SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD

Paralelizacija algoritama na heterogenim platformama uz pomoć sustava OpenCL

Veljko Dragšić
Voditelj: Domagoj Jakobović

Zagreb, Veljača, 2010.

Sadržaj

1 Uvod

1.1 Paralelni algoritmi

Uobičajen način razmišljanja o radu procesora je da slijedno prolazi kroz tekst programa odnosno njegov strojni jezik i izvršava naredbe. Prilikom oblikovanja algoritama također razmišljamo na taj način, tj. izrađujemo slijedni algoritam. Ako je slijedni algoritam procesorski prezahtjevan, odnosno ako bi se predugo izvršavao na jednom procesoru, nameće se potreba da se on paralelizira kako bi se mogao izvršavati na više procesora ili računala istovremeno, te se na taj način ubrzati. Paralelizacija algoritama može biti trivijalna, ali i izrazito kompleksna, ovisno o samom problemu koji se želi riješiti; stoga postoje i različiti načini paralelizacije. Ponekad nije moguće neki problem u potpunosti paralelizirati, pa onda govorimo o slijednom i paralelnom dijelu algoritma, a o tome ovisi i ubrzanje koje se može postići (veći udio paraleliziranog dijela podrazumijeva i veće ubrzanje). Kako algoritmi postaju sve zahtjevniji i kompleksniji, a procesori sadrže nekoliko jezgri i mnogo logike koja pospješuje paralelnu izvedu naredbi, tako se i nameće sve veća potreba za paralelizacijom algoritama i posla koje obavljamo na računalu.

1.2 Heterogene platforme

Pojam heterogene platforme podrazumijeva mikroprocesore različitih namjena i arhitektura. Do sredine '90-ih godina prošlog stoljeća smo u osobnim računalima imali samo centralne mikroprocesore (engl. central processing unit, CPU), a kasnije su im se pridružili i grafički mikroprocesori (engl. graphics processing unit, GPU) ponajviše specijalizirani za ubrzavanje prikaza trodimenzionalne grafike. Kroz vrijeme su se proširivali setovi instrukcija, broj bitova sa kojima barataju je skočio sa 32 na 64, pojavljivale su se nove arhitekture, radni takt je rastao dok nije naišao na fizička ograničenja postojeće tehnologije izrade, da bi se zatim pojavili višejezgreni mikroprocesori. Danas su nam na raspolaganju višejezgreni centralni mikroprocesori opće namjene, visokoparalelizirani grafički procesori i mnoštvo mikroprocesora specifičnije namjene koje nalazimo u lako dostupnim igračim konzolama pa sve do specijaliziranih poslužitelja. Logično se javlja potreba za što učinkovitijim i lakšim iskorištavanjem potencijala mikroprocesora različitih arhitektura.

1.3 OpenCL

Open Computing Language (kraće: *OpenCL*) je sustav koji omogućava jednostavniju paralelizaciju algoritama i njihovo izvođenje na heterogenim platformama. *OpenCL* se sastoji od programskog jezika baziranog na C-u (ISO C99) i sučelja za pristup hardverskim platformama. Kroz navedeni princip se prvo oblikuje željeni algoritam koji zovemo programska jezgra (engl. *kernel*), a zatim se izvodi u postavljenoj okolini na na dostupnim platformama, poput centralnog ili grafičkog mikroprocesora. Bitno je napomenuti da je *OpenCL* prvi sustav koji omogućuje da se isti tekst programa, točnije programska jezgra, izvodi paralelno na različitim platformama, što je upravo i njegova glavna prednost. Projekt je krajem 2008. godine inicijalno započela tvrtka *Apple inc.*, dok je prva potpuna implementacija postala dostupna u operacijskom sustavu *Mac OS X* u kolovozu 2009. godine. Upravljanje projektom je kasnije prepušteno konzorciju *Khronos* koji okuplja gotovo sve značajne IT kompanije u tom području, a specifikacija standarda je otvorenog tipa.

1.4 Zadatak rada

Opis i upoznavanje sa sustavom *OpenCL* je primarni zadatak ovog rada. Pomoću sustava se na praktičnim primjerima treba ispitati složenost paralelizacije algoritama za heterogene platforme, uzimajući u obzir određene optimizacije za pojedine platforme. Također se pri analizi sustava uzima u obzir učinkovitost, performanse, dostupne implementacije i podržane platforme. Analiza sustava uključuje i usporedbu sa drugim tehnologijama za paralelizaciju, te isticanje prednosti i mana *OpenCL*-a. Rad nije samo usmjeren na sustav samo sa softverske strane, već će predočiti opis i usporedbu hardverskih platformi koje su danas dostupne, ali i onih koje tek možemo očekivati u skorijoj budućnosti. Na temelju rezultata ispitivanja u *OpenCL*-u na dostupnim platformama, iznijet će se okvirne smjernice o opremanju računala namijenjenog izvedbi paralelnih algoritama, uzimajući u obzir procesorsku snagu, učinkovitost i cijenu.

2 Open Computing Language

2.1 Uvod

OpenCL je nastao s ciljem olakšavanja paralelizacije algoritama i omogućavanja njihovog izvršavanja na različitim platformama. Dok se do prije nekoliko godina za procesorski zahtjevne algoritme oslanjalo isključivo na centralne mikroprocesore opće namjene, s vremenom su ih po performansama i kompleksnosti dostigli specijalizirani grafički mikroprocesori. Različita namjena podrazumijeva i različite arhitekture mikroprocesora, stoga tekst programa nije moguće jednostavno prenositi između dvije poprilično različite arhitekure. Glavna značajka grafičkih mikroprocesora je određena što bržim prikazom trodimenzionalne grafike, a to je zapravo paralelno obavljanje istovjetnih matematičkih operacija na velikoj količini podataka. Uslijedilo je iskorištavanje njihovog potencijala i u općenitije svrhe, nazvano *General Purpose computing on Graphics Processing Units*, *GP/GPU*. Glavni proizvođači grafičkih mikroprocesora su pružili sučelja za njihovo iskorištavanje, *Nvidia inc.* tehnologiju zvanu *CUDA*, a *AMD/ATI*[1] pandan zvan *ATI Stream.* Bilo je logično očekivati tehnologiju koja će omogučiti lakšu paralelizaciju algoritama i njihovo izvođenje na centralnom i/ili grafičkom mikroprocesoru. Upravo tu prazninu popunjava *OpenCL*, koji omogućava da se jednom napisan tekst programa izvršava na različitim arhitekturama.

Idejni začetnik projekta je tvrtka *Apple*, poznata po prodaji računala u kombinaciji sa svojim operacijskim sustavom *Mac OS X* baziranim na *Unixu* odnosno *BSD*-a[2]. Kako se kompanija bavi sklopovljem i programskom opremom bilo je u neku ruku prirodno za očekivati da će prva pokrenuti razvoj takvog sustava. U realizaciji projekta su osim *Applea* sudjelovale i kompanije *AMD/ATI*, *IBM*, *Intel* i *Nvidia*. U srpnju 2008. godine je upravljanje daljnim razvojem *OpenCL*-a prepušteno konzorciju *Khronos*, koji je oformljen 2000. godine od strane kompanija koje se bave proizvodnjom grafičkog sklopovlja i multimedijom. *Khronos Group* već upravlja razvojem nekih izrazito bitnih standarda u grafičkoj industriji, poput *OpenGL*-a. Specifikacija *OpenCL*-a je otvorena tako da ju bilo koji proizvođač može implementirati za svoje mikroprocesore. Danas u razvoju sudjeluju sve relevantnije kompanije na području računalne grafike i mikroprocesora, od proizvođača računalnih igara poput *Blizzarda* i *Electronic Artsa*, pa sve do kompanija *ARM*, *Ericsson*, *General Electric*, *Texas Instruments* i mnogih drugih.

2.2 Implementacije i podržane platforme

U vrijeme pisanja ovog rada je bilo dostupno tek nekoliko implementacija, te je podržan manji broj platformi. Do sada su kompanije *AMD/ATI*, *Nvidia* i *Via* barem djelomično podržali svoje mikroprocesore kroz upravljačke programe (engl. *drivers*) koje izdaju.

Kompanija *AMD* je u potpunosti implementirala *OpenCL* kroz svoj skup upravljačkih programa i *ATI Stream SDK 2.0* za operacijske sustave *GNU/Linux* i *Windows*. *Nvidia* je učinila isto kroz svoje upravljačke programe i *SDK* (engl. *Software Developers Kit*). Appleov *Mac OS X* dolazi već sa potpunom implementacijom *OpenCL*-a od verzije *Snow Leopard*, u kojoj

[1] Kompanija *Advanced Micro Devices* (*AMD*) je 2006. godine preuzela *ATI Technologies Inc.* *AMD* je do onda bio drugi proizvođač x86 mikroprocesora, iza *Intela*, a *ATI* se natjecao sa *Nvidiom* na području grafičkih mikroprocesora.

[2] *BSD*, *Berkeley Software Distribution*, je varijanta *Unix* operacijskog sustava nastala na sveučilištu Berkeley u SAD-u. Mnogi danas poznati operacijski sustavi se djelomično baziraju na njemu, od zatvorenog *Mac OS X*-a do otvorenih *FreeBSD*-a i *OpenBSD*-a.

Tablica 1: Podržane *OpenCL* platforme

Kompanija	Podržane platforme	OS
AMD	ATI Radeon grafičke kartice s mikroprocesorima R700 i R800, AMD x86 mikroprocesori sa setom instrukcija SSE3	GNU/Linux, Mac OS X Snow Leopard, Windows XP, Vista, 7
Nvidia	GeForce 8 i noviji grafički mikroprocesori, Quadro i Tesla grafički mikroprocesori	GNU/Linux, Mac OS X Snow Leopard, Windows XP, Vista, 7
Via	ChromotionHD 2.0 video mikroprocesor	

se i općenito pojavila prva potpuna implementacija. *Intel* kao proizvođač najraširenijih mikroprocesora opće namjene još nije izdao nikakvu implementaciju, a također se čeka da i *IBM* izda podršku za svoje *Cell/BE* mikroprocesore[3].

Zgodno je napomenuti kako se *AMD*-ovi upravljački programi za centralne mikroprocesore mogu iskoristiti i za Intelove, jer su oba iste arhitekture, x86. Za to je u računalu potrebno imati *AMD*-ovu grafičku karticu, jer upravljački programi za grafički i centralni mikroprocesor, a koji implementiraju *OpenCL*, dolaze zajedno.

Kako su *AMD* i *Nvidia* već otprije omogućili GP/GPU na svojim grafičkim mikroprocesorima kroz sustave *Stream* i *CUDA*, tako se *OpenCL* može smatrati samo nadogradnjom na njihove već postojeće tehnologije.

U ovom radu se za potrebe ispitivanja koristi implementacija PyOpenCL [4]. *PyOpenCL* omogućava pristupanje sučelju *OpenCL*-a kroz programski jezik *Python*, dok se programske jezgre i dalje programiraju u *OpenCL C*-u. To svejedno ne isključuje potrebu za instaliranim upravljačkim programima za platforme koje želimo korisiti. Na taj način se osjetno skraćuje tekst programa potreban za izradu željene aplikacije i postavljanje okoline za izvođenje programskih jezgri, dok njihova brzina izvođenja na ciljanim platformama ostaje nepromijenjena.

2.3 Opis sustava i osnovne funkcionalnosti

Open Computing Language je standard namijenjen paralelnom programiranju opće namjene na heterogenim platformama, pritom podrazumijevajući danas dostupne mikroprocesore različitih namjena i arhitektura, a velike računalne snage. Standard *OpenCL* cilja na područje od ugradbene (engl. *embedded*) i potrošačke (engl. *consumer*) programske opreme, pa sve do razine računarstva visokih performansi (engl. *High performance Computing, HPC*). Sustav je zadržan na niskoj razini koja je blizu samog sklopovlja, što omogućuje postizanje izrazito dobrih performansi. Također je bitno da je apstrakcija koju sustav pruža prenosiva među platformama, tako da se u budućnosti može očekivati njegova raširena primjena, pogotovu u području grafičkih, procesorski zahtjevnih i paralelnih aplikacija. U vrijeme pisanja rada nije bilo dostupnih aplikacija šire upotrebe iz razloga što je *OpenCL* kao tehnologija praktično

[3] Cell mikroprocesor je nastao u suradnji IBM-a, Toshibe i Sonya, te se nalazi u Playstation 3 igračim konzolama. Mikroprocesor je zanimljive arhitekure je se sastoji od jedne jezgre općenite namjene, te nekoliko specijaliziranih vektorskih jezgri.

[4] Dokumentacija projekta je dostupna na web adresi http://documen.tician.de/pyopencl/, a glavna stranica projekta na http://mathema.tician.de/software/pyopencl

postao dostupan tek u proteklih nekoliko mjeseci.

Standard *OpenCL* se sastoji od sučelja za pristup sklopovskim platformama, biblioteka i sustava za raspodjelu poslova, te prenosivog programskog jezika:

- Podrška za paralelne programske modele bazirane na podjeli podataka i/ili zadataka
- Podskup programskog jezika ISO C99, proširen ekstenzijama za paralelizaciju
- Definira konzistentne numeričke zahtjeve bazirane na standardu IEEE 754
- Definira konfiguracijski profil za ručne i ugradbene uređaje
- učinkovito komunicira se tehnologijama *OpenGL* i ostalim sučeljima za pristup grafičkim karticama

Za korištenje *OpenCL*-a nisu potrebna uobičajena grafička sučelja poput otvorenog *OpenGL*-a ili Microsoftovog *DirectX*-a, već sustav direktno komunicira sa sklopovljem putem upravljačkih programa proizvođača.

2.4 Arhitektura OpenCL-a

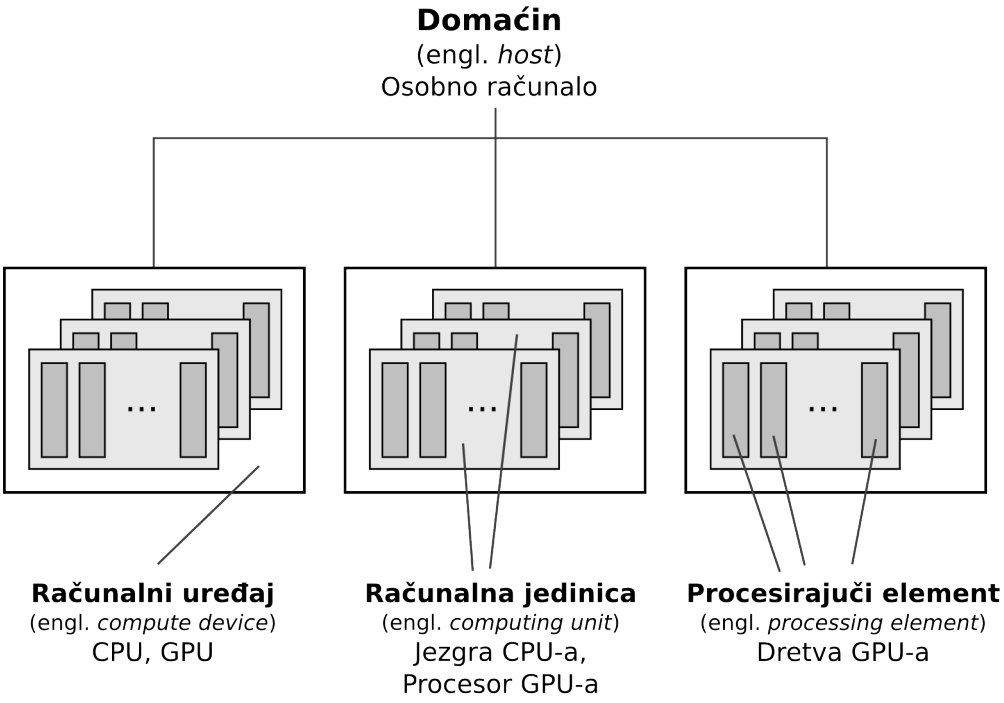
Arhitektura OpenCL-a se sastoji od hijerarhije u nekoliko modela:

- platformski model,
- memorijski model,
- izvršni (engl. *execution*) model,
- programski (engl. *programming*) model.

2.4.1 Platformski model

OpenCL-ov platformski model se sastoji od domaćina (engl. *host*) spojenog na jedno ili više računalnih uređaja (engl. *compute device*). Svaki računalni uređaj može imati jednu ili više računalnih jedinica (engl. *computing unit*), a svaka računalna jedinica jedan ili više procesirajućih elemenata (engl. *processing element*). Procesirajući elementi barataju sa podacima za obradu.

Kako model djeluje previše apstraktno, ilustracija u stvarnosti bi bila da domaćina zamijenimo za naše stolno ili prenosivo računalo, koje zatim u sebi ima više računalnih uređaja, pritom misleći na jedan ili više centralnih ili grafičkih mikroprocesora, odnosno CPU-a i GPU-a. Svaki taj računalni uređaj može imati više računalnih jedinica, poput višejezgrenih CPU koji su danas uobičajeni, te imaju po dvije, četiri ili čak više jezgri. GPU je ponešto drugačije arhitekture, te jedan mikroprocesor može imati na desetke jezgri, pa čak i stotine jezgri. Procesirajući element kod GPU-a odgovara dretvi (engl. *thread*) koje on izvršava u velikom broju, čak i po nekoliko tisuća istovremeno.



Slika 1: *OpenCL* platformski model

Aplikacija u *OpenCL*-u se pokreće na domaćinu, koji priprema okolinu za izvođenje programskih jezgri, prenosi podatke računalnim uređajima, te zatim pokreće izvođenje programskih jezgri na procesirajućim elementima. Procesirajući elementi prilikom rada djeluju po *SIMD* principu (engl. *Single Instruction, Multiple Data*).

2.4.2 Memorijski model

Memorijski model standarda *OpenCL* raspoznaje četiri memorijska područja, a radna jedinica koja izvršava programsku jezgru može pristupiti svakoj od njih.

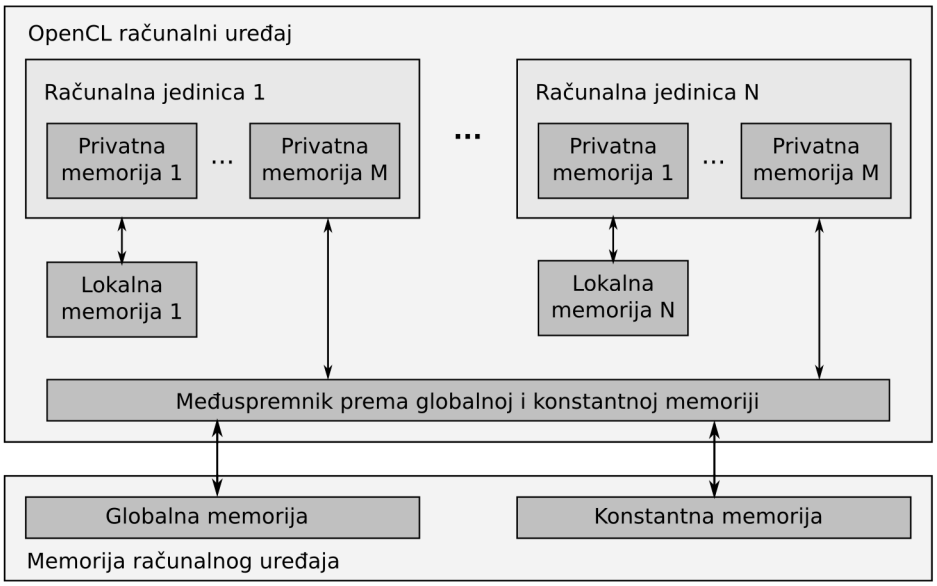
- **Globalna memorija**: Sve radne jedinice, neovisno u kojoj se radnoj grupi nalaze, mogu pisati ili čitati iz bilo koje lokacije globalne memorije. Fizički globalna memorija odgovara radnoj memoriji (engl. *Random access memory, RAM*) uređaja. Točnije u slučaju CPU-a je to radna memorija na matičnoj ploči, a kod GPU-a je riječ o radnoj memoriji na grafičkoj kartici.

- **Konstantna memorija**: Područje globalne memorije koje ostaje nepromijenjeno za vrijeme izvođenja programskih jezgri. Za zauzimanje i stavljanje podataka u konstantnu memoriju brine se domaćin.

- **Lokalna memorija**: Memorijsko područje namijenjeno radnim jednicama koje se nalaze unutar iste radne grupe. Radne jedinice iz druge grupe joj ne mogu pristupati. Ovisno o *OpenCL* implementaciji i tipu uređaja, za lokalnu memoriju može biti rezerviran dio globalne memorije ili se mogu koristiti neka druga memorijska područja. U praksi lokalna memorija kod CPU-a i GPU-a odgovara njihovim pričuvnim memorijama (engl. *cache*

memory) koje se nalaze u samom mikroprocesoru. Pristup lokalnoj memoriji je značajno brži nego globalnoj memoriji, tako da se, uzimajući u obzir koju ćemo memoriju koristiti, može postići znatno ubrzanje prilikom pisanja algoritma. Količina pričuvne memorije je znatno manja kod GPU-a nego kod CPU-a.

- **Privatna memorija**: Područje memorije namijenjeno svakoj radnoj jedinici pojedinačno. Svaka radna jedinica ima pristup samo svojoj privatnoj memoriji.



Slika 2: *OpenCL* memorijski model

Tablica 2 prikazuje u kojem memorijskom području domaćin i programske jezgre mogu zauzimati (alocirati) memoriju. Podrazumijevaju se dva načina zauzimanja: statično, koje se odvija u vrijeme prevođenja programa i dinamično, u vrijeme izvođenja programa. Drugi redak svakog polja označuje mogučnost pisanja i čitanja iz memorije.

Tablica 2: OpenCL memorijski model

	Globalna	Konstantna	Lokalna	Privatna
Domaćin	Dinamična Čitanje / Pisanje	Dinamična Čitanje / Pisanje	Dinamična Bez pristupa	Dinamična Bez pristupa
Programska Jezgra	Bez alokacije Čitanje / Pisanje	Statična Čitanje / Pisanje	Statična Čitanje / Pisanje	Statična Čitanje / Pisanje

OpenCL aplikacija pokrenuta na domaćinu kroz *OpenCL* sučelje zauzima lokacije u globalnoj memoriji i stvara memorijske objekte, te se brine oko izvođenja memorijskih naredbi. Memorijski objekti su zauzeta polja memorije kojima može pristupati domaćin i *OpenCL* uređaji. Kroz njih aplikacija na domaćinu i programske jezgre koje se izvode na *OpenCL* uređaju razmjenjuju podatke. Memorije na domaćinu i *OpenCL* uređaju su neovisne jedna o drugoj

iz razloga što je domaćin izvan *OpenCL* sustava. Komunikacija se međutim može ostvariti na dva načina: eksplicitnim kopiranjem podataka iz memorije na domaćinu u memoriju na *OpenCL* uređaju i obrnuto ili preslikavanjem (engl. *mapping*) njihovih memorijskih područja.

Eksplicitno kopiranje podataka između memorijskih objekata i memorije na domaćinu se odvija na zahtjev domaćina, a može biti blokirajuće ili neblokirajuće, tj. izvođenje programa na domaćinu može čekati da kopiranje se završi ili nastaviti dalje ne čekajući završetak kopiranja i njegov ishod.

Kod memorijskog preslikavanja domaćin može *OpenCL* memorijske objekte preslikati u svoj adresni prostor, a naredba za preslikavanje može biti blokirajuća i neblokirajuća. Kada se memorijski objekt preslika, domaćin može pisati ili čitati po tom adresnom prostoru.

OpenCL jamči memorijsku konzistenciju među lokalnom memorijom koja je dostupna radnim jedinicama unutar iste radne grupe prilikom sinkronizacijske granice (engl. *barrier*). Isto vrijedi i za globalnu memoriju, ali ne u slučaju radnih jedinica koje pripadaju različitim radnim grupama.

2.4.3 Izvršni model

Izvođenje *OpenCL* aplikacije se odvija u dva dijela: programske jezgre se izvode na jednom ili više računalnih uređaja, a na domaćinu se izvodi tekst programa koji upravlja izvođenjem programskih jezgri, točnije brine se za stvaranje konteksta (engl. *contex*) za izvođenje, programskih slijedova (engl. *programming queues*) i stvaranje memorijskih međuspremnika (engl. *memory buffer*) za pisanje i čitanje podataka iz memorije uređaja.

Za pokretanje *OpenCL* aplikacije se moraju definirati veličine dimenzija problema koji se obrađuje, jer se prilikom pokretanja programskih jezgri na uređajima stvara indeksni prostor koji odgovara tim dimenzijama. Dimenzije problema se najlakše mogu ilustrirati na primjeru matrice (Slika 3.) koja se sastoji od n redaka (Gx) i m stupaca (Gy). Indeksni prostor odgovara dimenzijama matrice, te se za svaki njegov element pokreće po jedna radna jedinica, pretpostavimo dretva, koja obrađuje element matrice koji se nalazi na dotičnom indeksu. Veličina indeksnog prostora ne mora nužno odgovarati dimenzijama matrice, moguće je i definirati manji indeksni prostor od dimenzija matrice, pa programski unutar svake radne jedinice obraditi više elemenata matrice.

Možemo definirati jednu, dvije ili tri dimenzije problema, te za svaku dimenziju postavljamo veličinu odnosno prirodni broj elemenata u rasponu od 1 do N. Na temelju tih postavki se stvara indeksni prostor istih dimenzija, a prilikom pokretanja svaka radna jedinica (engl. *working-item*) obrađuje jedan element iz tog indeksnog prostora, dakle broj radnih jednica odgovara veličini indeksnog prostora. Svaka radna jedinica pokreće istu programsku jezgru, ali nad različitim podacima. Taj princip se naziva *SIMD*[5], u prijevodu *"jedna instrukcija, višestruki podaci"* . Radne jedinice su grupirane u radne grupe (engl. *working-group*) koje omogućuju veću zrnatost prilikom izrade algoritma.

Svaka radna jedinica se može jednoznačno identificirati u indeksnom prostoru pomoću globalnih indeksa (engl. *global ID*). Svaka radna grupa se također može identificirati pomoću indeksa grupa (engl. *work-group ID*). Radne jedinice se također mogu indentificirati i pomoću lokalnih indeksa (engl. *local ID*) i grupa kojim pripadaju, odnosno svaka radna jedinica ima i svoj jednoznačni lokalni indeks u grupi u kojoj se nalazi (jedna radna jedinica se može nalaziti

[5]U literaturi vezanoj uz GPU-ove se često nailazi na sličan izraz *SIMT* (engl. *Single Instruction, Multiple Thread*)

samo u jednog radnoj grupi).

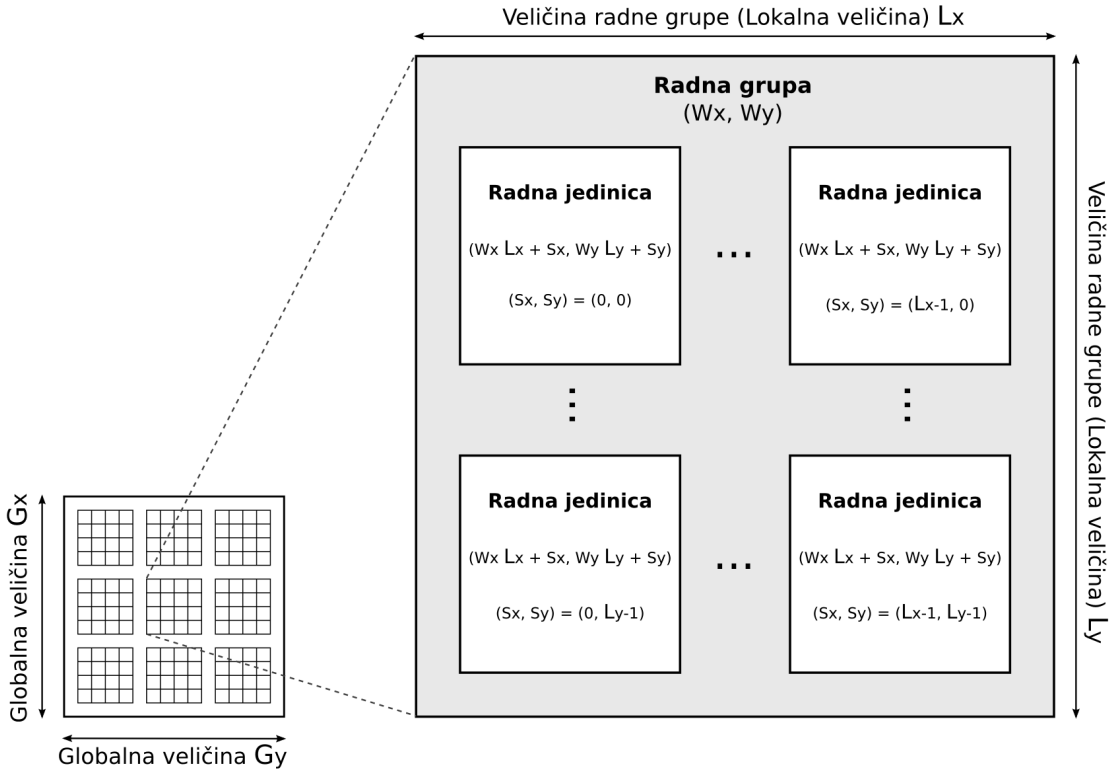
Pretpostavimo da koristimo dvodimenzionalni prostor za naš problem; prvo definiramo veličine dimenzija indeksnog prostora (G_x, G_y), a zatim veličine dimenzija lokalnog indeksnog prostora (L_x, L_y), a veličina radne grupe odgovara istim vrijednostima. Radne jedinice možemo dohvatiti preko globalnih (g_x, g_y) i lokalnih (s_x, s_y) indeksa. Veličinu radnih grupa označavamo sa (W_x, W_y), a njihove indekse sa (w_x, w_y). Sljedeće funkcije opisuju kako možemo dohvatiti željene indekse i veličine:

Određivanje globalnih indeksa radne jednice preko lokalnih indeksa i indeksa grupe:
$(g_x, g_y) = (w_x \times L_x + s_x, w_y \times L_y + s_y)$

Određivanje broja radnih grupa preko globalne i lokalne veličine indeksnog prostora:
$(W_x, W_y) = (G_x/L_x, G_y/L_y)$

Određivanje indeksa radne grupe u kojoj se radna jedinica nalazi, preko globalnih indeksa radne jedinice i veličine radne grupe:
$(w_x, w_y) = ((g_x - s_x)/L_x, (g_y - s_y)/L_y)$

OpenCL specifikacija propisuje postojanje funkcija za dohvaćanje globalne i lokalne veličine indeksnog prostora po dimenzijama, dohvaćanje globalnog i lokalnog indeksa radne jedinice, indeksa radne grupe i broja radnih grupa ovisno o dimenziji.



Slika 3: *OpenCL* izvršni model

Navedeni pristup znatno olakšava paralelizaciju algoritama, jer korisnik treba definirati veličinu problema po dimenzijama, a sustav se sam do neke mjere brine za raspodjelu posla i podataka, te olakšava korisniku kontrolu nad paralelizacijom. Korisnik može definirati lokalnu veličinu indeksnog prostora odnosno veličinu grupa, a ako to ne uradi sustav sam raspodjeljuje indeksni prostor među radnim grupama. Navedeni model omogućava preslikavanje dva programska modela, podatkovnu paralelizaciju i paralelizaciju podjelom na poslove.

Aplikacija na domaćinu stvara kontekst za izvođenje programskih jezgri na *OpenCL* uređajima. Tekst programske jezgre prevodi *OpenCL* prevodilac (engl. *compiler*) koji dolazi sa implementacijom. Kontekst sadrži sljedeće resurse:

- skup OpenCL uređaja koje domaćin koristi,
- programske jezgre, odnosno OpenCL funkcije koje se izvršavaju na OpenCL uređajima,
- programske objekte, odnosno tekst programa i izvršne datoteke koje sadrže programsku jezgru,
- memorijske objekte koji su vidljivi domaćinu i OpenCL uređajima.

Kontekstom upravlja domaćin preko funkcija koje definira OpenCL programsko sučelje . Domaćin potom stvara programski slijed putem kojeg se određuje redoslijed i način izvođenja programskih jezgi.

2.4.4 Programski model

OpenCL sustav podržava dva programska modela, podatkovnu paralelizaciju i paralelizaciju po raspodjeli poslova. Moguća je i kombinacija dva navedena modela, a primarni je podatkovna paralelizacija.

Prilikom *podatkovne paralelizacije* niz operacija se obavlja na višestrukim elementima memorijskog objekta. Ovisno o indeksnom prostoru stvaraju se radne jedinice i raspodjeljuju podaci za obradu među njima. Preslikavanje može biti jedan podatak na jednu radnu jedinicu, ali OpenCL dozvoljava da i više podataka obrađuje jedna radna jedinica. Broj radnih jedinica odgovara broju elemenata u indeksnom polju, a korisnik može eksplicitno definirati veličinu globalnog i lokalnog polja. Ako definira samo veličinu globalnog polja, sustav implicitno određuje veličinu lokalnog, podjela ovisi o implementaciji *OpenCL*-a. Za svaku radnu jedinicu se pokreće paralelno po jedna programska jezgra.

U modelu paralelizacije po poslovima programska jezgra se pokreće neovisno o indeksnom prostoru, a pretpostavljamo da jedna radna grupa sadrži samo jednu radnu jednicu. U ovom slučaju korisnik postiže paralelizam koristeći vektorske tipove podataka na *OpenCL* uređaju i stavljajući u red izvršavanja više poslova.

OpenCL poznaje dvije vrste sinkronizacije, radnih jedinica unutar iste radne grupe i stavljanjem poslova u red izvršavanja unutar istog konteksta. Sinkronizaciju radnih jedinica postižemo pozivanjem naredbe sinkronizacijske granice radne grupe (engl. *working-group barrier*). Pritom sve radne jedinice unutar radne grupe moraju pozvati naredbu sinkronizacije, te nakon toga mogu dalje nastaviti izvršavati naredbe u programskoj jezgri. Mehanizam sinkronizacije među radnim grupama ne postoji.

Sinkronizacija poslova u redoslijedu izvršavanja se također postiže pozivanjem barijere (engl. *command-queue barrier*), a pritom se čeka da svi poslovi u redu izvršavanja završe

i pospreme rezultate u memorijske objekte tako da postanu vidljivi. Također se može čekati na određeni posao u redu izvršavanja.

2.5 Pregled osnovnih funkcija sustava

Sustav *OpenCL* omogućuje aplikacijama korištenje domaćina i dostupnih *OpenCL* uređaja kao jedno heterogeno paralelno računalo. Sustav se sastoji od sljedećih komponenata:

- *OpenCL* platformskog sloja: omogućuje programima na domaćinu otkrivanje *OpenCL* uređaja i njihovih karakteristika, te stvaranje *OpenCL* konteksta;
- *OpenCL* izvršnog okruženja (engl. *runtime*): omogućuje programu na domaćinu upravljanje kontekstom;
- *OpenCL* prevodioca: služi za stvaranje programa koji sadrže programske jezgre koje se izvršavaju na *OpenCL* uređajima. Tekst programa se piše u *OpenCL C* programskom jeziku.

U ovom poglavlju će biti opisane samo relevantnije funkcije za osnovno korištenje *OpenCL*-a, tj. one koje su neophodne za kasnije primjere i njihovo objašnjenje.

2.5.1 OpenCL platformski sloj

Dohvaćanje informacija o platformi

Za dohvaćanje liste dostupnih platformi koristimo funkciju

```
cl_int clGetPlatformIDs (cl_uint num_entries,
                         cl_platform_id *platforms,
                         cl_uint *num_platforms)
```

Argument *num_entries* označava broj platformi koje mogu biti dodane u listu *platforms*. Funkcija u listu *platforms* postavlja identifikatore dostupnih platformi, a kroz *num_platforms* njihov broj. Ako je funkcija uspješno izvedena vraća vrijednost konstante *CL_SUCCESS*. Većina funkcija vraća iste vrijednosti ovisno o uspjehu tako da to neće biti posebno naznačeno kod opisa daljnih funkcija.

Dalje se kroz funkciju

```
cl_int clGetPlatformInfo (cl_platform_id platform,
                          cl_platform_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)
```

mogu dobiti detaljnije informacije o željenoj platformi tako da se identifikator platforme postavi u argument *platform.* Argument *param_name* sadrži konstantu koja identificira platformu, *param_value* je pokazivač na memorijski prostor u kojem je opis platforme, a *param_value_size* određuje veličinu toga prostora. Opis platforme sadrži podatke poput naziva, proizvođača i podržane verzije *OpenCL*-a .

Dohvaćanje informacija o uređajima

Za dohvaćanje liste uređaja na željenoj platformi se koristi funkcija

```
cl_int clGetDeviceIDs (cl_platform_id platform,
                       cl_device_type device_type,
                       cl_uint num_entries,
                       cl_device_id *devices,
                       cl_uint *num_devices)
```

U argument *platform* se postavlja identifikator platforme, kroz *device_type* se može filtrirati lista uređaja ovisno o njihovom tipu (CPU, GPU, ...), a preko argumenta *num_entries* se vraća broj dohvaćenih uređaja, tj. onih u koji se nalaze u listi *devices*.

Opširnije informacije o uređaju se dobivaju funkcijom

```
cl_int clGetDeviceInfo (cl_device_id device,
                        cl_device_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

Identifikator uređaja se određuje argumentom *device*, a argument param_name služi za dohvaćanje željene informacije o uređaju. Pokazivač *param_value* određuje memorijsku lokaciju u koju će se zapisati tražane informacije, a *param_value_size* definira njenu veličinu. Na taj način se mogu dobiti informacije o uređaju poput vrste, proizvođača, broja procesnih jedinica (engl. *compute units*), maksimalnom broju radnih jedinica po dimenziji indeksnog prostora, maksimalnoj veličini radnih grupa i slično.

Funkcije za baratanje kontekstom

OpenCL kontekst se koristi prilikom izvođenja *OpenCL* aplikacije za upravljanje memorijskim objektima (engl. *buffers*), programskim redovima (engl. *command-queues*) i pokretanjem programskih jezgri (engl. *kernels*) na uređajima (engl. *devices*).

Za stvaranje *OpenCL* konteksta koristimo funkciju

```
cl_context clCreateContext (const cl_context_properties *properties,
                            cl_uint num_devices,
                            const cl_device_id *devices,
                            void (*pfn_notify)(const char *errinfo,
                                               const void *private_info,
                                               size_t cb,
                                               void *user_data),
                            void *user_data,
                            cl_int *errcode_ret)
```

Argument *proporties* služi za postavljanje željenih opcija konektstu, može biti *NULL* i onda ovisi o implementaciji. Uređaje koje želimo da kontekst koristi dodajemo tako da njihove identifikatore stavimo u listu *devices*, a njihov broj stavimo u argument *num_devices*. Argumenti *pfn_notify* i *user_data* služe za prijavljivanje eventualnih grešaka i događaja koji se mogu dogoditi u kontekstu, može se postaviti i samo *NULL* pa se prijava takvih događaja zanemaruje. U *errcode_ret* se postavi kod eventualne greške ili uspjeha pri stvaranju konteksta, a funkcija nazad vraća kontekst koji je stvorila.

Također je dostupna funkcija *clCreateContextFromType* koja je jednostavnija verzija prethodne. Kod nje su argumenti *num_devices* i *devices* zamijenjeni sa *device_type* kojom se definira tip uređaja koji želimo koristiti, odnosno CPU i/ili GPU (*CL_DEVICE_TYPE_{CPU|GPU|ACCELERATOR|*

Informacije o stvorenom kontekstu se kasnije mogu dobiti funkcijom

```
cl_int clGetContextInfo (cl_context context,
                         cl_context_info param_name,
                         size_t param_value_size,
                         void *param_value,
                         size_t *param_value_size_ret)
```

U argument *context* se postavlja željeni kontekst, a u *param_name* se postavlja konstanta naziva parametra koji želimo dohvatiti. Rezultat se zapisuje na memorijsku lokaciju na koju pokazivač *param_value* pokazuje, a *param_value_size* definira njegovu veličinu.

2.5.2 OpenCL izvršno okruženje

Funkcije vezane uz programski slijed

Nakon što se stvori kontekst i objekti u njemu (memorijski, jezgre, ...), njima se upravlja preko programskog slijeda (engl. *command-queue*). U programski slijed se mogu stavljati naredbe koje će se redom izvoditi, a također je moguće definirati više programskih slijedova u koje će se stavljati naredbe i izvršavati neovisno jedna o drugoj. Prilikom korištenja više programskih slijedova se ne smiju djeliti isti objekti, točnije podaci u memoriji, među slijedovima zbog nedostatka sinkronizacije.

Programske slijedove stvaramo funkcijom

```
cl_command_queue clCreateCommandQueue (cl_context context,
                                       cl_device_id device,
                                       cl_command_queue_properties properties,
                                       cl_int *errcode_ret)
```

Argumentom *context* se određuje kontekst za koji se želi stvoriti programski slijed, a lista uređaja se određuje argumentom *device*, dok uređaji moraju biti povezani sa konekstom koji se koristi. Dodatne opcije se mogu definirati kroz argument *properties*, a ishod funkcije se stavlja u *errcode_ret*. U slučaju uspjeha funkcija vraća upravo stvoreni programski slijed.

Dodatne informacije o programskom slijedu se dobivaju kroz funkciju

```
cl_int clGetCommandQueueInfo (cl_command_queue command_queue,
                              cl_command_queue_info param_name,
                              size_t param_value_size,
                              void *param_value,
                              size_t *param_value_size_ret)
```

Argumentom *command_queue* se određuje željeni programski slijed preko njegovog indentifikatora, *param_name* označava informaciju koju želimo dohvatiti, pokazivač *param_value* pokazuje na memorijsku lokaciju na koju će biti upisan rezultat, a *param_value_size* veličinu memorijske lokacije.

Korištenje memorijskih objekata

Memorijski objekti mogu pripadati tipovima objekata međuspremnika (engl. *buffer object*) ili slika (engl. *image object*). Objekti slika neće biti obrađeni ovdje jer nisu bitni za kasnije primjere. Ukratko opisano, služe za stvaranje dvodimenzionalnih ili trodimenzionalnih slika, tekstura i međuspremnika slika za prikaz na zaslonu.

Objekti tipa međuspremnika, koji sa nadalje koriste, su jednodimenzionalni nizovi u koje se stavljaju podaci, poput cijelih brojeva i onih sa pomičnim zarezom, vektora podataka i struktura

podataka definiranih od strane korisnika. Služe za prijenos podataka između aplikacije na domaćinu i programskih jezgri koje se izvode na *OpenCL* uređajima. Podaci u objektima međuspremnika moraju biti podatkovnog tipa koji je čitljiv od stane aplikacije i programskih jezgri.

Za stvaranje objekta međuspremnika koristi se funkcija

```
cl_mem clCreateBuffer (cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *errcode_ret)
```

Pritom argument *context* određuje kontekst u kojem se stvara objekt. Atribut *flags* je niz bitova kojim se postavljaju određeni atributi za kreiranje objekta međuspremnika, poput prava pristupa i načina zauzimanja memorije. Argumentom *size* se određuje veličina međuspremnika u oktetima, *host_ptr* je pokazivač na memorijsku lokaciju na kojoj već mogu biti podaci namijenjeni međuspremniku. U *errcode_ret* će se postaviti eventualna greška nastala prilikom stvaranja međuspremnika. Funkcija u slučaju uspješnog izvođenja vraća objekt međuspremnika.

zastavica	opis
CL_MEM_READ_WRITE	Jezgra može čitati i pisati u mem. objekt
CL_MEM_WRITE_ONLY	Jezgra može samo pisati u objekt
CL_MEM_READ_ONLY	Jezgra može samo čitati iz objekta
CL_MEM_USE_HOST_PTR	Koristi se memorija domaćina za mem. objekt
CL_MEM_ALLOC_HOST_PTR	OpenCL implementacija zauzima memoriju za objekt dostupnu domaćinu
CL_MEM_COPY_HOST_PTR	OpenCL implementacija zauzima memoriju i u nju kopira sadržaj sa pokazivača *host_ptr*

Sljedeće funkcije su relativno slične, a služe za čitanje objekta međuspremnika i postavljanje tih podataka u memoriju domaćina, te upisivanje podataka u objekt međuspremnika iz memorije domaćina.

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t cb,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)

cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t cb,
                             const void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

Željni programski slijed u kojem će se operacija izvesti se određuje argumentom *command_queue*, a objekt međuspremnika argumentom *buffer*. Argumentima *blocking_write* i *blocking_read* se određuje hoće li ponašanje funkcije biti blokirajuće ili neblokirajuće, odnosno hoće li povratak iz funkcije uslijediti tek nakon izvršene operacije ili ne. Argument *offset* određuje pomak od memorijske lokacije međuspremnika, *cb* veličinu podataka koji se čitaju ili upisuju, a pokazivač *ptr* pokazuje na memoriju domaćina iz koje se čita ili u koju se upisuje. Argumentima *num_events_in_wait* list i *event_wait_list* se mogu odrediti događaji na koje funkcija mora pričekati da se izvrše, također mogu biti postavljeni na *NULL*. Kroz argument *event* funkcija vraća objekt događaja uz pomoć kojeg se kasnije može provjeriti je li naredba izvršena ili ne.

Funkcijom *clEnqueueCopyBuffer* se može sadržaj jednog objekta međuspremnika kopirati u drugi, a funkcijom *clReleaseMemObject* se smanjuje brojač memorijskih objekata, te navedeni objekt briše nakon što se izvedu naredbe u programskom slijedu kojem pripada.

Memorijski objekti se također mogu i preslikavati, tj. dio međuspremnika se može preslikati u memorijski prostor domaćina. Za to se koriste funkcije *clEnqueueMapBuffer* i *clEnqueueUnmapMemObject*.

Stvaranje programskih objekta

OpenCL program je skupina jedne ili više programskih jezgri. Deklariramo ih tako da u *OpenCL* programskom kodu ispred funkcije postavimo atribut *__kernel*. Programski objekt sadrži sljedeće informacije:

- informaciju kojem kontekstu pripada,
- izvorni tekst programa programa ili njegov prevedeni binarni zapis,
- izvršnu datoteku programa, listu uređaja za koje je prevedena, te opcije koje su pritom korištene,
- broj programskih jezgri koje se nalaze u njemu.

Programski objekt se stvara fukcijom

```
cl_program clCreateProgramWithSource (cl_context context,
                                      cl_uint count,
                                      const char **strings,
                                      const size_t *lengths,
                                      cl_int *errcode_ret)
```

Argumentom *context* se određuje kontekst u kojem se stvara objekt, lista pokazivača *strings*, duljine *count*, pokazuje na linije izvornog programskog koda, a njihove duljine su postavljene u listu *lengths*. U *errcode_ret* se postaviti eventualna greška nastala prilikom stvaranja programa. Funkcija u slučaju uspješnog izvođenja vraća programski objekt.

Osim stvaranja programskog objekta iz izvornog programskog koda, on se može stvoriti i iz binarnog zapisa uz pomoć funkcije *clCreateProgramWithBinary*.

Funkcijom *clReleaseProgram* se smanjuje brojač programskih objekata, te navedeni objekt uništava nakon što se unište sve programske jezgre povezane sa njim.

Stvaranje izvršnih datoteka programa

Funkcija

```
cl_int clBuildProgram (cl_program program,
                       cl_uint num_devices,
                       const cl_device_id *device_list,
                       const char *options,
                       void (*pfn_notify)(cl_program, void *user_data),
                       void *user_data)
```

se koristi za prevođenje i povezivanje izvršne datoteke programa iz izvornog teksta programa. Stvara se izvršna datoteka za sve *OpenCL* uređaje definirane u kontekstu kojem program pripada. Navedenu funkciju je obavezno pozvati nakon stvaranja programskog objekta sa nekom od prethodne dvije funkcije.

Argumentom *program* definira se prije stvoreni programski objekt, listom *device_list*, duljine *num_devices*, mogu se specificirati uređaji za koje je izvršna datoteka namijenjena, u slučaju *NULL* vrijednosti se uređaji određuju po povezanosti sa programskim objektom. Argumentom *options* se mogu postaviti neki dodatni parametri prevodiocu poput optimizacijskih tehnika, razine upozorenja i slično. Argumenti *pfn_notify* i *user_data* se mogu koristiti za pozivanje željene funkcije prilikom završetka prevođenja. Ovisno o ishodu prevođenja, funkcija vraća vrijednost koja može biti *CL_SUCCESS* u slučaju uspjeha ili neka druga konstanta koja opisuje tip greške.

Dodatne informacije o programskom objektu se mogu dobiti funkcijom

```
cl_int clGetProgramInfo (cl_program program,
                         cl_program_info param_name,
                         size_t param_value_size,
                         void *param_value,
                         size_t *param_value_size_ret)
```

Argument *program* specificira programski objekt, *param_name* naziv informacije koja se želi dohvatiti, a postavlja je na memorijsku lokaciju na koju pokazuje *param_value*, njena veličina je definirana argumentom *param_value_size*. Navedenom funkcijom se mogu saznati vrijednosti poput konteksta kojem objekt pripada, *OpenCL* uređaji sa kojima je povezan, te tekstualni i binarni zapis programa.

Status stvaranja programskog objekta se može saznati funkcijom *clGetProgramBuildInfo*, ovisno o *OpenCL* uređaju.

Stvaranje objekata programskih jezgri

Objekt programske jezgre stvaramo funkcijom

```
cl_kernel clCreateKernel (cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret)
```

gdje je argument *program* programski objekt za izvršnom datotekom, *kernel_name* naziv funkcije u izvornom kodu programske jezgre, a *errcode_ret* se koristi za prijavu identifikatora eventualnih grešaka. Funkcija u slučaju uspjeha vraća objekt programske jezgre.

Umjesto da stvaramo jedan po jedan objekt programske jezgre, mogu se stvoriti i svi odjednom u pojedinom programu uz pomoć funkcije *clGetProgramBuildInfo*.

Programske jezgre se stvaraju nakon što se stvori program sa odgovarajućim izvornim ili binarnim zapisom *OpenCL* programskog koda.

Izrazito je bitna funkcija za postavljanje argumenata programskoj jezgri, kako bi se jezgra kasnije mogla izvesti.

```
cl_int clSetKernelArg (cl_kernel kernel,
                       cl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value)
```

Argumentom *kernel* se postavlja objekt programske jezgre, *arg_index* služi za određivanje rednog broja argumenta u funkciji programske jezgre koji postavljamo, počevši od 0. Pokazivač *arg_value* pokazuje na vrijednost koja se želi proslijediti kao argument, najčešće je to memorijski objekt (međuspremnik) prije stvoren u istom kontekstu kao i program. U slučaju da argument označava lokalnu OpenCL memoriju (*__local*) se umjesto pokazivača stavlja *NULL*. Argument *arg_size* označava veličinu argumenta odnosno međuspremnika koji želimo proslijediti funkciji, u slučaju da se radi o lokalnoj memoriji gdje je vrijednost *NULL*, onda se preko *arg_size* označava veličina lokalne memorije koja će se zauzeti.

Dodatne informacije o objektu programske jezgre, poput konteksta i broja parametara, se mogu dobiti funkcijom *clGetKernelInfo*. Također je od koristi i funkcija

```
cl_int clGetKernelWorkGroupInfo (cl_kernel kernel,
                                 cl_device_id device,
                                 cl_kernel_work_group_info param_name,
                                 size_t param_value_size,
                                 void *param_value,
                                 size_t *param_value_size_ret)
```

kojom možemo dobiti informacije o objektu jezgre ovisno o *OpenCL* uređaju. Argumentom *kernel* se definira objekt jezgre, za uređaj postavimo *device*, a naziv parametra koji se želi dohvatiti kroz *param_name*. Rezultat se upisuje na memorijsku lokaciju na koju pokazuje *param_value*, a veličine je *param_value_size*. Parametrom *CL_KERNEL_WORK_GROUP_SIZE* se dohvaća maksimalana veličina radnih-grupa koje se mogu koristiti u dotičnoj jezgri na uređaju, a *CL_KERNEL_LOCAL_MEM_SIZE* količina lokalne memorij koju koristi jezgra.

Izvođenje programskih jezgri

Za stavljanje naredbe za izvršavanje programske jezgre u programski slijed koristi se funkcija

```
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                               cl_kernel kernel,
                               cl_uint work_dim,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event)
```

Argumentom *commad_queue* se određuje u kojem se programskom slijedu treba izvršiti programska jezgra, *OpenCL* uređaj je određen programskim slijedom. Argument *kernel* je objekt programske jezgre koji želimo izvesti. Broj dimenzija indeksnog prostora se postavlja argumentom *work_dim*, čija vrijednost mora biti između 1 i 3. *Global_work_offset* se mora u trenutnoj verziji *OpenCL*-a postaviti na *NULL*. Argumentima *global_work_size* i *local_work_size* se definira broj globalnih odnosno lokalnih radnih jedinica po dimenzijama, a svakom argumentu se pridružuje niz brojeva duljine jednake broju dimenzija. *Local_work_size* se može postaviti i na *NULL*, u kojem slučaju veličina lokalne radne grupe ovisi o *OpenCL* implementaciji. Na taj način se prilikom izvođenja specificira veličina indeksnog prostora i broj

globalnih i lokalnih radnih jedinica, tj. paralelizira se problem koji rješavamo. Argumentima *num_events_in_wait_list* i *event_wait_list* se mogu odrediti koji se događaji moraju završiti prije nego se navedena jezgra počne izvoditi. Kroz *event* se vraća objekt događaja koji identificira izvođenje programske jezgre, što se može koristiti prilikom redoslijeda izvođenja događaja, tj. paralelizacije po poslovima.

Programska jezgra se može izvesti i funkcijom *clEnqueueTask*, pri čemu je izvodi samo jedna radna jedinica, tj. ishod je isti kao da smo definirali samo jednu dimenziju veličine jedan.

Objekti događaja

Objekti događaja se mogu koristiti za praćenje izvođenja programskih jezgri, operacija pisanja i čitanja memorijskih objekata. Na taj način se može raspoređivati izvođenje poslova. Funkcijom

```
cl_int clWaitForEvents (cl_uint num_events,
                        const cl_event *event_list)
```

domaćin može pričekati da se završe događaji u listi *event_list*, duljine *num_events*.

Dodatne informacije o stanju objekta događaja se dohvaćaju funkcijom

```
cl_int clGetEventInfo (cl_event event,
                       cl_event_info param_name,
                       size_t param_value_size,
                       void *param_value,
                       size_t *param_value_size_ret)
```

Argumentom *event* se odredi objekt događaja, *param_name* naziv željene informacije, npr. *EXECUTION_STATUS* za praćenje izvršavanja. Rezultat se upisuje na mjesto pokazivača *param_value*, veličine *param_value_size*.

Čekanje na izvršenje jednog ili više događaja se postiže funkcijom

```
cl_int clEnqueueWaitForEvents (cl_command_queue command_queue,
                               cl_uint num_events,
                               const cl_event *event_list)
```

Definira se željeni programski slijed argumentom *command_queue*, lista događaja koji se žele pričekati se stavlja u listu *event_list*, duljine *num_events*. Objekti događaja moraju biti dobiveni iz neke od prethodnih funkcija, poput *clEnqueueNDRangeKernel* i *clEnqueue{Read|Write}Buffer*.

Čekanje na kraj izvršavanja svih poslova u programskom slijedu se postiže funkcijom

```
cl_int clEnqueueBarrier (cl_command_queue command_queue)
```

Vrijeme duljine izvođenja programskih jezgri odnosno događaja se može pratiti funkcijom

```
cl_int clGetEventProfilingInfo (cl_event event,
                                cl_profiling_info param_name,
                                size_t param_value_size,
                                void *param_value,
                                size_t *param_value_size_ret)
```

U argument *event* se postavi objekt željenog događaja, sa *param_name* se definira vrijednost koja se želi dohvatiti, npr. *CL_PROFILING_COMMAND_START* i *CL_PROFILING_COMMAND_*, a povratna vrijednost se upisuje u memoriju pokazivača *param_value*, veličine *param_value_size*.

2.5.3 OpenCL C programski jezik i prevodilac

OpenCL C je proširenje ISO C99 programskog jezika koji se najčešće naziva samo *"C"*. Točnije osim što je jezik proširen, tako su i neke njegove funkcionalnosti izbačene, tj. nisu dozvoljene, najčešće zbog načina izvođenja OpenCL programskih jezgri. To su pokazivači na funkcije, mogućnost rekurzije, varijabilne duljine nizova, te neke druge specifičnosti. S druge strane proširenja uključuju rad sa radnim jedinicama i grupama, vektorske tipove podataka, mogućnost sinkronizacije, te općenito neka proširenja za paralelizam, baratanje sa slikama, povezivanje sa OpenGL sustavom i tako dalje.

Na raspolaganju su skalarni tipovi podataka koje uobičajeno posjeduju i drugi programski jezici, poput znakova (*char*), cijelih brojeva (*short*, *ushort*, *int*, *uint*, *long*, *ulong*) i brojeva sa pomičnim zarezom (*half*, *float*), zatim vektorski tipovi podataka mogućih duljina 2, 4, 8, i 16 (*char2*, *short4*, *int8*, *float16*, ...) i neke predefinirane konstante. Naravno cijeli skup matematičkih i logičkih operatora nad tipovima podataka je na raspolaganju, funkcije za konverziju među različitim tipovima podataka, uobičajene matematičke funkcije poput trigonometrijskih, zatim neke specifičnije funkcije za baratanje geometrijom, rad sa slikama i povezivanje sa *OpenGL*-om.

Bitno je posebno izdvojiti atribute (kvalifikatore) koji se mogu pridružiti varijablama, a definiraju adresni prostor kojem varijabla pripada.

Tablica 3: *OpenCL C*: memorijski atributi varijabli

kvalifikator	**opis**
__global	referencira se na memorijske objekte poput međuspremnika u globalnom memorijskom prostoru
__local	opisuje varijable u lokalnom memorijskom prostoru, mogu im pristupati radne jedinice unutar iste radne grupe
__constant	definira varijable u globalnoj memoriji kojima mogu pristupati sve radne jedinice, ali samo sa pravom čitanja
__private	sve varijeable unutar funkcija programske jezgre ili koje su kao argumenti prosljeđene pripadaju privatnom adresnom prostoru

Također su neophodne funkcije za baratanje radnim jedinicama.

Tablica 4: *OpenCL C*: funkcije za indeksni prostor

funkcija	**opis**
get_work_dim()	vraća broj dimenzija adresnog prostora, do 1 do 3
get_global_size(D)	vraća broj globalnih radnih jedinica za dimenziju D
get_global_id(D)	vraća jedinstven identifikator radne jedinice za dimenziju D
get_local_size(D)	vraća broj lokalnih radnih jedinica za dimenziju D, tj. koliko ih se nalazi u jednoj radnoj grupi po dimenziji D
get_local_id(D)	vraća jedinstven identifikator radne jedinice u radnoj grupi za dimenziju D, radne jedinice u različitim radnim grupama mogu imati iste identifikatore
get_num_groups(D)	vraća broj radnih grupa po dimenziji D
get_group_id(D)	vraća jedinstveni identifikator radne grupe za dimenziju D

Uz pomoć navedenih funkcija paralelizira se obavljanje posla u programskim jezgrama, odnosno ovisno o identifikatoru radnih jedinica se mogu specificirati podaci koje će ona obrađivati.

Za sinkronizaciju su dostupne funkcije u tablici 5 .

Tablica 5: OpenCL C: sinkronizacijske funkcije

funkcija	**opis**
barrier(*flags*)	sve radne jedinice u istoj radnoj grupi čekaju
mem_fence(*flags*)	sva čitanja i pisanja po memoriji se čekaju
read_mem_fence(*flags*)	čeka se samo završetak čitanja iz memorije
write_mem_fence(*flags*)	čeka se samo završetak pisanja u memoriju

Svaka funkcija za argument prima konstante (zastavice) *CLK_LOCAL_MEM_FENCE* i *CLK_GLOBAL_MEM_FENCE*, te njihovu kombinaciju, njima se određuje radili li se o globalnoj i/ili lokalnoj memoriji.

OpenCL C je osmišljen s ciljem olakšavanja paralelizacije po podacima i poslovima, a zadržan je na niskoj razini kako bi bio efikasan što se tiče performansi, jer je to područje kojem je OpenCL namijenjen. Prevodi ga OpenCL prevodilac koji dolazi uključen u OpenCL implementaciji.

2.6 Ilustracija OpenCL-a na primjeru

Sljedeći primjer ukratko ilustrira općenitu *OpenCL* aplikaciju koja sadrži tekst programa domaćina i programske jezgre. Aplikacija nema neku posebnu namjenu, već samo zbraja elemente dva ulazna niza i rezultat vraća u izlaznom nizu. Zbrajanje se u ovom slučaju odvija na GPU uređaju. Ulazni nizovi se na domaćinu alociraju i popunjavaju slučajnim podacima , zatim se prosljeđuju uređaju koji obavlja zbrajanje elemenata, a rezultate sa uređaja domaćin kopira u svoju memoriju te ih ispisuje.

Za ovu zadaću je potrebno stvoriti kontekst, programski slijed, memorijske objekte međuspremnika i programa, prevesti programsku jezgru za željene *OpenCL* uređaje, postaviti argumente i izvesti programsku jezgru, osloboditi zauzetu memoriju i uništiti stvorene objekte. Ovaj primjer prikazuje sve neophodne radnje za stvaranje *OpenCL* aplikacije. Svaka naredba u programskom kodu je dokumentirana, a zbog jednostavnosti i čitljivosti nema nikakve provjere uspješnosti izvođenja funkcija i njihovih rezultata, poput neuspjelog stvaranja konteksta i programskog slijeda, memorijskih objekata, alociranja memorije i slično.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <CL/cl.h>

#define SIZE 1024
// Izvorni tekst programa OpenCL programske jezgre
const char* oclSource[] = {
  "__kernel void sum_arrays(__global int *in_a, "
  "                         __global int *in_b, __global int *out_c)",
  "{",
  "    uint n = get_global_id(0);",
```

```
  "    out_c[n] = in_a[n] + in_b[n];",
  "}",
};

int main (int argc, char **argv) {

  // Alokacija memorije za dva ulazna niza
  int array_1[SIZE], array_2[SIZE];
  // Inicijalizacija nasumičnih vrijednosti ulaznih nizova
  srandom(time(0));
  for (int i=0; i<SIZE; ++i) {
    array_1[i] = ((float)random() / RAND_MAX) * 100;
    array_2[i] = ((float)random() / RAND_MAX) * 100;
  }

  // Stvaranje OpenCL konteksta za GPU OpenCL uređaje
  cl_context oclContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                                  NULL, NULL,NULL);

  // Dobivanje veličine memorije za listu uređaja u kontekstu
  size_t paramSize;
  clGetContextInfo(oclContext, CL_CONTEXT_DEVICES, 0, NULL, &paramSize);

  // Dobivanje liste uređaja u kontekstu
  cl_device_id* oclDevices = (cl_device_id*)malloc(paramSize);
  clGetContextInfo(oclContext, CL_CONTEXT_DEVICES, paramSize,
                   oclDevices, NULL);

  // Stvaranje programskog slijeda u kontekstu za prvi uređaj
  cl_command_queue oclCommandQueue =
          clCreateCommandQueue(oclContext, oclDevices[0], 0, NULL);

  // Stvaranje memorijskih objekata međuspremnika koje uređaj može
  // samo čitati; Alokacija memorije na uređaju i kopiranje
  // vrijednosti iz ulaznih nizova na domaćinu
  cl_mem oclInBuffer_1 = clCreateBuffer(oclContext, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, array_1, NULL);
  cl_mem oclInBuffer_2 = clCreateBuffer(oclContext, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, array_2, NULL);

  // Stvaranje memorijskog objekta međuspremnika,
  // uređaj može samo pisati u njega
  cl_mem oclOutBuffer = clCreateBuffer(oclContext, CL_MEM_WRITE_ONLY,
                                       sizeof(int) * SIZE, NULL, NULL);

  // Stvaranje objekta programa iz izvornog programskog koda
  cl_program oclProgram =
        clCreateProgramWithSource(oclContext, 5, oclSource, NULL, NULL);

  // Prevođenje programa za sve uređaje u konektstu
  // (engl JIT, Just In Time Compilation)
  clBuildProgram(oclProgram, 0, NULL, NULL, NULL, NULL);

  // Stvaranje poveznice sa funkcijom (sum_arrays) unutar programske jezgre
  cl_kernel oclSumArrays = clCreateKernel(oclProgram, "sum_arrays", NULL);
```

```
  // Postavljanje argumenata za programsku jezgru u GPU memoriju
  clSetKernelArg(oclSumArrays, 0, sizeof(cl_mem), (void*)&oclInBuffer_1);
  clSetKernelArg(oclSumArrays, 1, sizeof(cl_mem), (void*)&oclInBuffer_2);
  clSetKernelArg(oclSumArrays, 2, sizeof(cl_mem), (void*)&oclOutBuffer);

  // Pokretanje izvođenja programske jezgre
  // Argumentima se specificira programski slijed, objekt
  // programske jezgre i veličina indeksnog prostora
  size_t WorkSize[1] = {SIZE};
  clEnqueueNDRangeKernel(oclCommandQueue, oclSumArrays, 1, NULL,
                         WorkSize, NULL, 0, NULL, NULL);

  // Alokacija memorije na domaćinu za rezultate
  // Kopiranje vrijednosti iz objekta međuspremnika u memoriju domaćina
  int array_res[SIZE];
  clEnqueueReadBuffer(oclCommandQueue, oclOutBuffer, CL_TRUE,
                      0, sizeof(int) * SIZE, array_res,0, NULL, NULL);

  // Ispis rezultata
  for (int i=0; i<SIZE; i++)
    printf("%d\t%d\t%d\n", array_1[i], array_2[i], array_res[i]);

  // Oslobađanje memorije objekata
  free(oclDevices);
  clReleaseKernel(oclSumArrays);
  clReleaseProgram(oclProgram);
  clReleaseCommandQueue(oclCommandQueue);
  clReleaseContext(oclContext);
  clReleaseMemObject(oclInBuffer_1);
  clReleaseMemObject(oclInBuffer_2);
  clReleaseMemObject(oclOutBuffer);
  return 0;
}
```

Navedeni primjer koristi prvi dostupni GPU od strane *OpenCL*-a. Za to je potrebno imati instalirane upravljačke programe za grafičke kartice koji podržavaju *OpenCL*, kao što je prije navedeno. To su trenutno *"novije"* grafičke kartice sa mikroprocesorima kompanija *Nvidia* i *AMD/ATI*. Korištenje GPU-a u primjeru može se jednostavno zamijeniti sa CPU-om.

Za prevođenje izvornog programskog koda u izvršni program se pod *GNU/Linux* operacijskim sustavom koristi *GNU C Compiler* (*gcc*), odnosno njegova varijanta za C++ (*g++*). Pritom je potrebno definirati putanju do datoteka zaglavlja (engl. *headers*) i povezati ga sa *OpenCL* bibliotekama.

Primjer za razvojni paket od *Nvidie*:

```
g++ -Wall -I ~/NVIDIA_GPU_Computing_SDK/OpenCL/common/inc/ \
    -lOpenCL opencl-test.cpp -o opencl-test
```

Općeniti dijagram tijeka razvoja *OpenCL* aplikacije je prikazan na slici 4.

Slika 4: Tijek razvoja *OpenCL* aplikacije

3 Druge tehnologije za paralelizaciju

U ovom je poglavlju dan kratak opis drugih tehnologija za paralelizaciju algoritama. Navedene su gotovo sve koje su danas relevantne, bez obzira na arhiketuru i svrhu kojoj su namijenjene. Sve imaju dovoljno dodirnih točaka sa *OpenCL*-om za usporedbu, ali nijedna ne omogućava korištenje na heterogenim platformama, već su sve ciljane na točno određene arhitekture mikroprocesora. Svaka tehnologija je ukratko opisana i uspoređena sa *OpenCL*-om, istaknute su njihove različitosti, nedostaci i prednosti kako bi se istaknula područja u kojima je bolje koristiti jednu ili drugu tehnologiju.

3.1 CUDA

Compute Unified Device Architecture, skraćeno *CUDA*, je sustav razvijen od strane kompanije *Nvidia*, a namijenjen izvršavanju programskog koda opće namjene (*GP/GPU*) na grafičkim mikroprocesorima iste kompanije. Na taj način je *Nvidia* omogučila širu primjenu svojih mikroprocesora, osim prikaza zahtjevne računalne grafike, za općenito zahtjevne matematičke kalkulacije. Sustav je relativno novijeg datuma, prvo javna verzija je postala dostupna u veljači 2007. godine, podržani su *"noviji"* mikroprocesori, točnije oni *GeForce8* serije i noviji, a na operacijskim sustavima *GNU/Linux*, *Windows* i *Mac OS X*.

Sustav je SIMD modela, točnije SIMT (*engl. single instruction, multiple thread*), model koji se učestalo koristi za grafičke mikroprocesore zbog njihove arhitekture. Paralelizirani algoritam se također sastoji od programskih jezgri koje se izvode na jezgrama mikroprocesora u velikoj količini dretvi. Sustav poznaje dijeljenu memoriju (engl. *shared memory*) smještenu na samom GPU-u i lokalnu memoriju[6] koja je smještena na grafičkoj kartici uz GPU. Sustav se može koristiti iz mnogih programskih jezika, *C/C++*, *Fortran*, *Python*, *Java* i *Matlab*. Tekst programa za jezgre se piše u *C*-u proširenom za *CUDA*-u. Sustav koristi modificirani *Open64* programski prevodilac za prevođenje izvornog programskog koda u strojni kod nazvan *Parallel Thread eXecution* (*PTX*). *PTX* je strojni kod koji se izvodi na grafičkim mikroprocesorima od *Nvidie*. Kroz *CUDA*-u je *Nvidia* stvorila sloj apstrakcije nad arhitekturom svojih mikroprocesora, što im omogućuje lagano unapređivanje mikroprocesora, a da se ne dovodi u pitanje izvođenje i kompatibilnost aplikacija napisanih za *CUDA*-u. Cilj *CUDA*-e je pružiti relativno jednostavan pristup iskorištavanju velikog računalnog potencijala GPU-a.

Nvidia je svoju implementaciju *OpenCL*-a temeljila na *CUDA* sustavu, zapravo izvođenje *OpenCL* programskih jezgri je gotovo istovjetno onima u *CUDA*-i, time su i performanse oba sustava u istom redu veličine. Glavna prednost *OpenCL*-a je prenosivost, tj. da se tekst programa može izvršavati i na drugim platformama osim GPU-a.

3.2 ATI Stream SDK

Početna inačica *Stream SDK* paketa se zvala *Close To the Metal* (*CTM*), a bila je namijenjeno isključivo za *GP/GPU* računarstvo. Kako naziv nije zaživio na dulje vrijeme, promijenjen je u *Stream*, a ciljano područje primjene je prošireno. *CTM* je zapravo bio *ATI*-jev pandan *CUDA*-i od *Nvidie*, kako obje kompanije proizvode grafičke mikroprocesore njihova arhitektura

[6] Lokalnu memoriju kod sustava *CUDA* ne treba mješati sa lokalnom memorijom *OpenCL*-a, već ona odgovara globalnoj memoriji *OpenCL*-a, a dijeljena memorija lokalnoj *OpenCL* memoriji.

je uglavnom identična, te su i sustavi iste svrhe, tj. omogućiti *GP/GPU*. Kasnije se *AMD/ATI* odlučio *Stream* iskoristi za implementaciju *OpenCL*-a na svojim mikroprocesorima.

Stream softverski paket osim što implementira *OpenCL* za GPU-ove kompanije *ATI*, također je ponudio i *OpenCL* implementaciju za centralne mikroprocesore kompanije *AMD*, točnije sve x86_64 mikroprocesore sa SSE3 skupom instrukcija što uključuje i *Intel*. Time je *AMD/ATI Stream* prvi ponudio implementaciju *OpenCL*-a za obje platforme na *GNU/Linux* i *Windows* operacijskim sustavima, nakon Appleove implementacije u *Mac OS X*-u.

3.3 MPI

Message Passing Interface je programsko sučelje (engl. *Application Programming Interface, API*) koje omogučava međusobnu komunikaciju više računala, ali i komponenata, točnije centralnih mikroprocesora, unutar jednog računala. Sustav je namijenjen razvoju aplikacija koje će se izvoditi na skupu računala sa raspodijeljenom memorijom, riječ je o SPMD modelu (engl. *Single Process, Multiple Data*), tj. sva računala izvode isti program, ali na različitim podacima. Komunikacija, točnije razmjena podataka, se odvija porukama među procesima koji se izvršavaju na centralnim mikroprocesorima odnosno njihovim jezgrama. Razmjena poruka između različitih računala se ostvaruje upotrebom TCP mrežnih paketa. Sustav podržava više tipova komunikacije poput okupljanja procesa u komunikatore i razmjenu poruka *jedan-na-jedan*, te blokirajuću i neblokirajuću sinkronizaciju.

Razlikuju se dvije inačice, MPI-1 nastao 1994. godine i MPI-2 (1997. g.) koji je proširenje MPI-1 inačice, time je zadržana kompatibilnost izvođenja MPI-1 programa na MPI-2 sustavu. *MPI* se nametnuo kao najraširenij standard u industriji za paralelizaciju izvođenja poslova, dvije najpoznatije implementacije su *MPICH* (*http://www.mcs.anl.gov/mpi/mpich/*) i *OpenMPI* (*http://www.open-mpi.org/*). Sam standard je nastao kroz suradnju većeg broja kompanija, te je otvorenog tipa. Dostupan je u svim relevantnim operacijskim sustavima danas i podržan u većini programskih jezika (*C*, *C++*, *Fortran*, *Java*, *Python*, ...), te razvojnih okruženja. *MPI* je najzastupljeniji u računarstvu visokih performansi (*HPC*) poput računalnih grozdova (engl. *cluster*), spletova (engl. *grid*) i općenito superračunala (engl. *supercomputer*).

Nedostak *MPI*-a u odnosu na *OpenCL* su ciljane arhitekture, tj. *MPI* je namijenjen isključivo za centralne mikroprocesore arhitektura x86 i x86_64, a *OpenCL* općenito za heterogene platforme (CPU, GPU, ...). *MPI* izvodi poslove kroz pokretanje procesa, dok se *OpenCL* više oslanja na dretve, što je učinkovitije kod višejezgrenih mikroprocesora, poput GPU-a i višejezgrenih CPU-a. Nedostatak *OpenCL*-a je što standard ne specificira nikakvu razmjenu podataka između više računala, već samo između uređaja unutar jednog računala, također je *MPI* daleko stariji standard i time znatno raširenij, što *OpenCL*-u ne treba uzimati kao nedostatak iz razloga što još nije ni u potpunosti podržan na svim platformama.

Iako su oba sustava osmišljena sa različitim ciljevima dodirna točka im je paralelizacija algoritama koji će se izvoditi unutar jednog računala, te se oba mogu koristiti u tu svrhu.

3.4 OpenMP

Open Multi-processing je poput *MPI*-a programsko sučelje za paralelizaciju algoritama, ali temeljeno na zajedničkoj memoriji i višedretvenosti (engl. *multithreading*). Za razliku od *MPI*-a barata sa dretvama umjesto procesa, te je namijenjeno izvedbi na jednom računalu, a ne skupu računala. Standard je također otvorenog tipa i definiran od strane više kompanija na

području sklopovlja i softvera.

Moguća je kombinacija sa sustavom *MPI* kako bi se dobio hibridni model kojim se kroz *MPI* razmjenjuju poruke između više računala namijenjene *OpenMP* programima na svakom računalu. *OpenMP* se temelji na višedretvenosti, tj. omogućava stvaranje paralelnih algoritama tako da se poslovi ili podaci raspodijele između više dretvi koje se paralelno izvode na dostupnoj platformi odnosno jezgrama CPU-a. Sustav je također široke primjene i podržan na većini operacijskih sustava i programskih jezika, te razvojnih okružja. Prva inačica se pojavila 1997. godine, a zadnja (3.0) 2008. godine.

OpenMP je dosta bliži svojim načinom rada *OpenCL*-u od *MPI*-a. Oba sustava barataju sa dretvama i zajedničkom memorijom, tj. *OpenCL* barata zajedničkom memorijom unutar istog *OpenCL* uređaja. Prednost *OpenCL*-a je opet podrška različitim platformama, dok *OpenMP* cilja samo na CPU. Oba sustava pružaju visoke performanse i relativno jednostavnu paralelizaciju algoritama, te su prenosivi. *OpenMP* je raširenij i podržaniji standard iz razloga što je dulje u primjeni, što se ne može pripisati kao manjak *OpenCL*-u. Paralelizacija je ipak nešto složenija na *OpenCL*-u iz razloga podrške različitim platformama, tj. moraju se prvo stvoriti programski konteksti i slijedovi, memorijski objekti i programske jezgre. *OpenMP* je zapravo proširenje već postojećih programskih prevodilaca, poput *GCC*-a (engl. *GNU C Compiler*), te mu se kroz predprocesorske naredbe u programskom kodu specificiraju dijelovi koji se trebaju izvoditi paralelnog odnosno višedretveno.

3.5 Cell BE

Cell Broadband Engine je nastao na sasvim novoj arhitekturi mikroprocesora *Cell* (*http://www.research.ibn*
razvijenih od strane *IBM*-a, *Sonya* i *Toshibe*. Mikroprocesor je kombinacija uobičajenih mikroprocesora opće namjene i specijaliziranih mikroprocesora, poput onih grafičkih od *Nvidie* i *ATI*-a. Sastoji se od jedne jezgre opće namjene, zvane *PPE*, koja je bazirana na *PowerPC* arhitekturi, te osam jezgri, zvanih *SPE*, koje su specijalizirane za brzu obradu velikih količina podataka. U praksi se mikroprocesor pokazao izuzetno brzim u određenim poljima spram mikroprocesora opće namjene.

Sustav pruža programsko sučelje za upravljanje mikroprocesorima, podržani su programski jezici *C* i *C++*. Posao se izvodi tako da se *PPE* jezgra koristi za operacijski sustav i raspodjelu poslova i njihovu sinkronizaciju, koji se zatim izvode na *SPE* jezgrama. *Cell* mikroprocesore se može pronaći u *Sony PlayStation3* igračim konzolama, superračunalima i poslužiteljima razvijenim od strane *IBM*-a, te specijaliziranim karticama koje se mogu ugraditi u standardno računalo. Kako je riječ o poprilično specifičnoj tehnologiji koja se oslanja isključivno na *Cell* mikroprocesore, njezina zastupljenost je ograničena. Šira primjena je ostvarena kroz *GNU/Linux* distribuciju *Yellow Dog Linux* koja se može instalirati na *PlayStation3*, te je čest slučaj povezivanje više konzola u grozd, time se za relativno malu cijenu mogu ostvariti izrazito velike računalne performanse.

Cell BE se u praksi pokazao kao izrazito dobra tehnologija što se tiče performansi, na kojoj *IBM* i dalje radi te ju unapređuje. Nažalost tehnologija nije lako dostupna široj populaciji jer ovisi isključivo o *Cell* mikroprocesorima, najjednostavniji i najjeftiniji način je nabavka *PlayStation3* konzole. Kako *IBM* isto sudjeluje u razvoju *OpenCL* sustava, za očekivati je da će uskoro izdati podršku za *Cell* mikroprocesore, čime će se i njihova primjena raširiti.

3.6 DirectCompute

DirectCompute je tehnologija namijenjena za *GP/GPU*, a razvijena od strane *Microsofta*. Sadržana je u *DirectX* sustavu verzije 11, koji se pojavio krajem 2009. godine za operacijske sustave *Windows Vista* i *Windows 7*. Podržani su grafički mikroprocesori koji podržavaju *DirectX 10* i *11*. *Nvidia* i *AMD* su već najavili svoju podršku ovoj tehnologiji. Kako je tehnologija nešto novija čak i od *OpenCL*-a teško je dati neku širu analizu, nedostaci su podrška samo za GPU, te ograničenost na novije verzije *Windows* operacijskog sustava.

3.7 Intel Ct

Intel je 2009. godine, kupnjom kompanije *RapidMind*, stekao vlasništvo i nad tehnologijom *RapidMind Multi-core Development Platform*. Riječ je o platformi koja se sastoji od *C++* biblioteka koje omogučuju podatkovnu paralelizaciju algoritama, ciljane platforme su *CPU*, *GPU* i *Cell*. *Intel* je na temelju te tehnologije izgradio svoju *Ct* tehnologiju, za koju navodi kako je tekst programa prenosiv i kroz sustav izvodiv na heterogenim platformama, jednostavan za implementaciju željenih aplikacija jer se oslanja na *C++*, te visokih performansi.

Kako je sustav trenutno u fazi testiranja (beta) veće usporedbe sa drugim tehnologijama nisu moguče. U svakom slučaju treba pratiti razvoj ovog sustava/platforme jer po opisu djeluje poprilično slično *OpenCL*-u, zapravo je uz *OpenCL* jedini sustav koji cilja na heterogene platforme. Moguća mana u razvoju bi mu mogla biti ovisnost o jednoj kompaniji, dok je *OpenCL* nastao kroz konzorcij u kojem sudjeluju svi proizvođači sklopovlja.

4 Arhitekture mikroprocesora

Pojavom osobnog računala (engl. *personal computer, PC*) i njegovom popularizacijom 80' i 90' godina prošlog stoljeća je računalna snaga mikroprocesora postala dostupna i široj populaciji. Do onda su *"brza"* (super)računala bila dostupna samo uskim akademskim krugovima, te zatvorenim ustanovama poput nekih kompanija i vladinih agencija. Kroz vrijeme su se razni proizvođači mikroprocesora natjecali tko će ponuditi brži, učinkovitiji i povoljniji mikroprocesor kako bi osvojili veći dio tržišta.

Pravilo koje poprilično precizno opisuje napredak razvoja mikroprocesora je Mooreov zakon[7], a kaže da se broj tranzistora na mikroprocesoru udvostruči svaka 24 mjeseca. Zakon se pokazao ispravnim sve do danas, a vjerovatno će se nastaviti još nekoliko godina dok se ne dođe do fizičke granice smanjivanja tranzistora i problema pretjerane disipacije topline.

Time su broj tranzistora i radni taktovi dulji niz godina samo rasli, pojavljivale se nove arhitekture, postojeće arhitekture se proširivale, ali neke i padale u zaborav zajedno sa kompanijama koje su stajale iza njih. Također se i svrha mikroprocesora mijenjala, dok se je prije cjelokupna obrada podataka izvodila samo na centralnim mikroprocesorima, u drugoj polovici 90' su se pojavili i grafički mikroprocesori drugačije svrhe i arhitekture.

Danas osobna računala ostvaruju rezultate kakve su prije par desetak godina mogla samo superračunala. Uobičajeno je imati višejezgreni centralni mikroprocesor u svoj računalu, te u većini slučajeva i grafički mikroprocesor sa mogučnošću akceleracije 3D grafike. Postavlja se pitanje kako u potpunosti, na što jednostavniji način, iskoristi taj potencijal, bilo da se radi o računarstvu visokih performansi ili nekim općenitijim radnjama, poput dekodiranja videa visoke kvalitete, sve zahtjevnijim web aplikacijama ili pak računalnim igrama.

Bitno je napomenuti kako porast potrebe za računalnom snagom, recimo u slučaju web aplikacija, često nije uzrokovan kompeksnošću algoritama, već visokim stupnjem programske apstrakcije. Ako se neka aplikacija realizira na niskom sloju, poput strojnog jezika ili *C*-a, može se očekivati velika učinkovitost, ali ako se radi o aplikaciji napisanoj u nekom dinamičnom jeziku višeg razreda koji komunicira sa nižim slojevima kroz nekakvo virtualno sučelje, performanse će naravno padati, čak i za nekoliko redova veličine. Naravno odabir višeg programskog jezika ima prednost kraćeg vremena potrebnog za razvoj, veće fleksibilnosti u naknadnim promjenama i prenosivosti. Zato je bitno prilikom razvoja neke aplikacije ili sustava dobro procjeniti odabir programskog jezika, moguće je i njihovo kombiniranje ovisno o kritičnim djelovima aplikacije. Dobar primjer je korištenje *pyOpenCL* implementacije za postavljanje okruženja za izvođenje programskih jezgri, dok su one i dalje zadržane u *OpenCL C*-u kako bi postizale visoke performanse.

Zadnjih godina je poseban naglasak stavljen na paralelizaciju obrade podataka na mikroprocesoru. Razlog je dostizanje granice postojeće tehnologije prilikom izvođenja slijednog niza naredbi, to je potaknulo pojavu višejezgrenih centralnih mikroprocesora, dok su grafički mikroprocesori u samom početku više orijentirani na paralelizam zbog primarne svrhe kojoj su namijenjeni. Proces paralelizacije slijednog algoritma nije jednostavan, neki alogritmi se po svojoj prirodi trivijalno paraleliziraju, dok neke uopće nije moguće. Danas ne postoji nikakvo posve automatizirano rješenje navedenog problema već pada u domenu programera, a programski prevodioci mogu do neke mjere učinkovito prevesti izvorni tekst programa u različite strojne kodove ovisno o ciljanoj platformi, npr. *OpenCL Just-in-Time* prevodilac. Sama učinkovitost optimiranja prevođenja programskog koda u niz instrukcija je bitna odlika prevodilaca.

[7] Gordon Moore je jedan od suosnivača kompanije Intel, a svoje zapažanje je iznio oko 1970. godine.

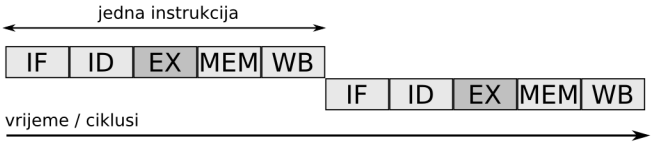
Ovdje će detaljnije biti opisani centralni mikroprocesori arhitekture x86, odnosno njene 64-bitne varijante x86_64, kakve danas nalazimo u svim osobnim računalima, ali i superračunalima. Grafički mikroprocesori će biti ilustrirani na primjeru kompanije *Nvidia*, te *Cell* mikroprocesor kompanije *IBM*, za koje bi se moglo reći da su hibrid prethodne dvije arhitekture. Riječ je o mikroprocesorima koji su najrašireniji i lako dostupni, a visokih performansi, naravno tu priča o mikroprocesorima ne staje, ali prije navedeni ulaze u područje koje je zanimljivo za paralelizaciju algoritama.

Osim ovdje navedenih danas su dostupne arhitekture poput *ARM*-a koji se nalaze u mobilnim uređajima i *MIPS*-a koja je česta kod ugradbenih i mrežnih računala, poput usmjerivača mrežnog prometa (engl. *router*). Također su još uvijek žive neke arhitekture namijenjene isključivo poslužiteljima, poput *PA-RISC* mikroprocesora kompanije *HP* i *Sparc* od *Sun Microsystemsa*.

Cilj je ukratko opisati trenutno dostupne arhitekture na kojima su ostvareni mikroprocesori visokih performansi, dati kratak uvid u neke arhitekture koje tek dolaze, te pokušat pretpostaviti u kojem će se smjeru razvijati naše aplikacije i njima dostupne platforme. Područje mikroprocesora je danas izuzetno zanimljivo jer je na pomolu redefiniranje pojmova CPU i GPU, barem načina na koji ih danas koristimo. Osnovno poznavanje arhitektura mikroprocesora je bitno za kasniju analizu rezultata *OpenCL* testova, te općenito za odabir arhitekture koja će učinkovitije izvoditi željene algoritme.

4.1 Centralni mikroprocesor - CPU

Razvoj mikroprocesora je još započeo 60' godina prošlog stoljeća, od onda je ostvaren ogroman napredak u njihovom razvoju, vjerovatno i najveći uzevši bilo koju industriju u obzir. Osnovni princip funkcioniranja je uglavnom ostao isti do danas, a temelji se na Von Neumannovoj arhitekturi. Mikroprocesor dohvaća naredbe redom iz memorije uz pomoć programskog brojila (engl. *program counter*), dekodira naredbu, ovisno o njoj dohvati podatke u memoriji, točnije registrima unutar samog mikroprocesora, te je izvede na aritmeto-logičkoj jedinici, dobivene podatke zapiše nazad na neku memorijsku lokaciju. Primjer izvođenja jedne instrukcije je prikazan slici, za jednu instrukciju je potrebno pet ciklusa.



Slika 5: Slijedno izvođenje instrukcija

Proizvođači mikroprocesora su se godinama, osim unapređivanja arhitekture, oslanjali i na konstantno podizanje radnog takta kako bi se postizale bolje performanse. Svoju ulogu je igrao i marketing jer je oznaka radnog takta ukazivala na snagu mikroprocesora u odnosu na druge, tako da su je proizvođači isticali uz naziv mikroprocesora. Međutim podizanje radnog takta se zaustavilo prije par godina na okvirno 3 GHz i danas su uglavnom na tržištu x86_64 mikroprocesori radnog takta između 2 i 3 GHz. Razlog tomu je da viši radni takt znači i veće zagrijavanje, a veliku količinu topline je teško odvoditi sa tako malene površine poput mikroprocesora. Za ilustraciju je dovoljno zamisliti potrošnju nekog mikroprocesora od

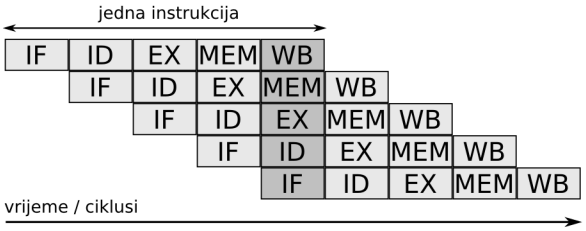
100 watta sa otprilike svega 1cm^2 površine, a fizička veličina mikroprocesora i cijelog računala mora biti manjih dimenzija zbog latencije i refleksija signala, te drugih neželjenih pojava. Zbog toga radni takt više nije glavni faktor podizanja brzine mikroprocesora, već su se proizvođači orijentirali na paralelizaciju rada i višejezgrenost, što povlači problem paralelizacije algoritama koji često nije odveć trivijalan. Problem radnog takta i topline vrijedi za sve elektroničke komponente općenito. Još jedan problem je što rast radnog takta mikroprocesora nije pratila i memorija, tako da je pristup memoriji u nekim slučajevima isto usko grlo sustava, što je još jedan od razloga što veće pričuvne memorije (engl. *cache*) na samom mikroprocesoru, koja opet opet zauzima dosta veliku fizičko površinu odnosno broj tranzistora.

Kako nikad neće postojati mikroprocesor dovoljno brz za sve potrebe, tj. u tom slučaju bi trebao biti beskonačno brz, logično je korištenje više mikroprocesora zajedno. Jedan od načina je smještanje više mikroprocesora u jedno računalo, no i to ima svoje granice zbog troškova sinkronizacije, fizičke veličine sabirnica koje ih povezuju, hlađenja, cijene i drugih praktičnih razloga. Tipično je imati osobno računalo sa jedim mikroprocesorom, dok ona poslužiteljska mogu po 2, 4, 8 i u nekim specifičnim slučajevima više. Slijedeći korak je povezivanje više računala u grozdove, te raspoređivanje poslova po njima uz pomoć sustava poput *MPI*. Prilikom paralelnog izvođenja programa svakako treba uzeti u obzir i trošak, preciznije vrijeme, komunikacije, koji je unutar jednog računala izuzetno malo, ali između više računala može imati značajan utjecaj na cjelokupne performanse.

Zbog prije navedenih problema današnji mikroprocesori su izuzetno kompleksni, sadrže čak preko 2 milijarde tranzistora, te imaju cijeli niz ugrađenih tehnika kako bi im se pospješila učinkovitost. Posjeduju velik skup instrukcija koje izvode (CISC arhitektura), mnogo registara i velike količine pričuvne memorije na samom mikroprocesoru, uobičajeno 1 do 8MB. Pristup pričuvnoj memoriji je daleko brži nego radnoj memoriji računala. Kako se visinom radnog takta dosegao maksimum slijednog izvođenja naredbi, već dulje vrijeme se veći naglasak stavlja na paralelizam koji se ostvaruje tehnikama poput cjevovoda (engl. *pipelining*), superskalarnosti (engl. *superscalar*), višedretvenosti (engl. *multithreading*), a u zadnje vrijeme i višejezgrenosti (engl. *multicore*).

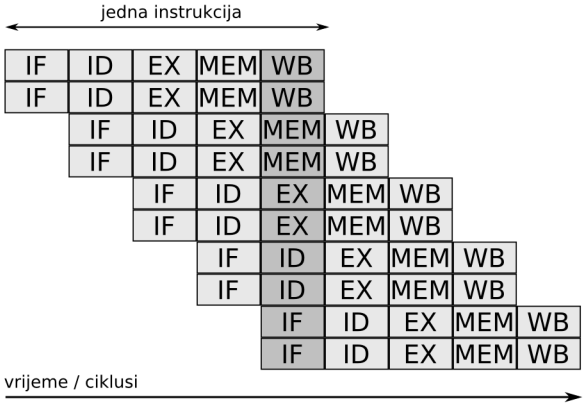
4.1.1 Cjevovodi i superskalarnost

Za izvedbu jedne instrukcije mikroprocesoru trebaju jedan ili više radnih perioda. Trajanje instrukcije se nikako ne može smanjiti na manje od jednog takta, ali je moguće pokušati izvesti u jednom periodu više različitih instrukcija. Pristup koji to omogućuje se naziva instrukcijski cjevovod (engl. *instruction pipeline*), koji se pojednostavnjeno može zamisliti kao započinanje obrade jedne instrukcije dok druga nije završila, tako u nekoliko razina. Problemi koji se javljaju je mogučnost nejednolikog broja perioda za izvršenje jedne instrukcije, pa za to vrijeme ostale čekaju, te ovisnost izvođenja jedne instrukcije o rezultatu prethodne. Ilustracija prikazuje princip cjevovoda duljine pet razina, za razliku od *"običnog"* slijednog izvođenja u kojem za jednu instrukciju treba pet ciklusa, u ovom primjeru se u idealnom slučaju može izvesti jedna instrukcija u jednom ciklusu.



Slika 6: Cjevovodi CPU-a

Sljedeći korak je superskalarnost koja uključuje dulje cjevovode sa više identičnih jedinica koje obrađuju instrukcije. Na taj način se pokušava više instrukcija izvesti u jednakom vremenskom periodu, a glavnu ulogu tu preuzima raspodjeljivač instrukcija (engl. *dispatcher*) koji odlučuje kako će preraspodijeliti instrukcije po cjevovodima. Na slici je vidljivo kako u idealnom slučaju mikroprocesor može efektivno izvesti dvije instrukcije u jednom ciklusu.



Slika 7: Superskalarnost CPU-a

Osim ove dvije bitne karakteristike, današnji mikroprocesori posjeduju i druge napredne tehnologije, poput predviđanja granjanja (engl. *branch prediction*), brzog prebacivanja između dretvi (engl. *out of order exectution*), te uvođenja novih setova instrukcija. Skup instrukcija SSE omogučava vektorsku obradu podataka (SIMD), te su tu još i posebne instrukcije za virtualizaciju i mnoge druge.

Danas svi centralni mikroprocesori u svojoj realizaciji posjeduju cjevovode i superskalarnost. Bitno je napomenuti kako na navedene principe programer nema utjecaj, tj. njih logika mikroprocesora sama primjenjuje na slijedno izvođenje algoritama i tako pokušava u jednom radnom periodu izvesti što više instrukcija, što u nekim slučajevima znatno unapređuje brzinu, dok kod nekih drugih baš i ne. Za ostvarenje logike CPU-a koja upravlja navedenim tehnikama je potrebna znatna količina tranzistora, dok je GPU *"jednostavniji"* što se logike tiče, pa je veći broj tranzistora iskorišten za ALU jedinice, što mu omogućuje daleko brže matematičke kalkulacije u mnogo dretvi.

4.1.2 Višedretvenost i višejezgrenost

Na današnjim računalima nalazimo pokrenute na desetke ili stotine programskih procesa od kojih svaki može imati više dretvi, dok se za redoslijed i priorite njihovog izvođenja brinu

raspoređivači unutar operacijskog sustava (engl. *scheduler*). Konkretan primjer upotrebe dretvi je situacija u kojoj jedna obavlja posao dok druga dohvaća nove podatke iz memorije. Višedretvenost je sposobnost mikroprocesora da odjednom izvodi više od jedne programske dretve, po principu da se mikroprocesor može u svega par ciklusa prebaciti sa izvođenja jedne dretve na drugu ili da paralelizira njihovo izvođenje, tj. dijelova koje može. Svi današnji operacijski sustavi za osobna računala već dulji niz godina mogu paralelno izvoditi procese i dretve, te na taj način iskorištavaju mogućnosti mikroprocesora. *Intel* je u svojem *Pentium 4* mikroprocesoru uveo tehnologiju nazvanu *HyperThreading* (*HT*), koja je jedan mikroprocesor operacijskom sustavu predstavljala kao dva. Učinkovitost je ostala upitna te je *HT* isključen iz kasnijih generacija mikroprocesora, da bi se u onim novijim opet uključio.

Višejezgrenost je aktualniji trend od prije opisanih, kada se dostigla fizička granica radnog takta i s time problema odvođenja topline, proizvođači su se zadržali na istom radnom taktu, ali su počeli daljni rast broja tranzistora iskorištavati za ostvarenje više procesorskih jezgri. Time korisnici efektivno dobivaju dva ili više mikroprocesora na raspolaganje, a fizički je riječ o više jezgi unutar jednog mikroprocesora koje dijele zajedničku pričuvnu memoriju. Do pojave višejezgrenih CPU-a je jedini način posjedovanja više od jednog CPU-a u jednom osobnom računalu bio posjedovanje poslužitelja koji su su naravno većeg cjenovnog ranga.

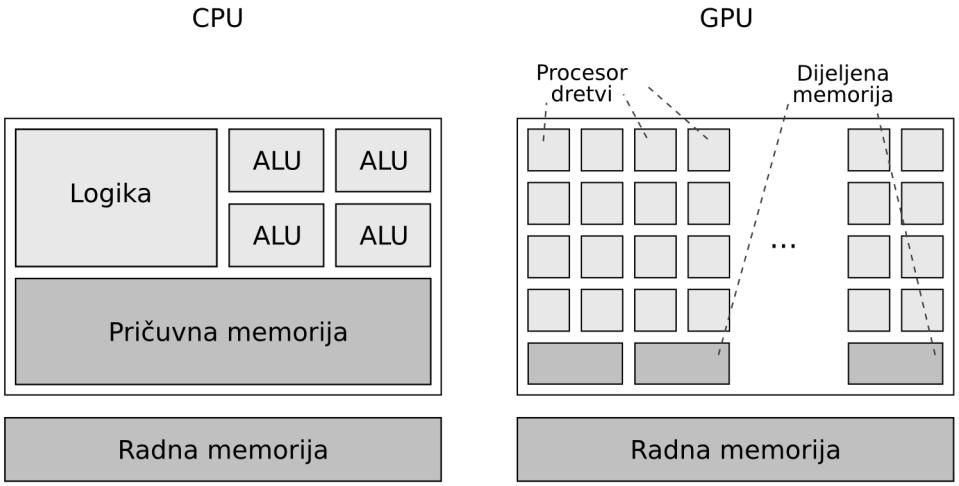
Danas na raspolaganju imamo centralne mikroprocesore arhitekture x86_64 kompanija *Intel* i *AMD* koji su međusobno kompatibilini, tj. barataju sa istim skupovim instrukcija uz neke manje razlike, sadrže po nekoliko jezgri, veće količine pričuvne memorije i napredne načine optimiranja izvođenja programskog koda. Centralni mikroprocesori su opće namjene što naravno povlači određene mane, imaju velik skup instrukcija i moraju zadržavati kompatibilnost unazad jer se i tekst programa napisan prije 20 godina mora izvoditi na novom mikroprocesoru. Iz tog razloga su izrazito kompleksni, što povlači određeni danak u performansama na specifičnija područa, jer je njihova namjena općenito obavljanje rada u računalu, poput operacijskog sustava i korisničkih aplikacija, dok su grafički mikroprocesori usmjereni na paralelni rad nad većim količinama podataka.

4.2 Grafički mikroprocesor - GPU

Značajniji razvoj grafičkih mikroprocesori je, za razliku od centralnih mikroprocesora, krenuo mnogo kasnije, tek u drugoj polovici 90' godina, za potrebe ubrzanja prikaza trodimenzionalne grafike. Iz tog razloga je njihova arhitektura poprilično drugačija od CPU-a, namjena im je što brži prikaz 3D grafike što je zahtjeva što brže obavljanje matematičkih operacija, veliku propusnost i paralelizaciju.

Dok CPU pokušava cjevovodima i superskalarnošću paralelizirati izvođenje instrukcija, a tek zadnjih godina višejezgrenošću, grafički mikroprocesori su u samom početku dizajnirani tako da imaju više *"manjih"* jezgri i da se drže SIMD odnosno SIMT principa. Prva bitna razlika je da CPU ima veliku količinu pričuvne memorije što uzima velik broj tranzistora, dok kod GPU-a to nije slučaj, tako da mu više tranzistora ostaje za ALU jedinice. U slučaju *Nvidie* je GPU organiziran tako da se sastoji od procesora dretvi (engl. *thread processor*) koji su organizirani u grupe, svaka grupa ima svoju dijeljenu memoriju (engl. *shared memory*) koja odgovara pričuvnoj memoriji kod CPU-a. Svaki procesor dretvi u grupi ima pristup istoj dijeljenoj memoriji, ali ne i memoriji druge grupe. Također svaki procesor dretvi može simultano izvršavati veći broj dretvi. *Nvidia* grupe procesora dretvi naziva multiprocesor, a svi noviji

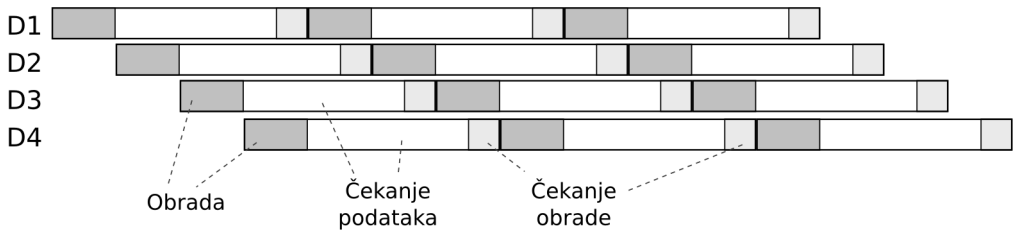
GPU-ovi imaju više multiprocesora, za primjer korištena *Nvidia FX570m* se sastoji od četiri multiprocesora.



Slika 8: CPU - GPU usporedba

Konkretan primjer je *GeForce8* mikroprocesor (2006. godina) koji ima 128 procesora dretvi, svakih 8 je u istoj grupi te dijele po 16kB dijeljene memorije, a svaki može istovremeno izvršavati po 96 dretvi. Time dolazimo do ukupnog broja od 12,288 dretvi koje mikroprocesor može istovremeno izvršavati. Jasno je kako GPU spram CPU-a ima daleko veću mogučnost paralelnog izvršavanja programskog koda, s druge strane mu je mana mala pričuvna odnosno dijeljena memorija. Iz tog razloga će se GPU dobro ponašati za obradom velike količine kontinuiranih podataka, poput 3D grafike, obrade videa i zvuka, te općenito paraleliziranih algoritama koji zahtjevaju mnogo matematičkih operacija.

GPU ostvaruje jako slabe performanse ako program koji izvodimo nije visoko paraleliziran, razlog je što posjeduje veliku količinu procesora dretvi koji sami po sebi nisu brzi, ali kako ih ima puno njihova ukupna učinkovitost drastično raste. GPU također izvodi po nekoliko dretvi na jednom procesoru dretvi iz razloga što dok jednu obrađuje, priprema memoriju za sljedeću, princip je donekle sličan cjevovodima kod CPU-a, te se naziva prikrivanje latencije (engl. *latency hiding*) pristupa memoriji.



Slika 9: Prikrivanje latencije kod GPU-a

Danas se na području grafičkih mikroprocosora orijentiranih na 3D performanse natječu kompanije *Nvidia* i *AMD/ATI*. GPU-ovi također kao i CPU-ovi imaju po par milijardi tranzis-

tora, radni takt im najčešće ne prelazi 1 GHz, a proizvođači se hvale kako su u vršnim performansama po čak nekoliko puta brži od CPU-a, naravno pod određenim tipovima opterećenja. Prije par godina su proizvođači uvidjeli da potencijal svojih GPU-ova mogu iskoristiti i u šire svrhe od samog prikaza 3D grafike, te pružili korisnicima sučelja za njihovo iskorištavanje, poput prije opisane *CUDA*-e i *Streama*. Nvidia je čak ponudila i grafičke kartice koje uopće nisu namijenjene prikazu slike, već sadrže po nekoliko grafičkih mikroprocesora visoke klase i služe isključivo za paralelnu obradu podataka kroz *CUDA*-u ili *OpenCL*, riječ je o *Tesla* seriji grafičkih kartica sa po tisuću i više jezgri. Tako je poprilično velika računalna snaga postala lakše dostupna, jer jedna takva kartica uvelike nadmašuje i najskuplje poslužitelje sa po nekoliko CPU-a, naravno u užem području primjene.

Za razliku od razvoja programa za CPU, kod GPU treba voditi računa o nekoliko principa kojima se značajno poboljšavaju performanse, poput da je njegova pričuvna memorija mala, te da ju treba *"pažljivo"* koristi jer je pristup njoj daleko brži nego radnoj memoriji, treba minimizirati prijenos podataka između domaćina (CPU + radna memorija) i GPU-a jer je sabirnica kojom je povezan često usko grlo sustava, osigurati velik broj dretvi kako bi GPU mogao primjenjivati princip prikrivanja latencije, te izbjegavati mogučnost grananja unutar dretvi. Također treba biti svjestan mane da GPU ne barata dobro sa tipovima podataka koji zahtjevaju dvostruku preciznost (64 bita) brojeva sa pomičnim zarezom, performanse su značajno lošije nego prilikom rada sa brojevima jednostruke preciznosti (32 bita), GPU-ovi zadnje generacije nude jednake performanse za oba tipa podataka.

Još jedna praktična osobina GPU-a je što ih se više može relativno jednostavno udružiti, kako već po samom dizajnu jesu paralelni, a ne moraju se brinuti oko mnogo drugih podsustava u računalu za koje se CPU brine, npr. upravljačkih programa za pristup tvrdim diskovima, računalnoj mreži, periferijama i tako dalje. Mogučnost uparivanja kartica sa GPU-om je već dulji niz godina dostupna kod određenih modela, a recimo *Tesle* i fizički imaju po 4 i više GPU-a na jednoj kartici. Za korištenje više CPU-a u jednom računalu je potrebno imati matičnu ploču sa više CPU ležišta (od 2 do 8) i mikroprocesore koji to podržavaju, a sve to ulazi u poslužiteljsku domenu i znatno diže cijenu, često neproporcionalno dobivenim performansama. Naravno ovisno o tipu problema koji se rješava se donosi odluka je li GPU ili CPU bolje zadovoljava potrebe i nudi bolje rješenje, u svakom slučaju je potrebno imati računalo sa barem jednim CPU-om na kojem će se izvršavati operacijski sustav i na taj način biti svojevrstan hipervizor nad GPU-ovima.

Nvidia je u vrijeme pisanja ovog rada najavila novu generaciju svojih grafičkih mikroprocesora, nazvanu Fermi. Neke od karakteristika i unapređenja koje najavljuju jasno pokazuju da *Nvidia* sa svojim GPU-ovima više ne cilja samo na prikaz 3D grafike, već i na širu primjenu svojih mikroprocesora kroz *GP/GPU*, tj. *CUDA*-u i *OpenCL*. Osim povećanja broja tranzistora (3 milijarde) i broja jezgri (32 *CUDA* jezgre po multiprocesoru) tu su još i neka specifičnija poboljšanja poput podrške za 64-bitne tipove podataka, povećanje dijeljene (*OpenCL* lokalne) memorije, uvođenje L1 i L2 priručne memorije, brža izmjena konteksta koji se izvodi, uvođenje dva raspoređivača (engl. *dispatcher*) redoslijeda izvođenja dretvi, sposobnost *Out-of-Order* izvođenja dretvi, podrška za *ECC*[8] memoriju i mnoga druga poboljšanja.

Danas CPU i GPU često koegzistiraju u određenim računalima zbog potrebe za brzim prikazom 3D grafike, dok se u poslužiteljima isključivo nalaze samo CPU-ovi, no i taj trend bi se mogao početi mijenjati, dovoljno je napomenuti kako nova superračunala kombiniraju CPU

[8]ECC memorija se koristi u poslužiteljima, a garantira da se prilikom pristupa memoriji neće dogoditi pogreška, tj. da je dohvaćeni podatak valjan.

i GPU kakve danas nalazimo u osobnim računalima, naravno u daleko većem broju. Time GPU postaje svojevrstan matematički koprocesor CPU-u, kao što smo prije dvadestak godina imali mogučnost ugradnje dodatnog matematičkog koprocesora u računalo, koji se potom kroz vrijeme integrirao u sam mikroprocesor.

4.3 Cell BE

Za arhitekturu mikroprocesora *Cell* se može reći da je u neku ruku hibrid CPU-a i GPU-a, također ju nazivaju heterogenom višejezgrenom arhitekturom. *Cell* je predstavljen 2005. godine od strane konzorcija *IBM*, *Sony* i *Toshiba* koji su ga razvijali za potrebe igrače konzole *Sony Playstation3*, za tu svrhu je gotovo pa u potpunosti ispočetka izdizajniran, a ne nadogradnjom neke postojeće arhitekture, iz tog razloga je uveo određene novitete u arhitekture mikroprocesora.

Neki od ciljeva su bili proširenje već postojeće *IBM*-ove *Power* jezgre s dodatnim akceleratorskim jezgrama, primjena paralelizma visokih performansi, velika propusnost prema radnoj memoriji, brz odaziv u realnom vremenu zbog osnovne namjene igračoj konzoli, te jednostavnost i fleksibilnost u programiranju.

Cell se sastoji od jedne jezgre zvane *Power Processor Element* (*PPE*) i osam jezgri zvanih *Synergistic Processor Elements* (*SPE*). *PPE* je jezgra opće namjene temeljene na prijašnjoj *Power* arhitekturi, riječ je o 64-bitnom RISC mikroprocesoru koji radi na 3.2 GHz, te je zapravo riječ o dvojezgrenom procesoru. Namjena *PPE* jezgre je je izvođenje operacijskog sustava, kontrola nad cijelim mikroprocesorom i upravljanje nad ostalim *SPE* jezgrama. *SPE* jezgre su pojednostavljene RISC arhitekture od kojih svaka ima ugrađeni *DMA* (engl. *direct memory access*) kontroler za pristup memoriji, to im omogućava u isto vrijeme izvođenje naredbi i dohvaćanje sljedećih potrebnih podataka iz memorij. Iz toga razloga *Cell* dostiže velike brzine jer se ne gubi vrijeme potrebno za dohvat novih podataka iz memorije. *PPE* i *SPE* jezgre su povezane unutar mikroprocesora sabirnicom zvanom *Element Interconnect Bus* (*EIB*) koja omoguća veliku propusnost između elemenata mikroprocesora, čak preko 300 GB/s. U praksi se učinkovitost *Cella* pokazala za red veličene bolja u odnosu na ostale aktualne mikroprocesore u područjima kritpografije, grafike i video procesiranja. Naravno pritom treba uzeti u obzir je riječ o 2005./2006. godini kada su tek počeli biti dostupni dvojezgreni mikroprocesori.

Cell mikroprocesor je nadostupniji nabavkom *PlayStationa3* na koji se može instalirati *Yellow Dog* distribucija *GNU/Linuxa* i *IBM*-ovo razvojno okruženje (*SDK*) za *Cell*. U vrijeme pojave *PlayStationa* je njegova cijena bila relativno mala u odnosu na dobiven mikroprocesor, pa su mnogi uočili priliku izgradnje *"uradi sam"* superračunala od nekoliko desetaka ili čak stotina umreženih *PS3*-a sa *Cellovima*. Konkretan primjer je upotreba 200 *PS3* konzola 2008. godine za računanje kolizija MD5 sažetka koji se koristio prilikom izdavanja *PKI* certifikata. *IBM* je 2007. godine ponudio na tržištu *blade* poslužitelje sa *Cell* mikroprocesorima. Njihovim korištenje se može na jedan od učinkovitijih načina sastaviti izrazito jako računalo, naravno cijenom je rezervirano za viši rang kupaca.

Cell je ponudio jedan sasvim novi pristup arhitekturi mikroprocesora i nagovijestio trend koji tek slijedi, a to spajanje jezgri opće namjene i specijaliziranih jezgri u jedan mikroprocesor. Posebno je zanimljivo da bi se uskoro mogla očekivati *OpenCL* podrška za *Cell* mikroprocesore, kako je i *IBM* jedan od članova konzoricija koji je radio na *OpenCL* sustavu.

4.4 Buduće arhitekture

U kojem smjeru će se razvijati arhitekture i mikroprocesori bazirani na njima je relativno teško za procijeniti, no neke osnovne naznake se mogu pretpostaviti. Danas imamo na raspolaganju mikroprocesore x86_64 arhitekture kompanija Intel i *AMD*, grafičke mikroprocesore *Nvidie* i *AMD*-a odnosno *ATI*-ja, a *IBM* ima svoj *Cell* dostupan na užem području.

Prije bilo kakve procjene potrebno je sagledati i trenutnu situaciju podrške softvera određenim arhitekturama. Danas sva osobna računala, stolna i prijenosna, sadrže CPU arhitekture x86, također i gotovo svi poslužitelji, te superračunala. Mikroprocesori arhitekture x86 su se još 90' nametnuli osnovna arhitektura svakog računala, te se danas na njoj izvode svi operacijski sustavi, *Windows*, *GNU/Linux*, **BSD* i *Mac OS X*. Grafički mikroprocesori nikad nisu bili namijenjeni izvođenju operacijskog sustava, a zbog njihove poprilično drugačije arhitekture postojeće operacijske sustave bi bilo u praksi gotovo nemoguće modificirati u tom smjeru. GPU se oduvijek koristio za dodatnu akceleraciju određenih poslova za koje se CPU pokazao presporim. Za korištenje potencijala GPU-a je bilo potrebno imati instalirane upravljačke programe na operacijskom sustavu. Operacijski sustavi otvorenog i slobodnog programskog koda (*Open Source*, *Free Software*) poput *GNU/Linuxa* i inačica **BSD*-a se uz x86 mogu izvoditi i na cijelom nizu drugih arhitektura, poput *ARM*-a u mobilnim uređajima, *MIPS*-a u ugradbenim uređajima, *PPE* jezgre (*Power* arhitektura) u *Cellu*, te mnogim drugim. Može se zaključiti kako u računalu bilo koje svrhe moramo imati arhitekturu CPU-a na kojoj se može izvoditi operacijski sustav, a opcionalno i GPU ili neki drugi akceleratorski mikroprocesor u svrhu izvedbe algoritama za koje je CPU prespor. Time su moguće tri kombinacije mikroprocesora u računalu:

1. jedan ili više CPU, svaki može biti višejezgreni
2. jedan ili više CPU, svaki može biti višejezgreni + jedan ili više GPU
3. mikroprocesor koji je *"hibrid"* CPU-a i GPU-a: *Cell*, *Intel Larrabee*, ...

Odabir neke od prethodnih kombinacija ovisi o namjeni računala i poslu koji će obavljati. Kako je već prije opisan problem prenosivosti programskog koda među platformama, ovo je mjesto gdje *OpenCL* savršeno popunjava prazninu. Neovisno o arhitekturi mikroprocesora, tekst programa napisan u *OpenCL* bi trebao detektirati platforme na kojima se nalazi i izvoditi se na njima. Je li riječ o zahtjevnim znanstvenim proračunima ili korisničkim aplikacijama koje zahtjevaju veću obradu podataka nije odviše bitno, jer će isti tekst programa imati mogučnost izvođenja na uobičajenom dvojezgrenom CPU-u kakav danas nalazimo u osobnim računalima, poslužitelju sa nekoliko višejezgrenih CPU-a ili GPU-u nižeg odnosno višeg cijenovnog ranga. Kao trenutne mane *OpenCL*-a se mogu navesti trenutni nedostak implementacija za sve platforme i nemogučnost jednostavnog izvršavanja na više računala istovremeno.

Trenutno sve upućuje da kroz sljedećeih nekoliko godina možemo očekivati mikroprocesore koji će dobro izvoditi poslove opće namjene poput CPU-a, ali i posjedovati mogučnost obrade velike količine podataka po SIMD prinicpu. Sigurno je da će se realizirati kroz višejezgrenost poput *Cella*. Tu se javlja pitanje orijentacije kompanija, te tehnologija i vlasničkih prava nad patentima. Trenutno patentna prava za x86 tehnologiju i njezina proširenja (npr. SSE setovi instrukcija) imaju pretežito *Intel* i tek iza njega *AMD*, točnije mnoge licence su odlučili dijeliti ili pak međusobno plačati, kako bi njihovi mikroprocesori bili međusobno kompatibilni.

U tom pogledu je trenutno *AMD/ATI* u najboljem položaju, jer se *AMD* bavi proizvodnjom x86 mikroprocesora, a *ATI* grafičkih mikroprocesora, što im osigurava posjedovanje

tehnologije i znanja za dizajn mikroprocesora koji bi mogao izvoditi naredbe x86 arhitekture, a opet imati i SIMD karakteristike grafičkih mikroprocesora. *Intel* je već u tom smjeru najavio sasvim novi mikroprocesor zvan *Larrabee*, bio bi baziran na x86 arhitekturi sa velikim brojem jezgri i mogučnošću paralelne obrade podataka. Zanimljivo je da *IBM* već ima *Cell* koji se poprilično dobro uklapa u prethodni opis, no nije riječ o x86 arhitekturi već od *Power* sa *SPE* jezgrama, što onemogučuje znatniji prodor u osobna računala. Također se dugo priča kako *Nvidia* planira izdati svoj x86 mikroprocesor, no zbog patentnih prava je to poprilično upitno, druga mogučnost je licenciranje *ARM* arhitekture i njena kombinacija sa već postojećim GPU-ovima, no to opet nebi omogučilo prodor u domenu centralnih mikroprocesora osobnih računala. Može se zaključiti kako je pitanje vremena kada ćemo u osobnim računalima imati mikroprocesore koji kombiniraju karakteristike CPU-a i GPU-a, samo je pitanje u kojem obliku, osnovni problem je zadržavanje kompatibilnosti sa postojećim operacijskim sustavima i aplikacijama. Naravno to ne isključuje i dalje postojanje akceleratorskih mikroprocesora koji će se opcionalno ugrađivati u računala koja već imaju nekakav centralni mikroprocesor. Još jedan pokazatelj razvoja u tom smjeru je *Intelova* kupovina udjela u kompaniji *Imagination Technologies* koja proizvodi 3D akceleratorske mikroprocesore za mobilne uređaje.

Ne treba zanemarivati i činjenicu da je dizajn novog mikroprocesora i njegova proizvodnja izuzetno skupo, te si proizvođači ne smiju dozvoljavati prevelike pogreške u procjenama tržišta. Ako se odmaknemo od osobnih prema računalima specifičnije namjene poput nekih poslužitelja i superračunala, zadržavanje x86 arhitekture i ne stvara veći problem iz razloga što se i danas za sve zahtjevnije poslove, računala i pripadajući softver posebno prilagođavaju, jer nema potrebe za širom namjenom, npr. *Cell blade* poslužitelji.

4.4.1 Intel Larrabee

Intel je najavio mikroprocesor kodnog naziva *Larrabee* koji bi trebao biti višejezgrena kombinacija CPU-a i GPU-a, baziran na x86 arhitekturi s mogučnošću izvođenja SIMD operacija. Za razliku od procesora *Cell* sve jezgre bi mu bile istovjetne, a broj bi po najavama bio oko 32. Kako se sastoji od x86 jezgri sa određenim proširenjima, teoretski bi trebao biti sposoban izvoditi tekst programa koji se danas izvodi na *intelovim* mikroprocesorima. *Intelove* najave i izvešća su mnogo varirala, po prvim najavama je trebao biti na tržištu 2010. godine namijenjen svim potrošaćima, dok zadnja spominju njegovu primjenu samo u istraživačke i razvojne svrhe poput računarstva visokih performansi. U svakom slučaju, bez obzira na vrijeme njegovog pojavljivanja, *Intel* je jasno nagovjestio nove tipove mikroprocesora koji bi objedinili karakteristike CPU-a i GPU-a, ali i pokazao kako drastična promjena trenutne arhitekture nije odviše jednostavna, bilo u tehničkom pogledu ili pitanju kompatibilnosti sa postojećim softverom.

5 Ispitivanje i usporedba performansi OpenCL-a

Za ocjenu performansi su izvedeni primjeri u *OpenCL*-u na više platformi, te još u tehnologijama *MPI* i *OpenMP* kako bi se dobili okvirni omjeri njihovih performansi. Odmah treba uzeti u obzir činjenicu kako primjeri u različitim tehnologijama ne mogu biti izvedeni u istim uvjetima iz razloga što se djelomično izvode na različitim platformama, implementacijama i operacijskim sustavima. Cilj ispitivanja je upoznavanje sa samim tehnologijama, dobivanje okvirnih rezultata, te što je najbitnije omjera među rezultatima kako bi se mogao izvući zaključak kako koja tehnologija i platforma stoji u odnosu na drugu. Također je zanimljivo vidjeti kako se koja tehnologija i tip mikroprocesora nosi sa porastom dimenzija problema i njegovom paralelizacijom.

Glavni primjer je množenje matrica različitih veličina i tipova podataka, odbran zbog jednostavnosti, relativno lagane paralelizacije u svim tehnologijama i njegove sličnosti sa stvarnim zahtjevima u područjima znanstvenih istraživanja i procesiranja većih količina podataka. Ispitivanja su provedena za kvadratne matrice veličina[9] 256, 512, 1024, 1536 i 2048, te cjelobrojne tipove podataka (engl. *integer*), sa pomičnim zarezom (engl. *float*) i pomičnim zarezom dvostruke preciznosti (engl. *double*).

Primjeri u *OpenCL*-u se izvode na platformama GPU i CPU, dok se *MPI* i *OpenMP* izvode samo na CPU-ovima. Na temelju tih rezultata se jasno može razlučiti razlika u namjeni pojedine platforme i pripadajuće arhitekture. Primjeri za *OpenCL* su postupno optimizirani kako bi se uočila razlika dobivenih rezultata kroz primjenu optimizacija koje proizvođači preporučaju, poput načina pristupa memoriji i raspodjeli problema.

Za provedbu ispitivanja su bile dostupne sljedeće platforme:

1. **Thinkpad T61p**

CPU:	Intel Core 2 Duo T7300, 2GHz, 4MB cache, 4GB RAM	MPI, OMP
GPU:	Nvidia Quadro FX570m, 256MB, 475Mhz	OpenCL
OS:	GNU/Linux Debian, 64-bit	

2. **Mac Mini**

CPU:	Intel Core 2 Duo P7350, 2GHz, 4MB cache, 1.5GB RAM	OpenCL
GPU:	Nvidia GeForce 9400, 256MB	OpenCL
OS:	Mac OS X Snow Leopard Server	

3. **Marvin**

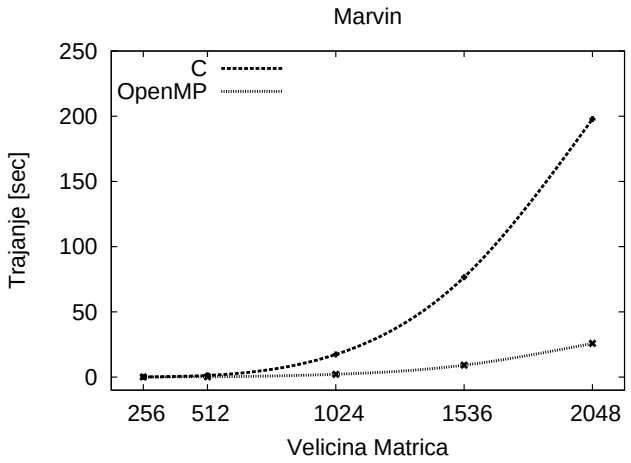
CPU:	2x Intel Xeon E5504 QuadCore, 2GHz, 4MB cache, 8GB RAM	MPI, OMP
GPU:	-	
OS:	GNU/Linux Debian, 64-bit	

5.1 OpenMP

OpenMP je zanimljiv za usporedbu sa *OpenCL*-om na arhitekturi CPU-a jer također paralelizira algoritam pokretanjem više dretvi i svi mikroprocesori imaju pristup zajedničkoj radnoj memoriji. Tako se mogu relativno pouzdano usporediti performanse ovih alata na istom centralnom mikroprocesoru.

[9] U tekstu se pod veličinom matrica misli na broj elemenata po dimenziji kvadratne matrice, a ne na ukupan broj elemenata matrice.

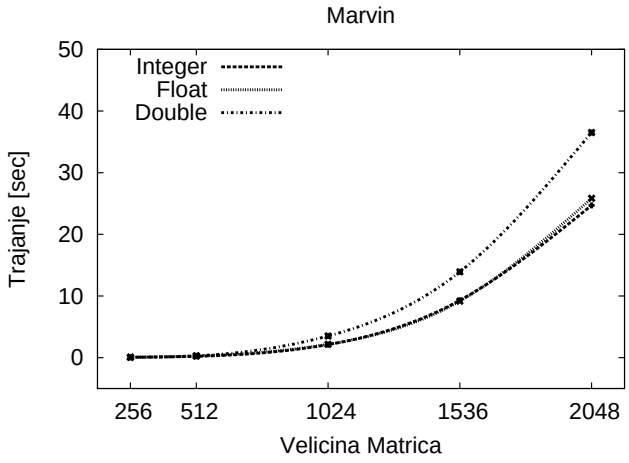
Slika 10 prikazuje trajanje množenja matrica u jednodretvenom *C* tekstu programa i višedretvenom kodu paraleliziranom kroz *OpenMP*. Očekivano je višedretveni kod brži jer se izvodi na osam jezgri, spram jednodretvenog koji se može izvoditi samo na jednoj jezgri. Isto tako je očekivan rezultat da višedretvena verzija nije osam puta brža od jednodretvene, što bi bio idealan slučaj, već se ubrzanje sa porastom mikroprocesora odnosno jezgri usporava zbog troškova sinkronizacije.



Slika 10: *OpenMP*, različite veličine matrica na Marvinu

Centralni mikroprocesori moraju biti posebne serije koja je namijenjena višeprocesorskom radu, najčešće je riječ o serijama namijenjenim za ugradnju u poslužitelje. Jedna od bitnijih odlika im je ugrađena logika za sinkronizaciju, jer dijele istu radnu memoriju (*RAM*) računala. Najčešće se serije CPU-a namijenjenih višeprocesorskom radu dijele na one koje mogu raditi do dva zajedno, te one koje mogu do osam zajedno, ali su one već značajno skuplje.

Slika 11 prikazuje razliku trajanja primjera ovisno o tipu podataka. Očekivano su cjelobrojni i elementi sa pomičnim zarezom postigli slične rezultate, dok je primjer sa tipom dvostruke preciznosti trajao nešto dulje. Razlog je duljina tipa podataka: dvostruka preciznost je duljine osam okteta, dok su druga dva tipa duljine četiri okteta.



Slika 11: *OpenMP*, različiti tipovi podataka na Marvinu

Uočena je zanimljivost prilikom ispitivanja. *OpenMP* je zapravo proširenje programskog prevodioca, točnije skup predprocesorskih naredbi pomoću kojih zatim prevodilac paralelizira tekst programa, tako da brzina izvršnog programa uvelike ovisi o kvaliteti samog prevodioca. U ovom slučaju je korišten *GCC 4.4.2* (*GNU C Compiler*) koji je u početku generirao višedretveni program koji se izvodio značajno dulje nego jednodretveni program. Rješenje problema je bilo eksplicitno označavanje varijabli privatnima (engl. *autoscoping*) jer prevodilac očigledno to ne radi sam, te alokacija matrice kroz pokazivač na pokazivače svakog reda.

```
#pragma omp parallel for private(i, j, k, sum)
for (i = 0; i < rank; ++i)
        ...
```

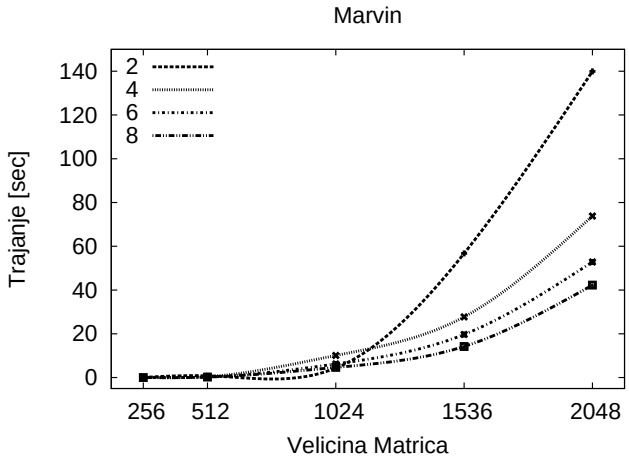
Pretraživanjem Interneta je utvrđeno da su i neki drugi korisnici imali istih problema sa *GCC*-om, dok im drugi programski prevodioci nisu stvarali slične probleme. Ovaj primjer samo dokazuje da većina optimizacija leži u dobro napisanom programskom kodu i kvalitetnom prevodiocu.

5.2 MPI

Sustav *MPI* za razliku od *OpenMP*-a paralelizira algoritme pokretanjem procesa, a ne dretvi, te omogućava komunikaciju porukama između više računala. Paralelizacija je kompliciranija za programera jer se mora brinuti oko razmjene poslova i podataka porukama te za sinkronizaciju, dok se kod *OpenMP*-a za paralelizaciju većinom brine prevodilac.

Slika 12 prikazuje trajanje testova ovisno o broju pokrenutih procesa odnosno korištenih jezgri CPU-a.

Očekivano je veći broj jezgri završio primjer u kraćem vremenu, a isto tako očekivano sa porastom jezgri se vrijeme nije linearno smanjivalo zbog troškova komunikacije i sinkronizacije. Prikazani rezultati se odnose na podatke sa pomičnim zarezom, dok su oni dvostruke preciznosti također ostvarili nešto dulja trajanja kao kod *OpenMP*-a. U trajanje nije uračunato vrijeme potrebno za kopiranje matrica svakom procesu, već samo računanje i prikupljanje rezultata



Slika 12: *MPI*, različite veličine matrica i brojevi procesa na Marvinu

od svakog procesa. Razlog je taj što se u navedenom primjeru kopiraju cijele matrice svakom procesu, dok bi u optimiziranijem slučaju to bili samo dijelovi matrica koji su potrebni procesu za izračun.

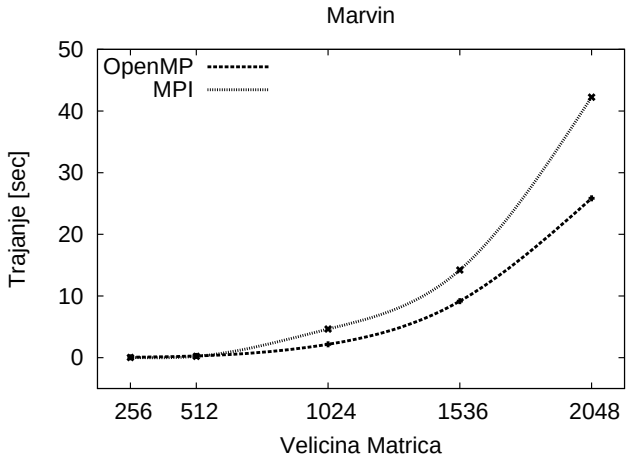
Može se zaključiti kako povećanje broja CPU-ova, tj. jezgri, u računalu poboljšava performanse, ali samo do neke mjere, tj. rast performansi odgovara logaritamskoj funkciji, dok cijena nažalost raste eksponencijalno. Iz tog razloga je danas najčešća gornja granica od osam CPU-a u jednom računalu. Navedeno ograničenje se odnosi na sve tehnologije koje barataju sa CPU-ovima, uključujući i *OpenCL* dok se izvodi na CPU. Sljedeći korak je povećanje broja računala, čime naravno raste kompleksnost algoritama u smislu raspodjele poslova i podataka. Bitno je, bez obzira na korišteni sustav, voditi računa o optimizaciji programskog koda, poput izbjegavanja uskih grla u sustavu, učinkovite raspodjele poslova, optimizacije raspodjele podataka i pristupa memoriji.

Slika 13 prikazuje usporedbu trajanja izvođenja primjera između *OpenMP* programa sa više dretvi i *MPI* programa sa osam procesa; očekivano je *OpenMP* nešto brži iz razloga svoje jednostavnosti u odnosu na *MPI*.

5.3 OpenCL

Za *OpenCL* primjere su bile dostupne dvije platforme, prijenosno računalo *Thinkpad t61p* sa *Nvidia Quadro FX570m* grafičkim mikroprocesorom i *Mac Mini* sa *Nvidia GeForce 9400* i *Intel Core 2 Duo* mikroprocesorom. Cilj je usporediti rezultate izvođenja primjera množenja matrica napisanih u *OpenCL*-u sa dobivenim rezultatima u drugim sustavima, te dati okviran omjer postignutih performansi među različitim arhitekturama, CPU i GPU. *OpenCL* primjeri uključuju uspoređivanje performansi sa različitim veličinama matrica, globalnog i lokalnog indeksnog prostora, te različitih tipova podataka.

Primjeri su izvedeni korištenjem *python* sučelja za *OpenCL* naziva *pyOpenCL*. Razlog je lakše postavljanje okoline za izvođenje programskih jezgri, koje su opet napisane u *OpenCL C*-u. Program *opencl-mm.py* služi za pokretanje primjera, a kroz komandnu liniju mu se mogu



Slika 13: *OpenMP - MPI*, Marvin

postaviti parametri poput dimenzija matrica i podmatrica, tipa podataka i ispisa matrica u svrhu provjere rezultata. U sustavu *OpenCL* veličine globalnog i lokalnog indeksnog prostora odgovaraju dimenzijama matrica i podmatrica. Sintaksa za pokretanje je sljedeća:

```
veljko:~/diplomski/src/opencl-mm$ ./opencl-mm.py -h
Parameters:
  [-d | --dimensions] MxNxP : Matrices dimensions, [m n] * [n p]
  [-b | --block] N          : Block size, N x N
  [-t | --datatype] type    : DataType [int|float|double]
                              (default float)
  [-v | --verbose]          : Prints matrices for checking purpose
  [-h | --help]             : Shows this help
```

Dimenzije se označavaju sintaksom *MxNxP*, gdje je *M* broj redaka matrice *A*, *N* stupaca matrice *A* i redaka matrice *B*, te *P* broj stupaca matrice *B*. Nisu sve veličine bloka dozvoljene, već dimenzije moraju biti višekratnik veličine bloka. U primjerima su korišteni blokovi veličine 1, 4, 8 i 16. Tipovi podataka mogu biti cjelobrojni (4 okteta), sa pomičnim zarezom (4 okteta) i dvostruke preciznosti (8 okteta), bitno je obratiti pažnju da *–datatype* parametar samo postavlja okolinu *OpenCL*-a za željeni tip podataka, dok se u programskoj jezgri zasebno mora pripaziti kakav se tip podataka definira.

Primjer otkrivanja dostupnih *OpenCL* platformi na *Apple Mac OS X* operacijskom sustavu, te ispis podrobnijih podataka:

```
Platform name: Apple
Platform profile: FULL_PROFILE
Platform vendor: Apple
Platform version: OpenCL 1.0 (Oct 16 2009 04:12:08)
-----------------------------------
Device name: GeForce 9400
Device type: GPU
Device memory:  256 MB
Device max clock speed: 1100 MHz
Device compute units: 2
```

```
-----------------------------------
Device name: Intel(R) Core(TM)2 Duo CPU P7350  @ 2.00GHz
Device type: CPU
Device memory:  1536 MB
Device max clock speed: 2000 MHz
Device compute units: 2
```

Za potrebe ispitivanja su napisane tri programske jezgre koje se razlikuju po korištenju indeksnog prostora i globalne odnosno lokalne memorije. Tako su prikazana dva osnovna koncepta pisanja *OpenCL* programskih jezgri, granulacija problema kroz indeksni prostor i odnos brzine pristupa globalnoj i lokalnoj memoriji.

5.3.1 Primjer: zbrajanje matrica, programska jezgra 1

Prvi primjer prikazuje najjednostavniji način paralelizacije algoritama koristeći globalne indekse. Na domaćinu se stvaraju memorijski međuspremnici za matrice, koji se zatim kopiraju u memoriju uređaja, a također se proslijede i dimenzije matrica kroz argumente m, n i p. Za svaki element indeksnog prostora koji odgovara veličini matrice C, u koju se sprema rezultat, pokreće se po jedna programska jezgra, svaka sa svojim globalnim indeksom. Konačno se kroz petlju obavlja množenje odgovarajućeg retka matrice A sa stupcem matrice B, te se rezultat sprema u element matrice C određen prije dobivenim indeksima.

```
__kernel void matrix_mul(const __global float* A,
                         const __global float* B,
                         __global float* C,
                         uint m, uint n, uint p)
{
        // određivanje retka i stupca u indeksnom prostoru
    uint row = get_global_id(0);
    uint col = get_global_id(1);

    // množenje elemenata matrica, C = A * B
    C[row * p + col] = 0;
    for (uint k = 0; k < n; ++k)
        C[row * p + col] += A[row * n + k] * B[k * p + col];
}
```

5.3.2 Primjer: zbrajanje matrica, programska jezgra 2

Razlika u odnosu na prvi primjer je što se ilustrira korištenje radnih grupa i lokalnih indeksa unutar njih. Drugi primjer u pogledu performansi nema značajnu prednost u odnosu na prvi, već više služi kao primjer granulacije problema kroz grupe. Zanimljivo je kako je na korisniku da sam odredi globalni i lokalni indeksni prostor, te kasnije u programskim jezgrama iskoristi tu dodatnu granulaciju za učinkovitije rješavanje problema. Na temelju indeksa radne grupe i lokalnog indeksa, može se za svaku programsku jezgru odrediti njezin globalni indeks.

```
__kernel void matrix_mul(const __global float* A,
                         const __global float* B,
                         __global float* C,
                         uint m, uint n, uint p)
{
    // određivanje veličine bloka
    uint blockSize = get_local_size(0);
```

```
    // određivanje indeksa radne grupe
    uint row = get_group_id(0);
    uint col = get_group_id(1);

    // određivanje lokalnih indeksa u grupi
    uint x = get_local_id(0);
    uint y = get_local_id(1);

    // određivanje indeksa u matrici kroz grupe i lokalne indekse
    uint pos_x = (row * blockSize + x) * p;
    uint pos_y = col * blockSize + y;

    // množenje elemenata matrica, C = A * B
    C[pos_x  + pos_y] = 0;
    for( uint k = 0; k < n; ++k)
        C[pos_x + pos_y] += A[pos_x + k] * B[k * p + pos_y]; }
```

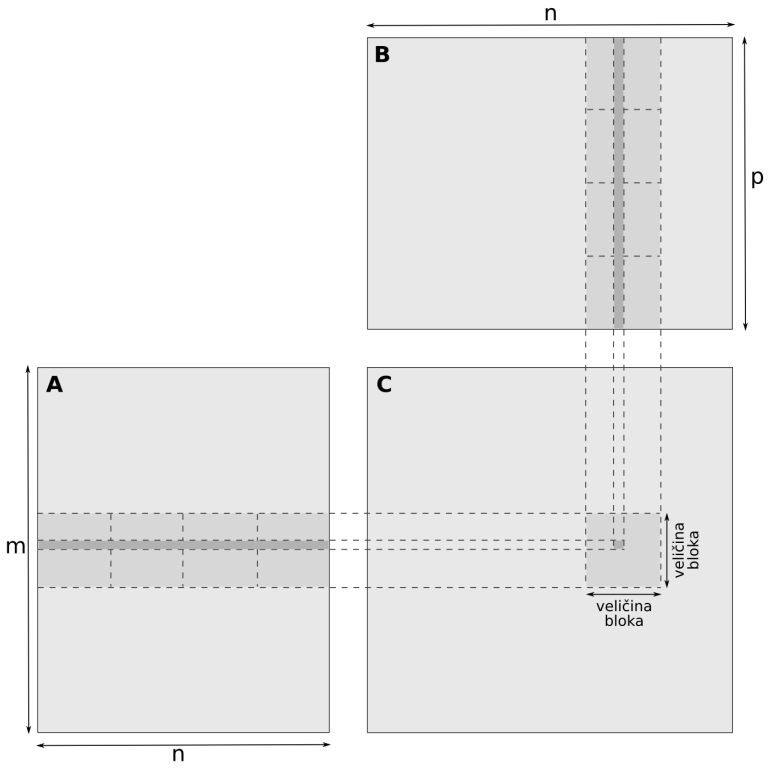
5.3.3 Primjer: zbrajanje matrica, programska jezgra 3

Treći primjer za razliku od prethodna dva koristi i lokalnu memoriju, te dijelove matrica kopira iz globalne u lokalnu memoriju. Kako se svaki element matrica koristi više puta, svaki pristup globalnoj memoriji oduzima značajno vrijeme. Kopiranjem blokova elemenata u znatno bržu lokalnu memoriju mogu se povećati performanse i za jedan red veličine. Razlika u deklaraciji funkcije programske jezgre je što se među argumentima nalazi i lokalna memorija, kroz te argumente se ništa ne prosljeđuje već služe samo za alokaciju memorije.

```
__kernel void matrix_mul(const __global float* A,
                         const __global float* B,
                         __global float* C,
                         uint m, uint n, uint p,
                         __local float* subA,
                         __local float* subB)
{
    // broj radnih grupa i njihove dimenzije
    uint numGroups = get_num_groups(0);
    uint blockSize = get_local_size(0);

    // indeksi radne grupe  (2 dimenzije)
    uint row = get_group_id(0);
    uint col = get_group_id(1);

    // lokalni indeksi jezgre u radnoj grupi
    uint x = get_local_id(0);
    uint y = get_local_id(1);

    float subC = 0;

    // iteracija kroz sve blokove matrica A i B potrebnih za računanje
    for(int blockA = row * blockSize * n , blockB = col * blockSize;
        blockA <= row * blockSize * n + numGroups * blockSize - 1;
        blockA += blockSize, blockB += blockSize * p)
    {
        // kopiranje elemenata matrica A i B u podmatrice u lokalnoj mem.
```

```
        // svaka radna jedinica kopira po jedan element
        subA[x * blockSize + y] = A[blockA + x * n + y];
        subB[x * blockSize + y] = B[blockB + x * p + y];

        // čekanje da sve radne jednice završe kopiranje
        barrier(CLK_LOCAL_MEM_FENCE);

        // množenje elemenata podmatrica A i B
        for(int k = 0; k < blockSize; ++k)
            subC += subA[x * blockSize + k] * subB[k * blockSize + y];

       // opet čekanje da sve radne jedinice završe računanje
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // postavljanje rezultata u globalnu matricu C
    C[get_global_id(0) * get_global_size(0) + get_global_id(1)] = subC;
}
```

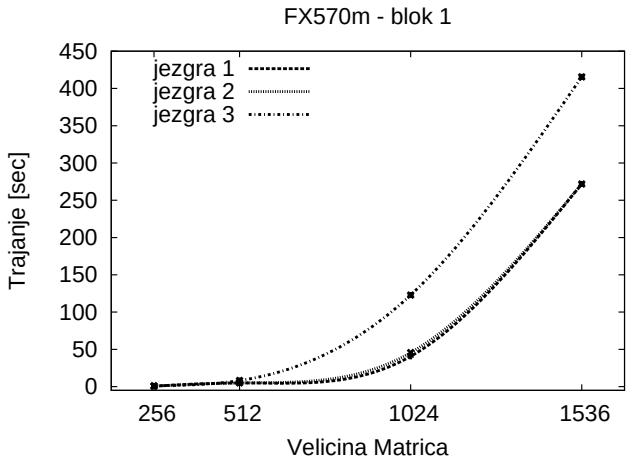
Slika 14 ilustrira raspodijelu matrica po blokovima. Tamniji blokovi matrica A i B se kopiraju u lokalnu memoriju prilikom računanja elemenata označenog bloka matrice C. Svi blokovi odnosno radne grupe su istih dimenzija, tj. kvadratnih su veličina. Algoritam kopira blok (podmatricu) matrica A i B u lokalnu memoriju, izračuna rezultat tih podbloka, te ponovi isto za sljedeće. Tako se izbjegava višestruko pristupanje istim podacima u globalnoj memoriji, već se potrebni podaci kopiraju u lokalnu memoriju, te im se zatim višestruko pristupa.



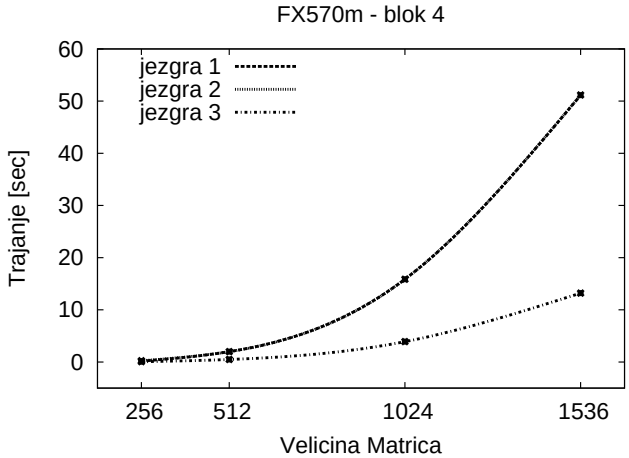
Slika 14: *OpenCL* Primjer 3, podjela problema na blokove

5.3.4 Rezultati primjera zbrajanja matrica

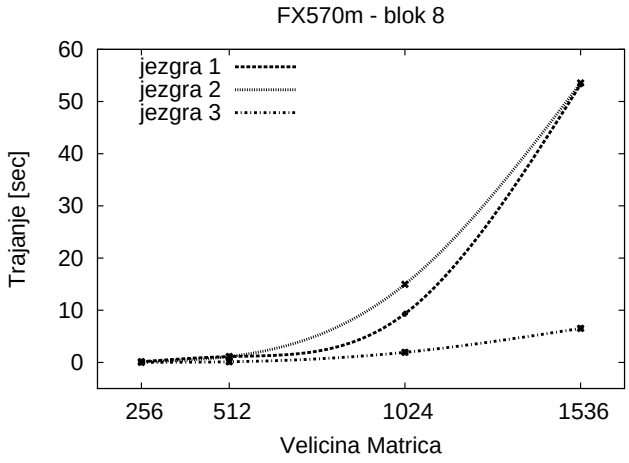
Slike 15, 16, 17 i 18 prikazuju rezultate koje su ostvarili primjeri programskih jezgri 1, 2 i 3, ovisno o veličini bloka, redom 1, 4, 8 i 16, te veličini matrica, korišten je tip podataka sa pomičnim zarezom. Primjeri su izvedeni na GPU platformi *Nvidia FX570m*.



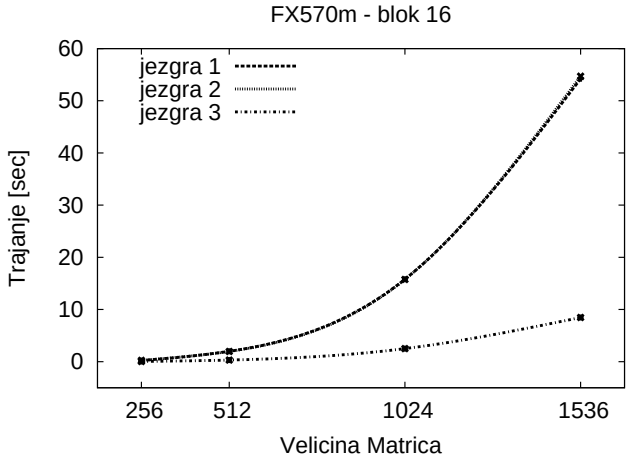
Slika 15: *OpenCL*, veličina bloka 1



Slika 16: *OpenCL*, veličina bloka 4



Slika 17: *OpenCL*, veličina bloka 8



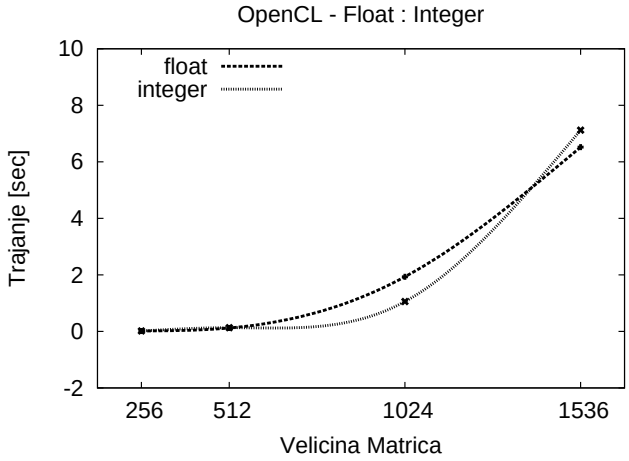
Slika 18: *OpenCL*, veličina bloka 16

Svi grafovi pokazaju jednako trajanje izvođenja primjera 1 i 2, bez obzira na veličinu bloka i tip podataka. Razlog je što jedan u odnosu na drugi nema nikakvu prednost što se tiče optimalnosti programskog koda, već se prvi oslanja na globalne indekse, dok se drugi oslanja na indekse grupa i lokalne indekse.

Na slici 15 prikazano je trajanje izračuna u slučaju kada je veličina bloka jednaka jedan, u kojem slučaju test 3 postiže najlošije rezlutate. Razlog je što se u ovom primjeru sadržaj blokova kopira u lokalnu memoriju, a kako je veličina bloka jednaka jedan, efektivno se svaki element matrica kopira zasebno u lokalnu memoriju. Kopiranjem samo jednog elementa se ne postiže nikakvo ubrzanje, već se samo nepotrebno troši vrijeme na kopiranje elementa iz globalne u lokalnu memoriju, te se čeka sinkronizacija dretvi.

Slike 16, 17 i 18 prikazuju rezultate za veličine grupa 4, 8 i 16. U tim slučajevima su se svi testovi pokazali bržima, a test 3 značajno nadmašuje ostale testove, čak i za red veličine. Razlog je što jedan multiprocesor u GPU-u izvodi odjednom skupinu dretvi, te na taj način prikriva latenciju pristupa memoriji, što utječe na sve testove. Primjer 3 je značajno brži jer iz globalne memorije kopira cijeli blok podataka u lokalnu memoriju, a taj blok odgovara kvadratu veličine bloka. Tako više dretvi više puta pristupa lokalnoj memoriji, a samo jednom globalnoj prilikom kopiranja podataka. Za konkretan slučaj se veličina bloka od 8 pokazala najučinkovitijom. Sami proizvođači GPU-ova predlažu da na veličinu bloka, točnije lokalnu veličinu indeksnog prostora, treba obratiti pažnju iz razloga što sam sustav ne može pouzdano odrediti njenu optimalnu veličinu, već je bolje da sam programer procijeni i isproba koja postiže najbolje rezultate.

Slika 19 prikazuje odnos performansi za cjelobrojne tipove podataka i one sa pomičnim zarezom. Uzeta je veličina bloka 8 jer se pokazala najučinkovitijom.



Slika 19: *OpenCL*, usporedba tipova podataka

Očekivano je kao i kod CPU-a test sa cjelobrojnim podacima trajao otprilike jednako kao i onaj sa pomičnim zarezom. Primjer sa dvostrukom preciznošću nije bilo moguće izvesti jer je sustav tražio navođenje predprocesorske naredbe u programskoj jezgri kako bi uključio mogučnost rada sa 64-bitnim podacima, no i nakon toga je tvrdio da nema mogućnosti za to. Razlog je što je ta podrška kod *Nvidie* uključena tek u grafičkim mikroprocesorima zadnje generacije. Kako je osnovna namjena GPU-a prikaz grafike za koju nije bitna dvostruka preciznost, ta mogućnost nije bila ni potrebna. U novijim mikroprocesorima je dodana i ta mogućnost upravo iz razloga znanstvenih izračuna koji se mogu izvoditi na GPU-u, a često zahtjevaju dvostruku preciznost odnosno rad sa podacima duljine osam okteta.

5.3.5 Primjer: simulacija međudjelovanja čestica

Uz primjer množenja matrica, koji je napisan za više različitih sustava, za potrebe ispitivanja *OpenCL*-a je napisan i primjer simulacije međudjelovanja čestica (engl. *N-body simulation*).

Radi se o učestalom primjeru simuliranja fizikalnih pojava, poput međusobnog djelovanja velikog broja čestica u prostoru. Riječ je o računalno poprilično zahtjevnom poslu jer je odlika dobre simulacije korištenje što većeg broja čestica s pripadajućim parametrima, te uzimanje što manjeg vremenskog perioda u simulaciji, što rezultira velikim brojem iteracija. Za ilustraciju je dovoljno zamisliti simulaciju gibanja fluida, poput valovitog mora pod utjecajem vremenskih (tlak, vjetar) i fizikalnih (privlačne sile čestica) prilika.

Napravljeni primjer za ove potrebe nije odviše kompleksan, već se u izračun uzimaju samo trenutne pozicije čestica u prostoru, te njihova masa i trenutna brzina. Svakom iteracijom se za svaku česticu računaju privlačne (gravitacijske) sile kojom sve druge čestice djeluju na nju, te se na temelju toga izračunava nova pozicija u prostoru. Početno stanje sustava se postavlja slučajnim odabirom početne pozicije svake čestice i njene mase, a početne brzine su jednake nuli.

Aplikacija je također napisana u *pyOpenCL*-u i kroz parametre joj se može određivati broj čestica koji odgovara globalnom indeksnom prostoru, zatim veličina bloka/grupe koji odgovara lokalnom indeksnom prostoru, te ukupan broj iteracija u simulaciji.

```
veljko:~/diplomski/src/opencl-nbody $ ./opencl-nbody.py -h
Parameters:
  [-n | --size] N   : Number of particles (default: 1024) (power of 2)
  [-b | --block] N  : Block size (default: 4)
  [-i | --iters] N  : Number of iterations (default: 128)
  [-h | --help]     : Shows this help
```

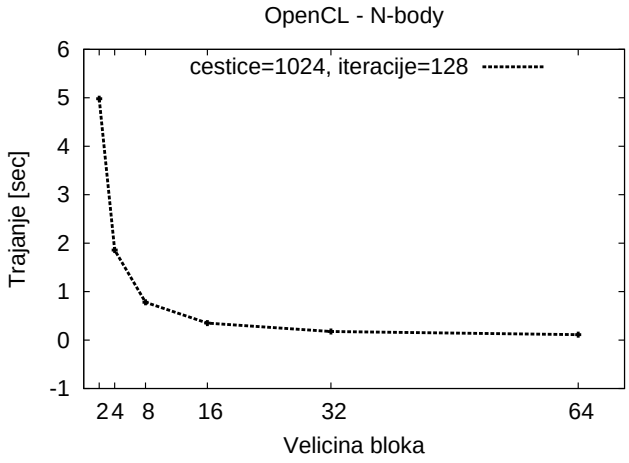
Razlika u odnosu na primjer množenja matrica je što se koristi jednodimenzionalni indeksni prostor. Za pohranu trenutne pozicije i mase se koristi vektorski tip podataka[10] *float4*. Vektorski tipovi podataka se kod prikaza grafike koriste za spremanje pozicija u prostoru, a mogu se zamisliti kao uobičajen niz podataka, ali kraće i unaprijed određene duljine (2, 4, 8 ili 16). U ovom primjeru se koristi tip *float4* koji se sastoji od četiri vrijednosti sa pomičnim zarezom, prve tri se koriste za poziciju u prostoru, a četvrta za masu čestice. Isti tip se koristi i za brzinu čestica, ali se kod njega četvrti element zanemaruje. Slijedi zakomentirani ispis teksta programske jezgre:

```
__kernel void nbody_simulation(int iters, float _dt, float limit,
                               __global float4* _position,
                               __global float4* _velocity,
                               __local float4* block)
{
  // dohvaćanje lokalne veličine i broja grupa
  int l_size = get_local_size(0);
  int n_groups = get_num_groups(0);

  // dohvaćanje globalnog i lokalnog indeksa dretve
  int g_tid = get_global_id(0);
  int l_tid = get_local_id(0);

  // definiranje vremenskog intervala za simulaciju
  // (primjer korištenja vektorskog tipa podataka)
  const float4 dt = (float4)(_dt, _dt, _dt, 0.0f);

  // željeni broj iteracija za simulaciju
  for (int i=0; i < iters; i++)
```

[10]Vektorski tipovi podataka su odlika GPU-a, ali i novijih CPU-a sa SSE3 setom instrukcija.

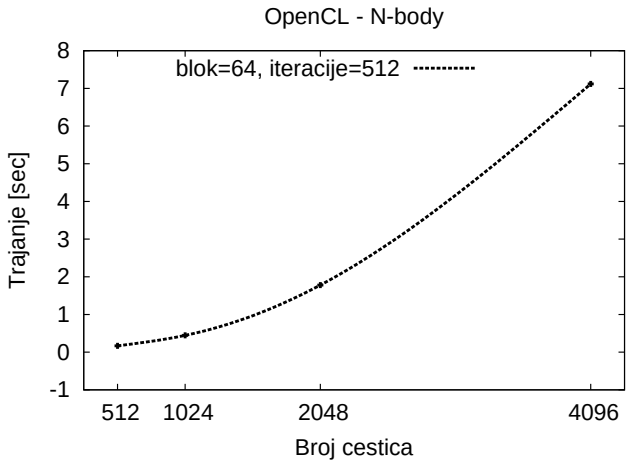
```
  {
    // početna pozicija, brzina i akceleracija
    float4 position = _position[g_tid];
    float4 velocity = _velocity[g_tid];
    float4 a = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

    // prolazak kroz sve grupe
    for (int gid=0; gid < n_groups; ++gid)
    {
      // svaka dretva kopira po jedan element iz
      // globalne u lokalnu memoriju, sinkronizacija
      block[l_tid] = _position[gid * l_size + l_tid];
      barrier(CLK_LOCAL_MEM_FENCE);

      // prolazak kroz sve čestice u grupi
      for (int k=0; k<l_size; ++k)
      {
        // izračun udaljenosti između čestica
        float4 d = block[k] - position;

        // izračun akceleracije čestice
        // primjer kdohvata elemenata tipa podataka float4
        // (vektor.x, vektor.y, vektor.z, vektor.w)
        float invr = rsqrt(pown(d.x,2) + pown(d.y,2) + pown(d.z,2) + limit);
        a += block[k].w * invr * invr * invr * d;
      }

      // sinkronizacija za prelazak na sljedeći blok
      barrier(CLK_LOCAL_MEM_FENCE);
    }

    // izračun nove pozicije i brzine
    position += dt * velocity + 0.5f*dt*dt*a;
    velocity += dt * a;

    // osvježavanje podataka u globalnom nizu čestica
    _position[g_tid] = position;
    _velocity[g_tid] = velocity;

    // sinkronizacija za sljedeću iteraciju
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

Slika 20. prikazuje odnos vremena izvođenja primjera ovisno o veličini bloka čestica odnosno veličini lokalnog indeksnog prostora *OpenCL*-a. U primjeru množenja matrica je veličina lokalnog indeksnog prostora rasla kvadratom veličine bloka jer je problem bio dvodimenzionalan, dok je ovdje riječ o jednodimenzionalnom problemu, tj. čestice su smještene u jednodimenzionalan niz vektora. Očekivano je vrijeme izvođenja manje za veću veličinu bloka jer se više elemenata u jednom koraku kopira u bržu lokalnu memoriju. Svakako treba pripazi da se ne stavi prevelika veličina bloka iz razloga što je kod GPU-a lokalna memorija izuzetno mala, konkretno kod *Nvidie FX570m* svega 16kB za svaku radnu grupu. U primjeru se već kod veličine bloka od 512 rušio program, ali to nije ni toliko bitno, jer se već kod veličina većih od 32 ne postiže brže izvođenje primjera.



Slika 20: *OpenCL*, *N-body*, veličine lokalnih blokova

Slika 21. prikazuje eksponencijalan rast trajanja izvođenja primjera ovisno o broju čestica. Rezultat je očekivan jer se za svaku česticu uzima utjecaj svih ostalih čestica, tako da je složenost $O(n^2)$.



Slika 21: *OpenCL*, *N-body*, broj čestica

Navedeni primjer je sam po sebi jednostavno paralelizirati, jer se radi o simulaciji velikog broja čestica nad kojima se obavljaju istovjetne operacije. Zanimljivo je korištenje vektorskih tipova podataka jer se postižu veće performanse[11], naravno ako uređaj zna baratati sa njima,

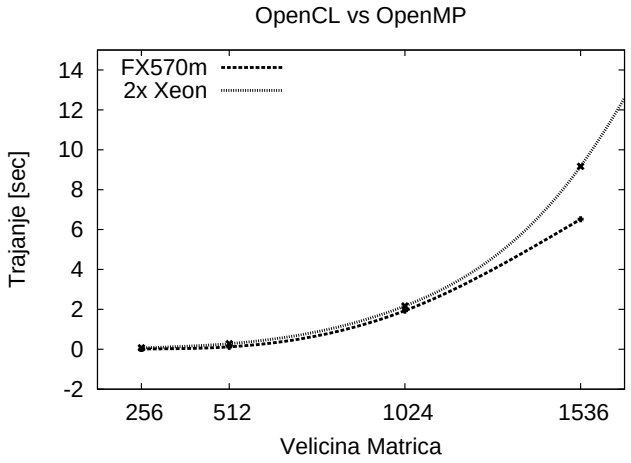
[11] Mikroprocesor u jednom koraku obavi istu operaciju nad svim elementima vektora, kao da se radi o *"običnom"* tipu podataka podataka (*integer*, *float*).

što je jedan od preduvjeta za *OpenCL*. Primjer također ilustrira općenitu praksu korištenja lokalne memorije kod *OpenCL*-a, neovisno o broju dimenzija problema.

Za daljnja ispitivanja bi bilo zanimljivo ostvariti neki kompleksniji algoritam u *OpenCL*-u, koji se po svojoj prirodi teže paralelizira. *OpenCL* u odnosu na druge sustave donekle olakšava paralelizaciju korištenjem indeksnog prostora i globalne i lokalne memorije, no njegova osnovna namjena je stvaranje apstrakcije nad heterogenim platformama.

5.3.6 Odnos performansi OpenCL-a i OpenMP-a

Na slici 22 je zanimljiv omjer performansi množenja matrica u *OpenCL*-u i *OpenMP*-u. *OpenCL* test je izveden na GPU-u *Nvidia FX570m* u prijenosnom računalu koristeći programsku jezgru primjera 3 sa veličinom grupe od osam, a *OpenMP* test na dva *Intel Xeon* CPU-a u poslužitelju.



Slika 22: *OpenCL FX570m* : *OpenMP* Marvin

Iznenađujuće je da je GPU kroz *OpenCL* sa korištenjem lokalne memorije za kraće vrijeme izveo primjer od dva *Intel Xeona*. Paralelna arhitektura GPU-a još više dolazi do izražaja kada se uzme u obzir da radi na daleko manjem radnom taktu, *FX570m* na 475Mhz spram *Xeona* na 2GHz, te da se navedeni GPU pojavio 2007. godine, dok je CPU arhitekture *Nehalem* koju je *Intel* predstavio 2009. godine.

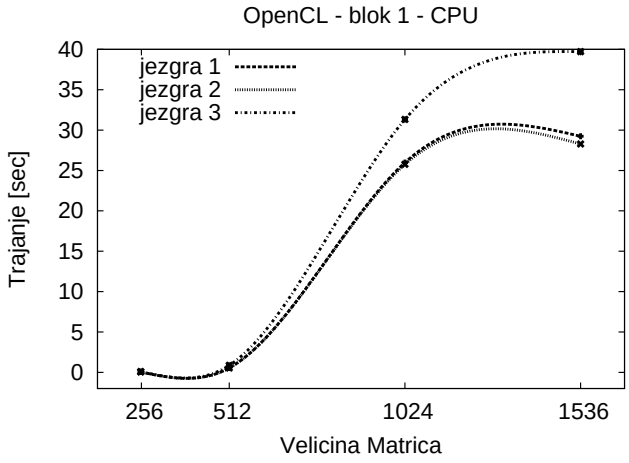
Dobiveni rezultat povlači za sobom pitanje je li se u svrhu računarstva visokih performansi (*HPC*) bolje oslanjati na GPU ili CPU. Prije svega treba uvidjeti o kakvom je problemu riječ, te odgovara li paralelna GPU arhitektura rješavanju tog problema. Ako je u pitanju jedoobrazna obrada velike količine podataka (engl. *stream processing*) poput raznih simulacija, rada sa videom i 3D grafikom, može se sa velikom sigurnošću odlučiti za GPU. Naravno neizostavno je i trenutno stanje potpore *OpenCL*-u ili nekom drugom *GP/GPU* sustavu, u praksi ono je trenutno jako malo zastupljeno za korisničke aplikacije[12].

[12] Za primjer se može uzeti aplikacija za uređivanje slika Photoshop CS4, kompanije Adobe, koja može koristiti i prednosti GPU-a u računalu.

Moguće je napraviti grubu usporedbu: *FX570m* posjeduje 4 multiprocesora i prema testovima postiže rezultate jednake kao dva *Xeon* CPU-a na 2GHz; dalje se za primjer može uzeti *Tesla* grafička kartica zadnje generacije koja posjeduje 4 grafička mikroprocesora svaki sa po 30 multiprocesora. *Tesla* bi pod tom pretpostavkom trebala postići 30 puta bolje rezultate od *FX570m*, a ako se uzme još dvostruko veći radni takt na kojem radi i 60 puta bolje rezultate. Naravno dio performansi se gubi porastom paralelizacije, ali znatno manji nego kod CPU-a. S druge strane je dovoljno pretpostaviti 120 *Xeon* CPU-a koji bi teoretski trebali biti ekvivalent jednoj *Tesli* prema provedenim testovima. Naravno unutar jednog računala je nemoguće zamisliti 120 centralnih mikroprocesora, već bi se trebalo računati na nekoliko poslužitelja, svaki sa po nekoliko CPU-a. Osim samih performansi postiže se i značajna financijska ušteda, te daleko manja potrošnja energije odnosno zagrijavanje.

Jedan od zaključaka ovog rada je da se svakako isplati odlučivati za GPU-ove, točnije investirati u matične ploče sa mogućnošću ugradnje jednog ili dva CPU-a za potrebe operacijskog sustava i raspodjele podataka, te što više *PCIexpress x16* utora u koje se danas ugrađuju grafičke kartice sa pripadajućim GPU-ovima. Na taj način se ostavlja dovoljno prostora za kasnija eventualna proširenja ugradnjom dodatnih grafičkih kartica. Može se reći kako danas GPU novije generacije u stolnom računalu može nadmašiti računalnu moć nekoliko skupih poslužitelja sa x86 mikroprocesorima.

Primjere na *Apple Mac Mini* stolnom računalu nažalost nije bilo moguće provesti u potpunosti. Sustav se prilikom pokretanja primjera sa većim dimenzijama matrica na GPU-u rušio. Razlog je vjerovatno *"slabiji"* grafički mikroprocesor koji dijeli radnu memoriju računala, te upaljeno grafičko sučelje (grafički poslužitelj *X*) koje mu oduzima resurse. Drugi mogući razlog je još nedotjerana implementacija *OpenCL*-a za dotičnu platformu ili pak što je manje vjerovatno greška u *pyOpenCL* sučelju koje nije pokazalo probleme na *GNU/Linuxu*. Primjeri su pokrenuti na CPU-u, ali su rezultati pretjerano varirali ovisno o veličini dimenzija matrica, slika 23. Vjerovatan uzrok je operacijski sustav koji se izvodi na CPU-u i sama *OpenCL* implementacija. Također je čudno što je programska jezgra 3 postigla najlošije rezultate, a očekivano bi bilo da postigne najbolje jer koristi lokalnu memoriju koja bi trebala odgovarati brzoj priručnoj (engl. *cache*) memoriji CPU-a; ili možda ne? Kod svih ostalih rezultata, neovisno o korištenom sustavu, trajanje izvođenja primjera je eksponencijalno raslo u skladu sa veličinom matrica, a ovdje to nije slučaj. Moguć razlog je što manje veličine matrica zauzimaju naravno manje memorijskog prostora, pa iz toga razloga, neovisno o lokalnoj i globalnoj memoriji *OpenCL*-a, završe u priručnoj memoriji. CPU posjeduje priručnu memoriju veličine 4MB, a matrica veličine 512 zauzima 1MB memorije, dok veličine 1024 zauzima 4MB memorije. Kako cijela matrica veličine 512 stane u priručnu memoriju, to je možda razlog velikog pada performansi za veće matrice. Moguće je i da se radi o izdvojenom slučaju, ali je svakako zanimljiva pojava koju bi trebalo dalje ispitati, za početak na nekoj drugoj implementaciji koja podržava rad sa CPU-om poput *ATI Streama*.



Slika 23: OpenCL, CPU

Primjeri na prijenosnom računalu *thinkpad t61p* su izvedeni pod *GNU/Linux* operacijskim sustavom sa ugašenim grafičkim sučeljem (*Xorg* poslužitelj). Korištena je dostupna *OpenCL* implementacija od *Nvidie* koja je još uvijek u fazi testiranja. Osim navedenih bilo bi zanimljivo isprobati *OpenCL* implementaciju kompanije *AMD/ATI*, posebno iz razloga što podržava *Radeon* GPU-ove i u kombinaciji sa njima x86 CPU-ove. Također bi daljna istraživanja trebala ići u smjeru ostvarivanja raznovrsnijih algoritama u *OpenCL*-u, prepisivanja dijela korisničkih aplikacija, te nekih dodatnih optimizacija u samom *OpenCL*-u poput izbjegavanja konflikata prilikom pristupa memorijskim područjima i uključivanja optimizacija *OpenCL* prevodiocu.

6 Zaključak

Na temelju provedenog istraživanja mogu se postaviti dva zaključka: jedan vezan uz *OpenCL* kao sustav, a drugi uz razvoj heterogenih platforma u smislu arhitektura mikroprocesora. *OpenCL* je prvi sustav koji omogućuje izvođenje istog programskog koda na različitim platformama, a njegova primjena može varirati od korisničkih aplikacija pa sve do računarstva visokih performansi. Danas na raspolaganju imamo mikroprocesore različitih arhitektura i svrha, te se nameće pitanje kako ih što jednostavnije i bolje iskoristiti.

OpenCL kao sustav za sada omogućuje iskorištavanje CPU-a i GPU-a koje danas nalazimo u svim osobnim računalima. Kako je riječ o novom sustavu, njegova implementacija na svim operacijskim sustavima i platformama se ne može smatrati potpunom za neku širu upotrebu u korisničkim aplikacijama. Kao standard iznimno obećava, te su između ostalog u njemu sudjelovale sve relevantije kompanije, pa se može zaključiti kako će njegova dostupnost sa vremenom rasti, tj. ako se želimo ograničiti na GPU-ove ona je već dostupna od *Nvidie* i *ATI*-a za sve bitne operacijske sustave. Trenutno je raspoloživo izuzetno malo aplikacija koje koriste *GP/GPU*, a uz *OpenCL* gotovo nijedna, no taj će se broj sa vremenom sigurno povećavati. *OpenCL* je sustav koji prvi koji stvara apstrakciju nad različitim platformama i nije standard koji ovisi o jednom proizvođaču, tako da se može pretpostaviti da će se proizvođači korisničkih aplikacija u području simulacija, obrade zvuka, slike i videa odlučivati za njega. S druge strane, pruža visoke performanse tako da je primjenjiv i u području računarstva visokih performansi. Tu mu se kao nedostatak može nametnuti nepostojanje definicije mrežne komunikacije u standardu, ali kako je *HPC* specifično područje tako taj nedostatak ne bi trebao stvarati veću zapreku. Također mu je prednost što se kod *HPC*-a često znaju koristiti različite platforme za koje je on upravo i namijenjen.

Drugi zaključak je vezan uz razvoj mikroprocesora. Sa korisničke strane je neophodno da svaki novi mikroprocesor pruža kompatibilnost unatrag (x86 arhitektura), ali to ne sprečava daljni razvoj u pogledu povećanja višejezgrenosti i uključivanja jezgri specifičnije namjene u mikroprocesore. Taj trend se već može iščitati po najavama proizvođača mikroprocesora. Također je danas moguće imati jedan ili više mikroprocesora opće namjene i uz njih razne akceleratorske mikroprocesore poput GPU-a. Kod *HPC*-a taj problem unazadne kompatibilnosti nije toliko izražen zbog specifičnosti same namjene *HPC*-a. Ako se danas odlučujemo za izgradnju paralelnog računala svakako se treba odstupiti od dosadašnje prakse pukog gomilanja CPU-a, već se orijentirati i na druge arhitekture poput GPU-a i *Cell*-a.

Konačno je za same korisnike dobro što danas za malu cijenu mogu dobiti poprilično jako osobno računalo za koje će kroz dogledno vrijeme biti dostupne uobičajene aplikacije, ali sa mogućnošću iskorištavanja paraleliziranih arhitektura. Programeri, znanstvena zajednica i napredniji korisnici danas mogu iskorištavati puni potencijal heterogenih platformi kroz sustav *OpenCL* za svoje specifične ciljeve. Time se *OpenCL* kao standard iznimno dobro uklopio na trenutno dostupne platforme, ali i one koje će se tek pojaviti.

Ključne riječi: paralelizacija algoritama, heterogene platforme, *OpenCL*, *GP/GPU*, *MPI*, *OpenMP*, centralni/grafički mikroprocesor, *Cell/BE*

Keywords: parallel algorithms, heterogeneous platforms, *OpenCL*, *GP/GPU*, *MPI*, *OpenMP*, CPU, GPU, *Cell/BE*

7 Dodatak: Tablični prikaz rezultata primjera

Tablica 6: (Slika 10.) *OpenMP*, različite veličine matrica na Marvinu

Veličina	C	OpenMP
256	0.14	0.07
512	1.32	0.28
1024	17.39	2.17
1536	76.62	9.17
2048	197.78	25.84

Tablica 7: (Slika 11.) *OpenMP*, različiti tipovi podataka na Marvinu

Veličina	int	float	double
256	0.07	0.07	0.07
512	0.22	0.28	0.30
1024	2.12	2.17	3.52
1536	9.30	9.17	13.92
2048	27.71	25.84	36.51

Tablica 8: (Slika 12.) *MPI*, različite veličine matrica i brojevi procesa na Marvinu

Veličina	2	4	6	8
256	0.089	0.045	0.030	0.029
512	0.794	0.402	0.276	0.216
1024	4.351	10.092	6.223	4.639
1536	56.763	27.741	19.719	14.222
2048	139.930	73.83	52.833	42.248

Tablica 9: (Slika 13.) *OpenMP* - *MPI*, Marvin

Veličina	OpenMP	MPI, 8
256	0.07	0.029
512	0.28	0.216
1024	2.17	4.639
1536	9.17	14.222
2048	25.84	42.248

Tablica 10: (Slika 15.) *OpenCL*, veličina bloka 1

Veličina	Jezgra 1	Jezgra 2	Jezgra 3
256	0.652	0.651	1.020
512	4.934	4.933	8.138
1024	40.634	45.598	122.875
1536	271.867	271.771	415.517

Tablica 11: (Slika 16.) *OpenCL*, veličina bloka 4

Veličina	Jezgra 1	Jezgra 2	Jezgra 3
256	0.166	0.248	0.061
512	1.984	1.984	0.484
1024	15.867	15.867	3.903
1536	51.139	51.177	13.202

Tablica 12: (Slika 17.) *OpenCL*, veličina bloka 8

Veličina	Jezgra 1	Jezgra 2	Jezgra 3
256	0.140	0.140	0.015
512	1.123	1.122	0.121
1024	9.332	14.968	1.934
1536	53.297	53.565	6.520

Tablica 13: (Slika 18.) OpenCL, veličina bloka 16

Veličina	Jezgra 1	Jezgra 2	Jezgra 3
256	0.222	0.243	0.039
512	1.989	1.959	0.312
1024	15.792	15.760	2.503
1536	54.227	54.708	8.462

Tablica 14: (Slika 19.) *OpenCL*, usporedba tipova podataka

Veličina	float	integer
256	0.015	0.016
512	0.121	0.132
1024	1.934	1.056
1536	6.520	7.122

Tablica 15: (Slika 20.) *OpenCL*, *N-body*, veličine lokalnih blokova; čestice 1024, iteracije 128

Blok	Trajanje
2	4.982
4	1.858
8	0.778
16	0.350
32	0.177
64	0.111

Tablica 16: (Slika 21.) *OpenCL*, *N-body*, broj čestica

Broj čestica	Trajanje
512	0.167
1024	0.445
2048	1.780
4096	7.117

Tablica 17: (Slika 22.) *OpenCL FX570m* : *OpenMP* Marvin

Veličina	OpenCL	OpenMP
256	0.015	0.07
512	0.121	0.28
1024	1.934	2.17
1536	6.520	9.17

Tablica 18: (Slika 23.) *OpenCL*, CPU

Veličina	Jezgra 1	Jezgra 2	Jezgra 3
256	0.040	0.040	0.099
512	0.548	0.545	0.869
1024	25.980	25.760	31.319
1536	29.237	28.278	39.716

Literatura

[1] Khronos OpenCL Working Group, 2009: The OpenCL Specification, version 1.0 rev 48

[2] Nvidia, 2009: OpenCL Programming for the CUDA Architecture, version 2.3

[3] Nvidia, 2009: OpenCL Programming Guide for the CUDA Architecture, version 2.3

[4] Nvidia, 2009: OpenCL Best Practices Guide, version 1.0

[5] Andreas Klöckner, prosinac 2009: http://mathema.tician.de/software/pyopencl/

[6] Wikipedia, 12.02.2010: http://en.wikipedia.org/wiki/N-body_problem