

GRAYSCALE - EDGE DETECTION

Labo Geavanceerde Computertechniek

(JLIZNM)

Jona CAPPELLE & JONAS BOLLE

17 mei 2020



Sessie Datum: 4 Mei, 2020

Partners: Jona Cappelle

Jonas Bolle

Klas: MELICTEES

Begeleider: Stijn Crul

Inhoudsopgave

1	Inleiding	2
2	Probleemstelling en analyse performantie	2
2.1	Grayscale	2
2.1.1	Tijd op GPU	2
2.1.2	Tijd op CPU	3
2.2	Edge detection	4
2.2.1	Tijd op GPU	4
2.2.2	Tijd op CPU	5
3	Besluit	7
A	CODE GRayscale	8
B	CODE EDGE DETECTION	13

Lijst van figuren

1	Opbouw PNG afbeelding	2
2	Resultaat grayscale	3
3	Meting tijd op GPU bij verschillende blocksizes	3
4	Resultaat edge detection GPU	5
5	Resultaat edge detection CPU	6

Lijst van tabellen

1	Samenvattende tabel tijden CPU en GPU	7
---	---	---

List of Listings

1	Keuze Blocksize en nBlock	5
---	-------------------------------------	---

1 Inleiding

In dit labo van geavanceerde computertechniek gaan we een grayscale van een afbeelding maken en edge detection toepassen.

Een grayscale operatie zet alle kleurenwaarden om naar zwart-wit waarden met een equivalente luminantie. Een edge detection operatie detecteert grote verschillen tussen naburige pixels.

2 Probleemstelling en analyse performantie

2.1 Grayscale

De bedoeling van het eerste deel van het labo is het omzetten van een kleurenafbeelding in een grayscale afbeelding. Het resultaat hiervan is terug te vinden in figuur 2. Bij de grayscale zijn er verschillende mogelijkheden om deze te implementeren. Men kan het gemiddelde nemen van de RGB waarden en dit gemiddelde toekennen aan elke R, G en B waarde. Er kan ook gewerkt worden met verschillende verhoudingen voor RGB waarden. In dit labo hebben we voor deze eerste optie gekozen.

De ingelezen afbeelding zetten we met 'lodepng' om naar een ééndimensionale array met structuur zoals weergegeven in figuur 1.

image[i]	image[i+1]	image[i+2]	image[i+3]
Red	Green	Blue	Alpha

Figuur 1: Opbouw PNG afbeelding

Hier moeten we er rekening mee houden dat het 'alpha' kanaal, dat de opacity bepaalt, altijd de waarde 255 moet hebben om de afbeelding niet doorzichtig te maken.

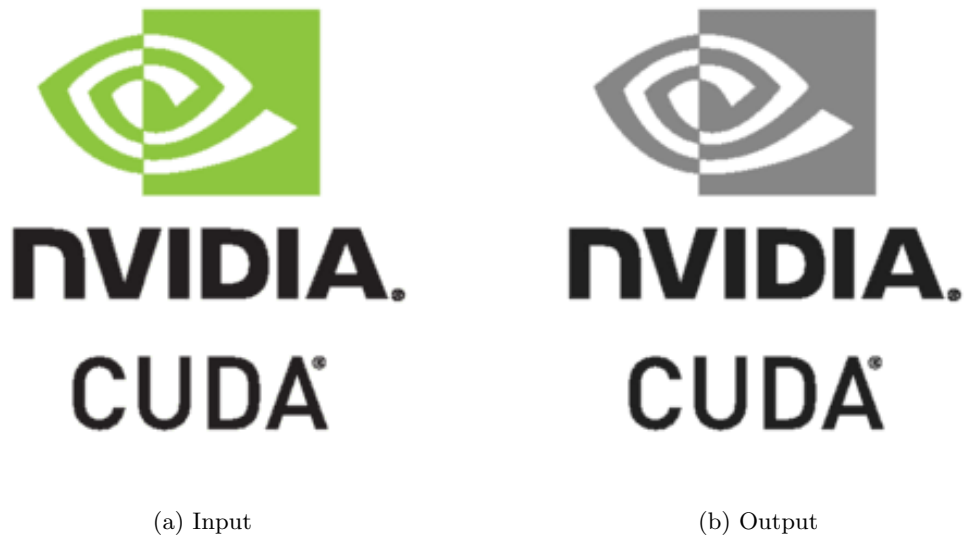
De code van grayscale is terug te vinden in deel A op pagina 8 en volgende.

2.1.1 Tijd op GPU

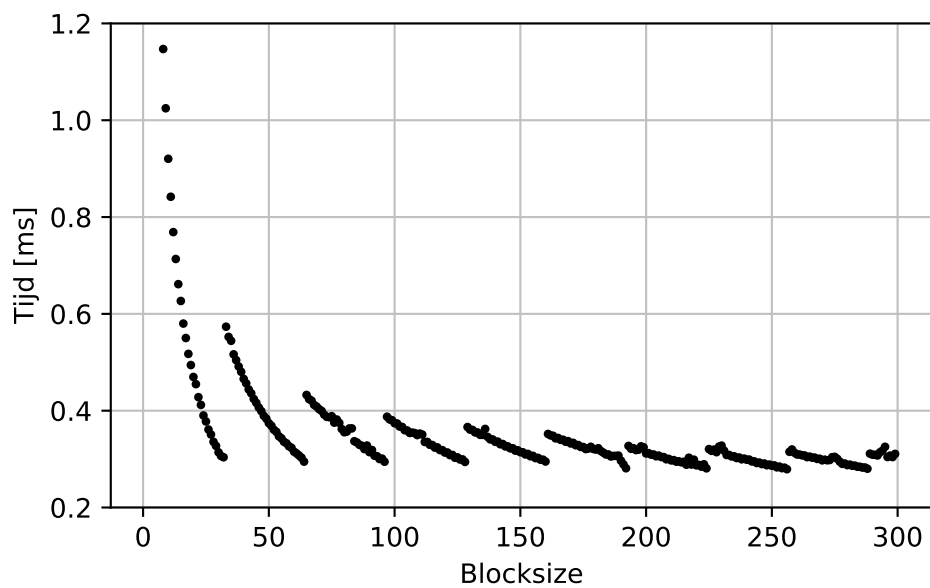
We meten de uitvoertijd op de GPU bij verschillende blocksizes. Hier werd het kopiëren van de data van en naar de GPU niet mee in rekening gebracht. De laagste tijden worden bekomen bij blocksizes met veelvoud van 32. Een grafiek hiervan is terug te vinden in figuur 3.

De tijd die het duurt om de grayscale uit te voeren is:

$$t_{GPU} = 304 \mu s$$



Figuur 2: Resultaat grayscale



Figuur 3: Meting tijd op GPU bij verschillende blocksizes

2.1.2 Tijd op CPU

Wanneer we een gray scale implementatie op de CPU schrijven, duurt het veel langer om deze uit te voeren namelijk:

$$t_{CPU} = 656 \mu s$$

Dit is meer dan dubbel zo lang dan de uitvoering op de GPU (zonder mee kopiëren van de data). Uit vorig labo konden we al besluiten dat het langer zal duren met het mee kopiëren van de data maar dat bij een groot aantal fotos die omgezet moeten worden de GPU met zijn parallelisatie toch veel sneller zal zijn. Om deze reden werd deze meting hier niet opnieuw gedaan.

2.2 Edge detection

In het tweede deel van het labo gaan we edge detection toepassen op een afbeelding. Het resultaat van deze operatie is terug te vinden in figuur 4. Edge detection is een mooi vervolg op grayscale, daar we de grayscale ook nodig hebben om aan edge detection te doen. Voor de edge detection werd gewerkt met de gekende sobel filter. Het werkt op basis van een 3x3 matrix vermenigvuldiging met elke pixel van het beeld. Wanneer er veel verschil is naburige waarden van pixels, zal deze bewerking ofwel een zeer groot, of een zeer klein resultaat opleveren. We maken gebruik van twee 3x3 matrices, één voor de veranderingen in de x-richting te detecteren (G_x) en één voor de veranderingen in de y-richting te detecteren (G_y). Om deze waarden samen te voegen (G) en negatieve getallen te vermijden, wordt hier ook de wortel van de kwadraten van beide x en y resultaten genomen.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

[1]

De code van edge detection is terug te vinden in deel B op pagina 13 en volgende.

2.2.1 Tijd op GPU

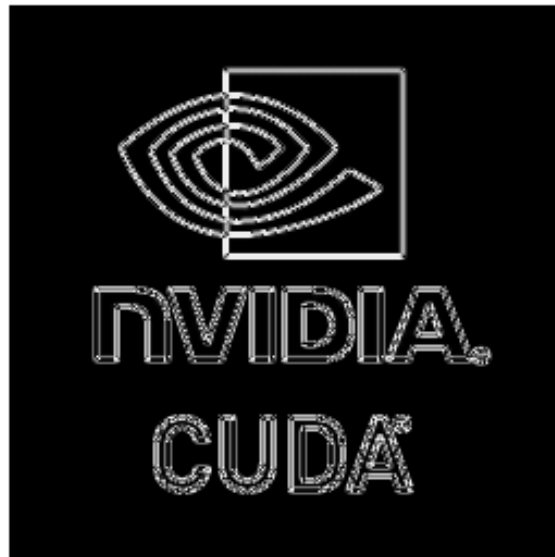
Als we de edge detectie op de GPU uitvoeren, duurt het:

$$t_{GPU} = 336 \mu\text{s}$$

Hierbij werd rekening gehouden met het optimaal aantal threads per block een veelvoud te kiezen van 32. Zie listing 2.2.1. Er werd gekozen voor een Blocksize van 64 op 16 = 1024. nBlocks volgt hieruit: 64 (4 op 16). Door hier in 2D de threads te organiseren, laat dit ons toe de edge detection op de gpu intuïtiever te coderen. De threads die naast



(a) Input



(b) Output

Figuur 4: Resultaat edge detection GPU

en onder elkaar staan, zullen ook de waarden van de pixels berekenen die naast en onder elkaar staan.

Listing 1: Keuze Blocksize en nBlock

```
// Choose Blocksize & nBlock in 2D  
dim3 BLOCKSIZE(64,16);  
dim3 nBlocks(ceil(width/64),ceil(height/16));
```

2.2.2 Tijd op CPU

Als we de edge detectie op de CPU uitvoeren, duurt het:

$$t_{CPU} = 3496 \mu s$$

Dit is 10 keer langer dan de bewerking op de GPU! Hoe intensiever de taken worden, hoe groter de tijdswinst als we te werk gaan met een GPU.

Het resultaat van de edge detection op de CPU is terug te vinden in figuur 5.



(a) Input



(b) Output

Figuur 5: Resultaat edge detection CPU

3 Besluit

Grayscale	
Tijd op GPU	304 μ s
Tijd op CPU	656 μ s
Edge detection	
Tijd op GPU	336 μ s
Tijd op CPU	3496 μ s

Tabel 1: Samenvattende tabel tijden CPU en GPU

Uit tabel 1 kunnen we besluiten dat het uitvoeren van bewerkingen op de GPU veel sneller kan gebeuren dan op een CPU.

Ook hier zien we terugkomen dat 32 threads per block optimaal is voor performantie. Dit komt aangezien CUDA GPU's kernels runnen die blokken van threads gebruiken met een veelvoud van 32. Indien de gebruiker ook dit veelvoud van 32 hanteert, zal er zoveel mogelijk parallelisatie optreden. Wanneer niet voor dit veelvoud gekozen wordt, zullen enkele threads niet gebruikt worden en zal het programma bijgevolg trager uitgevoerd worden.

Een andere conclusie die we kunnen maken is dat de edge detection langer duurt dan de grayscale. Bij de grayscale wordt enkel het gemiddelde van de pixels genomen: twee optellingen en één vermenigvuldiging. Bij de edge detection wordt er vermenigvuldigd met twee maal een 3x3 matrix, wat meer tijd vergt (meer vermenigvuldigingen en optellingen).

Conclusie: Bij zeer kleine workloads heeft het kopiëren van data van en naar de GPU een grote invloed op de uitvoeringstijd. Bij grotere parallele workloads is het veel sneller om een GPU te gebruiken dan een CPU. Dit kunnen we ook zien aan de speedup factor van de grayscale functie en de edge detection functie: de GPU voert de grayscale functie 2x sneller uit dan de CPU, bij de edge detection is dit x10!

A CODE GRAYSCALE

```
1  //////////////////////////////////////
2  // GRAYSCALE -- Jona Cappelle -- Jonas Bolle
3  //////////////////////////////////////
4
5  // includes, system
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <math.h>
10
11 // includes CUDA
12 #include <cuda_runtime.h>
13
14 // includes, project
15 #include <helper_cuda.h>
16 #include <helper_functions.h> // helper functions for SDK examples
17
18 // own includes
19 #include "iostream"
20 #include "cstdlib"
21 #include "time.h" // timing on cpu
22 #include "lodepng.h" // PNG image read
23
24 extern "C"
25
26
27 void decodeOneStep(const char* filename) {
28     unsigned error;
29     unsigned char* image = 0;
30     unsigned width, height;
31
32     error = lodepng_decode32_file(&image, &width, &height,
33     ↪ filename);
34     if(error) printf("error %u: %s\n", error,
35     ↪ lodepng_error_text(error));
36
37     /*use image here*/
38
39     free(image);
40 }
```

```

40 void encodeOneStep(const char* filename, const unsigned char* image,
    ↪ unsigned width, unsigned height) {
41     /*Encode the image*/
42     unsigned error = lodepng_encode32_file(filename, image, width,
    ↪ height);
43
44     /*if there's an error, display it*/
45     if(error) printf("error %u: %s\n", error,
    ↪ lodepng_error_text(error));
46 }
47
48 ////////////////////////////////////
49 // KERNEL GRAYSCALE
50 ////////////////////////////////////
51 int BLOCKSIZE;
52
53 // GPU
54 __global__ void grayscale(unsigned char* image, unsigned char*
    ↪ grayImage, unsigned width, unsigned height)
55 {
56     int j = (blockIdx.x*blockDim.x + threadIdx.x)*4;
57
58     if(j < width*height*4)
59     {
60         grayImage[j] = (image[j]+image[j+1]+image[j+2])/3;
61         grayImage[j+1] = (image[j]+image[j+1]+image[j+2])/3;
62         grayImage[j+2] = (image[j]+image[j+1]+image[j+2])/3;
63         grayImage[j+3] = 255;
64     }
65
66 }
67
68 // CPU
69 void grayscale_cpu(unsigned char* image, unsigned width, unsigned
    ↪ height)
70 {
71     for(int j=0; j < (width*height*4); j+=4)
72     {
73         image[j] = (image[j]+image[j+1]+image[j+2])/3;
74         image[j+1] = (image[j]+image[j+1]+image[j+2])/3;
75         image[j+2] = (image[j]+image[j+1]+image[j+2])/3;
76     }
77 }

```

```
78
79 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
80 // Program main
81 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
82 int main()
83 {
84     ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
85     // Load PNG file
86     ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
87     float millis = 0;
88     unsigned char *image_in, *image_out, *image_in_dev,
89     ↪ *image_out_dev;
90     unsigned width, height, width_dev, height_dev;
91
92     const char* filename = "test.png";
93
94     unsigned error;
95     unsigned char* image = 0;
96
97     error = lodepng_decode32_file(&image, &width, &height,
98     ↪ filename);
99     if(error) printf("error %u: %s\n", error,
100     ↪ lodepng_error_text(error));
101
102     // allocate arrays on host
103     image_in = (unsigned char *)malloc(width*height*4 *
104     ↪ sizeof(char));
105     image_out = (unsigned char *)malloc(width*height*4 *
106     ↪ sizeof(char));
107
108     // File to store measured time data
109     FILE *f = fopen("data.csv", "w");
110
111     for (int BLOCKSIZE = 1; BLOCKSIZE < 300; BLOCKSIZE++)
112     {
113         int nBlocks = (width*height*4) / BLOCKSIZE + ((width*height*4) %
114         ↪ BLOCKSIZE == 0 ? 0 : 1);
115         printf("nBlocks: %d", nBlocks);
116
117         // StopwatchInterface *timer = 0;
118         // sdkCreateTimer(&timer);
119         // sdkResetTimer(&timer);
120         // sdkStartTimer(&timer);
```

```
115
116 //      grayscale_cpu(image, width, height);
117
118 //      sdkStopTimer(&timer);
119 //      printf("Tijd: %f\n", sdkGetTimerValue(&timer));
120 //      sdkDeleteTimer(&timer);
121
122 //      // allocate arrays on device
123 cudaMalloc((void **)&image_in_dev, width*height*4 *
124 ↪      sizeof(char));
125
126 cudaMalloc((void **)&image_out_dev, width*height*4 *
127 ↪      sizeof(char));
128
129 cudaMemcpy(image_in_dev, image, width*height*4*sizeof(char),
130 ↪      cudaMemcpyHostToDevice);
131
132 cudaMemcpy(image_out_dev, image_out,
133 ↪      width*height*4*sizeof(char), cudaMemcpyHostToDevice);
134
135 //      unsigned *width_1 = &width;
136 //      cudaMemcpy(width_dev, width_1, sizeof(unsigned),
137 ↪      cudaMemcpyHostToDevice);
138 //      cudaMemcpy(height_dev, &height, sizeof(unsigned),
139 ↪      cudaMemcpyHostToDevice);
140
141 // Record time on GPU with cuda events
142 cudaEvent_t start, stop;
143 cudaEventCreate(&start);
144 cudaEventCreate(&stop);
145
146 //////////////////////////////////////
147 cudaEventRecord(start);
148 grayscale <<< nBlocks, BLOCKSIZE >>> ( image_in_dev,
149 ↪      image_out_dev, width, height );
150 cudaEventRecord(stop);
151 //////////////////////////////////////
152
153 cudaMemcpy(image_in, image_in_dev, width*height*4*sizeof(char),
154 ↪      cudaMemcpyDeviceToHost);
155
156 cudaMemcpy(image_out, image_out_dev,
157 ↪      width*height*4*sizeof(char), cudaMemcpyDeviceToHost);
158
159
160 cudaEventSynchronize(stop);
161 cudaEventElapsedTime(&millis, start, stop);
```

```
149
150     printf("Tijd op GPU: %f\n", millis);
151     fprintf(f, "%d,%f\n", BLOCKSIZE, millis);
152 }
153
154     // Close the file
155     fclose(f);
156
157     // Save the result image
158     const char* output_filename = "output.png";
159     encodeOneStep(output_filename, image_out, width, height);
160
161     // Free memory
162     free(image_in);
163     free(image_out);
164
165     cudaFree(image_in_dev);
166     cudaFree(image_out_dev);
167
168     printf("Done!");
169
170     return 0;
171 }
```

B CODE EDGE DETECTION

```
1  //////////////////////////////////////
2  // EDGE DETECTION -- Jona Cappelle -- Jonas Bolle
3  //////////////////////////////////////
4
5  // includes, system
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <math.h>
10
11 // includes CUDA
12 #include <cuda_runtime.h>
13
14 // includes, project
15 #include <helper_cuda.h>
16 #include <helper_functions.h> // helper functions for SDK examples
17
18 // own includes
19 #include "iostream"
20 #include "cstdlib"
21 #include "time.h" // timing on cpu
22 #include "lodepng.h" // PNG afbeelding inlezen
23
24 extern "C"
25
26 void decodeOneStep(const char* filename) {
27     unsigned error;
28     unsigned char* image = 0;
29     unsigned width, height;
30
31     error = lodepng_decode32_file(&image, &width, &height,
32     ↪ filename);
33     if(error) printf("error %u: %s\n", error,
34     ↪ lodepng_error_text(error));
35
36     /*use image here*/
37
38     free(image);
39 }
```

```

39 void encodeOneStep(const char* filename, const unsigned char* image,
    ↪ unsigned width, unsigned height) {
40     /*Encode the image*/
41     unsigned error = lodepng_encode32_file(filename, image, width,
    ↪ height);
42
43     /*if there's an error, display it*/
44     if(error) printf("error %u: %s\n", error,
    ↪ lodepng_error_text(error));
45 }
46
47 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
48 // KERNEL EDGE DETECTION
49 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
50 int BLOCKSIZE;
51
52 // GPU
53 __global__ void edge(unsigned char* orig, unsigned char* result,unsigned
    ↪ width,unsigned height)
54 {
55
56     int x = (threadIdx.x + blockIdx.x * blockDim.x)*4;
57     int y = (threadIdx.y + blockIdx.y * blockDim.y);
58     float dx, dy;
59     width=4*width;
60     if( x > 0 && y > 0 && x < (width-1) && y < (height-1)) {
61         dx = (-1* orig[(y-1)*width + (x-4)]) + (-2*orig[y*width+(x-4)])
    ↪ + (-1*orig[(y+1)*width+(x-4)]) +
62         ( orig[(y-1)*width + (x+4)]) + ( 2*orig[y*width+(x+4)])
    ↪ + ( orig[(y+1)*width+(x+4)]);
63         dy = ( orig[(y-1)*width + (x-4)]) + ( 2*orig[(y-1)*width+x])
    ↪ + ( orig[(y-1)*width+(x+4)]) +
64         (-1* orig[(y+1)*width + (x-4)]) + (-2*orig[(y+1)*width+x])
    ↪ + (-1*orig[(y+1)*width+(x+4)]);
65         result[y*width + x] = sqrt( (dx*dx) + (dy*dy) );
66         result[y*width + x + 1] = sqrt( (dx*dx) + (dy*dy) );
67         result[y*width + x + 2] = sqrt( (dx*dx) + (dy*dy) );
68         result[y*width + x + 3] = 255;
69     }
70
71 }
72
73 // CPU

```



```

74 void edge_cpu(unsigned char* image,unsigned char* image_out, unsigned
    ↪ width, unsigned height)
75 {
76
77     for(int j=0; j < (width*height*4); j+=4)
78     {
79         image[j] = (image[j]+image[j+1]+image[j+2])/3;
80         image[j+1] = (image[j]+image[j+1]+image[j+2])/3;
81         image[j+2] = (image[j]+image[j+1]+image[j+2])/3;
82     }
83     int dx,dy,val;
84     int sobel_x[3][3] =
85     { { -1, 0, 1 },
86       { -2, 0, 2 },
87       { -1, 0, 1 } };
88
89     int sobel_y[3][3] =
90     { { -1, -2, -1 },
91       { 0, 0, 0 },
92       { 1, 2, 1 } };
93
94     for (int x=1; x < (width-1); x++)
95     {
96         for (int y=1; y < (height-1); y++)
97         {
98
99             dx = (sobel_x[0][0] * image[width*4 * (y-1) + (x-1)*4])
100                 + (sobel_x[0][1] * image[width*4 * (y-1) + x*4
101                 ↪ ])
102                 + (sobel_x[0][2] * image[width*4 * (y-1) +
103                 ↪ (x+1)*4])
104                 + (sobel_x[1][0] * image[width*4 * y +
105                 ↪ (x-1)*4])
106                 + (sobel_x[1][1] * image[width*4 * y + x *4
107                 ↪ ])
108                 + (sobel_x[1][2] * image[width*4 * y +
109                 ↪ (x+1)*4])
110                 + (sobel_x[2][0] * image[width*4 * (y+1) +
111                 ↪ (x-1)*4])
112                 + (sobel_x[2][1] * image[width*4 * (y+1) + x *4
113                 ↪ ])
114                 + (sobel_x[2][2] * image[width*4 * (y+1) +
115                 ↪ (x+1)*4]);

```

```

108
109         dy = (sobel_y[0][0] * image[width*4 * (y-1) + (x-1)*4])
110             + (sobel_y[0][1] * image[width*4 * (y-1) + x *4
111                 ↪ ])
112             + (sobel_y[0][2] * image[width*4 * (y-1) +
113                 ↪ (x+1)*4])
114             + (sobel_y[1][0] * image[width*4 * y +
115                 ↪ (x-1)*4])
116             + (sobel_y[1][1] * image[width*4 * y + x *4
117                 ↪ ])
118             + (sobel_y[1][2] * image[width*4 * y +
119                 ↪ (x+1)*4])
120             + (sobel_y[2][0] * image[width*4 * (y+1) +
121                 ↪ (x-1)*4])
122             + (sobel_y[2][1] * image[width*4 * (y+1) + x *4
123                 ↪ ])
124             + (sobel_y[2][2] * image[width*4 * (y+1) +
125                 ↪ (x+1)*4]);
126
127         int val = (int)sqrt((dx * dx) + (dy * dy));
128
129         if(val < 0) val = 0;
130         if(val > 255) val = 255;
131
132         image_out[width*4 * (y-1) + (x-1)*4] = val;
133         image_out[width*4 * (y-1) + (x-1)*4+1] = val;
134         image_out[width*4 * (y-1) + (x-1)*4+2] = val;
135         image_out[width*4 * (y-1) + (x-1)*4+3] = 255;
136     }
137 }
138
139 //////////////////////////////////////
140 // Program main
141 //////////////////////////////////////
142 int main()
143 {
144     //////////////////////////////////////
145     // Load PNG file
146     //////////////////////////////////////
147     float millis = 0;

```

```
142     unsigned char *image_in, *image_out, *image_in_dev,
        ↪ *image_out_dev;
143     unsigned width, height, width_dev, height_dev;
144
145     const char* filename = "test.png";
146
147     unsigned error;
148     unsigned char* image = 0;
149
150     error = lodepng_decode32_file(&image, &width, &height,
        ↪ filename);
151     if(error) printf("error %u: %s\n", error,
        ↪ lodepng_error_text(error));
152
153     // allocate arrays on host
154     image_in = (unsigned char *)malloc(width*height*4 *
        ↪ sizeof(char));
155     image_out = (unsigned char *)malloc(width*height*4 *
        ↪ sizeof(char));
156
157
158     FILE *f = fopen("data.csv", "w");
159
160     //     StopwatchInterface *timer = 0;
161     //     sdkCreateTimer(&timer);
162     //     sdkResetTimer(&timer);
163     //     sdkStartTimer(&timer);
164
165     //     edge_cpu(image, image_out, width, height);
166
167     //     sdkStopTimer(&timer);
168     //     printf("Tijd: %f\n", sdkGetTimerValue(&timer));
169     //     sdkDeleteTimer(&timer);
170
171     // Grayscale on CPU
172     for(int j=0; j < (width*height*4); j+=4)
173     {
174         image[j] = (image[j]+image[j+1]+image[j+2])/3;
175         image[j+1] = (image[j]+image[j+1]+image[j+2])/3;
176         image[j+2] = (image[j]+image[j+1]+image[j+2])/3;
177     }
178
179     // Allocate arrays on device
```

```
180     cudaMalloc((void **)&image_in_dev, width*height*4 *
      ↪     sizeof(char));
181     cudaMalloc((void **)&image_out_dev, width*height*4 *
      ↪     sizeof(char));
182
183     cudaMemcpy(image_in_dev, image, width*height*4*sizeof(char),
      ↪     cudaMemcpyHostToDevice);
184     cudaMemcpy(image_out_dev, image_out,
      ↪     width*height*4*sizeof(char), cudaMemcpyHostToDevice);
185
186     // Record time on GPU with cuda events
187     cudaEvent_t start, stop;
188     cudaEventCreate(&start);
189     cudaEventCreate(&stop);
190
191     ////////////////////////////////////
192     // Choose Blocksize & nBlock in 2D
193     dim3 BLOCKSIZE(64,16);
194     dim3 nBlocks(ceil(width/64),ceil(height/16));
195     ////////////////////////////////////
196
197     cudaEventRecord(start);
198     edge <<< nBlocks, BLOCKSIZE >>> ( image_in_dev, image_out_dev,
      ↪     width, height );
199     cudaEventRecord(stop);
200
201     cudaMemcpy(image_in, image_in_dev, width*height*4*sizeof(char),
      ↪     cudaMemcpyDeviceToHost);
202     cudaMemcpy(image_out, image_out_dev,
      ↪     width*height*4*sizeof(char), cudaMemcpyDeviceToHost);
203
204     cudaEventSynchronize(stop);
205     cudaEventElapsedTime(&millis, start, stop);
206
207     printf("Tijd op GPU: %f\n", millis);
208
209     //      fprintf(f, "%d,%f\n", BLOCKSIZE, millis);
210
211     fclose(f);
212
213     const char* output_filename = "output.png";
214     encodeOneStep(output_filename, image_out, width, height);
215
```

```
216     free(image_in);
217     free(image_out);
218
219     cudaFree(image_in_dev);
220     cudaFree(image_out_dev);
221
222     printf("Done!");
223
224     return 0;
225 }
```

Referenties

- [1] “Sobel operator - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Sobel_operator