SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4836

MASIVNA PARALELNA OBRADA

Rudolf Lovrenčić

Zagreb, lipanj 2017.

SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Masivna paralelna obrada

Završni rad br. 4836

Autor: Rudolf Lovrenčić

Mentor: izv. prof. dr. sc. Domagoj Jakobović

Zagreb, lipanj 2017.

Zahvaljujem prof.dr.sc. Domagoju Jakoboviću na prijateljskom i motivirajućem mentorstvu, puno truda u radu sa studentima i pomoći u izradi rada. Hvala dobrim prijateljima, strpljivoj obitelji i prekrasnoj Dani – uvijek ste tu kad je buka glasnija od misli.

1 UVOD

C++ AMP je razvio Microsoft s namjerom da učini programiranje grafičkih procesora (i ostalih potencijalnih akceleratora programa) jednostavnije za programera, ali istovremeno pruža i mogućnosti za podrobnu kontrolu koju zahtijevaju napredniji korisnici. Implementiran je povrh DirectX 11 programskog sučelja te je otvorenog standarda – javno je dostupan, nema propisane načine uporabe i implementacija niske razine nije definirana. Problem Microsoftove implementacije je dostupnost DirectX alata samo na Windows sustavima.

Već krajem 2013. godine, što je 2 godine kasnije od početka razvoja C++ AMP tehnologije, HSA organizacija je najavila prevoditelj čiji izlaz neće biti DirectX pozivi - cilj je bio prenosivo rješenje. Sljedeće godine, izrađen je prevoditelj koji uspješno preveo C++ AMP u OpenCL programsko sučelje, a Linux platforma je postala službeno podržana.

2 OPIS PROGRAMSKOG ALATA C++ AMP

Cilj tehnologije je pružiti potporu za univerzalno programiranje grafičkih procesora unutar jezika C++. Funkcije koje se izvršavaju na grafičkim procesorima, svaka na svojoj nezavisnoj dretvi, nazivamo *kernel* funkcije. Kernel funkcije se obično pišu na posebnim programskim platformama (npr. CUDA). Posebnost C++ AMP alata je da se kernel funkcija piše direktno unutar C++ programa, uz određena ograničenja koja će detaljnije biti razrađena kasnije.

Centralni procesor računala se u kontekstu GPGPU programiranja često naziva *host*, što se prevodi kao "domaćin" ili "matični uređaj". Pojam uređaj (engl. *device*) okuplja sve procesore raspoložive programu, bili oni CPU ili neka vrsta akceleratora programa kao što je grafička kartica.

Kako bi se omogućio rad s akceleratorima programa unutar jezika C++, potrebno je enkapsulirati sve ključne dijelove GPGPU programiranja u razrede i funkcije dostupne programeru. C++ AMP stoga nudi razrede za upravljanje memorijom i akceleratorima te stvaranje, indeksiranje i blokiranje dretvi.

2.1 Uspostavljanje programskog alata

Najlakše korištenje C++ AMP tehnologije jest uz Microsoft Visual Studio 2012 ili noviju verziju. Knjižnica dolazi s instalacijom programa te sve što je potrebno je uključiti *amp.h* u zaglavlju programa. Knjižnica dolazi sa *debuggerom* i alatom za detaljno proučavanje memorije (engl. *memory profiler*) što su velike prednosti u odnosu na ostale razvojne okoline gdje je uspostavljanje znatno kompliciranije.

Uspostavljanje na Ubuntu Linux operacijskom sustavu zahtjeva korištenje HCC prevoditelja. Prevoditelj je izrađen s namjerom da se programi iz viših jezičnih standarda kao što su: C++ AMP, OpenMP i C++ ParallelSTL, prevedu u skup instrukcija AMD GCN (*Graphics Core Next*) [3]. Prevoditelj je dostupan na službenom GitHub repozitoriju [1]. Nakon kloniranja repozitorija u lokalni, potrebno je prevesti izvorni kod prevoditelja – za ovaj korak potrebno je osnovno poznavanje CMake-a. Ako je instalacija uspješno završena, poziv HCC prevoditelja za C++ AMP izvorni kod radi se iz naredbenog retka naredbom prikazanom u nastavku.

```
hcc `clamp-config --cxxflags –ldflags` foo.cpp
```

***Isječak 1** Pokretanje HCC prevoditelja*

Iako je Ubuntu službeno podržana platforma, iskustva nekih korisnika govore da je instalacija na Linuxu prilično mukotrpna. U nekim slučajevima se zahtijevaju izmjene jezgre operacijskog sustava, pa je generalna preporuka instalacija na Windows operacijskom sustavu ukoliko korisnik nema napredno znanje Linux platforme.

2.2 Razredi

2.2.1 array

Polje sa kojim akceleratori znaju raditi. Prilikom stvaranja objekta, stvara se kopija podataka nad kojima akcelerator radi – time se podaci prebacuju na radnu memoriju akceleratora. Ne postoji podrazumijevani (engl. *default*) konstruktor za ovu klasu. Svi konstruktori koje ovaj razred podržava ograničeni su na izvođenje na procesoru, tj. ne mogu izvršavati u kernel funkciji.

Alokacija memorije preko *array* objekata ima automatiziranu kontrolu životnog ciklusa. Time dobivamo ponašanje da dokle god postoji referenca ili pogledarray_view (*array_view*) na dio memorije koji je početno rezerviran objektom *array*, legalno je pristupati tom dijelu memorije iako je došlo do brisanja početnog *array* objekta (pozvan je destruktor).

2.2.2 array_view

Pogled na cijeli objekt *array* ili na jedan njegov dio. Druga funkcionalnost koju *array_view* pruža jest pogled na podacima domaćina, tj. podataka u procesora smještenih u radnoj memoriji. Razred ne sadrži podrazumijevani konstruktor već se podaci moraju predati ukoliko se želi stvoriti *array_view* objekt.

Najčešći scenariji korištenja:

- pogled na podatke u sistemskoj memoriji koji se koristi na akceleratorima
- pogled na podatke u memoriji akceleratora koji se koristi na drugim akceleratorima
- pogled na podatke u memoriji akceleratora koji se koristi na domaćinu (CPU)

U svakom od navedenih scenarija, podaci se implicitno kopiraju na memoriju uređaja s kojega pristupamo podacima. Ukoliko dođe do promjene tih podataka na uređaju koji ima pogled na podatke, promjene se obavljaju i nad matičnim podacima. Microsoftova verzija C++ AMP tehnologije ne specificira prenosi li se cijelo polje podatak natrag ili samo promijenjeni podaci, te u velikoj većini slučajeva za programera to nije bitno.

U slučaju pogleda nad podacima u sistemskoj memoriji, programer je dužan slijediti sljedeće smjernice:

1. Mijenjati podatke samo preko pogleda.
2. Pozvati funkciju *synchronize* pripadnog pogleda nad podacima neposredno prije direktnog pristupa podacima, a ako su se temeljni podaci promijenili, prije daljnjeg pristupa podacima preko pogleda potrebno je pozvati funkciju *refresh*. Ovim pozivima se obavještava pogled da su se izvorni podaci promijenili te da je kopija na akceleratoru zastarjela, pa se mora izvršiti sinkronizacija.
3. Ukoliko se radi o pogledu nad izvornim CPU podacima ("*Plain old data*", std::vector), potrebno je osigurati da temeljni podaci imaju dulji životni ciklus od pogleda nad njima. Pristup oslobođenom dijelu sistemske memorije preko *array_view* objekata rezultira nedefiniranim ponašanjem.

Važno je napomenuti da je programer dužan ručno sinkronizirati scenarij u kojima se istim podacima pristupa preko dva različita pogleda. Primjer stvaranja objekta klase *array_view* koja sadrži cijele brojeve u tri dimenzije iz standardnog C++ polja:

```
int aCPP[] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
};
array_view<int, 3> a(2, 3, 4, aCPP);
```

***Isječak 2** Stvaranje pogleda nad podacima*

Ako ispred tipa pogleda nad podacima navedemo modifikator *const*, podaci se samo trebaju kopirati na akcelerator ukoliko se oni već tamo ne nalaze. Promjene nad podacima se ne mogu dogoditi jer se podaci sintaksno mogu samo čitati, pa pogled ne mora pratiti promjene nad podacima.

2.2.3 index

Pruža mogućnost pomaka po poljima podataka namijenjenim za akceleratore – to su *array* i *array_view.* Primjer stvaranja objekta klase *Index* za trodimenzionalno polje podataka. Indeksira se član na dubini nula, retku jedan i stupcu tri:

```
index<3> idx(0, 1, 3);
```

Isječak 3 *Stvaranje objekta klase Index*

2.2.4 extent

Specificira oblik podataka, tj. duljinu podataka u svaku dimenziju. *Array* i *Array_view* posjeduju atribut ove klase kako bi specificirali veličinu podataka.

2.2.5 accelerator

Razred `accelerator` jest apstrakcija sklopovlja optimiziranog za paralelnu obradu velike količine podataka. Grafički procesor računala je najčešće podrazumijevani akcelerator, ali razred može predstavljati i virtualni entitet procesora (npr. Microsoft DirectX REF, Windows Advanced Rasterization Platform - *WARP*).

Preko ovog razreda, C++ AMP pruža jednostavan pogled i manipulaciju raspoloživim računalnim resursima što se može vidjeti kratkim isječkom programskog kôda:

```
accelerator defaultAcc(accelerator::default_accelerator);
std::wcout << "Default accelerator:\n\t"
           << defaultAcc.description;
```

Isječak 4 *Ispis naziva podrazumijevanog akceleratora*

```
Default accelerator:
      NVIDIA GeForce GTX 1070
```

Ispis 1 *Ispis gornjeg programskog odsječka*

2.3 Zajednička memorija

Zajednička memorija (*Shared memory)* se koristi kako bi se izbjeglo kopiranje podataka na akcelerator prije obrade te kopiranje obrađenih podataka po završetku obrade. Stoga,

ako akcelerator podržava korištenje zajedničke memorije, procesor i akcelerator imaju zajednički bazen memorije kojemu ne mogu istodobno pristupati, ali se time izbjegava problem kopiranja podataka.

Korištenje zajedničke memorije se preporuča uz *array* polja kojima se zadaje tip pristupa (engl. *access_type*). Tip pristupa određuje je li objekt samo za čitanje, pisanje ili oboje. Takve informacije pomažu poboljšati performanse programa jer kernel može optimizirati pristup zajedničkim podacima.

Primjer stvaranja objekta razreda *array* sa jednom dimenzijom koja podržava deset elemenata. Objekt služi samo za čitanje:

```
array<int, 1> arr_r(10, acc_v, access_type_read);
```

***Isječak 5** Stvaranje read-only array objekta*

Array i *array_view* objekti kojima je dodijeljeno svojstvo *access_type_read* automatski se koriste u zajedničkoj memoriji ako akceleratori koji im pristupaju podržavaju zajedničku memoriju. Odluka o korištenju zajedničke memorije donosi se prilikom izvođenja programa.

2.4 Podjela memorije na blokove

Podjela bloka podataka na manje blokove može maksimizirati ubrzanje programa. Takav pristup dijeli dretve na jednako velike skupine dretvi. Efikasna upotreba ovakve podjele podataka i grupiranja dretvi se preporuča kako bi se iskoristile sve mogućnosti knjižnice C++ AMP.

Ukratko, prvo što program mora učiniti jest podijeliti podatke na manje cjeline i zatim te cjeline kopirati u posebne *tile_static* varijable koje su dostupne samo određenim dretvama.

2.4.1 tile_static

Glavno ubrzanje dobiveno grupiranjem se očekuje od pristupa varijablama tipa *tile_static*. Pristup varijablama u statičnoj memoriji grupe dretvi (*tile_static* varijable) može biti znatno brži od pristupa podacima u globalnom prostoru akceleratora (to su

klasični *array* i *array_view* objekti). Svaka grupa dretvi ima zasebnu *tile_static* varijablu i svaka dretva te grupe može pristupati toj varijabli.

Algoritmi koji dobro koriste mogućnosti grupiranja najčešće kopiraju podatke u *tile_static* memoriju samo jednom, a zatim im dretve iz pripadne grupe pristupaju mnogo puta.

2.4.2 tile_barrier::wait

Poziv funkcije *wait* zaustavlja dretvu dok sve dretve iste grupe također ne pozovu *wait.* Kako nema uređenog redoslijeda izvršavanja dretvi, ova funkcija omogućava razinu sinkronizacije tako što garantira da niti jedna dretva neće krenuti dalje od poziva *wait* funkcije prije no što taj poziv učine sve dretve iste grupe.

Ova funkcija nam otvara mogućnost da organiziramo obradu podataka na razini grupa dretvi, a ne na razini pojedinačne dretve. U programima se stoga odmah nakon inicijalizacije *tile_static* memorije poziva *wait* funkcija. Nakon toga slijedi obrada statične memorije grupe dretvi.

2.4.3 tiled_extent i tiled_index

Analogno globalnom indeksiranju, za lokalno adresiranje unutar grupe dretvi, koristi se *tiled_index.* Za opisivanje dimenzija lokalnih podataka koristi se *tiled_extent* objekt.

2.4.4 Primjer programa

Pogledajmo jednostavan primjer programa koji koristi grupiranje dretvi za obradu podataka. Program dijeli matricu sa 8 redaka i 6 stupaca na manje dijelove podataka veličine 2×3. Matrica se sastoji od elemenata sljedeće strukture:

```
struct Description {
        int value;
        int tileRow;
        int tileColumn;
        int globalRow;
        int globalColumn;
        int localRow;
        int localColumn;
};
```

***Isječak 6** Elementi matrice*

Atribut *value* jest vrijednost polja koja će ujedno biti i indeks elementa u jednodimenzionalnom polju. *tileRow* i *tileColumn* određuju mjesto segmenta memorije u globalnoj memoriji, *globalRow* i *globalColumn* određuju položaj elementa u odnosu na početak globalne memorije, a *localRow* i *localColumn* određuju položaj elementa u odnosu na početak segmenta memorije u kojem se element nalazi.

```
std::vector<Description> descs;
for (int i = 0; i < ROWS * COLS; i++) {
    Description d = {i, 0, 0, 0, 0, 0, 0};
    descs.push_back(d);
}

extent<2> matrix(ROWS, COLS);
array_view<Description, 2> descriptions(matrix, descs);

parallel_for_each(descriptions.extent.tile<2, 3>(),
 [=] (tiled_index<2, 3> t_idx) restrict(amp) {
        descriptions[t_idx].globalRow = t_idx.global[0];
        descriptions[t_idx].globalColumn = t_idx.global[1];
        descriptions[t_idx].tileRow = t_idx.tile[0];
        descriptions[t_idx].tileColumn = t_idx.tile[1];
        descriptions[t_idx].localRow = t_idx.local[0];
        descriptions[t_idx].localColumn= t_idx.local[1];
});
```

Isječak 7 *Ulazna točka za primjer podjele memorije*

Isječak 7 prikazuje dodjeljivanje vrijednosti svakom elementu te podatke o položaju (globalne i lokalne). Broj dretvi koji se stvara je isti kao i da nema segmentacije (npr. *Ispis 2* ima stvorene 24 dretve), ali se sada pruža mogućnost sinkronizacije dretvi koje rade nad istim segmentom te brži pristup elementima pripadnog segmenta - *tile_static* pristup.

Value: 0 Tile: (0,0) Global: (0,0) Local: (0,0)	Value: 1 Tile: (0,0) Global: (0,1) Local: (0,1)	Value: 2 Tile: (0,0) Global: (0,2) Local: (0,2)	Value: 3 Tile: (0,1) Global: (0,3) Local: (0,0)	Value: 4 Tile: (0,1) Global: (0,4) Local: (0,1)	Value: 5 Tile: (0,1) Global: (0,5) Local: (0,2)
Value: 6 Tile: (0,0) Global: (1,0) Local: (1,0)	Value: 7 Tile: (0,0) Global: (1,1) Local: (0,0)	Value: 8 Tile: (0,0) Global: (1,2) Local: (1,2)	Value: 9 Tile: (0,1) Global: (1,3) Local: (1,0)	Value: 10 Tile: (0,1) Global: (1,4) Local: (1,1)	Value: 11 Tile: (0,1) Global: (1,5) Local: (1,2)
Value: 12 Tile: (1,0) Global: (2,0) Local: (0,0)	Value: 13 Tile: (1,0) Global: (2,0) Local: (0,1)	Value: 14 Tile: (1,0) Global: (2,2) Local: (0,2)	Value: 15 Tile: (1,1) Global: (2,3) Local: (0,0)	Value: 16 Tile: (1,1) Global: (2,4) Local: (0,1)	Value: 17 Tile: (1,1) Global: (2,5) Local: (0,2)
Value: 18 Tile: (1,0) Global: (3,0) Local: (1,0)	Value: 19 Tile: (1,0) Global: (3,1) Local: (1,1)	Value: 20 Tile: (1,0) Global: (3,2) Local: (1,2)	Value: 21 Tile: (1,1) Global: (3,3) Local: (1,0)	Value: 22 Tile: (1,1) Global: (3,4) Local: (1,1)	Value: 23 Tile: (1,1) Global: (3,5) Local: (1,2)

Ispis 2 *Ispis programa za prikaz mogućnosti podjele memorije*

(tablica je dodana zbog lakšeg čitanja ispisa, segment memorije je obojan istom bojom)

2.5 Ulazna točka za akcelerator

2.5.1 parallel_for_each

Funkcija *parallel_for_each* predstavlja ulaznu točku za akcelerator. Funkcija očekuje *extent* objekt koji specificira dimenzije podataka te tekst programa koji će se izvršavati na akceleratorima. Tekst programa se može poslati funkciji na više načina:

- lambda funkcijom,
- objektom funkcije.

Sve dodatne funkcije koje se pozivaju unutar poslane Lambda funkcije ili funkcijskog objekta moraju biti funkcije ograničene po propisima C++ AMP knjižnice. To se ostvaruje tako da se prije otvaranja bloka funkcije piše sljedeća ključna riječ: `restrict(amp)`

Pogledajmo primjer za ubrzanje jednostavnog zbrajanja polja. *sum* jer rezultantno polje, a element na indeksu *idx* dobiva se zbrajanjem elemenata polja *a* i *b* na istim indeksima. Pomoću lambda funkcije, tekst programa izgledao bi ovako:

```
parallel_for_each(
        sum.extent,
        [=](index<1> idx) restrict(amp) {
                sum[idx] = a[idx] + b[idx];
        }
);
```

Isječak 8 *Ulazna točka korištenjem lambda funkcije*

Lambda funkcije je prikladno koristiti kada ovaj tekst programa više nećemo koristiti na drugim mjestima, no ponekad je urednije i korisnije pisati funkcijski objekt. Razred mora nadgraditi operator poziva funkcije što je prikazano u programskom isječku u nastavku.

```
class AdditionFunctionObject{
public:
     AdditionFunctionObject(const array_view<int, 1>& a,
     const array_view<int, 1>& b,
     const array_view<int, 1>& sum)
        : a(a), b(b), sum(sum)
        {}

      void operator()(index<1> idx) restrict(amp) {
         sum[idx] = a[idx] + b[idx];
      }
private:
     array_view<int, 1> a;
     array_view<int, 1> b;
     array_view<int, 1> sum;
};
```

Isječak 9 *Funkcijski razred*

Poziv funkcije *parallel_for_each* ako koristimo funkcijski objekt:

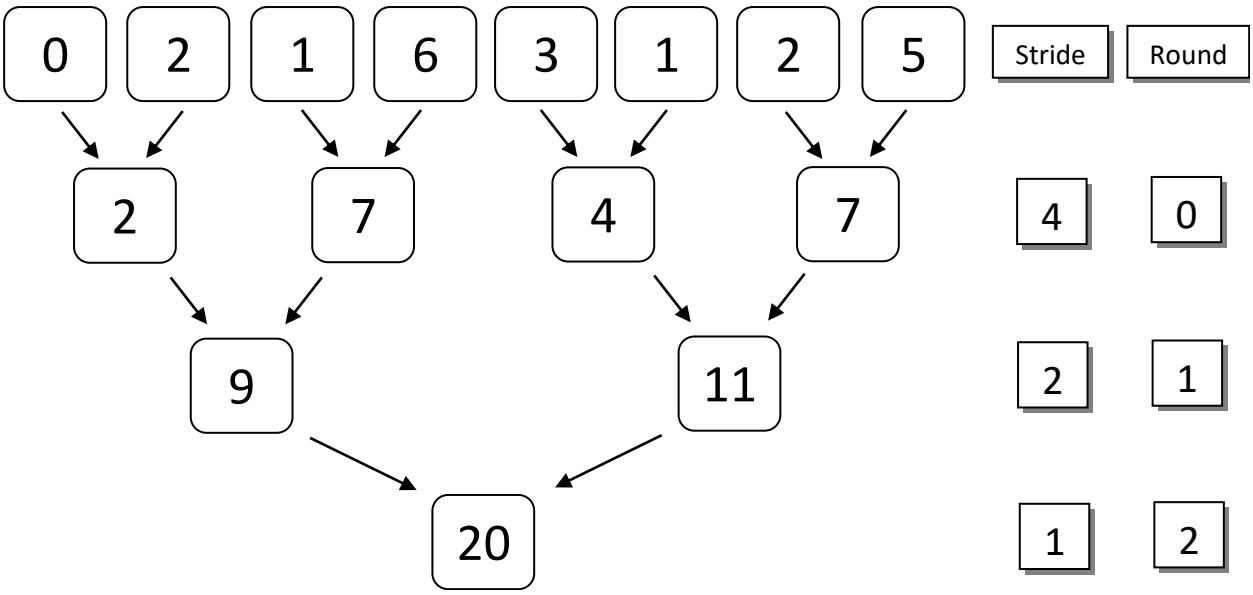
```
    parallel_for_each(
         sum.extent,
         AdditionFunctionObject(a, b, sum));
```

Isječak 10 *Ulazna točka korištenjem funkcijskog objekta*

2.6 Redukcija

Metoda redukcije je standardna metoda za obradu polja podataka u paralelnom programiranju. Koristi se kada je na temelju velike količine podataka potrebno izračunati jednu vrijednost. Naravno, kao i u većini slučajeva u paralelnoj obradi podataka, potrebno je da su podaci međusobno nezavisni. U iznimnim scenarijima dozvoljena je određena doza zavisnosti među podacima, ali to nije predmet ovog rada te će sve redukcije u nastavku biti provedene nad potpuno međusobno neovisnim elementima polja.

Redukcija kombinira sve elemente kolekcije u jedan koristeći operator sa dva ulaza i jednim izlazom. Bitno je da je željeni operator asocijativan, tj. da redoslijed operacija nije bitan. Nužnost za ovo svojstvo biti će objašnjeno nakon upoznavanja sa konceptom redukcije.



Slika 1 *Numerička redukcija*

Slika 1 prikazuje redukciju polja od osam cijelih brojeva operacijom zbrajanja. *Round* definira korak redukcije, a *Stride* određuje broj operacija koje će se izvršiti u tom koraku što je ujedno i broj dretvi koji se stvara za taj korak.

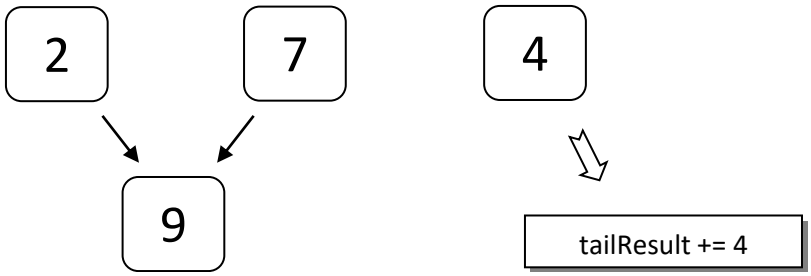
```
for (int stride = (elementCount / 2); stride > 0; stride /= 2) {
      parallel_for_each(extent<1>(stride),
      [=, &accArray](index<1> idx) restrict(amp) {
            accArray[idx] += accArray[idx + stride];

            if ((idx[0] == 0) && (stride & 0x1) && (stride != 1))
                        tailResultView[idx] += accArray[stride - 1];
      });
}
```

Isječak 11 *Realizacija redukcije zbrajanja pomoću C++ AMP tehnologije*

Implementacija redukcije zbrajanjem korištenjem C++ AMP knjižnice nije složeno. *Isječak 11* dočarava koliko je malo potrebno napisati za realizaciju redukcije. U isječku nije prikazana cijela funkcija nego samo njezin najbitniji dio – iterativni poziv *parallel_for_each* koji pri svakom pozivu stvara dvostruko manje dretvi. Generička funkcija redukcije prima standardni C++ vektor koji se kopira na akcelerator u *accArray* objekt razreda *array*. Kopija je potrebna jer redukcija mijenja kolekciju koja se reducira – element s indeksom nula poprima vrijednost rezultata operacije redukcije primijenjene na nulti i prvi element. Time se postiže da je na kraju rezultat redukcije pohranjen upravo u tom elementu s indeksom nula.

Drugi potreban objekt je *tailResultView* koji je tipa *array_view* i predstavlja pogled na lokalnu varijablu tipa *int*. Objekt služi za akumulaciju elementa koji nemaju svog para u nekom od koraka redukcije (*Slika 2*). Pojava elementa bez para javlja se kada broj elemenata polja koje se reducira nije potencija broja dva.



Slika 2 *Akumulacija elemenata bez para u koraku redukcije*

Posljednja dva retka kernel funkcije (*Isječak 11*) obavljaju funkcionalnost akumulacije elemenata bez para. Prvi uvjet *if* naredbe osigurava da samo dretva s indeksom nula obavlja akumulaciju, drugi pak provjerava postoji li u ovom koraku neparan broj elemenata. Posljednji uvjet osigurava da se akumulacija ne izvrši u posljednjem koraku redukcije - preostao je samo jedan živući element, pa da nema ovog uvjeta on bi se zbrojio sam sa sobom i krajnja vrijednost bi bila pogrešna.

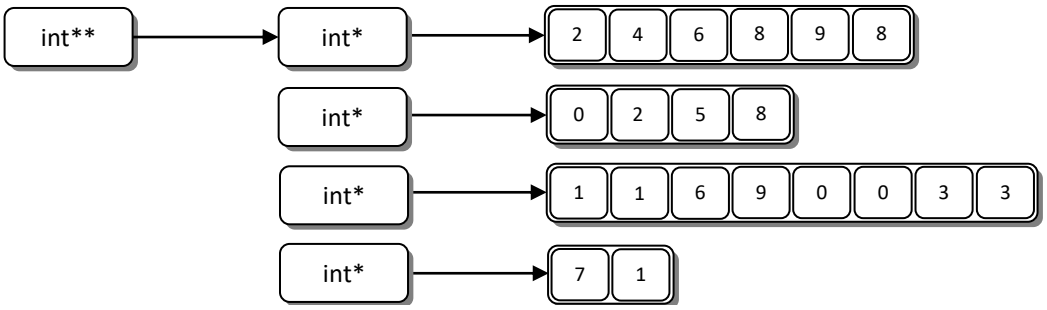
Spomenuto svojstvo asocijativnosti operatora je potrebno kako bi u svakom trenutku mogli biti sigurni da je redukcija obavljena onako kako smo željeli. Ovo nije problem ako uvijek radimo nad problemima koji imaju broj elemenata jednak nekoj potenciji broja dva. U tom slučaju imamo uređeni redoslijed primjene operatora, no u svim ostalim slučajevima ne možemo predvidjeti redoslijed primjene bez poznavanja implementacije. Razlog tome je činjenica da ukoliko imamo broj elemenata koji je jednak nekoj potenciji broja dva, u svakom koraku redukcije imamo paran broj elemenata, pa se dva najbliža elementa reduciraju u jedan, tj. svaki element ima svog para. Implementacija odlučuje kako će se riješiti problem neparnog broja elemenata u nekom koraku redukcije. Najčešći način je jednostavno reducirati element viška sa posljednja dva uspješno reducirana elementa.

2.7 Prijenos višedimenzionalnih polja

Rad sa višedimenzionalnim poljima u sklopu tehnologije C++ AMP je omogućen, ali je njihov prijenos na akcelerator znatno kompliciraniji. Slična pojava javlja se i u drugim rješenjima kao što su CUDA i OpenCL. Razlog tome jest inzistiranje na zgusnutim (engl.

dense) poljima prilikom prijenosa na akceleratore što znači da se bajtovi, redovi i stupci polja moraju nalaziti jedan iza drugog. Drugim riječima, kopiranje na akcelerator ne *slijedi* pokazivače već se kopira samo blok podataka.

Iz navedenog zaključujemo da strukturu polja prikazanu na slici ne možemo direktno preslikati na akcelerator:



Slika 3 *Nazubljeno (engl. jagged) polje*

Savjetuje se da ovakvo polje najprije *zgusnemo*, tj. pretvorimo u standardno više-dimenzijsko polje stalnih dimenzija te nakon toga stvaramo *array* ili *array_view* objekte na temelju izrađenog polja.

2.8 Problem ubrzanja vektora vektora

Sljedeći kod uzrokuje rušenje C/C++ Optimizator-Compilera:

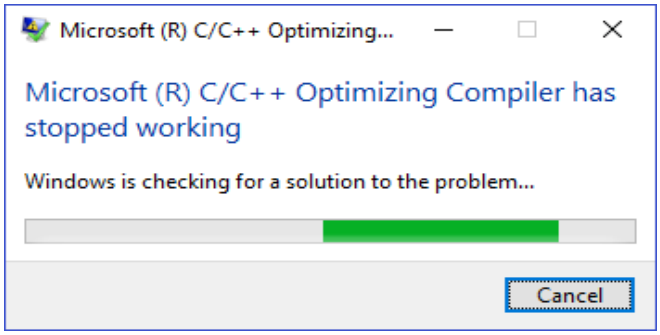
```
#include <vector>
#include <amp.h>

using namespace concurrency;

int main(void) {
      std::vector<std::vector<int>> a(5);
      array<vector<int>, 1> ampArray(a.size(), a);
      return 0;
}
```

Isječak 12 *Kreiranje concurrency::array objekta iz vektora vektora*

Naime, Visual Studio ne javlja pogreške prilikom prevođenja i povezivanja programa, već se dobiva sljedeći prozor:



Slika 4 *Greška C++ optimizacijskog prevoditelja*

Jednodimenzionalni vektori su podržani, ali sudeći po ovoj grešci, može se zaključiti da ukoliko želimo neki vektor vektora prebaciti na akcelerator, vektor se mora pretvoriti u jednodimenzionalni ili u primitivne tipove (*POD*). Službena dokumentacija govori da element podatkovnih struktura *array* i *array_view* smije sadržavati AMP-kompatibilne tipove koji nisu rekurzivni, pa ne možemo imati *array* unutar drugog *array* objekta.

Poznata restrikcija za C++ AMP je da *array* mora biti regularno polje, a ne nazubljeno (engl. *jagged*) te, kao što je već navedeno, polje mora biti zgusnuto. No, to i dalje ne objašnjava zašto se greška ova ne dojavi programeru na prihvatljiviji način od ovakve poruke operacijskog sustava.

3 OBRAĐENI PROBLEMI

C++ AMP tehnologija može biti pogodna za sve C++ programe koji se izvršavaju na sklopovlju koje ima raspoložive akceleratore, a isplati se uložiti u razvoj paralelizacije. U sklopu ovog rada, naglasak je na optimizacijskim problemima jer oni često pate od visoke vremenske složenosti, pa su sva ubrzanja dobro došla.

Obrađeni primjeri:

- ***Operation Speed Test*** – primjer konstruiran za demonstraciju mogućnosti ubrzanja
- ***GATSP*** – rješavanje problema trgovačkog putnika genetskim algoritmima[1]
- ***ECF-NeuralNetwork*** – promjena težina neuronske mreže genetskim algoritmima[1]
- ***PUF optimizer*** – "Physically unclonable function" optimizacija[1]

3.1 Operation Speed Test

Primjer prikazuje mogućnosti ubrzanja bez optimiziranja podjele memorije akceleratora (*2.4 Podjela memorije na blokove*). Program se sastoji od jedne *for* petlje u kojoj se zadana matematička operacija ponavlja određeni broj puta nad cijelim vektorom. Drugim riječima, zadani broj puta se obavlja skalarni produkt – operacija se primjenjuje nad elementima polja odgovarajućih indeksa. Operacija nad podacima se najprije izvrši slijedno na glavnoj procesorskoj jedinici računala, a zatim se podaci kopiraju u radnu memoriju akceleratora te se operacija odvija paralelno.

3.1.1 Parametri

Modifikacije rada programa zahtijevaju izmjenu izvornog koda s obzirom da program radi u "*headless*" modu – ne čitaju se parametri zadani preko komandne linije ili sa standardnog ulaza. Za jednostavnije modifikacije, potrebno je jednostavno izmijeniti nekoliko redaka datoteke *Main.cpp* jer su u njoj definirani svi bitni parametri za rad programa.

[1] *Problem je razrađen u sklopu okvira ECF [2]*

```
#define N_OF_ELEMS 1000000U
#define N_OF_REPETITION 100000U
```

Isječak 13 *Konstante koje određuju količinu posla*

Konstanta N_OF_ELEMS određuje broj elemenata u vektorima, a N_OF_REPETITION određuje broj ponavljanja "*dot*" operacije. Ukupni elementarnih operacija određen je umnoškom te dvije konstante, pa je ukupan broj operacija iz gornjeg isječka jednak 10^{11}.

```
sequentialOperationAndTime(N_OF_REPETITION, N_OF_ELEMS,
            first, second, result, Operations::SigmoidAddition);
concurrentOperationAndTime(N_OF_REPETITION, N_OF_ELEMS,
            first, second, result, Operations::SigmoidAddition);
```

Isječak 14 *Poziv funkcija za izračunavanje*

Isječak 14 prikazuje pozive funkcija koje obavljaju zadanu operaciju onoliko puta koliko je to zadano konstantama (*Isječak 13*). U pozivu gornjih funkcija koristi se operacija `SigmoidAddition`. Ova operacija primjenjuje sigmoidalnu funkciju na članove polja na pripadnim indeksima te ih potom zbraja i sprema rezultat u polje `result`. Važno je napomenuti da navedene funkcije rade izbor operacije samo jednom, a ne neposredno prije primjene operacije. Time je postignuto da dobiveni rezultati pokazuju ubrzanje isključivo matematičkih funkcija.

3.1.2 Ispis

Program ispisuje osnovne podatke o procesoru računala te informacije o raspoloživim akceleratorima.

```
System CPU:
        Intel(R) Core(TM) i5 CPU         760 @ 2.80GHz
Available accelerators:
        AMD Radeon HD 7900 Series
                Double precision: Yes
                Shared Memory: Yes
        Microsoft Basic Render Driver
                Double precision: Yes
                Shared Memory: Yes
        Software Adapter
                Double precision: Yes
                Shared Memory: No
        CPU accelerator
                Double precision: No
                Shared Memory: Yes
```

Ispis 3 *Primjer ispisa informacija o raspoloživim računalnim resursima*

Ispis 3 prikazuje da se na računalu na kojem je pokrenut program *Operation Speed Test* nalazi procesor Intel i5-760 čija je osnovna radna frekvencija 2.80GHz. Slijedi ispis raspoloživih akceleratora. Akcelerator koji se nalazi na vrhu ispisa je podrazumijevani akcelerator te će ga program koristiti za paralelno izvođenje matematičkih operacija. Podrazumijevani akcelerator sa gornjeg ispisa je grafička kartica AMD Radeon iz 7900HD serije. Nešto važniji podaci za programera su informacije o mogućnosti rada s brojevima dvostruke preciznosti i zajedničkom memorijom (*Zajednička memorija*). Naime, ukoliko se koriste funkcije iz prostora imena `amp::precise_math`, a akcelerator ne podržava operacije s brojevima dvostruke preciznosti, baca se iznimka uz odgovarajuću poruku:

```
concurrency::parallel_for_each uses features (full double_precision)
unsupported by the selected accelerator.
```

Ispis 4 *Iznimka - korištenje nepodržane preciznosti*

Navedena iznimka se baca prilikom izvođenja, pa se savjetuje da se prije slanja podataka na akcelerator provjere mogućnosti istog. Preciznost možemo provjeriti preko objekta razreda *accelerator* (*2.2.5*) i njegovog atributa `supports_double_precision`.

Nakon ispisa kontrolnih informacija, učitavaju se elementi polja iz datoteka te se kreće u izvršavanje zadane naredbe. Na posljetku, ispisuju se rezultati testiranja:

```
Matrix operation sequential time: 1257.2 seconds
Matrix operation concurrent time: 4.7462 seconds
```

Ispis 5 *Rezultat izvršavanja operacija*

3.1.3 Rezultati

Rezultati sintetičkih testova kao što je ovaj obično ne odgovaraju stvarnim situacijama, ali mogu biti korisni za određivanje reda veličine mogućnosti ubrzanja. U sklopu rada, provedena su testiranja na tri različita računala:

Tablica 1 *Specifikacije raspoloživih računala za testiranje ubrzanja*

		Računalo A	Računalo B	Računalo C
CPU	Naziv	Intel Core i3-4010U	Intel Core i5-760	Intel Core i7-6700K
	Radni takt (GHz)	1.7, Haswell	2.8, Lynnfield	4.0, Skylake
GPU	Naziv	NVIDIA GeForce 840M	AMD Radeon HD7950	NVIDIA GeForce GTX 1070
	Jezgra (MHz)	1024, Maxwell	925, Southern Islands	1797, Pascal
	Memorija (MHz)	1920, 2GB DDR3	1250, 3GB GDDR5	8108, 8GB GDDR5
	Broj jezgara	384	1792	1920

U tablici nisu navedene specifikacije radne memorija računala jer se pretpostavlja da je dovoljno velika za potrebe programa (jamči se da neće doći do straničenja), a sama frekvencija memorije ne pravi značajnu razliku u ovakvim testovima.

Količina operacija za testiranje se kreće od 10^3 do 10^{11}, a svaki test je pokrenut barem pet puta na x86 Release verziji programa. Valja napomenuti da iako su testovi pokrenuti nekoliko puta, ne možemo znatno utjecati na dodjeljivanje procesorskog vremena operacijskog sustava Windows, pa su manja rasipanja u rezultatima očekivana.

3.1.3.1 RAČUNALO A

Računalo A je prijenosnik osrednjih specifikacija iz 2014. godine. U nastavku su prikazani rezultati ubrzanja za navedeno računalo. Rezultati su prikazani na logaritamskoj skali zbog velikih razlika među rezultatima, a u podnožju grafa (*Dijagram 1*) se mogu vidjeti stvarni brojevi ubrzanja dobiveni kao količnik CPU vremena i GPU vremena potrebnih za obavljanje operacija. Ukoliko je taj broj manji od jedan, ubrzanje je neuspjelo te dodatni poslovi pripreme podataka za paralelizaciju u konačnici usporavaju program.

***Dijagram 1** Ubrzanje računala A*

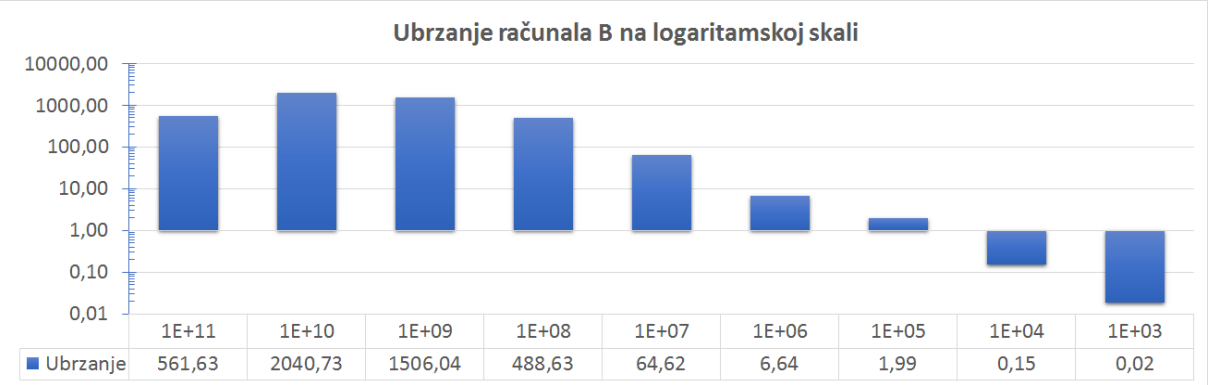
(na horizontalnoj osi se nalazi broj operacija, a vertikalna os prikazuje ubrzanje na logaritamskoj skali)

Dobiveni rezultati odgovaraju očekivanjima, ali se javljaju neke anomalije. Za 10^{11} operacija dobivamo lošiji rezultate nego za 10^{10} za čak cijeli red veličine. Pretpostavlja se da je razlog tome loša raspoloživost procesorskog vremena za vrijeme testiranja ubrzanja za 10^{10} operacija, pa je procesor ispao sporiji nego što u stvarnosti jest. Ipak, i prilikom višestrukog ponavljanja testiranja, dobiveni su slični rezultati, pa empirijski prihvaćamo da je najveće ubrzanje dobiveno upravo za obavljanje 10^{10} operacija. Analogna anomalija javlja se za 10^8 podataka, samo što su ovaj put rezultati u korist procesorske jedinice.

Valja primijetiti da za program od 10^6 jednostavnih matematičkih operacija nema smisla provoditi paralelizaciju na razini grafičkog procesora. Ubrzanje koje bi dobili jednostavnijom raspodjelom poslova na više jezgri procesora bilo bi identično ili čak veće. Za manje od 10^6 operacija, tehnologija C++ AMP daje negativne rezultate i usporava program.

3.1.3.2 RAČUNALO B

Računalo B je stolno računalo iz 2010 godine. Po specifikacijama ga možemo smjestiti u višu-srednju klasu računala svog vremena. Iako jezgra grafičke kartice računala B ima nešto manju frekvenciju od jezgre grafičke kartice računala A (*Tablica 1*), ubrzanje koje se ovdje primjećuje je višestruko veće. Razlog tome je višestruko veći broj jezgara grafičke kartice računala B što omogućava masivniju paralelizaciju koja pridonosi mnogo više u ovakvim testu nego brzina radnog takta jer se u svakom primjeru koji sadrži 10^6 ili više operacija, generira 10^6 nezavisnih dretvi (za ostale je broj dretvi jednak broju operacija). Takva količina međusobno nezavisnih poslova se brže izvršava na više raspoloživih jezgri.



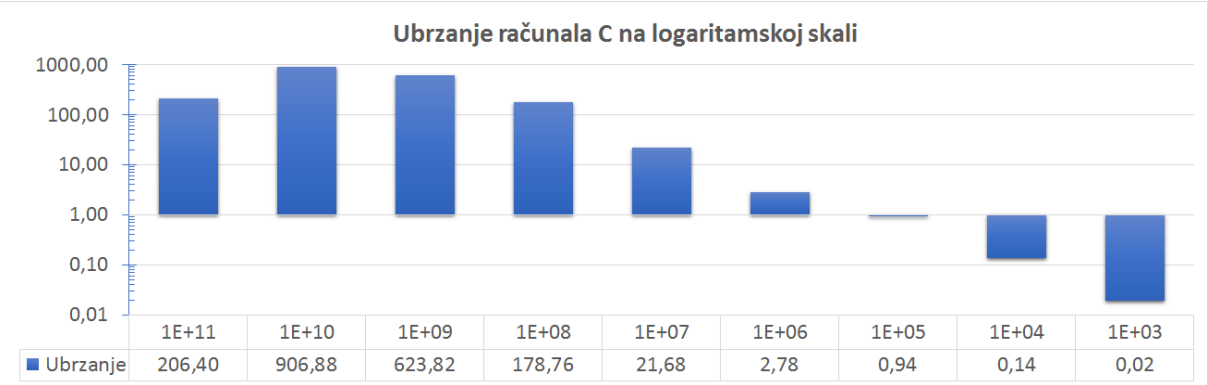
Dijagram 2 *Ubrzanje računala B*

(na horizontalnoj osi se nalazi broj operacija, a vertikalna os prikazuje ubrzanje na logaritamskoj skali)

Najveća ubrzanja se ponovno javljaju za 10^{10} operacija gdje primjećujemo ubrzanje od preko 2000 puta. Zanimljivo je da se zbog veće snage grafičke kartice negativni rezultati (usporenje) prvi put javljaju na 10^4 operacija. Na računalu A, to je bio slučaj već kod 10^5 operacija (*Dijagram 1*).

3.1.3.3 RAČUNALO C

Računalo C je osobno računalo više klase iz 2016. godine. Radni takt CPU i GPU jedinica je znatno veći od takta odgovarajućih komponenti na prethodna dva računala, ali broj jezgara grafičkog procesora veći za samo 7% od broja procesora grafičke kartice računala B. Za usporedbu, razlika u broju grafičkih procesora između računala A i B bila je 467% (*Tablica 1*).



Dijagram 3 *Ubrzanje računala C*

(na horizontalnoj osi se nalazi broj operacija, a vertikalna os prikazuje ubrzanje na logaritamskoj skali)

Rezultati su vrlo slični prethodno dobivenim rezultatima. Prvo pogoršanje performansi primjećuje se za 10^5 operacija, kao i kod računala A. Ovi rezultati potvrđuju intuiciju da je veća korist od paralelizacije na računalima gdje je razlika između procesorske snage i

snage akceleratora veća. U provedenim testiranjima, najbolje rezultate daje računalo B jer je tamo omjer snage grafičke kartice i procesora najveći.

3.2 GATSP

GATSP predstavlja rješavanje problema trgovačkog putnika genetskim algoritmom. Primjer je realiziran u sklopu okvira ECF, a sva paralelizacija se događa u evaluacijskom operatoru. Paralelizira se *for* petlja koja zbraja udaljenosti između gradova:

```
for (unsigned int i = 0; i <  permVariablesSize - 1; i++) {
      fitnessV +=
            weights[perm->variables[i]][perm->variables[i+ 1]];
}
```

Isječak 15 *Slijedno računanje dobrote*

Postupak paralelizacije je zahtjeva sljedeće preinake u tekstu programa:

1. Vektor težina *weights* mora se prebaciti u jednodimenzionalni vektor kako bi se nad njim izgradio pogled koji akcelerator može koristiti.

2. Stvaranje pogleda na *perm->variables* vektor kako bi se mogao koristiti na akceleratoru:

```
array_view<const int, 1> AMP_perm_variables(perm->variables);
```

3. Stvaranje objekta *array* veličine *permVariablesSize* kako bi svaka dretva računala svoj dio vrijednosti dobrote (*fitnessV*):

```
vector<int> fitnessVec(permVariablesSize, 0);
concurrency::array<int, 1> AMP_fitnessVec(
     permVariablesSize,
     fitnessVec.begin(),
     fitnessVec.end() );
```

4. Zamjena početne *for* petlje pozivom funkcije *parallel_for_each*:

```
concurrency::parallel_for_each(nOfThreads,
     [=, &AMP_fitnessVec](concurrency::index<1> idx) restrict(amp)
{
      AMP_fitnessVec[idx] += AMP_weights[AMP_perm_variables[idx]
                                 * weightsRowSize
                                 + AMP_perm_variables[idx + 1]];
});
```

5. Redukcija *array* objekta stvorenog u prethodnom koraku kako bi se dobila konačna vrijednost dobrote (*2.6. Redukcija*).

Iz navedenih koraka se može zaključiti da preinake znatno povećavaju količinu teksta programa, a većina tog teksta je zapravo najavljivanje i pretvorba podataka za akcelerator. Dodatnim tekstom, smanjuje se čitljivost programa, no to je za očekivati s obzirom da se programira na razini kernel funkcija.

3.2.1 Rezultati

Ubrzanje se vrši u evaluacijskom operatoru u kojem se iterira po broju gradova. *for* petlja se zamjenjuje *parallel_for_each* programskim odsječkom i time se dobiva onoliko neovisnih dretvi koliko ima gradova u danom TSP problemu. Broj gradova se kreće od nekoliko desetaka do nekoliko tisuća, pa čak i desetaka tisuća. Ipak, taj broj je još uvijek relativno malen u odnosu na broj iteracija iz *Operation Speed Test* problema te ne bi bilo pogrešno očekivati marginalna poboljšanja ili čak pogoršanja performansi programa.

Mjerenje vremena je obavljeno barem pet puta za svaki primjer na računalu B (*Tablica 1* sadrži specifikacije računala) i to za tri različita pristupa.

***Tablica 2** Vrijeme u sekundama potrebno za rješavanje TSP-a*

Broj gradova	**51** (100 generacija)	**225** (100 generacija)	**1084** (10 generacija)
Slijedno	17	242	466
Paralelno A	62	302	634
Paralelno B	50	277	598
Paralelno C	45	262	516

Paralelno A jest prvi i najlošiji pokušaj paralelizacije. Pretvaranje vektora vektora u jednodimenzionalno polje radi se svaki put u *evaluate* funkciji. Navedena funkcija je ključna funkcija svakog evaluacijskog operatora unutar okvira ECF. Poziva se nakon svake promjene genotipa kako bi se ocijenila dobrota rješenja, tj. stanja genotipa. Evaluacijska funkcija se poziva mnogo puta, pa se puno vremena troši na pretvorbu u jednodimenzionalno polje koje se može učiniti samo jednom jer pretvorba ne ovisi o stanju genotipa.

Paralelno B predstavlja poboljšanje prethodnog pokušaja. Pretvaranje u jednodimenzionalno polje izlučeno je u funkciju *initialize* koja se obavlja samo jednom. Poboljšanje u radu programa je očigledno, no još uvijek je program znatno sporiji od svoje slijedne inačice.

Zadnje poboljšanje koje je moguće napraviti na razini paralelizacije po gradovima jest konstrukcija specijalizirane redukcije. Redukcija opisana u poglavlju 2.6. prima standardni C++ vektor kao parametar, pa objekt *array* koji se puni elementima izračunatima u *parallel_for_each* funkciji trebamo najprije pretvoriti u standardni vektor da bi ga dosad definirana funkcija redukcije ponovno slala na akcelerator (kreirala objekt *array* nad njime), obavila redukciju i vratila vrijednost. Specijalizirana redukcija prima objekt *array* koji se već nalazi na akceleratoru, reducira ga i vraća vrijednost. Na taj način su spriječena nepotrebna kopiranja između sistemske memorije i memorije akceleratora.

Usprkos specijaliziranoj redukciji, *Tablica 2* pokazuje da ni pristup "Paralelno C" ne daje zadovoljavajuće rezultate već drastično usporava rad programa. Negativni učinak je umanjen za probleme s više gradova, ali TSP problemi na kojima bi se moglo vidjeti ubrzanje su teški za pronaći i toliko su veliki da bi slijedno testiranje trajalo tjednima.

Rezultati nisu iznenađujući s obzirom da je na *Operation Speed Test* problemu pokazano da moramo imati dovoljno veliku količinu nezavisnih podataka za mogućnost ubrzanja. Na navedenom primjeru, potrebno je imati barem 10^5 operacija da bi se došlo do marginalnih ubrzanja (*Dijagram 2*).

3.3 PUF optimizer

Funkcije bez mogućnosti kopiranja (engl. p*hysically unclonable functions, PUF*) su funkcije koje se ne mogu reproducirati jer koriste fizička svojstva sklopovlja kojim su realizirane kako bi osigurale svoju jedinstvenost (npr. parazitski kapacitet). Program *PUF optimizer* je simulator modela takvih funkcija ostvaren unutar okruženja ECF s ciljem korištenja evolucijskih algoritama za optimizaciju istih [7]. Tijekom optimizacije, algoritam mijenja parametre funkcije u cilju pronalaska najboljeg rješenja. Dobrota rješenja se prilikom svake promjene parametara ocjenjuje pozivom funkcije *evaluate* evaluacijskog operatora okvira ECF.

Paralelizacija se odvija u evaluacijskom operatoru, jer se tamo troši najveća količina vremena na neovisno računanje. Operator je realiziran s tri savršeno ugniježđene *for* petlje – prva naredba svake *for* petlje, osim posljednje, je petlja (najavu varijabli možemo zanemariti):

```
unsigned int error = 0;
for (unsigned int resp = 0; resp < responseSize; resp++) {
      int myResponse = 0;
      for (unsigned int chain = 0; chain < nPUFs; chain++) {
            double delay = 0,
            for (unsigned int stage = 0; stage < nStages; stage++) {
                        delay += feature[resp * nStages + stage]
                              * gen->realValue[chain * nDelays +
                                                      stage];
            }
            delay += 1 * gen->realValue[chain * nDelays + nStages];
            if (delay < 0)
                  myResponse ^= 1;
            else
                  myResponse ^= 0;
      }
      if (myResponse == 0)
            myResponse = -1;
      if (myResponse != response[resp])
            error++;
}
```

Isječak 16 *Slijedni kod programa PUF optimizer*

Cilj dijela programa kojeg prikazuje *Isječak 16* je izračunati vrijednost varijable *error*. Iz isječka se može zaključiti da su iteracije vanjske *for* petlje međusobno nezavisne, a priroda problema je takva da će ta petlja raditi najviše iteracija, pa je ona idealna za paralelizaciju. Traženu vrijednost ćemo izračunati na način da svaka dretva računa svoj dio *error* vektora te nakon toga taj vektor reduciramo u jednu vrijednost (*2.6. Redukcija*).

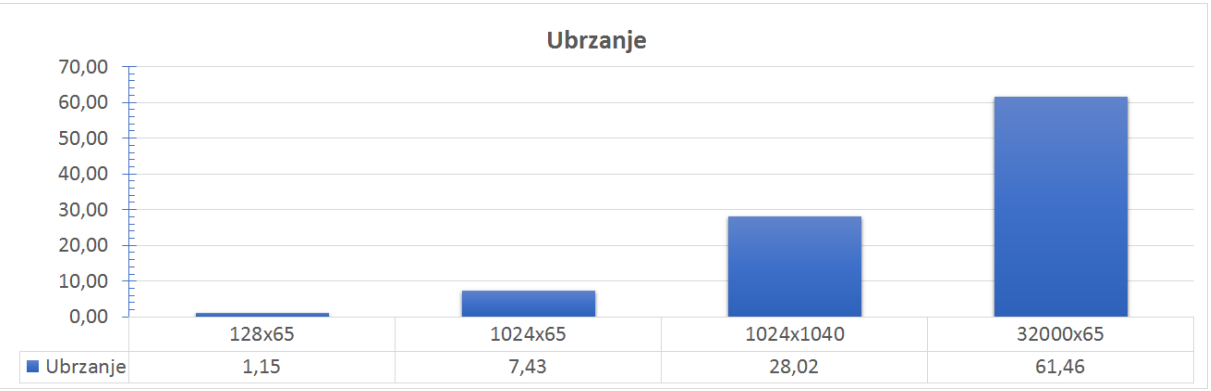
Vektor značajki (engl. *feature*) i vrijednosti genotipa treba omotati *array_view* objektom kako bi bili dostupni na akceleratoru. Dodatno, *nStages* i *nDelays* je potrebno spremiti u lokalne varijable jer se u *restrict(amp)* kontekstu ne mogu nalaziti privatni atributi klase. Broj dretvi jednak je vrijednosti varijable *responseSize* jer ona određuje broj okretaja vanjske *for* petlje. Na temelju nje se stvara *extent* objekt i poziva *parallel_for_each*.

```
concurrency::parallel_for_each(concurrency::extent<1>(responseSize),
      [=, &AMP_errorVec](Concurrency::index<1> idx) restrict(amp) {
            int myResponse = 0;
            for (uint chain = 0; chain < numberOfPUFs; chain++) {
                  double delay = 0;
                  for (uint stage = 0; stage < numberOfStages; stage++) {
                        delay += AMP_feature[idx * numberOfStages + stage]
                              * AMP_genotypeRealValue[chain *
                                          numberOfDelays +  stage];
                  }
                  delay += 1 * AMP_genotypeRealValue[chain *
                                    numberOfDelays + numberOfStages];
                  if (delay < 0)
                        myResponse ^= 1;
                  else
                        myResponse ^= 0;
            }
            if (myResponse == 0)
                  myResponse = -1;
            if (myResponse != AMP_response[idx])
                  AMP_errorVec[idx]++;
      });
```

Isječak 17 *Paraleliziran evaluacijski operator programa PUF optimizer*

3.3.1 Rezultati

U ovom realnom primjeru rasipanje je manje nego u *Operation Speed Test* primjeru, pa će broj podataka rasti u smjeru x-osi, a skala neće biti logaritamska već standardna. Prikazani rezultati su za testiranje na računalu B (*Tablica 1* sadrži specifikacije računala), a za ostala računala rezultati nisu prikazani jer su analogni – nema razlika do na red veličine.



Dijagram 4 *Postignuto ubrzanje u ovisnosti o veličini podataka*

(Broj A u zapisu AxB prikazuje broj okretaja vanjske *for* petlje (ujedno i broj dretvi kod paralelizacije). Broj B prikazuje umnožak okretaja unutarnjih *for* petlji što efektivno daje broj operacija koje svaka dretva izvršava)

Skladno očekivanjima, *Dijagram 4* prikazuje da se najveća dobit ostvaruje na najvećoj količini podataka. Naime, to je slučaj jer se za posljednji primjer stvara 32000 međusobno

neovisnih dretvi, pa valja naglasiti da je skalabilnost ovog programa iznimno dobra te bi izvođenje u sustavu s više akceleratora dovelo do još boljih rezultata.

3.4 GP classifier

Sljedeći razmatrani problem je binarni klasifikator genetičkim računanjem. Program na temelju izračunate dobrote jedinke odlučuje hoće li joj dodijeliti nulu ili jedinicu i time obavlja klasifikaciju podataka. Dobrota se izračunava iz vrijednosti značajki svake jedinke. U realnim problemima ima mnogo značajki te se često kod klasifikacije pretpostavlja da su one međusobno nezavisne, pa se ovdje očekuje mogućnost ubrzanja.

Većina posla se obavlja upravo u *evaluate* funkciji gdje se petlje gnijezde na dvije razine (kratko objašnjenje *evaluate* funkcije nalazi se u poglavlju 3.3.1). Priroda problema pokazuje da bi dobici paralelizacije bili najveći za paralelizaciju na vanjskoj petlji, a da bi paralelizacija po unutarnjim petljama vrlo vjerojatno rezultirala sporijim radom programa. Problem koji je odmah uočen jest da velika količina teksta programa unutar vanjske for petlje ne obavlja neke jednostavne matematičke funkcije već poziva funkcije definirane u programskom okviru ECF. Sve funkcije koje se pozivaju unutar kernel funkcije moraju biti ograničene ključnom riječju *restrict(amp)*, što predstavlja problem za pozive funkcija jer to povlači izmjenu funkcija okvira – pretvorbu svih struktura podataka koje se referenciraju u *array* ili *array_view* objekte i promjenu poziva standardnih matematičkih funkcija na funkcije iz *concurrency::fast_math* prostora imena. Program koristi i složene strukture podataka kao što su višedimenzionalni vektori što znatno otežava rad jer mogu biti nazubljene strukture, pa je potrebna dodatna operacija pretvorbe u pravilna polja što ponovno zahtjeva veću opreznost prilikom indeksiranja kako ne bi došlo do indeksiranja praznog prostora.

Iz razloga navedenih u prošlom odlomku, odlučeno je da se program neće paralelizirati jer zahtjeva previše izmjena radnog okvira te je dobitak na performansama upitan zbog previše pretvorbi struktura podataka. Primjer pokazuje da nisu svi programi dobri za paralelizaciju ovom tehnologijom, osobito programi složene strukture poziva funkcija i velike količine naprednih struktura podataka koji nisu u počecima bili osmišljeni za paralelizaciju.

3.5 ECF-NeuralNetwork

Primjer spaja područje neuro-računarstva i genetskog programiranja. Neuronska mreža u sklopu programskog okvira ECF ostvarena je na način da se težine veza između neurona mijenjaju genetskim algoritmima [6]. Dobro je poznato da se sustavi koji koriste neuronske mreže mogu višestruko ubrzati izvođenjem programa na grafičkim karticama, pa se od ovog primjera očekuju pozitivni rezultati.

Konfiguracija programa se obavlja pomoću XML datoteke. Genotip *NeuralNetwork* zahtjeva da se definira struktura neuronske mreže koju čine broj neurona po slojevima te aktivacijske funkcije po slojevima. Za paralelizaciju je podržana sigmoidalna aktivacijska funkcija na svim slojevima osim na ulaznom sloju – tamo se koristi linearna funkcija kako bi se ulazi jednostavno propagirali na prvi skriveni sloj.

Većina posla obavlja se u *NeuralNetwork::getTotalError* funkciji gdje se izračunava pogreška izlaza neuronske mreže u odnosu na očekivani izlaz primjera za učenje. Paralelizacija se odvija na razini primjera za učenje.

```
-4.96637 -2.06261
-4.86637 -1.90187
-4.76637 -1.77074
-4.66637 -1.66955
-4.56637 -1.5983
-4.46637 -1.5567
-4.36637 -1.54418
```

Isječak 18 *Dio datoteke za učenje funkcije* $x + 3sin(x)$

(Prvi broj predstavlja ulaz u mrežu, a drugi očekivani izlaz)

Paralelizacija mreže nad podacima koje prokazuje *Isječak 18* odvijala bi se u sedam nezavisnih dretvi jer je dano sedam različitih primjera za učenje. Stvarna datoteka sadrži mnogo više primjera, pa paralelizacija ima više smisla nego na ovom primjeru. Kako bi mreža mogla naučiti zadanu funkciju, ona mora imati jedan ulazni neuron i jedan izlazni te proizvoljno mnogo sakrivenih slojeva od kojih svaki ima proizvoljan broj neurona.

```
for (unsigned int i = 0; i < inputData_.size(); i++) {
      result = outputData_[i];
      setCurrentInput(inputData_[i]);
      output = calculateOutput();
    // Izračunavanje greške na temelju output vektora.
}
```

Isječak 19 *Slijedno izračunavanje greške*

Kako bi paralelizacija C++ AMP tehnologijom bila moguća na razini primjera za učenje, potrebno je pretvoriti sve strukture podataka koje se koriste unutar petlje koju prikazuje *Isječak 19* u *array* ili *array_view* objekte kako bi podaci bili dostupni na akceleratorima. Iz isječka se također može vidjeti da funkcija za računanje izlaza mreže (*calculateOutput*) koristi atribute genotipa *NeuralNetwork* za izračunavanje izlaza što također predstavlja problem jer se atributi razreda ne mogu koristiti u *restrict(amp)* kontekstu. Svi atributi koji se koriste za izračunavanje greške moraju postati lokalne varijable ili barem pogledi na njih moraju postojati unutar lokalnog dosega. Pretvorba vektora u jednodimenzionalne strukture će biti obavljena u *initialize* funkciji kako bi se pretvorba izvršila samo jednom, a ne prilikom svakog računanja greške.

Problem ubrzanja neuronske mreže je znatno kompliciraniji od prethodno obrađenih problema zbog količine složenih struktura podataka koje se koriste u izračunu. Indeksiranje tih struktura podatak postaje komplicirano zbog neizbježnog nadopunjavanja nepravilnih struktura kako bi tvorile pravilno polje. Primjer koji to dočarava jest najava zajedničke memorije za sve dretve koje one koriste u izračunu izlaza mreže.

```
array<double, 1> allThreadsMemoryPool(inputData_.size() *
                                                largestLayerSize);
```

Isječak 20 *Alokacija memorije za sve dretve*

Alocirana memorija je dimenzionirana tako da svaka dretva ima dovoljno prostora za izračun izlaza sloja s najvećim brojem neurona. *Isječak 20* prikazuje tu činjenicu jer je broj stvorenih dretvi upravo *inputData_.size()*, a *largestLayerSize* je broj neurona unutar najvećeg sloja mreže.

```
array_view<double, 1> thisThreadsMemoryPool =
      allThreadsMemoryPool.section(
            index<1>(thread_index * largestLayerSize),
            extent<1>(largestLayerSize)
      );
```

Isječak 21 *Dretva stvara pogled na svoj dio alocirane memorije*

Svaka dretva koristi svoj dio memorije alocirane na akceleratoru. Prilikom instanciranja, dretva uzima svoj dio memorije na način da stvori pogled na dio ukupne alocirane memorije (*Isječak 21*). Prikazane komplikacije s memorijom su jedan od

primjera složenosti paralelizacije programa. Ostali problemi pripreme podataka za paralelizaciju su nešto kompliciraniji, ali su ideje analogne, pa ih ovaj rad neće detaljnije prikazivati.

Tekst programa zbog navedenih prepravki postaje znatno nečitljiviji. Na primjer, funkcija *calculateOutput* koja se izvršava na akceleratoru prima jedanaest argumenata, pa dokumentiranje C++ AMP programa postaje iznimno važno. Broj lokalnih varijabli funkcija brzo raste, a podrška otkrivanja pogreški za GPGPU programe unutar razvojnog okruženja Visual Studio postoji, ali zahtjeva složeno uspostavljanje i napredno znanje programiranja kako bi se njime služilo. Navedene činjenice dovode do sporijeg razvoja programa, no za višestruka ubrzanja izvođenja na programeru je da odluči je li isplativo uložiti vrijeme u paralelizaciju.

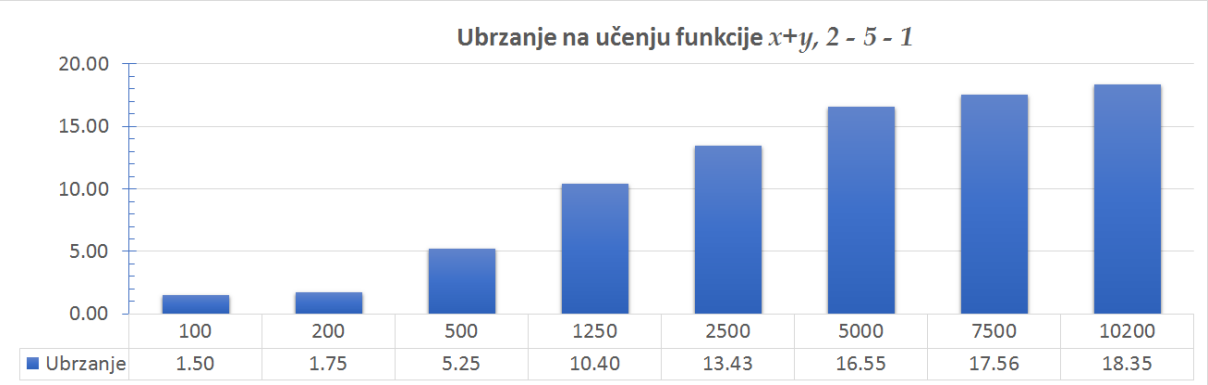
3.5.1 Rezultati

Rezultati su prikupljeni na *debug* konfiguracijama programa, no rezultati ne bi trebali značajnije odstupati od *release* konfiguracije jer se i slijedni i paralelni program pokreću u istim konfiguracijama. Ipak, valja imati na umu da su u *debug* konfiguraciji isključene optimizacije C++ prevoditelja te je moguće da jedna od inačica programa gubi više performansi od druge. Cilj testiranja je dati red veličine ubrzanja, a ne točne brojke, pa nas konfiguracija programa u ovakvim testovima neće smetati.

Testiranje je provedeno na više različitih funkcija: $x+y$, $x+3\sin(x)$, x^2+y^2, $\mathrm{tg}(x)$ i druge. Funkcije su odabrane jer pokrivaju različite strukture mreža. Neke funkcije zahtijevaju mrežu koja ima dva neurona u ulaznom sloju, a druge samo jedan jer su funkcija jedne varijable. Sva testiranja su provedena na računalu B čije specifikacije prikazuje *Tablica 1*.

3.5.1.1 FUNKCIJA $X+Y$

Neuronska mreža u sklopu okvira ECF može vrlo dobro naučiti ovu funkciju što je i očekivano s obzirom da je ovo vrlo jednostavna funkcija dvije varijable. Za učenje se upotrebljava mreža sa strukturom 2 - 5 - 1 što ukupno daje 16 težina koje genetski algoritam nastoji optimizirati. Najveće ubrzanje dobiveno paralelizacijom očekujemo za najveći broj primjera za učenje jer je paralelizacija rađena upravo nad petljom koja obilazi primjere za učenje.



Dijagram 5 *Ubrzanje na učenju funkcije $x+y$. Struktura mreže: 2 - 5 - 1*

(Na horizontalnoj osi se nalazi broj primjera za učenje, a vertikalna os prikazuje ubrzanje)

Dijagram 5 prikazuje rezultate skladne očekivanjima - ubrzanje raste kako raste broj primjera za učenje. Na boljim grafičkim procesorima očekuju se ubrzanja i do 100 puta na učenju neuronske mreže ukoliko je dan dovoljno velik skup primjera (barem 10000).

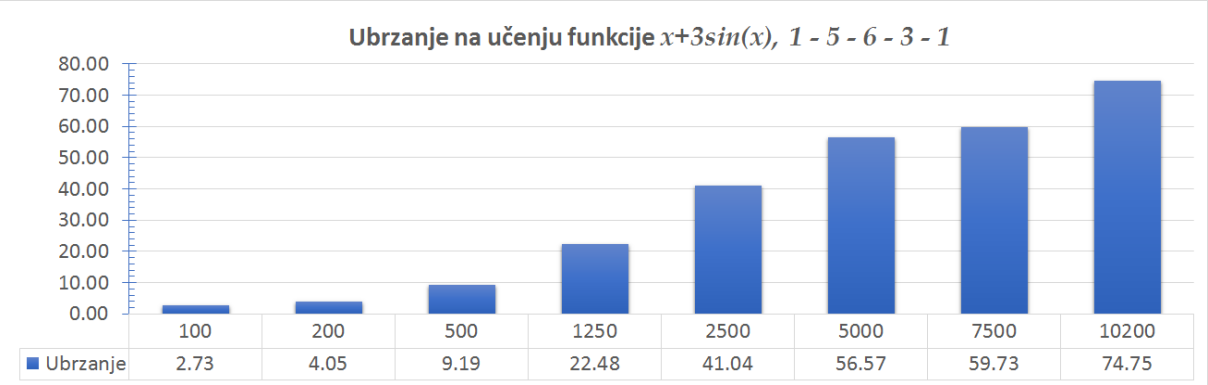
Računanje ukupne greške iz izračunatih izlaza mreže podešeno je tako da se izvršava na akceleratoru jedino ako je broj testnih primjera strogo veći od 100. Time su dobiveni pozitivni rezultati i za 100 testnih primjera. Na tako malom skupu za učenje dobivaju se usporenja rada programa zbog trošenja vremena na pripremu podataka za akcelerator i slanja podataka na akcelerator, pa su potrebne *mini-optimizacije* poput ove kako bi se zadržali pozitivni rezultati.

3.5.1.2 OSTALE FUNKCIJE NA SLIČNOJ STRUKTURI

Ostala testiranja ubrzanja učenja matematičkih funkcija na ovoj strukturi mreže daju vrlo slične rezultate kao i *Funkcija* $x+y$, pa neće biti posebno navedena. Ubrzanje raste porastom testnih primjera, a veća dobit se primjećuje na primjerima koji koriste složenije matematičke funkcije (npr. trigonometrijske funkcije). Razlog tome je veća količina posla koja dretva može obaviti umjesto da se ta povećana količina mora obavljati slijedno na procesoru.

3.5.1.3 FUNKCIJA X+3SIN(X)

U prethodnom poglavlju je zaključeno da matematička složenost funkcije koja se uči ima malo utjecaja na brzinu izvođenja, pa su za ovo testiranje znatno prošireni kapaciteti mreže u odnosu na mrežu iz poglavlja *3.5.1.1*. Mreži su dodana dva skrivena sloja pa je konačna struktura 1 - 5 - 6 - 3 - 1 što daje 71 težinu za optimizaciju.



***Dijagram 6** Ubrzanje na učenju funkcije $x+3sin(x)$. Struktura mreže: 1 - 5 - 6 - 3 - 1*

Ubrzanje na ovom primjeru poraslo je dva do četiri puta u odnosu na jednostavnu mrežu ispitanu ranije. Rezultati su očekivani jer je kapacitet mreže oko četiri puta veći što povlači činjenicu da svaka dretva ima četiri puta više posla. Procesor se već na ovakvim mrežama znatno rasterećuje, a na većim mrežama iz područja dubokog učenja rezultati bi trebali biti i znatno bolji.

Tehnologija se pokazala konzistentna i na problemu ubrzanja neuronskih mreža. Rezultati su sukladni predviđanjima što programeru daje dozu povjerenja u alat jer se ubrzo može steći osjećaj za ubrzanje koje se može očekivati na danom problemu.

4 ZAKLJUČAK

Dobiveni rezultati ukazuju na optimističnu budućnost C++ AMP tehnologije. Jednostavniji programi se brzo mogu pretvoriti u učinkovitije inačice bez iskakanja izvan okvira C++ programskog jezika. Prednost tehnologije jest objektna apstrakcija pojmova vezanih uz paralelizaciju kao što su akceleratori, polja podataka i pogled na podatke na akceleratoru, doseg i veličina objekata na akceleratoru itd. Granularna kontrola nad podacima je dostupna, ali je prema nekima izvorima znatno otežana u odnosu na niže tehnologije poput OpenCL. Dodatna prednost jest prenosivost programa pisanih C++ AMP tehnologijom, no ukoliko se program piše na Microsoftovoj distribuciji tehnologije potrebno je paziti na manja odstupanja od standarda.

Loše strane tehnologije su manjak kontrole nad različitim tipovima memorije akceleratora što je posljedica inzistiranju na jednostavnosti. Loša kontrola podijele memorije (engl. *tiling*) je jedan od glavnih razloga zašto OpenCL prednjači pred C++ AMP tehnologijom u većini GPGPU problema. Iako je dokumentacija dostupna na službenim stranicama Microsofta, podrška je znatno oslabjela jer se više od godinu dana nije ništa ažuriralo niti objavljivalo vezano uz tehnologiju. Ade Miller, jedan od najaktivnijih članova C++ AMP zajednice programera i autor knjige "*C++ AMP*", je izjavo da se je entuzijazam splasnuo s obzirom da Microsoft daje slabu podršku te da nema daljnjih planova raditi i unaprjeđivati C++ AMP [5]. Za AMD se pretpostavlja da još koristi tehnologiju iako se na službenim stranicama tvrtke može vidjeti da ja zadnja objava koja sadrži riječi "C++ AMP" iz druge polovice 2014. godine – upravo u vrijeme kada je aktualna serija grafičkih kartica bila HD7000, a u ovom radu je testirana pripadnica te serije (HD7950) koja je pokazala odlične rezultate.

Činjenice upućuju da C++ AMP neće biti korišten u mnogo velikih projekata, no smatram da tehnologija može pronaći svoje mjesto u problemima srednje i manje veličine gdje ubrzanja doprinose korisničkom iskustvu ili pak je priroda problema takva da obrada podataka dugo traje, a ne želi se ulagati mnogo resursa u paralelizaciju programa. Neuronske mreže i genetsko programiranje dolaze iz *mekog* računarstva, te se za njih već zna da mogu imati velike koristi od paralelizacije. C++ AMP se može primijeniti na takve probleme znatno lakše nego alternative kao OpenCL, iako ne pruža mogućnost da

istisnemo svaku kap performansi iz paralelnih algoritama. Naglasak je na jednostavnosti koja bi trebala programeru omogućiti da razmišlja vrlo slično klasičnom programiranju u okvirima C++ jezika. Nadalje, pojednostavljena struktura i sintaksa GPGPU programiranja omogućava da se C++ AMP uči prije naprednijih tehnologija za paralelizaciju.

Sintetički testovi pokazuju da na računalima koja raspolažu naprednijim akceleratorima možemo vidjeti ubrzanja i preko 2000 puta. Na manjim primjerima i nešto standardnijim problemima u programiranju, očekivana ubrzanja su između 50 i 500 puta što pokazuje računalnu snagu koja ostaje neiskorištena pisanjem isključivo slijednih programa. Je li uopće moguće paralelizirati program ovisi o problemu, ali dobiveni rezultati i jednostavnost korištenja bi trebali potaknuti na korištenje C++ AMP tehnologije.

5 IZVORI

[1] AMD, 2014., GitHub - HCC:
https://github.com/RadeonOpenCompute/hcc

[2] Jakobović, D. -*Evolutionary Computation Framework*:
http://ecf.zemris.fer.hr/

[3] Wikipedia - Graphics Core Next. (2012):
https://en.wikipedia.org/wiki/Graphics_Core_Next

[4] C++ AMP: Language and Programming Model, Version 1.0, (2012)

[5] Stack Overflow: "*What is the current status of C++ AMP?* ":
https://stackoverflow.com/questions/34969287/what-is-the-current-status-of-c-amp

[6] ECF-NeuralNetwork, (2017):
https://ecf-neural.herokuapp.com/

[7] Burda, M. – Modeliranje logičkih funkcija bez mogućnosti kopiranja evolucijskim algoritmima, Završni rad, Fakultet elektrotehnike i računarstva, (2017)

Masivna paralelna obrada

Sažetak

Ubrzanje programa pomoću grafičkih procesora je u stalnom usponu proteklih desetak godina. Snaga grafičkih kartica primjenjuje se na raznolike probleme iz računarske prakse, a ne samo za iscrtavanje računalne grafike. Za potrebe rada, ostvaren je idealizirani program za paralelizaciju kako bi se pokazala mogućnost ubrzanja programskim alatom C++ AMP. Obrađeno je nekoliko dodatnih problema iz područja genetskog programiranja i neuro-računarstva. Testovi su odrađeni i na različitim računalima u cilju pronalaska ovisnosti između ubrzanja i raspoloživih računalnih resursa. Rezultati su prikazani na dijagramima i tablicama u kojima se vidi ovisnost ubrzanja o količini operacija.

Ključne riječi: masivna paralelna obrada, programiranje grafičkih procesora, paralelno programiranje, C++ AMP, ovisnost ubrzanja i računalnih resursa, ubrzanje neuronskih mreža i genetskih algoritama

Accelerated massive parallelism

Abstract

Field of program acceleration using graphics processing units has seen constant growth over the past ten years. Computing power of GPUs is used for general-purpose problem-solving, instead of using their resources exclusively on rendering computer graphics. An ideal program for parallelization has been implemented for the purpose of this paper in order to show possible gains using the C++ AMP technology. Few other programs from fields of genetic programming and neurocomputing have been accelerated as well. Tests have been done on different computers in order to find connection between acceleration and available hardware. Results are displayed on graphs and in tables which show connection between number of operations and gained acceleration.

Keywords: Accelerated massive parallelism, GPGPU, parallel programming, C++ AMP, connection between acceleration and available hardware, acceleration of artificial neural networks and genetic algorithms