ADD - INVERT
Labo Geavanceerde Computertechniek

(JLIZNM)

Jona Cappelle & Jonas Bolle

23 maart 2020

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

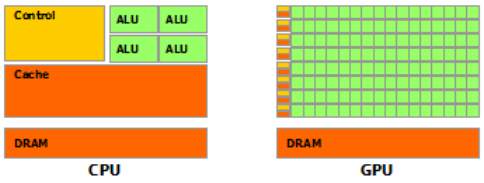
Inhoudsopgave

Lijst van figuren

Lijst van tabellen

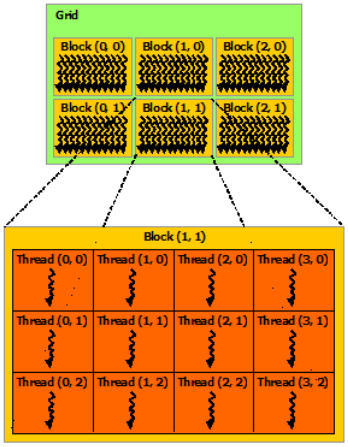
1 Inleiding

In dit eerste labo geavanceerde computerarchitectuur maken we kennis met CUDA, een extensie van de C programmeertaal gemaakt door nVidia. Het grote voordeel hierbij is dat de code op de GPU kan uitgevoerd worden. Hierdoor kan de programmeur gebruik maken van de zeer grote parallele rekenkracht van een nVidia grafische kaart. Cuda is enkel geschikt voor algoritmes die geparalleliseerd kunnen worden.
In dit eerste labo gaan we twee kleine cuda programmaatjes schrijven. Het eerste is om twee arrays op te tellen, het tweede is om een arrray te inverteren. [2]
De architectuur van een CPU en GPU verschilt drastisch (zie figuur 1).



Figuur 1: Vergelijking architectuur CPU vs GPU [1]

Een GPU bestaat uit grids, blocks en threads. Een grid bestaat uit verschillende blocks, die op hun beurt bestaat uit verschillende threads. Nu is het aan de programmeur om deze threads op zo'n efficiënt mogelijke manier te gebruiken. [1]



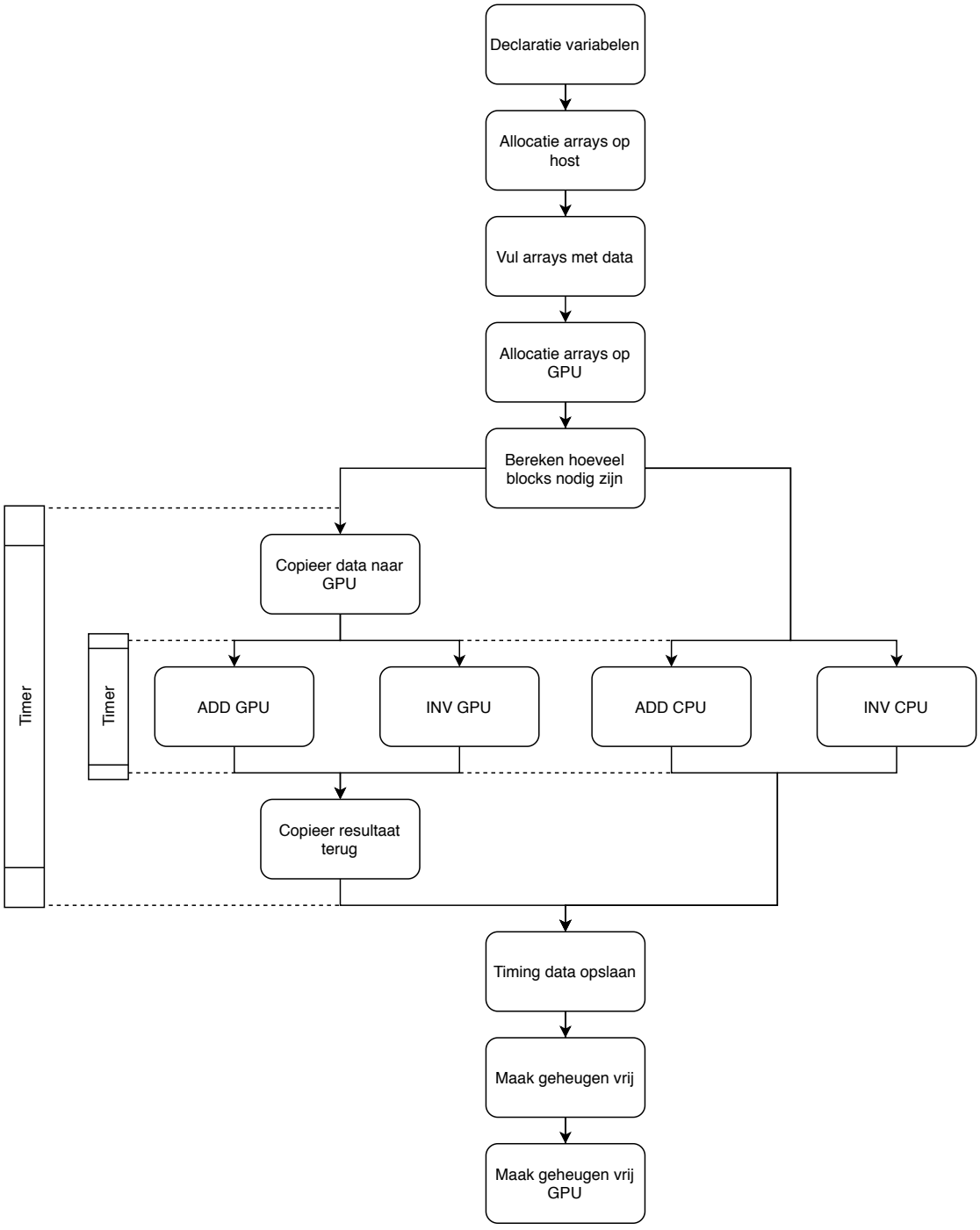
Figuur 2: Grids, blocks en threads [1]

2 Probleemstelling

We zullen in dit labo twee arrays optellen en een array inverteren. We zullen dit een aantal keren doen. De timing om al deze operaties uit te voeren zal opgenomen worden, dit zowel op de GPU als op de CPU. Hieruit zullen we laten conclusies kunnen trekken. We onderzoeken ook de invloed van de blocksize op de GPU. De operaties op de GPU zullen getimed worden met het kopiëren van data naar de GPU en terug en zonder het kopiëren van de data.

3 Oplossing

3.1 Flowchart code



Figuur 3: Flowchart ADD - INV

3.2 Code

De volledige cuda code kan u terug vinden in bijlage B op pagina 8.

4 Analyse performantie

4.1 ADD

4.1.1 Tijd op GPU

In bijlage A op pagina 6 zijn grafieken terug te vinden van uitvoeringstijd in functie van het aantal blocks. Zie hiervoor figuur 4 voor deze met kopiëren van de data naar de GPU en figuur 5 voor de tijd zonder kopiëren van de data. We zien dat de snelste GPU tijden rond de $\pm 380\,\mu s$ liggen met mee copiëren van de data naar de GPU. Zonder het copiëren van de data bekomen we een resultaat van $\pm 25\,\mu s$.

4.1.2 Tijd op CPU

Met een array grootte van 10 000 000 bekomen we op de CPU een tijd van:

$$\text{Tijd op CPU} \Rightarrow 37.12\,\mu s$$

4.2 INV

4.2.1 Tijd op GPU

In bijlage A op pagina 6 zijn grafieken terug te vinden van uitvoeringstijd in functie van het aantal blocks. Zie hiervoor figuur 6 voor deze met kopiëren van de data naar de GPU en figuur 7 voor de tijd zonder kopiëren van de data. We zien dat de snelste GPU tijden rond de $\pm 375\,\mu s$ liggen met het mee copiëren van de data naar de GPU. Zonder het copi#eren van de data bekomen we een sneller resultaat van $\pm 22\,\mu s$.

4.2.2 Tijd op CPU

Met een array grootte van 100 000 000 bekomen we op de CPU een tijd van:

$$\text{Tijd op CPU} \Rightarrow 33.33\,\mu s$$

4.3 Vergelijking

Onderstaande tabel 1 geeft een duidelijk overzicht van de verschillende tijden op de GPU (met en zonder copiëren van de data) en op de CPU.

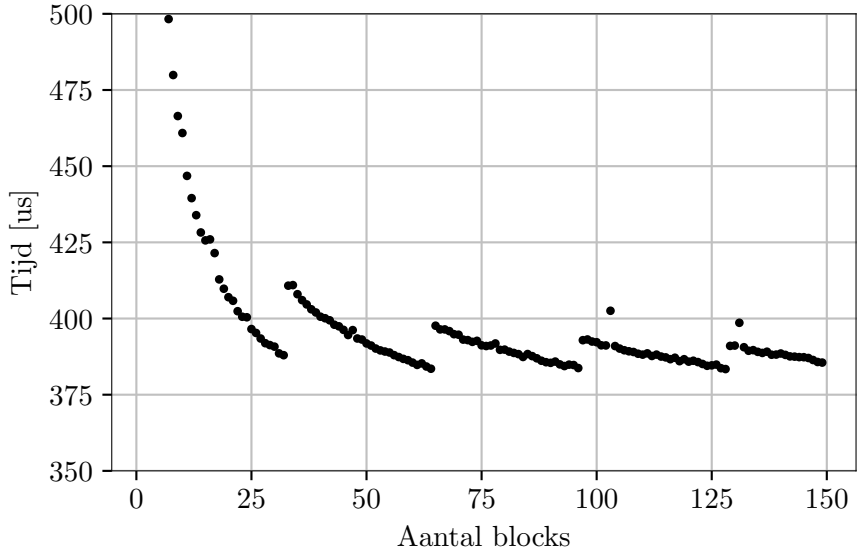
	ADD	INV
Tijd GPU (met copiëren) [μs]	380	375
Tijd GPU (zonder copiëren) [μs]	25	22
Tijd CPU [μs]	37.12	33.33

Tabel 1: Vergelijkende tabel tijden GPU en CPU (met en zonder copiëren van de data)

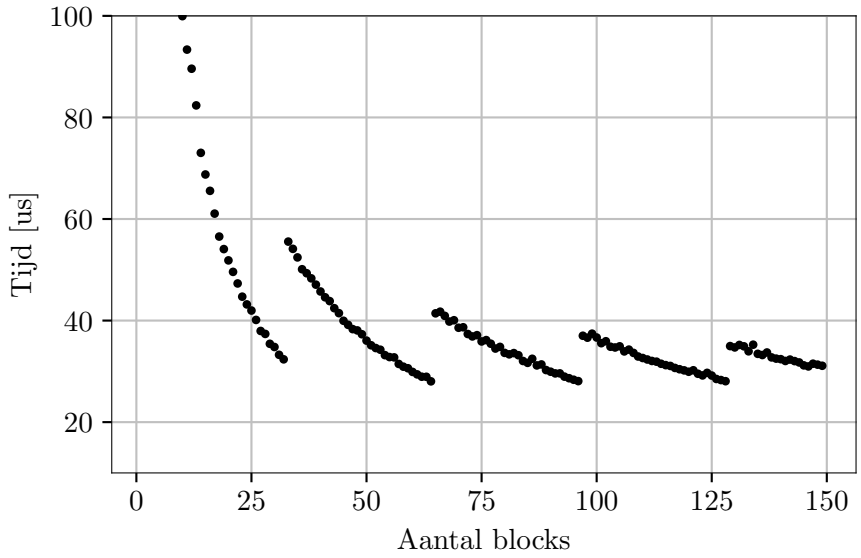
5 Besluit

Uit bovenstaande tabel 1 kunnen we besluiten dat het uitvoeren van een bewerking op de GPU veel sneller kan verlopen dan op een CPU. Dit is net omdat een GPU gebruik kan maken van zeer veel parallele processoren om het rekenwerk te verdelen, terwijl een CPU krachtigere processoren heeft maar het werk meer sequentiëel moet werken.
We zien duidelijk dat het kopiëren van data van en naar de GPU zeer veel tijd in beslag neemt. Dit weegt echter niet op tegen de prestatieverbetering van het uitvoeren van een parallel algoritme op de GPU. Bij grote workloads wordt het snelheidsverlies door het copiëren wordt meer dan goed gemaakt door het sneller berekenen.
Bij veelvouden van 32 blocks werkt de GPU het snelste. Alle threads in één bock steken is zeker niet aan te raden, dit gaf de langste uitvoeringstijd. We zien ook naar mate het aantal blocks stijgt het minder uitmaakt of we met een veelvoud van 32 aan het werken zijn.
We kunnen ook besluiten dat een optelling meer bewerkingstijd in beslag neemt dan het reallocceren van geheugenplaatsen. We zien dit verschil duidelijk tussen INV, die sneller uitvoert dan ADD.
Het besluit is dus, mits het correct alloceren van data op de GPU, het voor grote parallelle workloads veel interessanter is om een GPU te gebruiken dan een CPU.

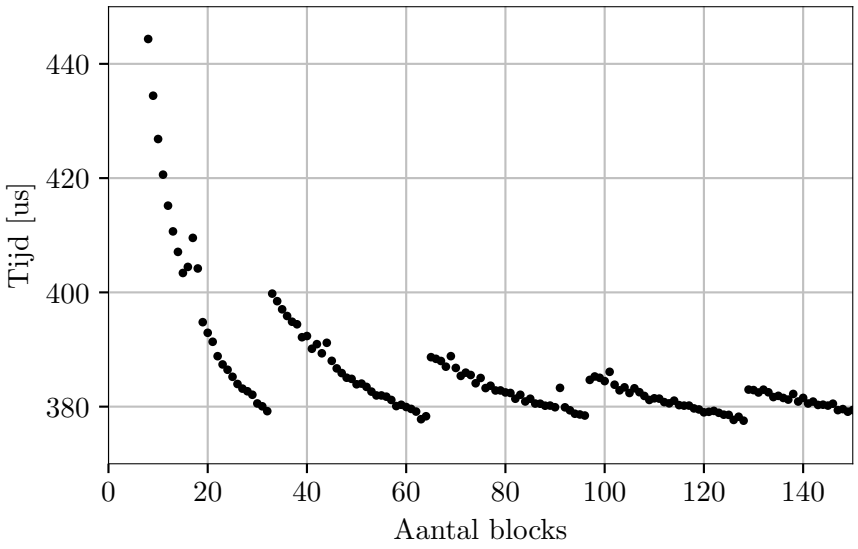
A Grafieken



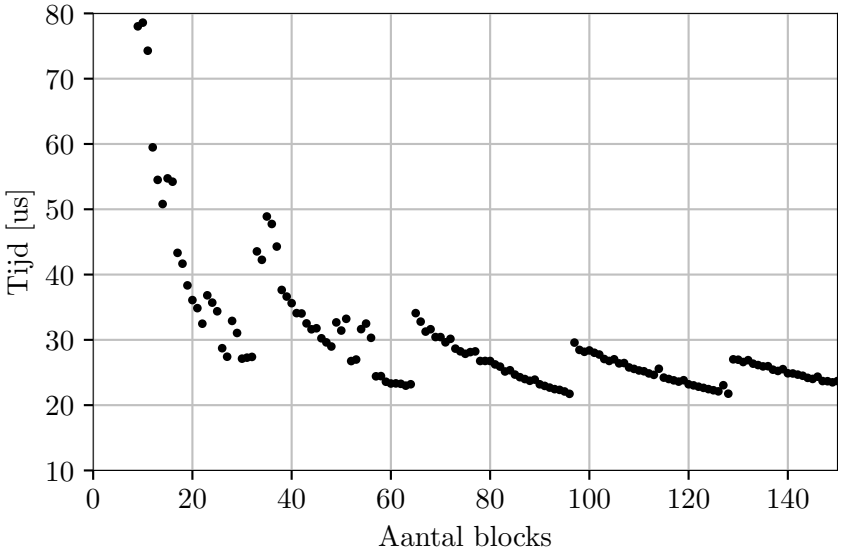
Figuur 4: ADD op GPU met memory copy (10 miljoen elementen)



Figuur 5: ADD op GPU zonder memory copy (10 miljoen elementen)



Figuur 6: INV op GPU met memory copy (100 miljoen elementen)



Figuur 7: INV op GPU zonder memory copy (100 miljoen elementen)

B CODE

```
////////////////////////////////////////////////////////////////////////////////
// ADD / INVERT -- Jona Cappelle -- Jonas Bolle
////////////////////////////////////////////////////////////////////////////////

// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
#include <cuda_runtime.h>

// includes, project
#include <helper_cuda.h>
#include <helper_functions.h> // helper functions for SDK examples

// own includes
#include "iostream"
#include "cstdlib"
#include "time.h"        // timing on cpu

extern "C"
#define ARRAYSIZE 100000000 // Is also the number of threads that will be used

////////////////////////////////////////////////////////////////////////////////
// SELECT GPU - CPU TIMING
// #define GPU
// #define CPU
////////////////////////////////////////////////////////////////////////////////
// RUN ADD - INV
// #define ADD
// #define INV
////////////////////////////////////////////////////////////////////////////////

// Helper function
void init_array(int *a)
{
        for (int i = 0; i < ARRAYSIZE; i++)
        {
                a[i] = i;
        }
}

////////////////////////////////////////////////////////////////////////////////
// KERNEL ADD
////////////////////////////////////////////////////////////////////////////////
```

```
48  int BLOCKSIZE;
49
50  // GPU
51  __global__ void add(int *a, int *b, int *out)
52  {
53          int idx = blockIdx.x * blockDim.x + threadIdx.x;
54          if (idx < ARRAYSIZE)
55          {
56                  out[idx] = a[idx] + b[idx];
57          }
58  }
59
60  // CPU
61  void cpu_add(int *a, int *b, int *out)
62  {
63          for (int i = 0; i < ARRAYSIZE; i++)
64          {
65                  out[i] = a[i] + b[i];
66          }
67  }
68
69  ////////////////////////////////////////////////////////////////////////////////
70  // KERNEL INVERT
71  ////////////////////////////////////////////////////////////////////////////////
72
73  // GPU
74  __global__ void invert(int *a, int *out)
75  {
76          int idx = blockIdx.x * blockDim.x + threadIdx.x;
77          if (idx < ARRAYSIZE)
78          {
79                  out[idx] = a[ARRAYSIZE - 1 - idx];
80          }
81  }
82
83  // CPU
84  void cpu_invert(int *a, int *out)
85  {
86          for (int i = 0; i < ARRAYSIZE; i++)
87          {
88                  out[i] = a[ARRAYSIZE - 1 - i];
89          }
90  }
91
92  ////////////////////////////////////////////////////////////////////////////////
93  // Program main
94  ////////////////////////////////////////////////////////////////////////////////
95  int main()
96  {
```

```
97          // declare variables
98          int *a_host, *b_host, *out_host;
99          int *a_dev, *b_dev, *out_dev;
100
101         // allocate arrays on host
102         a_host = (int *)malloc(ARRAYSIZE * sizeof(int));
103         b_host = (int *)malloc(ARRAYSIZE * sizeof(int));
104         out_host = (int *)malloc(ARRAYSIZE * sizeof(int));
105
106         // initialize arrays with zeros
107         init_array(a_host);
108         init_array(b_host);
109
110         // allocate arrays on device
111         cudaMalloc((void **)&a_dev, ARRAYSIZE * sizeof(int));
112         cudaMalloc((void **)&b_dev, ARRAYSIZE * sizeof(int));
113         cudaMalloc((void **)&out_dev, ARRAYSIZE * sizeof(int));
114
115         // Record time on GPU with cuda events
116 #ifdef GPU
117         cudaEvent_t start, stop;
118         cudaEventCreate(&start);
119         cudaEventCreate(&stop);
120 #endif
121
122         // Timer on CPU
123 #ifdef CPU
124         clock_t start, end;
125         double cpu_time_used;
126 #endif
127
128         // Initialize data file where the timing results will be stored
129         FILE *f = fopen("data.csv", "w");
130
131 #ifdef GPU
132         for (int BLOCKSIZE = 1; BLOCKSIZE < 300; BLOCKSIZE++)
133         {
134 #endif
135
136 #ifdef CPU
137                 float millis = 0;
138 #endif
139
140                 // Calculate amount of blocks needed
141                 int nBlocks = ARRAYSIZE / BLOCKSIZE + (ARRAYSIZE % BLOCKSIZE ==
                    ↪  0 ? 0 : 1);
142                 printf("Nblocks: %i", nBlocks);
143
144                 // Start timer
```

```
145 #ifdef CPU
146                 StopWatchInterface *timer = 0;
147                 sdkCreateTimer(&timer);
148                 sdkStartTimer(&timer);
149 #endif
150
151 #ifdef GPU // Start cuda event on GPU
152                 cudaEventRecord(start);
153 #endif
154
155                 //Step 1: Copy data to GPU memory
156                 cudaMemcpy(a_dev, a_host, ARRAYSIZE * sizeof(int),
                    ↪  cudaMemcpyHostToDevice);
157                 cudaMemcpy(b_dev, b_host, ARRAYSIZE * sizeof(int),
                    ↪  cudaMemcpyHostToDevice);
158                 cudaMemcpy(out_dev, out_host, ARRAYSIZE * sizeof(int),
                    ↪  cudaMemcpyHostToDevice);
159
160                 ////////////////////////////////////////////////////////////////////////////
161                 // GPU -- define ADD - INV to run
162                 ////////////////////////////////////////////////////////////////////////////
163 #ifdef ADD && GPU
164                 add<<<nBlocks, BLOCKSIZE>>>(a_dev, b_dev, out_dev);
165 #endif
166 #ifdef INV && GPU
167                 invert <<< nBlocks, BLOCKSIZE >>> ( a_dev, out_dev );
168 #endif
169
170 #ifdef GPU // Stop GPU event timer
171                 cudaEventRecord(stop);
172 #endif
173
174                 ////////////////////////////////////////////////////////////////////////////
175                 // CPU -- define ADD - INV to run
176                 ////////////////////////////////////////////////////////////////////////////
177 #ifdef CPU // CPU timer
178                 start = clock();
179 #endif
180
181 #ifdef ADD && CPU
182                 cpu_add( a_host, b_host, out_host);
183 #endif
184 #ifdef INV && CPU
185                 cpu_invert ( a_host, out_host );
186 #endif
187
188 #ifdef CPU
189                 end = clock();
190                 cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

```
191                 printf("%f", cpu_time_used);
192 #endif
193
194                 //Step 4: Retrieve result
195                 cudaMemcpy(a_host, a_dev, ARRAYSIZE * sizeof(int),
                    ↪  cudaMemcpyDeviceToHost);
196                 cudaMemcpy(b_host, b_dev, ARRAYSIZE * sizeof(int),
                    ↪  cudaMemcpyDeviceToHost);
197                 cudaMemcpy(out_host, out_dev, ARRAYSIZE * sizeof(int),
                    ↪  cudaMemcpyDeviceToHost);
198
199 #ifdef GPU
200                 cudaEventSynchronize(stop);
201                 cudaEventElapsedTime(&millis, start, stop);
202 #endif
203
204 #ifdef CPU        // Stop timer
205                 sdkStopTimer(&timer);
206 #endif
207
208                 // Print time to console
209 #ifdef CPU
210                 printf("Processing time: %f (ms)\n", sdkGetTimerValue(&timer));
211 #endif
212 #ifdef GPU
213                 printf("Processing time: %f (ms)\n", millis);
214 #endif
215
216                 // Write timing results to file
217 #ifdef CPU
218                 fprintf(f, "%f\n", sdkGetTimerValue(&timer));
219 #endif
220 #ifdef GPU
221                 fprintf(f, "%d,%f\n", BLOCKSIZE, millis);
222 #endif
223
224 #ifdef CPU        // Verwijder timer
225                 sdkDeleteTimer(&timer);
226 #endif
227
228 #ifdef GPU
229         } //End for
230 #endif
231
232         // Close the file
233         fclose(f);
234
235         // Free up the used memory
236         free(a_host);
```

```
237         free(b_host);
238         free(out_host);
239 
240 #ifdef GPU // Free up the cuda memory
241         cudaFree(a_dev);
242         cudaFree(b_dev);
243         cudaFree(out_dev);
244 #endif
245 
246         return 0;
247 }
```

Referenties

[1] "Programming Guide :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] "What is CUDA? An Introduction — The Supercomputing Blog." [Online]. Available: http://supercomputingblog.com/cuda/what-is-cuda-an-introduction/