

Kierunek: **Informatyka techniczna**
Specjalność: **Grafika i systemy multimedialne**

PRACA DYPLOMOWA MAGISTERSKA

**Autorska implementacja algorytmów
transformacji Hough'a dla wybranych
metod akceleracji obliczeń
w środowiskach języka JavaScript**

**Proprietary implementation of Hough
transformation algorithms for selected
calculation acceleration methods in
JavaScript language environments**

Damian Koper

Opiekun pracy
Dr inż. Marek Woda

Streszczenie

W tej pracy zaprezentowane zostały popularne metody akceleracji obliczeń w środowiskach języka JavaScript. Zbadano wydajność w środowiskach przeglądarek internetowych Google Chrome oraz Mozilla Firefox, a także w środowiskach serwerowych - NodeJS i Deno. Metodę wykrywania kształtów z użyciem algorytmów Standard Hough Transform i Circle Hough Transform zaimplementowano z użyciem metody poprawy wykonania sekwencyjnego, wykorzystania natywnych rozszerzeń NodeJS, komplikacji i wykonania kodu asm.js i WebAssembly, również w wariantie SIMD. Zaimplementowano również metody współbieżne z wykorzystaniem Worker'ów oraz GPU używając WebGL API. Najlepszym ze środowisk okazała się przeglądarka Google Chrome, a najwolniejszym Mozilla Firefox. Najbardziej wydajną metodą sekwencyjną są natywne rozszerzenia NodeJS, a współbieżną WebGL. Opracowane wyniki stanowić mogą podstawę do wyboru metody akceleracji w implementowanych algorytmach, aby móc konkurować ze środowiskami innych języków i tworzyć wydajne algorytmy intensywne obliczeniowo.

Słowa kluczowe: javascript, akceleracja obliczeń, sht, standard hough transform node, przeglądarka internetowa, deno, webgl, webpack, wasm, simd, workers

Abstract

This paper presents popular acceleration methods in JavaScript execution environments. Performance of browser environments of Google Chrome and Mozilla Firefox was examined as well as the server ones - NodeJS and Deno. Algorithms used for benchmarking were Standard Hough Transform and Circle Hough Transform used for pattern detection in images. Acceleration method implemented are sequential execution improvement, NodeJS native addons, WebAssembly with asm.js and SIMD variants. Parallel implemented method are usage of Workers and GPU with WebGL API. It appears that the most performant environment is Google Chrome and the least one is Mozilla Firefox. The fastest sequential method is the usage of native addons in NodeJS and the parallel one is WebGL. Summarized results may help choosing the most suitable acceleration method for algorithm to be implemented, to be able to compete with other languages' environments and to create efficient compute-intensive algorytms.

Keywords: javascript, acceleration, sht, standard hough transform node, web browser, deno, webgl, webpack, wasm, simd, workers

Spis treści

1. Wstęp	13
1.1. Istota rzeczy	14
1.1.1. Duża i rosnąca rola technologii webowych	15
1.2. Cel badań i zawartość pracy	16
1.2.1. Perspektywa wydajności	17
1.2.2. Perspektywa kompatybilności i budowania bibliotek	17
1.2.3. Perspektywa wygody użytkowania	17
1.2.4. Transformacja Hough'a jako wspólny mianownik analizy	17
1.2.5. Zawartość pracy	18
2. Język JavaScript	19
2.1. Model wykonania	19
2.2. Środowiska JavaScript	21
2.2.1. Przeglądarka internetowa	21
2.2.2. NodeJS	21
2.2.3. Deno	22
2.3. Modularność i kompatybilność kodu	22
2.3.1. Modularność	22
2.3.2. Kompatybilność	24
2.4. Metody akceleracji	24
2.4.1. Optymalizacja wykonania sekwencyjnego	24
2.4.2. Natywne rozszerzenia	25
2.4.3. WebAssembly	25
2.4.4. Współbieżność	26
2.4.5. GPGPU	27
3. Transformacja Hough'a	29
3.1. Standard Hough Transform	29
3.2. Circle Hough Transform	31
3.2.1. Wariant z wykorzystaniem gradientu	32
3.2.2. Próbkowanie i złożoność obliczeniowa	33
4. Metodologia pomiarów	35
4.1. Biblioteka benchmark	36
4.2. Badane aspekty	39
5. Implementacja algorytmów	41
5.1. Organizacja plików	41
5.2. Budowanie bibliotek	42
5.3. Implementacja algorytmów	44

5.3.1. Wyniki działania algorytmów	45
6. Implementacja metod akceleracji i rezultaty pomiarów	49
6.1. Wykonanie sekwencyjne	49
6.1.1. Wyniki pomiarów	49
6.2. NodeJS Native C++ Addon	51
6.2.1. Wyniki pomiarów	52
6.3. WebAssembly i asm.js	52
6.3.1. Implementacja procesu budowania	53
6.3.2. Warianty testów	54
6.3.3. Wyniki pomiarów	55
6.4. Współbieżność	57
6.4.1. Wyniki pomiarów	59
6.5. GPGPU	61
6.5.1. Standard Hough Transform	61
6.5.2. Circle Hough Transform	62
6.5.3. Wyniki pomiarów	62
6.6. Problem zimnego startu	64
6.7. Przykład przetwarzania obrazu z kamery w przeglądarce internetowej	66
7. Podsumowanie	69
Literatura	75
A. Opis załączonej płyty CD/DVD	79

Spis rysunków

2.1. Uproszczony model pętli zdarzeń środowisk języka JavaScript w wariantie z wyroznieniem API przeglądarek internetowych.	20
2.2. Model architektury środowiska NodeJS.	21
2.3. Potok graficzny WebGL API, który może być wykorzystany do obliczeń ogólnego przeznaczenia.	27
3.1. Ogólny schemat przetwarzania obrazu z wykorzystaniem transformacji Hough'a.	29
3.2. Demonstracja transformacji Hough'a w wariantie równania kierunkowego prostej. .	30
3.3. Prosta opisana za pomocą odległości ρ i kąta θ od środka układu współrzędnych biegunowych.	30
3.4. Wynik transformacji Hough'a dla obrazu na rysunku 3.2a w wariantie współrzędnych biegunowych.	31
3.5. Wynik transformacji	32
4.1. Diagram klas biblioteki Benchmark.	37
5.1. Układ najważniejszych plików w repozytorium projektu z pakietami implementującymi metody akceleracji zaznaczonymi na czerwono.	42
5.2. Zależności pomiędzy pakietami w projekcie.	42
5.3. Definicja interfejsu algorytmów.	44
5.4. Wynik działania zaimplementowanego algorytmu SHT w wariantie sekwencyjnym. .	46
5.5. Wynik działania zaimplementowanego algorytmu CHT w wariantie sekwencyjnym. .	47
6.1. Wyniki pomiarów czasu wydajności dla wykonania sekwencyjnego SHT i CHT. . . .	50
6.2. Wynik profilowania wykonania sekwencyjnego algorytmu CHT w przeglądarce Google Chrome.	51
6.3. Znormalizowana do przedziału [0, 255] absolutna różnica wartości akumulatorów z wynikiem głosowania pomiędzy wykonaniem sekwencyjnym SHT <i>non-LUT</i> . . .	51
6.4. Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem natywnego rozszerzenia C++ w środowisku NodeJS.	53
6.5. Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem komplikacji do asm.js.	56
6.6. Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem komplikacji do WASM.	57
6.7. Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem komplikacji do WASM z ręczną implementacją instrukcji SIMD.	58
6.8. Sieć zależności pomiędzy modułami implementującymi algorytm SHT w wersji <i>non-LUT</i> , która zapewnia kompatybilność ze wszystkimi środowiskami.	59
6.9. Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem czterech Worker'ów.	60

6.10. Wynik profilowania wykonania algorytmu CHT w przeglądarce Google Chrome z użyciem Worker'ów.	60
6.11. Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem biblioteki <i>gpu.js</i>	64
6.12. Wynik profilowania wykonania algorytmu CHT w przeglądarce Google Chrome z wykorzystaniem biblioteki <i>gpu.js</i>	64
6.13. Wyniki pomiarów pierwszych czasów wykonania algorytmu SHT <i>non-LUT</i> znormalizowane do przedziału $[0, 1]$ dla $S_\theta = 1$	65
6.14. Wyniki pomiarów pierwszych czasów wykonania algorytmu SHT <i>LUT</i> znormalizowane do przedziału $[0, 1]$ dla $S_\theta = 1$	66
6.15. Wyniki pomiarów pierwszych czasów wykonania algorytmu CHT znormalizowane do przedziału $[0, 1]$ dla $n = 1$	67
6.16. Wizualizacja wykrytych kształtów na podstawie obrazu z kamery w przeglądarce internetowej.	68
6.17. Wynik profilowania wykonania sekwencyjnego algorytmu SHT <i>LUT</i> wraz z procesem pozyskania obrazu, przetwarzania wstępnego, wykrywania krawędzi i wyświetlenia wyników.	68

Spis tabel

1.1. Popularne biblioteki do przetwarzania danych w języku Python i ich odpowiedniki w języku JavaScript. Dane pochodzą z serwisów kolejno PyPI Stats oraz NPM, a w nawiasach znajduje się tygodniowa liczba pobrań biblioteki na dzień 2022-04-27.	15
1.2. Zaimplementowane metody akceleracji dla badanych środowisk.	16
4.1. Opis parametrów uruchomienia benchmarku w bibliotece <i>benchmark</i>	39
4.2. Wersje środowisk testowych wraz z ich wersjami silnika JavaScript.	39
6.1. Przyspieszenie i jego efektywność dla metody akceleracji z wykorzystaniem Worker'ów ($S_\theta = 1, n = 1, p = 4$).	59
7.1. Porównanie zaimplementowanych metod akceleracji dla badanych środowisk. Wykonanie dla SHT przy $S_\theta = 1$, dla CHT przy $n = 10$. Wykonanie dla CHT WebGL nie zostało uwzględnione.	72

Spis listingów

2.1. Przykład kodu demonstrujący mechanizmy asynchroniczności w języku JavaScript	20
2.2. Przykład wykorzystania ECMAScript Modules	23
2.3. Funkcja licząca silnię w języku C/C++	25
2.4. Funkcja licząca silnię w języku WASM	26
4.1. Kompozycja funkcji w trybie <i>extracted</i>	38
4.2. Przykładowy benchmark mierzący czas wykonania funkcji <code>Function.prototype()</code> 2000 razy.	38
5.1. Konfiguracja narzędzia Webpack służąca do budowania biblioteki <i>benchmark</i>	43
5.2. Definicja typów funkcji wariantu SHT i CHT	45
6.1. Plik powiązania kodu C++ z JavaScript	52
6.2. Komenda wykorzystana podczas komplikacji kodu C++ do modułu WebAssembly.	54
6.3. Wybrane fragmenty kodu warstwy powiązania WASM pomiędzy C++ i JavaScript.	55
6.4. Funkcja <code>getBounds</code> algorytmu CHT z wykorzystaniem instrukcji SIMD.	55
6.5. Przykład minifikacji kodu błędnie transpilowanego do języka ESSL przez bibliotekę <code>gpu.js</code>	62
6.6. Funkcja tworząca kernel dla wariantu SHT <i>non-LUT</i>	63
6.7. Funkcja przetwarzająca w pętli kolejne klatki obrazu z kamery.	68

Skróty

IDE (ang. *Integrated Development Environment*)

API (ang. *Application Programming Interface*)

CPU (ang. *Central Processing Unit*)

GPU (ang. *Graphics Processing Unit*)

GPGPU (ang. *General-purpose computing on graphics processing units*)

ESSL (ang. *OpenGL ES Shading Language*)

SPA (ang. *Single Page Application*)

DOM (ang. *Document Object Model*)

URL (ang. *Uniform Resource Locator*)

NPM (ang. *Node Package Manager*)

UMD (ang. *Universal Module Definition*)

AMD (ang. *Asynchronous Module Definition*)

ESM (ang. *ESModules, ECMAScript Modules*)

LUT (ang. *Lookup Table*)

PTLM (ang. *Point-to-line mapping*)

LUT (ang. *Lookup Table*)

SHT (ang. *Standard Hough Transform*)

CHT (ang. *Circle Hough Transform*)

WASM (ang. *WebAssembly*)

SIMD (ang. *Single Input Multiple Data*)

Rozdział 1

Wstęp

Prawo Moore'a mówi, że liczba tranzystorów w układach scalonych podwaja się co około dwa lata. Prawo Koomey'a opisuje natomiast trend wzrostu liczby obliczeń na jeden dżul energii, która podwaja się co 1.57 lat. Choć w ostatnich latach, w związku ze zmniejszającym się tempem miniaturyzacji tranzystorów, wartości te przestały być aktualne to wciąż mamy do czynienia ze zjawiskiem ustawicznego wzrostu mocy obliczeniowej. Dodatkowo, zgodnie z obserwacją nazwaną prawem Huang'a - prezesa firmy NVIDIA, wzrost wydajności układów graficznych wzrasta więcej niż dwukrotnie co dwa lata[16], co świadczy o obecnym rozwoju możliwości optymalizacji architektur i programów wykorzystujących przetwarzanie masowo równolegle. Wzrost wydajności z kolei zwiększa możliwości wykorzystania algorytmów, które wcześniej były zbyt intensywne obliczeniowo i nie mogły być wykorzystane w rozwiązaniach produkcyjnych. Powszechnie dostępne wydajne maszyny dają również możliwości szybszego prototypowania rozwiązań większej liczby osób, co z kolei wpływa na rozwój samych algorytmów i ich zastosowań. Algorytmy i obliczenia opierają się na osiągnięciach analizy numerycznej oraz matematyki dyskretnej.

Analiza numeryczna zajmuje się opisywaniem i analizą metod pozyskiwania wyników dla problemów matematycznych. Dzięki jej osiągnięciom możliwe jest budowanie algorytmów, które są kompletnym i jednoznaczny opisem metody konstruowania rozwiązania owych problemów. Konstruowane algorytmy mogą mieć różne poziomy skomplikowania zaczynając od obliczania wartości funkcji, wielomianów, czy też znajdować rozwiązania układów równań. Dzięki nim możemy aproksymować wartości funkcji - obliczać pochodne oraz całki korzystając z metod prostokątów, trapezów, czy Simpson'a. Możemy obliczać przybliżenia funkcji trygonometrycznych korzystając z szeregów Taylor'a. Dzięki obliczeniom opartym o algebrę liniową i rachunek różniczkowy mogły rozwinać się pola związane z uczeniem maszynowym. Znajdowanie wektorów i wartości własnych macierzy w metodzie PCA w celu redukcji wymiarowości danych, a w końcu algorytmy propagacji wstecznej szczególnie wykorzystywane w intensywnie rozwijającym się obszarze uczenia głębokiego[20].

Niezależnie od skali zaawansowania algorytmów dążą one do stanowienia rozwiązania konkretnych problemów świata rzeczywistego. Przykładem ich zastosowania jest przewidywanie pogody w oparciu o modele meteorologiczne, które zaczęły intensywnie rozwijać się wraz z dostępem do coraz większej mocy obliczeniowej i udoskonalaniem samych modeli. W ciągu 15 lat, od 1971 roku, spełnialność prognoz 36-godzinnych zrównała się ze spełnialnością prognoz 72-godzinnych[12]. Kolejnym wątkiem przytoczenia przykładem jest obszar analizy obrazów z wyszczególnieniem zagadnienia ich klasyfikacji, gdzie wzrost mocy obliczeniowej pozwolił na rozwój algorytmów. W ciągu 8 lat metryka dokładności klasyfikacji obrazów na zbiorze danych ImageNet wzrosła z 63% do 90%[12, 3].

Dynamiczny rozwój algorytmów napędzany jest przede wszystkim przez poszerzanie i budowanie wiedzy domenowej, specyficznej dla rozwiązywanego problemu. Jednak, aby uczynić taki rozwój możliwym, musi być on oparty na niezbędnych filarach jakimi są oprogramowanie, które służy do prototypowania, a następnie wdrażania rozwiązań oraz środowisko sprzętowe, które umożliwia przeprowadzanie obliczeń, wielokrotnych testów, prototypów na małą oraz wdrożeń na dużą skalę. Wraz ze wzrostem złożoności problemu liczba obliczeń niezbędnych do jego rozwiązania również rośnie i w przypadku ich większości ten wzrost jest wykładniczy lub większy. Niezbędnym zatem jest, aby oprogramowanie potrafiło wykorzystać wszelkie dostępne metody akceleracji zarówno te związane z optymalizacją samego algorytmu, jak i te związane z mechanizmami akceleracji sprzętowej.

1.1. Istota rzeczy

Wykonywanie obliczeń numerycznych wśród obecnie popularnych rozwiązań można podzielić na dwie grupy. Pierwsza z nich używa specjalnie do tego celu stworzonego języka programowania, często również w połączeniu ze zintegrowanym środowiskiem programistycznym (Integrated Development Environment, ang. IDE). Przykładem takiego rozwiązania jest środowisko i język MATLAB[45] oraz R[54], czy też środowisko Mathematica z językiem Wolfram[44].

Druga grupa używa języka ogólnego przeznaczenia do wykonywania obliczeń w oparciu o zewnętrzne biblioteki w zdecydowanej większości udostępniane jako oprogramowanie open-source, które dostarczają wymagany zestaw funkcjonalności i niwelują potrzebę ich ręcznej implementacji. Języki takie możemy podzielić na te niskiego poziomu, zapewniające wysoką wydajność, oraz te wysokiego poziomu, interpretowane, zapewniające większą wygodę użytkowania. Najbardziej popularnym tego typu środowiskiem jest język Python, którego społeczność stworzyła liczne biblioteki (w postaci pakietów) do przetwarzania danych, obliczeń numerycznych i statystycznych, analizy obrazów, czy uczenia maszynowego. Najpopularniejsze z nich podane zostały w tabeli 1.1. W nawiasach została ujęta liczba pobrań danego pakietu w przeciągu ostatniego tygodnia na dzień 2022-04-27. Dla punktu odniesienia warto dodać, że w przeciągu tego samego tygodnia, w całym ekosystemie, pobrano łącznie 3.409.997.407 pakietów [51].

Biblioteki takich języków, czego przykładem jest biblioteka OpenCV, często implementowane są w językach kompilowanych bezpośrednio do kodu maszynowego takich jak C++ czy Rust udostępniając jednolity interfejs, którego metody, poprzez powiązania, wywoływać mogą języki skryptowe wysokiego poziomu, takie jak już wspomniany Python. Takie rozwiązania zapewnia możliwość zbudowania wersji biblioteki kompatybilnej z wieloma środowiskami i językami na podstawie jednego kodu bazowego, oddając adaptacji tylko elementy architektury integrującej bibliotekę niskopoziomową i język wysokiego poziomu. Skompilowana biblioteka poddana może być również procesom optymalizacji, co zwiększyć może jej wydajność. Wadę takiego rozwiązania stanowi konieczność kompilowania biblioteki niskiego poziomu dla wielu systemów operacyjnych, czy architektur procesora. Taka komplikacja odbywać może się przed publikacją samej biblioteki, a gotowe artefakty pobierane są przez użytkownika w momencie instalacji. Kompilacja może odbywać się również bezpośrednio na maszynie użytkownika w momencie instalacji.

Opisany podział nie skłania do uznania przewagi jednej z grup nad drugą w żadnym z aspektów. W środowiskach specyficznych i zintegrowanych brakujące funkcjonalności mogą zostać dodane przez twórców jako biblioteki standardowe języka oraz zaimplementowane przez społeczność w bibliotekach języków ogólnego przeznaczenia. Obie grupy rozwiązań z reguły są w stanie działać w środowisku tego samego systemu operacyjnego i wchodzić w interakcje z tym samym sprzętem, i wykorzystać idące za tym możliwości akceleracji obliczeń. Jednak nie wszystkie języki ogólnego przeznaczenia i technologie zyskały jednakową popularność w ob-

Tab. 1.1: Popularne biblioteki do przetwarzania danych w języku Python i ich odpowiedniki w języku JavaScript. Dane pochodzą z serwisów kolejno PyPI Stats oraz NPM, a w nawiasach znajduje się tygodniowa liczba pobrań biblioteki na dzień 2022-04-27.

Python	JavaScript	Zastosowanie
numpy (26.067.844)	numjs (533)	Operacje na macierzach
pandas (19.778.648)	danfojs (1.029)	Operacje na strukturach danych
scipy (9.986.376)	simple-statistics (87.882) fft.js (8.027)	Operacje związane z analizą numeryczną, przetwarzanie sygnałów, algebra liniowa.
scikit-learn (7.705.438)	ml (156)	Uczenie maszynowe
matplotlib (6.457.099) plotly (1.632.246)	plotly.js (149.542) c3 (83.564)	Wizualizacja danych
tensorflow (3.348.986)	@tensorflow/tfjs (91.233)	Sieci neuronowe
opencv-python (1.236.711)	OpenCV.js (b.d [50]) jimp (1.479.783) image-js (4.103)	Operacja na obrazach, computer vision

szarze obliczeń numerycznych mimo swojej ogólnej popularności. Przykładem takowych są technologie webowe z językiem JavaScript na czele.

1.1.1. Duża i rosnąca rola technologii webowych

Technologie webowe zdefiniować można jako narzędzia i techniki umożliwiające wymianę danych pomiędzy różnymi urządzeniami przez internet. Technologie webowe mogą występować i spełniać różne zadania na wielu poziomach architektury aplikacji. W przypadku architektury klient-serwer środowiskiem frontend'owym umożliwiającym wyświetlanie i obsługę graficznego interfejsu użytkownika zwykle jest przeglądarka internetowa, gdzie za jego implementację na najniższym poziomie abstrakcji odpowiadają języki HTML, CSS i JavaScript. Serwerem może być na przykład aplikacja komunikująca się z klientem za pomocą API zaimplementowanego w architekturze REST, czy też za pomocą języka zapytań GraphQL uruchamiana w środowisku NodeJS.

Warto zaznaczyć, że serwer, jak i klient, nie muszą być zaimplementowane z wykorzystaniem języka JavaScript by być uznawanym jako aplikacja webowa. Języki takie jak PHP, Python, Ruby, Java, czy C# także oferują zaawansowane frameworki, jednak to język JavaScript, ze względu na historię swojego rozwoju ściśle powiązaną z przeglądarką internetową, uważany jest jako główne narzędzie i czynnik rozwoju technologii webowych zarówno w części klienckiej jak i serwerowej. Potwierdzają to badania, gdzie JavaScript w 2021 roku dziewiąty rok z rzędu został wyłoniony jako najpopularniejsza technologia wśród developerów [58].

JavaScript w obliczeniach numerycznych

Pomimo swojej popularności, w przeciwieństwie do języka Python, język JavaScript nie zyskał popularności wśród obliczeń inżynierskich, naukowych, statystycznych, obróbki i analizy obrazów. Jest on starszy od języka JavaScript i w przeciwieństwie do niego od początku mógł pełnić zadanie narzędzia do implementacji algorytmów obliczeniowych, podczas gdy JavaScript ograniczony był jedynie do jednowątkowego środowiska przeglądarki internetowej. Dopiero

w 2009 roku, wraz z pojawieniem się środowiska NodeJS, możliwe stało się uruchamianie kodu JavaScript po stronie serwera. Umożliwia to również powstałe w 2018 roku środowisko Deno.

Innym czynnikiem, który warunkuje tempo rozwoju i potencjalny przepływ użytkowników ku lepszym ich zdaniem środowiskom jest dostępność metod akceleracji obliczeń, która warunkuje możliwości poprawy wydajności. W konsekwencji przekłada się to na wygodę użytkowania i możliwości prototypowania bardziej złożonych algorytmów dla problemów o większych rozmiarach. Przykładem takich są algorytmy uczenia maszynowego, w szczególności uczenia głębokiego.

Ostatnim z analizowanych czynników jest architektura systemu pakietów, która jest ściśle powiązana ze środowiskami uruchomieniowymi, które z nich korzystają. Pobierane pakiety zawierają biblioteki, które importowane są do projektu w postaci modułów. Python posiada jeden format modułów w przeciwieństwie do języka JavaScript ze względu na heterogeniczność środowisk jego wykonywania. Przeglądarka internetowa, NodeJS i Dino mimo, że wykonują kod w tym samym języku, posiadają znaczące różnice w sposobie komunikacji z użytkownikiem, systemem operacyjnym, w dostępności metod akceleracji i zarządzaniu modułami.

Obecnie jednak, z technicznego punktu widzenia, nie istnieją zasadnicze różnice pomiędzy środowiskami języków Python i JavaScript, które hamowałyby w znacznym stopniu rozwój bibliotek służącym do prototypowania i wdrażania aplikacji zajmującymi się obliczeniami numerycznymi w środowiskach języka JavaScript.

W tabeli 1.1 przedstawiono porównanie popularnych bibliotek stosowanych przy obliczeniach numerycznych języka Python z ich odpowiednikami w języku JavaScript. Widać wyraźnie dysproporcję w liczbie pobrań bibliotek pomiędzy językami. W wielu przypadkach autorowi nie udało się znaleźć dokładnego odpowiednika danej biblioteki, a znalezione zestawy bibliotek języka JavaScript nie pokrywają w całości funkcjonalności biblioteki języka Python.

1.2. Cel badań i zawartość pracy

Celem niniejszej pracy jest zbadanie przy-
stosowania środowisk języka JavaScript
do przeprowadzania obliczeń numerycznych
oraz wykonywania złożonych algorytmów.
Analizie poddano popularne środowiska -
NodeJS, Deno i przeglądarki internetowe
Google Chrome i Mozilla Firefox. Dla
tych środowisk zbadano dostępne metody
akceleracji obliczeń, gdzie pojęcie akcele-
racji rozumiane jest jako każda modyfika-
cja algorytmu wpływająca pozytywnie na
jego wydajność. Tyczy się to optymalizacji
wersji sekwencyjnej algorytmu bez zmiany
jego złożoności, jak i jego modyfikację do
wersji zrównoległej. Metody akcelera-
cji zaimplementowane dla badanych środo-
wisk przedstawione zostały w tabeli 1.2. Zo-
staną one opisane dokładnie w dalszej części
pracy. Analiza zorientowana jest na algorytmy analizy obrazów.

Na to złożone zagadnienie trzeba spojrzeć z wielu perspektyw. Autor pracy abstrahuje jednak od zagadnienia poprawy złożoności obliczeniowej samego algorytmu w wersji sekwencyjnej,

Tab. 1.2: Zaimplementowane metody akceleracji dla badanych środowisk.

	Chrome	Firefox	Node	Deno
Sequential	✓	✓	✓	✓
Native addon	✗ ¹	✗ ¹	✓	✗ ²
asm.js	✓	✓	✓	✗ ²
WASM	✓	✓	✓	✗ ²
WASM+SIMD	✓	✓	✓	✗ ²
Workers	✓	✓	✓	✓
WebGL	✓	✓	✗ ²	✗ ¹
WebGPU	✗ ³	✗ ³	✗ ¹	✗ ³

¹ Niedostępne w środowisku

² Wymaga zewnętrznych pakietów lub kodu bazowego
w języku innym niż C++

³ Niestabilne lub eksperymentalne

ponieważ rozważania takie wykraczają poza zakładaną w pracy tematykę, która koncentruje się na środowiskach języku JavaScript, możliwościach tych środowisk, jak i samego języka.

1.2.1. Perspektywa wydajności

Pierwszą z perspektyw jest wydajność algorytmu postrzegana przez pryzmat środowiska, w którym jest on wykonywany oraz wykorzystywanych przez niego metod akceleracji. W pracy zbadano wydajność zaimplementowanego algorytmu z wykorzystaniem metod wymienionych w tabeli 1.2, które szczegółowo omówione zostały w rozdziale 2.4.

1.2.2. Perspektywa kompatybilności i budowania bibliotek

Innym punktem widzenia jest kompatybilność budowanych bibliotek w wielu środowiskach. Każde z nich jest na tyle zróżnicowane, że zbudowanie jednej wersji kodu źródłowego kompatybilnego z nimi wszystkimi naraz jest wymagające, a niekiedy niemożliwe. Analiza tego zagadnienia, możliwości i ograniczeń modularności kodu języka JavaScript, dodatkowo w aspekcie wykorzystywanych metod akceleracji, opisana została w rozdziale 2.2.

1.2.3. Perspektywa wygody użytkowania

Celem autorów bibliotek jest dostarczenie rozwiązań, które przede wszystkim będą aktywnie wykorzystywane. Autor, razem z opisem implementacji algorytmów i ich późniejszego wykorzystania, w rozdziale 5 analizuje możliwości i sposoby interakcji ze środowiskami, wyświetlania wyników, ładowania danych oraz wewnętrzne mechanizmy przetwarzania danych związane również z modelem wykonania języka JavaScript.

Aspektem łączącym budowanie bibliotek, kompatybilność oraz wygodę użytkowania jest użycie języka TypeScript, nadzbioru języka JavaScript umożliwiającego wykorzystanie statycznego typowania oraz wszystkie zalety za tym idące. Są nimi między innymi dostępność typów generycznych, dekoratorów, czy zmiennych wyliczeniowych. W momencie budowania biblioteki, w procesie transpilacji kodu do języka JavaScript, do kodu dodane mogą być mechanizmy zapewniające kompatybilność ze starszymi wersjami języka oraz środowiskami.

1.2.4. Transformacja Hough'a jako wspólny mianownik analizy

W celu porównania wydajności pomiędzy środowiskami dla tej samej metody akceleracji oraz porównania ich w ramach jednego środowiska zaimplementowano algorytmy transformacji Hough'a (czyt. Hafa), która dokładniej opisana została w rozdziale 3. Jest to rodzina algorytmów deterministycznych wykorzystywana do detekcji kształtów parametrycznych i nieparametrycznych na obrazach. Własna implementacja algorytmów z tej rodziny, w przeciwieństwie do zestawów testowych takich jak na przykład Ostrich[11], daje możliwości granularnej kontroli nad samą implementacją i możliwości dostosowania jej do wszystkich analizowanych środowisk i metod akceleracji. Samo wykrywanie kształtów na podstawie transformacji Hough'a jest intensywne obliczeniowo, wieloetapowe, wykorzystuje operacje zmienoprzecinkowe, w tym funkcje trygonometryczne, oraz zakłada wykonanie wielu iteracji po dużych, zależnych od rozmiaru problemu i próbkowania, strukturach danych. Czynniki te czynią tę rodzinę algorytmów, zdaniem autora, dobrym punktem zaczepienia podczas szerokiej analizy środowisk i metod akceleracji, które one udostępniają. Pozwolić to może uzyskać ogólną orientację w analizowanych perspektywach oraz wskazać wartościowe kierunki przyszłych badań.

1.2.5. Zawartość pracy

Rozdział drugi opisuje język JavaScript, wprowadza w zagadnienie jego modelu wykonania, środowisk oraz opisuje dostępne dla nich metody akceleracji i sposoby budowania bibliotek. Wspólny mianownik analizy, czyli algorytmy transformacji Hough'a i sposób ich działania opisany został w rozdziale trzecim. Rozdział czwarty opisuje metodykę przeprowadzanych testów wydajności, dla potrzeb których zaimplementowana została osobna biblioteka odpowiedzialna za pomiary czasu wykonania, mając na uwadze jej kompatybilność ze wszystkimi środowiskami. Implementację algorytmów i konsekwencje idące za stosowaniem poszczególnych rozwiązań opisuje rozdział piąty. Rozdział szósty przedstawia rezultaty testów wydajności ze wskazaniem na porównanie środowisk i metod z bazowym wariantem sekwencyjnego wykonania algorytmu, a rozdział siódmy stanowi podsumowanie całości rozważań i eksperymentów oraz wskazuje zidentyfikowane problemy i proponowane kierunki przyszłych badań.

Rozdział 2

Język JavaScript

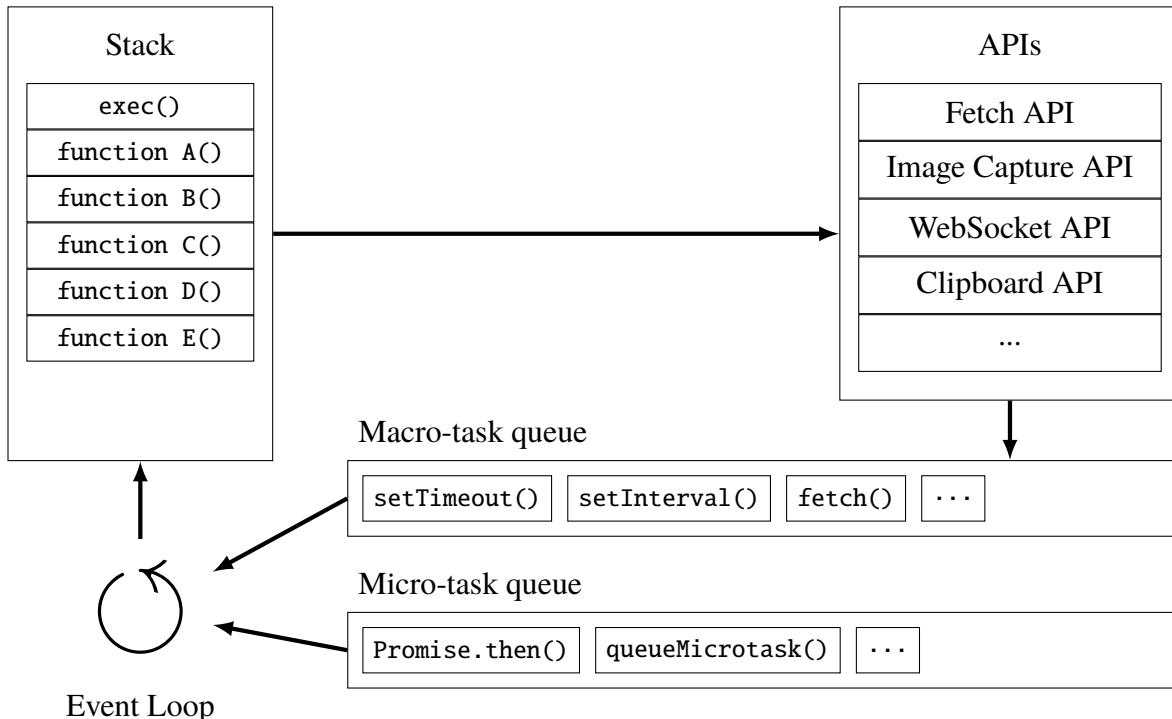
JavaScript od momentu swojego powstania w 1995 roku stanowi jeden z filarów rozwoju technologii webowych, zaczynając od dodania prostych mechanizmów interaktywności do statycznych stron internetowych, a kończąc na byciu nieraz jedynym samodzielnym budulcem pełnowymiarowych aplikacji działających po stronie klienta i serwera, aplikacji działających w środowisku przeglądarki internetowej, ale też w środowiskach natywnych, desktopowych i mobilnych. Dlatego, aby zrozumieć w pełni specyfikę problemu, który stanowi przystosowanie języka do wykonywania obliczeń numerycznych, w tym rozdziale przybliżone zostały zagadnienia związane z modelem wykonania, środowiskami oraz sposobami na podział kodu na moduły i późniejsze ich wykorzystanie. Na końcu opisane zostały metody akceleracji, dla których przeprowadzono badania.

2.1. Model wykonania

Model wykonania języka JavaScript skoncentrowany jest w głównej mierze na obsłudze zdarzeń. W przeglądarce internetowej zdarzeniami takimi mogą być interakcje z użytkownikiem, na przykład kiedy naciśnięty zostanie przycisk, albo interakcje z siecią, kiedy otrzymamy odpowiedź na zapytanie z wykorzystaniem obiektu XMLHttpRequest lub skorzystamy z Fetch API. Po stronie serwera zdarzeniami takimi mogą być odebranie zapytania, które serwer musi obsłużyć, obsługa strumieni, ale także wszelkie odpowiedzi na interakcje z systemem operacyjnym. Podstawowymi interakcjami mogą być obsługa sygnałów, dostęp do plików, czy też obsługa sieci, która umożliwia połączenie na przykład z bazą danych.

Zdarzenia te obsługuje pętla zdarzeń (ang. event loop). Na rysunku 2.1 pokazano jej uproszczony model. Wyróżnia ona zadania, zwane także makro zadaniami, oraz mikro zadania. Dla każdego typu zadań utworzona zostaje osobna kolejka. Jeśli aktualnie wykonywane przetwarzanie sekwencyjne, którego ramki wywołań śledzone są na stosie, zakończy się, wtedy z pętli zdarzeń pobierane i wykonywane jest makro lub mikro zadanie. W pierwszej kolejności wykonywane są wszystkie mikro zadania, a gdy ich kolejka jest pusta, wykonywane jest kolejne makro zadanie.

Makro zadania dodawane są do kolejki, aby obsługiwać wspomniane już zdarzenia związane z działaniami użytkownika lub inne zewnętrzne zdarzenia. Są one również dodawane do kolejki, kiedy mija czas zadany podczas wywołań funkcji `setTimeout()` oraz `setInterval()`. Warto zaznaczyć, że wywołania tych funkcji nie gwarantują wykonania dokładnie po zadanym czasie, ale traktują go jako próg czasowy, po jakim zadana funkcja zostanie dodana do kolejki makro zadań [56]. Makro zadania dodane podczas jednej iteracji pętli nigdy nie zostaną wykonane w tej samej iteracji.



Rys. 2.1: Uproszczony model pętli zdarzeń środowisk języka JavaScript w wariancie z wyróżnieniem API przeglądarek internetowych.

Mikro zadania pochodzą tylko i wyłącznie z kodu użytkownika lub bibliotek i wykorzystywane są do obsługi asynchronicznych zadań, zarządzania ich kolejnością i obsługą błędów [38]. Do ich tworzenia wykorzystuje się głównie obiekt `Promise` reprezentujące zadanie, które w przyszłości zakończy się pomyślnie lub błędem. Dla takiego obiektu zdefiniować możemy funkcje, które wykonają się podczas scenariusza pomyślnego (`then`), błędnego (`catch`) oraz zawsze (`finally`) [62]. Aby uprościć z tymi obiekty, jako alternatywę do stosowania tych metod, utworzono wyrażenia wykorzystujące słowa kluczowe `async/await`. Mikro zadania pochodzić mogą również od obserwatorów na przykład `MutationObserver`, czy `ResizeObserver`.

Zrozumienie sposobu wykonywania kodu w asynchronicznym modelu języka JavaScript jest kluczowe do efektywnego wykorzystania możliwości, jakie idą za metodami akceleracji, których użycie możliwe jest tylko i wyłącznie poprzez asynchroniczne wywołania. Opisane w sekcji 2.4 metody bazujące na Worker'ach oraz WebGL wymagają interakcji poprzez wywołania asynchroniczne. Na listingu 2.1 pokazano przykład kodu asynchronicznego. Wywołania `console.log` wykonają się, zawsze drukując liczby w kolejności od 1 do 6.

Listing 2.1: Przykład kodu demonstrujący mechanizmy asynchroniczności w języku JavaScript.

```

1 console.log(1);
2 Promise.resolve().then(() => setTimeout(() => console.log(6)));
3 setTimeout(() => console.log(4), 0);
4 setTimeout(() => Promise.resolve().then(() => console.log(5)));
5 Promise.resolve().then(() => console.log(3));
6 console.log(2);

```

Linijki 1 oraz 6 zostają wykonane synchronicznie. `Promise` w linijce 5 zostaje wykonany jako następny, ponieważ jako mikro zadanie, wykona się zaraz po operacjach synchronicznych. Następnie funkcje `setTimeout` wykonają się w kolejności ich wywołania, gdzie w linijkach 2-4 na ich kolejność wpływ mają mechanizm `Promise`. Timeout z linijki 3, a potem 4 zostają wywołane jaki pierwsze. Jako ostatni wykonuje się timeout z linijki 2, ponieważ jego wywołanie, w postaci mikro zadania, przeniesione zostało na koniec wykonania synchronicznego.

2.2. Środowiska JavaScript

Rosnąca popularność języka JavaScript i idących za jego stosowaniem możliwości przyspieszyła rozwój środowisk, w których kod języka mógł być wykonywany. Pierwszym z nich była przeglądarka internetowa.

2.2.1. Przeglądarka internetowa

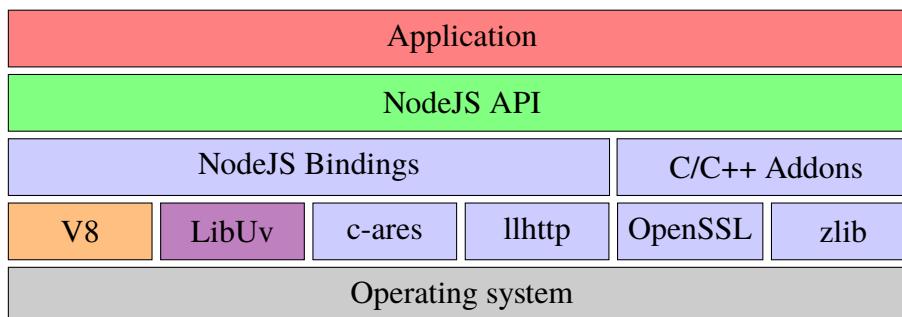
Głównym zadaniem przeglądarki internetowej jest pobieranie z sieci i wyświetlanie zawartości użytkownikowi oraz obsługa jego interakcji. Najważniejszym komponentem przeglądarki jest silnik renderujący, który zawiera między innymi silniki odpowiedzialne za parsowanie i renderowanie struktury modelu DOM, adaptery służące do wywołań dostępnych bibliotek graficznych (OpenGL, Vulcan, DirectX) oraz silnik JavaScript. Z perspektywy problemu stawianego w tej pracy to właśnie silnik JavaScript jest najważniejszym komponentem silnika renderującego. Silnik języka zajmuje się parsowaniem, komplikacją do bytecode'u, interpretowaniem oraz późniejszą optymalizacją kodu. Posiada wiele możliwości optymalizacji spekulatywnej z racji na specyfikę języka, który jest dynamicznie typowany [14].

Popularnymi silnikami renderującymi są Blink z silnikiem JavaScript V8 [63] oraz Gecko z silnikiem JavaScript SpiderMonkey [57]. Silniki JavaScript skupione są na szybkim i efektywnym wykonywaniu kodu i nie zajmują się asynchronicznością i pętlą zdarzeń. Odpowiedzialne za to są równolegle biblioteki. W przeglądarce Google Chrome pętlę zdarzeń implementuje biblioteka LibEvent [42].

2.2.2. NodeJS

NodeJS jest powstały w 2009 roku środowiskiem, które było odpowiedzią na architekturę pozostałych rozwiązań serwerowych, które angażują wiele procesów i wątków do obsługi wielu zapytań w tym samym czasie. Rodzi to problemy związane z koniecznością przełączania kontekstu pomiędzy procesami oraz większe zapotrzebowanie na pamięć. Również każda operacja wejścia-wyjścia musi być synchroniczna, co prowadzi do zablokowania całego procesu w oczekiwaniu na odpowiedź [2].

Problemy te rozwiązało środowisko NodeJS, napisane w C++ i oparte na silniku V8. Razem z biblioteką LibUv [43] implementującą pętlę zdarzeń w ramach jednego procesu wykonującą kod JavaScript użytkownika oraz wielu wątków, które realizują oczekiwanie na operacje asynchroniczne, pozwoliło rozwiązać problem operacji wejścia-wyjścia umożliwiając w ramach tych samych zasobów sprzętowych osiągnąć lepszą wydajność niż popularny serwer Apache [5]. Na rysunku 2.2 pokazano architekturę środowiska NodeJS, która stoi pomiędzy aplikacją, a systemem operacyjnym.



Rys. 2.2: Model architektury środowiska NodeJS.

2.2.3. Deno

Deno powstał w 2018 roku, a jego wersja 1.0.0 wydana została w 2020 roku. Jest to środowisko aspirujące do bycia następcą NodeJS rozwiązujejąc jego problemy związane z bezpieczeństwem, systemem budowania zależności bibliotek, czy importowania zależności [1]. Podobnie jak NodeJS do wykonywania kodu JavaScript wykorzystuje silnik V8, ale napisany jest w języku Rust. Do obsługi asynchroniczności i pętli zdarzeń wykorzystuje bibliotekę Tokio [61]. W przeciwieństwie do NodeJS i przeglądarek internetowych obsługuje natywnie TypeScript.

2.3. Modularność i kompatybilność kodu

Kolejnym ważnym aspektem rozważań jest modularność i kompatybilność kodu pomiędzy środowiskami. Rozwój bibliotek języków jest naturalnym krokiem ewolucji ich ekosystemów i aby taki ekosystem był ogólnodostępny, biblioteki dostępne są dla wszystkich w postaci scentralizowanego rejestru pakietów.

2.3.1. Modularność

Dla środowiska NodeJS najpopularniejszym rejestrem jest *npm registry* (node package manager). Za pomocą narzędzia o tej samej nazwie można instalować i zatraskiwać wersję pakietów, które trafiają do katalogu `node_modules`, w którym środowisko NodeJS domyślnie poszukuje kodu podczas importu pakietów. Zainstalowane pakiety są opisane wraz z ich wersją w pliku `package.json` i `package-lock.json`, gdzie pierwszy z nich zawiera wymaganą wersję zapisaną w konwencji Semver [55], a drugi zatrzaśnięte zainstalowane wersje, co pozwala odtworzyć dokładną strukturę zależności.

Przeglądarki internetowe z kolei pobierają dodatkowe biblioteki poprzez umieszczenie tagów `<script/>` w dowolnym miejscu na stronie, często z użyciem sieci CDN (ang. content delivery network). Rejestr *npm* jest również popularnym rozwiązaniem służącym do dostarczania modułów niezbędnych do funkcjonowania stronom internetowym, jednak odbywa się to pośrednio. Za pomocą narzędzi zwanych bundler'ami, ze wszystkich niezbędnych zależności - właściwego kodu strony oraz zewnętrznych bibliotek instalowanych przy pomocy narzędzia *npm*, budowany jest pojedynczy plik, który następnie jest ładowany przez przeglądarkę. Pomaga to zaoszczędzić liczbę połączeń przeglądarki do serwerów, a co za tym idzie zaoszczędzić czas spędzony na inicjowaniu połączenia i pobieraniu danych. Znaczenie ma to w szczególności, że liczba możliwych otwartych połączeń przez przeglądarkę internetową jest limitowana (10 w Google Chrome). Często obecnie budowane są dwa lub więcej plików - jeden z bibliotekami zewnętrznymi oraz jeden lub więcej z właściwym kodem strony w celu wykorzystania mechanizmów pamięci podręcznej przeglądarki oraz dynamicznego i opóźnionego ładowania niezbędnego kodu, co przyspiesza ładowanie strony. Popularnymi bundler'ami są Webpack, Snowpack, Parcel, Rollup oraz Vite.

Deno próbuje rozwiązać problem importowania modułów w NodeJS, który wynika z scentralizowanego rejestru pakietów oraz z faktu istnienia plików `package*.json` i konieczności wykonania procesu instalacji. Pobiera on zależności bezpośrednio z sieci i umieszcza w pamięci podręcznej. Link do zależności jest jej jednoznacznym identyfikatorem, a to znaczy, że powinien zawierać wersję pakietu i nigdy nie zmienić swojej zawartości. Pobranie i komplikacja zależności w czasie wykonania programu likwiduje potrzebę przechowywania listy zależności i ich wersji oraz niweluje potrzebę ich instalacji. Deno wykorzystuje opisany w sekcji 2.3.1 format modułów ESM.

Niezależnie od tego gdzie przechowywane są moduły oraz w jaki sposób są zarządzane przez środowisko, muszą one być finalnie przez nie skonsumowane. Mogą być one użyte bezpośred-

nio, ale też przetransformowane w procesie budowania biblioteki, która będzie potem dalej konsumowana, czy pakietu dla przeglądarki internetowej. W procesie ewolucji ekosystemu języka JavaScript wykształciło się wiele formatów modułów. Niektóre z nich są kompatybilne tylko z przeglądarką internetową, niektóre tylko ze środowiskiem NodeJS bądź Deno.

AMD

AMD (Asynchronous Module Definition) jest sposobem ładowania zależności w przeglądarkach internetowych. Rozwija wzorzec modułów JavaScript [39] poprzez dodanie asynchronicznego pobierania i ładowania zależności. Moduł jest funkcją, dzięki czemu zadeklarowane zmienne nie wyciekają poza jej zakres, a jej wartość zwracana stanowi wartość, którą taki moduł eksportuje. Zadeklarowanie modułu i jego zależności w formacie AMD umożliwia funkcja `define`.

CommonJS

Format CommonJS utworzony został na potrzeby środowiska NodeJS i jest tam do dzisiaj wykorzystywany. Używa on globalnie dostępnej funkcji `require`, która jako argument przyjmuje nazwę modułu lub relatywną ścieżkę do pliku `*.js`, jednak z pominięciem jego rozszerzenia, co stanowi problem podczas wyszukiwania modułów przez środowisko. Moduł może eksportować funkcje i wartości poprzez dodanie ich do obiektu `module.exports`. Pliki z modułami w tym formacie, aby lepiej je identyfikować, mogą mieć rozszerzenie `*.cjs`.

UMD

UMD (Universal Module Definition) nie stanowi samodzielnego formatu, ale integruje formaty AMD, CommonJS oraz użycie zmiennych globalnych do definicji modułu i jego zależności. Wyewoluował on z potrzeby tworzenia bibliotek kompatybilnych z wieloma środowiskami, dla których nie trzeba budować wielu wersji w różnych formatach.

Moduły ECMAScript

Brak kompatybilności i wiele formatów modułów, gdzie każde środowisko zaproponowało swój własny, wymusiło ich standaryzację w specyfikacji języka. ECMAScript, którego implementacją jest JavaScript, w wersji 2015 (zwanej również ES6) wprowadza definicję modułów zwanych ESMODules, ESM [31]. Obecnie wspierane są one przez wszystkie analizowane tutaj środowiska i są zalecaną metodą importowania zależności. Używają one słów kluczowych `import` oraz `export` tak, jak zostało to pokazane na listingu 2.2. Od wersji ES11 specyfikacji możliwe stało się dynamiczne importowanie modułów podczas wykonania, gdzie jako rezultat otrzymujemy obiekt `Promise`. Pliki z modułami w tym formacie, aby lepiej je identyfikować, mogą mieć rozszerzenie `*.mjs`. Wszystkie biblioteki wykorzystujące metody akceleracji badane w tej pracy budowane są z wykorzystaniem ESM.

Listing 2.2: Przykład wykorzystania ECMAScript Modules

```

1 // main.mjs
2 import { add } from "./module"
3 console.log(add(2+2));
4
5 // module.mjs
6 export function add(foo, bar) { return foo + bar; }

```

2.3.2. Kompatybilność

Szerokie starania w standaryzacji modułów umożliwiają tworzenie kodu kompatybilnego z wieleoma środowiskami. Jeśli jednak kod ten korzysta z funkcjonalności samego środowiska, która istnieje w pozostałych środowiskach, ale ich API nie są ze sobą zgodne, problem kompatybilności między środowiskami wciąż występuje. Wprowadza to niechciane mechanizmy do kodu wykrywające środowisko i wymusza wykorzystanie wzorców projektowych takich jak *adapter* w celu obsługi wszystkich wariantów API.

Przykładem takiego rozwiązania jest biblioteka *axios*, która służy do wykonywania zapytań HTTP. W środowisku przeglądarki internetowej do wykonywania zapytań wykorzystuje obiekt `XMLHttpRequest`, a w środowisku NodeJS wbudowany moduł `http`. Rozwiązaniem problemu w tym przypadku może być użycie Fetch API, które jako pierwsze zadebiutowało w przeglądarkach internetowych oraz zaadaptowane zostało przez NodeJS, a w Deno jest ono domyślnie przewidzianą formą wykonywania zapytań HTTP.

Innym przykładem braku kompatybilności pomiędzy podobnymi funkcjonalnościami jest wielowątkowość, która istnieje pod abstrakcją Worker'ów i dokładnie zostanie omówiona w rozdziale 2.4. Przeglądarki internetowe oraz Deno, który ma na celu możliwie zbliżyć się do nich ze swoim API, implementują Web Worker API. NodeJS z kolei do obsługi Worker'ów wykorzystuje wbudowany moduł `worker_threads`, który różni się od jego odpowiedników w pozostałych środowiskach.

2.4. Metody akceleracji

Akceleracja obliczeń jest niekończącą się pogonią za nieskończenie krótkim czasem wykonania algorytmów. Zdaniem autora powinna być ona brana pod uwagę na każdym etapie ich rozwoju - od etapu prototypowania, do wdrożeń produkcyjnych, ponieważ na każdym z nich może przynieść wymierne korzyści. Wspomnianą w poprzednim rozdziale akcelerację praca traktuje jako wszelkie metody przyspieszające wykonywanie algorytmu.

2.4.1. Optymalizacja wykonania sekwencyjnego

Pierwszym aspektem, na który trzeba zwrócić uwagę jest sama interakcja z silnikiem języka i wykorzystanie jego możliwości oraz tego, w jaki sposób potrafi on optymalizować wykonywany kod. Poprzez komplikację kodu Just-In-Time (JIT), czyli bezpośrednio przed jego wykonaniem oraz możliwość powtarzania tego procesu wykorzystując heurystyki dla zebranych danych, możliwe jest przyspieszenie wykonania kodu [14]. W przypadku silnika V8 kod najpierw trafia do interpretera o nazwie Ignition, gdzie kompilowany jest do bytecode'u, który zamieniany jest na instrukcje zgodne z architekturą procesora. Równolegle bytecode wysyłany jest do kompilatora TurboFan, gdzie przechodzi optymalizację na poziomie funkcji, w przeciwieństwie do innych rozwiązań JIT, które identyfikują wielokrotnie wykonywaną część bytecode'u [15]. Optymalizacja ta jest ograniczona do funkcji o maksymalnym rozmiarze bytecode'u równym 60KB, więc ważne jest, aby umiejętnie rozbijać kod algorytmu na funkcje i moduły. W przeszłości podjęto również próby wykorzystania spekulacyjnego zrównoleglania kodu po stronie silnika JavaScript (Thread-level Speculation). Przyniosło to dobre rezultaty, jednak ta metoda optymalizacji nie jest implementowana w popularnych silnikach [13].

Osoba projektująca dany algorytm może celowo unikać konstrukcji języka oraz niektórych wzorców, aby wspomóc mechanizmy optymalizacji i zwiększyć wydajność w skrajnych przypadkach nawet o 25% ([7], [22]). Dobrą praktyką jest stosowanie typowanych tablic (Typed Array) do przechowywania danych binarnych oraz danych o znanym typie. Innym sposobem poprawny wydajności jest modyfikacja algorytmu tak, aby zredukować obciążenie związane

z obliczeniami zmiennoprzecinkowymi, na przykład poprzez stosowanie stablicowanych wartości funkcji trygonometrycznych (LUT).

2.4.2. Natywne rozszerzenia

Środowiska NodeJS i Deno działające po stronie serwera pozwalają na uruchomienie z ich poziomu bibliotek skompilowanych do kodu maszynowego konkretnej architektury. Deno pozwala uruchomić funkcję ze skompilowanego kodu języka Rust do postaci biblioteki na różnych platformach (*.so, *.dll, *.dylib). NodeJS natomiast posiada własny system budowania natywnych rozszerzeń z kodu C++ o nazwie *node-gyp*, którego zależnością jest język Python.

Zaletą korzystania z natywnych rozszerzeń jest możliwość wykorzystania metod optymalizacji specyficznych dla platformy sprzętowej, jakie oferują języki niskiego poziomu kompilowane do kodu maszynowego. Metodę tę można rozwinać o metody akceleracji dostępne w językach źródłowych, których przykładem może być wielowątkowość z wykorzystaniem biblioteki *pthreads*. Do wad takiego rozwiązania zaliczyć można konieczność pobierania lub budowania natywnych zależności w momencie instalacji biblioteki oraz ogólną kompatybilność, która wykracza poza środowisko języka JavaScript. Wadą, która wpływa w znaczącym stopniu na wydajność w specyficznych przypadkach jest brak bezpośredniego dostępu do pamięci silnika JavaScript, co skutkuje koniecznością kopiowania danych w warstwie powiązania natywnego rozszerzenia z kodem języka JavaScript. Dla dużych danych proces ten okazać się może wąskim gardłem algorytmu.

2.4.3. WebAssembly

Kolejną metodą akceleracji wykonania sekwencyjnego jest WebAssembly (WASM), który jest językiem niskiego poziomu. Stanowi on cel komplikacji języków takich jak C++ czy Rust i pozwala uruchamiać w środowisku webowym złożone aplikacje, których wcześniejsze uruchomienie nie było możliwe ze względu na konieczność parsowania i komplikacji dużej ilości kodu. Umożliwiło to na przykład łatwe przeniesienie aplikacji napisanych w języku C++ z wykorzystaniem biblioteki Qt i uruchomienie ich w przeglądarce internetowej [53]. Kolejnym przykładem jest jeden z backendów biblioteki Tensorflow.js [60], która umożliwia trenowanie i wdrażanie modeli uczenia maszynowego. Dzięki bibliotece PyScript, która portuje implementację interpretera języka Python napisaną w C do języka WebAssembly, możliwe stało się wykonywanie kodu tego języka w przeglądarce internetowej [52].

Listing 2.3: Funkcja licząca silnię w języku C/C++

```

1 int factorial(int n) {
2     if (n == 0)
3         return 1;
4     else
5         return n * factorial(n-1);
6 }
```

Wydajność kodu WebAssembly z założenia powinna być zbliżona do wykonania natywnego. Moduły operują na liniowym modelu danych, który nie jest współdzielony z kodem języka JavaScript. Podobnie jak w przypadku natywnych rozszerzeń występuje konieczność transformacji i kopiowania danych do pamięci modułu, co może rodzić problemy z wydajnością. Na listingu 2.3 przedstawiono funkcję liczącą silnię, która na listingu 2.4 została zapisana w jej odpowiedniku w WASM. WebAssembly przetwarzany jest w postaci binarnej i działa jak maszyna stosowa, a na potrzeby analizy zapisywany jest w formacie tekstowym w postaci S-wyrażeń.

Listing 2.4: Funkcja licząca silnię w języku WASM

```
1 (func (param i64) (result i64)
2     local.get 0
3     i64.eqz
4     if (result i64)
5         i64.const 1
6     else
7         local.get 0
8         local.get 0
9         i64.const 1
10        i64.sub
11        call 0
12        i64.mul
13    end)
```

asm.js

Poprzednikiem WebAssembly był asm.js - podzbiór języka JavaScript w procesie komplikacji z języków źródłowych zoptymalizowany pod kątem wydajności wykonania i specjalnie interpretowany przez przeglądarki wykorzystując komplikację Ahead-Of-Time. Kompilację tę wspiera do dziś jedynie przeglądarka Mozilla Firefox, jednak optymalizacje silnika V8 Google Chrome 28 zapewniły dwukrotny wzrost wydajności podczas wykonania asm.js [30]. Nie jest on już rozwijany i został wyparty przez WebAssembly.

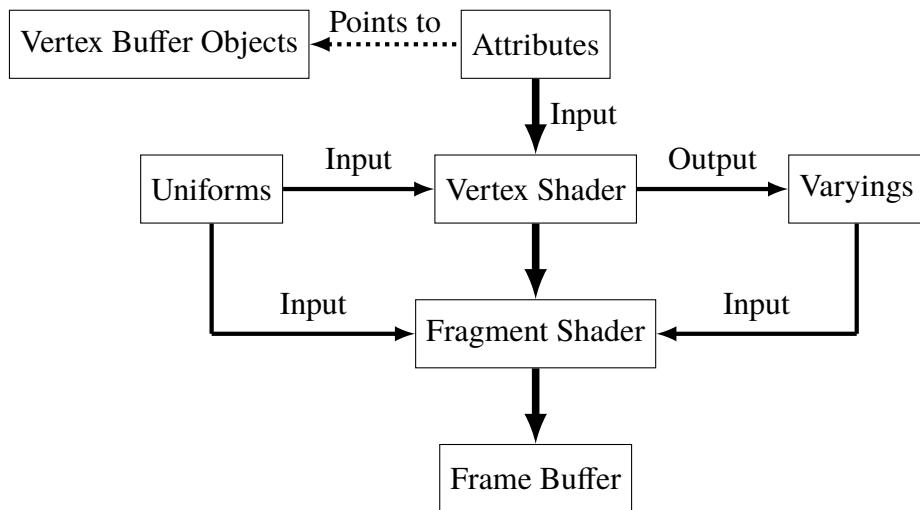
SIMD

WebAssembly jest w stanie wykorzystać architekturę SIMD (Single Instruction, Multiple Data), używając rejestrów i instrukcji wektorowych [33]. Kompilatory takie jak LLVM potrafią w procesie auto-wektoryzacji wykorzystać instrukcje wektorowe, aby przyspieszyć działanie pętli oraz połączyć inne masowe operacje logiczne i arytmetyczne. Wektoryzację taką można również przeprowadzić ręcznie wykorzystując funkcje, które operują bezpośrednio na typach wektorowych.

2.4.4. Współbieżność

W środowisku przeglądarki internetowej współbieżność osiągnięta może być tylko na poziomie wielu wątków, poprzez zastosowanie Worker'ów. Każdy Worker funkcjonuje jako osobny wątek w ramach jednego procesu. Posiada on własną pętlę zdarzeń i z głównym wątkiem komunikuje się jedynie asynchronicznymi wiadomościami. Poza obiektem SharedArrayBuffer, wspieranego przez interfejs operacji atomowych `Atomic`, aby zapobiec problemowi wyścigów, Worker'y nie współdzielą pamięci. Dane w wiadomościach przekazywane są z użyciem algorytmu klonowania strukturalnego, który wspiera natywne typy takie jak tablice, mapy, czy sety [59]. Nie możliwe jest natomiast kopiowanie funkcji zdefiniowanych przez użytkownika oraz węzłów drzewa DOM, ponieważ tylko wątek główny może być odpowiedzialny za renderowanie widoku strony. Aby uniknąć procesu klonowania dużych zmiennych, które chcemy przenieść do Worker'a, i które nie są potrzebne nam w wątku głównym, możemy użyć obiektów `Transferrable`, które jeśli wskazane, zostają przeniesione do kontekstu Worker'a, a nie skopowane, oszczędzając w ten sposób pamięć i czas procesora.

W środowiskach działających po stronie serwera współprzejne wykonanie możliwe jest przy zastosowaniu Worker'ów, ale również poprzez uruchomienie podprocesów, z którymi można komunikować się poprzez standardowe strumienie wejścia oraz wyjścia, a w przypadku NodeJS,



Rys. 2.3: Potok graficzny WebGL API, który może być wykorzystany do obliczeń ogólnego przeznaczenia.

również poprzez mechanizm wiadomości, który zajmuje się ich automatyczną serializacją i parsowaniem.

2.4.5. GPGPU

GPGPU (ang. General-purpose computing on graphics processing units) zakłada użycie układów graficznych do wykonywania obliczeń ogólnego przeznaczenia, które do tej pory wykonywane były na CPU. W środowiskach desktopowych, możemy wykorzystać takie układy używając rozwiązań powiązanych z ich architekturą, czego przykładem jest NVIDIA CUDA. Możemy również skorzystać z framework'ów implementujących warstwę abstrakcji będąc kompatybilnymi z wieloma platformami i architekturami jak na przykład OpenCL.

Środowiska webowe zorientowane są na jak największą kompatybilność pomiędzy platformami, a pierwotnie interakcja z układami graficznymi możliwa była tylko w środowisku przeglądarki internetowej poprzez wykorzystanie WebGL API do generowania grafiki w elemencie `<canvas>`. Dla obliczeń ogólnego przeznaczenia stworzono standard WebCL, jednak nie zyskał on popularności i nie był powszechnie implementowany. Sposobem, który okazał się skuteczny, aby użyć układ graficzny do innych rzeczy niż generowanie grafiki, paradoksalnie okazało się wykorzystanie procesów odpowiedzialnych za generowanie grafiki [21]. Każdy piksel generowanego obrazu stanowić może pojedynczy wynik działania kernela czyli funkcji, której działanie jest masowo zrównoległe. Dostarczenie danych wejściowych w postaci tekstur oraz konstrukcja programu Fragment Shader wyliczającego kolor każdego wynikowego piksela pozwala wykonać obliczenia w takiej samej abstrakcji jak dedykowane rozwiązania.

Na rysunku 2.3 przedstawiono najważniejsze elementy potoku graficznego WebGL API. Na podstawie atrybutów, które wskazują na bufory z danymi wierzchołków, program *Vertex Shader* dla każdego z nich oblicza ich współrzędne. Następnie po procesie transformacji wierzchołków na prymitywy (najczęściej trójkąty) i ich rasteryzacji, program *Fragment Shader* zajmuje się obliczeniem koloru każdego piksela wynikowego obrazu na podstawie interpolowanych wartości dla wierzchołków dostarczanych w postaci *Varyings*. Omijając etap związany z pozycją wierzchołków i wymuszając tylko kolorowanie właściwej liczby pikseli, możemy przenieść obliczenia do programu *Fragment Shader*, gdzie dane wejściowe dostarczane są w postaci tekstur za pomocą *Uniforms*, czyli stałych dla całego potoku.

Podejście to, z racji na specyficzność tego potoku, ma swoje ograniczenia. W przeciwieństwie do pozostałych rozwiązań GPGPU jedynym wynikiem działania kernela jest wartość pik-

selą. Niemożliwe zatem jest zapis do pamięci współdzielonej w trakcie wykonywania algorytmu. Wąskie gardło wydajności stanowi odczyt wyników. Proces wysłania bufora ramki i wyświetlenia go na elemencie <canvas/> jest zoptymalizowany, jednak pobranie go z GPU do postaci typowanej tablicy w języku JavaScript jest już kosztowne czasowo.

W środowisku przeglądarki internetowej WebGL jest dostępny , a co za tym idzie, istnieje możliwość wykorzystanie tej metody akceleracji obliczeń. Środowiska NodeJS oraz Deno, jako rozwiązania serwerowe, nie skupiły się na mechanizmach generowania grafiki. W środowisku NodeJS istnieją jednak biblioteki implementujące kontekst WebGL, na przykład *headless-gl* [36]. Środowisko Deno, z racji na swój młody wiek, na chwilę obecną nie posiada implementacji natywnej, jak i w postaci bibliotek.

WebGPU

Popularyzacja i potrzeba tworzenia coraz bardziej wydajnych aplikacji webowych spowodowała powstanie specyfikacji WebGPU API, która stanowi uogólnienie przetwarzania masowo równoległego dostarczając bezpośrednio abstrakcję GPU [64]. WebGPU może być wykorzystane do obliczeń ogólnego przeznaczenia, ale również do generowania grafiki. Obecnie jest jednak wciąż dostępne jako funkcjonalność eksperymentalna i do działania w środowisku przeglądarki internetowej oraz Deno potrzebuje specjalnej flagi. Środowisko NodeJS nie implementuje WebGPU.

Rozdział 3

Transformacja Hough'a

Transformacja Hough'a (czyt. Hafa) wykorzystywana jest w procesie analizy obrazów i służy do wykrywania na nim kształtów parametrycznych oraz nieparametrycznych w zależności od jej wariantu [17]. Samo pojęcie transformacji odnosi się do odwzorowywania pojedynczych pikseli obrazu binarnego lub ich zbioru w procesie głosowania w przestrzeni akumulatora. Obraz wejściowy wcześniej poddany być musi procesowi wykrywania krawędzi. Dane zebrane w akumulatorze biorą następnie udział w procesie, w którym wyłonione zostają potencjalne kształty poprzez wykrywanie największych wartości w akumulatorze. W zależności od specyfiki problemu oraz wykrywanych kształtów wykrywanie maksimów może odbywać się na różne sposoby. Użyte może zostać proste progowanie, wykrywanie i uśrednianie skupisk, czy też filtracja przestrzeni akumulatora. Ogólny schemat przetwarzania przedstawiony jest na rysunku 3.1.

3.1. Standard Hough Transform

Pracą, która jako pierwsza opisała tę transformację jest zgłoszony w 1962r. patent Paula Hough'a [8]. Opisał on wykrywanie linii poprzez zastosowanie odwzorowania PTLM (point-to-line mapping). Odwzorowanie to dla każdego piksela rysuje linię w dwuwymiarowej przestrzeni akumulatora zgodnie z kierunkowym równaniem prostej z równania (3.1) przekształcone do postaci (3.2). Stosując odwzorowanie odwrotne dla punktów o największych wartościach możemy otrzymać potencjalne linie na obrazie. Transformację stosującą pełne odwzorowanie wszystkich punktów obrazu na parametry kształtów nazywamy standardową transformacją Hough'a (Standard Hough Transform, SHT).

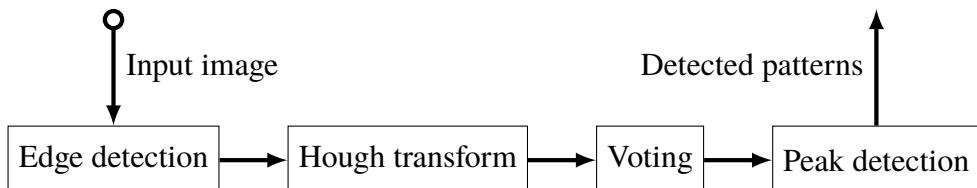
$$y(x) = mx + c \quad (3.1)$$

$$c(m) = -xm + y \quad (3.2)$$

gdzie: x, y — współrzędne piksela na obrazie;

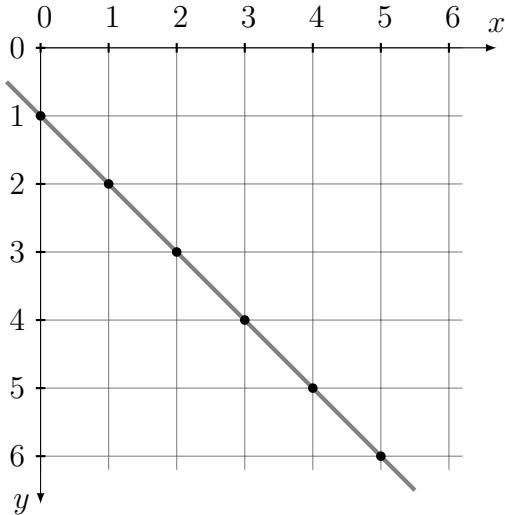
m — zbocze prostej;

c — punkt przecięcia prostej z osią Y.

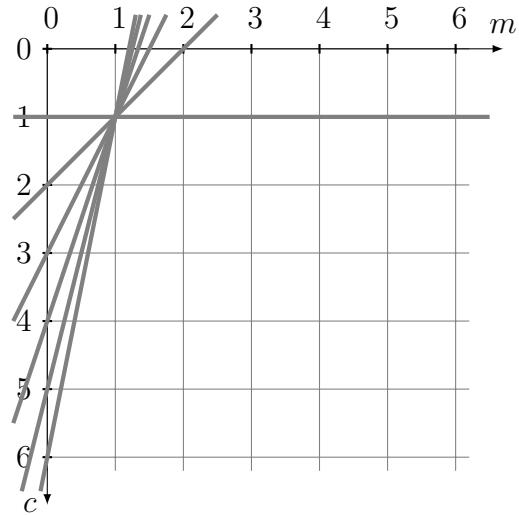


Rys. 3.1: Ogólny schemat przetwarzania obrazu z wykorzystaniem transformacji Hough'a.

3. Transformacja Hough'a



(a) Binarny obraz wejściowy



(b) Wynik transformacji

Rys. 3.2: Demonstracja transformacji Hough'a w wariancie równania kierunkowego prostej.

Na rysunku 3.2 przedstawiono obraz wejściowy (rys. 3.2a) oraz wynik transformacji (rys. 3.2b). Na rysunkach reprezentujących obraz oś Y jest skierowana w dół, co ułatwia interpretację w zgodności ze sposobem indeksowania pikseli obrazów podczas ich przetwarzania, gdzie punkt $(0, 0)$ znajduje się w lewym górnym rogu. W wyniku transformacji każdy jasny piksel obrazu (x, y) został odwzorowany na linię zgodnie z równaniem (3.2). Linie te przecięły się w jednym punkcie $(1, 1)$. W wariancie dyskretnym transformacji wartość akumulatora w punkcie $(1, 1)$ miałaby największą wartość. W tym wypadku punkt $(1, 1)$ akumulatora przekłada się na prostą o równaniu $y = x + 1$, co zgadza się z prostą na rysunku 3.2a.

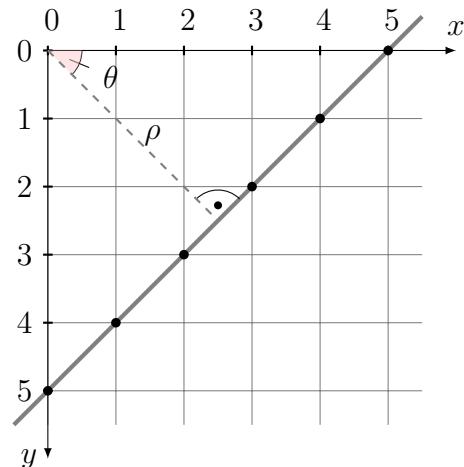
Taka reprezentacja punktu w przestrzeni akumulatora rodzi jednak problem w przypadku wykrywania linii pionowych. Piksele obrazu zorientowane w pionie w przestrzeni akumulatora utworzą linie równoległe. Brak punktu przecięcia takich linii uniemożliwia wykrycie linii na obrazie. Rozwiązaniem tego problemu jest zaproponowana w 1972 roku zmiana reprezentacji prostej, gdzie zamiast zbocza i punktu przecięcia z osią Y użyto bieguno-wego układu współrzędnych oraz prostej normalnej do wykrywanej prostej [4]. Przykładowa prosta została zaprezentowana na rysunku 3.3 i reprezentowana jest przez wartości odległości $\rho = \frac{5\sqrt{2}}{2}$ i kąta obrotu $\theta = \frac{\pi}{4}$ wokół środka układu współrzędnych. Prosta odwzorowywana jest na sinusoidę zgodnie z równaniem 3.3.

$$\rho(\theta) = x \cos \theta + y \sin \theta \quad (3.3)$$

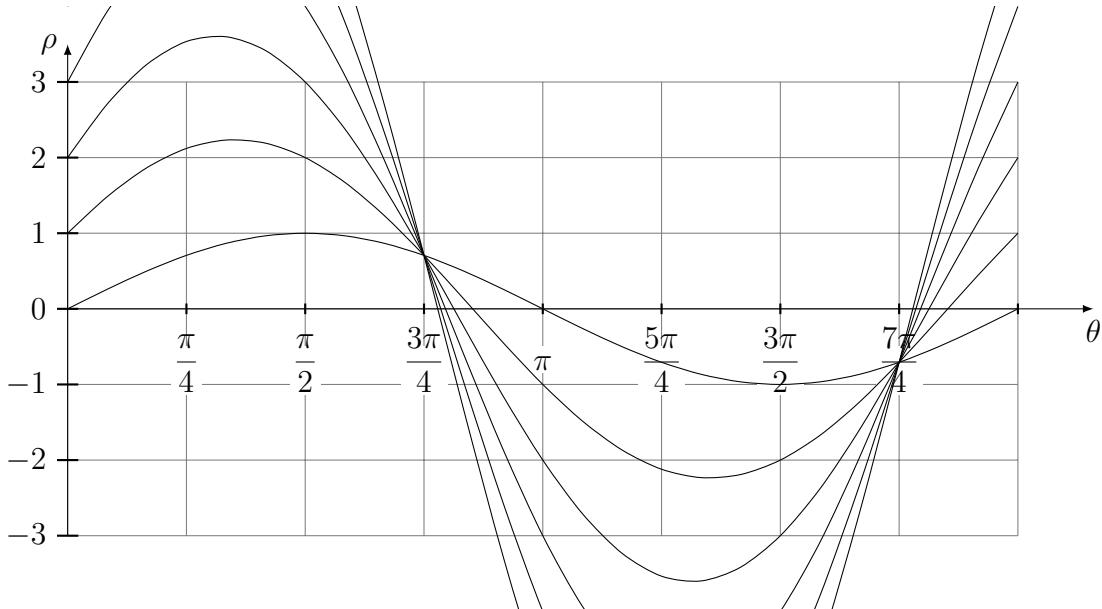
gdzie: x, y — współrzędne piksela na obrazie;

ρ — odległość prostej od środka układu współrzędnych;

θ — obrót prostej od wokół układu współrzędnych.



Rys. 3.3: Prosta opisana za pomocą odległości ρ i kąta θ od środka układu współrzędnych bieguno-wych.



Rys. 3.4: Wynik transformacji Hough'a dla obrazu na rysunku 3.2a w wariancie współrzędnych biegunowych.

Na rysunku 3.4 przedstawiono zawartość akumulatora po transformacji obrazu z rysunku 3.2a. Zgodnie z oczekiwaniemi sinusoidy te przecinają się w punktach, które reprezentują wykryte linie. Dwa punkty $(\frac{3\pi}{4}, \frac{\sqrt{2}}{2})$ oraz $(\frac{7\pi}{4}, -\frac{\sqrt{2}}{2})$, z racji na okresowość funkcji trygonometrycznych reprezentują tę samą linię. Przestrzeń akumulatora dla kąta obrotu można zatem ograniczyć do $\theta \in [0, \pi)$ [9].

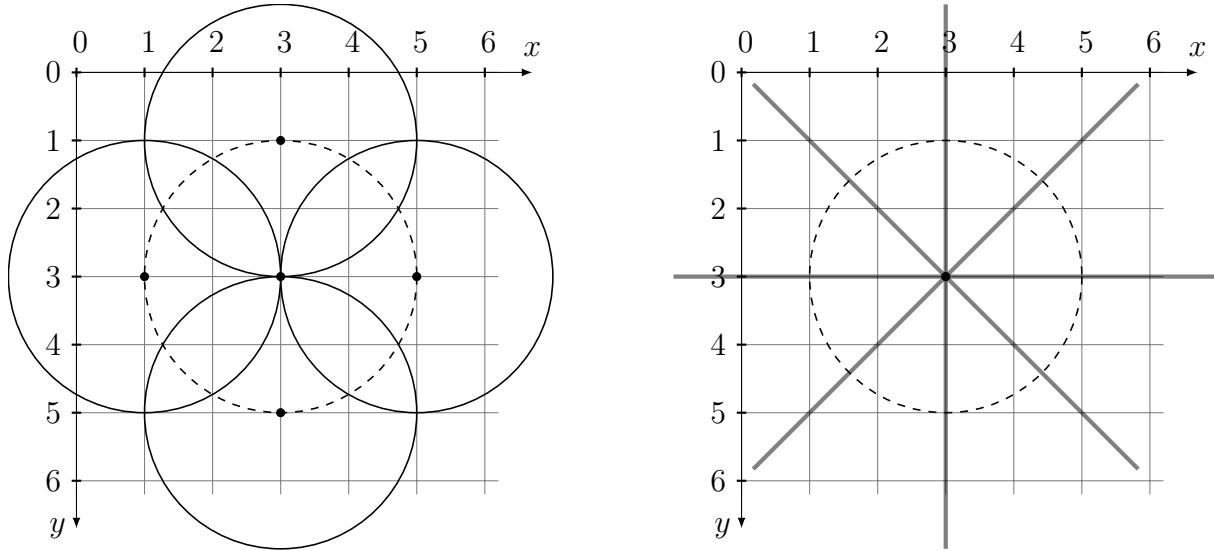
W czasie transformacji, na końcowy rezultat mają wpływ szумy, które powstały na skutek niedoskonałości procesu wykrywania krawędzi, są bardzo krótkimi krawędziami lub są krawędziami, ale nieuwzględnianymi w rozwiązywanym problemie. Przykładem mogą być okręgi podczas wykrywania linii na obrazie. Proces głosowania w przestrzeni akumulatora musi uwzględniać takie sytuacje i proste progowanie może zostać zastąpione bardziej złożonymi metodami [18, 19].

Na potrzeby prowadzonych badań zaimplementowany został algorytm wykrywania prostych na obrazie binarnym, który jako wykrywanie maksimów w akumulatorze wykorzystuje proste progowanie.

3.2. Circle Hough Transform

Prosta do swojej reprezentacji potrzebuje dwóch parametrów - zbocza i punktu przecięcia z osią Y. Okrąg natomiast jest kształtem parametrycznym, które potrzebuje trzech parametrów - współrzędnych środka i promienia. Idąc dalej, możemy rozszerzać liczbę parametrów, zyskując możliwość stosowania transformacji Hough'a do wykrywania coraz bardziej skomplikowanych kształtów.

Akumulator w Circle Hough Transform (CHT) ma trzy wymiary, po jednym na każdy parametr tak samo jak w SHT, która do wykrywania linii wykorzystywała dwuwymiarowy akumulator. Dla każdego z możliwych promieni, dla każdego piksela obrazu w przestrzeni akumulatora rysujemy okrąg, którego ten piksel jest środkiem. Efektywnie jeden piksel obrazu mapowany jest na stożek w trójwymiarowej przestrzeni akumulatora. Na rysunku 3.5a przedstawiono przykładowe okręgi w dwóch wymiarach dla stałego promienia. Wszystkie te narysowane okręgi



(a) CHT dla stałego promienia w wariancie standardowym.

(b) CHT w wariancie gradientowym dla wybranych kierunków.

Rys. 3.5: Wynik transformacji

przecinają się w środku właściwego okręgu [17]. W takiej trójwymiarowej przestrzeni akumulatora jego największe wartości wskazywać będą na konkretne środki oraz promienie potencjalnych okręgów na obrazie wejściowym.

3.2.1. Wariant z wykorzystaniem gradientu

Wraz ze skomplikowaniem kształtu parametrycznego rośnie liczba jego parametrów, których liczba stanowi liczbę wymiarów akumulatora. Dla każdego jasnego piksela obrazu konieczne jest uaktualnienie akumulatora we wszystkich jego wymiarach, co prowadzi do zwiększenia wykładowiczej złożoności obliczeniowej. Dlatego dąży się do redukcji wymiarowości problemu zazwyczaj łącząc przetwarzanie charakterystyczna dla SHT z dodatkową informacją z przestrzeni obrazu. Biblioteka OpenCV implementuje mniej złożony wariant transformacji [10], który oparty jest na wykorzystaniu gradientu wykrytych krawędzi. W wariancie tym najpierw w dwuwymiarowym akumulatorze następuje głosowanie nad centrum potencjalnego okręgu.

W pierwszej kolejności binarny obraz poddawany jest operacji splotu z filtrem Sobela osobno w kierunku pionowym oraz poziomym. Pozwala to na uzyskanie pochodnych cząstkowych w danym punkcie, które razem tworzą gradient, czyli prostopadły kierunek przebiegu krawędzi potencjalnie wskazujący środek wyszukiwanego okręgu. W dwuwymiarowym akumulatorze od analizowanego punktu w dwóch kierunkach rysowana jest linia o długości maksymalnego poszukiwanego promienia. W przypadku okręgu wszystkie te linie przecinają się w jednym punkcie, co pokazane zostało na rysunku 3.5b. Kolejnym krokiem jest wykrycie maksimów w akumulatorze. Aby zmniejszyć szумy i liczbę okręgów leżących blisko siebie, po procesie progowania środki leżące bliżej siebie niż ustalona odległość są łączona w jeden. Następnie dla każdego punktu po wykryciu maksimów w akumulatorze, w drugim jednowymiarowym akumulatorze, dla każdego możliwego poszukiwanego promienia następuje głosowanie. Zliczana jest liczba jasnych pikseli obrazu, które znajdują się w danej odległości od środka, co po wykryciu maksimum w akumulatorze uzupełnia dane okręgu o najbardziej prawdopodobny promień.

Na potrzeby prowadzonych badań zaimplementowany został algorytm wykrywania okręgów na obrazie binarnym, który wykorzystuje metodą gradientów oraz jako wykrywanie maksimów

w akumulatorze podczas głosowania dla środków jak i promieni wykorzystuje proste progowanie.

3.2.2. Próbkowanie i złożoność obliczeniowa

Głównym elementem wpływającym na złożoność obliczeniową transformacji Hough'a jest liczba analizowanych parametrów kształtów. Dla SHT złożoność wynosi $O(n)$ dla procesu głosowania i $O(S_\rho S_\theta)$ dla procesu wykrywania maksimum, gdzie n jest liczbą jasnych pikseli obrazu, a S_ρ i S_θ parametrami próbkowania parametrów w przestrzeni akumulatora. Widać zatem, że złożoność w tym wypadku zależy od jakości danych wejściowych, gdzie wszelkie szумy zwiększą czas wykonania algorytmu. Kolejnym elementem jest próbkowanie w przestrzeni akumulatora. Zwiększenie próbkowania pozwala uzyskać większą precyzję detekcji, ale zwiększa liniowo (dla jednego wymiaru) rozmiar akumulatora, a co za tym idzie ilość obliczeń wymaganych w procesie głosowania. Ważnym czynnikiem jest specyfika problemu, który rozwiązywała transformacja Hough'a. Możemy zmniejszyć czas wykonania algorytmu ograniczając zakres poszukiwań głosując w ustalonej podprzestrzeni akumulatora, na przykład analizując linie nachylone tylko pod określonym zakresem kątów i leżące w danej odległości od punktu odniesienia.

Transformacja Hough'a użyta została w algorytmach wspólnych dla wszystkich testów w środowiskach, które zostały przystosowane do badanych metod akceleracji. Wybrana została do tego celu ze względu na swoją złożoność obliczeniową, podział na wiele etapów oraz, w wariancie SHT, wykorzystania funkcji trygonometrycznych. Operuje ona również na dużych zbiorach danych, co dodatkowo pozwala rzucić światło na konieczność zarządzania pamięcią i transferu danych dla wybranych metod akceleracji.

Rozdział 4

Metodologia pomiarów

Testowanie wydajności języka JavaScript, który głównie wykorzystywany jest w przeglądarkach internetowych, z racji na ich szeroką kompatybilność oraz mnogość środowisk jest szczególnie problematyczny [37]. Rozdział ten porusza problematykę testowania syntetycznego oraz rzeczywistego. Opisuje również bibliotekę stworzoną na potrzeby prowadzonych badań.

Pierwotnie testy wydajności wykonywane były za pomocą testów syntetycznych, mikrobenchmarków - krótkich fragmentów kodu, które miały określić wydajność pojedynczej lub małego podzbioru funkcjonalności języka, na przykład porównując wydajność zwykłych tablic `Array` do tablic typowanych, której przykładem jest obiekt `Int8Array`. Popularnym narzędziem do budowanie takich benchmarków była strona jsPerf [41], która obecnie nie jest już utrzymywana.

Rosnąca liczba API i elementów ekosystemu wykorzystująca coraz bardziej złożone mechanizmy doprowadziła do ewolucji mikrobenchmarków do statycznych zestawów testów. Przykładami takowych są wspomniany wcześniej Ostrich [11], który wykonuje różnego rodzaju algorytmy numeryczne takie jak algorytm Bauma-Welcha, czy szybką transformację Fouriera. Innymi przykładami są benchmarki JetStream 2 oraz Octane sprawdzające, oprócz ogólnych algorytmów takich jak algorytmy sortowania, elementy specyficzne dla ekosystemu JavaScript takie jak czas komplikacji kompilatora TypeScript, działanie WebAssembly oraz wyrażeń regularnych [49, 40].

Jednak w przeglądarkach internetowych z perspektywy ich głównego przeznaczenia liczy się wygoda użytkowania. Czas poświęcony na wykonywanie samego kodu JavaScript, razem z pobocznymi procesami takimi jak Garbage Collector, kompilacja i optymalizacja stanowią ok 40% całego nakładu obliczeń przeglądarki, która musi oprócz tego parsować i renderować DOM oraz reagować na zdarzenia. Popularnym narzędziem do pomiarów wydajności strony z perspektywy czasu ładowania i renderowania jest Google Lighthouse [35]. Popularnymi metrykami używanymi w tego typu pomiarach jest First Contentful paint, czyli czas do narysowania czegokolwiek na ekranie, czy Largest Contentful Paint, czyli czas po którym nastąpiła największe przerysowanie elementów strony. W środowiskach serwerowych problem ładowania strony naturalnie nie występuje. Mierzy się natomiast czas zimnego startu, czyli czasu wykonania kodu razem z czasem uruchomienia samego środowiska. Ma to szczególne znaczenie w środowiskach *serverless* takich jak AWS Lambda [25].

Prowadząc badania w ramach tej pracy nie musimy skupiać się na metrykach czasu ładowania strony lub uruchamiania środowiska serwerowego, ponieważ przy intensywnych lub powtarzalnych obliczeniach stanowią one pomijalną składową stałą. Problem zimnego startu jednak występuje, choć w różnym stopniu. Wpływ na to mają procesy optymalizacji wykonania sekwencyjnego oraz czas inicjalizacji metod akceleracji [29].

Do pomiaru samego czasu wykonania można podejść na kilka sposobów. Dla krótkich fragmentów kodu istnieje ryzyko, że czas ich wykonania nie zmieści się w precyzji pomiaru czasu.

Rozwiązaniem tego problemu jest pomiar czasu wykonania t danej liczby iteracji n , a finalnym wynikiem pomiaru będzie iloraz $\frac{t}{n}$. Innym podejściem jest wykonywanie testów w pętli, aż do osiągnięcia zadeklarowanego całkowitego czasu. Hybrydowym rozwiązańiem jest dynamiczne dostosowanie liczby cykli pojedynczego pomiaru czasu, aby zaspokoić wymagania całkowitego czasu testu.

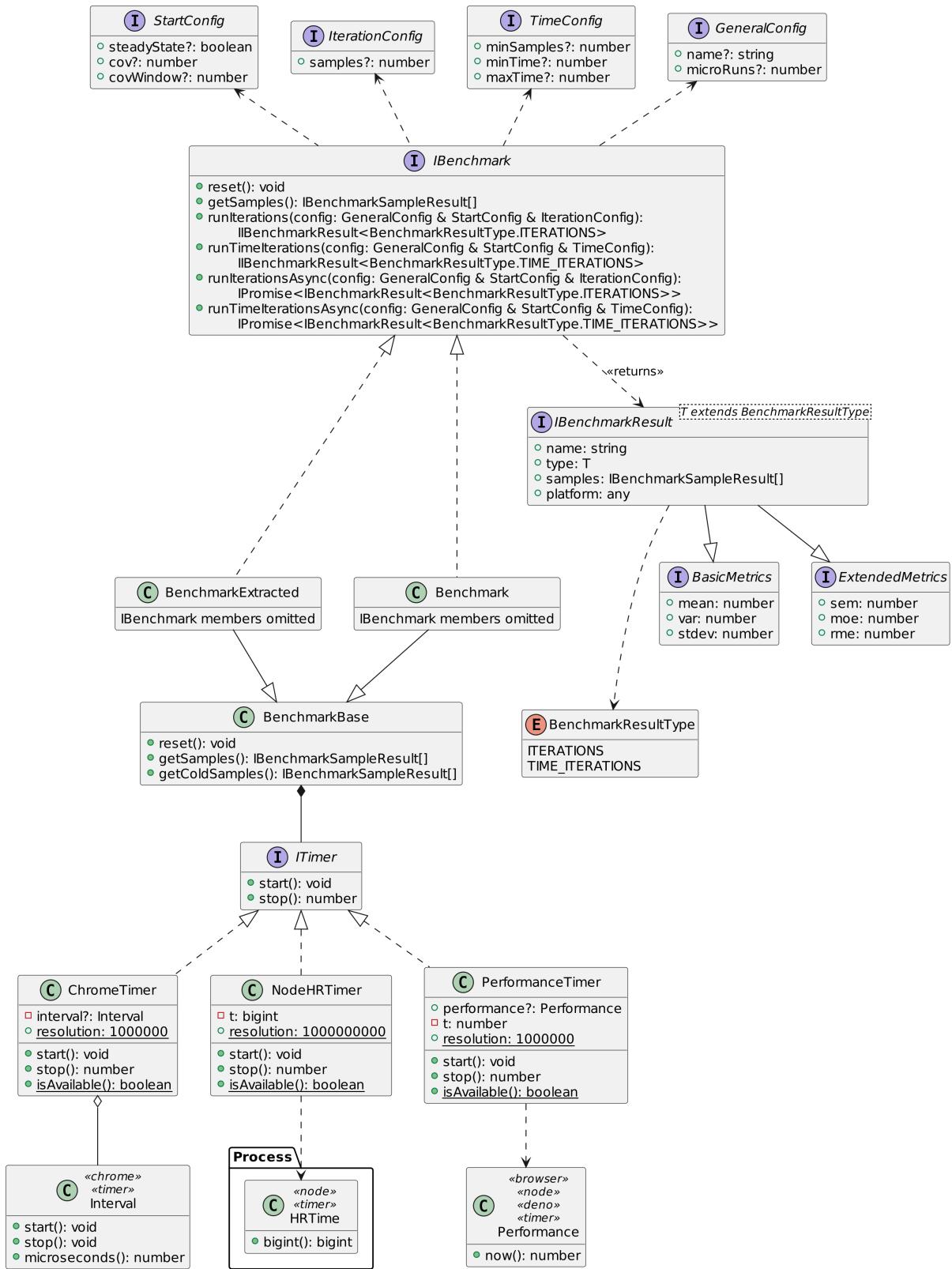
Funkcje i pętle stosowane do sterowania cyklami, analizując krótkie czasy wykonania, mogą stanowić istotną składową stałą wyników. Aby temu zapobiec stosuje się dynamicznie tworzoną funkcję testową na podstawie serializowanej funkcji testowanej, co możliwe jest poprzez wywołanie `Function.prototype.toString()`. Wywołanie to zwraca funkcję - argumenty i jej ciało w postaci ciągu znaków, a następnie ciało duplikowane jest wymaganą liczbę razy. Rozwiązań to, w przypadku bibliotek, które w procesie budowania przeszły proces minifikacji, może prowadzić do błędów związanych z nazwami symboli. Dzieje się tak, ponieważ w procesie minifikacji nazwy symboli, w celu zmniejszenia rozmiaru kodu, zamieniane są na ich krótsze odpowiedniki, często kolejne kombinacje liter alfabetu. Nie jest to możliwe dynamicznie w przypadku funkcji zserializowanej do ciągu znaków.

Problemem wartym rozważenia są również mechanizmy pomiaru czasu. Różnią się one w zależności od środowiska i w celu zapewnienia bezpieczeństwa mają one często ograniczoną rozdzielcość, aby zapobiec atakom czasowym jak Spectre i Meltdown. Standardowym rozwiązaniem kompatybilnym ze wszystkimi analizowanymi środowiskami jest Performance API z metodą `performance.now()`, która zwraca liczbę zmiennoprzecinkową reprezentującą czas w milisekundach od załadowania strony. W przeglądarce Mozilla Firefox wartość ta zaokrąglona jest do 1ms, a w Google Chrome do 0.1ms. Google Chrome po użyciu flagi `--enable-benchmarking` przy uruchomieniu udostępnia obiekt `chrome.Interval`, którego pomiary mają rozdzielcość $1\mu s$. Deno po użyciu flagi `--allow-hrtime` zwiększa rozdzielcość pomiarów przy użyciu Performance API, a NodeJS udostępnia obiekt `process.hrtime`, którego pomiary wykonywane są z rozdzielcością $1ns$.

Biblioteka `benchmark.js`, która jest przykładem biblioteki służącej do pomiarów czasu wykonania kodu, łączy przedstawione wcześniej rozwiązania, dynamicznie wykrywa środowisko, dynamicznie dostosowuje liczbę iteracji pojedynczego pomiaru czasu zgodnie ze zdefiniowanym czasem minimalnym i maksymalnym całego benchmarku oraz dynamicznie buduje funkcję testującą, która odpakowuje ciało funkcji testowanej, aby uniknąć narzutu związanego ze utworzeniem dodatkowej ramki na stosie, oraz aby móc wykorzystać zdefiniowane zmienne lokalne. Biblioteka ta jednak od 4 lat nie jest utrzymywana. Jako format modułu wykorzystuje format UMD, gdzie założeniem badań jest użycie wyłącznie modułów ECMAScript. Nie ma również możliwości zrezygnowania z dynamicznego budowania funkcji testującej, co prowadzi do błędów związanych z budowaniem środowiska testowego. Z tych powodów zadecydowano o własnej implementacji biblioteki do przeprowadzania benchmarków opartą o moduły ECMAScript oraz kompatybilną z badanymi środowiskami.

4.1. Biblioteka `benchmark`

Zaimplementowana biblioteka `benchmark`, której diagram klas przedstawiono na rysunku 4.1, pozwala wykonywać testy w dwóch wariantach wywołania - *zwykłym* i *extracted*. Tryb *zwykły* wykonuje funkcję testową dostarczoną bezpośrednio w konstrukcji klasy `Benchmark`. Tryb *extracted* odpakowuje funkcję dostarczoną do obiektu `BenchmarkExtracted` i umieszcza jej ciało w budowanej funkcji testującej. Na listingu 4.1 pokazano szablon takiej funkcji. Jako argumenty funkcja ta przyjmuje jeden obiekt kontekstu. W jej ciele definiowane są zmienne liczby iteracji oraz klasa `timer`. Następnie w pętli `while` umieszczone zostaje ciało funkcji testowanej. Całą pętlę otaczają definiowalne funkcje `setup` i `teardown` oraz rozpoczęcie i zakończenie pomiaru



Rys. 4.1: Diagram klas biblioteki Benchmark.

czasu. Funkcja ta budowana jest dla każdej iteracji testu za każdym razem zastępując znaki @ innym numerem, co efektywnie zmienia nazwy zmiennych. Działanie to jest charakterystyczne dla trybu *extracted* i pozwala na uniemożliwienie silnikowi optymalizacji kodu, co umożliwia za każdym razem uwzględnienie czasu zimnego startu.

Listing 4.1: Kompozycja funkcji w trybie *extracted*.

```

1 const template = '
2   return (
3     ${async ? "async\u2022" : ""}function(@context) {
4       let @n = @context.config.microRuns;
5       let @t = new @context.timer()
6       ${setupString}
7       @t.start();
8       while(@n--){
9         ${fnString}
10      }
11      let @tt = @t.stop();
12      ${teardownString}
13      return @tt;
14    }
15  ).replace(/@/g, config.name + (this.nameCounter++).toString());

```

Oprócz podziału na dwie klasy, które reprezentują obydwa tryby, każda z nich może wykonać testy będąc w wariancie *Iterations*, ograniczonym liczbą iteracji. W wariancie *TimeIterations* test ograniczony jest czasem minimalnym i maksymalnym, a liczba iteracji dobierana jest dynamicznie jedynie na podstawie minimalnej ich liczby, aby spełnić warunek minimalnego czasu testu. Na diagramie klas (rys. 4.1) przedstawiony interfejs **IBenchmark** definiuje metody klas biblioteki dla dwóch trybów, dodatkowo w wariantach synchronicznym i asynchronicznym.

W trybie *zwykłym* istnieje możliwość wyeliminowania problemu zimnego startu. Pierwsze wykonania, które cechować się mogą dłuższym niż reszta czasem są pomijane. Kryterium decydującym o rozpoczęciu właściwego zbierania próbek jest próg współczynnika wariancji dla czasów próbek z okna o konfigurowalnej długości, który zdefiniowany jest wzorem (4.1) [6].

$$c_v = \frac{\sigma}{\mu} \approx \hat{c}_v = \frac{s}{\bar{x}} \quad (4.1)$$

gdzie: c_v, \hat{c}_v — współczynnik wariancji i jego estymator;
 σ, s — odchylenie standardowe populacji i próby;
 μ, \bar{x} — średnia arytmetyczna populacji i próby.

Na listingu 4.2 przedstawiono przykładowe wywołanie benchmarku. Możliwe do ustawienia parametry wraz z ich opisem pokazane zostały w tabeli 4.1.

Listing 4.2: Przykładowy benchmark mierzący czas wykonania funkcji **Function.prototype()** 2000 razy.

```

1 const ben = new Benchmark(function () {
2   for (let index = 0; index < 2000; index++) Function.prototype();
3 });
4 const results = ben.runTimeIterations({
5   minTime: 200,
6   minSamples: 30,
7   microRuns: 20,
8   steadyState: true,
9   cov: 0.01,
10  covWindow: 10,
11 });

```

Ze względu na to, że problem zimnego startu stanowi pomijalną część algorytmów intensywnych obliczeniowo, które dodatkowo, jak to ma miejsce w przypadku przetwarzania obrazów, często są wykonywane wielokrotnie w ramach tej samej instancji aplikacji, przeprowadzone

Tab. 4.1: Opis parametrów uruchomienia benchmarku w bibliotece *benchmark*.

Parametr	Wartość dla testów	Opis
GeneralConfig.name	—	Nazwa identyfikująca benchmark.
GeneralConfig.microRuns	1	Liczba iteracji n pojedynczej próbki, której rzeczywisty czas t_r kalkulowany jest wg. wzoru $t_r = \frac{n}{t_c}$ na podstawie czasu całkowitego t_c . Wartość jest dostosowywana dynamicznie w wariancie <i>TimeIterations</i> i dotyczy wtedy pierwszej próbki.
StartConfig.steadyState	true	Czy pomijać pierwsze wykonania na podstawie progowania CoV.
StartConfig.cov	0.01	Próg stabilności czasu wykonania.
StartConfig.covWindow	5	Szerokość okna CoV.
IterationsConfig.samples	0.01	Liczba iteracji w wariancie <i>Iterations</i> .
TimeConfig.minSample	50	Minimalna zebrana liczba próbek w wariancie <i>TimeIterations</i> .
TimeConfig.minTime	1000	Minimalny czas benchmarku w wariancie <i>TimeIterations</i> w milisekundach.
TimeConfig.maxTime	30000	Maksymalny czas benchmarku w wariancie <i>TimeIterations</i> w milisekundach.

badania w pierwszej części nie uwzględniają tego czynnika oraz dodatkowo go eliminują wykorzystując wyżej opisany mechanizm.

Badania wykonano tylko w trybie zwykłym w wariancie *TimeIterations*. Zaimplementowano i zbadano czasy wykonania algorytmów transformacji Hough'a w wariancie SHT z zastosowaniem tablic LUT dla funkcji trygonometrycznych oraz bez. Takie same badana wykonano również dla wariantu CHT. Dla każdej z metod akceleracji zbadano również czas zimnego startu, który różni się pomiędzy metodami akceleracji. W tabeli 4.1 opisano parametry benchmarków biblioteki *benchmark*, z jakimi wykonano testy, a w tabeli 4.2 opisano dokładne wersje środowisk wraz z wersjami ich silników JavaScript, na których przeprowadzono testy.

Tab. 4.2: Wersje środowisk testowych wraz z ich wersjami silnika JavaScript.

Środ.	Wersja	Silnik JS
Chrome	97.0.4692.71	V8 9.7.106.18
Firefox	96.0	SpiderMonkey 96.0
Node	16.13.2	V8 9.4.146.24-node.14
Deno	1.18.0	V8 9.8.177.6

4.2. Badane aspekty

Jednym z analizowanych aspektów jest czas wykonania algorytmu dla stałego rozmiaru problemu. W przypadku transformacji Hough'a rozmiarem problemu jest liczba jasnych pikseli obrazu, ale także częstotliwość próbkowania akumulatora, która odpowiada za dokładność detekcji. Czas wykonania dla tego samego rozmiaru problemu został porównany pomiędzy środowiskami i metodami akceleracji. Jako punkt odniesienia pomiarów czasów użyto implementację algorytmów w języku C++ w wariancie sekwencyjnym.

Kolejnym aspektem jest zmiana czasu wykonania algorytmu w zależności od rozmiaru problemu, również w odniesieniu do metod akceleracji i środowiska. Zmiana rozmiaru problemu

została dobrana tak, że czas wykonania powinien rosnąć liniowo. Pozwoli to zaobserwować wszelkie optymalizacje i odchylenia od liniowego wzrostu czasu wykonania. Rozmiarem problemu dla algorytmu w wariancie SHT jest próbkowanie kąta obrotu wokół środka układu współrzędnych S_θ , którego wartość mówi ile pikseli wymiaru akumulatora przypada na jeden stopień. Dla wariantu CHT rozmiarem problemu jest zakres długości wykrywanych promieni, co skutkuje koniecznością rysowania dłuższych linii wzdłuż kierunku gradientu. Długość ta jest obliczana na podstawie współczynnika n ze wzoru $r_{max} = 20 + 10n$.

Ostatnim z badanych aspektów jest zjawisko zimnego startu i jego wpływ na pierwsze wykonania algorytmu. Pierwszy wykonania, które w poprzednich przypadkach są pomijane, aż do ustabilizowania się ich czasów, będą przeanalizowane.

Testy wykonane zostały na maszynie wyposażoną w procesor Intel® Core™ i7-12700KF i układ graficzny Nvidia 970 GTX w systemie operacyjnym Ubuntu 20.04.1. Procesor na potrzeby testów miał wyłączone skalowanie częstotliwości oraz ze względu na swoją hybrydową architekturę, tylko 4 rdzenie typu P-Core były przypisane dla procesów testowych, co zostało osiągnięte przez wykorzystanie narzędzia taskset. Testy wykonania sekwencyjnego w języku C++ zostały wykonane przy pomocy biblioteki *google/benchmark* [34].

Rozdział 5

Implementacja algorytmów

W rozdziale tym opisana została implementacja oraz wyniki pomiarów algorytmów dla przewidzianych środowisk oraz metod akceleracji (tabela 1.2). Na początku opisana została organizacja plików projektu oraz sposób budowania bibliotek. Następnie omówione zostały wyniki detekcji z wykorzystaniem transformacji Hough'a, które powinny być spójne dla wszystkich implementowanych metod akceleracji. Wszelkie problemy i różnice wyników pomiędzy implementacjami poszczególnych metod opisane zostaną w rozdziale 6.

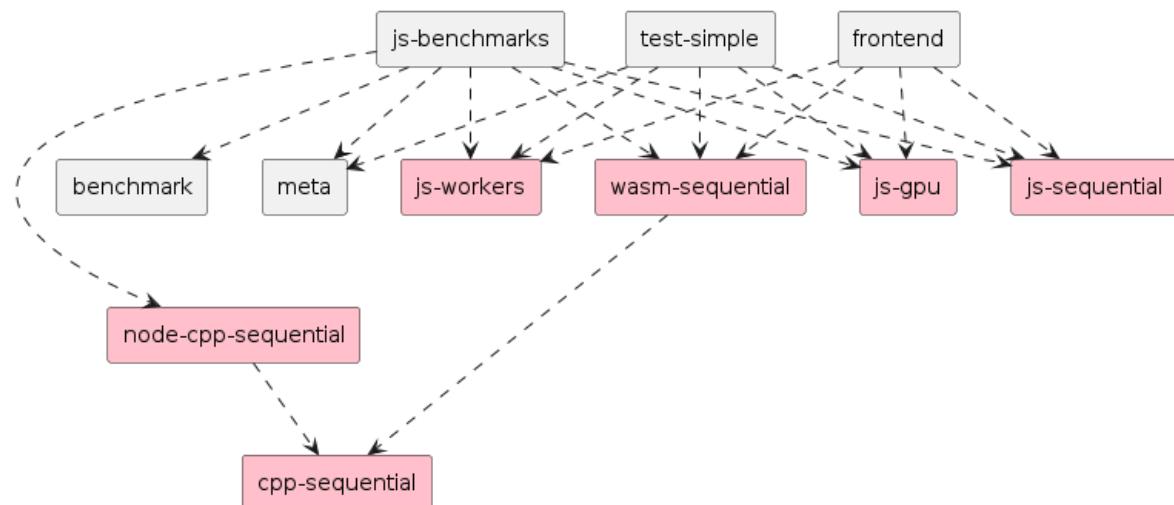
5.1. Organizacja plików

Jednym z założeń tej pracy jest analiza metod budowania bibliotek, które wykorzystują badane metody akceleracji obliczeń. Projekt, w którym znajdują się zaimplementowane metody, został zbudowany w oparciu o strategię *monorepo*, która zakłada, w ramach jednego repozytorium, współistnienie wielu projektów ze wzajemnie zdefiniowanymi relacjami [32]. W implementacji tej metody zostało wykorzystane narzędzie *Lerna* [47]. Zarządza ono projektami w abstrakcji pakietów, pozwala instalować w nich zewnętrzne zależności, wykonywać komendy w zdefiniowanym zakresie oraz tworzyć relację między pakietami. Każdy pakiet posiada swój katalog `node_modules`, w którym zdefiniowane są zależności. *Lerna*, tworząc relację między pakietami, tworzy w nim dowiązanie symboliczne do katalogu powiązanego pakietu, co pozwala innym narzędziom traktować go, jakby był zależnością instalowaną z zewnątrz. *Lerna*, jako prekursor omawianej strategii organizacji projektu, jest już wypierana przez konkurencyjne narzędzia, które rozwiązują problemy związane z zarządzaniem zależnościami i dostępem, czy dodają możliwości rozproszonego wykonania i obsługi pamięci podręcznej dla wyników komend. Przykładami takowych są Nx, Rush, Turborepo oraz Bazel.

Na rysunku 5.1 przedstawiono drzewo, reprezentujące układ najważniejszych z perspektywy organizacji repozytorium plików. Na czerwono zaznaczone zostały pakiety, które implementują metody akceleracji. Oprócz nich w pakiecie *test-simple* zaimplementowano prostą stronę internetową, która importuje zbudowane biblioteki i wyświetla wyniki przetwarzania dla testowego obrazu w celu sprawdzenia poprawności zaimplementowanych algorytmów. Pakiet *frontend* implementuje stronę internetową, która symuluje realistyczny scenariusz przetwarzania obrazu w przeglądarce internetowej. Pakiet *meta* zawiera definicje interfejsów w języku TypeScript, z których korzystają wszystkie zaimplementowane metody akceleracji. Daje to pewność spójności implementacji i możliwości porównania koniecznych działań dla metod i środowisk, aby taką spójność osiągnąć. Zależności pomiędzy pakietami zaprezentowana została na rysunku 5.2

root	Główny katalog projektu
benchmark	Pliki *.csv z wynikami pomiarów
node_modules	Zainstalowane zależności, w tym narzędzie <i>Lerna</i>
packages	Lokalizacja wszystkich pakietów
benchmark	Biblioteka <i>benchmark</i>
cpp-sequential	Implementacja metody sekwencyjnej w C++
frontend	Rzeczywisty przykład użycia
js-benchmarks	Przeprowadzanie pomiarów wydajności
js-gpu	Implementacja metody WebGL
js-sequential	Implementacja metody sekwencyjnej w JavaScript
js-workers	Implementacja metody Workers
meta	Typy języka TypeScript współdzielone pomiędzy implementacje
node-cpp-sequential	Implementacja metody Native C++ Addon w NodeJS
test-simple	Strona testowa dla implementowanych metod
wasm-sequential	Implementacja metody WASM oraz jej wariantów
test	Katalog z obrazami testowymi
js-acceleration	Konfiguracja <i>workspace</i> IDE VsCode
lerna.json	Konfiguracja narzędzia <i>Lerna</i>
package.json	Konfiguracja instalowanych zależności
package-lock.json	Zablokowane wersje instalowanych zależności

Rys. 5.1: Układ najważniejszych plików w repozytorium projektu z pakietami implementującymi metody akceleracji zaznaczonymi na czerwono.



Rys. 5.2: Zależności pomiędzy pakietami w projekcie.

5.2. Budowanie bibliotek

Głównym wymaganiem stawianym przed biblioteką jest bycie kompatybilną z możliwie wieloma środowiskami, bycie eksportowaną w formacie modułu ECMAScript oraz zapewniającą wygodę użytkowania, na którą zalicza się eksportowanie typów języka TypeScript. W głównej mierze format modułu zapewnia jego kompatybilność, ponieważ wszystkie badane środowiska obsługują moduły ECMAScript. W tej sekcji, na przykładzie biblioteki *benchmark* omówiony zostanie proces jej budowania. Pozostałe biblioteki zawierające metody akceleracji nie różnią się u podstaw w procesie budowania, a wszelkie różnice specyficzne dla metod akceleracji zostaną opisane wraz z opisem ich implementacji.

Biblioteki budowane są w wykorzystaniem narzędzia Webpack w wersji piątej. Transformuje ono pliki wejściowe, analizując i również transformując ich zależności. W zależności od konfiguracji i użytych *loader*'ów, które stanowią ogniwa łańcucha transformacji, mogą obsługiwać wiele formatów plików i produkować wyjścia w wybranych konfiguracjach. Mogą generować pojedynczy plik z kodem aplikacji lub podzielony na części.

Listing 5.1: Konfiguracja narzędzia Webpack służąca do budowania biblioteki *benchmark*

```

1 const config = {
2   entry: "./src/main.ts",
3   devtool: "cheap-module-source-map",
4   output: {
5     path: resolve(__dirname, "dist"),
6     library: { type: "module" },
7     environment: { module: true },
8   },
9   plugins: [
10     new ForkTsCheckerWebpackPlugin({
11       typescript: { build: true, mode: "write-dts" },
12     }),
13   ],
14   module: {
15     rules: [
16       {
17         test: /\.ts|tsx$/i,
18         loader: "ts-loader",
19         exclude: ["/node_modules/"],
20         options: {
21           transpileOnly: true,
22           configFile: resolve(__dirname, "./tsconfig.build.json"),
23         },
24       },
25     ],
26   },
27   // ...
28   experiments: { outputModule: true },
29   externalsType: "module",
30   optimization: { minimize: false },
31 };

```

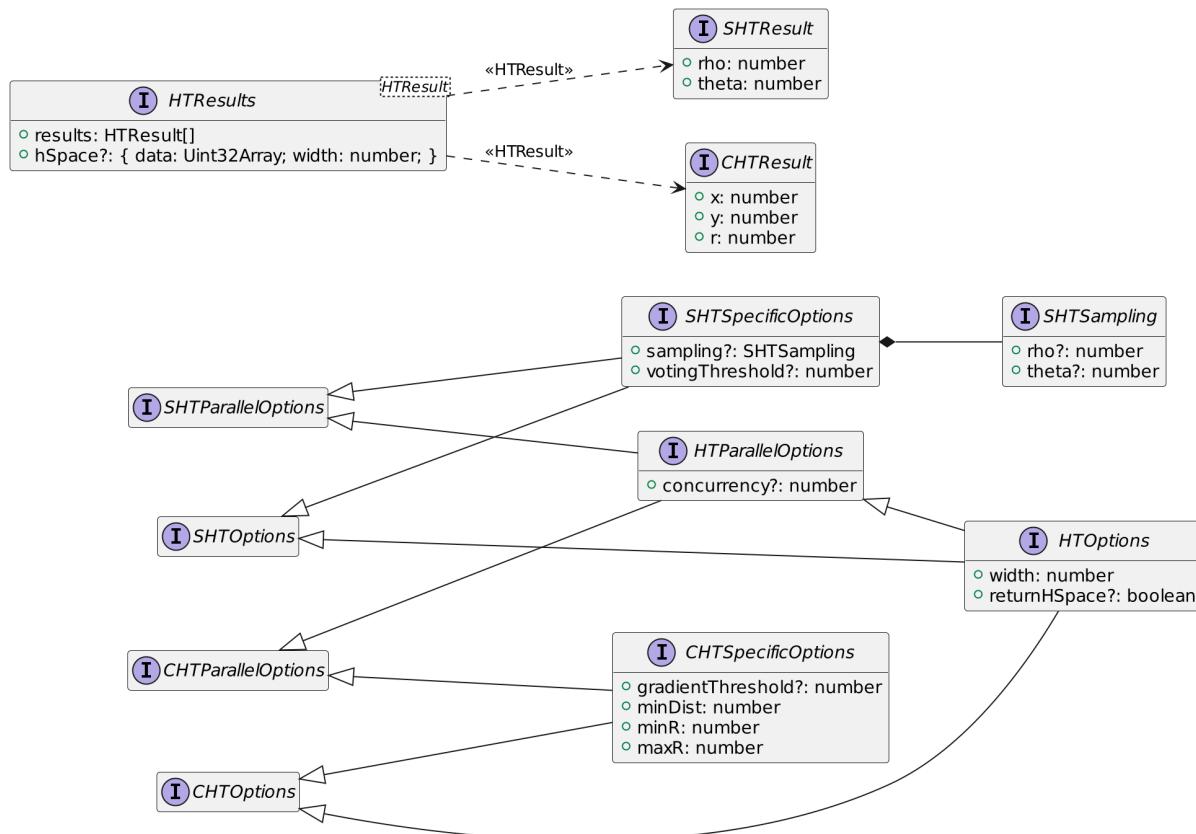
Na listingu 5.1 przedstawione zostały najważniejsze części obiektu, który stanowi konfigurację procesu budowania biblioteki *benchmark*. Pole `entry` wskazuje na pojedynczy plik wejściowy, który importuje i eksportuje to, co ma stanowić wyjście budowanego modułu. Dla pola `devtool`, które określa sposób mapowania kodu wygenerowanego na kod źródłowy dla narzędzi developerskich, ustawiono wartość "cheap-module-source-map". Wykorzystanie domyślnej wartości "eval" nie jest możliwe, ponieważ definicje map kodu biblioteki budowana w trybie *development* oraz importowane przez inną bibliotekę interferują z procesem jej budowania generując błędy. W polu `output` zdefiniowano miejsce zapisu plików wynikowych, typ generowanej biblioteki oraz właściwości środowiska, gdzie obsługa modułów ECMAScript jest możliwa. Webpack 5 obsługuje moduły dostarcza jako opcję eksperymentalną, która musi być jawnie zadeklarowana w polu `experiments.outputModule`. Pole `externalsType: "module"` definiuje format importu zewnętrznych zależności w generowanym kodzie. Plugin `ForkTsCheckerWebpackPlugin` odpowiedzialny jest za, równolegle z procesem budowania kodu JavaScript, generowanie plików `*.d.ts` zawierające definicję typów języka TypeScript. Sprawdza on również statycznie poprawność kodu i informuje o ewentualnych błędach. Jako, że pliki wynikowe biblioteki *benchmark* stanowią formę pośrednią, która będzie stanowić wejście innych procesów budowania, proces minifikacji kodu nie jest wymagany, co definiuje pole `optimization.minimize`.

Najważniejszym elementem tej konfiguracji jest pole `module.rules`, które definiuje proces transformacji plików. Skonfigurowano tutaj *ts-loader*, który odpowiedzialny jest za transpilację plików w języku TypeScript do plików języka JavaScript, które obsługiwane są przez przeglądarkę internetową i NodeJS. Pomimo tego, że Deno natywnie obsługuje język TypeScript nic

nie stoi na przeszkodzie, aby podczas przeprowadzania testów posłużyć się modułami w języku JavaScript, będącymi wynikiem budowania.

5.3. Implementacja algorytmów

Opis algorytmów należy zacząć od definicji interfejsu funkcji, który opisuje format danych wejściowych i wyjściowych. W definicjach zawarte są interfejsy dla dwóch wariantów algorytmów transformacji Hough'a - SHT i CHT.



Rys. 5.3: Definicja interfejsu algorytmów.

Na rysunku 5.3 znajduje się diagram klas reprezentujący definicję typów wykorzystywanych w implementacji algorytmów dla wszystkich metod akceleracji. Interfejsy posegregowano zgodnie z wariantami transformacji, używając typów generycznych, oraz zgodnie z wymaganiami metod akceleracji wydzielając opcje dla metod ze zrównolegleniem. Na listingu 5.2 pokazano definicję typów funkcji generycznych z podziałem na te synchroniczne (HT) i asynchroniczne (HTAsync), na podstawie których zdefiniowane zostały aliasy funkcji dla wariantów transformacji, metod akceleracji oraz współbieżności. Funkcja zawsze przyjmuje jako argumenty binarny obraz w zmiennej typu Uint8Array oraz obiekt opcji, który rozszerza interfejs HTOptions w zależności od typu algorytmu i metody akceleracji. Opcja returnHSpace określa czy razem z wynikami zwracać akumulator i na potrzeby testów jest wyłączona. Zwracanie akumulatora razem z wynikami przynieść może spadek wydajności jeśli będzie stosowane w interfejsach wymuszających kopowanie danych. Od wariantu transformacji zależy również zwracany typ rozszerzający interfejs HTResult.

Listing 5.2: Definicja typów funkcji wariantu SHT i CHT

```

1 export type HT<O extends HTOptions, HTResult> = (
2   binaryImage: Uint8Array,
3   options: O
4 ) => HTResults<HTResult>;
5
6 export type HTAsync<O extends HTOptions, HTResult> = (
7   binaryImage: Uint8Array,
8   options: O
9 ) => Promise<HTResults<HTResult>>;
10
11
12 export type SHTResults = HTResults<SHTResult>;
13 export type SHT = HT<SHTOptions, SHTResult>;
14 export type SHTAsync = HTAsync<SHTOptions, SHTResult>;
15 export type SHTParallelAsync = HTAsync<SHTParallelOptions, SHTResult>;
16
17 export type CHTResults = HTResults<CHTResult>;
18 export type CHT = HT<CHTOptions, CHTResult>;
19 export type CHTAsync = HTAsync<CHTOptions, CHTResult>;
20 export type CHTParallelAsync = HTAsync<CHTParallelOptions, CHTResult>;

```

5.3.1. Wyniki działania algorytmów

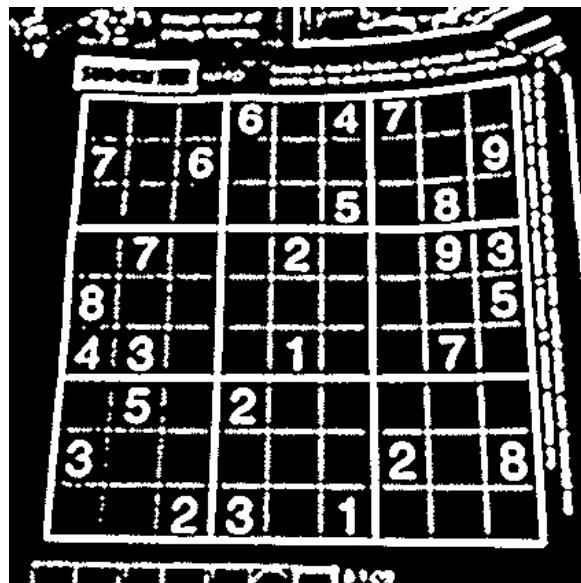
W tej sekcji przedstawiono obrazy będące wejściem, wyjściem i wizualizacją dwuwymiarowego akumulatora, która została wygenerowana poprzez odwzorowanie zakresu wartości komórek na zakres wartości $[0, 255]$.

Standard Hough Transform

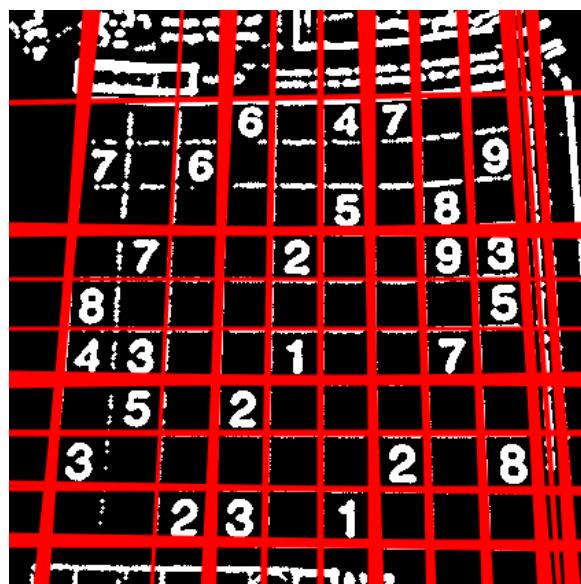
Na rysunku 5.4 przedstawiono obraz wejściowy, który stanowił podstawę weryfikacji poprawności (rys. 5.4a) oraz wynik transformacji z naniesionymi wykrytymi liniami (rys. 5.4b). W odróżnieniu od typowych obrazów wejściowych, ten został poddany operacji progowania, a nie wykrywania krawędzi. Pozwoliło to zwiększyć liczbę jasnych pikseli obrazu, a tym samym rozmiar problemu. Próbkowanie akumulatora (rys. 5.4c) dla odległości i kąta obrotu wynosi kolejno $S_\rho = 1$ i $S_\theta = 1$, co na podstawie szerokości $w = 419$ i $h = 423$ obrazu wejściowego definiuje rozmiar akumulatora jako $w_{acc} = 360$ i $h_{acc} = \lfloor \sqrt{w^2 + h^2} \rfloor = 595$.

Circle Hough Transform

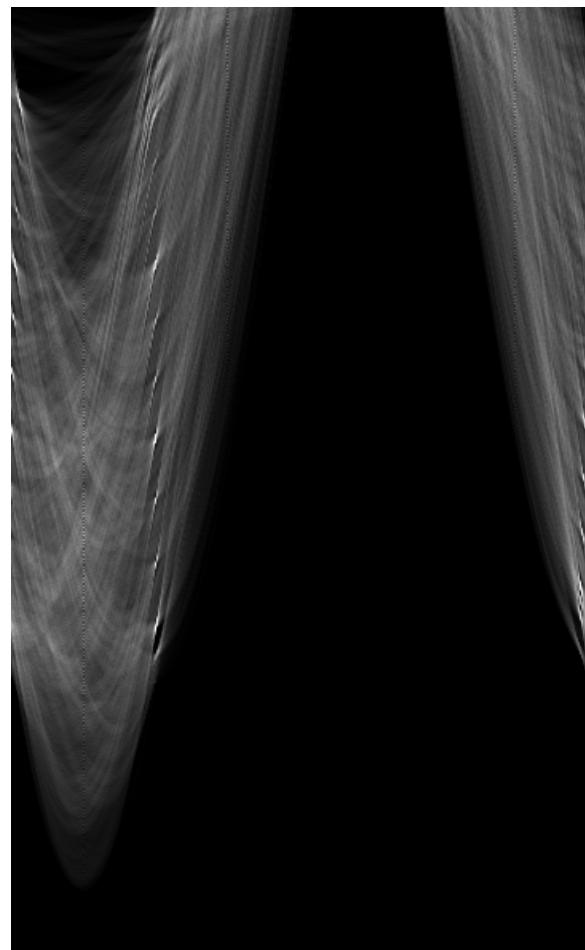
Obraz testowy dla wariantu CHT przedstawiono na obrazie 5.5a, a wynik detekcji na 5.5b. Poprzez rysowanie prostopadłych do kierunku krawędzi linii o zakresie długości odpowiadającym długości wyszukiwanych promieni, w procesie głosowania wygenerowany został akumulator przedstawiony na rysunku 5.5c. Jego rozmiar odpowiada rozmiarowi obrazu wejściowego, a najjaśniejsze punkty są kandydatami na środki okręgów. Oprócz akumulatora dwuwymiarowego do detekcji promienia CHT wykorzystuje nieprzedstawiony tutaj akumulator jednowymiarowy, który bierze udział w głosowaniu nad najlepszym promieniem. Dzieje się to osobno dla każdego kandydata na środek okręgu.



(a) Binarny obraz wejściowy

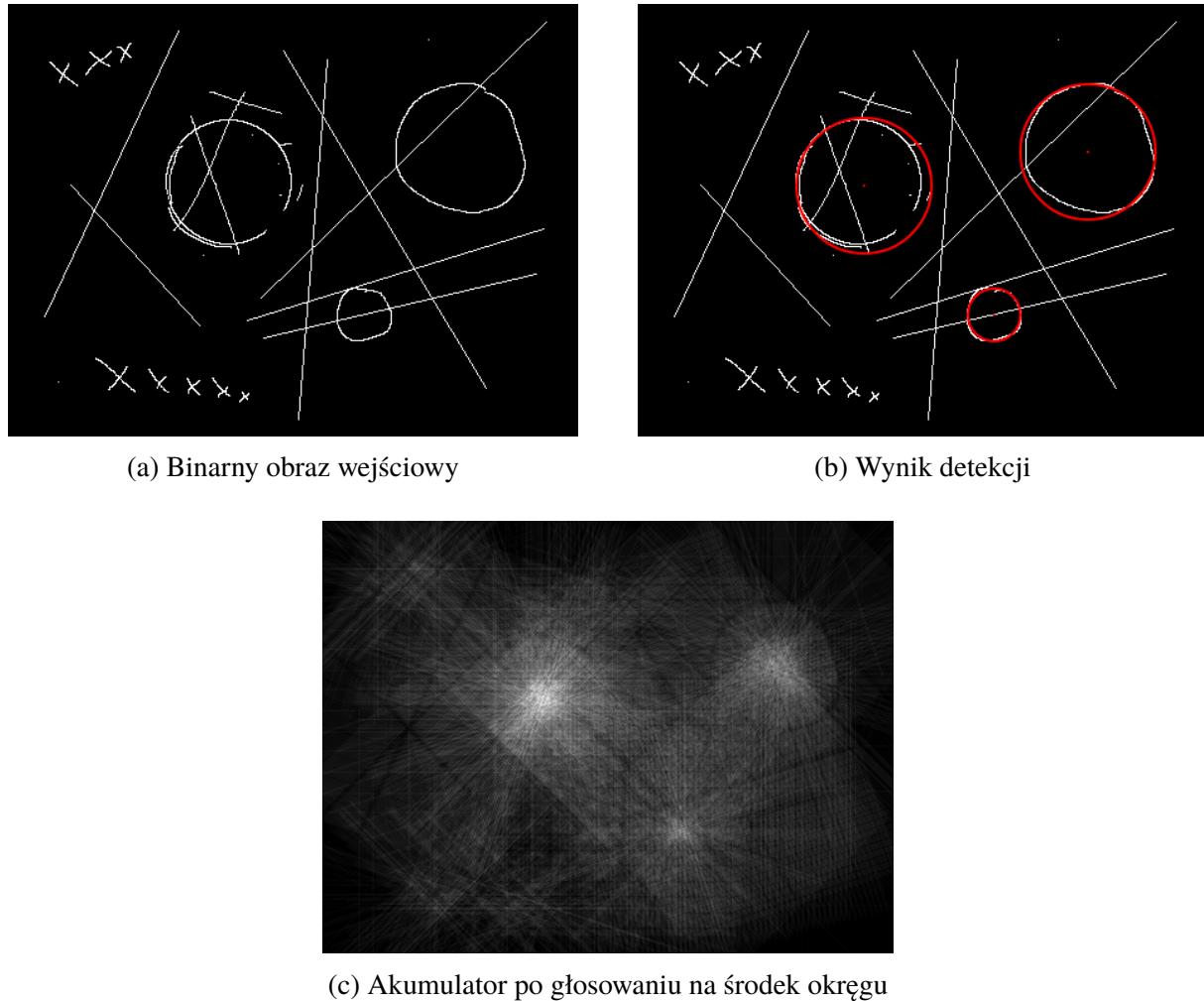


(b) Wynik detekcji



(c) Akumulator po głosowaniu

Rys. 5.4: Wynik działania zaimplementowanego algorytmu SHT w wariancie sekwencyjnym.



Rys. 5.5: Wynik działania zaimplementowanego algorytmu CHT w wariantie sekwencyjnym.

Rozdział 6

Implementacja metod akceleracji i rezultaty pomiarów

W rozdziale tym opisane zostały sposoby implementacji badanych metod akceleracji obliczeń oraz zaprezentowany zostały rezultaty przeprowadzonych testów wydajności. Wskazane zostały również wszelkie aspekty, na które trzeba zwrócić uwagę w procesie budowania bibliotek z daną metodą akceleracji oraz ich możliwości i ograniczenia. Wszystkie wyniki pomiarów porównywane zostały do wykonania sekwencyjnego w C++, a dla pozostałych metod akceleracji na wykresach kolorem szarym zaznaczony został zakres czasów wykonania sekwencyjnego wszystkich środowisk języka JavaScript.

Jak już wcześniej wspomniano w rozdziale 4.1, dla każdej z metod akceleracji zaimplementowano trzy algorytmy. Niżej przedstawiono nazwy i oznaczenia używane na potrzeby implementacji i prezentacji wyników.

- Standard Hough Transform (SHT *non-LUT*, SHT_Simple) – algorytm detekcji linii wykorzystujący intensywne obliczenia z użyciem funkcji trygonometrycznych.
- Standard Hough Transform (SHT *LUT*, SHT_Simple_Lookup) – algorytm detekcji linii wykorzystujący tablicę LUT do zapisania potrzebnych wartości funkcji trygonometrycznych.
- Circle Hough Transform (CHT, cht_Simple) – algorytm detekcji okręgów wykorzystujący metodę gradientu w celu redukcji złożoności obliczeniowej (rozmiaru akumulatora). Zmienny parametr maksymalnego promienia obliczany jest ze wzoru $r_{max} = 20 + 10n$, gdzie n to współczynnik inkrementowany w kolejnych iteracjach pomiarów.

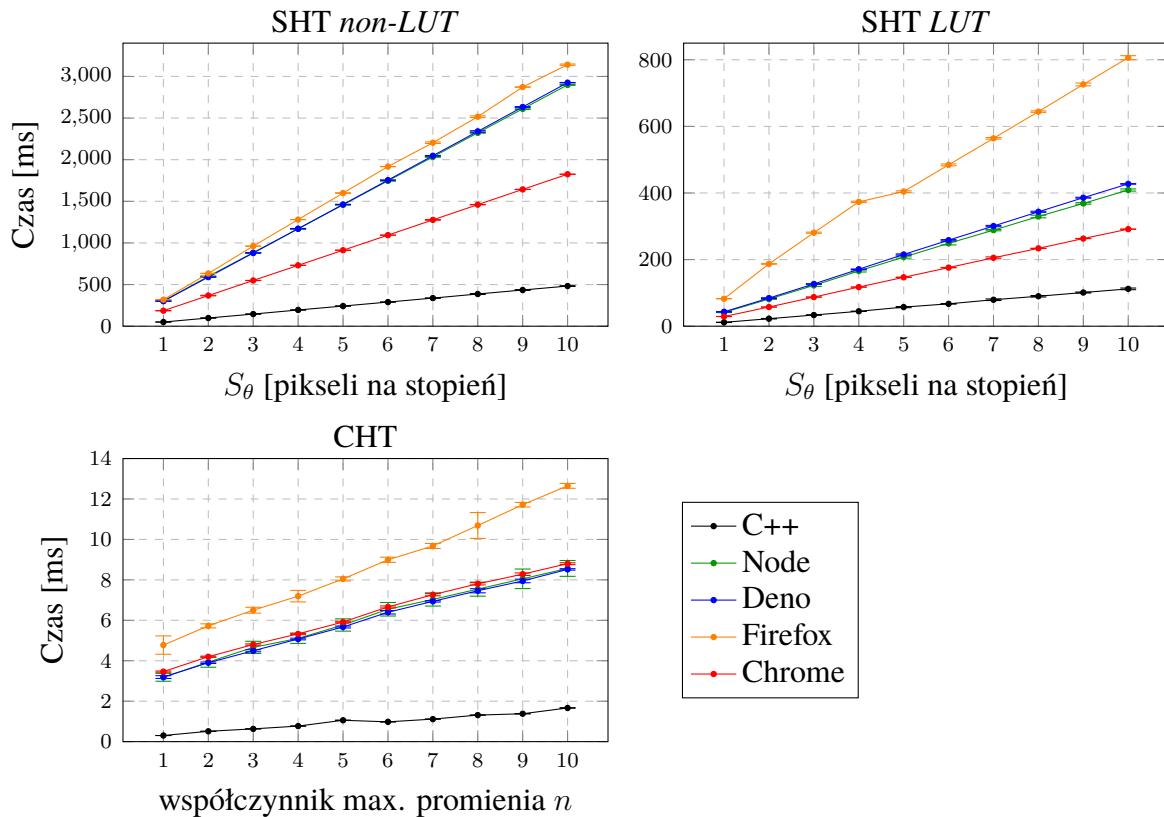
6.1. Wykonanie sekwencyjne

Implementacja procesu budowania biblioteki z algorytmem transformacji w wariancie wykonania sekwencyjnego nie wymagała zmian w procesie budowania w porównaniu do biblioteki *benchmark*, będącej punktem odniesienia. Na rysunku 6.1 pokazano rezultaty pomiarów czasu wykonania w postaci wykresów dla każdego z implementowanych algorytmów.

6.1.1. Wyniki pomiarów

We wszystkich wariantach transformacji implementacja w C++ osiągnęła najlepsze czasy wykonania, co jest oczekiwany rezultatem. Dla algorytmu SHT najlepiej zoptymalizowanym okazało się środowisko przeglądarki Google Chrome będące $3.71 \times$ wolniejsze od implementacji w C++ dla wariantu *non-LUT* i $S_\theta = 1$. Dla wszystkich algorytmów środowiska serwerowe NodeJS oraz Deno osiągnęły porównywalne wyniki z minimalną przewagą środowiska NodeJS,

która była powtarzalna pomiędzy wieloma uruchomieniami testów, jednak jest pomijalnie mała. Najgorzej zoptymalizowana okazuje się przeglądarka Mozilla Firefox, będąc $1.71 \times$ wolniejszą od Google Chrome dla $S_\theta = 1$. Interesującym dla niej zjawiskiem jest optymalizacja zachodząca dla SHT *LUT* i $S_\theta \geq 5$.



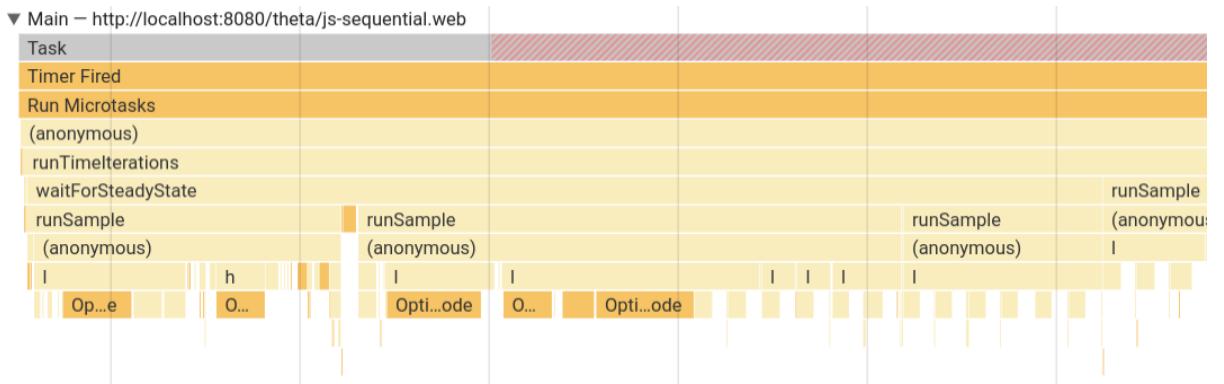
Rys. 6.1: Wyniki pomiarów czasu wydajności dla wykonania sekwencyjnego SHT i CHT.

Analizując różnice pomiędzy wykonaniami wariantów *non-LUT* i *LUT* widać, że wszystkie środowiska zyskują na optymalizacji związanej z używaniem zmiennych do przechowywania wartości funkcji trygonometrycznych, ponieważ optymalizacja ta stanowi jedyną różnicę w implementacji. Jednak przeglądarka Mozilla Firefox zdecydowanie gorzej radzi sobie z wykorzystaniem tej optymalizacji, co prowadzi do zwiększenia przewagi NodeJS i Deno z $1.05 \times$ do $1.99 \times$ większego czasu wykonania dla $S_\theta = 1$.

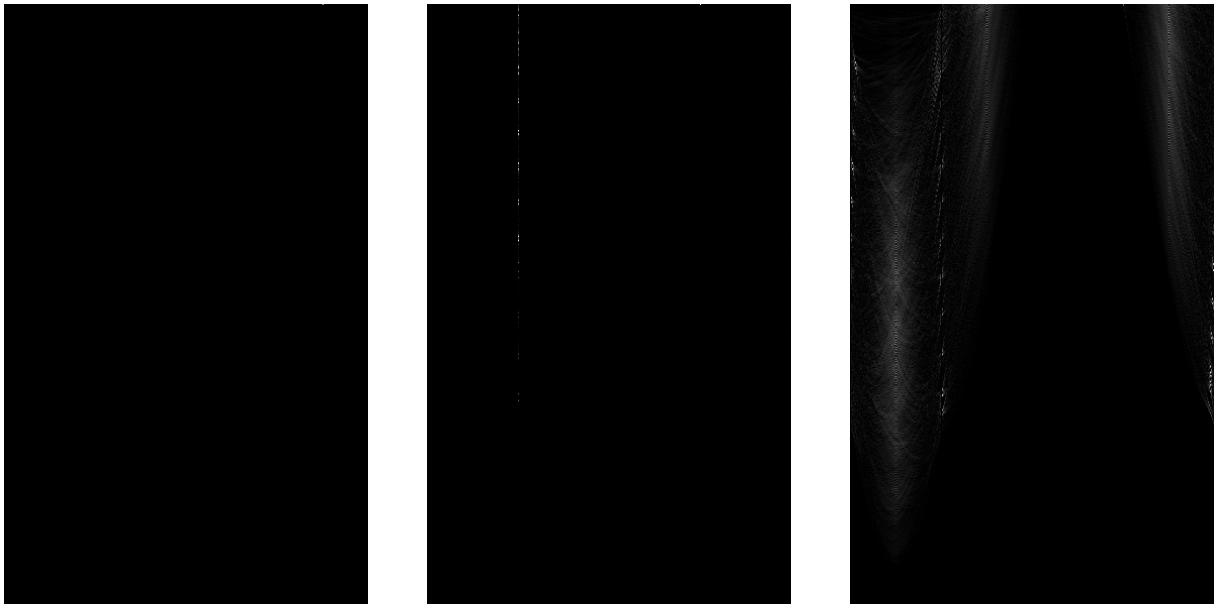
Dla algorytmu CHT środowiska NodeJS i Deno, jak i przeglądarka Google Chrome dają porównywalne rezultaty. Również i tutaj przeglądarka Mozilla Firefox okazała się być wolniejsza od pozostałych środowisk wykonując ten sam algorytm na tych samych danych.

Na rysunku 6.2 widać wynik profilowania wykonania sekwencyjnego algorytmu CHT dla $n = 1$. Biblioteka *benchmark* wewnętrznej metodzie *waitForSteadyState* czeka, aż czas wykonania kodu się ustabilizuje. Pozwala to silnikowi JavaScript optymalizować czas wykonania, co w pokazanych wynikach profilowania dzieje się w blokach *Optimize Code*, które występują tylko na początku pomiarów.

Zbadano również różnicę pomiędzy wartościami akumulatorów, która pomiędzy wariantami *non-LUT* i *LUT* powinna być równa zero. Wykryto jednak różnicę dla jednego piksela, która widoczna jest na rysunku 6.3a. Pochodzić ona może z różnicy w precyzyi operacji zmiennoprzecinkowych. Tablica LUT funkcji trygonometrycznych zbudowana została z wykorzystaniem pojedynczej precyzyji i obiektu tablicy typu *Float32Array*. JavaScript wewnętrznie do reprezentacji liczb zmiennoprzecinkowych używa podwójnej precyzyji.



Rys. 6.2: Wynik profilowania wykonania sekwencyjnego algorytmu CHT w przeglądarce Google Chrome.



Rys. 6.3: Znormalizowana do przedziału $[0, 255]$ absolutna różnica wartości akumulatorów z wynikiem głosowania pomiędzy wykonaniem sekwencyjnym SHT *non-LUT*.

6.2. NodeJS Native C++ Addon

Implementacja natywnego rozszerzenia w środowisku NodeJS wymaga utworzenia, oprócz właściwego kodu w języku C++ oraz warstwy abstrakcji, która zapewnia obsługę interfejsu po stronie języka JavaScript, pozwala na określenie typów zmiennych, kopowanie danych, czy wywoływanie funkcji. Warstwa ta zaimplementowana została za pomocą Node-API, która dostarcza niezbędne interfejsy i funkcje w języku C++, aby efektywnie powiązać kod C++ i JavaScript [48]. Implementacja natywnych rozszerzeń korzysta dokładnie z tego samego kodu, dla którego przeprowadzony pomiary w natywnym C++. Kod ten został skompilowany do postaci bibliotek współdzielonych, które w procesie linkowania są dołączane do natywnego rozszerzenia NodeJS.

Na listingu 2.2 przedstawiony został plik z pakietu *node-cpp-sequential*, który dołącza nagłówki z pakietu *cpp-sequential*. Funkcja `Napi::Object Init(Napi::Env env, Napi::Object exports)` zwraca obiekt, który definiuje wartości eksportowane przez moduł. W niej właściwie definiowane są funkcje takie jak `SHTSimple`, której implementacją stanowi `SHTSimpleBind`. Funkcja ta odpowiedzialna jest za stworzenie obiektów z danymi wejściowymi zgodnie z inter-

fejsem dołączanych bibliotek (`getTestImage` oraz `getSHTOptions`), wywołania właściwej funkcji oraz zbudowanie obiektu z odpowiedzią (`getSHTResultBind`).

Listing 6.1: Plik powiązania kodu C++ z JavaScript

```
1 #define NAPI_DISABLE_CPP_EXCEPTIONS
2 #include "CHTSimple.h"
3 #include "SHTSimple.h"
4 #include "SHTSimpleLookup.h"
5 #include "napi.h"
6 #include <cstdint>
7
8 using namespace Napi;
9
10 Napi::Object SHTSimpleBind(const Napi::CallbackInfo &info) {
11     Napi::Env env = info.Env();
12     auto testImageBind = info[0].As<Napi::Uint8Array>();
13     auto testImage = getTestImage(testImageBind);
14     auto optionsBind = info[1].As<Napi::Object>();
15     SHTOptions options = getSHTOptions(optionsBind);
16     SHTResults results = SHTSimple(testImage, options);
17
18     return getSHTResultBind(env, options, results);
19 }
20 // ...
21 Napi::Object Init(Napi::Env env, Napi::Object exports) {
22     exports.Set(Napi::String::New(env, "SHTSimple"),
23                 Napi::Function::New(env, SHTSimpleBind));
24     // ...
25     return exports;
26 }
27
28 NODE_API_MODULE(addon, Init)
```

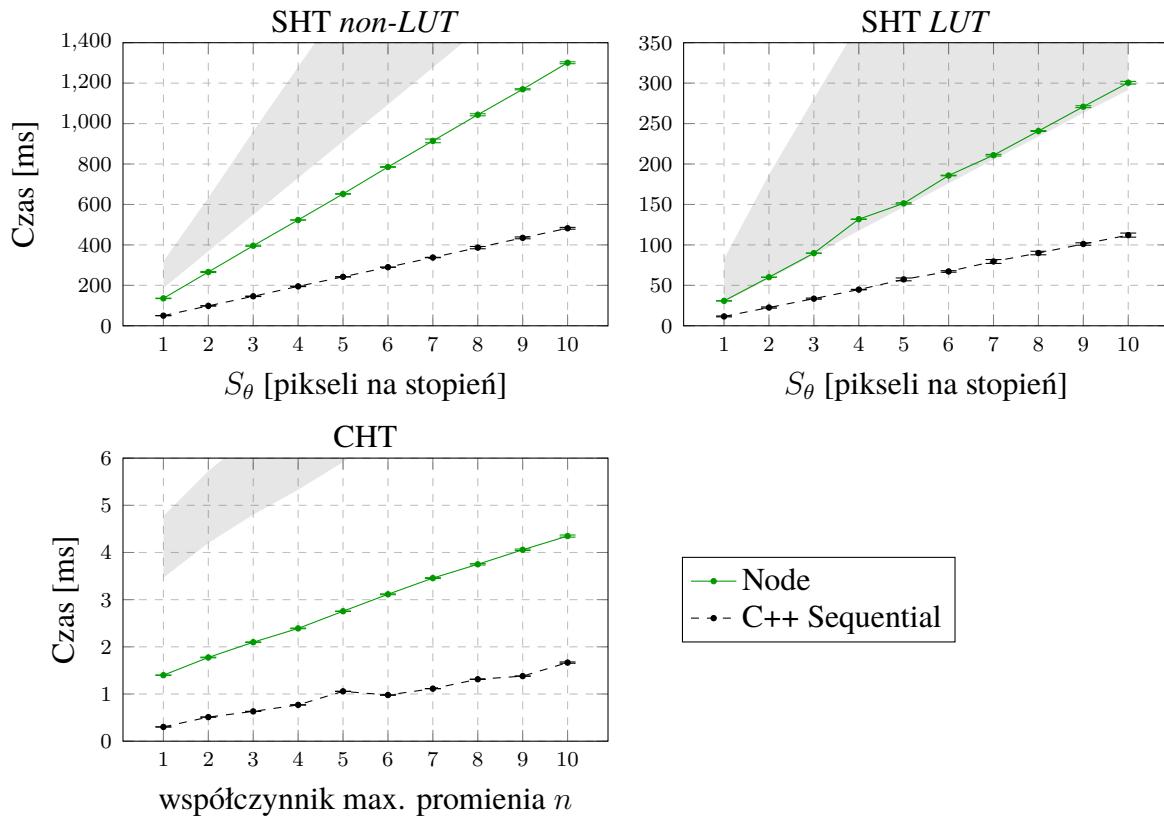
6.2.1. Wyniki pomiarów

Na rysunku 6.4 widać, że wykorzystanie tej metody akceleracji w każdym przypadku przyspieszyło działania algorytmu względem odpowiadającego mu wykonania sekwencyjnego w środowisku NodeJS. Przyspieszenie nastąpiło również względem wszystkich innych środowisk poza przypadkiem wariantu SHT *LUT*, gdzie wydajność osiągnęła poziom wydajności odpowiednika wykonania sekwencyjnego w przeglądarce Google Chrome.

Wariant SHT *LUT* był $4.40 \times$ szybszy od SHT *non-LUT*, co wraz z wynikami algorytmu CHT pozwala wyciągnąć wniosek, że jeśli w algorytmie nie znamy z góry niezbędnych wartości funkcji trygonometrycznych lub obliczenia są oparte głównie na liczbach całkowitych, to wykorzystanie tej metody akceleracji przynosi poprawę wydajności algorytmów. Oczywiście nie można zapominać o wadach takiego rozwiązania, jaką stanowi konieczność implementacji warstwy wiążącej obydwa języki. Istnieje również konieczność zapewnienia kompatybilności biblioteki z różnymi środowiskami uruchomieniowymi, dla których biblioteka musi zostać zbudowana zawczasu, lub w momencie instalacji.

6.3. WebAssembly i asm.js

Budowanie biblioteki wykorzystującą WebAssembly, tak jak w przypadku natywnych rozszerzeń NodeJS, wymaga wykorzystania specyficznych narzędzi. Do budowy biblioteki we wszyst-



Rys. 6.4: Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem natywnego rozszerzenia C++ w środowisku NodeJS.

kich wariantach wykorzystany został Emscripten [46] - zestaw narzędzi, dzięki którym dowolny przenośny kod w językach kompatybilnych z backendem kompilatora LLVM może zostać skompilowany do WebAssembly. Możliwym stało się uruchamianie w środowiskach webowych kodu języków takich jak Python czy Lua, poprzez komplikację interpreterów tych języków.

6.3.1. Implementacja procesu budowania

Na listingu 6.2 przedstawiono wywołanie komendy `emcc`, które inicjuje proces kompilacji. Z perspektywy modularności oraz wygody użytkowania niezbędne jest omówienie opcji dostarczonych do komendy. Flaga `--bind` uruchamia narzędzie Embind i dodaje do kodu wynikowego modułu powiązania pomiędzy funkcjami eksportowanymi przez moduł, a tymi zdefiniowanymi w kodzie C++. Na listingu 6.3 pokazano część kodu definiującego warstwę powiązania obydwu języków. Aby zapewnić zgodność z docelowym interfejsem funkcja `SHTSimpleBind` eksportowana jest jako `SHTSimple`. W porównaniu do natywnych rozszerzeń NodeJS w tym wypadku Embind sam zajmuje się przetwarzaniem obiektów wejściowych na podstawie dostarczonych odwzorowań (linijka 16) i powiązane funkcje zamiast argumentu z całym kontekstem, otrzymywać mogą argumenty w docelowych typach. Jednak również tutaj wszystkie dane muszą zostać skopiowane do liniowego modelu pamięci WebAssembly. Za przetwarzanie obrazu wejściowego do postaci wektora liczb w formacie `uint8_t` odpowiedzialna jest funkcja `emscripten::convertJSToNumberVector`, która wymusza traktowanie wartości jako liczby, wpływając pozytywnie na wydajność. Opcja `ALLOW_MEMORY_GROWTH=1` umożliwia powiększenie obszaru pamięci, który domyślnie ma rozmiar 16.0MB, co może nie wystarczyć, aby pomieścić przestrzeń akumulatora dla większych wartości próbkowania. Aby wynik kompilacji funkcjonował jako biblioteka eksportująca moduł ES6, należy użyć flag `--no-entry` oraz opcji `EXPORT_ES6=1`,

MODULARIZE oraz SINGLE_FILE=1, aby binarny kod WebAssembly był zagnieżdżony w pliku *.mjs w postaci kodu *base64*. Nie można również zapomnieć o możliwościach optymalizacji samego kompilatora, czyli o flagach -O3, czy -ffast-math, która wprowadza optymalizacje działań matematycznych rezygnując z poprawności wyników na ostatnich bitach.

Listing 6.2: Komenda wykorzystana podczas komplikacji kodu C++ do modułu WebAssembly.

```
1 COMMON_ARGS="-Inode_modules/cpp-sequential/include
2   --bind \
3   -s MODULARIZE \
4   -s ALLOW_MEMORY_GROWTH=1 \
5   -s FILESYSTEM=0 \
6   -s SINGLE_FILE=1 \
7   -s ENVIRONMENT=web \
8   -s EXPORT_ES6=1 \
9   --no-entry \
10  -std=c++17\
11  -ffast-math\
12  -O3"
13
14 ### Non-SIMD
15 NON SIMD_ARGS="$COMMON_ARGS \
16   src/wasm_sequential.cc \
17   node_modules/cpp-sequential/src/SHTSimpleLookup.cpp \
18   node_modules/cpp-sequential/src/SHTSimple.cpp \
19   node_modules/cpp-sequential/src/CHTSimple.cpp"
20
21 emcc $(echo $NON SIMD_ARGS -o build/wasmSequential.mjs)
```

6.3.2. Warianty testów

Dla każdego wariantu algorytmu transformacji przewidziano pomiary wydajności dla każdej z wymienionych niżej metod.

- asm.js – wydajny podzbiór języka JavaScript zoptymalizowany pod kątem możliwości optymalizacji silnika i inferencji typów,
- WASM – standardowy wynik komplikacji kodu C++
- WASM SIMD (implicite) – wynik komplikacji kodu C++ z włączonym procesem automatycznej wektoryzacji wykonania, w celu wykorzystania instrukcji wektorowych SIMD,
- WASM SIMD (explicite) – wynik komplikacji kodu C++ z ręcznie przeprowadzonym procesem wektoryzacji wykonania.

Zbudowanie biblioteki w asm.js wymaga dodania opcji WASM=0. Wariant SIMD explicite i implicite budowany jest z dodatkową flagą -msimd128. Wariant SIMD explicite jest ręcznie przy stosowany do obsługi operacji na wektorowych rejestrach 128b. Przykładem takiego zastosowania jest pokazana na listingu 6.4 funkcja obliczająca minimalne i maksymalne współrzędne w przestrzeni akumulatora za jednym razem dla osi OX i OY. W procesie implementacji wersji SIMD explicite wykorzystano operacje na czteroelementowych wektorach liczb całkowitych bądź zmiennoprzecinkowych dla każdej intensywnie obliczeniowo pętli redukując liczbę ich iteracji czterokrotnie.

Listing 6.3: Wybrane fragmenty kodu warstwy powiązania WASM pomiędzy C++ i JavaScript.

```

1 // ...
2 #include <emscripten.h>
3 #include <emscripten/bind.h>
4
5 extern "C" {
6 // ...
7 EMSCRIPTEN_KEEPALIVE
8 SHTResults SHTSimpleBind(emscripten::val binaryImageBind, SHTOptions
    ↪ options) {
9     auto testImage =
10     emscripten::convertJSArrayToNumberVector<uint8_t>(binaryImageBind);
11     return SHTSimple(testImage, options);
12 }
13 // ...
14 }
15
16 EMSCRIPTEN_BINDINGS(wasm_sequential) {
17     emscripten::register_vector<uint32_t>("VectorUint32");
18     emscripten::register_vector<uint8_t>("VectorUint8");
19     emscripten::value_object<SHTSamplingOptions>("SHTSamplingOptions")
20         .field("rho", &SHTSamplingOptions::rho)
21         .field("theta", &SHTSamplingOptions::theta);
22     // ...
23     emscripten::function("SHTSimple", &SHTSimpleBind);
24     // ...
25 }
```

Listing 6.4: Funkcja getBounds algorytmu CHT z wykorzystaniem instrukcji SIMD.

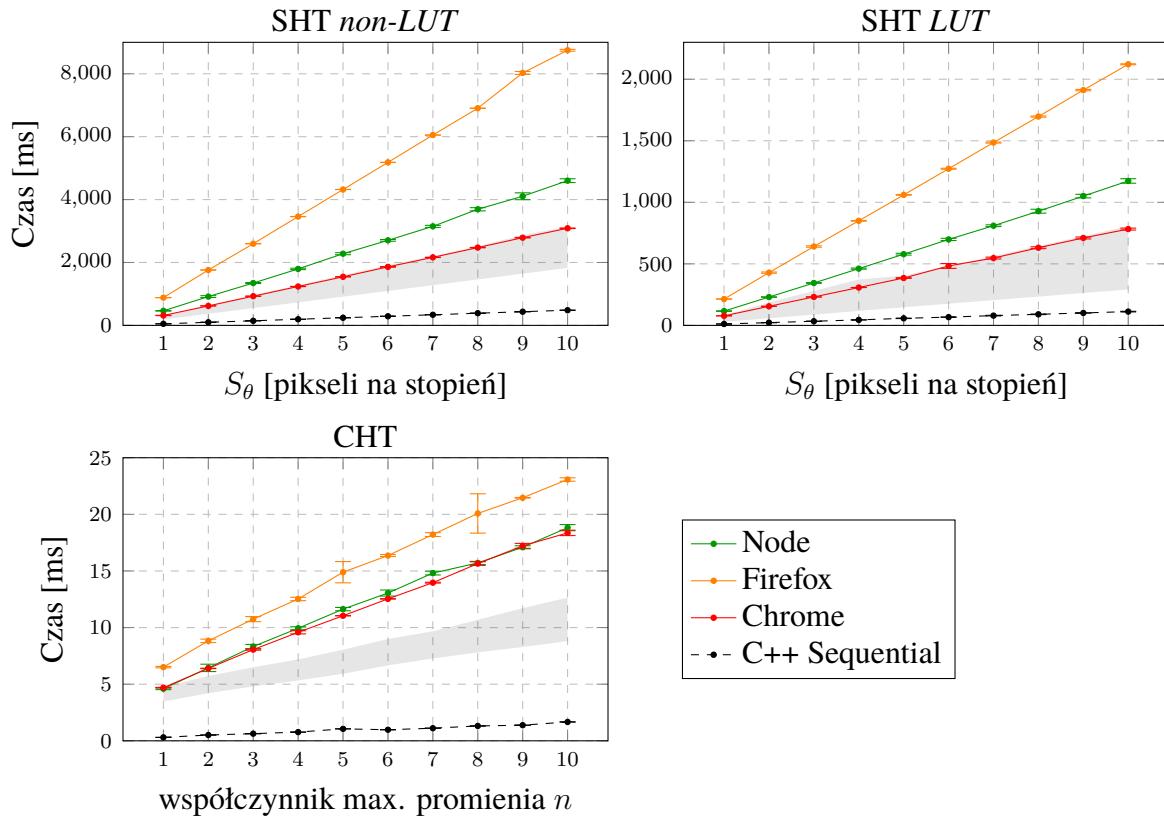
```

1 #include <wasm_simd128.h>
2 / ...
3 void getBounds(int32_t x, int32_t max, v128_t vRBounds, int32_t *out) {
4     v128_t vZero = wasm_i32x4_const_splat(0);
5     v128_t vMax = wasm_i32x4_splat(max);
6     v128_t vX = wasm_i32x4_splat(x);
7     vX = wasm_i32x4_add(vRBounds, vX);
8     vX = wasm_i32x4_max(vX, vZero);
9     vX = wasm_i32x4_min(vX, vMax);
10    wasm_v128_store(out, vX);
11 }
```

6.3.3. Wyniki pomiarów

Analizując wykres przedstawiający wyniki pomiarów czasów wykonania algorytmów z wykorzystaniem komplikacji do asm.js przedstawiony na rysunku 6.5 zobaczyć można, że ta metoda akceleracji w każdym przypadku daje gorsze rezultaty od wykonania sekwencyjnego. Może to mieć dwie potencjalne przyczyny. Pierwszą z nich jest nieprzeprowadzanie komplikacji kodu przez środowiska, które z asm.js nie są w pełni kompatybilne, co w przypadku tych konkretnych algorytmów powoduje utratę wydajności. W zbudowanym module nie znajdują się funkcje trygonometryczne `Math.sin` i `Math.cos`. Oznacza to, że w kodzie znajduje się ich ręczna implementacja, która nie korzysta z biblioteki standardowej. Drugim możliwym powodem, w szczególności w przypadku przeglądarki Mozilla Firefox, która z asm.js jest w pełni kompatybilna, może być problem z wykrywaniem kodu asm.js, który jest umieszczony w jednym pakiecie z resztą

kodu w procesie budowania. Świadczyć może o tym fakt, że podczas profilowania wykonania nie występowały bloki *Compile*, co ma miejsce podczas wykonania wyizolowanych przykładów.

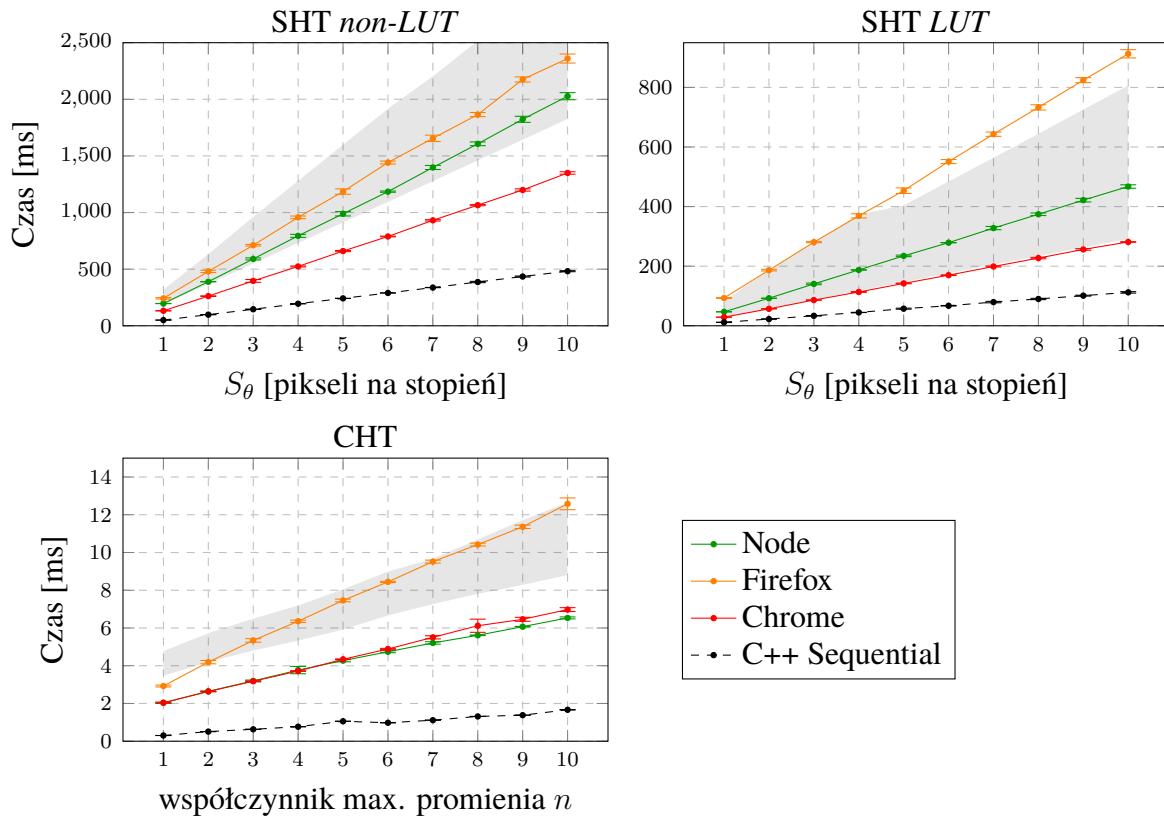


Rys. 6.5: Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem kompilacji do asm.js.

Kompilacja do WebAssembly zwiększyła wydajność wariantu algorytmu SHT *non-LUT* we wszystkich testowanych środowiskach (rys. 6.6). Jednak największy wpływ miała na środowiska przeglądarek internetowych, które zdają się lepiej reagować na te metodę akceleracji niż środowisko NodeJS. W wariantie SHT *LUT* za to nie zaobserwowano żadnej poprawy względem wykonania sekwencyjnego, dodatkowo nie zaobserwowano optymalizacji czasu wykonania dla $S_\theta \geq 5$. Warianty te różnią się jedynie zmniejszoną liczbą wywołań funkcji trygonometrycznych. Założyć można zatem, że zaobserwowana optymalizacja przeglądarki Mozilla Firefox dotyczyła właśnie ich i nie odbyła się w momencie komplikacji do WebAssembly.

Przypadek algorytmu CHT pokazuje jak różne środowiska mogą reagować na ten sam kod. W przeglądarce Google Chrome z silnikiem V8 zaobserwowano poprawę wydajności, gdzie wykonanie z wykorzystaniem WebAssembly było $1.70\times$ szybsze od wykonania sekwencyjnego dla $n = 1$. Mozilla Firefox jednak zachowuje się inaczej. Dla $n = 1$ wykonanie z wykorzystaniem WebAssembly było $1.63\times$ szybsze, niż wykonanie sekwencyjne, jednak dla $n = 10$ czas wykonania był już taki sam jak czas wykonania sekwencyjnego. Na tej podstawie można wyciągnąć wniosek, że w przypadku przeglądarki Mozilla Firefox duża liczba iteracji generuje narzut, który niweluje korzyści, jakie płyną z ogólnej poprawy wydajności dzięki WebAssembly.

Kompilacja do WebAssembly z włączonym procesem automatycznej wektoryzacji w narzędziu Emscripten nie przyniosła żadnej różnicy w porównaniu do wykonania zwykłego kodu WebAssembly i dlatego nie będzie dalej omawiana. Natomiast w przypadku ręcznej optymalizacji, którego przykład pokazany został na listingu 6.4, poprawa wydajności zależała od środowiska. Na rysunku 6.7 zaobserwować można, że wszystkie środowiska pozytywnie zareagowały na zastosowaną metodę dla wariantu algorytmu SHT *non-LUT*. Najszybsza znów okazała się



Rys. 6.6: Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem komplikacji do WASM.

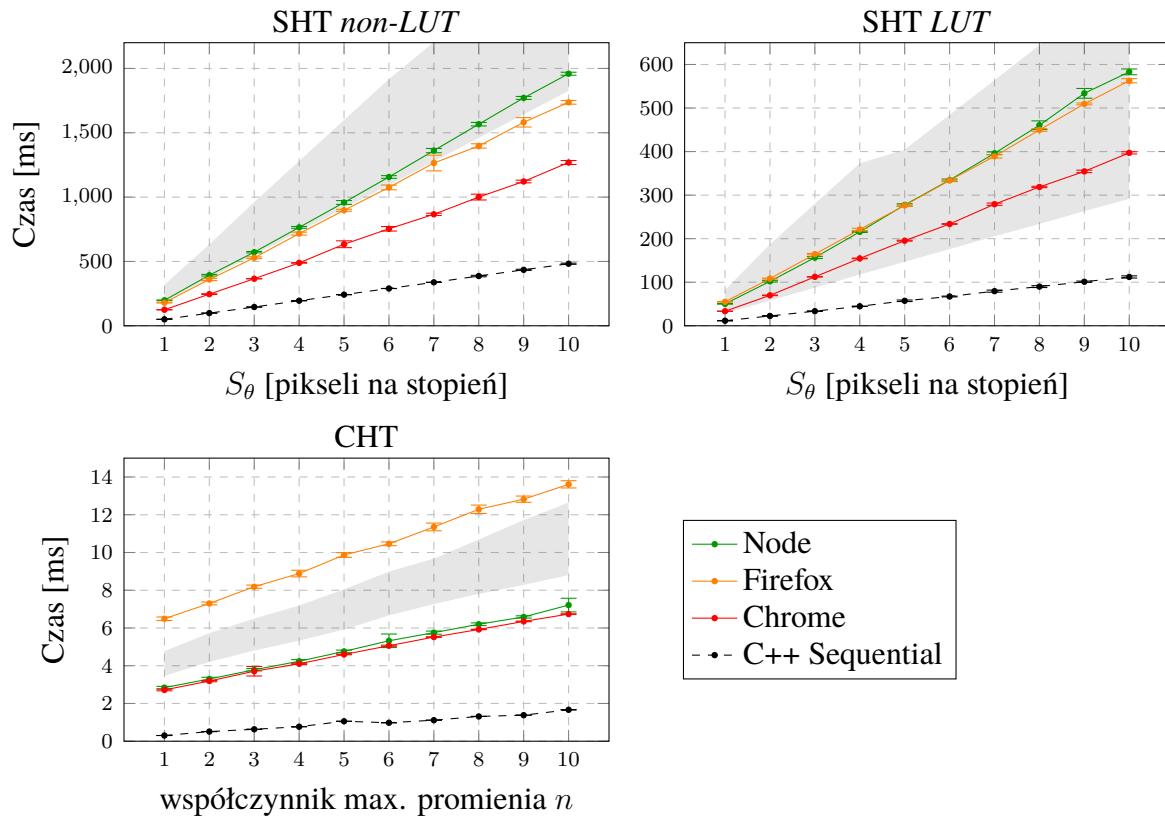
przeglądarka Google Chrome, osiągając wynik $1.50\times$ lepszy od wykonania sekwencyjnego. W wariantie SHT *LUT* jednak wykonanie sekwencyjne było $1.15\times$ szybsze. Wykonanie wariantu CHT w przeglądarce Mozilla Firefox, w porównaniu do wykonania sekwencyjnego, było wolniejsza, a wykonanie w środowiskach Google Chrome i NodeJS zyskało na wydajności.

Dla analizowanego kąta obrotu $\theta = \frac{\pi}{2}$ wykryto duże różnice w akumulatorze pomiędzy wykonaniem SHT WASM a sekwencyjnym SHT. Różnica widoczna jest na rysunku 6.3b. Oprócz wyraźnej linii różnica występuje też dla wielu pojedynczych pól akumulatora i może wynikać z różnic w reprezentacji w algorytmie liczb zmiennoprzecinkowych, co generuje różnice w wyliczeniach pól akumulatora.

6.4. Współbieżność

Zrównoleglenie wykonania bezdyskusyjnie przynosi wzrost wydajności. Jednak, aby mogło być zastosowane algorytm musi dać się zrównoleglić. Również docelowe środowiska muszą tę współbieżność wspierać. Wspólnym mianownikiem wszystkich badanych środowisk jest wielowątkowość w postaci Worker'ów. Oprócz nich środowisko NodeJS posiada natywne mechanizmy komunikacji między procesami, jednak nie są one przedmiotem badań tej pracy.

Komunikacja pomiędzy Worker'ami odbywa się z wykorzystaniem interfejsów WebWorker API. Dostarczają one mechanizm wiadomości, które są asynchronicznie przetwarzane przez Worker'y, które jako niezależne wątki mają osobną pętlę zdarzeń. Wiadomość jest w abstrakcji przetwarzaną zdarzeniem. Metoda `Worker.postMessage(...)` oraz definiowana funkcja `self.→ onmessage` pozwala na wysyłanie i obsługę zdarzeń wiadomości pomiędzy wątkami. Mechanizm ten jest bardzo prosty, niweluje problemy wyścigów oraz współdzielenia zasobów, ponieważ wszystkie dane są albo kopiowane, albo transferowane z jednego wątku do drugiego.



Rys. 6.7: Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem komplikacji do WASM z ręczną implementacją instrukcji SIMD.

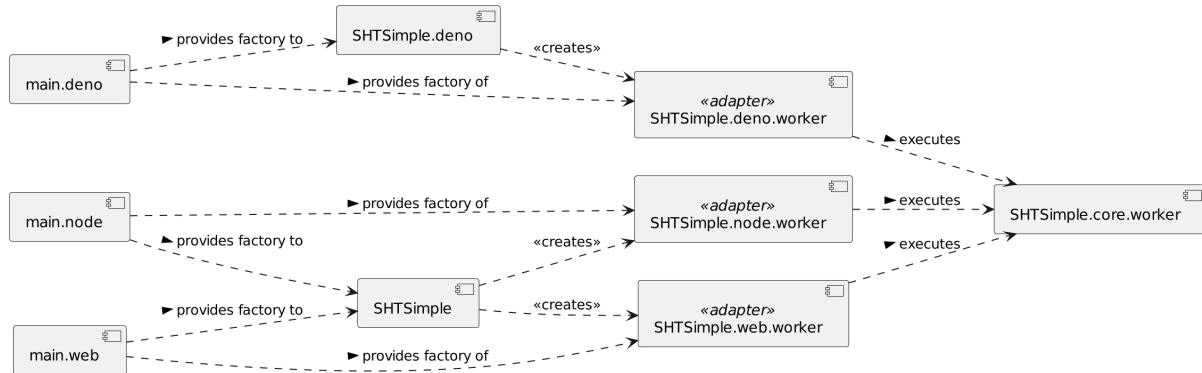
Jego wadą jest sam interfejs, który wymaga tworzenia dodatkowych mechanizmów serializacji i parsowania złożonych wiadomości. Z pomocą przychodzi, zastosowana przy implementacji Worker'ów na potrzeby badań, biblioteka Comlink [27]. Pozwala ona na zdalne wywoływanie funkcji i synchronizację zmiennych, co nie odbiega interfejsem od sposobu interakcji z metodami i atrybutami standardowych obiektów. Jest ona kompatybilna ze wszystkimi badanymi środowiskami.

Implementacja Worker'ów wymagała przystosowania algorytmów. Podzielono wykonanie głównych ich pętli pomiędzy wątki. W testach badano czasy wykonania dla liczby wątków $c = 4$. Aby zapewnić kompatybilność ze wszystkimi środowiskami konieczne było zbudowanie dwóch wersji biblioteki. Biblioteka w środowisku Deno była uruchamiana z pominięciem procesu budowania, ponieważ Deno pozwala bezpośrednio na uruchomienie kodu w języku TypeScript.

Dla środowiska przeglądarki internetowej konieczne jest użycie loader'a `worker-loader`, który, z opcją `inline: 'no-fallback'`, odpowiedzialny jest za dołączenie kodu Worker'a do kodu biblioteki, aby taka biblioteka mogła być ponownie zbudowana razem z docelową aplikacją. W standardowym scenariuszu, kiedy Worker budowany jest razem z aplikacją, jego kod jest pobierany jako osobny plik przez przeglądarkę. Kod musi być dołączony w tym samym pliku, ponieważ niemożliwym jest, bez dodatkowych skryptów, wskazanie i skopiowanie pliku Worker'a w procesie budowania.

Dla środowiska NodeJS konieczne jest jawne wskazanie środowiska, dla którego przeznaczona jest budowana biblioteka poprzez ustawienie opcji `target: 'node'`. NodeJS do obsługi Worker'ów używa wbudowanego modułu, WebWorker API jest lekko zmodyfikowane, a konstruktor obiektu Worker nie jest dostępny jako symbol globalny. Narzędzie Webpack musi uwzględnić wszystkie te elementy. Niekompatybilność ta uniemożliwia w wypadku tej me-

tody akceleracji zbudowanie pliku kompatybilnego ze wszystkimi środowiskami. Na rysunku 6.8 przedstawiono diagram zależności modułów w kodzie, w tym adapterów, które zapewniają kompatybilność kodu algorytmu ze wszystkimi środowiskami. Kod części właściwego algorytmu, który tworzy i wywołuje funkcje Worker'ów w środowisku Deno jest oderwany od jego odpowiednika wspólnego dla przeglądarek i NodeJS. Jest to spowodowane niekompatybilnością API biblioteki Comlink w wersji na środowisko Deno.



Rys. 6.8: Sieć zależności pomiędzy modułami implementującymi algorytm SHT w wersji *non-LUT*, która zapewnia kompatybilność ze wszystkimi środowiskami.

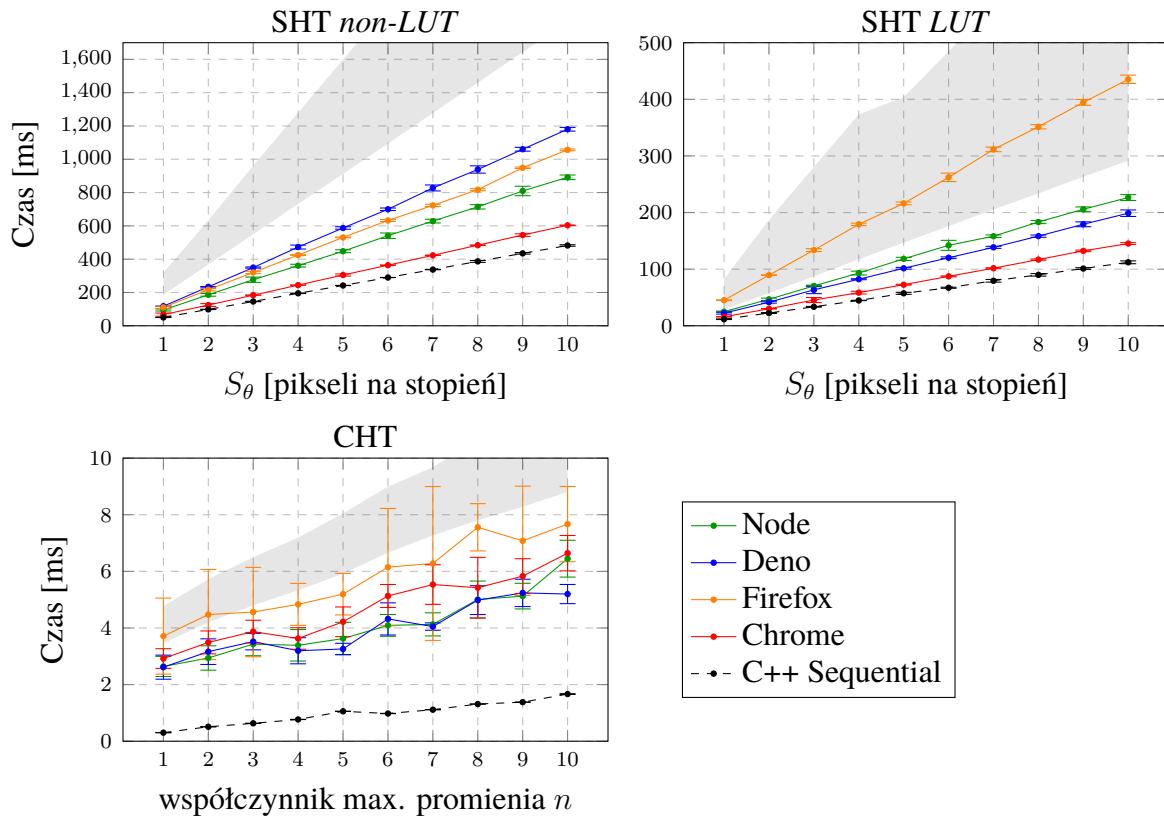
6.4.1. Wyniki pomiarów

Pomiar dla wykonania wielowątkowego przeprowadzone były przy przeznaczonych do tego 4 fizycznych rdzeniach procesora, aby uniknąć wpływu hyper-threadingu na czas wykonania. Wymagało to przypisania 8 wirtualnych rdzeni używając maski `0x000000ff` jako konfiguracji narzędzia taskset.

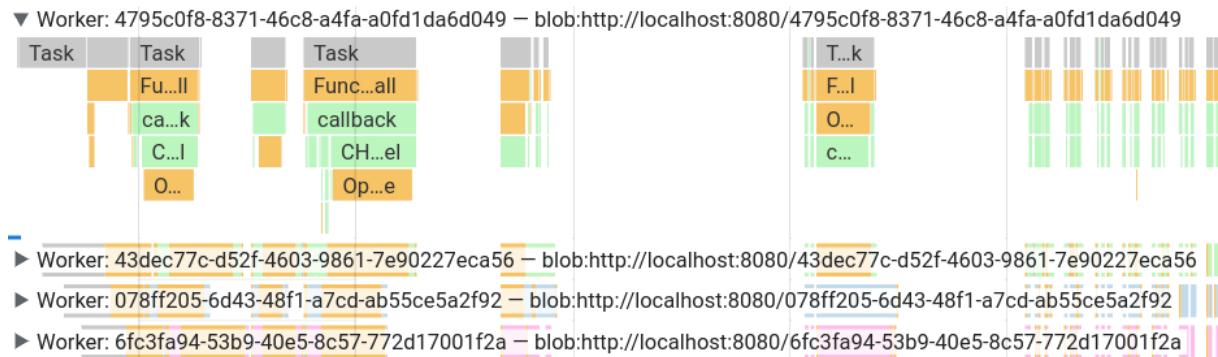
Na rysunku 6.9 pokazane zostały wykresy z wynikami pomiarów czasów wykonania z wykorzystaniem czterech Worker'ów. Jak w większości przypadków również i tutaj, dla obydwu wariantów SHT, prym we wzroście wydajności wiedzie przeglądarka Google Chrome. Przy krótszych czasach wykonania dla wariantu CHT uwydatnia się większa niż dla poprzednich metod wartość odchylenia standardowego. Szczególnie widoczna jest w przypadku przeglądarki Mozilla Firefox. Spowodowane jest to asynchronicznością komunikacji z Worker'em, gdzie wiadomości nie są obsługiwane od razu, a trafiają do pętli zdarzeń. W zależności od implementacji, jak i liczby zdarzeń do obsłużenia przez pętlę, czasy te mogą się różnić. Kod Worker'ów jest również osobno optymalizowany, co zwiększa skalę zjawiska zimnego startu w przypadku tej metody akceleracji. Na rysunku 6.10 widać wynik profilowania pierwszych wykonanów kodu na stworzonych instancjach Worker'ów. W pierwszych wykonaniach widać bloki *Optimize Code*, a dalsze wykonania wykonują się już po optymalizacji. Podczas długiej przerwy pomiędzy wykonaniami proces optymalizacji zachodził dla wątku głównego.

Tab. 6.1: Przyspieszenie i jego efektywność dla metody akceleracji z wykorzystaniem Worker'ów ($S_\theta = 1, n = 1, p = 4$).

Alg.	Środ.	Przysp.	Efek.
SHT <i>non-LUT</i>	Chrome	2.78	0.69
SHT <i>non-LUT</i>	Firefox	2.89	0.72
SHT <i>non-LUT</i>	Node	3.19	0.80
SHT <i>non-LUT</i>	Deno	2.56	0.64
SHT <i>LUT</i>	Chrome	1.84	0.46
SHT <i>LUT</i>	Firefox	1.83	0.46
SHT <i>LUT</i>	Node	1.66	0.41
SHT <i>LUT</i>	Deno	1.94	0.48
CHT	Chrome	1.19	0.30
CHT	Firefox	1.29	0.32
CHT	Node	1.20	0.30
CHT	Deno	1.22	0.31



Rys. 6.9: Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem czterech Worker'ów.



Rys. 6.10: Wynik profilowania wykonania algorytmu CHT w przeglądarce Google Chrome z użyciem Worker'ów.

W tabeli 6.1 przedstawiono miarę przyspieszenia obliczeń oraz jej efektywność. Największe przyspieszenie, a zarazem jego efektywność ma miejsce w przypadku wariantu SHT *non-LUT* i środowiska NodeJS. Różnica wyników pomiędzy wariantami SHT *non-LUT* i SHT *LUT* jest znacząca i kolejny raz wskazuje na funkcje trygonometryczne i liczby zmiennoprzecinkowe jako czynnik, na który trzeba zwrócić uwagę podczas optymalizacji algorytmów. W wariantie CHT przyspieszenie jest niewielkie, ponieważ zrównolegleniu poddany został sam proces głosowania na środek okręgu. Jeśli w danych wejściowych większa liczba pikseli byłaby zapalona, etap głosowania byłby bardziej wymagający, co zwiększyłoby efektywność przyspieszenia.

Worker'y współdzielą pamięć akumulatora w postaci obiektu `SharedArrayBuffer`. Z uwagi na to, że każdy Worker działał na własnym fragmencie pamięci podczas procesu głosowania,

nie było konieczne stosowanie operacji atomowych z wykorzystaniem metod interfejsu `Atomic`. Użycie takiego interfejsu spowalniało wykonanie algorytmu w testach podczas implementacji.

6.5. GPGPU

Implementacja akceleracji z użyciem układu graficznego wymaga dostosowania algorytmu do masowego zrównoleglenia wydzielając funkcję jądra (kernel), która wykonywać się będzie równolegle na setkach rdzeni jednocześnie zawsze rozwijając wszystkie gałęzie wyrażeń warunkowych. Stworzona implementacja korzysta z mechanizmów generowania grafiki WebGL API, co nakłada dodatkowe ograniczenia w przeciwieństwie do standardowego podejścia. Kernel nie ma możliwości modyfikacji danych w pamięci współdzielonej, a jego jedynym wynikiem jest wektor liczb. W zależności od algorytmu liczby te mogą być różnie interpretowane - jako zmienne logiczne, liczby stałe i zmiennoprzecinkowe, czy też w standardowy sposób, jako kolor piksela.

6.5.1. Standard Hough Transform

Przystosowanie procesu głosowania dla wariantu SHT wymaga podejścia skoncentrowanego na generowaniu jednej wartości elementu akumulatora. Proces ten jest masowo zrównoległy przez układ graficzny równolegle generując wszystkie wartości w akumulatorze.

Jeśli jeden piksel na obrazie odwzorowywany jest na sinusoidę w przestrzeni akumulatora, a jeden punkt w przestrzeni akumulatora to jedna linia na obrazie, to odwrotnym podejściem do głosowania jest zliczenie, dla jednego elementu akumulatora, ile pikseli na obrazie przechodzi przez potencjalną linię, na którą może być on odwzorowany. Żeby uwzględnić wszystkie piksele linii, których zbocze $m > 1$, iteracja po przestrzeni obrazu odbywa się również w wariancie, w którym układ współrzędnych jest obrócony o 90° . Dlatego równania (6.4) i (6.5) obliczające sumę zapalonej pikseli zamieniają współrzędne x i y miejscami w zależności kąta nachylenia. Końcowy wzór na wartość pola akumulatora przedstawiony został na równaniu (6.6).

$$\rho(\theta) = x \cos \theta + y \sin \theta \quad (6.1)$$

$$y_a(\rho, \theta, x) = \frac{\rho - x \sin \theta}{\cos \theta} \quad (6.2)$$

$$x_a(\rho, \theta, y) = \frac{\rho - y \cos \theta}{\sin \theta} \quad (6.3)$$

$$A_x(\rho, \theta) = \begin{cases} \sum_{x=0}^{w-1} I(x, y_a(\rho, \theta, x)), & \rho \in \left[\frac{\pi}{4}, \frac{3\pi}{4}\right) \vee \rho \in \left[\frac{5\pi}{4}, \frac{7\pi}{4}\right) \\ 0, & \text{dla pozostałych } \rho \end{cases} \quad (6.4)$$

$$A_y(\rho, \theta) = \begin{cases} \sum_{y=0}^{h-1} I(x_a(\rho, \theta, y), y), & \rho \in \left[0, \frac{\pi}{4}\right) \vee \rho \in \left[\frac{3\pi}{4}, \frac{5\pi}{4}\right) \vee \rho \in \left[\frac{7\pi}{4}, 2\pi\right) \\ 0, & \text{dla pozostałych } \rho \end{cases} \quad (6.5)$$

$$A(\rho, \theta) = A_x(\rho, \theta) + A_y(\rho, \theta) \quad (6.6)$$

gdzie: x, y — współrzędne piksela obrazu;

$I(x, y)$ — wartość piksela obrazu.

A_x, A_y — częściowa zawartość akumulatora w zależności od nachylenia zbocza prostej;
 A — zawartość akumulatora.

w, h — szerokość i wysokość obrazu;

x_a, y_a — funkcje obliczające drugą współrzędną linii na podstawie odwzorowania z akumulatora;

ρ — odległość prostej od środka układu współrzędnych;

θ — obrót prostej od wokół układu współrzędnych;

Zbudowanie biblioteki nie wymagało dodatkowych zmian konfiguracji bazowej narzędzia Webpack. Do implementacji obliczeń z wykorzystaniem WebGL wykorzystano bibliotekę *gpu.js*. Listing 6.6 przedstawia funkcję tworzącą kernel dla wariantu SHT*non-LUT*. Jedynym argumentem funkcji kernela jest tablica z wejściowym obrazem. Pozostałe parametry dostarczane są jako stałe, co przyspiesza do nich dostęp. Sprawia to jednak, że dla każdego zestawu opcji istnieje konieczność stworzenia nowego kernela. W implementowanym algorytmie kernele zapisywane są w mapie, której kluczem są wartości opcji. Kernel zwraca jedną liczbę stanowiącą wynik równania (6.6).

Na listingu również zobaczyć możemy utworzenie funkcji za pomocą konstruktora, która zwraca właściwa funkcję i jest od razu wykonywana - `new Function('return function () { testImage/* : number[] */{}/*...*/})();`. Jest to niezbędne, aby mieć pełną kontrolę nad wejściem do mechanizmów transpilacji biblioteki *gpu.js*, ponieważ kod w procesie budowania biblioteki jest minifikowany, co zmienia nazwy zmiennych i stosuje skrócone notacje, z którymi biblioteka *gpu.js* ma problem z transpilacją. Na listingu 6.5 widać przykład zminifikowanego kodu błędnie transpilowanego do języka ESSL, gdzie problem stanowi połączenie post-inkrementacji z wyrażeniem logicznym.

Listing 6.5: Przykład minifikacji kodu błędnie transpilowanego do języka ESSL przez bibliotekę *gpu.js*.

```
1 // before
2 if(offset < this.constants.width * this.constants.height &&
   testImage[offset] == 1) {
3     acc+=1;
4 }
5 // after
6 t<this.constants.width*this.constants.height&&1==e[t]&&a++;
```

6.5.2. Circle Hough Transform

W wariantie CHT zaimplementowano łącznie 3 kernele. Dwa z nich odpowiedzialne są za obliczenie splotu obrazu z operatorem Sobela, aby wyznaczyć gradient krawędzi. Pętle iterujące po macierzy 3x3 operatora zostały rozwinięte do sześciu wyrażeń w celu redukcji narzutu związanego z obsługą pętli. Kernele te zwracają wynik w postaci obiektu tekstuury, której powiązane dane znajdują się bezpośrednio na GPU. Tekstyry te są następnie wejściem do kernela wyznaczającego wartość pola akumulatora w procesie głosowania na środki okręgów. Dzięki takiemu złożeniu kerneli dane splotów nie przechodzą przez CPU, aby trafić z powrotem do układ graficznego. Wartość piksela wyznaczana jest na podstawie liczby linii, o długości wyznaczanej na podstawie maksymalnego i minimalnego wyszukiwanego promienia, przechodzących przez dany punkt w przestrzeni akumulatora.

6.5.3. Wyniki pomiarów

Na rysunku 6.11 przedstawiono wyniki pomiarów czasu wykonania algorytmów zaimplementowanych z wykorzystaniem biblioteki *gpu.js*. Widać, że metoda ta cechuje się bardzo niestabilnym z odchyleniami od liniowego, czasem wykonania. Brak dużego odchylenia standardowego sugeruje, że czas zależeć może od ogólnego obciążenia systemu oraz od optymalizacji wykorzystanych zasobów procesora graficznego, co sugeruje podobny czas wykonania dla $S_\theta \in \{5, 6, 7\}$ w wariantie SHT *LUT*. Dla SHT w każdym środowisku i wariantie algorytmu udało się przyspieszyć czas wykonania bądź osiągnąć porównywalny względem wykonania sekwencyjnego

Listing 6.6: Funkcja tworząca kernel dla wariantu SHT *non-LUT*

```

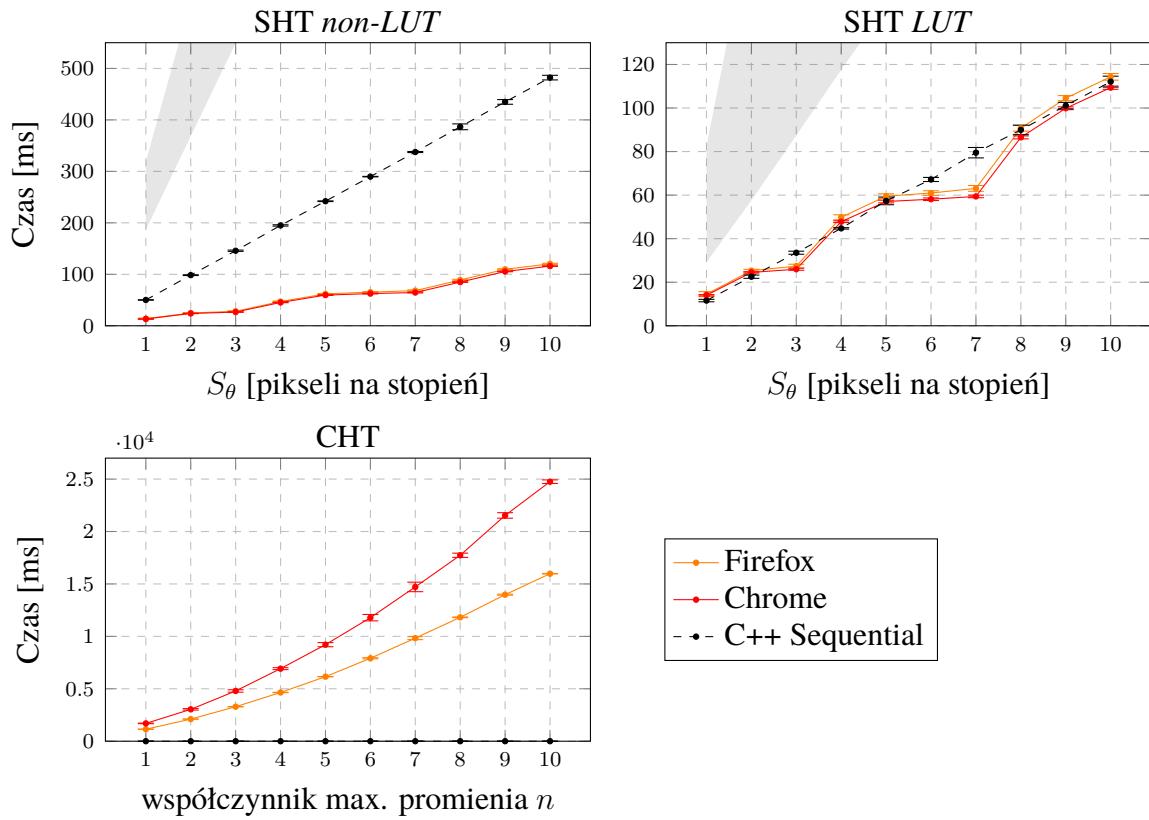
1 import { GPU, IKernelRunShortcutBase } from "gpu.js";
2
3
4 export const gpu = new GPU();
5 export function createSHTSimpleKernel(
6     hsWidth: number, hsHeight: number, width: number, height: number,
7     samplingRho: number, samplingTheta: number
8 ) {
9     return gpu
10    .createKernel(
11        new Function('return function (testImage/* : number[] */) {
12            // ...
13            let acc = 0;
14            // ...
15            return acc;
16        })(),
17        {
18            output: [hsWidth * hsHeight],
19            constants: {hsWidth, width, height, samplingRho, samplingTheta},
20        }
21    )
22    .setLoopMaxIterations(Math.max(width, height))
23    .setOptimizeFloatMemory(true) as IKernelRunShortcutBase<Float32Array>;
24 }

```

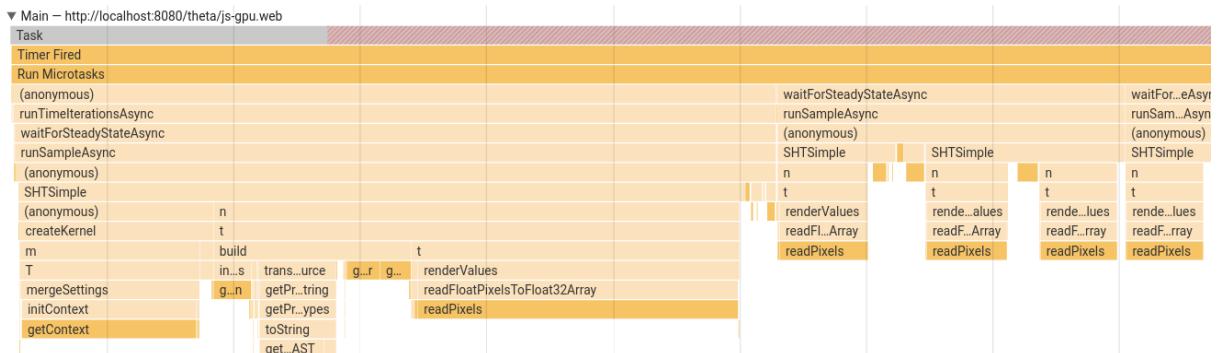
w języku C++. Przyspieszenie wynosi $3.81 \times$ dla wariantu SHT *non-LUT* i oscyluje w okolicy $1 \times$ dla SHT *LUT* oraz $S_\theta = 1$. Ze względu na odwrotne podejście do generowania wartości w akumulatorze oraz na zmniejszoną precyzyje obliczeń dla liczb zmiennoprzecinkowych w procesorach graficznych, różnica pomiędzy wygenerowanym akumulatorem dla tej metody, a wariantem sekwencyjnym jest znacząca i została pokazana na rysunku 6.3c. Nie wpływa ona jednak znacząco na ocenioną empirycznie jakość detekcji.

Wąskim gardłem w metodach akceleracji z wykorzystaniem układów graficznych jest komunikacja CPU z GPU. Wysyłanie i odbieranie danych zajmuje nieproporcjonalnie dużo czasu, co widać na rysunku 6.12. Widać na nim również proces transpilacji kodu kernela (bloki `createKernel` i `build`). Konieczność zbudowania kernela zwiększa czas zimnego startu, który jest szczególnie duży w przypadku tej metody akceleracji, co zostanie opisane w sekcji 6.6.

W przypadku wersji CHT nie udało się uzyskać przyspieszenia, a dodatkowo implementacja w taki sam sposób zwiększyła złożoność obliczeniową, która stała się ponad-liniowa. Dla każdego punktu akumulatora wszystkie punkty wokół niego na obrazie, nie dalej niż maksymalnie szkany promień, muszą zostać sprawdzone pod kątem generowania linii, która przecina generowany punkt akumulatora. Na tej podstawie można oszacować złożoność jako $O(n^2)$. Dla dużych promieni rzędu 220 pikseli, czas wykonania potrafi wynieść 25s. W ramach pracy nie analizowano dalej tego problemu, a wszelkie próby poprawy złożoności algorytmu nie przyniosły efektu. Doświadczenie z udaną akceleracją wariantu SHT algorytmu pozwala sądzić, że zaimplementowanie wersji z jednym trójwymiarowym akumulatorem przyniosłoby oczekiwane rezultaty. Ograniczeniem w takiej implementacji byłby rozmiar akumulatora. W obydwu przeglądarkach maksymalny rozmiar wyjścia to 16384×16384 . Rozwiążaniem tego problemu może być podział wykonania na części, gdzie każda z nich generuje maksymalny możliwy fragment akumulatora.



Rys. 6.11: Wyniki pomiarów czasu wydajności dla wykonania SHT i CHT z wykorzystaniem biblioteki *gpu.js*.



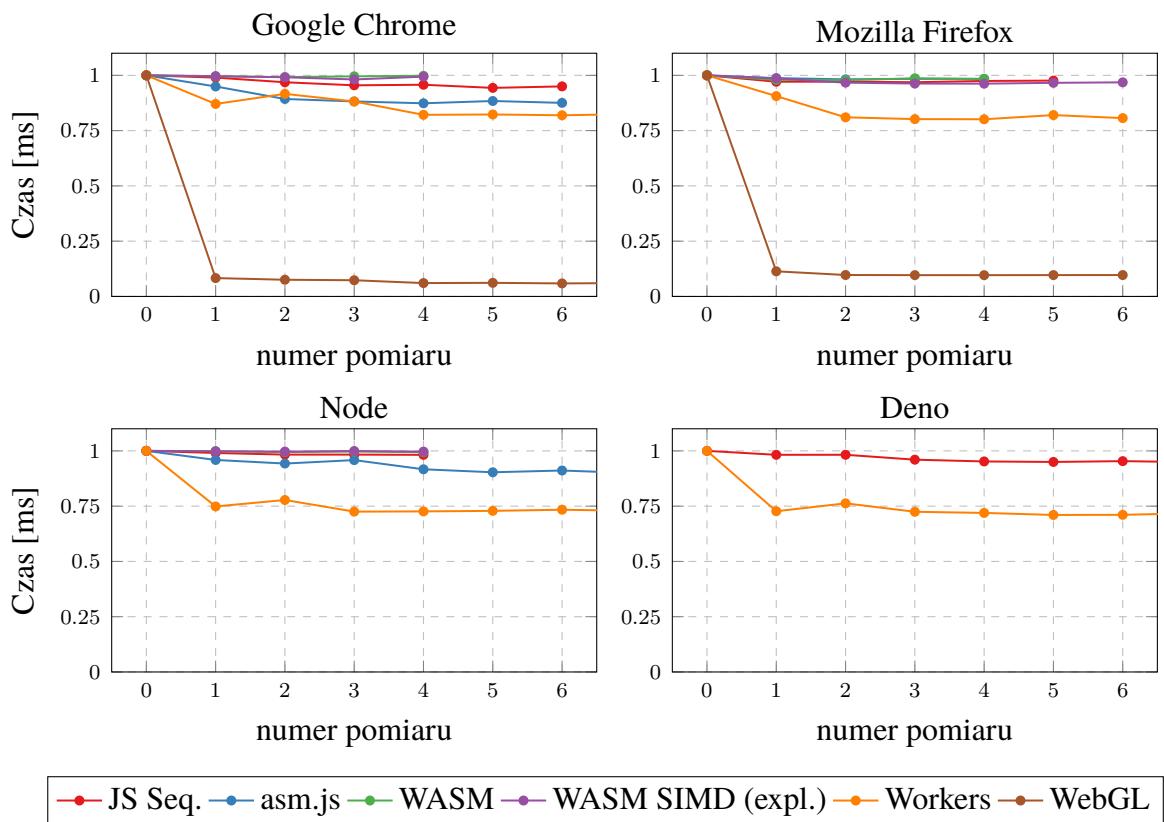
Rys. 6.12: Wynik profilowania wykonania algorytmu CHT w przeglądarce Google Chrome z wykorzystaniem biblioteki *gpu.js*.

6.6. Problem zimnego startu

Problem zimnego startu związany jest z narzutem czasowym występującym przy uruchomieniu środowiska, komplikacji i optymalizacji kodu, czy też uruchomienia dodatkowych mechanizmów obsługujących wątki lub komunikację z GPU. W zależności od sytuacji może być on pomijalny, ale i krytyczny. Pominąć można go w procesie prototypowania, kiedy wielokrotnie uruchamiamy środowisko po zamianach w algorytmie co jakiś czas. Wtedy dodatkowy czas nie stanowi problemu, oczywiście jeśli jest akceptowalnie krótki. Krytyczny okazać się może w produkcyjnych rozwiązańach *serverless*, gdzie każdemu wykonaniu funkcji towarzyszy osobna instancja środowiska, a optymalizacje nie są przeprowadzane. Oczywiście w zależności od obciążenia pojedyncza instancja środowiska może obsługiwać wiele zapytań, co przy odpowiednio długim czasie trwania pozwala na optymalizację kodu i komplikację JIT.

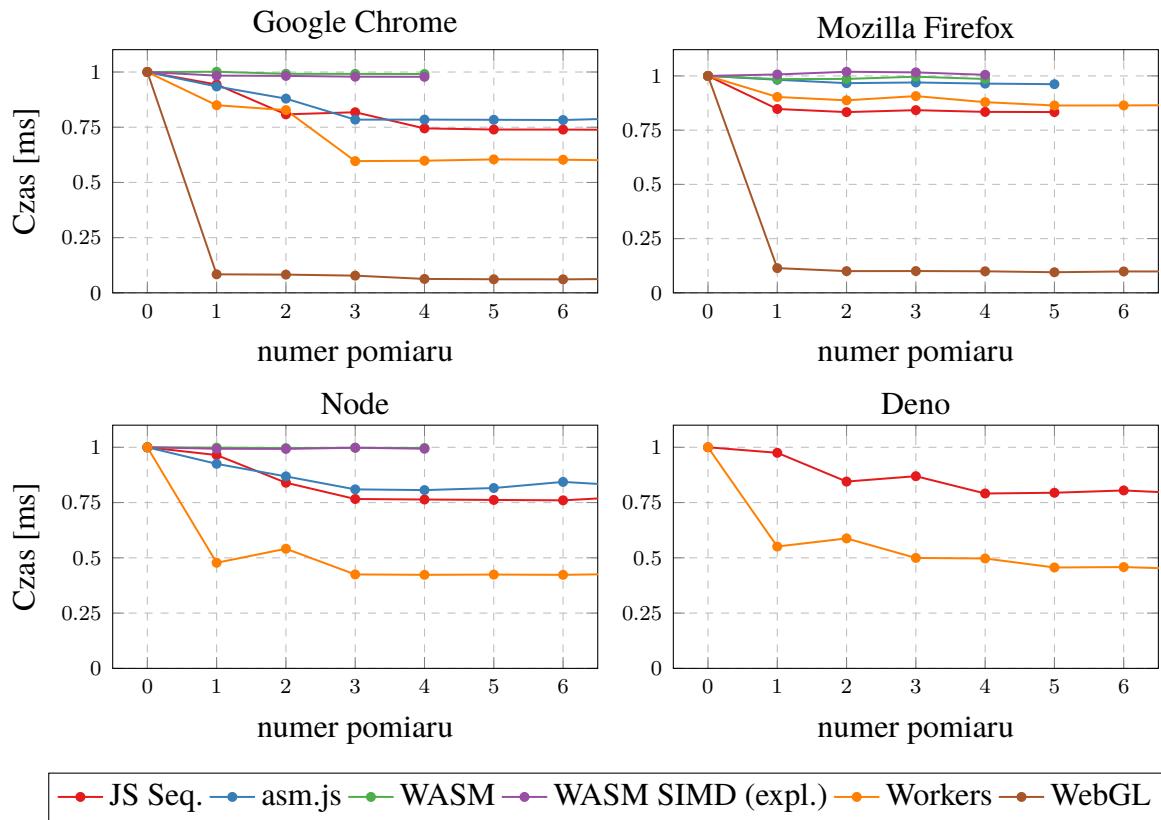
Każda z badanych metod akceleracji posiada inną charakterystykę zimnego startu. Na rysunku 6.13 i 6.14 przedstawiono czasy wykonania algorytmu SHT kolejno *non-LUT* i *LUT* znormalizowane do przedziału $[0, 1]$ dla $S_\theta = 1$. Pomiary wykonywano, dopóki współczynnik wariancji dla okna o długości 5 był mniejszy lub równy 0.01. Na wykresach pokazano tylko sześć pierwszych pomiarów, ponieważ to dla nich występuje największa zmiana czasu wykonania. Widać, że największa różnica w czasie występuje dla metod wywoływanego asynchronicznie - Worker'ów oraz WebGL. Metody te mają problem z osiągnięciem stabilnych czasów również po stabilizacji wykrytej przez metrykę współczynnika wariancji.

Metody wykorzystujące WebAssembly nie zyskują na wydajności w kolejnych pomiarach. Jest to zrozumiałe, ponieważ skompilowany kod nie jest już optymalizowany przez silnik. Oznacza to jednak również, że widocznej optymalizacji nie ulegają mechanizmy powiązania kodu JavaScript i WebAssembly - głównie mechanizmy kopiowania danych. Dla wariantu SHT *LUT*, dla metod akceleracji wykonujących kod JavaScript, wydać wyraźną poprawę wydajności w kolejnych pomiarach. Kolejny raz widać zatem, że funkcje trygonometryczne i ich wielokrotne wywoływanie spowalniają czas wykonania, a dodatkowo wpływają negatywnie na optymalizację związaną z pierwszymi uruchomieniami algorytmu w ramach tej samej instancji środowiska.



Rys. 6.13: Wyniki pomiarów pierwszych czasów wykonania algorytmu SHT *non-LUT* znormalizowane do przedziału $[0, 1]$ dla $S_\theta = 1$.

Analizując przyspieszenie związane z optymalizacją na rysunku 6.15 przedstawiającym pierwsze czasy wykonania algorytmu CHT dla $n = 1$, możemy zobaczyć dużo większe przyspieszenie dla kolejnych uruchomień niż dla algorytmów SHT. Z racji na mniejszy rozmiar problemu i krótszy czas iteracji implementacja ta lepiej pokazuje możliwości optymalizacji środowisk, ponieważ obliczenia nie są skupione tak bardzo na powtarzających się iteracjach. W wariancie sekwencyjnym wspólną cechą środowisk opartych na silniku V8 jest zwiększenie czasu wykonania dla drugiego wywołania funkcji, gdzie kolejne są już optymalizowane. Widać dla metod wykorzystujących WebAssembly, optymalizację instrukcji wywołania modułów WebAs-



Rys. 6.14: Wyniki pomiarów pierwszych czasów wykonania algorytmu SHT LUT znormalizowane do przedziału $[0, 1]$ dla $S_\theta = 1$.

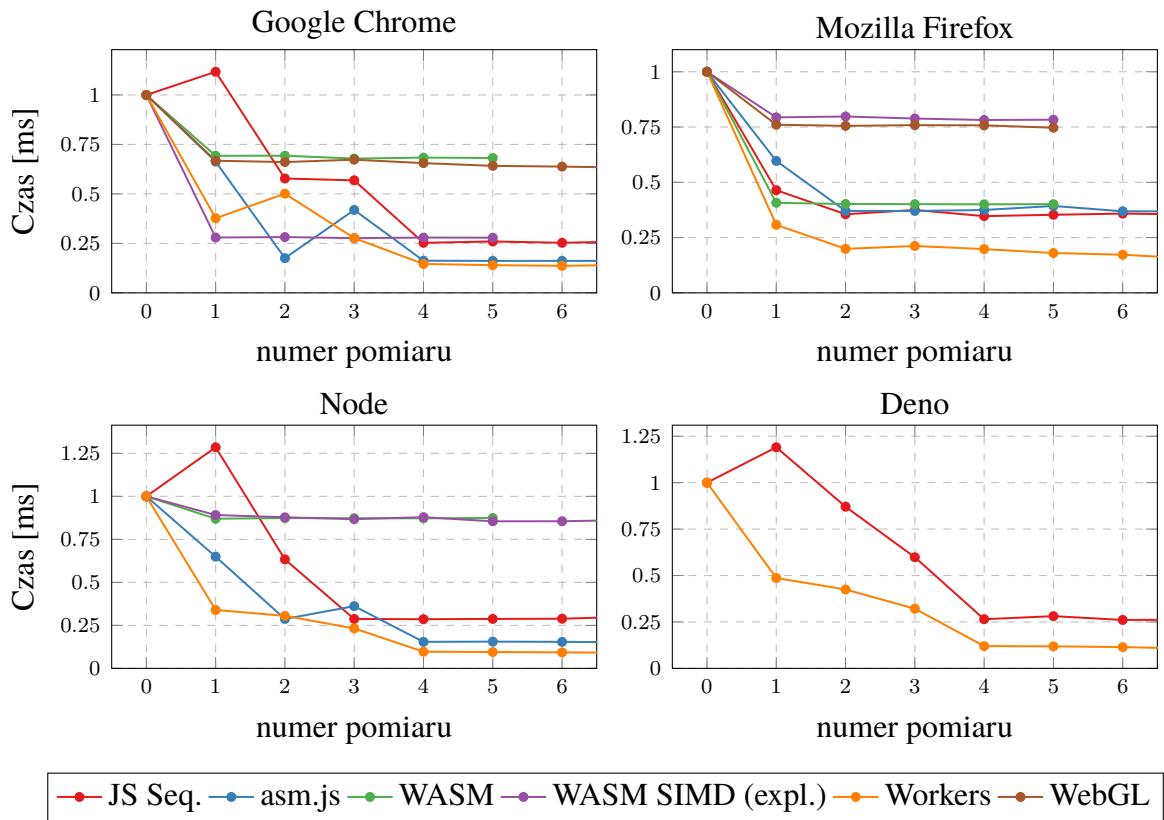
sembly, która raz wykonana daje stałe czasy w kolejnych wykonaniach algorytmu. Środowiska oparte na silniku V8 cechują się ogólną skutecznością w optymalizacji kodu JavaScript w porównaniu do silnika SpiderMonkey środowiska przeglądarki Mozilla Firefox.

6.7. Przykład przetwarzania obrazu z kamery w przeglądarce internetowej

Implementacja przetwarzania obrazu z kamery w przeglądarce wykonana została w celu zbadania i wykrycia ewentualnych problemów z integracją zbudowanych bibliotek z aplikacją SPA oraz z procesem pozyskiwania obrazu z kamery. Obraz na potrzeby testów dostarczany był w formie strumieniowania plików *.mp4 o rozdzielcości 720p do wirtualnego urządzenia kamery stworzonego w wykorzystaniem narzędzia *v4l2loopback*.

Pobranie obrazu z kamery wymaga skorzystania z WebRTC API, które pozwala uzyskać dostęp do kamery i mikrofonu użytkownika i zarządzać nimi w abstrakcjach strumieni, które zawierają ścieżki audio i wideo. Jednym ze standardowych sposobów na uzyskanie obrazu do przetworzenia jest utworzenie obiektu HTML5 `<video>` z atrybutem `srcObject`, który przyjąć musi wartość obiektu zwróconego przez zapytanie `navigator.MediaDevices.getUserMedia(...)`. Następnie, na podstawie obiektu `<video>`, na obiekcie `<canvas>` narysować możemy wyświetlana obecnie klatkę. Aby pobrać taką klatkę do przetwarzania w formie binarnej pliku obrazu w wybranym formacie należy użyć metody `canvas.toBlob(...)`. Aby pobrać wartości konkretnych pikseli należy użyć metody kontekstu rysowania obiektu `<canvas>` - `ctx.getImageData(...)`.

Przechodząc od obrazu źródłowego z kamery do wartości konkretnych pikseli musimy przejść przez formy pośrednie obiektu `<video>` i `<canvas>`. Wpływ to na wydajność i wygodę użytko-

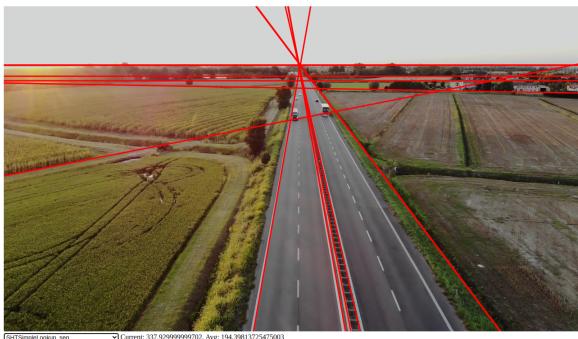


Rys. 6.15: Wyniki pomiarów pierwszych czasów wykonania algorytmu CHT znormalizowane do przedziału $[0, 1]$ dla $n = 1$.

kowania, która jest zdecydowanie mniejsza, jeśli do niezbędnych operacji trzeba wykorzystać obiekty modelu DOM. Rozwiązaniem, które nie wykorzystuje modelu DOM jest zaimplementowane w ramach przykładu ImageCapture API. Działa on jako wrapper ścieżki wideo pochodzącej z strumienia uzyskanego za pomocą metody `getUserMedia` i dzięki metodzie `grabFrame` możemy pobrać obiekt `ImageBitmap`. Rozwiązanie to wciąż wykorzystuje obiekt `<canvas/>` do narysowania na nim obrazu z obiektu `ImageBitmap`, a następnie pobrania poszczególnych pikseli. ImageCapture API jest specyfikacją eksperymentalną, która wspierana jest tylko w przeglądarkach opartych na Chromium.

Przetwarzanie zostało zaimplementowane dla obydwu algorytmów i przetestowane z dwoma obrazami wejściowymi (rys. 6.16). Wejściem algorytmu detekcji kształtów wykorzystujący transformację Hough'a jest binarny obraz z wykrytymi krawędziami. Obraz z kamery musi zatem zostać pozyskany, narysowany na obiekcie `<canvas/>`, przekształcony do skali szarości, a następnie poddany operacji wykrywania krawędzi. Implementacja wykorzystuje bibliotekę `image-js`[28] oraz `canny-edge-detector`[26], które odpowiadają kolejno za ogólne przetwarzanie obrazów oraz za implementację operatora Canny, który służy do wykrywania krawędzi na obrazie.

Na listingu 6.7 przedstawiona została główna pętla przetwarzania obrazu. Na podstawie tego przykładu i wyników profilowania wykonania widać zatem, jak ważne jest zapewnienie wydajności całego łańcucha przetwarzania. Na rysunku 6.17 widać, że z czasu przetwarzania, które trwało 173ms, przeglądarka powróciła 120ms, czyli zdecydowaną większość, na wykonanie zewnętrznej i nieozymalizowanej pod względem wydajności metodzie `CannyEdgeDetector` (69% czasu). Wstępne przetwarzanie obrazu trwało 7.5ms (4% czasu), a implementowana detekcja linii 40ms (23% czasu).



(a) SHT [23]



(b) CHT [24]

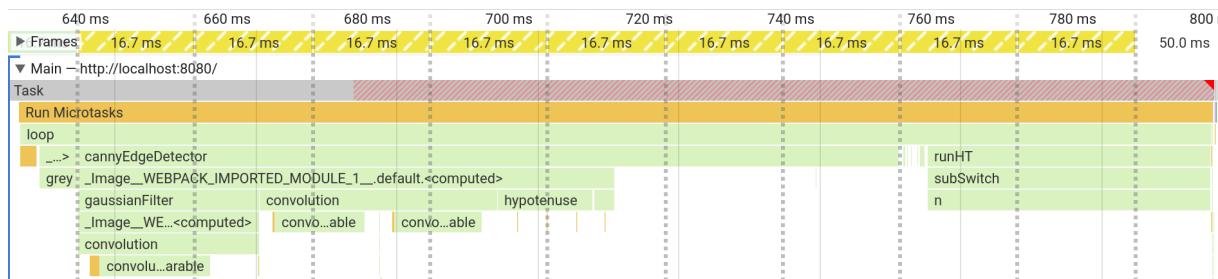
Rys. 6.16: Wizualizacja wykrytych kształtów na podstawie obrazu z kamery w przeglądarce internetowej.

Listing 6.7: Funkcja przetwarzająca w pętli kolejne klatki obrazu z kamery.

```

1  async function loop() {
2    if (resultCtx && imageCapture.value) {
3      const imageBitmap = await imageCapture.value.grabFrame();
4      resultCtx.drawImage(imageBitmap, 0, 0);
5      image.data.set(resultCtx.getImageData(0, 0, width, height).data);
6      const grayImage = image.grey();
7      const canny = cannyEdgeDetector(grayImage, {
8        lowThreshold: 50,
9        highThreshold: 50,
10       }) as Image;
11      const inputImage = canny.data.map((v) => (v > 0 ? 1 : 0))
12      await runHT(inputImage);
13    }
14    !stopLoop ? requestAnimationFrame(loop) : console.log("Loop_stop");
15  }

```



Rys. 6.17: Wynik profilowania wykonania sekwencyjnego algorytmu SHT *LUT* wraz z procesem pozyskania obrazu, przetwarzania wstępnego, wykrywania krawędzi i wyświetlenia wyników.

Ta próba implementacji procesu od pozyskania obrazu, przez jego przetwarzanie, aż do wyświetlenie wyników pokazuje nieprzystosowanie środowiska przeglądarki internetowej do przetwarzania obrazów. Również wiele do życzenia pozostawia cały ekosystem bibliotek, którym często brakuje kompatybilności i popularności na tle ich odpowiedników w języku Python.

Rozdział 7

Podsumowanie

Czy język JavaScript, jego ekosystem i środowiska są przystosowane do wykonywania intensywnych obliczeń numerycznych? Czy asynchroniczny model wykonania wspiera, czy też przeszkadza w wykonywaniu obliczeń? Czy środowiska zapewniają odpowiednie mechanizmy, aby wygodnie ładować dane, przyspieszać obliczenia oraz wyświetlać wyniki? Która z metod akceleracji jest najbardziej wydajna oraz jak wygląda ich kompatybilność wśród środowisk. Czy istnieje możliwość bezkompromisowego utworzenia biblioteki kompatybilnej ze wszystkimi środowiskami? Na te i inne pytania postarano się uzyskać odpowiedź w tej pracy. Nie skorzystano w tym celu z syntetycznych testów wydajności środowisk języka JavaScript i dostępnych metod akceleracji obliczeń oraz równoległej do nich analizy ich kompatybilności i metod budowania bibliotek. Zamiast tego zastosowano podejście całościowe. Zaimplementowano algorytmy przetwarzania obrazów bazujące na transformacji Hough'a. Jest ona wykorzystywana w algorytmach detekcji kształtów parametrycznych i nieparametrycznych. W zależności od kształtu i wydajności powstały różne jej warianty. W ramach tej pracy zaimplementowano wariant *Standard* (Standard Hough Transform, SHT) i *Circle* (Circle Hough Transform, CHT) z wykorzystaniem metody analizy gradientu do detekcji środków okręgów. Jako format eksportowanych bibliotek przyjęto moduły ES6 (ESModules).

Dla każdej z analizowanych metod akceleracji przeprowadzono pomiary czasu wykonania we wszystkich udostępniających ją środowiskach. Do badanych środowisk należały przeglądarki internetowe Google Chrome i Mozilla Firefox oraz środowiska serwerowe NodeJS i Deno. Wszystkie z nich korzystają z V8 jako silnika języka JavaScript, co razem z implementacją dla środowiska z silnikiem SpiderMonkey - przeglądarki Mozilla Firefox, umożliwiło ich wzajemne porównanie.

Jako definicję metody akceleracji przyjęto wszelkie działania, które potencjalnie mogą przyspieszyć wykonania algorytmu. Zaimplementowano algorytmy w wariancie sekwencyjnym. Algorytm SHT dodatkowo został zaimplementowany z wykorzystaniem stabilizowanych wartości funkcji trygonometrycznych, co w dużym stopniu zwiększyło wydajność. Do testów pozostałych metod akceleracji wykorzystano zatem trzy implementacje - dwie algorytmu SHT i jedną CHT.

Podstawową metodą akceleracji była poprawa wydajności wykonania sekwencyjnego. kolejną z nich stanowiło wykorzystanie natywnych rozszerzeń środowiska NodeJS, których bazę stanowił kod C++. Następną z nich było WebAssembly, którego bazę również stanowił kod w języku C++ komplikowany do niskopoziomowego języka kompatybilnego z wszystkimi środowiskami. Oprócz standardowej komplikacji, kod C++ skompilowano do kodu *asm.js* oraz uruchomiono procesy automatycznej wektoryzacji wykonania podczas komplikacji do WebAssembly, aby wykorzystać wektorowe rejestrów i instrukcje procesora (SIMD). Przeprowadzono również komplikację po manualnej wektoryzacji kodu. Pierwszą z metod wykorzystujących współbieżne wykonanie jest implementacja Worker'ów, gdzie zrównoleglone zostały iteracje najbardziej ob-

ciążonych pętli w algorytmach. Ostatnią z analizowanych metod było wykorzystanie potoku graficznego WebGL do przeprowadzenia masowo równoległych obliczeń.

Dla zaimplementowanych metod akceleracji podjęto próbę stworzenia biblioteki kompatybilnej ze wszystkimi środowiskami, co nie zawsze było możliwe. Zadbano jednak, aby każda z bibliotek implementowała ten sam interfejs. Było to zapewnione przez transpilator języka TypeScript, w którym to zaimplementowano wszystkie elementy bibliotek. Podczas transpilacji plików źródłowych do języka JavaScript, korzystał on z tych samych symboli definiujących interfejs eksportowanych przez bibliotekę obiektów i funkcji. Biblioteka zbudowana dla wariantu sekwencyjnego jest kompatybilna ze wszystkimi środowiskami, ponieważ opiera się tylko i wyłącznie na standardowych mechanizmach języka JavaScript. Metoda wykorzystująca natywne rozszerzenia została zbudowana tylko dla środowiska NodeJS, które jako jedyne z badanych udostępnia ją w tej formie. Środowisko Deno również pozwala na zbudowanie i wykonanie natywnych rozszerzeń, jednak bazą dla nich jest język Rust. Nie zostały one zaimplementowane, ponieważ implementacja algorytmów w tym języku wykracza poza zakres tej pracy, która skupia się na bazowym kodzie w języku C++. Dla tej metody, niezbędne było utworzenie warstwy powiązania, która zarządzała kopiowaniem danych oraz wywoływaniem właściwych funkcji. Narzędzie *node-gyp* wraz z Node-API niestety nie udostępniają automatycznych mechanizmów konwersji obiektów i tablic z języka JavaScript na ich odpowiedniki w języku C++. Rodziło to konieczność ręcznego pobierania wartości do struktur na podstawie obiektu kontekstu wykonania. Kolejna zaimplementowana metoda wykorzystuje WebAssembly. Również tam niezbędne było utworzenie warstwy powiązania, jednak narzędzie *Emscripten* udostępnia mechanizm automatycznej konwersji obiektów po zdefiniowaniu ich struktury, co zdecydowanie upraszcza proces implementacji. Jako, że proces inicjalizacji modułu WebAssembly musi odbyć się asynchronicznie, oraz nie zaimplementowano w kodzie C++ mechanizmów łączenia zagnieżdzonych struktur, aby zapewnić domyślne wartości obiektom opcji, w implementacji konieczne okazało się opakowanie modułu WebAssembly w obiekt, który zajmuje się jego inicjalizacją oraz dostarcza domyślne dane opcji. Samo wywołanie funkcji implementowanych algorytmów odbywa się synchronicznie. Kompilacja z użyciem narzędzia *Emscripten*, aby uzyskać zakładany format modułu, wymaga użycia dodatkowych flag, co ma miejsce również w przypadku wariacji formatów wyjściowych (*asm.js*) i używanych instrukcji (SIMD). Również i w tym wypadku wszystkie środowiska, na których przeprowadzone zostały badania tej metody akceleracji, były kompatybilne z tym samym kodem źródłowym.

Kompatybilności jednego kodu źródłowego ze wszystkimi środowiskami nie udało się uzyskać w przypadku implementacji Worker'ów. Pomimo użycia biblioteki Comlink, która nałożyła warstwę abstrakcji ułatwiając komunikację głównego wątku z Worker'ami, konieczne było utworzenie dwóch wersji kodu osobno dla środowisk przeglądarek internetowych i osobno dla NodeJS. Środowisko Deno również wymagało utworzenia osobnej implementacji, ale w odróżnieniu od pozostałych środowisk, gdzie biblioteka została zbudowana z plików źródłowych, jest ono w stanie wykonać pliki w języku TypeScript. Konieczność budowania wielu wersji biblioteki wymusza niekompatybilny pomiędzy środowiskami przeglądarek, Deno i NodeJS interfejs obsługi Worker'ów. Dlatego też, aby zapewnić możliwość współdzielenia kodu samego algorytmu dla tej metody akceleracji, niezbędne jest użycie adapterów oraz zbudowanie osobnych wersji biblioteki dla każdego ze środowisk.

Kolejną metodą niekompatybilną ze wszystkimi badanymi środowiskami jest wykorzystanie GPGPU poprzez użycie WebGL API, czyli potoku graficznego do obliczeń ogólnego przeznaczenia. Jest ona możliwa do wdrożenia natywne jedynie w przeglądarkach internetowych. W NodeJS istnieje konieczność zainstalowania zewnętrznej implementacji WebGL, a środowisko Deno nie wspiera jej wcale. Metoda ta, w przeciwieństwie do użycia GPGPU w przystosowanych do tego rozwiązańach, nie umożliwia interakcji z pamięcią wspólnie dzieloną pomiędzy wątkami, a wynikiem obliczeń jest tylko kolor piksela. Jako liczba może być on jednak dowolnie

interpretowany. Metoda ta, pomimo swoich wad i wykorzystania potoku niezgodnie z jego przeznaczeniem, może przynieść zdecydowaną poprawę wydajności algorytmów. Muszą one jednak być podatne na masowe zrównoleglenie oraz nie korzystać z pamięci współdzielonej. Przykładem najprostszego z nich może być mnożenie macierzy, czy operacja splotu. Dołączona biblioteka *gpu.js*, służąca do obsługi interakcji z potokiem graficznym, transpiluje kod JavaScript na kod programu *Fragment Shader* potoku graficznego. Proces budowania musi uwzględnić mechanizmy minifikacji kodu, ponieważ mogą one interferować z ową transpilacją i produkować kod i jego konstrukcje niezrozumiałe przez bibliotekę *gpu.js*.

W tabeli 7.1 przedstawiono podsumowanie czasów wszystkich pomiarów wydajności dla wartości próbkowania $S_\theta = 1$ dla wariantu SHT algorytmu i współczynnikiem maksymalnego promienia $n = 10$ dla CHT. Wykonanie dla algorytmu CHT z metodą WebGL nie zostało uwzględnione, ponieważ w implementacji wystąpiły błędy, co opisano w sekcji 6.5.2. W ogólnym podsumowaniu najbardziej wydajnym środowiskiem okazało się Google Chrome. Kolejnymi środowiskami pod względem wydajności są NodeJS i Deno, których wydajność jest porównywalna. Najwolniejszym ze środowisk jest przeglądarka Mozilla Firefox. Osiągnęła ona wyniki porównywalne z Google Chrome jedynie w przypadku akceleracji z użyciem WebGL, gdzie ciężar obliczeń oddelegowany jest do GPU, a przeglądarka odpowiedzialna jest za przekazanie danych i skompilowanie programów shader'ów.

Najlepszą z metod, których bazę stanowił kod w C++, jest użycie natywnych rozszerzeń w środowisku NodeJS. WebAssembly, również przynosi poprawę wydajności we wszystkich środowiskach w porównaniu do wykonania sekwencyjnego. Wykorzystanie automatycznej wektoryzacji kodu, aby wykorzystać instrukcje i rejesty wektorowe nie przyniosło rezultatów w żadnym środowisku w porównaniu do zwykłego WebAssembly. Manualna wektoryzacja, która wymagała modyfikacji kodu C++ i użycia specjalnych instrukcji operujących na tych rejestrach, przyniosła poprawę wydajności dla algorytmów SHT, którego dane wejściowe wymuszały większą intensywność obliczeń w modyfikowanych pętlach. Dla algorytmów CHT manualna wektoryzacja nie przyniosła żadnych rezultatów, bądź nawet spowolniła wykonanie. Pozwala to wysnuć wniosek, że manualna wektoryzacja dołożyła czas związany z jej obsługą, który zrekompensowany został poprzez intensywne jej wykorzystanie, co miało miejsce dla testów algorytmów SHT i nie dla CHT. Metoda wykorzystująca *asm.js* jako jedyna nie przyniosła poprawy wydajności. Spowodowane to być może sposobem budowania biblioteki z kodem *asm.js*, który uniemożliwił prawidłową jego interpretację przez przeglądarki, ale również to, że *asm.js* jest natywnie wspierany - komplikowany Ahead-of-Time jedynie w przeglądarce Mozilla Firefox.

Największe przyspieszenie umożliwiło wykorzystanie Worker'ów oraz WebGL z oczywistą przewagą tego drugiego. Wyniki te nie dziwią, ponieważ naturalnym jest osiągnięcie lepszych wyników wykonując obliczenia równolegle. Przewaga WebGL jest o tyle specyficzna, że implementowany algorytm musi dać się masowo zrównoleglić bez wpływu na opisane wcześniej wady tej metody, podczas gdy nie trzeba się o to martwić w przypadku Worker'ów.

Badania te pozwalają stwierdzić, że środowiska języka JavaScript są gotowe, aby zapewnić wysoką wydajność obliczeń numerycznych. Wszystkie zaimplementowane metody akceleracji poza *asm.js* zmniejszyły lub nie zmieniły czasu wykonania w ramach środowiska. Dalsze prace nad testowaniem wydajności mogą sprawdzać działanie hybrydowych metod akceleracji, czego przykładem może być zastosowanie Worker'ów, a w nich WebAssembly. Dodatkowo można zastosować podejście SIMD. Niezbędna jest również analiza implementacji algorytmu CHT, który osiągnął niewspółmiernie długie czasy wykonania podczas testów metody opartej na WebGL. Na podstawie ogólnego rozumnego w możliwościach każdej z metod akceleracji, implementacji algorytmów oraz sposobów budowania bibliotek, możliwym staje się wybór interesujących elementów składowych algorytmów, ich ekstrakcja i izolowane testy syntetyczne, co może również być przedmiotem dalszych badań. Możliwym kierunkiem dalszych badań z pewnością powinna stać się implementacja i analiza wydajności tych algorytmów dla akceleracji obliczeń z użyciem

Tab. 7.1: Porównanie zaimplementowanych metod akceleracji dla badanych środowisk. Wykonanie dla SHT przy $S_\theta = 1$, dla CHT przy $n = 10$. Wykonanie dla CHT WebGL nie zostało uwzględnione.

LUT	Alg.	Metoda	Czas [ms]			
			Chrome	Firefox	Node	Deno
	CHT	JS Sequential	8.80 (1.00)	12.65 (1.44)	8.57 (0.97)	8.52 (0.97)
	CHT	C++ addon	- (-)	- (-)	4.35 (-)	- (-)
	CHT	asm.js	18.36 (1.00)	23.08 (1.26)	18.82 (1.03)	- (-)
	CHT	WASM	6.97 (1.00)	12.58 (1.80)	6.53 (0.94)	- (-)
	CHT	WASM SIMD (impl.)	6.98 (1.00)	12.55 (1.80)	6.66 (0.95)	- (-)
	CHT	WASM SIMD (expl.)	6.74 (1.00)	13.61 (2.02)	7.21 (1.07)	- (-)
	CHT	Workers	6.64 (1.00)	7.67 (1.16)	6.45 (0.97)	5.20 (0.78)
	SHT	JS Sequential	186 (1.00)	319 (1.72)	302 (1.62)	301 (1.62)
	SHT	C++ addon	- (-)	- (-)	136 (-)	- (-)
	SHT	asm.js	315 (1.00)	883 (2.80)	463 (1.47)	- (-)
	SHT	WASM	133 (1.00)	242 (1.82)	196 (1.47)	- (-)
	SHT	WASM SIMD (impl.)	131 (1.00)	245 (1.87)	198 (1.51)	- (-)
	SHT	WASM SIMD (expl.)	124 (1.00)	180 (1.45)	198 (1.60)	- (-)
	SHT	Workers	67 (1.00)	111 (1.66)	95 (1.42)	118 (1.76)
	SHT	WebGL	13 (1.00)	14 (1.08)	- (-)	- (-)
✓	SHT	JS Sequential	29 (1.00)	83 (2.86)	41 (1.41)	43 (1.48)
✓	SHT	C++ addon	- (-)	- (-)	31 (-)	- (-)
✓	SHT	asm.js	78 (1.00)	214 (2.74)	116 (1.49)	- (-)
✓	SHT	WASM	29 (1.00)	93 (3.21)	47 (1.62)	- (-)
✓	SHT	WASM SIMD (impl.)	29 (1.00)	93 (3.21)	48 (1.66)	- (-)
✓	SHT	WASM SIMD (expl.)	33 (1.00)	55 (1.67)	50 (1.52)	- (-)
✓	SHT	Workers	16 (1.00)	45 (2.81)	25 (1.56)	22 (1.38)
✓	SHT	WebGL	14 (1.00)	15 (1.07)	- (-)	- (-)
Średnia geometryczna (CHT)		(1.00)	(1.55)	(0.99)	(0.87)	
Średnia geometryczna (SHT non-LUT)		(1.00)	(1.51)	(1.51)	(1.69)	
Średnia geometryczna (SHT LUT)		(1.00)	(2.36)	(1.54)	(1.43)	
Średnia geometryczna (wszystkie)		(1.00)	(1.76)	(1.32)	(1.28)	

WebGPU i sprawdzenie, czy i w jakim stopniu rozwiązuje on problemy wynikające z niedoskonałości metody opartej na WebGL.

JavaScript, pomimo bycia rozwiniętym językiem z bogatym ekosystemem, nie pozwala na budowę szeroko kompatybilnych bibliotek ze względu na różnice wśród środowisk i ich różną specyfikę. Do najszybszych obliczeń, co pokazują przeprowadzone testy, i prezentacji ich wyników najlepszym środowiskiem jest przeglądarka Google Chrome. Jednak konieczność budowania, trudność w prototypowaniu rozwiązań oraz bycie środowiskiem odizolowanym od systemu operacyjnego skutecznie powstrzymało rozwój jego zastosowania w obszarze obliczeń numerycznych. Środowiska serwerowe leżą po drugiej stronie barykady. Mają możliwość bezpośrednią interakcji z systemem operacyjnym. Pomiędzy nimi występują jednak braki w kompatybilności, czego przykładem jest interfejs obsługi Worker'ów w NodeJS i Deno. Prezentacja wyników w formie graficznej wymaga wygenerowanie ich i wyświetlenie w postaci strony internetowej lub zapisania plików, podczas gdy w języku Python biblioteki są w stanie wyświetlić wyniki w osobnych oknach.

Różnice środowisk, ich wzajemne wady i zalety, oraz oczywiście docelowe zastosowanie samego języka były przyczyną powolnej ewolucji ekosystemu bibliotek, który jest w tej chwili głównym hamulcem w rozwoju tej gałęzi obliczeń w języku JavaScript. Bibliotek jest mało, są mało popularne, co przekłada się na tempo ich rozwoju i podążania za wydajnością i kompatybilnością ze środowiskami. Jednak jak to zawsze bywa, najtrafniejszą odpowiedzią na większość pytań z początku tego rozdziału będzie: *To zależy*. W zależności od wymaganych opóźnień i zakładanego obciążenia konsekwencje asynchroniczności oraz braku bezpośredniego wsparcia dla wątków mogą stanowić problem. W zależności od wymaganego środowiska dla wdrożenia aplikacji oraz zakładanego formatu prezentacji wyników środowisko przeglądarki może okazać się odpowiednie. W zależności od implementowanego algorytmu będzie możliwe zastosowanie wybranych metod akceleracji. Pewnym jest jednak, że bezkompromisowe utworzenie biblioteki bez zważania na stosowana metodę akceleracji jest niemożliwe. Dla całego spektrum rozwiązań na temat algorytmów i języków implementujących obliczenia numeryczne, te o przystosowaniu środowisk języka JavaScript stanowią jedynie ich niewielką część. Powinny być one bowiem skoncentrowane w pierwszej kolejności na złożoności obliczeniowej samych algorytmów, co jest najlepszym sposobem poprawy ich wydajności.

Literatura

- [1] R. Dahl. 10 Things I Regret About Node.js - Ryan Dahl - JSConf EU. <https://www.youtube.com/watch?v=M3BM9TB-8yA> Na dzień 2022-05-03.
- [2] R. Dahl. Original Node.js presentation. <https://www.youtube.com/watch?v=ztspvPYybIY> Na dzień 2022-05-03.
- [3] Z. Dai, H. Liu, Q. V. Le, and M. Tan. Coatnet: Marrying convolution and attention for all data sizes. *Advances in Neural Information Processing Systems*, 34:3965–3977, 2021.
- [4] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.
- [5] emilioSp. Discussion of nodejs vs Apache Performance Battle for the conquest of my heart, Aug 2019.
- [6] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.
- [7] L. Gong, M. Pradel, and K. Sen. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 357–368, 2015.
- [8] P. V. Hough. Method and means for recognizing complex patterns, Dec. 18 1962. US Patent 3,069,654.
- [9] J. Immerkær. Some remarks on the straight line hough transform. *Pattern Recognition Letters*, 19(12):1133–1135, 1998.
- [10] Y. Ito, W. Ohyama, T. Wakabayashi, and F. Kimura. Detection of eyes by circular Hough transform and histogram of gradient. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 1795–1798. IEEE, 2012.
- [11] Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick. Ostrich benchmark suite. <https://github.com/Sable/Ostrich> Na dzień 2022-04-27.
- [12] P. Lynch. The origins of computer weather prediction and climate modeling. *Journal of computational physics*, 227(7):3431–3444, 2008.
- [13] J. K. Martinsen, H. Grahn, and A. Isberg. Combining thread-level speculation and just-in-time compilation in Google’s V8 JavaScript engine. *Concurrency and computation: practice and experience*, 29(1):e3826, 2017.
- [14] B. Meurer. An Introduction to Speculative Optimization in V8, Nov 2017.
- [15] B. Meurer. JavaScript Performance Pitfalls in V8, Mar 2019.
- [16] C. Mims. Huang’s law is the new Moore’s law, and explains why Nvidia wants arm. *The Wall Street Journal*, Sep 2020.

- [17] P. Mukhopadhyay and B. B. Chaudhuri. A survey of Hough Transform. *Pattern Recognition*, 48(3):993–1010, 2015.
- [18] P. L. Palmer, J. Kittler, and M. Petrou. An optimizing line finder using a hough transform algorithm. *Computer Vision and Image Understanding*, 67(1):1–23, 1997.
- [19] S. Perantonis, N. Vassilas, T. Tsenoglou, and K. Seretis. Robust line detection using weighted region based Hough transform. *Electronics Letters*, 34(7):648–650, 1998.
- [20] G. M. Phillips and P. J. Taylor. *Theory and applications of numerical analysis*. Elsevier, 1996.
- [21] F. Sapuan, M. Saw, and E. Cheah. General-purpose computation on GPUs in the browser using GPU.js. *Computing in Science & Engineering*, 20(1):33–42, 2018.
- [22] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72, 2016.
- [23] Marian Croitoru - Pexels <https://www.pexels.com/pl-pl/video/samochody-droga-ruch-drogowy-pojazdy-5200378/> Na dzień 2022-05-24.
- [24] Cottonbro - Pexels <https://www.pexels.com/pl-pl/video/reka-bogaty-pieniadze-zloto-3943971/> Na dzień 2022-05-24.
- [25] AWS Lambda. <https://aws.amazon.com/lambda/> Na dzień 2022-05-09.
- [26] Biblioteka canny-edge-detector. <https://www.npmjs.com/package/canny-edge-detector> Na dzień 2022-05-20.
- [27] Biblioteka Comlink. <https://github.com/GoogleChromeLabs/comlink> Na dzień 2022-05-17.
- [28] Biblioteka image-js. <https://image-js.github.io/> Na dzień 2022-05-20.
- [29] Bulletproof JavaScript benchmarks. <https://mathiasbynens.be/notes/javascript-benchmarking> Na dzień 2022-05-09.
- [30] Chrome 28 Beta: A more immersive web, everywhere. <https://blog.chromium.org/2013/05/chrome-28-beta-more-immersive-web.html> Na dzień 2022-05-04.
- [31] ESModules. <https://v8.dev/features/modules> Na dzień 2022-05-03.
- [32] Everything you need to know about monorepos, and the tools to build them. <https://monorepo.tools/> Na dzień 2022-05-13.
- [33] Fast, parallel applications with WebAssembly SIMD. <https://v8.dev/features/simd> Na dzień 2022-05-04.
- [34] Google Benchmark. <https://github.com/google/benchmark> Na dzień 2022-05-13.
- [35] Google Lighthouse. <https://developers.google.com/web/tools/lighthouse> Na dzień 2022-05-09.
- [36] HeadlessGL. <https://github.com/stackgl/headless-gl> Na dzień 2022-05-05.
- [37] How V8 measures real-world performance. <https://v8.dev/blog/real-world-performance> Na dzień 2022-05-09.
- [38] In depth: Microtasks and the JavaScript runtime environment. https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide/In_depth Na dzień 2022-05-02.

-
- [39] Javascript module pattern: In-depth. <http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html> Na dzień 2022-05-03.
 - [40] JetStream 2. <https://browserbench.org/JetStream/> Na dzień 2022-05-09.
 - [41] JsPerf. <https://github.com/jsperf/jsperf.com> Na dzień 2022-05-09.
 - [42] LibEvent. <https://libevent.org/> Na dzień 2022-05-03.
 - [43] LibUv. <https://libuv.org/> Na dzień 2022-05-03.
 - [44] Mathematica. <https://www.mathematica.pl/> Na dzień 2022-04-24.
 - [45] Matlab. <https://www.mathworks.com/products/matlab.html> Na dzień 2022-04-24.
 - [46] Narzędzie Emscripten. <https://emscripten.org/> Na dzień 2022-05-16.
 - [47] Narzędzie lerna. <https://lerna.js.org/> Na dzień 2022-05-13.
 - [48] Node-API. <https://nodejs.org/api/n-api.html> Na dzień 2022-05-15.
 - [49] Octane. <http://chromium.github.io/octane/> Na dzień 2022-05-09.
 - [50] OpenCV.js. https://docs.opencv.org/4.x/d4/da1/tutorial_js_setup.html Na dzień 2022-04-27.
 - [51] PyPI Stats. https://pypistats.org/packages/__all__ Na dzień 2022-04-27.
 - [52] PyScript: Python in the Browser. <https://anaconda.cloud/pyscript-python-in-the-browser> Na dzień 2022-05-04.
 - [53] Qt for WebAssembly. <https://doc.qt.io/qt-5/wasm.html> Na dzień 2022-05-04.
 - [54] R. <https://www.r-project.org/about.html> Na dzień 2022-04-24.
 - [55] Semver. <https://semver.org/> Na dzień 2022-05-03.
 - [56] setTimeout: Reasons for delays longer than specified. https://developer.mozilla.org/en-US/docs/web/api/settimeout#reasons_for_delays_longer_than_specified Na dzień 2022-05-02.
 - [57] SpiderMonkey. <https://spidermonkey.dev> Na dzień 2022-05-03.
 - [58] StackOverflow 2021 Developer Survey. <https://insights.stackoverflow.com/survey/2021> Na dzień 2022-04-27.
 - [59] The structured clone algorithm. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm#supported_types Na dzień 2022-05-04.
 - [60] Tensorflow.js. <https://www.tensorflow.org/js> Na dzień 2022-05-04.
 - [61] Tokio. <https://docs.rs/tokio> Na dzień 2022-05-03.
 - [62] Using Promises. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises Na dzień 2022-05-02.
 - [63] V8. <https://v8.dev> Na dzień 2022-05-03.
 - [64] WebGPU. <https://gpuweb.github.io/gpuweb/> Na dzień 2022-05-09.

Dodatek A

Opis załączonej płyty CD/DVD

Dołączona płyta zawiera wszystkie pliki wykorzystane do stworzenia projektu, dokumentacji i niniejszej pracy. Na płycie znajduje się cała zawartość projektu zaimplementowanego w strategii *monorepo* i w katalogu głównym znajdują się pliki odpowiedzialne za jej obsługę. Pozostałe pliki podzielono na następujące katalogi:

- **.vscode** - ustawienia środowiska VsCode,
- **benchmark** - wyniki testów wydajności w formacie *.csv,
- **docs** - notatki oraz treść i materiały niniejszej pracy,
 - **notes** - notatki
 - **out** - pliki będące wynikiem kompilacji plików źródłowych pracy
 - W04N_241292_2022_praca_magisterska.pdf - plik PDF z zawartością pracy
 - **src** - pliki źródłowe pracy
- **packages** - zbiór pakietów *monorepo*
 - **benchmark** - Biblioteka *benchmark*
 - **cpp-sequential** - Implementacja metody sekwencyjnej w C++
 - **frontend** - Rzeczywisty przykład użycia
 - **js-benchmarks** - Przeprowadzanie pomiarów wydajności
 - **js-gpu** - Implementacja metody WebGL
 - **js-sequential** - Implementacja metody sekwencyjnej w JavaScript
 - **js-workers** - Implementacja metody Workers
 - **meta** - Typy języka TypeScript współdzielone pomiędzy implementacje
 - **node-cpp-sequential** - Implementacja metody Native C++ Addon w NodeJS
 - **test-simple** - Strona testowa dla implementowanych metod
 - **wasm-sequential** - Implementacja metody WASM oraz jej wariantów
- **test** - zbiór testowych obrazów,