GRAYSCALE - EDGE DETECTION
Labo Geavanceerde Computertechniek

(JLIZNM)

Jona Cappelle & Jonas Bolle

15 mei 2020

Sessie Datum:	9 Maart, 2020
Partners:	Jona Cappelle
	Jonas Bolle
Klas:	MELICTEES
Begeleider:	Stijn Crul

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

List of Listings

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 7 en volgende.

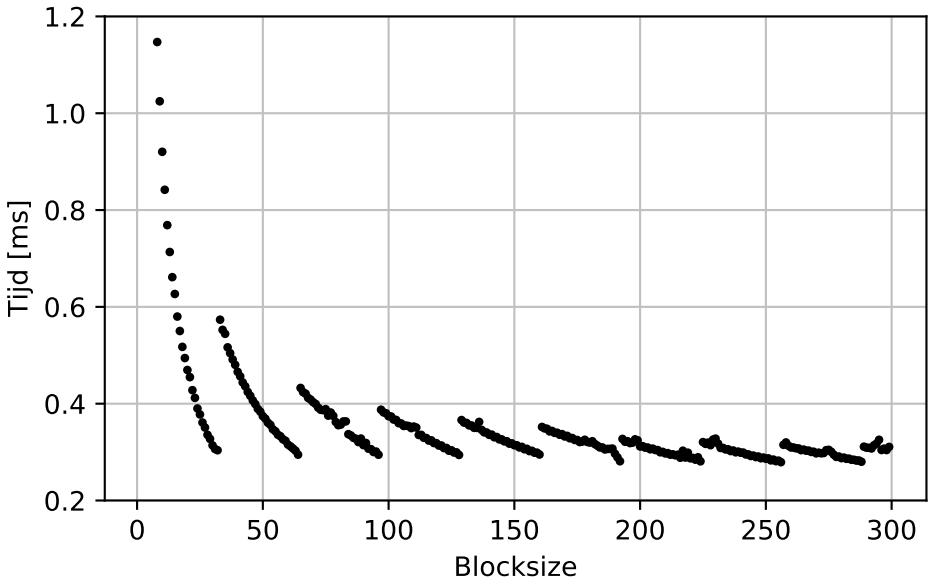
2.1.1 Tijd op GPU

We meten de uitvoertijd op de GPU bij verschillende blocksizes. Hier werd het copiëren van de data van en naar de GPU niet mee in rekening gebracht. De laagste tijden worden bekomen bij blocksizes met veelvouden van 32. Een grafiek hiervan is terug te vinden in figuur 3.
De tijd die het duurt om de grayscale uit te voeren is:

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

(a) Input (b) Output

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 copiëren van de data). Uit vorig labo konden we al besluiten dat het langer zal duren met het mee copiëren 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]

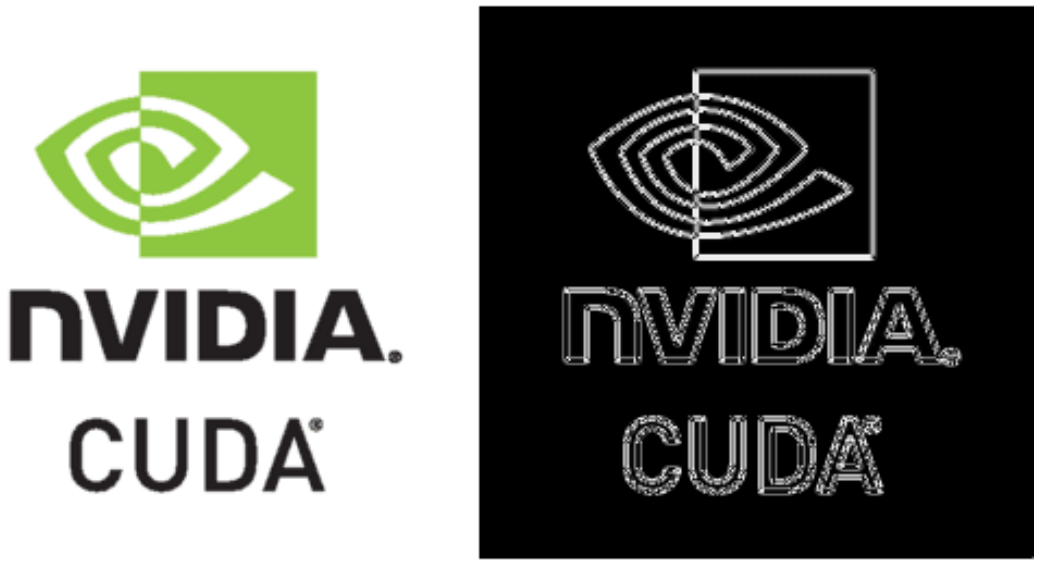
De code van edge detection is terug te vinden in deel B op pagina 12 en volgende.

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

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

3 Besluit

Grayscale		
	Tijd op GPU	304 µs
	Tijd op CPU	656 µs
Edge detection		
	Tijd op GPU	336 µs
	Tijd op CPU	0 µs

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 tragen uitgevoerd worden.
Een ander 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 grotere parallele workloads is het veel sneller om een GPU te gebruiken dan een CPU. Bij zeer kleine workloads heeft het copiëren van data van en naar de GPU een grote invloed op de uitvoeringstijd.

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,
           ↪  filename);
33         if(error) printf("error %u: %s\n", error,
           ↪  lodepng_error_text(error));
34
35         /*use image here*/
36
37         free(image);
38 }
39
```

```
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,
            ↪  *image_out_dev;
89          unsigned width, height, width_dev, height_dev;
90
91          const char* filename = "test.png";
92
93          unsigned error;
94          unsigned char* image = 0;
95
96          error = lodepng_decode32_file(&image, &width, &height,
            ↪  filename);
97          if(error) printf("error %u: %s\n", error,
            ↪  lodepng_error_text(error));
98
99          // allocate arrays on host
100         image_in = (unsigned char *)malloc(width*height*4 *
            ↪  sizeof(char));
101         image_out = (unsigned char *)malloc(width*height*4 *
            ↪  sizeof(char));
102
103         // File to store measured time data
104         FILE *f = fopen("data.csv", "w");
105
106 for (int BLOCKSIZE = 1; BLOCKSIZE < 300; BLOCKSIZE++)
107 {
108         int nBlocks = (width*height*4) / BLOCKSIZE + ((width*height*4) %
            ↪  BLOCKSIZE == 0 ? 0 : 1);
109         printf("nBlocks: %d", nBlocks);
110
111 //        StopWatchInterface *timer = 0;
112 //        sdkCreateTimer(&timer);
113 //        sdkResetTimer(&timer);
114 //        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 *
            ↪  sizeof(char));
124         cudaMalloc((void **)&image_out_dev, width*height*4 *
            ↪  sizeof(char));
125
126         cudaMemcpy(image_in_dev, image, width*height*4*sizeof(char),
            ↪  cudaMemcpyHostToDevice);
127         cudaMemcpy(image_out_dev, image_out,
            ↪  width*height*4*sizeof(char), cudaMemcpyHostToDevice);
128
129 //        unsigned *width_1 = &width;
130 //        cudaMemcpy(width_dev, width_1, sizeof(unsigned),
    ↪  cudaMemcpyHostToDevice);
131 //        cudaMemcpy(height_dev, &height, sizeof(unsigned),
    ↪  cudaMemcpyHostToDevice);
132
133         // Record time on GPU with cuda events
134         cudaEvent_t start, stop;
135         cudaEventCreate(&start);
136         cudaEventCreate(&stop);
137
138 ////////////////////////////////////////////////////////////////////////////
139         cudaEventRecord(start);
140         grayscale <<< nBlocks, BLOCKSIZE >>> ( image_in_dev,
            ↪  image_out_dev, width, height );
141         cudaEventRecord(stop);
142 ////////////////////////////////////////////////////////////////////////////
143
144         cudaMemcpy(image_in, image_in_dev, width*height*4*sizeof(char),
            ↪  cudaMemcpyDeviceToHost);
145         cudaMemcpy(image_out, image_out_dev,
            ↪  width*height*4*sizeof(char), cudaMemcpyDeviceToHost);
146
147         cudaEventSynchronize(stop);
148         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,
          ↪  filename);
32         if(error) printf("error %u: %s\n", error,
          ↪  lodepng_error_text(error));
33
34         /*use image here*/
35
36         free(image);
37 }
38
```

```
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 GRAYSCALE
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 width, unsigned height)
75  {
76          printf("test1");
77
78          for(int j=0; j < (width*height*4); j+=4)
79          {
80                  image[j] = (image[j]+image[j+1]+image[j+2])/3;
81                  image[j+1] = (image[j]+image[j+1]+image[j+2])/3;
82                  image[j+2] = (image[j]+image[j+1]+image[j+2])/3;
83
84                  if((image[j+4] - image[j]) > 5 )
85                  {
86                          image[j] = image[j+1] =image[j+2]= 255;
87                  }else{
88                          image[j] = image[j+1] =image[j+2]= 0;
89                  }
90
91          }
92          printf("test2");
93  }
94
95
96
97  ////////////////////////////////////////////////////////////////////////////////
98  // Program main
99  ////////////////////////////////////////////////////////////////////////////////
100 int main()
101 {
102         ////////////////////////////////////////////////////////////////////////////////
103         // Load PNG file
104         ////////////////////////////////////////////////////////////////////////////////
105         float millis = 0;
106         unsigned char *image_in, *image_out, *image_in_dev,
          ↪  *image_out_dev;
107         unsigned width, height, width_dev, height_dev;
108
109         const char* filename = "test.png";
110
111         unsigned error;
112         unsigned char* image = 0;
113
114         error = lodepng_decode32_file(&image, &width, &height,
          ↪  filename);
```

```
115         if(error) printf("error %u: %s\n", error,
            ↪  lodepng_error_text(error));
116
117         // allocate arrays on host
118         image_in = (unsigned char *)malloc(width*height*4 *
            ↪  sizeof(char));
119         image_out = (unsigned char *)malloc(width*height*4 *
            ↪  sizeof(char));
120
121
122         FILE *f = fopen("data.csv", "w");
123
124 //        StopWatchInterface *timer = 0;
125 //        sdkCreateTimer(&timer);
126 //        sdkResetTimer(&timer);
127 //        sdkStartTimer(&timer);
128
129 //        edge_cpu(image, width, height);
130
131 //        sdkStopTimer(&timer);
132 //        printf("Tijd: %f\n", sdkGetTimerValue(&timer));
133 //        sdkDeleteTimer(&timer);
134
135         // Grayscale on CPU
136         for(int j=0; j < (width*height*4); j+=4)
137         {
138                 image[j] = (image[j]+image[j+1]+image[j+2])/3;
139                 image[j+1] = (image[j]+image[j+1]+image[j+2])/3;
140                 image[j+2] = (image[j]+image[j+1]+image[j+2])/3;
141         }
142
143         // Allocate arrays on device
144         cudaMalloc((void **)&image_in_dev, width*height*4 *
            ↪  sizeof(char));
145         cudaMalloc((void **)&image_out_dev, width*height*4 *
            ↪  sizeof(char));
146
147         cudaMemcpy(image_in_dev, image, width*height*4*sizeof(char),
            ↪  cudaMemcpyHostToDevice);
148         cudaMemcpy(image_out_dev, image_out,
            ↪  width*height*4*sizeof(char), cudaMemcpyHostToDevice);
149
150         // Record time on GPU with cuda events
```

```
151         cudaEvent_t start, stop;
152         cudaEventCreate(&start);
153         cudaEventCreate(&stop);
154 
155         ////////////////////////////
156         // Choose Blocksize & nBlock in 2D
157         dim3 BLOCKSIZE(64,16);
158         dim3 nBlocks(ceil(width/64),ceil(height/16));
159         ////////////////////////////
160 
161         cudaEventRecord(start);
162         edge <<< nBlocks, BLOCKSIZE >>> ( image_in_dev, image_out_dev,
            ↪  width, height );
163         cudaEventRecord(stop);
164 
165         cudaMemcpy(image_in, image_in_dev, width*height*4*sizeof(char),
            ↪  cudaMemcpyDeviceToHost);
166         cudaMemcpy(image_out, image_out_dev,
            ↪  width*height*4*sizeof(char), cudaMemcpyDeviceToHost);
167 
168         cudaEventSynchronize(stop);
169         cudaEventElapsedTime(&millis, start, stop);
170 
171         printf("Tijd op GPU: %f\n", millis);
172 
173 //         fprintf(f, "%d,%f\n", BLOCKSIZE, millis);
174 
175         fclose(f);
176 
177         const char* output_filename = "output.png";
178         encodeOneStep(output_filename, image_out, width, height);
179 
180         free(image_in);
181         free(image_out);
182 
183         cudaFree(image_in_dev);
184         cudaFree(image_out_dev);
185 
186         printf("Done!");
187 
188         return 0;
189 }
```

Referenties

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