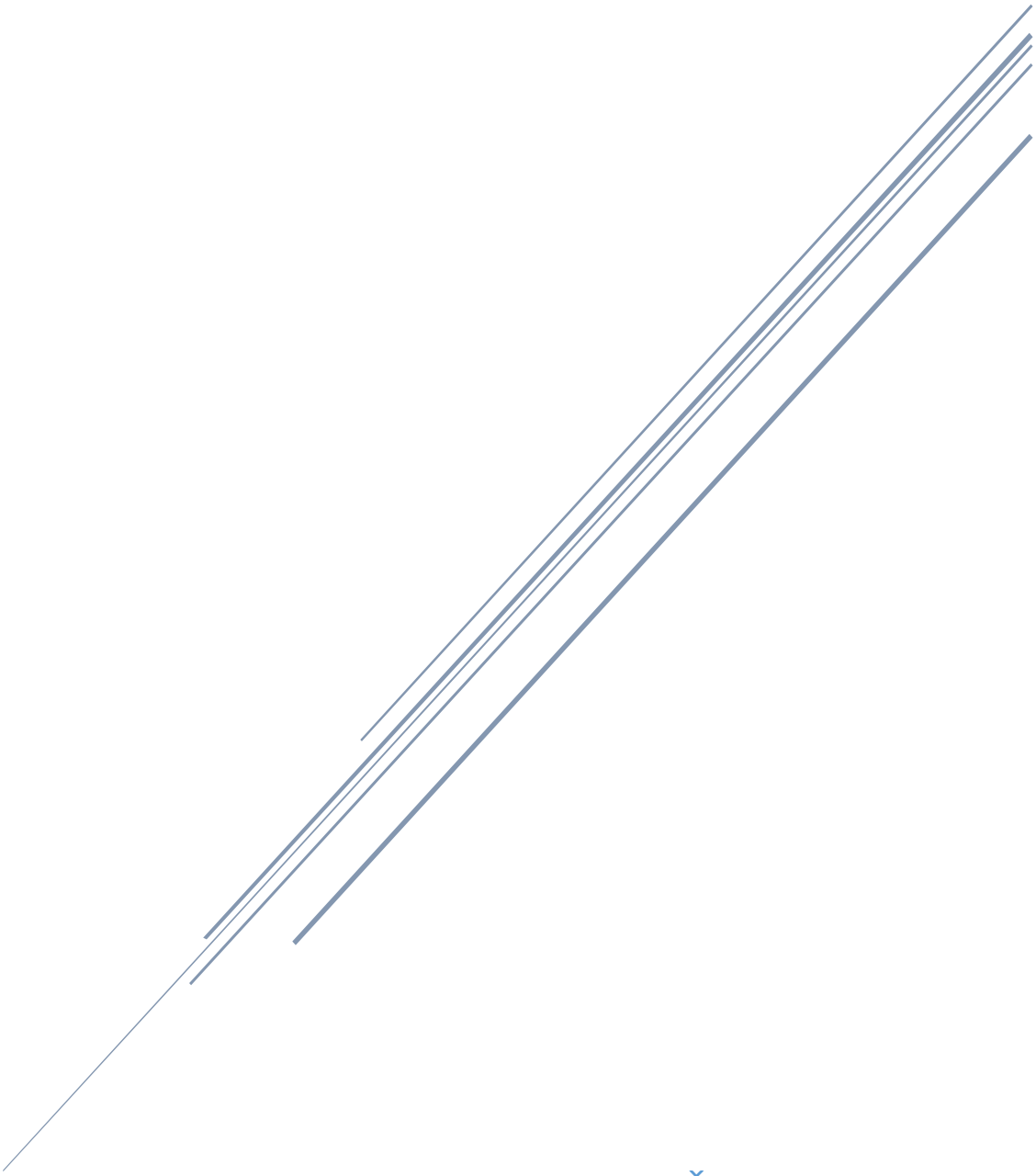
PARALLEL AND DISTRIBUTED SYSTEMS AND ALGORITHMS

Izenačenje histograma - seminarska naloga



Nejc Žun (63170329)
Ljubljana 2020

Kazalo

Uvod 2

Paralelna implementacija algoritma 2

Izračun histograma 2

Predpostavke 2

Kernel koda 2

Delovanje 3

Razlog za implementacijo 3

Izračun komulativnega histograma 4

Predpostavke 4

Kernel koda 4

Delovanje 4

Razlog za implementacijo 5

Izračun izenačenja 6

Predpostavke 6

Kernel koda 6

Delovanje 7

Razlog za implementacijo 7

Eksperiment 8

Naprava za testiranje: 8

Uporabljene slike: 8

Merjenje časa 8

Rezultati 9

Velike slike 9

Manjše slike 10

Ugotovitve 11

Možne nadgradnje 11

Uvod

Izenačenje histograma je tehnika procesiranja slik s katero lahko izboljšamo kontrast v slikah. To dosežemo tako, da raztegnemo najpogostejše intenzitetne vrednosti čez celotni histogram slike, ter s tem pridobimo nov histogram katerega vrednosti so enakomerno razporejene.

Metoda še posebej deluje, ko so kontrastne vrednosti stisnjene ena zraven druge. Slike z zelo nizkim kontrastom ali zelo temne slike posledično postanejo ostrejše / svetlejše (odvisno od izvorne slike).

Paralelna implementacija algoritma

Metoda izenačenja poteka v treh delih. Izračun histograma podane slike, izračun kumulativnega histograma ter izenačenje. Vsako izmed naslednjih sem implementiral v svoji funkciji. V glavnem programu inicializiram objekte v strukturi tako da se vsaka funkcija začne s inicializacijo kernela (s tem pridobimo na času ponovnega prebiranja / povezave na grafično kartico).

Izračun histograma

Predpostavke:

- Lokalna velikost: 32
- 2 dimenzionalen NDRange
- BINS = 256

Kernel koda

```
__kernel void HistogramGPU(__global unsigned char* imageIn, __global unsigned int* histogram, int width, int height) {

            //global indexes
            int gidx = get_global_id(1);
            int gidy = get_global_id(0);
            int global_index = gidy * width + gidx;

            //local indexes
            __local unsigned int local_hist[BINS];
            int lidx = get_local_id(1);
            int lidy = get_local_id(0);
            int local_index = lidy * get_local_size(0) + lidx;

            int rowGroups = get_group_id(1);
            int colGroups = get_group_id(0);

            unsigned int pixel;

            //nastav local histogram na 0
            if (local_index < BINS) local_hist[local_index] = 0;
            barrier(CLK_LOCAL_MEM_FENCE);
            if ((lidx + colGroups * get_local_size(0))< height && (lidy + rowGroups * get_local_size(0) < width)) {
                        pixel = imageIn[(lidx + colGroups * get_local_size(0)) * width + (lidy + rowGroups * get_local_size(0))];
                        atom_add(&local_hist[pixel], 1);
            }
            barrier(CLK_LOCAL_MEM_FENCE);

            //fill the histogram with the temp histogram
            if (local_index < BINS) atom_add(&histogram[local_index], local_hist[local_index]);

}
```

Figure 1 Koda histograma v kernel.cl

Delovanje

V funkcijo podamo izvorno sliko, ki je bila predhodnje spremenjena v sivinsko obliko ter razbita na “surove” (raw) bite, spremenljivko histogram, v katero bomo shranili rezultat ter višino in širino slike.

Nato v funkciji izračunamo lokalne in globalne indekse. Globalni indeks zračunamo po formuli `gidy * width + gidx` (da se skozi spreminjanje pomikamo čez celotno sliko potrebujemo v izračun všteti še širino). Lokalni index deluje na podoben način, vendar tukaj uporabimo velikost work groupa. Nato definiramo lokalni histogram in ga sprva napolnimo z vrednostmi 0. Za tem sledi barrier – počakamo, da se vrednosti napolnijo nato nadaljujemo. Nato sledi dolg pogojni stavek v katerem testiramo, če se še nahajamo znotraj mej slike (višina in širina). V kolikor se nahajamo znotraj slike pridobimo vrednost piksla (0-255). Nato pa v naš lokalni histogram katerih vsebuje števila pojavitev teh pikslov podamo piksel kot indeks ter ga povečamo za 1. Za pogojnim stavkom sledi še en barrier, ki nam omogoča da počakamo, da se vse niti izvedejo in naš lokalni histogram napolni.

Takoj za tem zapolnimo naš izhodni histogram z vrednostmi lokalnega histograma v kolikor je lokalni index manjši od števila pikslov.

Razlog za implementacijo

Odločil sem se za to implementacijo, ker smo jo obdelali na vajah, mi je bila razumljiva, ter skozi testiranje optimalna.

Izračun komulativnega histograma

Predpostavke:

- Lokalna velikost: BINS
- 1 dimenzionalen NDRange
- BINS = 256

Kernel koda

```
__kernel void CDF_GPU(__global unsigned int* histogram, __global unsigned int* cdf){

	/*nieve way: */
	/*cdf[0] = histogram[0];
	for (int i = 1; i < BINS; i++) {
		cdf[i] = cdf[i - 1] + histogram[i];
	} */

	int lid = get_local_id(0);
	int offset = 1;
	int n = BINS;

	__local unsigned int local_cdf[BINS + 1];

	local_cdf[lid] = histogram[lid];

	for (int d = n >> 1; d > 0; d >>= 1){
		__syncthreads();
		if (lid < d){
			int ai = offset * (2 * lid + 1) - 1;
			int bi = offset * (2 * lid + 2) - 1;
			local_cdf[bi] += local_cdf[ai];
		}
		offset *= 2;
	}
	if (lid == 0) {
		local_cdf[n] = local_cdf[n - 1];
		local_cdf[n - 1] = 0;
	}
	for (int d = 1; d < n; d *= 2){
		offset >>= 1;
		__syncthreads();
		if (lid < d){
			int ai = offset * (2 * lid + 1) - 1;
			int bi = offset * (2 * lid + 2) - 1;
			float t = local_cdf[ai];
			local_cdf[ai] = local_cdf[bi];
			local_cdf[bi] += t;
		}
	}
	__syncthreads();

	//prvi korak so same 0-le.
	cdf[lid] = local_cdf[lid + 1];
}
```

Figure 2 Koda CDF v kernel.cl

Delovanje

Deluje na principu AVL dreves. Na začetku inicializiramo lokalni index ter lokalni cdf velikosti 257 (256 + 1 – zaradi tega ker je prvi korak prenos ničle). Nato zapolnimo lokalni cdf z vrednostmi histograma, ki smo jih izračunali predhodnje. Prvi del algoritma predstavlja redukcijo. Pomikamo se po drevesu ter izračunamo delne vrednosti na podlagi odmika ter lokalnega indeksa. Redukcija je zato ker se na koncu tega dela na izvornem indeksu nahaja seštevek vseh pod-delov polja.

Med prvim in drugim delom preverjamo ali je lokalni indeks enak 0. V tem primeru pomaknemo vrednost lokalnega cdf za eno nazaj ter predhodnjo nastavimo na 0. S tem popravimo prvi korak algoritma ki samo prenese ničlo na konec.

V drugem delu pa se pomikamo nazaj in sestavljamo nazaj polje tako, da se ničle pomaknejo nazaj na začetek, ter vmesne seštete vrednosti pomaknejo na njihovo prvotno mesto. Na koncu ignoriramo prvi korak ter cdf napolnimo z lokalnim cdf z zamaknjenim indeksom za 1.

Razlog za implementacijo

Iskreno sem imel s tem algoritmom največ težav. Povezava, ki ste nam ga objavili v navodilih je kar pripomorel pri reševanju tega dela, ter vizualizacije poteka algoritma so prav tako pomagale pri reševanje algoritma findmin (vendar o tem nekoliko pozneje). Kot v dokumentu piše je zgoraj narejen algoritem optimalen, a ne popolnoma zaradi konfliktov spomina na strani grafične kartice. Potrebno bi bilo implementirati še, da do tega ne bi prišlo vendar mi ni uspelo razrešiti problema bank konfliktov.

Izračun izenačenja

Predpostavke:

- Lokalna velikost: 32
- 2 dimenzionalen NDRange
- BINS = 256

Kernel koda

```
__kernel void EqualizeGPU(__global unsigned int *histogram, __global unsigned int *cdf, __global unsigned char* imageOut, int
width, int height) {

	int i = get_global_id(0);
	int j = get_global_id(1);
	unsigned long imageSize = width * height;
	unsigned long cdfmin = findMin(cdf);
	if (get_global_id(1) == 0 && get_global_id(0) == 0) {
		printf("Min: %u\n", cdfmin);
	}

	if (j < width && i < height) {
		imageOut[i * width + j] = scale(cdf[imageOut[i * width + j]], cdfmin, imageSize);
	}
}
inline unsigned char scale(unsigned long cdf, unsigned long cdfmin, unsigned long imageSize) {
	float scale;
	scale = (float)(cdf - cdfmin) / (float)(imageSize - cdfmin);
	scale = round(scale * (float)(BINS - 1));
	return (int)scale;
}
inline unsigned int findMin(__global unsigned int* cdf) {
	/* ------------------------ Naiven nacin ------------------------------*/
	/*unsigned int min = 0;
	for (int i = 0; min == 0 && i < BINS; i++) {
		min = cdf[i];
	}
	return min;*/
	/*---------------------------- Reduction---------------------------------*/
	//isto kot pri cdfju
	int local_x = get_local_id(1);
	int local_y = get_local_id(0);
	int local_index = local_x * get_local_size(0) + local_y;

	int offset = 1;

	__local unsigned long cdf_local[BINS];

	if (local_index < BINS)cdf_local[local_index] = cdf[local_index];

	barrier(CLK_LOCAL_MEM_FENCE);
	for (int d = BINS >> 1; d > 0; d >>= 1) {
		if (local_index < d) {
			int normal = offset * (2 * local_index + 1) - 1;
			int next = offset * (2 * local_index + 2) - 1;

			if (cdf_local[next] > cdf_local[normal] && cdf_local[normal] != 0) {
				cdf_local[next] = cdf_local[normal];
			}

		}
		offset *= 2;
		barrier(CLK_LOCAL_MEM_FENCE);
	}
	return cdf_local[BINS-1];
	/* ------------------------ faster way ----------------------*/
	/*int local_x = get_local_id(1);
	int local_y = get_local_id(0);

	int local_index = local_x * get_local_size(0) + local_y;
	__local unsigned int min;

	if (local_index != 0 && cdf[local_index] != 0 && cdf[local_index - 1] == 0)min = cdf[local_index];
	barrier(CLK_LOCAL_MEM_FENCE);

	return min; */
}
```

Figure 3 Funkcija Izenačenja v kernel.cl

Delovanje

Delovanje bomo pri zadnjem primeru razdelili na tri dele:

Izenačenje (EqualizeGPU)

To je glavna funkcija v kateri se kličeta funkciji scale in findmin. Glavni namen funkcije je zapisovanje slike. In kopiranje rezultatov nazaj v glavni program, kjer se slika nato ustvari.

Scale

Funkcija ni nič spremenjena (nič paraleliziranja).

CdfMin

Pristopa sem se lotil na dva načina:

1. Redukcija: Deluje na podoben način kot pri računanju cdfa. Cdf shranimo v lokalni cdf ter se pomikamo skozi njega ter na podlagi odmika primerjamo 2 vrednosti. V kolikor je prva vrednost manjša od druge ter različna od 0 povozimo vrednost z vrednostjo odmika ter s tem manjšamo polje. Na koncu nam minimum vrednost ostane v zadnjem elementu polja.
2. Min: Pomikamo čez polje samo enkrat. V primeru če najdemo vrednost, ki ni nič in predhodno vrednost, ki je enaka nič nam vrne vrednost. Na ta način se lahko izognemo redukciji in pospešimo iskanje.

Razlog za implementacijo

Redukcijo sem implementiral zaradi omembe v navodilih, nato sem se pa vprašal ali je še kakšen hitrejši način. Oba načina sem nato med seboj tudi primerjal in prišel do ugotovitve, da kljub temu oba vrneta podobno sliko, algoritma ne delujeta isto na vseh slikah. Obstajajo primeri, kjer min način vrne vrednost na napačnem indeksu. To nastane zaradi tega, ker se na indexsu 0 cdfja začne vrednost različna od 0. Oči tega sicer ne zaznajo, vendar je sprememba očitna v vrednosti – saj se vsi indexi končnega histograma zamaknejo v vrednosti za 1 pixel.

Eksperiment

Naprava za testiranje:

Asus laptop - ASUSTek Computer Inc.
CPU: Intel® Core™ i7-7700HQ CPU @ 2.80GHz 2.81GHz
Memory: 16GB
System type: 64-Bit Operating System, x64-based processor
GPU: GeForce GTX 1050 Ti

- Max work item sizes 1024x1024x64
- Max work group size 1024
- Preferred work group size multiple 32

Uporabljene slike:

Slika	Dimenzija	Sivinska
big.png	3072x2048	DA
bog.png	874x544	DA
bridge.png	2736x1824	DA
car.png	600x338	DA
chang.png	512x512	NE
crowd.png	800x600	NE
hill.png	1024x683	DA
nature.png	1920x1080	NE
sneg.png	1024x768	DA
tick.png	2560x1920	DA

Merjenje časa

Čas se meri pri funkciji init(), ki inicializira večino spremenljivk za vzpostavljanje povezave grafične kartice, kot prebiranje slike. Za tem se pa čas meri in sešteva v vsaki funkciji posebej – pri kopiranju rezultatov iz grafične v glavno strukturo. Na koncu pa merimo še čas pri zapisu slike v GPU_output.png.

V namen testiranja algoritma min (najmanši element cdf-ja) sem testiral program z redukcijo in min algoritmom posebej. V primeru, da min algoritem ni vedno vrnil pravilnega rezultata (kot sem že omenil zgoraj) sem pri tisti sliki obarval čas min-a z rdečo barvo in ga NISEM upošteval pri končnih ugotovitvah.

Rezultati

Skozi testiranje sem uporabil 10 fotografij. Vsaka fotografija je bila testirana vsaj okoli 15 krat, da sem pridobil povprečen čas. Slike sem razdelil na dve večji skupini:

Velike slike

V to kategorijo spadajo slike nad 1920x1080 oz. slike katere vsebujejo več kot 2,073,600 število pikslov.

- big.png (3072x2048)

Figure 4 Big.png pred izenačenjem

Figure 5 Big.png po izenačenju

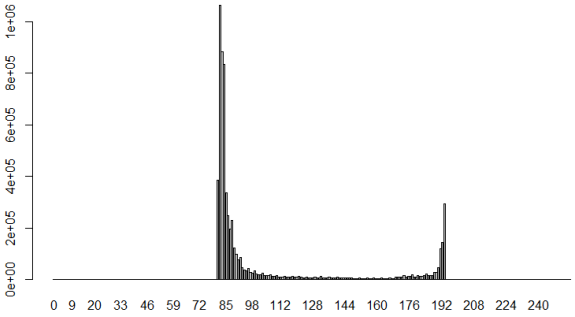


Figure 6 Histogram pred izenačenjem

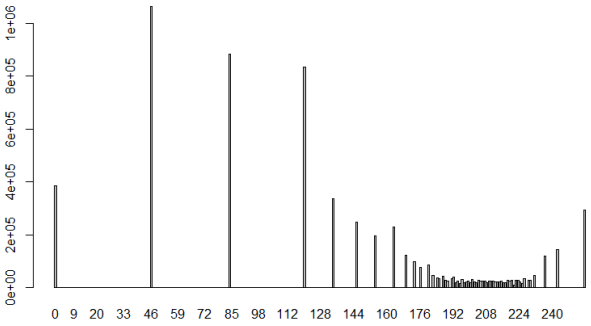


Figure 7 Histogram po izenačenju

Tabela izračuna časa za velike slike:

Slika	Velikost	Čas CPE	Čas GPE (redukcija)	Čas GPE (min)
big.png	3072x2048	1.2848 s	0.8760 s	0.7846 s
bridge.png	2736x1824	1.2292 s	0.8740 s	0.8323 s
tick.png	2560x1920	0.9467 s	0.6235 s	0.5813 s
nature.png	1920x1080	0.4673 s	0.3331 s	0.3262 s

V tej kategoriji slik je razvidno da je algoritem na grafični kartici:

- z redukcijo pohitril za povprečno: 1.4x
- z min algoritmom pohitril za povprečno: 1.7x

Manjše slike

V to kategorijo spadajo slike, ki se nahajajo pod velikostjo 1920x1080 oz. pod 2,073,600 število pikslov.

- chang.png (512x512)

Figure 8 Chang.png pred izenačenjem

Figure 9 Chang.png po izenačenju

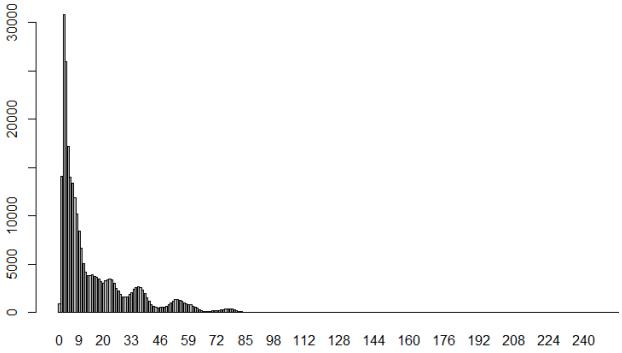


Figure 10 Histogram pred izenačenjem

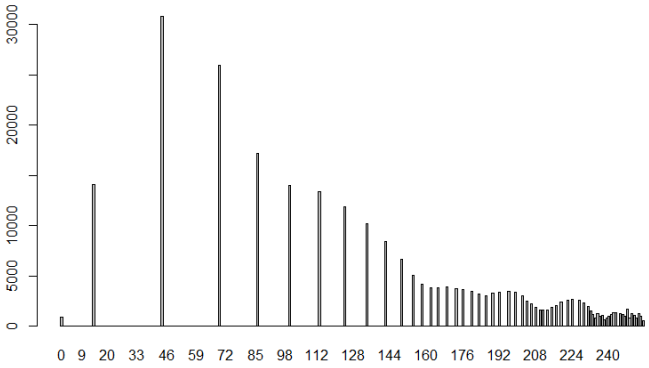


Figure 11 Histogram po izenačenju

Tabela izračuna časa za majhne slike:

Slika	Velikost	Čas CPE	Čas GPE (redukcija)	Čas GPE (min)
sneg.png	1024x768	0.1422 s	0.0890 s	0.0820 s
hill.png	1024x683	0.1400 s	0.0870 s	0.0810 s
crowd.png	800x600	0.0842 s	0.0565 s	0.0625 s
bog.png	874x544	0.0875 s	0.0890 s	0.0565 s
chang.png	512x512	0.0465 s	0.0480 s	0.0334 s
car.png	600x338	0.0360 s	0.0330 s	0.0260 s

V tej kategoriji slik je razvidno da je algoritem na grafični kartici:

- z redukcijo pohitril za povprečno: 1.4 – 1.5x
- z min algoritmom pohitril za povprečno: 1.5x

Slike do okoli velikosti 874x544 so še imele opazno pohitritev, vse pod njo pa se bližajo ali segajo čez izvrševalni čas centralno procesne enote.

Ugotovitve

Paraleliziran algoritem je optimalen na slikah katere velikosti so nad 850 x 540.

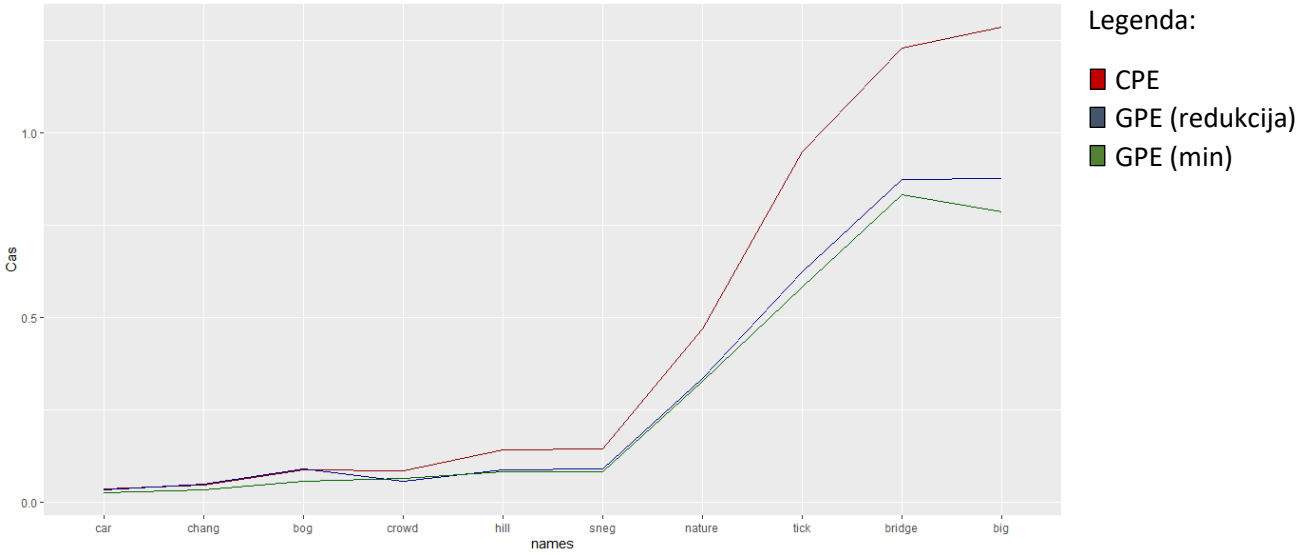


Figure 12 Prikaz posameznih izvršilnih časov za vse teste (v sekundah)

Iz grafa lahko razberemo, da je redukcija veliko bolj učinkovita pri večjih slikah. Prav tako pa testiranje z min odvisno od sreče, kako hitro bo našel tisto vrednost, katerih sledi predhodnji element 0. Hitrost algoritma min začne iztopati šele pri večjih slikah.

Možne nadgradnje

V glavnem programu bi prišparal na času tako, da ne bi prebiral iz kernela ven podatkov vendar samo nadaljeval z njimi v naslednjem. To sem tudi želel narediti vendar nisem razumel kako lahko potem spremenim NDRange medtem ko se koda še izvaja znotraj kernela. V kolikor bi odpravil to težavo, bi pričakoval pohitritev za kar nekaj milisekund, saj vračanje ne bi bilo potrebno.