Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra managementu

Využití WebAssembly pro vývoj webových aplikací

Bakalářská práce

Autor: Tadeáš Polák
Studijní obor: Informační management
Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Hradec Králové leden 2022

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Českém Meziříčí dne 21.4.2022 Tadeáš Polák

Anotace

Název: Využití WebAssembly pro vývoj webových aplikací

Bakalářská práce se zaměřuje na analýzu dopadů vybraných způsobů využití WebAssembly na výkon a rychlost webových aplikací na straně klienta. Nejdříve se zaměříme na základní definici WebAssembly a struktury WebAssembly modulů. Dále krátce popíšeme Asm.js a NaCl a jaký vliv měli na návrhu WebAssembly. Následně popíšeme, jak lze WebAssembly nasadit ve webových aplikacích. Na konec teoretické části se zaměříme na hlavní limitace WebAssembly a na výkonnostní vlastnosti a jak se porovnávají s Javascriptem. V praktické části se nejprve zaměříme na popis knihoven a nástrojů, které byly využiti pro vývoj aplikace. Na závěr se budeme zabývat analýzou WebAssembly na rychlost a výkon webových aplikací pomocí nástroje Lighthouse, pro rychlost, a provedením predikce modelu umělé inteligence přímo na klientovi, pro výkon.

Anotation

Title: Use of WebAssembly for web application development

The Bachelors Thesis focuses on the analysis of WebAssembly on the speed and performance of client-side web applications. To start we will provide a basic definition of what WebAssembly is and describe its structure. Then we will shortly describe projects Asm.js and NaCl and the influence they had on the design of WebAssembly and selected ways WebAssembly can be used in web applications. To end the theoretical part, we will describe the current major limitations of WebAssembly in the browser and compare its performance related properties to Javascript. In the next part of the thesis, we will first cover some libraries and tools that were used to develop the test applications. Next, we will focus on the analysis itself using Lighthouse for speed and machine learning model running directly in the browser, for performance.

Obsah

1. Úvod

Náplní této bakalářské práce, jak již název napovídá, je WebAssembly (dále wasm). WebAssembly je relativně nový kompilační cíl (nebo také jazyk), který je branný jako čtvrtý jazyk, který je nativně podporován webovými prohlížeči. Můžeme říct, že do většího povědomí se dostal s příchodem frameworku Blazor. Blazor je zajímá tím, že umožňuje psát spa (*single page application)* webové aplikace s použitím jazyka C#. C# je jazyk, který se historicky používal exkluzivně na desktopové, serverové či mobilní aplikace. Ale koncem roku 2017 se stal jedním prvních jazyků, s automatickou správou paměti (GC), který bylo možné s wasm využívat. Více o něm bude řeč v dalších kapitolách, neboť jejich přístup ke zprovoznění C# ve wasm je celkem unikátní a dobře ilustruje co je s touto technologií možné dělat. Jaký je však význam wasm? Významů je několik. Tím nejočividnějším je možnost jednoduše používat jiné programovací jazyky, než je Javascript (důraz na slovo jednoduše). To samo osobě má zajímá implikace. Otevírají se dveře do bohatých ekosystémů jiných jazyků (např. wasm port ffmpeg) a existuje možnost využívat jazyky se silným typovým systémem jako například Rust nebo výše zmínění C#. Tato práce se však zaměří na jiný význam, a to na dopad na výkon webových aplikací, neboť WASM má vlastnosti, které jí umožňují vyšší, a hlavně konzistentnější výkon oproti Javascriptu. Tímto však výčet významů nekončí, jedná se však, dle mého názoru, o ty hlavní, od kterých další odvíjí. Z předchozích vět je jasné, že wasm slibuje mnohé a potencionální dopad této technologie na vývoj nejen webových aplikací může být obrovský.

2. Cíl, metodika

Cílem této práce je analyzovat dopad vybraných způsobů využití WebAssembly na výkon a rychlost webových aplikací na straně klienta.

Můžeme identifikovat alespoň čtyři oblasti využití WebAssembly. První oblastí je lepší využití procesoru, u výpočetně náročných úkolů, které jsou vykonávány přímo v prohlížeči klienta, například strojové učení, zpracování a přehrávání streamovaného videa (Amazon prime video), nebo základní editace a komprese obrázků (Squoosh). Jako další oblast můžeme identifikovat portování nativních aplikací, na web s minimální ztrátou funkcionalit, zde můžeme mezi příklady uvést iniciativu společnosti Adobe přinést své programy na web počínaje s Adobe Photoshop (projekt se nachází v rané fázi), port programu AutoCad od společnosti Autodesk, nebo LibreOffice . Třetí oblast úzce souvisí

se dvěma předchozími, a to je využití kódu, který byl původně využíván v nativních aplikacích například knihovny ffmpeg (práce s videem), tract (strojové učení). Poslední oblastí je nahrazení Javascriptu pro vývoj SPA webů jinými programovacími jazyky jako je C#, Rust a další.

I s předchozím rozdělením je relativně složité určit výčet typů aplikací, kde využití WebAssembly je rozumné, a to především kvůli univerzálnosti využití, ale s jistotou můžeme řict, že využití bude nacházet v moderních, nebo právě vznikajících aplikacích. Konkrétněji pak v aplikacích, které imitují nativní aplikace, nebo se původně o nativní aplikace jednalo; aplikace, nebo části aplikací, které více zatěžují procesor, například různé způsoby integrace strojového učení. V této práci se budeme zabývat především první kategorií, a to integrací strojového učení do platformy, která umožňuje adopci koček a WebAssembly zde bude využito pro jednoduší identifikaci plemene, pro zadávání do systému a pro filtraci. Základem pro měření bude jednoduchá aplikace, která bude existovat ve třech variantách základních, a to pouze Javascript (referenční), Javascript a WebAssembly modul a pouze WebAssembly (celá aplikace napsaná s použitím WebAssembly) s využitím frameworku Blazor pro jazyk C#. U každé varianty bylo aplikováno minimum optimalizačních technik, aby výsledná data bylo jednoduší porovnat Tyto varianty budou mezi sebou následně porovnávány ve dvou kategoriích.

První kategorií je vykonání predikcí pomocí modelu umělé inteligence. Přesněji se jedná o kategorizování obrázku pomocí modelu *Mobilenet V2*. Pro vykonání v Javascriptu bude využita state of the art knihovna *TensorflowJs*. Pro WebAssembly, zde bude taktéž využita knihovna *TensorflowJs,* neboť nabízí režim vykonání modelu pomocí WebAssembly, které nabízí některé pokročilé optimalizace (např. SIMD, využití vláken), dále zde bude využita implementace napsaná v jazyce Rust. Modul napsaný v Rustu bude využit pro získání informací o tom, jak výkonný může být i naivní a jednoduchý modul. Nakonec budeme testovat i implementaci pomocí frameworku Blazor, který by nám umožní pozorovat výkon WebAssembly v prostředí, které není ideální. Důležité je zmínit, že v případě Blazoru bude využit kód pro predikci, který je taktéž napsán v jazyce Rust, a to kvůli tomu, že v době psaní se jednalo o nejlepší variantu. Avšak kód bude stále vykonáván v rámci jazyka C#. Měření bude probíhat na desktopu s operačním systémem Windows 10 a procesorem Intel-core I5 7660K, Macbook pro 2017 s procesorem Intel-core I5-7360U a Iphone11. Zároveň zde bude provedeno testování na desktopu se 4x

zpomaleném procesoru, kterým budeme simulovat zařízení se slabším hardwarem. Tedy zařízení s různými úrovněmi výkonosti procesoru, na kterých budeme moci pozorovat, jak dobře se dokáže WebAssembly škálovat. U všech variant bude provedeno kontinuální provedení predikce na 240 obrázcích o s podobnými rozměry, kde kratší strana odpovídá 1024 pixelů a delší strana mezi 1400 - 1600px, které budou postupně získávány ze serveru, normalizovány a bude na nich provedena predikce. Všechny obrázky jsou ve stejném formátu (jpeg) a komprimovány stejným algoritmem (Mozjpeg) se stejným nastavením. Po dokončení všech 240 obrázků budou výsledná data odeslána na server a uložena do databáze. Všechny zde zkoumané varianty jsou v produkčním sestavení a následně hostované ve službě Firebase. Do rychlosti predikce započítáváme proces od normalizace obrázků až po konec predikce.

Druhou kategorií je rychlost webové aplikace neboli to, jak dlouho se aplikace načítá, a jak dlouho trvá, než se stane interaktivní. Pro tato měření, zde budou využity vybrané metriky, které patří do sady *Web vitals*, prosazované a využívané společností Google pro měření UX (uživatelského zážitku) na webu. Konkrétně se zde budeme zajímat o některé metriky spadající do sekce *Performance.* Tedy *Largest Contentful Paint, Total Blocking Time* a *Time to Interactive*, neboť tyto metriky jsou, mimo jiné, přímo ovlivňovány Javascriptem a velikostí souborů přenesených přes síť. Budou tak nějak ovlivněny i WebAssembly. Pro jejich měření zde využijeme nástroj Lighthouse a budou měřeny na domovské stránce, která je spíše statická a načítá malé množství dat a na stránce */pets,* která načítá mnohem větší množství dat, včetně modelů UI. Dále bude měřen dopad *cachování* na Web Vitals všech výše zmíněných variant.

Vykonání predikce umělé inteligence bude měřeno jak na hlavních desktopových (Google Chrome 97.0, Mozilla Firefox 95.0 a Safari 15.1), tak i na některých mobilních (safari) prohlížečích. Tyto prohlížeče byly vybrány, protože se jedná o majoritu webových prohlížečů, které jsou používány uživately, nebo některé další prohlížeče využívají jejich technologii.

Testováním dopadu WebAssembly na výkon strojového učení a rychlost pomocí Web Vitals, se zde zaměřujeme kvůli malému množství informací ohledně dopadu WebAssembly na rychlost načítání (UX), a dále kvůli tomu jak jednoduché je docílit

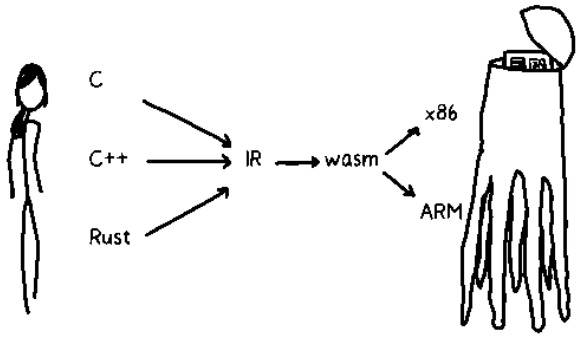
vysokého výkonu ve strojovém učení a jak na výkon, který je vázán na procesor působí vlastnosti WebAssembly.

3. Představení WebAssembly

Na začátek je dobrým nápadem položit si otázku co wasm vůbec je. Pokud bychom chtěli být co nejstručnější můžeme říct, že WebAssembly nám umožňuje spouštět kód napsaný v teoreticky libovolném jazyce na webu. Tato definice však nezachycuje wasm dostatečně komplexně. Lepší odpověď můžeme například najít na oficiálních stránkách WebAssembly, kde se můžeme dočíst: „Web *assembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as portable compilation target for programming languages, enabling deployment on the web for client and server applications.“*(1). Tento popis je velmi dobrý, neboť obsahuje všechny charakteristiky, které js ou pro wasm definující. Na následujících řádcích se touto definicí bude zabývat více, přiblížíme si některé termíny a jejich dopad na WebAssembly.

Binární instrukční formát

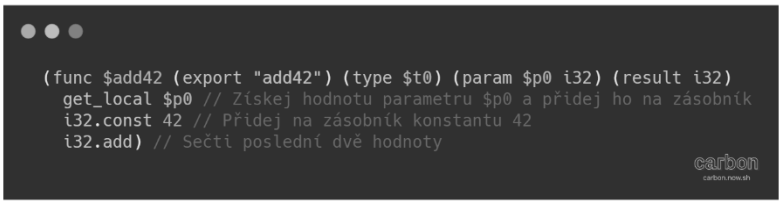
WebAssembly je definováno jako binární formát, to samo osobě má zajímavé implikace, o kterých se více dozvíte v následujících kapitolách. Kromě binárního formátu wasm nabízí taktéž formát textový tzv. wat formát, který má svoji vlastní syntax a strukturu, která se podobá assembly, zde však podobnosti s assembly končí (2). Hlavní oblast, ve které se wasm liší od assembly jsou instrukce. Jak je již v úvodní definice napsáno, wasm je instrukční formát pro virtuální stroj. To znamená, že instrukce, které můžeme ve wasm



Obrázek 1Vztah programovacích jazyků, WebAssembly a instrukcí konkrétních architektur. Zdroj: Lin Clark

najít, nejsou instrukcemi pro konkrétní architekturu, ale jedná se o virtuální instrukce, které můžeme na ty konkrétní namapovat jednodušeji, nežli je to možné u jednodušeji, nežli je to možné u jazyků jako je Javascript. Jak píše Lin Clark: „*So WebAssembly is a little bit different than other kinds of assembly. It's a machine language for a conceptual machine, not an actual, physical machine.".* Vztah programovacích jazyků, wasm a instrukcí konkrétních architektur dále ilustruje obrázek-1 (3). Dalším rozdílem oproti assembly je

fakt, že wasm je zásobníkový stroj. To není u programovacích jazyků, které využívají virtuální stroje, jako je například Java nic neobvyklého, neboť jsou relativně jednoduché na implementaci. Ukázku toho, jak zásobníkoví stroj funguje lze najít na obrázku dva. Jedná se o jednoduchou funkci, která příjme 32bitové celé číslo, sečte ho s konstantou 42 a vrátí výsledek. Ukázka taktéž poukazuje na další vlastnost wasm, která je pro prostředí



Obrázek 2 Ukázka WAT formátu. Zdroj: Autor

Javascriptu novinkou, a to explicitně definované datové typy. O jejich dopadu bude řeč v následujících kapitolách.

Kompilační cíl

V předchozí části se psalo o lidsky čitelném textovém formátu, který wasm nabízí. Avšak stejně jako u assembly, přímá interakce s wasm kódem bude ve většině případů vyhraněno pro speciální použití například pro další optimalizace. Je pravděpodobnější, že většina wasm kódu vznikne kompilací z nějakého jiného, vyššího programovacího jazyka. Zde může nastat otázka: *„Jaké programovací jazyky lze pro vývoj s WebAssembly použít?“.* Jak již bylo zmíněno, teoreticky by mělo být možné použít libovolný jazyk, pokud daný jazyk má kompilátor, který dokáže produkovat WebAssembly. Seznam podporovaných jazyků můžeme nalézt na oficiální webové stránce. Mezi nimi můžeme nalézt systémové programovací jazyky jako jsou C, C++ nebo Rust. Tyto jazyky byli první oficiálně podporované, především protože se jedná o jazyky s manuální správou paměti, což je činilo jednoduššími na implementaci (4). Dále zde můžeme nalézt i některé vyšší programovací jazyky, např. C#, ty však musí řešit velký problém. WebAssembly zatím nemá garbage collector (v době psaní se jedná o návrh). Můžeme tu také nalézt kompletně nové programovací jazyky, které jsou vyvíjeny exkluzivně pro využití pro vývoj s wasm. Jako příklad lze uvést AssemblyScript, jedná se o jazyk, který svou syntaxi modeluje po vzoru TypeScriptu a umožňuje přímou interakci s wasm kódem.

Vývoj pro web i mimo něj

Ať se to může zdát zvláštní, tak technologie, která má ve svém názvu slovo web není exkluzivní pouze pro použití na webu. Důvod je jednoduchý. WebAssembly nebylo navrhováno s konkrétní platformou na mysli. Jak již bylo výše zmíněno wasm se skládá z omezeného množství základních virtuálních instrukcí. Neobsahuje tak funkce, které jsou odlišné napříč platformami, nebo na nich jsou omezené nebo nedostupné, jako je například I/O, nebo práce s vlákny v prohlížeči. (6). WebAssembly tak potřebuje něco, co tyto, a další, funkce bude schopno zprostředkovávat. Dále však potřebuje prostředí, které umožní wasm kód zkompilovat a vykonat. Toto je práce pro tzv. embedder, který si lze představit jako most mezi wasm modulem a prostředím, ve kterém se bude vykonávat. Tento vztah si můžeme zkráceně popsat na využití v prohlížeči. Prostředím, ve kterém budeme vykonávat wasm funkce je prohlížeč. Embedder bude v tomto případě Javascript, který obsahuje API pro kompilaci, modulů, instancování, importování Javascriptových funkcí do wasm (např. console.log) a exportování funkcí z wasm modulu. Proces tak může vypadat takto (detailněji bude rozebrán později):

1) Získání modulu, případné importování funkcí a paměti
2) Kompilace
3) Získání přístupu k exportovaným funkcím z wasm modulu

Vykonávacím prostředím však nemusí být pouze prohlížeč, můžou jim být i desktopová prostředí jako je wasmer.

Zatímco v prohlížeči, nebo v programovacích jazycích lze relativně jednoduše importovat funkce pro zajištění funkcí, které samotné wasm nemá k dispozici, v desktopovém prostředí tomu tak není. A není tomu tak z několika důvodů. Tím hlavním je fakt, že jednotlivé wasm moduly, jsou vykonávány v tzv. sandboxovaném prostředí, kdy kód nemá přístup ke všem funkcím operačního systému, velmi podobné modelu, které používají prohlížeče (7). Řešením pro tento problém je WASI – WebAssembly system interface. Ve zkratce se jedná o projekt, který zamýšlí vytvořit standardizovaný způsob, který by wasm umožnil přístup k funkcím operačního systému jako je souborový systém nebo sokety. WASI přináší i další zajímavou funkcionalitu a tou je „*capability-based security*“. Tato funkcionalita umožňuje limitovat k jakým systémovým zdrojům (soubory, sokety …) má program přístup. Programu tak může být udělen přístup k souborovému systému, ale nikoliv k soketům. Tento přístup k bezpečnosti není úplně novou myšlenkou, protože podobně již dlouho funguje bezpečnost v prohlížečích a na mobilních zařízeních. Filozofie

za wasi není exkluzivní pouze pro využití v desktopových prostředích, ale v jiné podobě se jeho využití uvažuje i v prostředí prohlížečů.

Koncepty WebAssembly

Nyní, když už máme lepší přehled o tom, co wasm je, co dokáže, a kde všude se dá použít je vhodné abychom si něco řekli o tom, jak tato technologie vůbec funguje. Na

Obrázek 3 Jednoduchý wasm modul a jeho části. Zdroj: Autor

následujících řádcích si rozeberme velmi jednoduchý WebAssembly modul (viz obrázek-3) a seznámíme se s některými klíčovými koncepty. Pro zjednodušení lze budeme uvažovat využití v prohlížeči.

Module

Programy napsané ve wasm, nebo kompilovány do wasm jsou děleny do samostatných binárních souborů, které se sestávají z různých sekcí. Mezi nimi třeba funkce (ob-3 Function), podpisy funkcí (ob-3 Types), tabulka (ob-3 table) a paměť (ob-3 memory). Zde je však nutné podotknout, že platným modulem je i ten modul, který obsahuje pouze označení, tj. má následující formu *(module).* Modul je tedy statická reprezentace programu. (3, 8)

Function

Samotný kód je v modulu organizován do funkcí, které popisují požadovanému chování. Definice funkce může odpovídat následujícímu formátu *(func <signature> <locals>*

<body>). Signature neboli podpis, který se nachází v sekci Types (ob-3). Obsahuje výčet vstupních parametrů a výstupů. Sekce Types obsahuje podpisy všech funkcí, včetně těch importovaných, které se v modulu využívají. Poté je možné definovat výčet lokální proměnných. Nakonec je možné definovat jakou hodnotu funkce vrací. Aby mohla být funkce volán zvenčí modulu, tedy například js musí být označena slovem *export* a musí být specifikováno jméno pomocí kterého bude funkce volána.

Memory

Paměť ve WebAssembly je symbolizována lineárním polem bajtů. Pořadí jednotlivých bajtů v paměti je organizováno dle Little-endian. Paměť je potřeba inicializovat. Může být inicializována přímo ve wasm modulu, jak lze vidět na obrázku-3, a poté exportována, nebo může být inicializována mimo wasm modul a následně importována do modulu pomocí elementu *import*. Pro inicializaci paměti je potřeba specifikovat její počáteční velikost. Tato velikost je udávána ve stránkách (v našem případě specifikujeme 17 stránek viz. Obrázek-3), přičemž každá stránka má 64KiB.

Pokud je třeba inicializovat paměť v Javascriptu činíme tak pomocí funkce *WebAssembly.Memory(),* která vrací objekt *Memory*. Tento objekt obsahuje samotnou paměť, která je uvnitř reprezentována typem *ArrayBuffer* nebo *SharedArrayBuffer*. To znamená, že adresy paměti jsou zastupovány indexy do pole. Tento přístup zajišťuje relativní bezpečí, co se týče přístupu k paměti, neboť není možné přistoupit k libovolné adrese, která by se mohla nacházet mimo přiřazenou paměť, a protože velikost paměti (pole) je vždy známá může být zajištěno, že adresa (index), ke které se modul snaží přistoupit, se nachází v přiřazené paměti. WebAssembly tímto způsob dává uživatelům volnost rozhodnout se, jak přistoupit rozšiřování paměti. Pokud je známo, kolik paměti bude modul potřebovat, je možno vytvořit při instancování modulu dostatečně velké pole a používat ho beze změny po celou dobu používání modulu. Na druhou stranu také umožnuje paměť dynamicky rozšiřovat v případech, kdy nemusí být známy požadavky na paměť.

Host má tak přímý přístup k paměti, kterou wasm modul využívá. Toto se hodí u několika případů. První je využití vláken, které budou detailněji rozebrány později a druhým je využívání komplexních datových typů. Zpočátku wasm obsahovalo pouze čtyři datové

typy a všechny byly číselné. To způsobovalo problémy, pokud bylo potřeba přenášet komplexní datové typy.

i32	32bitový celé číslo
i64	64bitové celé číslo
f32	32bitové desetinné číslo
f64	32bitové desetinné číslo
externref	Referenční typ

Dalo se to však vyřešit s využitím paměti, kdy se hodnota, například typu String, uložila do paměti a samotné funkci, která tuto hodnotu využívala, se jako parametry předaly začátek a délka řetězce. Toto řešení však velice primitivní, a ne vždy vhodné. Komplexnější řešení, může vyžadovat napsání nebo vygenerování relativně velkého množství tzv. „glue code“, neboli kódu, jehož účelem je zajistit, aby dva nekompatibilní programovací jazyky mohly spolupracovat. Pro zajištění lepší komunikace mezi hostem a wasm modulem byl přidán pátý datový typ *externref*.

Externref je datový typ, který reprezentuje odkaz na objekt, který se nachází mimo wasm modul, tedy v hostiteli, a umožňuje modulu s tímto objektem interagovat napřímo

```
// Původní kód
#[wasm_bindgen]
pub fn takes_js_value(a: &JsValue) {
    // ...
}

// Glue code
const heap = new Array(32).fill(undefined);

heap.push(undefined, null, true, false);

let stack_pointer = 32;

function addBorrowedObject(obj) {
    if (stack_pointer == 1) throw new Error('out of js stack');
    heap[--stack_pointer] = obj;
    return stack_pointer;
}

export function takes_js_value(a) {
    try {
        wasm.takes_js_value(addBorrowedObject(a));
    } finally {
        heap[stack_pointer++] = undefined;
    }
}
//

// Funkce exportována do js
export function takes_js_value(a) {
    wasm.takes_js_value(a);
}
```

Obrázek 4 Ukázka glue kódu. Adaptováno

na rozdíl od řešeních, která v minulosti vyžadovala speciální glue kód na straně hostitele, jak lze vidět na obrázku 4. Pro zajištění sandboxu má *externref* jedno zásadní omezení, že WebAssembly modul nesmí vracet reference, které sám vytvoří, ale pouze ty, kterému byly poskytnuty hostem, případně hodnotu *null* Kromě odstranění některého glue kódu je *externref* důležitý i pro budoucí rozvoj WebAssembly, o kterém bude řeč v části *Limitace WebAssembly*.

U paměti je potřeba se, alespoň krátce, zmínit problematiku správy paměti. U správy paměti se můžeme setkat s mnoha problémy, nejčastěji pak s úniky paměti. V rámci prohlížeče můžeme u WebAssembly říct to samé jako u Javascriptu a jiných programovacích jazyků s automatickou správou paměti, což je, že úniky paměti jsou téměř nemožné. Tato vlastnost vychází z toho, že paměť, kterou modul využívá je obyčejný objekt Javascriptu, což znamená, že je sledován garbage collectorem a jakmile paměť není používána bude automaticky zničena. (2, 9)

Table

Table je jedna z volitelných částí, které mohou být definované ve wasm modulu, nebo mimo modul, jejíž funkce je relativně jednoduchá. Jedná se o dynamické pole referencí na funkce, které existuje mimo paměť modulu, což umožnuje funkcím bezpečně volat další funkce, které jsou zapsané v table, pomocí indexu, pokud jsou zaznamenané v *table.* Největší užitečnost však nachází v případě, kdy volaná funkce není předem známá během kompilace, ale je závislá na chodu aplikace. Má však ještě jedno teoretické využití. Protože neexistuje přímo v modulu a je přímo přístupná pro hostitele, je možné *table* sdílet mezi vícero moduly a umožňuje tak sdílení funkcí mezi nimi. Nejedná se však o řešení, které by se mělo preferovat před jinými.

Co se týče inicializace, pokud je *table* inicializována v modulu je třeba do elementu *table* specifikovat počet funkcí, které budou zaznamenány, a to že se bude jednat o reference na funkce (*anyfunc)*. Pokud se funkce, které budou zaznamenány v table nachází v modulu, pak je možné je zapsat do elementu *elem,* v pořadí, které bude odpovídat jejím indexům. Je možné ji také importovat, pomocí elementu *import,* pak je však nutné ji vytvořit pomocí Javascriptu. Inicializace přes Javascript je podobná inicializaci paměti, a to pomocí *WebAssembly.Table,* která vrátí instanci. Pro zapsání funkcí je poté potřeba zavolat metodu *set* na instanci.

Instance

Pokud modul je statická reprezentace programu, tak instance je dynamická reprezentace, která obsahuje vše, co potřebuje k běhu. Instancování modulu je, jak již bylo zmíněno, úkol embeddera, v tomto případě Javascriptu. Ten musí zařídit, že modul má vše, co potřebuje, může to tak být Javascriptem vytvořená paměť, potřebné importy funkcí. Pokud instancování proběhne v pořádku je možné volat funkce, které modul exportuje. (2, 9)

Importování funkcí

Poslední klíčovou sekci wasm modulu, která zde bude zmíněna je *import* Už byla řeč o dvou typech importů, a to *table a memory*, zde se bude zabývat importováním funkcí. Importování funkcí je relativně přímočaré. V modulu stačí nadefinovat element *import*, který obsahuje *jmenný prostor*, pod kterým bude funkce importována, *jméno* importované funkce, a nakonec podpis importované funkce, včetně jména, pomocí něhož bude funkce volána v rámci modulu.
Při inicializaci modulu v Javascriptu je pak potřeba specifikovat objekt *imports* a v něm nadefinovat potřebné funkce.

Nyní když máme již lepší představu o tom, co wasm je a jak funguje, můžeme se přesunout na další část, ve které budou představeni předchozí projekty, které se pokoušeli dosáhnout toho samého čeho se pokouší dosáhnout wasm.

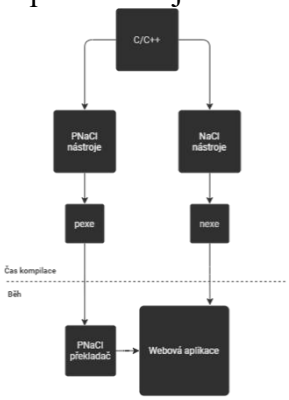
4. WebAssembly v prohlížeči

Předchůdci

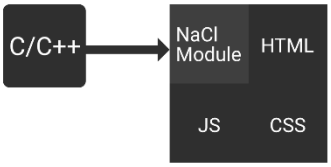
WebAssembly není první pokusem o spouštění kódu, jiného než Javascriptového ve webovém prostředí. V průběhu let existovalo mnoho projektů, které se tohoto cíle snažilo dosáhnout, a to z různých důvodů a různým způsobem. Obvykle se však snažili dosáhnout něčeho co v dané době nebylo na tehdejší webové platformě možné nebo jednoduše proveditelné. Například Adobe Flash umožnil relativně jednoduchou tvorbu animací, her a interaktivních webových aplikací předtím, než to bylo možné se standartními webovými technologiemi. Pro tuto práci jsou však relevantnější ty projekty, jejichž hlavním cílem bylo zvýšení výkonu, v některých případech pomocí velmi rozdílných přístupů. Na následujících řádcích se seznámíme se dvěma projekty, a to Native client od Google a Asm.js od Mozilla.

Native client

Native client byl projekt společnosti Google, který umožňoval spouštění speciálních modulů tzv. NaCl Module přímo v prohlížeči uživatele. Webová aplikace se v tomto module sestávala z klasických webových technologií a C/C++ kódu, který byl zkompilován do již zmíněných modulů. Tento model umožňoval implementovat výpočetně



Obrázek 6 Rozdíl mezi NaCl a PNaCl. Adaptováno



Obrázek 5 Architektura Webové aplikace, která využívá NaCl. Adaptováno.

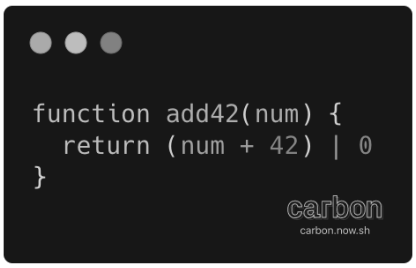
náročnější části aplikací, neboť umožňoval spouštění nativního zkompilovaného kódu a využívání instrukcí, ke kterým Javascript nemá přístup, jako je například SIMD. Neboť se některé instrukce liší mezi platformami byla vyvinut další verze, PNaCl neboli Portable Native client. Oproti klasickému NaCl je kód přeložen, do formátu pexe, který je nezávislí na operačním systému. Tento kód je po načtení prohlížečem zkompilován do formátu nexe, který je využíván i klasickým NaCl, a je proveden. Protože spouštění nativního kódu může být nebezpečné NaCl běží ve svém vlastním sandboxu na rámec sandboxu prohlížeče. Toto zajišťuje, že kód využívá pouze API operačního systému, kterému jsou povoleny a zamezuje interakci s jiným běžícím kódem. (10, 11)

Asm.js

Native client vypadal velmi slibně. Mněl však problémy, které zamezovaly rozsáhlé adopci. Mezi nimi třeba fakt, že se jednalo o formát, který bylo možné využít pouze v prohlížečích založených na Chromium. V té době tedy především Google Chrome. Toto omezení se snažil vyřešit projekt Asm.js.

Podobně jako u Native Client se jednalo o kompilační cíl, který se však lišil v tom, že namísto využívání speciálního formátu využíval „standartní“ Javascript. Přesněji řečeno Asm.js byl podmnožinou Javascriptu, která umožňovala psaní výkonného kódu, který byl obvykle dvakrát pomalejší než nativní kód. Navíc oproti standartnímu Javascriptu byl výkon asm.js předvídatelnější. Těchto vlastností bylo dosaženo tím, že asm.js nevyužíval

některé vlastnosti Javascriptu, jako je garbage collector, objekty, ale třeba i řetězce a jiné dynamické vlastnosti (12, 14). Celý programovací model byl založen aritmetice s explicitně definovanými celými a desetinnými čísly a virtuální haldě, která je reprezentována pomocí *TypedArray* jako například *UInt8Array*. *TypedArray* umožňuje přístup k binárním datům, které se nachází v *ArrayBuffer* neboli bufferu o fixni délce. Dále



Obrázek 7 Ukázka principu Asm.js. Zdroj: Autor

využíval různé triky, které dále pomáhaly zvýšit výkon. Příkladem takové triku je obrázek 7. Obrázek 7 ukazuje trik, který spočívá v aplikování bitové operace OR na výsledek součtu dvou čísel. Co tento trik udělá je, že nehledě na číslo bude tato operace pro procesor stejná jako sčítání celých čísel, které je často rychlejší než sčítání desetinných čísel (13). Tento přístup, kvůli výše zmíněným vlastnostem, znamenal, že Asm.js mohl být rychlejší než Javascript ve všech js enginech. Dále však mohly být provedeny další optimalizace. Například pokud kód obsahoval speciální direktivu, a prohlížeč tuto funkci podporoval, mohl se, kvůli zmíněným omezením, mohl být okamžitě dále přeložen a optimalizován (14,15). Asm.js byl však limitován samotným Javascriptem; například nepřítomností SIMD instrukcí, 64bitových adres a nutností být kompatibilní se samotným Javascriptem. (14)

Jak již je z přechozích řádků jasné WebAssembly není originální nápad. Některé vlastnosti, kterými se pyšní WebAssembly existovaly i u jiných projektů. Můžeme zde třeba zmínit využití sandboxovaného prostředí, speciálních formátů, díky kterým kód funguje nezávisle na platformě, využití pouze číselných typů a to, že se často jednalo o kompilační cíle. Proč má tedy WebAssembly větší šanci na úspěch než její předchůdci? Odpovědí je, že se jedná o skutečný webový standard, který byl vyvinut inženýry, kteří pracují na hlavních webových prohlížečích a nemusí se zabývat držením kompatibility s normálním Javascriptem (14). Funguje tedy ve všech hlavních prohlížečích a není využitelný v pouze

v jednom z nich. Zároveň ke svému fungování nepotřebuje plugin, jako to bylo například u Adobe Flash. Oproti asm.js pak wasm nabízí o něco lepší výkon, ale hlavně by měl být tento výkon konzistentnější mezi prohlížeči.

Způsoby použití

Nyní se přesuneme do části, ve které se seznámíme se způsoby, jak lze WebAssembly využít ve webových aplikacích. Seznámíme se zde se třemi možnostmi. Všechny mají stejný cíl, a to zajistit fungování nějakého wasm modulu, liší se však v tom, co všechno musí vývojář udělat, aby funkci zajistil a rozsah využití WebAssembly v aplikaci.

Samostatný modul

V této části načrtneme, co vše je potřeba pro přímé využití modulu. Tento přístup je tím, nejpřímějším a v některých případech nejsložitějším, neboť veškerá zodpovědnost za

Obrázek 8 Ukázka instancování jednoduchého wasm modulu. Adaptováno

správné inicializování modulu spadá na vývojáře. V závislosti na modulu může inicializace obsahovat mnoho kroků, které mohou být rozdílných složitostí. Samotný proces inicializace může probíhat dvěma způsoby. Buď pomocí *WebAssembly.Instantiate* nebo *WebAssembly.InstantiateStreaming.* Tyto dvě metody se výrazně liší pouze v přístupu kompilování. Zatímco *Instantiate* vyžaduje, aby byl celý modul načten do paměti ve formě *ArrayBuffer, InstantiateStreaming* pracuje nad proudem bajtů ze sítě. Je tak efektivnější a mělo by být preferováno před starším *Instantiate*. V obou případech kromě zdroje, ve vyžadované podobě, mají další, tentokrát volitelný, parametr a to *imports.* Jedná se o objekt, který obsahuje vše, co se importuje do modulu. Tedy funkce, popřípadě paměť a tabulku odkazů. Obě funkce následně zajistí kompilaci, případné importování funkcí do modulu a vytvoří instanci. Pokud vše proběhne úspěšně získáme přístup ke dvěma objektům. Prvním je *instance*, který jak název napovídá, reprezentuje instanci modulu,

pomocí které můžeme přistupovat k exportům, které obsahují funkce, paměť a případně další. Druhým objektem je *module*, který obsahuje samotný zkompilovaný modul. Ten poté můžeme využít pro vytváření dalších instancí. (16) Celý proces je ilustrovaný obrázku 9. Tímto však obslužní kód končit nemusí, jak již bylo v předchozích částech zmíněno v některých případech je nutno využít paměti pro přesouvání hodnot typu String, případně dalších komplexních datových typů, které jsou reprezentované např. ve formátu json, mezi wasm modulem a Javascriptem. Z těchto a jiných důvodů je vhodné využít knihoven a dalších nástrojů, které dokážou tento obslužný kód automaticky vygenerovat a snížit tím šanci špatné inicializace.

Frameworky

Dalším způsobem, jak lze WebAssembly využít ve webových aplikacích jsou frameworky. V této části uvažujeme typ podobné těm Javascriptovým jako jsou react nebo vue. Takové frameworky umožňují napsání celé klientské aplikace, tj. UI a logiky pomocí Javascriptu. Wasm frameworky fungují na stejném principu, až na to, že místo Javascriptu využívají nějaký jiný jazyk. V praxi to tedy znamená, že pokud někdo, kdo chce napsat dynamickou webovou stránku a nezná nebo nechce používat Javascript, ale zná c#, může využít Blazor a pokud nepotřebuje využívat api prohlížeče, tak se teoreticky nemusí Javascriptu dotknout. Zde, však tento hypotetický vývojář končit nemusí. Jak již bylo několikrát zmíněno wasm není samo o sobě závislé na platformě, ale některý kód být může, třeba kód pro uživatelské rozhraní. Pokud by se tento problém podařilo vyřešit, wasm by mohl umožnit vývoj webových, mobilních i desktopových aplikací, které by mohly sdílet stejný kód. Tohoto cíle se snaží dosáhnout minimálně dva projekty. Prvním je Flutter od společnosti Google, který využívá wasm pro vykreslování uživatelského rozhraní na webu, při zachování stejného API, to znamená žádné HTML a minimální úpravy UI kódu, který je poté využit na mobilních zařízeních a některých desktopových operačních systémech. Místo klasického HTML využívá kombinaci canvas elementu a wasm. Konkrétněji se jedná o knihovnu Skia, která je napsána primárně v jazyce c++ a následně zkompilována do WebAssembly (17, 18). Druhým projektem je Uno platform. Zatímco flutter využívá wasm pro UI, Uno wasm využívá spíše pro logiku a umožňuje ji tak sdílet napříč platformami, tj. pokud je cíleno na web bude logika aplikace zkompilována do WebAssembly.

Co je však tou největší výhodou těchto frameworků je jejich integrovanost, která usnadňuje proces vývoje, a to nejen co se týče načítání modulů a dodržování best-practices, ale i integrování dalších nástrojů, které se při vývoji webových aplikací používají jako jsou například css preprocesory jako *sass*, případně další Javascriptové balíčky.

Knihovny

Poslední zde probíranou možností je využití wasm v knihovnách. Knihovny stojí na pomezí samostatného modulu a frameworku, kdy téměř veškerá práce s wasm modulem, jako je inicializace, vytvoření dalšího obslužného kódu a veřejného rozhraní je provedená přímo v knihovně a uživatel tak ani nemusí vědět, že knihovna wasm používá, neboť pouze pracuje s veřejným rozhraním, které bylo uzpůsobeno Javascriptu nebo případně typescriptu. Často jediné, co musí uživatel poskytnout je způsob, jak daný wasm modul získat, např. URL. Nakonec tak můžeme říct, že využívání knihoven usnadňuje použití wasm a umožňuje je lépe integrovat do stávajících aplikací.

Limitace WebAssembly

V následující části budou stručně rozebrány vybrané problémy, které autor považuje za důležité, a které v době psaní brání větší adopci WebAssembly na webu. Přesněji zde bude řeč o chybějící podpoře pro automatickou správu paměti pomocí Garbage Collectiru (dále GC) a o efektivita komunikace s hostitelským prostředím. V obou případech existují návrhy, které tyto problémy řeší, avšak v době psaní se stále nachází v první fázi návrhu.

Garbage collector

Hned ze začátku je potřeba říct, že model automatické správy paměti není pro současnou, funkčnost WebAssembly zásadně důležitý. Pro co však důležitý je, a co je primární motivací představení modelu automatické správy paměti je podpora vyšších programovacích jazyků. Chybějící GC nutně neznamená, že neexistují vyšší programovací jazyky, které je možné zkompilovat do WebAssembly, naopak je jich několik a často se potýkají se stejnými problémy a tím největším je samozřejmě poskytnutí samotného GC. Objevilo se několik řešení např. tým za jazykem C# se rozhodl zkompilovat celý runtime včetně GC do WebAssembly. Do prohlížeče se poté stáhne jeden *wasm* soubor (*dotnet.wasm*) a libovolný počet *dll* souborů, které budou vykonány pomocí výše zmíněného WebAssembly souboru. Nevýhodou tohoto přístupu není pouze výsledná velikost aplikace, kdy i velmi jednoduchá aplikace vyžaduje se pohybuje v rámci

megabajtů, které je potřeba doručit přes síť, ale může se se projevit i na výkonu např. pozastavení vykonávání kódu během uvolňování paměti.

WebAssembly však nechce poskytovat svůj vlastní GC naopak chce využít hostitelů, pro poskytnutí GC, a přidat instrukce, které by umožnily tento GC využívat z libovolného jazyku. Zásadní taktéž je to, že využití GC je volitelné. Zároveň se spolu s GC zásadně rozšíří množství datových typů, kterým wasm bude rozumět. Mezi nimi jsou například pole a třídy.

Komunikace s prostředím hostitele

O této problematice již byla řeč v části *Koncepty WebAssembly*, kde byla prezentována na příkladu s předáváním hodnoty typu *String* mezi Javascriptem a WebAssembly module, obdobně lze uvažovat i o předávání dalších komplexních datových typů, jako listy, nebo slovníky. Tento přístup má však dva problémy prvním je již zmíněný glue kód.

Glue kódu má sám o sobě mnoho problémů, v kontextu webového prostředí můžeme identifikovat velikost, kdy velikost Javascriptového glue kódu může být v některých případech stejná jako velikost samotného WebAssembly modulu, fakt, že jeho existence může narušit bezpečnost, a nakonec nutnost přepínání mezi js a wasm může snižovat potencionální výkon.

Druhým problémem je existence funkcí, které patří do api prohlížeče, a jejich užívání z WebAssembly. Je sice možné takové funkce do modulu importovat, ale v téměř všech případech se jedná o funkce, které jako své parametry využívají komplexní datové typy, což znamená, že je potřeba se znovu přepnout do Javascriptu a využít glue kódu.

V obou případech již částečné řešení existuje a tím je *externref*. Jak je již známo samotné využití *externref* dokáže eliminovat velké množství glue kódu a dokáže tak částečně pomoci s prvním problémem. Co se týče druhého problému, tak jedna z vlastností *externref* je, že se jedná o referenci, a tudíž je teoreticky možné na ní volat funkce., ale neřeší druhou část problému, a to jsou datové typy parametrů. Kvůli těmto, i jiným, problémům existuje návrh pro rozšíření WebAssembly standartu o tzv. *Interface Types*.

To hlavní, a nejdůležitější, co tento návrh přináší jsou tzv. adaptéry. Jedná se o funkce, které se nachází ve speciální části WebAssembly modulu, a které dovolují využívat abstraktní reprezentaci komplexních datový typů (není přesně stanoveno, jak jsou tyto typy reprezentovány v paměti) jako své parametry, nebo návratové hodnoty. Jak by takovýto modul mohl vypadat lze vidět na obrázku 9. Jak lze již z tohoto obrázku usoudit *Interface types* eliminují glue kód tím, že ho přesouvají přímo do WebAssembly modulu. Tím, že bude glue kód přesunut do modulu samozřejmě snižujeme množství kódu, které je potřeba

```
(adapter_module
 // Funkce původního modulu, která vrací pointer a length
  (module $CORE
    (func $greet (export "greet") (result i32 i32)
      i32.const 0
      i32.const 11
    )
    (memory $memory (export "mem") 1)
    (data (i32.const 0) "hello there")
  )
  (instance $core (instantiate $CORE))
  // Funkce zabalená v adapter_func
  // vrací reprezentaci řetězce.
  (adapter_func (export "greet")) (result (string))
    ...
    // zavolání původní funkce, vrátí pointer a length
    call $core.greet
    memory-to-string $core.$memory
    ...
  )
)
```

a zvyšujeme bezpečnost a výkon. Co se týče volání webových api, tak ty *Interface types* neřeší přímo. Co ale umožňují je vytvoření
potřebných mapování, které by umožnilo WebAssembly modulu volat funkce, které patří do api prohlížeče napřímo, tedy bez nutnosti využít Javascript a ztratit část výkonu. Mapování jako takové by poté bylo vytvořeno na základě Web IDL neboli dokumentu, který definuje, mimo jiné, datové typy, které jako své parametry využívají metody patřící do api prohlížeče. Například funkce *createElement* vyžaduje parametr typu *DOMString*. Mohlo by tak být vytvořeno mapování mezi *DOMString* a dvěma čísly, které reprezentují začátek a délku řetězce, nebo vyššími datovými typy, které ve WebAssembly umožní Garbage Collector. (39, 40)

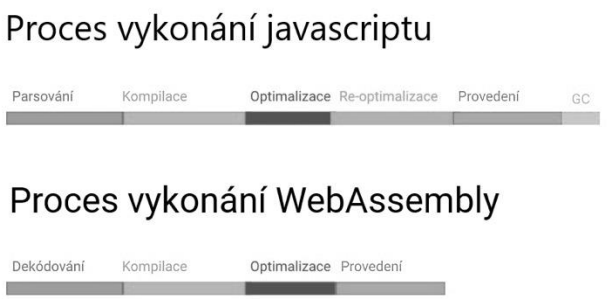
Aplikace citlivé na výkon

V této části se zaměříme na otázku, proč by se WebAssembly mělo využít. Kvůli zaměření této práce se zde zaměříme na oblast, která je s wasm do jisté míry synonymní a to výkon. Na následujících řádcích se více seznámíme s některými vlastnostmi WebAssembly, které mají implikace na výkon a jak se porovnávají s Javascriptem.

Prvním vlastností, která má dopad na výkon WebAssembly je samotný proces kompilace. Cílem kompilace je vzít zdrojový jazyk a přeložit ho do jiného, cílového jazyka, kterým může být strojový kód, bajtkód nebo jiné (20). Tohoto cíle lze dosáhnout v základu dvěma způsoby, a to využitím kompilátoru tzv. *Ahead of time compilation (AoT)* nebo interpreta. Interpret funguje na principu překládání kódu přímo za běhu a nevyžaduje tak předchozí kompilaci a umožňuje tak velmi rychlý vývojový cyklus (21). AoT kompilace je pravý opak, neboť vyžaduje zkompilování kódu před spuštěním. Vývojový cyklus tak může být celkem dlouhý, hlavně pro rozsáhlé projekty, ale často produkuje optimalizovanější kód. Javascript začal jako čistě interpretovaný jazyk a z dobrého důvodu. Původně totiž neměl sloužit k vytváření komplexních aplikací, jako je tomu dnes, ale k zajištění interaktivity. Zároveň rychlost vývoje byla a stále je požadovaná za velmi důležitou část Javascriptu. Postupem času se však webový svět dostal do fáze, ve které pouze interpret už nestačil, neboť se zvětšovaly nároky na webové stránky. Především s příchodem webových stránek a aplikací, které můžeme zařadit do *Web 2.0*. Řešení se našlo v tzv. *Just in time* kompilátorech, které „kombinují“ interprety a klasické kompilátory.

Just in time kompilace

Pozornost nyní věnujte obrázku 10. Na tomto obrázku je přibližně ilustrován životní cyklus Javascriptového a WebAssembly kódu. Jak již je očividné životní cyklus Javascriptu je o něco delší a komplikovanější. V tuto chvíli nás budou zajímat hlavně pět části, tj. parse,



Obrázek 10 Porovnání procesu vykonání js a wasm. Adaptováno

kompilace, optimalizace, re-optimalizace a garbage collection (19). Následují proces bude ilustrovány tak, jak funguje ve V8 (Javascriptový engine, který využívá Chromium), ale principy jsou podobné napříč prohlížeči. Kompilační průběh ve V8 probíhá způsobem, který je ilustrován na obrázku 11; jedná se o nejjednodušší možnou strukturu (22) a jak jsme již stanovily, samotnou kompilaci Javascriptu můžeme rozdělit do čtyř částí.

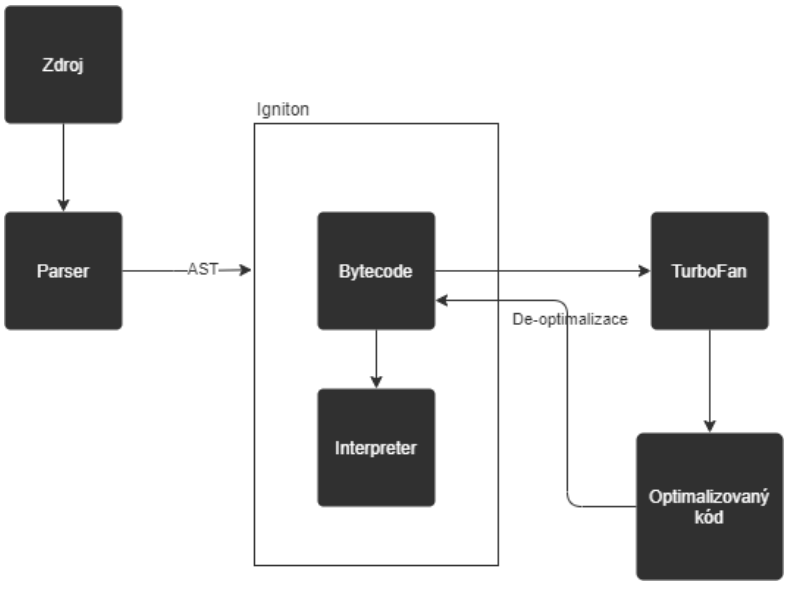
Parse

Aby mohl být kód proveden musí být převed do formátu, se kterým může dané prostředí dále pracovat. Tento formát se nazývá *intermediate representation* dále IR a jedná se o kód, kterým se v kompilátorech nebo virtuálních strojích reprezentuje strojový kód (24). Javascript pro tento kód využívá *Abstract syntax tree* (Ast) což je stromová reprezentace původního zdrojového kódu (26). Tato reprezentace je dále předána do tzv. *baseline compiler,* ve V8 *Ignition.* (23, 25)

WebAssembly na druhou stranu tímto procesem projít nemusí, protože jak jsme již v části „Představení WebAssembly“ stanovily wasm se sestává z instrukcí pro virtuální stroj, tudíž se jedná o relativně jednoduchou formu IR. Kód tak nemusí projít samotným procesem pársování, ale pouze dekódováním, protože se jedná o binární formát, a validací, která zajišťuje, že modul lze bezpečně použít. Poté je předán k první kompilaci. (25)

Kompilace

Po přijmutí AST je co nejrychleji vygenerován neoptimalizovaný, ale relativně malý bajtkód. Tento bajtkód je dále předán interpretu, který ho začne okamžitě vykonávat, aby



Obrázek 11Proces kompilace ve V8 enginu. Adaptováno

bylo možné s webovou stránkou co nejrychleji interagovat. Zároveň bude tento bajtkód držen v paměti a využit pro zajištění dalších optimalizací, neboť oproti původnímu AST vyžaduje méně paměti. Ve V8 oba procesy zajišťuje *Ignition. (23)*

U WebAssembly je tento proces velmi podobný. Namísto *Ignition* je kód předán do *Liftoff*. *Liftoff* je taktéž *baseline compiler* jehož úkolem není vyprodukovat optimalizovaný kód, ale zkompilovat kód rychle, tak aby se zamezilo dlouhému načítání webové aplikace, neboť wasm kód musí být celý zkompilován před vykonáním. Avšak oproti Javascriptu, je kód okamžitě předán do druhého kompilátoru, který se jmenuje *TurboFan.* Zatímco *Liftoff,* nebo *Ignition* generují neoptimalizovaný kód, ale generují ho velmi rychle; *TurboFan* je pravý opak, dokáže generovat vysoce optimalizovaný strojový kód, ale chvíli to trvá. WebAssembly tak projde krokem optimalizace okamžitě a téměř od samotného počátku je vysoce optimalizovaná. (19, 25, 27)

Optimalizace

Zatímco u WebAssembly je téměř optimalizace okamžitá u Javascriptu to tak není a chvíli tak trvá, než kód bude optimalizován. Namísto kompilování celého kódu je zde dále zkompilován pouze ten kód, který je často využívaný tzv. *hot code*. Tento kód je předán optimalizačnímu kompilátoru, kterým je i v tomto případě *TurboFan*. Protože Javascript je dynamický jazyk kompilátor si nemůže být jistý, že kód bude vždy využíván stejným způsobem. To se především týká volání funkce s různými datovými typy. Vygeneruje tak novou verzi, tzv. *stub*, u které udělá určité předpoklady, například, že se v té funkci budou použít pouze celá čísla, původní (neoptimalizovanou) verzi si však ponechá. Poté, když bude tato funkce zavolána s celými čísly, tak namísto staré neoptimalizované funkce bude využita nová optimalizovaná funkce. Pokud se v této hypotetické funkci začnou využívat třeba desetinná čísla, tak optimalizovaná funkce už nemůže být použita a namísto toho bude využita původní verze. Verzi optimalizovanou pro celá čísla si však ponechá a bude jí moci dále využívat. Tomuto procesu se říká de-optimalizace. Funkce však může být dále hodně využívána a může být znovu poslána k optimalizaci neboli k re-optimalizaci. (19, 25)

Re-optimalizace

Jak již bylo výše zmíněno re-optimalizace je proces „zahození“ předem zkompilované a optimalizované funkce a vygenerování nové verze stejné funkce, která splňuje potřebné

požadavky, např. využití jiného datového typu. Lin Clark například poukazuje na dva problémy, které se vyskytují v tomto procesu (25). Prvním je fakt, že opakovaným kompilováním stejné funkce zbytečně plýtváme výkon. Tím druhým je prodleva, které nastává při nahrazení optimalizované funkce funkcí neoptimalizovanou.
WebAssembly díky explicitně definovaným datovým typům procesem re-optimalizace procházet vůbec nemusí. (25)

Garbage Collection

Zbývají poslední dvě části, o kterých zde zatím nebyla řeč. První je provedení, to však bude předmětem praktické části. Druhou je Garbage Collection (GC). Garbage Collection je velmi komplexní a složitá součást mnoha moderních programovacích jazyků, včetně Javascriptu, která má dva hlavní úkoly, a to identifikovat živé a mrtvé objekty (objekty, na které se nikdo nereferuje) a uvolnit paměť okupovanou mrtvými objekty. Aby mohly být tyto úkoly splněny musí být někdy přerušeno vykonávání kódu, tzv. *zastavení světa*.
V tomto intervalu poté proběhne proces GC a po ukončení znovu začne vykonávání kódu. Ve většině případů není toto přerušení problémem, neboť jsou využívané metody a optimalizace, které minimalizují čas, který musí být vynaložen na GC. Existují však i případy, kdy GC může představovat problém například tím, že nelze přesně stanovit, kdy by měl proces GC proběhnout, což může mít negativní dopad na výkon, rychlost a případně i na interaktivitu aplikace.

WebAssembly samo o sobě GC nemá a pokud k němu v budoucnosti bude mít přístup, tak bude volitelný a mohlo by tak zjednodušit vývoj aplikací, které jsou na problémy spojené s GC citlivé.

Speciální instrukce

Druhým důvodem, proč WebAssembly může být rychlejší, než Javascript je možnost využít optimalizací, které u Javascriptu nejsou k dispozici, jako je například už jednou zmíněné SIMD. SIMD neboli *Single instruction Multiple data* je speciální instrukce, která umožňuje provádět stejnou operaci na více datech paralelně ve stejný čas (28). Důležité je zmínit to, že SIMD obecně mohou zlepšit výkon, ale nemusí být kompilátorem všude rozpoznány a implementovány, a často musí být implementovány ručně. Využití tak nachází ve výpočetně náročných aplikacích, jako je zpracování obrazu. I když je současná WebAssembly implementace omezená, především kvůli požadavku fungování nezávisle na

procesorové architektuře (29), tak i přesto má velmi dobré výsledky. V některých případech může nabízet razantní zvýšení výkonu, například čas klasifikace obrazu s využitím modelu MobileNetV2 se pouhou aktivací SIMD snížil ze 182ms na 82ms (30). U SIMD však wasm nekončí a již dnes jsou navrženy další optimalizace jako je *Quasi-Fused Multiply-Add/Subtract instructions*, které by mohly zvýšit přesnost a rychlost výpočtů s desetinnými čísly. (30)

Vlákna

Předposledním důvodem, který je zde rozebrán jsou vlákna a jejich využití. Vlákna ve WebAssembly zatím nejsou úplně standardizované, ale v době psaní již mají podporu v téměř všech prohlížečích, které wasm podporují. Samotná podpora vláken nepřináší do WebAssembly nic úplně nového, naopak pouze přináší rozšíření instrukcí o atomické instrukce, které umožnují práci s vlákny a možnost deklarovat exportovanou paměť jako sdílenou. Neobsahují tak mechanismy, jak s vlákny pracovat. Tato práce s vlákny, jako je vytváření vláken, je poté přesunuta na jazyk, který je do WebAssembly kompilován a na hostitelské prostředí, ve kterém běží. V prohlížečích se tak stejně jako Javascript spoléhají na web workers.

Web workers jsou součástí webové platformy již relativně dlouho a umožňují vykonávat kód paralelně s kódem, který běží na hlavním vlákně, kde probíhá kód a vykreslování samotné webové aplikace. Kde se Javascript a WebAssembly liší je ve způsobu, jakým tyto „vlákna“ používají. Klasický Javascript ke komunikaci mezi „vlákny“ využívá zasílání zpráv pomocí metody *postMessage,* neboť jednotlivá „vlákna“ jsou izolovaná a nesdílí tak mezi sebou kód ani objekty. Zavolání *postMessage* zašle zprávu s kopií poskytnutých dat a zpřístupní je kódu, který běží ve workeru. Ten poté může obdobným způsobem komunikovat s jinými vlákny.

WebAssembly funguje na podobném principu, kdy pomocí *postMessage* zkopíruje zkompilovaný modul a potřebný glue kód do workera. Tímto však využití *postMessage* končí, neboť WebAssembly umožňuje sdílet jednu paměť mezi více moduly, které se nachází v samostatných workrech. Tímto způsobem umožňuje emulovat chování, které je podobné nativním vláknům, dále to také znamená, že umožnuje jednoduše sdílet stav programu (31). WebAssembly pro sdílení paměti využívá *SharedArrayBuffer,* ten ale může instancovat a sdílet i Javascript přes *postMessage* a komunikovat tak s jinými workery.

Rozdíl je však v tom, že hodně jazyků, které s wasm používají mají pro tento model lepší podporu, než Javascript a sdílení komplexních objektů je tak jednoduší a v některých případech i rychlejší, protože není potřeba kopírovat data, což u velmi komplexních, především hlubokých, objektů může být, dle dostupných dat, výpočetně náročné. (32, 41)

Doba kompilace

Poslední zde probírané téma úzce souvisí s již probíraným tématem kompilace WebAssembly. Přesněji řečeno souvisí s tím, jak lze WebAssembly kompilovat.
Jak již bylo v části *Just-in-time kompilace* načrtnuto, proces kompilace WebAssembly a Javascriptu se liší v několika ohledech. Co ale nebylo probráno je, jak dlouhý proces kompilace může být a co ho může ovlivňovat.

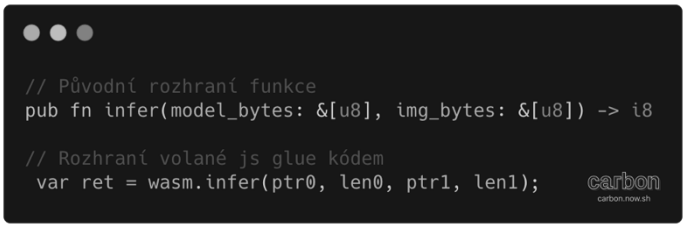
Aby bylo možné začít kompilovat Javascript je potřeba zpracovat potřebný kód předtím, než může započít samotný proces kompilace. Toto zpracování je úzce propojené s množstvím kódu, které je potřeba zkompilovat a jeho organizací. Pokud je třeba zpracovat a zkompilovat větší množství kódu, které se nachází v jednom souboru, je možné že celý proces proběhne na hlavní vlákně, což znamená, že může být relativně dlouhý a může negativně ovlivnit výkon a rychlost aplikace. Proto je nutné využít další metody, které umožní kompilátorům paralelizovat a dále optimalizovat svoji práci. (43)

U WebAssembly je tomu jinak. Díky formátu, ve kterém je WebAssembly reprezentováno, je prvotní zpracování kódu rychlejší a ukázalo se, že je možné kompilovat WebAssembly řádek po řádku. Samotné kompilování může tedy probíhat přímo na proudu bajtů ze sítě a proces může být jednodušeji rozložen mezi více vláken a nemusí dlouho blokovat hlavní vlákno. Díky těmto vlastnostem je možné kompilovat WebAssembly brzo a velmi rychle, ovšem i zde záleží na množství kódu. (42)

5. Vývoj s WebAssembly

V této části se zaměříme na to, jak se s WebAssembly vyvíjí. Budeme zde probírat jazyk Rust a co ho činní vhodný pro využití s WebAssembly.

Prvním důležitým rozhodnutím je samozřejmě výběr programovacího jazyku a v době psaní WebAssembly podporuje kolem 40 jazyků, ovšem ne u všech se dá mluvit o „plné podpoře“. Pro ilustraci na tomto projektu byl vybrán jazyk Rust. Rust je celkem unikátní systémový jazyk, je tedy rychlý a nespoléhá se na GC, ale oproti dalším systémový jazykům je velmi bezpečný, neboť se spoléhá na pravidla, pomocí kterých je schopný zachytit běžné chyby, na které lze narazit v C/C++, během procesu kompilace. Rychlost se



Obrázek 12 Ukázka transformace funkcí z jazyku Rust do js

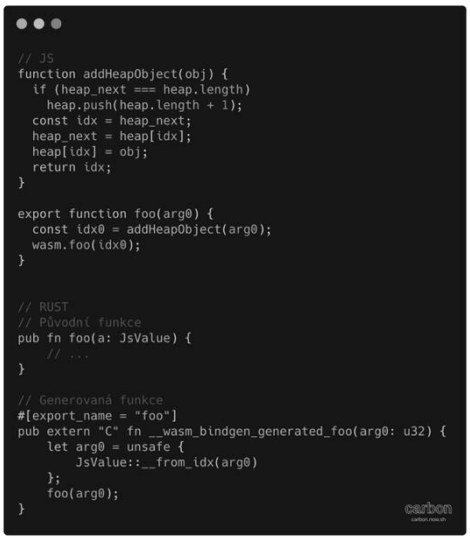
při aplikování strojového učení hodí, avšak hlavní důvod, proč byl Rust vybrán a proč je obecně považován za jeden z lepších jazyků pro wasm, je jeho ekosystém. Jak již bylo během této práce několikrát zmíněno vývoj s wasm nekončí zkompilováním modulu, například je třeba zajistit kompatibilitu datových typů, zajistit správu paměti a v některých případech je třeba transformovat rozhraní samotné volané funkce, jak lze vidět na obrázku – 13.

Mohly by také existovat požadavky na manipulaci s DOM, nebo jinými webovými api, jako například fetch. Bylo by tak vhodné abychom pro tyto, a další, potřeby mít dobře podporované knihovny, které můžou takovýto vývoj urychlit a usnadnit, nebo úzkou integraci s WebAssembly. Rust se spoléhá na podporu WebAssembly skrze několik knihoven. První a hlavní knihovnou je *wasm-bindgen*.

Wasm-bindgen

Úkol této knihovny a utility je prostý, a to zajistit komunikaci mezi wasm modulem a Javascriptem tak, aby bylo možné používat jiné datové typy než ty, které jsou WebAssembly definované a běžně dostupné, což jsou v současnosti pouze číselné datové

typy. Dále umožňuje jednodušeji využívat webové api v kódu, který bude zkompilován do WebAssembly. Jedná se tedy o částečnou, dočasnou náhradu *Interface types*, která byla více probíraná v rámci části *Limitace WebAssembly*. Podobně jako v případě *Interface*



Obrázek 13 - Zasílání livobolné hodnoty do wasm modulu. Adaptováno.

types i wasm-bindgen využívá pro vytváření těchto mapování Web Idl, které využívá i Javascript. (34). Prozatím je toto zprostředkováno glue kódem na straně Javascriptu a Rustu, jak je ilustrováno na obrázku-13. Na obrázku-13 je zobrazen kód, který importuje libovolnou Javascriptovou hodnotu do wasm modulu, toho je dosáhnuto pomocí proměnné *heap*, do které se uloží hodnota uloží a je vrácen index, tento index je poté poslán do wasm modulu, kde ho přijme transformovaná funkce, pomocí indexu najde objekt a zavolá námi definovanou funkci. (35)

Wasm-pack

Druhým nástrojem, který Rust nabízí je wasm-pack. Zatímco wasm_bindgen se soustředí na kompilaci a zjednodušení komunikace WebAssembly a Javascriptu, wasm-pack nabízí mnohem více. Jeho primární funkcí je vygenerovat, z námi poskytnutého kódu, npm balíček. V Javascriptovém prostředí se poté dá využít jako klasická závislost, která je deklarovaná v *package.json.* Celý proces primárně zahrnuje:

1. Zkompilování do WebAssembly
2. Použít wasm_bindgen a vygenerovat potřebný glue kód a rozhraní
3. Optimalizovat wasm modul pomocí nástrojů jako *wasm-opt*
4. Vygenerovat package.json

Wasm-pack dále nabízí další knihovny, které je možné ve vývoji využít jako je *wee_alloc*, miniaturní alokátor, který se zaměřuje na to, aby jeho velikost byla co nejmenší a aby se tak snížila velikost i výsledného wasm modulu. Dále nabízí velmi užitečný *console_error_panic_hook*, který chyby, které nastanou za běhu modulu vypisuje do konzole ve srozumitelném formátu. (36)

Ladění

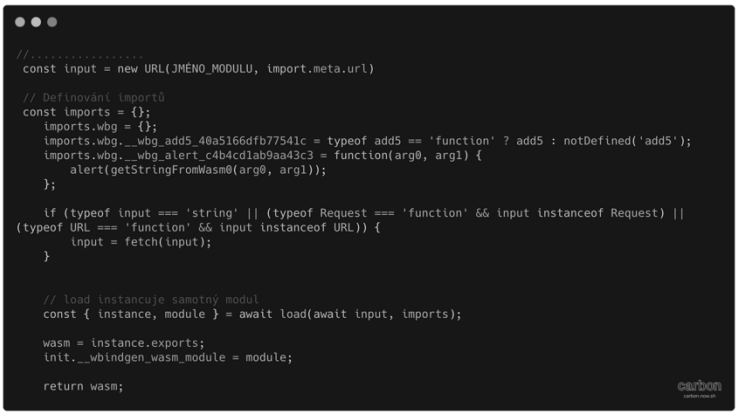
Ladění je částí každého vývoje, je tedy nutné se seznámit se způsoby, kterými lze ladit wasm moduly. Rust v současné chvíli nabízí dva způsoby, kterými lze ladit aplikaci. Prvním je výše zmíněný *console_error_panic_hook* a druhým je klasické vypisování do konzole. Co však nenabízí je možnost využít ladič. Důvod je ten, že současné ladiče, které rust podporují, kvůli chybějícím informacím, umožňují pouze pohybování se v rámci wasm instrukcí, a ne ve zdrojovém kódu. Co však rust nabízí je knihovna, která umožňuje testování WebAssembly kódu.

Většina výše zmíněných knihoven a nástrojů je využita v tomto projektu. Wasm_bindgen je zde využit pro využití konzole ve wasm modulu a k zajištění toho, aby bylo možné využití poslat do funkce model a obrázek, které jsou reprezentovány polem bajtů. Celý rust kód je poté zkompilován a zpracován nástrojem wasm-pack. Za zmínku taktéž stojí, že samotný kód napsaný v Rustu není o tolik odlišný, od kódu, který by fungoval naklasickém desktopu. Lišil by se pouze ve způsobu, kterým by byl obrázek načten.

Načítání wasm

Nyní se zaměříme na to, co mají oba přístupy stejné, a to je načítání wasm modulů, a jak se ukázalo, může tento krok být celkem problémový. Problémovost je proměnlivá a závisí na tom, jestli projekt využívá bundleru a jakou úroveň podpory pro WebAssembly nabízí.

Pokud projekt bundler nevužívá a wasm modul je hostován na stejném serveru jako inicializační kód, tak inicializace a využití WebAssembly je vcelku přímočaré viz. Obrázek-14. Po návratu z této funkce má kód přístup k funkcím, pamětem, které jsou z wasm modulu exportované.



Obrázek 14 - Instancování modulu. Zdroj: Autor

Pokud však projekt využívá nějaký bundler, a pokud projekt využívá moderní js frameworky jako react, je velká šance, že ano, může být tento proces komplikovanější. Problém spočívá především v tom, že bundlery sice umožňují lepší integraci a management různých typů souborů, ale musí vědět, jak s těmito soubory zacházet. Uživatel ho tak musí nakonfigurovat a případně poskytnout potřebné knihovny. A toto může být problém, nejen kvůli tomu, že bundlery a jejich konfigurace může být komplexní a časově náročná, ale také kvůli tomu, že některé nástroje pro generování projektů tyto konfigurace

Obrázek 15 – Jednoduché pravidlo pro bundlery. Zdroj: Autor

před uživatelem schovávají, a to nejen kvůli v předešle větě zmíněným důvodům. Pokud uživatel má přístup ke konfiguraci bundleru, tak nejjednodušší možné nastavení je zobrazeno na obrázku-15. Toto však zpřístupní pouze wasm modul, skrze vygenerovanou URL adresu. Další práce je pak v rukou knihovny nebo uživatele. Takto je WebAssembly využívané knihovnou například knihovnou tensorflow. Důležité je zde také říci, že ne všechny bundlery podobná pravidla nepotřebují.

Druhou možností, která je v tomto projektu použita je využití speciálního *loaderu*. Loadery obecně umožnují Webpacku předzpracovat soubory, může zde tak využít loader, který sám zajistí vygenerovaní kódu potřebného k inicializaci wasm modulu. Tento přístup je vhodnější pro složitější aplikace, obsahující mnoho obslužného kódu, který je exportován z WebAssembly modulu, nebo musí být správně importován, to znamená i včetně jména. Nebo pokud vývojář, chce mít svůj C++ nebo Rust kód na stejném místě jako zbytek aplikace. V těchto případech je poté jednoduší, a vhodnější, nechat tyto operace provést automaticky během sestavování aplikace.

6. Popis aplikace

Nyní se přesuneme k popisu samotné aplikace. Nejdříve se zaměříme na architekturu.

Architektura

Referenční aplikace je postavená na spa frameworku React. Mohla zde být využita libovolný framework, ale react byl vybrán, protože se jedná o pravděpodobně nejpopulárnější a nejvyužívanější Javascriptovou knihovnu po vytváření klientských aplikací. Využití knihovny bylo vybráno z důvodu ilustrování, jak lze WebAssembly využít v prostředí, které má často komplexní sestavovací prostředí např. již zmíněné bundlery. Dále je aplikace obsahuje dva způsoby nasazení WebAssembly, jedním je využití knihovny TensorflowJS a druhý je využití samostatného WebAssembly modulu. K rozdělení došlo z důvodu prezentace různých přístupů jak, lze s wasm moduly pracovat a jak složité je získat z WebAssembly dobrý výkon a rychlost.

Co se týče samotného WebAssembly modulu, tak ten je napsaný v jazyce Rust jsou zde využity knihovny wasm-bindgen, Image pro načítání obrázků a tract pro provedení samotné klasifikace. Modul exportuje dvě funkce *main*, která je volána při inicializaci modulu a *infer*, která provádí klasifikace. Funkce *infer* má dva parametry *image_bytes* a *model_bytes*. V obou případech se jedná o pole 8bitových čísel, neboť takto lze reprezentovat jak obrázek, který bude klasifikován, tak i model, který bude pro klasifikaci využit.

Pro porovnání je zde varianta aplikace, která je z převážné většiny napsaná v jazyce C# s využitím frameworku Blazor a následně zkompilovaná do WebAssembly. Pro provedení predikce využívá lehce upravený kód, který využívá i výše zmíněný WebAssembly modul, a to z toho důvodu, že během psaní této práce neexistovaly jiné způsoby, jak provést

predikci, které by byly podporované využívaným frameworkem. Framework Blazor byl vybrán především z těchto důvodů, Nachází ve fázi, kdy je v něm možné vyvíjet aplikace, které lze nasadit do produkce. Jazyk C# využívá automatické správy paměti, a není tak tím nejlepším jazykem pro současné WebAssembly.

Fungování aplikace

Jak již bylo na začátku zmíněno jedná se prototyp aplikace na adopci koček. Stránka uživateli nabízí informace o jednotlivých plemenech, která jsou na stránce dostupná k adopci. Prezentované informace pochází z několika zdrojů (cattime.com, petfinder.com a wikipedia). Dále nabízí uživateli možnost procházet seznam, všech koček, které je možná adoptovat. Tento seznam lze dále filtrovat, například tím, že uživatel poskytne fotku nějaké kočky, která bude vyhodnocena modelem umělé inteligence a seznam bude filtrován podle výsledku, kterým je identifikátor plemene. Dále obsahuje stránky pro zobrazení samotné kočky a editor, který umožňuje vytvářet záznamy o nových kočkách k adopci. Zde je také využit model umělé inteligence k určení plemena.

Dále je ještě potřeba zmínit přesnost modelu. Obě aplikace využívají stejný základní model, který je založen na modelu *Mobile Net 2.0* a přetrénován na klasifikování kočičích plemen, nakonec byl uložen ve formátu *saved model*, který využívá tensorflow. Ovšem pro využití v aplikacích musel být přeložen do dvou rozdílných formátů. Pro aplikaci, která využívá wasm modul přímo (what-cat-wasm), byl model přeložen do formátu *onnx* a zachoval si svoji přesnost. Pro využití v aplikaci, která používá TensorflowJs (what-cat-tensorflow) musel být model přeložen do jednoho ze dvou formátů. V jednom případě model nelze načíst a v druhém případě je model extrémně nepřesný. Tento fakt by však neměl ovlivnit výkon knihovny a získaná data o výkonu by měla být validní.

7. Výkon WebAssembly

Dopad WebAssembly na velikost stránek

Velmi důležitá oblast, která ovlivní Web Vitals, nehledě na to, jestli využívá WebAssembly, je velikost dat a množství kódu, které je potřeba přenést přes síť, aby webová aplikace mohla fungovat a zobrazit obsah uživateli. V následujících tabulkách jsou zachyceny velikosti jednotlivých variant. Měříme první návštěvu neboli jako kdyby uživatel navštívil danou stránku (/ nebo /pets) bez navštívení jiné.

Tabulka 1 Velikost na domovské stránce

Velikost aplikace na /			
Varianta	Přeneseno přes síť (s kompresí)	Přeneseno přes síť + cache	Celková velikost (bez komprese)
Referenční	442 kB	57.7 kB	2.1 MB
Javascript + modul	253 kB	17.8 kB	441 kB
Blazor	8.2 MB	38 kB	49.7 MB

Tato tabulka nám ukazuje data, která byl do jisté míry očekávatelná. Kvůli způsobu, jak Blazor funguje a jak je tato aplikace implementována, je potřeba přenést obrovské množství dat a už na jejich základě těchto můžeme předpokládat, že samotné Web Vitals pro využití C# pomocí WebAssembly nebudou ty nejlepší. Tato nadměrná velikost se převážně nachází v souboru *dotnet.wasm,* neboli prostředí, které bude vykonávat *.DLL* soubory aplikace. Ve většině případů je tento soubor relativně malý. V tomto případě bude jeho velikost navýšena o kód, který je potřeba k provedení klasifikace, který je napsán v jazyce rust, zkompilován a následně využit v C# kódu. Tento extra kód má sám o sobě přibližně 30 MB. Zvláštní může být velikost varianty *Javascript + wasm*, která je ve všech případech nižší než referenční aplikace. Tato redukce ve velikosti je nejspíše způsobena tím, že v této variantě nevyužíváme knihovny *Tensorflow-Js* a samotný glue kód, který wasm modul potřebuje je relativně malý. Před minifikací má 272 řádek kódu. Zároveň samotný WebAssembly modul, který bude provádět klasifikaci ještě není načtený, neboť není potřeba.

Tabulka 2 Velikost na stránce /pets

Velikost aplikace na /pets			
Varianta	Přeneseno přes síť (s kompresí)	Přeneseno přes síť + cache	Celková velikost (bez komprese)
Referenční	5.2 MB	504 kB	7.5 MB
Javascript + modul	9.5 MB	11 kB	15 MB
Blazor	16.7 MB	112 kB	59 MB

Jak lze vidět už samotné načítání modelů strojového učení způsobí relativně dramatické navýšení velikosti dat. U referenční aplikace dojde k nejmenšímu navýšení, což bude způsobeno tím, že samotný formát modelu je lépe uzpůsobený prostředí webu než formát,

který je využívaný v dalších dvou variantách. Referenční aplikace využívá model ve formátu *tfjs*, do kterého sice není možné konvertovat libovolný model, ale za to je lépe optimalizovaný, co se týče velikosti (celková velikost modelu je přibližně 4.6 MB). Zbylé dvě varianty využívají stejný model ve formátu *onnx*, tedy naprosto standartní formát modelu strojového učení, který by dokázala využít jakákoliv jiná knihovna v libovolném programovacím jazyce. Avšak není úplně dobře optimalizovaný, co se týče velikosti (celková velikost modelu je přibližně 9 MB).

Znovu se zde pozastavíme nad variantou *Javascript + wasm*, neboť u ní můžeme pozorovat zajímavou skutečnost a to, že samotné použití Web Assembly nemusí nutně negativně ovlivnit aplikaci co se týče velikosti. V této variantě má využitý WebAssembly modul velikost přibližně 5 MB, avšak po kompresi se tato velikost zmenší na přibližně 686 kB, přičemž původní kód byl napsán v jazyce Rust, který je znám, tím že produkuje velké wasm soubory. Pokud by byl tento modul napsán v C++ je pravděpodobné, že výsledná velikost by byla ještě menší. Samozřejmě je nutné dále počítat s velikostí glue kódu, který dále navýší velikost, ale jak již bylo v této části práce načrtnuto, velikost tohoto kódu nemusí být nějak drastická. Ovšem stále je velikostně větší než Javascriptové soubory, což může navýšit dobu, která je potřebná pro stažení webové aplikace, především na pomalejším připojení. Nyní, když už máme vice informací, jak WebAssembly ovlivňuje velikost aplikace je na čase se podívat na to, jak ovlivní Web Vitals.

Dopady na WebVitals - Largest Contentful Paint

První metrikou, kterou se zde budeme zabývat je jedna z těch důležitějších, metrik, která přímo ovlivňuje zážitek uživatele, a to Largest Contentful Paint (LCP). LCP je velmi přímočará metrika, která měří dobu, která uplyne od načtení dokumentu, až po vykreslení největšího dom elementu ve viewportu. Pro tuto práci je však důležité to, že je tato metrika ovlivňována vlastnostmi, které mají i testovací aplikace, a to vykreslování na straně klienta (spa aplikace) a velkou velikostí dat. Dopad těchto vlastností je dobře vidět v následujících tabulkách.

Tabulka 3 Hodnoty LCP měřené s presetem Desktop

LCP, Desktop, meřené v sekundách		
Varianta a stránka	Bez cache	S cache

Referenční na /	**0,9**	**0,3**
Blazor na /	7,40	1
Javascript + modul na /	0,8	0,3
Referenční na /pets	**1,5**	**0,4**
Blazor na /pets	45,3	2,4
Javascript + modul na /pets	1,1	0,5

Jak lze již z této tabulky vidět, tak verze s WebAssembly modulem se oproti té referenční nějak zásadně neliší. Jediný větší rozdíl můžeme pozorovat na */pets,* kde varianta s WebAssembly modulem je o 0,4 sekundy rychleji načtená. To však může být způsobeno externími vlivy, jako je využití procesoru. Co je však zajímavější jsou data, která nám poskytuje Blazor. Jak lze vidět, tak i na desktopovém hardware se tato varianta načítá velmi dlouhou dobu. Největším viníkem se zdá být prvotní stažení všech potřebných *.DLL* souborů a hlavně *dotnet.wasm*, kvůli své velikosti. Možná nejdůležitější jsou však data, která počítají s využitím cache. Jak z nich můžeme vyčíst dochází k výrazné redukci v době načítání a doba, která stále zbývá bude nejspíše rozdělená mezi samotné vykreslení DOM a načítání dat z API. Vzhledem k tomu, že doba načítání ostatních variant je v rámci milisekund, je možné, že varianta Blazor stráví většinu načítání vykreslováním DOM, což může poukazovat na pomalou interakci WebAssembly s api prohlížeče. Avšak je třeba mít na mysli, že blazor využívá *Virtual Dom*, takže tato doba není reprezentativní samotného WebAssembly. Dále způsob, kterým Blazor využívá WebAssembly a celková velikost wasm a dll souborů může výrazně přispět k prvotnímu načtení.

Tabulka 4 Hodnoty LCP měřené s presetem Mobile

LCP, Mobile, meřené v sekundách		
Varianta a stránka	Bez cache	S cache
Referenční na /	**4,3**	**0,9**
Blazor na /	45,3	2,4
Javascript + modul na /	3,2	0,9
Referenční na /pets	**5,3**	**1**
Blazor na /pets	44	2,9
Javascript + modul na /pets	4,3	1

Co se týče dat z testů s omezeném hardwaru, které odpovídá slabšímu mobilnímu zařízení můžeme pozorovat, že se od dat z desktopu zas tak zásadně neliší. I zde však můžeme pozorovat důležitou roli cáchování.

Dopady na WebVitals - Time to Interactive

Další metrikou, kterou zde budeme probírat je Time to Interactive (TTI). TTI je o něco komplexnější než předchozí metrika, ale v základu se jedná o dobu, kterou aplikace potřebuje by se stala interaktivní. I tato metrika je ovlivňována vlastnostmi testovacích aplikacích. Znovu se jedná o množství Javascriptu. Dále se však jedná o množství práce, které je třeba vykonat na hlavním vlákně a množství kódu, které je třeba vykonat. Především poslední dvě vlastnosti nemusí být složité snížit pro většinu aplikací, ale pro spa aplikace, které vyžadují Javascript pro svoji funkci, a pro aplikace, které potřebují vykonat výpočty, nebo provést nastavení při načítání aplikace, jako v případě knihovny *Tensorflow-Js* může být tato metrika celkem složitá. Toto chování můžeme pozorovat v následujících tabulkách.

Tabulka 5 Hodnoty TTI měřená s presetem Desktop

TTI, Desktop, meřené v sekundách		
Varianta a stránka	Bez cache	S cache
Referenční na /	**0,9**	**0,3**
Blazor na /	7,3	1
Javascript + modul na /	0,8	0,3
Referenční na /pets	**2,3**	**1,3**
Blazor na /pets	7,6	1,2
Javascript + modul na /pets	0,8	0,2

Nejdříve je třeba říci, že výsledků, kterých dosáhla referenční aplikace v obou případech jsou naprosto v pořádku. Oproti tomu Blazor znovu vychází jako nejhorší a to především kvůli skriptu *blazor.webassembly.js*, který je velmi důležitý neboť zodpovídá za stáhnutí všech potřebných souborů aplikace (dotnet.wasm a .DLL soubory) a její následnou inicializaci. Znovu zde tedy můžeme vidět důvod proč jazyky jako je C# nejsou v současnou chvíli ideální pro vývoj s WebAssembly. Zajímavější data lze znovu nalézt u varianty *Javascript + modul*, které jak můžeme vidět z tabulky, má v obou případech nižší TTI. Avšak pouze ten na */pets* můžeme považovat za signifikantní. Je složité přesně určit

co tento rozdíl způsobuje, můžeme tak pouze spekulovat, ale v reportu, který vygeneroval Lighthouse můžeme nalézt, že je dlouho vykonáván nějaký kód, a hodně práce je vykonáváno na hlavním vlákně. Nejpravděpodobněji tento rozdíl způsobuje doporučení, pro využití *Tensorflow-Js* a to je provedení jedné predikce nanečisto a dává smysl provést tuto první predikci při načítání stránky, především pokud se pracuje s Javascriptem. Zde tak můžeme pozorovat velkou výhodu WebAssembly a to, že WebAssembly nic takového nepotřebuje.

Tabulka 6 Hodnoty TTI meřené s prestem Mobile

TTI, Mobile, meřené v sekundách		
Varianta a stránka	Bez cache	S cache
Referenční na /	**4,3**	**0,2**
Blazor na /	45,7	2,7
Javascript + modul na /	0,8	0,3
Referenční na /pets	**9,5**	**4,7**
Blazor na /pets	44,6	4,2
Javascript + modul na /pets	2,9	0,1

Jak můžeme vidět, tak na „mobilním“ zařízení výsledky sledují stejný trend jako na desktopu, avšak v případech *referenční aplikace* a Blazoru můžeme na */pets* pozorovat větší zvýšení TTI než u *Javascript + modul*. Částečnou odpověď můžeme nalézt v reportu, kdy si můžeme všimnout, že problémy jsou znovu vykonávání kódu a práce na hlavním vlákně. Největší zpomalení můžeme vidět u Blazoru, což by mohlo znovu poukazovat na nevhodnost jazyku C# pro WebAssembly.

Dopady na WebVitals - Total Blocking Time

Tato metrika je podobná metrice TTI s tím rozdílem, že zde měříme celkový čas, pokaždé, když je hlavní vlákno blokováno na vice než 50ms. Tato metrika je tak velmi důležitá hlavně co se týče interaktivity. Mimo jiné, je tato metrika ovlivňována stejnými problémy jako TTI. I zde data více méně sledují podobný trend.

Tabulka 7 Hodnoty TBT měřené s presetem Desktop

TBT, Desktop, meřené v milisekundách

Varianta a stránka	Bez cache	S cache
Referenční na /	**0**	**0**
Blazor na /	280	340
Javascript + modul na /	0	0
Referenční na /pets	**1720**	**910**
Blazor na /pets	390	500
Javascript + modul na /pets	0	0

Kvůli podobnosti s TTI zde můžeme pozorovat podobné problémy a jejich zdroje. U Blazoru v obou případech bude na vině hlavně *blazor.webassembly.js.* Zajímavé může být že v případě */pets* varianta Blazor méně často blokuje hlavní vlákno, to nejspíše bude způsobené provedením prvotní predikce. Zvláštnost je, že Blazor má v obou případech horší výsledek, když využíváme cache. To je způsobené tím, že z neznámého důvodu, je provedení kódu v *blazor.webassembly.js* rozděleno na dvě části, nebo se začne provádět dvakrát, ale na podruhé se přeruší, což samozřejmě zvětší TBT.

Tabulka 8 Hodnoty TBT měřené s presetem Mobile

TBT, Mobile, meřené v milisekundách		
Varianta a stránka	Bez cache	S cache
Referenční na /	**140**	**20**
Blazor na /	2690	2270
Javascript + modul na /	0	70
Referenční na /pets	**7680**	**4270**
Blazor na /pets	3290	3530
Javascript + modul na /pets	0	40

I na mobilních zařízeních pozorujeme stejné chování, jako na desktopu akorát ovlivněné slabším hardwarem a připojením k síti.

Na závěr této části tedy můžeme říct, že v závislosti na způsobu použití nemusí využití WebAssembly mít negativní dopad na rychlost webových stránek a v některých případech může i přinést větší či menší přínosy ve všech zkoumaných metrikách. Velikost kladných či negativních dopadů na rychlost bude nejspíše záležet na množství a primárně velikosti

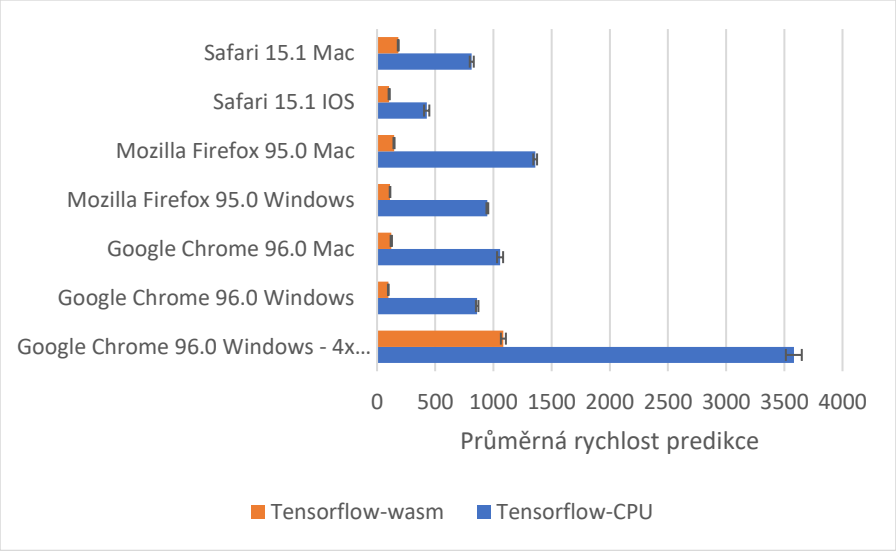
jednotlivých WebAssembly modulů. Pokud se pracuje s několika málo WebAssembly moduly, které jsou velikostně v rámci nižších megabajtů, před kompresí, tak nejspíše nebude znát citelný dopad, na rychlost, kromě potencionálně většího množství dat ke stažení. Jak už zde jde poznat důležité bude také to, jaký jazyk typ programovacího jazyku bude s WebAssembly využi, jak můžeme pozorovat u Blazoru.

Výpočetní výkon WebAssembly

V této části se zaměříme na výkon WebAssembly pro využití u strojového učení. Bude nás tu tedy zajímat, jak si WebAssembly povede ve vykonání predikce v porovnání s javascritpem. Podle probrané teorie a dalších článků můžeme očekávat, že by wasm mělo být rychlejší, otázkou je však o kolik rychlejší a jak jednoduché je tohoto výkonu dosáhnout a jestli bude výkon konzistentnější napříč prohlížeči. Pro lepší přehlednost bude tato část rozdělená do částí, ve kterých se budou porovnávat jednotlivé varianty k referenční variantě, která zde bude reprezentována *Tensorflow-CPU*.

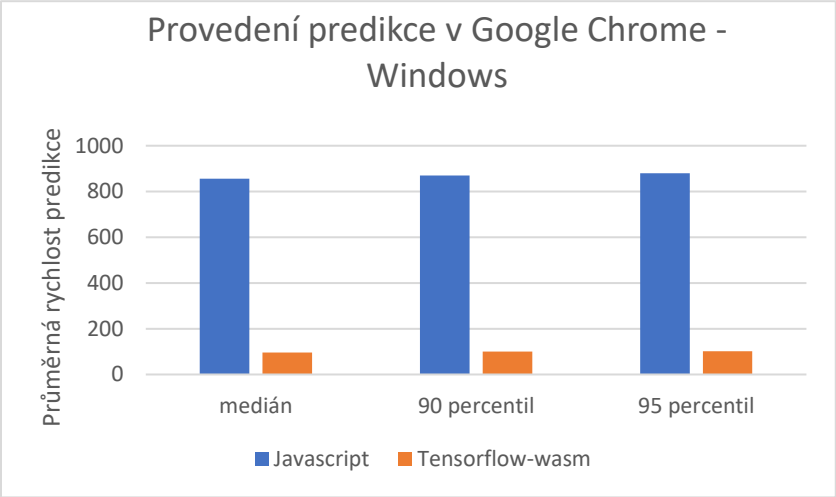
Optimalizované Tensorflow-Wasm a jeho variant

Jako první variantou se zde budeme zabývat wasm backendy knihovny Tensorflow-Js, především z toho důvodu, že nabízí jak jednoduchý wasm modul, tak i moduly, které



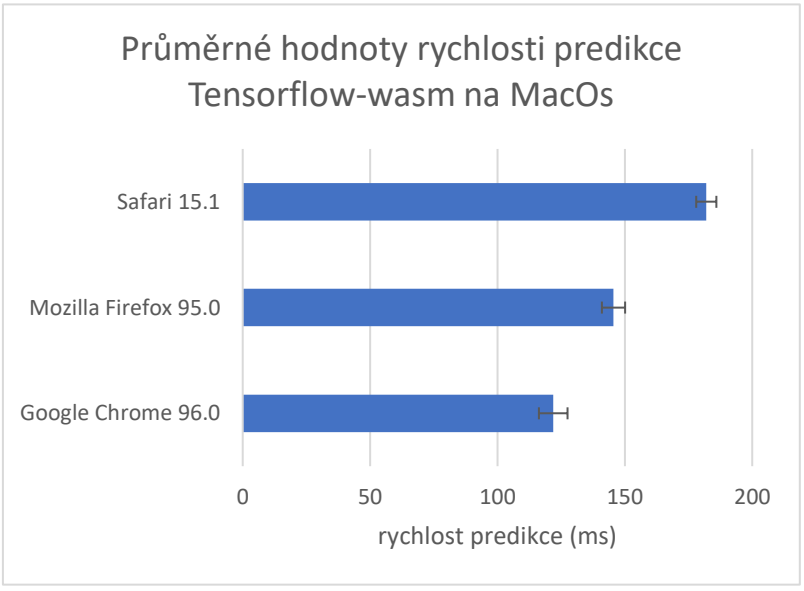
Graf 1Porovnání průměrné rychlosti predikce mezi Javascriptem a Tensorflow-wasm

nabízejí pokročilejší funkce WebAssembly, jako SIMD a využití vláken. Můžeme zde tedy pozorovat, jak velký dopad na výkon mají. Na následujícím grafu můžeme pozorovat, jak se porovná doba predikce u varianty s Javascriptem a u standartního WebAssembly modulu. Jak již z tohoto grafu můžeme pozorovat i základní WebAssembly modul bez využití jakýchkoliv další schopností WebAssembly orientovaných na výkon dokáže



Graf 3 Perencitly v porovnání js a Tensorflow-wasm

s přehledem porazit samotný Javascript v celkem náročném výpočetním testu. Dále můžeme pozorovat, a toto by se mělo opakovat u všech WebAssembly variant, je relativně konzistentní výkon napříč prohlížeči, například na zařízení s MacOs. Jak naznačuje přiložený graf, můžeme celkově pozorovat relativně podobný výkon mezi prohlížeči, kde rozdíl mezi nerychlejším a nejpomalejším prohlížečem je přibližně 60 ms, v Javascriptu se jedná o rozdíl více jak 500ms.

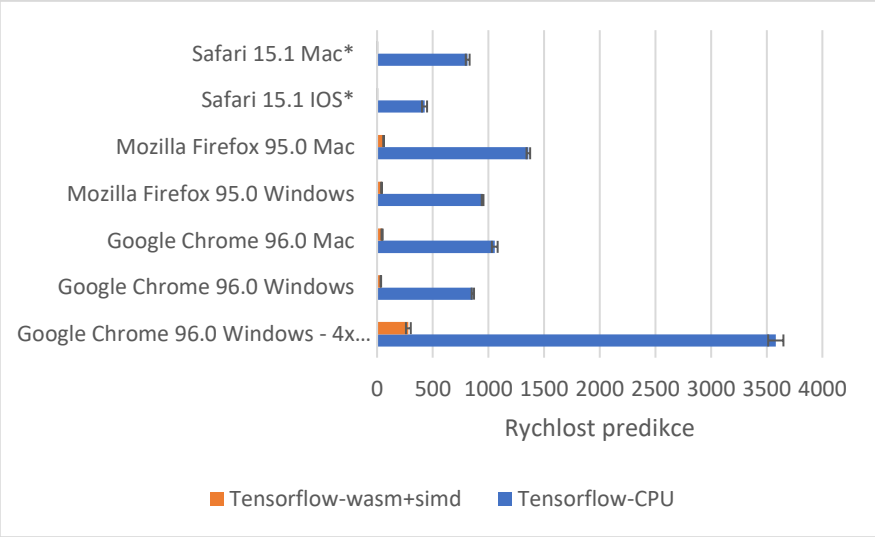


Graf 2 Průměrné hodnoty Tensorflow-wasm na MacOS

V obou případech jsou variační koeficienty menší než 10 %, průměrné hodnoty by tak měly dobře reprezentovat celý set. Jediným prohlížečem, který má hodnoty relativně razantně vyšší (tedy nižší výkon) je Safari, což by mohlo naznačovat horší implementaci, WebAssembly kompilátoru. Pokud se však podíváme na konzistenci v rámci provedení predikcí v jednom prohlížeči, můžeme vidět, že rozdíl není nějak závratný a při porovnání var. koeficientů vychází, Javascript dokonce o něco lépe (1,2 oproti 3,1). Konzistence výkonu i mezi prohlížeči je jedna z oblastí, ve které by WebAssembly mělo nabízet zlepšení oproti Javascriptu a detailněji se na tuto oblast zaměříme později.

Zároveň na grafu 1 můžeme vidět, že výkon WebAssemly vypadá velmi konzistentně i napříč různými zařízeními s různě rychlím hardwarem. Nakonec na grafu 1 můžeme také pozorovat relativně dobré škálování i na slabší hardware, kdy WebAssembly bylo schopno provést predikci na 4krát zpomaleném procesoru pod 1 sekundu, zatímco Javascriptu to trvalo 3,5 sekundy.

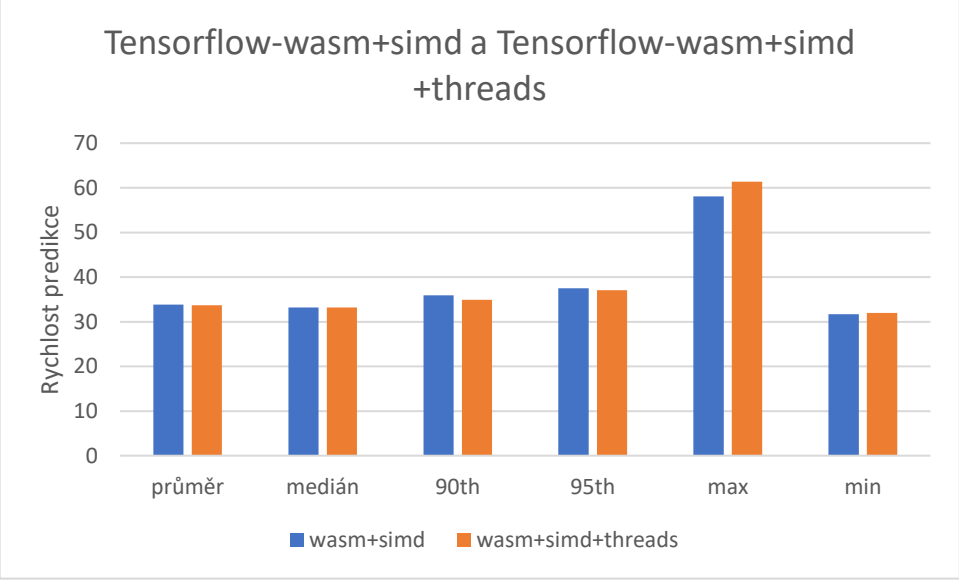
Nyní se přesuneme na variantu WebAssembly modulu, který je zkompilovaný s podporou SIMD, což by mělo dále snížit čas potřebný k provedení predikce. A jak následující graf prezentuje, tak v některých případech může využití SIMD odemknout celkem drastické zvýšení výkonu. I v tomto případě zde pozorujeme stejný trend, jako v předchozím



Graf 4 Porovnání průměrné rychlosti predikce mezi Javascriptem a Tensorflow-wasm+simd

případě, kdy WebAssembly jednoduše porazí Javascript. Tím nejpozoruhodnějším, co můžeme z tohoto grafu vyčíst, je predikce na 4krát zpomaleném procesoru, kde můžeme pozorovat přibližně 3.8krát vyšší rychlost predikce. Tyto výsledky tak může být dalším důkazem toho, že WebAssembly může najít využití pro aplikace, které jsou více náročné na procesor, a to kvůli tomu, že SIMD, nenachází exkluzivní využití pouze u strojového učení, ale i v dalších oblastech. Dále to také může znamenat, že WebAssembly by mohlo najít uplatnění pro aplikace, či programy, které cílí na zařízení se slabým hardwarem. Může, zde také pozorovat podobnou konzistenci napříč prohlížeči, kromě Safari, kde SIMD nebylo v době psaní podporováno.

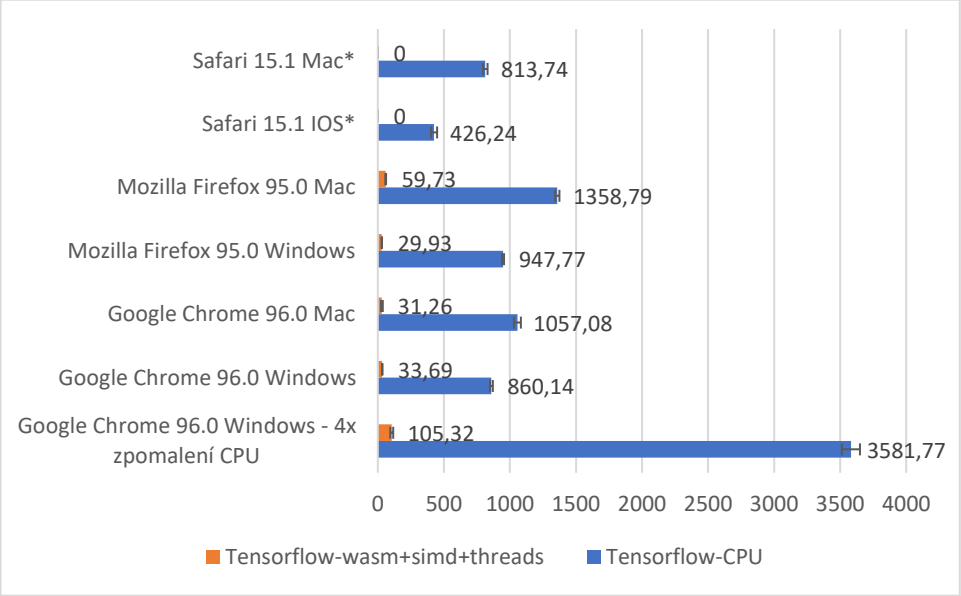
Jako poslední možnost nabízí knihovna Tensorflow-Js modul, který dokáže pracovat s vlákny. Dávalo by smysl, že vlákna by měla odemknout další výkon, který by aplikace mohla využít. Následující graf poskytuje náhled.



Graf 5 Porovnání dat na zařízení s Windows. Data zachycena na Google Chrome

I zde pozorujeme vyšší výkon než u Javascriptu, ale v tuto chvíli je nutné zasadit tento výkon do kontextu předchozích zmíněných. Jak můžeme pozorovat, z následujících dat z testů v Google Chrome (ale podobný trend je i u ostatních testovacích prostředích) tak podle těchto dat nemůžeme říct, že vlákna obecně zvýší výkon, co se týče využití procesoru, což je zvláštní. Je možné, že v knihovně byly provedeny změny, které toto chování zapříčinily, neboť podle dat, které autoři knihovny poskytují by se měla rychlost predikce pohybovat v okolí 12 milisekund.

Je však pravděpodobné, že jinak měříme rychlost predikce, ale i tak bychom měli být schopni pozorovat zásadnější rozdíl ve všech případech. V jednom případě však k většímu navýšení výkonu došlo, a to v případě zpomalení cpu. V tomto případě došlo znovu k celkem drastickému navýšení výkonu. Znovu zde tedy dostáváme najevo, že hypotéza ohledně vhodnosti využití WebAssembly na slabším hardware, může být pravdivá. Dále



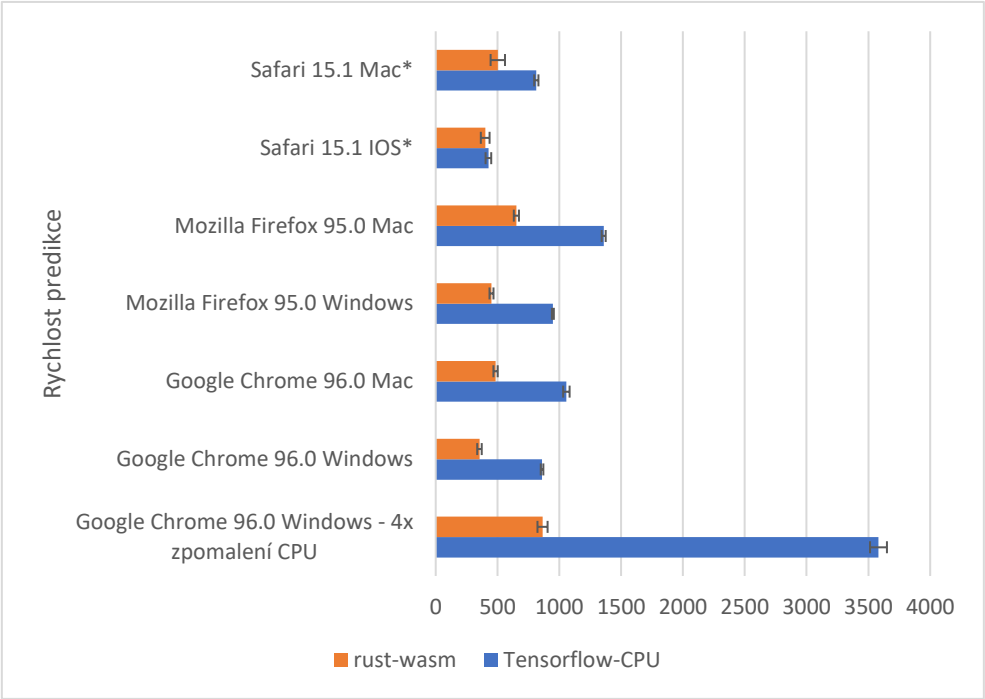
Graf 6 Porovnání průměrné rychlosti predikce mezi Javascriptem a Tensorflow-wasm+simd+threads

můžeme říct, že pokud bychom naměřili podobný nárůst výkonu i v ostatních testech mohly bychom, zde říci, že výpočetní potenciál WebAssembly je skutečně výtečný. Ovšem s těmito výsledky je složité říct, jak přesně se výkon škáluje. Zároveň je i v této možnosti využito SIMD, a je tedy složité přesně určit, jak velký dopad na výkon samotná vlákna mají.

Jednoduchý Rust modul na provedení predikce

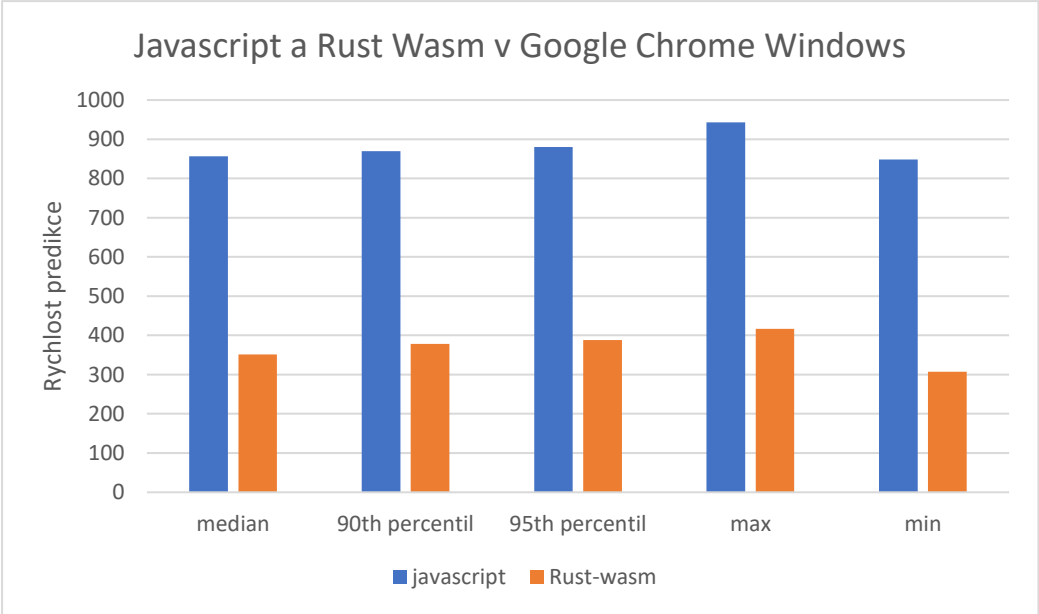
V této části se budeme zabývat WebAssembly modulem, který obsahuje jednoduchou implementaci provedení predikce s knihovnou *tract* a bude se zde zabývat dvěma otázkami. První otázkou bude, jak složité, je napsat WebAssembly modul, který bude výkonnější než Javascript. Druhou otázkou, jak moc ovlivní výkon použití jazyku C#, který není, jak jsme již stanovily, ideální pro vývoj s WebAssembly.

Odpověď na první otázku můžeme najít v následujícím grafu, který zobrazuje průměrnou dobu predikce Rust WebAssembly modulu v porovnání s Javascriptovou variantou.



Graf 7 Porovnání průmerných predickí rust wasm a javascript

Jak můžeme vidět, tak i tento relativně jednoduchý a neoptimalizovaný WebAssembly modul má oproti Javascriptu celkem solidní výkon. Není sice tak dobrý jako v případě knihovny Tensorflow-Js, ale i tak se jedná o dobrý výsledek, hlavně s ohledem na to, jak jednoduché bylo tento modul naprogramovat a poté použít v aplikaci. Jsou zde dva zajímavé případy.

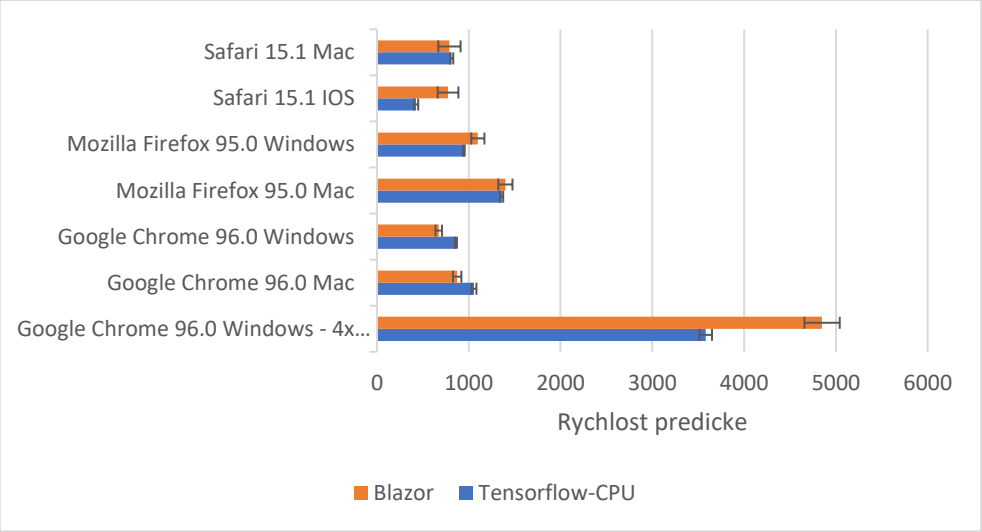


Graf 8 Detailnější data rust wasm a javascript

První je, že rozdíl mezi rychlostí predikcí na zařízení s IOS nebyl zas tak velký, což jen dokazuje optimalizaci Safari. Tím druhým případem je, že WebAssembly v tomto případě není o tolik rychlejší v testu se zpomaleným procesorem, jako tomu bylo u předchozích variant. Což by mohlo narušovat hypotézu o WebAssembly na slabším hardware, ale také to může naznačovat, jak moc výkonu lze z WebAssembly získat dalšími optimalizacemi, které nejsou Javascriptu možné nebo jednoduché. Lepší pohled na rozdíl mezi Javascriptem a modulem poskytuje graf-8.

Co se týče otázky, jak moc bude výkon ovlivněn použitím C#, tedy jazyku, který není v současné situaci nejlepší volbou, následující graf může poskytnout, alespoň teoretický pohled, neboť se zde používá stejný kód, který je i ve WebAssembly modulu, tj. kód napsaný v jazyce rust a poté volán ze C#. Takovéto využití bude mít své implikace co se týče výkonu.

A jak zde můžeme vidět, tak využití v jazyce C# rychlost vykonání predikce skutečně ovlivnilo. Ve většině případů je v implementace v Blazoru o něco málo pomalejší než

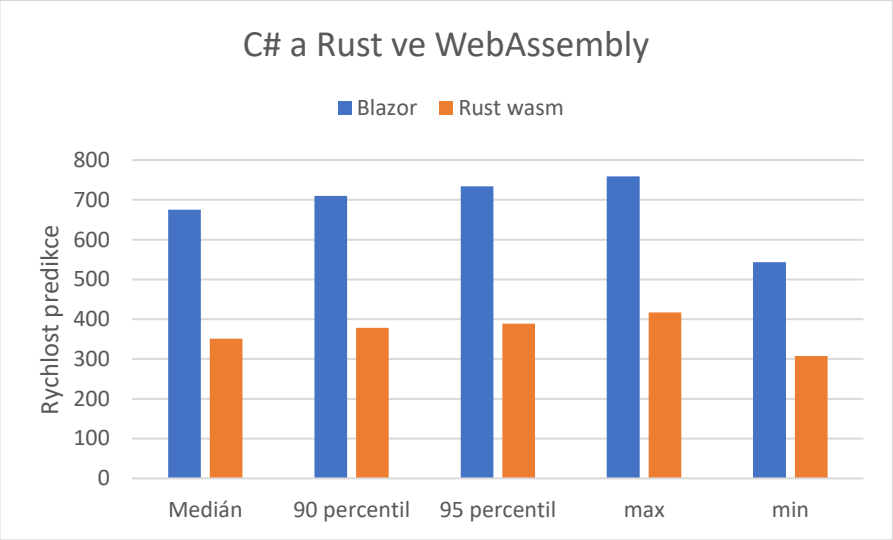


Graf 9 Porovnání průmerných predickí Blazor a javascript

referenční varianta v javascritpu, jediné případy, kdy je rychlejší můžeme nalézt v Google Chrome jak na Windows, tak i na MacOs. I v tomto případě můžeme pozorovat ve většině případů relativně podobný výkon napříč prohlížeči. Nemusí se to tak zdát, ale i tyto výsledky jsou až překvapivě dobré. Je potřeba si uvědomit, že v této variantě využíváme programovací jazyk, který potřebuje svůj vlastní garbage collector, JIT kompilátor a celkově své vlastní prostředí, ve kterém bude kód vykonáván, a ještě k tomu vykonává

kód, který je napsán v jiném programovacím jazyce. Zároveň zde tak máme, alespoň částečné potvrzení, že C# a podobné jazyku nejsou v současnou chvíli úplně vhodné, pokud za využitím WebAssembly je touha po vyšším výkonu ve strojovém učení, ale nejsou ani úplně špatné.

Celkové porovnání mezi samotným Wasm modulem a využitím v C#, v prohlížeči Google Chrome, lze nalézt v následujícím grafu.



Graf 10 Porovnání Rust a C# na Windows v Google Chrome

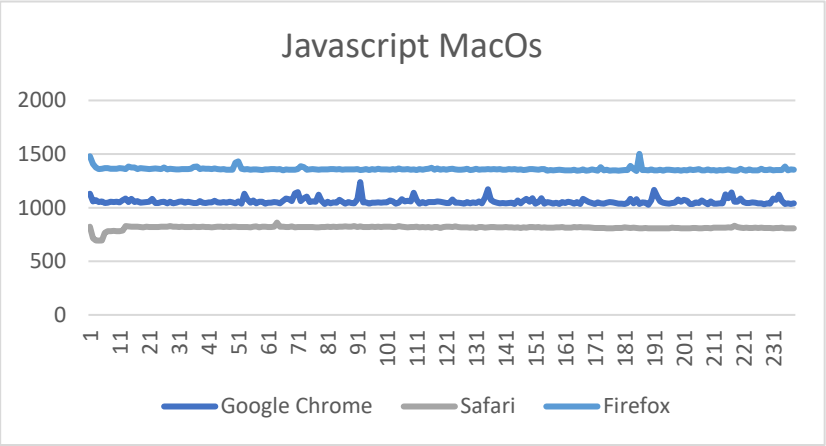
Co se týče průměrů nejsou mezi nimi dramatičtější rozdíly, avšak jednotlivé percentily ukazují, jak moc je výkon, v tomto případě, ovlivněn použitím C#. Což dále potvrzuje to, že jazyk C# a nejspíše i další podobné jazyku nejsou pro vývoj s WebAssembly úplně ideální, avšak nemůže říct, že by se neměli používat. Je také složité ukázat přesně co způsobuje nižší výkon u C#, tj. je to způsobeno jazykem, nebo způsobem, kterým tento jazyk využívá WebAssembly.

Z přiložených dat můžeme prohlásit, že co se týče strojového učení, a nejspíše i v dalších podobných případech, WebAssembly skutečně může přinést ve většině případů dramatické navýšení výkonu, a to i v relativně primitivních implementacích. Výkon, který zde pozorujeme bude z části způsoben tím, že WebAssembly je předem zkompilovaná, s pevnými datovými typy, což umožní prohlížeči, nebo jinému prostředí lépe optimalizovat cílový strojový kód. Zároveň zde pozorujeme i dopad věcí, které WebAssembly přináší na webovou platformu, jako je lepší práce s vlákny, využití SIMD, nebo další optimalizace,

které byly zpočátku určeny pro nativní kód, a které dále zlepšují výkon i oproti základnímu WebAssembly. Jediný případ, kde k takovému navýšení došlo je v případě Blazor (C#), což není překvapení, a jak jsme již navrhly, tak se stále jedná o dobrý výsledek. Znovu zde však vidíme, že ač jde do WebAssembly teoreticky kompilovat velké množství různých jazyků, tak ne všechny dosáhnou stejných výsledků.

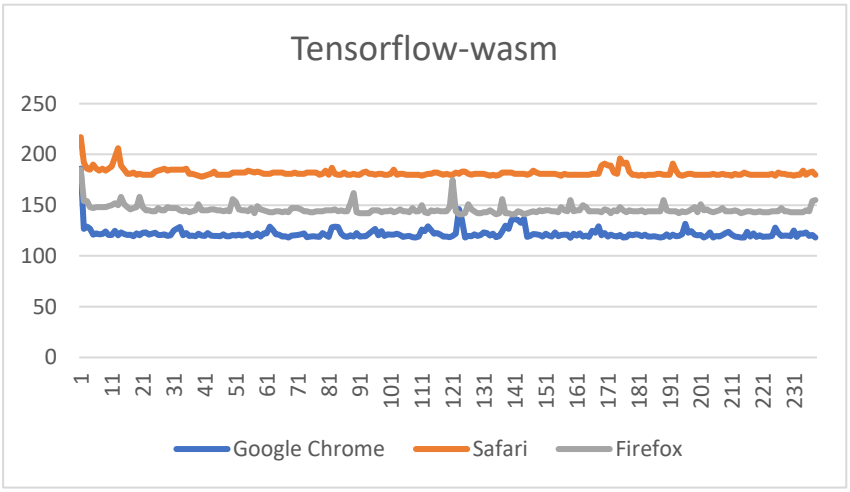
Konzistence WebAssembly

U měření výkonu, tedy využití procesoru, je velmi dobré vědět, jak moc je tento výkon přenositelný, a jak moc je závislí na prostředí, ve kterém je prováděn, zároveň konzistence je často zmiňovaná jako oblast, ve které by mělo být WebAssembly lepší než Javascript Jak jsme již načrtly v předchozí části, a co ilustrují i následující grafy můžeme pozorovat,



Graf 11 Konzistence provedení predikce v javascriptu na zařízení s MacOS

že co se konzistence v rychlosti provedení často vychází Javascript lépe, než WebAssembly varianty. Jak můžeme vidět, tak Firefox a Safari jsou velmi konzistentní

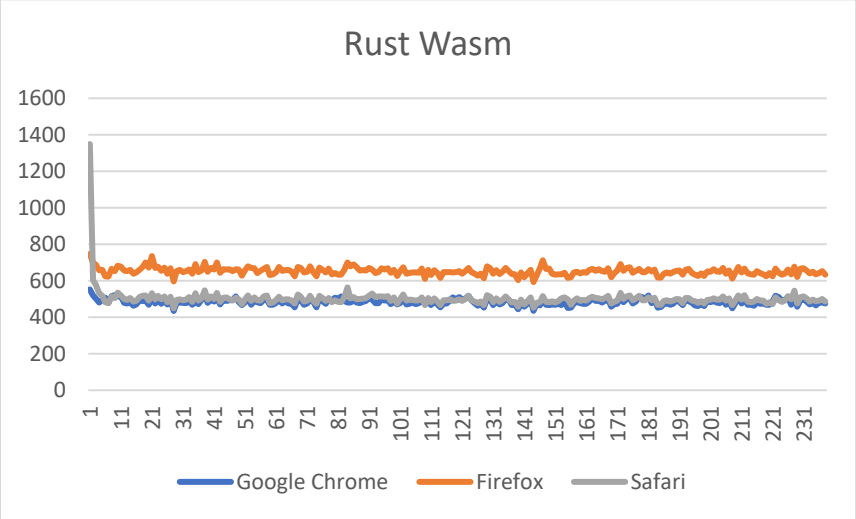


Graf 12 Konzistence provedení predikce v Tensorflow-wasm na zařízení s MacOS

s několika málo výkyvy. Google Chrome je poté méně stabilní. Zároveň je vhodné

připomenout, že rozdíl mezi nejrychlejším a nejpomalejším prohlížečem je přibližně 545ms tedy zpomalení přibližně 67 %. Pokud se nyní přesuneme na variantu Tensorflow-Wasm, tak narazíme na podobné chování.

Jak z grafu 12 můžeme vypozorovat, znovu jsou zde dva prohlížeče, v tomto případě Google Chrome a Firefox, které mají velmi podobný průběh. Rozdíl mezi nejrychlejším a nejpomalejším prohlížečem je v tomto případě kolem 50ms (zpomalení o 49,3%). Velmi podobný trend můžeme pozorovat i v Tensorflow-Wasm+SIMD, zpomalení o přibližně 31,1 % a Rust wasm, zpomalení o 34,8 %. Kde ovšem kde tento trend neplatí je Tensorflow-Wasm+SIMD+Threads, zpomalení o 91 %, a Blazor, zpomalení 77, 3 % oproti nejrychlejšímu prohlížeči.



Graf 13 Konzistence provedení predikce v Rust wasm na zařízení s MacOS

V tuto chvíli je nutné zmínit relativně velké občasné výkyvy hodnot, především u Tensorflow, WebAssembly variant. V tuto chvíli není jisté, čím tyto velké hodnoty mohou být způsobeny. Jednou hypotézou může být, že se jedná o nějakou vlastnost knihovny, neboť u modulu napsaném v Rustu se tolik neobjevují. Výjimkou je prvotní hodnota v Safari, což mohlo být způsobeno začínáním testu o chvíli předtím, než byl modul plně zkompilován, nebo byl z nějakého důvodu využit neoptimalizovaný modul, neboť první dva testované obrázky mají podobné vlastnosti, a tak velkou hodnotu má pouze první. Další hypotézou může být dopad velikosti a rozměru testovaných obrázků, avšak rozměry jsou si velice podobné a velikost by u Tensorflow-Wasm variant, nemělo mít velký vliv.

Kde by velikost mohla mít vliv by byl Rust modul, ale jak můžeme vidět, žádný větší vliv zde nepozorujeme.

I s těmito informacemi není úplně jednoduché stanovit, jestli tvrzení o lepší konzistenci je pravdivé, neboť existují různé úhly pohledu. I když jako hlavní ukazatel vezmeme rozdíl mezi nejrychlejším a nejpomalejším prohlížečem nemůžeme říct, že pro využití u strojového učení může být WebAssembly nějak zásadně konzistentnější mezi prohlížeči než Javascript. Avšak výkyvy mezi jednotlivými variantami, které využívají WebAssembly, nejsou mezi téměř všech případy nějak velké, a to i napříč testovanými zařízeními. Všechny nárůsty lze nalézt v následující tabulce, ignorujeme zde zařízení s iOS a simulovaný 4x zpomalený procesor.

Tabulka 9 Přehled zpomalení doby provedení predikce mezi nejrychlejším a nejpomalejším prohlížeči

	zpomalení – MacOs	zpomalení - Windows
Tensorflow-CPU	67 %	10,2 %
Tensorflow-Wasm	49,3 %	15, 4 %
Tensorflow-Wasm+SIMD	31,1 %	18,6 %
Tensorflow-Wasm+SIMD+Threads	91,1 %	12,6 %
Rust-wasm	34,8 %	27, 5 %
Blazor	77,3 %	63,2 %

8. Shrnutí výsledků

V rámci této práce bylo popsáno, co WebAssembly je, jak funguje, jak je možno WebAssembly využít a o významných předchůdcích. Dále zde byly představeny charakteristiky WebAssembly, které by měly vyústit v lepší využití procesoru a tím i lepší výkon. V praktické části byl velmi krátce představen jazyk Rust a co ho dělá vhodný pro vývoj s wasm, byly zde představeny některé způsoby, jak lze wasm načítat a ladit. Nakonec zde byly provedeny experimenty, které zkoumaly výpočetní výkon a rychlost WebAssembly, Jako první byla zkoumaná rychlost načítání a velikost stránek.
Z výsledných dat vyšlo najevo, že pokud je WebAssembly modul rozumně velký, tedy v rámci jednotek megabajtů před kompresí, tak se zdá, že žádný negativní dopad na načítání nemá, a dokonce může i načítání zrychlit. WebAssembly tak nemusí způsobovat problémy u samotného načítání, ale může potencionálně prodlužovat stahování, protože soubory WebAssembly jsou pořád relativně velké v porovnání s Javascriptovými soubory.
Co se týče výpočetního výkonu, tak ve strojovém učení, vidíme velkou redukci v času potřebného na provedení predikce. Jediným případem, kde čas predikcí byl podobný, a někdy i větší, než javascript byla varianta, která využívala jazyk C#, tedy jazyk, který není v současnosti úplně vhodný, ale i tak se jedná o dobré výsledky. Prezentovaná data však nelze brát jako důkaz, že WebAssembly bude vždy rychlejší než Javascript. Tato práce se pouze zabývala aplikací (strojové učení), která vytíží procesor, je tedy možné, že u podobných případů použití bude dosahovat podobných výsledků, ale budou existovat i taková použití, u kterých bude Javascript stále o něco rychlejší.
S výkonem souvisí i jak moc je tento výkon konzistentní mezi prohlížeči. V tomto ohledu nelze jasně říct, že WebAssembly je nějak závratně konzistentnější než Javascript.

9. Závěry a Doporučení

Webová platforma je velmi pozoruhodná, neboť úplně neví, jestli se vydat směrem stránek, které se zaměřují na pouhé prezentování a sdílení obsahu, nebo na plnohodnotné náhrady historicky desktopových aplikací, jako je, z nejbližší doby, webová verze Adobe Photoshop a nástroje typu Codesandbox, které nabízí relativně použitelné vývojové prostředí. Zároveň je vhodné, kvůli charakteristikám webové platformy, těchto cílů s co nejmenší velikostí. Dále stoupající popularit jazyku Typescript, která implikuje touhu webových vývojářů po využívání vlastností, které staticky typované jazyky jako je C++ mají od samotného začátku. A nyní zde máme WebAssembly, jazyk, který byl vytvořen s cílem umožnit publikovat určité typy aplikací, které by bylo složité naprogramovat v Javascriptu, na webovou platformu. Avšak dnes můžeme I uvažovat napsání celé webové stránky s využitím WebAssembly a někdy I bez využití HTML.

Co se týče psaní celých webových stránek s WebAssembly, není zde možné udělat obecný závěr, neboť kromě Blazoru existují i alternativy, které jsou napsané v jazycích, které je jednoduší s WenAssembly používat, což ovlivní velikost i výkon aplikace. Avšak technologii Blazor lze doporučit především pokud bude především pokud bude výsledná aplikace využívaná interně, nebo několika málo lidmi, hlavně na desktopových zařízení, nebo na zařízeních, která jsou připojená k WI-FI. Kde pozorované problémy, nebyly tak velké. Hlavním doporučením je zde nepodceňovat dopad, který má na rychlost aplikace a UX uživatele cachovaní. Je také preferováno věnovat čas implementaci PWA (Progressive Web App), která umožňuje, kromě cáchování, využívat aplikaci i v případě kdy uživatel není připojen k internetu. Toto doporučení můžeme rozšířit i na použití s jinými frameworky i pro případ, kdy by byl využíván pouze jeden, rozumně velký WebAssembly modul.

Ohledně použití WebAssembly pro výpočetně náročné situace nebo aplikace zde narážíme na podobný problém. V této práci byl jako benchmark použit jeden specifický případ a výsledky tak nebudou aplikovatelné na všechno. Jako rozumné doporučení můžeme považovat to, že určitě dává smysl uvažovat o použití WebAssembly pokud se narazí na situace, kdy implementace v Javascriptu má slabý výkon, nebo v situaci, kdy existuje knihovna napsaná C/C++ nebo Rust. Důležité je však tyto implementace porovnat a rozhodnout se podle výsledků. Pokud se však jedná o využití se strojovým učením, tak

pokud je potřeba využívat modely, které jsou podporovány knihovnou Tensorflow-Js, tak je nejlepší využít WebGl back-end, neboť je stále rychlejší než WebAssembly a jakmile bude nové API WebGPU dostupné bude pravděpodobně ještě rychlejší a problémy s pamětí budou nejspíše vyřešeny. Ovšem pokud modely nejsou knihovnou podporované, tak napsání kódu a zkompilování pro predikci je také relativně jednoduché a stále dosáhne dobrého výkonu. Zároveň se však musí počítat s tím, že použití WebAssembly v současné době zvýší množství dat, které je potřeba stáhnout, což může být v některých případech velká překážka. Kromě aplikacích, které vytěžují procesor existují i další potenciální oblasti, ve kterých by se dalo WebAssembly využít, a které by stály za další bádání, jako je například prostředek pro interoperaci jazyků, jako alternativa pro kontejnery, využití v orchestračních systémech, vývoj beck-endových částí webových aplikací, vývoj webových stránek s využitím cross-platform frameworků a přístupů jako například nahrazení HTML DOM canvas elementem pro vykreslování uživatelského rozhraní.

10. Použitá literatura

[1]: *WebAssembly* [online]. [cit. 2021-4-30]. Dostupné z: https://webassembly.org/

[2]: Bringing the web up to speed with WebAssembly. *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* [online]. 2017, **2017**, 185 - 200 [cit. 2021-4-30]. Dostupné z: doi: https://doi.org/10.1145/3062341.3062363

[3]: CLARK, Lin. Creating and working with WebAssembly modules. *Mozzila hacks* [online]. 2017 [cit. 2021-4-30]. Dostupné z: https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/

[4]: ROURKE, Mike. *Learn WebAssembly* [online]. Packt, 2018 [cit. 2021-4-30]. ISBN 9781788997379. Dostupné z: https://Javascript.packtpub.com/product/learn-webassembly/9781788997379

[5]: WebAssembly Community Group. WebAssembly Specification. [online], 2021, 149 – 156. [cit. 2021-5-2]. Dostupné z: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[6]: Bringing the web up to speed with WebAssembly. *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* [online]. 2017, **2017**, 195 [cit. 2021-4-30]. Dostupné z: doi: https://doi.org/10.1145/3062341.3062363

[7]: CLARK, Lin. Standardizing WASI: A system interface to run WebAssembly outside the web. *Mozzila hacks* [online]. 2017 [cit. 2021-5-2]. Dostupné z: https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/

[8]: WebAssembly Community Group. WebAssembly Specification. [online], 2021, 15 – 21. [cit. 2021-5-2]. Dostupné z: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[9]: *Understanding WebAssembly text format* [online]. [cit. 2021-5-4]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format#reference_types

[10]: *Technical Overview* [online]. [cit. 2021-5-5]. Dostupné z: https://developer.chrome.com/docs/native-client/overview/

[11]: *NaCl and PNaCl* [online]. [cit. 2021-5-5]. Dostupné z: https://developer.chrome.com/docs/native-client/nacl-and-pnacl/

[12]: URBAN, Hynek. *WebAssembly a cesta k němu* [online]. 2015 [cit. 2021-5-7]. Dostupné z: https://fragaria.cz/blog/2015/07/23/webassembly-cesta-k-nemu/

[13]: ZAKAI, Alon. *Why WebAssembly is Faster Than asm.js* [online]. 2017 [cit. 2021-5-7]. Dostupné z: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/

[14]: *Asm.js* [online]. 2020 [cit. 2021-5-7]. Dostupné z: https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js

[15]: *Asm.js - frequently asked questions* [online]. [cit. 2021-5-7]. Dostupné z: http://asmjs.org/faq.html

[16]: *Loading and running WebAssembly code* [online]. [cit. 2021-5-9]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running

[17]: *Web support for Flutter* [online]. [cit. 2021-5-10]. Dostupné z: https://flutter.dev/web

[18]: About Skia. *Skia* [online]. [cit. 2021-5-10]. Dostupné z: https://skia.org/about/

[19]: CLARK, Lin. A crash course in just-in-time (JIT) compilers. *Mozilla hacks* [online]. [cit. 2021-5-11]. Dostupné z: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

[20]: Compiler. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-11]. Dostupné z: https://en.wikipedia.org/wiki/Compiler

[21]: Compilers versus interpreters. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-11]. Dostupné z: https://en.wikipedia.org/wiki/Interpreter_(computing)#Compilers_versus_interpreters
[22]: *BlinkOn 6 Day 1 Talk 2: Ignition - an interpreter for V8 – BlinkOn. In: Youtube* [online]. 2016 [cit. 2021-5-11]. Dostupné z: https://youtu.be/r5OWCtuKiAk?t=589
[23]: VERWAEST, Toon. Blazingly fast parsing, part 1: optimizing the scanner. *V8* [online]. 2019 [cit. 2021-5-12]. Dostupné z: https://v8.dev/docs/wasm-compilation-pipeline
[24]: Intermediate representation. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-12]. Dostupné z: https://en.wikipedia.org/wiki/Intermediate_representation
[25]: CLARK, Lin. *What makes WebAssembly fast?* [online]. 2017 [cit. 2021-5-12]. Dostupné z: https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/
[26]: Abstract syntax tree. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-12]. Dostupné z: https://en.wikipedia.org/wiki/Abstract_syntax_tree
[27]: *WebAssembly compilation pipeline* [online]. [cit. 2021-5-12]. Dostupné z: https://v8.dev/docs/wasm-compilation-pipeline
[28]: SIMD. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-13]. Dostupné z: https://en.wikipedia.org/wiki/SIMD
[29]: Fast, parallel applications with WebAssembly SIMD. *V8* [online]. 2021 [cit. 2021-5-13]. Dostupné z: https://v8.dev/features/simd
[30]: YUAN, Ann a Marat DUKHAN. Supercharging the TensorFlow.js WebAssembly backend with SIMD and multi-threading. *Blog.tensorflow.org* [online]. 2020 [cit. 2021-5-13]. Dostupné z: https://blog.tensorflow.org/2020/09/supercharging-tensorflowjs-webassembly.html
[31]: DANILO, Alex. WebAssembly Threads ready to try in Chrome 70. *Google Developers* [online]. 2018 [cit. 2021-5-13]. Dostupné z: https://developers.google.com/web/updates/2018/10/wasm-threads
[32]: *WebAssembly threads – http 203. In: Youtube* [online]. 2020 [cit. 2021-5-11]. Dostupné z: https://Javascript.youtube.com/watch?v=x9RP-M6q2Mg
[33]: Introduction. *Rust and WebAssembly* [online]. [cit. 2021-5-19]. Dostupné z: https://rustwasm.github.io/docs/wasm-bindgen/
[34]: CLARK, Lin. WebAssembly Interface Types: Interoperate with All the Things! *Mozilla hacks* [online]. 2019 [cit. 2021-5-19]. Dostupné z: https://hacks.mozilla.org/2019/08/webassembly-interface-types/
[35]: Polyfill for "JS objects in wasm." *Rust and WebAssembly* [online]. [cit. 2021-5-19]. Dostupné z: https://rustwasm.github.io/wasm-bindgen/contributing/design/js-objects-in-rust.html
[36]: *Our Vision for Rust and WebAssembly* [online]. 2018 [cit. 2021-5-20]. Dostupné z: https://rustwasm.github.io/2018/06/25/vision-for-rust-and-wasm.html
[37]: *Debugging Rust-generated WebAssembly* [online]. 2019 [cit. 2021-5-20]. Dostupné z: https://rustwasm.github.io/docs/book/reference/debugging.htm
[38]: CLARK, Lin. *WebAssembly table imports… what are they? – Mozilla Hacks - the Web developer blog* [online]. [cit. 2021-10-12]. Dostupné z: https://hacks.mozilla.org/2017/07/webassembly-table-imports-what-are-they/
[39]: FITZGERALD, Nick. WebAssembly Reference Types in Wasmtime. *Bytecode Alliance* [online]. 2019 [cit. 2021-10-20]. Dostupné z: https://bytecodealliance.org/articles/reference-types-in-wasmtime

[40]: EBERHARDT, Colin. *What Is WebAssembly?* [online]. O'Reilly Media, 2019 [cit. 2021-10-21]. ISBN 9781492076896. Dostupné z: https://Javascript.oreilly.com/library/view/what-is-webassembly/9781492076902/
[41]: STEPANYAN, Ingvar. Using WebAssembly threads from C, C++ and Rust. *Web.dev* [online]. [cit. 2021-10-25]. Dostupné z: https://web.dev/webassembly-threads/
[42]: CLARK, Lin. Making WebAssembly even faster: Firefox's new streaming and tiering compiler. *Mozilla hacks* [online]. 2018 [cit. 2021-10-25]. Dostupné z: https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/
[43]: OSMANI, Addy a Mathias BYNENS. The cost of JavaScript in 2019. *V8 JavaScript engine* [online]. 2019 [cit. 2021-10-26]. Dostupné z: https://v8.dev/blog/cost-of-javascript-2019

Seznam Obrázků

Seznam Grafů

Seznam Grafů

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Tadeáš Polák**
Studium:	I1800449
Studijní program:	B6209 Systémové inženýrství a informatika
Studijní obor:	Informační management
Název bakalářské práce:	**Využití WebAssembly pro vývoje webových aplikací**
Název bakalářské práce AJ:	Use of WebAssembly for web application development

Cíl, metody, literatura, předpoklady:

Cíl práce

Analyzovat dopad vybraných způsobů využití WebAssembly na výkon a rychlost webových aplikací na straně klienta.

Osnova

1. Úvod
2. Cíl, metodika
3. Představení WebAssembly
4. WebAssembly v prohlížeči
5. Vývoj s WebAssembly
6. Popis aplikace
7. Výkon WebAssembly
8. Shrnutí výsledků
9. Závěr a doporučení

Bringing the web up to speed with WebAssembly. *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* [online]. 2017, **2017**, 185 - 200 [cit. 2021-4-30]. Dostupné z: doi: https://doi.org/10.1145/3062341.3062363

ROURKE, Mike. *Learn WebAssembly* [online]. Packt, 2018 [cit. 2021-4-30]. ISBN 9781788997379. Dostupné z: https://www.packtpub.com/product/learn-webassembly/9781788997379

AKINSHIN, Andrey. *Pro .NET benchmarking: the art of performance measurement*. Berkeley: APress, [2019]. For professionals by professionals. ISBN 978-1484249406.

WebAssembly Community Group. WebAssembly Specification. [online], 2021, 15 – 21. [cit. 2021-5-2]. Dostupné z: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

Garantující pracoviště:	Katedra informačních technologií, Fakulta informatiky a managementu
Vedoucí práce:	Mgr. Daniela Ponce, Ph.D.
Datum zadání závěrečné práce:	21.1.2020