Heterogeno programiranje u jeziku C++

Crnković, Matej

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:369038

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

MATEJ CRNKOVIĆ

HETEROGENO PROGRAMIRANJE U JEZIKU C++

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, studeni, 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Zahvaljujem se svom mentoru, prof. dr. sc. Mladenu Juraku, na korisnim savjetima i naputcima vezanim uz ovaj rad. Također, zahvaljujem mu na izvrsnim kolegijima tijekom fakultetskog obrazovanja koji su me zainteresirali za područje ovog rada.

Zahvaljujem se Ani na lektoriranju i dragocjenim savjetima, kao i Vlatku na dobronamjernim opaskama.

Zahvaljujem se prijateljima s fakulteta, a posebno Lovri, Matiji, Mihovilu i Ivki, na pomoći, dobroj atmosferi i brojnim uspomenama.

Veliko hvala svim profesorima tijekom mog obrazovanja na njihovom trudu i znanju koje su mi prenijeli. Također, zahvaljujem svim izviđačkim prijateljima uz koje sam naučio mnoge vrijednosti koje se ne mogu naučiti u školi. BP.

Ovaj rad posvećujem svojoj obitelji — mami Mandi, ocu Zdravku i bratu Vlatku, koji su mi velika podrška u dobrim trenucima, a još veća u teškim. Hvala im na bezuvjetnoj ljubavi i vjeri u mene, koja mi je uvijek bila velika snaga.

Najveće hvala i najveću posvetu upućujem Ani. Hvala ti što si bila uz mene svaki korak ovog puta, slavila sa mnom sve položene ispite i tješila me kada stvari nisu išle kako sam želio. Hvala ti.

Sadržaj

Uvod

Ubrzani razvoj tehnologije i sve veće potrebe za računalnim resursima dovele su do značajnog povećanja važnosti visokoučinkovitog računarstva (*High-Performance Computing*, HPC). HPC sustavi, poput superračunala i računalskih klastera, podrazumijevaju spajanje više računalnih resursa u visokoperformantne sustave. Oni omogućuju rješavanje kompleksnih problema u područjima kao što su znanstvene simulacije, analiza vremenskih modela, istraživanje novih materijala i razvoj lijekova. Osim toga, sve veća primjena strojnog učenja i neuronskih mreža za zadatke poput prepoznavanja slika, analize jezika i autonomnih sustava također zahtijeva iznimnu računalnu snagu.

Ključnu ulogu u postizanju visoke performanse u ovim područjima imaju heterogeni sustavi, koji kombiniraju različite vrste procesorskih jedinica poput CPU-a (Central Processing Unit), GPU-a (Graphics Processing Unit) i specijaliziranih akceleratora, čime se omogućuje učinkovita upotreba dostupnih resursa.

Na primjer, algoritmi za simulaciju dinamike fluida (*Computational Fluid Dynamics*, CFD) koriste GPU akceleraciju za paralelne proračune, dok se treniranje neuronskih mreža često oslanja na specijalizirane akceleratore poput TPU (Tensor Processing Unit) i LPU-a (Language Processing Unit). Sličan pristup koristi se u područjima kao što su genomska istraživanja, gdje akceleratori omogućuju brzu analizu velikih bioloških skupova podataka, kao i simulacijama molekularne dinamike u biotehnologiji i kemiji, gdje se paralelizacija na GPU-ima ubrzava računske procese za modeliranje ponašanja molekula.

S obzirom na rastuću upotrebu GPU-a za opće računanje, poznatu kao GPGPU (General-Purpose Computing on Graphics Processing Units), sve je veća potreba za standardima koji omogućuju jednostavno korištenje tih akceleratora u različitim aplikacijama. GPGPU omogućava upotrebu GPU-a ne samo za grafiku, već i za opće računanje, čime se ubrzava širok spektar aplikacija, od znanstvenih simulacija do strojnog učenja i analize velikih podataka.

Jezik C++ sa svojom modernom sintaksom i podrškom za otvorene standarde donosi značajne prednosti u razvoju prijenosivih aplikacija koje ciljaju različite akceleratore, osiguravajući unificirani pristup i smanjenje kompleksnosti. Iako C++ omogućuje visoku apstrakciju, također nudi snažnu kontrolu nad hardverskim resursima, što programerima omogućuje optimizaciju performansi na različitim vrstama hardverskih platformi.

Ova kombinacija visoke apstrakcije i fine kontrole nad hardverom čini C++ moćnim alatom za razvoj visokoučinkovitih aplikacija u heterogenim okruženjima.

Ovaj rad istražuje osnove heterogenog programiranja, ključne koncepte **SYCL**-a kao otvorenog standarda za heterogeno programiranje u jeziku C++, te načine njegove implementacije putem *oneAPI DPC++* kompajlera.

Poglavlje 1

Heterogeno programiranje

U ovom poglavlju dan je pregled homogenih i heterogenih računalnih sustava. Također, dan je kratki pregled temeljnih digitalnih sklopova koje koriste računalni sustavi te CPU-GPU heterogenih sustavima na kojima su u idućim poglavljima izučavani modeli programiranja. Na kraju poglavlja istaknuti su problemi koje donose heterogeni sustavi te je samim time motivirano iduće poglavlje.

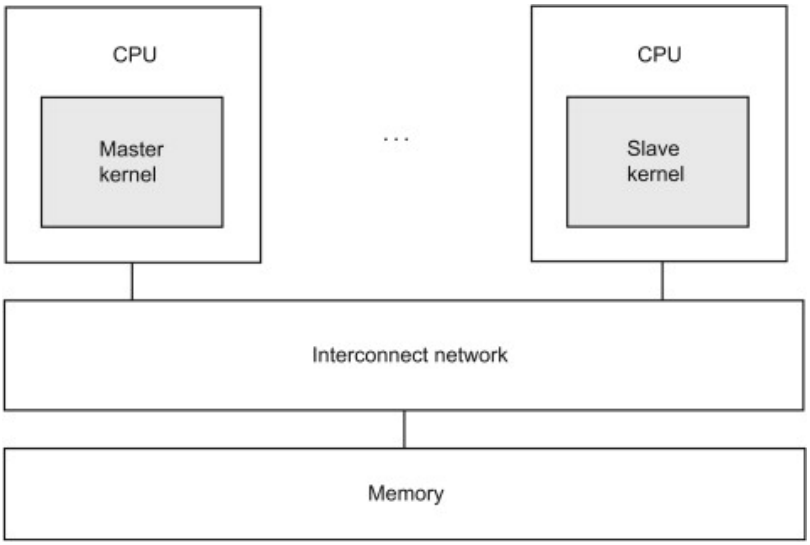
1.1 Heterogenost i homogenost u računarstvu

U modernom svijetu sve više rastu zahtjevi na računalne sustave koji podrazumijevaju visoke performanse, energetsku učinkovitost i obradu velike količine podataka.
Kako bi se navedeni zahtjevi ispunili većina današnjih računalnih sustava su multiprocesorski, višejezgreni ili kombinacija tog dvoga. Ukratko, multiprocesorski sustavi sačinjeni su od više računalnih jedinica dok su višejezgreni sustavi sačinjeni od više jezgara na istoj fizičkoj jedinici. I multiprocesorski i višejezgreni sustavi mogu biti homogeni ili heterogeni. Temelj ovog rada je programiranje na heterogenim sustavima, no za potpunost razumijevanja dan je i kratak uvid u homogene sustave.

1.2 Homogeni sustavi

Homogeni multiprocesorski sustavi generalno definiraju arhitekture sačinjene od više računalnih jedinica istog tipa (slika 1.1). U takvim sustavima računalne jedinice nisu specijalizirane za posebne zadatke već balansirano izvršavaju iste tipove zadataka. Samo neke od prednosti ovakvih sustava uključuju jednostavnost dizajna kao posljedice replikacije komponenti te jednostavnost programiranja i egzekucije, koja proizlazi iz mogućnosti izvršavanja programa na bilo kojoj komponenti uz gotovo jednake performanse izvršavanja.

Međutim, dodavanje sve većeg broja istih računalnih jedinica ubrzo prestaje donositi poboljšanja u performansama jer korisnici često ne mogu optimalno iskoristiti dodatne jedinice za svoje potrebe, a računalne jedinice opće namjene nisu dizajnirane da u svim uvjetima efikasno skaliraju s povećanjem broja istovrsnih komponenata.
Stoga se u praksi pojavila potreba za malo drugačijim računalnim sustavima koji se odmiču od klasične homogene paradigme.

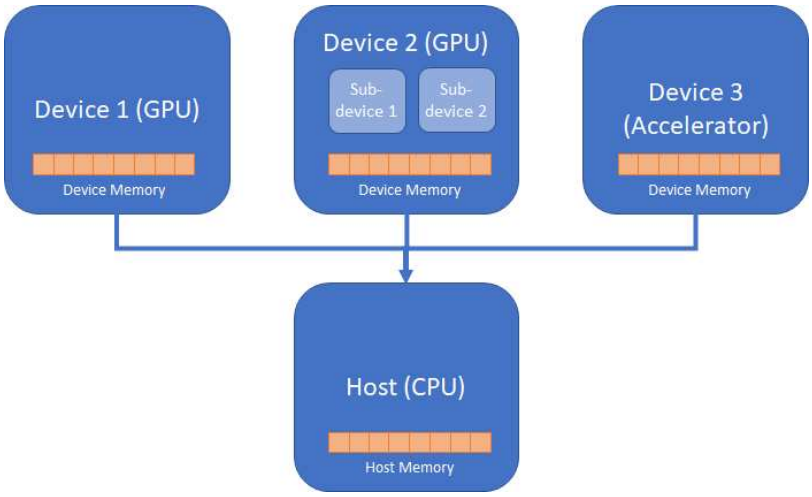


Slika 1.1: Primjer homogenog multiprocesorskog sustava koji implementira *master-slave* paradigmu [26].

1.3 Heterogeni sustavi

Fizička ograničenja u proizvodnji homogenih sustava te potrebe za ubrzanjem vrlo specijaliziranih računarskih zadataka dovela su do razvoja heterogenih računarskih sustava. Pojam heterogenosti u računarstvu obično se referira na paralelno postojanje različitih arhitektura procesora, konkretnije različitih ISA (instruction-set architectures) modela, gdje jedan procesor koristi jedan model, a drugi procesor koristi drugi vrlo različiti model.
Takvi sustavi donose poboljšanja u performansama i energetskoj učinkovitosti inkorporirajući različite računalne jedinice koje su specijalizirane za izvršavanje posebnih tipova zadataka.
U svojoj suštini **Heterogeni sustavi** označavaju računalne sustave koji su sačinjeni od dvije ili više različitih računalnih jedinica poput centralnih procesorskih jedinica (CPU), grafičkih kartica (GPU), FPGA jedinica ili AI-akceleratora poput TPU-a (Tensor Proce-

ssing Unit) i NPU-a (Neural Processing Unit) te njihove softverske podrške (slika 1.2). U nastavku rada isključivo će biti opisano programiranje na heterogenim sustavima koji su sačinjeni od jednog ili više CPU-a te jednog ili više grafičkih akceleratora (GPU-a).

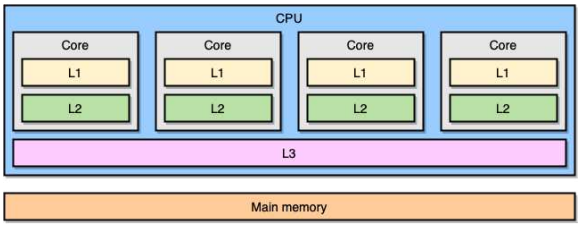


Slika 1.2: Primjer heterogenog multiprocesorskog sustava sačinjenog od jednog CPU domaćina, dva GPU uređaja i jednog dodatnog akceleratora [8]

1.4 CPU-GPU heterogeni sustavi

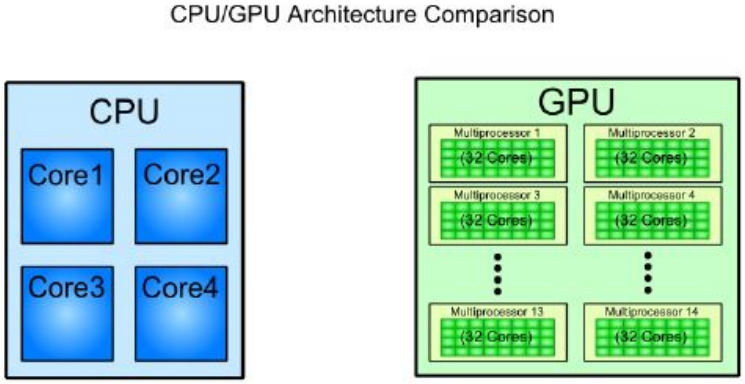
Kao što je spomenuto ranije, fokus ovog rada bit će primarno heterogeni sustavi koji su sačinjeni od barem jedne centralne procesorske jedinice (CPU) i barem jedne grafičke procesorske jedinice (GPU).

Centralne procesorske jedinice (*CPU*, centralni procesor, glavni procesor, procesor) je elektronički digitalni sklop koji izvršava osnovne operacije i upravlja komponentama računalnog sustava. CPU obično ima manji broj jezgi (2 - 64), no svaka jezgra može izvršavati vrlo složene sekvencijalne zadatke poput upravljanja podacima, logičkih i aritmetičkih operacija. Jezgre modernih procesora obično imaju dva mala hardverska dijela memorija zvana L1 i L2 *cache* koji su smješteni vrlo blizu jezgri te L3 *cache* (ponekad i L4) kojeg jezgre međusobno dijele (slika 1.3). Hardverski *cache*-ovi čuvaju podatke s često posjećenih memorijskih lokacija kako bi se smanjila latencija i ubrzale operacija nad podacima.



Slika 1.3: Memorijska shema četverojezgrene centralne procesorske jedinice [21]

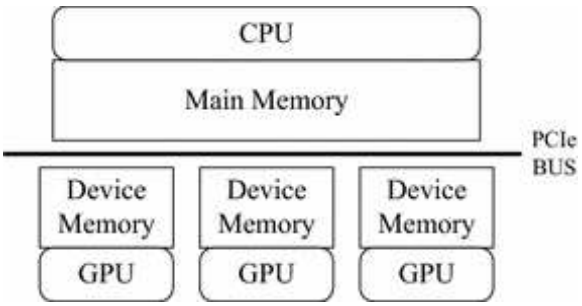
Grafičke procesorske jedinice (*GPU*) specijalizirani su digitalni sklopovi koji se koriste za paralelnu obradu velike količine podataka. Inicijalno su se koristile u grafičke svrhe poput renderiranja grafike za video igre, no u moderno vrijeme imaju veliku primjenu na svim područjima koji zahtjevaju visoku razinu paralelizacije poput strojnog učenja. GPU ima velik broj jezgri (do nekoliko tisuća), no za razliku od jezgara CPU-ova, jezgre na grafičkim procesorima su jednostavne odnosno mogu izvršavati jednostavne zadatke (slika 1.4). Jezgre grafičkih kartica obično izvode aritmetičke operacije poput zbrajanja, množenja, dijeljenja itd. te bitne operacije poput AND, OR, XOR operacija. Međutim, korist GPU-a je upravo u tome što se jednostavnije računske operacije mogu izvršavati u potpunoj paralelnosti na nekoliko tisuća podataka što uvelike ubrzava složenije operacije poput množenja vektora i matrica koje je osnova za izradu grafike, primjenu grafičkih filtera, velike znanstvene proračune, fizikalne simulacije itd.



Slika 1.4: Usporedba shema CPU-a i GPU-a [7]

U CPU-GPU heterogenim sustavima važno je shvatiti uloge **domaćina** (eng. *host*) i **uređaja** (eng. *device*).
Iako razna literatura navodi malo drugačije formalne definicije pogotovo u sustavima koji imaju nekoliko CPU-a, u radu će biti objedinjene sljedećim definicijama.

Domaćin (eng. *host*) je glavna računarska jedinica koja upravlja cijelim sustavom, inicira izvršavanja te raspoređuje zadatke među ostalim računarskim jedinicama unutar sustava. Domaćin također upravlja I-O (ulazno-izlaznim, eng. *input-output*) operacijama te komunikacijom među komponentama sustava. Domaćin je najčešće CPU te se to u nastavku rada podrazumijeva.
Uređaj (eng. *device*) je računarska jedinica koja izvršava zadatke dodijeljene od strane domaćina. Takva jedinica je specijalizirana za akceleraciju te ima drugačiji ISA i arhitekturu od domaćina.
U sustavima koji imaju veći broj CPU-ova i dalje se samo jedan (glavni) CPU smatra domaćinom. Dodatni CPU-ovi obično služe kao podrška domaćinu koji na njih prebacuje dio svojih zadataka, ali se ne smatraju akceleratorima niti uređajima jer nemaju tehničke sposobnosti klasičnih akceleratora. U nekom vrlo apstraktnom pogledu, glavni CPU, zajedno sa svim dodatnim CPU-ovima na koje *offload*-a (rasterećuje) svoje zadatke, možemo smatrati domaćinom sustava.



Slika 1.5: Prikaz jednostavnog CPU-GPU heterogenog sustava sačinjenog od jednog CPU domaćina i tri GPU uređaja s ulogama akceleratora [16].

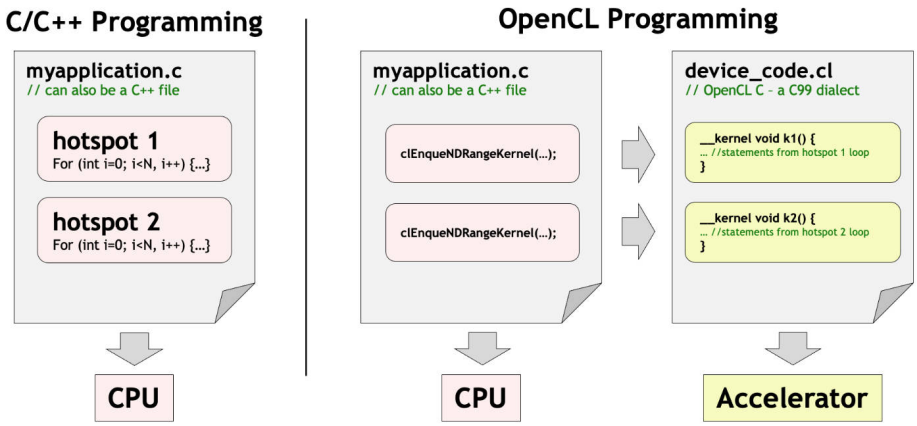
1.5 Problemi heterogenih sustava

Heterogeni sustavi sa sobom nose niz poteškoća i problema koji se ne pojavljuju u standardnim homogenim računalnim sustavima. Najveće poteškoće za razvojne inženjere heterogeni sustavi donose zbog fizičke različitosti računalnih jedinica. Na primjer, kako jedinice imaju različite ISA, one su binarno nekompatibilne, što znači da strojni kod za jednu arhitekturu ne može direktno biti izvršen na drugoj bez dodatnih prilagodbi.
Računalne jedinice mogu imati različite načine interpretacije memorije poput razlika u bajtnom redoslijedu (endianness), strukture *cache*-ova, pristupa memoriji i tako dalje. Navedeni problemi povećavaju opterećenje na razvojne inženjere zbog povećane složenosti programskog koda i smanjene portabilnosti jer računske jedinice zahtjevaju specifičan kod prilagođen konkretnom hardveru.

Dodatan teret donosi nužnost balansiranja radnog opterećenja aplikacije među jedinicama kada se složeni izračuni moraju podijeliti među različitim jezgrama s promjenjivim karakteristikama i performansama. U nastavku rada izložen je pregled standarda koji nudi potencijalna rješenja navedenih problema.

1.6 Heterogeno programiranje i prenosivost koda

Nakon upoznavanja s prednostima i manama heterogenih sustava, važno je upoznati se s načinom pisanja koda za aplikacije koje se izvršavaju na heterogenim sustavima te izazovima koje takvo programiranje nosi.
Budući da su heterogeni sustavi sačinjeni od različitih jedinica (u terminima ovog rada barem jednog CPU-a i jednog GPU-a), na niskom nivou inženjeri ustvari moraju pisati dva različita dijela koda, od kojih je jedan usmjeren na domaćina, a drugi na uređaj. Međutim, kod napisan na niskoj razini za neku jedinicu, recimo grafičku karticu, gotovo je uvijek zaključan isključivo na kartice istog proizvođača. Time se gubi jedna od temeljnih želja pri pisanju programa, a to je prenosivost i međukompatibilnost koda odnosno mogućnost da se isti kod izvršava na različitim sustavima.
Programski modeli i alati koji eksplicitno iskorištavaju *low-level* arhitekturalni dizajn grafičkih kartica su također često kompletno zaključani za kartice nekog od proizviđača. Konkretno, kao primjer ističu se **CUDA** platforma i API (*Application Programming Interface*), koji su kompletno zaključani za grafičke kartice proizvođača NVIDIA te ROCm za grafičke kartice proizvođača AMD.
Jedan od najpoznatijih razvojnih okvira koji je pokušao doskočiti problemu pisanja različitog koda za grafičke kartice je **OpenCl** (*Open Computing Language*). OpenCl standard razvijen je od strane **Khronos** grupe, dok proizvođači grafičkih kartica poput NVIDIA, Intel, AMD i ostalih implementiraju vlastite OpenCL drivere i OpenCL runetime koji osiguravaju da se generičke OpenCL instrukcije izvršavaju na konkretnim procesnim jedinicama. OpenCl u svojoj suštini koristi C programski jezik te zahtjeva od korisnika eksplicitno upravljanje memorijom, raspored zadataka i sinkronizaciju što daje veću kontrolu nad detaljima izvršenja, ali zahtjeva više koda (slika 1.6).



Slika 1.6: Usporedba C++ aplikacije koja se izvršava na CPU-u i OpenCL aplikacije koja dodatno sadrži i izvorni kod namijenjen za izvršavanje na GPU-u [25]

Postoji niz drugih *high-level* programskih modela koji se koriste kao sloj apstrakcije nad raznim platformama i omogućuju portabilnost koda među platformama. Ti slojevi su zamišljeni kao portabilna API proširenja C++ jezika koja koriste razne biblioteke, jezike i specifikacije hardvera za paralelnu egzekuciju i upravljanje strukturama podataka. Najpoznatiji primjeri su **Kokkos**, **Raja** i **SYCL**. Iako svi modeli imaju svoje prednosti i mane, u zadnjih nekoliko godina SYCL vrlo brzo raste u popularnosti, pokrivenosti raznih platformi te brojnosti fizičkih implementacija kompajlera. U nastavku rada bit će objašnjen isključivo SYCL standard uz poneke usporedbe s drugim modelima te eventualna direktna mapiranja određenih instrukcija na razne *backend*-e poput CUDE.

Poglavlje 2

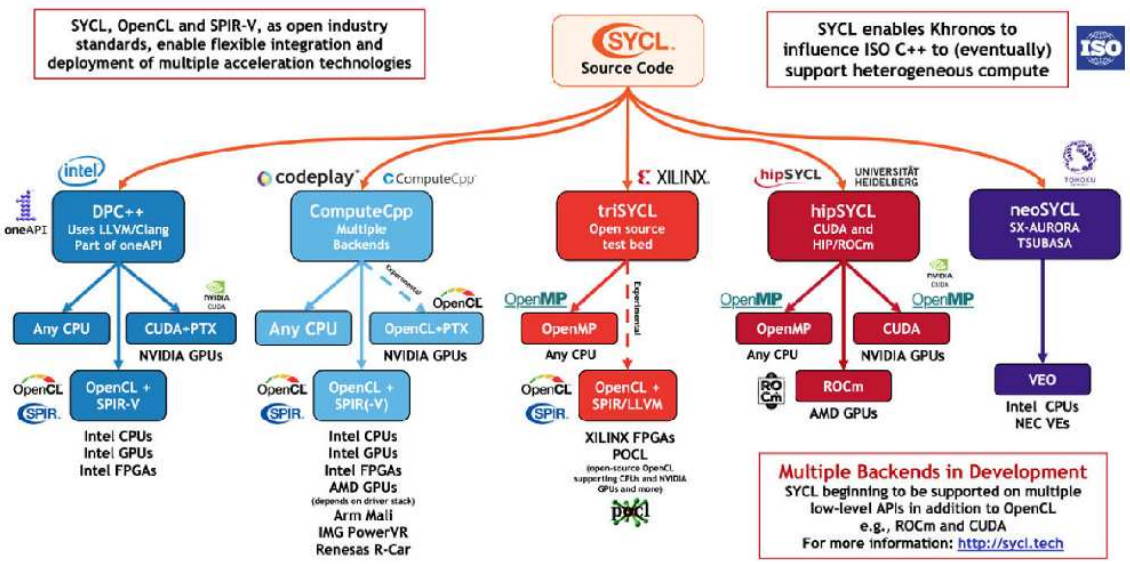
SYCL

U prethodnim su poglavljima samo spomenuti SYCL i njegove različite komponente, dok se u ovom poglavlju sustavno definiraju ključni pojmovi i koncepti SYCL-a te njegova arhitektura. Također se odgovara na pitanja poput: *Što je to sycl? Što su SYCL backend-i, kernel funkcije itd.*

2.1 SYCL standard

SYCL ("sikl") je besplatan, standardizirani apstrakcijski sloj za programiranje heterogenih i *offload* procesora koristeći moderni ISO C++ (C++17) [22]. SYCL je, poput OpenCL-a, razvijen od strane Khronos grupe s ciljem povećanja produktivnosti razvojnih inženjera na širokom spektru hardverskih akceleratora. SYCL standard definira *single-source* eDSL jezik (*embedded domain-specific language*) odnosno jezik koji nije opće namjene, već se koristi za specijalnu domenu i implementiran je poput biblioteke u svom jeziku domaćinu (u ovom slučaju C++). *Single-source* pristup omogućava razvoj koda za domaćina i uređaj unutar iste C++ izvorne datoteke.
SYCL je relativno nov standard, no postoji nekoliko kompajlera koji ga implementiraju. Najpoznatiji je **Data Parallel C++** (DPC++), open-source projekt pod okriljem Intel-a. DPC++ je ujedno prvi, i za sada jedini, kompajler koji je postigao potpunu implementaciju najnovijeg SYCL 2020 standarda [20].
Uz DPC++ svakako treba izdvojiti **AdaptiveCpp** (prije znan kao *hipSYCL*). AdaptiveCpp je *open-source comunitiy-driven* projekt pod vodstvom Sveučilišta Heidelberg u Njemačkoj. Još nije u potpunosti implementirao SYCL 2020 standard, no ipak donosi neke prednosti. Osim mogućnosti definiranih SYCL standardom, AdaptiveCpp podržava standarnu C++ paralelizaciju odnosno nudi podršku za automatski *offload* standardnih algoritama. Osim toga, AdaptiveCpp je trenutačno jedini SYCL kompajler koji podržava *single-pass* parsiranje koda za domaćina i uređaj [1].

Uz dva navedena kompajlera, razvijaju se još i *triSYCL*, koji je proširenje DPC++ baziran na C++ 20, *neoSYCL*, *Sylkan*, *Polygeist* i drugi (slika 2.1).
U nastavku se rad temelji isključivo na DPC++ kompajleru, budući da je jedini koji u potpunosti implementira aktualni SYCL 2020 standard.



Slika 2.1: Prikaz SYCL implementacija nekoliko radnih skupina [23]

2.2 SYCL arhitektura

Anatomija SYCL aplikacije

Prije razmatranja detalja SYCL arhitekture, prikazan je primjer (Listing 2.1) osnovne anatomije SYCL aplikacije te su opisani osnovni dijelovi programa i specifičnosti korištene za upravljanje podacima.

```
1 #include <iostream>
2 #include <sycl/sycl.hpp>
3 using namespace sycl; // (opcionalno) kako bi izbjegli navođenje "sycl
     ::" imeničkog prostora prije SYCL imena
4
5 int main() {
6   int data[1024]; // alociranje podataka s kojima se radi
7
8   // Kreiranje dodijeljenog reda za dodavanje zadataka na dodijeljeni
       uređaj
```

```
9   queue myQueue;
10
11  // Stavljajući sav specifičan SYCL posao u {} blok osigurava se
12  // da će se svi SYCL zadaci završiti prije nego izađemo iz scope-a
        bloka
13  // jer se tada poziva destruktor resultBuf buffera koji čeka izvršenje
        SYCL zadataka
14  {
15    // podaci spremljeni u data varijabli zamataju se u buffer
16    // buffer kao memorijski spremnik upravlja podacima između glavne
        memorije i uređaja
17    buffer<int, 1> resultBuf { data, range<1> { 1024 } };
18
19    // grupa naredbi stavlja se za red myQueue
20    myQueue.submit([&](handler& cgh) {
21      // stvara se aksesor za pisanje u resultBuf bez inicijalizacije
22      accessor writeResult { resultBuf, cgh, write_only, no_init };
23
24      // u red se stavlja parallel_for zadatak s 1024 radne jedinice
25      cgh.parallel_for(1024, [=](id<1> idx) {
26        // svaki element aksesora inicijalizira se s vrijednosti
            vlastitog indeksa
27        writeResult[idx] = idx;
28      }); // Kraj kernel (jezgrine) funkcije
29    });   // kraj grupe naredbi za red myQueue
30  }       // Kraj scope-a, čeka se završetak dovršetak svih zadataka nad
        spremnikom resultBuf
31
32  // Ispis rezultata rezultat
33  for (int i = 0; i < 1024; i++)
34    std::cout << "data[" << i << "] = " << data[i] << "\n";
35
36  return 0;
37 }
```

Listing 2.1: Prikaz jednostavne SYCL aplikacije

U liniji 2. uključene su SYCL datoteke zaglavlja koje osiguravaju korištenje svih SYCL značajki.
SYCL aplikacija izvršava se na SYCL platformi i strukturirana je u tri područja koja specificiraju :*kernel scope*, *command group scope* i *application scope* [22].

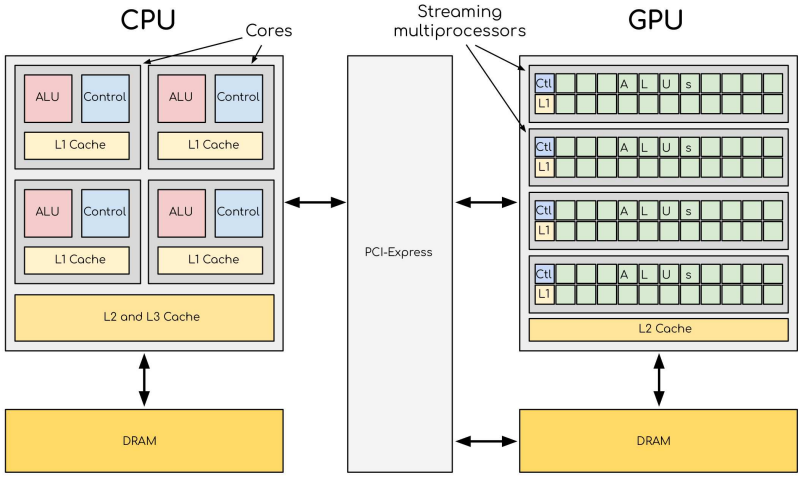
Kernel scope, odnosno područje jezgre, specificira jednu jezgrinu funkciju, *kernel*, koja će biti kompajlirana od strane kompajlera uređaja i u konačnici izvršena na samom uređaju. U navedenom primjeru područje jezgrine funkcije definirano je u linijama 26. i 27.

Command group scope, odnosno područje zapovijedne grupe, specificira radnu jedinicu koja je sačinjena od SYCL jezgrine funkcije i aksesora (*accessors*). U gornjem primjeru područje zapovjedne grupe definirano je linijama 21.-29. U liniji 22. pojavljuje se ***sycl::accessor*** - klasa koja regulira pristup podacima koji su zamotani u *buffer*. Preko *aksesora*, zadaci u redu uređaja pristupaju podacima koje čuva buffer. Također, preko *aksesora* SYCL jezgrine funkcije pristupaju memoriji domaćina. U primjeru, konkretno u liniji 27., pomoću aksesora se piše u *buffer* podataka.

Application scope, odnosno područje aplikacije, označava sav ostali kod izvan područja zapovjedne grupe.

Memorijski model

Osnovna prednost SYCL-a je visok nivo apstrakcije za korisnika i *single-source* pristup za kod koji se izvršava na domaćinu i na uređaju. To povlači nužnost postojanja efektivnog memorijskog modela koji upravlja i sinkronizira podatke među domaćinom i uređajima. Dodatno, kako se jezgrine funkcije pišu u čistom C++ kodu, nužno je da, za vrijeme izvršavanja aplikacije, SYCL okruženje osigura da su podaci dostupni za izvršavanje na jezgrama uređaja po pravilima i konstruirani onako kako to propisuju implementacije standarda za konkretne uređaje.
U konkretnom slučaju heterogenog sustava s CPU domaćinom i GPU kao uređajem, komunikacija, odnosno transfer podataka, teče kroz PCI sabirnicu (slika 2.2). Isti proces zna uzrokovati 'usko grlo' u izvršavanju aplikacija jer se podaci moraju kopirati s domaćina na uređaj, tamo transformirati i naposlijetku ponovno vratiti na domaćina.



Slika 2.2: Komunikacija CPU-a i GPU-a preko PCI sabirnice [6]

Za upravljanje memorijom SYCL koristi tri glavna pristupa [19]:

Buffer objekti

sycl::buffer je generička klasa koja služi za upravljanje podacima između SYCL C++ aplikacije na domaćinu i SYCL jezgri koje se izvršavaju na uređaju. *Buffer* klasa (zajedno s *accessor*-om) upravlja prijenosom podataka i garantira konzistentnost podataka među različitim jezgrinim funkcijama koje se trebaju izvršavati na istim podacima. Svojim životnim vijekom *buffer* objekti definiraju koliko dugo i kako SYCL *runtime* upravlja memorijskim alokacijama za dane podatke na domaćinu i uređaju.
Objekt tipa *sycl::buffer* ne mapira se nužno u samo jedan memorijski objekat na uređaju, već SYCL *runtime* može mapirati *buffer* na nekoliko memorijskih objekata unutar iste zapovjedne grupe na specifičnom uređaju.
Konstruktori klase *sycl::buffer* generiraju dijeljena jedno-, dvo- ili trodimenzionalna polja koje mogu koristiti jezgrine funkcije. Neka preopterećenja konstruktora inicijaliziraju podatke u *buffer*-u kopirajući ih s domaćina dok drugi ne kopiraju podatke već inicijaliziraju *buffer* dodijeljenim vrijednostima (ako takva postoje) ili samo alociraju 'čistu' memoriju ukoliko ne postoje dodijeljene vrijednosti za konkretan tip memorije.
Budući da je *sycl::buffer* dijeljeno polje, njegovo ponašanje u vezi životnog vijeka podsjeća na *std::shared_ptr*.
Naime, *buffer* bilježi broj uređajskih podataka koji se referenciraju na njega i ne može se uništiti dok se ne unište svi mapirani uređajski podaci odnosno dok svi ne zapišu svoj rezultat natrag u *buffer* i 'otkače' svoju zavisnost na njega. Takvo blokiranje destruktora, kada *buffer* izlazi iz opsega, dovodi do željenog ponašanja unutar asinkronog izvršavanja zadataka u redu uređaja. Naime, nije dozvoljen nastavak glavnog programa dok nisu sinkronizirani svi podaci na domaćinu koji su povezani s *buffer*-om. Kako bi se u potpunosti iskoristili učinci ovog RAII koncepta, *buffer* objekti često se definiraju u najmanjem mogućem opsegu u kojem trebaju postojati (zamatajući taj opseg s *{}*) kako bi korisnici implicitno upravljali pozivom destruktora).
Iako postoji velik broj preopterećenja konstruktora za *sycl::buffer*, generalno se *buffer* konstruira tako da se specificira njegova veličina u memorijskom smislu i memoriju u koju *buffer* gleda.
Objekt klase *sycl::buffer* ustvari nema vlasništvo nad memorijom nego nudi ograničeni pogled u memoriju. Iz navedenih razloga memorija za *buffer* se može inicijalizirati samo iz trivijalno kopirajućih postojećih objekata (eng. *TriviallyCopyable* - C++11). Također, memoriji u *buffer*-u se ne pristupa direktno, kao pomoću *getter* i *setter* funkcija, već se koriste *sycl::accessor* objekti za definiranje pristupa.

Idući primjer (Listing 2.2) prikazuje osnovan način rada s *buffer*-ima, a detalji su objašnjeni u narednim poglavljima.

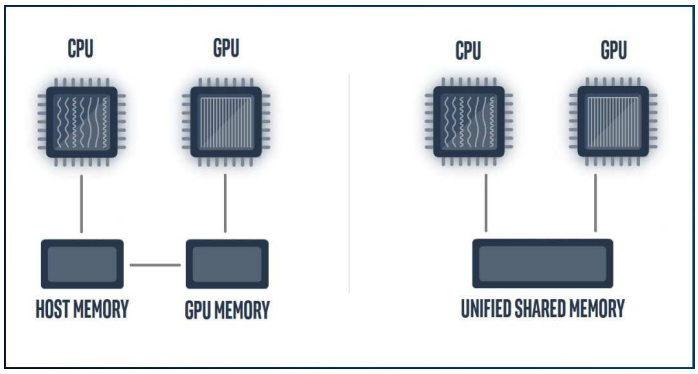
```
...
constexpr size_t N = 1024;
std::vector<int> host_data(N, 1);
sycl::queue queue;
...
// kreiranje buffer za prijenos podataka između domaćina i uređaja
// buffer u konstruktoru prima podatke s domaćina te range objekat kojime se definira broj
// radnih jedinica koje će se izvršavati paralelno odnosno paralelno pristupati podacima
{
    sycl::buffer<int> data_buffer(host_data.data(), sycl::range<1>(N));

    // slanje zadataka u SYCL red
    queue.submit([&](sycl::handler& h) {
        // kreiranje accessor-a za pristup bufferu na uređaju
        // s sycl::write_only definirano je da se kroz accessor-a može samo pisati u buffer, a s
        // sycl::no_init se izbjegava nepotrebna inicijalizacija buffera
        sycl::accessor data_accessor(data_buffer, h, sycl::write_only, sycl::no_init);

        // definira se rad na uređaju pomoću lambda funkcije
        // idx označava ID radne jedinice, a pomoću idx[0] se dobija numerički ID same jedinice
        h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> idx)
        {
            data_accessor[idx] = idx[0] * 2;
        });
    });
    // opseg završava ovdje i čeka da podaci budu sinkronizirani natrag u host_data na domaćina
}
...
```

Listing 2.2: Prikaz osnovnog toka rada s *buffer* objektima

USM

Unified Shared Memory (USM) je alternativni SYCL model upravljanja memorijom koji se temelji na pokazivačima. USM je izgrađen snažno se oslanjajući na **Unificirano adresiranje** (*Unified Addressing*) - koncept koji omogućava korištenje jednog adresnog prostora (*unified address space*) koji obuhvaća domaćina i uređaje (Slika 2.3).

U tom adresnom prostoru svi pokazivači pokazuju na konzistentne memorijske lokacije, što znači da isti pokazivač referencira istu lokaciju u memoriji na svim uređajima.



Slika 2.3: Usporedba heterogenog sustava s odvojenim memorijama domaćina i uređaja te onog koji koristi USM [14]

USM model nije dostupan za sve platforme i sve uređaje jer nemaju svi podršku za virtualni zajednički adresni prostor, no u situacijama kada je dostupan pruža jaku mogućnost alokacije memorije u dijeljenom prostoru.
Kroz USM moguća su tri tipa alokacije memorije opisana u tablici 2.1:

Tablica 2.1: Tipovi USM alokacija

Tip USM alokacije	Opis
host	Alokacije u memoriji domaćina koje su dostupne i uređaju
device	Alokacije u memoriji uređaja koje nisu dostupne domaćinu
shared	Alokacije u dijeljenoj memoriji koje su dostupne domaćinu i uređaju

Kroz **alokacije na domaćinu** (*host allocations*) uređaji mogu direktno čitati i pisati u domaćinsku memoriju unutar svojih jezgrinih funkcija. Količina memorije koja se može alocirati na domaćinu je generalno ograničena količinom *pinnable* memorije u sustavu. Iako uređaj može direktno pisati u memoriju domaćina, ona je cijelo vrijeme u vlasništvu domaćina, odnosno ne kopira se na uređaj.
Iz tog razloga korisnici trebaju osigurati pravilnu sinkronizaciju pristupa memoriji alociranoj na domaćinu, kako bi se spriječili konflikti između operacija koje se izvode na domaćinu i onih koje se izvode unutar jezgri.

Alokacije na uređaju (*device allocations*) se koriste za eksplicitno upravljanje memorijom na uređajima. Memorija alocirana na uređaju nije direktno dostupna domaćinu, već se potrebni podaci moraju eksplicitno kopirati između domaćina i uređaja. Ipak, zbog očuvanja svojstva konzistentnosti unificiranog adresnog prostora, vrijednosti pokazivača na alociranu memoriju moraju ostati konzistentne između uređaja i domaćina. Konkretno, vrijednost pokazivača dobivenog alokacijom na uređaju uvijek će pokazivati na unikatnu memoriju i nije moguće da na domaćinu postoji pokazivač s istom vrijednosti koji pokazuje na neku drugu memoriju izvan unificiranog adresnog prostora. Količina memorije koja se može alocirati ovakvim putem je obično ograničena samo ukupnom dostupnom memorijom na uređaju. Primjer SYCL programa koji inicijalizira polje koristeći USM alokaciju memorije na uređaju dan je u primjeru Listing 2.3.

```
...
#include <sycl/sycl.hpp>
#include <iostream>

int main() {
    sycl::queue myQueue;

    // alociranje memorije na uređaju koja je vezana za kontekst
       asociran s redom
    int *data = sycl::malloc_device<int>(1024, myQueue);

    //jednostavna jezgrina funkcija
    myQueue.parallel_for(1024, [=](sycl::id<1> idx) {
        data[idx] = idx;
    });

    // nužno je eksplicitno čekati završetak jezgrine funkcije kako ne
       postoji aksesor
    myQueue.wait();


    int hostData[1024];
    // Kopiranje memorije s uređaja na domaćina
    myQueue.memcpy(hostData, data, 1024 * sizeof(int));
    myQueue.wait();

    sycl::free(data, myQueue);

    for (int i = 0; i < 1024; i++)
    std::cout << "hostData[" << i << "] = " << hostData[i] << "\n";
...
}
```

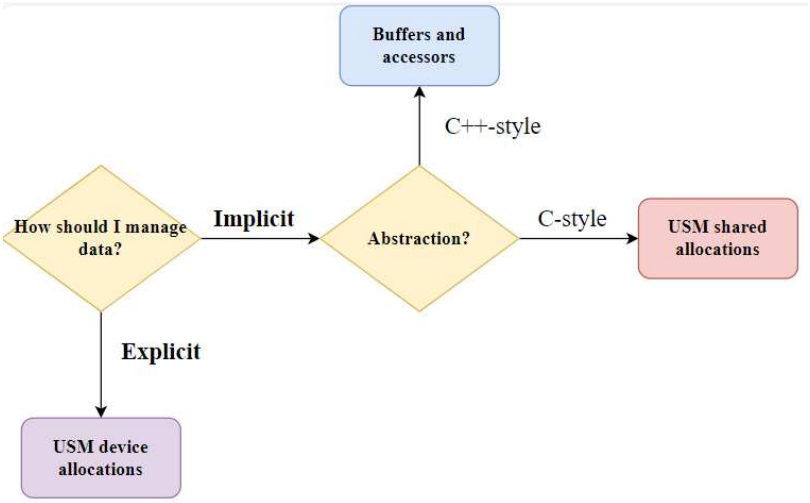
Listing 2.3: SYCL aplikacija koja koristi USM alokaciju memorije na uređaju

Alokacije u dijeljenoj memoriji (*shared allocations*) stvara memoriju koja se implicitno dijeli između domaćina i uređaja. Za razliku od primjera alokacije na uređaju, u narednom primjeru (Listing 2.4) nije potrebno kopirati eksplicitno podatke nazad na domaćina jer su oni (implicitno) dostupni tamo gdje se koriste.
SYCL runtime automatski upravlja kopiranjem podataka između domaćina i uređaja tamo gdje je to potrebno.

```
...
#include <sycl/sycl.hpp>

#include <iostream>

int main() {
    sycl::queue myQueue;

    int *data = sycl::malloc_shared<int>(1024, myQueue);

    myQueue.parallel_for(1024, [=](sycl::id<1> idx)
    {
        data[idx] = idx;
    });
    myQueue.wait();

    for (int i = 0; i < 1024; i++)
        std::cout << "data[" << i << "] = " << data[i] << "\n";

    sycl::free(data, myQueue);
}
```

Listing 2.4: SYCL aplikacija koristi USM alokaciju memorije u dijeljenoj memoriji domaćina i uređaja

Ne postoji egzaktno pravilo kada se treba koristiti USM, a kada *buffer*-i. Odabir obično ovisi o željenoj strategiji - želi li korisnik samostalno kontrolirati transfer podataka ili želi dopustiti SYCL izvršenju implicitno prebacivanje potrebne memorije. Također, SYCL se često uvodi u već postojeći projekt, gdje je USM, baziran na pokazivačima, vjerojatno bolji izbor jer smanjuje količinu programskog koda koji se treba prilagoditi *buffer* modelu (Slika 2.4).



Slika 2.4: Praktično pravilo za odabir memorijskog modela između USM i *buffer* paradigme [5]

IMAGES

Vrlo slično *buffer* memorijskom modelu, u SYLC-u je moguće baratati slikama koristeći integrirani *images* model. Ovaj model uvodi dvije temeljne klase : ***sycl::unsampled_image*** i ***sycl::sampled_image*** koje predstavljaju dijeljenu memoriju s dodatnom podrškom za slike koja može koristiti aksesore u jezgrinim funkcijama kao i *buffer*-a.
Životni vijek i sinkronizacija slijede pravila kao kod *buffer*-a, no navedene klase pružaju dodatnu podršku za rad na slikama kao i njihovo uzorkovanje što je posebno korisno za rad s grafikom na grafičkim akceleratorima.
sycl::unsampled_image najčešće se koriste za rad sa slikama u sirovom formatu koje nije potrebno dalje uzorkovati te navedena klasa nema podršku za filtriranje i uzorkovanje, ali je rad s podacima često brži.
sycl::sampled_image se koristi za rad sa slikama koje se uzorkuju te sama klasa pruža podršku za razna filtriranja i obradu slika.

2.3 Istaknute SYCL klase

Ovo poglavlje daje pregled eksplicitnih i implicitnih klasa koje pruža SYCL API, a svojom funkcionalnošću tvore jezgru SYCL modela [22]. Navode se klase koje su samo površno, ili nisu uopće, objašnjene u ostalim dijelovima rada.

sycl::context

Klasa koja predstavlja SYCL kontekst - reprezentaciju *runtime* objekata i njihovih stanja koje zahtjeva SYCL API za komunikaciju s jednim ili više uređaja asociranih s konkretnom platformom. Objekt klase sycl::context može se konstruirati koristeći jedan od mnogo preopterećenih konstruktora primajući objekte poput sycl::device, selektora uređaja, platforme ili liste uređaja. Kod kreiranja sycl::context objekta iz liste sycl::device uređaja nužno je da svi uređaji pripadaju istoj platformi. SYCL kompajler često implicitno supstituira syc::context bez korisnikove intervencije na mjestima gdje je on nužan (kao na primjer kod kreiranja dodijeljenog reda sycl::queue).

sycl::device

Klasa sycl::device enkapsulira jedinstveni SYCL uređaj na kojem se mogu izvršavati SYCL jezgrine funkcije odnosno sycl::kernel. Objekt klase sycl::device kreira se koristeći dodijeljeni konstruktor ili konstruktor koji prima selektor uređaja. Sami objekti služe za dohvat informacija o selektiranom uređaju kao i sučelje ostalim elementima SYCL aplikacije (poput sycl::queue) za komunikaciju s uređajem.

sycl::event

Objekti klase sycl::event predstavljaju status operacije koju izvršava SYCL runtime. Preko sycl::event objekata dohvaćaju se informacije poput vremena izvršavanja zadataka na uređaju, a također služi i domaćinskom programu za sinkronizaciju s uređajem.

sycl::handler

Klasa sycl::handler služi kao sučelje kroz koje se svaki zadatak u zapovijednoj grupi šalje u red uređaja (sycl::queue). Također, pruža sučelje za pristup podacima unutar zapovijedne grupe. Instancu klase sycl::handler može stvoriti isključivo SYCL runtime, a onemogućeno je svako kopiranje ili *move*-anje sycl::handler objekta.

Prilikom kreiranja zapovijedne grupe, SYCL kompajler automatski supstituira objekt klase sycl::handler. Svi aksesori (sycl::accessor) unutar zapovijedne grupe primaju instancu sycl::handler kao parametar, a sve jezgrine funkcije koje se pozivaju u zapovijednoj grupi su metode članice ove klase.

sycl::kernel

Ova klasa služi kao apstrakcija jezgre (*kernel*) i nema javnih konstruktora, no nudi nekoliko metoda za dohvaćanje informacija o jezgri poput asociranog konteksta.

Sve funkcije SYCL koje se izvršavaju na uređaju zovu se SYCL jezgrine funkcije (*SYCL kernel functions*), a kompajlirana SYCL jezgrina funkcija, zajedno s ostalim funkcijama koje su potrebne za njezino izvršavanje na uređaju, tvore jezgru. Jezgra se implicitno stvara kada se SYCL jezgrina funkcija, preko naredbe za pozivanje jezgri, preda uređajskom redu na izvršavanje. Naredbe za pozivanje jezgri, poput *singe_task* i *parallel_for*, su metode članice sycl::handler klase.

sycl::queue

Klasa sycl::queue služi za povezivanje domaćinskog koda s točno jednim uređajem. Pomoću sycl::queue objekata domaćin inicira izvršavanje zadataka na uređaju te prati završetak rada navedenih zadataka. Zadatci se iniciraju slanjem zapovijedne grupe sycl::queue objektu od strane domaćina, a domaćin čeka na završetak izvršavanja zadataka pozivajući *wait* ili *wait_and_throw* metode na sycl::queue objektu.
U slučaju predaje kratkih zadataka na izvršenje uređaju, mogu se koristiti prečac funkcije (*shortcut functions*) sycl::queue-a. Njihovim korištenjem pojednostavljuje se generiranje zapovijedne grupe bez eksplicitnog korištenja sycl::handler objekta. Pozivanjem neke od prečac funkcija implicitno se generira zapovijedna grupa s jednom naredbom, na odgovarajućem sycl::handler objektu se pozove odgovarajuća metoda te se zapovijedna grupa preda u red izvršavanja na uređaj. Prečac funkcija ima nekoliko, a najkorištenije su *single_task*, *parallel_for*, *memcpy*, *cpy* i *fill*. Najveća razlika u korištenju prečac funkcija i pozivanju funkcija iz sycl::handler-a, izuzev pojednostavljenog zapisa, je u povratnim vrijednostima funkcija. Prečac funkcije vraćaju sycl::event objekt, dok metode sycl::handler-a vraćaju *void*.

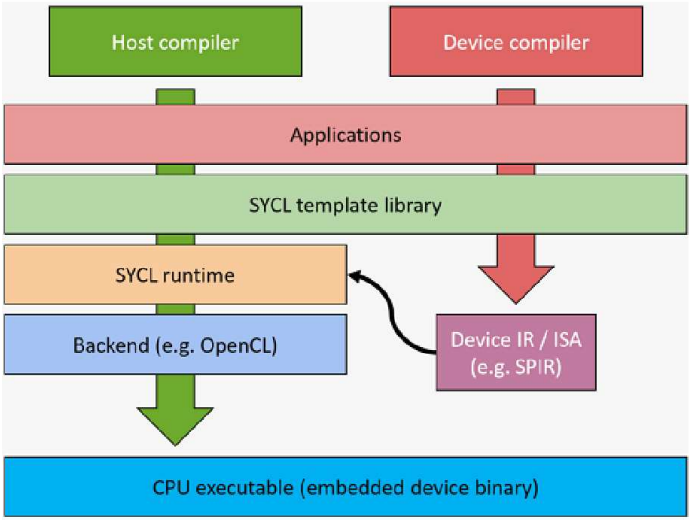
2.4 Karakteristike

Kompatibilnost sa standardnim C++ jezikom

SYCL je dizajniran da bude što bliži standardnom C++ jeziku. U praksi, to znači da sve dok nema posebne ovisnosti o SYCL integraciji s osnovnim implementacijama, standardni C++ kompajler može kompilirati SYCL programe i omogućiti njihovo izvršavanje na CPU-u domaćina. SYCL može biti implementiran i kao čista biblioteka, što znači da čak nije potrebno posebno kompajlersko okruženje, no tada će se svi zadaci izvršavati na domaćinu.
Također, SYCL koristi napredne tehnike C++ jezika poput generičkog programiranja, predložaka, nasljeđivanja itd. što omogućuje razvoj modernih generičkih biblioteka za izvršavanje na heterogenim sustavima.

SMCP

SYCL implementira **SMCP** (*single-source multiple compiler-passes*) tehniku kompajliranja. Ova tehnika omogućava da se izvorni kod za domaćina i uređaje ne mora čuvati u različitim datotekama, već se sav kod nalazi unutar jedinstvenog izvornog dokumenta. Različiti kompajleri prolaze kroz izvorni kod, parsiraju ga te generiraju izvršni kod za svoju radnu jedinicu. Standardni C++ kompajler generira kod za domaćina dok SYCL uređajni kompajler prolazi kroz izvorni kod, pronalazi dijelove koji su vezani za uređaj, parsira ih i stvara izvršni kod koji će se izvršavati na uređaju (Slika 2.5).



Slika 2.5: Prikaz paralelnog kompajliranja domaćinskog i uređajskog kompajlera [23]

Ograničenja jezika

SYCL standard inspiriran je OpenCL standardom uz proširanja koje donose mogućnost pisanja koda koristeći standardnu C++ sintaksu i kompatibilnost s C++ programima. Ipak, postoje značajke C++ jezika koje nisu podržane tokom pisanja dijelova koda za uređaje. Na primjer, SYCL za uređaje ne podržava virtualne funkcije, općenito pokazivače na funkcije, iznimke, informacije o tipu tijekom izvođenja (runtime type information), niti cijeli niz C++ biblioteka koje ovise o tim značajkama ili o određenim značajkama kompajlera na domaćinskoj strani [22].

Poglavlje 3

Implementacija raznih primjera koristeći oneAPI DPC++ kompajler

Kao što je ranije najavljeno, ovo poglavlje izlaže nekoliko osnovnih primjera kratkih programa koji ujedno služe i kao platforma za konkretan prikaz važnih dijelova strukture SYCL programa. Nadalje, poglavlje daje općenite upute vezane uz lokalno postavljanje SYCL okoline.

3.1 Postavljanje SYCL okoline

Postavljanje lokalne okoline

Kako je navedeno na kraju prošlog poglavlja, ovaj rad se zasniva na korištenju DPC++ kompajlera - Intel-ove implementacije SYCL standarda unutar oneAPI ekosustava.
Budući da je sam SYCL kao projekt još uvijek u povojima, do studenog 2024. godine nije kreirano programsko rješenje kojime korisnici mogu jednostavno postaviti SYCL okolinu, instalacijom DPC++ kompajlera, na sustave upravljane različitim operacijskim sustavima, koji sadrže i hardverske komponente različitih proizvođača.

OneAPI DPC++ kompajler može se instalirati kao zasebna komponenta ili kao dio Intel-ovo *oneAPI Base Toolkit* seta alata za razvijanje visokoperformantnih aplikacija. Navedena instalacija pruža automatski generirano okruženje za razvijanje SYCL aplikacija na Windows i Linux operativnim sustavima koji sadrže Intel hardverske komponente (CPU i akceleratore). Također, instalacijom je moguće integrirati SYCL okolinu s razvojnim okolinama poput *Microsoft Visual Studio*-a.

Za sustave koji sadrže akceleratore drugih proizvođača nužno je instalirati dodatne priključke kako bi se akceleratori integrirali u SYCL okruženje.

Za kreiranje SYCL okoline na Windows ili Linux operacijskim sustavima koji koriste NVIDIA grafičke kartice potrebno je instalirati odgovarajuće GPU upravljačke softvere i CUDA razvojne alate. Osim toga, potrebno je instalirati dodatan *oneAPI for NVIDIA GPUs* priključak. Za vrijeme pisanja ovog rada jedini takav priključak javno je distribuiran od strane tvrtke Coldplay. Kreiranje SYCL okoline na sustavima koji sadrže AMD-ove akceleratore trenutačno je moguće samo na Linux operacijskim sustavima. Za ispravno kreiranje SYCL okoline potrebno je dodatno instalirati AMD-ov ROCm set razvojnih alata kao i *oneAPI for AMD GPUs* priključak kojeg također distribuira tvrtka Coldplay.

Detalji oko instalacije nalaze se na službenim *Intel stranicama* [9].

Nakon uspješne instalacije potrebno je pokrenuti odgovarajuće skripte za postavljanje varijabli okruženja. Detalji oko lokacije skripti kao i načina pokretanja mogu se pronaći u službenoj dokumentaciji za postavljanje SYCL okoline na Windows odnosno Linux sustavima [9].

Prije samog pokretanja SYCL aplikacija potrebno je provjeriti jesu li odgovarajuće hardverske komponente prepoznate od strane SYCL okruženja. Najjednostavniji način za provjeru prepoznavanja hardverskih komponenti u SYCL okruženju je pomoću naredbe

```
$ sycl-ls
```

u oneAPI terminalu. Rezultati *sycl-ls* mogu se usporediti s rezultatima *clinfo* naredbe koja daje detaljnije informacije o svim platformama i hardverskim komponentama sustava bili oni podržani od strane SYCL okruženja ili ne.

Ukoliko je okruženje ispravno postavljeno, SYCL programi mogu se pokretati unutar integriranih razvojnih okolina poput Visual Studio-a odabirući DPC++ kompajler u postavkama projekta. Programe je moguće kompajlirati i direktno iz terminala koristeći naredbe ***icpx***, ***icx-cl*** ili ***icx-cl*** za poziv kompajlera uz dodatnu opciju *-fsycl* za kompajliranje SYCL programa. Konkretno, *hello-world.cpp* SYCL primjer bi se na Windows sustavu, koristeći icx za poziv DPC++ kompajlera, kompajlirao ovako :

```
$ icx -fsycl hello-world.cpp
```

Moguće je kompajleru zadati dodatne uvjete koristeći integrirane zastavice poput onih za postavljanje ciljeva za kompajliranje akceleriranog koda. Tako se program može kompajlirati uz zastavicu *-fsycl-targets* koja definira za koji GPU cilj se izgrađuju SYCL jezgrine funkcije. Naredba

```
$ icx-cl -fsycl -fsycl-targets=nvptx64-nvidia-cuda hello-world.cpp -o
    hello-world
```

izgrađuje SYCL jezgre za NVIDIA GPU ciljeve. Kompajlerske zastavice omogućavaju i definiranje više ciljeva tokom izgradnje kako bi se izvršni program mogao pokretati na drugačijim sustavima. Na primjer, zastavica

```
$ ... -fsycl-targets=spir64,amdgcn-amd-amdhsa,nvptx64-nvidia-cuda \ ...
```

će osigurati kreaciju jedinstvene binarne datoteke koja se može izvršiti na Intel, NVIDIA i AMD grafičkim karticama.

Upotreba gotovih *online* okolina

Budući da je SYCL moderna tehnologija, postoji nekoliko gotovih, besplatnih SYCL okolina kojima korisnici mogu pristupati preko interneta i u njima razvijati i testirati jednostavne SYCL programe. Naravno, takve okoline donose niz poteškoća kao što su integracija raznih biblioteka, ograničeni računalni resursi, nužnost internetske konekcije itd., ali vrlo su korisni u procesu učenja i upoznavanja sa SYCL-om.

Najjednostavnija internetska razvojna okolina dana je direktno od strane Kronos grupe i služi za upoznavanje s anatomijom SYCL programa. Okruženje *SYCL playground* [2] nudi nekoliko uvodnih SYCL primjera te mogućnost pokretanja svojih SYCL programa. Ipak, korisnici su ograničeni na korištenje jedne C++ datoteke kao i samo na standardni set C++ biblioteka [24].

Intel također nudi svoju internetsku razvojnu okolinu u oblaku kroz *Intel Tiber AI Cloud* [10]. Unutar navedene okoline nalazi se cijeli set uvodnih SYCL primjera koje je moguće pokretati i prilagođavati kroz sučelje *Jupyter* bilježnice. Dodatno, moguće je definirati vlastite C++ datoteke, koristiti programski kod iz više datoteka te dodati resurse za rad poput slika ili tekstualnih dototeka. Intelova razvojna okolina nudi nekoliko CPU i GPU uređaja naprednih performansi.

Razvojne okoline korištene u radu

Za pokretanje i testiranje SYCL primjera u ovom radu korištene su dvije SYCL razvojne okoline. Slijedi pregled uređaja i platformi koje definiraju obje okoline.

Lokalna okolina

Lokalna okolina kreirana je na osobnom prijenosnom računalu pod Windows operacijskim sustavom. Uz *oneAPI base Toolkit* koji donosi DPC++ kompajler, okolina sadrži CUDA 12.6 razvojne alate te Microsoft Visual Studio 2022 IDE za razvoj aplikacija. Popis dostupnih CPU i GPU uređaja unutar lokalne okoline vidljiv je na slici 3.1.

```
Platform: Intel(R) OpenCL
        Device: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
Platform: Intel(R) OpenCL HD Graphics
        Device: Intel(R) UHD Graphics
Platform: NVIDIA CUDA BACKEND
        Device: NVIDIA GeForce MX330
```

Slika 3.1: Detalji o platformama i uređajima lokalne okoline

Online Intel okolina

Pri pokretanju većine primjera korištena je SYCL okolina na *Intel Tiber AI Cloud* servisu [10].
Popis dostupnih CPU i GPU uređaja unutar Intel *online* okruženja prikazan je na slici 3.2.

```
Platform: Intel(R) OpenCL
    Device: Intel(R) Xeon(R) Platinum 8468V
Platform: Intel(R) OpenCL Graphics
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
    Device: Intel(R) Data Center GPU Max 1100
Platform: Intel(R) Level-Zero
    Device: Intel(R) Data Center GPU Max 1100
```

Slika 3.2: Detalji o platformama i uređajima Intel okoline na oblaku

3.2 Otkrivanje informacija sustava

Otkrivanje platformi i odgovarajućih uređaja

Određeni heterogeni sustav može imati više platformi te se svaka od platformi može sastojati od više uređaja. *Sycl API* nudi jednostavan način dohvaćanja popisa svih platformi sustava te odgovarajućih uređaja. Jedan od načina korištenja navedene funkcionalnosti izložen je u idućem primjeru (Listing 3.1):

```
#include <sycl/sycl.hpp>

int main()
{
    for (auto platform : sycl::platform::get_platforms())
    {
        std::cout << "Platform: "
            << platform.get_info<sycl::info::platform::name>()
            << "\n";

        for (auto device : platform.get_devices())
        {
            std::cout << "\tDevice: "
                << device.get_info<sycl::info::device::name>()
                << "\n";
        }
    }
}
```

Listing 3.1: Otkrivanje informacija o dostupnim uređajima sustava koristeći SYCL API

Postavlja se pitanje zašto je važno imati mogućnost dohvaćanja dostupnih platformi i uređaja? Nije li osnovna ideja SYCL-a upravo apstrakcija tih detalja od korisnika? U nekim slučajevima odgovor je potvrdan, no postoje situacije u kojima je potrebna kontrola nad eksplicitnim izvršavanjem zadataka. Moguće je koristiti ugrađene SYCL selektore za odabir uređaja na kojem će se izvršavati zadaci, pri čemu se u potpunosti oslanja na SYCL runtime. Međutim, ako se želi postići veća preciznost, moguće je iskoristiti prezentiranu funkcionalnost za odabir specifičnog uređaja na određenoj platformi.

Ispis informacija o uređajima

U prethodnom primjeru pokazano je kako ispisati imena svih uređaja na dostupnim platformama, no SYCL API nudi dodatne informacije o uređajima poput proizvođača, verzije sustava, veličine globalne memorije, broja radnih jedinica itd. što je prikazano u primjeru Listing 3.2.

```
...

void print_device_info(const sycl::device& device) {
    std::cout << "\tDevice: "
              << device.get_info<sycl::info::device::name>() << "\n";

    // Ispis tipa uređaja
    auto device_type =
            device.get_info<sycl::info::device::device_type>();
```

```
10
11      std::cout << "\t\tType: ";
12
13      switch (device_type) {
14          case sycl::info::device_type::cpu:
15              std::cout << "CPU" << "\n";
16              break;
17          case sycl::info::device_type::gpu:
18              std::cout << "GPU" << "\n";
19              break;
20          case sycl::info::device_type::accelerator:
21              std::cout << "Accelerator" << "\n";
22              break;
23          case sycl::info::device_type::custom:
24              std::cout << "Custom" << "\n";
25              break;
26          default:
27              std::cout << "Unknown" << "\n";
28              break;
29      }
30
31      // Ispis veličine globalne memorije
32      std::cout << "\t\tGlobal Memory Size: "
33          << device.get_info<sycl::info::device::global_mem_size>()
34          / (1024 * 1024)
35          << " MB" << "\n";
36
37      // Ispis maksimalnog broja radnih stavki
38      std::cout << "\t\tMax Compute Units: "
39          << device.get_info<sycl::info::device::max_compute_units>()
40          << "\n";
41
42      // Ispis maksimalne veličine radne stavke
43      std::cout << "\t\tMax Work Group Size: "
44          << device.get_info<sycl::info::device::max_work_group_size>()
45          << "\n";
46
47      // Ispis podržane verzije
48      std::cout << "\t\tVendor: "
49          << device.get_info<sycl::info::device::vendor>()
50          << "\n";
51
52      std::cout << "\t\tVersion: "
53          << device.get_info<sycl::info::device::version>()
54          << "\n";
55  }
```

Listing 3.2: Otkrivanje detalja uređaja koristeći SYCL API

Selektori za odabir uređaja

Kako je prava moć *sycl-a* pisanje prenosivog koda za heterogene sustave, to zahtjeva mogućnost otkrivanja uređaja na sustavu ispitivanjem informacija o dostupnom *hardware-u*. Važno je naglasiti da je ova funkcionalnost omogućena putem SYCL *API*-ja, uz pretpostavku da nije poznato postoji li na sustavu neki akcelerator, pri čemu je cilj osigurati da se kod izvrši bez grešaka. U nastavku se nalazi primjer (Listing 3.3) upravo opisanog postupka odabira jedinice na kojoj će se sigurno izvršiti program.

```
1  #include <sycl/sycl.hpp>
2
3  int main() {
4      sycl::device device;
5
6      try {
7          device = sycl::device(sycl::gpu_selector_v);
8      } catch (sycl::exception const &e) {
9          std::cout << "Nije moguće pronaći GPU na sustavu\n"
10             << e.what() << "\n";
11         std::cout << "Izabran je CPU umjesto uređaja.\n";
12         device = sycl::device(sycl::cpu_selector_v);
13     }
14
15     std::cout << "Koristi se: " <<
16         device.get_info<sycl::info::device::name>();
17 }
```

Listing 3.3: Prikaz korištenja selektora za odabir GPU uređaja te selektora za odabir CPU-a kao *fallback*

U liniji 7. korišten je selektor za GPU uređaj koji će, koristeći unaprijed propisanu heuristiku i ispitujući informacije sustava o *hardware*-skim komponentama, odabrati najprikladniji uređaj za izvršavanje zadataka. Ovaj primjer koristan je za razjašnjavanje osnovnih tipova SYCL selektora [22].

- Dodijeljeni selektor *sycl::default_selector_v* vraća neki SYCL-uređaj iz bilo kojeg podržanog *SYCL-backenda*. Heuristika kojom se bira dodijeljeni uređaj nije specificirana SYCL standardom već je definirana u samoj implementaciji kompajlera. Ovaj selektor uvijek mora vratiti neki uređaj.
- GPU selektor *gpu_selector_v* vraća neki uređaj čiji je tip *info::device_type::gpu*. Ukoliko se takav uređaj ne nalazi na sustavu program izbacuje grešku.
- Selektor akceleratora *accelerator_selector_v* vraća neki uređaj čiji je tip *info::device_type::accelerator*. Ukoliko se takav uređaj ne nalazi na sustavu program izbacuje grešku.

- CPU selektor *cpu_selector_v* vraća neki uređaj čiji je tip *info::device_type::cpu*. Ukoliko se takav uređaj ne nalazi na sustavu program izbacuje grešku.

Dobro je napomenuti kako je moguće proširiti SYCL *API* konstruirajući vlastite selektore (Listing 3.4) uz odabranu heuristiku za kreiranje selektora.

```
...
auto custom_selector = [](const sycl::device& d) {
    // Ignoriraju se uređaji bez podrške za dvostruku preciznost (double
       -precision)
    if (!d.has(sycl::aspect::fp64)) {
        return -1; // Eliminacija iz selekcije
    }

    // Ako je uređaj GPU, preferiraju se NVIDIA GPU
    if (d.is_gpu()) {
        std::string vendor = d.get_info<sycl::info::device::vendor>();

        // NVIDIA GPU ima najveći prioritet
        if (vendor.find("NVIDIA") != std::string::npos) {
            return 4;
        }
        return 3; // Ostali GPU-ovi
    }
    // Ako je uređaj akcelerator (ali nije GPU), ima srednji prioritet
    if (d.is_accelerator()) {
        return 2;
    }
    // Ako je CPU, ima niži prioriteta
    if (d.is_cpu()) {
        return 1;
    }
    // eventualni ostali uređaji dobijaju najniži prioritet
    return 0;
};
...
sycl::queue q(custom_selector);
...
```

Listing 3.4: Definiranje vlastitog selektora

3.3 Uvodni primjeri

Hello World!

Kao i u svim uvodnim lekcijama nekog programskog jezika ili tehnologije, funkcioniranje programa objašnjava se *Hello World!* primjerom - jednostavnim kodom koji za cilj ima ispisati tekst *"Hello World!"* na konzolu [2]. Ovaj osnovni primjer koristi se i za ukazivanje na određene specifičnosti SYCL programa (Listing 3.5).

```
#include <sycl/sycl.hpp>

class hello_world;

int main() {
    auto device_selector = sycl::default_selector_v;

    sycl::queue queue(device_selector);

    std::cout << "Odabrani uređaj: "
        << queue.get_device().get_info<sycl::info::device::name>()
        << "\n";

    queue.submit([&] (sycl::handler& cgh) {
        auto os = sycl::stream{128, 128, cgh};
        cgh.single_task<hello_world>([=]()
        {
            os << "Hello World! (s uređaja)\n";
        });
    });

}
```

Listing 3.5: Hello World! aplikacija

Paralelizacija jezgara

Temeljna zadaća uređaja je paralelno izvršavanje jezgara, odnosno jednostavnih operacija na neovisnim podacima. Ta zadaća je analogna izvršavanju standardne *for* petlje u kojoj je svaka iteracija neovisna o prethodnoj. SYCL kroz svoj *API* nudi jednostavan način za kreiranje paralelnih jezgara kroz *parallel_for* funkciju čiji detaljni pregled slijedi u nastavku. Kao što je navedeno ranije, *parallel_for* je metoda članica *sycl::handler* klase, no može se pozvati i direktno na *sycl::queue* objektu. U slučaju poziva na *sycl::queue* objektu, SYCL implicitno stvara *sycl::handler* objekt i na njemu poziva odgovarajuću metodu.

Funkcionalnost paralelnih jezgara je izložena preko ***sycl::range***, ***sycl::item*** i ***sycl::id*** klasa. Pomoću *sycl::range* klase definira se **prostor iteracija**, odnosno 1D, 2D ili 3D polje koje definira dimenziju i ukupan broj radnih stavaka koje se pokreću na radnim jedinicama GPU-a. Svaka je radna stavka (**work item**) fizička instanca jezgre te posjeduje svoj indeks koji koristi za pristup podacima.

```
...
    // parallel_for kao parametar prima sycl::range objekat koji
       definira 1D polje dimenzije 1024 te lambda funkciju ili
       funkcijski objekat.
    // U ovom slučaju lambda prima 1D sycl::id koji označava indeks
       konkretne radne stavke
    myQueue.parallel_for(range<1>(1024), [=](id<1> ID)
    {
    // kod koji se izvršava na uređaju
    // preko sycl::id ID radna stavka pristupa podacima
        A[ID] = B[ID] + C[ID];
    });
...
```

Listing 3.6: Pregled poziva *parallel_for* funkcije

Gornji primjer (Listing 3.6) prikazuje korištenje 1D prostora iteracija. U slučaju korištenja 2D ili 3D prostora iteracija svaki ID je ujedno i polje odgovarajuće dimenzije. Tada radna stavka pristupa podacima preko indeksnog operatora ([]).

Ukoliko se unutar zapovijedne grupe trebaju koristiti informacije o samom prostoru iteracija, poželjno je umjesto *sycl::id* koristiti *sycl::item* klasu. Klasa *sycl::item* predstavlja instancu funkcijskog objekta koja se izvršava, a sadrži dodatne metode za dohvat informacija poput *get_range()* i *get_id*.

IOTA

Još jedan tip *Hello World!* primjera za paralelno računanje su *IOTA* programi. U računarstvu, **IOTA** program označava postupak generiranja sekvencijalnog niza cijelih brojeva. Navedeni primjer ujedno služi i za verifikaciju SYCL radnog okruženja.

```
#include <sycl/sycl.hpp>
#include <array>
#include <iostream>
#include <numeric>

using namespace std;

constexpr size_t array_size = 10000;
typedef array<int, array_size> IntArray;

```

```
11 void SyclIota(sycl::queue &q, IntArray &a_array, int value)
12 {
13   sycl::range num_items{a_array.size()};
14   sycl::buffer a_buf(a_array);
15
16   q.submit([&](auto &h) {
17     sycl::accessor a(a_buf, h, sycl::write_only, sycl::no_init);
18     h.parallel_for(num_items, [=](auto i) { a[i] = value + i; });
19   });
20 }
21
22 int main() {
23     IntArray sequential, parallel;
24     constexpr int value = 100000;
25     auto selector = sycl::default_selector_v;
26
27     // sekvencijalna iota na CPU
28     std::iota(sequential.begin(), sequential.end(), value);
29
30     try {
31         // red za dodijeljeni uređaj
32         sycl::queue q(selector);
33         // Ispis informacija o dodijeljenom uređaju
34         cout << "Izvršavanje na uređaju: "
35             << q.get_device().get_info<sycl::info::device::name>()
36             << "\n";
37
38         // poziv palalelne sycl iote
39         SyclIota(q, parallel, value);
40     } catch (std::exception const &e) {
41         cout << "Dogodila se pogreška tokom rada na uređaju :";
42         << e.what() << "\n";
43         terminate();
44     }
45
46     // verifikacija jednakosti sekvenci
47     if (std::equal(sequential.begin(), sequential.end(),    parallel.
       begin())) {
48         std::cout << "Iota je uspješno izvršena na uređaju." << "\n";
49     }
50     else {
51         std::cout << "Iota na uređaju nije proizvela pravilnu sekvencu."
               << "\n";
52     }
53
54 }
```

Listing 3.7: IOTA aplikacija uz korištenje *buffer* modela

Glavni dio ovog programa (Listing 3.7) vezan za SYCL izvršavanje odvija se u funkciji *SyclIota*.
U liniji 13. kreiran je *range* objekat koji definira dimenzije radnog prostora odnosno označava koliko će se paralelnih zadataka pokrenuti na uređaju. Budući da je veličina polja *a_array* $10,000$, *range* će osigurati pokretanje $10,000$ paralelnih zadataka.
U liniji 14. korišten je *buffer* objekat za omotavanje polja *a_array*. Kreirani *buffer* omogućava kopiranje memorije između domaćina i uređaja.
U red *q* , koji je red zadataka na dodijeljenom uređaju, dodaje se novi zadatak preko *submit* metode te ona koristi *handler* objekat *h* za definiranje zadataka i upravljanje memorijom.
Aksesor *a* definiran u liniji 17. omogućava pristup *bufferu a_buff* kodu unutar *parallel_for* funkcije koja se izvršava na uređaju. S *write_only* je definirano da uređajski kod može samo pisati u *buffer*.
Handler objekat u liniji 18. pokreće zadatak *parallel_for* koji se izvodi na uređaju. *Parallel_for* kao ulazne parametra prima *range* objekat *num_items*, koji definira koliko radnih zadataka će se pokrenuti u paraleli, te jezgrinu funkciju (kernel) koja preko aksesora upisuje generirane vrijednosti u zadani *buffer*.

USM IOTA

U prethodnom primjeru korišten je *buffer* memorijski SYCL model te tehnike poput *submit*-a, aksesora i *buffer* objekta za automatsko kontroliranje pristupa zajedničkoj memoriji i životnog vijeka jezgrinih funkcija.
Kroz idući program prikazan je identičan IOTA program koji generira niz na domaćinu i uređaju te uspoređuje njihovu jednakost, no uz korištenje USM memorijskog modela.

```
1  #include <sycl/sycl.hpp>
2  #include <array>
3  #include <iostream>
4
5  using namespace std;
6
7  constexpr size_t array_size = 10000;
8
9  void IotaParallel(sycl::queue &q, int *a, size_t size, int value)
10 {
11     sycl::range num_items{size};
12
13     q.parallel_for(num_items, [=](auto i) {
14         a[i] = value + i;
15     }).wait();
16 }
17
18 int main() {
```

```
constexpr int value = 100000;

try {
    sycl::queue q;


    int *sequential = sycl::malloc_shared<int>(array_size, q);
    int *parallel = sycl::malloc_shared<int>(array_size, q);

    if (!sequential || !parallel) {
        if (sequential) sycl::free(sequential, q);
        if (parallel) sycl::free(parallel, q);
        cerr << "Pogreška pri alokaciji dijeljene memorije.\n";
        return -1;
    }

    // IOTA na CPU
    for (size_t i = 0; i < array_size; ++i) {
        sequential[i] = value + i;
    }

    // paralelna IOTA na SYCL uređaju
    IotaParallel(q, parallel, array_size, value);

    for (size_t i = 0; i < array_size; ++i) {
        if (parallel[i] != sequential[i]) {
            cerr << "Neuspješno generiranje sekvence na uređaju.\n";
            sycl::free(sequential, q);
            sycl::free(parallel, q);
            return -1;
        }
    }


    sycl::free(sequential, q);
    sycl::free(parallel, q);
    cout << "Uspješno generiranje sekvence na uređaju.\n";

} catch (std::exception const &e) {
    cerr << "Došlo je do pogreške tijekom računanja na uređaju: "
        << e.what() << "\n";
    return -1;
}

return 0;
}
```

Listing 3.8: IOTA aplikacija uz korištenje USM modela

Razlika između USM programa (Listing 3.8) i prethodnog (Listing 3.7) najviše se očituje kroz eksplicitno alociranje zajedničke memorije u linijama 24. i 25. Funkcija *sycl::malloc_shared* pokušava napraviti zajedničku USM alokaciju za domaćina i uređaj povezan s redom *q*, te ukoliko uspije vrati pokazivač na dijeljenu memoriju u virtualnom dijeljenom prostoru. Provjere valjanosti memorijskih pokazivača u linijama 27.-29., kao i eksplicitna oslobađanja memorije poput onih u linijama 45. i 46., ukazuju na nedostatke u USM modelu. Također, kako u *IotaParallel parallel_for* nije stavljen u red pomoću *submit* metode, već je direktno invociran, nužno je staviti poziv *wait* metode u liniji 14.

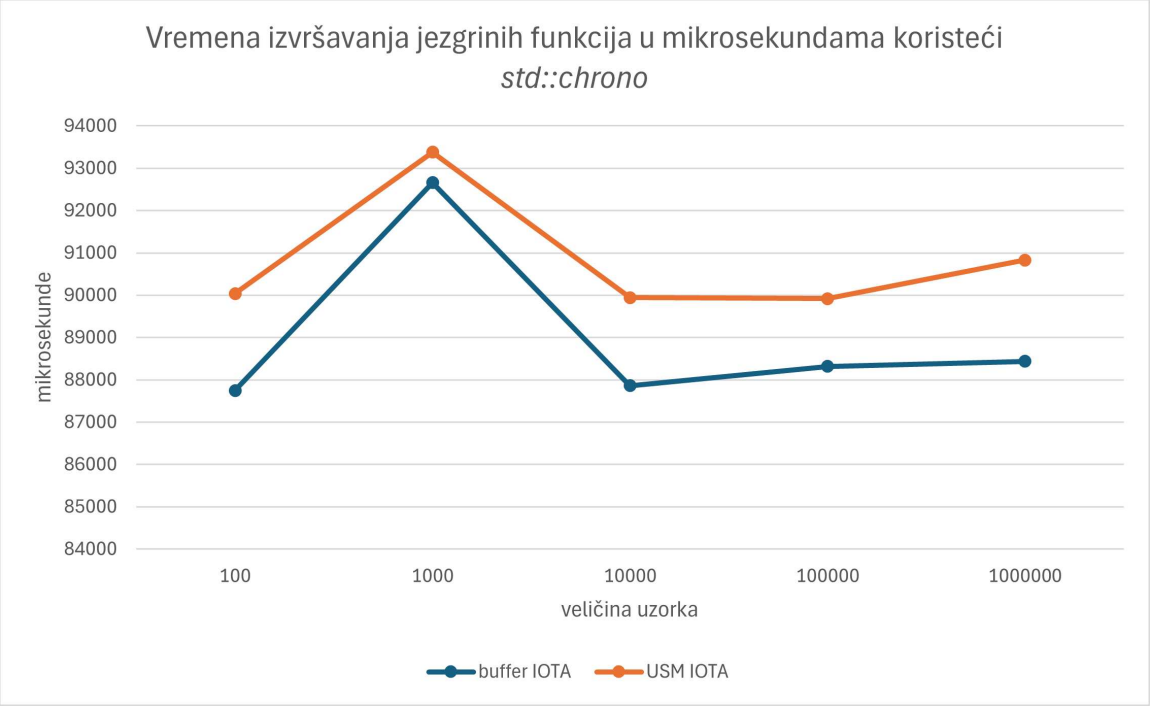
Mjerenje vremena izvršavanja SYCL jezgrinih funkcija

Praćenje vremena izvršavanja zadataka na uređaju važno je radi optimizacije rasporeda radnih zadataka između domaćina i akceleratora. U nekim situacijama nije isplativo prebaciti izvršavanje zadataka na akcelerator jer višak uzrokovan prebacivanjem memorije između domaćina i uređaja znatno nadilazi koristi dobivene prebacivanjem zadataka na uređaje. Heterogenost nije garancija akceleraciji programa, već sam program mora biti dobro osmišljen i isprogramiran.
Standardni C++ način mjerenja vremena je korištenjem ***std::chrono*** biblioteke [3]. Naredbama poput *std::chrono::high_resolution_clock::now()* prije i poslije jezgrinih funkcija moguće je odrediti vrijeme izvršavanja iste (Listing 3.9). Međutim, dobiveno vrijeme često ne označava samo vrijeme izvršavanja jezgrine funkcije, već ukupno vrijeme utrošeno na transfer podataka između domaćina i uređaja, izvršavanje jezgrine funkcije i sinkronizaciju. Dodatno, kroz SYCL API, jezgrine se funkcije često ne izvršavaju odmah, već se predaju u red odabranog uređaja. Stoga, dodavanje mjerenja vremena oko *sycl::queue.submit* metode mjeri dodatno i vrijeme proteklo od početka kreacije zapovijedne grupe zadataka te vrijeme dodavanja iste grupe u red uređaja.

```
1  ...
2  sycl::queue myQueue;
3  ...
4  // uzimanje vremenske oznake prije izvršavanja jezgre
5  auto device_start = std::chrono::high_resolution_clock::now();
6
7  // izvršavanje jezgre
8  myQueue.parallel_for(...).wait();
9
10 // uzimanje vremenske oznake poslije izvršavanja jezgre
11 auto device_end = std::chrono::high_resolution_clock::now();
12
13 // pretvorba razlike vremenskih oznaka u vremenski interval u
      mikrosekundama koristeći std::chrono::duration_cast
14 auto device_duration =
15     std::chrono::duration_cast<std::chrono::microseconds>
```

```
16          (device_end - device_start).count();
17
18  std::cout << "Vrijeme izvršavanja jezgre: " << device_duration
19      << " mikrosekundi" << "\n";
20
21  ...
```

Listing 3.9: Mjerenje vremena izvršavanja SYCL jezgre koristeći *std::chrono*



Slika 3.3: Usporedba prosječnog vremena izvršavanja jezgrinih funkcija buffer i USM verzija IOTA programa u mikrosekundama koristeći *std::chrono* na 'Intel Data Center GPU Max 1100' grafičkoj kartici

Budući da je slika 3.3 izražena u mikrosekundama, može se zaključiti da ne postoji velika razlika u prosječnim vremenima izvršavanja zapovijednih grupa. Glavni razlog za to je već naveden - većina vremenskog intervala mjerenog pomoću *std::chrono* biblioteke zauzme inicijalizacija same zapovijedne grupe odnosno predavanje grupe u red izvršavanja uređaja.

Alternativni način mjerenja je korištenje *sycl::handler* klase i njezine članske metode ***get_profiling_info()*** [4]. Funkciji predloška *get_profiling_info()* kao parametar se može proslijediti jedan od *struct*-ova iz *sycl::info::event_profiling* imeničkog prostora opisanih u Tablici 3.1.

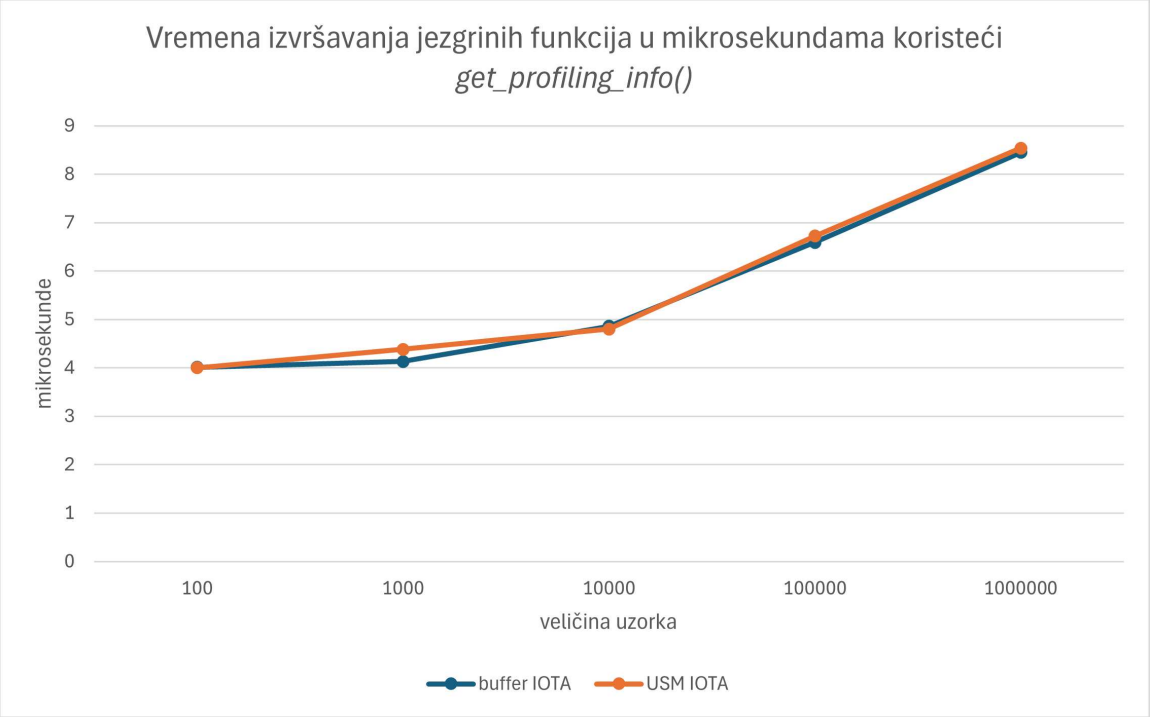
Tablica 3.1: *sycl::info::event_profiling* imenički prostor

Struct	Opis
command_submit	Označava vremenski trenutak u kojem se zapovijedna grupa predala u red za izvršavanje
command_start	Označava vremenski trenutak u kojem se bilo koji zadatak iz asocirane zapovijedne grupe krenuo izvršavati na uređaju
command_end	Označava vremenski trenutak u kojem se završilo izvršavanje zadataka na uređaju

Nužan uvjet za korištenje navedene tehnike je uključivanje informacija o profiliranju za odabrani red kroz njegov konstruktor.
Pregled tehnike mjerenja vremena izvršavanja SYCL jezgrine funkcije dan je u narednom isječku koda (Listing 3.10):

```
...
sycl::queue myQueue (sycl::default_selector_v,
    sycl::property::queue::enable_profiling{});
// omogućeno je profiliranje za myQueue putem
// sycl::property::queue::enable_profiling{} parametra
...
// prečac funkcija vraća \textit{sycl::event} objekat
auto event = myQueue.parallel_for(...);
event.wait();
auto start =
 event.get_profiling_info<sycl::info::event_profiling::command_start>();
auto end =
 event.get_profiling_info<sycl::info::event_profiling::command_end>();

// start i end su vremenske oznake u nanosekundama stoga ih je ponekad
   korisno pretvoriti u milisekunde, mikrosekunde ili sekunde
std::cout << "Vrijeme izvršavanje jezgre: " << (end - start) / 1.0e3 <<
   " mikrosekundi\n";

...
```

Listing 3.10: Mjerenje vremena izvršavanja SYCL jezgre koristeći *get_profiling_info()*



Slika 3.4: Usporedba prosječnog vremena izvršavanja jezgrinih funkcija buffer i USM verzija IOTA programa u mikrosekundama koristeći *get_profiling_info()* na 'Intel Data Center GPU Max 1100' grafičkoj kartici

Prosječna vremena izvršavanja jezgrinih funkcija mjerena profiliranjem (slika 3.4) podudaraju se s intuicijom. Povećanje broja uzoraka, odnosno veličine IOTA polja, donosi gotovo isto vremensko povećanje u *USM* i *buffer* verzijama programa.

AXPY

Još jedan od uvodnih primjera za heterogeno programiranje su *AXPY* programi :
Neka je α skalar te $\mathbf{x}$ i $\mathbf{y}$ dva vektora. Operacija AXPY izvodi sljedeći izraz:

$$\mathbf{y} = \alpha \cdot \mathbf{x} + \mathbf{y}$$

U nastavku je dana generička implementacija koja će radit s bilo kojim aritmetičkim tipom. Kroz primjer (Listing 3.11) se vidi kako se sva snaga modernog C++ može koristiti kroz SYCL.

```
#include <cassert>
#include <iostream>
#include <numeric>
#include <vector>
#include <cmath>
#include <limits>

#include <sycl/sycl.hpp>

using namespace sycl;

template <typename T>
std::vector<T> axpy(sycl::queue &Q, T alpha, const std::vector<T> &x,
    const std::vector<T> &y)
{
    assert(x.size() == y.size());
    auto vec_size = x.size();
    std::vector<T> z(vec_size);
    sycl::range<1> work_items { vec_size };

    // otvara se novi opseg kako bi se iskoristilo blokirajuće svojstvo
       buffer-a
    // i implicitno pozvalo wait()
    {
        sycl::buffer<T> bff_x(x.data(), work_items);
        sycl::buffer<T> bff_y(y.data(), work_items);
        sycl::buffer<T> bff_z(z.data(), work_items);

        Q.submit([&](handler &h) {
            auto acc_x = sycl::accessor(bff_x, h, read_only);
            auto acc_y = sycl::accessor(bff_y, h, read_only);
            auto acc_z = sycl::accessor(bff_z, h, write_only, no_init);

            h.parallel_for(work_items, [=](id<1> t_id) {
                acc_z[t_id] = alpha * acc_x[t_id] + acc_y[t_id];
            });
        });
    }
    return z;
}

template <typename T>
std::vector<T> axpy_host(T alpha, const std::vector<T> &x,
    const std::vector<T> &y)
{
    assert(x.size() == y.size());
    size_t vec_size = x.size();
```

```
46      std::vector<T> z(vec_size);
47
48      for (size_t i = 0; i < vec_size; ++i) {
49          z[i] = alpha * x[i] + y[i];
50      }
51
52      return z;
53  }
54
55  int main() {
56      constexpr auto vec_size = 1024;
57
58      std::vector<double> a(vec_size, 0.0);
59      std::iota(a.begin(), a.end(), 0.0);
60      std::vector<double> b(vec_size, 0.0);
61      std::iota(b.rbegin(), b.rend(), 0.0);
62
63      sycl::queue Q;
64
65      std::cout << "Izrvšavanje na uređaju: " <<
66          Q.get_device().get_info<info::device::name>() << "\n";
67
68      // APXY na uređaju
69      auto z_device = axpy(Q, 1.0, a, b);
70
71      // APXY na domaćinu
72      auto z_host = axpy_host(1.0, a, b);
73
74      std::cout << "Usporedba rezultata..." << "\n";
75
76      // tolerancija
77      const double epsilon = std::numeric_limits<double>::epsilon() * 100;
78
79      for (size_t i = 0; i < z_device.size(); ++i) {
80          // Usporedba rezultata
81          if (std::abs(z_device[i] -  z_host[i]) >= epsilon) {
82              std::cout << "Greška! z_device[" << i << "] = "
83              << z_device[i] << ", očekivano = " << z_host[i] << "\n";
84              std::cout << "Neuspješno izvršen apxy." << "\n";
85              return;
86          }
87      }
88
89      std::cout << "Uspješno izvršen axpy!" << "\n";
90  }
```

Listing 3.11: SYCL kod za generički axpy program

Navedeni primjeri prikazuju kako je, uz osnovno poznavanje C++ jezika, vrlo jednostavno pisati uvodne SYCL programe koristeći *oneAPI DPC++* kompajler. Svega par SYCL koncepata i osnovnih klasa dovoljno je za rasterećenje programa na akceleratore. Ipak, za stvarnu akceleraciju programa nužno je poznavati sustav za koji se program razvija, prirodu problema koji on rješava te pratiti stvarna vremena izvršavanja na domaćinu i akceleratorima. Dodatni alati za profiliranje, poput *Intel VTune Profiler*, pomažu u mjerenju performansi i otkrivanju uskih grla što dodatno olakšava prilagodbu akceleriranih programa. Iako SYCL nudi unificiran pristup akceleratorima svih proizvođača, u nekim je situacijama potrebno razumjeti njihovu fizičku strukturu i prema tome prilagoditi program.

Poglavlje 4

Složeniji primjeri

Naredno poglavlje izlaže nekoliko složenijih primjera koji pokazuju razne tehnike za kreiranje heterogenih programa kroz SYCL standard. Prikazuje se način kombiniranja višedretvenosti u C++ za paralelan prijenos zadataka na više uređaja. Detaljnije se opisuju radne stavke, grupe i podgrupe te *ND-range* objekti. Pred kraj poglavlja dan je pregled oneAPI DPC++ biblioteke.

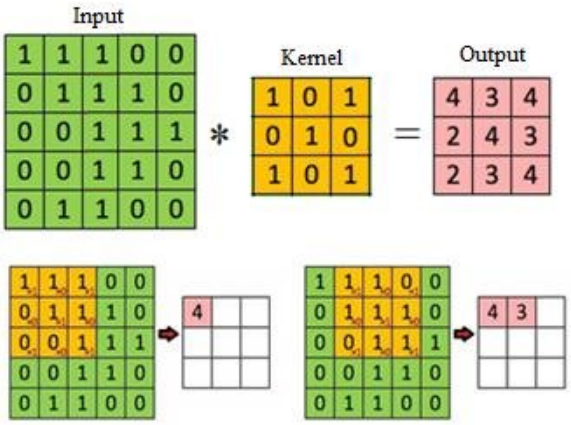
4.1 Konvolucija matrica

U domeni obrade slika ili fotografija jedna od glavnih korištenih tehnika za izvlačenje specijaliziranih informacija iz ulaznog signala ili dodavanja informacija ulaznom signalu je primjena filtera. Filteri se koriste za izoštravanje ili zamućivanje fotografija, detekciju horizontalnog ili vertikalnog ruba kao i za dodavanje kozmetičkih filtera.
U praksi, fotografije se promatraju kao dvodimenzionalne matrice, a filteri koji se primjenjuju na fotografije su zapravo konvolucija matričnih reprezentacija fotografija i jezgre konvolucije. Jezgre su obično kvadratne matrice malih dimenzija (npr. 3x3), koje su nerijetko i simetrične.

Definicija 4.1.1. *Neka je A ulazna matrica dimenzija $N \times M$ i K jezgra konvolucije (ili filter) dimenzije $k \times k$, gdje su $N, M, k \in \mathbb{N}$, k neparan te $N, M \geq k$. Konvolucija matrica A i K daje izlaznu matricu B dimenzije $(N - k + 1) \times (M - k + 1)$ i definira se kao:*

$$B(i, j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} A(i + m, j + n) \cdot K(m, n)$$

za sve i i j takve da $0 \leq i \leq N - k$ i $0 \leq j \leq M - k$, pri čemu se retci i stupci matrice kreću indeksirati od 0.



Slika 4.1: Konvolucija matrica [18]

Kako je vidljivo iz definicije 4.1.1 i slike 4.1 ovakvom se konvolucijom gubi $k/2$ ruba svake strane od ulazne matrice. Za dobivanje izlazne matrice istih dimenzija ulaznoj matrici, mogu se koristiti razne tehnike poput jednostavnog prepisivanja rubova ulazne matrice ili proširivanja izvorne matrice virtualnim vrijednostima.

Definicija 4.1.2. *[17] Neka je $A \in \mathbb{R}^{M\times N}$ ulazna matrica, a $K \in \mathbb{R}^{n\times n}$ jezgra konvolucije (ili filter). Pretpostavit ćemo da je n neparan broj oblika $n = 2r + 1$ gde su $N, M, r, n \in \mathbb{N}$ te r zovemo radijus konvolucije. Konvolucija ulazne matrice i jezgre konvolucije daje matricu $B \in \mathbb{R}^{M\times N}$, koja je definirana kao:*

$$B_{i,j} = \sum_{l=-r}^{r} \sum_{k=-r}^{r} K_{l,k} \cdot A_{i+l,j+k}$$

Ovdje su stupci i retci matrice K indeksirani od $-r, ..., 0, ...r$ pa, uz primjenu indeksacije od 0 do $2r + 1$, rezultat konvolucije postaje:

$$B_{i,j} = \sum_{m=0}^{2r} \sum_{n=0}^{2r} K_{m,n} \cdot A_{i+m-r,j+n-r}$$

Ovako definirana konvolucija zahvaća vrijednosti izvan ulazne matrice. Zbog toga, potrebno je definirati vrijednosti za *r* redova i stupaca 'prije' i 'nakon' matrice. Za virtualne vrijednosti najčešće se uzima nula, a virtualne vrijednosti nije nužno alocirati tokom programiranja već je dovoljno naredbama uvjeta izuzeti virtualne vrijednosti iz računanja konvolucije.

U nastavku je prikazan SYCL program konvolucije (Listing 4.1 , 4.2 i 4.3) koji kombinira nekoliko tehnika za paralelizaciju i ubrzanje programa poput višedretvenosti i korištenja višestrukih uređaja. Nadalje, prikazuje kako se u SYCL C++ okruženju mogu standardnim načinom koristiti dodatne biblioteke. U navedenom primjeru učitana je slika *Lena* koja se često koristi u primjerima vezanim uz procesiranje slika. Većina slika u boji sastavljena je od tri kanala za tri osnovne bolje (***R**ed*, ***G**reen*, ***B**lue*). Konvolucija slike u boji nekom jezgrom ekvivalentna je konvolucijama matrica koje predstavljaju njene kanale tom istom jezgrom. Slika u boji dimenzija $M \times N$ piksela, zapravo je sačinjena od tri $M \times N$ matrice čiji su elementi brojevi između 0 i 255.

Naredni primjer (Listing 4.3) učitava sliku te pokreće po jednu dretvu za svaki od kanala slike. Svaka dretva (Listing 4.2) izdvaja odgovarajući kanal slike, te ga predaje GPU-u koji nad kanalom vrši konvoluciju standardnom jezgrom za otkrivanje ruba slike (Listing 4.1).

Konvolucija na GPU

Pomoćna funkcija *convolution_channel* zadužena je za pripremu podataka i definiranje konvolucije koja će se izvršavati na uređaju (Listing 4.1). Kao ulazne parametre prima *input* i *kernel* vektore koji predstavljaju matricu kanala i matricu jezgre konvolucije spljoštene u jednu dimenziju. Dodatno, prima dimenzije matrica *N*, *M* i *K* te *sycl::queue* asociran s odabranim uređajem.

```
...
using namespace sycl;

// Funkcija za izvođenje konvolucije na jednom kanalu
std::vector<unsigned char> convolution_channel(
    const std::vector<unsigned char>& input, const std::vector<float>&
       kernel,
    int N, int M, int K, queue& q)
{
    std::vector<unsigned char> output_channel(N * M); //#vektor za
       razultat konvolucije
    int padding = K / 2; // Radijus konvolucije

    // definiranje buffer omotača oko podataka
    buffer input_buf(input.data(), range<1>(N * M));
    buffer kernel_buf(kernel.data(), range<1>(K * K));
    buffer output_buf(output_channel.data(), range<1>(N * M));

    // predaja zapovijedne skupine redu na izvršavanje
    q.submit([&](handler& h) {
        //# definiranje aksesora
```

```
20          auto input_acc = input_buf.get_access<access::mode::read>(h);
21          auto kernel_acc = kernel_buf.get_access<access::mode::read>(h);
22          auto output_acc = output_buf.get_access<access::mode::write>(h);
23
24          // paralelizacija SYCL jezgri
25          // korišten je 2D prostor iteracija radi jednostavnijeg rada s
                matricama
26          h.parallel_for(range<2>(N, M), [=](id<2> idx) {
27              int row = idx[0];
28              int col = idx[1];
29              float sum = 0.0f;
30
31              // računanje konvolucije matrice
32              for (int i = 0; i < K; i++) {
33                  for (int j = 0; j < K; j++) {
34                      int input_row = row + i - padding;
35                      int input_col = col + j - padding;
36
37                      // provjera rubnih uvjeta
38                      if (input_row >= 0 && input_row < N &&
39                          input_col >= 0 && input_col < M)
40                      {
41                          sum += input_acc[input_row * M + input_col] *
                                kernel_acc[i * K + j];
42                      }
43                  }
44              }
45
46              // izlazni podatak je ograničen na interval [0, 255]
47              output_acc[row * M + col] =
48                  std::clamp(static_cast<int>(sum), 0, 255);
49          });
50      });
51      q.wait();
52      return output_channel;
53  }
54  ...
```

Listing 4.1: *convolution_channel()* funkcija korištena od strane programske dretve za usmjeravanje zadataka na uređaj i računanje konvolucije kanala

Radna funkcija za kreiranje dretvi

Funkcija *process_channel* (Listing 4.2) služi kao radna funkcija za kreiranje dretvi. Glavna uloga joj je izoliranje odgovarajućeg kanala iz slike te pozivanje *convolution_channel* koja u suštini šalje kanal uređaju na daljnju obradu.

```
...
// radna funkcija korištena za kreiranje dretvi
void process_channel(int channel, const std::vector<unsigned char>&
    input, const std::vector<float>& kernel, int N, int M, int K,
    queue& q, std::vector<unsigned char>& output_channel)
{
    // svakom GPU se predaje samo relevantni kanal
    // stoga se privremeno kreira vektor \textit{input_channel} koji
       predstavlja spljoštenu matricu pojedinog kanala slike
    std::vector<unsigned char> input_channel(N * M);
    for (int i = 0; i < N * M; ++i) {
        input_channel[i] = input[i * 3 + channel];
    }
    // dretva rezultat konvolucije dobiven izvršavanjem na uređaju
       zapisuje u output_channel
    output_channel =
        convolution_channel(input_channel, kernel, N, M, K, q);
}
...
```

Listing 4.2: Radna funkcija za kreiranje novih dretvi

Glavni program

Ovaj program (Listing 4.3) pretpostavlja postojanje tri različita GPU uređaja na sustavu. Iako to nije slučaj na većini sustava, ovdje se prikazuje način pronalaska i odabira više različitih uređaja.

U glavnom programu slika se učitava u liniji 24. korištenjem *stb_image* biblioteke. U linijama 49. - 70. program pokušava pronaći tri dostupna GPU uređaja. Vektor *channel_outputs* definiran u liniji 72., predstavlja rezultate konvolucija po kanalima slike. Unutar *for* petlje definirane u liniji 80., kreiraju se tri nove dretve koje izoliraju odgovarajući kanal iz slike te ga predaju uređaju na daljnju obradu. Kako bi se osiguralo ispravno stanje *channel_outputs* vektora, u linijama 88.-90. sinkroniziraju se kreirane dretve. Izolirani kanali slike zatim se 'spajaju' nazad u sliku u linijama 93.-97. redoslijedom kojim su iz nje izolirani. Naposljetku, ukupni rezultat konvolucije slike sprema se u izlaznu datoteku *lena-rubovi.png* koristeći *stb_image_write* biblioteku. Funkcija *stbi_write_png* iz navedene biblioteke kao ulazne parametre prima put do izlazne datoteke, širinu i visinu slike i pikselima, broj kanala slike, pokazivač na podatke slike te broj bajtova u svakom redu slike.

```
#define STB_IMAGE_IMPLEMENTATION
#include "../stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "../stb_image_write.h"
```

```
#include <sycl/sycl.hpp>
#include <iostream>
#include <vector>
#include <thread>
#include <future>

using namespace sycl;

std::vector<unsigned char> convolution_channel(...)
{...}

void process_channel(...)
{...}

int main() {
    int width, height, channels;

    // slika se učitava pomoću stbi_load iz stb_image bibloteke
    // učitani podaci spremaju se kao niz bajtova
    unsigned char* img =
        stbi_load("lena.png", &width, &height, &channels, 0);

    if (img == nullptr) {
        std::cerr << "Nije moguće učitati sliku!" << "\n";
        return -1;
    }

    // Učitana slika pretvara se u vector radi lakšeg rukovanja
    std::vector<unsigned char> image_data(img, img +
        (width * height * channels));

    // Prikaz informacija o slici
    std::cout << "Širina slike: " << width << ", visina: " << height
        << ", broj kanala: " << channels << "\n";

    // Izlazna matrica treba biti iste veličine kao ulazna
    std::vector<unsigned char> output_image(width*height*channels, 0);

    // Standardna jezgra konvolucije za detekciju ruba
    std::vector<float> kernel = {
        -1, -1, -1,
        -1, 8, -1,
        -1, -1, -1
    };

    // Raspodjela posla na tri GPU uređaja za obradu svakog kanala
    std::vector<queue> queues;
```

```
52      try {
53          for (auto& device : sycl::device::get_devices()) {
54              if (device.is_gpu()) { // Provjera je li uređaj GPU
55                  // Koristi se najviše tri GPU-a
56                  if (queues.size() >= 3) break;
57                  queues.emplace_back(queue(device));
58                  std::cout << "Koristi se GPU uređaj: " <<
59                      device.get_info<info::device::name>() << "\n";
60              }
61          }
62
63          if (queues.empty()) {
64              throw std::runtime_error("Nije pronađen niti jedan GPU!");
65          }
66      } catch (const exception& e) {
67          std::cerr << "Pogreška tokom odabira GPU uređaja: " << e.what()
               << "\n";
68          return -1;
69      }
70
71      // Pretpostavlja se postojanje 3 odvojena GPU uređaja
72      if (queues.size() < 3) {
73          std::cerr << "Nema dovoljno GPU uređaja za obradu!" << "\n";
74          return -1;
75      }
76
77      std::vector<std::vector<unsigned char>> channel_outputs(3);
78      std::vector<std::thread> threads;
79
80      //# kreira se zasebna dretva za svaki kanal slike
81      //# uloga svake dretve je izolacija kanala te dodijeljivanje
           zadataka
82      //# zasebnom GPU uređaju. Rezultat računanja svakog GPU-a odgovaraju
           ća dretva
83      //# zapisuje u channel_outputs. Budući da svaka dretva pristupa svom
            komadu memorije
84      //# unutar channel_outputs nema stanja natjecanja.
85      for (int channel = 0; channel < 3; ++channel) {
86          //# podaci za koje je nužno da se ne kopiraju već koriste po
               referenci zamataju se u
87          //# std::ref
88          threads.push_back(std::thread(process_channel, channel,
89              std::ref(image_data), std::ref(kernel), height, width, 3,
90              std::ref(queues[channel]),
91              std::ref(channel_outputs[channel])));
92      }
93
```

```
94      //# Čekanje na završetak svih dretvi
95      for (auto& t : threads) {
96          t.join();
97      }
98
99      //# Spajanje kanala u konačni izlaz
100     for (int i = 0; i < height * width; ++i) {
101         for (int channel = 0; channel < 3; ++channel) {
102             output_image[i * 3 + channel] = channel_outputs[channel][i];
103         }
104     }
105
106     //# Spremanje rezultata u novu sliku
107     stbi_write_png("lena-rubovi.png", width, height, channels,
            output_image.data(), width * channels);
108
109     stbi_image_free(img);
110
111     std::cout << "Izlazna slika je spremljena kao: 'lena-rubovi.png'."
            << "\n";
112     return 0;
113 }
```

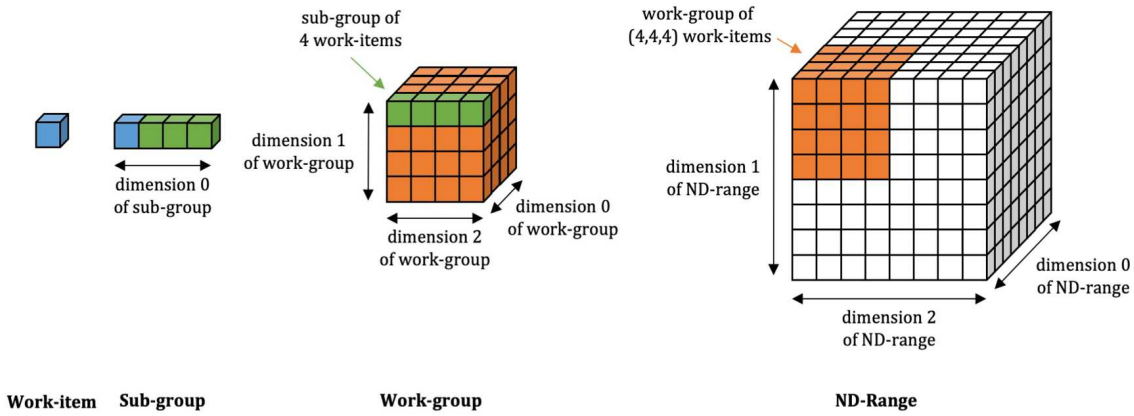
Listing 4.3: Glavni program konvolucije slike *Lena*

Ovaj primjer (Listing 4.3) u praksi ne donosi veliku akceleraciju programa ponajviše zbog vrlo malog uzorka odnosno slike koja nije velikih dimenzija u terminima računalne snage modernih sustava. Računalni višak koji se stvara kreiranjem i sinkronizacijom dodatnih dretvi, te prijenosom podataka između domaćina i uređaja, se na većini arhitektura neće nadomijestiti prikazanim tehnikama akceleracije. Ipak, ovaj program služi kao prikaz mogućnosti koje pruža SYCL podrška za prijenos rada na višestruke uređaje u kombinaciji s višedretvenošću.

Slika 4.2: Prikaz učitane slike *Lena* na lijevoj strani te rezultata konvolucije s jezgrom za detekciju ruba na desnoj strani

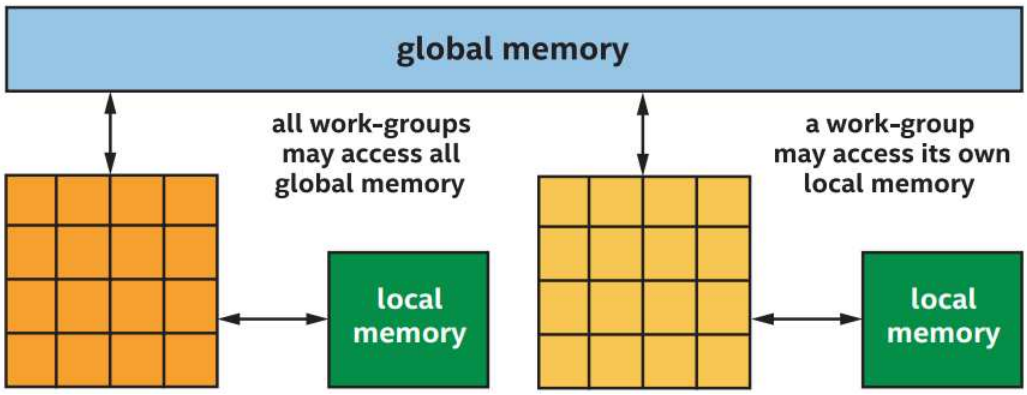
4.2 ND-Ranges

U ranijim dijelovima rada uveden je pojam *radne stavke* - entiteta koji se izvršava na domaćinu uređaju i najčešće predstavlja konkretnu instancu SYCL jezgrine funkcije. Radne stavke imaju jedinstvene globalne identifikatore u radnom indeksnom prostoru koji mogu biti jedno-, dvo- ili trodimenzionalni. Više povezanih radnih stavki moguće je reprezentirati radnom grupom (*work-group*). Radne stavke unutar iste radne grupe izvršavaju se na istoj fizičkoj jedinici obrade (*compute unit*) unutar uređaja. Također, radne stavke unutar iste radne grupe izvršavaju istu jezgru te dijele lokalnu memoriju. Dodatnu međugranulaciju predstavljaju radne podgrupe (*sub-groups*) koje označavaju kolekciju usko povezanih radnih stavki. Radne podgrupe obično tvore radne stavke koje izvršavaju (*SIMD* - single instruction multiple data) istovjetnu operaciju u isto vrijeme, ali na različitim podacima (analogno *warp*-u u CUDA terminologiji). Radne podgrupe izvršavaju se na izvršnim jedinicama (*execution unit*) odnosno procesnim elementima jedinica obrade (*processing element*).
Ukupnu organizaciju radnog indeksnog prostora jezgre moguće je reprezentirati kroz ***ND-range*** objekt. *ND-range* (Slika 4.3) je jedno-, dvo- ili trodimenzionalni indeksni radni prostor definiran globalnim i lokalnim *range*-om.



Slika 4.3: Prikaz odnosa *ND-range*, radne grupe, radne pod-grupe i radne stavke [15]

Primarni cilj korištenja *ND-Range* je iskorištavanje hardverskih značajki za optimizaciju izvršavanja jezgre. Sve radne stavke mogu komunicirati preko globalne memorije uređaja, no ona je spora i neefikasna. S druge strane, radne stavke unutar radne podrgupe koje se izvršavaju na istim izvršnim jedinicama uređaja efikasno dijele komuniciraju i podatke, no broj radnih stavki unutar podgrupe je jako ograničen. Kako bi se doskočilo problemu visoke latencije i neefikasne komunikacije među radnim stavkama iste radne grupe koje ne pripadaju zajedničkoj podgrupi, uvode se razna hardverska rješenja. Proizvođači grafičkih kartica tim ciljem dizajniraju lokalnu dijeljenu memoriju (***SLM*** - *shared local memory*) koja se nalazi direktno na GPU ploči (Slika 4.4). Lokalna dijeljena memorija može se smatrati *cache*-om radne grupe. Kada se radna grupa pokreće njezine radne stavke mogu eksplicitno podatke učitati iz globalne memorije u lokalnu. Lokalna memorija ostaje aktivna za vrijeme aktivnosti radne grupe te se može ekplicitno prepisati nazad u globalnu memoriju pri završetku životnog vijeka radne grupe. Veličina lokalne memorije uređaja može se dohvatiti pomoću
device::get_info<sycl::info::device::local_mem_size>() metode. Također, SYCL *API* nudi i metodu *device::get_info<sycl::info::device::local_mem_type>()* za provjeru je li dijeljena memorija na uređaju implementirana softverski ili postoji prava fizička komponenta dijeljene memorije na hardveru.



Slika 4.4: Shema memorije dostupne radnim grupama. Sve radne grupe imaju pristup istoj globalnoj memoriji i vlastitoj lokalnoj memoriji koja je dijeljena među radnim stavkama odgovarajuće radne grupe [19]

Dijeljenje memorije u radnoj grupi

Za upravljanje dijeljenom memorijom unutar grupe koriste se dva dodatna koncepta. Klasa ***sycl::local_accessor*** služi za deklariranje podataka unutar dijeljene memorije grupe. Slično kao i *sycl::accessor*, lokalni *aksesori* mogu biti jedno-, dvo- i trodimenzionalni te se njihova konstrukcija odvija unutar zapovijedne grupe. Lokalnom *aksesoru* mogu pristupiti sve radne stavke unutar iste radne grupe, no ne može mu pristupiti niti jedna radna stavka iz druge radne grupe. Za sinkronizaciju podataka u dijeljenoj memoriji koristi se ***sycl::group_barrier*** funkcija (Listing 4.4). Funkcija *sycl::group_barrier* osigurava konzistentnost podataka u dijeljenoj memoriji među radnim stavkama te se obično poziva nakon modifikacija lokalnog *aksesora*.

```
#include <sycl/sycl.hpp>
...
size_t N = 16;
sycl::queue q;
std::vector<float> data(N);
...
sycl::buffer data_buffer(data);
...
q.submit([&](handler &h){
    // definiranje aksesora za globalnu memoriju
    sycl::accessor data_acc(data_buffer, h, read_only);
    range<1> global_size(N);
    range<1> work_group_size(N);

    // definiranje aksesora za lokalnu memoriju
    local_accessor<float, 1> data_local_acc(range<1>(N), h);
```

```
17
18      // U situacijama kada se koristi \textit{sycl::nd\_range}, radna
            stavka reprezentirana je klasom \textit{nd::item}.
19      h.parallel_for(nd_range<1>{global_size, work_group_size}, [=](
            nd_item<1> item){
20          // dohvati informacija o radnoj stavci u globalnom i lokalnom
                indeksnom prostoru
21          const int i = item.get_global_id(0);
22          const int x = item.get_local_id(0);
23
24          // kopiranje iz globalne u lokalnu memoriju
25          data_local_acc[x] = data_acc[...];
26
27          // barijera koja sinkronizira lokalnu memoriju unutar radne
                grupe
28          group_barrier(item.get_group());
29          ...
30      });
31  });
```

Listing 4.4: Primjer kopiranja podataka iz globalne u dijeljenu lokalnu memoriju te sinkronizacije barijerom grupe

Dodatan način osiguravanja konzistentnosti i sinkroniziranosti podataka je korištenje atomskih operacija (***Atomic operations***). Atomske operacije osiguravaju višestruki pristup dijeljenoj memoriji bez izazivanja stanja natjecanja. Idući isječak iz koda prikazuje poziv SYCL jezgre koja se izvršava na N paralelnih radnih stavki (Listing 4.5). Svaka stavka pokušava mijenjati isto memorijsko mjesto što uzrokuje stanje natjecanja.

```
1  ...
2  q.parallel_for(N, [=](auto i) {
3      sum[0] += data[i];
4  });
5  ...
```

Listing 4.5: Isječak iz SYCL programa koji uzrokuje stanje natjecanja

SYCL *API* za potrebe rješavanja stanja natjecanja nudi ***sycl::atomic_ref*** klasu. Navedena klasa služi za izvršavanje atomskih operacija na uređaju te je sintaksom vrlo slična *std::atomic_ref*. Zamatanje varijable u *sycl::atomic_ref* osigurava da će se istoj pristupati atomarno tijekom života reference. Sada se gornji primjer (Listing 4.5) može zapisati koristeći *sycl::atomic_ref* (Listing 4.6):

```
1  ...
2  using namespace sycl;
3  ...
4  q.parallel_for(N, [=](auto i) {
5      // sum[0] se zamata u atomic_ref što osigurava atomarne pristupe
```

```
    auto atomic_var = atomic_ref<int, memory_order::relaxed,
        memory_scope::device, access::address_space::global_space>(sum
        [0]);

    // atomic_ref nudi integriranu metodu za dohvat i zbrajanje
    atomic_var.fetch_add(data[i]);
});
...
```

Listing 4.6: Primjer korištenja *sycl::atomic_ref* za izbjegavanje stanja natjecanja

Klasa *sycl::atomic_ref* je tipizirana s 4 parametra (Tablica 4.1), a kao ulazni parametar konstruktora prima referencu na varijablu koju treba *zamotati*.

Tablica 4.1: Tipizirani parametri *sycl::atomic_ref* klase

Parametar	Opis
T	Tip podataka nad kojim se izvodi atomska operacija
sycl::memory_order	Definira redoslijed memorijskih operacija
sycl::memory_scope	Definira memorijski opseg na kojem se osigurava atomarnost
sycl::access::address_space	Adresni prostor za pristup podacima (lokalni ili globalni)

U nastavku je dan primjer (Listing 4.7) koji pokazuje kako se *sycl::atomic_ref* može koristiti u programima koji upotrebljavaju USM model.

```
#include <sycl/sycl.hpp>

static constexpr size_t N = 1024;

int main() {
    sycl::queue q;
    std::cout << "Izvršava se na : " <<
    q.get_device().get_info<sycl::info::device::name>() << "\n";

    // alokacija i inicijalizacija podataka u dijeljenoj memoriji domać
        ina i uređaja
    auto data = sycl::malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) data[i] = i;

     // alokacija izlazne sume u dijeljenoj memoriji domaćina i uređaja
    auto sum = sycl::malloc_shared<int>(1, q);
    sum[0] = 0;

    //# SYCL jezgra za redukciju
    q.parallel_for(N, [=](auto i) {
```

```
20          // omatanje sume u sycl::atomic_ref
21          auto sum_atomic = sycl::atomic_ref<int,
22            sycl::memory_order::relaxed,
23            sycl::memory_scope::device,
24            sycl::access::address_space::global_space>(sum[0]);
25
26          // atomarno zbrajanje podataka u sumu
27          sum_atomic += data[i];
28      }).wait();
29
30      std::cout << "Ukupna suma je = " << sum[0] << "\n";
31      return 0;
32  }
```

Listing 4.7: *sycl::atomic_ref* u USM programima

Dijeljenje memorije u radnoj podgrupi

Kako je već objašnjeno radna podgrupa sastoji se od radnih stavki koji se izvršavaju u paraleli na istoj dretvi izvršne jedinice. Radne stavke unutar podgrupe komuniciraju direktno koristeći *shuffle* operacije, bez eksplicitnih memorijskih operacija. Sinkronizacija radnih stavki unutar podgrupe vrši se barijerama podgrupe, a konzistentnost memorije osigurava se memorijskim ogradama podgrupe. Radne stavke unutar radne podgrupe imaju pristup funkcijama i algoritmima podgrupe što uvelike ubrzava implementacije učestalih paralelnih algoritama.

Za potpuno iskorištavanje navedenih optimizacija ponekad je potrebno detaljnije poznavati arhitekturu uređaja na kojem se izvršavaju jezgre. Na primjer, *Intel*-ove grafičke kartice podržavaju podgrupe veličina 8, 16 i 32, dok je ta vrijednost na *NVIDIA* grafičkim karticama 32. DPC++ kompajler u većini slučajeva bira optimalnu veličinu podgrupa, no ona se može eksplicitno navesti. Moguće je saznati i veličinu podgrupa nekog uređaja što je prikazano u narednom primjeru (Listing 4.8).

```
1  #include <sycl/sycl.hpp>
2
3  int main() {
4    sycl::queue q;
5    std::cout << "Testirani uređaj : " << q.get_device().get_info<sycl::
       info::device::name>() << "\n";
6
7    // dohvaćanje svih podržanih veličina radnih podgrupa
8    auto sg_sizes = q.get_device().get_info<sycl::info::device::
       sub_group_sizes>();
9    std::cout << "Podržane veličine radnih podgrupa : ";
10   for (int i=0; i<sg_sizes.size(); i++) std::cout << sg_sizes[i]
11     << " "; std::cout << "\n";
```

```
12
13    // ispis maksimalne veličine radne podgrupe
14    auto max_sg_size = std::max_element(sg_sizes.begin(), sg_sizes.end());
15    std::cout << "Maksimalna radna podgrupa : " << max_sg_size[0] << "\n";
16
17    q.submit([&](sycl::handler &h) {
18      auto out = sycl::stream(1024, 768, h);
19
20      h.parallel_for(sycl::nd_range<1>(64, 64), [=](sycl::nd_item<1> item)
          [[intel::reqd_sub_group_size(32)]]
21      {
22        // dohvaćanje radne podgrupe trenutačne radne stavke
23        auto sg = item.get_sub_group();
24
25        // ispis informacija o radnoj podgrupi za samo jednu radnu stavku
            unutar podgrupe
26        if (sg.get_local_id()[0] == 0) {
27          out << "ID radne podgrupe: " << sg.get_group_id()[0] <<
28              " od " << sg.get_group_range()[0] << ", veličina="
29              << sg.get_local_range()[0] << "\n";
30        }
31      });
32    }).wait();
33  }
```

Listing 4.8: SYCL program za prikaz informacija o radnim podgrupama

U liniji 19. konstruiran je jednodimenzionalni *nd_range* objekat koji prima dva argumenta : dimenziju cijelog radnog indeksnog prostora i dimenziju radne grupe. Kako su oba argumenta 64, radni prostor će imati samo jednu radnu grupu sa 64 radne stavke. Veličina radne podgrupe postavljena je na 32 koristeći atribut *[[intel::reqd_sub_group_size(32)]]*.

Resursi poput dijeljene memorije i sinkronizacijskih značajki omogućavaju efikasniju komunikaciju radnih stavki unutar radne grupe. Iz navedenih razloga generalno je dobro odabrati najveću podržanu veličinu radne grupe na nekom uređaju. Informacija o najvećoj mogućoj veličini radne grupe dohvaća se pomoću metode
device::get_info<sycl::info::device::max_work_group_size>().

SYCL nudi razne funkcije i algoritme povezane uz rad radne podgrupe, izložene kao funkcije biblioteke. ***Shuffle*** funkcije poput *select_by_group*, *shift_group_left* i *permute_group_by_xor* služe za direktnu komunikaciju unutar podgrupe bez eksplicitnih memorijskih operacija.
Slično funkciji *std::reduce*, SYCL nudi *sycl::joint_reduce* i *sycl::reduce_over_group* funkcije. U nastavku je dan primjer (Listing 4.9) korištenja *sycl::reduce_over_group* za pronalazak maksimalnog elementa radne podgrupe.

```
#include <sycl/sycl.hpp>

static constexpr size_t global_size = 256; // global size
static constexpr size_t work_group_size = 64;  // work-group size

int main() {
    sycl::queue q;
    std::cout << "Izvršava se na : " << q.get_device().get_info<sycl::
        info::device::name>() << "\n";

    // initialize data array using usm
    int *data = sycl::malloc_shared<int>(global_size, q);
    for (int i = 0; i < global_size; i++) data[i] = i;
    for (int i = 0; i < global_size; i++) std::cout << data[i] << " ";
    std::cout << "\n\n";

    q.parallel_for(sycl::nd_range<1>(global_size, work_group_size),
    [=](sycl::nd_item<1> item)
    {
        auto sg = item.get_sub_group();
        auto i = item.get_global_id(0);

        // izdvajanje maksimalnog elementa iz podgrupe koristeći '
            sub_group' algoritam
        // reduce_over_group kao zadnji parametar prima binarnu
            operaciju koja je u ovom slučaju funkcijski objekat
        // sycl::maximum koji vraća veći od dva ulazna elementa
        int result = sycl::reduce_over_group(sg, data[i],
        sycl::maximum<>());

        // rezultat se zapisuje na prvu lokaciju svake podgrupe
        if (sg.get_local_id()[0] == 0) {
          data[i] = result;
        } else {
          data[i] = 0;
        }
    }).wait();

    for (int i = 0; i < global_size; i++) std::cout << data[i] << " ";
    std::cout << "\n";

    free(data, q);
    return 0;
}
```

Listing 4.9: Korištenje *sycl::reduce_over_group* za pronalazak maksimalnog elementa radne podgrupe

4.3 oneAPI DPC++ biblioteka

Uz *oneAPI* DPC++ kompajler, Intel za razvoj SYCL aplikacija nudi i oneAPI DPC++ biblioteku (***oneDPL***) [11]. Biblioteka dolazi integrirana uz instalaciju *oneAPI base Toolkit*-a te služi za pokretanje programa na heterogenim sustavima. Biblioteka se temelji na SYCL standardu uz brojna proširenja koja omogućavaju jednostavniju i kraću sintaksu te dodatne funkcionalnosti. oneDPL podržava korištenje C++ standardnog *API*-a, proširuje standardne STL paralelne algoritme i nudi dodatan *API* za nestandardne algoritme [13].

Sortiranje vektora na uređaju

oneAPI DPC++ biblioteka služi za paralelizaciju i rasterećivanje zadataka na akceleratore. Za razliku od SYCL standardnog koda koji korisnicima pruža više kontrole nad pojedinim akcijama, oneAPI DPC++ biblioteka nudi visok nivo apstrakcije prema korisnicima. Korištenjem biblioteke moguće je sortirati vektor na uređaju (Listing 4.10).

```
...
sycl::queue q(sycl::gpu_selector{});
oneapi::dpl::sort(oneapi::dpl::execution::make_device_policy(q),
    v.begin(), v.end());
...
```

Listing 4.10: Sortiranje vektora na uređaju koristeći oneAPI DPC++ biblioteku

Popunjavanje vektora

Kako biblioteka sakriva i apstrahira mnoštvo *boilerplate* koda, standardne SYCL aplikacije moguće je isprogramirati korištenjem znatno manje linija koda. Za to se koriste algoritmi integrirani u oneAPI DPC++ biblioteku. Jedan primjer 'pokrate' SYCL koda vidljiv je u iduća dva isječka koda (Listing 4.11 i Listing 4.12).

```
#include<sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 4;

int main() {
    queue q;
    std::vector<int> v(N);
    {
        buffer<int> buf(v.data(),v.size());
        q.submit([&](handler &h){
           auto V = buf.get_access<access::mode::read_write>(h);
           h.parallel_for(range<1>(N),[=] (id<1> i){ V[i] = 20; });
        });
```

```
14      }
15      for(int i = 0; i < v.size(); i++) std::cout << v[i] << "\n";
16      return 0;
17  }
```

Listing 4.11: Popunjavanje vektora koristeći standardni SYCL kod

```
1   #include <oneapi/dpl/algorithm>
2   #include <oneapi/dpl/execution>
3   using namespace sycl;
4   constexpr int N = 4;
5
6   int main() {
7       queue q;
8       std::vector<int> v(N);
9
10      // oneDPL koristi oneapi::dpl imenički prostor za sve vlastite
            ekstenzije
11      oneapi::dpl::fill(oneapi::dpl::execution::make_device_policy(q),
12          v.begin(), v.end(), 20);
13
14      for(int i = 0; i < v.size(); i++) std::cout << v[i] << "\n";
15      return 0;
16  }
```

Listing 4.12: Popunjavanje vektora koristeći oneAPI DPC++ biblioteku

U liniji 11. (Listing 4.12) definiran je *execution policy* koji specificira gdje i kako će se pokretati paralelni STL algoritam. Uz to što oneDPL *execution policy* nasljeđuje standardni C++ *execution policy*, dodatno enkapsulira SYCL uređaj ili red.

buffer iteratori

U slučaju poziva višestrukih oneDPL algoritama, memorija se kopira s domaćina na uređaj i nazad za svaki poziv algoritma. Višestruko kopiranje je ukratko prikazano idućim primjerom (Listing 4.13):

```
1   ...
2   // vektor se kopira s domaćina na uređaj, izvršava se for_each algoritam
        i podaci se kopiraju nazad na domaćina
3   oneapi::dpl::for_each(make_device_policy(q), v.begin(), v.end(),
4       [](int &a){ a *= 2; });
5
6   // vektor se ponovno kopira s domaćina na uređaj, izvršava se sort
        algoritam i podaci se kopiraju nazad na domaćina
7   oneapi::dpl::sort(make_device_policy(q), v.begin(), v.end());
```

```
...
```

Listing 4.13: Pozivi višestrukih oneDPL algoritama

Kako bi se izbjeglo nepotrebno kopiranje memorije, potrebno je definirati *buffer*-e i *buffer* iteratore za pristup *buffer*-ima iz STL paralelnih algoritama (Listing 4.14). Za iteratore se koriste *oneapi::dpl::begin* i *oneapi::dpl::end* funkcije koje primaju *buffer* i osiguravaju da memorija ostane na uređaju sve dok se *buffer* ne uništi.

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>

using namespace oneapi::dpl::execution;

int main(){
    sycl::queue q;
    std::cout << "Uređaj : " << q.get_device().get_info<sycl::info::
        device::name>() << "\n";
    std::vector<int> v{2,3,1,4};

    {
        // kreira se buffer
        sycl::buffer buf(v);

        // buffer iteratori osiguravaju da se memorija ne kopira
            bespotrebno
        // između domaćina i uređaja
        auto buf_begin = oneapi::dpl::begin(buf);
        auto buf_end   = oneapi::dpl::end(buf);

        oneapi::dpl::for_each(make_device_policy(q), buf_begin, buf_end,
            [](int &a){ a *= 3; });
        oneapi::dpl::sort(make_device_policy(q), buf_begin, buf_end);
    }

    for(int i = 0; i < v.size(); i++) std::cout << v[i] << "\n";
    return 0;
}
```

Listing 4.14: Korištenje *buffer* iteratora iz oneDPL

USM model kroz oneDPL

oneDPL nudi vrlo jednostavne načine za korištenje biblioteke unutar USM memorijskog modela. Može se koristiti funkcionalnost oneDPL-a kroz obične USM pokazivače (Listing 4.15) ili kroz USM alokatore (Listing 4.16).

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>

using namespace oneapi::dpl::execution;
const int N = 4;
...
sycl::queue q;
int* data = malloc_shared<int>(N, q);

// izvršavanje paralelnog STL algoritma koristeći USM pokazivač
// nije nužno koristiti iteratore već se dovoljno osloniti na
// aritmetiku pokazivača te tako proslijediti početak i kraj podataka
oneapi::dpl::fill(make_device_policy(q), data, data + N, 20);
q.wait();

for (int i = 0; i < N; i++) std::cout << data[i] << "\n";
free(data, q);
...
```

Listing 4.15: Paralelni STL algoritam koristeći USM pokazivač

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>

const int N = 4;

int main() {
    sycl::queue q;

    // kreacija usm alokatora koji omogućava alokaciju u dijeljenoj
       memoriji
    sycl::usm_allocator<int, sycl::usm::alloc::shared> alloc(q);

    // kreacija standardnog vektora pomoću alokatora. Vektor se može
       koristiti na domaćinu i uređaju, to jest alociran je u
       dijeljenoj memoriji
    std::vector<int, decltype(alloc)> v(N, alloc);

    // izvršavanje paralelnog algoritma koristeći alokator, odnosno
       vektor kreiran putem alokatora
    oneapi::dpl::fill(oneapi::dpl::execution::make_device_policy(q),
        v.begin(), v.end(), 20);
    q.wait();
}
```

Listing 4.16: Paralelni STL algoritam koristeći USM alokator

Biblioteka oneDPL nudi dodatno proširenje *oneDPL - Parallel API*, kao podršku za nestandardne algoritme, iteratore i funkcije. Proširanje nudi šest dodatnih paralelnih algoritama, četiri dodatna tipa iteratora i nekoliko pomoćnih klasa.

Histogram koristeći oneDPL

U nastavku je dan program (Listing 4.17) za računanje histograma ulaznih podataka. Ulazni podaci predstavljeni su vektorom koji sadrži cijele brojeve iz $[0, ..., N-1]$, pri čemu se svaki broj može pojaviti više puta ili uopće ne pojaviti. Računanje histograma znači određivanje učestalosti svake vrijednosti unutar skupa $\{0, ..., N\}$ [12].

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/numeric>

#include <sycl/sycl.hpp>
#include <iostream>
#include <random>

/*
IDEJA :
     Neka je ulazni vektor = [4, 0, 2, 1, 4, 0, 3, 2, 4]
    -> sortiranje vektora :
    [0, 0, 1, 2, 2, 3, 4, 4, 4].
    -> upper_bound za svaki broj daje:
    [2, 3, 5, 6, 9].
    -> adjacent_difference računa broj pojavljivanja odgovarajućih
       elemenata u ulaznom vektoru:
    [2-0 = 2, 3 -2 = 1, 5 - 3 = 2, 6 - 5 = 1, 9 - 6 = 3] =
        [2, 1, 2, 1, 3]
    -> Završni histogram je:
    [0 : 2, 1 : 1, 2 : 2, 3 : 1, 4 : 3]
*/

using u_short = unsigned short;

void calculate_histogram(std::vector<u_short> &input, sycl::queue& q,
    int num_bins)
{
    const int N = input.size();
    // zamatanje ulaznih podataka u buffer
    sycl::buffer<u_short> input_buff{input.data(),
        sycl::range<1>(N)};

    // sortiranje svih podataka koristeći oneDPL algoritam sort
    oneapi::dpl::sort(oneapi::dpl::execution::make_device_policy(q),
```

```
34              oneapi::dpl::begin(input_buff), oneapi::dpl::end(input_buff)
                    );
35      q.wait();
36      // kreiranje novog buffera koji predstavlja buckete histograma
37      sycl::buffer<u_short> buckets_buff{sycl::range<1>(num_bins)};
38      auto num_begin_it = oneapi::dpl::counting_iterator<int>{0};
39
40      // za svaki bucket traži se gornja granica pojavljivanja vrijednosti
            indeksa bucket-a u sortiranim ulaznim podacima.
41      // Gornje granice se zapisuju u buckets_buff
42      oneapi::dpl::upper_bound(
43          oneapi::dpl::execution::make_device_policy(q),
44          oneapi::dpl::begin(input_buff),
45          oneapi::dpl::end(input_buff),
46          num_begin_it, num_begin_it + num_bins,
47          oneapi::dpl::begin(buckets_buff));
48      q.wait();
49
50      // Vrijednost pojavljivanja brojeva za određeni bucket odgovaraju
           razlici između dvije susjedne gornje granice
51      oneapi::dpl::adjacent_difference(
52          oneapi::dpl::execution::make_device_policy(q),
53          oneapi::dpl::begin(buckets_buff),
54          oneapi::dpl::end(buckets_buff),
55          oneapi::dpl::begin(buckets_buff));
56      q.wait();
57      std::cout << "Kreirani histogram:\n";
58      {
59          // preko host aksesora pristupa se bufferu kako bi se ispisali
                podaci
60          sycl::host_accessor host_acc(buckets_buff,
61              sycl::read_only);
62          for (int i = 0; i < num_bins; i++)
63          {
64              std::cout << "(" << i << ", " << host_acc[i] << ") ";
65          }
66          std::cout << std::endl;
67      }
68  }
69
70
71  int main(void) {
72      sycl::queue q;
73      constexpr int N = 1000;
74      constexpr short num_bins = 10;
75      std::vector<u_short> input_vector(N);
76
```

```
77      // popunjavanje vektora brojevima od 0 do 9 uniformnom distribucijom
78      std::random_device rd;
79      std::mt19937 gen(rd());
80      std::uniform_int_distribution<> dist(0, num_bins-1);
81      std::generate(input_vector.begin(), input_vector.end(),
82          [&]() { return dist(gen); });
83
84      calculate_histogram(input_vector, q, num_bins);
85  }
```

Listing 4.17: Paralelni STL algoritam koristeći USM alokator

Prikazani složeniji primjeri ilustriraju snagu i fleksibilnost SYCL-a u rješavanju raznolikih problema, od optimiziranih matematičkih operacija poput konvolucije matrica do naprednog korištenja ND-range paradigme. Korištenje *oneAPI DPC++* biblioteke dodatno olakšava implementaciju i optimizaciju heterogenih aplikacija, omogućujući programerima stvaranje prijenosivih i visokoučinkovitih rješenja. Upotreba navedene biblioteke dodatno je pojednostavljena zahvaljujući intuitivnoj sintaksi u skladu s modernim C++ jezikom te integriranim paralelnim algoritmima.

Bibliografija

[1] *AdaptiveCpp*, https://github.com/AdaptiveCpp/AdaptiveCpp.

[2] *SYCL playground*, https://sycl.tech/playground.

[3] https://en.cppreference.com/w/cpp/chrono/duration.

[4] https://github.khronos.org/SYCL_Reference/iface/event.html#event-elapsed-time.

[5] EuroCC National Competence Center Sweden, *Buffer-accessor model vs unified shared memory*, Online dokumentacija, 2021, https://enccs.github.io/sycl-workshop/buffer-accessor-vs-usm/, © Copyright , Copyright 2021, Roberto Di Remigio and individual contributors.

[6] _____, *The GPU hardware and software ecosystem*, Online dokumentacija, 2024, https://enccs.github.io/gpu-programming/2-gpu-ecosystem/.

[7] Gabe Rudy, *Video Graphics and Genomics: A Real Game Changer?*, Online dokumentacija, 2010, https://www.goldenhelix.com/blog/video-graphics-and-genomics-a-real-game-changer/.

[8] Intel Corporation, *Heterogeneous Computing*, Online dokumentacija, 2023, https://intelpython.github.io/DPEP/main/heterogeneous_computing.html, © Copyright 2020–2023, Intel Corporation.

[9] _____, *Get Started with the Intel® oneAPI DPC++/C++ Compiler*, Online dokumentacija, https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/get-started-guide/2025-0/overview.htm, © Copyright , Intel Corporation.

[10] _____, *Intel Tiber AI Cloud*, https://www.intel.com/content/www/us/en/developer/tools/tiber/ai-cloud.html, © Copyright , Intel Corporation.

[11] _____, *Intel® oneAPI DPC++ Library Developer Guide and Reference*, https://www.intel.com/content/www/us/en/docs/onedpl/developer-guide/2022-7/overview.html.

[12] _____, *oneDPL examples*, https://github.com/oneapi-src/oneDPL/tree/main/examples.

[13] _____, *oneDPL Specification*, https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.3-rev-1/elements/onedpl/source/.

[14] _____, *SYCL* Unified Shared Memory Code Walkthrough*, Online dokumentacija, https://www.intel.com/content/www/us/en/developer/articles/code-sample/dpcpp-usm-code-sample.html, © Copyright , Intel Corporation.

[15] _____, *Thread Mapping and GPU Occupancy*, Online dokumentacija, https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2024-2/thread-mapping-and-gpu-occupancy.html, © Copyright , Intel Corporation.

[16] Tao Li, Qiankun Dong, Yifeng Wang, Xiaoli Gong i Yulu Yang, *Dual buffer rotation four-stage pipeline for CPU–GPU cooperative computing*, Soft Computing **23** (2019).

[17] Mladen Jurak, *Primjena paralelnih računala - Konvolucija*, 2022., https://web.math.pmf.unizg.hr/nastava/ppr/html/cuda.html#_konvolucijal.

[18] Abinaya Ramaiyan, Indira Dnvsls i Dhanalakshmi Lanka, *Acoustic based Scene Event Identification Using Deep Learning CNN*, Turkish Journal of Computer and Mathematics Education (TURCOMAT) (2021), 1398–1405.

[19] J. Reinders, *Data Parallel C++*, Apress Berkeley, CA, 2020, https://link.springer.com/book/10.1007/978-1-4842-5574-2.

[20] Steve Hikida - Intel Corporation, *Intel® Compiler First to Achieve SYCL* 2020 Conformance*, 2023, https://www.intel.com/content/www/us/en/developer/articles/technical/compiler-first-full-sycl2020-conformance.html, © Copyright 2020-2023, Intel Corporation.

[21] Teiva Harsanyi, *Go and CPU Caches*, Online dokumentacija, 2020, https://teivah.medium.com/go-and-cpu-caches-af5d32cc5592.

[22] The Khronos Group, *SYCL 2020 Specification*, Online specifikacija, august 2024, https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html.

[23] _____, *SYCL Academy - What is SYCL*, PDF dokument, 2024, https://github.com/codeplaysoftware/syclacademy/blob/main/Lesson_Materials/What_is_SYCL/What_Is_SYCL.pdf.

[24] _____, *An Introduction to SYCL*, Online specifikacija, https://www.khronos.org/sycl/.

[25] _____, *OpenCL - Open Standard for Parallel Programming of Heterogeneous Systems*, Online dokumentacija, https://www.khronos.org/opencl/.

[26] Marilyn Wolf, *High-Performance Embedded Computing (Second Edition)*, Morgan Kaufmann, 2014, https://www.sciencedirect.com/science/article/pii/B9780124105119120015.

Sažetak

U doba rastućih potreba za sustavima visokih performansi nužno je razvijati alate koji će biti u korak s vremenom i pratiti razvoj tehnologije. Heterogeno programiranje u kombinaciji s C++ jezikom donosi ključan korak prema učinkovitoj upotrebi računalnih resursa u tehničkim i znanstvenim područjima poput paralelnog računanja, računalnih simulacija te razvoja modela umjetne inteligencije.
U ovom radu prikazani su temelji heterogenih sustava i heterogenog programiranja, zajedno s detaljnim pregledom SYCL standarda i njegove implementacije kroz *oneAPI DPC++* kompajler. Objašnjen je postupak postavljanja lokalne SYCL okoline te mogućnosti korištenja gotovih okolina u oblaku. Prezentirani su osnovni primjeri koji ilustriraju temljne SYCL koncepte i klase, kao i metode mjerenja vremena izvršavanja SYCL jezgri. Konvolucijom matrica pokazano je kako povezati C++ višedretvenosti i izvršavanje SYCL programa na nekoliko akceleratora. Uvođenjem *ND-Ranges* paradigme objašnjena je razlika globalne i lokalne memorije na uređaju kao i načini sinkronizacije radnih stavki unutar iste podgrupe. Na kraju rada opisana je *oneAPI DPC++* biblioteka koja omogućuje jednostavnije pisanje heterogenih programa kroz viši nivo apstrakcije i integrirane paralelne mehanizme.
SYCL-ova prenosivost koda i mogućnost unificiranog pristupa različitim akceleratorima značajno pojednostavljuju razvoj heterogenih aplikacija, no zahtijevaju pažljivo razumijevanje hardverske arhitekture za postizanje optimalnih rezultata. Iako su izazovi, poput otežanog postavljanja okoline i prilagodbe specifičnom hardveru, i dalje prisutni, SYCL predstavlja snažan alat za razvoj budućih heterogenih sustava. Uz kontinuirani razvoj SYCL standarda i kompajlera koji ga implementiraju, njegova će prisutnost zasigurno rasti, ponajviše zbog smanjene kompleksnosti u odnosu na druge slične standarde te zbog podrške za akceleratore raznih proizvođača.

Summary

In an era of growing demand for high-performance systems, it is essential to develop tools that keep pace with technological advancements. Heterogeneous programming, combined with the C++ language, represents a crucial step towards the efficient use of computational resources in technical and scientific fields such as parallel computing, computational simulations, and the development of artificial intelligence models. This paper presents the fundamentals of heterogeneous systems and heterogeneous programming, along with a detailed review of the SYCL standard and its implementation through the oneAPI DPC++ compiler. The process of setting up a local SYCL environment is explained, as well as the possibilities of using ready-made cloud environments. Basic examples are provided to illustrate fundamental SYCL concepts and classes, as well as methods for measuring the execution time of SYCL kernels. Matrix convolution demonstrates how to link C++ multithreading and the execution of SYCL programs on multiple accelerators. The introduction of the ND-Ranges paradigm explains the difference between global and local memory on the device, as well as synchronization methods for work items within the same subgroup. Finally, the oneAPI DPC++ library is described, which facilitates writing heterogeneous programs through a higher level of abstraction and integrated parallel mechanisms. SYCL's code portability and the ability to provide a unified approach to various accelerators significantly simplify the development of heterogeneous applications, but they require a careful understanding of hardware architecture to achieve optimal results. While challenges, such as the complexity of setting up environments and adapting to specific hardware, remain, SYCL represents a powerful tool for developing future heterogeneous systems. With the continuous development of the SYCL standard and the compilers that implement it, its presence will undoubtedly grow, primarily due to reduced complexity compared to other similar standards and its support for accelerators from various manufacturers.

Životopis

Rođen sam 14. kolovoza 1997. godine u Zagrebu, gdje sam odrastao s roditeljima i bratom. Osnovnu školu započeo sam u Osnovnoj školi Pavleka Miškine, a nakon preseljenja u Veliku Goricu u trećem razredu, nastavio obrazovanje u Osnovnoj školi Eugena Kumičića. Već tada sam se zainteresirao za STEM područja, sudjelujući na brojnim natjecanjima iz različitih predmeta. U 8. razredu sam se plasirao na državno natjecanje iz fizike, gdje sam osvojio 3. mjesto. Uz školu, aktivno sam se bavio nogometom i svirao u školskom bendu.

Nakon osnovne škole, upisao sam XV. gimnaziju u Zagrebu, gdje je moj interes za matematiku dodatno narastao, pa sam nakon završetka srednje škole odlučio upisati preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Od jeseni 2020. godine radim u tvrtci AVL-AST. U listopadu 2022. godine nastavio sam obrazovanje upisavši diplomski studij Matematika i računarstvo na istom fakultetu.

Od malih nogu član sam Odreda izviđača Jarun, a 2013. godine počeo sam voditi aktivnosti za mlađe članove. U jesen 2015. godine preuzeo sam ulogu načelnika udruge, odnosno glavnog programskog koordinatora, tu funkciju obnašam i danas. Sudjelujem u radu Saveza izviđača Hrvatske te Saveza izviđača Zagreba gdje sam jedan mandat bio član Upravnog Odbora. U slobodno se vrijeme bavim sportom i uživam u boravku u prirodi.