UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Vene

Izravnava histograma

PROJEKTNA NALOGA

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM

RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: izr. prof. dr. Patricio Bulić

Ljubljana, 2024

Kazalo vsebine

Kazalo slik

Kazalo tabel

Uvod

Izravnava histograma je proces obdelave slike, ki nam omogoča kreiranje enakomerno porazdeljenega histograma. Posledično, je intenziteta barv bolje porazdeljena na histogramu. Rezultat le-tega, je slika s povečanim kontrastom. To metodo lahko predvsem uporabimo na slikah, kjer je ozadje in ospredje svetlo ali temno. Prednost uporabe te metode, je predvsem v njeni dokaj enostavni implementaciji, zato lahko algoritem razdelimo na tri korake:

1. Kreiranje histograma.
   Izračun histograma prvotne slike z namenom, da pridobimo njeno tonsko porazdelitev sivih vrednosti.
2. Izračun kumulativne distribucije za ustvarjeni histogram.
   Pridobimo v vednost, koliko slikovnih točk ima določeno barvo, ali manjšo.
3. Uporaba transformacijske funkcije.
   Uporabimo formulo za izravnavo histograma na podatkih, ki smo jih prejeli od prejšnjega koraka (CDF).

$$l_{new} = \left\lfloor \left(\frac{cdf(l) - cdf_{min}}{(N \times M) - cdf_{min}} (L - 1) \right) \right\rfloor$$

Slika 1 Transformacijska funkcija

Paralelna implementacija izravnave histograma v programskem jeziku CUDA

Paralelna implementacija izračuna histograma

Osnovna izvedba je, da kreiramo toliko niti kolikor je slikovnih točk, torej da bi vsaka nit računala histogram. Tu nastane problem, da več niti poskuša pisati v enaki element histograma, kar povzroči napačne vrednosti. Le-to lahko rešimo z uporabo atomičnih operacij, in sicer:

```
atomicAdd(&localHisto[image[y * width + x]], 1);
```

Slika 2 Atomična operacija

Vendar ni zagotovila, da bosta dve sosednji niti v snopu dostopala do dveh sosednjih elementov histograma, torej ne implementira združevanja pomnilniških dostopov. V ta namen, uporabimo lokalni pomnilnik, kar pomeni, da niti v svojem bloku izračunajo svoj lasten histogram.

```
__shared__ unsigned int localHisto[GRAY_LEVELS];
```

Slika 3 Lokalni pomnilnik

Sedaj nastane toliko delnih oz. lastnih histogramov, kolikor je blokov. Posledično je potrebno sešteti histograme in pri tem ponovno uporabiti atomične operacije:

```
atomicAdd(&(histogram[blockDim.x * localY +
localX]), localHisto[blockDim.x * localY +
localX]);
```

Slika 4 Atomična operacija 2

Celotna implementacija (na naslednji strani):

```
__global__ void calculateHistogram(unsigned char
*image, int width, int height, unsigned int
*histogram) {

    int x = blockIdx.x * blockDim.x + threadIdx.x,
        y = blockIdx.y * blockDim.y + threadIdx.y;

    int localX = threadIdx.x,
        localY = threadIdx.y;

    __shared__ unsigned int
localHisto[GRAY_LEVELS];

    localHisto[blockDim.x * localY + localX] = 0;
    __syncthreads();

    if (x < width && y < height) {
        atomicAdd(&localHisto[image[y * width +
x]], 1);
    }
    __syncthreads();

    atomicAdd(&(histogram[blockDim.x * localY +
localX]), localHisto[blockDim.x * localY +
localX]);

}
```

Slika 5 Implementacija histograma

Paralelna implementacija »Parallel prefix sum«

Implementacija tega algoritma vsebuje težavo, da je potrebnih preveliko število seštevanj. To se predvsem materializira pri večjih slikah. Asimptotična zahtevnost tega algoritma je posledično $O(n \times log_2(n))$. Rešitev je uporaba »Up-sweep« ter »Down-sweep« redukcije, ki postavi zahtevnost na $O(n)$, vendar v tej izvedbi ni uporabljena.

Algoritem za namene lokalnega dostopa uporablja lokalni pomnilnik, ki je dvojne velikosti, ker si niti v vsakem koraku prepisujejo elemente, ki jih berejo.

```
__shared__ unsigned int tmp[GRAY_LEVELS * 2];
```

Slika 6 Lokalni pomnilnik 2

Posledično potrebujemo tudi način preklapljanja med njima:

```
int threadID = threadIdx.x,
        pout = 0,
        pin = 1;
```

Slika 7 Spremenljivke

Po končani operaciji je potrebno izhodni del dvojnega pomnilnika prepisati v globalni pomnilnik:

```
cdf[threadID] = tmp[pout * GRAY_LEVELS + threadID];
```

Slika 8 Prenos vrednosti iz lokalnega v globalni pomnilnik

Celotna implementacija (na naslednji strani):

```
__global__ void parallelPrefixSum(unsigned int
*histogram, unsigned int *cdf) {

    __shared__ unsigned int tmp[GRAY_LEVELS * 2];
    int threadID = threadIdx.x,
        pout = 0,
        pin = 1;

    tmp[threadID] = histogram[threadID];
    __syncthreads();

    for (int offset = 1; offset<GRAY_LEVELS;
offset<<=1) {
        pout = 1 - pout;
        pin = 1 - pout;
        if (threadID >= offset) {
            tmp[pout * GRAY_LEVELS + threadID] =
tmp[pin * GRAY_LEVELS + threadID] + tmp[pin *
GRAY_LEVELS + threadID - offset];
        } else {
            tmp[pout * GRAY_LEVELS + threadID] =
tmp[pin * GRAY_LEVELS + threadID];
        }
        __syncthreads();
    }

    cdf[threadID] = tmp[pout * GRAY_LEVELS +
threadID];
}
```

Slika 9 Implementacija Parallel prefix sum algoritma

Paralelna implementacija iskanje najmanjšega ne ničelnega števila

Implementacija tega algoritma uporablja redukcijo z namenom združevanja niti, dokler ne ostane le ena, ki vsebuje končni rezultat. V vsaki iteraciji zanke je potrebno število niti zmanjšati, in sicer za faktor 2. V ta namen lahko uporabimo operacijo deljenja, vendar zaradi hitrosti delovanja algoritma to rajši izvedemo preko bitne operacije premika za eno mesto v desno – to ima enak učinek kot deljenje z dve. Potrebno je hkrati uporabiti prepreko za namene sinhronizacije niti. Algoritem bi najverjetneje lahko izboljšali z uporabo lokalnega pomnilnika.

Zmanjšamo število niti za polovico vsako iteracijo:

```
for (unsigned int stride = blockDim.x >> 1;
stride>0; stride >>= 1)
```

Slika 10 Zmanjševanje števila niti

Zapomnimo najmanjši ne ničelni element:

```
cdf[threadID] = (cdf[threadID] != 0) ?
min(cdf[threadID], cdf[threadID + stride]) :
cdf[threadID + stride];
```

Slika 11 Iskanje najmanjšega ne ničelnega števila

Zadnja nit zapiše rezultat:

```
if (threadID == 0) {
        minimum[blockIdx.x] = cdf[0];
}
```

Slika 12 Zapis rezultata najmanjšega števila

Celotna implementacija (na naslednji strani):

```
__global__ void calcuateMinReduction(unsigned int
*cdf, unsigned int *minimum) {
    int threadID = threadIdx.x;

    // x >> 1 is the same as x / 2
    // and because bitwise op. are faster, so
    // we are gonna use it.
    // So it is stride / 2; stride>0; stride /= 2
    // this reduces the number of threads involved
in the reduction
    // by half each iteration until there is only
one left that performs
    // the final combination
    for (unsigned int stride = blockDim.x >> 1;
stride>0; stride >>= 1) {
        // this is reduction
        __syncthreads();
        if (threadID < stride) {
            cdf[threadID] = (cdf[threadID] != 0) ?
min(cdf[threadID], cdf[threadID + stride]) :
cdf[threadID + stride];
        }
    }
    // one thread remaining
    if (threadID == 0) {
        minimum[blockIdx.x] = cdf[0];
    }

}
```

Slika 13 Implementacija iskanje najmanjšega ne ničelnega števila

Paralelna implementacija transformacijske funkcije

Implementacija tega algoritma je bila dokaj enostavna, saj je bilo možno transformacijsko funkcijo izračunati za vsako slikovno točko neodvisno od drugih. Za lokalni dostop uporablja lokalni pomnilnik:

```
__shared__ unsigned int localCdf[GRAY_LEVELS];
```

Slika 14 Lokalni pomnilnik 3

Funkcija, ki izračuna formulo je predpisana kot __device__, torej se izvaja na GPU ter kliče s strani GPU.

Celotna implementacija (na naslednji strani):

```
__device__ inline unsigned char scale(unsigned int
cdf, unsigned int min, unsigned int imageSize) {

    float scale;
    scale = (float)(cdf - min) / (float) (imageSize
- min);
    scale = round(scale * (float) (GRAY_LEVELS -
1));

    return (int) scale;

}

__global__ void equalize(unsigned char *imageIn,
unsigned char *imageOut, int width, int height,
unsigned int *cdf, unsigned int *cdfMin) {

    int x = blockIdx.x * blockDim.x + threadIdx.x,
        y = blockIdx.y * blockDim.y + threadIdx.y;

    int localX = threadIdx.x,
        localY = threadIdx.y;

    unsigned int imageSize = width * height;

    __shared__ unsigned int localCdf[GRAY_LEVELS];

    localCdf[blockDim.x * localY + localX] =
cdf[blockDim.x * localY + localX];
```

```
    __syncthreads();

    if (x < width && y < height) {
        imageOut[y * width + x] =
scale(localCdf[imageIn[y * width + x]], *cdfMin,
imageSize); // cdf, cdfMin, imageSize
    }

}
```

Slika 15 Implementacija izračuna transformacijske funkcije

Eksperiment

Specifikacije računalnika, na katerem je potekal eksperiment:

- Sistem vsebuje 4256 jeder:
  - Vozlišča:
    - 62 x 64 jeder, AMD Epyc 7702P
    - 24 x 12 jeder, AMD Epyc 7272, 2x Nvidia V100
- 17,9 TB RAM
- 3 PB HDD, 320 TB SSD
- WAN:
  - 100 Gb/s
- Programska oprema:
  - OS AlmaLinux 8
  - Sistem za upravljanje s posli – SLURM

Podatki o uporabljenih slikah:

Slika	Širina [px]	Višina [px]
S_kolesar-neg	400	250
Slika2	625	348
Unequalized_Hawkes_Bay_NZ	1024	683
gray	1880	1204
grayscale-photo-of-elephant-2	4333	2424
r597Y	991	637
Town	360	540
Plane	1024	589
sceene	1928	941
Slika10	2736	1824

Tabela 1 Velikosti slik

Merjenje časa izvedbe programa je bilo izvedeno z uporabo:

- GPU: cudaEventCreate, cudaEventElapsedTime. Za vsak ščepec oz. kernel je bil ustvarjen svoj dogodek in pretekel čas dodan h globalni vsoti.
- CPU:

```
#include <time.h>
struct timespec timeStart, timeEnd;
clock_gettime(CLOCK_REALTIME, &timeStart);
clock_gettime(CLOCK_REALTIME, &timeEnd);
double timeTaken = (timeEnd.tv_sec -
timeStart.tv_sec) + (timeEnd.tv_nsec -
timeStart.tv_nsec) / 1e9;
printf("Time taken: %f seconds\n", timeTaken);
```

Slika 16 Merjenje časa

Eksperiment je uporabil 10 slik različnih velikosti. Vsako sliko se je obdelalo 10-krat, sekvenčno in paralelno. Spodnja tabela prikazuje povprečen čas izvajanja. Povprečni časi izvedbe bo se izračunali s pomočjo Python programa.

Slika	Povprečni čas izvajanja - sekvenčno [sekund]	Povprečni čas izvajanja – CUDA [sekund]
S_kolesar-neg	0,0582589	0,0133949
Slika2	0,1002774	0,0022431
Unequalized_Hawkes_Bay_NZ	0,2040542	0,0072940
gray	0,4445167	0,0083450
grayscale-photo-of-elephant-2	2,5918127	0,0013065
r597Y	2,6121799	0,0010043
Town	0,0676366	0,0006364
Plane	0,1898799	0,0010379
sceene	0,5007837	0,0010793
Slika10	1,3028210	0,0010161

Tabela 2 Povprečni časi izvajanja v sekundah

Rezultati in diskusija

Za vsako od desetih uporabljenih slik v prejšnjem eksperimentu je tu prikazan njihov izgled prej in po izravnavi histograma, izračunane pohitritve ter točka, kjer CUDA postane hitrejša od sekvenčne izvedbe. Pohitritev je bila izračunana kot:

Pohitritev = čas izvajanja CPU / čas izvajanja GPU

S_kolesar-neg

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,044780	0,000940	47-krat

Tabela 3 S_kolesar-neg pohitritev

Prej:

Slika 17 S_kolesar-neg prej

Potem:

Slika 18 S_kolesar-neg potem

Slika2

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,026818	0,001119	23-krat

Tabela 4 Slika2 pohitritev

Prej:

Slika 19 Slika2 prej

Potem:

Slika 20 Slika2 potem

Unequalized_Hawkes_Bay_NZ

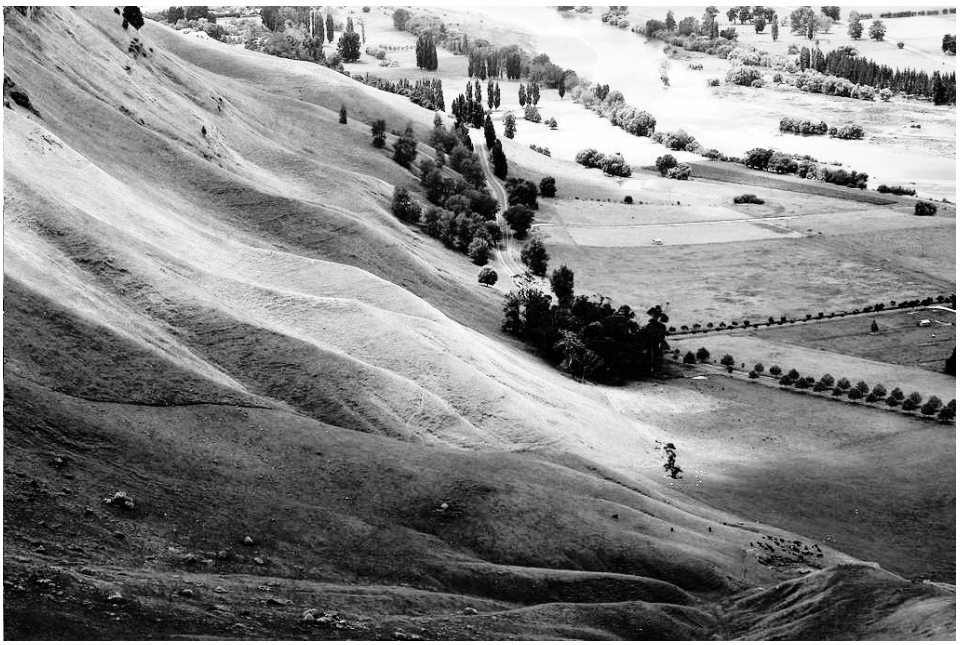
Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,065195	0,001270	50-krat

Tabela 5 Unequalized_Hawkes_Bay_NZ pohitritev

Prej:

Slika 21 Unequalized_Hawkes_Bay_NZ prej

Potem:



Slika 22 Unequalized_Hawkes_Bay_NZ potem

Gray

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,144692	0,001990	72-krat

Tabela 6 Gray pohitritev

Prej:

Slika 23 Gray prej

Potem:

Slika 24 Gray potem

grayscale-photo-of-elephant-2

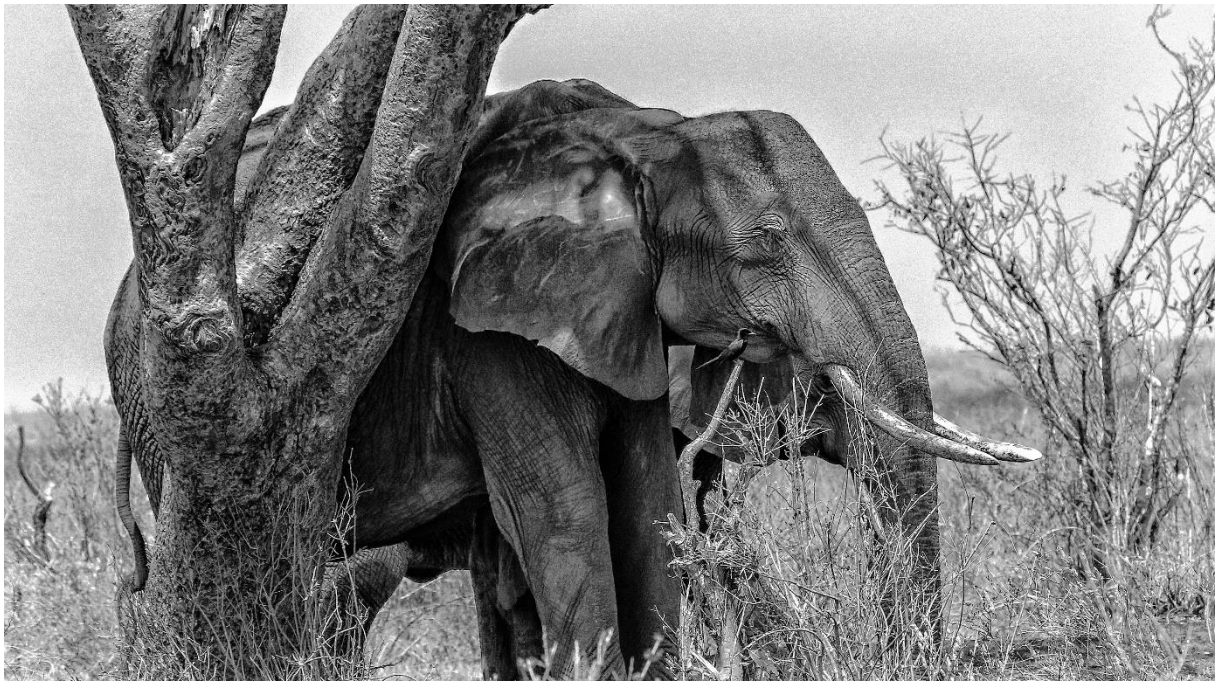
Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,985154	0,159150	6-krat

Tabela 7 grayscale-photo-of-elephant2 pohitritev

Prej:

Slika 25 grayscale-photo-of-elephant2 prej

Potem:



Slika 26 grayscale-photo-of-elephant2 potem

r597Y

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,044037	0,000735	59-krat

Tabela 8 r597Y pohitritev

Prej:

Slika 27 r597Y prej

Potem:

Slika 28 r597Y potem

Town

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,022008	0,000971	22-krat

Tabela 9 Town pohitritev

Prej:

Slika 29 Town prej

Potem:

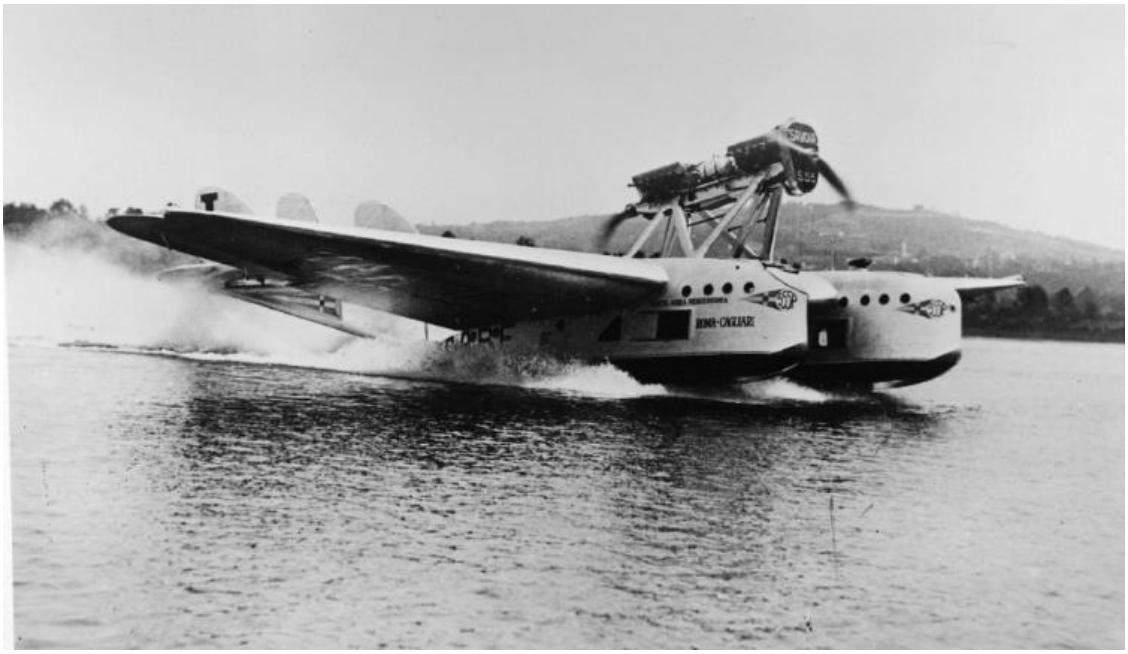
Slika 30 Town potem

Plane

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,054333	0,000871	62-krat

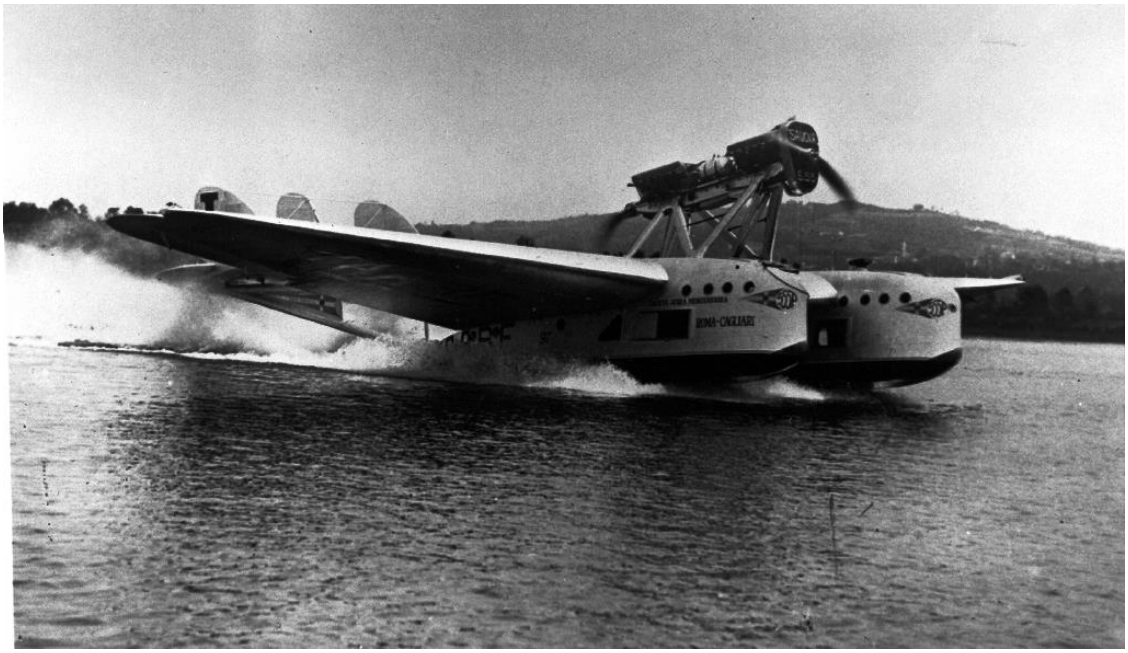
Tabela 10 Plane pohitritev

Prej:



Slika 31 Plane prej

Potem:



Slika 32 Plane potem

Sceene

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,154323	0,001524	100-krat

Tabela 11 Sceene pohitritev

Prej:

Slika 33 Sceene prej

Potem:

Slika 34 Sceene potem

Slika10

Čas izvajanja CPU [sekund]	Čas izvajanja GPU [sekund]	Pohitritev
0,441347	0,190704	2-krat

Tabela 12 Slika10 pohitritev

Prej:

Slika 35 Slika10 prej

Potem:

Slika 36 Slika10 potem

Ugotovitve

Projektna naloga nam je pomagala ugotovit točko, kjer CUDA postane hitrejša od sekvenčne izvedbe. Izkazalo se je, da na to najbolj vpliva velikost oz. resolucija slike. CUDA implementacija izravnave histograma bi se najverjetneje še lahko izboljšala, in sicer na področju iskanja najmanjše ne ničelne vrednosti, zmanjšanje količine seštevanja pri »Parallel prefix sum« algoritmu ter izračunu transformacijske funkcije. Izračuni pohitritve desetih različnih slik se zelo razlikujejo, in sicer od izmerjene 2-kratne pohitritve do 100-kratne pohitritve, kar je presenetljivo, vendar mogoče nakazuje težave pri merjenju časa enega izmed implementacij, ali samemu načinu izračuna.