Auteur : Jalu Glissenaar, Thomas Smeele
Datum : 1 April 2019
Project : Vision
Team : Kaasbroodje

Meetrapport Snelheid

1. Doel

In ons experiment willen we kijken hoe snel onze implementatie van de randdetectie eigenlijk is in vergelijking met de originele implementatie. De meetresultaten worden bepaald door de tijdmeting vanaf de aanroep van de randdetectie functie tot het einde van de randdetectie (inclusief thresholding). Onze hoofdvraag is:

"Hoe snel is de door ons geïmplementeerde randdetectie in vergelijking met het standaard geïmplementeerde randdetectie deel (beide inclusief de thresholding)."

Wij hopen daar door middel van deze meetrapport, deze onderzoeksvraag te kunnen beantwoorden.

2. Hypothese

Wij verwachten dat de door ons geïmplementeerde vorm van randdetectie minder snel is dan het standaard geïmplementeerde deel omdat de manier van implementatie van de door ons gemaakte randdetectie op de meest eenvoudige manier geïmplementeerd is. De standaard implementatie maakt gebruik van de library OpenCV waar mensen lang mee bezig zijn geweest om de implementatie van verschillende soorten functies zo snel mogelijk te maken.

3. Werkwijze

We gaan voor elke stap in de randdetectie kijken hoe lang die duurt (in nano seconden). Ook bepalen we de totale duur van de randdetectie van alle stappen bij elkaar. Dit doen we door middel van 100x de test te herhalen op zowel de zelf geïmplementeerde code en de standaard code.

Deze test wordt uitgevoerd op een Windows machine.

Stappenplan

Bij elke uitvoerende functie of variabel declaratie meten we de begin- en eindtijd in nano seconden zoals hier onder te zien.

```
auto begin = std::chrono::high_resolution_clock::now(); // Start the timer
```

... // De op snelheid te testen code

```
auto end_gaussian_kernel_generator = std::chrono::high_resolution_clock::now(); // End the timer
auto secs = std::chrono::duration_cast<std::chrono::nanoseconds>(_Dur:end_gaussian_kernel_generator - begin).count(); // Calculate secs out of clock ticks
std::cout << "~wat je net hebt getest: ~" << secs << '\n'; // Print secs to cout
```

Omdat we elke functie en variabele een *cout* doen (die heel traag zijn) kunnen we niet het begin en einde meten van de hele randdetectie en thresholding. We moeten daarom de waardes van de gemeten tijden bij elkaar optellen. Dit doen we door middel van een extra variabel aan te maken voor elke meetwaarde en onder de *std::cout* lijn de waarde er bij optellen:

```
long long int time_1 = 0;
long long int time_2 = 0;
long long int time_3 = 0;
long long int time_4 = 0;
long long int time_5 = 0;
```

... // code

```
auto end_gaussian_kernel_generator = std::chrono::high_resolution_clock::now(); // End the timer
auto secs = std::chrono::duration_cast<std::chrono::nanoseconds>(_Dur:end_gaussian_kernel_generator - begin).count();
std::cout << "Gaussian blur lambda creation -> elapsed nano seconds = " << secs << '\n'; // Print secs to cout
time_1 += secs;
```

Als laatste regel in de cout willen we de total nano seconden printen. Dit doen we door middel van de volgende code:

```
std::cout << "Total -> elapsed nano seconds = " << (time_1 + time_2 + time_3 + time_4 + time_5 + time_6 + time_7 + time_8) << "\n\n";
```

Als we dit bij elke functie & variabel uitvoeren krijgen we voor het zelf geïmplementeerde deel de volgende output in de console:

```
Gaussian blur lambda creation -> elapsed nano seconds = 800
Gaussian blur kernel generator -> elapsed nano seconds = 11700
Edge kernel generator -> elapsed nano seconds = 8000
IntensityImage to matrix conversion -> elapsed nano seconds = 3154500
Gaussian blur convolution -> elapsed nano seconds = 107599300
Edge kernel convolution -> elapsed nano seconds = 328875800
Thresholding -> elapsed nano seconds = 3095400
Matrix to IntensityImage ptr converting -> elapsed nano seconds = 3965000
Total -> elapsed nano seconds = 44913543092
```

En omdat het standaard geïmplementeerde deel opgesplitst is in het randdetectie deel en het thresholding deel moeten we voor 2 functies de test schrijven op dezelfde manier. De resultaten van het randdetectie deel van de standaard implementatie ziet er in de console ongeveer zo uit:

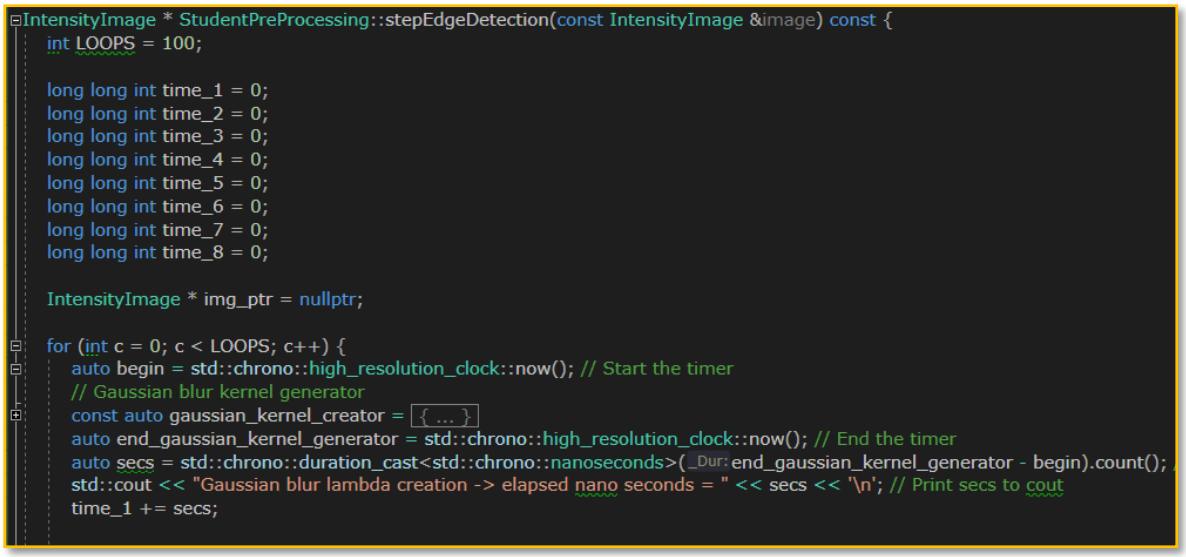
```
Creating empty math -> elapsed nano seconds = 1100
IntensityImage to mat conversion -> elapsed nano seconds = 4524900
Creating edge kernel -> elapsed nano seconds = 16900
Creating empty math -> elapsed nano seconds = 1300
Edge kernel convolution -> elapsed nano seconds = 527300
Math to IntensityImage prt converting -> elapsed nano seconds = 3928700
Total -> elapsed nano seconds = 1063621900
```

En het thresholding deel ziet er ongeveer zo uit:

```
Creating empty math -> elapsed nano seconds = 1100
IntensityImage to mat conversion -> elapsed nano seconds = 4288300
Thresholding -> elapsed nano seconds = 9900
Math to IntensityImage prt converting -> elapsed nano seconds = 4302600
```

Maar we willen niet 1 meting van de eigen implementatie & de standaard implementatie maar we willen bijvoorbeeld 100 verschillende metingen doen van elk deel en daar het gemiddelde van berekenen.

Dit doen we door in de functies een for loop te maken waar bijna alle code in staat:



```
IntensityImage * StudentPreProcessing::stepEdgeDetection(const IntensityImage &image) const {
    int LOOPS = 100;

    long long int time_1 = 0;
    long long int time_2 = 0;
    long long int time_3 = 0;
    long long int time_4 = 0;
    long long int time_5 = 0;
    long long int time_6 = 0;
    long long int time_7 = 0;
    long long int time_8 = 0;

    IntensityImage * img_ptr = nullptr;

    for (int c = 0; c < LOOPS; c++) {
        auto begin = std::chrono::high_resolution_clock::now(); // Start the timer
        // Gaussian blur kernel generator
        const auto gaussian_kernel_creator = { ... }
        auto end_gaussian_kernel_generator = std::chrono::high_resolution_clock::now(); // End the timer
        auto secs = std::chrono::duration_cast<std::chrono::nanoseconds>(_Dur: end_gaussian_kernel_generator - begin).count();
        std::cout << "Gaussian blur lambda creation -> elapsed nano seconds = " << secs << '\n'; // Print secs to cout
        time_1 += secs;
```

... // De rest van de test code

En in de standaard code doen we precies hetzelfde alleen dan voor de edge detection functie en de thresholding functie:

Edge detection:

```
IntensityImage * DefaultPreProcessing::stepEdgeDetection(const IntensityImage &src) const
    int LOOPS = 100;

    long long int time_1 = 0;
    long long int time_2 = 0;
    long long int time_3 = 0;
    long long int time_4 = 0;
    long long int time_5 = 0;
    long long int time_6 = 0;

    IntensityImage * ThoroughFloodThoroughFire = nullptr;

    for (int c = 0; c < LOOPS; c++) {
        auto begin_create_empty_math_1 = std::chrono::high_resolution_clock::now(); // Sta
        cv::Mat OverHillOverDale;
        auto end_create_empty_math_1 = std::chrono::high_resolution_clock::now(); // End t
        auto secs_create_empty_math_1 = std::chrono::duration_cast<std::chrono::nanoseco
        std::cout << "Creating empty math -> elapsed nano seconds = " << secs_create_empt
        time_1 += secs_create_empty_math_1;

        auto begin_IntensityImage_to_mat_conversion = std::chrono::high_resolution_clock::r
```

Thresholding:

```
IntensityImage * DefaultPreProcessing::stepThresholding(const IntensityImage &src) const {
    int LOOPS = 100;

    long long int time_1 = -1;
    long long int time_2 = -1;
    long long int time_3 = -1;
    long long int time_4 = -1;

    IntensityImage * ThoroughBushThoroughBrier = nullptr;

    for (int c = 0; c < LOOPS; c++) {
        auto begin_create_empty_math_1 = std::chrono::high_resolution_clock::now(); // Start the
        cv::Mat OverHillOverDale;
        auto end_create_empty_math_1 = std::chrono::high_resolution_clock::now(); // End the tim
        auto secs_create_empty_math_1 = std::chrono::duration_cast<std::chrono::nanoseconds>(
        std::cout << "Creating empty math -> elapsed nano seconds = " << secs_create_empty_ma
        time_1 += secs_create_empty_math_1;

        auto begin_IntensityImage_to_mat_conversion = std::chrono::high_resolution_clock::now();
```

We willen natuurlijk ook dat op het einde na x verschillende testen we het gemiddelde van de x aantal loop resultaten laten zien in de command line. Dit is nu heel makkelijk te implementeren omdat we variabelen die we langer nodig hebben buiten de forloop hebben gedeclareerd en we ook speciaal de IntensityImage buiten de loop hebben aangemaakt zodat we die pas teruggeven na x aantal loops waar in wij alle berekeningen al gedaan hebben. De code die we na de forloop moeten schrijven ziet er ongeveer als volgt uit:

```
    auto begin_matrix_to_IntensityImage_ptr_converter = std::chrono::high_resolution_clock::now(); // Start the timer
    // convert the matrix to a IntensityImage type which is required.
    img_ptr = img.get_intensity_image_ptr();
    auto end_matrix_to_IntensityImage_ptr_converter = std::chrono::high_resolution_clock::now(); // End the timer
    auto secs_matrix_to_IntensityImage_ptr_converter = std::chrono::duration_cast<std::chrono::nanoseconds>(_Dur: end_matrix_to_IntensityImage_ptr_converter - begin_m
    std::cout << "Matrix to IntensityImage ptr converting -> elapsed nano seconds = " << secs_matrix_to_IntensityImage_ptr_converter << '\n'; // Print secs to cout
    time_8 += secs_matrix_to_IntensityImage_ptr_converter;

    std::cout << "Total -> elapsed nano seconds = " << (time_1 + time_2 + time_3 + time_4 + time_5 + time_6 + time_7 + time_8) << "\n\n"; // Total elapsed time
}; // End for loop

std::cout << "################################## TOTAL ##################################\n\n";

std::cout << "Gaussian blur lambda creation (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_1/LOOPS << '\n';
std::cout << "Gaussian blur kernel generator (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_2/LOOPS << '\n';
std::cout << "Edge kernel generator (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_3/LOOPS << '\n';
std::cout << "IntensityImage to matrix conversion (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_4/LOOPS << '\n';
std::cout << "Gaussian blur convolution (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_5/LOOPS << '\n';
std::cout << "Edge kernel convolution (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_6/LOOPS << '\n';
std::cout << "Thresholding (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_7/LOOPS << '\n';
std::cout << "Matrix to IntensityImage ptr converting (average out of " << LOOPS << " ) -> elapsed nano seconds = " << time_8 / LOOPS << '\n';
std::cout << "Total (average out of " << LOOPS << " ) -> elapsed nano seconds = " << ((time_1 / LOOPS) + (time_2 / LOOPS) + (time_3 / LOOPS) + (time_4 / LOOPS) + (tir
```

Met als laatste regel de optelling van alle verschillende gemiddelde metingen.

Om de test zo eerlijk mogelijk te houden tussen de eigen implementatie en de standaard implementatie moeten we ook de lege thresholding functie opmeten in de eigen implementatie. Deze meting heeft maar 1 resultaat en is niet zo bijzonder omdat de functie alleen een nieuwe IntensityImage aanmaakt en de waardes over gekopieerd en dan dezelfde waardes teruggeeft.

Dit is dan ook de volgende code:

```
IntensityImage * StudentPreProcessing::stepThresholding(const IntensityImage &image) const {
    // The next lines of code are required to create a copy of the given IntensityImage parameter and return it.
    // We couldn't just make a pointer of the IntensityImage reference becouse it's a const parameter which is declared in the abstract
    // class so we couldn't change const to non-const.
    int LOOPS = 100;

    long long int total = 0;

    IntensityImage * img_ptr = nullptr;
    for(int c = 0; c < LOOPS; c++){
        auto begin = std::chrono::high_resolution_clock::now();  // Start the timer
        img_ptr = ImageFactory::newIntensityImage();
        img_ptr->set(image.getWidth(), image.getHeight());
        for (int y = 0; y < image.getHeight(); y++) {
            for (int x = 0; x < image.getWidth(); x++) {
                img_ptr->setPixel(x, y, image.getPixel(x, y));
            }
        }
        auto end = std::chrono::high_resolution_clock::now(); // End the timer
        total += std::chrono::duration_cast<std::chrono::nanoseconds>(_Dur: end - begin).count(); // Calculate secs out of clock ticks
    }

    std::cout << "################################## TOTAL ##################################\n\n";

    std::cout << "Total (average out of " << LOOPS << " ) -> elapsed nano seconds = " << total / LOOPS << '\n';
    return img_ptr;
}
```

De resultaten in de prompt zullen er als volgt uitzien:

Eigen implementatie edge detectie & thresholding:

```
################################## TOTAL ####################################

Gaussian blur lambda creation (average out of 100 ) -> elapsed nano seconds = 783
Gaussian blur kernel generator (average out of 100 ) -> elapsed nano seconds = 11925
Edge kernel generator (average out of 100 ) -> elapsed nano seconds = 8710
IntensityImage to matrix conversion (average out of 100 ) -> elapsed nano seconds = 2994509
Gaussian blur convolution (average out of 100 ) -> elapsed nano seconds = 105783186
Edge kernel convolution (average out of 100 ) -> elapsed nano seconds = 330979441
Thresholding (average out of 100 ) -> elapsed nano seconds = 3065061
Matrix to IntensityImage ptr converting (average out of 100 ) -> elapsed nano seconds = 4068755
Total (average out of 100 ) -> elapsed nano seconds = 446912370
```

Eigen implementatie thresholding functie (doet niks)

```
################################## TOTAL ####################################

Total (average out of 100 ) -> elapsed nano seconds = 6427168
```

Standaard edge detection implementatie:

```
############################## TOTAL EDGE DETECTION ##############################

Creating empty math (average out of 100 ) -> elapsed nano seconds = 1571
IntensityImage to mat conversion (average out of 100 ) -> elapsed nano seconds = 4960270
Creating edge kernel (average out of 100 ) -> elapsed nano seconds = 21700
Creating empty math (average out of 100 ) -> elapsed nano seconds = 1402
Edge kernel convolution (average out of 100 ) -> elapsed nano seconds = 671029
Math to IntensityImage prt converting (average out of 100 ) -> elapsed nano seconds = 4846498
Total (average out of 100 ) -> elapsed nano seconds = 10502470
```

Standaard thresholding implementatie:

```
############################## TOTAL THRESHOLDING ##############################

Creating empty math (average out of 100 ) -> elapsed nano seconds = 1298
IntensityImage to mat conversion (average out of 100 ) -> elapsed nano seconds = 4238660
Thresholding (average out of 100 ) -> elapsed nano seconds = 4238660
mat to IntensityImage conversion (average out of 100 ) -> elapsed nano seconds = 4246412
Total (average out of 100 ) -> elapsed nano seconds = 12725030
```

4. Resultaten

Eigen implementatie:

```
################################## TOTAL ###################################

Gaussian blur lambda creation (average out of 100 ) -> elapsed nano seconds = 783
Gaussian blur kernel generator (average out of 100 ) -> elapsed nano seconds = 11925
Edge kernel generator (average out of 100 ) -> elapsed nano seconds = 8710
IntensityImage to matrix conversion (average out of 100 ) -> elapsed nano seconds = 2994509
Gaussian blur convolution (average out of 100 ) -> elapsed nano seconds = 105783186
Edge kernel convolution (average out of 100 ) -> elapsed nano seconds = 330979441
Thresholding (average out of 100 ) -> elapsed nano seconds = 3065061
Matrix to IntensityImage ptr converting (average out of 100 ) -> elapsed nano seconds = 4068755
Total (average out of 100 ) -> elapsed nano seconds = 446912370
```

Eigen implementatie thresholding (doet niks behalve kopiëren):

```
################################## TOTAL ###################################

Total (average out of 100 ) -> elapsed nano seconds = 6427168
```

Standaard edge detection implementatie:

```
################################## TOTAL EDGE DETECTION ###################################

Creating empty math (average out of 100 ) -> elapsed nano seconds = 1571
IntensityImage to mat conversion (average out of 100 ) -> elapsed nano seconds = 4960270
Creating edge kernel (average out of 100 ) -> elapsed nano seconds = 21700
Creating empty math (average out of 100 ) -> elapsed nano seconds = 1402
Edge kernel convolution (average out of 100 ) -> elapsed nano seconds = 671029
Math to IntensityImage prt converting (average out of 100 ) -> elapsed nano seconds = 4846498
Total (average out of 100 ) -> elapsed nano seconds = 10502470
```

Standaard thresholding implementatie:

```
################################## TOTAL THRESHOLDING ###################################

Creating empty math (average out of 100 ) -> elapsed nano seconds = 1298
IntensityImage to mat conversion (average out of 100 ) -> elapsed nano seconds = 4238660
Thresholding (average out of 100 ) -> elapsed nano seconds = 4238660
mat to IntensityImage conversion (average out of 100 ) -> elapsed nano seconds = 4246412
Total (average out of 100 ) -> elapsed nano seconds = 12725030
```

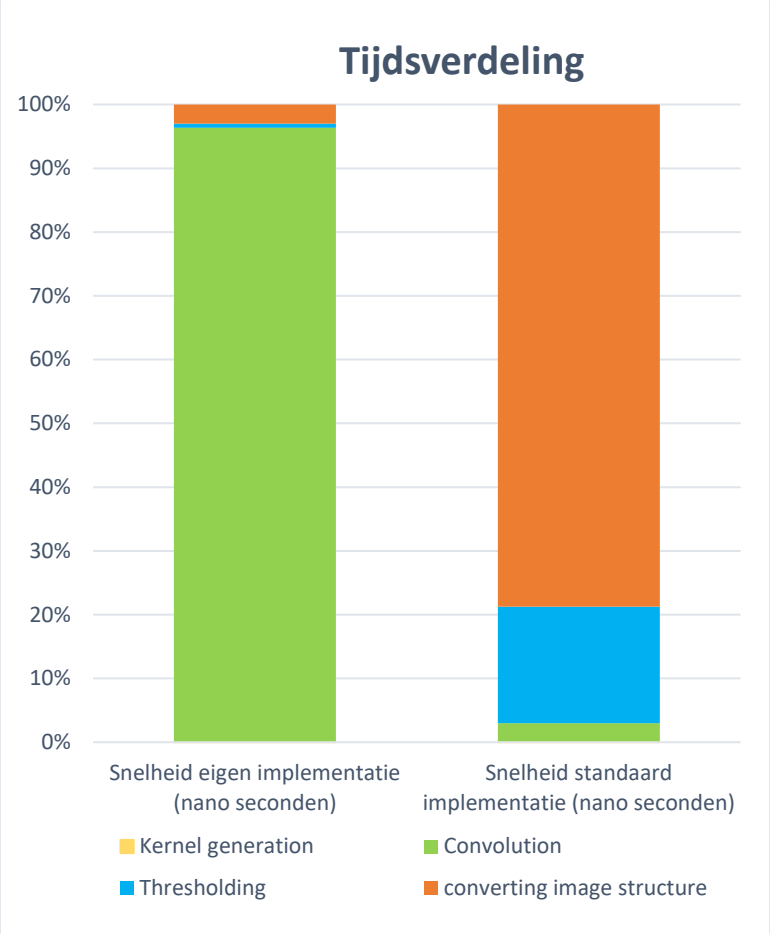
Als we de resultaten verwerken in een tabel krijgen we de volgende tabel:

	Snelheid eigen implementatie (nano seconden)	Snelheid standaard implementatie (nano seconden)
Gaussian blur lambda	783	
Gaussian blur kernel generation	11925	
Edge kernel generation	8710	21700
Empty matrix creation 1		1571
IntensityImage to matrix 1	2994509	4960270
gaussian kernel convolution	105783186	
edge kernel convolution	330979441	671029
Empty matrix creation 2		1298
IntensityImage to matrix 2		4238660
Thresholding	3065061	4238660
Matrix to IntensityImage 1	4068755	4846498
Matrix to IntensityImage 2	6427186	4246412
Totaal:	**453339556**	**23226098**

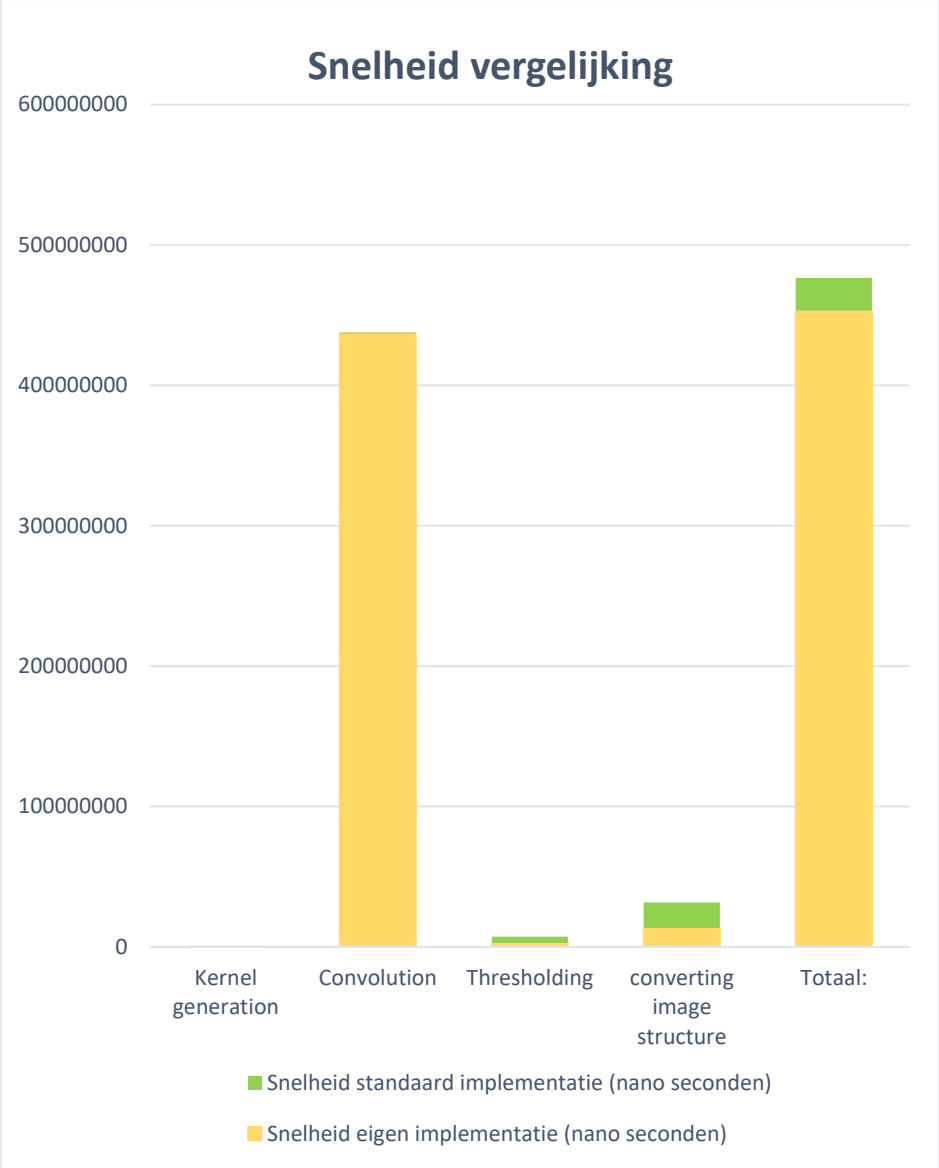
Als we de gegevens van de tabel onderverdelen in verschillende categorieën en die in een tabel verwerken krijgen we de volgende tabel:

	Snelheid eigen implementatie (nano seconden)	Snelheid standaard implementatie (nano seconden)
Kernel generation	21418	21700
Convolution	436762627	671029
Thresholding	3065061	4238660
converting image structure	13490450	18294709
Totaal:	**453339556**	**23226098**

Deze tabel kunnen we weer makkelijk in een tijdsverdeling grafiek zetten om te kijken hoe beide code het doet qua tijdsverdeling van verschillende processen:



Maar om makkelijk te kunnen vergelijken hoe snel beide implementaties zijn kunnen we de waardes van de tabel ook in een tijds vergelijking grafiek zetten:



5. Verwerking

Alle resultaten die verwerkt zijn een gemiddelde van 100 metingen. Dit kan natuurlijk altijd accurater door dit getal op te hogen maar ondanks het vaker te testen blijven de waardes ongeveer hetzelfde.

Door de resultaten te vergelijken met onze hypothese kunnen we stellen dat we deels gelijk hadden. Er zijn verschillende resultaten die ons verbazen, het aanmaken van verschillende kernels is meer dan 2x zo snel & de thresholding is ook een stuk sneller dan de standaard implementatie. Maar als we gaan kijken naar de elementen die de meeste invloed hebben op de tijdsmeting, de convolution, kunnen we wel stellen dat de standaard implementatie een stuk sneller is dan de door ons geïmplementeerde versie van de randdetectie (bijna 500x sneller).

6. Conclusie

Door de resultaten te bekijken kunnen we zien dat de eigen implementatie sneller is in de volgende onderdelen: Kernel generatie (± 2.5 x sneller); Omzetten van een IntensityImage naar een matrix (±1.7 x sneller); Omzetten van een matrix naar IntensityImage (± 1.2 x sneller); Thresholding (±1.4 x sneller).

Maar de vlakken waar de eigen implementatie sneller uit de testen komt wegen niet op tegen de vlakken waar de standaard implementatie sneller in is, namelijk de convolution. De standaard convolution functie is ± 500 x sneller dan het door ons geïmplementeerde deel. Ook gebruikt de standaard implementatie maar één keer de convolution functie voor de edge kernel en in onze implementatie gebruiken we de convolution functie 2x, namelijk voor de gaussian blur kernel en de edge kernel.

Het totale tijdsverschil is door het tijdsverschil van de convolution functies dan ook flink. De standaard implementatie doet er ongeveer 23.2miliseconden over (23.226.098 ns / 1000.000 ≈ 23.2ms). Terwijl de eigen implementatie er ongeveer 453.3 milliseconden over doet (453.339.556 ns / 1000.000 ≈ 453.3ms). Dit betekent dat de standaard implementatie ongeveer 20x zo snel is als het eigen geïmplementeerde deel.

Het antwoord op de hoofdvraag “Hoe snel is de door ons geïmplementeerde randdetectie in vergelijking met het standaard geïmplementeerde randdetectie deel (beide inclusief de thresholding).” Luidt dan ook: ongeveer 20x langzamer.

7. Evaluatie

Bij het maken van het implementatieplan hadden we een duidelijk doel voor ogen. Dit doel was om een werkende vorm van edge detectie te implementeren door middel van een eigen implementatie. Dit is uiteindelijk zeker gelukt en sommige onderdelen van onze implementatie zijn zo te zien ook sneller. Jammer genoeg is de convolution functie zeer traag in vergelijking met de standaard implementatie. Dit had dan ook verbetert kunnen worden door andere technieken toe te passen voor de convolution. We weten ook hoe we andere manieren zouden moeten implementeren die sneller zijn alleen zouden we die niet helemaal zelf hebben geïmplementeerd maar voor een groot deel van internet af hebben gehaald.