UNIVERZITET U SARAJEVU
ELEKTROTEHNIČKI FAKULTET
ODSJEK ZA RAČUNARSTVO I INFORMATIKU

# Paralelizacija algoritma brze Fourierove transformacije na GPU

SEMINARSKI RAD

- DRUGI CIKLUS STUDIJA -

**Mentor**:

Red. prof. dr Novica Nosović

**Asistent**:

Mr. dipl. ing. Tarik Hrnjić

**Studenti**:

Kemal Halilović
Kenan Omić
Miralem Memić
Faris Mehmedović

Sarajevo,
decembar 2021.

# Sažetak

U ovom radu opisana je brza Fourierova transformacija, njena teoretska postavka sa matematskom formulacijom i primjenom na različitim hardverskim platformama. Izvršena je analiza tehnologija pogodnih za paralelnu implementaciju računanja brze Fourierove transformacije, te obrazloženi su razlozi zbog kojih je odabrana CUDA platforma. Zatim je izvršena analiza nekoliko radova na sličnu temu, te su prikazani načini izvršenja paralelne brze Fourierove transformacije. Na kraju, opisani su očekivani ciljevi samog rada.

**Ključne riječi**: brza Fourierova transformacija, CUDA platforma,

# Sadržaj

# Popis slika

# Popis tabela

# Poglavlje 1

# Brza Fourierova transformacija

Fourierova transformacija ima mnogo primjena u nauci i inženjerstvu. Koristi se za širok spektar područja, kao što je digitalna obrada signala, slike, prepoznavanje glasa i tako dalje. Diskretna Fourierova transformacija je posebna vrsta Fourierove transformacije. Ona mapira sekvencu tokom vremena u drugu sekvencu preko frekvencije. Međutim, iako se diskretna Fourierova transformacija implementira jednostavno, njena vremenska složenost je $O(n^2)$. Shodno tome, da ovakav pristup nije najbolji u praksi koristi se brza Fourierova transformacija (eng. *fast Fourier transformation - FFT*) sa vremenskom kompleksnošću $O(nlog(n))$, koja je bazirana na manjem broju matematskih operacija [9].

Dostupnost hardvera posebne namjene u komercijalnom i vojnom sektoru doveo je do sofisticiranih sistema za obradu signala na osnovu karakteristika FFT-a. O popularnosti FFT-a svjedoči širok spektar primjena oblasti poput: konvencionalnog radara, sonara i aplikacija za obradu signala. Trenutna polja upotrebe FFT uključuju biomedicinski inženjering, spektroskopija, metalurška analiza, analiza nelinearnih sistema, mehanička analiza, geofizička analiza, simulacija, muzička sinteza i određivanje varijacije težine u proizvodnji papira iz celuloze. [10]

## 1.1 Diskretna Fourierova transformacija

Diskretna Fourierova transformacija je posebna vrsta Fourierove transformacije, koja zahtijeva ulaznu diskretnu funkciju i da nenulte vrijednosti imaju konačnu dužinu. Ulazna funkcija može biti konačna sekvenca realnih ili kompleksnih brojeva, tako da DFT je idealna za obradu informacija pohranjenih u računarima. Međutim, ukoliko su podaci kontinualni, podaci moraju biti uzorkovani kada koristimo DFT. Postoji mogućnost da ako je interval uzorkovanja previše širok, može izazvati efekat neprepoznavanja podataka, također ukoliko je uzorak podataka previše uzak, veličina digitaliziranih podataka može biti povećana. Definicija se izražava na sljedeći način:

Data sekvenca $x(n)$ za $k = 0,1,2,...,N$ je transformisana u sekvencu $X(k)$ od strane DFT preko navedene formule [9]:

$$X(k) = DFT_N\{x(n)\} = \sum_{n=0}^{N-1} x(n) e^{\frac{-2\pi i}{N} kn} \tag{1.1}$$

## 1.2 Formulacija brze Fourierove transformacije

Najprije, razmotrimo različite reprezentacije polinoma i vrijeme potrebno za završetak operacija na osnovu reprezentacije. Postoje tri glavne reprezentacije koje treba uzeti u obzir [7]:

1. Vektor koeficijenta sa monomskom osnovom
$$A(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{1.2}$$
2. Korijeni
$$A(x) = (x - r_0) \cdot (x - r_1) \cdot \cdots \cdot (x - r_{n-1}) \cdot c \tag{1.3}$$
3. Uzorci
$$(x_0, y_0), (x_1, y_1), ....(x_{n-1}, y_{n-1}) \tag{1.4}$$
gdje je $A(x_i) = y_i (\forall i)$ i svaki $x_i$ je različit.

Svaka od navedenih reprezentacija ima prednosti i mane, kada su u pitanju operacije evaluacije, sabiranja i množenja. U nastavku ćemo prikazati vremensku kompleksnost za operacije navedenih reprezentacija u sljedećoj tabeli (Tabela 2.1):

**Tabela 1.1:** Prikaz vremenske kompleksnosti operacija za tri glavne reprezentacije polinoma [7]

| Operacije | Koeficijenti | Korijeni | Uzorci |
|---|---|---|---|
| Evaluacija | $O(n)$ | $O(n)$ | $O(n^2)$ |
| Sabiranje | $O(n)$ | $\infty$ | $O(n)$ |
| Množenje | $O(n^2)$ | $O(n)$ | $O(n)$ |

Analizirajući navedenu tabelu, vidimo svaka reprezentacija polinoma posjeduje barem jedno nelinearno vremensko izvršavanje operacija respektivno. Ukoliko bismo se odlučili na reprezentaciju polinoma preko korijena, vidimo da je operacija sabiranja nemoguća u artimetičkom modelu (neophodno je beskonačno mnogo vremena za izvršavanje operacije), stoga okrenuti ćemo se ka reprezentaciji polinoma preko vektora koeficijenata sa monomskom osnovom i preko uzoraka. Shodno tome, da nećemo u potpunosti dobiti minimum prednosti koji pružaju navedene reprezentacije, fokusirati ćemo se na algoritam koji će moći konvertovati između ove dvije reprezentacije po potrebi operacija za vremensku kompleknost $O(nlog(n))$. Navedeni algoritam se naziva brza Fourieova transformacija.

Razmotrimo sada polinom u matričnoj formi:

$$V \cdot A = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} \tag{1.5}$$

gdje $V$ predstavlja Vandermondeovu matricu sa ulazima $v_{jk} = x_j^k$. Tada možemo konvertovati iz jedne u drugu formulaciju (koeficijenti i uzorci) koriteći proizvod $V \cdot A$. Vremenska kompleksnost navedenog proizvoda iznosi $O(n^2)$. Kako bismo postigli bolje rezultate od $\theta(n^2)$ kada radimo konverziju iz reprezentacije preko koeficijenata u uzorke, i obrnuto, neophodno je da izaberemo posebne vrijednosti za $x_0, x_1, ..., x_{n-1}$. Do sada smo samo napavili predpostavku da su vrijednosti $x_i$ različite.

### 1.2.1 "Podijeli i vladaj" pristup

Množenje polinoma možemo formulirati kao algoritam podijeli pa vladaj sa sljedećim koracima za polinom $A(x)$ $\forall x \in X$.

1. Podjelimo polinom $A$ na njegove parne i neparne koeficijente:

$$A_{parni}(x) = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k} x^k = \langle a_0, a_2, a_4, ... \rangle \tag{1.6}$$

$$A_{neparni}(x) = \sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1} x^k = \langle a_1, a_3, a_5, ... \rangle \tag{1.7}$$

2. Rekuzivno izračunajmo $A_{parni}(y)$ za $y \in X^2$ i $A_{neparni}(y)$ za $y \in X^2$, gdje je $X^2 = \{ x^2 \mid x \in X \}$.

3. Spojimo izraze:

$$A(x) = A_{parni}(x^2) + x \cdot A_{neparni}(x^2) \tag{1.8}$$

za $x \in X$.

Međutim, rekurentna relacija za ovaj algoritam glasi:

$$T(n, |X|) = 2 \cdot T(\frac{n}{2}, |X|) + O(n + |X|) = O(n^2) \tag{1.9}$$

što nije ništa bolje nego ranije.

Bolje performanse možemo postići ukoliko smanjujemo $X$, ili je $|X| = 1$ (bazni slučaj) ili je $X^2 = \frac{|X|}{2}$ i $X^2$ se (rekurzivno) smanjuje. Tada je rekurenta relacija u sljedećoj formi:

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n) = O(nlog(n)). \tag{1.10}$$

Redukcija vremenske kompleksnosti je postignuta sa "Podijeli i vladaj" algoritmom, što nam dozvoljava da formulu 1.1 raspišemo u sljedećem formatu [11]:

$$T(2k) = DFT_{N/2}\{x(n) + x(n+N/2)\} \tag{1.11}$$

$$T(2k) = DFT_{N/2}\{x(n) + x(n+N/2)\} \tag{1.12}$$

gdje je $k = 0, 1, ...N/2 - 1$ i $W_N^n = exp(-i2\pi n/N)$. Tada se dekompozicijske formule 1.11 i 1.12 mogu primjeniti za $N = 2^p$, gdje je $p$ cijeli broj.

## 1.3 Motivacija za paralelizaciju algoritma brze Fourierove transformacije

Glavna prednost algoritama brze Fourier transformacije je da se izvršavaju u manje vremena u poređenju sa naivnom implementacijom. Moguće je još optimizirati proračune FFT algoritama efektivnim korištenjem hardvera na kojima se izračunavaju. Paralelna implementacija FFT dozvoljava da se različiti dijelovi algoritma izvršavaju u isto vrijeme, postižući bolja vremena od sekvencijalne implementacije. Tabele 1.2 i 1.3 prikazuju izvršavanja različitih algoritama brze Fourierove tranformacije za sekvencijalnu i paralelnu implementaciju respektivno [8].

**Tabela 1.2:** Prikaz vremena izvršavanja algoritma brze Fourierove implementacije putem sekvencijalne implementacije [8]

| Ulazna/izlazna sekvenca | 8 uzoraka | 16 uzoraka |
|---|---|---|
| DFT | 178 ms | 231 ms |
| DIT | 159 ms | 185 ms |
| DIF | 161 ms | 202 ms |

**Tabela 1.3:** Prikaz vremena izvršavanja algoritma brze Fourierove implementacije putem paralelene implementacije [8]

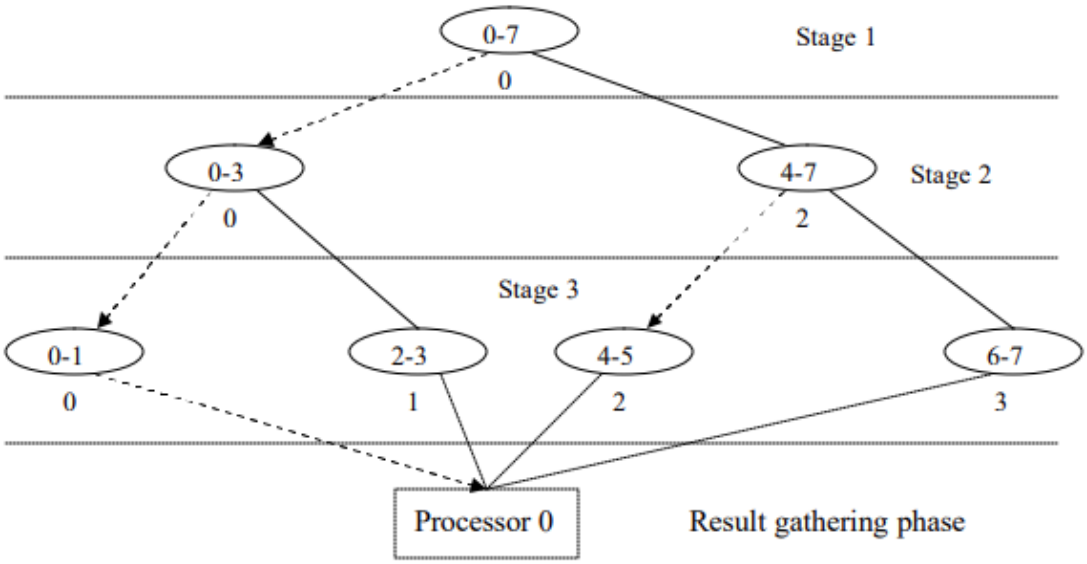
| Ulazna/izlazna sekvenca | 8 uzoraka | 16 uzoraka |
|---|---|---|
| DFT | 136 ms | 164 ms |
| DIT | 155 ms | 170 ms |
| DIF | 150 ms | 166 ms |

Rezultati pokazuju da za svaki algoritam, paralelna implementacija postiže bolje rezultate. Navedena konstacija postaje očiglednija sa povećanjem broja uzoraka za koje treba izračunati DFT. FFT algoritmi koriste prednosti brojnih svojstava opće DFT jednačine kako bi smanjili redundantnost i poboljšali ograničenja vremena i memorije. Stoga, za istu količinu ulaznih i izlaznih uzoraka, FFT algoritmi su brži i potrebno im je manje vremena da se izvrše u sličnim uvjetima.

Dalja optimizacija proračuna se postiže efektivnim korišćenjem procesorske jedinice. Računari su vremenom dovoljno poboljšani da bi mogli imati istinsku hardversku konkurentnost koju čine višeprocesorski računari ili procesori s više jezgara. Da bi se mogla koristiti ova hardversku konkurentnost, program mora biti paraleliziran kako bi se omogućilo da se različiti dijelovi programa izvršavaju na različitim procesorima/procesorskim jezgrama u isto vrijeme. Paralelizacija programa također poboljšava brzinu izvršavanja. Kao što pokazuju rezultati, također je ekonomičniji kada se izračunava DFT više uzoraka. Stoga je evidentno da su paralelni FFT algoritmi pogodni za višeprocesorsku implementaciju kako bi se osigurala efikasnost programa.

## 1.4 Izazovi implementacije paralelnog algoritma

Mogući način paralelizacije jeste da se opseg paralelizma povećava sa fazama. U prvoj fazi, svi podaci su zajedno i postavljaju se na jedan procesor. Izvršava se DFT u onoliko tačaka kolika je dužina algoritma i podaci se dijele na dva DFT-a sa pola tačaka. U drugoj fazi, oba DFT se postavljaju na zasebne procesore i nastavlja se pristup. Ako više nije moguće dijeliti elemente podataka, tada će procesor koji je trebao podijeliti skup podataka nastaviti da obavlja proračune za cijeli skup podataka iz prošle faze. Kada svi procesori završe proračune podataka, rezultati se skupe na jednom procesoru. Na slici 1.2 se vidi ovakav pristup sa 8-tačaka FFT algoritmom na sistemu sa tri procesora [1].




**Slika 1.1:** Paralelna kompozicija za FFT u 8 tačaka na sistemu sa 3 procesora [1].

Potencijalni izazov prilikom paralelne implementacije algoritma brze Fourierove transformacije jeste optimalna podjela posla, kao i komunikacija između procesora radi skupljanja podataka. Mogući problemi komunikacije u paralelnim sistemima su zastoji, stanje protočne strukture, itd. Također, sa našim pristupom, optimizacija procesora jeste problem, pošto u početnim fazama većina procesora nisu u stanju aktivnosti. Tek nakon što nekoliko faza paralelizacije prođe većina procesora se koristi za proračune. Izazov je dakle moguća upotreba neaktivnih procesora za bržu paralelizaciju algoritma.

# Poglavlje 2

# Odabir platforme i hardverskih resursa

Kada govorimo o tehnologiji koju ćemo koristiti za paralelizaciju brze Fourierove transformacije, naš izbor je između OpenCL i CUDA. Na kraju smo se odlučili za CUDA. Da bismo bolje uvidjeli zbog čega je baš CUDA naš odabir, pogledajmo detaljno navedene tehnologije:

## 2.1 OpenCL

OpenCL je *framework* za pisanje programa koji se izvršavaju na heterogenim platformama koje se sastoje od centralnih procesorskih jedinica, grafičkih procesnih jedinica, digitalnih procesora signala, polja koji se mogu programirati na terenu i drugih procesora ili hardverskih akcelaratora [12].

OpenCL navodi programske jezike koji su zasnovani na C99 i C++11 za programiranje uređaja i interfejsa koje koristimo za programiranje aplikacija, odnosno API-ja, za kontrolu platforme i izvršavanje programa na drugim računskim uređajima. Također, pruža standardni interfejs za paralelno računanje, zasnovan na taskovima i podacima. OpenCL je standard koji održava *Khronos Group*, kasnije je usvojen od strane kompanija poput Apple, Intel, Qualcomm, AMD, Nvidia, Samsung.

OpenCL na računarski sistem gleda kao da se sastoji od nekoliko računarskih uređaja koji mogu biti centralne procesne jedinice ili akcelaratori, poput grafičke procesne jedinice, priključene na host procesor. Pojedinačni računarski uređaj se najčešće sastoji od nekoliko računarskih jedinica koje sadrže procesne elemente. Ovdje je bitno napomenuti da sve funkcije koje su izvršene na OpenCL uređaju se nazivaju *kernelima* . Izvršavanje jednog jezgra se u ovom slučaju može izvoditi paralelno, na većini ili čak na svim procesnim elementima. Način na koji se računarski uređaj dijeli na računarske jedinice i procesne elemente ovisi o dobavljaču. Računarska jedinica može se smatrati jezgrom, ali pojam jezgra teško je definirati za sve tipove uređaja podržanih od strane OpenCL-a, a broj računarskih jedinica možda neće odgovarati broju jezgri traženih u marketinškoj literaturi dobavljača, od kojeg zapravo i zavisi na koji način se računarski uređaj dijeli na računarske jedinice i procesne elemente [12].

Programski jezik koji se koristi za pisanje računskih jezgri naziva se OpenCL C. Iz samog naziva može se pretpostaviti da je zasnovan na C99, ali ipak u dozi prilagođen da odgovara modelu uređaja u OpenCL-u. Memorijski međuspremnici nalaze se na određenim nivoima memorijske hjerarhije, dok su pokazivači označeni regionalnim kvalifikatorima : __global, __local, __constant i __private. Kao što smo ranije napomenuli, umjesto da program uređaja ima

glavnu funkciju, OpenCL C funkcije su označene kao __kernel, signalizirajući na taj način da su ulazne tačke u program koji se poziva iz host programa. Osobine ovog programskog jezika su i da postoje pokazivači na funkcije, nizovi promjenjive dužine su izostavljeni, dok je rekurzija zabranjena. Standardna biblioteka programskog jezika C zamijenjena je prilagođenim skupom standardnih funkcija, fokusiranih ka matematičkom programiranju.

Ranije smo pomenuli memorijsku hijerarhiju koja u OpenCLu je raspoređena na posebne nivoe hijerarhije, sa posebnim kvantifikatorima:

- globalna memorija : dijeljena na svim procesnim elementima, ima veliko kašnjenje kod pristupanja (__global)
- ROM : manjeg kapaciteta, manje kašnjenje, može se editovati od strane hosta, ali ne sa računarskim uređajima (__constant)
- lokalna memorija : dijeljena u grupi procesnih elemenata (__local)
- privatna memorija : dodijeljena privatno po elementu, registri (__private)

Ovdje je važno napomenuti da se ne očekuje da svaki uređaj implementira sve nivoe hijerarhije u hardveru. Konzistentnost između različitih nivoa hijerarhije nije rigorozna, samo se provodi eksplicitnim sinhronizacijskim konstrukcijama, ponajviše preprekama. Uređaji mogu ili ne moraju dijeliti memoriju s glavnim procesorom. Host API pruža kontrole na međuspremnicima memorije uređaja i funkcije za prenos podataka naprijed-nazad između hosta i uređaja [12].

OpenCL je proširen da olakša upotrebu paralelizma s vektorskim tipovima i operacijama, sinhronizaciju i funkcije za rad s radnim stavkama i radnim grupama. Konkretno, pored skalarnih tipova poput float i double, OpenCL nudi vektorske tipove fiksne dužine kao što je *float4* (4-vektor jednostruke preciznosti floats). Takvi vektorski tipovi su dostupni u dužinama 2, 4, 8 i 16 za različite osnovne tipove. Vektorizirane operacije na ovim tipovima namijenjene su mapiranju na SIMD skupove instrukcija, npr. SSF ili VMX kada pokrećemo OpenCL na CPU-ima. Drugi specijalizirani tipovi uključuju 2D i 3D tipove slika.

Računarski programi se prosljeđuju do OpenCL runtime kroz API pozive. API pozivi očekuju vrijednosti tipa *char. Česta praksa je čuvati ove programe u različitim datotekama.

### 2.1.1 Osnove programiranja u OpenCL frameworku

Kao i kod bilo kog drugog vanjskog API-ja pri korištenju C++, neophodno je uključiti određena zaglavlja, kao npr. CL direktorij : cl.h. Uobičajna osobina većine OpenCL API poziva je da li vraćaju kod greške tipa cl_int kao rezultat funkcije ili čuvaju greške na lokaciji koja je proslijeđena od strane korisnika kao parametar poziva. Zbog toga je potrebno implementirati funkciju koja provjerava da li se određeni poziv okončao uspješno.

**Program 2.1:** Funkcija checkErr provjerava okončanje poziva

```
inline void checkErr(cl_int err, const char *name) {
    if (err != CL_SUCCESS) {
        std::cerr << "ERROR: "<< name << "(" << err << ") " <<std::
           endl;
        exit(EXIT_FAILURE);
    }
}
```

Prvi korak ka inicijalizaciji i korištenju OpenCL-a je kreiranje konteksta. Kreiranje uređaja i memorija, kompajliranje i pokretanje programa se obavlja unutar konteksta. Kontekst može imati mnogo povezanih uređaja npr. CPU i GPU uređaje. Potrebno je alocirati OpenCL bafer koji drži rezultat kernela koji će se pokrenuti na uređaju. Primjer jednog takvog bafera je dat u nastavku:

**Program 2.2:** Buffer koji drži rezultat kernela

```
char * outH = new char[hw.length()+1];
cl::Buffer outCL(
context,
CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
hw.length()+1,
outH,
&err);
checkErr(err, "Buffer::Buffer()");
```

Program može imati mnogo ulaznih tački koje nazivamo kernelima. Za njihov poziv je potrebno izgraditi kernel objekat :

**Program 2.3:** Izgradnja kernel objekta

```
cl::Kernel kernel(program, "hello", &err);
checkErr(err, "Kernel::Kernel()");
err = kernel.setArg(0, outCL);
checkErr(err, "Kernel::setArg()");
```

Sva računanja se obavljaju korištenjem komandnog reda, svaki komandni red ima 1 na 1 mapiranje sa dodijeljenim uređajem. Što se tiče korištenja programa, potrebno je uraditi build i run. Build na Linux operativnom sistemu se radi pomoću jedne komande:

**Program 2.4:** Komanda za build na Linux OS

```
gcc -o hello_world -Ipath-OpenCL-include -Lpath-OpenCL-libdir
   lesson1.cpp -lOpenCL
```

Za pokretanje se koristi komanda

**Program 2.5:** Komanda za pokretanje na Linux OS

```
LD_LIBRARY_PATH=path-OpenCL-libdir ./hello_world
```

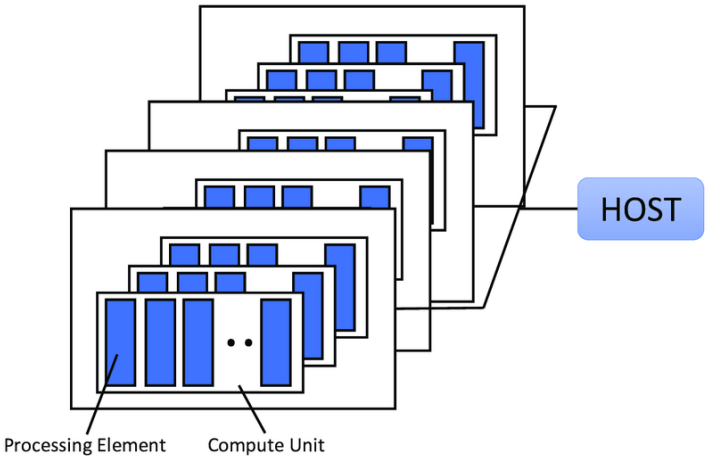
### 2.1.2 Platformski model OpenCL. Grupisanje radnih stavki. Memorijska organizacija

OpenCL je sačinjen od tri glavna dijela : specifikacija jezika, sloj API platforme, te vrijeme izvršenja API.

Specifikacija je bazirana na ISO C99 specifikaciji sa dodatnim ekstenzijama i restrikcijama. Dodaci uključuju vektorske tipove i operacije, optimizovan pristup slici i kvalifikatore prostora adrese. Restrikcije su izostanak podrške za funkcionalne pokazivače, bit polja i rekurziju.

Sloj API platforme radi pristup rutama softverske aplikacije. Ovaj sloj dopušta korištenje raznih koncepata za selektovanje i inicijalizovanje OpenCL uređaja, spremanje rada na uređaje i omogućavanje transfera podatka od jednog do drugih uređaja. Vrijeme izvršavanja API koristi kontekst za upravljanje jednim ili više OpenCL uređaja.

OpenCL platformski model definisan je kao host konektovan s jednim ili više OpenCL uređaja. Na sljedećoj slici vidimo jedan platformski model koji sadrži jedan host i više računarskih uređaja od kojih svaki ima više računarskih jedinica koje se sastoje od više procesirajućih elemenata [4].




**Slika 2.1:** Platformski model OpenCL [2]

Veliki broj puta je dosad pomenut pojam *host* bez eksplicitnog definisanja pojma.To je zapravo bilo kakav računar sa centralnom procesnom jedinicom, koja radi na standardnom operativnom sistemu. OpenCL uređaji također mogu biti i GPU, DSP ili pak CPU sa više jezgara. Računarska jedinica koja čini OpenCL uređaj se sastoji od jednog ili više procesirajućih elemenata. Oni izvršavaju instrukcije kao SIMD ili SPMD. SPMD instrukcije se najčešće izvršavaju na uređajima opšte namjene kao što je CPU, dok je za izvršavanje SIMD instrukcija neophodan vektor procesor, kao što je to GPU ili vektor jedinica koji se nalazi u CPU.

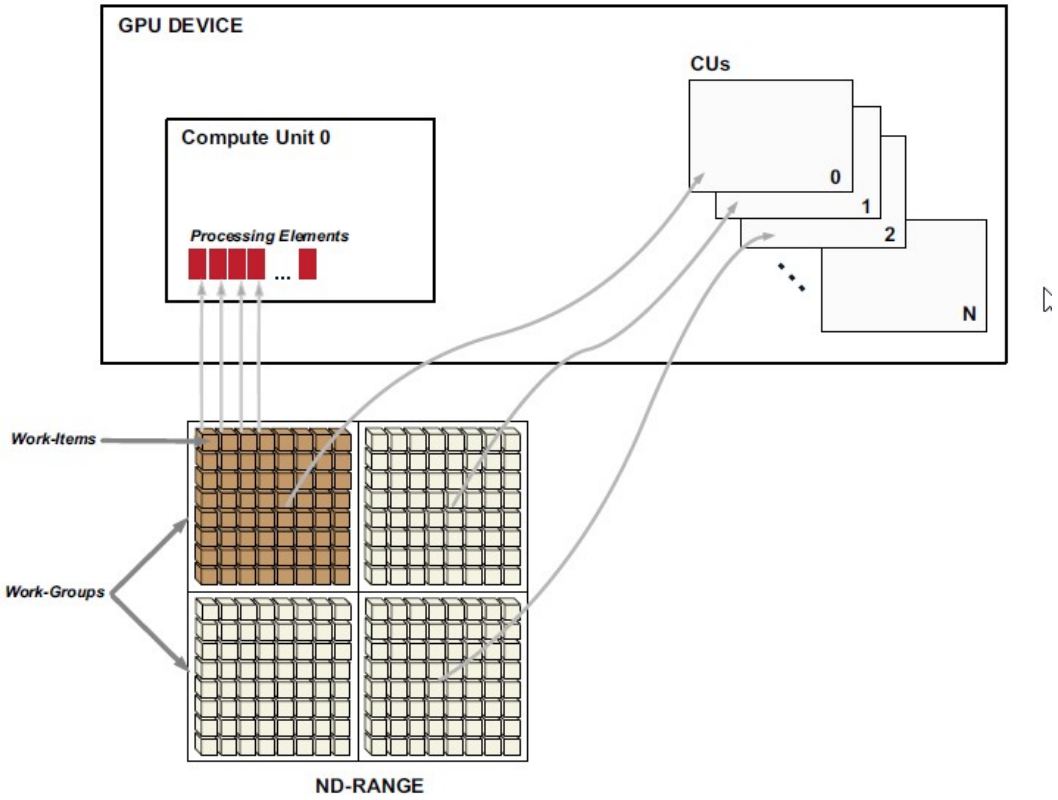
OpenCL model izvršavanja poredi dvije komponente kernele i host programe. Kernele smo već definisali kao osnovne jedinice koda koji se izvršava na jednom ili više OpenCL uređaja. Slijed funkcionisanja OpenCL modela teče otprilike ovako :

- Host program se izvršava na host sistemu
- Host program definiše kontekst uređaja
- Host program definiše red izvršavajuće instance kernela koristeći red naredbi
- Kerneli su poredani, ali mogu se izvršavati i van poretka

Konačno, dolazimo do najvažnijeg dijela o OpenCLu, a to je osobina paralelizma. OpenCL iskorištava paralelno računanje definisanjem problema u n-dimenzionalni prostor. Indeks prostora biva definisan onda kada se kernel smjesti u red za izvršavanje. U ovom indeksnom prostoru, svaki onaj elemenet koji je jedinstven i neovisan, se naziva radna stavka. Svaka radna

stavka izvršava istu kernel funkciju, ali na različitim podacima. Kako bi uređaj pratio ukupan broj radnih stavki koje zahtijevaju izvršenje, pri smještanju kernel komande u komandni red neophodno je definisati indeks prostora [2]. N-dimenzionalni indeks prostora može biti 1, 2 ili 3. Ukoliko je N=1, u pitanju je procesiranje linearnog niza podataka.Procesiranje slike je u pitanju ukoliko je N=2, dok je procesiranje 3D zapremnine u pitanju ukoliko je N=3.

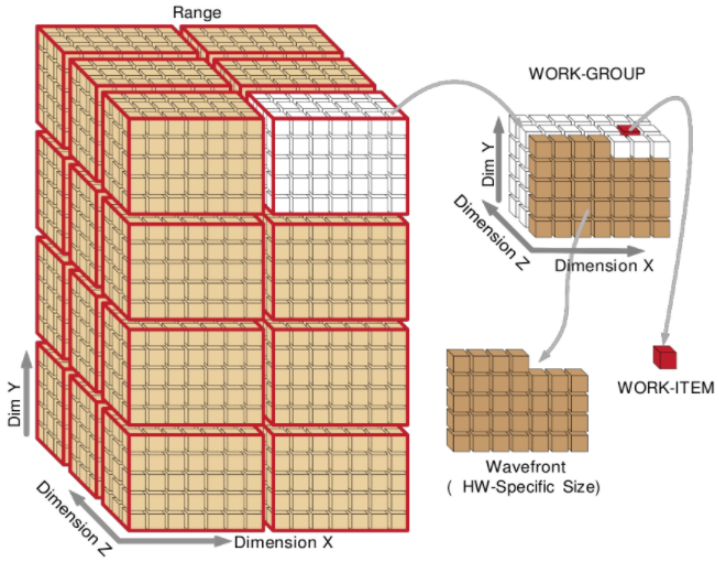
Dopušteno je grupisanje radnih stavki u radne grupe. Veličina svake radne grupe definisana je indeksom lokalnog prostora. Karakteristika radnih stavki koje su smještene u jednoj radnoj grupi je da se one izvršavaju skupa na istom uređaju. To je zato što je dopušteno da radne stavke dijele lokalnu memoriju i čine međusobnu sinhronizaciju. Globalne radne stavke su nezavisne, samim tim ne mogu biti sinhronizovane.




**Slika 2.2:** Grupisanje radnih stavki u radne grupe [3]

OpenCL model izvršavanja podržava dvije kategorije kernela OpenCL kernele i Native kernele. Razlika je ta što su OpenCL kerneli napisani u C i kompajliraju se sa OpenCL kompajlerom, dok su native ekstenzije kernela koji mogu da budu neke specijalne funkcije koje su definisane u kodu aplikacije, ili su pak eksportane iz neke biblioteke.

Radne grupe se dalje grupišu u *wavefronte*. Većinom se svaka radna grupa dodjeljuje određenoj računarskoj jedinici. Unutar jednog wavefront-a radne stavke se izvršavaju paralelno, dok se različiti wavefront-i izvršavaju sekvencijalno. Koliko se radnih stavki može dodijeliti određenom wavefrontu? Najčešće se to određuje brojem elemenata obrade u računarskoj jedinici. Programeri direktno ne mogu kontrolisati wavefront-e, s obzirom da se podjelom grupa u wavefront-e interno bave hardver i OpenCL [3].




**Slika 2.3:** Grupisanje radnih grupa u wavefront, zatim grupisanje wavefrontova u n-dimenzionalni grid [3]

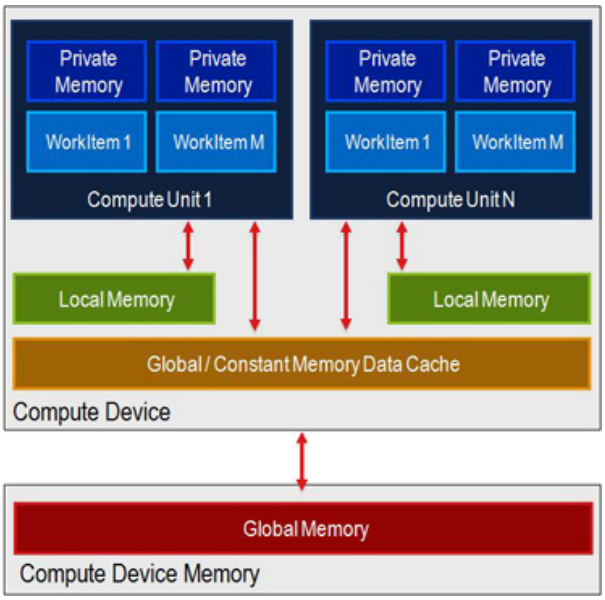
Host program je odgovoran je za postavljanje i upravljanje izvršenja kernela korištenjem konteksta. Korištenjem OpenCL API host može kreirati i manipulirati kontekstom uključivanjem :

- Uređaja ili seta OpenCL uređaja korištenih od strane hosta za izvršenje kernela
- Objekata programa koji implementiraju kernel(e)
- Običnih funkcija koje se izvršavaju na OpenCL uređaju
- Seta bafera memorije uobičajenih za hosta i OpenCL uređaje

Nakon što se kreira kontekst, kreiraju se komandni redovi da upravljaju izvršenjem kernela na OpenCL uređajima, koji su povezani na kontekst. Svaki komandni red prihvata tri tipa komandi :

- Komande koje se koriste za izršavanje kernela, pokreću kernel komandu na OpenCL uređaju
- Komande koje prebacuju memorijske objekte između memorijskog prostora OpenCL uređaja i hosta
- Komande sinhronizacije koje definišu poredak izvršavanja komandi. Kada se komande izvršavaju po redu, to je onda serijsko izvršavanje, ukoliko je bez reda onda je to bazirano na sinhronizacijskom ograničenju, koje je postavljeno na komandu.

Memorijski prostor je nedostupan na hostu i na OpenCL uređajima, samim tim je neophodno da OpenCL model obezbijedi regione memorije koji će biti dostupni za radne stavke pri izvršavanju kernela [4].




**Slika 2.4:** Memorijski model hijerarhije OpenCL [4]

Region globalne memorije predstavlja region u kom sve stavke rada i radne grupe imaju pristup čitanju i pisanju na računarskom uređaju i na hostu. Alocira se samo od strane hosta u toku vremena izvršavanja.

Region konstantne memorije je zapravo region globalne memorije koji ostaje konstantan pri izvršavanju kernela. Radne stavke imaju samo pristup čitanja ovog regiona, dok host nema ni pristup pisanja ni pristup čitanja.

Region lokalne memorije je korišten za dijeljenje podataka od strane radnih stavki u radnoj grupi. Sve radne stavke iz jedne grupe imaju i pristup čitanja i pisanja.

Region privatne memorije je označeni region koji je dostupan samo jednoj radnoj stavki. Većinom su host memorija i memorija računarskog uređaja neovisne jedna od druge. Komande koje se smještaju u red mogu biti blokirajuće ili neblokirajuće. Blokiranje znači da komanda host memorije čeka dok se završi memorijska transakcija prije nastavljanja. Neblokirajuća znači da host stavlja komandu u red i nastavlja, ne čekajući završetak memorijske transakcije [2].

## 2.2 CUDA

CUDA (ili Compute Unified Device Architectrue) je paralelna računarska platforma i API koji omogućava softveru da koristi određene tipove grafičke procesrske jedinice (GPU) za opće namjene. Ovakav pristup se naziva računarstvo opšte namjene na GPU-ovima (GPGPU).

CUDA platforma je dizajnirana za rad sa programskim jezicima kao što su C, C++ i Fortran. Pored toga, CUDA platforma podržava druge interface, kao što su OpenCL i OpenGL. CUDA je podržana od strane NVIDIA korporacije.

U srži CUDE su tri važne apstrakcice, hijerahija grupa thread-ova, zajednička memorija i sinhronizacija barijera. *Thread* u CUDI nije isto kao što je CPU thread, već je to osnovni element podataka koji se obrađuju. Grupa niti se naziva *warp*, što predstavlja minimalna veličinu podataka koje obrađuje CUDA. Napokon blokovi *warpova* se spajaju u gridove. CUDA runtime razbija te gridove i dozvoljava proširivu kontrolu nad hardverom. Ako hardver ima malo resursa, moguće je postaviti da se blokovi izvršavaju uzastopno, a ako ima veliki broj procesorskih jedinica, blokovi se mogu obrađivati paralelno.

Programski jezik koji se koristi za pisanje kernela se naziva CUDA C++. Ovaj jezik je ekstenzija C++ jezika koja omogućava kreiranje kernela. Slično kao u OpenCL, funkcije izvršene u CUDI nazivaju se *kerneli*, koji kada pozvani izvršavaju se određeni broj puta na različitim CUDA *threadovima*. Važno je napomenuti da CUDA kerneli se izvršavaju na GPU uređaju, dok ostatak C++ programa se izvršava na host uređaju, što je često CPU. Sa ovime, CUDA dozvoljava paralelno izvršenje sljedeći zadataka:

- Računanje na hostu
- Računanje na uređaju
- Prenos memorije sa hosta na uređaj
- Prenos memorija sa uređaja na host.
- Prenos memorije unutar memorije datog uređaja.
- Transfer memorije između uređaja.

### 2.2.1 Osnove programiranja u CUDA modelu

Da bi se kod unutar C++ programa pokrenuo na GPU uređaju, potrebno je samo imati CPU compiler pored host compilera. Tada postaje moguće kreirate kernele pomoću zagljavlja (__global__). Kada je funkcija definisana kao kernel, moguće je pozvati nju preko specijalne sintakse sa brojem blokava i thread-ova po bloku.

**Program 2.6:** Kernel funkcija add

```
1 __global__ void add(float* A, float* B, float* C)
2 {
3     *c = a + b;
4 }
```

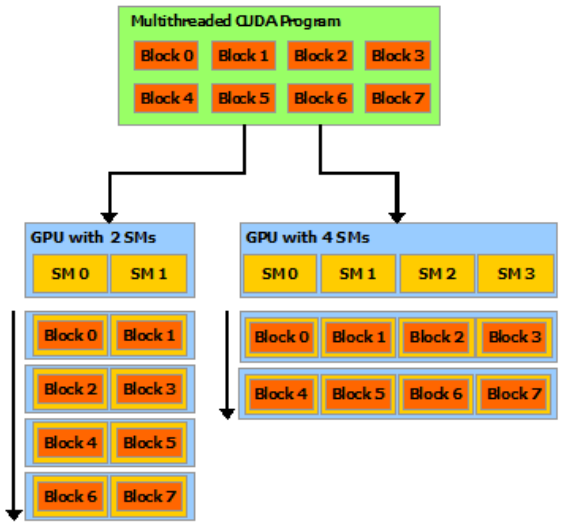
Da bi se ova funkcija pokrenula potrebno je samo pozvati funkciju preko specjalne sintakse «<X.Y»>. Vrijednost X predstavlja broj blokova na kojoj će se funkcija pokrenut, a Y broj thread-ova. Sa ovime postižemo paralelizaciju, pošto na svakom bloku se može ista funkcija sa drugačijim vrijednostima izvršavati. Ono što je važno je poziv cudaMalloc, koji umjesto da alocira memorija na hostu, zapravo je alocira na uređaju. Sa time se dobija pokazivač na pokazivača gdje se drži adresa novo alocirane memorije[13].

**Program 2.7:** Poziv add u mainu

```
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
    add<<<1,1>>>( 2, 7, dev_c );
    HANDLE_ERROR( cudaMemcpy( &c,
                              dev_c,
                              sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );
    return 0;
}
```

### 2.2.2 CUDA programerski i hardverski model

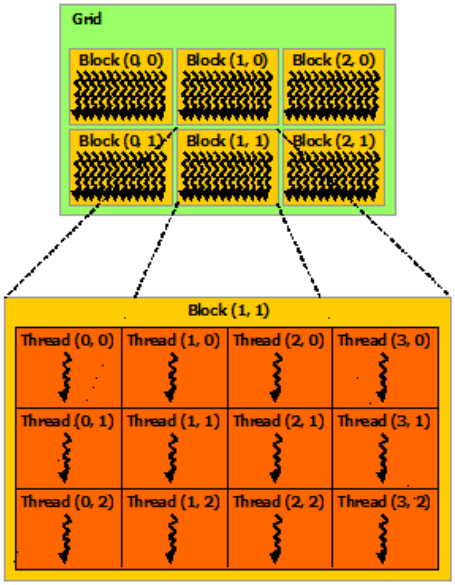
Do sada su bile spomenute mnogi termini u vezi CUDE, i potrebno je pogledat kako to radi kao čitav model. CUDA hardverski je sačinjena od dva dijela, hosta-a i uređaja. Host je bilo kakav računar na kojem se kod izvršava, i CUDA nema interakcije sa njime. Uređaj je bilo koji uređaj spojen sa hostom koji podržava izvršavanje CUDA koda, kao što je GPU. Ovo razdvajanje je važno pošto omogućava već spomenute apstrakcije, koje obezbeđuju paralelizam podataka i thread-ova. Ovime se problemi dijele na pod-probleme koji se mogu rješavati nezavisno na paralelnim blokovima, a svi pod-problemi se mogu razdvojiti na finije dijelove koje se mogu riješavati paralelno preko svih thread-ova u blokovima. Zaista, svaki blok thread-ova može biti zakazan na bilo kojem od dostupnih multiprocesora unutar uređaja, GPU, bilo kojim redoslijedom, istovremeno ili sekvencijalno, tako da se kompajlirani CUDA program može izvršiti na bilo kojem broju multipocesora.




**Slika 2.5:** Automatska skalabilnost GPU-a[5]

Ova ideja je ono što omogućava paralelizam uz pomoć CUDE. CUDA dozvoljava paralelno riješavanje problema do N dimenzija, što opisuje dimenzije matrice blokova koji riješavaju probleme. Kada je *N* jednak 1, riješava se linearan niz blokova, kada je jednak 2 to je procesiranje dvo-dimenzionalne matrice blokova, ili slike, a kade je 3 to je tro-dimenzionalna matrica blokova, ili procesiranje 3D objekta. Broj blokova u ovim dimenzijama je ograničen samo od strane snage uređaja.

Kao što je već spomenuto, grupisanje thread-ova se naziva blok. Postoji ograničenje za broj thread-ova po bloku, budući da se očekuje da sve niti blokova borave na istoj procesorskoj jezgri i moraju dijeliti ograničene memorijske resurse te jezgre. Međutim kernel se može izvršit sa više blokova thread-ova jednakog oblika, tako da je ukupan broj thread-ova jednak broju thread-ova po bloku pomnoženom sa brojem blokova.




**Slika 2.6:** Mreža Thread Blokova GPU-a[5]

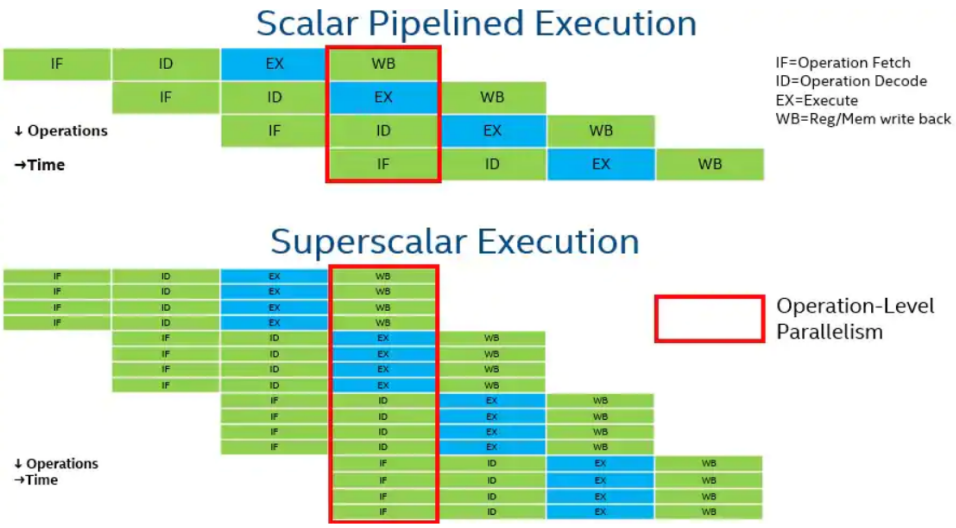
## 2.3 Poređenje CPU i GPU arhitekture

Nakon što smo se upoznali sa odabranom platformom, odnosno softverskom stranom, u ovom poglavlju će biti objašnjena hardverska strana. CUDA implementira CUDA device kompajler koji može generirati kod za ciljni hardver koji može uključivati GPU-ove. Kompajler diktira koje će ciljeve podržavati. CUDA aplikacija ima dijelove koje rade na CPU-u, ali cilj je razbijanje aplikacije na dijelove na GPU. Tipično, GPU će imati mnogo više jezgara nego čak i višejezgreni CPU. Imati puno jezgri može pomoći u nekim aplikacijama, ali svako ubrzanje ima tendenciju da bude specifično za aplikaciju. Neke aplikacije će raditi bolje na višejezgrenom CPU-u, dok će druge biti bolje sa GPU-ovima. Mnogo toga zavisi od toga kako su različiti kerneli napisani i izvođeni, kao i od vrste operacija koje se izvode. Operacije koje iskorištavaju funkcionalnost GPU-a obično će raditi bolje na GPU-u. Ponekad kombinacija hardvera može biti najbolja alternativa. Moguće je imati CUDA sistem koji obuhvata oba. S obzirom na navedeno, u nastavku će biti opisana arhitektura CPU-a i GPU-a te korištene tehnike kako bi se ostvario paralelizam a samim tim i bolje performanse [5].

### 2.3.1 CPU arhitektura

CPU arhitektura se ponekad naziva skalarnom arhitekturom jer je dizajnirana da efikasno obrađuje serijske instrukcije. CPU-i su, kroz mnoge dostupne tehnike, optimizirani za povećanje paralelizma na nivou instrukcija (ILP) tako da se serijski programi mogu izvršavati što je brže moguće.

Skalarna protočna struktura može izvršiti instrukcije, podijeljene u faze, brzinom do jedne instrukcije po ciklusu takta (IPC) kada nema zavisnosti. Da bi se poboljšale performanse, moderne CPU jezgre su višenitni superskalarni procesori sa sofisticiranim mehanizmima koji se koriste za pronalaženje paralelizma na nivou instrukcija i izvršavanje više instrukcija van reda po ciklusu takta. Oni preuzimaju mnogo instrukcija odjednom, pronalaze graf zavisnosti tih instrukcija, koriste sofisticirane mehanizme predviđanja grananja i izvršavaju te instrukcije paralelno (obično sa 10x performansama skalarnih procesora u smislu IPC) [6]. Dijagram ispod pokazuje pojednostavljeno izvođenje skalarne protočne strukture i superskalarnog CPU-a.




**Slika 2.7:** Poređenje izvršenja instrukcija skalarne protočne strukture i superskalarnog CPU-a [6].

Postoje mnoge prednosti korištenja CPU-a za računanje u poređenju sa prenošenjem na koprocesor, kao što je GPU ili FPGA. Prvo, pošto podaci ne moraju da se prebacuju, latencija je smanjena uz minimalne troškove prenosa podataka. Budući da su visokofrekventni CPU-i podešeni da optimiziraju skalarno izvršenje i da je većina softverskih algoritama serijska po prirodi, visoke performanse se lako mogu postići na modernim CPU-ima. Za dijelove algoritma koji se mogu vektorski paralelizirati, moderni procesori podržavaju jedna instrukcija-više podataka (SIMD) instrukcije. Kao rezultat, CPU-i su prilagođeni širokom spektru radnih opterećenja. Čak i za masovno paralelna opterećenja, CPU-i mogu nadmašiti akceleratore za algoritme sa velikom divergencijom grana ili visokim paralelizmom na nivou instrukcija, posebno tamo gdje je odnos veličine podataka i izračunavanja visok.

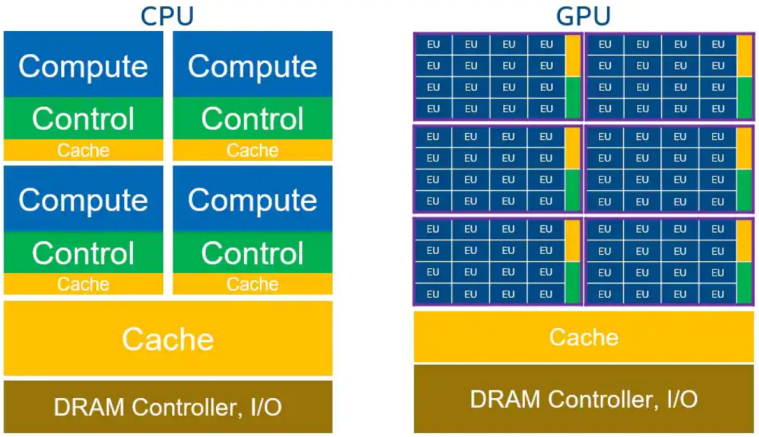
Prednosti CPU-a:

- Superskalarno out-of-order izvršenje
- Sofisticirana kontrola za izdavanje ogromnog paralelizma na nivou instrukcija
- Precizno predviđanje grananja
- Automatski paralelizam na sekvencijalnom kodu
- Veliki broj podržanih instrukcija
- Manja latencija u poređenju sa ubrzanjem rasterećenja
- Sekvencijalno izvršavanje koda rezultira lakoćom razvoja

CPU-i se oslanjaju na nekoliko tipova paralelizma da bi postigli visoke performanse: SIMD paralelizam podataka, paralelizam na nivou instrukcija i paralelizam na nivou niti sa više niti koje se izvršavaju na različitim logičkim jezgrima [6].

### 2.3.2 GPU arhitektura

GPU-ovi su procesori napravljeni od masovno paralelnih, manjih i specijalizovanih jezgara od onih koji se generalno nalaze u CPU-ima visokih performansi. Arhitektura GPU-a je optimizirana za ukupnu propusnost kroz sva jezgra, ne naglašavajući latencije i performanse pojedinačnih niti. Arhitektura GPU-a efikasno obrađuje vektorske podatke (niz brojeva) i često se naziva vektorska arhitektura. Kod GPU-ova, više prostora je izdvojeno za računanje, a manje za keširanje i kontrolu. Kao rezultat toga, GPU hardver istražuje manje paralelizma na nivou instrukcija i oslanja se na softverski dat paralelizam da bi postigao performanse i efikasnost. GPU-ovi su in-order procesori i ne podržavaju sofisticirano predviđanje grananja. Umjesto toga, oni imaju mnoštvo aritmetičko-logičkih jedinica (ALU) i dubokih protočnih struktura. Performanse se postižu višenitnim izvršavanjem velikih i nezavisnih podataka, čime se amortizuju troškovi jednostavnije kontrole i manjih keš memorija. GPU-ovi koriste SIMT model izvršavanja (jedna instrukcija-višestruke niti). SIMT je model izvršenja koji se koristi u paralelnom računarstvu gdje se SIMD model (jedna instrukcija, višestruki podaci) kombinuje sa višenitnim radom [14]. Višestruki SIMD tokovi instrukcija se mapiraju u jednu izvršnu jedinicu (EU), a GPU može da prebaci EU između tih tokova SIMD instrukcija kada se jedan tok zaustavi. Slika 2.6 u nastavku pokazuje razliku između CPU-a i GPU-a.




**Slika 2.8:** Razlika između CPU i GPU arhitektura [6].

EU je osnovna jedinica obrade na GPU-u. Svaka EU može obraditi više SIMD tokova instrukcija. U istom silikonskom prostoru, GPU-ovi imaju više jezgara/EU-a od CPU-a. GPU-ovi su organizovani hijerarhijski. Višestruki EU se kombinuju kako bi formirali računarsku jedinicu sa zajedničkom lokalnom memorijom i mehanizmima sinhronizacije (poznati kao sub-slice ili streaming multiprocesori, označeni ljubičastom bojom). Računalne jedinice se kombinuju da formiraju GPU.

Prednosti GPU-a:

- Masovni paralelizam, do hiljada malih i efikasnih SIMD jezgara/EU
- Efikasno izvršavanje podatkovno-paralelnog koda
- Veliki dinamički propusni opseg memorije sa slučajnim pristupom (DRAM).

GPU-i se oslanjaju na velika radna opterećenja paralelnih podataka kako bi se postigle visoke performanse. Kao rezultat toga, jezgra sa jednim zadatkom se rijetko koriste. Prilikom izvršavanja kernela na GPU-u, svaka radna stavka je mapirana u SIMD traku. Podgrupa se formira od radnih stavki koje se izvršavaju na SIMD način, a podgrupe se mapiraju na GPU EU. Radne grupe, koje uključuju radne stavke koje mogu sinhronizirati i dijeliti lokalne podatke, dodjeljuju se za izvršenje na računskim jedinicama [6].

# Poglavlje 3

# Referentni radovi

## 3.1 Effectiveness of Fast Fourier Transform implementations on GPU and CPU

Autori Dariusz Puchała, Kamil Stokfiszewski, Mykhaylo Yatsymirskyy i Bartłomiej Szczepaniak u radu *Effectiveness of Fast Fourier Transform Implementations on GPU and CPU* [11] predstavljaju rezultate poređenja efikasnosti odabranih varijanti radix-2 brze Fourierove transformacije (BFT) algoritma implementiranih na grafičkim (GPU) i centralnim (CPU) procesorskim jedinicama. U ovom radu razmatrana su dva pristupa: realni i imaginarni dijelovi pohranjeni kao parovi u jednom baferu podataka (1B) i dva odvojena bufera, koji se koriste za skladištenje realnih i imaginarnih dijelova (2B). Također su razatrane su dvije verzije BFT algoritma: MY FFT i CT FFT.

Na 50 ispitivanja, testovi su obavljeni pomoću Intel Core i7, 3.5 GHz, procesor sa 16 GB RAM-a i NVidia GeForce GTX 970 grafičkom karticom, sa 4GB DRAM-a. Sve procedure izračunavanja BFT-a za CPU i GPU su prvobitno implementirane od strane autora u jeziku C koristeći Microsoft Visual C++ 2012 kompajler, sa opcijama optimizacije kôda u cilju postizanja perfomanse vremenske efikasnosti. U svrhu izvršavanja procedura na GPU, autori su koristili NVIDIA R CUDA R Toolkit 7.5 NVCC C++ kompajler jezika sa standardnim opcijama optimizacije koda.

Rezultati rada pokazuju da GPU implementacije, za velike veličine $N$ kod dvije posmatrane brze transformacije, su bile znatno bolje od njihovih odgovarajućih CPU implementacija, za njihove B1 i B2 varijante, što snažno dokazuje prednost implementacije BFT algoritama na GPU-ovima. Pored navedenog, najbolji rezultat među svim razmatranim implementacijama za velike vrijednosti $N$ postignut je B1 varijantom GPU implementacija MY FFT algoritma. S druge strane, kod CPU implementacija verzija algoritma MY FFT za B1 i B2 pristupe je 2 i 2.5 puta respektivno brža od verzije algoritma CT FFT. Sumarno, i za CPU i za GPU implementacije od razmatranih FFT algoritama, MY FFT B1 postiže najkraća vremena izvršenja.

## 3.2 Accelerating Fast Fourier Transformation for Image Processing using Graphics Processing Unit

Mohammad Nazmul Haque i Mohammad Shorif Uddin su napisali naučni rad po nazivom *Accelerating Fast Fourier Transformation for Image Processing using Graphics Processing Unit* [15], unutar kojeg su uporedili performanse obrade slike zasnovane na BFT algoritamu za pokretanje na CPU i GPU. Ovaj rad opisuje ubrzanje FFT algoritma na NVIDIA GeForce G 103M GPU-u i Intel Core2 Duo procesoru.

Za ispitivanje preformansi autori referentog rada su se odlučili, između ostalog, za rad sa CUDA 3.2 GeForce G 103M, uključujući CUDA SDK kompajler, kao i CUFFT i CUBLAS biblioteke. Testovi su obavljeni pomoću Intel(R) Core(TM)2 Duo CPU, 2.0 GHz, procesor sa 2048 MB RAM i NVIDIA GPU 3.2 skupom alata potrebnih za CUDA programiranje.

Eksperimentalna analiza je vršena sa tri predložene slike, različitih dimenzija, kroz 100 iteracija. Eksperimentalni rezultati pokazuju značajno ubrzanje algoritma na GPU-u nego kod implementacije zasnovane na CPU-u. Uočeno je da su performanse algoritma na CPU i GPU sa jako malim razlikama (maksimalna vrijednost vremenskog trajanja je 0.4 i 0.2 ms respektivno) za slike rezolucije 512x512. Sa povećenjem rezolucije slike na 1024x1024 znatne razlike između implemantacija na CPU i GPU su primjetne, gdje je implementacija na GPU čak 410.00% brža od implementacije na CPU. Minimalna postignuta razlika između navedenih implementacija bila je 4.05x, dok maksimalna razlika je iznosila 4.128x. Naposlijetku, za slike sa rezolucijom 2048x2048, implementacija na GPU je 434.90% brža od implementacije na CPU.

Osim prednosti performansi obrade slike zasnovane na algoritmu brze Fourierove transofrmacije na GPU-u u odnosu na CPU-u, izražne su i druge prednosti. Naime, CPU može biti preokupirama sa vremenskim kritičnim zadacima, kao što je upravljanje hardverom pri akviziciji podataka. U tom slučaju, preporuka je koristiti GPU za procesiranje slike, a ostaviti akviziciju podataka CPU. Razlog tome je da GPU radi bez interapta, pa ima bolje performanse od CPU-a koji radi sa interaptima iz operativnog sistema.

## 3.3 Fourier Transform Based GPU Acceleration

Jingshen Li je napisao naučni rad sa nazivom *Fourier Transform Based GPU Acceleration* [16], koji se fokusira na ubrzavanje Brze Fourierove transformacije za obradu slika pomoću GPU-a. Unutar rada se upoređuju simluacije obrade slika preko CPU i GPU. Odnos veličine podataka i količina utrošenog vremena za izračun je analiziran na tradicionalnoj metodi serijskog izvršavanja operacija na CPU i paralelnog izvršavanja operacija na GPU. Kompjuterske simulacije potvrđuju da se brzina izvršavanja može poboljšati odnosno skratiti u slučaju korištenja GPU nad velikim podacima.

U digitalnoj obradi slike, Fourierova transformacija je slika iz prostornog domena u frekvencijski domen, a inverzna transformacija je slika iz frekvencijskog u prostorni domen. Ova transformacija je široko korištena u poljima kompresije, segmentacije, rekonstrukcije slike i slično. Za kompjutersku analizu i simulaciju korištene su slike rezolucija 200x200, 400x400, 600x600, 800x800 i 1000x1000. Korišteni operativni sistem je Windows 7, platforma MATLAB2009A a grafička kartica je GTX580 koja ima podršku za CUDA paralelno procesiranje.

Rezultati pokazuju da, bez obzira preciznost podataka, povećavanjem količine podataka vrijeme izvršavanja na CPU se povećava znatno, dok se vrijeme izvršavanja na GPU jako sporo povećava. Da bi uvidjeli ovu razliku u performansama potrebni su veliki ulazni podaci, konkretno sa ovom konfiguracijom počevši sa slikama minimalne rezolucije 600x600.

Uz razvoj digitalnih slika velikog kapaciteta, poboljšati brzinu obrade što je prije moguće te karakteristika slike i informacija, posebno u pravovremenom prepoznavanju igra vrlo važnu ulogu. Sa razvojem računarskih tehnologija visokih performansi, GPU operacija u pokretnom zarezu, aritmetičkog i paralelnog računanja, računarski kapacitet se ubrzano poboljšao. Ovaj rad uglavnom uvodi upotrebu baziranu na GPU ubrzanom pristupu analizi Brze Fourierove transformacije. Kroz kompjutersku simulaciju zaključujemo da, za velike podatke, uz pomoć snažnog računarskog kapaciteta i mogućnosti paralelnog procesiranja GPU-a, dolazimo do poboljšanja vremena potrebnog za izvršavanje operacija. Upotreba visokih brzina, visoke efikasnosti, niskih troškova procesiranja slika velikih rezolucija ili mnoštva slika, baziranog na GPU ubrzanoj metodi, će zasigurno imati važnu ulogu u procesiranju digitalnih slika.

## 3.4 High Performance Discrete Fourier Transforms on Graphics Processors

Naga K. Govidaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith i John Manferdelli u radu *High Performance Discrete Fourier Transforms on Graphics Processors* [17] su predstavili različite algoritme za računanje Fourier transformacije na grafičkim procesorskim jedinicama. U radu su hijerarhijski predstavljeni korijenski FFT algoritmi za dimenzije stepena dvojke, kao i za dimenzije koje nisu stepen dvojke. Hijerarhijski FFT algoritmi efikasno iskorištavaju dijeljenu memoriju GPU-a koristeći *Stockham formulation*. FFT algoritmi analizirani u radu su: *Global Memory FFT*, *Shared Memory FFT*, *Hierachical FFT*, *Mixed-radix FFT*, *Bluestein's FFT*, *Multi-dimensional FFTs* i *Real FFTs*. Algoritmi su testirani na tri različita GPU-a, 8800GTX, 8800GTS i GTX280. U radu su zaključili da *Hierachical FFT* ima najbolje performanse u ovom hardveru, sa poboljšanjima od 8-40x u poređenju sa CPU-ima.

Razlog zbog kojeg GPU predstavljaju odličnu metu za računanje je vrlo jednostavan - visoke performanse uz malu cijenu. Npr. GPU koji košta 300$, može teoretski da peaka performanse koje iznose preko 1 TFLop/s, i da kreira saobraćaj od preko 100GiB/s. Tipično, algoritmi opće namjene, namijenjeni za GPU moraju da budu adekvatno mapirani programskom modelu grafičkog API-ja. U skorije vrijeme, alternativni API-ji su omogućili da hardveri niskog nivoa izlažu svojstva koja mogu omogućiti značajna unapređenja pefromansi.

Veliki broj implementacija FFT algoritama na GPU koristi grafički API poput trenutne verzije OpenGL ili DirectX. Nedostatak ovih API-ja je što oni direktno ne podržavaju scatter, pristup dijeljenoj memoriji ili sitnozrnastu sinhronizaciju omogućenu na modernim GPU-ovima. Pristup ovim svojstvima može biti omogućen samo od strane specifičnih prodavača API-ja. NVIDIA-ina FFT biblioteka zvana CUFFT, koristi CUDA API za dostizanje većih performansi od onih koje su dostupne na grafičkim API-jima.

Za ispitivanje performansi algoritama autori rada su koristili NVIDIA CUDA API i uporedili rezultate sa NVIDIA CUFFT bibliotekom i Intel CPU optimizacijom na high-end quad core centralnoj procesnoj jedinici. Na NVIDIA GPU, autori rada su dostigli performanse do 300 Gflopsa (1 Gflop je jednak jednoj milijardi floating-point operacija po sekundi), što je u odnosu na tipične performanse na CUFFT biblioteci bolje za 2-4 puta, dok se u odnosu na MKL uvidjelo se poboljšanje od 8-40 puta.

## 3.5 Fast Simulation of Laser Heating Processes on Thin Metal Plates with FFT Using CPU/GPU Hardware

Daniel Mejia-Parra, Ander Arbelaiz, Oscar Ruiz-Salguero, Juan Lalinde-Pulido, Aitor Moreno i Jorge Posada su u radu *Fast Simulation of Laser Heating Processes on Thin Metal Plates with FFT Using CPU/GPU Hardware*[18] predstavljaju korištenje Brze Fourierove transformacije u oblasti simulacije laserskog zagrijavanja/rezanja tankih pravouglih ploča. Trenutne najsavremenije metode u ovoj oblasti koriste spektralna analitička rješenja koja značajno umanjuju potrebno vrijeme računanja u poređenju na industrijski standard koji je zasnovan na pristupu konačnih elemenata. Međutim, ta spektralna rješenja nisu bila prethodno predstavljena u smislu Fourierovih metoda i FFT. Autori su u ovom radu predstavili rješenja koja koriste FFT i na taj način smanjili složenost procesa sa $O(M^2N^2)$ na $O(M \cdot N \cdot log(MN))$ gdje je MxN veličina diskretizacije ploče. Rezultati testiranja pokazuju da ovaj pristup nadmašuje non-FFT pristup u performansama i u CPU i GPU hardveru, što rezultira i do 100 puta bržim izvršavanjem.

Za maksimalnu iskorištenost hardvera korišteni su visoko optimizirane FFT biblioteke. Mjesto programskog jezika je zauzeo Python koji uključuje u svoj naučni paket omote visokog nivoa za C/C++ biblioteke, radi brze izrade prototipova. Za optimizaciju CPU-a izabrane su FTTPACK, MKL i FFTW biblioteke, a za GPU cuFFT biblioteka iz NVIDIA CUDA Toolkita. Sve ove biblioteke koriste paralelizaciju sa više jezgara, instrukcije vektorizacije, efikasno korištenje memorije i primjenu specifičnog FFT algoritma za iskorištavanje osnovnog hardvera do najvišeg stepena. Dvije testne platforme su korištene za mjerenje performansi: (i) desktop PC sa Windows 10 OS, Intel Core i5-6500 (CPU), 16GB RAM i NVIDIA GeForce GTX 960 (GPU) i (ii) desktop PC sa Manjaro (GNU/Linux), Intel Core i7-4700K (CPU), 16GB RAM i NVIDIA GeForce RTX 2060 (GPU).

Izvršena su testiranja na pomenute tri biblioteke i dva različita CPU-a. Stepen optimizacije postignut za FFT algoritme sa MKL i FFTW bibliotekama je viši u odnosu na FFTPACK. Ove biblioteke bolje koriste osnovni hardver, postižući brže rezultate od FFTPACK biblioteka za metode zasnovane na FFT-u. Rezultati dobijeni sa FFTW bibliotekom su neznatno bolji (brži) od MKL. Međutim, to može biti zbog upotrebe omotača, kao što je pyfftw (FFTW) omotač koji nudi veću kontrolu nad implementacijom. Ipak, dobijeni rezultati uvelike nadmašuju trenutne najsavremenije tehnike, i FFTW i MKL su pokazali vrijeme izvršenja ispod 1 s za ploču veličine do 4096 × 4096. U sveukupnom poređenju, uzimajući u obzir i CPU i GPU, FFTPACK je najsporija i cuFFT je najbrža implementacija. Vremena izvršenja za biblioteke MKL i FFTW su slične, gdje je FFTW ostvario malo bolje rezultate. Sve u svemu, hardversko ubrzanje GPU-a (sa cuFFT) pruža značajno ubrzanje, što ga čini dobrom alternativom za razmatranje za simulacije na pločama sa velikim veličinama diskretizacije. Ovo poređenje pokazuje da GPU hardver efikasno ubrzava vreme računanja, između najbržeg CPU-a (i7-4700K) i najsporiji GPU (GTX 960), postižući do 2× ubrzanje za veličine ploča veće od 1024 × 1024. Razlika u performansama se povećava kako se ulazna ploča povećava u veličini.

Procjena performansi pokazuje da minimalno potrebno vrijeme računanja zavisi u odnosu na korištene biblioteke, posebno za velike ulazne veličine. Nadalje, dobiveni rezultati kazuju da je i CPU i GPU implementacija efikasnija od trenutno korištenih u industriji, s tim što je GPU znatno brža od CPU rješenja. Konkretno, korištenjem GPU hardvera, vrijeme izračunavanja za procjenu temperature se smanjuje sa 1 s na 0,01 s (100×brže), mjereno u NVIDIA GeForce GTX 960 (GPU), dok moderniji GeForce RTX2060 (GPU) postiže još bolje rezultate.

## 3.6 An efficient parallel implementation of 3D-FFT on GPU

Trodimenzionalna brza Fourierova transformacija (3D-FFT) je vrlo zahtjevna za izračun na jezgrima u brojnim poljima primejne. Kako bi uspješno uspjeli da stanu rame uz rame sa enormnom količinom zahtjeva, predložene su mnogobrojne naučne solucije za izračun FFT od strane raznih naučnih grana. Svejedno, mnoge ove solucije imaju jako dug razvojni ciklus, veliku cijenu dizajna i fiksnu funkcionalnost. S druge strane, softverske solucije su dosta jeftinije, skalabilne su i fleksibilne i imaju kraći razvojni ciklus. Autori rada *An efficient parallel implementation of 3D-FFT on GPU* [19], *Selcuk Keskin, Ertunc Erdil i Taskin Kocak* predstavljaju 3D-FFT algoritam koji radi na GPU. Demonstrirane su performanse ovakvog pristupa na vještački generisanim 3D slikama različitih dimenzija. Eksperimentalni rezultati su pokazali da 3D-FFT implementacija na GPU doseže do 486 GFlop/s sa memorijskim i algoritamskim optimizacijama.

Procesiranje slika i kompjuterska vizija su dva istraživačka polja gdje 3D-FFT se najčešće koristi. Neka polja primjene u ovim granama nauka su ekstraktovanje specifičnog signala iz slike, segmentacija tekstura i rekonstrukcija slika. Grafička procesna jedinica (GPU) je postala jedna od najpopularnijih platformi za razvoj računanja visokih performansi. Razlog za to je što GPU pruža paralelnu arhitekturu, koja kombinuje programabilnost sa suhom moći izračunavanja. Izvršavajući hiljade niti istovremeno i kreirajući veliku količinu prometa, GPU je postala neizbježna platforma za implementiranje FFt algoritama visokih performansi. Jedina javna FFT biblioteka za GPU je cuFFT biblioteka, obezbijeđena od strane NVIDIA koja podržava višedimenzionalnu transformaciju kompleksnih podataka jednostruke preciznosti i ID izvršavanja podataka

U ovom radu, predložena je implementacija 3D-FFT algoritma koja se izvršava na GPU sa visoko-zahtjevom podrškom tokom izvršavanja. Za uspješno testiranje performansi ovog algoritma, autori rada su vještački generisali 3D fotografije raznih dimenzija. Eksperimentalni rezultati su demonstrirali efikasnost predloženog pristupa u odnosu na postojeće 3D-FFT implementacije. GPU omogućava ekstremno visoku mogućnost računanja zapošljavajući veliki broj jezgri da rade na velikom skupu podataka. *Compute UNified Device Architecture* CUDA, razvijena od strane NVIDIA corp., predstavlja široko korišten programski pristup u velikom broju paralelnih računarskih sistema. CUDA C pruža jednostavan rad za programere upoznate sa programskim jezikom C, i brzo izvršavanje na uređaju. CUDA C proširuje programski jezik C, omogućujući programeru da definiše C funkcije, poziva kernele koji se izvršavaju N puta paralelno na CUDA nitima, za razliku od običnih C funkcija.

Ovaj rad sadrži dizajn, metodologiju i implementaciju GPU baziranog 3D-FFT algoritma za dimenzije stepena dvojke. Globalni pristup memoriji je reduciran i reprezentacija podataka je poboljšana da koristi spajanje memorije. Optimizirani 3D FFT algoritam doseže do 486 GFlop/s računa, što je 1.14x brže rezultata koji daje cuFFT. GPU baziran 3D-FFT algoritam autora, može da se koristi u raznim poljima primjene.

# Opis cilja

Opišimo ciljeve koje želimo postići pri izradi rada na temu, "Paralelizacija algoritma brze Fourierove transformacije na GPU".

Prvi cilj, koji bi bilo idealno postići u sklopu ovog rada, jeste postići ubrzanje algoritma brze Fourierove transformacije, čije su teoretske postavke prikazane u prvom poglavlju ili barem postići komparabilne rezultate, te uputiti na dalja unaprijeđenja.

Drugi cilj je bio da uporedimo CUDA i OpenCL platforme za paralelno računanje i zaključimo koja platforma daje veći broj mogućnosti i više funkcionalnosti za implementaciju paralelizacije algoritma brze Fourierove transformacije.

Treći cilj je bio da uporedimo CPU i GPU arhitekturu i na taj način zaključimo koja arhitektura ima veći potencijal za implementaciju algoritma brze Fourierove transformacije, radi paralelizacije i postizanja što većih performansi.

Četvri cilj je dokumentovanje rezultata na način usporedbe sa benchmark programima, ukoliko postoje standardni, po kojima bi dokumentovali rezultate, te testiranje najlošijih, srednjih i najboljih vremena izvršavanja.

# Literatura


[1] Meiyappan, S., "Implementation and performance evaluation of parallel fft algorithms", School of Computing. National University of Singapore, Singapore, 2003.

[2] Tay, R., OpenCL parallel programming development cookbook. Packt Publishing Birmingham, UK, 2013.

[3] "OpenCL Programming flow", dostupno na: https://new-final.readthedocs.io/en/latest/Programming_Guides/Opencl-programming-guide.html

[4] "OpenCL : Platform model and Memory hierarchy", dostupno na: https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

[5] Guide, "Cuda c++ programming guide", NVIDIA, July, Vol. 29, 2013.

[6] "Compare benefits of cpus, gpus, and fpgas for different oneapi compute workloads", dostupno na: https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html#gs.geft64

[7] Demaine, E. D., Leiserson, C. E., Lee, W. S., "6.046 j/18.410 j introduction to algorithms, fall 2001", 2001.

[8] Farhan, M., "Glance on parallelization of fft algorithms", Land Forces Academy Review, Vol. 24, No. 1, 2019.

[9] Liu, B., "Parallel Fast Fourier Transform", dostupno na: https://cs.wmich.edu/gupta/teaching/cs5260/5260Sp15web/studentProjects/tiba&hussein/03278999.pdf

[10] Brigham, E. O., The fast Fourier transform and its applications. Prentice-Hall, Inc., 1988.

[11] Puchała, D., Stokfiszewski, K., Yatsymirskyy, M., Szczepaniak, B., "Effectiveness of fast fourier transform implementations on gpu and cpu", in 2015 16th International Conference on Computational Problems of Electrical Engineering (CPEE). IEEE, 2015, str. 162–164.

[12] "OpenCL Specification", dostupno na: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[13] Sanders, J., Kandrot, E., CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

[14] "Single instruction, multiple threads", dostupno na: https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads

[15] Haque, M. N., Uddin, M. S., "Accelerating fast fourier transformation for image processing using graphics processing unit", 2011.

[16] Li, J. S., "Fourier transform based gpu acceleration", in Applied Mechanics and Materials, Vol. 347. Trans Tech Publ, 2013, str. 2926–2929.

[17] Govindaraju, N. K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J., "High performance discrete fourier transforms on graphics processors", in SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Ieee, 2008, str. 1–12.

[18] Mejia-Parra, D., Arbelaiz, A., Ruiz-Salguero, O., Lalinde-Pulido, J., Moreno, A., Posada, J., "Fast simulation of laser heating processes on thin metal plates with fft using cpu/gpu hardware", Applied Sciences, Vol. 10, No. 9, 2020, str. 3281.

[19] Keskin, S., Erdil, E., Kocak, T., "An efficient parallel implementation of 3d-fft on gpu", in Proc. IEEE High Perform. Extreme Comput. Conf., 2017, str. 1–4.

# Indeks pojmova