

Implementacija histograma in vzporednega scan algoritma

Tim Rekelj, 63210277

February 2024

1 Kazalo

1 Kazalo	2
2 Uvod	3
3 Paralelna implementacija izenačitve histograma v CUDA	3
3.1 Računanje histograma slike	3
3.2 Računanje kumulativne distribucijske funkcije	3
3.3 Transformacija prvotne slike	3
4 Eksperiment	4
5 Rezultati in diskusija	7
5.1 Primeri enačevanja	7
5.2 Primerjava hitrosti	8
5.3 Srečanje hitrosti med paralelnim in sekvenčnim programom . . .	9
6 Zaključek	10
6.1 GCC	10

2 Uvod

Izenačevanje slike je postopek, ki se uporablja za izboljšanje kontrasta slike in poudarjanje podrobnosti. Algoritem za izenačevanje slike deluje tako, da priлагodi svetlost in kontrast posameznih pikslov na sliki. Najprej se izračuna histogram slike, ki prikazuje porazdelitev svetlosti med pikslji. Nato se izvede transformacija histograma, kjer se porazdelitev enakomerno razporedi po celotnem razponu svetlosti. To ima za posledico izboljšano razločevanje med različnimi področji slike in poudarjanje detajlov.

3 Paralelni implementaciji izenačitve histograma v CUDA

3.1 Računanje histograma slike

Vsak blok niti vzpostavi svoj lokalni histogram s pomočjo skupnega pomnilnika *localHistogram*.

Niti v bloku preštejejo pojavitve posameznih intenzitet v svojem delu slike in uporabijo atomicAdd za varno povečanje ustreznega bin-a v lokalnem histogramu. Synchronizacijska ovira (*__syncthreads()*) se uporablja za zagotovitev, da so vsi blokovi končali z izračunom svojih lokalnih histogramov.

Uporaba lokalnega histograma na ravni blokov pomaga zmanjšati konflikte pri sočasnem pisanju v pomnilnik in izboljšati učinkovitost.

Velikost lokalnega histograma je omejena z zmogljivostjo skupnega pomnilnika bloka, kar lahko vpliva na natančnost izenačevanja za slike z velikim razponom intenzitet.

Uporaba atomicAdd lahko povzroči čakalne čase v primeru visoke konkurence za posamezne bin-e v lokalnem histogramu.

3.2 Računanje kumulativne distribucijske funkcije

V vsakem bloku niti izračunajo kumulativno distribucijsko funkcijo (CDF) iz lokalnega histograma. Uporabljajo zaporedno akumulacijo v *localHistogram*, kjer trenutni element enačimo s števkom prejšnjega elementa in trenutnega v CDF.

Ponovno se uporablja sinhronizacijska ovira za zagotovitev končanja izračunov vseh blokov.

Kumulativna distribucijska funkcija (CDF) se izračuna vzporedno z računanjem lokalnih histogramov za hitrejšo izvedbo.

3.3 Transformacija prvotne slike

Vsaka nit prejme intenziteto svojega piksla v sliki in uporabi izračunano CDF za transformacijo te intenzitete.

Ta transformacija se izvede neposredno na sliki v pomnilniku GPU.

Po zaključku transformacije se vsi rezultati združijo v transformirani sliki.
 Za zelo velike slike ali slike z veliko intenzitetami bi bilo treba prilagoditi strategijo za obvladovanje pomnilnika in optimizirati deljenje podatkov med niti.

4 Eksperiment

```
===== cudaDeviceGetProperties =====
Device 0: "Tesla V100S-PCIE-32GB"
    GPU Clock Rate (MHz):          1597
    Memory Clock Rate (MHz):        1107
    Memory Bus Width (bits):       4096
    Peak Memory Bandwidth (GB/s):  1133.57
    CUDA Cores/MP:                 64
    CUDA Cores:                    5120
    Total amount of global memory: 32 GB
    Total amount of shared memory per block: 48 kB
    Total number of registers available per block: 65536
    Warp size:                     32
    Maximum number of threads per block: 1024
    Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
    Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

===== cudaDeviceGetAttribute =====
Device 0: "Tesla V100S-PCIE-32GB"
    Max number of threads per block:      1024
    Max block dimension X:                1024
    Max block dimension Y:                1024
    Max block dimension Z:                64
    Max grid dimension X:                2147483647
    Max grid dimension Y:                65535
    Max grid dimension Z:                65535
    Max shared memory per block:         49152
    Warp size:                          32
    Peak clock frequency in kilohertz:   1597000
    Peak memory clock frequency in kilohertz: 1107000
    Global memory bus width in bits:     4096
    Size of L2 cache in bytes:           6291456
    Maximum resident threads per SM:    2048
    Major compute capability version number: 7
    Minor compute capability version number: 0
    Max shared memory per SM in bytes:  98304
    Max number of 32-bit registers per SM: 65536
    Max per block shared mem size on the device: 98304
    Max thread blocks that can reside on a SM: 32
```

Figure 1: Strojna oprema naprave

V tem koraku sem naredil meritve paraleliziranega algoritma, sekvenčnega algoritma ter sekvenčnega algoritma v C-ju (brez CUDA, v gcc compilerju - za testiranje razlike med nvcc ter gcc).

Uporabil sem 11 verzij ene slike, katerih resolucija se enakomerno raztega med 10px širine ter 9148px širine (slika je približno kvadratna, tako da je razmerje približno 1:1), vse tri verzije programa pa sem ponovil 20-krat.

Za konec sem pa primerjal še sliko *kolesar-neq.jpg* iz predavanj.

Merjenje časa poteka od konca branja slike do začetka izdelave nove slike (ta dva ukaza sta input in output, zato dolžina njunega izvajanja ni odvisna od algoritma za histogram ter pretvorbo). Program kot argument sprejme ime

slike, ki jo želimo pretvoriti in pa število ponovitev, izračuna pa povprečje vseh ponovitev. Za testiranje vseh treh algoritmov sem naredil bash skripto, ki požene vse tri programe z enako sliko ter enakim številom ponovitev.

Slika	Paralelno [ms]	Sekvenčno (CUDA) [ms]	Sekvenčno (C) [ms]
<i>sun00.jpg, 10px x 10px</i>	0.3	0.01	0.01
<i>sun01.jpg, 922px x 918px</i>	0.71	19.67	17.78
<i>sun02.jpg, 1836px x 1829px</i>	1.78	63.73	56.78
<i>sun03.jpg, 2750px x 2739px</i>	3.46	137.64	123.43
<i>sun04.jpg, 3664px x 3650px</i>	7.12	242.35	215.81
<i>sun05.jpg, 4578px x 4560px</i>	10.26	378.53	336.36
<i>sun06.jpg, 5492px x 5471px</i>	14.41	537.85	483.15
<i>sun07.jpg, 6406px x 6381px</i>	19.25	718.64	643.75
<i>sun08.jpg, 7320px x 7292px</i>	24.5	932.65	833.11
<i>sun09.jpg, 8234px x 8202px</i>	30.49	1178.19	1047.36
<i>sun10.jpg, 9148px x 9112px</i>	37.44	1454.16	1294.96
<i>kolesar-neq.jpg, 1024px x 640px</i>	0.62	16.42	15.38

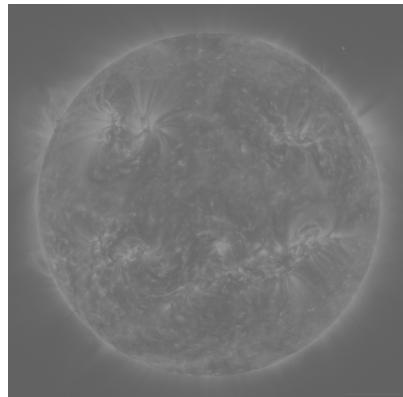


Figure 2: sun.jpg



Figure 3: kolesar-neq.jpg

Figure 4: Slike, uporabljeni v eksperimentu

5 Rezultati in diskusija

5.1 Primeri enačevanja

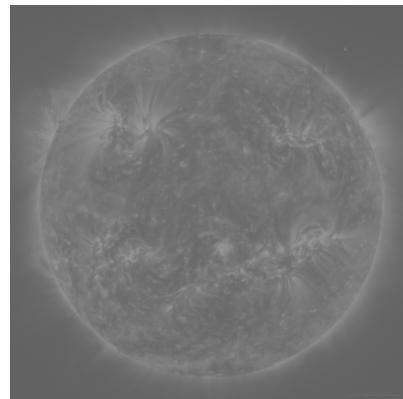


Figure 5: sun.jpg

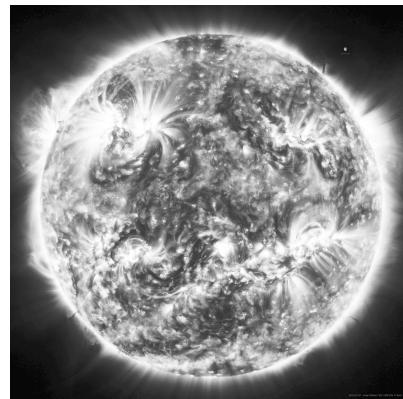


Figure 6: kolesar-neq.jpg

Figure 7: Primer enačevanja histograma slike *sun.jpg*



Figure 8: sun.jpg



Figure 9: kolesar-neq.jpg

Figure 10: Primer enačevanja histograma slike *kolesar-neq.jpg*

5.2 Primerjava hitrosti

Več kot očitno je, da se pri večjih slikah bolj splača uporabiti paralelno verzijo programa. Po velikosti $1836px \times 1829px$ je pospešitev enakomerna, približno 37x hitrejša. Točni časi so zapisani v tabeli v sekciji eksperiment.

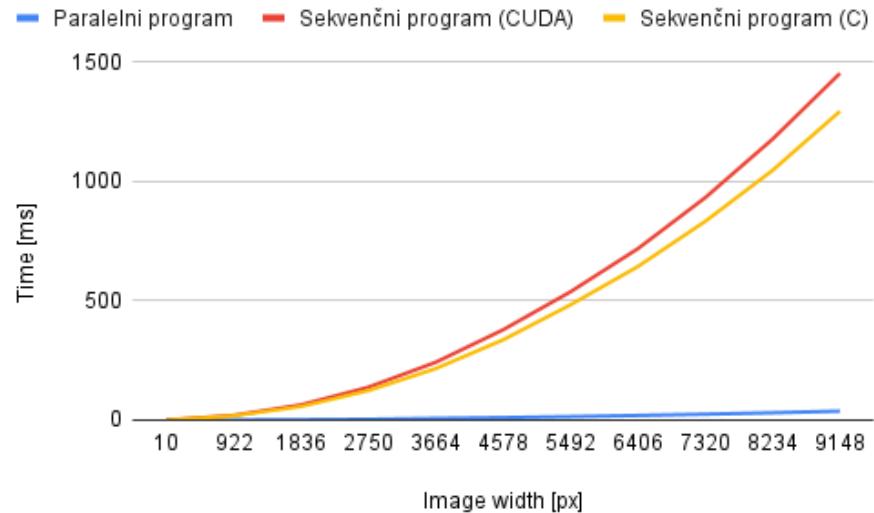


Figure 11: Graf povprečne hitrosti algoritma za 11 različnih velikosti slike *sun.jpg*

5.3 Srečanje hitrosti med paralelnim in sekvenčnim programom

Prva slika (10px širine) je hitrejša v sekvenčnem algoritmu, zato sem naredil poizkuse do katere velikosti se bolj splača sekvenčna verzija algoritma.

Spodnji graf prikazuje, da se hitrosti križata točno pri širini slike 100px. Višina te slike je 99px, torej se algoritma sekata pri 9900 pikslih.

Čas pri paralelnem algoritmu je pri majhnih slikah konstanten, ker predvidevam, da se porabi konstantna količina časa za razporejanje niti. To je tudi razlog, zakaj je paralelni algoritem počasnejši pri manjših številnih pikslov.

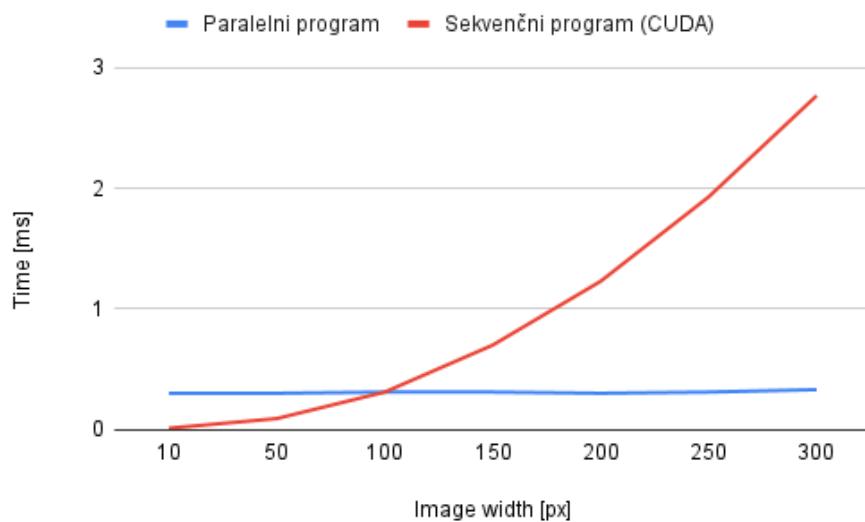


Figure 12: Graf povprečne hitrosti algoritma za 11 različnih velikosti slike *small-sun.jpg*

6 Zaključek

V raziskavi sem ugotovil, da faktor, kolikokrat hitrejši je paralelni algoritem od sekvenčnega, narašča do približno 3358044 pikslov, potem pa postane enakomerna (približno 37x hitrejši). Pri 9900 pikslih je meja, do katere se bolj splača uporabljati sekvenčni algoritem, saj razporejanje niti vzame konstantno količino časa.

6.1 GCC

Kot zanimivost bi poudaril, da sem v mojem testiranju ugotovil tudi, da je sekvenčni program, preveden z *gcc* prevajalnikom konstantno za približno 1.1x hitrejši od programa prevedenega z *nvcc*.