Fakultet elektrotehnike, strojarstva i brodogradnje

Optimizacija vremena treniranja duboke neuronske mreže hardverskim i softverskim pristupom

Napredne arhitekture računala - seminarski rad

Mislav Ivanda
Ivan Lukšić
Roko Smoljić
Akademska godina 2021./22

Uvod

Motiv pisanja ovog seminara proizlazi iz teme koje nas zanima, strojno učenje te primjene različitih optimizacijskih opcija, što hardverskih, što softverskih na vrijeme izvršavanja treniranje modela što je jedna od odrednica ovog kolegija.
Vremena treniranja modela su najveće usko grlo prilikom korištenja strojnog učenja jer mogu trajati danima, tjednima, pa i mjesecima. Razvijanjem hardvera, ponajprije grafičkih kartica, ubrzava se treniranje za određene, ne male postotke te posljedično štedi vrijeme i novac uložen u infrastrukturu.
Cilj ovog seminara je usporediti vremena treniranja modela strojnog učenja na različitim grafičkim karticama u što je više moguće sličnim uvjetima. Proći ćemo okruženje u kojem smo radili, što smo radili, usporediti grafičke kartice te pričati o optimizaciji za treniranje u kodu.

Model strojnog učenja

Okruženje

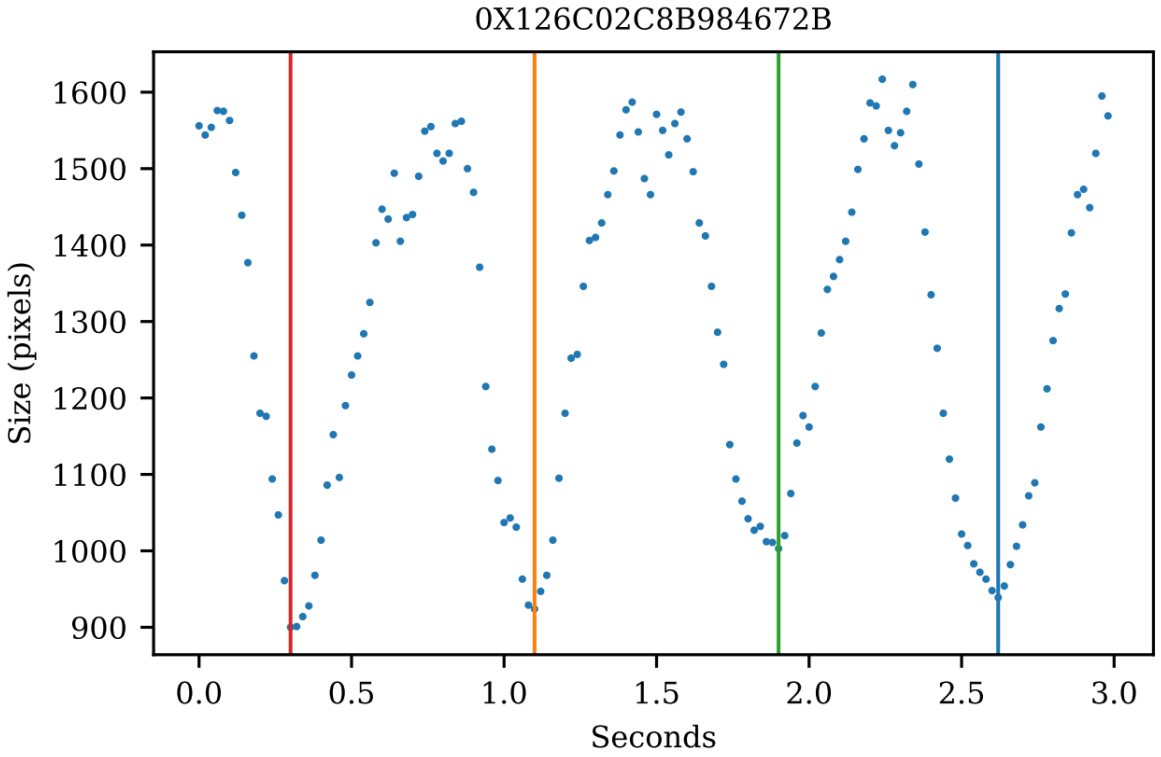
Za vršenje testova treniranja modela smo koristili Googleovo Colab okruženje koje služi upravo za svrhe primjerice treniranja modela strojnog učenja jer daje korisniku na raspolaganje Linux virtualnu mašinu s alociranim grafičkim karticama na kojima prosječan korisnik ne bi mogao trenirati svoj model jer nisu ekonomski pristupačne za nekoga tko ne trenira često modele za strojno učenje jer su same grafičke u ponudi namjenjene za podatkovne centre. To su primjerice nVidia Tesla K80 i T4 kod besplatne verzije Colaba te nVidia Tesla P100 kod Pro verzije Colaba. Colab Pro+ daje mogućnost korištenja nVidia Tesla V100, međutim zbog cijene Pro+ verzije smo ostali na Pro verziji. Kako u Colabu nije moguće samostalno birati grafičku karticu nismo uspjeli, u mnogobrojnim pokušajima, dobiti na korištenje Tesla K80 grafičku karticu već smo testove provodili na Tesla T4 i Tesla P100 grafičkim karticama.

Testovi

Testove smo radili na EchoNet Dynamic datasetu sveučilišta Stanford. Koristeći većinski njihov kod, uz manje promjene, testirajući prosječna vremena treniranja modela po epohama na različitim grafičkim karticama te testiranja promjene različitih parametara u kodu te analizirali promjene vremena treniranja, o čemu više dalje u seminaru.
Testove smo, zbog konzistentnosti radili na 256 snimki i 200 epoha. Parametri su odabrani zato što je u besplatnoj verziji Colaba runtime grafičke kartice maksimalno 12 sati u danu što bi samo s dvostruko većim datasetom premašili te nebi imali mjerodavne podatke za obe kartice. Naknadni testovi vezani za softverske parametre su svi rađeni na Tesla P100 grafičkoj kartici.
Vremena treniranja su bila od 5 do nešto više od 7 sati.

Rezultati

Kao rezultat smo dobili model video segmentacije ehokardiograma te veličine segmentiranog područja u pikselima. Iako je model dosta dobar, takav je jer vrši segmentaciju na snimkama na kojima je treniran i validiran tako da u praksi je potreban znatno veći broj snimaka kako bi model imao širu primjenu.

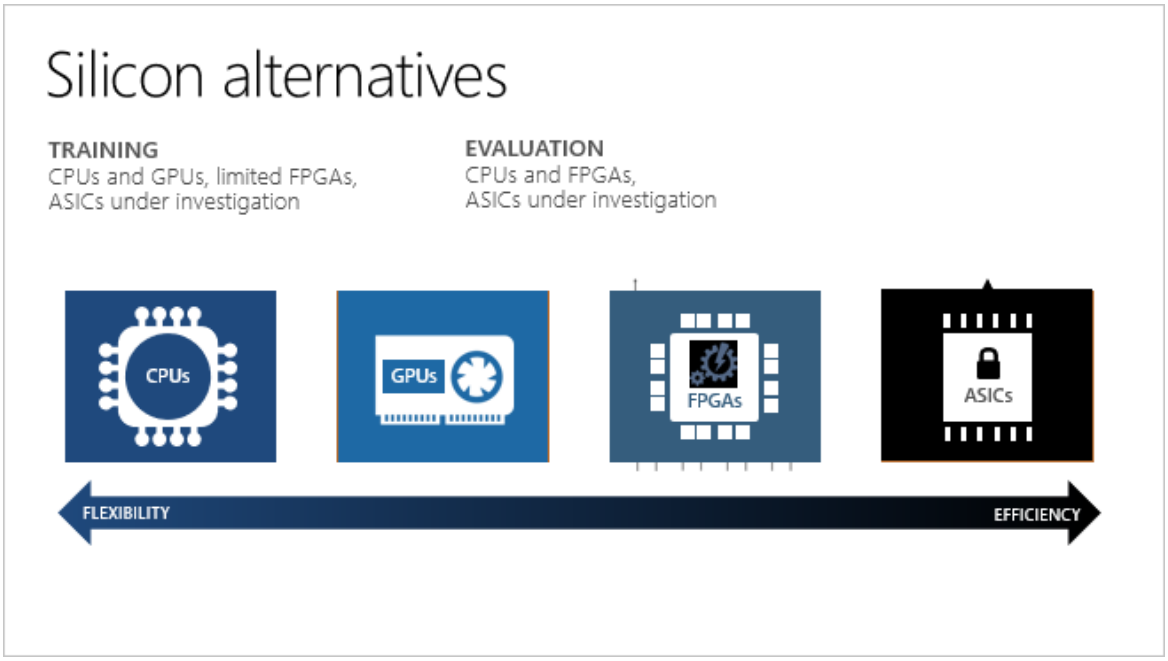


Analizom vremena treniranja po epohama smo dobili da je prosječno vrijeme treniranja modela po epohi za Tesla T4 grafičku karticu **01:41.509 min** dok je to vrijeme za Tesla P100 karticu **01:32.815 min.** Zašto je tome tako te koje su moguće optimizacije sa softverske strane će se razlagati kroz naredna poglavlja.

Optimizacija treniranja duboke neuronske mreže hardverskim pristupom

Različite hardverske varijante

Duboke neuronske mreže postale su uobičajena praksa u mnogim aplikacijama strojnog učenja. Njihova sposobnost da postignu točnost poput ljudi, pa čak i nadljudske, učinila ih je prekretnicom u povijesti umjetne inteligencije. Međutim, postizanje takve razine točnosti je pretjerano skupo u smislu računalne snage. Moderne arhitekture izvode milijarde operacija s pomičnim zarezom (FLOP), što predstavlja značajan izazov za njihovu upotrebu u praksi. Iz tog razloga, mnoge su tehnike optimizacije razvijene na razini hardvera za obradu ovih modela s visokim performansama i energetskom/energetskom učinkovitošću — bez utjecaja na njihovu točnost.



CPU-ovi su 'mozak' računala koji obrađuju upute za izvođenje niza traženih operacija. Današnji CPU-ovi su iznimno robusni, sadrže milijarde tranzistora, a iznimno su skupi za dizajn i proizvodnju. Sadrže snažne jezgre koje su u stanju podnijeti goleme količine operacija i potrošnju memorije. Ovi procesori podržavaju bilo koju vrstu operacija bez potrebe za pisanjem prilagođenih programa. Međutim, njihova opsežna univerzalnost također znači da sadrže suvišne operacije i logičke provjere. Štoviše, ne iskorištavaju u potpunosti mogućnosti paralelizma dostupne u dubokom učenju.

- Prednosti - isplativ, pogodan za opću namjenu, snažne jezgre, veliki kapacitet memorije
- Nedostaci - nemojte u potpunosti iskoristiti paralelizam, performanse niske propusnosti

GPU je specijalizirana hardverska komponenta dizajnirana za obavljanje mnogih jednostavnih operacija istovremeno. Izvorno, GPU-ovi su bili namijenjeni ubrzavanju grafike za računalnu grafiku u stvarnom vremenu, posebno aplikacije za igre. Opća struktura GPU-a ima neke sličnosti sa strukturom CPU-a; obje pripadaju obitelji prostornih arhitektura. Ali, za razliku od CPU-a, koji se sastoje od nekoliko ALU-ova optimiziranih za sekvencijalnu serijsku obradu, GPU se sastoji od tisuća ALU-ova koji omogućuju paralelno izvođenje ogromne količine jednostavnih operacija. Ovo nevjerojatno svojstvo čini GPU-ove idealnim kandidatom za izvršenje algoritama dubokog učenja. Na primjer, u slučaju aplikacija računalnog vida, struktura operacije konvolucije, gdje se jedan filtar primjenjuje na svaki dio u ulaznoj slici.

Iako su grafički procesori trenutno zlatni standard za obuku dubokog učenja, slika nije toliko jasna kada je riječ o zaključivanju. Potrošnja energije GPU-a onemogućuje njihovu upotrebu na raznim uređajima široke upotrebe. Na primjer, NVIDIA GeForce GTX 590 ima maksimalnu potrošnju energije od 365 W. Ovu količinu energije nikako ne mogu isporučiti pametni telefoni, dronovi i mnogi drugiuređaji. Štoviše, postoji značajno kašnjenje za prijenos s hosta na uređaj, pa se ni paralelizacija ne može u potpunosti iskoristiti, upotreba GPU-a postaje suvišna.

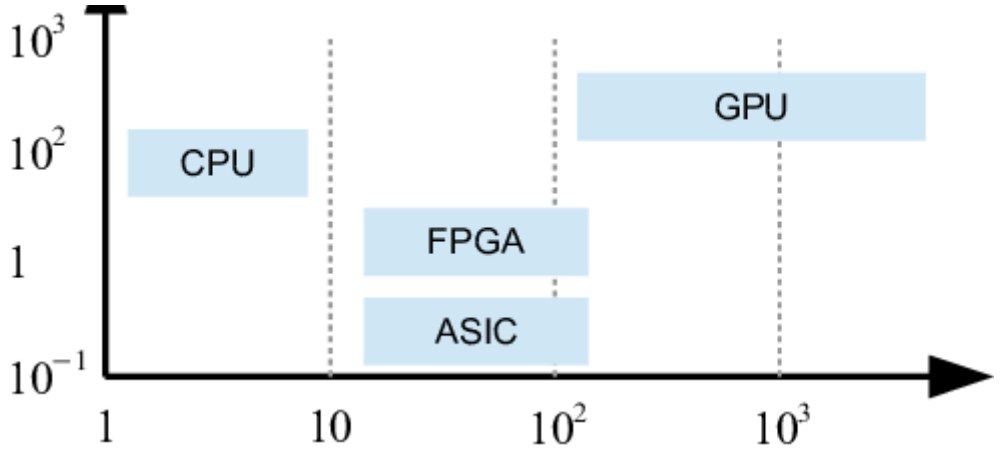
- Prednosti - Izvedba visoke propusnosti, dobro se uklapa u moderne arhitekture (ConvNets)
- Nedostaci - skupo, gladno energije, ima IO latenciju, ograničenja memorije

FPGA je još jedna vrsta specijaliziranog hardvera koji je dizajniran da se konfigurira nakon proizvodnje. Sadrži niz programabilnih logičkih blokova i hijerarhiju konfigurabilnih međuveza koje omogućuju međusobno povezivanje blokova u različitim konfiguracijama. U praksi, korisnik piše kod u jeziku opisa hardvera (HDL), kao što je Verilog ili VHDL. Ovaj program određuje koje se veze izvode i kako se provode pomoću digitalnih komponenti. Posljednjih godina, FPGA su počeli podržavati sve više i više operacija množenja i akumuliranja, omogućujući im implementaciju vrlo paralelnih sklopova. No, s druge strane, njihovo korištenje u praksi moglo bi biti izazovno. Na primjer, znanstvenici većinom treniraju svoje modele koristeći Python biblioteke (TensorFlow, Pytorch, itd.) na GPU-ovima i koriste FPGA samo za zaključivanje odnosno upotrebu. Stvar je, HDL nije pravi programski jezik; to je kod napisan za definiranje hardvera kao što su registri i brojači. To znači da je pretvorba iz Python biblioteke u HDL kod vrlo teška i moraju je obaviti stručnjaci.

- Prednosti - Čip, energetski učinkovit, fleksibilan
- Nedostaci - Izuzetno težak za korištenje, nije uvijek bolji od CPU/GPU-a

Do sada su opisani hardverski sklopovi koji su transformirani da odgovaraju svrhama dubokog učenja, a to im nije primarna namjena. Prilagođeni AI čipovi (ASIC) pokrenuli su brzo rastuću industriju koja gradi prilagođeni hardver za AI. Trenutno više od 100 tvrtki diljem svijeta razvija integrirane sklopove (ASIC) ili sustave na čipu (SoC) za aplikacije dubokog učenja. Štoviše, gigantske tehnološke tvrtke također razvijaju vlastita rješenja kao što su Googleov TPU, Amazonov Inferentia, NVIDIA-in NVDLA, Intelov Habana Labs, itd. Jedina

svrha ovih prilagođenih AI čipova je izvođenje operacija dubokog učenja brže od postojećih rješenja (tj. GPU-ovi).



Na slici iznad je grafički prikazan odnos potrošnje energije (x-os) i podatkovne propusnosti (y-os) različitih tehnologija.

Što GPU čini kvalitetnim GPU-om?

Budući da je GPU tehnologija postala toliko tražen proizvod ne samo za industriju strojnog učenja već i za računalstvo općenito, na tržištu postoji nekoliko GPU-ova za potrošače i poduzeća.

Općenito govoreći, ako tražimo GPU koji se može uklopiti u hardversku konfiguraciju za strojno učenje, tada će neke od važnijih specifikacija za tu jedinicu uključivati sljedeće:

- **Velika propusnost memorije** : Budući da GPU-ovi preuzimaju podatke u paralelnim operacijama, oni imaju potrebu za velikom propusnosti memorije. Za razliku od CPU-a koji radi sekvencijalno (i koji oponaša paralelizam kroz promjenu konteksta, odnosno stanja registara i programskog brojila), GPU može uzeti puno podataka iz memorije istovremeno. Veća propusnost s većom VRAM-om obično je bolja
- **Tenzorske jezgre** : Tenzorske jezgre omogućuju brže množenje matrice u jezgri, povećavajući propusnost i smanjujući kašnjenje. Ne dolaze svi GPU-ovi s tenzorskim jezgrama, ali kako tehnologija napreduje, oni su češći, čak i u GPU-ovima za obične privatne potrošače.
- **Značajnija zajednička memorija** : GPU-ovi s višim L1 predmemorijama mogu povećati brzinu obrade podataka čineći podatke dostupnijima — ali je navedeno rješenje skupo. GPU-ovi s više predmemorije općenito su poželjniji, ali radi se o kompromisu između cijene i performansi.
- **Međusobno povezivanje** : rješenje u oblaku ili lokalno rješenje koje koristi GPU-ove za radna opterećenja visokih performansi obično će imati nekoliko jedinica međusobno povezanih. Međutim, svi GPU-ovi ne surađuju kvalitetno jedni s drugima.

Koristiti obični potrošački ili profesionalni GPU?

Ako tražimo grafičku karticu koja odgovara zahtjevnim zahtjevima softvera profesionalne razine, velika je vjerojatnost da će dobavljač vašeg profesionalnog softvera preporučiti Quadro ili Radeon Pro GPU. Iako ovi "profesionalni" GPU-ovi izgledaju privlačno na papiru, imaju obećavajuću podršku poslovnog ranga i certificirane profesionalne drajvere, njihova visoka cijena nas tjera da dvaput razmislimo treba li nam jedan ili će nam to učiniti obični GPU za potrošače.

Ključne specifikacije	GeForce RTX 3070	RTX A4000	Razlika
CUDA jezgre	5888	6144	256
Tenzorske jezgre	184	192	8
RT jezgre	46	48	2
ROP-ovi	96	96	0
Jedinice teksture	184	192	8
Grafička memorija	8 GB GDDR6	16 GB GDDR6 s ECC	8 GB + ECC
Širina sabirnice memorije	256-bitni	256-bitni	0
Cijena	499 dolara	999 dolara	500 dolara

Suprotno očekivanjima veći trošak za profesionalne GPU-ove ne znači nužno i znatno bolje performanse u odnosu na uobičajene kartice. Umjesto toga, ono što nude profesionalni GPU-ovi su certificirani hardver, optimizirani upravljački programi i opsežna podrška, što ih čini većom investicijom za one s dubokim džepovima koji traže veću pouzdanost ili specifične značajke ekskluzivne za ove GPU-ove.

Budući da se ove kartice koriste za rad s vrlo složenim 3D modelima ili ogromnim skupovima podataka, vrlo je važno da imaju dovoljno memorije. Ako to ne učine, kartica će morati početi mijenjati memorijski sadržaj s vlastitim RAM-om računala, drastično usporavajući rad. Ne radi se samo o čistom kapacitetu jer baš kao i kod poslužitelja i radne stanice RAM-a, profesionalna GPU memorija je posebna vrsta koja ispravlja pogreške. Kada dođe do oštećenja memorije na kartici za igranje, rezultat je obično mala vizualna greška ili sitni nedostatak koji korisnik neće ni primijetiti.

Osim posebne memorije, profesionalni GPU-ovi izgrađeni su prema višim standardima od već visokih standarda za potrošačke kartice. Ispod haube će imati bolje komponente za regulaciju napona, kondenzatore i ploče.

Kartice za igre imaju vrlo snažan naglasak na sirovim performansama. Potrošači se brinu o postizanju najviših grafičkih postavki uz guranje onoliko sličica u sekundi koliko hardver može prikupiti. U ovoj potrazi dobivamo povremenu nestabilnost drajvera ili druge probleme koji mogu nakratko pokvariti iskustvo igranja.

Profesionalne kartice također moraju raditi dobro, ali nikada po cijenu stabilnosti. Iz te temeljne vrijednosti proizlazi kako ove kartice djeluju u stvarnom životu.

NVIDIA T4 vs. NVIDIA TESLA P100

Specifikacije grafičke kartice NVIDIA T4:

- Up to 8.1 TFLOPS single-precision floating-point performance
- Up to 65 TensorTFLOPS of Deep Learning Training Performance; 260 INT4 TOPS of Inference Performance
- NVIDIA "Turing" TU104 graphics processing unit (GPU)
- 2560 CUDA cores, 320 Tensor Cores
- 16GB of GDDR6 GPU memory
- Memory bandwidth up to 320GB/s
- PCI-E x16 Gen3 interface to system
- Passive heatsink only, suitable for specially-designed GPU servers

Specifikacije grafičke kartice NVIDIA TESLA P100:

- Up to 5.3 TFLOPS double- and 10.6 TFLOPS single-precision floating-point performance
- NVIDIA "Pascal" GP100 graphics processing unit (GPU)
- 3584 CUDA cores
- 12GB or 16GB of on-die HBM2 CoWoS GPU memory
- Memory bandwidth up to 732GB/s
- NVLink or PCI-E x16 Gen3 interface to system
- Passive heatsink only, suitable for specially-designed GPU servers

Usporedba performansi ovih dviju grafičkih kartica:

Za test sa operacijama dvostruke preciznosti:

GPU	Tesla T4	Tesla P100
Max Flops (GFLOPS)	253.38	4736.76
Fast Fourier Transform (GFLOPS)	132.60	756.29
Matrix Multiplication (GFLOPS)	249.57	4256.08
Molecular Dynamics (GFLOPS)	105.26	402.96
S3D (GFLOPS)	59.97	161.54

Za test sa operacijama jednostruke preciznosti:

GPU	Tesla T4	Tesla P100
Max Flops (GFLOPS)	8073.26	9322.46
Fast Fourier Transform (GFLOPS)	660.05	1510.49
Matrix Multiplication (GFLOPS)	3290.94	8793.33
Molecular Dynamics (GFLOPS)	572.91	480.02
S3D (GFLOPS)	99.42	295.20

GFLOP (gigaflop) – jedinica koja odgovara brzini obrade milijardu operacija sa pomičnim zarezom u sekundi

Rezultati jednostruke preciznosti pokazuju da Tesla T4 radi dobro za svoju veličinu, iako zaostaje za dvostrukom preciznošću u usporedbi s NVIDIATesla P100 GPU-ovima. Aplikacije koje zahtijevaju dvostruku preciznost nisu prikladne za Tesla T4. Međutim, izvedba za jednostruke preciznosti je poprilično dobra i dovoljno kvalitetna je za izvedbu aplikacija koje su optimizirane za nižu ili mješovitu preciznost.

T4 ima značajne performanse usmjerene na jednostruko/mješovitu preciznost strojnog učenja, s cijenom znatno nižom od većih Tesla GPU-ova. Ono što T4 nedostaje u dvostrukoj preciznosti, nadoknađuje impresivnim rezultatima jednostruke preciznosti. Dostupne performanse jednostruke preciznosti uvelike će se moći pobrinuti za algoritme strojnog učenja s potencijalom primjene na mješovitu preciznost.

Uz potpuno nižu cijenu, T4 također radi na 70 W, u usporedbi s 250+ Watta potrebnih za Tesla P100 GPU. Rad na jednu četvrtinu snage znači da ju je i jeftinije kupiti i jeftinije raditi.

Rad s CUDA u PyTorch

PyTorch je open source okvir za strojno učenje baziran na Pythonu. Omogućuje vam izvođenje znanstvenih i tenzorskih izračuna uz pomoć grafičkih procesorskih jedinica (GPU). Možete ga koristiti za razvoj i obuku neuronskih mreža dubokog učenja pomoću automatske diferencijacije (proces izračuna koji daje točne vrijednosti u stalnom vremenu).

Ključne značajke PyTorch su:

- **Jednostavno sučelje** — uključuje API jednostavan za korištenje koji se može koristiti s Pythonom, C++ ili Javom.
- **Pythonic u prirodi** — jednostavno se integrira s Pythonovim konceptima i omogućuje vam korištenje usluga i funkcionalnosti Pythona.
- **Računalni grafovi** —uključuje mogućnosti za dinamičke računske grafove koje možete prilagoditi tijekom izvođenja.

CUDA je programski model i računalni alat koji je razvila NVIDIA. Omogućuje brže izvođenje računalno intenzivnih operacija paraleliziranjem zadataka na GPU-ovima. CUDA je

dominantni API koji se koristi za duboko učenje iako su dostupne i druge opcije, kao što je OpenCL. PyTorch pruža podršku za CUDA u biblioteci torch.cuda.

PyTorchova CUDA biblioteka omogućuje vam da pratite koji GPU koristite i uzrokuje da se svi tenzori koje kreirate automatski dodijele tom uređaju. Nakon što se tenzor dodijeli, s njim možete izvoditi operacije, a rezultati se također pripisuju istom uređaju.

GPU operacije su prema zadanim postavkama asinkrone kako bi se omogućilo paralelno izvođenje većeg broja izračuna. Asinkrone operacije općenito su nevidljive korisniku jer PyTorch automatski sinkronizira podatke kopirane između CPU-a i GPU-a, odnosno GPU-a i GPU-a. Također, operacije se izvode redoslijedom čekanja. To osigurava da se operacije izvode na isti način kao da su izračuni sinkroni. Ako morate koristiti sinkrone operacije, možete forsirati ovu postavku pomoću varijable okoline CUDA_LAUNCH_BLOCKING=1 . To je od korisno kada na primjer vidimo greške na svojim GPU-ovima. Sinkrono izvršenje osigurava da se pogreške prijavljuju kada se pojave i olakšava identifikaciju koji je zahtjev pokrenuo pogrešku.

Kako koristiti CUDA s PyTorch?

Na samom početku potrebno je provjeriti imamo li potrebne CUDA biblioteke i NVIDIA upravljačke programe te raspoloživ GPU za rad. To možete provjeriti sljedećom naredbom:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Naredbom torch.device( ) postavljamo uređaj na kojem će biti alocirani naši tenzori. Svakom tenzoru koji izradimo dodijeljena je funkcija člana to(). Ova funkcija dodjeljuje navedeni tenzor uređaju koji definiramo, bilo CPU ili GPU. Kada se koristi ovu funkciju, potrebno je proslijediti objekt torch.device kao ulaz. Naš program koristi metodu to() za prebacivanje tenzora kojima će se trenirati naš model, odnosno za segmentaciju ehokardiograma.

```
model = torchvision.models.segmentation.__dict__[model_name](pretrained=pretrained, aux_loss=False)
```

Također, prije samog prosljeđivanja modela GPU-u potrebno je izvršiti metodu *torch.nn.DataParallel( ).* Implementira paralelizam podataka na razini modula. Ova metoda paralelizira primjenu zadanog modela dijeleći ulaz na navedene uređaje dijeljenjem u skupnoj dimenziji (drugi objekti će se kopirati jednom po uređaju). U prolazu naprijed, model se replicira na svakom uređaju, a svaka replika upravlja dijelom ulaza. Tijekom prolaska unatrag, gradijenti iz svake replike se zbrajaju u izvorni modul. U našem kodu ovo je implementirano naredbom:

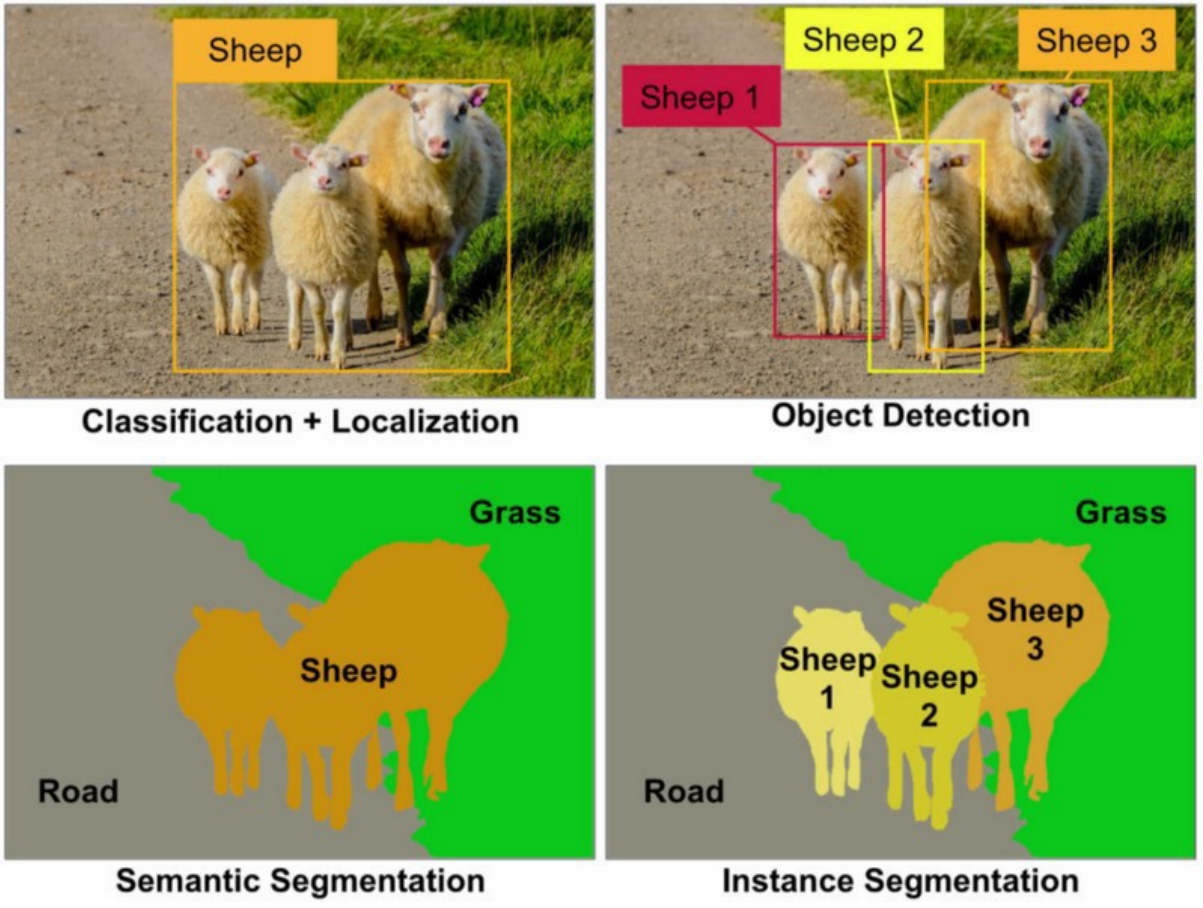
```
if device.type == "cuda":
    model = torch.nn.DataParallel(model)
```

Optimizacija treniranja duboke neuronske mreže softverskim pristupom u PyTorch programskom okviru

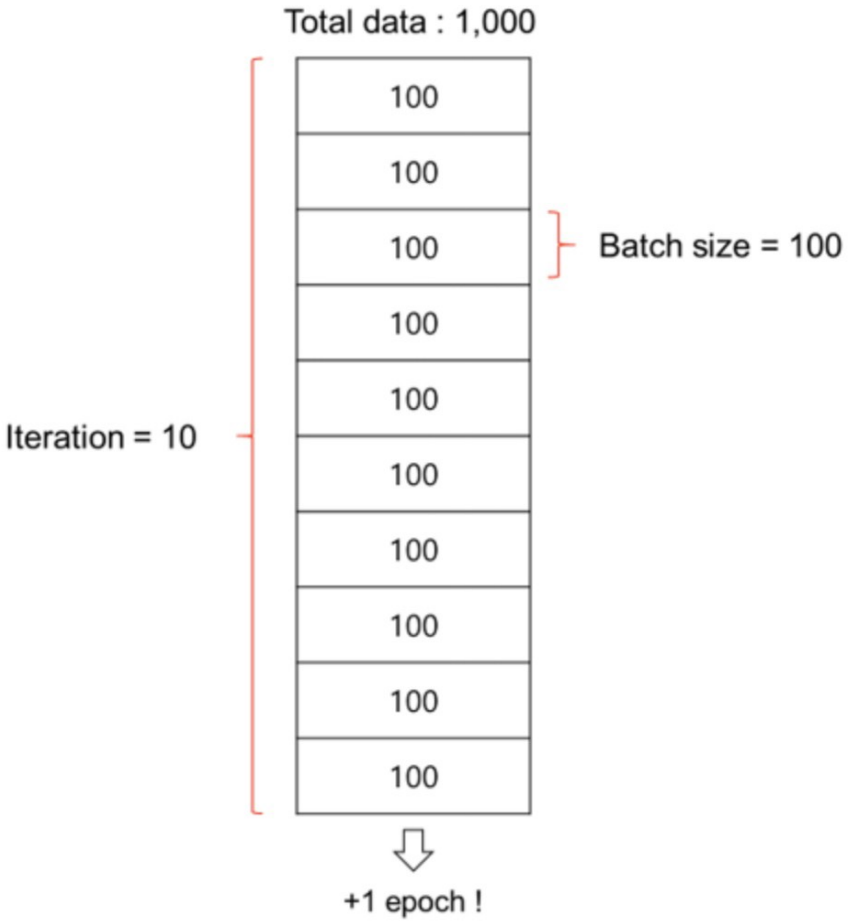
Model i podaci

Središnji dio koncepta strojnog učenja su podaci. Naime, kako bi računalo moglo “naučiti” određeni proces potrebne su velike količine podataka koji predstavljaju “iskustvo računala” u zadanom segmentu koji u kombinaciji s naprednim matematičkim postupcima mogu “naučiti” računalo obavljanju određenog procesa. Kao što i ljudi povećanjem iskustva i treniranjem unaprjeđuju svoje vještine i znanje tako će i računalo s povećanjem iskustva poboljšati svoje sposobnosti u obavljanju procesa za koje ga želimo istrenirati.

Zbog svoje široke primjene i revolucionarnih koncepata, područje strojnog učenja se ovisno o primjeni dijeli na nekoliko disciplina. Ukoliko je riječ o primjeni modela strojnog učenja u kojima su ulazni podaci slike ili videa tada govorimo o području računalnog vida(*Computer vision*). Također, unutar ovog područja razlikujemo različite tipove zadataka koje želimo realizirati, a oni najvažniji prikazani su na sljedećoj slici :



Budući da je cilj modela u našem slučaju segmentacija lijeve komore/klijetke srca na videima ehokardiograma zaključujemo da je riječ o semantičkoj segmentaciji. Također, *dataset* se sastoji od 10030 videa ehokardiograma koji zauzimaju otprilike 8.5 GB. Izuzev superračunala, rijetko koje računalo bi moglo učitati cjelokupni *dataset* i osigurati dovoljno CPU i GPU resursa za izvođenje zahtjevnih operacija nad pikselima.
Stoga zaključujemo da ulazne podatke modela strojnog učenja moramo učitavati “komad po komad” u memoriju te na koncu na GPU. Ovakav princip poznat je pod nazivom *lazy loading*. Prilikom dohvata svake instance koja je pohranjena na sekundarnom memorijskom mediju(disku) u glavnu memoriju moramo raditi pristup memoriji koji pristupa sabirnici preko koje se podaci prenose u blokovima koji ovise o veličini sabirnice.
Pristup memoriji je “skupa” operacija u vidu potrošnje resursa i vremena. Ukoliko dohvaćamo 1 po 1 instancu *dataset-a* tada se može dogoditi situacija da GPU izvršava svoje zadatke brže nego CPU dohvaća podatke s diska što će uzrokovati “prazni hod” GPU-a pa time i lošu iskoristivost sustava.
Stoga zaključujemo da je optimalnije dohvatiti podatke u blokovima od nekoliko instanci kako bi smanjili broj pristupa memoriji i eliminirali mogućnost “praznog hoda” grafičke kartice.
Blok podataka koji se sastoji od instanci koje dohvaćamo u 1 iteraciji naziva se *batch*.
Prolazak kroz sve podatke *dataset-a* naziva se epoha. Sljedeća slika jasno prikazuje razliku između zadanih pojmova:



PyTorch nam omogućava postupak *lazy loading-a* kroz *Dataloader* klasu. Instanciranjem *dataloader* klase dobivamo objekt koji implementira koncept iteratora odnosno u svakoj iteraciji učitava broj instanci definiran veličinom *batch-a* te omogućuje programeru da unutar trenutne iteracije izvodi željene operacije nad podacima prije ulaska u sljedeću iteraciju i učitavanja novih podataka.
Prilikom kreiranja *dataloader* objekta konstruktoru klase proslijeđujemo različite parametre koji se tiču modaliteta učitavanja podataka *dataset-a*. Popis parametara konstruktora prikazan je na sljedećoj slici**:**

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

Fokus ovog dijela seminarskog rada će biti na 2 parametra:

- *num_workers*
- *pin_memory*

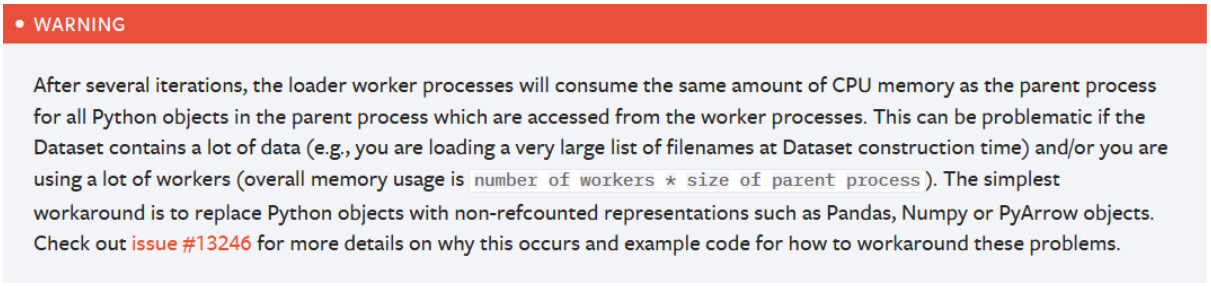
num_workers

Budući da je u posljednjih nekoliko godina razvoj performansi čipova dosegnuo limit jedini način povećanja resursa i ubrzanja izvođenja predstavlja paradigma paralelizacije koja može biti na razini procesora jednog uređaja ili na razini povezivanja procesorskih resursa više računala.
Ukoliko govorimo o paralelizaciji na razini procesora jednog uređaja tada je bitno razlikovati 2 temeljna koncepta za koje se može naslutiti da predstavljaju identične principe, ali to naravno nije slučaj:

- *Multithreading:* paralelizacija se zasniva na alociranju više niti(*thread-ova)* unutar jednog procesa. Svaki proces u početku ima jednu, glavnu nit. Ukoliko se proces sastoji od više neovisnih zadataka njegovo izvršenje je moguće ubrzati na način da svaki zadatak pridjelimo zasebnoj niti. Važno je naglasiti da alokacijom niti ne nastaje novi proces odnosno resursi niti i njen adresni prostor se zadržavaju unutar procesa tj. definira se samo zaseban kontekst.
- *Multiprocessing:* paralelizacija se zasniva na podjeli izvođenja zadatka na više procesa. Alokacija procesa je različita od alokacije niti u smislu da svaki proces ima vlastiti, zasebni adresni prostor. Stoga lako zaključujemo da *multiprocessing* metoda zahtjeva više resursa od *multithreading-a.*

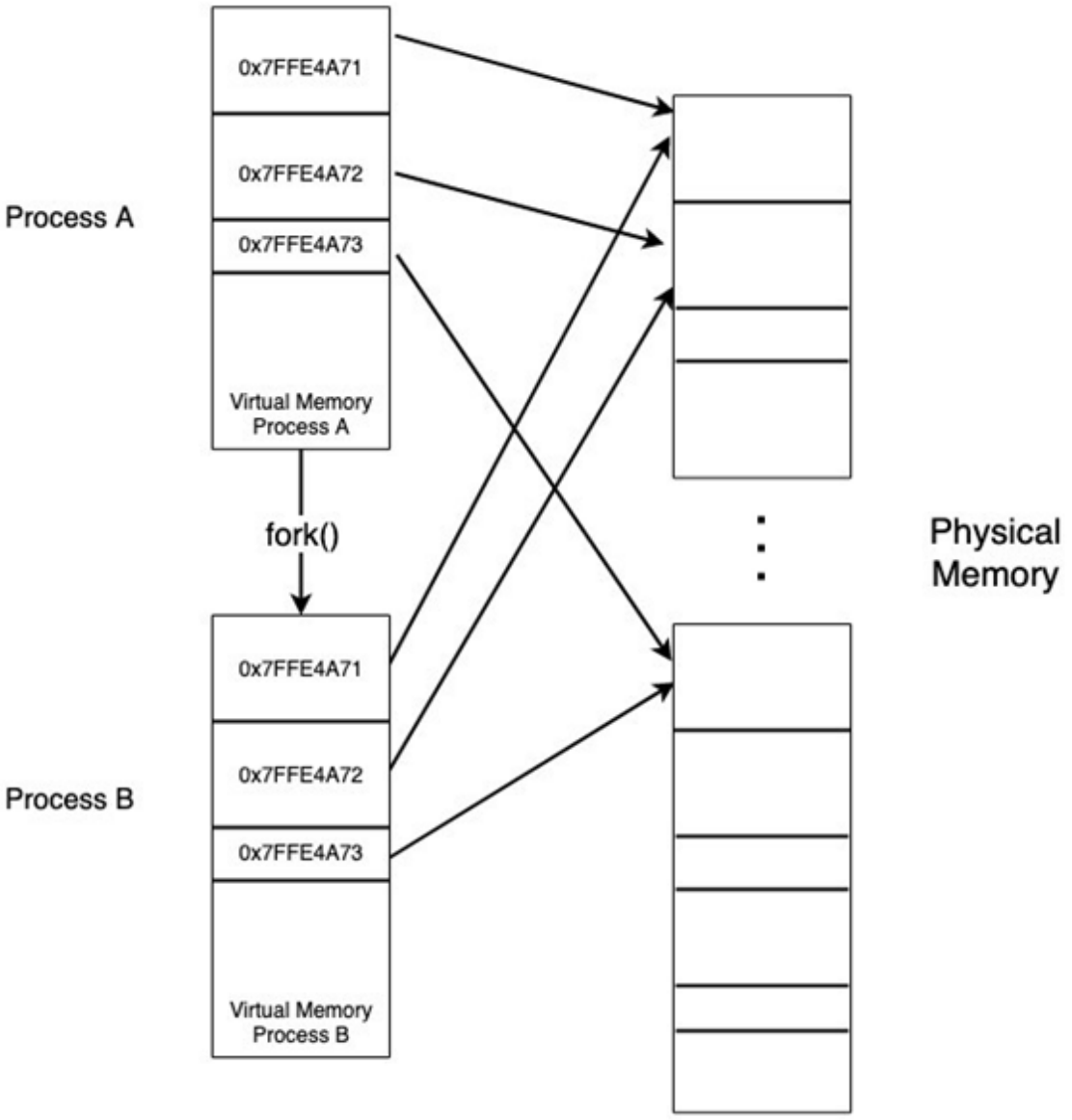
Načini implementacije mehanizma paralelizacije razlikuju se ovisno o programskom jeziku i okruženju na kojem se program odvija. Budući da je riječ o PyTorch programskom okviru iz naziva se lako da zaključiti da će naš fokus biti na Python programskom jeziku.
Python je interpretirani programski jezik pri čemu postoji više verzija implementacije interpretera koji se razlikuju u određenim postupcima prilikom prevođenja Python programa u izvršnu datoteku. Referentna implementacija python interpreter jest CPython napisan u C programskom jeziku.
Međutim, mana ove implementacije jest u tome što nije *thread-safe* odnosno ne omogućava korištenje više niti unutar programa zbog mogućih *race condition-a*.

Stoga CPython implementira *Global Interpreter Lock(GIL) mutex* varijablu koja osigurava da se u jednom trenutku može izvoditi samo 1 nit. Time je zapravo onemogućeno *multithreading* programiranje u Pythonu. Budući da je riječ o referentnoj implementaciji interpretera udio programa koji koriste ovu verziju interpretera je zasigurno velik pa stoga opcija isključivanja GIL-a nije preporučljiva.
Stoga zaključujemo da je jedini način implementacije paralelizacije *multiprocessing* paradigma.
PyTorch nam omogućuje paralelizaciju učitavanja podataka alociranjem dodatnih procesa korištenjem *num_workers* parametra *dataloader-a*.
Predefinirano učitavanje podataka je učitavanje korištenjem jednog, glavnog procesa tzv. *single-process.* Mana ovog prostupa je u blokiranju izvršavanja glavnog procesa koji mora čekati na pristup i dohvat podataka s diska pa je stoga onemogućeno izvođenje ostalih operacija programa.
Ukoliko postavimo num_workers parametar na određeni broj tada će PyTorch koristi multi-process paradigmu prilikom učitavanja podataka na način da će se u svakoj iteraciji dataloader-a alocirati num_workers procesa koji će učitavati podatke.
Iako *multi-process* pristup daje značajno bolje rezultate od *single-process* pristupa, prilikom njegove implementacije u PyTorch-u treba biti jako oprezan zbog moguće pojave problema **zauzeća cjelokupne radne memorije resursa**. Ovaj problem je neočekivan i neuočljiv te je bio predmet velike rasprave prije nekoliko godina kada je uočen(https://github.com/pytorch/pytorch/issues/13246#issuecomment-445770039) do te mjere da je naveden i u samoj dokumentaciji:

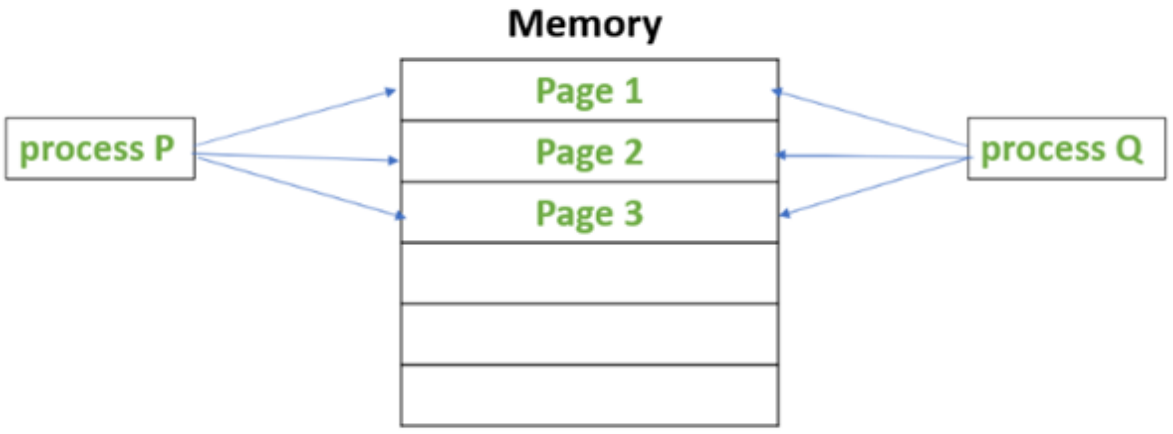


Za razumijevanje razloga koji dovode do ovog problema potrebno je poznavanje operacija koje se odvijaju na *low level* razinama kako Python programskog jezika tako i postupka alokacije procesa na razini operacijskog sustava.

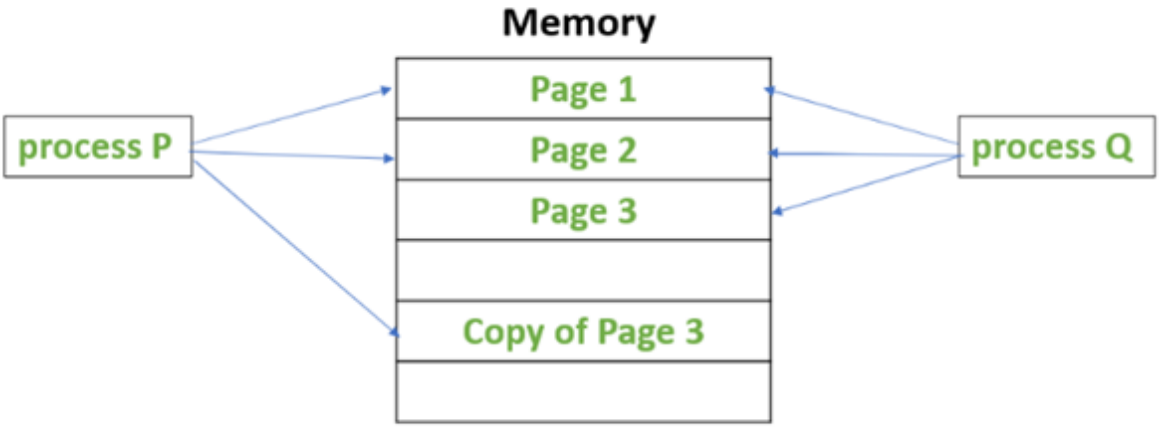
Proces alokacije procesa ovisan je o platformi koju koristimo, no u većini slučajeva hardverski resursi koji služe za izvođenje modela strojnog učenja su temeljeni na Unix operacijskom sustavu.
Unix operacijski sustav pozivom *fork()* metode alocira *child* proces unutar *parent* procesa u kojem je metoda pozvana. Prethodno smo objasnili da je svaki proces nezavisan i ima vlastiti adresni prostor. Međutim, varijable koje su zajedničke *child* i *parent* procesu se ne kopiraju u njihove virtualne adresne prostore već u početku *child* i *parent* proces imaju reference na zajednička područja u memoriji. Ova situacija najbolje je opisana na sljedećoj slici**:**



Međutim, procesi A i B s gornje slike moraju biti neovisni pa stoga u slučaju da proces A promijeni vrijednost varijable čiju referencu također sadrži i proces B, proces B ne bi trebao "vidjeti" odnosno koristiti promijenjenu vrijednost nego vlastitu.
Zadani uvjet se osigurava korištenjem tzv. *copy-on-write* mehanizma. U prethodno opisanom slučaju, *copy-on-write* će procesu A koji mijenja vrijednost zajedničke varijable alocirati dodatni prostor u glavnoj memoriji, pohraniti u njega promijenjenu vrijednost te ažurirati njegovu referencu na novu lokaciju. *Copy-on-write* mehanizam prikazan je na sljedeće 2 slike:



Before process P modifies Page 3



After process P modifies Page 3

Time smo definirali procedure koje se događaju na razini operacijskog sustava prilikom alokacije novih procesa.
Sada je potrebno objasniti utjecaj Python-a na pojavljivanje zadanog problema.
Python je interpretirani jezik i posjeduje *garbage collector.* Jedan od najvažnijih mehanizama koji osigurava rad *garbage collectora* vezan je uz pojam *refcount-ova.*
Uloga *refcount-a* može se naslutiti iz njegovog naziva. *Refcount* određenog objekta ili strukture podataka predstavlja broj referenci u programu koje se odnose na taj objekt.
Važnost *refcount-a* u kontekstu *garbage collectora* jest u tome što u trenutku kada određena varijabla ima *refcount=0 garbage collector* je dealocira.
Zaključujemo da *refcount* mehanizam od Pythona zahtijeva da prilikom dodavanja svake nove reference ili brisanja prethodno postojeće mora ažurirati *refcount* zadane varijable.
Zadano ažuriranje *refcount-a* se tretira kao promjena pa će se stoga u slučaju da je riječ o dijeljenoj varijabli između *parent* i *child* procesa aktivirati *copy-on-write* mehanizam koji će alocirati dodatne memorijske resurse.

Pristup svakoj instanci *map-style dataset-a* u PyTorchu se odvija pozivom __getitem__() metode kojoj se prosljeđuje index koji jedinstveno identificira svaku instancu. Ova metoda se poziva iz *num_workers* alociranih procesa.
Stoga, ukoliko postoji zajednička varijabla koju dijeli __getitem__() metoda ili glavni proces imat ćemo pojavu *copy-on-write* mehanizma prilikom svakog poziva __getitem__() metode budući da u njoj referenciramo dijeljenu varijablu kojoj će se pri tome promijeniti *refcount.*
U svakoj iteraciji ćemo imati aktivaciju *copy-on-write* mehanizma te će se zauzeće resursa memorije povećavati kroz iteracije. Stoga će svaki *num_workers* proces zauzimati jednaku količinu memorije kao i glavni proces. Ukoliko glavni proces sadrži varijable koje zauzimaju veliku količinu memorije dolazi do problema zauzeća svih resursa glavne memorije i rušenja programa.
Nerijetko je riječ upravo o tom slučaju, odnosno glavni proces sadrži varijable koje zauzimaju veliku količinu memorije poput niza koji sadržava podatke o svim instancama *dataset-a.*
Tipičan primjer implementacije *dataset-a* korištenjem PyTorch *Dataset* klase prikazan je na sljedećoj slici**:**

```
class DataIter(Dataset):
     def __init__(self):
          path = "path/to/data"
          self.data = []
          for cls in os.listdir(path):
               for img in os.listdir(os.path.join(path, cls)):
                    self.data.append(os.path.join(path, cls, img))

     def __getitem__(self, idx):
          with Image.open(self.data[idx]) as img:
               img = img.convert('RGB')
               return transforms.functional.to_tensor(img)
```

Primjerice, ukoliko imamo 30000 slika tada će *self.data* niz zauzeti značajan dio memorije. Budući da je zadani niz kreiran u konstruktoru klase koji će biti pozvan prije *dataloader-a(*jer *dataloader* prima *dataset* instancu kao parametar) on pripada glavnom, *parent* procesu iz kojeg će biti izvedeni *num_workers* procesi.
Prilikom pristupanja članu *self.data* niza u *__getitem__* metodi koja će biti pozvana u nekom od *num_workers* procesa promijenit će se *refcount* zadanog člana i aktivirati *copy-on-write* mehanizam.
Srećom postoji nekoliko načina na koje možemo izbjeći zadani problem.
Prvi način temelji se na ideji implementacije strukture podataka koja neće pamtiti *refcount* za svaki svoj član nego će sadržavati jedan *refcount* za cijelu strukturu. U Pythonu podaci poput *list* i *dictionary* tipa pamte *refcount* za svaki član dok primjerice liste implementirane u Python bibliotekama poput NumPy, pandas, PyArrow održavaju samo jedan *refcount* pa stoga predstavljaju način eliminacije zadanog problema.
Drugi način je korištenje PyTorch tenzora koji ne implementiraju *copy-on-write* mehanizam.
Treći način je korištenje Python *multiprocessing* modula kojim eksplicitno navodimo da je određena varijabla zajednička među procesima te zbog toga neće aktivirati copy-on-write mehanizam. U našem konkretnom slučaju s nizom bilo bi potrebno koristiti *multiprocessing.Array()* oznaku.

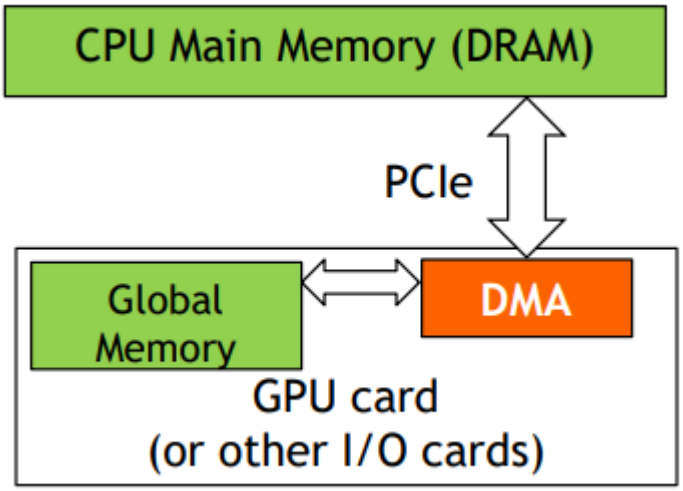
Također korištenje *multiprocess* učitavanja je neophodno prilikom treniranja na velikom *dataset-u* što potvrđuju i podaci prikazani u sljedećoj tablici u kojima je prosječno trajanje epohe prilikom *singleprocess* učitavanja **46%** duže nego prilikom *multiprocess* učitavanja.

Prosječno vrijeme treniranja po epohama:	
Multiprocess *num_workers=4*	*Singleprocess* *num_workers=0*
01:32.815 min	**02:15.667 min**

pin_memory

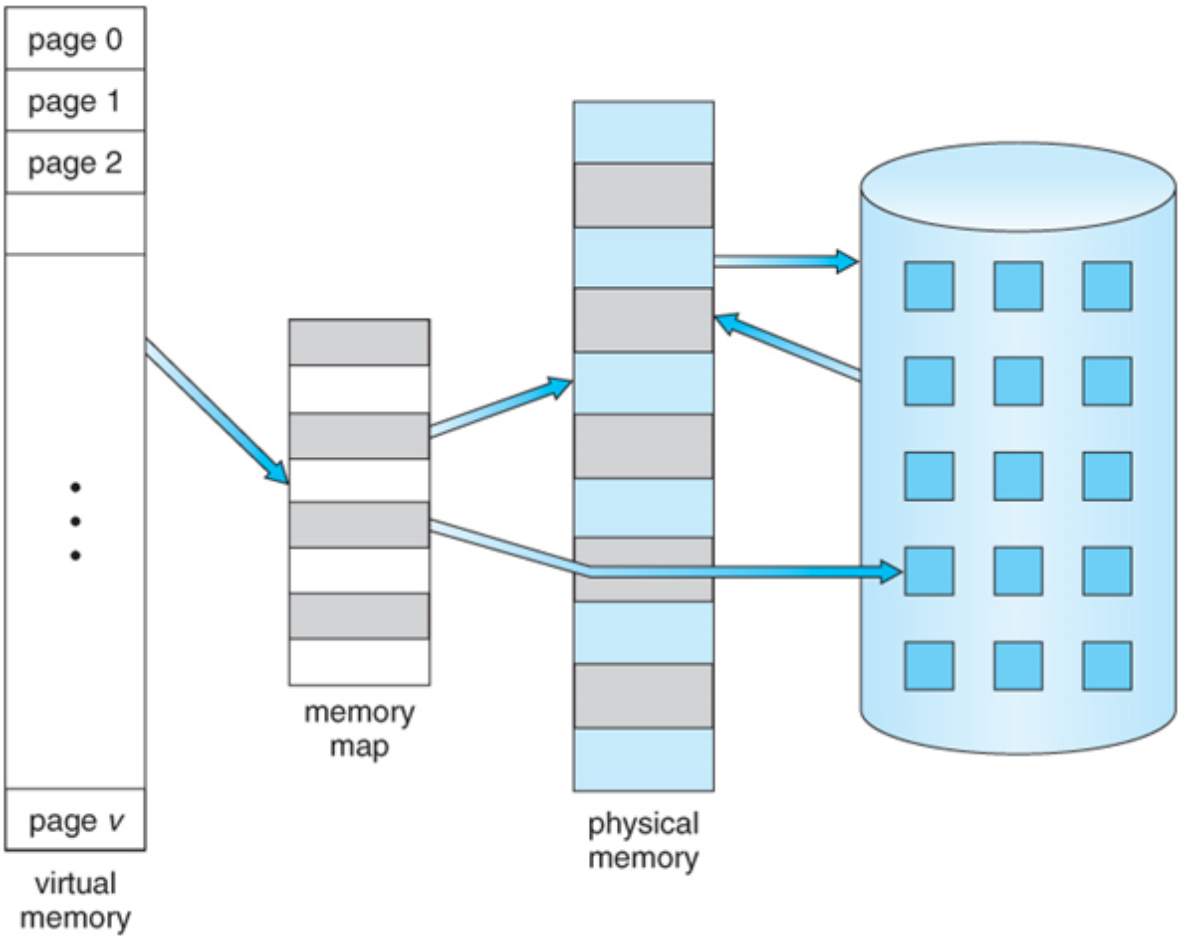
Zbog arhitekture GPU-a koja je pogodna za izvođenje *Single Instruction Multiple Data(SIMD)* operacija procesi treniranja modela strojnog učenja pretežno se odvijaju na GPU. Budući da GPU predstavlja zaseban sustav s vlastitom memorijom i procesorskim jedinicama, a podaci se u glavnu memoriju učitavaju od strane CPU-a potrebno je učitane podatke prenijeti na GPU. Također, na GPU pohranjujemo i model koji je predstavljen sa svojim različitim parametrima.
Kako bi razlikovali sudionike i smjerove prijenosa podataka koriste se definirani nazivi za GPU i CPU. Stoga se GPU označava pojmom *device*, a CPU i glavna memorija pojmom *host*.
PyTorch omogućava prijenose u oba smjera kroz jednostavno sučelje pozivom *.to()* metode koja kao parametar prima odredište prijenosa varijable nad kojom se poziva metoda.
Preostaje pitanje načina na koji se odvija prijenos podataka između *host-a* i *device-a* koji mora biti efikasan zbog nerijetko velike količine podataka koji sudjeluju u prijenosu.
Zbog razvoja kvalitetnih uređaja i CUDA sučelja tvrtka Nvidia zauzima najveći udio tržišta u GPU svijetu te je važno naglasiti da je CUDA sučelje dostupno samo za rad s Nvidia GPU-ovima.
CUDA sučelje definira određenu vrstu standarda kod prijenosa podataka između *host-a* i *device-a* koja se sastoji od 2 važna uvjeta:

- Podaci se prenose korištenjem posebnog *Direct Memory Access(DMA)* sklopa na GPU-u čija je osnovna uloga rasteretiti CPU od upravljanja zadanim memorijskim prijenosom. Stoga za vrijeme prijenosa korištenjem DMA sklopa CPU može neometano izvršavati druge operacije. Zadani mehanizam prikazan je na sljedećoj slici.
- Podaci se iz memorije *host-a* prenose isključivo iz posebnog dijela označenog kao *pinned memory.*



+

Pinned memory predstavlja važan uvjet i dodatni zahtjev za računalni sustav iz čega se da zaključiti da je potreba za njegovim uvođenjem utemeljena na čvrstim i praktičnim razlozima. Da bi u potpunosti razumjeli razlog uvođenja *pinned memory* zahtjeva potrebno se prisjetiti načina na koji operacijski sustav održava i razlikuje različite tipove memorije.
Naime, programi koriste naredbe i podatke koji su logički adresirani relativno u odnosu na početnu adresu programa zbog čega logička adresa ne predstavlja stvarnu, fizičku adresu na disku. Stoga operacijski sustav mora prilikom dohvaćanja naredbi ili operanada programa napraviti pretvorbu između logičke u fizičku adresu.
Efikasna realizacija zadanog mehanizma sastoji se u dijeljenju memorije na blokove i korištenju tablice stranica kao baze znanja o transformaciji logičke u fizičku adresu. Logički adresni prostor se često označava pojmom virtualne memorije. Opisani mehanizam zorno je predočen na sljedećoj slici**:**



S obzirom na zadanu arhitekturu razlikujemo različite moguće situacije:

- Ukoliko je podatak koji zahtijeva program preko logičke adrese prisutan u fizičkoj memoriji tada će podatak biti dohvaćen bez dodatnih operacija
- Ukoliko podatak koji zahtijeva program preko logičke adrese **nije** prisutan u fizičkoj memoriji(*page miss*) tada je potreban dodatni pristup disku iz kojeg će se zadani podatak(točnije njegova stranica) učitati u glavnu memoriju te će se ažurirati tablica stranica
- Ukoliko je fizička memorija popunjena tada učitavanje tražene stranice s diska neće biti moguće. U ovom slučaju potrebno je izbaciti određenu stranicu(*page-out*) iz fizičke memorije kako bi bilo moguće učitati tražene podatke. Odabir stranice koja će biti zamijenjena traženom stranicom s diska može se vršiti na različite načine korištenjem različitih algoritama

S obzirom na prethodno opisani mehanizam mogući su sljedeći scenariji prilikom *host to device* prijenosa:

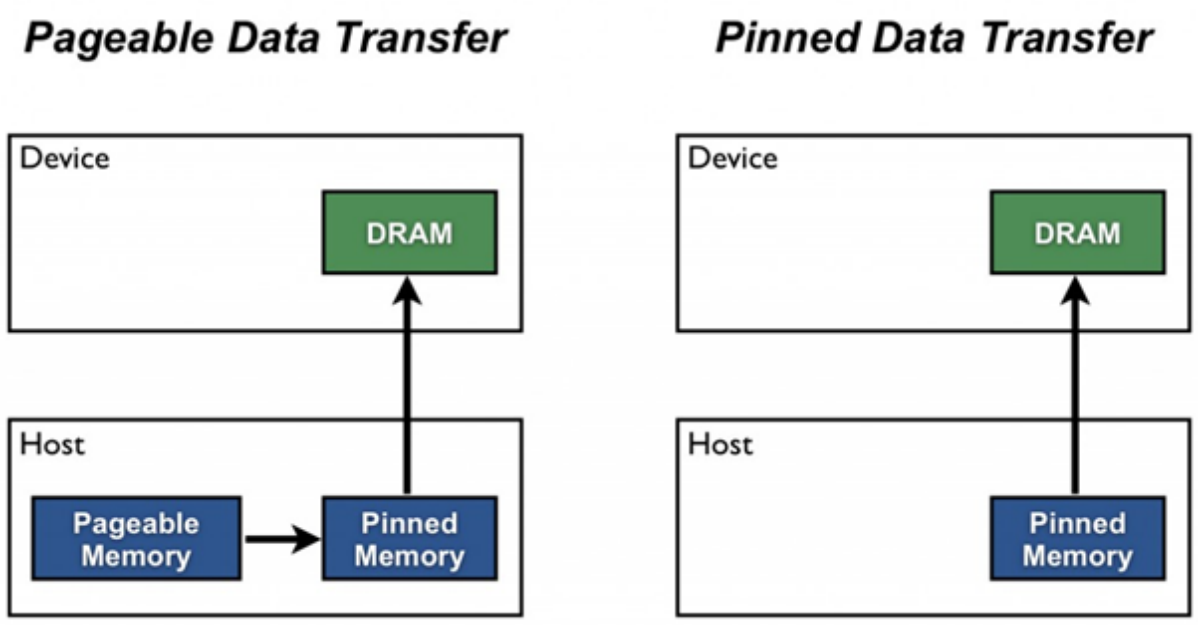
- Nakon što su *host* i *device* utvrdili koji će dijelove memorije biti transferirani, prije početka samog transfera neka od stranica koja upada u prethodno dogovoreni dio memorije može biti *page-outana* te će GPU primiti krive podatke
- Tijekom *host to device* prijenosa neka od stranica koja sudjeluje u prijenosu može biti *page-outana* te će GPU primiti krive podatke

Kako bi izbjegli problem prijenosa krivih podataka na GPU zaključujemo da moramo zabraniti mogućnost *page-outanja* onih stranica koje sudjeluju u *host to device* prijenosu.

Dio memorije *host-a* koji sadrži stranice koje ne mogu biti *page-outane* je tzv. *page locked* dio poznatiji pod nazivom *pinned memory.*
Zaključujemo da je uloga *pinned memory* uvjeta za *host to device* prijenos ključna i nužna za ispravni prijenos podataka.
Stoga ukoliko želimo prenijeti određene podatke iz fizičke memorije *host-a* na *device* potrebno je napraviti 2 memorijska transfera:

1. Prijenos podataka iz *pagable* memorije *host-a* u *pinned memory* dio
2. Prijenos podataka iz *pinned memory* dijela *host-a* na *device* korištenjem *DMA* sklopa

Memorijski transferi su prilično “skupe” operacije pogotovo ukoliko je riječ o velikoj količini podataka koji se mogu prenijeti.
U našem konkretnom slučaju, PyTorch *dataloader* će učitati podatke s diska u *pagable* dio memorije *host-a* te će ih nakon toga morati prenijeti u *pinned memory* dio iz kojeg će biti moguć transfer na GPU.
Međutim, ukoliko dohvat podataka realiziramo na način da učitane podatke odmah učitamo u *pinned memory* dio tada ćemo imati 1 transfer manje što će ubrzati cjelokupni proces *host to device* prijenosa.
Upravo tu mogućnost nam nudi PyTorch *dataloader* kroz postavljanje *pin_memory* parametra konstruktora na vrijednost *true* što će rezultirati smanjenjem dodatnog prijenosa podataka kako je prethodno opisano. Razlika *host to device* prijenosa za slučajeve *pin_memory=false*(lijevi dio slike) i *pin_memory=true*(desni dio slike) prikazana je na sljedećoj slici:



Prethodno pisani teorijski principi poduprijeti su konkretnim empirijskim dokazima prikazanima na sljedećoj tablici pri čemu je u slučaju *pin_memory=false* prosječno trajanje epohe **10%** veće.

Prosječno vrijeme treniranja po epohama:	
pin_memory=false	*pin_memory=true*
01:42.200 min	**01:32.815 min**

Zaključak

Kroz ovaj seminar smo se dotakli tema strojnog učenja, grafičkih kartica, optimizacije koda za strojno učenje među ostalim. Usporedili smo vremena treniranja na dvije grafičke kartice dobro optimizirane za strojno učenje i umjetnu inteligenciju i objasnili zašto je tome tako. Također smo proveli testove te pokazali kako su neke optimizacije u kodu dovele do značajne optimizacije vremena treniranja te objasnili što su te optimizacije i zašto su tako učinkovite.

Kao jednu od odrednica seminara bi htjeli naglasiti kako primjerice razlika od 10% koju smo mi dobili na usporedbi dviju kartica je u praksi značajna. Kod treniranja modela mi smo uzeli 256 od 10 030 snimaka u setu te je treniranje trajalo od 5 i nešto do 7 sati. Kada bi uzeli cijeli set snimaka tih 10% poboljšanja bi značilo dane treniranja manje te dane korištenja grafičke manje što je, primjerice za Googleov servis Colab značajna ušteda u infrastrukturi.

Kroz ovaj seminar smo naučili mnogo o arhitekturi grafičkih kartica i kako su optimizirane za strojno učenje te što sve možemo kao programeri sa softverske strane optimizirati te stekli znanja koja ćemo u budućnosti, nadamo se, što više koristiti.

Literatura

[1] CPU vs GPU in Machine Learning Algorithms: Which is Better?, s Interneta, https://thinkml.ai/cpu-vs-gpu-in-machine-learning-algorithms-which-is-better/
[2] TORCH.CUDA, s Interneta, https://pytorch.org/docs/stable/cuda.html
[3] FPGA for Deep Learning, s Interneta, https://www.run.ai/guides/gpu-deep-learning/fpga-for-deep-learning
[4] NVIDIA TESLA P100, s Interneta, https://www.nvidia.com/en-us/data-center/tesla-p100/
[5] NVIDIA® Tesla™ T4 GPU Computing Accelerator - 16GB GDDR6 - PCIe 3.0 x16 - Passive Cooling, s Interneta, https://www.thinkmate.com/product/nvidia/900-2g183-0000-001
[6] What's the difference between Consumer and Pro graphics cards?, s Interneta, https://www.redsharknews.com/technology-computing/item/318-what-s-the-difference-between-consumer-and-pro-graphics-cards
[7] DataLoader num_workers > 0 causes CPU memory from parent process to be replicated in all worker processes #13246, s Interneta, https://github.com/pytorch/pytorch/issues/13246
[8] Single- and Multi-process Data Loading , s Interneta, https://pytorch.org/docs/stable/data.html#single-and-multi-process-data-loading
[9] GlobalInterpreterLock, s Interneta, https://wiki.python.org/moin/GlobalInterpreterLock
[10] Copy on Write, s Interneta, https://www.geeksforgeeks.org/copy-on-write/
[11] How Does Python Code Run: CPython And Python Difference, s Interneta, https://www.c-sharpcorner.com/article/why-learn-python-an-introduction-to-python/
[12] Understanding Reference Counting in Python, s Interneta, https://towardsdatascience.com/understanding-reference-counting-in-python-3894b71b5611
[13] Page-Locked Host Memory for Data Transfer, s Interneta, https://leimao.github.io/blog/Page-Locked-Host-Memory-Data-Transfer/
[14] Module 14 – Efficient Host-Device Data Transfer, s Interneta, https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwirsMHB1LT4AhWC8rsIHRCuC5sQFnoECAcQAw&url=https%3A%2F%2Fengineering.purdue.edu%2F~smidkiff%2Fece563%2FNVidiaGPUTeachingToolkit%2FMod14DataXfer%2FMod14DataXfer.pdf&usg=AOvVaw3BIkpPbpyqs_hPKd0_-PBi
[15] EchoNet-Dynamic, s Interneta, https://echonet.github.io/dynamic/