuna glede na svojo vrednost iz prejšnje iteracije po enačbi

$$theta_n = \frac{1 + \sqrt{1 + 4 * {theta_{n-1}}^2}}{2}, \tag{6.2}$$

kjer je n številka iteracije.

Računanje vrednosti *theta* je vedno enako in neodvisno od vhodnih spremenljivk. Prvih 100.000 vrednosti smo zato sami izračunali z ločenim programom in jih shranili v datoteko, podobno kot so shranjeni ostali statični podatki. Tako se lahko vnaprej izračunane vrednosti *theta* uporablja v zanki, brez da bi jih bilo potrebno vsako iteracijo ločeno izračunati.

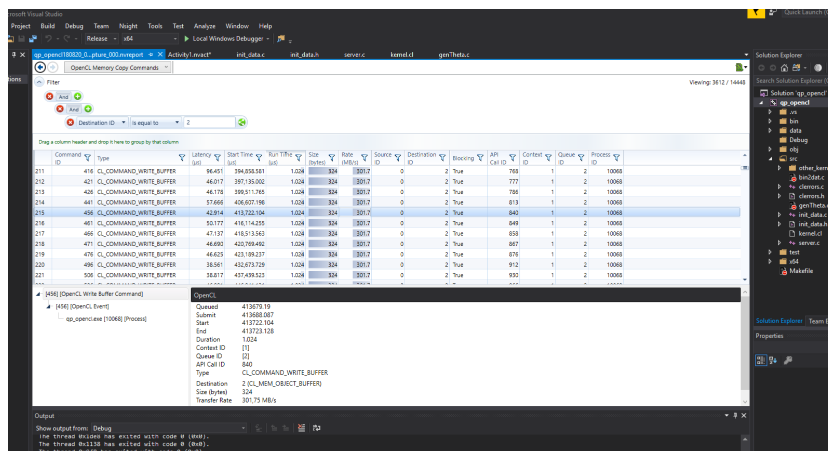
Poglavje 7

Meritve in rezultati

Vse meritve smo naredili na računalniku s procesorjem *Intel Core i7-6700K*. Pri merjenju časa izvedbe, narejene z orodjem OpenCL, smo program izvajali z grafično kartico *NVIDIA GeForce GTX 1070*. Frekvenco GPE smo imeli povečano za 100 MHz od privzete vrednosti. Najvišja frekvenca GPE med izvajanjem je bila 2.050 MHz, frekvenca grafičnega pomnilnika pa okoli 3.910 MHz. Za izvajanje meritev smo imeli na voljo operacijski sistem *Microsoft Windows 10* in operacijski sistem *Ubuntu 16.04*.

Meritve natančnosti in časa izvajanja smo vedno izvajali z enakim ogrodjem programa, ki je opisano v poglavju 4.2. To pomeni, da so vse meritve narejene s skupino istih 3.600 vhodnih podatkov. Spreminjali smo med drugim število notranjih iteracij v funkciji ali ščepcu `qp`.

Za podrobno analizo izvajanja programa na GPE smo uporabili orodje NVIDIA Nsight, s katerim se lahko analizira ukaze programskega vmesnika OpenCL. Orodje je na voljo kot dodatek za razvojno okolje Visual Studio. Z njim se lahko na primer spremlja prenose podatkov v pomnilnik GPE in nazaj v pomnilnik gostitelja.



Slika 7.1: Orodje za profiliranje NVIDIA Nsight v razvojnem okolju Visual Studio.

7.1 Meritve natančnosti

Rezultate vsake izvedbe smo z relativno napako RE, opisano v poglavju 4.3, primerjali z referenčnimi rezultati obstoječe zaporedne izvedbe. Primerjali smo samo pomembnih prvih enajst signalov v vsakem izmed izhodnih vektorjev. Referenčne rešitve so pridobljene na podlagi 100.000 iteracij z namenom da so čim bolj natančne.

Ko smo primerjali natančnost rešitev izvedb, smo za vsak izhodni vektor izračunali RE in nato naredili povprečje $\overline{RE}$, ki predstavlja povprečje vseh 3.600 nastalih relativnih napak za izvedbo. Zanimala nas je tudi maksimalna relativna napaka RE_{max}, ki predstavlja največjo relativno napako. Vrednost $RE_{max} = 0{,}01$ na primer pomeni, da se najslabši izhodni vektor signalov izvedbe od referenčne rešitve razlikuje za 1 odstotek, relativno glede na območja signalov.

ime izvedbe, natančnost	**št. iteracij**	$\overline{RE}$	RE_{max}
obstoječa zaporedna, dvojna	100.000	0	0
prilagojena zaporedna, dvojna	10.000	$0,90 \times 10^{-5}$	$0,14 \times 10^{-3}$
	1.000	$8,72 \times 10^{-5}$	$2,97 \times 10^{-3}$
	500	$9,78 \times 10^{-5}$	$3,64 \times 10^{-3}$
	250	$10,43 \times 10^{-5}$	$6,73 \times 10^{-3}$
prilagojena zaporedna, enojna	10.000	$0,97 \times 10^{-5}$	$0,21 \times 10^{-3}$
	1.000	$8,74 \times 10^{-5}$	$2,98 \times 10^{-3}$
	500	$9,79 \times 10^{-5}$	$3,64 \times 10^{-3}$
	250	$10,44 \times 10^{-5}$	$6,73 \times 10^{-3}$
paralelna OpenCL, enojna	10.000	$0,97 \times 10^{-5}$	$0,23 \times 10^{-3}$
	1.000	$8,74 \times 10^{-5}$	$2,98 \times 10^{-3}$
	500	$9,79 \times 10^{-5}$	$3,64 \times 10^{-3}$
	250	$10,44 \times 10^{-5}$	$6,73 \times 10^{-3}$

Tabela 7.1: Tabela meritev natančnosti glede na izvedbo in število iteracij v funkciji ali ščepcu `qp`.

Za različne izvedbe smo merili čas izvajanja funkcij ali ščepca `qp`, katerim smo nastavljali različno število notranjih iteracij. Več kot je nastavljenih iteracij, večkrat se ponovi korak iterativnega iskanja metode DHGM, kar pomeni, da so izhodni vektorji bolj natančni.

V tabeli 7.1 so predstavljene vrednosti RE za vsako izmed izvedb, katerih izhodni vektorji so se razlikovali, kljub istemu številu nastavljenih iteracij. Do tega je prišlo, ker nekatere izvedbe računajo z dvojno natančnostjo plavajoče vejice (vrednostmi tipa `double`), nekatere pa z enojno natančnostjo (vrednostmi tipa `float`). Sklepali smo, da lahko do razlik pride tudi zaradi računskih napak, ki nastanejo zaradi morebitnih razlik v izvedbi tipov ali računskih operacij.

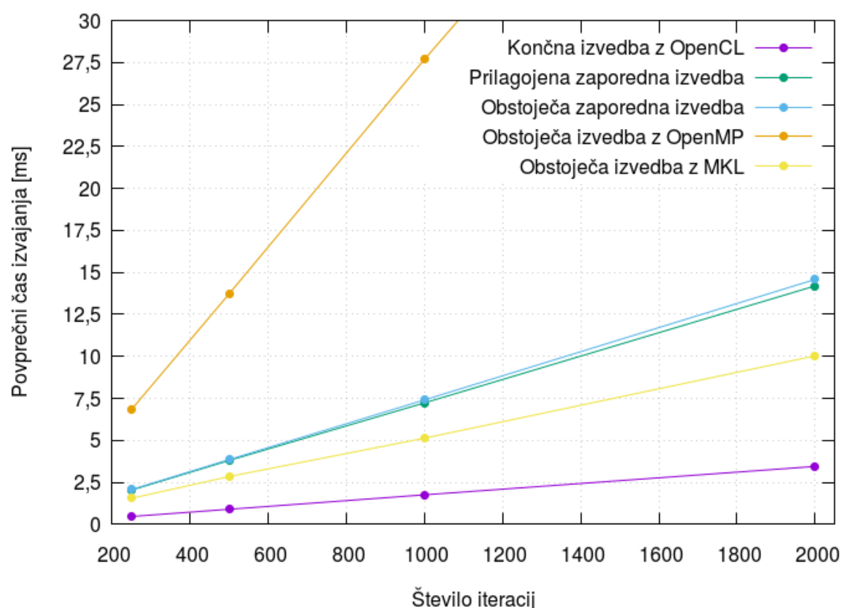
Opazimo lahko, da se vrednosti RE pri različnih izvedbah in istem številu nastavljenih iteracij skoraj ne razlikujejo. To pomeni, da so relativne napake

odvisne predvsem od števila korakov iterativnega iskanja in ne toliko od natančnosti plavajoče vejice.

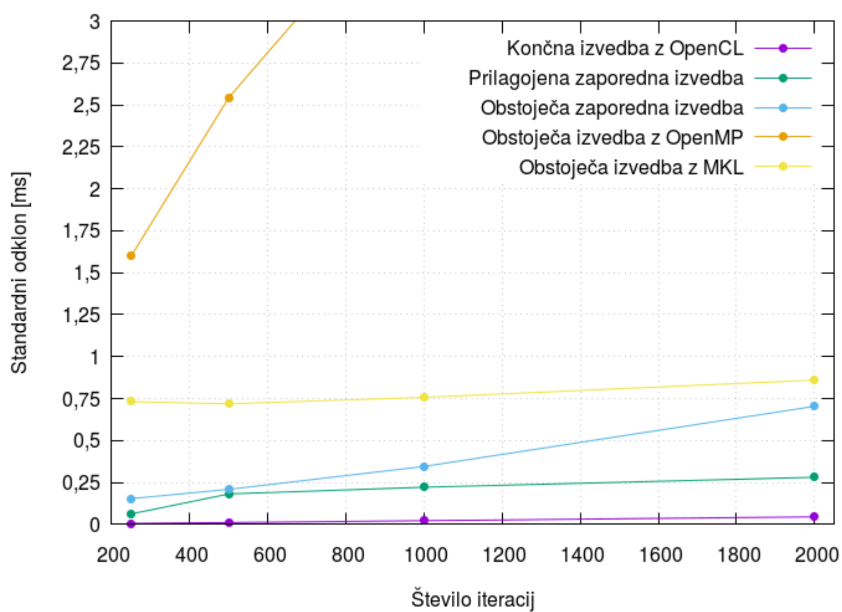
Zelo majhne razlike relativnih napak se lahko opazi med prilagojeno sekvenčno izvedbo, ki računa s tipom `double`, in izvedbami, ki računajo s tipom `float`. Razlike relativnih napak so tako majhne in konsistentne, da smo kljub temu lahko še vedno prepričani v pravilnost programa.

7.2 Meritve časa izvajanja

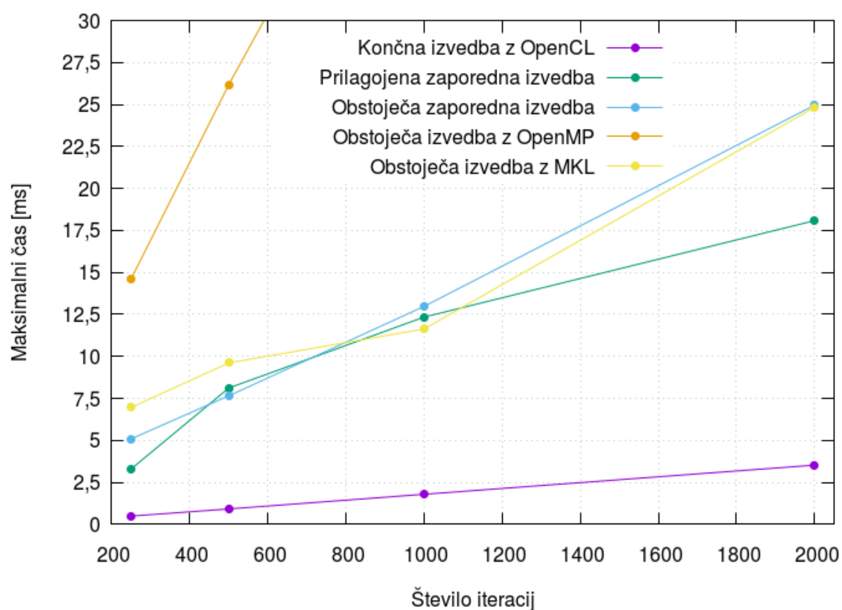
Za vsako izvajanje funkcije ali ščepca `qp` smo izmerili čas izvajanja t in izračunali povprečje $\bar{t}$, maksimalno vrednost t_{max} in standardni odklon σ. Pri izračunu vrednosti $\bar{t}$, t_{max} in σ upoštevamo samo zadnjih 2.700 od 3.600 zaporednih izvajanj. S tem je zagotovljeno, da morebitni začetni pojavi pri izvajanju z manj iteracijami ne vplivajo na izračune. Vse meritve časa izvajanja so predstavljene v milisekundah.



Slika 7.2: Prikaz povprečnih izmerjenih časov izvajanja.



Slika 7.3: Prikaz standardnih odklonov pri meritvah časa izvajanja.



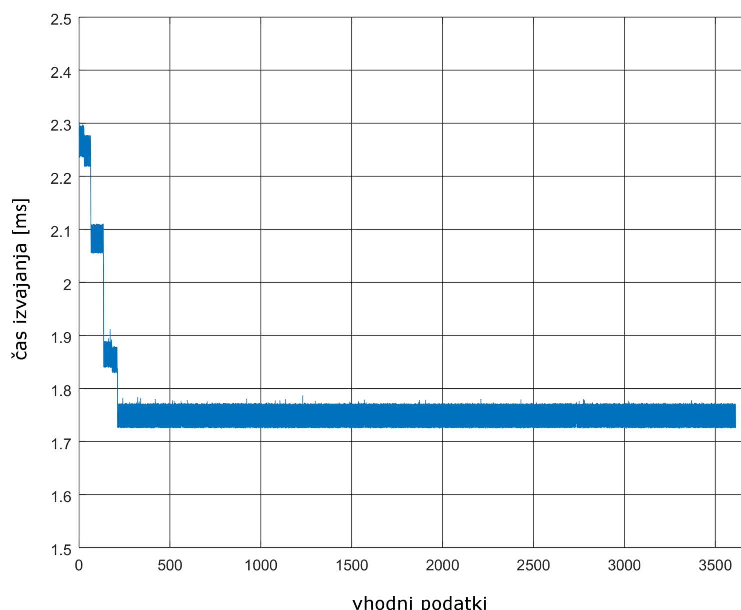
Slika 7.4: Prikaz izmerjenih maksimalnih časov izvajanja.

Čas izvajanja smo izmerili za obstoječo zaporedno in prilagojeno zaporedno izvedbo. Izmerili smo tudi čas izvajanja izvedb z obstoječimi poskusi paralelizacije (izvedba z OpenMP in izvedba z MKL). Naštete meritve primerjamo z izvedbo z ogrodjem OpenCL na GPE. Programe smo prevajali z optimizacijo O3. Izvedbe smo merili z različnim številom iteracij funkcije ali ščepca `qp`. Meritve časa izvajanja so prikazane na slikah 7.2, 7.3 in 7.4.

Iz grafa na sliki 7.2 se lahko prepričamo, da imata obstoječa zaporedna in prilagojena zaporedna izvedba zelo podobna časa izvajanja. Pri obstoječi izvedbi z OpenMP lahko opazimo zelo povečan čas izvajanja v primerjavi z zaporedno izvedbo. Sklepali smo, da je vzrok prepogosta delitev dela, ki se izvede pri skoraj vsaki matrični operaciji. Od obstoječih izvedb ima najboljši čas izvajanja izvedba z MKL. Končna izvedba z OpenCL pa na testirani GPE dosega najhitrejši čas izvajanja. V primerjavi z zaporedno izvedbo, ki se izvaja na CPE, se na testirani GPE izvaja približno štirikrat hitreje. Opazimo lahko tudi, da je izvajanje na GPE tudi zelo konsistentno in ima pri vsaki meritvi zelo majhen standardni odklon. Najverjetneje je vzrok za to majhno število vejitev.

Na sliki 7.5 je prikazano kako konsistentne so meritve izvedbe z OpenCL. Pri prvih 200 meritvah lahko vidimo posledice dinamičnega spreminjanja napetosti in frekvence na GPE. Na omenjeno dinamično spreminjanje smo bili vnaprej pripravljeni, zato teh meritev pri izračunih $\bar{t}$, σ in t_{max} ne upoštevamo.

Opravili smo tudi meritve časa izvajanja posameznih OpenCL različic brez nekaterih sprememb, ki so bile dodane tekom razvoja. Izmerili smo čas izvedbe z OpenCL brez uporabe lokalnega pomnilnika. Zanimalo nas je tudi, kakšen je bil vpliv dodajanja sinhronizacije delovnih niti s preprekami. Izvedbi brez vseh nujnih preprek sicer dajeta napačne rešitve in služita samo za meritve izvajanja. Meritve so prikazane v tabeli 7.2.



Slika 7.5: Prikaz meritev časa izvajanja končne izvedbe z OpenCL pri 1.000 iteracijah. Prikazanih je vseh 3.600 zaporednih meritev.

ime izvedbe	$\bar{t}$ [ms]	σ [ms]	t_{max} [ms]
brez preprek	3,240	0,027	3,320
s samo eno prepreko v zanki	1,677	0,028	1,794
brez uporabe lokalnega pomnilnika	6,013	0,032	6,187
končna OpenCL	1,760	0,023	1,797

Tabela 7.2: Čas izvajanja različic izvedbe z OpenCL pri 1.000 iteracijah.

Po pričakovanju lahko opazimo, da je čas izvajanja izvedbe brez uporabe lokalnega pomnilnika bistveno daljši. Prenos pogosto uporabljenih podatkov iz globalnega v lokalni pomnilnik predstavlja najučinkovitejšo optimizacijo, ki je bila narejena tekom razvoja izvedbe z ogrodjem OpenCL. Ugotovili smo, da je izvajanje različice brez vseh preprek približno dvakrat daljše od končne izvedbe z OpenCL. Do tega verjetno pride zaradi nesinhronih dostopov do

pomnilnika. Pri dodajanju samo ene prepreke v zanko smo dobili najboljši čas izvajanja, vendar taka izvedba seveda še vedno proizvaja napačne rešitve.

ime prenosa	smer prenosa	velikost [B]	$\bar{t}$ [μs]
vektor **in_x**	vhodni	324	0.60
vektor **in_ref**	vhodni	84	0.52
vektor **out_sol**	izhodni	132	0.67

Tabela 7.3: Meritve časa vhodno-izhodnih prenosov pri izvedbi z OpenCL.

Iz analize ukazov za prenos podatkov v pomnilnik GPE in nazaj v pomnilnik gostitelja je razvidno, da so povprečni časi vhodno-izhodnih prenosov relativno kratki v primerjavi z izvajanjem ščepca. Meritve in velikosti vhodno-izhodnih prenosov so predstavljeni v tabeli 7.3.

Poglavje 8

Zaključek

V diplomski nalogi smo se seznanili z ozadjem in namenom metode DHGM, ki se uporablja v prediktivnem regulatorju za omejevanje oblike in toka plazme v napravah tipa tokamak. Opisali smo napravo tokamak in se seznanili s konceptom prediktivnega vodenja, ki ga rešuje metoda DHGM.

Po podrobni analizi obstoječe izvedbe in strukture projekta smo napisali bolj strnjeno zaporedno izvedbo metode DHGM. Naredili smo tudi novo strukturo in ogrodje projekta, ki omogoča sprotno testiranje pravilnosti in merjenje povprečnega časa izvajanja programa.

Ko smo v okviru projekta FMPCFMPC sprejeli nalogo paralelizacije metode DHGM z orodjem OpenCL, je sam problem, sestavljen iz matrik in vektorjev, na prvi pogled izgledal dokaj preprost za paralelizacijo. V času analize smo hitro ugotovili, da sta iterativna narava algoritma in majhne velikosti vektorjev ter matrik zelo otežili paralelizacijo na GPE.

Pri prvih poskusih je bil čas izvajanja paralelne izvedbe bistveno slabši od zaporedne izvedbe. Potrebni so bili popravki napak, ki so se pojavile tekom razvoja. Pri vsakem koraku je bilo potrebno sprotno testiranje. Vendar nam je po premiku celotne metode DHGM v en ščepec in po nekaj optimizacijah uspelo dobiti dokaj kratke čase izvajanja v primerjavi s prejšnjimi izvedbami. Pri merjenju končne različice izvedbe z OpenCL je bil čas izvajanja na testirani GPE približno trikrat hitrejši od obstoječe izvedbe z MKL, kar se nam

je zdel zelo dober dosežek. Čas izvajanja pa bi moral biti za uporabo in testiranje v manjših tokamakih še krajši.

Nadaljnjo optimizacijo izvedbe z OpenCL bi lahko naredili s še podrobnejšo analizo uporabe pomnilnika in poteka metode DHGM. Ena od možnosti za nadaljnji razvoj bi lahko bila prilagoditev programske kode OpenCL za optimizirano izvajanje na FPGA [13].

Literatura

[1] ABB. Model predictive control technology demystified. Dosegljivo: https://new.abb.com/control-systems/features/model-predictive-control-mpc, 2018. [Dostopano 25. 8. 2018].

[2] Eduardo F. Camacho, Carlos Bordons Alba. Introduction to model predictive control. In *Model Predictive Control, Second Edition*, pages 1–3. Springer Science & Business Media, 2013.

[3] OpenMP Architecture Review Board. OpenMP. Dosegljivo: https://www.openmp.org, 2018. [Dostopano 25. 8. 2018].

[4] Kristen Boydstun. Introduction OpenCL. Dosegljivo: http://www.tapir.caltech.edu/~kboyds/OpenCL/opencl.pdf, 2018. [Dostopano 11. 8. 2018].

[5] EUROfusion. JET - Europe's Largest Fusion Device – Funded and Used in Partnership. Dosegljivo: https://www.euro-fusion.org/faq/what-is-the-q-value-for-jet, 2018. [Dostopano 1. 2. 2018].

[6] EUROfusion. Ohmic Heating Networks. Dosegljivo: https://www.euro-fusion.org/news/detail/detail/News/fusion-spin-off-ohmic-heating-networks, 2018. [Dostopano 25. 8. 2018].

[7] Pontus Giselsson. QPgen. Dosegljivo: http://www.control.lth.se/QPgen/index.html, 2018. [Dostopano 5. 2. 2018].

[8] Khronos Group. About The Khronos Group. Dosegljivo: https://www.khronos.org/about, 2018. [Dostopano 25. 8. 2018].

[9] Khronos Group. OpenCL Overview. Dosegljivo: https://www.khronos.org/opencl, 2018. [Dostopano 25. 8. 2018].

[10] Khronos Group. The OpenCL C 2.0 Specification, Document Revision: 40, Version 2.2-7. Dosegljivo: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_C.pdf, 2018. [Dostopano 13. 8. 2018].

[11] Khronos Group. The OpenCL Specification, Version 2.1, Document Revision 24. Dosegljivo: https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf, 2018. [Dostopano 10. 8. 2018].

[12] Samo Gerkšič, Boštjan Pregelj, Matija Perne, Matic Knap, Iztok Ramovš, Gianmaria De Tommasi, Marco Ariola in Alfredo Pironti. EUROfusion Work Programme 2015 Enabling Research, Project: Fast Model Predictive Control for Magnetic Plasma Control. Deliverable D6: Evaluation of fast MPC for ITER RWM control, 2017.

[13] Intel. Intel FPGA SDK for OpenCL. Dosegljivo: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, 2018. [Dostopano 26. 8. 2018].

[14] Intel. Intel Math Kernel Library. Dosegljivo: https://software.intel.com/en-us/mkl, 2018. [Dostopano 25. 8. 2018].

[15] ITER. What is tokamak? Dosegljivo: https://www.iter.org/mach/tokamak, 2017. [Dostopano 7. 5. 2017].

[16] ITER. Plasma Heating. Dosegljivo: https://www.iter.org/sci/PlasmaHeating, 2018. [Dostopano 1. 2. 2018].

[17] ITER. Science - Making it Work. Dosegljivo: https://www.iter.org/sci/makingitwork, 2018. [Dostopano 1. 2. 2018].

[18] ITER. What is ITER? Dosegljivo: https://www.iter.org/proj/inafewlines, 2018. [Dostopano 1. 2. 2018].

[19] MathWorks. Quadratic Programming. Dosegljivo: https://www.mathworks.com/discovery/quadratic-programming.html, 2018. [Dostopano 5. 2. 2018].

[20] MathWorks. Simulink Documentation. Dosegljivo: https://www.mathworks.com/help/simulink, 2018. [Dostopano 10. 2. 2018].

[21] Inštitut Jožef Stefan. Fast Model Predictive Control for Magnetic Plasma Control. Dosegljivo: http://dsc.ijs.si/en/projects/FMPCFMPC, 2018. [Dostopano 25. 8. 2018].