AKADEMIA GÓRNICZO-HUTNICZA

WEAIiIB, Katedra Automatyki i Robotyki

Laboratorium Biocybernetyki

Przedmiot:	**Akceleracja Algorytmów Wizyjnych w GPU i OpenCL**			
			PR25Laaw07_CIRCLE	
Temat projektu:				
	Detekcja okręgów za pomocą transformaty Hough'a			
Wykonali:	Mikołaj	Pietrzyk	mpietrzyk@student.agh.edu.pl	
	Bartek	Stec	stecb@student.agh.edu.pl	
	Mateusz	Zajda	mzajda@student.agh.edu.pl	
	Automatyka i Robotyka		studia:	magisterskie
Rok akademicki:	2024/2025	semestr letni		
Prowadzący:	dr inż.	Mirosław Jabłoński		

wersja 3.4

Kraków, maj, 2025.

Spis treści

1. Streszczenie

Celem niniejszej pracy jest opracowanie metody detekcji współśrodkowych okręgów w obrazach cyfrowych, przy użyciu transformaty Hougha. Zaproponowane podejście odpowiada na wyzwania związane z obecnością szumów, deformacjami oraz nieciągłościami krawędzi w obrazach, co często utrudnia precyzyjne rozpoznanie okręgów przy użyciu klasycznych metod.

W opracowanej metodzie najpierw przeprowadza się wstępne przetwarzanie obrazu: wygładzanie i detekcję krawędzi. Następnie środek okręgów wykrywany jest za pomocą standardowej transformaty Hougha, która opiera się na zliczaniu punktów będących kandydatami na krawędź okręgu w akumulatorze dla każdego piksela.

W ramach realizowanego projektu osiągnięto wykrywanie krawędzi okręgów o promieniach mieszczących się w zadanym zakresie, nawet na obrazach o bardzo wysokiej rozdzielczości.

Ze względu na rosnące wymagania obliczeniowe przy analizie obrazów wysokiej rozdzielczości, rozwiązanie zostało zoptymalizowane pod kątem wykorzystania procesora graficznego, co umożliwia znaczące skrócenie czasu przetwarzania w porównaniu do tradycyjnych implementacji opartych wyłącznie na jednostce centralnej. Wykorzystano przy tym framework OpenCL, który umożliwia tworzenie międzyplatformowych aplikacji do obliczeń równoległych, co stanowi kluczowy element przyspieszający analizę danych w zastosowaniach obrazowych.

W wyniku podjętych działań powstała aplikacja napisana w języku C++, która poprzez dedykowany kernel algorytmu Hougha realizuje wykrywanie kół na przekazanym obrazie. Użytkownik, wybierając odpowiednią platformę obliczeniową, może uzyskać wynikowy obraz zawierający zaznaczone wykryte okręgi.

Głównym wnioskiem płynącym z realizacji projektu jest potwierdzenie, że framework OpenCL umożliwia efektywne zrównoleglenie obliczeń. Dodatkowo, przeprowadzone eksperymenty wykazały, iż metoda transformaty Hougha posiada pewne ograniczenia. W szczególności jest to podatność na deformacje krawędzi.

Projekt stanowi istotny dowód na potęgę obliczeniową GPU w połączeniu z technologiami równoległych obliczeń, takimi jak OpenCL, oraz podkreśla znaczenie zastosowania nowoczesnych, zrównoleglonych algorytmów w przetwarzaniu obrazów. Taka optymalizacja pozwala na realizację złożonych operacji analitycznych w skróconym czasie, co może mieć fundamentalne znaczenie w wielu dziedzinach zastosowań informatyki i analizy obrazów.

2. Wstęp

2.1 Cele i założenia projektu

Celem niniejszego projektu jest opracowanie i implementacja algorytmu detekcji współśrodkowych okręgów w obrazach cyfrowych z wykorzystaniem transformaty Hough'a. Metoda ta polega na analizie otoczenia każdego piksela i określeniu, ile pikseli z otoczenia kwalifikuje się do wyznaczanego w danym miejscu okręgu o zadanym promieniu. Ważnym elementem algorytmu jest początkowe wygładzanie obrazu oraz detekcja krawędzi, poprzez którą można wykrywać położenie środków okręgu. Rozwiązanie ma zapewniać wysoką odporność na zakłócenia, szumy, nieciągłości krawędzi oraz częściowe deformacje, przy jednoczesnym obniżeniu złożoności obliczeniowej względem pozostałych podejść do problemu detekcji figury przypominającej okrąg. Optymalizacja ma zostać osiągnięta za pomocą języka programowania C++ oraz frameworku OpenCL oraz akceleracji obliczeń na GPU. Czas działania algorytmu powinien być relatywnie krótki nawet dla obrazów z wysoką rozdzielczością.

Wykrywanie współśrodkowych okręgów jest stosowane w różnych dziedzinach począwszy od produkcji przemysłowej i rozpoznawaniu krągłych otworów fotoelektrycznych, wykrywania tęczówki oka w rozpoznawaniu twarzy, przemysłu spożywczego i wykrywania hamujących aureoli antybakteryjnych czy kalibracji systemów optycznych. [4]

Klasyczną metodą detekcji okręgów jest Standardowa Transformacja Hough'a (SHT), która wykorzystuje trójwymiarową przestrzeń parametryczną, środek okręgu (a,b) i promień r. Każdy punkt krawędziowy w obrazie przekształcany jest na stożek w przestrzeni, a przecięcia tych stożków wskazują potencjalny punkt, który jest środkiem okręgu. Odległość między każdym punktem krawędzi, a oszacowanym środkiem jest kandydatem na promień okręgu. Jednak w praktyce szacunki są niedokładne, dlatego wykrycie prawdziwego lokalnego maksimum może być trudne. [6]

Kolejną metodą jest Gerig i Klein Hough Transform (GKHT). Miała ona za zadanie rozwiązać problem dużego zapotrzebowania na pamięć, które występuje dla dużych promieni. Zamiast używania trójwymiarowej przestrzeni parametrycznej, korzysta się z trzech dwuwymiarowych tablic. Dwie z nich używane są do lokalizacji środka, a jedna do odpowiadającego mu promienia. W ten sposób udaje się zmniejszyć zapotrzebowanie na miejsce, jednak nie ma możliwości wykrywania koncentrycznych okręgów o różnych środkach.[6]

Rozwinięciem powyższego algorytmu jest Gerig Hough Transform z gradientem (GHTG). Jest on rozwinięty o kierunek gradientu, co pozwala na zwiększenie efektywności. Ogranicza to w akumulacji i późniejszym przetwarzaniu obszar całego koła.[6]

Ciekawymi i nietypowymi algorytmami są Randomized Hough Transform (RHT) i jej rozwinięta wersja Vector Quantization of Hought Tranfrom (VQHT), który wykrywa okręgi bardziej efektywnie. Jest to wykonane poprzez wcześniejsze rozłożenie krawędzi obrazu na wiele podobrazów. Punkty krawędzi znajdujące się w każdym podobrazie są rozpatrywane jako jedna grupa kandydatów na okręgi. Jest to szybka, ale skomplikowana metoda, która pozwoliła na wyeliminowanie wad RHT. [5]

Pojawiają się też algorytmy, które są połączeniem kilku z poprzednich. Takim algorytmem może być połączenie gradientowej transformaty Hough'a z jednowymiarową transformatą Hough'a. Wyniki badań pokazują, że algorytm ma wysoką wydajność wykrywania, zmniejsza złożoność czasową względem tradycyjnego algorytmu chord midpoint Hought transform. [4]

Innymi metodami wykorzystywanymi do wykrywania środka okręgów jest transformata Hough'a 2-1 czy szybka transformata Hough'a. [6]

Proponowane podejście powinno dążyć do zapewnienia kompromisu pomiędzy dokładnością, odpornością na zakłócenia, a efektywnością obliczeniową, która jest kluczowa w przetwarzaniu dużej ilości danych i czasem działania. Z tego powodu przeniesienie obliczeń na platformę równoległą jest mile widziane, a w przypadku obrazów o wysokiej rozdzielczości wręcz konieczne.

2.2 Zarys ogólny proponowanego rozwiązania

W ramach projektu opracowano oraz zaimplementowano algorytm wykrywania współśrodkowych okręgów w obrazach cyfrowych, wykorzystujący klasyczne podejście do badanego problemu. Metoda polega na wyznaczeniu dla każdego piksela obrazu wartości akumulatora, którego wartość odpowiada liczbie otaczających go pikseli krawędzi jądra będącego odpowiednikiem okręgu o promieniu *r*. Jądro zostało dodatkowo rozszerzone poprzez wprowadzenie parametru progowego (*threshold*), co umożliwia precyzyjniejsze wykrywanie oraz pożądaną odporność algorytmu na drobne zniekształcenia.

Rozwiązanie rozpoczyna się od klasycznego przetwarzania wstępnego: obraz poddawany jest konwersji na skalę szarości i detekcji krawędzi, z wykorzystaniem operatora Canny'ego.

Po wstępnym przygotowaniu obrazu delegowane jest poprzez OpenCL zadanie do platformy obliczeniowej polegające na zebraniu akumulatorów przestrzeni Hougha dla zadanego promienia. Następnie wykończona przestrzeń podlega etapowi wyszukiwania ekstremów na podstawie wartości progowej (zależącej od promienia) z dodatkowym parametrem pozwalającym na zmniejszenie wartości progowej w przypadku zniekształceń obrazu. Zastosowanie OpenCL pozwoliło zrealizować większość etapów w sposób zrównoleglony na poziomie GPU, co znacząco zwiększyło wydajność całego rozwiązania.

W projekcie osiągnięto następujące rezultaty:

- Zaprojektowano algorytm łączący dwa podejścia Hougha w celu wykrywania okręgów współśrodkowych.
- Zrealizowano implementację w OpenCL, umożliwiającą równoległe przetwarzanie danych na GPU.
- Przeprowadzono testy detekcji na obrazach syntetycznych i rzeczywistych, potwierdzające odporność metody na zakłócenia i wysoką efektywność czasową.

Metoda zachowuje wysoką jakość detekcji w warunkach szumu, deformacji i częściowego braku danych, co czyni ją użyteczną w rzeczywistych systemach przetwarzania obrazów.

2.3 Dyskusja alternatywnych rozwiązań

Alternatywnym podejściem jest zastosowanie gradientowych metod detekcji kół, co jednak wiąże się z koniecznością wdrożenia dodatkowego etapu przeszukiwania obrazu w celu precyzyjnego określenia gradientu. Dodatkowo innym rozwiązaniem jest zastosowanie jedego z opisanych wcześniej algorytmów takich jak transformata Houh'a 2-1 który bardzo dobrze się sprawdza, jednak jest to bardziej zaawansowany algorytm, a sam może być jeszcze mniej odporny na szum.

3. Analiza złożoności i estymacja zapotrzebowania na zasoby

Złożoność obliczeniowa algorytmu, ze względu na iterowanie po 3 wymiarze wynosi O(n^3), co jest bardzo wymagające obliczeniowo. Niemniej ze względu na użycie GPU nie jest to problem uniemożliwiający detekcji okręgów na obrazie w skończonym czasie.

4. Koncepcja proponowanego rozwiązania

Projektowany system został zrealizowany jako aplikacja hybrydowa, w której przetwarzanie danych obrazu jest podzielone pomiędzy jednostki programowe (CPU – PC) oraz sprzętowe (akcelerator OpenCL na GPU). Struktura opiera się na architekturze host-device, w której CPU zarządza przepływem danych, kontroluje sekwencję operacji i interfejs użytkownika, natomiast detekcja okręgów przy użyciu transformacji Hougha realizowana jest jako jądra OpenCL uruchamiane na akceleratorze sprzętowym.

Tabela 1 Specyfikacja komponentów fukcjonalnych.

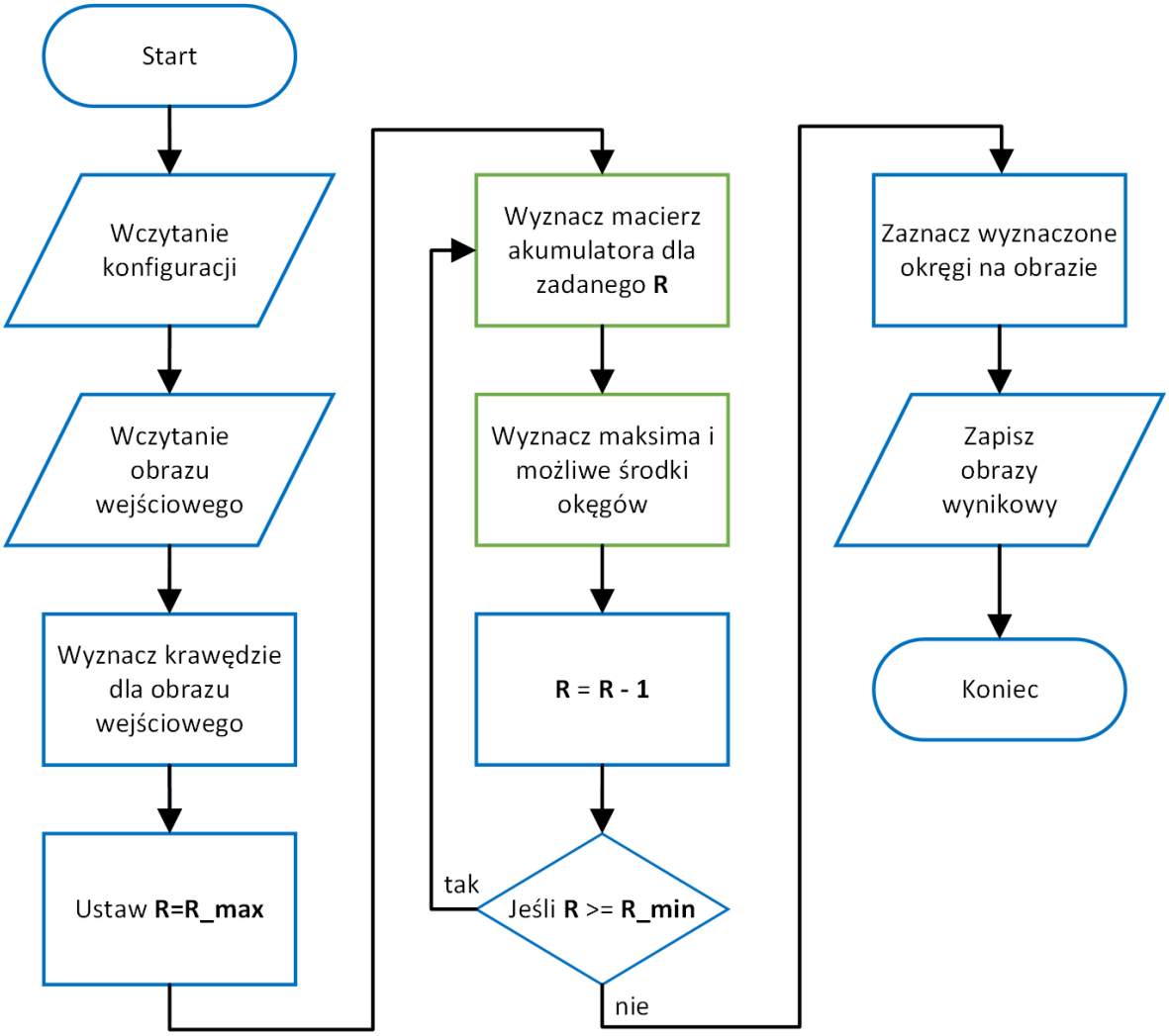
Komponent	Typ	Opis funkcjonalny
main.cpp	CPU	Załadunek obrazu, inicjalizacja platformy OpenCL, kompilacja kernelów, konrtola wykonania
FindCircle.cl	GPU	Plik jest wyposażony w dwa kernele 1) Wyznaczający akumulator Hougha 2) Wyznaczający ekstrema będące środkiem okręgu
KernelUtils.cpp	CPU	Obsługa plików z kodem kernel, wybór urządzeń OpenCL
create_circle_with_th.cpp	CPU	Generowanie maski do detekcji promieni (referencyjnie/testowo)

Tabela 2 Podział zadań – platforma programowa i sprzętowa.

Zadanie	Przypisanie	Uzasadnienie
Wczytanie obrazu i preprocesing (OpenCV)	CPU	Szybkość implementacji i niski koszt zasobowy
Wykrycie środka	GPU (OpenCL)	Duża liczba operacji akumulacyjnych i wektorowych
Wizualizacja wyników	CPU	Integracja z GUI i OpenCV

System został zaprojektowany w sposób sprzętowo-niezależny – przy użyciu OpenCL, dzięki czemu może zostać uruchomiony zarówno na CPU, jak i akceleratorze GPU (np. NVIDIA, Intel), a także platformach FPGA wspierających OpenCL. Komunikacja między hostem a urządzeniem sprzętowym odbywa się za pomocą standardowego interfejsu OpenCL:

- **Zarządzanie pamięcią:** clCreateBuffer, clCreateImage
- **Transfer danych**: clEnqueueWriteImage / clEnqueueReadImage
- **Zarządzanie kernalami**: clCreateKernel, clSetKernelArg, clEnqueueNDRangeKernel



Rys. 1 Schemat blokowy algorytmu

Na rysunku 1 kolorem niebieskim został zaznaczony host (PC), a kolorem zielonym urządzenie wykorzystujące OpenCL.

5. Symulacja i testowanie

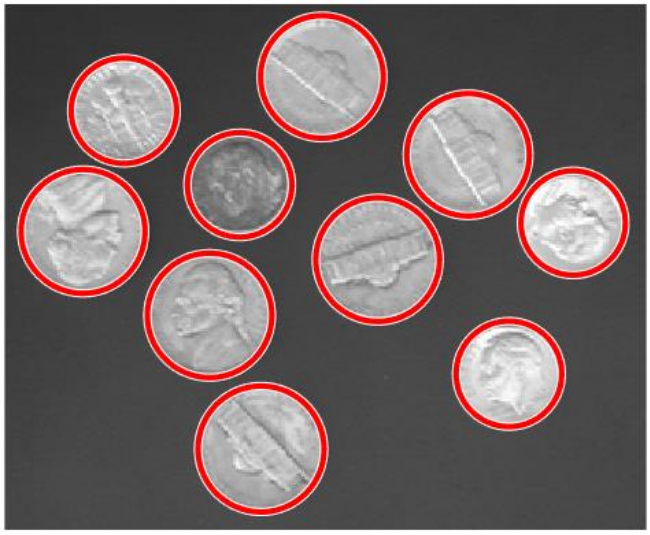
Testy jakościowe wykazały, że algorytm bardzo skutecznie rozpoznaje wzorzec idealnego koła. W przypadku obrazów zniekształconych oraz niedokładnych konturów niezbędne było wprowadzenie dodatkowego parametru, który obniża próg, powyżej którego lokalne ekstremum kwalifikowane jest jako środek koła dla sprawdzanego promienia.

Testy ilościowe przeprowadzone na CPU ujawniły wysoką czasochłonność algorytmu, co wymusiło wykorzystanie GPU do przyspieszenia obliczeń. Stanowi to doskonały przykład zastosowania zaawansowanych platform sprzętowych do rozwiązywania współczesnych wyzwań wynikających z przetwarzania bardzo dużych zbiorów danych.

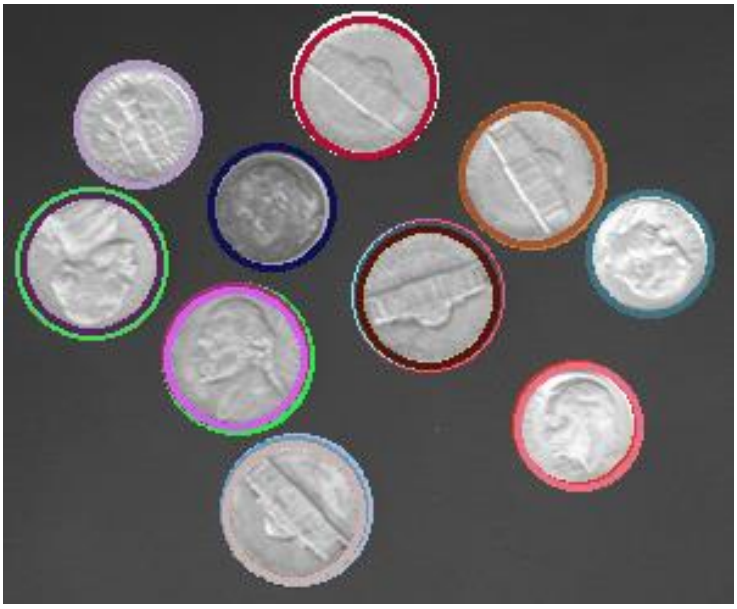
Poniżej przedstawiono przykładowe działanie algorytmu:

Rys. 2 Wykrycie słoi na przekroju drzewa w obrazie wysokiej rozdzielczości.

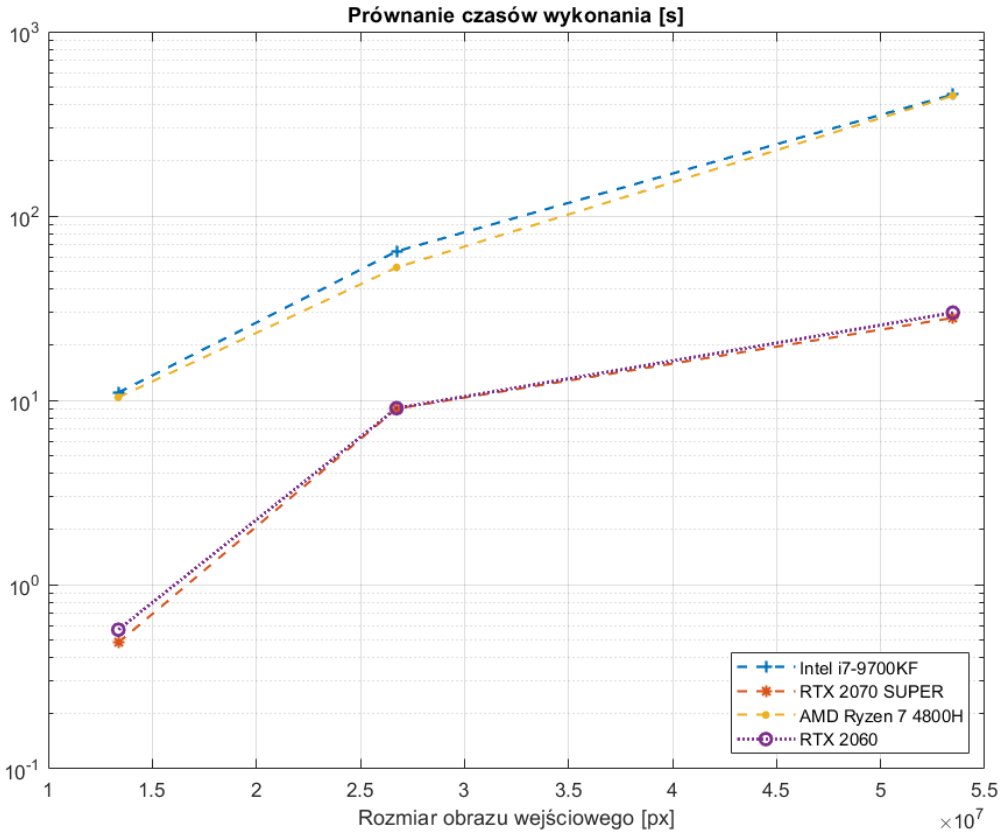
Algorytm bardzo dobrze radzi sobie z wykrywaniem współśrodkowych okręgów na obrazie wysokiej rozdzielczości. (Rys. 2) Obraz był przetwarzany dla resize 0.3. Było to spowodowane niedoskonałością algorytmu, dla którego przy większych rozmiarach obrazu zaszumienie przy krawędziach jest zbyt duże aby wykrywać poprawnie okręgi. Porównując działanie algorytmu (Rys. 4) do gotowej funkcji z matlaba (Rys. 3), można stwierdzić że osiąga ono porównywalną skuteczność detekcji względem matlaba, przy dodatkowej możliwości rozszerzenia na platformy sprzętowe (FPGA, GPU) i kontroli nad wszystkimi etapami przetwarzania.



Rys. 3 Wykrycie okręgów na monetach za pomocą wbudowanej funkcji w matlabie.



Rys. 4 Wykrycie okręgów na monetach za pomocą autorskiej aplikacji.



Rys. 5 Wykrycie słoi na przekroju drzewa w obrazie wysokiej rozdzielczości.

Na powyższym wykresie (Rys. 5) widzimy porównanie czasów z jakimi nasz algorytm dokonuje detekcji okręgów za pomocą transformaty Houh'a. Dwie jednostki jakimi są Intel i7-9700KF oraz AMD Ryzen 7 4800H mają znacząco gorsze wyniki od RTX 2060 i RTX 2070 SUPER. Większa szybkość działania na jednostkach GPU w porównaniu z CPU może być spowodowana zrównolegleniem obliczeń. Dodatkowo można zauważyć że wraz ze wzrostem obrazu wejściowego rośnie czas potrzebny do wykonania zadania. Jednak nie jest to zmiana liniowa. Porównując najmniejszy rozmiar obrazu a środkowy można zauważyć znaczący przeskok w działaniu GPU. Korzyści jakie zyskujemy poprzez zrównoleglenie obliczeń są minimalizowane poprzez dodatkową konieczność przesyłania danych. Dopiero przy największym obrazie mamy największą korzyść i większą efektywność.

6. Rezultaty i wnioski

Na podstawie przeprowadzonych badań wysnute zostały następujące wnioski dotyczące algorytmu SHT oraz przetwarzania obrazu w środowisku OpenCL na platformie obliczeniowej typu GPU:

- **Przewaga GPU nad CPU** Badania wykazały, że zastosowanie GPU znacząco przyspiesza obliczenia w porównaniu do tradycyjnych procesorów CPU. Wynika to z równoległego przetwarzania danych, co stanowi kluczowy atut w kontekście przetwarzania dużych zbiorów danych.
- **Zalety OpenCL** Implementacja algorytmu przy wykorzystaniu platformy OpenCL umożliwiła elastyczne i efektywne wykorzystanie zasobów sprzętowych. OpenCL zapewnia wysoką skalowalność i wydajność, co czyni go znakomitym narzędziem do realizacji intensywnych obliczeniowo operacji na dużych zbiorach danych.
- **Sugestie przyszłych badań** Wyniki eksperymentów wskazują na korzyści płynące z dalszej integracji metod gradientowych oraz optymalizacji obliczeń na CPU. Rozszerzenie badań nad tym zagadnieniem może przyczynić się do opracowania jeszcze bardziej efektywnych algorytmów detekcji, łącząc zalety różnych technik obliczeniowych.

7. Podsumowanie

Projekt przedstawia metodę detekcji współśrodkowych okręgów w obrazach cyfrowych z wykorzystaniem transformaty Hougha. Kluczowym elementem procedury jest wstępne wygładzanie obrazu oraz detekcja krawędzi, co umożliwia precyzyjne określenie środków okręgów, nawet przy analizie obrazów o bardzo wysokiej rozdzielczości. Optymalizacja osiągnięta dzięki implementacji w języku C++ z wykorzystaniem frameworku OpenCL oraz akceleracji obliczeń na GPU znacząco skraca czas przetwarzania, potwierdzając skuteczność równoległych obliczeń w analizie obrazów. Realizacja projektu została zakończona sukcesem. Zakresy promieni, określane przez użytkownika, są poprawnie wykrywane, a odpowiedni dobór progu pozwala na zminimalizowanie negatywnego wpływu szumu oraz deformacji kół. Wśród dalszych kierunków rozwoju warto rozważyć implementację algorytmu detekcji okręgów opartego na gradientach krawędzi, co może przyspieszyć obliczenia, choć kosztem pewnej utraty dokładności.

8. Literatura

[1] Przybyło J. „SIS – szablon raportu”, Kraków, Laboratorioum 2000

[2] Jeppe Jensen, “Hough Transform for Straight Line”

[3] Dana H. Ballard “Generalizing the Hough Transform to detect arbitrary shapes”, University of Rochester, 10.1979

[4] Xing Chen, Ling Lu, Yang Gao, “A New Concentric Circle Detection Method Based on Hough Transform”, The 7th International Conference on Computer Science & Education, July 14-17, 2012. Melbourne, Australia

[5] Bing Zhou, „Using Vector Quantization of Hough Transform for Circle Detection”, 14th International Conference on Machine Learning and Applications, 2015

[6] HK. Yuen, J. Princen, J. Illingworth, J. Kittler, “Comparative study of Hough Transform metods for circle finding”, University of Surrey, 2003

[7] YouTube: “First Principles of Computer Vision - Transform | Boundary Detection“ Dostępny: https://www.youtube.com/watch?v=XRBc_xkZREg (odwiedzona 13.05.2025)

[8] Week 5: Hough Transform (Line and Circle Detection). Dostępny: https://sbme-tutorials.github.io/2018/cv/notes/5_week5.html (odwiedzona 13.05.2025)

[9] Hough Circle Transform Dostępny: https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html (odwiedzona 13.05.2025)

[9] Circle Hough Transform. Dostępny: https://en.wikipedia.org/wiki/Circle_Hough_Transform (odwiedzona 13.05.2025)

9. Dodatek A: Szczegółowy opis zadania

Podział zadań w projekcie:
Bartek Stec: aplikacja sterująca
Mateusz Zajda: kernel i model referencyjny
Mikołaj Pietrzyk: przegląd literatury i raport

Projekt opiera się na detekcji współśrodkowych okręgów w obrazach cyfrowych z wykorzystaniem transformaty Hougha. Algorytm realizowany jest w formie hybrydowej: część przetwarzania wykonywana jest sekwencyjnie na CPU (host), a zasadnicze, obciążające obliczeniowo etapy są wykonywane współbieżnie na GPU za pomocą jąder OpenCL.

Konwersja obrazu do skali szarości:

Opis: Funkcja OpenCV pobiera obraz RGB i przelicza go na obraz w skali szarości według wzoru NTSC:

$$Y = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

Typ przetwarzania: równoległe (dla każdego piksela niezależnie)
Język implementacji: C++

Rozmycie obrazu metodą Gaussa:

Opis: Funkcja OpenCV pobiera obraz w skali szarości i rozmywa go jądrem rozmiaru 5x5. Ogólny wzór, który wykorzystywany jest do przekształcenia każdego piksela kontekstu:

$$G_0(x,y) = Ae^{\frac{-(x-\mu_x)^2}{2\sigma_x^2}+\frac{-(y-\mu_y)^2}{2\sigma_y^2}}$$

gdzie:
A – stała normalizacyjna przyjmowana tak by suma pikseli kontekstu była równa 1,
x, y – położenie każdego z pikseli kontekstu,
μ_x, μ_y – środkowe położenie jądra (dla symetrycznego jądra μ_x = μ_y= 0),

σ_x, σ_y – szerokość rozkładu (większa wartość prowadzi do większego rozmycia w osi x/y).

Następnie wykonywana jest konwolucja po obrazie:

$$I(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j) \cdot G_0(i,j)$$

Typ przetwarzania: równoległe (dla każdego piksela niezależnie)

Język implementacji: C++

Wykrywanie krawędzi metodą Canny:

Opis: Funkcja OpenCV pobiera rozmyty obraz w skali szarości i zwraca krawędzie o szerokości jednego piksela. Metoda opiera się o wyszukanie gradientów w celu wyliczenia ich modułów i kierunków. Następnie dokonuje się supresji niemaksymalne i progowania. Najistotniejsze wzory to (opis zakłada, że obraz wejściowy jest już rozmyty):

1. Obliczenie modułów i kierunków gradientów:

$$G_x(x,y) = \frac{\partial A}{\partial x}(x,y), \qquad G_y(x,y) = \frac{\partial A}{\partial y}(x,y)$$

$$M(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2}, \qquad \theta(x,y) = \text{arctg}(\frac{G_y(x,y)}{G_x(x,y)})$$

2. Supresja niemaksymalna:

$$M(x,y) \geq M\left(x + \cos\big(\theta(x,y)\big), y + sin\big(\theta(x,y)\big)\right)$$

$$M(x,y) \geq M\left(x - \cos\big(\theta(x,y)\big), y - sin\big(\theta(x,y)\big)\right)$$

*Jeżeli warunki nie są spełnione to wartość $M(x,y)$ jest zerowana.

3. Progowanie:

$M(x,y) \geq T_h$ – silna krawędź,

$T_h \geq M(x,y) \geq T_m$ – słaba krawędź, która jest krawędzią tylko gdy jest połączona z silną krawędzią,

$M(x,y) \leq T_m$ – brak krawędzi.

Typ przetwarzania: równoległe (dla każdego piksela niezależnie)

Język implementacji: C++

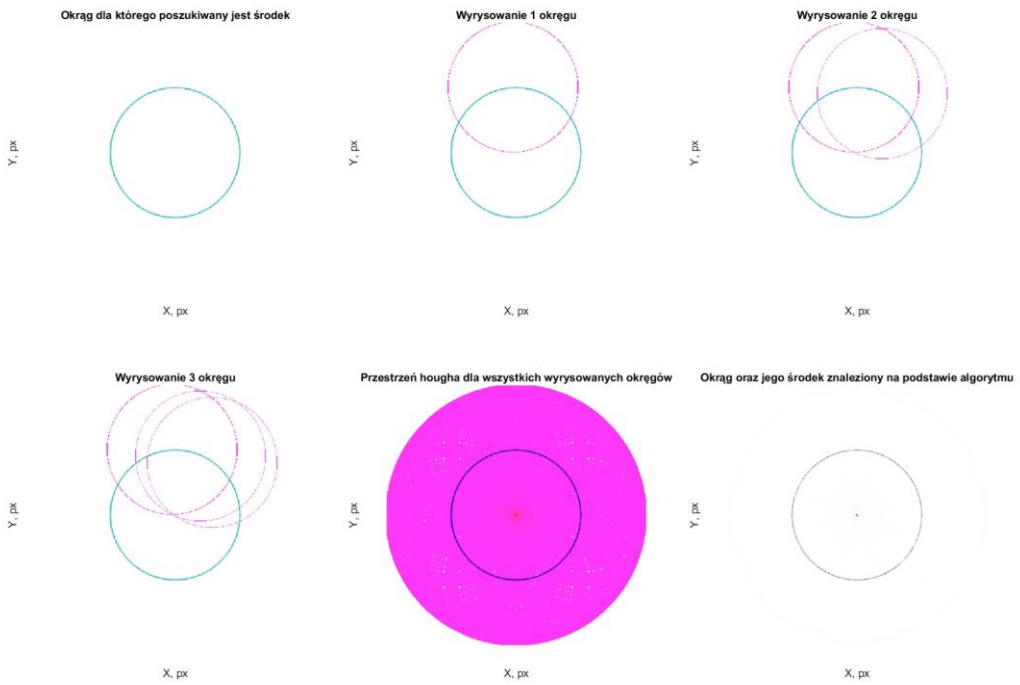
Standardowa transformata Hougha do wykrywania okręgów:

Kernele wykorzystywane w algorytmie wraz z wyjaśnieniem w pseudokodzie:

Kernel	**Objaśnienie pseudokodem (odpowiednik sekwencyjny)**

`__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;`
 	
`__kernel void FindCircle2(read_only image2d_t input_image,`
`const int base_radius,`
`const int tolerance,`
`const int angle_step,`
`__global uint* accumulator)`
`{`
`const int2 image_size = (int2)(get_global_size(0), get_global_size(1));`
`const int2 coord = (int2)(get_global_id(0), get_global_id(1));`
 	
`const uint4 center_pixel = read_imageui(input_image, sampler, coord.xy);`
 	
`if (center_pixel.x == 0)`
`return;`
 	
`#pragma unroll`
`for (int angle = 0; angle < 360; angle += angle_step)`
`{`
`const float theta = radians((float)angle);`
`const float cos_value = cos(theta);`
`const float sin_value = sin(theta);`
 	
`for (int dr = -tolerance; dr <= tolerance; ++dr)`
`{`
`const int radius = base_radius + dr;`
 	
`int a = (int)(coord.x - radius * cos_value);`
`int b = (int)(coord.y - radius * sin_value);`
 	
`if (a >= 0 && a < image_size.x && b >= 0 && b < image_size.y)`
`{`
`int idx = b * image_size.x + a;`
`atomic_inc(accumulator + idx);`
`}`
`}`
`}`
`}`
	`FUNCTION FindCircle2(input_image, base_radius, tolerance, angle_step, accumulator):`
	
	`# Pobierz rozmiary obrazu (szerokość i wysokość)`
	`image_size ← width and height of input_image`
	
	`# Odczytaj koordynaty piksela`
	`coord ← index x and y of pixel on input_image`
	
	`# Iteruj po każdym pikselu obrazu`
	`FOR each pixel with coordinates (x, y) in input_image DO:`
	`# Odczytaj wartość piksela centralnego`
	`center_pixel ← image(coord.x, coord.y)`
	
	`# Jeśli piksel centralny jest zerowy, przejdź do następnego`
	`IF center_pixel == 0 THEN:`
	`CONTINUE to next pixel`
	`END IF`
	
	`# Przechodź po kątach od 0 do 360 stopni z krokiem angle_step`
	`FOR angle FROM 0 TO 360 STEP angle_step DO:`
	`# Przekonwertuj wartość angle ze stopni na radiany i wyznacz wartości sinusa i cosinusa dla tego kąta`
	`theta ← convert(angle) to radians`
	`cos_value ← cos(theta)`
	`sin_value ← sin(theta)`
	
	`# Iteruj w zakresie od -tolerance do tolerance z krokiem 1`
	`FOR dr FROM -tolerance TO tolerance DO:`
	`# Do promienia dodaj dodatkową wartość dr by uwzględnić threshold wokół promienia`
	`radius ← base_radius + dr`
	
	`# Oblicz współrzędne punktu na okręgu`
	`a ← coord.x - radius * cos_value`
	`b ← coord.y - radius * sin_value`
	
	`# Jeśli punkt (a, b) mieści się w granicach obrazu`
	`IF (a, b) is within input_image bounds THEN:`
	`# Ikrementuj wartość akumulatora w odpowiadającym badanemu pikselowi miejscu w tablicy accumulator`
	`idx ← convert (a, b) to corresponding 1D index in accumulator`
	`accumulator[idx] ← 1 + accumulator[idx]`
	`END IF`
	`END FOR`
	`END FOR`
	`END FOR`
	`END FUNCTION`

Kernel FindCircle2 dokonuje wyznaczenia wartości przestrzeni Hougha w badanym pikselu na podstawie: otrzymanego binarnego obrazu *input_image*, promienia *base_radius*, tolerancji promienia *tolerance*, kroku kąta *angle_step* oraz zapisuje tę wartość do akumulatora *accumulator*. Jeżeli wartość badanego piksela wynosi 0 to funkcja nie aktualizuje wartości akumulatora przestrzeni Hougha. W przeciwnym wypadku rozpoczyna się iteracja w pętli for po kącie *angle* od 0 do 360 stopni z krokiem *angle_step*, po czym wartość *angle* konwertowana jest na radiany w celu wyznaczenia wartości cosinusa oraz sinusa tego kąta. Następnie rozpoczyna się zagnieżdżona pętla for, która iteruje po wartości zmiennej *dr* od *-tolerance* do *+tolerance*, w celu poszerzenia promienia *base_radius* by uwzględnić wpływ odchylenia badanych okręgów od idealnego (teoretycznego) kształtu. W kolejnym kroku wyznaczane są wartości *a* oraz *b* (na podstawie wartości zmiennej *radius* równej *base_radius* + *dr* oraz wartości sinusa i cosinusa kąta *angle*) reprezentujące kandydata na środek koła w przestrzeni Hougha. Jeżeli współrzędne *a* i *b* mieszczą się w graniach obrazu to są one zamieniane na odpowiedni indeks *idx* akumulatora *accumulator*, w którym zostanie on zwiększony. Poniżej przedstawiony został rysunek (Rys. 6) działania kernela, przy czym ostatni obrazek (2 rząd, 3 kolumna) jedynie symbolizuje centrum, które zostanie dokładnie wyznaczone w kernelu FindRadius, a także okrąg (rysowany jest on poza kernelami), który powinien powstać przez wyznaczenie środka dla zadanego na wejściu promienia.



Rys. 6 Przedstawienie wykrywania środka okręgu

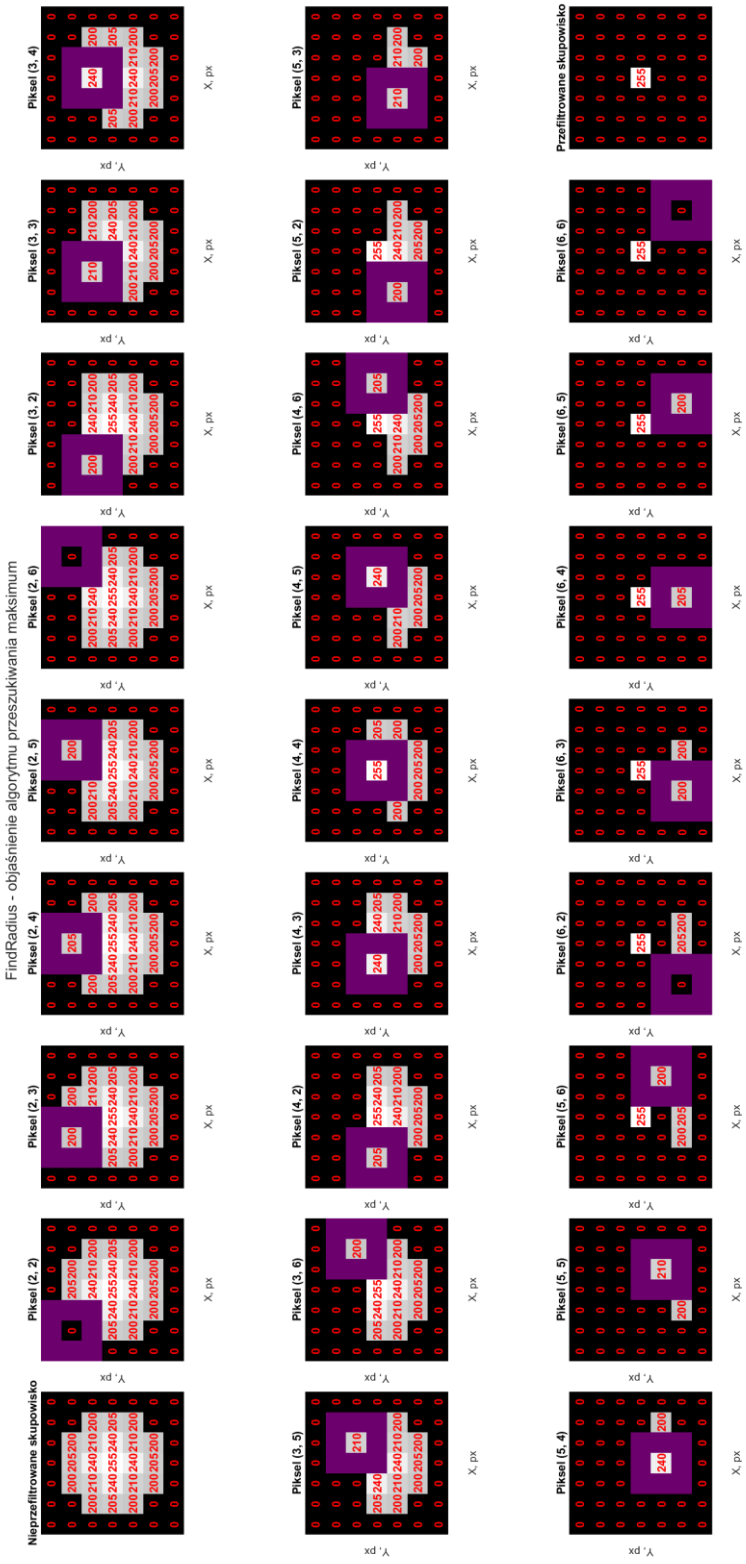
Kernel	Objaśnienie pseudokodem (odpowiednik sekwencyjny)

constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;
 	
__kernel void FindRadius(__global uint* input_image,
uint threshold,
int hough_max_radius,
write_only image2d_t output_image)
{
const int2 image_size = (int2)(get_global_size(0), get_global_size(1));
const int2 coord = (int2)(get_global_id(0), get_global_id(1));
 	
const int idx = coord.y * image_size.x + coord.x;
const uint center_pixel = input_image[idx];
uint4 output = (uint4)(0);
 	
bool has_larger_neighbor = false;
 	
if (center_pixel > threshold)
{
for (int dy = -hough_max_radius; dy <= hough_max_radius; ++dy)
{
for (int dx = -hough_max_radius; dx <= hough_max_radius; ++dx) {
int2 neighbor_coord = coord + (int2)(dx, dy);
 	
if (neighbor_coord.x < 0 || neighbor_coord.x >= image_size.x ||
neighbor_coord.y < 0 || neighbor_coord.y >= image_size.y)
continue;
 	
int neighbor_idx = neighbor_coord.y * image_size.x + neighbor_coord.x;
uint neighbor_pixel = input_image[neighbor_idx];
 	
if (neighbor_pixel > center_pixel) {
has_larger_neighbor = true;
break;
}
}
if (has_larger_neighbor) break;
}
if (!has_larger_neighbor) {
output = (uint4)(255);
}
}
write_imageui(output_image, coord, output);
}
	FUNCTION FindRadius(input_image, threshold, hough_max_radius, output_image):
	
	# Pobierz rozmiary obrazu (szerokość i wysokość)
	image_size ← width and height of input_image
	
	# Odczytaj koordynaty piksela
	coord ← index x and y of pixel on input_image
	
	# Iteruj po każdym pikselu obrazu
	FOR each pixel with coordinates (x, y) in input_image DO:
	# Wyznacz wartość piksela centralnego
	idx ← coord.y * image_size.x + coord.x
	center_pixel ← input_image[idx]
	
	# zainicjalizuj wartość, która zostanie przypisana do wyjścia na współrzędnych coord jeśli badany piksel nie jest lokalnym maksimum
	output ← (0, 0, 0, 0)
	
	# zainicjalizuj zmienną determinującą, że badany piksel jest lokalnym maksimum
	has_larger_neighbor ← FALSE
	
	# Jeżeli piksel centralny jest większy niż threshold
	IF center_pixel > threshold THEN:
	
	# Iteruj w zakresie od -hough_max_radius do hough_max_radius z krokiem 1
	FOR dy FROM -hough_max_radius TO hough_max_radius DO:
	
	# Iteruj w zakresie od -hough_max_radius do hough_max_radius z krokiem 1
	FOR dx FROM -hough_max_radius TO hough_max_radius DO:
	neighbor_coord ← coord + wektor(dx, dy)
	
	# Jeżeli neighbor_coord.x lub neighbor_coord.y nie mieści się w granicach obrazu przejdź do następnego piksela
	IF neighbor_coord is not within input_image bounds THEN:
	CONTINUE
	END IF
	
	# Odczytaj wartość piksela sąsiadującego na podstawie jego koordynatów
	neighbor_idx ← neighbor_y * image_size.width + neighbor_x
	neighbor_pixel ← input_image[neighbor_idx]
	
	# Jeżeli piksel sąsiadujący ma większą wartość to ustaw wartość has_larger_neighbor na pozytywną i wyjdź z obu pętli
	IF neighbor_pixel > center_pixel THEN:
	has_larger_neighbor ← TRUE
	BREAK OUT OF BOTH LOOPS
	END IF
	END FOR
	END FOR
	
	# Jeżeli piksel nie ma w otoczeniu sąsiada o większym ekstremum to przypisz do zmiennej out wektor 4 elementowy z wartościami [255, 255, 255, 255]
	IF NOT has_larger_neighbor THEN:
	output ← (255, 255, 255, 255)

```
            END IF
        END IF

        output[coord] = output
    END FOR
END FUNCTION
```

Kernel FindRadius wyznacza środki poszukiwanych okręgów poprzez detekcję lokalnych maksimów wśród kandydatów, czyli pikseli, których wartość przekracza zadany próg i znajduje się w przestrzeni Hougha wyznaczonej przez kernel FindCircle2. Na wejściu kernel otrzymuje: przestrzeń Hougha jako tablicę *input_image*, próg do wyznaczenia kandydatów *threshold,* wartość promienia *hough_max_radius,* który służy do wyznaczenia otoczenia w poszukiwaniu lokalnych maksimów oraz obraz wynikowy, który przechowa centra okręgów *output_image.* Na samym początku wartość wyjściowa *output* z kernela jest zakładana jako zerowa (tj. zakładamy, że nie występuje w miejscu piksela maksimum lokalne badanego otoczenia) oraz zakładamy, że w otoczeniu nie występuje piksel o wyższej wartości (*has_larger_neighbor* jest inicjalizowane wartością false). Jeżeli piksel nie spełnia warunku zostania kandydatem (jego wartość nie jest większa od *threshold*) to wartość *output* nie jest aktualizowana i jest ona przypisywana do wyjściowego obrazu *output_image.* W przeciwnym wypadku sprawdzane jest otoczenie wokół piksela (podwójną pętlą for) w kierunku x oraz y w zakresie od -*hough_max_radius* do *hough_max_radius*. Jeżeli piksel otoczenia wykracza poza obraz to wewnętrzna pętla przechodzi do następnej iteracji. Jeśli jednak piksel leży wewnątrz obrazu to pobierana jest jego wartość do zmiennej *neighbor_pixel* po czym jest porównywana z wartością piksela, dla którego kernel został wywołany. Jeżeli wartość *neighbor_pixel* jest wyższa to badany piksel nie jest maksimum, a co za tym idzie przerywane są obie pętle for i zwracana do obrazu wyjściowego wartość jest równa 0. Jeśli jednak żaden sąsiad badanego piksela nie ma wyższej wartości, do obrazu wyjściowego *output_image* zapisywana jest wartość 255, co oznacza lokalne maksimum, a tym samym wyznaczony środek poszukiwanego okręgu o zadanym promieniu. Schematyczne przedstawienie idei działania kernela zostało zamieszczone na poniższym rysunku:



Rys. 7 Przedstawienie działania kernela

Typ przetwarzania: równoległe (dla każdego piksela niezależnie)

Język implementacji: OpenCL C

10. Dodatek B: Dokumentacja techniczna

Komputer w laboratorium na którym był testowany program miał numer: PCL172, a nazwa konta na którym znajdują zainstalowane odpowiednie biblioteki to: Crircle25

10.1 Środowisko programowania:

- **System operacyjny**: Linux/Windows (kompatybilny z OpenCL SDK)
- **Języki programowania**: C++, OpenCL C
- **Biblioteki zewnętrzne**:
  - OpenCV >= 4.0 – do wczytywania i przetwarzania obrazów
  - OpenCL >= 2.0 – do akceleracji algorytmu
- **System budowania**: CMake

10.2 Procedura symulacji i testowania

Wymagane narzędzia:

- CMake >= 3.10
- OpenCV >= 4.5
- OpenCL SDK

Kroki:

1. Skonfiguruj projekt: `cmake -Bbuild -DCMAKE_BUILD_TYPE=Release`
2. Zbuduj: `cmake --build build`
3. Przejdź do folderu z aplikacją: `cd build/bin`
4. Uruchom aplikacje: `./Hough_transform`

W celu przestawania obrazu o dużej rozdzielczości należy go pobrać ze strony kursu do folderu *build/bin/data*, oraz ustawić odpowiednią nazwę obrazu w pliku konfiguracyjnym *config.toml*.

Wersja testowana:

- GPU: NVIDIA RTX / AMD Radeon z OpenCL 2.0

11. Dodatek D: Spis zawartości dołączonego nośnika (płyta CD room)

- **CMAKE** - folder zawierający pomocnicze skrypty wykorzystane podczas budowy projektu
- **DATA** – folder zawierający przykładowe dane wejściowe aplikacji.
- **DOC** – folder zawierający sprawozdanie.
- **SRC** – folder z plikami źródłowymi aplikacji wraz z plikami kernela OpenCL.
- **TEST** – folder zawierający pliki referencyjne wykorzystane w czasie sprawdzania poprawności wykonania aplikacji.

12. Dodatek E: Historia zmian:

Tabela 3 Historia zmian.

Autor	Data	Opis zmian	Wersja
Mikołaj Pietrzyk	2025-04-29	Utworzono stronę tytułową. Dodano rozdziały pochodzące z dokumentu [1]	0.0
Mikołaj Pietrzyk	2025-05-13	Napisano rozdział 1 Napisano rozdział 2.1 Napisano rozdział 2.2 Napisano rozdział 4 Napisano rozdział 8 Napisano rozdział 9 Napisano rozdział 10	1.1
Mateusz Zajda	2025-05-13	Korekta rozdziałów 1 – 6	1.2
Bartłomiej Stec	2025-05-13	Poprawki merytoryczne	1.3
Mateusz Zajda	2025-05-20	Poprawiono rozdział 1 Dodano rozdział 7 Poprawiono rozdział 9	2.1
Mikołaj Pietrzyk	2025-05-21	Zredagowanie	2.2
Bartłomiej Stec	2025-05-22	Uzupełnienie rozdziału 4	2.3
Mateusz Zajda	2525-05-26	Uzupełnienie rozdziału 9	3.1
Mikołaj Pietrzyk	2025-05-26	Poprawa rozdziału 2 Uzupełnienie rozdziału 5	3.2
Mateusz Zajda	2025-05-27	Poprawki merytoryczne	3.3
Bartłomiej Stec	2025-05-27	Uzupełnienie Dodatku D	3.4

PR25Laaw07_CIRCLE			**Karta oceny projektu**
Data	Ocena	Podpis	Uwagi