SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Izvedba diferencijabilnih transformacija pod programskim okvirima CUDA i PyTorch

Filip Oreč
Voditelj: *Siniša Šegvić*

Zagreb, svibanj 2020.

SADRŽAJ

1. Uvod

Razvojem hardvera koji podržava paralelnu obradu podataka u zadnjih nekoliko godina postignut je veliki napredak na području dubokog učenja. Kao posljedica toga razvili su se i mnogi programski okviri za rad na tom području. Ovaj seminarski rad razmatra implementaciju diferencijabilnih transformacija u programskom okviru PyTorch.

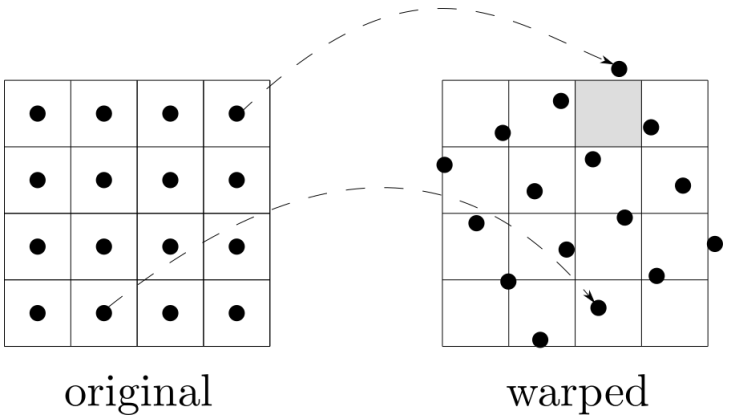
PyTorch je programski okvir dubokog učenja otvorenog koda koji se koristi u razvoju aplikacija računalnog vida. Prvenstveno je razvijen od strane Facebook-ovog istraživačkog laboratorija za umjetnu inteligenciju (FAIR). Postoji sučelje prema programskom jeziku C++ i sučelje prema programskom jeziku Python na kojemu je veći naglasak. Jedan od glavnih modula PyTorch-a je njegov *Autograd* modul koji nudi mogućnost automatske diferencijacije. Druga važna mogućnost je iskorištavanje hardvera koji podržava paralelnu obradu podataka (GPU). Kako bi to ostvario PyTorch se oslanja na programski okvir CUDA.

CUDA je platforma za paralelno računanje koju je razvila Nvidia i koja omogućuje jednostavno korištenje grafičkog procesora (GPU) za računanje opće namjene. CUDA omogućava programerima da ubrzaju računalno zahtjevne aplikacije iskorištavanjem snage GPU-a za paralelni dio računanja.

Stoga ovaj seminarski rad razmatra implementaciju vlastitih diferencijabilnih transformacija i njihovu integraciju s *Autograd* modulom PyTorch-a kako bi se iskoristilo svojstvo automatske diferencijacije. Kako bi se iskoristila snaga GPU-a rad razmatra i implementaciju navedenih transformacija u programskom okviru CUDA. Također, provedena je studija slučaja na softmax splatting transformaciji za video interpolaciju (Niklaus i Liu, 2020).

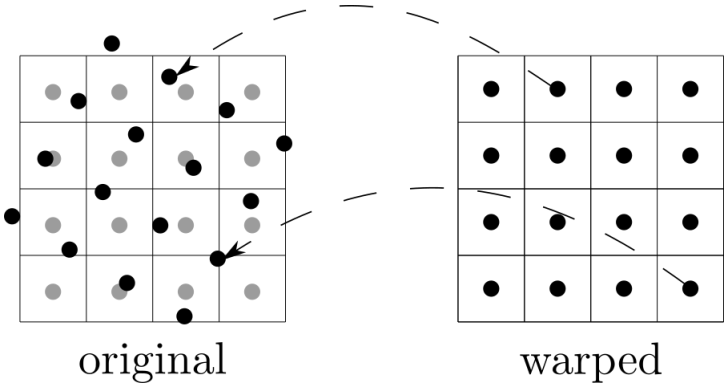
2. Unatražne i unaprijedne transformacije slike

Video interpolacija je klasičan problem računalnog vida s puno praktičnih primjena. Pristupi za ovaj problem mogu se kategorizirati kao one bazirane na toku (engl. *flow-based*), bazirane na jezgri (engl. *kernel-based*) i fazno bazirane (engl. *phase-based*). Softmax splatting (Niklaus i Liu, 2020) je diferencijabilna transformacija bazirana na optičkom toku (engl. *optical flow*) koja je detaljnije opisana u poglavlju 3. Čest pristup ovih metoda je procjena optičkog toka $F_{t\to0}$ i $F_{t\to1}$ između dvije ulazne slike I_0 i I_1 iz perspektive slike I_t koju treba stvoriti. Rezultantna slika I_t može se tada dobiti unatražnim savijanjem (engl. *backward warping*) slike I_0 s obzirom na optički tok $F_{t\to0}$ i slike I_1 s obzirom na optički tok $F_{t\to1}$. Ova metoda otežava korištenje gotovog procjenitelja optičkog toka i sprječava stvaranje nove slika za proizvoljan t na prirodan način. Niklaus i Liu (2020) predlažu korištenje unaprijednog savijanja (engl. *forward warping*) slike I_0 s obzirom na optički tok $t \cdot F_{0\to1}$ i slike I_1 s obzirom na optički tok $(1-t) \cdot F_{1\to0}$, čime se izbjegava aproksimacija tokova $F_{t\to0}$ i $F_{t\to1}$.



Slika 2.1: Metoda unaprijednog savijanja. Za svaki piksel izvorišne slike izračunava se njegova nova pozicija u odredišnoj slici. Problem nastaje jer izračunate pozicije piksela u odredišnoj slici nisu cijeli brojevi. Može doći do pojave rupa, označeno sivo, te mapiranja više piksela izvorišne slike na istu poziciju odredišne slike.

Problem kod ove metode je što proces mapiranja piksela izvorišne slike u odredišnu sliku nije dobro definiran kad nove pozicije piksela nisu cijeli brojevi, tj. može se dogoditi da se



Slika 2.2: Metoda unatražnog savijanja gdje je svaki piksel iz odredišne slike uzorkovan iz izvorišne slike. Za svaki piksel odredišne slike izračunava se pozicija u izvorišnoj slici, te se uzorkuje taj piksel izvorišne slike, što rješava nastanak rupa. Još važnije, ponovno uzorkovanje slike s pozicija koje nisu cijeli brojevi je dobro izučen problem i visoko kvalitetni filtri koji kontroliraju aliasing se mogu koristiti.

više piksela izvorišne slike mapira u istu poziciju odredišne slike. Softmax splatting izvodi diferencijabilno unaprijedno savijanje i rješava taj problem.

3. Unaprijedna transformacija slike zasnovana na funkciji softmax

Unatražno savijanje je uobičajena tehnika koja ima široku primjenu, za razliku od unaprijednog savijanja koje nema toliku primjenu zbog toga što se više različitih piksela iz I_0 mogu mapirati na istu poziciju u I_t. Postoje razni pristupi koji rješavaju ovu nejednoznačnost. Kako bi se objasnio softmax splatting u nastavku će se prvo obraditi uobičajeni pristupi za rješavanje te nejednoznačnosti.

3.1. Summation splatting

Prvi od njih je summation splatting. Vrlo intuitivan pristup za rješavanje gore spomenute nejednoznačnosti je da se zbroje svi doprinosi. Summation splating $\overrightarrow{\Sigma}$ definira se sljedećim izrazima:

$$\mathbf{u} = \mathbf{p} - (\mathbf{q} + F_{0\to t}[\mathbf{q}]) \tag{3.1}$$

gdje $\mathbf{q}$ predstavlja poziciju piksela izvorišne slike I_0, a $\mathbf{p}$ predstavlja poziciju piksela odredišne slike I_t, te $F_{0\to t}[\mathbf{q}]$ predstavlja optički tok. Na ovaj način se izračunava udaljenost dobivene necjelobrojne pozicije piksela $(\mathbf{q} + F_{0\to t}[\mathbf{q}])$ od cjelobrojne pozicije piksela $\mathbf{p}$ u odredišnoj slici I_t.

$$b(\mathbf{u}) = max(0, 1 - |\mathbf{u}_x|) \cdot max(0, 1 - |\mathbf{u}_y|) \tag{3.2}$$

Izraz 3.2 predstavlja bilinearnu jezgru koja izračunava doprinos piksela $\mathbf{q}$ iz izvorišne slike I_0 za piksel $\mathbf{p}$ odredišne slike I_t. Piksel $\mathbf{q}$ će doprinositi vrijednosti piksela $\mathbf{p}$ samo ako je udaljenost izračunate necjelobrojne pozicije piksela $(\mathbf{q} + F_{0\to t}[\mathbf{q}])$ i cjelobrojne pozicije piksela $\mathbf{p}$ manja od 1, tj. piksel $\mathbf{q}$ će doprinositi vrijednosti samo četiri piksela odredišne slike I_t koja okružuju izračunatu necjelobrojnu poziciju $(\mathbf{q} + F_{0\to t}[\mathbf{q}])$.

$$I_t^{\Sigma}[\mathbf{p}] = \Sigma_{\forall \mathbf{q} \in I_0} b(\mathbf{u}) \cdot I_0[\mathbf{q}] \tag{3.3}$$

$$\overrightarrow{\Sigma}(I_0, F_{0\to t}) = I_t^{\Sigma} \tag{3.4}$$

I_t^{Σ} predstavlja izračunatu vrijednost odredišne slike I_t na poziciji $\mathbf{p}$. Računa se kao suma svih doprinosa iz I_0 do I_t prema $F_{0\to t}$ podložnih bilinearnoj jezgri b. Ova transformacija se kraće zapisiva pomoću izraza 3.4.

Na slici 3.1 dole lijevo vidi se da ovaj pristup stvara nekonzistentnost u svjetlini kod preklapajućih područja kao prednji dio automobila. Nadalje, zbog bilinearne jezgre nastaju pikselu u I_t koji primaju samo dio doprinosa piksela I_0 što ponovo dovodi do nekonzistentnosti u svjetlini kao na cesti. Međutim ovaj se pristup koristi kao osnova za ostale pristupe. Derivacije za x komponentu su definirane kao:

$$\frac{\partial I_t^{\Sigma}[\mathbf{p}]}{\partial I_0[\mathbf{q}]} = b(\mathbf{u}) \tag{3.5}$$

$$\frac{\partial I_t^{\Sigma}[\mathbf{p}]}{\partial F_{0\to t}^{x}[\mathbf{q}]} = \frac{\partial b(\mathbf{u})}{\partial F_{0\to t}^{x}} \cdot I_0[\mathbf{q}] \tag{3.6}$$

$$\frac{\partial b(\mathbf{u})}{\partial F_{0\to t}^{x}} = max(0, 1 - |\mathbf{u}_y|) \cdot \begin{cases} 0, \text{if } |\mathbf{u}_x| \geq 1 \\ -sgn(\mathbf{u}_x), \text{ else} \end{cases} \tag{3.7}$$

i analogno za y komponentu optičkog toka $F_{0\to t}$. U poglavlju 4 opisana je PyTorch implementacija summation splatting $\overrightarrow{\Sigma}$ transformacije pisana u CUDA-i.

3.2. Average splatting

Kako bi se riješila nekonzistentnost sa svjetlinom I_t^{Σ} se mora normalizirati. Može se iskoristiti definicija $\overrightarrow{\Sigma}$ i odrediti average splatting $\overrightarrow{\Phi}$ kao:

$$\overrightarrow{\Phi}(I_0, F_{0\to t}) = \frac{\overrightarrow{\Sigma}(I_0, F_{0\to t})}{\overrightarrow{\Sigma}(\mathbf{1}, F_{0\to t})} \tag{3.8}$$

Na slici 3.1 gore desno vidi se kako ovaj pristup rješava nekonzistentnosti sa svjetlinom i održava izgled I_0. Ali ovaj pristup izračunava prosječnu vrijednost preklapajućih regija, kao prednji dio automobila s travom u pozadini.

3.3. Linear splatting

Kako bi se što bolje odvojile preklapajuće regije može se I_0 linearno otežati s maskom težina Z i definirati linear splatting $\overrightarrow{*}$ kao:

$$\overrightarrow{*}(I_0, F_{0\to t}) = \frac{\overrightarrow{\Sigma}(Z \cdot I_0, F_{0\to t})}{\overrightarrow{\Sigma}(Z, F_{0\to t})} \tag{3.9}$$

gdje Z može biti dubina svakog piksela. Na slici 3.1 dole desno vidi se da ovaj pristup bolje odvaja automobil od trave u pozadini. Ali ova transformacija nije invarijantna na translaciju s

Slika 3.1: Za dvije ulazne slike I_0 i I_1 i procjenu optičkog toka $F_{0\to1}$ slika prikazuje primjer unaprijednog savijanja slike I_0 do I_t prema $F_{0\to t} = t \cdot F_{0\to1}$ s četiri različita pristupa (Niklaus i Liu, 2020).

obzirom na Z. Ako Z predstavlja recipročne vrijednosti dubine onda će biti jasna odvojivost ako je automobil na $Z = 1/1$ i pozadina na $Z = 1/10$. Ako je automobil na $Z = 1/101$ i pozadina na $Z = 1/110$ onda će se samo izračunavati prosječna vrijednost iako su jednako udaljeni.

3.4. Softmax splatting

Kako bi se jasno odvojila preklapajuća područja s invarijantnošću na translaciju Niklaus i Liu (2020) predlažu softmax splatting $\overrightarrow{\sigma}$ definiran kao:

$$\overrightarrow{\sigma}(I_0, F_{0\to t}) = \frac{\overrightarrow{\Sigma}(\exp(Z) \cdot I_0, F_{0\to t})}{\overrightarrow{\Sigma}(\exp(Z), F_{0\to t})} \tag{3.10}$$

gdje Z može odgovarati dubini svakog piksela. Na slici 3.1 gore lijevo vidi se da je prednji dio automobila jasno odvojen od trave u pozadini. Vidi se da je ova transformacija slična softmax funkciji, te je invarijantna na translaciju s obzirom na Z, ali nije invarijantna na skaliranje.

4. Implementacija u programskim okvirima PyTorch i CUDA

PyTorch pruža jako puno operacija vezanih uz neuronske mreže, algebru s tenzorima, obradu podataka i druge svrhe. Ipak te operacije mogu biti nedovoljne za određene implementacije, te se stvara potreba za implementiranjem vlastitih operacija i njihovom integracijom s PyTorchem. Kako bi se iskoristila snaga GPU-a te operacije se mogu implementirati u CUDA-i, te se zatim integrirati u PyTorch. CUDA dolazi s programskom podrškom koja omogućava korištenje C++-a kao programskog jezika visoke razine. Zbog toga i brzine izvršavanja C++ koda koristi se sučelje PyTorcha prema C++-u za implementaciju vlastitih operacija, koje se zatim integriraju s PyTorchovim sučeljem prema Pythonu. U nastavku će se razmatrati implementacija transformacije summation splatting $\overrightarrow{\Sigma}$ opisane u 3.1 u C++-u i CUDA-i.

4.1. Implementacija C++ i CUDA operacija

Općeniti pristup pisanju CUDA proširenja je prvo pisanje C++ datoteke koja definira funkcije koje će biti pozivane iz Pythona, te povezivanje tih funkcija s Pythonom. Također se deklariraju i funkcije koje se definiraju u CUDA (`.cu`) datoteci.

```
#include <torch/extension.h>

torch::Tensor sumsplat_update_output_cuda(
    torch::Tensor input,
    torch::Tensor flow
);
torch::Tensor sumsplat_update_gradinput_cuda(
    torch::Tensor input,
    torch::Tensor flow,
    torch::Tensor grad_output
);
torch::Tensor sumsplat_update_gradflow_cuda(
    torch::Tensor input,
    torch::Tensor flow,
```

```
    torch::Tensor grad_output
);
torch::Tensor sumsplat_update_output(
    torch::Tensor input,
    torch::Tensor flow
) {
  // Checking input arguments...
  return sumsplat_update_output_cuda(input, flow);
}
torch::Tensor sumspalt_update_gradinput(
    torch::Tensor input,
    torch::Tensor flow,
    torch::Tensor grad_output
) {
  // Checking input arguments...
  return sumsplat_update_gradinput_cuda(input, flow, grad_output);
}
torch::Tensor sumspalt_update_gradflow(
    torch::Tensor input,
    torch::Tensor flow,
    torch::Tensor grad_output
) {
  // Checking input arguments...
  return sumsplat_update_gradflow_cuda(input, flow, grad_output);
}
```

Listing 4.1: Primjer C++ datoteke

Naredba `#include <torch/extension.h>` uključuje *ATen* biblioteku, koja je primarni API za računanje s tensorima, *pybind11* biblioteku, koja omogućava povezivanje s Pythonom i ostala zaglavlja (engl. *headers*) koja upravljaju detaljima interakcije između biblioteka *ATen* i *pybind11*. Prve tri funkcije s nastavkom `_cuda` predstavljaju CUDA funkcije definirane u CUDA (`.cu`) datoteci koje pozivaju CUDA kernele. Ostale C++ funkcije provjeravaju ulazne argumente i prosljeđuju ih CUDA funkcijama. Primarni tip podataka za sva računanja je `torch::Tensor`. Funkcija `sumsplat_update_output` predstavlja unaprijedni prolaz, dok funkcija `sumsplat_update_gradinput` i funkcija `sumsplat_update_gradflow` predstavljaju unatražni prolaz, tj. računaju gradijente s obzirom na ulaznu sliku I_0 i optički tok $F_{0\to t}$. U nastavku je dan primjer implementacije CUDA funkcije `sumsplat_update_output_cuda` unutar `.cu` datoteke.

```
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>

template <typename scalar_t>
```

```
__global__ void sumsplat_update_output_cuda_kernel(
    const int n,
    const torch::PackedTensorAccessor32<scalar_t,4,torch::RestrictPtrTraits> input,
    const torch::PackedTensorAccessor32<scalar_t,4,torch::RestrictPtrTraits> flow,
    torch::PackedTensorAccessor32<scalar_t, 4, torch::RestrictPtrTraits> output
) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for(int i = index; i < n; i += stride) {
    const int N = (i / output.size(3) / output.size(2) / output.size(1)) % output.size(0);
    const int C = (i / output.size(3) / output.size(2)) % output.size(1);
    const int qy = (i / output.size(3)) % output.size(2);
    const int qx =  i % output.size(3);

    float flt_output_x = (float) qx + flow[N][0][qy][qx];
    float flt_output_y = (float) qy + flow[N][1][qy][qx];

    int p_up_left_x = (int) floor(flt_output_x);
    int p_up_left_y = (int) floor(flt_output_y);
    // Calculating rest of the positions of adjacent pixels...

    float b_up_left = ((float) (p_down_right_x) - flt_output_x) * ((float) (p_down_right_y) - flt_output_y);
    // Calculating rest of the bilinear kernel b(u)...

    if((p_up_left_x >= 0) & (p_up_left_x < output.size(3)) & (p_up_left_y >= 0) & (p_up_left_y < output.size(2))) {
      atomicAdd(&output[N][C][p_up_left_y][p_up_left_x],( input[N][C][qy][qx] * b_up_left));
    }
    // Calculating output at the rest positions of adjacent pixels...
  }
}
torch::Tensor sumsplat_update_output_cuda(
    torch::Tensor input,
    torch::Tensor flow
) {
  torch::Tensor output = torch::zeros_like(input);
  const int N = output.numel();
  const int blockSize = 256;
  const int numBlocks = 1024;
```

```
44   sumsplat_update_output_cuda_kernel<float><<<blockSize, numBlocks>>>(
45       N,
46       input.packed_accessor32<float,4,torch::RestrictPtrTraits>(),
47       flow.packed_accessor32<float,4,torch::RestrictPtrTraits>(),
48       output.packed_accessor32<float,4,torch::RestrictPtrTraits>()
49   );
50   cudaDeviceSynchronize();
51   return output;
52 }
```

Listing 4.2: Primjer `.cu` datoteke

Funkcija `sumsplat_update_output_cuda` je funkcija koja poziva CUDA kernel `sumsplat_update_output_cuda_kernel`. CUDA kernel se označava ključnom riječi `__global__` koja govori CUDA C++ kompajleru da je to funkcija koja se izvršava na GPU a poziva se iz CPU koda. Pozivi CUDA kernel funkcije označeni su s `<<< >>>` što se može vidjeti na liniji 44 primjera 4.2. Ta sintaksa govori CUDA runtime-u koliko paralelnih dretvi koristiti prilikom izvršavanja kernela. CUDA organizira dretve u blokove dretvi, te kako bi se iskoristila puna snaga GPU-a kerneli se pozivaju s više blokova dretvi. U primjeru 4.2 varijabla `blockSize` označava broj dretvi po bloku, a varijabla `numBlocks` označava ukupan broj blokova dretvi, dakle ukupan broj dretvi je umnožak te dvije vrijednosti. Zajedno blokovi dretvi čine rešetku (engl. *grid*).

CUDA kernel mora uzeti u obzir cijelu rešetku blokova dretvi, zato CUDA pruža strukturu podataka pomoću koje se mogu indeksirati pojedine dretve u rešetci; varijabla `gridDim.x` sadrži broj blokova dertvi u rešetci, varijabla `blockIdx.x` sadrži indeks trenutnog bloka dretvi u mreži, a varijabla `threadIdx.x` sadrži indeks trenutne dretve u bloku. Ideja je da svaka dretva dobije indeks računajući pomak do početka svog bloka (umnožak indeksa trenutnog bloka `blockIdx.x` i veličine bloka `blockDim.x`) i dodavanja indeksa dretve `threadIdx.x`. Svaka dretva će indeksirati jedan element `torch::Tensor`-a. Ako je broj dretvi manji od broja elemenata onda se treba pomicati za ukupan broj dretvi (umnožak `blockDim.x` i `gridDim.x`) sve dok svi elementi ne budu indeksirani. Ovaj postupak prikazan je u primjeru 4.2 na linijama 12 do 14.

Za pristupanje elementima `torch::Tensor`-a bez računanja pomaka koristi se sučelje `torch::packed_accesor32<>`. Osim pristupanja elementima ovo sučelje nudi i metodu `size` koja kao argument prima broj dimenzije i vraća njenu veličinu. Iako je u primjeru 4.2 indeksiranje dretvi jednodimenzionalno, moraju se moći indeksirati ulazni podatci koji su četverodimenzionalni. Stoga odgovarajući indeksi računaju se pomoću metode `size` kako je prikazano linijama 15 do 18.

Metoda `atomicAdd` je CUDA-ina funkcija koja pročita jednu 32-bitnu ili 64-bitnu riječ na zadanoj adresi, doda joj broj i zapiše rezultat na istu adresu. Važno svojstvo ove funkcije je

da osigurava izvođenje bez miješanja drugih dretvi.
Nakon poziva CUDA kernela poziva se funkcija `cudDeviceSynchronize` koja osigurava da CPU čeka kraj izvođenja CUDA kernela jer pozivanje CUDA kernela ne blokira pozivajuću CPU dretvu.

4.2. Povezivanje s Pythonom

Ovo poglavlje razmatra *ahead of time* izgradnju C++ proširenja opisanog u poglavlju 4.1 koristeći `setuptools` biblioteku. Za ovu vrstu izgradnje potrebno je napisati Python skriptu koja koristi `setuptools` biblioteku za kompajliranje C++ koda. Primjer Python skripte dan je u nastavku.

```
from setuptools import setup
from torch.utils.cpp_extension import BuildExtension, CUDAExtension

setup(
    name='sumsplat_cuda',
    ext_modules=[
        CUDAExtension('sumsplat_cuda', [
            'sumsplat_cuda.cpp',
            'sumsplat_cuda_kernel.cu',
        ])
    ],
    cmdclass={
        'build_ext': BuildExtension
    })
```

Listing 4.3: Primjer Python skripte za izgradnju C++ proširenja koristeći biblioteku `setuptools`

`cpp_extension` paket brine se za kompajliranje C++ datoteka s C++ kompajlerom (`gcc`) i CUDA datoteka s NVIDIA kompajlerom (`nvcc`). Na kraju se povezuju u jednu dijeljenu biblioteku (engl. *shared library*) koja će biti dostupna u Python kodu. `BuildExtension` izvršava brojne potrebne konfiguracijske korake i provjere, te također upravlja kompajliranjem u slučaju miješanih C++/CUDA proširenja. Modulu `CUDAExtension` se predaju `.cu` i `.cpp` datoteke koji prosljeđuje odgovarajuće *include* putanje i postavlja C++ i CUDA kao jezike proširenja.
Nakon gotove implementacije operacija, za povezivanje C++ funkcija ili razreda s Pythonom može se koristiti biblioteka *pybind11*. Za trenutni primjer implementacije summation splatting $\overrightarrow{\Sigma}$ na kraju C++ datoteke potrebno je dodati sljedeći kod:

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &sumsplat_update_output, "Summation splatting forward
    (CUDA)");
  m.def("backward_input", &sumspalt_update_gradinput, "Input gradients of
     summation splatting (CUDA)");
  m.def("backward_flow", &sumspalt_update_gradflow, "Flow gradients of
    summation splatting (CUDA)");
}
```

Listing 4.4: Povezivanje C++ funkcija s Python-om koristeći biblioteku *pybind11*

Nakon pokretanja Python skripte i izgradnje proširenja C++ funkcije se mogu koristiti u Pythonu:

```
>>> import torch
>>> import sumsplat_cuda
>>> sumsplat_cuda.forward
<built-in method forward of PyCapsule object at 0x7ff0c81fd510>
>>> help(sumsplat_cuda.forward)
Help on built-in function forward in module sumsplat_cuda:

forward(...) method of builtins.PyCapsule instance
    forward(arg0: at::Tensor, arg1: at::Tensor) -> at::Tensor

    Summation splatting forward (CUDA)
```

Listing 4.5: Primjer korištenja C++/CUDA proširenja povezanog s Python-om.

Nakon povezivanja s Pythonom potrebno je integrirati implementirane operacije u programski okvir PyTorch kako bi se iskoristilo svojstvo automatske diferencijacije. Potrebno je omotati C++ funkcije s `torch.autograd.Function` kako bi ih učinili pravim članovima PyTorcha.

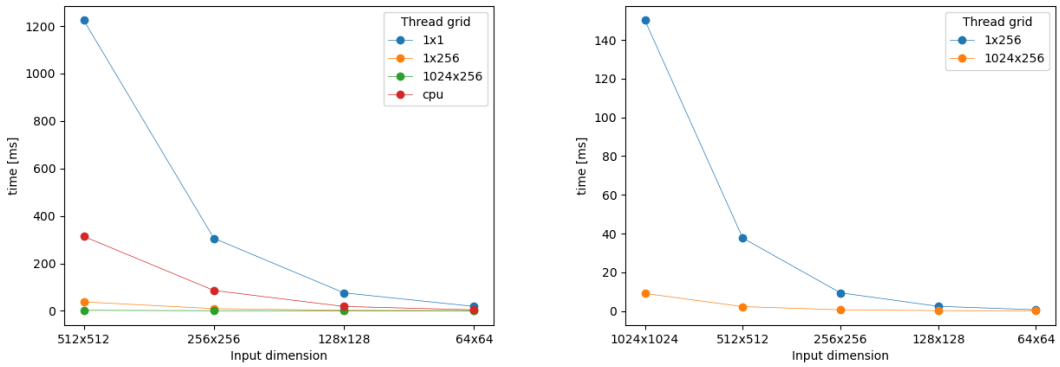
```
import torch
import sumsplat_cuda as ss

class SumSplatFunction(torch.autograd.Function):
  @staticmethod
    def forward(ctx, tenInput, tenFlow):
        ctx.save_for_backward(tenInput, tenFlow)
        return ss.forward(tenInput, tenFlow)

    @staticmethod
    def backward(ctx, gradOutput):
        tenInput, tenFlow = ctx.saved_tensors
        gradInput = ss.backward_input(tenInput, tenFlow, gradOutput)
        gradFlow = ss.backward_flow(tenInput, tenFlow, gradOutput)
```

```
15          return gradInput, gradFlow
```

Listing 4.6: Primjer integriranja C++/CUDA proširenja s programskim okvirom PyTorch.

4.3. Rezultati

Na slici 4.1 prikazana je brzina izvođenja unaprijednog prolaza implementirane transformacije za različit broj dretvi i različite veličine ulazne slike. Korištena grafička kartica je Nvidia GeForce MX110/PCIe/SSE2.



Slika 4.1: Graf brzine izvođenja unaprijednog prolaza implementirane transformacije u ovisnosti o veličini ulazne slike i broju dretvi.

Izvršavanje na GPU s jednom dretvom postiže najlošije rezultate, čak i lošije nego izvršavanje na CPU s jednom dretvom. Ali snaga GPU-a leži u višedretvenosti, što se može i vidjeti. Izvršavanje na GPU s $1024 \cdot 256$ dretvi drastično smanjuje vrijeme izvršavanja u odnosu na CPU, što se najbolje vidi za veće ulazne dimenzije. Kako je broj dretvi bitan pokazuje i desni graf na slici 4.1. Vidi se da je za ulazne dimenzije 1024x1024 brzina izvođenja drastično manja kad se koristi 1024 puta više dretvi.

5. Zaključak

Unaprijedna transformacija slika zasnovana na funkciji softmax koju su predložili Niklaus i Liu (2020) rješava problem mapiranja više različitih piksela na istu poziciju na učinkovit način. Bitno svojstvo ove transformacije je diferencijabilnost jer omogućava korištenje algoritama za učenje baziranih na računanju gradijenata. Prikazan je primjer implementacije navedene transformacije u programskom okviru PyTorch koji ima svojstvo automatske diferencijacije. Kako bi se iskoristila snaga grafičkog procesora implementacija je napisana u programskom okviru CUDA. Dobiveni rezultati pokazuju da izvršavanje na GPU drastično smanjuje vrijeme izvršavanja u odnosu na CPU.

6. Literatura

Peter Goldsborough. Custom c++ and cuda extensions. URL `https://pytorch.org/tutorials/advanced/cpp_extension.html`.

Simon Niklaus i Feng Liu. Softmax splatting for video frame interpolation. U *IEEE International Conference on Computer Vision*, 2020.

Richard Szelinski. *Computer Vision: Algorithms and Applications*. 1. izdanju, 2010. URL `http://szeliski.org/Book/1stEdition.htm`.

7. Sažetak

Kroz ovaj seminarski rad opisana je važnost diferencijabilnih transformacija te njihova implementacija. Kao konkretne transformacije opisane su unaprijedne i unatražne transformacije slike, te nova transformacija koju su predložili Niklaus i Liu (2020). Provedena je studija slučaja navedene nove transformacije, te su opisane njene prednosti u odnosu na druge transformacije. Nadalje, prikazan je primjer implementacije navedene transformacije u programskim okvirima PyTorch i CUDA. Implementacija transformacije u programskom okviru CUDA iskorištava svu snagu grafičkog procesora, te povezivanje te implementacije s programskim okvirom PyTorch iskorištava njegovu mogućnost automatske diferencijacije. Prikazanim primjerom dan je postupak implementiranja vlastitih diferencijabilni transformacija u programskim okvirima CUDA i PyTorch. Na kraju su prikazane brzine izvršavanja na CPU i GPU s različitim brojem dretvi.