SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Seminar

Paralelno programiranje na CELL procesoru

Nino Antulov-Fantulin

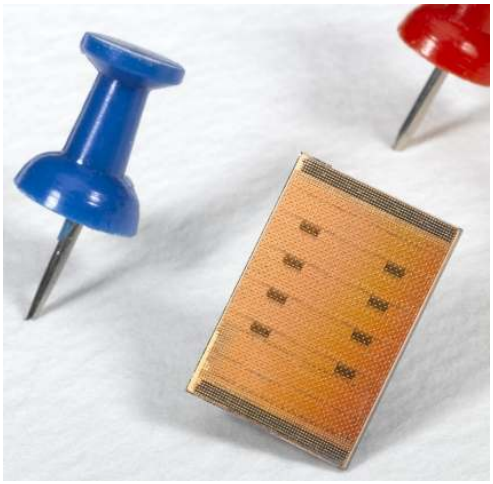
Voditelj: prof.dr.sc. Branko Jeren

Zagreb, svibanj, 2009.

Sadržaj

Uvod

Playstation 3 (**PS3**) igraća konzola sedme generacije tvrtke Sony . Centralni procesor igraće konzole zove se ***Cell*** koji je proizvod zajedničkog rada Sonyja, Toshibe i IBM-a. Cell integrira jedan centralni procesor i 8 jezgri (3.2 GHz). Centralni procesor daje hardversku prednost nad ostalim konkurentima. Cell sadrži jednu glavnu jezgru **PPE** (**PowerPC Processing Element**) koja je bazirana na PowerPC arhitekturi i 8 SPE jezgri (**Synergistic Processing Element**).



Slika 1. Cell procesor

1. Paralelno programiranje

Paralelno programiranje je razvoj paralelnog algoritma koji se izvodi paralelno na više različitih lokacija (računala, procesora, jezgri).

1.1. Modeli paralelnih računala

Osnovna podjela računala po odnosu **programskih instrukcija i podataka**:

- **SISD** (Single Instruction, Single Data)- jedna instrukcija – jedan podatak, koristi se na Von-Neumannovom modelu računala.
- **SIMD** (Single Instruction, Multiple Data Stream)- jedna instrukcija obrađuje više podataka tj. na čitavom vektoru podataka se obavlja ista operacija. Vektorski SIMD ima posebne instrukcije za vektorske podatke. SIMD je tehnika za postizanje podatkovnog paralelizma. Superkompjutori poput Cray X-MP je koristio SIMD arhitekturu. Paralelni SIMD model gdje polje procesora izvodi iste instrukcije na različitim podacima sinkrono u lock-step načinu rada.
- **MISD** (Multiple Instruction, Single Data Stream)- više procesora izvodi različite instrukcije na istim podacima. Npr. isti se podaci analiziraju s različitim neovisnim algoritmima.
- **MIMD** (Multiple Instruction, Multiple Data Stream) – više procesora izvodi različite instrukcije na različite podatke. Prednosti su: paralelno izvođenje više poslova i svaki je procesor neovisan o drugima. Nedostaci su: ujednačavanje opterećenja i teže je programirati za takav model.

Osnovna podjela računala po **memorijskoj strukturi**:

- Model zajedničke memorije (shared memory)- više procesora koristi isti memorijski spremnik ali rade neovisno. Čitanje i pisanje je ekskluzivno: samo jedan procesor u jednom trenutku može čitati ili pisati.
- Model raspodijeljene memorije (distributed memory) – više neovisnih procesora rade sa vlastitim spremnicima. Podjela podataka i sinkronizacija odvija se putem poruka.

- NUMA (Non Uniform Memory Access)- skup paralelnih računala sa zajedničkom memorijom ali sa različitom cijenom pristupa različitim memorijskim lokacijama

Kombinacije modela:

- Višeprocesorsko računalo (zajednička memorija + MIMD)- tu spada PRAM (Parallel Random Access Machine)
- Multi računalo (distribuirana memorija + MIMD) – tu spada APRAM (Asinkroni PRAM)
- Grozd računala (cluster)- skup računala povezanih lokalnom mrežom
- Splet računala (grid)- veći broj računala koji podržavaju splet (grid), različita brzina komunikacija

1.2. Modeli paralelnih programa

Postoje različite paradigme paralelnih programa:

- Komunikacija porukama – više zadataka izvodi se neovisno a podaci između njih se razmjenjuju porukama. Drugi naziv za taj model je SPMD (engl. single program multiple data) tj. jedan program se izvodi na više procesora. Ali unutar programa se mogu implementirati različite uloge (master i worker). Predstavnik paralelnog programiranja s komunikacijom porukama je MPI (engl. Message Passing Interface).
- Podatkovni paralelizam – (engl. dana parallelism) primjena jedne operacije na više elemenata podatkovne strukture. Primjer programski jezik High Performance Fortran
- Zajednička memorija – svi procesori dijele isti memorijski spremnik, gdje je čitanje i pisanje asinkrono. Potrebni eksplicitni mehanizmi zaštite memorije (semafori, mutex i sl.)
- Sustav zadataka i kanala – prikazan je usmjerenim grafom u kojemu su čvorovi zadaci a veze su kanali kojima oni komuniciraju.

1.3. Svojstva paralelnih algoritama

Definiramo poželjna svojstva paralelnih algoritama:

- **Istodobnost** (concurrency) – mogućnost izvođenja više radnji istovremeno
- **Skalabilnost** (scalability) – mogućnost prilagođavanja proizvoljnom broju procesora
- **Lokalnost** (locality) – veći omjer lokalnog u odnosu na udaljeni pristup memoriji
- **Modularnost** (modularity) – mogućnost uporabe različitih dijelova paralelnih algoritama u drugim programima

Amdahl-ov zakon – mjera učinkovitosti ubrzanja paralelnog algoritma. Potencijalno ubrzanje definirano udjelom slijednog programa koji se može paralelizirati. Npr. ako se 40 % programa može paralelizirati onda je ubrzanje 1.667.

```
ubrzanje = 1/(1-P)
```

Uključimo li u mjerenje i broj procesora koji izvode paralelni posao Amdahl-ov zakon se definira ovako:

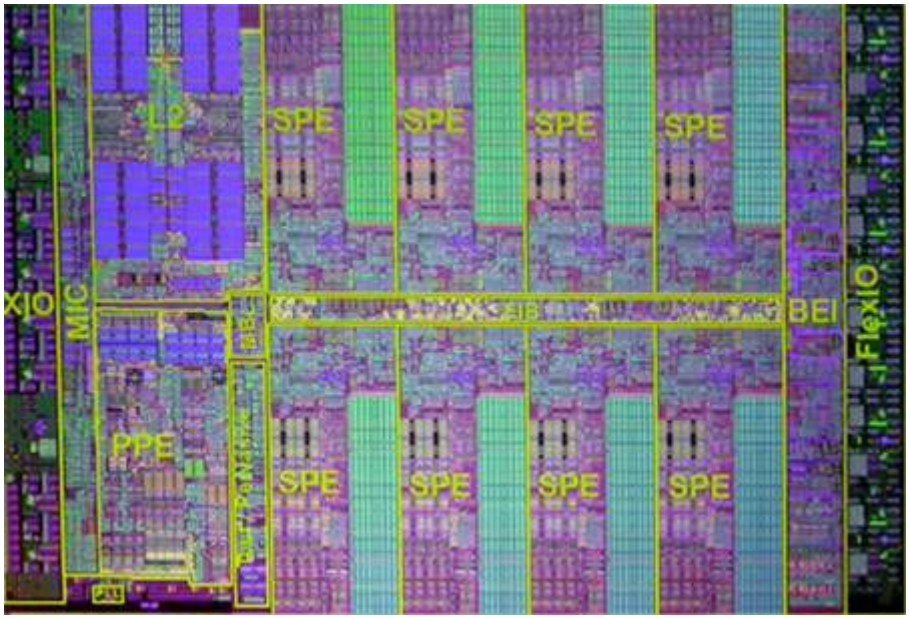
```
ubrzanje = 1/(S+P/N)
```

, gdje je S-slijedni udio programa, P-paralelni udio programa, N-broj procesora.

Druga svojstva paralelnih algoritama:

- **Ujednačenje opterećenja** (load balancing) – raspodjela količine poslova po procesorima u smislu optimalne vremenske iskoristivosti
- **Zrnatost** (granularity) – omjer između količine lokalnog računanja i količine komunikacije. Postoji sitnozrnatost (mala količina računanja između uzastopne komunikacije) i krupnozrnatost (veća količina računanja između uzastopne komunikacije)
- **Podatkovna ovisnost –** postoji kod višestruke uporabe istih memorijskih lokacija što uzrokuje težu paralelizaciju
- **Potpuni zastoj** (dead-lock) – stanje u kojem više procesa čekaju istodobno međusobno na ostvarenje nekog događaja koji se ne ostvaruje nikad.

2. Cell arhitektura

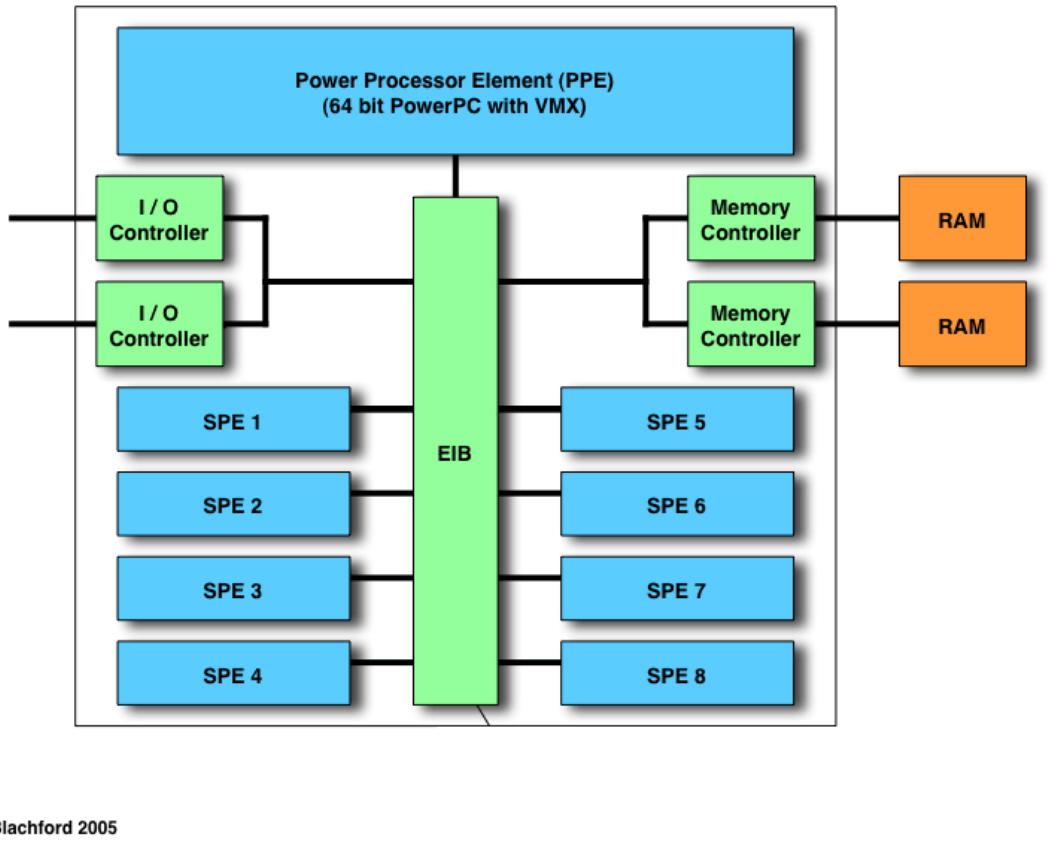


Slika 2. Arhitektura Cell procesora

Podrijetlo ideje o Cell arhitekturi potječe iz 1999 godine iz Japana kada je Ken Kutaragi „otac Playstationa“ razmišljao o računalu koje se ponaša kao biološka stanica. Današnja arhitektura Cella djelo je ove tri kompanije: Sony, Toshiba i IBM. Dizajn samog procesora nije obavljen pomoću automatiziranih alata već ručno za razliku od većine drugih proizvoda te vrste. Iako je Cell procesor primarno namijenjen Playstationu 3 on je puno više od toga. Puni naziv Cell procesora je Cell Broadband Engine koji je prvi proizvod nove familije mikroprocesora CBEA (Cell Broadband Engine Architecture). Cell je multiprocesor sa devet procesora koji rade na zajedničkoj memoriji ali sa različitim ulogama (PPE i SPE).

Cell je nova arhitektura namijenjena visokim performansama distribuiranog računarstva. Vjerujući IBM-u Cell procesori obavljaju posao 10 puta brže od postojećih procesora (CPU). Grafički procesori (GPU) već danas postižu veće performanse od standardnih procesora (CPU) i u ne grafičkim poslovima. Tehnologija Cell-a je slična grafičkim procesorima ali nije tako specifična te može biti korištena za širi skup poslova nego GPU. Cell može ići korak dalje, ne postoji razlog zašto se vaš sustav ne bi mogao distribuirati preko mreže ili interneta. Također je moguće je napraviti ad-hoc mrežu od puno različitih uređaja koji koriste Cell

(PDA, serveri, Playstation). Svaki Cell procesor ima teoretsku računarsku moć od 256 GFLOPS-a.



Slika 3. Arhitektura Cell procesora

Svaki Cell procesor se sastoji od sljedećih elemenata:

- 1. Glavna jezgra PPE (Power Processor Element)
- 8 Pomoćnih jezgri SPE (Synergistic Processing Element)
- Glavna sabirnica EIB (Element Interconnect Bus)
- Izlazni/Ulazni kontroler (76.8 Gb/s propusnost)
- Memorijski kontroler (25.6 Gb/s propusnost)

EIB (Element Interconnect Bus) je glavna sabirnica preko koje komuniciraju PPE procesor i SPE procesori međusobno, s glavnim spremnikom i ulazno/izlaznim kontrolerima.

EIB sadrži dva vanjska sučelja:

- MIC (Memory Interface Controller) kontrolor pruža sučelje između EIB sabirnice i glavnog spremnika.

- BEI (Broadband Engine Interface) kontrolor omogućava podatkovne transfere između EIB sabirnice i ulazni/izlaznih uređaja.

2.1. PPE procesor

U Cell arhitekturi PPE procesor je zadužen za pokretanje operativnog sustava i za izvršavanje većine aplikacija koji nisu računalno zahtjevni. Dok se računalno zahtjevne aplikacije izvršavaju i na SPE procesorima. PPE procesor raspodjeljuje poslove ostalim SPE procesorima. PPE procesor je 64 bitne arhitekture sa 512 K lokalne memorije koji radi na taktu od 4 GHz. PPE procesor je dizajniran kao RISC (Reduced Instruction Set Computing) procesor koji se temelji se na IBM-ovoj PowerPC arhitekturi. Zbog jednostavne arhitekture PPE procesora potrošnja električne energije je mnogo manja od drugih „Power PC“ procesora čak i pri visokim taktovima rada.

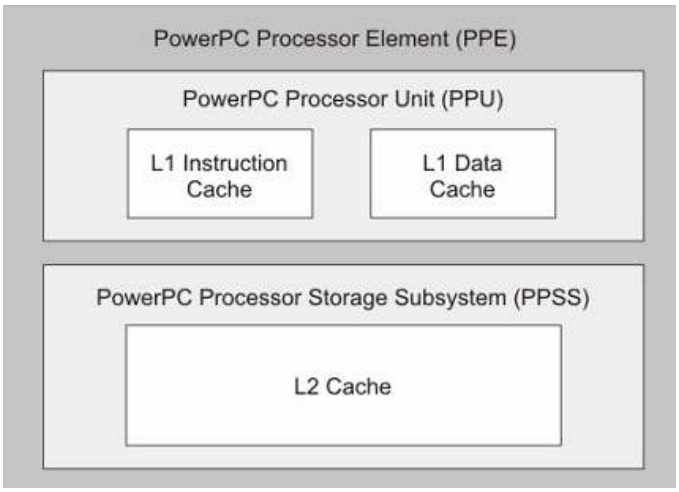
PPE procesor je naslijedio dio tehnologije od IBM-ove „Power“ serije procesora pa tako ima mogućnost izvođenja 2 dretve paralelno. Kada jedna dretva čeka podatke druga dretva može obrađivati instrukcije i tako držati jedinicu za obradu instrukcija zaposlenom. IBM-ova hypervisor tehnologija [1] je korištena u Cell procesoru omogućujući izvođenje više operativnih sustava paralelno.

Još jedna zanimljiva činjenica kod Cell procesora je podržavanje VMX vektor instrukcija[2] (SIMD arhitektura). PPE procesor podržava dva skupa instrukcija: PowerPC skup instrukcija i VMX vektor instrukcije.

---

[1] Hypervisor tehnologija ili VMM (virtual machine monitor) je računalni program/hardware koji omogućava izvedbu više operacijskih sustava na jednom računalu paralelno.

[2] Vektor instrukcije se izvode na vektor procesorima i omogućuju operacije koje se izvode na čitavim vektorima podataka paralelno za razliku od skalarnih procesora koji obrađuju jedan element pomoću više instrukcija.



Slika 4. Prikaz dijelova PPU procesora

PPE procesor se sastoji od dva dijela:

- PPU jedinica (Power Processor Unit)
- PPSS jedinica (Power Processor Storage Subsystem)

PPU jedinica sadrži:

- 64 bitne PowerPC registre
- 32 vektorska registra veličine 128-bita
- 32-KB L1 instrukcijski spremnik
- 32-KB L1 podatkovni spremnik
- druge jedinice (instrukcijska jedinica, jedinica za rad s posmičnim zarezom, …)

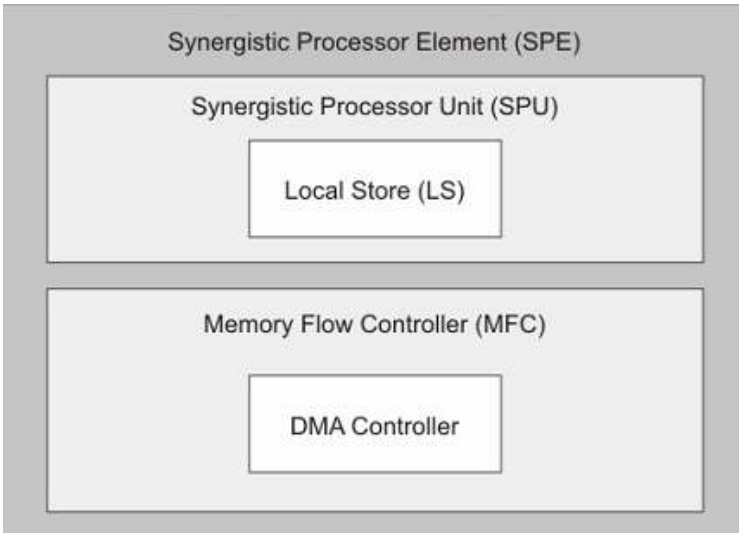
PPSS jedinica obrađuje zahtjeve za pristup glavnom spremniku od PPE procesora i drugih uređaja.

2.2. SPE procesor

Svaki Cell procesor integrira osam SPE (Synergist Processor Element) procesora temeljenih na RISC arhitekturi. Svaki SPE procesor sadrži 256 KB lokalnu memoriju za instrukcije i podatke i 128-bitne privatne registarske spremnike. SPE procesori podržavaju SIMD instrukcije koje se povezuju sa asinkronim DMA prijenosima za prijenos podataka i instrukcija između glavnog spremnika (PowerPC adrese) i lokalne memorije. SPE procesori ne mogu pokretati operativni sustav. SPU jedinica dohvaća instrukcije i podatke iz lokalne memorije.

Svaki SPE procesor se sastoji od sljedećih dijelova:

- SPU jedinica (Synergistic Processor Unit)
- MFC jedinica (Memory Flow Controller)



Slika 5. Prikaz dijelova SPE procesora

SPU jedinica se brine za kontrolu i izvođenje instrukcija. SPU sadrži:

- privatni registarski spremnik sa 128 registara od 128 bita
- zajednički (podaci i instrukcije) 256-KB lokalni spremnik (LS)
- druge jedinice (instrukcijska jedinica, jedinica za rad s posmičnim zarezom, …)

MFC jedinica sadrži DMA kontrolor koji služi za prijenos podataka i instrukcija između lokalne memorije i glavnog spremnika. MFC jedinica sadrži red za pohranu DMA naredbe. Nakon što se neka DMA komanda ubacila u red SPU može nastaviti sa izvršavanjem instrukcija dok MFC jedinica procesira DMA naredbe autonomno i asinkrono. MFC jedinicom može upravljati samo pripadni SPE procesor.

2.3. Pristup memoriji

Postoji znatna razlika u načinu pristupa memoriji između PPE i SPE procesora.

PPE procesor pristupa glavnom spremniku pomoću instrukcija za pisanje i čitanje (load & store) koje potpuno okupiraju procesor za vrijeme pisanja i čitanja u memoriju. Instrukcije za čitanje i pisanje prenose podatke između privatnih registarskih spremnika i glavnog spremnika.

SPE procesor pristupa glavnom spremniku preko DMA[3] kanala koji omogućava prijenos podataka između glavnog spremnika i privatne lokalne memorije koja služi za pohranu instrukcija i podataka bez okupiranja SPE procesorskog vremena. SPE instrukcije za dohvat i spremanje pristupaju lokalnoj memoriji brže nego glavnoj memoriji.

Ovakva podjela spremnika za pohranu podataka (privatne registarske spremnike, lokalna memorija i glavni spremnik) zajedno sa asinkronim DMA prijenosom između lokalne memorije i glavnog spremnika je radikalna promjena u odnosu na druge procesore.

Optimalni model pristup memoriji SPE procesora je onaj u kojemu se lista DMA prijenosa konstruira u lokalnoj memoriji tako da SPE-DMA kontroler može asinkrono obrađivati listu DMA prijenosa dok SPE obrađuje prethodno prenesene podatke. Ovakav način pristupa memoriji i obrade podataka omogućuje 10 puta bolje performanse CBE nad konvencionalnim PC računalom.

[3] DMA- skraćenica od Direct Memory Access, označava vrstu sabirnice koja omogućava vanjskim jedinicama izravan pristup glavnoj memoriji računala za čitanje i pisanje podataka i to bez izravnog posredovanja procesora.

3. Paralelno programiranje na Cell procesoru

Za prevođenje programa za Cell procesor potrebno je imati:

- **Gcc** prevodilac za prevođenje C programa u izvršne programe koji će se izvršavati na PPE procesoru. Potrebno je koristiti –m64 zastavicu za generiranje 64 bitnog izvršnog programa. (IBM SDK koristi `ppu-gcc`)
- **Spu-gcc** prevodilac za prevođenje C programa u izvršne programe za SPE procesor.
- **Embededspu** program koji služi za pretvorbu SPE programa u objekte koji se potom povezuju sa PPE izvršnim programima.(IBM SDK koristi `ppu-embedspu`)

Primjer prevođenja

```
# Prevođenje SPE programa

spu-gcc spe_distance.c -o spe_distance

# Pretvara SPE program (spe_distance) u ELF objekt (spe_distance_csf.o)
# koji je vidljiv preko globalne varijable: calculate_distance_handle

embedspu calculate_distance_handle spe_distance spe_distance_csf.o

# Prevođenje PPE programa zajedno sa SPE programom

gcc ppe_distance.c spe_distance_csf.o -lspe -o distance

#Pokretanje programa
./distance
```

3.1. Jednostavni primjer SPE i PPE programa u C/C++

```
// ppe_distance.c

#include <stdio.h>
#include <libspe.h>

//This global is for the SPE program code itself.  It will be created by
//the embedspu program.
extern spe_program_handle_t calculate_distance_handle;

//This struct is used for input/output with the SPE task
typedef struct {
        float speed;     //input parameter
        float num_hours; //input parameter
        float distance;  //output parameter
        float padding;   //pad the struct a multiple of 16 bytes
} program_data;
```

```
int main() {
        program_data pd __attribute__((aligned(16)));  //aligned for transfer

        //GATHER DATA TO SEND TO SPE
        printf("Enter the speed at which your car is travelling in miles/hr: ");
        scanf("%f", &pd.speed);
        printf("Enter the number of hours you have been driving at that speed: ");
        scanf("%f", &pd.num_hours);

        //USE THE SPE TO PROCESS THE DATA
        //Create SPE Task
        speid_t spe_id = spe_create_thread(0, &calculate_distance_handle, &pd, NULL,-1, 0);
        //Check For Errors
        if(spe_id == 0) {
                fprintf(stderr, "Error creating SPE thread!\n");
                return 1;
        }
        //Wait For Completion
        spe_wait(spe_id, NULL, 0);

        //FORMAT THE RESULTS FOR DISPLAY
        printf("The distance travelled is %f miles.\n", pd.distance);
        return 0;
}
```

Funkcija *spe_create_thread(0, &calculate_distance_handle, &pd, NULL,-1, 0)* posjeduje sljedeće parametre:

- Prvi parametar funkcije *spe_create_thread* je ID kreirane dretve (0 označava da će se kreirati nova grupa za ovu dretvu).
- Drugi parametar *calculate_distance_handle* je pokazivač na SPE program.
- Treći parametar *pd* je pokazivač na podatke za slanje i primanje.
- Četvrti parametar je dodatni pokazivač okoline (optional environment pointer).
- Peti parametar predstavlja oznaku SPE procesora na kojem će se izvršavati program (-1 opcija označava bilo koji SPE procesor)
- Zadnji parametar je lista dodatnih opcija (0 nema opcija)

```
//Pull in DMA commands
#include <spu_mfcio.h>

//Struct for communication with the PPE
typedef struct {
        float speed;     //input parameter
        float num_hours; //input parameter
        float distance;  //output parameter
        float padding;   //pad the struct a multiple of 16 bytes
} program_data;
```

```
int main(unsigned long long spe_id, unsigned long long program_data_ea, unsigned
long long env) {
        program_data pd __attribute__((aligned(16)));
        int tag_id = 0;

        //READ DATA IN
        //Initiate copy
        mfc_get(&pd, program_data_ea, sizeof(pd), tag_id, 0, 0);
        //Wait for completion
        mfc_write_tag_mask(1<<tag_id);
        mfc_read_tag_status_any();

        //PROCESS DATA
        pd.distance = pd.speed * pd.num_hours;

        //WRITE RESULTS OUT
        //Initiate copy
        mfc_put(&pd, program_data_ea, sizeof(program_data), tag_id, 0, 0);
        //Wait for completion
        mfc_write_tag_mask(1<<tag_id);
        mfc_read_tag_status_any();
        return 0;
}
```

Treći parametar pokazivač *pd* prilikom poziva funkcije *spe_create_thread* dolazi u SPE program kao *program_data_ea.* EA je oznaka za efektivnu adresu u glavnom spremniku od strane PPE programa. Pošto SPE nema direktan pristup glavnom spremniku nemoguće je direktno dereferencirati pokazivač *pd.* Zato je potrebno pokrenuti prijenos podataka iz glavnog spremnika u lokalnu memoriju SPE procesora. Jednom kada se podaci nalaze u lokalnoj memoriji SPE procesor im može pristupiti preko lokalnih adresa (LSA- local store adress). Funkcija *mfc_get(&pd, program_data_ea, sizeof(pd), tag_id, 0, 0)* pokreće prijenos podataka u lokalnu memoriju (MFC- Memory Flow Controller). Struktura za prijenos podataka *program_data* je u PPE i SPE programima je skraćena na 16 bajta zbog DMA prijenosa. Pomoću varijable *tag_id* možemo dohvatiti podatke o statusu DMA operacija.

Funkcije *mfc_write_tag_mask* i *mfc_read_tag_status_any* omogućavaju čekanje dok prijenos podataka ne završi. Pomoću funkcije *mfc_put* zapisuju se nazad rezultati iz lokalne memorije u glavni spremnik kako bi PPE procesor mogao dobiti rezultate izračuna.

3.2. Programiranje na SPU procesorima u C/C++

Za uključivanje SPU C/C++ programskih ekstenzija potrebno je uključiti `spu_intrinsics.h`.

3.2.1. Vektorske instrukcije na SPU

Glavna razlika između vektorskih i skalarnih procesora je da vektorski procesori imaju velike registre u koje spremaju veći broj elemenata istog tipa i obrađuju ih sa jednom vektorskom operacijom odjednom. Zbog toga je bilo potrebno uvesti taj novi koncept u C/C++ programski jezik sa posebnom ključnom riječi *vector* . Npr. naredba `vector unsigned int myvec` kreira 4 integer vektor *myvec* gdje se elementi zajedno učitavaju, spremaju ili obrađuju pomoću neke operacije. Ključna riječ `signed/unsigned` je potrebna za deklaracije bez pomične točke (non-floating point). Vektorski elementi konstante se u vektor dodaju navođenjem njihovih vrijednosti u vitičastim zagrada ispred kojih stoji oznaka tipa vektora.

```
vector unsigned int myvec = (vector unsigned int){1, 2, 3, 4};
```

Osim direktnog pridruživanja vrijednosti vektora postoje četiri osnovne naredbe koje služe prebacivanju skalarnih vrijednosti u vektorske vrijednosti i obrnuto.

- *Spu_insert* služi za ubacivanje skalarne vrijednosti na određeno mjesto u vektoru. Npr naredba `spu_insert(x, myvec, y)` vraća kopiju vektora `myvec` gdje je na poziciji `y` postavljena vrijednost `x`.
- *Spu_extract* vadi skalarnu vrijednost iz vektora. Npr. `spu_extract(myvec, x)` vraća skalar koji se nalazi na poziciji `x` u vektoru *myvec*.
- *Spu_promote* pretvara skalar u vektor. Tip vektora ovisi o tipu skalara koji se prevara u vektor. Npr naredba *spu_promote((unsigned int) x, y)* kreira vektor tipa *unsigned int* sa elementom x na poziciji y dok ostali elementi su nedefinirani (vektor je fiksne dužine 128 bajta – 4 * 32 bajta = 4 integera).
- *Spu_splats* radi kao i *spu_promote* osim što kopira skalarnu vrijednost na sve elemente vektora. Npr. *Spu_splats((unsigned int) x)* kreira vektor tipa *unsigned int* gdje svi elementi imaju istu vrijednost `x`.

Smatrati vektore kao kratke nizove fiksne veličine je krivo jer se ponašaju drugačije. Nizovima manipuliramo preko referenci dok sa vektorima nije tako. Npr. *Spu_insert* ne

modificira postojeći vektor već vraća novu kopiju vektora sa ubačenim skalarnim elementom. Operacije nad vektorima rezultiraju novim kopijama a ne modifikacijama već postojećim vektora.

```
#include <spu_intrinsics.h>

void print_vector(char *var, vector unsigned int val) {
        printf("Vector %s is: {%d, %d, %d, %d}\n", var, spu_extract(val, 0),
         spu_extract(val, 1), spu_extract(val, 2), spu_extract(val, 3));
}

int main() {
        /* Kreiranje četiri vektora */
        vector unsigned int a = (vector unsigned int){1, 2, 3, 4};
        vector unsigned int b;
        vector unsigned int c;
        vector unsigned int d;

        /* b je identičan vektoru a, osim zadnjeg elementa koji je promijenjen na 9 */
        b = spu_insert(9, a, 3);

        /* c vektor ima sva četiri elementa postavljena na 20 */
        c = spu_splats((unsigned int) 20);

        /* d ima drugi element postavljen na 5 dok su ostali elementi nedefinirani */
        d = spu_promote((unsigned int)5, 1);

        /* Prikaži rezultate */
        print_vector("a", a);
        print_vector("b", b);
        print_vector("c", c);
        print_vector("d", d);

        return 0;
}
```

Prevođenje i pokretanje programa na SPU procesoru se izvodi ovako:

```
spu-gcc vec_test.c -o vec_test
./vec_test
```

3.2.2. Vektorske intrinzične naredbe (engl. Vectors intrinsics)

Programske ekstenzije jezika C/C++ uključuju tipove podataka i intrinzične naredbe (intrinsics) omogućavaju pristup hardverskim dijelovima i SPU asemblerskim naredbama koje nisu jednostavno dostupne iz viših programskih jezika (C/C++). Postoje tri tipa vektorskih intrinzičnih naredbi:

- SI – (engl. Specific Intrinsics) specifične intrinzične naredbe imaju preslikavanje jedan na jedan sa SPU asemblerskim instrukcijama. Imaju prefiks si_ npr. si_cbd, si_lql , itd.
- GI – (engl. Generic instrinsics) generičke intrinzične naredbe su operacije koje se preslikavaju u jednu ili više SI naredbi. Imaju prefiks spu_.
- BI- (engl. Build-intrinsics) kompozitne intrinzične naredbe su instrukcije koje se preslikavaju u više SPU asemblerskih instrukcija. Imaju prefiks spu_.

Asemblerske instrukcije koje se razlikuju samo u tipu operanda su reprezentirani jednom C/C++ intrinzičnom naredbom koja selektira potrebnu asemblersku instrukciju u ovisnosti o tipu operanda. Npr. intrinzična naredba *spu_add* kada kao operande dobije dva vektora tipa `unsigned int` generira jednu „*a*“ 32-bitnu add-instrukciju ali ako se na ulazu nađu operandi vektori tipa `float` generira se jedna „*fa*“ (add instrukcija sa pomičnom točkom). Ukoliko je ulazni operand prevelik za neki instrukciju prevodilac će automatski promovirati skalarni operand u vektor i generirati vektorsku instrukciju koja će napraviti operaciju nad vektorima. Npr. *spu_add(myvec, 2)* generira „*ai*“ (add instrukciju) dok naredba *spu_add(myvec, 2000)* prvo promovira skalar 2000 u vektor i potom obavi „*ai*“ instrukciju.

Sve intrinzične naredbe sa prefiksom spu_ će pokušati pronaći najbolje SPE asemblerske instrukcije ovisno o ulaznim parametrima. Ukoliko ne želimo dopustiti prevodiocu da nam automatski pronalazi asemblerske instrukcije moguće je koristiti SI intrinzične naredbe (engl. Specific instrisics) sa prefiksom `si_nazivAsemblerskeInstrukcija`.

3.2.3. Korištenje intrinzičnih naredbi

Program radi konverziju malih slova u velika slova koristeći C/C++ programski jezik. Osnovni koraci prilikom pretvorbe jednog vektora su:

- Konvertiraj sve vrijednosti koristeći naredbu konverzije u velika slova.
- Napraviti vektor usporedbe svih bajta radi utvrđivanja da li su svi znakovi između slova 'a' i slova 'z'.
- Koristeći vektor usporedbe za odluku između konvertiranih i ne konvertiranih vrijednosti koristeći select instrukciju

U C/C++ moguće je pozivati in-line funkcije više puta i dopustiti prevoditelju da se brine o raspoređivanju izvršavanja instrukcija. Prikazana je C/C++ verzija *convert_buffer_to_upper* funkcija.

```
#include <spu_intrinsics.h>

unsigned char conversion_value = 'a' - 'A';

inline vec_uchar16 convert_vec_to_upper(vec_uchar16 values) {
        /* Process all characters */
        vec_uchar16 processed_values = spu_absd(values, spu_splats(conversion_value));
        /* Check to see which ones need processing (those between 'a' and 'z')*/
        vec_uchar16 should_be_processed = spu_xor(spu_cmpgt(values, 'a'-1),
        spu_cmpgt(values, 'z'));
        /* Use should_be_processed to select between the original and processed values */
        return spu_sel(values, processed_values, should_be_processed);
}

void convert_buffer_to_upper(vec_uchar16 *buffer, int buffer_size) {
        /* Find end of buffer (must be casted first because size is bytes) */
        vec_uchar16 *buffer_end = (vec_uchar16 *)((char *)buffer + buffer_size);

        while(__builtin_expect(buffer < buffer_end, 1)) {
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
        }
}
```

U ovom primjeru koristi se drugačija notacija za vektorska imena *vec_* . Za integer tip podataka idući znak u notaciji za vektorska imena je *u/s* ovisno da li se radi o tipu bez

predznaka ili s predznakom (unsigned/signed). Nakon toga navodi se osnovni tip podataka koji se koristi (*char, int, float itd).* A na kraju se nalazi broj koji označava broj elemenata u tom vektoru. Npr. `vec_ float4` označava vektor od 4 elementa tipa float.

Prvo se izračunala adresa kraja *buffera* tako da se napravi operacija cast na tip *char* i onda se tek zbroji veličina bufera u bajtima i ponovo napravi operacija cast na željeni tip. Vektorski pokazivač *buffer* pokazuje na vektore dugačke 128 bita (16 bajta) a svaki vektor sadrži 16 elemenata char koji svaki zauzima 1 bajt. Zato inkrementacija adrese pokazivača za 1 vrši pomak od 16 bajta u memoriji. Dok inkrementacija adrese pokazivača koji pokazuje na char vrši pomak od 1 bajta u memoriji.

Moguće je koristiti naredbu *__builtin_expect* koja pomaže prevodiocu odrediti kakav asemblerski kod će generirati s obzirom na neku pretpostavku o grananju. Ovdje je korištena pretpostavka da će adresa početka buffera biti manja od adrese kraja buffera. Pomoću te pretpostavke generira se asemblerski kod za while petlju.

Slijedeće naredbe govore prevodiocu da se ne očekuje da uvjet x bude ispunjen nego se očekuje da uvjet x ne bude ispunjen jer je drugi argument nula.

```
If (__builtin_expect(x, 0))
        Foo();
```

3.2.4. Kompozitne intrinzične naredbe i MFC programiranje

Kompozitne intrinzične naredbe se prevode u više asemblerskih instrukcija. Kompozitne intrinzične naredbe enkapsuliraju česte korištene obrasce na SPE radi jednostavnijeg programiranja. Dvije najvažnije kompozitne intrinzične naredbe su:

- *Spu_mfcdma64*- koristi šest parametara:
    - Lokalna memorijska adresa za prijenos
    - Viših 32 bita efektivne adrese
    - Nižih 32 bita efektivne adrese
    - Veličina prijenosa podataka
    - Oznaka (tag) prijenosa podataka
    - DMA naredba
- *Spu_mfcstat*

Za razdvajanje 64 bitne efektivne adrese na niže i više bitove koristi se naredba *mfc_ea2h* za ekstrakciju viših bita i *mfc_ea2l za ekstrakciju nižih bitova.* Oznaka (tag) je broj između 0 i 31 koji se koristi kao identifikacjia transfera ili grupe transfera. DMA naredbe čiji dio imena sadrži naziv *PUT* označava grupu transfera iz SPU lokalne memorije u glavni spremnik dok naziv *GET* označava suprotni transfer. Stoga ovakve naredbe imaju prefiks *MFC_PUT* ili *MFC_GET*. MFC naredbe rade samostalno ili u listi naredbi. Ako je DMA naredba naredba liste, DMA naredba u imenu ima sufiks *L*. DMA naredbe također mogu imati ugrađen određeni stupanj sinkronizacije. Za sinkronizaciju barijerom (barrier) dodaje se sufiks B, za sinkronizaciju ogradom (fence) dodaje se sufiks F u ime. Konačno sve DMA naredbe imaju *_CMD* sufiks u imenu.

DMA naredbe na MFC redu unutar SPE-a MFC obrađuje proizvoljnim redom. Ali ograde, oznake (tags) i barijere mogu se koristiti kako bi prisilili MFC obradu DMA transfera u željenom redoslijedu. Korištenje ograde postavlja se uvjet da se pripadni DMA transfer izvršava tek nakon što su se **sve prijašnje** DMA naredbe s istom oznakom završile. Korištenje barijere postavlja se uvjet da se pripadni DMA transfer izvršava tek nakon što su se **sve** DMA naredbe s istom oznakom završile.

```
typedef unsigned long long uint64;
typedef unsigned long uint32;
uint64 ea1, ea2, ea3, ea4, ea5; /* assume each of these have sensible
values */
void *ls1, *ls2, *ls3, *ls4; /* assume each of these have sensible values
*/
uint32 sz1, sz2, sz3, sz4; /* assume each of these have sensible values
*/
int tag = 3; /* Arbitrary value, but needs to be the same for all
synchronized transfers */

/* Transfer 1: System Storage -> Local Store, no ordering specified */
spu_mfcdma64(ls1, mfc_ea2h(ea1), mfc_ea2l(ea1), sz1, tag, MFC_GET_CMD);

/* Transfer 2: Local Storage -> System Storage, must perform after
previous transfers */
spu_mfcdma64(ls2, mfc_ea2h(ea2), mfc_ea2l(ea2), sz2, tag, MFC_PUTF_CMD);

/* Transfer 3: Local Storage -> System Storage, no ordering specified */
spu_mfcdma64(ls3, mfc_ea2h(ea3), mfc_ea2l(ea3), sz3, tag, MFC_PUT_CMD);

/* Transfer 4: Local Storage -> System Storage, must be synchronized */
spu_mfcdma64(ls4, mfc_ea2h(ea4), mfc_ea2l(ea4), sz4, tag, MFC_PUTB_CMD);

/* Transfer 5: System Storage -> Local Storage, no ordering specified */
spu_mfcdma64(ls4, mfc_ea2h(ea5), mfc_ea2l(ea5), sz4, tag, MFC_GET_CMD);
```

Gornji primjer ima nekoliko mogućih načina redoslijeda izvršavanja DMA naredbi. Ovo su ti slučajevi:

- Transfer 1, Transfer 2, Transfer 3, Transfer 4, Transfer 5

- Transfer 3, Transfer 1, Transfer 2, Transfer 4, Transfer 5
- Transfer 1, Transfer 3, Transfer 2, Transfer 4, Transfer 5

Transfer 2 koristi ogradu a transfer 3 ne koristi sinkronizacijske mehanizme pa može biti na proizvoljnoj poziciji reda izvođenja ali prije transfera 4 jer on koristi barijeru. Tu se vidi razlika između korištenja barijere i ograde kao dva sinkronizacijska mehanizma. Kod korištenja ograde transfer 3 može se izvršiti prije transfera sa ogradom transfer 2 jer je transfer 3 definiran poslije transfera 2 sa ogradom. Ograda transferu 2 osigurava jedino da se transfer 1 izvrši prije njega dok transfer 4 sa barijerom zahtjeva sinkronizaciju sa svim transferima prije i poslije definiranih.

Pažljivije pogledajte transfer 4 i transfer 5. Transfer 4. sprema podatke veličine *sz4* iz lokalne memorije s adrese *ls4* na adresu *ea4* u glavnom spremniku sa sinkronizacijskim mehanizmom barijere (engl. *barrier).* Transfer 5. čita iz glavnog spremnika s adrese *ea5* u lokalni spremnik na adresu *ls4*. Zbog korištenja iste lokalne adresa u transferu 4 i transferu 5. potrebno je koristiti sinkronizacijski mehanizam. Ovakav način spremanja i čitanja podataka prepušta MFC kontroleru autonomni rad spremanja i pisanja i za to vrijeme SPE procesor može raditi druge stvari. (Za naprednije korištenje ovog koncepta pogledati poglavlje double buffering.)

Funkcija *spu_mfcdma64* je korisna ali je potrebno eksplicitno pretvarati adrese pomoću naredbe *mfc_ea2h* i *mfc_ea2l.* Postoje naredbe u mfc_ klasi koje omogućavaju jednostavnije korištenje DMA transfer naredbi. Ulazni parametri u takve naredbe su 64 bitne efektivne adrese i DMA:

```
Mfc_putf ( ls2, ea2, sz2, tag, 0, 0 )
```

Dodatna dva parametra označavaju identifikatore klase za transfer i klase za zamjenu (engl. *transfer and the replacement class identifier).*

Oznake transfera (engl. *tag)* su korisne za provjeru statusa napredovanja transfera. Na SPE procesoru postoji kanal za maskiranje oznaka (engl. *tag mask channel)* koji služi za specifikaciju oznaka koji se trenutačno koriste za provjeru statusa. Postoji kanal koji je odgovoran za slanje zahtjeva statusa i kanal koji je odgovoran za vraćanje statusa. Funkcija *mfc_write_tag_mask* uzima varijablu (32-bitni integer) i koristi ga kao kanal za buduće promjene statusa. U maski se postave vrijednosti bitova svake oznake kojoj želimo provjeriti status na vrijednost 1. Npr. za provjeru statusa kanala 2 i 4 potrebno je postaviti vrijednost maske na 20 tj. *mfc_write_tag_mask(20).* Točnije vrijednost maske se postavlja na vrijednost

(1<<2[4] | 1<<4) što je upravo jednako 20. Za promjenu vrijednosti statusa potrebno je odabrati pravu statusnu naredbu i poslati je pomoću funkcije *spu_mfcstat(unsigned int command)*.

Naredbe za status su:

- *MFC_TAG_UPDATE_IMMEDIATE* – ova naredba uzrokuje vraćanje statusa DMA naredbe odmah. Svaki kanal koji je označen u kanalu maske će dobiti vrijednost 1 ako više nema naredbi s pripadnom oznakom u redu MFC-a (sve operacije koje su bile aktivne s tom oznakom su završile).
- *MFC_TAG_UPDATE_ANY* – ova naredba uzrokuje čekanje u SPE procesoru sve dok za barem jednu oznaku nema više naredbi u redu MFC-a. Onda se vraća status DMA kanala.
- *MFC_TAG_UPDATE_ALL* – ova naredba uzrokuje čekanje u SPE procesoru sve dok za **sve** oznake ne postoji više naredbe koje se izvode. Onda vraća vrijednost 0.

Za korištenje ovih naredbi potrebno je uključiti *spu_mfcio.h.* Korištenje naredbe *spu_mfcstat* dopušta čekanje i provjeru statusa DMA naredbi. Korištenje naredbe *MFC_TAG_UPDATE_ANY* omogućava pokretanje više DMA zahtjeva i dopuštanje MFC-u njihovo izvršavanje u proizvoljnom redoslijedu dok naš program odgovora ovisno o redoslijedu izvršavanja koje je MFC odabrao.

Koristeći novo znanje možemo isti program za konverziju znakova napisati optimalnije (convert_driver_c.c).

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
typedef unsigned long long uint64;

#define CONVERSION_BUFFER_SIZE 16384
#define DMA_TAG 0

void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);

char conversion_buffer[CONVERSION_BUFFER_SIZE];

typedef struct {
        int length __attribute__((aligned(16)));
        uint64 data __attribute__((aligned(16)));
} conversion_structure;

int main(uint64 spe_id, uint64 conversion_info_ea) {
        conversion_structure conversion_info; /* Information about the data from the PPE */
```

[4] (1 << 2) – označava broj koji se dobije operacijom lijevog pomicanja (left shift) broja 1 za 2 bita.

```
        /* We are only using one tag in this program */
        mfc_write_tag_mask(1<<DMA_TAG);

        /* Grab the conversion information */
        mfc_get(&conversion_info, conversion_info_ea, sizeof(conversion_info), DMA_TAG,0,0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */

        /* Get the actual data */
        mfc_get(conversion_buffer, conversion_info.data,conversion_info.length, DMA_TAG,0,0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        /* Perform the conversion */
        convert_buffer_to_upper(conversion_buffer, conversion_info.length);

        /* Put the data back into system storage */
        mfc_put(conversion_buffer, conversion_info.data, conversion_info.length, DMA_TAG,0,0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

Ovaj primjer se prevodi pomoću slijedećih naredbi:

```
spu-gcc convert_buffer_c.c convert_driver_c.c -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert
spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o
dma_convert
./dma_convert
```

Ovaj primjer je ograničen na veličinu DMA transfera. Što ako želimo maknuti to ograničenje? Možemo sve zavrtjeti u petlji i prebacivati komad po komad podataka sve dok ne prebacimo željenu količinu. Sljedeći primjer pokazuje SPE program koji omogućuje DMA transfer veće količine podataka.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h> /* constant declarations for the MFC */
typedef unsigned long long uint64;
typedef unsigned int uint32;

/* Renamed CONVERSION_BUFFER_SIZE to MAX_TRANSFER_SIZE because it is now
primarily used to limit the size of DMA transfers */
#define MAX_TRANSFER_SIZE 16384

void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);

char conversion_buffer[MAX_TRANSFER_SIZE];

typedef struct {
        uint32 length __attribute__((aligned(16)));
        uint64 data __attribute__((aligned(16)));
} conversion_structure;


int main(uint64 spe_id, uint64 conversion_info_ea) {
        conversion_structure conversion_info; /* Information about the data from the PPE */

        /* New variables to keep track of where we are in the data */
        uint32 remaining_data; /* How much data is left in the whole string */
        uint64 current_ea_pointer; /* Where we are in system memory */
```

```
        uint32 current_transfer_size; /* How big the current transfer is (may be smaller
        than MAX_TRANSFER_SIZE) */

        /* We are only using one tag in this program */
        mfc_write_tag_mask(1<<0);

        /* Grab the conversion information */
        mfc_get(&conversion_info, conversion_info_ea, sizeof(conversion_info), 0, 0, 0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */

        /* Setup the loop */
        remaining_data = conversion_info.length;
        current_ea_pointer = conversion_info.data;

        while(remaining_data > 0) {
                /* Determine how much data is left to transfer */
                if(remaining_data < MAX_TRANSFER_SIZE)
                        current_transfer_size = remaining_data;
                else
                        current_transfer_size = MAX_TRANSFER_SIZE;

                /* Get the actual data */
                mfc_getb(conversion_buffer, current_ea_pointer, current_transfer_size,0,0,0);
                spu_mfcstat(MFC_TAG_UPDATE_ALL);

                /* Perform the conversion */
                convert_buffer_to_upper(conversion_buffer, current_transfer_size);

                /* Put the data back into system storage */
                mfc_putb(conversion_buffer, current_ea_pointer, current_transfer_size,0,0,0);

                /* Advance to the next segment of data */
                remaining_data -= current_transfer_size;
                current_ea_pointer += current_transfer_size;
        }
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

Sada smo proširili veličinu podataka za mogući prijenos na 4 gigabajta ali čak i više ako postavimo varijablu veličine podataka *uint32 length* na 64 bitnu. Nije potrebno eksplicitno navoditi da SPE procesor čeka završetak DMA transfera spremanja i čitanja jer se koristi sinkronizacijski mehanizam barijere i iste oznake DMA prijenosa. Pripazite na postavljanje naredbe *spu_mfcstat*(MFC_TAG_UPDATE_ALL) poslije while petlje jer se inače svi podaci zadnjeg prijenosa ne bi nužno morali zapisati prije nego program završi.

3.3. Savjeti za programiranje u C/C++

- Vektori se mogu pretvoriti u druge tipove podataka i obrnuto.
- Vektorski i ne vektorski pokazivači se mogu pretvoriti međusobno jednu u druge. Ali kada se pretvara iz skalarnog pokazivača u vektorski zadaća je programera da je pokazivač poravnat na dužinu 64 bita (engl. *quadword-alligned).*
- Deklarirani vektori su uvijek poravnati na dužinu 64 bita (engl. *quadword-alligned)* prilikom alociranja
- DMA transferi moraju biti poravnati na dužine višekratnike 16-bajta na SPE i PPE procesoru. Manji transferi od 16 bajta moraju biti potencije broja 2. Optimalni transferi su višekratnici 128 bajta
- Ako niste sigurni u poravnavanje podataka na PPE procesoru koristite *memalign* ili *posix_memalign* funkcije za alociranje pokazivača koji si poravnati na heap-u. Koristite funkciju *memcopy* za pomicanje poravnatih podataka.
- Uvijek prevodite programe sa zastavicom –Wall i vodite računa o upozorenjima prevoditelja za nedostajuće prototipove. Netočni prototipovi između funkcija koje rade sa 32-bitnih i 64-bitnih podacima može dovesti do ozbiljnim grešaka.
- Efektivne adrese uvijek deklarirajte kao *unsigned long long* tako će se tretirati isto prilikom 32 bitnog ili 64 bitnog prevođenja (?)
- Izbjegavajte množenje integer brojeva s 32 bita. Potrebno je 5 instrukcija za izvođenje ove operacije. Ako baš morate prvo broj pretvorite u *unsigned short*
- U skalarnom kodu na SPE procesoru deklariranje skalarnih vrijednosti kao vektora i vektorskih pokazivača (čak i ako ih ne koristite kao vektore) može ubrzati kod jer se ne mora obavljati operacija store i load sa ne poravnatim podacima.
- Imajte na umu da na SPE procesoru tipovi *float i double* su drugačije implementirani i drugačije se zaokružuju također.

3.4. MFC programiranje s dvostrukim spremnikom (engl.*Double-buffering)*

Programi napisani do sada su uvijek slijedili jednostavan obrazac:

- SPU procesor postavi u red MFC-a DMA naredbu za dohvat podataka iz glavnog spremnika u lokalnu memoriju
- SPU procesor čeka MFC dok ne napuni lokalni spremnik podacima
- SPU obrađuje podatke
- SPU procesor postavi u red MFC-a DMA naredbu za spremanje podataka u glavni spremnik
- SPU čeka dok MFC ne spremi podatke

Problem u ovom pristupu je gubljenje dosta vremena na čekanje SPU procesorskog vremena dok se podaci dohvaćaju ili spremaju pomoću MFC kontrolera. Umjesto čekanja i gubljenja SPU procesorskih ciklusa čekajući podatke možemo imati dodatni spremnik (engl. *buffer*) podataka koji također trebamo obrađivati. Dok čekamo da se jedan spremnik napuni podacima s drugim spremnikom možemo raditi. Novi obrazac programiranja s korištenjem dva spremnika:

1. SPU procesor postavi u red MFC-a DMA naredbu za dohvat podataka iz glavnog spremnika u spremnik 1 u lokalnoj memoriji
2. SPU procesor postavi u red MFC-a DMA naredbu za dohvat podataka iz glavnog spremnika u spremnik 2 u lokalnoj memoriji
3. SPU procesor čeka da se spremnik 1 napuni podacima
4. SPU obrađuje podatke iz spremnika 1
5. SPU procesor postavi u red MFC-a DMA naredbu za spremanje podataka iz spremnika 1 u glavni spremnik i odmah zatim postavi DMA naredbu za dohvat novih podataka u spremnik 1 ali sa sinkronizacijskim mehanizmom barijere
6. SPU procesor čeka spremnik 2 da se napuni podacima
7. SPU procesor obrađuje podatke iz spremnika 2

8. SPU procesor postavi u red MFC-a DMA naredbu za spremanje podataka iz spremnika 2 u glavni spremnik i odmah zatim postavi DMA naredbu za dohvat novih podataka u spremnik 2 ali sa sinkronizacijskim mehanizmom barijere
9. Ide na korak 3 sve dok se ne obrade svi podaci.

Ovaj obrazac programiranja također ima nepotrebnog posla koje obavlja SPU ali je znatni pomak u performansama u odnosu na korištenje jednog spremnika. Također moguće je postaviti provjeru uvjeta da li smo gotovi s punjenjem svih podataka iz glavnog spremnika za jedan od spremnika. Ali takav način nam dodatno usporava program ukoliko prenosimo velike količine podataka. MFC kontroler DMA transfere bez podataka obrađuje kao operaciju koja ne troši puno vremena.

Također nakon svake obrade podataka u spremniku postavljaju se dvije DMA naredbe *PUT i GETB*. Druga naredba je postavljena zajedno sa sinkronizacijskim mehanizmom barijere što znači da će se sve naredbe koje su došle u red MFC-a prije naredbe *GETB* biti obrađene prije a također se nijedna buduća naredba ne može izvršiti prije naredbe *GETB*. Naravno prethodna izjava vrijedi samo za naredbe koje imaju istu oznaku kao i naredbe *GETB*.

Slijedi primjer već prije obrađenog programa za konverziju malih znakova u velike znakove ali sada pomoću dvostrukog spremnika.

Koristimo strukturu *buffer* koja sadrži:

- efektivnu adresu spremnika u glavnoj memoriji
- veličinu spremnika
- adresu spremnika u lokalnoj memoriji *data*

```
struct {
        uint64 effective_address __attribute__((aligned(16)));
        uint32 size __attribute__((aligned(16)));
        char data[MAX_TRANSFER_SIZE] __attribute__((aligned(16)));
} buffer;
```

Koristimo globalno polje gdje spremamo adrese spremnika u lokalnoj memoriji:

```
buffer buffers[2];
```

Zatim potrebno je podijeliti mehanizam konverzije u dva funkcijska dijela:

1. Inicijalizacija dohvaćanja podataka
2. Čekanje, obrada podataka i spremanje u spremnik podataka

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
/* Constants */
#define MAX_TRANSFER_SIZE 16384

/* Data Structures */
typedef unsigned long long uint64;
typedef unsigned int uint32;
typedef struct {
        uint32 length __attribute__((aligned(16)));
        uint64 data __attribute__((aligned(16)));
} conversion_structure;

typedef struct {
        uint32 size __attribute__((aligned(16)));
        uint64 effective_address __attribute__((aligned(16)));
        char data[MAX_TRANSFER_SIZE] __attribute__((aligned(16)));
} buffer;

/* Global Variables */
buffer buffers[2];

/* Utility Functions */
inline uint32 MIN(uint32 a, uint32 b) {
        return a < b ? a : b;
}

inline void wait_for_completion(uint32 mask) {
        mfc_write_tag_mask(mask);
        spu_mfcstat(MFC_TAG_UPDATE_ALL);
}

inline void load_conversion_info(uint64 cinfo_ea, uint64 *data_ea, uint32 *data_size) {
        conversion_structure cinfo;
        mfc_get(&cinfo, cinfo_ea, sizeof(cinfo), 0, 0, 0);
        wait_for_completion(1<<0);
        *data_size = cinfo.length;
        *data_ea = cinfo.data;
}

/* Processing Functions */
inline void initiate_transfer(uint32 buf_idx, uint64 *current_ea_pointer,
uint32 *remaining_data) {
        /* Setup buffer information */
        buffers[buf_idx].size = MIN(*remaining_data, MAX_TRANSFER_SIZE);
        buffers[buf_idx].effective_address = *current_ea_pointer;
        /* Initiate transfer using the buffer index as the DMA tag */
        mfc_getb(buffers[buf_idx].data, buffers[buf_idx].effective_address,
                buffers[buf_idx].size, buf_idx, 0, 0);
        /* Move the data pointers */
        *remaining_data -= buffers[buf_idx].size;
        *current_ea_pointer += buffers[buf_idx].size;
}

inline void process_and_put_back(uint32 buf_idx) {
        wait_for_completion(1<<buf_idx);
        /* Perform conversion */
        convert_buffer_to_upper(buffers[buf_idx].data, buffers[buf_idx].size);
        /* Initiate the DMA transfer back using the buffer index as the DMA tag */
        mfc_putb(buffers[buf_idx].data, buffers[buf_idx].effective_address,
                buffers[buf_idx].size, buf_idx, 0, 0);
}
```

```
/* Main Code */
int main(uint64 spe_id, uint64 conversion_info_ea) {
        uint32 remaining_data;
        uint64 current_ea_pointer;

        load_conversion_info(conversion_info_ea, &current_ea_pointer, &remaining_data);

        /* Start filling buffers to prepare for loop (loop assumes both buffers have
         * data coming in) */
        initiate_transfer(0, &current_ea_pointer, &remaining_data);
        initiate_transfer(1, &current_ea_pointer, &remaining_data);

        do {
                /* Process buffer 0 */
                process_and_put_back(0);
                initiate_transfer(0, &current_ea_pointer, &remaining_data);

                /* Process buffer 1 */
                process_and_put_back(1);
                initiate_transfer(1, &current_ea_pointer, &remaining_data);
        } while(buffers[0].size != 0);

        wait_for_completion(1<<0|1<<1);
}
```

Ideja korištenja dva spremnika kod MFC programiranja naziva se programsko ulančavanje (engl. software pipelining). Vrši se podjela dijelova za obradu podataka u više funkcionalno neovisnih dijelova koji se mogu preklapati tijekom izvođenja radi maksimizacije brzine obrade. U ovom slučaju imali smo dva funkcionalno neovisna dijela:

- Čitanje iz glavnog spremnika, spremanje podataka u glavni spremnik
- obrada podataka u spremniku

koji su se preklapali kod izvođenja.

Ali možemo otići korak dalje i generalizirati ideju na više funkcionalno neovisnih dijelova i više ulančanih koraka (engl. *pipeline stages)*. Svakom koraku dodijelimo vlastiti spremnik podataka. Ovaj koncept je poznat pod nazivom multibuffering. Za SPU procesor dva koraka i dva neovisna funkcionalna dijela su u većini slučajeva najbolje rješenje jer je MFC odgovoran za čitanje i spremanje podataka i zato koraci ulančavanja se mogu izvoditi paralelno.

4. Zaključak

Ideja o Cell arhitekturi procesora koji se ponaša kao biološka stanica potječe još iz 1999 godine iz Japana. Današnja arhitektura Cell procesora djelo je tri kompanije: Sony, IBM i Toshiba. Cell predstavlja prvi proizvod iz nove familije mikroprocesora CBEA (engl. Cell Broadband Engine Architecture). Cell procesor se nalazi ugrađen u svim Playstationima 3 generacije.

Svaki Cell procesor se sastoji od jednog glavnog PPE procesora (engl. PowerPc Processing Element) i osam SPE procesora (engl. Synergistic Processing Element) koji su međusobno povezani preko glavne sabirnice EIB (engl. Element Interconnect Bus). PPE procesor je zadužen za pokretanje operativnog sustava i podjelu poslova SPE procesorima koji služe za računanje zahtjevnijih operacija. PPE je 64 bitni RISC procesor koji radi na taktu od 4 GHz i podržava PowerPc i VMX vektorski skup instrukcija. SPE procesori su RISC procesori koji podržavaju SIMD instrukcije za rad s vektorima podataka. SPE procesor sadrži MFC (engl. Memory Flow Controller) kontroler za DMA transfere podataka.

Zbog svoje specifične arhitekture Cell procesor ima malu potrošnju električne energije, veliku memorijsku propusnost i veliku računalno moć (250 GFLOPS-a).

Za prevođenje programa napisanih u C/C++ programskom jeziku koristi se gcc prevodilac za PPE procesor i spe_gcc prevodilac za SPE procesor. Optimalnim korištenjem MFC kontrolora za DMA transakcije moguće je imati 10 puta bolje performanse korištenja memorije nego kod običnim PC računala.

5. Literatura

[1] Programmer's Guide, Software Development Kit for Multicore Acceleration Version 3.0, IBM

[2] Programming Turtorial, Software Development Kit for Multicore Acceleration Version 3.0, IBM

[3] One-Day IBM Cell Programming Workshop at Georgia Tech, 6.2.2007, http://www.cc.gatech.edu/~bader/CellProgramming.html, 20.4.2009.

[4] Introduction to Cell Processor, Dr. Michael Perrone, IBM,

http://www.cag.csail.mit.edu/ps3/lectures/6.189-lecture2-cell.pdf, 15.4.2009.

[5] Programming high-performance applications on the Cell BE processor, Part 1: An introduction to Linux on the PLAYSTATION 3, Jonathan Bartlett, 3.1.2007., http://www.ibm.com/developerworks/power/library/pa-linuxps3-1/index.html?S_TACT=105AGX16&S_CMP=EDU, 10.4.2009.

[6] Programming high-performance applications on the Cell BE processor, Part 2: Program the synergistic processing elements of the Sony PLAYSTATION 3, Jonathan Bartlett, 7.2.2007., http://www.ibm.com/developerworks/power/library/pa-linuxps3-2/index.html?S_TACT=105AGX16&S_CMP=EDU, 13.4.2009

[7] Programming high-performance applications on the Cell BE processor, Part 5: Programming the SPU in C/C++, Jonathan Bartlett, 20.3.2007., http://www.ibm.com/developerworks/power/library/pa-linuxps3-5/index.html?S_TACT=105AGX16&S_CMP=EDU, 15.4.2009

[8] Programming high-performance applications on the Cell BE processor, Part 6: Smart buffer management with DMA transfers, Jonathan Bartlett, 3.4.2007, http://www.ibm.com/developerworks/power/library/pa-linuxps3-6/index.html?S_TACT=105AGX16&S_CMP=EDU, 20.4.2009

6. Sažetak

Definirani su osnovni modeli paralelnih računala (SISD,SIMD,MISD i MIMD) po odnosu programskih instrukcija i podataka i modeli zajedničke i raspodijeljene memorije. Nakon toga napravljena je podjela osnovnih paradigmi paralelnih programa (komunikacija porukama, podatkovni paralelizam, zajednička memorija i sustav kanala i zadataka). Definirana su osnovna svojstva paralelnih algoritama (istodobnost, skalabilnost, lokalnost i modularnost) kao i jedna mjera ubrzanja paralelnih algoritama poznatija kao Amdahl-ov zakon. Također su definirani pojmovi kao što su ujednačenje opterećenja, zrnatost, podatkovna ovisnost i potpuni zastoj.

U idućem poglavlju smo pobliže objasnili arhitekturu Cell procesora. Svaki Cell procesor se sastoji od jednog glavnog PPE procesora (engl. PowerPc Processing Element) i osam SPE procesora (engl. Synergistic Processing Element) koji su međusobno povezani preko glavne sabirnice EIB (engl. Element Interconnect Bus). PPE procesor je zadužen za pokretanje operativnog sustava i podjelu poslova SPE procesorima koji služe za računanje zahtjevnijih operacija. PPE je 64 bitni RISC procesor koji radi na taktu od 4 GHz i podržava PowerPc i VMX vektorski skup instrukcija. SPE procesori su RISC procesori koji podržavaju SIMD instrukcije za rad s vektorima podataka. SPE procesor sadrži MFC (engl. Memory Flow Controller) kontroler za DMA transfere podataka.

U trećem poglavlju objašnjene su osnove paralelnog programiranja na Cell procesoru. Objašnjen je poseban način prevođenja programa za PPE (gcc prevodilac) i SPE (spu_gcc prevodilac) procesor i njihovo povezivanje pomoću programa Embededspu. Pokazan je način pokretanja SPE dretvi od strane PPE procesora. Objašnjeni su osnovni koncepti programiranja SPE procesora (vektorske instrukcije, vektorske intrinzične naredbe, itd.). Pokazani su načini korištenja MFC kontrolera u SPE procesoru i objašnjen je napredni obrazac MFC programiranja s dvostrukim spremnikom.