Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gregor Vitek

Prevajanje modela OpenACC na nivoju izvorne kode

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2018

Zahvala

Rad bi se zahvalil svojemu mentorju doc. dr. Boštjanu Slivniku za strokovno pomoč pri izdelavi diplomskega dela.

Prav tako se želim zahvaliti družini in prijateljem, ki so me vzpodbujali pri delu.

Gregor Vitek, 2018

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPE	Central processing unit	Centralna procesna enota
GPE	Graphical processing unit	Grafična procesna enota
GPGPU	General purpose computing on GPU	Računanje na GPE
SIMD	Single instruction multiple data	enojni ukaz z več podatki
KNG	Context free grammar	Kontekstno neodvisna gramatika
AST	Abstract syntax tree	Abstraktno sintaksno drevo

Povzetek

Naslov: Prevajanje modela OpenACC na nivoju izvorne kode

Razvili smo prevajalnik, ki omogoča dodajanje direktiv OpenACC v programski jezik Java. Implementirali smo ga s pomočjo prevajanja na nivoju izvorne kode. Prevajalnik deluje kot predprocesor, ki javansko kodo z oznakami OpenACC prevaja v javansko kodo, ki uporablja vmesnik OpenCL za računanje na grafičnih procesnih enotah. Izhodna koda prevajalnika je prevedljiva s katerimkoli javanskim prevajalnikom in lahko teče na vseh računalnikih z Javinim navideznim strojem in podporo za OpenCL. Zaradi obsega standarda OpenACC smo se pri implementaciji osredotočili le na osnovno paralelizacijo zank in prenos podatkov med napravami. S prevajalnikom smo dosegli pospešitve izvajanja kompleksnih problemov za okoli 50% in dosegli enak čas izvajanja kot z jezikom C++, ki uporablja OpenCL.

Ključne besede

Prevajalnik, GPE, OpenACC, OpenCL

Abstract

Title: Source-to-source Compilation of OpenACC

The OpenACC framework was created to alleviate the problems with programming graphical processing unit. We implemented a compiler, which added OpenACC support to the Java programming language. We used source-to-source translation to compile Java code annotated with OpenACC directives into Java code which uses OpenCL to execute computations on the GPU. The code compiled with our compiler is compatible with any Java compiler and runs on any machine that supports the Java virtual machine and OpenCL. Due to the size of the OpenACC standard, we focused mostly on the core features of the framework, such as loop parallelization and data transfer. The code generated with our compiler achieved around a 50% speed-up, and achieved parity with native C++ programs using OpenCL.

Keywords

Compiler, GPU, OpenACC, OpenCL

Poglavje 1

Uvod

Po dolgih letih razvoja računalniške opreme so se zaradi fizikalnih omejitev proizvajalci centralnih procesnih enot (CPE) (ang. *central processing unit (CPU)*) usmerili v vzporedno računanje. Vzporedno so se za potrebe računalniške grafike začele uveljavljati grafične procesne enote (GPE) (ang. *graphics processing unit (GPU)*). Zgodnje GPE so imele le statične grafične cevovode, ki so imeli zelo omejen nabor funkcionalnosti, namenjen pretvorbi in rasterizaciji geometrijskih objektov. Za potrebe računalniške grafike so te GPE počasi postale preveč toge in nefleksibilne, zato so jih nadomestile GPE s programirljivimi senčilniki (ang. *programmable shaders*). To pa je pomenilo, da so GPE postale bolj splošno namenske in odprle možnost za uporabo svoje računske moči za druga področja znanosti. Tako je prišlo do povečanja računanja na GPE (ang. *General purpose computing on GPU (GPGPU)*).

Problem pri programiranju GPE je, da je njihova arhitektura drugačna kot arhitektura, katere so programerji vajeni. Medtem ko imajo CPE širok nabor funkcionalnosti, so GPE namenjene predvsem izvajanju aritmetično-logičnih operacij. Moderne CPE imajo glavni pomnilnik z več nivoji predpomnilnika, ki ga programerji ne morejo nadzirati, GPE pa imajo po navadi hierarhičen pomnilnik, kjer imajo računska jedra poleg glavnega pomnilnika dostop tudi do svojih lokalnih pomnilnikov. Uporaba teh pomnilnikov ni transparentna kot pri CPE, saj mora programer sam določiti, kateri podatki

so shranjeni v katerem od teh pomnilnikov.

Današnji gonilniki GPE tako vsebujejo, poleg vmesnikov za računalniško grafiko (kot je OpenGL), tudi vmesnike za GPGPU. Vmesnikov je več vrst, večinoma pa so omejeni na nizko-nivojske programske jezike, kot sta Fortran in C++. Trenutno največji proizvajalec GPE, podjetje Nvidia, v svojih gonilnikih GPE ponuja več vmesnikov za računanje. Vmesnik Cuda je implementiran prek njihovega lastniškega prevajalnika za C/C++ `nvcc`, ki dostop do GPE omogoča preko vgrajenih funkcij in direktiv, v nasprotju pa je vmesnik za OpenCL implementiran kot zunanja knjižnica za ostale prevajalnike, kot so `gcc`, `clang` in `cl` (prevajalnik C++ podjetja Microsoft). V višje-nivojskih jezikih redko najdemo podporo za računanje na GPE s strani proizvajalca, zato pa obstaja veliko knjižnic, ki preko nizko-nivojskih programov dostopajo do GPE.

V tem delu smo se osredotočili na podporo računanju na GPE v jeziku Java. Java je zelo razširjen in pogosto rabljen jezik, ki pa nima direktnega vmesnika za GPGPU. Obstaja več knjižnic, ki po navadi implementirajo dostop preko OpenCL, vendar so večinoma implementirane kot zelo tanka ovojnica okoli programskega ogrodja v jeziku C, kar pomeni, da je njihov programski model imperativen. To je neskladno s programskim modelom Jave, ki spodbuja uporabo objektno orientiranega programiranja.

Pri vzporednem računanju obstaja več modelov programiranja. Najpogosteje od programerja zahtevajo, da eksplicitno nadzira potek računanja, upravlja z nitmi in jih programira v drugačnem jeziku, kot je jezik, v katerem program teče. Programsko ogrodje, ki uporablja tak model, je OpenCL. Nekateri modeli pa so bolj preprosti za programerje, saj omogočajo, da s posebnimi oznakami označijo sekvenčen program, katerega nato prevajalnik prevede v kodo, ki teče vzporedno. Taka modela sta na primer OpenMP za vzporedno računanje na CPE in OpenACC, ki je namenjen vzporednem računanju na GPE.

OpenACC [1] je programski vmesnik za heterogeno računanje, ki omogoča označevanje obstoječe kode, s katero prevajalniku sporočimo, da želimo, da

se del kode izvede vzporedno na GPE, CPE ali na drugih procesnih enotah. Kode, ki jo želimo paralelizirati, ne spreminjamo, ampak jo označimo s posebnimi oznakami, ki prevajalniku sporočijo, kako naj paralelizacijo izvede [2].

Za jezik C je najosnovnejši primer programa v OpenACC seštevanje vektorjev, ki ga vidimo v izseku kode 1.1.

```
int main(){
    const int N = 100;
    int a[N];
    int b[N];
    int c[N];

    /* inicializacija a in b*/

    #pragma acc parallel loop
    for(int i=0; i<N; i++){
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

Izsek kode 1.1: Seštevanje vektorjev v jeziku C z OpenACC

Želeli bi, da bi v Javi lahko storili podobno. Primer ekvivalentne kode, napisane v Javi, katero bi želeli podpreti, vidimo v izseku 1.2.

```
public class VectorAdd {
    public static void main(String[] args){
        final int N = 100;
        int[] a = new int[N];
        int[] b = new int[N];
        int[] c = new int[N];

        /* inicializacija a in b*/

        #acc parallel loop
        for(int i=0; i<N; i++){
```

```
                c[i] = a[i] + b[i];
            }
        }
    }
```

Izsek kode 1.2: Seštevanje vektorjev v Javi z OpenACC

Na teh dveh primerih vidimo primer uporabe OpenACC – v kodi le označimo zanko, za katero želimo, da se izvede vzporedno. Pomembno je, da je rezultat izvajanja kode z oznakami OpenACC enak izvajanju brez njih. To pomeni, da lahko prevajalnik, ki ne podpira OpenACC, označbe ignorira. Rezultat programa tako ostane isti, edina sprememba je način izvajanja.

V sklopu dela smo želeli implementirati prevajalnik, ki bi prevajal kodo programskega jezika Java, označeno z OpenACC, da bi jo lahko izvajali na GPE. Da se ne bi ukvarjali z združljivostjo naše kode z GPE različnih proizvajalcev, smo odločili uporabiti vmesnik OpenCL za Javo. Javo z OpenACC smo prevajali v ukaze OpenCL na nivoju izvorne kode, OpenCL C kodo pa smo prevedli z obstoječimi prevajalniki, ki jih najdemo v gonilnikih GPE.

V sledečih poglavjih najprej pregledamo ogrodja in podobna dela za računanje na GPE, nato podrobno opišemo OpenACC, predvsem dele, ki smo jih implementirali, predstavimo pa tudi nekaj primerov. Sledi opis implementacije prevajalnika za realizacijo OpenACC v jeziku Java. Na koncu še predstavimo teste in meritve kode, ustvarjene z našim prevajalnikom, ter predstavimo implementirane funkcionalnosti.

Poglavje 2

Pregled sorodnih del

Zgodnja podpora za odprto-kodno vzporedno računanje je prišla s programskim orodjem PVM (ang. *Parallel virtual machine*) [3]. PVM je ogrodje za porazdeljeno računanje, ki programerju omogoča, da programsko izrabi več osebnih računalnikov, CPE ali drugih procesorjev. PVM deluje po načelu mojster-suženj. Programer s svojega računalnika, ki deluje kot mojster, požene program, ki vsem sužnjem razdeli delo. Ko ti delo opravijo, sporočijo rezultat mojstru, ta pa izvajanje programa zaključi.

Uporaba PVM se je sčasoma prenehala, nadomestil jo je MPI (ang. *Message passing interface*) [4]. MPI je prav tako ogrodje za porazdeljeno procesiranje, vendar vsebuje veliko večji nabor funkcionalnosti, kot na primer podporo za delitev dela med niti in podporo za GPGPU.

Za razvoj OpenACC je bil zelo pomemben razvoj OpenMP [5], saj je OpenACC izšel iz OpenMP. OpenMP je vmesnik, narejen za vzporedno računanje v sistemih s skupnim pomnilnikom, npr. v sodobnih CPE, ki imajo več procesorskih jeder, ki pa si delijo skupen glavni pomnilnik. Specifikacija OpenMP obstaja za jezike C, C++ in Fortran. Uporaba OpenMP je skoraj identična uporabi OpenACC: obstoječo sekvenčno kodo označimo z oznakami OpenMP, ki prevajalniku povejo, kako naj kodo prevede v vzporedno kodo. Če prevajalnik ne podpira vmesnika OpenMP, lahko označbe preprosto ignorira, saj naj označbe same ne bi spremenile rezultata programa, le način izvajanja.

Z verzijo 4.0 je OpenMP dobil tudi podporo za računanje na GPE, ki pa je zelo primerljiva OpenACC.

Za eksplicitno računanje na GPE obstaja več vmesnikov. Podjetje Nvidia je za računanje na svojih GPE razvilo vmesnik CUDA [6], ki je implementiran prek njihovega lastniškega prevajalnika za C/C++ `nvcc` in dostop do GPE omogoča z uporabo vgrajenih funkcij in direktiv. CUDA je bil eden prvih vmesnikov za GPGPU, ki je postal razširjen in uporaben.

Zaradi želje po bolj odprti verziji vmesnika CUDA, je podjetje Apple, v sodelovanju s podjetji Nvidia, AMD, IBM, Qualcomm in Intel, predstavilo predlog za OpenCL skupini Khronos [7]. Khronos od takrat skrbi za specifikacijo OpenCL. OpenCL je programski vmesnik, ki omogoča izvajanje programov na GPE (in ostalih koprocesorjih). V OpenCL programer programira CPE drugače kot GPE. CPE deluje kot gostitelj in ga programiramo iz katerega koli jezika, ki ima OpenCL. GPE in ostale naprave, ki delujejo kot gosti, pa programira v posebnem jeziku OpenCL C. Program za CPE in program za GPE se tako pišeta ločeno. Gostitelj med tekom programa prevede kodo v strojno kodo za gosta, ki se nato izvede. Za Javo obstaja nekaj OpenCL vmesnikov. Eden izmed njih je JOCL [8]. JOCL je tanek vmesnik OpenCL v Javi, ki posnema vmesnik OpenCL za jezik C. Pred kratkim so pri podjetju Apple naznanili, da ukinjajo podporo za OpenCL. Podporo OpenCL so nadomestili s svojim zaprto-kodnim vmesnikom Metal Performance Shaders [9].

Podpora za OpenACC za jezike C, C++ in Fortran obstaja v nekaj komercialnih in odprtokodnih prevajalnikih. Vmesnikov za ostale jezike ni. Prevajalnik podjetja Cray ima podporo za oznake OpenACC, vendar prevajalniki delujejo le na računalniških sistemih njihove izdelave [10], OpenACC oznake pa prevajajo le v jezik Nvidia PTX [11]. PGI [12], prevajalnik za OpenACC podjetja Nvidia, deluje na splošnejših arhitekturah (na primer x86), vendar prav tako lahko prevaja kodo le za naprave, ki podpirajo jezik PTX [13]. Prevajalnik GCC podpira OpenACC, vendar ne v celoti [14, 15], podporo za OpenACC pa so napisali v podjetju Nvidia tako, da so podprte

le naprave, katerih gonilnik ima zmožnost prevajati PTX. V GCC so podporo za OpenACC v jeziku Fortran razvijali v podjetju Samsung [16]. Da so se izognili implementaciji proizvajanja strojne kode, so prevajali na ravni izvorne kode s pomočjo vmesnika OpenCL.

Obstaja nekaj poskusov prevajanja modela OpenACC v OpenMP 4.0 [17, 18], bodisi z ročno pretvorbo oznak OpenACC v oznake na OpenMP bodisi popolno prevajanje, ki omogoča prevajanje testnih programov EPCC [19]. Predstavljenih je tudi nekaj poskusov prevajanja OpenACC na nivoju izvorne kode v OpenCL ali CUDA [20, 21, 22, 23, 24]. Ti prevajalniki prevajajo direktno v ščepce OpenCL ali pa uporabljajo različne knjižnice, ki omogočajo prevajanje v več vmesnikov hkrati.

Podpora za GPGPU v visoko-nivojskih jezikih je zelo razširjena. V jeziku Python najdemo veliko število vmesnikov. Poznamo knjižnici PyCUDA in PyOpenCL, ki sta le prenešena vmesnika CUDA in OpenCL v Python [25, 26]. V jeziku Python poznamo tudi knjižnico Numba [27, 28], ki je po uporabi zelo podobna OpenACC. Namenjena je vektorizaciji in avtomatični paralelizaciji kode. Njena uporabnost pri računanju na GPE je omejena na vektorske operacije knjižnice numpy. Deluje tako, da funkcije, ki jih označimo z dekoratorji, med tekom programa s pomočjo LLVM prevede v strojno kodo za GPE.

V jeziku za statistično računanje R poznamo knjižnico gpuR, ki jeziku dodaja podporo GPUGP [29]. Knjižnica preko posebnih objektov, ki se obnašajo kot vgrajeni matrični objekti jezika, v ozadju računanje opravlja z GPE. Podobno podporo za računanje na GPU ima tudi MATLAB [30], kjer s posebnimi funkcijami na GPE ustvarimo matrike, nato pa z njimi delamo kot z matrikami, ki so vgrajene v jeziku. Obe knjižnici opravita prevod kode v kodo CUDA v ozadju, brez programerjeve pomoči.

Podporo za GPGPU poznamo tudi v nekaterih funkcijskih jezikih. Jezik F# pozna več vmesnikov za računanje na GPE, kot sta Alea GPU in Brahma.FSharp [31, 32], ki prevajata kodo F# v vmesnik CUDA in OpenCL ter knjižnico GpuLinq[33], ki prevaja poizvedbeno ogrodje LINQ iz jezikov

C# in F# v OpenCL.

Poglavje 3

OpenACC

OpenACC je standardiziran programski vmesnik [34] za programiranje splošno namenskih GPE. Osnovan je na vmesniku OpenMP. Kot v OpenMP lahko tudi v OpenACC v sekvenčni kodi označimo, kateri del kode želimo paralelizirati. To storimo s posebnimi direktivami, ki jih interpretira prevajalnik. Te direktive so enake, ne glede na to, na kakšni strojni opremi bo program tekel.

3.1 Model izvajanja OpenACC

OpenACC model izvajanja predvideva, da imamo dve napravi: gostitelja (ang. *host*) in gosta (ang. *guest*). Gostitelj je tista naprava, ki izvaja večino programa in koordinira izvajanje programa na vseh napravah. V praksi je to vedno CPE. Gost, imenovan tudi pospeševalnik (ang. *accelerator*), je naprava, namenjena predvsem izvajanju zelo računsko intenzivnih delov programa. Naprava izvaja vzporedna področja (ang. *parallel regions*) in področja ščepcev (ang. *kernels regions*), ki vsebujejo zanke, katere naprava izvaja vzporedno. Standard dopušča, da naprava izvaja tudi zaporedna področja (ang. *serial regions*), kjer se koda izvaja sekvenčno. Gostje so po navadi GPE, lahko pa so tudi CPE ali ostali pospeševalniki, kot na primer Intel Xeon Phi.

Gostitelj je odgovoren za izvajanje sekvenčnih delov programa, prav tako pa je odgovoren za koordinacijo med gostom in gostiteljem, prevajanje kode za gosta, dodelitev dela gostu, rezerviranje in sproščanje pomnilnika na gostu ter prenos podatkov in argumentov med gostom in gostiteljem. Za vsakega gosta ima gostitelj svojo ukazno vrsto (ang. *command queue*). Vse operacije, ki jih gostitelj želi izvesti na gostu, uvrsti v njegovo ukazno vrsto, gost ukaze jemlje iz ukazne vrste v enakem vrstnem redu, kot so bile vanj uvrščene in jih izvede. Gostje po navadi delujejo asinhrono ostalim napravam.

Poznamo tri nivoje paralelizma: grobo-zrnati (ang. *coarse grain*), drobnozrnati (ang. *fine grain*) in vektorski. Grobo-zrnati paralelizem je paralelizem nalog, kjer vsaki procesni enoti dodelimo nalogo. OpenACC ga imenuje gručni (ang. *gang*) paralelizem. Drobno zrnati paralelizem je paralelizem, kjer delo razdelimo delavcem (ang. *workers*). Vsak delavec teče v svoji programski niti, več niti pa izmenjujoče ali vzporedno teče na eni procesni enoti. Vektorski paralelizem pa je po navadi poznan pod kratico SIMD (ang. *single instruction multiple data*), kar pomeni, da z enim aritmetičnim ukazom operiramo na več podatkih hkrati.

OpenACC prepoveduje sinhronizacijo med delavci ali gručami. To je zato, ker standard implementacijam dovoljuje, da vseh gruč ne zaženejo hkrati, ampak jih izvajajo sekvenčno. To pomeni, da morajo biti programi, ki jih prevajamo z OpenACC, deljivi med niti tako, da ne potrebujemo sinhronizacije.

Iz vsake direktive, ki določa področje za izvajanje na GPE, prevajalnik ustvari kos kode, ki bo tekel na GPE. Temu kosu kode pravimo ščepec (ang. *kernel*), kdaj pa tudi računski senčilnik (ang. *compute shader*).

3.2 Pomnilniški model OpenACC

Velik problem pri programiranju zunanjih pospeševalnikov je, da imajo naprave, kot so GPE, svoj pomnilnik. To pomeni, da je potrebno podatke, ki jih obdeluje GPE, prenesti v njen pomnilnik, nato pa rezultate prenesti nazaj. V

bolj eksplicitnih vmesnikih za programiranje GPE, kot je OpenCL, mora to programer storiti ročno, v OpenACC pa mora za te operacije poskrbeti prevajalnik. Programer lahko prevajalniku poda namige v obliki podatkovnih direktiv, kdaj naj podatke prenese na napravo in kdaj nazaj v glavni pomnilnik. Prav tako imajo GPE lahko več-nivojski pomnilnik. Ta se po navadi deli na globalnega, do katerega lahko dostopajo vse niti, ki se izvajajo na napravi, in lokalnega, ki je skupen le nitim, ki se izvajajo na isti procesni enoti. To se razlikuje od predpomnilniške arhitekture na CPE v tem, da so predpomnilniki avtomatični in programer do njih ne more dostopati eksplicitno, uporaba lokalnega spomina na GPE pa je povsem prepuščena programerju. V OpenCL mora tako uporabo lokalnega pomnilnika eksplicitno uporabljati programer, v OpenACC pa njegovo morebitno uporabo nadzira prevajalnik. Tudi tu lahko programer pomaga z direktivami, vendar je to težko, saj je uporaba lokalnega pomnilnika na GPE zelo odvisna od oblike in velikosti gruč, ki se oblikujejo ob zagonu programa na GPE, te pa so odvisne predvsem od prevajalnika.

Ločenost pomnilnikov naprav prinese nekaj problemov pri programiranju GPE in pisanju prevajalnika za OpenACC. Hitrost prenosa podatkov med GPE in CPE je pri programiranju v OpenACC skrita, saj programerju ni potrebno nikjer specificirati točnih podatkov, za katere želi, da se prenesejo na GPE. To pomeni, da mora prevajalnik sam ugotoviti, katere podatke koda potrebuje. Pri tem so prevajalniki redko tako uspešni kot programerji, ker je njihov vpogled v delovanje programa slabši. Programerji morajo biti pazljivi, da prevajalnik ne premakne prevelike količine podatkov v pomnilnik GPE, ker to prinese velike upočasnitve v delovanju programa. Pri programskih jezikih, ki eksplicitno uporabljajo kazalce, je tudi potrebno biti pazljiv, saj pri prenosu na GPE njihova veljavnost preneha.

3.3 Direktive OpenACC

OpenACC deluje prek direktiv. To so ukazi, ki jih vključimo v kodo in s katerimi prevajalniku povemo, kaj in kako želimo, da se izvede na GPE. OpenACC je standardiziran za tri jezike: C, C++ in Fortran. Direktive so v C in C++ implementirane prek predprocesorskih ukazov `#pragma`, v Fortranu pa prek komentarjev s posebno oznako `!$acc`. Ker smo za to delo implementirali OpenACC v tretjem jeziku, bomo tu uporabljali sintakso, ki je bila uporabljena v našem prevajalniku. Ta je zelo podobna kot sintaksa za direktive v C in C++. Oblika direktive je sledeča:

```
#acc ime-direktive [seznam-dolocil]
```

Vsaka direktiva mora obvezno stati sama v eni vrstici. V našem prevajalniku so, tako kot v C in C++, vse direktive pisane z malimi črkami.

Direktiv in njihovih določil je veliko, glavne med njimi so:

```
#acc kernels
#acc parallel
#acc loop
#acc data
```

Z direktivo `kernels` specificiramo področje kode, kjer želimo, da prevajalnik opravi analizo, katere zanke se splača paralelizirati, nato pa jih spremeni v vzporedno kodo. Direktiva `parallel` pove, za katera področja želimo, da se izvedejo na GPE. Direktiva `loop` pove, katere zanke želimo, da se paralelizirajo. Ta direktiva je najbolj uporabna, saj predstavlja jedro funkcionalnosti OpenACC. Nahajati se mora v področju direktive `parallel` in neposredno pred zanko `for`. Lahko pa specificiramo obe hkrati.

```
#acc parallel loop
```

Direktiva `data` se uporablja za pomoč prevajalniku pri prenosu podatkov iz in na GPE. Ima veliko različnih določil, ki prevajalniku povejo, kdaj želimo podatke prenesti med GPE in CPE, ali želimo, da se sploh prenašajo, lahko pa prevajalniku tudi povemo, če so podatki že na GPE.

Poglavje 4

Realizacija OpenACC v Javi

OpenACC je standardiziran za jezike Fortran, C in C++. Pri izbiri višje nivojskega jezika, za katerega smo se odločili izdelati podporo za OpenACC, smo upoštevali nekaj kriterijev. Ta jezik smo poimenovali vhodni jezik.

Zaradi strukture OpenACC naš vhodni jezik potrebuje zanke. Za nekatere funkcijske jezike, ki ne vsebujejo zank, obstajajo knjižnice in orodja, ki kodo prevajajo v ščepce za GPE, vendar je njihov programski vmesnik povsem drugačen kot v OpenACC [30, 29].

Naš izbrani vhodni jezik je potreboval vmesnik za OpenCL. Zaradi predznanja smo za komunikacijo z GPE želeli uporabiti OpenCL. OpenCL je definiran za jezik C++, vendar obstaja ogromno vmesnikov OpenCL, prenesenih v druge jezike.

Najbolj pomembno pa je bilo, da smo znali naš vhodni jezik prevesti v jezik OpenCL C. Čeprav je ta prevedba možna iz velikega števila jezikov, smo se zaradi enostavnosti implementacije odločili, da izberemo jezik iz družine jezikov, ki so podobni jeziku C. Ti kriteriji so nas pripeljali do tega, da smo za izvorni jezik prevajalnika izbrali Javo.

Prednosti Jave pred jeziki, kot so C in Fortran, so:

- podpora za avtomatično čiščenje pomnilnika;
- dobra podpora za generične tipe in

- lažji ter hitrejši razvoj kompleksnih programov.

4.1 Implementacija prevajalnika

Za podporo OpenACC v Javi smo izdelali prevajalnik na nivoju izvorne kode. To pomeni, da smo izvorno kodo, napisano v Javi, skupaj z OpenACC oznakami prevajali v izvorno kodo v Javi, ki je uporabljala knjižnico JOCL za izvajanje ukazov na GPE. To kodo lahko nato posredujemo navadnemu prevajalniku Jave, kot je na primer `javac`, da dobimo končni program. Ko ta program zaženemo, se med tekom prevede tudi del programa, ki je namenjen izvajanju na GPE. Naš prevajalnik je sestavljen iz dveh delov: sprednjega dela (ang. *front end*) in zadnjega dela (ang. *back end*). Sprednji del je namenjen analizi vhodne izvorne kode, zadnji del pa sintezi izhodne izvorne kode.

4.2 Omejitve prevajalnika

Zaradi časovne omejitve tega dela smo se pri implementaciji prevajalnika omejili na majhen del standarda OpenACC. Implementirali smo omejene dele direktiv `loop` in `data`, saj te direktive tvorijo jedro funkcionalnosti OpenACC. Drugih direktiv nismo implementirali, saj bi bila njihova implementacija preveč časovno zahtevna.

Pri implementaciji OpenACC v Javi smo prav tako postavili nekaj strogih omejitev. V blokih, ki so se prevedli v ščepce, ne dopuščamo ne-primitivnih tipov, torej ne omogočamo uporabe razredov in objektov v OpenACC. Prav tako smo omejili uporabo večdimenzionalnih tabel. S tema dvema omejitvama smo se znebili problemov, ki nam bi jih povzročali v Javi sicer skriti kazalci za dostop do objektov. Prav tako v bloku OpenACC ne dovolimo funkcijskih klicev.

Vse zanke, ki jih opremimo z direktivo `loop`, morajo imeti kanonično obliko, ki jo vidimo v sledečem izseku kode.

```
for(int i=0; i</*st. iteracij*/; i++){
    ...
}
```

Med izvajanjem ene iteracije zanke ni dovoljeno spreminjati vrednosti iteracijske spremenljivke.

Vse implementacije direktive `data` morajo stati pred blokom, s katerim določimo življenjsko dobo objektov, ki jih bodo predstavljali v izhodni kodi. Zaradi izogibanja puščanja pomnilnika je prav tako prepovedana uporaba skokov, ki niso del zank, torej `return`, `break` in `continue`. Izjema je ključna beseda `continue` v samih ščepcih, saj jo tam lahko nadomestimo s ključno besedo `return` in s tem ohranimo pomen izvorne kode.

4.3 Sprednji del prevajalnika

Sprednji del prevajalnika je namenjen analizi vhodnega programa. Program odpre datoteko z izvorno kodo in jo preda prevajalniku. Prva dela prevajalnika sta leksikalna in sintaksna analiza. Za obe uporabimo knjižnico ANTLR4.

ANTLR4 (ang. *Another Tool For Language Recognition, version 4*) [35] je orodje, namenjeno ustvarjanju leksikalnih in sintaksnih analizatorjev (ang. *parser generation*). ANTLR je napisan v Javi, je odprtokoden in sposoben ustvarjati analizatorje za velik nabor programskih jezikov (Java, C++, Python ...). Deluje tako, da mu podamo gramatiko (ang. *grammar*), iz katere ustvari izvorno kodo za sintaksni analizator. To izvorno kodo lahko nato vključimo v poljuben projekt in jo prevedemo. Poleg sintaksnih analizatorjev nam ANTLR omogoči tudi ustvarjanje leksikalnih analizatorjev (ang. *lexer*), ki opravljajo leksikalno analizo.

ANTLR za gramatike uporablja poseben jezik, ki omogoča skupno pisanje leksikalnih pravil in gramatike za analizatorja. Za leksikalna pravila uporablja konstrukte, ki so zelo podobni regularnim izrazom UNIX (ang. *UNIX regex*), za gramatike sintaksnih analizatorjev pa uporablja konstrukte, po-

dobne obliki Backus-Naur. V obeh primerih ANTLR omogoča razširitve, ki olajšajo pisanje leksikalnih in sintaksnih pravil.

ANTLR ima na spletu veliko zbirko gramatik za različne programske jezike [36]. Pri razvoju prevajalnika za OpenACC smo zaradi velike kompleksnosti gramatik za Javo začeli z gramatiko za Javo različice 7, objavljeno v njihovi zbirki, ki smo jo priredili za prepoznavanje simbolov in konstruktov OpenACC.

4.3.1 Leksikalna analiza

Prva faza prevajalnika je leksikalna analiza. Vhod v fazo leksikalne analize je niz znakov izvorne kode. Namen leksikalne analize je, da iz niza znakov, ki tvorijo izvorno kodo, dobimo niz osnovnih simbolov jezika (ang. *token*). Ti so določeni z leksikalnimi pravili jezika. Prevajalnik iz vhodnega niza jemlje znake, enega po enega, in gleda, če se njihova zaporedja ujemajo s katerim izmed leksikalnih pravil jezika. Če se ujemajo, prevajalnik ustvari simbol za del vhodnega niza, ki se ujema z leksikalnim pravilom in ga poda naslednji fazi prevajalnika – sintaksni analizi. Pravila leksikalne analize se po navadi opišejo z regularnimi izrazi.

Za leksikalno analizo uporabimo leksikalni analizator, ki ga ustvarimo s pomočjo orodja ANTLR. Orodju ANTLR podamo datoteko z naborom leksikalnih pravil, iz katerih nam ANTLR ustvari izvorno kodo za leksikalni analizator. Vsako leksikalno pravilo je sestavljeno iz imena simbola in izraza, ki pove, kakšen niz znakov je potreben za stvaritev takega simbola. ANTLR jezik za gramatike prav tako omogoča, da ustvarimo fragmente (ang. *fragments*), ki sami po sebi ne tvorijo simbolov jezika, ampak nam pomagajo pri specifikaciji leksikalnih pravil za le-te.

Pomembna funkcionalnost ANTLR leksikalnih analizatorjev je možnost, da spreminjamo način njihovega delovanja. Vsak način delovanja ima svoja leksikalna pravila. Za prehod med načini prav tako uporabljamo leksikalna pravila, ki jim lahko dodamo posebne funkcije, katere spremenijo način delovanja.

```
IDENTIFIER:         Letter LetterOrDigit*;

fragment LetterOrDigit
    : Letter
    | [0-9]
    ;

fragment Letter
    : [a-zA-Z$_]


HASH : '#' -> pushMode(AccMode);

mode AccMode;

ACC             : 'acc';
ACC_PARALLEL        : 'parallel';
ACC_KERNELS        : 'kernels';
ACC_SERIAL        : 'serial';
ACC_LOOP         : 'loop';

ACC_NEWLINE: [\n] -> popMode, channel(HIDDEN);
```

Izsek kode 4.1: Primer leksikalnih pravil

V izseku kode 4.1 vidimo primer leksikalnih pravil našega prevajalnika. Prva tri pravila predstavljajo, kako bi lahko definirali identifikator v jeziku Java. Prvo pravilo pove, da je identifikator sestavljen iz črke, kateri sledi poljubno število črk ali cifer. V ANTLR jeziku velja konvencija, da so vsi osnovni simboli jezika pisani z velikimi tiskanimi črkami. Naslednji dve pravili specificirata fragmenta, ki povesta, kako je v našem jeziku definirana črka in kako števka.

Sledeča pravila prikazujejo, kako lahko prevajalnik premakne leksikalni analizator v drug način delovanja. Ko prevajalnik najde simbol `HASH` (znak #), se premakne v način `AccMode`, katerega sestavljajo vsa pravila, ki sledijo

direktivi `mode AccMode;`. Ta način je namenjen leksikalni analizi direktiv OpenACC. Ko prevajalnik najde znak za novo vrstico, se z ukazom `popMode` vrne v navadni način delovanja.

4.3.2 Sintaksna analiza

Druga faza prevajalnika je sintaksna analiza. Nalogo sintaksne analize opravlja sintaksni analizator (ang. *syntax analyzer, parser*). Vhod v sintaksni analizator so osnovni simboli jezika, ki jih tvori leksikalna analiza. Rezultat sintaksne analize je sintaksno drevo (ang. *syntax tree, parse tree*). Sintaksna pravila jezikov se po navadi podajo kot kontekstno neodvisne gramatike (KNG). Končni simboli (terminali) KNG so simboli leksikalne analize, spremenljivke (neterminali) pa predstavljajo pravila našega jezika. Sintaksno drevo, ki ga tvori sintaksni analizator, je drevo izpeljave KNG našega jezika.

Za sintaksno analizo uporabimo sintaksni analizator, ki nam ga ustvari ANTLR. Analizator, ki ga ustvari ANTLR, je tipa LL(*) [37]. LL(k) analizatorji so analizatorji za KNG, ki za določen vhod zmeraj ustvarijo skrajno levo izpeljavo (ang. *left-most derivation*) in imajo vhodno okno dolžine k. LL(*) je variacija sintaksnih analizatorjev LL(k), ki nima omejenega vpogleda vnaprej in je bila razvita posebej za prevajalnike ANTLR.

Sintaksni analizator ustvarimo z datoteko, v kateri specificiramo sintaksna pravila našega jezika. V izseku kode 4.2 vidimo strukturo teh pravil. Prvo pravilo pove, kakšni so v našem jeziku osnovni stavki (pravilo tu ni predstavljeno v celoti). Vrstico, ki se začne s simbolom tipa `HASH`, smo vrinili in predstavlja pravilo, ki določa strukturo OpenACC direktiv. Sledeča pravila opisujejo, kakšne direktive dopuščamo v naši razširitvi Jave.

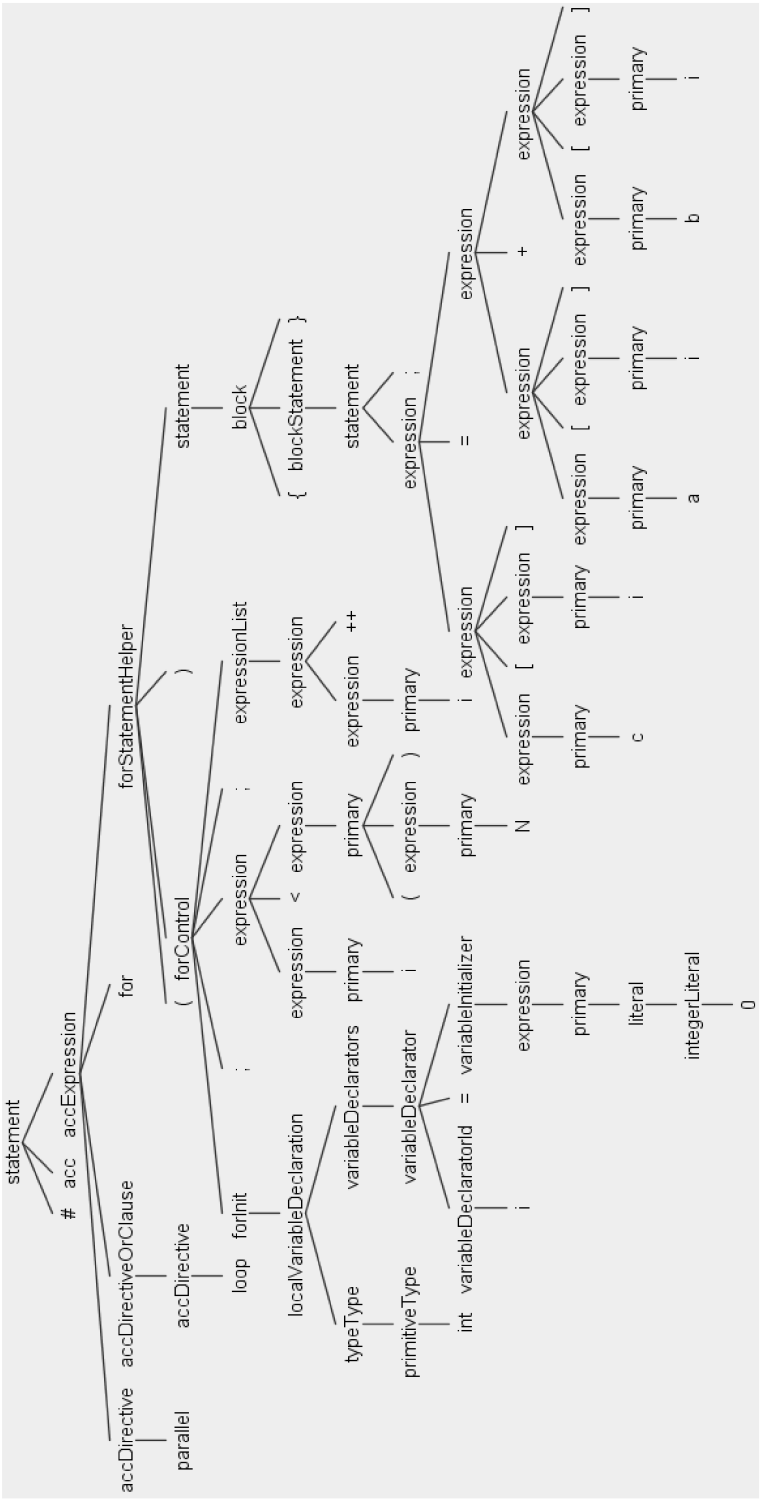
Z datotekami, ki specificirajo leksikalno in sintaksno strukturo Jave z našimi OpenACC razširitvami, ANTLR ustvari izvorno kodo za leksikalno in sintaksno analizo. To lahko ustvari za velik nabor jezikov, kot so Java, C++ in Python. Mi smo se odločili za Javo.

S to kodo lahko začnemo analizirati izvorno kodo, ki jo bo prevajal naš prevajalnik. ANTLR ponuja orodje Grun, ki omogoča vizualizacijo sinta-

```
statement
    : IF parExpression statement (ELSE statement)?
    | FOR forStatementHelper
    | SEMI
    | HASH ACC accExpression
    ... // ostala pravila za stavke
    ;

accExpression
    : accDirective accDirectiveOrClause* block
    | accDirective accDirectiveOrClause* FOR
       forStatementHelper
    ;

accDirectiveOrClause
    : accDirective
    | accClause
    ;

accDirective
    : ACC_PARALLEL
    | ACC_LOOP
    | ACC_DATA
    ;
```

Izsek kode 4.2: Primer sintaksnih pravil

ksnega drevesa izvorne kode. Na sliki 4.1 vidimo primer, kakšen je videti izsek tega drevesa na preprostem primeru zanke for, ki smo jo videli v izseku kode 1.2. Drevo se začne z neterminalom *statement*, ki se deli na dva terminala `#` in `acc` ter en neterminal *accExpression*. Ta se naprej deli na štiri dele, ki tvorijo preostanek OpenACC direktive in zanke `for`.



Slika 4.1: Sintaksno drevo direktive OpenACC int preproste zanke `for`.

4.3.3 Semantična analiza

Semantična analiza je del prevajalnika, ki je v enostavnih prevajalnikih namenjen predvsem razreševanju imen in analizi tipov. Po navadi po opravljeni sintaksni analizi dobljeno drevo pretvorimo v abstraktno sintaksno drevo (AST), ki olajša nadaljnjo analizo. Zaradi omejenega obsega prevajalnika se za to nismo odločili, saj smo se ukvarjali predvsem z OpenACC direktivami.

V sklopu semantične analize smo se opirali na programski vzorec, imenovan *obiskovalec*. ANTLR pri ustvarjanju sintaksnega analizatorja omogoča tudi stvaritev skeleta za obiskovalca, ki ga uporabljamo na sintaksnem drevesu. S tem skeletom lahko implementiramo svoje obiskovalce, ki jih uporabljamo za analizo drevesa. Rezultati te analize so prilastki (ang. *attributes*), ki jih obesimo na posamezne liste ali vozlišča drevesa. V sklopu semantične analize smo implementirali tri obiskovalce.

Prvi obiskovalec je namenjen identifikaciji programskih obsegov (ang. *scope*), iskanju in tipizaciji spremenljivk. Za naš prevajalnik nas je predvsem zanimalo, ali je določena spremenljivka primitivnega ali kompleksnega tipa. Primitivni tipi so celoštevilski tipi in tipi s plavajočo vejico. Prav tako nas je zanimalo, ali se deklaracija spremenljivke nahaja v obsegu bloka OpenACC ali ne, in v katerem programskem obsegu se nahaja.

Drugi obiskovalec je namenjen predvsem iskanju uporab spremenljivk v programu, njihovi tipizaciji in ugotavljanju, kje se uporabljajo v relaciji do blokov OpenACC. Pomembne so predvsem uporabe znotraj OpenACC blokov, saj smo s tem kasneje določili, katere spremenljivke je potrebno prepisati na GPE.

Tretji obiskovalec je namenjen identifikaciji in analizi blokov OpenACC. Ta obiskovalec identificira tipe OpenACC blokov, določi njihove parametre in vhode ter izhode ščepcev, ki jih tvorijo. Vhodi in izhodi ščepcev se določijo glede na uporabo spremenljivk v bloku. Zato je potrebno ugotoviti, kje so deklarirani in kako se uporabljajo v ščepcu. Bralni dostopi v bloku pomenijo, da moramo podatke prenesti s CPE na GPE. Pisalni dostopi do spremenljivke lahko pomenijo, da je potrebno podatke prenesti z GPE, vendar se lahko

zgodi, da so taki prenosi odvečni, saj ni nujno, da so podatki uporabljeni še kje na CPE po izvedbi ščepca. Za zmanjševanje števila odvečnih prenosov bi lahko uporabili analizo aktivnosti spremenljivk (ang. *liveness analysis*), ki pa je nismo uspeli implementirati.

4.4 Zadnji del prevajalnika

4.4.1 OpenCL

Izhodna koda, ki jo ustvari zadnji del prevajalnika, sloni na uporabi ogrodja OpenCL za komunikacijo z GPE in ostalimi napravami. OpenCL je programsko ogrodje, namenjeno programiranju heterogenih sistemov, ki vsebujejo CPE, GPE ali ostale pospeševalnike, lahko pa tudi procesorje za diskretno procesiranje signalov (ang. *digital signal processor – DSP*) in programirljiva vezja (ang. *field programmable gate arrays – FGPA*). Uporaba teh naprav je omogočena preko enotnega vmesnika. Implementacije OpenCL obstajajo za velik nabor arhitektur in naprav. To je velika prednost OpenCL pred ostalimi vmesniki za GPE, saj omogoča visoko stopnjo prenosljivosti med napravami in enoten model programiranja, ne glede na arhitekturo gosta ali gostitelja.

OpenCL je sestavljen iz več delov. Prvi del je programski vmesnik OpenCL (ang. *OpenCL API*). Preko tega vmesnika upravljamo izvajanje zunanjih naprav z gostitelja. Sestavljen je iz množice funkcij in objektov, s katerimi lahko nadziramo potek računanja in prenosa podatkov med CPE ter zunanjimi napravami. S tem delom zunanje naprave pripravimo na računanje, jim pripravimo podatke, podamo ukaze za začetek računanja in prebiramo rezultate računanja.

Drugi del OpenCL je koda, ki se izvaja na zunanjih napravah. To kodo pišemo v jeziku *OpenCL C*. To je jezik, ki je osnovan na jeziku C, verziji C99. Jezik *OpenCL C* je prirejen tako, da omogoča lažje programiranje naprav, ki izvajajo programe vzporedno. Jezik ima vgrajene funkcije za ugotavljanje identifikacijskega števila niti, njene skupine in njenega števila znotraj skupine. Jezik ima vgrajen tudi velik nabor matematičnih funkcij in

ima podporo za vektorske spremenljivke ter vektorske operacije.

Ključni del OpenCL implementacije je gonilnik naprave, na kateri želimo programe izvajati. Gonilniki vsebujejo prevajalnik za OpenCL C, ki prevaja izvorno kodo v strojno kodo zunanje naprave. Prav tako tolmačijo ukaze iz OpenCL vmesnika v ukaze, ki jih zunanja naprava razume.

OpenCL vmesnik vsebuje veliko število objektov in funkcij, vendar nas zanimajo le nekatere. Preden lahko vzpostavimo komunikacijo z napravo, se moramo odločiti, preko katere OpenCL platforme (`cl_platform`) bomo to storili. Vsaka OpenCL platforma je implementacija OpenCL, ki jo najdemo na našem sistemu. Po navadi vsaka platforma predstavlja implementacijo ene verzije OpenCL, enega proizvajalca naprav za vse svoje naprave. Nato iz platforme izberemo eno izmed naprav (`cl_device`), ki je na voljo v platformi, in zanjo ustvarimo kontekst OpenCL (`cl_context`). Kontekst je objekt OpenCL, prek katerega komuniciramo z eno ali več napravami. Nanj je navezana ukazna vrsta (`cl_command_queue`), pomnilniški objekti (`cl_mem`), programski objekti(`cl_program`) in ostali komunikacijski ter računski objekti.

Po stvaritvi konteksta moramo najprej ustvariti ukazno vrsto. Ukazna vrsta je objekt, v katerega pošiljamo ukaze, ti pa se predajo naprej, preko gonilnika, na zunanjo napravo. Nato moramo ustvariti programski objekt, ki vsebuje izvorno kodo, ki se bo izvajala na napravi. Prav tako pri stvaritvi programskega objekta gonilniku GPE tudi podamo ukaz, da izvorno kodo prevede. V programski objekt tako shranimo tudi podatke o tej strojni kodi, ki je na napravi. Preden lahko začnemo z izvajanjem, moramo določiti vstopno točko v naš program. To storimo z objektom, ki predstavlja naš ščepec. V ta objekt tudi podamo argumente ščepca. Nato lahko objekt ščepca potisnemo v ukazno vrsto, ki požene izvajanje ščepca na zunanji napravi.

Za prenos podatkov prav tako potrebujemo posebne OpenCL objekte, ki jih imenujemo pomnilniški objekti. Ob njihovi stvaritvi na zunanji napravi gonilnik rezervira zahtevan kos pomnilnika, v objekt pa shranimo kazalec nanj. Ta kazalec lahko predamo ščepcu kot argument. Za prepis podatkov na napravo moramo v ukazno vrsto podati ukaz, ki pove, katero spremenljivko

želimo prepisati na napravo, njeno velikost in kam jo želimo prepisati na napravi. Enako velja za pridobivanje podatkov iz naprave.

Na koncu izvajanja moramo poskrbeti, da vse te objekte odstranimo in počistimo z zunanje naprave, saj lahko v nasprotnem primeru pride do nepravilnega delovanja zunanje naprave.

Vsakič, ko ščepec pošljemo na GPE, moramo specificirati N-dimenzionalni računski prostor (ang. *ND range*). Ta pove GPE velikost problema, ki ga rešujemo. Računski prostor ima lahko največ tri dimenzije, vsaka točka v tem prostoru pa predstavlja eno nit, ki bo izvedla ščepec.

4.4.2 Sinteza izhodne kode

Zadnji del prevajalnika je namenjen sintezi izhodne kode. Sestavljen je iz dveh delov, sinteze OpenCL C kode in sinteze Javanske kode. Za razliko od sprednjega dela, zadnji del prevajalnika ne deluje na drevesih, ampak na sekvenčnem toku simbolov jezika. V sklopu semantične analize smo nekatere od izračunanih prilastkov pripisali listom drevesa. Te prilastke uporabimo pri sintezi izhodne kode, da vhodno kodo spremenimo v izhodno.

Sinteza izhodne kode poteka tako, da prevajalnik bere tok simbolov vhodne kode in iz njih sintetizira izhodno kodo. Če simboli nimajo prilastkov, jih preprosto prepiše v izhodno kodo. Ko naleti na simbol s prilastki, jih pregleda in se glede na njih odloči, ali bo spremenil vsebino programa. Glavne točke, kjer se to zgodi, so simboli `#`, { in }, torej direktive OpenACC ter vhodi in izhodi iz programskih blokov.

Ko tok pride do simbola `#`, vemo, da smo prišli do direktive OpenACC. Ta simbol nosi prilastke, ki povejo, za kakšno direktivo gre. Prevajalnik direktiv nikoli ne prepiše v izhodno kodo. Če gre za direktivo `data`, potem se prilastek, ki opisuje to podatkovno področje, vstavi v seznam trenutno aktivnih podatkovnih področij. Nato prevajalnik izpusti simbole, dokler ne najde znaka {, ki pomeni začetek novega bloka. Na tej točki mora prevajalnik vstaviti kodo za začetek podatkovnega področja. Začetek podatkovnega področja vsebuje ustvarjanje pomnilniških objektov in rezervacije pomnilnika

na GPE. To vidimo izseku kode 4.3.

```
memPointers[2] = Pointer.to(a);
memObjects[2] = CL.clCreateBuffer(context,CL.
    CL_MEM_READ_WRITE | CL.CL_MEM_COPY_HOST_PTR,
    Sizeof.cl_float * a.length, memPointers[2],
    null);
```

Izsek kode 4.3: Stvaritev pomnilniških objektov

Vsakič, ko ustvarimo pomnilniški objekt in rezerviramo pomnilnik na napravi, ta objekt uvrstimo na seznam. Tako lahko poskrbimo, da pri spremenljivki, za katero bi morali ustvariti pomnilniški objekt, ta pa že ima rezerviran pomnilnik na napravi, ne rezerviramo pomnilnika še enkrat.

Ko prevajalnik konča z inicializacijo podatkovnega področja, nadaljuje z normalnim branjem toka s simboli, vendar čaka na ustrezen simbol }, ki konča trenutno podatkovno področje. Ta simbol sproži zaključek ustreznega podatkovnega področja. Konec podatkovnega področja vsebuje dva koraka. Najprej z naprave prepišemo vse podatke, ki jih potrebujemo, na primer tiste, ki so zapisani v določilih `copyout`. Za tem pa je potrebno še sprostiti vse pomnilniške objekte, ustvarjene na začetku podatkovnega področja. Potrebno je poskrbeti, da se pomnilniški objekti pobrišejo na koncu tistega podatkovnega področja, kjer so bili ustvarjeni, in ne prej. V izseku kode 4.4 vidimo, katere ukaze v izhodno kodo vrine naš prevajalnik.

```
CL.clEnqueueReadBuffer(commandQueue, memObjects
    [2], CL.CL_TRUE, 0, a.length * Sizeof.cl_float,
     memPointers[2], 0, null, null);
CL.clReleaseMemObject(memObjects[2]);
```

Izsek kode 4.4: Brisanje pomnilniških objektov

Če simbolu `#` sledi direktiva `parallel loop`, začne prevajalnik sintetizirati kodo za zagon ščepca. Kodo direktive in zanke `for` izbrišemo iz izhodne kode ter jo nadomeščamo z ukazi OpenCL. Najprej ustvarimo kodo ščepca. Kako to storimo, je opisano kasneje. Kot pri direktivi `data`, imajo tudi ščepci vhodne in izhodne podatke, zato je potrebno, da se najprej ustvarijo

pomnilniški objekti. To se zgodi enako kot pri direktivi `data`. Poleg teh spremenljivk moramo posebej obravnavati redukcijske spremenljivke. Redukcijo opravimo tako, da vsaka nit ščepca izračuna svojo zasebno verzijo rezultata, ki ga zapiše v posebno tabelo. To tabelo nato prepišemo nazaj na gosta, ki to tabelo reducira. To pomeni, da moramo na tem mestu ustvariti pomnilniški objekt za izhodno tabelo redukcije. Ker je redukcij lahko več, to storimo za vsako redukcijsko spremenljivko.

V izsekih kode 4.5 in 4.5 vidimo naslednje korake izhodne kode. Najprej kodo ščepca prevedemo, ustvarimo objekt, ki ga predstavlja na gostu, in mu določimo argumente.

```
cl_program program = CL.clCreateProgramWithSource(
    context, 1, new String[]{ programSource }, null
    , null);
CL.clBuildProgram(program, 0, null, null, null,
    null);
cl_kernel kernel = CL.clCreateKernel(program, "
    kernel$0", null);

CL.clSetKernelArg(kernel, 0, Sizeof.cl_mem,
    Pointer.to(memObjects[0]));
...
```

Izsek kode 4.5: Priprava ščepca

Nato določimo število dimenzij in število niti v našem računskem prostoru. Te smo pridobili med semantično analizo. Nato ščepec zaženemo in z naprave prepišemo rezultate.

```
long global_work_size[] = new long[]{(N), };

CL.clEnqueueNDRangeKernel(commandQueue, kernel, 1,
     null, global_work_size, null, 0, null, null);
CL.clEnqueueReadBuffer(...);
```

Izsek kode 4.6: Izvedba ščepca

Če je potrebno opraviti redukcijo, jo opravimo tu. Za redukcijo je na gostu

potrebna le `for` zanka, ki rezultate v tabeli združi z originalno spremenljivko. To vidimo v izseku kode 4.7.

```
CL.clEnqueueReadBuffer(..., b_reduction_mem_object
    , ...);

for(int _reduction_i=0; _reduction_i < (N);
    _reduction_i++) {
     b = b +  b_reduction_buffer[_reduction_i];
}
```

Izsek kode 4.7: Koda za izvedbo redukcije

Zadnja stvar, ki jo je potrebno storiti, je še počistiti vse objekte OpenCL, ki smo jih ustvarili za ta ščepec, in sprostiti spomin, ki smo ga zasedli na GPE. To vidimo v izseku kode 4.8.

```
CL.clReleaseMemObject(b_reduction_mem_object);
CL.clReleaseKernel(kernel);
CL.clReleaseProgram(program);
```

Izsek kode 4.8: Čiščenje objektov OpenCL

Ob branju vhodnega toka čakamo tudi na posebne prilastke, ki označujejo začetek in konec funkcije `main`. To sta vstopna in izstopna točka našega celotnega programa. Na začetek funkcije `main` vstavimo inicializacijo nekaterih OpenCL objektov. To so objekti, katere si delijo vsi ščepci, ki se izvajajo v izhodnem programu. Inicializacija najprej najde pravo platformo OpenCL ter napravo in zanjo ustvari kontekst. V tem kontekstu za izbrano napravo ustvari ukazno vrsto. Ker si lahko ščepci delijo tudi pomnilniške objekte, je prostor za njih potrebno rezervirati med inicializacijo, sami pomnilniški objekti pa se ustvarijo med tekom programa.

Za sintezo kode ščepca je bilo potrebno pretvoriti izvorno kodo v Javi v OpenCL C. Tu nam dobro služi izbira Jave za izvorni jezik prevajalnika zaradi njene podobnosti z jezikom OpenCL C. Prav tako nam dobro služi odločitev, da se za pri ščepcih omejimo le na primitivne tipe, ker to pomeni, da nam ni treba prevajati bolj kompleksnih objektov v kodo ščepca.

Pretvorbo v OpenCL C začnemo z deklaracijo vhodne funkcije ščepca. Pri deklaraciji argumentov ščepca je potrebno paziti, da so deklarirani v enakem vrstnem redu, kot jih nato nastavimo v kodi za gosta. Na začetku ščepca najprej ustvarimo iteracijske spremenljivke, ki jim dodelimo položaj niti v računskem prostoru ščepca. Če mora ščepec opraviti redukcijo, je tu potrebno opraviti pripravo spremenljivk. Če redukcijska spremenljivka ni med vhodi v ščepec, jo deklariramo in ji pripišemo nevtralno vrednost glede na tip redukcije, npr. če elemente seštevamo, je nevtralna vrednost 0, če jih množimo, pa je ta vrednost 1. Prav tako naredimo kopijo te spremenljivke, da si zapomnimo njeno začetno vrednost.

Jedro ščepca tvori koda, ki je napisana v Javi. Zaradi podobnosti v sintaksi med Javo in C je večinoma potrebno le simbole iz vhoda prepisati v kodo ščepca. Določene simbole, kot so `byte` in `final`, je potrebno spremeniti v njihove ekvivalente jezika C. Prav tako je potrebno posebej obravnavati vse ključne besede `continue`. Če se nahajajo v zanki, v ščepcu ostanejo nespremenjene, sicer jih moramo spremeniti v ključno besedo `return`. Če lahko najdemo ekvivalentno funkcijo, potem matematične funkcije, ki jih v Javi najdemo v razredu `Math`, pretvorimo v vgrajene matematične funkcije OpenCL C.

Kakšen je preveden ščepec, vidimo v izseku kode 4.9. Ščepec opravi preprosto seštevanje vektorja z redukcijo.

Če ima ščepec redukcijo, na koncu ščepca zapišemo redukcijsko spremenljivko v redukcijsko tabelo. Pri tem je potrebno odšteti začetno vrednost spremenljivke. Če bi ta korak izpustili, bi vsaka nit v končni rezultat prištela začetno vrednost spremenljivke, namesto da bi se ta prištela le enkrat. Inverzna operacija za seštevanje je odštevanje, za množenje je operacija deljenje, za bitno operacijo xor pa je inverzna operacija kar xor.

```
__kernel void kernel$1(float b, __global float* a,
    int N, __global float* b_reduction_array)
{
    int i = get_global_id(0);
    float b_initial = b;
```

```
    {
        b += a [ i ] ;
    }
    b_reduction_array[i] = b - b_initial;
}
```

Izsek kode 4.9: OpenCL C koda ščepca, ki opravi redukcijo vektorja

4.5 Implementirane funkcionalnosti

Glavne implementirane funkcionalnosti OpenACC so sledeče:

1. Direktiva `parallel` je implementirana delno. Implementirana je njena najbolj pomembna funkcionalnost, ki je direktiva `loop`. Druge funkcionalnosti direktive `parallel`, ki je redundantno vzporedno izvajanje na GPE, nismo implementirali zaradi njene omejene uporabnosti.

   (a) Glavna funkcionalnost direktive `loop`, paralelizacija zank, je implementirana, prav tako je implementirano implicitno prepisovanje podatkov, ki jih potrebujejo ščepci, ustvarjeni s pomočjo te direktive. Pri direktivi `loop` smo prav tako implementirali nekaj pomembnih določil.

   (b) Določilo `reduction` je določilo, ki omogoča izvajanje redukcij na koncu ščepcev. Redukcija je operacija, pri kateri tabelo rezultatov preko specificirane operacije združimo v skalar, ki predstavlja končni rezultat. Redukcije so implementirane tako, da vsaka nit delne rezultate zapiše v tabelo, ki se nato prenese na CPE, tam pa se na njej izvede veriga operacij, ki opravi redukcijo. Podprte operacije za redukcijo so seštevanje, množenje, resničnostni operaciji *in* ter *ali*, bitne operacije *in*, *ali* ter *xor* in operaciji *max* in *min*.

   (c) Določilo `collapse` je določilo direktive `loop`, ki omogoča združitev več zank v en ščepec. Implementiran je tako, da vsaka zanka, ki

jo združujemo v ščepcu, predstavlja drugo dimenzijo računskega prostora OpenCL.

2. Druga pomembna direktiva, ki smo jo implementirali, je direktiva `data`. Ta omogoča, da podatke prenašamo na in z GPE tudi eksplicitno, na mestih, kjer to želimo mi, ne le tik pred začetkom izvajanja ščepca. Implementirana je nekoliko drugače kot v standardu, saj mora zmeraj stati pred programskim blokom in ne more stati prosto v kodi. Na koncu bloka, pred katerim stoji, prevajalnik poskrbi, da se pomnilniški objekti OpenCL počistijo.

3. Direktiva `data` za delovanje uporablja tri implementirana določila: `copyin`, `copyout` in `create`. `copyin` na začetku bloka na napravi rezervira blok pomnilnika, ki ga spremenljivka potrebuje, in vanj prepiše njeno vsebino. Določilo `create` na začetku bloka le rezervira prostor za spremenljivko na napravi, vanj pa nič ne prepiše. To je uporabno predvsem takrat, kadar želimo na GPE predhodno ustvariti prostor za zapis rezultatov, nočemo pa, da se vanj kar koli prepisuje. Določilo `copyout` na koncu bloka z naprave prepiše vrednost spremenljivke nazaj na CPE.

Poglavje 5

Meritve in rezultati

Rezultat našega dela je prevajalnik, ki uspešno prevede javansko kodo, označeno z oznakami OpenACC, v javansko kodo, ki uporablja JOCL za prenos računanja s CPE na GPE. Pri vrednotenju prevajalnika nas zanimata dve značilnosti: nabor funkcionalnosti prevajalnika in hitrost kode, ki jo prevajalnik ustvari.

5.1 Nabor funkcionalnosti

Za pregled, katere funkcionalnosti delujejo, smo prevedli del testov OpenACC EPCC (Edinburgh Parallel Computing Centre)[19] v Javo. Od osnovnih 16 testov je bilo v Javo nemogoče prevesti štiri od njih, od ostalih pa jih nepravilno deluje šest. Dva zaradi pomanjkanja določila `if`, dva zaradi odločitve, da prevajalnik ne bo podpiral skalarjev kot izhodov iz ščepcev, preostali pa zaradi nepopolne podpore direktivama `kernels` in `parallel`. Vsi nedelujoči testi niso delovali zaradi odločitve, da funkcionalnosti, ki jih testirajo, niso zanimive za večino programerjev, in smo jih zato izpustili.

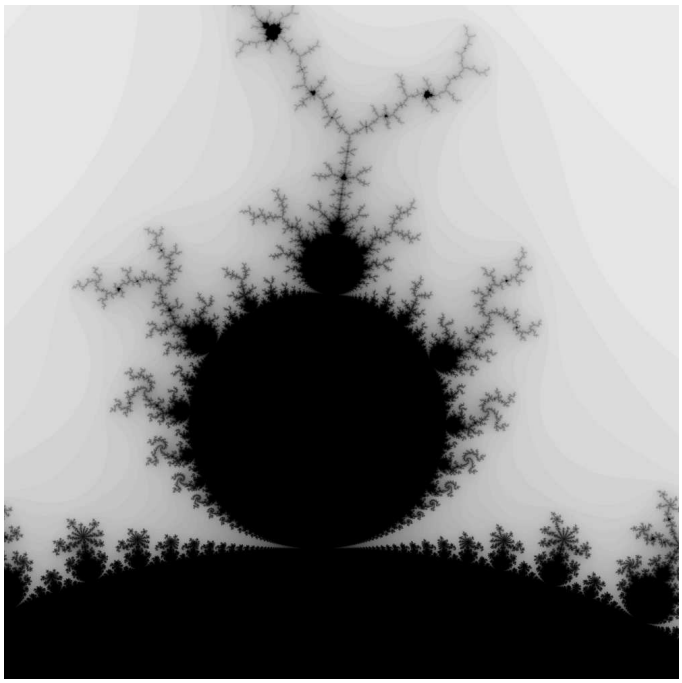
5.2 Meritve hitrosti izvajanja

Za testiranje hitrosti računanja smo izbrali dva večja problema, ju implementirali in ju z našim prevajalnikom prevedli. Problema sta morala biti

ne-trivialna, sicer bi testirali samo prenose podatkov med CPE in GPE namesto hitrosti računanja. Za problema smo izbrali računanje Mandelbrotove množice in simulacijo črede.

Meritve smo izvajali na CPE Intel Core 2 Duo E8400 in GPE Nvidia GeForce 9600 GT. Oba testa smo implementirali v Javi in C++, v Javi z direktivami OpenACC in v C++ s pomočjo OpenCL.

5.2.1 Mandelbrotova množica



Slika 5.1: Del Mandelbrotove množice

Prvi problem je bil računanje Mandelbrotove množice. Mandelbrotova množica je množica števil c v kompleksni ravnini, za katera velja, da zaporedje, podano kot $z_{n+1} = z_n^2 + c$ kjer $n \in \mathbb{N}$ in $z_0 = 0$, ostane omejeno. Algoritem za izračun, ali število c pripada množici ali ne, vidimo v algoritmu 1.

Za naš prvi test smo razdelili kompleksno ravnino, kjer računamo pripadnost množici, na $N \times N$ točk. Za vsako točko poženemo algoritem 1.

Algoritem 1 Izračun pripadnosti Mandelbrotovi množici

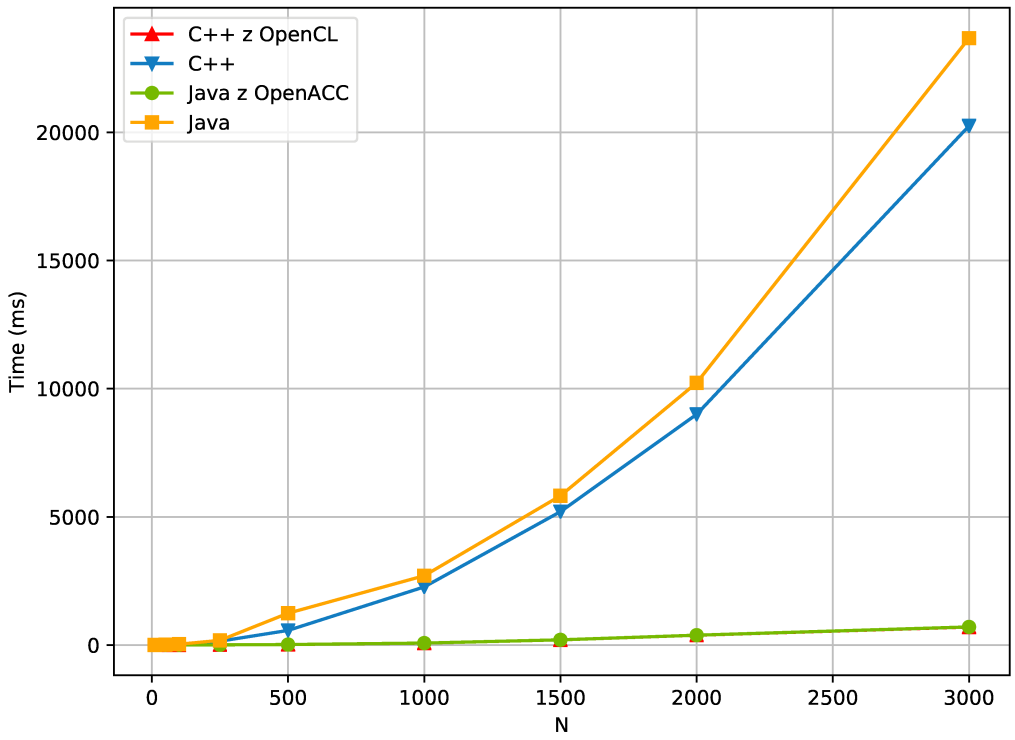
```
z ← 0
iter ← 0
for iter < število_iteracij do
    z ← z^2 + c
    if |z| > 2 then
        return true
    end if
end for
return false
```

Izračun pripadnosti ene točke ne vpliva na izračun ostalih točk, torej je problem trivialno deljiv med niti.

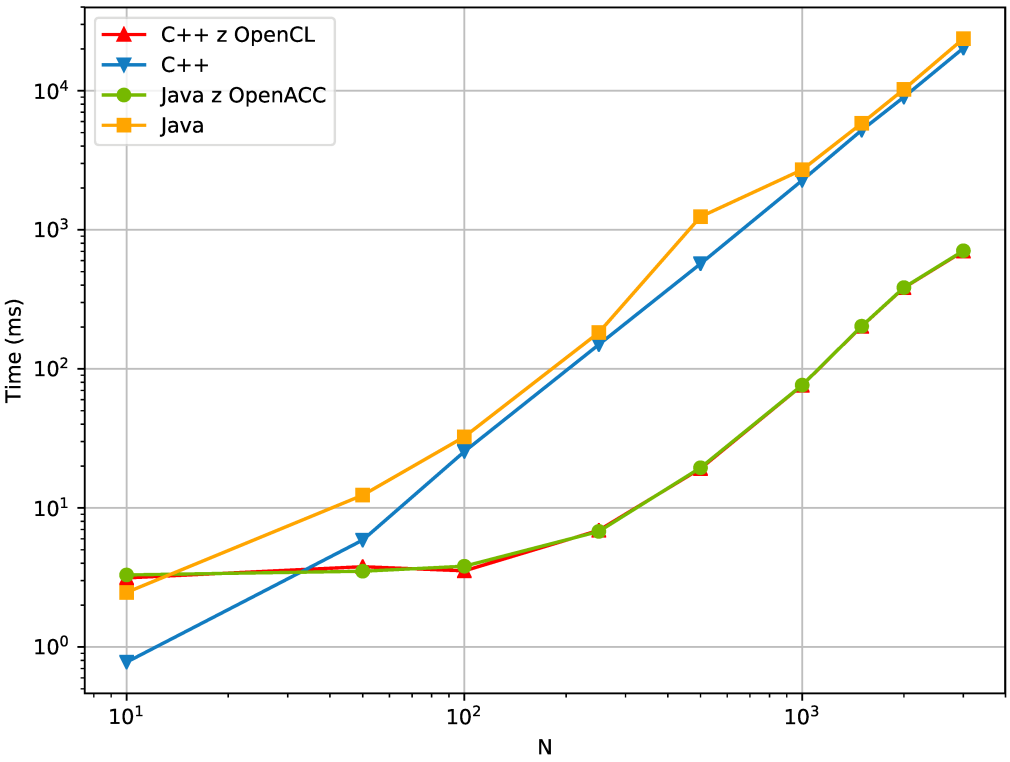
Rezultate testov vidimo v tabeli 5.1, izris tabele pa na sliki 5.2. V tabeli 5.3 pa vidimo tudi izrisan log-log graf, ki bolje prikazuje razlike med meritvami na celotnem razponu merjenja. Iz log-log grafa prav tako lažje vidimo časovno kompleksnost primera.

N	Java	C++	Java z OpenACC	C++ z OpenCL
10	2,469 ms	0,778 ms	3,297 ms	3,139 ms
50	12,352 ms	5,873 ms	3,506 ms	3,780 ms
100	32,493 ms	25,438 ms	3,810 ms	3,528 ms
250	182,438 ms	149,445 ms	6,777 ms	6,902 ms
500	1240,736 ms	569,85 ms	19,450 ms	19,183 ms
1000	2703,823 ms	2267,41 ms	76,441 ms	76,236 ms
1500	5822,582 ms	5202,57 ms	202,689 ms	202,297 ms
2000	10229,576 ms	8999,66 ms	384,278 ms	382,739 ms
3000	23674,366 ms	20251,1 ms	706,5976 ms	701,888 ms

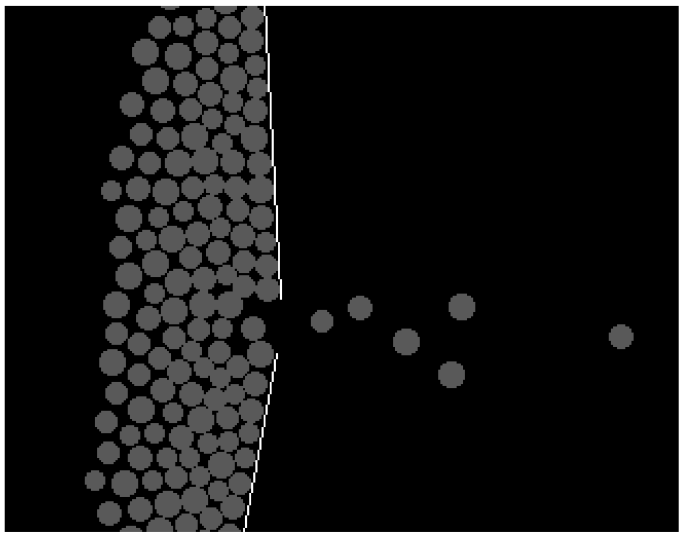
Tabela 5.1: Časi računanja $N \times N$ točk Mandelbrotove množice



Slika 5.2: Časi računanja $N \times N$ točk Mandelbrotove množice. Krivulji OpenCL in OpenACC se prekrivata.



Slika 5.3: Log-log izris časov računanja $N \times N$ točk Mandelbrotove množice.
Krivulji OpenCL in OpenACC se prekrivata.



Slika 5.4: Simulacija čred

5.2.2 Simulacija čred

Drugi testni primer je bila simulacija čred, ker smo želeli testirati prevajalnik na realnem problemu. Simulacija čred je simulacija človeškega ali živalskega obnašanja. Simulacija čred se uporablja predvsem pri načrtovanju prostorov in dogodkov, kjer je predvideno večje število ljudi. Simulacije pomagajo načrtovalcem predvideti nevarna področja zgradb, kjer ob paniki obstaja nevarnost, da ljudi poteptajo ali stisnejo. Simulacija čred je bila narejena na podlagi modela, ki so ga predstavili Helbing, Illés in Tamas [38]. V simulaciji imamo veliko število agentov, ki poskušajo doseči svoj cilj, pri tem pa jih ovirajo drugi agenti. Cilj agenta je, da pride do določene točke.

Model simulacije črede je opisan z enačbama (5.1) in (5.2). V simulaciji imamo N agentov i, z maso m_i, hitrostjo v_i, ki imajo želeno hitrost v_i^0 v smeri $\mathbf{e}_i^0$. Vsak od njih želi doseči to hitrost z neko karakteristično hitrostjo τ_i. Vsak agent je v simulaciji predstavljen kot krog s polmerom r_i.

Delovanje zunanjih in notranjih sil na agente simulacije opisuje enačba

$$m_i \frac{d\mathbf{v}_i}{dt} = m_i \frac{v_i^0(t)\mathbf{e}_i^0(t) - v_i(t)}{\tau_i} + \sum_{j(\neq i)} \mathbf{f}_{ij} + \sum_{W} \mathbf{f}_{iW} \tag{5.1}$$

Levi del enačbe (5.1) predstavlja spremembo hitrosti posameznega agenta.

Desni del enačbe je razdeljen na tri dele. Prvi del predstavlja željo agenta, da se premika v v smeri $\mathbf{e}_i^0$ s hitrostjo v_i^0. Če se premika počasneje od želene hitrosti, agent pospešuje, če pa se premika hitreje od nje, pa se ustavlja. Drugi del enačbe predstavlja interakcije med agenti. Tretji del enačbe predstavlja interakcije med agenti in zidovi W simulacije. Ker v našem testu nismo uporabljali zidov, je ta del ničeln.

Podroben opis ene interakcije med agentoma i in j predstavlja enačba

$$\mathbf{f}_{ij} = \left(A_i \exp\left(\frac{r_{ij} - d_{ij}}{B_i} \right) + kg(r_{ij} - d_{ij}) \right) \mathbf{n}_{ij} + \kappa g(r_{ij} - d_{ij}) \Delta v_{ij}^t \mathbf{t}_{ij} \quad (5.2)$$

Ta enačba predstavlja tri dele interakcije agentov. $\mathbf{n}_{ij}$ predstavlja enotski vektor v smeri interakcije agentov od j proti i, $\mathbf{t}_{ij}$ pa predstavlja enotski vektor, ki je ortogonalen na in $\mathbf{n}_{ij}$. Ta predstavlja smerni vektor tangencialne sile interakcije med agentoma.

Člen $A_i \exp\big((r_{ij} - d_{ij})/B_i\big)$ enačbe (5.2) deluje kot šibka odbojna sila med agenti. Predstavlja tendenco ljudi, da se želijo med gibanjem nekoliko odmakniti od drugih ljudi. A_i in B_i sta konstanti. r_{ij} je seštevek polmerov agentov i in j, d_{ij} pa je razdalja med središčema dveh agentov. $r_{ij} - d_{ij}$ je torej pozitivna, kadar se agenta dotikata, in negativna, kadar se ne.

Člen $kg(r_{ij} - d_{ij})$ je fizična interakcija med agenti, ki predstavlja dejanske kontakte med ljudmi. k predstavlja odbojni faktor agentov in je konstanta. Funkcija $g(x)$ je ničelna, če se agenta ne dotikata, sicer pa je njena vrednost x. To pomeni, da se ta interakcija upošteva le takrat, ko se agenta dotikata. Konstanta k je dovolj velika, da kadar se agenta dotikata, ta člen enačbe usmerja agenta in je dosti močnejši od prvega člena enačbe.

Člen $\kappa g(r_{ij} - d_{ij})$ predstavlja drsno silo med agenti, kar predstavlja umikanje ljudi, kadar pridejo v stik z drugimi. κ je karakteristična konstanta te sile, Δv_{ij}^t pa predstavlja tangencialno razliko hitrosti agentov. Ta člen se upošteva le, ko se agenti dotikajo, in simulira drsenje ljudi mimo drugih v gneči.

Simulacija deluje tako, da v enačbi (5.1) diskretiziramo časovni prostor in nato simuliramo dogajanje v kratkih časovnih korakih. Za vsako sekundo

simulacijskega časa želimo narediti čim več časovnih korakov zato, da izboljšamo natančnost in da se izognemo napakam v simulaciji. Napake, ki se lahko zgodijo zaradi premajhne natančnosti simulacije, so agenti, ki lahko skočijo skozi zidove, in prevelike odbojne sile med agenti.

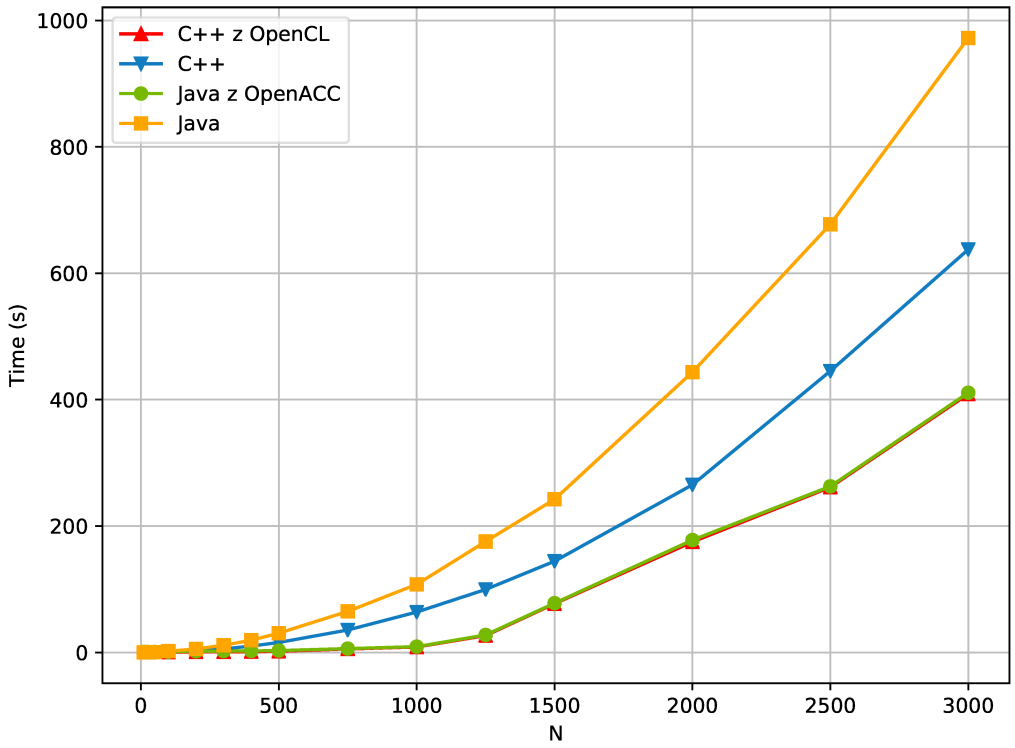
Če pogledamo enačbo (5.1), vidimo, da je računska kompleksnost izračuna enačbe za prvi del $\Theta(1)$, za drugi del $\Theta(N)$ in za zadnji del $\Theta(N_W)$, kjer N_W predstavlja število zidov. Te sile moramo izračunati za vsakega agenta, kar pomeni, da imajo izračuni posameznih členov za vsako iteracijo časovno kompleksnost $\Theta(N)$, $\Theta(N^2)$ in $\Theta(NN_W)$. Ker je po navadi v simulaciji število zidov nizko, sklepamo, da ima tudi zadnji člen enačbe skupno časovno kompleksnost $\Theta(N)$. Zaradi njegove manjše računske kompleksnosti in njegove manjše pomembnosti pri simulaciji smo ga lahko pri naših testih izpustili iz izračuna.

Opravili smo tudi dinamično analizo časa izračuna in ugotovili, da za večje N računanje interakcij med agenti vzame večino računskega časa – tudi več kot 99%. Zato smo se odločili, da za test na GPE prenesemo le računanje interakcij med agenti. Nit poženemo za vsako interakcijo med dvema agentoma, nato pa poženemo še en ščepec, kjer za vsakega agenta ena od niti sešteje vse rezultate in jih vrne na CPE. To pomeni, da je računska kompleksnost $\Theta(N^2)$, kompleksnost prenosa podatkov pa je le $\Theta(N)$.

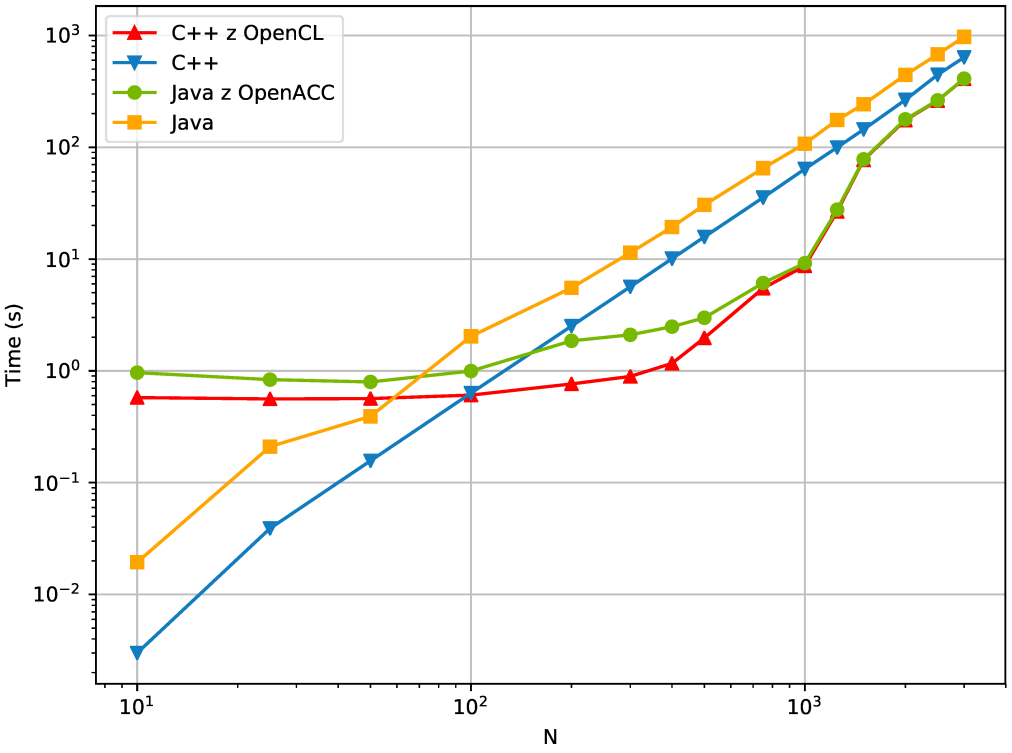
Teste smo izvajali na različnem številu agentov. Za vsak test smo izvedli 300 iteracij simulacije, kar je predstavljalo 6 sekund simulacijskega časa. Poleg simulacije v Javi in Javi z OpenACC smo meritve opravili tudi v jeziku C++. Rezultate vidimo v tabeli 5.2, prikazani pa so tudi na slikah 5.5 in 5.6.

N	Java	C++	Java z OpenACC	C++ z OpenCL
10	0,019 s	0,002 s	0,964 s	0,577 s
25	0,210 s	0,039 s	0,835 s	0,561 s
50	0,392 s	0,206 s	0,796 s	0,565 s
100	2,037 s	0,693 s	0,997 s	0,607 s
200	5,531 s	2,749 s	1,857 s	0,764 s
300	11,401 s	5,652 s	2,101 s	0,892 s
400	19,329 s	10,080 s	2,485 s	1,167 s
500	30,512 s	15,780 s	2,984 s	1,968 s
750	64,966 s	35,500 s	6,108 s	5,485 s
1000	107,658 s	63,920 s	9,186 s	8,677 s
1250	175,521 s	99,646 s	27,660 s	26,686 s
1500	242,351 s	144,350 s	77,974 s	77,151 s
2000	443,612 s	265,335 s	177,804 s	174,974 s
2500	677,418 s	445,308 s	262,826 s	261,472 s
3000	972,194 s	637,639 s	410,914 s	408,966 s

Tabela 5.2: Časi 300 iteracij simulacije črede velikosti N



Slika 5.5: Časi 300 iteracij simulacije črede velikosti N. Krivulji OpenCL in OpenACC se prekrivata.



Slika 5.6: Log-log izris časov 300 iteracij simulacije črede velikosti N

5.3 Komentar meritev

Meritve pričakovano kažejo, da je uporaba OpenACC hiter in učinkovit način za pohitritev kode, ki teče v programskem jeziku Java. It rezultatov je razvidno, da je to vredno storiti le, če imamo dovolj velik problem. Programi, ki jih prevedemo z OpenACC, morajo pred izvajanjem ščepcev ustvariti veliko število OpenCL objektov in na GPE prenesti veliko število podatkov. To pomeni, da so za majhne probleme neučinkoviti in je bolje, da GPE v takih primerih ne uporabljamo.

Mandelbrotova množica pokaže učinkovitost vzporednega računanja z GPE. Ker je problem povsem trivialno deljiv na niti, je izračun množice za 100 % hitrejši na GPE kot na CPE. Implementacija na CPE z Javo je za približno 33 % počasnejša kot implementacija C++. Razlike med Javo z OpenACC in C++ z OpenCL praktično ni, saj se skoraj ves potek programa izvaja na GPE, kar pomeni, da razlika v hitrosti med C++ in Javo v tem primeru ne pride do izraza.

Pri simulaciji čred vidimo podoben rezultat. Medtem ko pri manjšem številu agentov implementacija C++ močno prehiteva GPU implementacijo, vidimo, da ko postane problem dovolj velik, postane implementacija OpenACC dosti hitrejša. Računska kompleksnost tega problema je $\Theta(N^2)$, vendar je pri implementaciji na GPU najpočasnejša operacija prenos podatkov, ta pa ima kompleksnost $\Theta(N)$. To pomeni, da pri manjšem številu agentov problem na GPE raste linearno, na CPE pa raste z N^2.

To dobro vidimo na sliki 5.6, ki predstavlja log-log graf rezultatov. Vemo, da so polinomske funkcije na log-log grafu videti kot linearne funkcije, kjer njihov naklon pove stopnjo polinoma. Kot vidimo, je čas izvajanja na GPE do $N = 100$ konstanten. Do te točke je največ časa porabljenega za vzpostavljanje konteksta in pripravo ščepca za izvajanje na GPE ter prepisovanja podatkov, ki imajo konstantno velikost. Samo računanje vzame zelo malo časa. Pri $100 \leq N \leq 1000$ vidimo, da problem raste linearno, saj tu časovno prevladuje prenos podatkov. Od $N > 1000$ pa začne prevladovati izračun in težavnost problema se začne obnašati po $O(N^2)$. Na istem grafu vidimo

iz naklona črte, da se problem, implementiran s C++, povsod obnaša kot $O(N^2)$.

Opazimo tudi, da je zaradi večjega deleža kode, ki se izvaja sekvenčno, pri simulaciji čred čas izvajanja na GPE bolj podoben času izvajanja na CPE, še posebej kode, napisane v C++. Koda na GPE je okoli 50 % hitrejša kot najhitrejša koda na CPE. Še vedno pa je to opazna pospešitev, ki jo pridobimo brez večjega napora.

Pri tem tudi opazimo, da je razlika med Javo, ki uporablja GPE, in C++ kodo, ki uporablja GPE, zelo majhna. Čeprav je tu delež programa, ki se izvaja na CPE, večji kot pri računanju Mandelbrotove množice, se zaradi računske kompleksnosti dela, ki ga izvajamo na GPE, zelo hitro zgodi, da večino časa izvajamo le del programa, ki uporablja GPE. To pomeni, da tudi pri tem ni razlike med tema implementacijama.

Ti rezultati so zelo spodbudni, vendar je potrebno omeniti, da je implementacija C++ z OpenCL zelo podobna kodi, ki jo ustvari naš prevajalnik. Če bi želeli, bi z implementacijo C++ lahko dosegli boljše rezultate, saj OpenCL ponuja več možnosti za optimizacijo kode.

Poglavje 6

Zaključek

V delu smo zasnovali in razvili prevajalnik, ki je iz javanske kode, označene z direktivami OpenACC, prevajal kodo v javansko kodo, ki je označene dele kode izvajala na GPE. Prevajalnik omogoča preprosto pospešitev javanskih programov, ki so primerni za paralelizacijo in omogoča učinkovito izrabo računalniškega sistema. Narejen je kot predprocesor. Njegov izhod je izvorna koda, ki je prevedljiva s katerim koli javanskim prevajalnikom. Izhodna koda potrebuje le dostop do knjižnice JOCL, ki ji omogoči dostop do funkcionalnosti OpenCL.

Prevajalnik smo implementirali z leksikalnim in sintaksnim analizatorjem, ki smo ga ustvarili z orodjem ANTLR4. Z njima smo opravili delitev vhodne kode na osnovne simbole jezika in stvaritev sintaksnega drevesa. Semantično analizo smo opravili z obiskovalci, ki smo jih ustvarili, da so analizirali uporabo spremenljivk, potek programa in direktive OpenACC. S podatki, pridobljenimi v semantični analizi, smo nato sintetizirali izhodno kodo. Izhodno kodo je sestavljala izvorna koda, v katero smo vrinili ukaze OpenCL za vzpostavitev konteksta in ostalih objektov za komunikacijo z grafično procesno enoto, ukaze za prenos podatkov na in z naprave ter kodo ščepca, ki so ga direktive OpenACC določile za izvajanje na zunanji napravi.

Implementirali smo samo omejen nabor funkcionalnosti, ki jih ponuja OpenACC, a so že te zadostne, da lahko z njimi pospešimo realne probleme.

Nabor funkcionalnosti smo testirali s testi za OpenACC EPCC, ki smo jih prevedli v Javo. Testi so pokazali, da naš prevajalnik pravilno deluje na nekaterih primerih, vendar pa obstaja velik del standarda, ki ga naš prevajalnik ne podpira.

Teste hitrosti smo opravili na dveh dovolj težkih problemih, da smo lahko pokazali, kakšne pospešitve lahko pričakujemo od Jave skupaj z OpenACC. Oba problema sta bila, brez večjih sprememb, prenesena v Javo z OpenACC tako da smo opazili velike pospešitve. Pokazali smo, da so naši prevodi kode v OpenCL dobri in da je prevajanje visokonivojske kode na GPE lahko rešitev za tiste, ki potrebujejo veliko računske moči v višje nivojskih jezikih.

Trenutno prevajalnik podpira funkcionalnosti, ki so potrebne za osnovno delovanje OpenACC. To je dobro izhodišče za dodajanje nadaljnjih funkcionalnosti. Zelo uporabna bi bila implementacija določila `if`, ki omogoča, da se lahko programi, ki jih ustvarimo s prevajalnikom, odločijo dinamično med tekom ali bodo določeno zanko izvedli na GPE ali kar na CPE. Zelo uporabna bi bila tudi implementacija delitve dela v zanki. Nekatere zanke zahtevajo preveč pomnilnika ali preveč niti, da bi jih lahko zagnali na GPE naenkrat. To bi lahko rešili z implementacijo dinamičnega dodeljevanja dela, ki bi v končnem programu med tekom analiziral in poganjal ščepce take velikosti, ki bi jih GPE lahko izvajal. Prav tako bi lahko dinamično dodeljevanje dela izkoristili za izrabo več naprav naenkrat. Hitrost izhodne kode bi lahko izboljšali tako, da bi ob morebitnem večkratnem zagonu ščepca nekatere OpenCL objekte ponovno uporabili, namesto da jih vmes pobrišemo in nato ponovno ustvarimo.

Literatura

[1] OpenACC Community, OpenACC, more science, less programming, https://www.openacc.org/community, accessed: 2018-08-10 (2011–2017).

[2] A. W. U. Munipala, S. V. Moore, Code complexity versus performance for GPU-accelerated scientific applications, in: Proceedings of the 2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), IEEE, Salt Lake City, UT, USA, 2016, pp. 50–50.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. S. Sunderam, PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing, MIT press, 1994.

[4] W. D. Gropp, W. Gropp, E. Lusk, A. Skjellum, Using MPI: portable parallel programming with the message-passing interface, Vol. 1, MIT press, 1999.

[5] OpenMP Architecture Review Board, OpenMP, http://www.openmp.org/, accessed: 2018-08-10 (2012–2018).

[6] Nvidia Developer Zone, CUDA C programming guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, accessed: 2018-08-10 (2017).

[7] Khronos OpenCL Working Group, OpenCL Overview - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl, accessed: 2018-08-10 (2011).

[8] M. Hutter, JOCL - Java bindings for OpenCL, http://www.jocl.org/, accessed: 2018-09-11 (2017).

[9] Apple, Metal performance shaders, https://developer.apple.com/documentation/metalperformanceshaders, accessed: 2018-08-30 (2018).

[10] H.-W. Peng, J. J.-J. Shann, Translating OpenACC to LLVM IR with SPIR kernels, in: Proceedings of the 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), IEEE, Okayama, Japan, 2016, pp. 597–602.

[11] Nvidia Developer Zone, Parallel thread execution ISA version 6.0, http://docs.nvidia.com/cuda/parallel-thread-execution/index.html, accessed: 2018-8-30 (2017).

[12] D. Blatner, OpenACC and the PGI Compiler.

[13] S. Sawadsitang, J. Lin, S. See, F. Bodin, S. Matsuoka, Understanding performance portability of OpenACC for supercomputers, in: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), IEEE, Hyderabad, India, 2015, pp. 699–707.

[14] GCC Wiki, OpenACC, https://gcc.gnu.org/wiki/OpenACC, accessed: 2018-08-10 (2017).

[15] V. G. V. Larrea, O. Hernandez, C. Philippidis, R. Allen, et al., An in-depth evaluation of GCC's OpenACC implementation on Cray systems, Cray User Group 2017 Proceedings (CUG2017 Proceedings).

[16] M. Larabel, Samsung brings OpenACC 1.0+ support to GCC Fortran, `https://www.phoronix.com/scan.php?page=news_item&px=MTU4MjQ`, accessed: 2018-08-10 (2014).

[17] O. Hernandez, W. Ding, W. Joubert, D. Bernholdt, M. Eisenbach, C. Kartsaklis, Porting OpenACC 2.0 to OpenMP 4.0: Key similarities and differences, `http://openmpcon.org/wp-content/uploads/openmpcon2015-oscar-hernandez-portingacc.pdf`, Oak Ridge National Laboratory, US Dept. of Energy, accessed: 2018-8-20 (2016).

[18] N. Sultana, A. Calvert, J. L. Overbey, G. Arnold, From OpenACC to OpenMP 4: Toward automatic translation, in: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale (XSEDE16), ACM, Miami, FL, USA, 2016, pp. 44:1–44:8.

[19] EPCC OpenACC benchmark suite, `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite`, accessed: 2018-08-10 (2013).

[20] T. Vanderbruggen, J. Cavazos, Generating OpenCL C kernels from OpenACC, in: Proceedings of the International Workshop on OpenCL 2013 & 2014 (IWOCL '14), ACM, Bristol, United Kingdom, 2014, pp. 9:1–9:10.

[21] A. Lashgar, A. Majidi, A. Baniasadi, IPMACC: Open source OpenACC to CUDA/OpenCL translator, arXiv preprint arXiv:1412.1127 (2014).

[22] A. Lashgar, A. Majidi, A. Baniasadi, IPMACC: Translating OpenACC API to OpenCL, Presented at the poster session of The 3rd International Workshop on OpenCL (IWOCL '15), (2015).

[23] R. Reyes, I. López-Rodríguez, J. Fumero, F. de Sande, accULL: an OpenACC implementation with CUDA and OpenCL support, Proceedings of the 18th International Conference Euro-Par 2012 (2012) 871–882.

[24] A. Lashgar, A. Baniasadi, Employing software-managed caches in OpenACC: Opportunities and benefits, ACM Transactions on Modeling and Performance Evaluation of Computing Systems 1 (1) (2016) 2:1–2:34.

[25] A. Klöckner, Pycuda, Courant Institute of Mathematical Sciences, New York University.

[26] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing 38 (3) (2012) 157–174.

[27] S. Lam, A. Pitrou, S. Seibert, Numba, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC-LLVM 2015, 2015.

[28] S. K. Lam, A. Pitrou, S. Seibert, Numba: A LLVM-based python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, ACM, 2015, p. 7.

[29] D. Eddelbuettel, CRAN task view: High-performance and parallel computing with R.

[30] J. Reese, S. Zaranek, GPU programming in MATLAB, MathWorks News&Notes. Natick, MA: The MathWorks Inc (2012) 22–5.

[31] QuantAlea, Alea GPU, `http://www.aleagpu.com/release/3_0_4/doc/`, accessed: 2018-08-30 (2018).

[32] YaccConstructor, Brahma.FSharp, `http://yaccconstructor.github.io/Brahma.FSharp/`, accessed: 2018-08-30 (2018).

[33] Nessos, GpuLinq, `https://github.com/nessos/GpuLinq/`, accessed: 2018-08-30 (2018).

[34] OpenACC-Standard.org, The OpenACC application programming interface, version 2.6, `https://www.openacc.org/sites/default/`

files/inline-files/OpenACC.2.6.final.pdf, accessed: 2018-08-10 (2017).

[35] T. Parr, et al., Antlr: Another tool for language recognition (2006).

[36] grammars-v4 - Grammars written for ANTLR v4, https://github.com/antlr/grammars-v4, accessed: 2018-08-30 (2018).

[37] T. Parr, K. Fisher, LL (*): the foundation of the ANTLR parser generator, ACM Sigplan Notices 46 (6) (2011) 425–436.

[38] D. Helbing, I. Farkas, T. Vicsek, Simulating dynamical features of escape panic, Nature 407 (6803) (2000) 487.