**Politechnika Wrocławska**
**Wydział Informatyki i Telekomunikacji**

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

## Propietary implementation of Hough transformation algorithms for selected calculation acceleration methods in JavaScript language environments

Damian Koper

Opiekun pracy
**Dr inż. Marek Woda**

WROCŁAW, 2022

# Streszczenie

blabla

**Słowa kluczowe:** raz, dwa, trzy, cztery

# Abstract

blabla in english

**Keywords:** one, two, three, four

# Spis treści

# Spis rysunków

# Spis tabel

# Spis listingów

# Skróty

**EXAMPLE** (ang. *Example*)

# 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[15], co świadczy o obecnym rozwoju możliwości optymalizacji architektur i programów wykorzystujących przetwarzanie masowo równoległe. 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 liczbie 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 jednoznacznym 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 rozwinąć się pola związana 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[19].

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 zaczęło 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[11]. Kolejnym wartym 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%[11, 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[37] oraz R[43], czy też środowisko Mathematica z językiem Wolfram[36].

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 niwelując 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 [40].

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, poddając 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 kompilacja 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.

| **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)<br>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)<br>plotly (1.632.246) | plotly.js (149.542)<br>c3 (83.564) | Wizualizacja danych |
| tensorflow (3.348.986) | @tensorflow/tfjs (91.233) | Sieci neuronowe |
| opencv-python (1.236.711) | OpenCV.js (b.d [39])<br>jimp (1.479.783)<br>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 [47].

#### JavaScript w obliczeniach numerycznych

Pomimo swojej popularności, w przeciwieństwie do języka Python, język JavaScript nie zyskał popularności wśród zadań 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 powstały 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. Rodzi to problemy

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 przystosowania 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 akceleracji rozumiane jest jako każda modyfikacja algorytmu wpływająca pozytywnie na jego wydajność. Tyczy się to optymalizacji wersji sekwencyjnej algorytmu bez zmiany jego złożoności, jak i jego modyfikację do wersji zrównoleglonej. Metody akceleracji zaimplementowane dla badanych środowisk przedstawione zostały w tabeli 1.2. Zostaną one opisane dokładnie w dalszej części pracy. Analiza zorientowana jest na algorytmy analizy obrazów.

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 paczek lub kodu bazowego w języku innym niż C++
[3] Niestabilne lub eksperymentalne

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

ponieważ rozważania takie wykraczają poza zakładaną w pracy tematykę, która koncentruje się na środowiskach języku JavaScript i 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. Jednak 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, wyświetlanie wyników, ładowanie danych oraz wewnętrzne mechanizmy przetwarzania danych związane również z modelem wykonania języka JavaScript.

### 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 został 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[10], 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 zmiennoprzecinkowe, 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, co pozwala 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 do bazowych wariantów 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

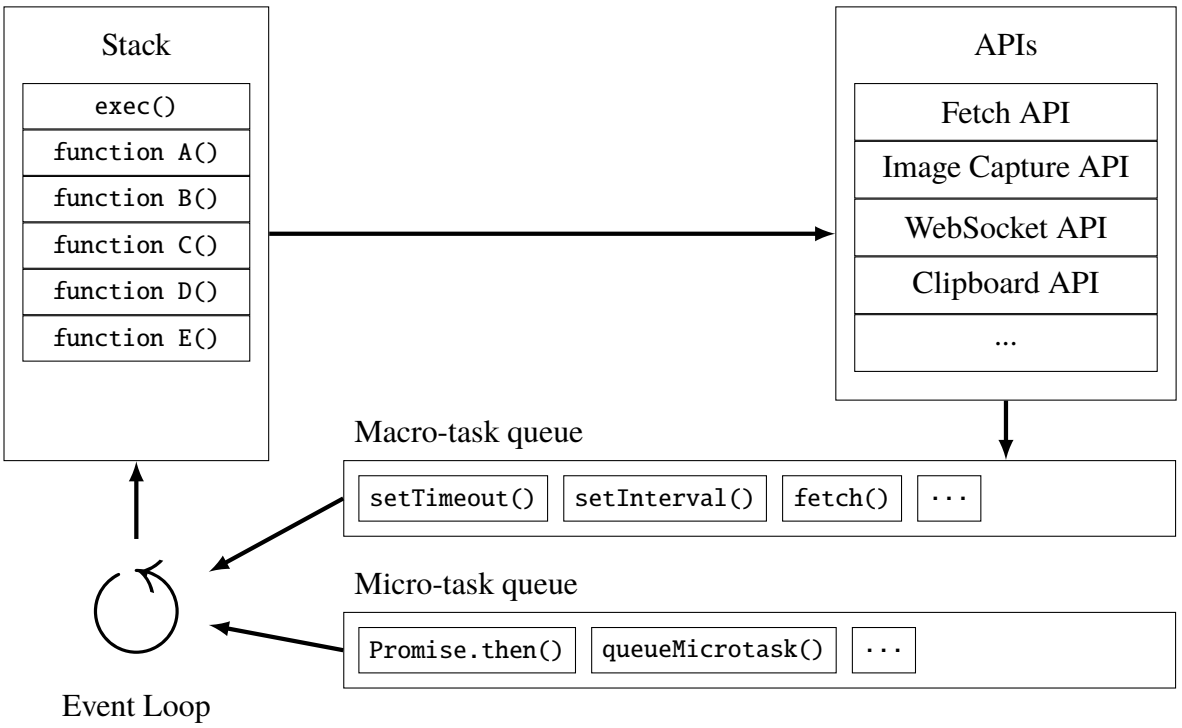
JavaScript od momentu swojego powstania w 1995 roku stanowi jeden z filarów rozwoju technologii webowych, zaczynając od dodania prostych mechanizmów interaktywności do statycznych stron internetowym, a kończąc na byciu nieraz jedynym samodzielnym budulcem pełnowymiarowych aplikacji działających po stronie 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 obsłużyć 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 [45]. 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.

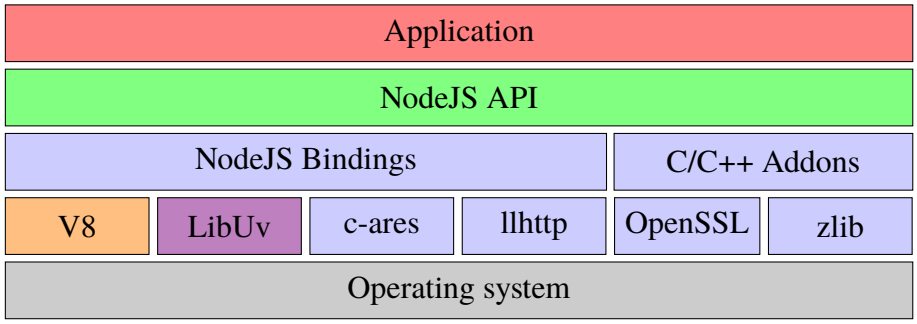
Mikro zadania pochodzą tylko i wyłącznie z kodu użytkownika, bądź bibliotek i wykorzystywane są do obsługi asynchronicznych zadań, zarządzania ich kolejnością i obsługą błędów[30]. Do ich tworzenia wykorzystuje się głównie obiekt `Promise`, który reprezentuje zadanie, które w przyszłości zakończy się pomyślnie lub błędem. Dla takiego obiektu zdefiniować możemy funkcje, które wykonają się podczas scenariusza pomyślnego (`then`), błędnego (`catch`) oraz zawsze (`finally`) [51]. 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 maja 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 taska, przeniesione zostało na koniec wykonania synchronicznego.




Rys. 2.2: Model architektury środowiska NodeJS.

## 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, kompilacją do bytecode'u, interpretowaniem oraz późniejszą optymalizacją kodu. Posiada wiele możliwości optymalizacji spekulatywnej z racji na specyfikę języka, który jest dynamicznie typowany [13].

Popularnymi silnikami renderującymi są Blink z silnikiem JavaScript V8 [52] oraz Gecko z silnikiem JavaScript SpiderMonkey [46]. 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 [34].

### 2.2.2. NodeJS

NodeJS jest powstałym 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 biblioteką LibUv [35] implementującą pętlę zdarzeń w ramach jednego procesu wykonującego kod JavaScript użytkownika oraz wielu wątków, które realizują oczekiwanie na operacje asynchroniczne, pozwoliło rozwiązać problem operacji wejścia-wyjścia pozwalając w ramach tych samych zasobów sprzętowych osiągnąć lepsza wydajność niż popularny serwer Apache [5]. Na rysunku 2.2 pokazano architekturę środowiska NodeJS, która stoi pomiędzy aplikacją, a systemem operacyjnym.

### 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 rozwiązując 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 [50]. W przeciwieństwie do NodeJS i przeglądarek internetowych obsługuje natywnie TypeScript - nadzbiór języka JavaScript umożliwiający wykorzystanie statycznego typowania oraz wszystkie zalety za tym idące.

## 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 paczek.

### 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 zatrzaskiwać wersję paczek, które trafiają do folderu `node_modules`, w którym środowisko NodeJS domyślnie poszukuje kodu podczas importu paczek. Zainstalowane paczki 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 [44], 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). *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 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 i paczek 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 paczki i nigdy nie zmienić swojej zawartości. Pobranie i kompilacja zależności w czasie wykonania programu likwiduje potrzebę przechowywania listy zależności i ich wersji oraz niweluje potrzebę ich instalacji. Deno wykorzystuje opisany w dalszej części rozdziału format modułów ESM.

Niezależnie od tego gdzie przetrzymywane są moduły oraz w jaki sposób są zarządzane przez środowisko, muszą one być finalnie przez nie skonsumowane. Mogą być one użyte bezpośrednio, ale też przetransformowane w procesie budowania biblioteki, która będzie potem dalej konsumowana, czy paczki 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 [31] 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 [25]. 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 wieloma 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 kompilację 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 [13]. 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 [14]. 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 [12].

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% ([6], [21]). 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 moduły

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 modułów z kodu C++ o nazwie *node-gyp*, którego zależnością jest język Python.

Zaletą korzystania z natywnych modułów jest możliwość wykorzystania metod optymalizacji specyficznych dla platformy sprzętowej, jakie oferują języki niskiego poziomu kompilowane do kodu maszynowego. Metodę tę można rozwinąć 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 modułu 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 kompilacji 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 kompilacji dużej ilości kodu. Umożliwiło to na przykład łatwe przeniesienie aplikacji napisanych w języku C++ z wykorzystaniem biblioteki Qt i uruchomienie ich w przeglądarce internetowej [42]. Kolejnym przykładem jest jeden z backend'ów biblioteki Tensorflow.js [49], która umożliwia trenowanie i wdrażanie modeli uczenia maszynowego. Dzięki bibliotece PyScript, która portuje implementację języka Python napisaną w C do języka WebAssembly, możliwe stało się wykonywanie kodu języka w przeglądarce internetowej [41].

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  }
```

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)
```

Wydajność kodu WebAssembly zbliżona jest do wykonania natywnego. Moduły operują na liniowym modelu danych, który nie jest współdzielony z kodem języka JavaScript. Podobnie jak w przypadku natywnych modułów 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.

#### asm.js

Poprzednikiem WebAssembly był asm.js - podzbiór języka JavaScript w procesie kompilacji z języków źródłowych zoptymalizowany pod kątem wydajności wykonania i specjalnie interpretowany przez przeglądarki wykorzystując kompilację Ahead-Of-Time. Kompilację tę wspiera do dziś jedynie przeglądarka Firefox, jednak optymalizacje silnika V8 Google Chrome 28 zapewniły dwukrotny wzrost wydajności podczas wykonania asm.js [24]. Nie jest on już rozwijany i został wyparty przez WebAssembly.

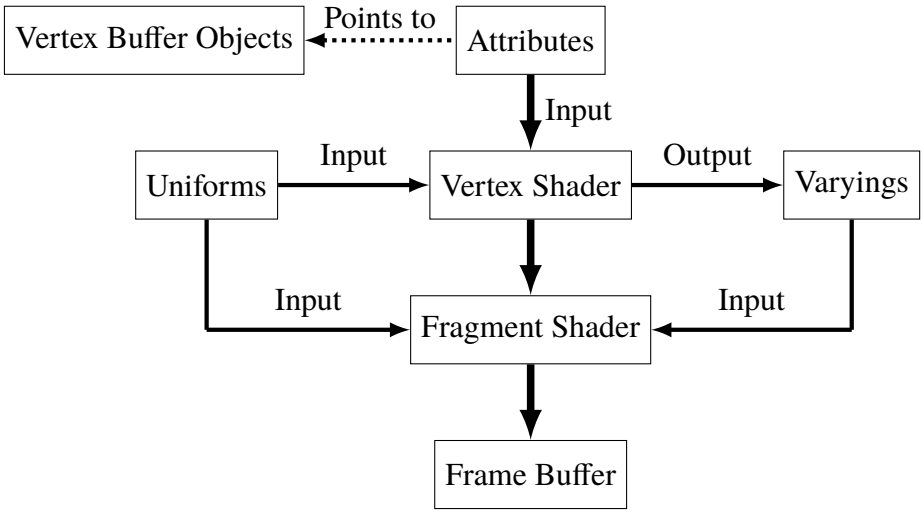
#### SIMD

WebAssembly jest w stanie wykorzystać architekturę SIMD (Single Instruction, Multiple Data), używając rejestrów i instrukcji wektorowych [26]. 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 tego 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 [48]. 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 skopiowane, oszczędzając w ten sposób pamięć i czas procesora.

W środowiskach działających po stronie serwera współbieżne 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, również poprzez mechanizm wiadomości, który zajmuje się ich automatycznyą serializacją i parsowaniem.




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

### 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 architekturą układu, 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 [20]. Każdy piksel generowanego obrazu stanowić może pojedynczy wynik działania kernela czyli funkcji, której działanie jest masowo zrównoleglane. 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 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 piksela. 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 dostępny jest WebGL, 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* [28]. 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, sprowokowała powstanie specyfikacji WebGPU API, która stanowi uogólnienie przetwarzania masowo równoległego dostarczając bezpośrednio abstrakcję GPU [53]. 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 [16]. Samo pojęcie transformacji odnosi się do odwzorowywania pojedynczych pikseli obrazu binarnego lub ich zbioru w przestrzeni akumulatora. Obraz wejściowy wcześniej poddany być musi procesowi wykrywania krawędzi. Dane zebrane w akumulatorze biorą następnie udział w procesie głosowania, w którym wyłonione zostają potencjalne kształty poprzez wykrywanie największych wartości w akumulatorze. W zależności od specyfiki problemu oraz wykrywanych kształtów głosowanie nad kandydatami 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.

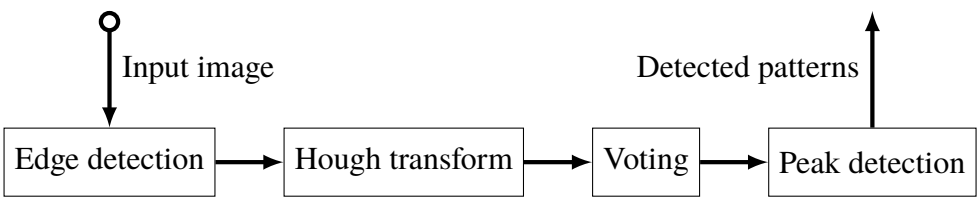
## 3.1. Standard Hough Transform

Pracą, która jako pierwsza opisała tę transformację jest zgłoszony w 1962r. patent Paula Hough'a [7]. 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).

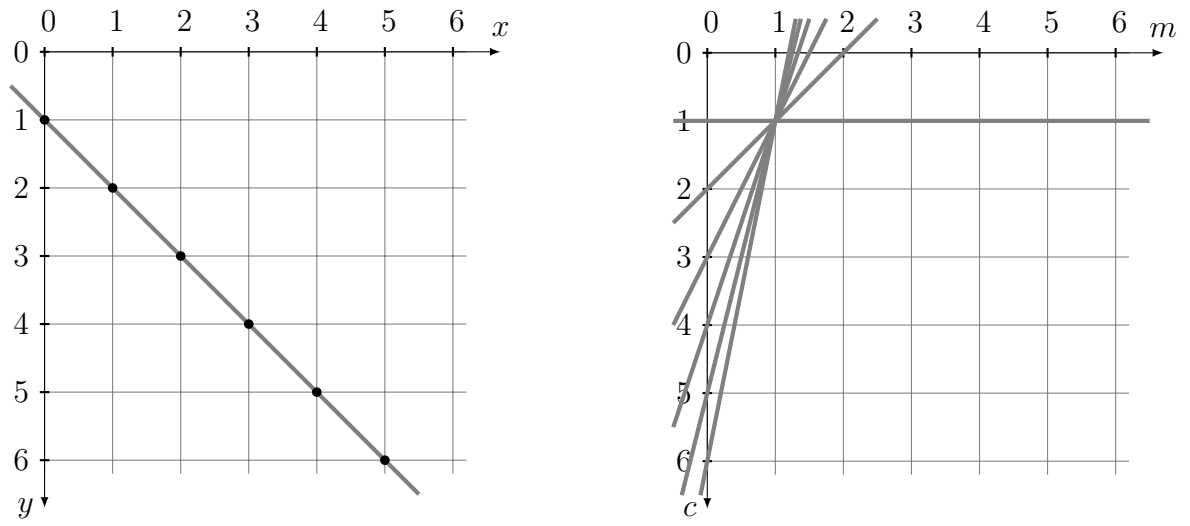
$$y(x) = mx + c \tag{3.1}$$

$$c(m) = -xm + y \tag{3.2}$$

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.



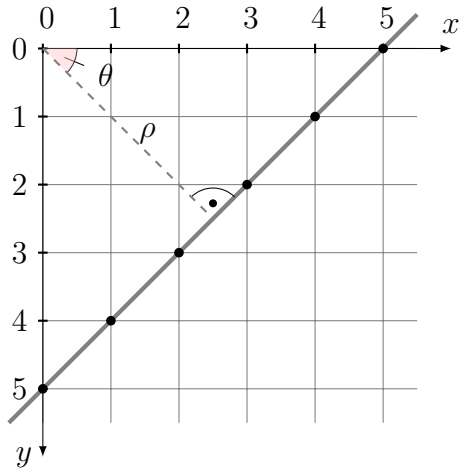

(a) Binarny obraz wejściowy

(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.

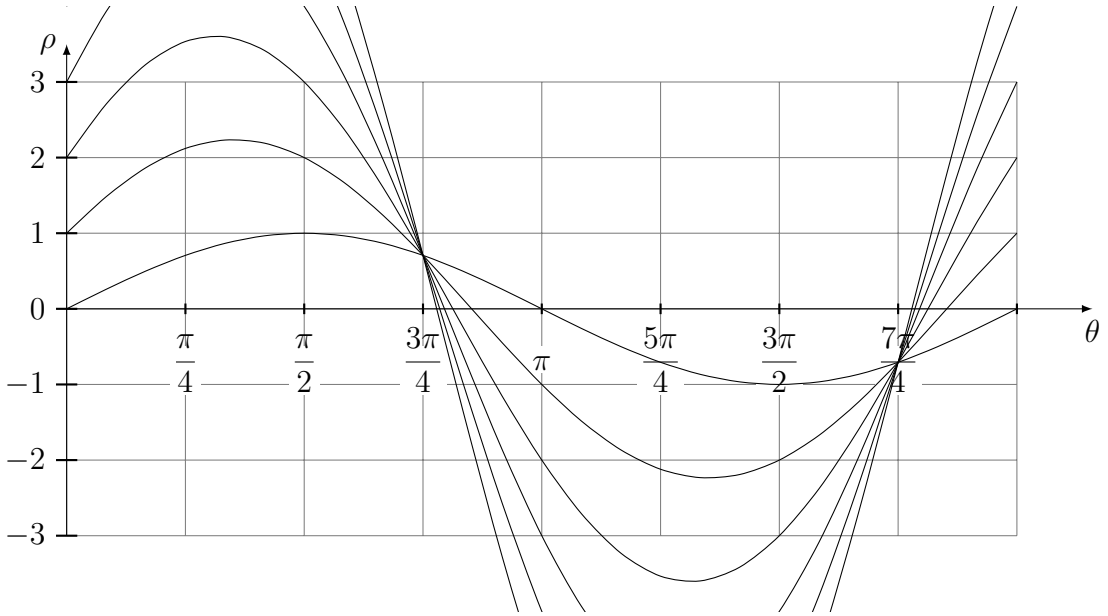
Taka reprezentacja punktu w przestrzeni akumulatora rodzi jednak problem w przypadku wykrywania linii pionowych. Piksele obrazu zorientowane w pionie w przestrzeni akumulatora utworzą linie równoległe. Brak punktu przecięcia takich linii uniemożliwia wykrycie linii na obrazie. Rozwiązaniem tego problemu jest zaproponowana w 1972 roku zmiana reprezentacji prostej, gdzie zamiast zbocza i punktu przecięcia z osią Y użyto biegunowego 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.




Rys. 3.3: Prosta opisana za pomocą odległości $\rho$ i kąta $\theta$ od środka układu współrzędnych biegunowych.

$$\rho(\theta) = x \cos\theta + y \sin\theta \tag{3.3}$$

gdzie: $x, y$ — współrzędne piksela na obrazie;
$\rho$ — odległość prostej od środka układu współrzędnych ;
$\theta$ — obrót prostej od wokół układu współrzędnych.




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 oczekiwaniami 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 $\theta \in [0, \pi\rangle$ [8].

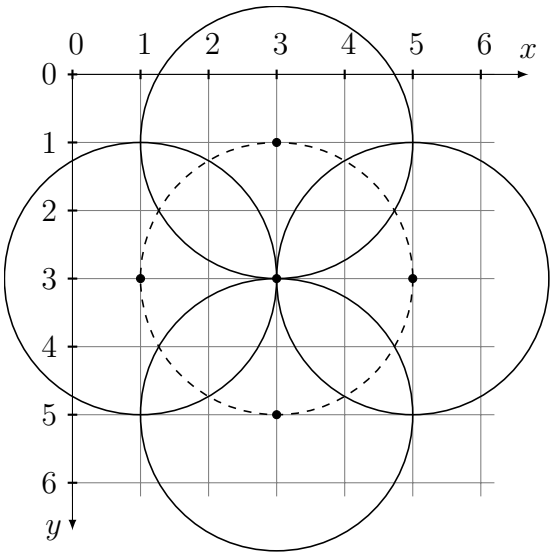
W czasie transformacji, na końcowy rezultat mają wpływ szumy, które powstały na skutek niedoskonałości procesu wykrywania krawędzi, są bardzo krótkimi krawędziami lub są krawędziami, ale nieuwzględnianymi w rozwiązywanym problemie. Przykładem mogą być okręgi podczas wykrywania linii na obrazie. Proces głosowania w przestrzeni akumulatora musi uwzględniać takie sytuacje i proste progowanie może zostać zastąpione bardziej złożonymi metodami [17, 18].

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

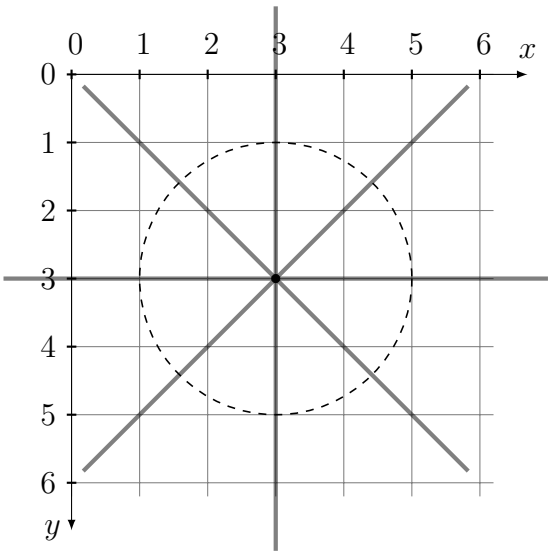
## 3.2. Circle Hough Transform

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 [16]. 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 wykładniczej 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 [9], 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 w wariancie 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. 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 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 $S_\rho$ i $S_\theta$ parametrami próbkowania parametrów w przestrzeni akumulatora. Widać zatem, że złożoność w tym wypadku zależy od jakości danych wejściowych, gdzie wszelkie szumy zwiększają czas wykonania algorytmu, oraz od zakładanej dokładności. 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 liczbę obliczeń wymaganych w procesie głosowania. Ważnym czynnikiem jest specyfika problemu, który rozwiązywać ma 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 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 [29]. 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 [33], 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 [10], 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 kompilacji kompilatora TypeScript, działanie WebAssembly, czy wyrażeń regularnych [38, 32].

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 [27]. 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 [22].

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 [23].

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 pomiary będzie iloraz $\frac{t}{n}$. Innym podejściem jest wykonywanie testów w pętli, aż do

osiągnięcia zadeklarowanego całkowitego czasu. Hybrydowym rozwiązaniem 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 wywołania `Function.prototype.toString()`. Wywołanie to zwraca funkcję - argumenty i jej ciało w postaci ciągu znaków, a następnie ciało duplikowane jest wymaganą liczbę razy. Rozwiązanie 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 rozważenie są również mechanizmy pomiaru czasu. Różnią się one w zależności od środowiska i w celu zapewnienia bezpieczeństwa mają one często ograniczoną rozdzielczość, 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 Firefox wartość ta zaokrąglona jest to 1ms, a w Google Chrome do 0.1ms. Chrome po użyciu flagi `--enable-benchmarking` przy uruchomieniu udostępnia obiekt `chrome.Interval`, którego pomiary mają rozdzielczość $1\mu s$. Deno po użyciu flagi `--allow-hrtime` zwiększa rozdzielczość pomiarów przy użyciu Performance API, a NodeJS udostępnia obiekt `process.hrtime`, którego pomiary wykonywane są z rozdzielczością $1ns$.

Biblioteka benchmark.js 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

# Rozdział 5

# Implementacja i wyniki pomiarów

# Rozdział 6

# Podsumowanie

asdsad

asdsad

# 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] 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.

[7] P. V. Hough. Method and means for recognizing complex patterns, Dec. 18 1962. US Patent 3,069,654.

[8] J. Immerkær. Some remarks on the straight line hough transform. *Pattern Recognition Letters*, 19(12):1133–1135, 1998.

[9] 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.

[10] 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.

[11] P. Lynch. The origins of computer weather prediction and climate modeling. *Journal of computational physics*, 227(7):3431–3444, 2008.

[12] 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.

[13] B. Meurer. An introduction to speculative optimization in v8, Nov 2017.

[14] B. Meurer. Javascript performance pitfalls in v8, Mar 2019.

[15] C. Mims. Huang's law is the new Moore's law, and explains why Nvidia wants arm. *The Wall Street Journal*, Sep 2020.

[16] P. Mukhopadhyay and B. B. Chaudhuri. A survey of hough transform. *Pattern Recognition*, 48(3):993–1010, 2015.

[17] 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.

[18] 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.

[19] G. M. Phillips and P. J. Taylor. *Theory and applications of numerical analysis*. Elsevier, 1996.

[20] 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.

[21] 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.

[22] Aws lambda. https://aws.amazon.com/lambda/ Na dzień 2022-05-09.

[23] Bulletproof javascript benchmarks. https://mathiasbynens.be/notes/javascript-benchmarking Na dzień 2022-05-09.

[24] 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.

[25] Esmodules. https://v8.dev/features/modules Na dzień 2022-05-03.

[26] Fast, parallel applications with webassembly simd. https://v8.dev/features/simd Na dzień 2022-05-04.

[27] Google lighthouse. https://developers.google.com/web/tools/lighthouse Na dzień 2022-05-09.

[28] Headlessgl. https://github.com/stackgl/headless-gl Na dzień 2022-05-05.

[29] How v8 measures real-world performance. https://v8.dev/blog/real-world-performance Na dzień 2022-05-09.

[30] 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.

[31] Javascript module pattern: In-depth. http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html Na dzień 2022-05-03.

[32] Jetstream 2. https://browserbench.org/JetStream/ Na dzień 2022-05-09.

[33] Jsperf. https://github.com/jsperf/jsperf.com Na dzień 2022-05-09.

[34] Libevent. https://libevent.org/ Na dzień 2022-05-03.

[35] Libuv. https://libuv.org/ Na dzień 2022-05-03.

[36] Mathematica. https://www.mathematica.pl/ Na dzień 2022-04-24.

[37] Matlab. https://www.mathworks.com/products/matlab.html Na dzień 2022-04-24.

[38] Octane. http://chromium.github.io/octane/ Na dzień 2022-05-09.

[39] Opencv.js. https://docs.opencv.org/4.x/d4/da1/tutorial_js_setup.html Na dzień 2022-04-27.

[40] Pypi stats. https://pypistats.org/packages/__all__ Na dzień 2022-04-27.

[41] Pyscript: Python in the browser. https://anaconda.cloud/pyscript-python-in-the-browser Na dzień 2022-05-04.

[42] Qt for webassembly. https://doc.qt.io/qt-5/wasm.html Na dzień 2022-05-04.

[43] R. https://www.r-project.org/about.html Na dzień 2022-04-24.

[44] Semver. https://semver.org/ Na dzień 2022-05-03.

[45] 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.

[46] Spidermonkey. https://spidermonkey.dev Na dzień 2022-05-03.

[47] Stackoverflow 2021 developer survey. https://insights.stackoverflow.com/survey/2021 Na dzień 2022-04-27.

[48] 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.

[49] Tensorflow.js. https://www.tensorflow.org/js Na dzień 2022-05-04.

[50] Tokio. https://docs.rs/tokio Na dzień 2022-05-03.

[51] Using promises. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises Na dzień 2022-05-02.

[52] V8. https://v8.dev Na dzień 2022-05-03.

[53] 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. Pliki podzielono na następujące katalogi:

1. `.vscode` - ustawienia środowiska VsCode,