Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**Покращення швидкодії обчислень у веббраузері за допомогою використання WebGPU**»

Виконав: студент 4-го року навчання,

Спеціальності

122 Комп’ютерні науки

Козодой Максим Олександрович

Керівник: Калітовський Б. В.

Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою _____
Секретар ЕК_____
“_____” _____ 2023 р.

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,

Жежерун О.П.

“___” _____ 2023 р.

ЗАВДАННЯ

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТЦІ

Козодоя Максима Олександровича

1. Тема роботи: Покращення швидкодії обчислень у веббраузері за допомогою використання WebGPU
   керівник роботи Калітовський Богдан Віталійович
   затверджені наказом вищого навчального закладу від «__»_____20__року №____
2. Строк подання студентом роботи _____
3. План роботи _____

1. **Календарний план виконання роботи**:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	12.10.2022	
2.	Пошук тематичної літератури	14.11.2022	
3.	Ознайомлення з літературою	17.12.2022	
4.	Створення плану роботи	01.02.2023	
5.	Написання теоретичної частини роботи	20.03.2023	
6.	Подання першої версії записки науковому керівнику	25.03.2023	
7.	Розробка застосунку для дослідження	15.04.2023	
8.	Описання практичної частини роботи	20.04.2023	
9.	Проведення дослідження	01.05.2023	
10.	Аналіз дослідження та висновки	05.05.2023	
11.	Перегляд змісту роботи керівником	10.05.2023	
12.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	13.05.2023	

Студент Козодой М. О.

Керівник Калітовський Б. В.

“____” _____

Зміст

АНОТАЦІЯ

У ході цієї кваліфікаційної роботи було досліджено технологію WebGPU. Були детально оглянуті й проаналізовані всі її можливості, основні концепції й описані особливості роботи з нею. Для демонстрації переваг WebGPU над WebGL, її прямим конкурентом, були розроблені демонстраційні застосунки, які показали, що WebGPU обходить WebGL у всіх головних критеріях, а саме: швидкодії, продуктивності та оптимізації. На останок були викладені роздуми щодо можливих майбутніх перспектив цієї технології у якості потенційного вебстандарту.

ВСТУП

У минулому можливості графічних процесорів були значно меншими, ніж зараз. По-перше, самі пристрої були набагато менш потужними. А по-друге, вони зазвичай обмежувалися невеликим набором фіксованих і спеціалізованих функцій, вбудованих в самі пристрої.

На противагу їм, сучасне графічне обладнання є набагато потужнішим і, що найголовніше, високопрограмованим. Останнє означає, що користувачам дозволяється виконувати довільні програми на графічному процесорі. Це можна використовувати для розробки широкого спектру програмного забезпечення, яке орієнтоване на роботу з графікою, а також для інших програм, які можуть виграти від високопаралельної природи графічних процесорів. Використання графічного процесора (англ. GPU) для будь-яких обчислень часто називають GPGPU (General-Purpose GPU), що перекладається як графічний процесор загального призначення.

Для керування графічним процесором команди повинні надсилатися з центрального процесора. І хоча низькорівневі деталі цього зв’язку можуть бути пропрієтарними, існують доступні стандартизовані прикладні програмні інтерфейси (англ. Application Programming Interface або API), які можуть бути реалізовані виробником обладнання у графічних драйверах, щоб дозволити програмістам взаємодіяти з різними пристроями уніфікованим способом.

На сьогоднішній день на ринку представлене велике різноманіття таких інтерфейсів, деякі з яких є орієнтованими виключно на конкретні операційні системи, як-от: DirectX від Microsoft для Windows, Metal від Apple для macOS та iOS. Окрім них існують й інші прикладні програмні інтерфейси, які не залежні від конкретної платформи, наприклад OpenGL і Vulkan.

У 2021 році група програмістів під назвою “GPU FOR THE WEB COMMUNITY GROUP” у складі W3C (World Wide Web Consortium) продемонструвала прототип свого нового продукту під назвою WebGPU. Це робоча назва потенційного вебстандарту та JavaScript API для відображення сучасної графіки і прискорення обчислень у браузері. Революційним у цій технології є те, що вона дозволяє повноцінно використовувати потужності графічного процесору користувача у веббраузері, чого раніше ніхто не міг навіть і уявити. Це дозволить перевернути індустрію вебтехнологій з ніг на голову, запропонувавши небачену раніше обчислювальну потужність і швидкодію для вебзастосунків. Також варто наголосити на тому, що ця технологія призначена для використання у сфері вебтехнологій, а отже її не потрібно буде адаптовувати під різні операційні системи, адже вона від самого початку є платформо-незалежною по своїй природі.

Об’єкт дослідження цієї кваліфікаційної роботи – методи покращення швидкодії обчислень у веббраузері. Предмет дослідження – технологія WebGPU. Отже, мета цієї роботи – дослідити WebGPU, описати й проаналізувати основні особливості цієї технології, її перспективи, переваги у порівнянні з конкурентами. Під час написання прикладних програм буде дана оцінка можливостей цієї технології, її зручності у застосуванні, а також викладені роздуми щодо можливих майбутніх перспектив

1 ПОНЯТТЯ ГРАФІЧНОГО ПРОЦЕСОРА

Графічний процесор (англ. Graphic Processing Unit, GPU) – це спеціалізований електронний пристрій, призначений для виконання завдань з графічного рендерингу. Особливості їхньої архітектури, що дозволяють їм ефективно виконувати великі паралельні обчислення, роблять їх більш ефективними, ніж центральні процесори загального призначення (CPU) для алгоритмів, які обробляють великі блоки даних паралельно. Вперше подібний пристрій був представлений у 1970 році на сімействі 8-бітних комп'ютерів Atari.

1.1 Основне призначення

Такі індустрії розваг, як кіно та відеоігри, стали головними рушійними силами розвитку технологій GPU. Графічний процесор швидко став одним із найважливіших типів обчислювальних пристроїв як для персонального, так і для комерційного використання. Будучи початково розробленими для паралельних обчислень, потужності графічних процесорів широко використовуються у багатьох різних типах програмних продуктів.

1.1.1 Рендеринг графіки

Одними із перших, хто відчув покращення від впровадження і розвитку графічних процесорів, стали художники, монтажери, графічні дизайнери та геймери. Останні в особливості, адже відеоігри стали більш обчислювально інтенсивними, з гіперреалістичною графікою та великими, складними ігровими світами. Завдяки передовим технологіям відображення, таким як екрани 4K і висока

частота оновлення, разом із розвитком ігор у віртуальній реальності вимоги до обробки графіки швидко зростають. Також до таких процесів відноситься створення графіки, яка використовується у фільмах, телевізійних шоу, рекламах та картинах цифрових художників.

Сучасні графічні карти підтримують програмне забезпечення, яке використовується для кодування відео (процесу, за допомогою якого відеодані редагуються та форматуються перед відтворенням). Кодування відео – це ресурсомісткий процес, який може зайняти надзвичайно багато часу, якщо використовується лише центральний процесор. За допомогою графічного процесора кодування відео можна виконуватися швидко, не перевантажуючи системні ресурси.

1.1.2 Обчислення загального призначення на графічних процесорах (GPGPU)

Два десятиріччя тому графічні процесори використовувалися в основному для прискорення рендерингу графіки, особливо тривимірних графічних програм у реальному часі, таких як відеоігри. Однак із початком XXI століття вчені у галузі комп’ютерних наук зрозуміли, що графічні процесори мають потенціал для вирішення деяких із найскладніших обчислювальних задач. Це усвідомлення дало початок уже згаданій раніше методиці обчислення загального призначення на графічних процесорах (англ. General-purpose computing on graphics processing units, GPGPU). Сучасні графічні процесори є більш програмованими, ніж будь-коли раніше, що надає їм гнучкість для прискорення широкого спектру програм, які виходять далеко за рамки традиційного рендерингу графіки. Деякі з найбільш цікавих сфер застосувань включають в себе розробку штучного інтелекту та машинного навчання. Оскільки графічні процесори мають надзвичайні потужності,

вони можуть забезпечити неймовірне прискорення робочих обчислень, які використовують переваги високопаралельної природи графічних процесорів, наприклад, для розпізнавання зображень. Багато сучасних технологій глибокого навчання покладаються на графічні процесори, які працюють в парі із центральними процесорами.

Популярність обчислень загального призначення на графічних процесорах значно зросла із появою NVIDIA Compute Unified Device Architecture (CUDA). CUDA – це програмно-апаратна архітектура паралельних обчислень створена компанією NVIDIA для їхніх власних графічних процесорів. CUDA дозволяє використовувати графічні процесори NVIDIA для обчислень загального призначення, надаючи програмістам прямий доступ до віртуального набору інструкцій графічного процесора та паралельних обчислювальних елементів. Відтоді графічні процесори стали однією з домінуючих технологій у сфері високопродуктивних обчислень (англ. High-performance computing, HPC) для швидкої обробки надзвичайно великих обсягів даних.

2 ПОНЯТТЯ ШЕЙДЕРІВ

Програми, створені для виконання на графічному процесорі, називаються шейдерами (англ. shaders). Зазвичай вони написані на спеціалізованій мові програмування шейдерів. Кожен програмний інтерфейс для взаємодії з графічним процесором надає власні мови шейдерів, що і зображено в таблиці, наведеній нижче:

API	Мова шейдерів	Платформа
DirectX	HLSL	Windows
Metal	MSL	macOS, iOS
Vulkan	SPIR-V	Windows, Linux
OpenGL	GLSL	Windows, Linux

Таблиця 1. API та їхні мови шейдерів

Термін шейдер, який походить від англійського слова “shade”, що у перекладі означає відтінок або тінь, може збивати з пантелику, адже за допомогою нього можна робити значно більше, ніж просто реалізовувати процес “затінення”. Але раніше, у 80-х роках XX століття , цей термін був доречним: у ті часи шейдери були невеликим фрагментом коду, який запускався на графічному процесорі для визначення кольору кожного пікселя, щоб “відтінювати” різноманітні об’єкти, досягаючи цим самим ілюзії ефекту освітлення й відображення тіней. Загалом існує дві категорії шейдерів – графічні та обчислювальні. Графічні шейдери використовуються для виконання певних завдань у процесі візуалізації графіки, наприклад визначення кольору кожного пікселя. Обчислювальні ж шейдери можуть використовуватися для виконання довільних обчислень.

3 WEBGL ЯК ПОПЕРЕДНИК WEBGPU

WebGL (Web Graphics Library) був представлений у 2011 році і до цього моменту залишався єдиним низькорівневим програмним інтерфейсом для доступу до графічного процесору з веббраузеру.

WebGL – це фактично OpenGL з деякими тонкими обгортками та невеликими надбудовами, які роблять його вебсумісним. Обидва цих API розробляються та підтримуються організацією Khronos Group.

Сам OpenGL є ще більш давнім і за сучасними стандартами є відчутно застарілим. Його архітектурний дизайн зосереджений навколо внутрішнього глобального об’єкта стану. Раніше, враховуючи менш потужні графічні процесори та повільнішу швидкість мережі, це рішення, можливо, і мало сенс, оскільки дозволяло мінімізувати кількість даних, які необхідно було передавати до графічного процесора та надсилати з нього для будь-якого виклику. Однак це також несло з собою істотно підвищену складність розробки та зневадження застосунків.

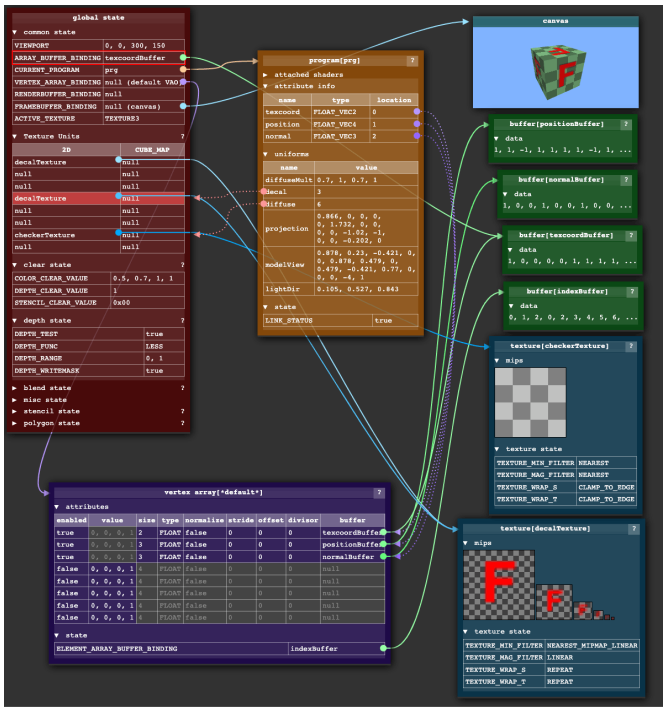

Рисунок 1 Візуалізація роботи у WebGL із внутрішнім глобальним об’єктом стану

Внутрішній глобальний об’єкт стану – це, в основному, набір вказівників (англ. pointers). Виклики API, які здійснює користувач, можуть впливати як на об’єкти, на які спрямовані ці вказівники, так і на сам об’єкт стану. Як наслідок, порядок викликів API неймовірно важливий, що ускладнює створення власних абстракцій та бібліотек. Ба більше, це вимагає бути більш прискіпливим та уважним, пильно контролювати усі ресурси й структури, задіяні в рендерингу, не забуваючи очищати зайві вказівники та об’єкти, на які вони були спрямовані. Відсутність інформування про помилки ще більше ускладнює цей процес, адже замість повідомлення з текстом помилки та причиною її виникнення користувач отримає чорний екран.

Також варто зазначити, що і для обчислень загального призначення на графічних процесорах WebGL не є хорошим рішенням. Для того, щоб обробити довільні дані на графічному процесорі за допомогою WebGL, їх спочатку необхідно закодувати як текстуру, декодувати в шейдері, виконати бажане обчислення, а потім ще повторно закодувати результат обчислень як текстуру.

4 ОГЛЯД WEBGPU

Протягом останніх десяти років за межами вебіндустрії з'явилося та продовжує розвиватися нове покоління програмних інтерфейсів для керування графічними процесорами, які надають ще більш низькорівневі можливості для управління цілими відеокартами ніж коли-небудь раніше. Ці нові API враховують сучасні особливості як самих пристроїв, так і способів їхнього використання, яких просто не існувало під час розробки OpenGL. Так, наприклад, у сучасному світі графічні процесори почали використовуватися майже скрізь. Навіть мобільні пристрої, будь то смартфони чи планшети, обзавелися потужними вбудованими графічними процесорами. У результаті програмування, пов'язане із відображенням сучасної графіки, наприклад, рендеринг 3-D об'єктів із використанням технології трасування променів (англ. ray tracing), чи обчислень загального призначення на графічних процесорах стає все більш поширеним і популярним. Також, враховуючи, що більшість пристроїв сьогодні мають багатоядерні процесори, цікавою і перспективною сферою стала оптимізація взаємодії з графічним процесором із кількох потоків.

Найпопулярнішими і найкращими API для взаємодії з графічним процесором нового покоління є Vulkan від Khronos Group, Metal від Apple та DirectX 12 від Microsoft. Саме для того, щоб перенести їх революційні здобутки у царину вебзастосунків, і був створений WebGPU.

4.1 Основні поняття

При створенні WebGPU розробники обрали новий підхід: у той час як WebGL є лише тонкою обгорткою навколо OpenGL, у WebGPU використовуються власні абстракції, а ще він не є прямою адаптацією жодного зі створених раніше нативних

інтерфейсів. Частково це пов’язано з тим, що жоден з існуючих API не є доступним на всіх платформах, а також тому, що багато концепцій, наприклад, надзвичайно низькорівневе керування пам’яттю, не критично важливі для вебінтерфейсу. Натомість WebGPU розроблено таким чином, щоб бути гнучким, зручним, кросплатформним і ввібрати в себе лише найкраще з нативних графічних API, абстрагуючись від їхніх особливостей. Завдяки тому, що WebGPU розробляється організацією W3C, він стандартизований серед усіх найпопулярніших веббраузерів.

4.1.1 Контекст

Оскільки WebGPU все ще залишається експериментальною технологією, для роботи з нею необхідно спеціальним чином налаштувати веббраузер. Для демонстрації прикладів роботи з WebGPU у ході цієї кваліфікаційної роботи використовувався браузер Google Chrome Canary з увімкненими прапорцями #enable-unsafe-webgpu та #enable-webgpu-developer-features, які можна знайти на вкладці chrome://flags/.

Враховуючи написане вище, у будь-якому застосунку, де використовується WebGPU, перш за все обов’язковою є перевірка, чи підтримує веббраузер користувача цю технологію. Це може мати наступний вигляд:

```
const entry = navigator.gpu;
if (!entry) {
  throw Error(
    "WebGPU is not supported. Please enable it in about:flags in Chrome or in about:config in Firefox."
  );
}
```

Рисунок 2 Перевірка підтримки WebGPU браузером користувача

Початкове налаштування середовища для роботи з WebGPU загалом подібне до WebGL: так наприклад для роботи з графікою необхідною умовою є наявність

на сторінці елемента HTMLCanvasElement, в якому вона і буде відображатися. Перш за все необхідно створити цей елемент та сконфігурувати контекст.

```
const canvas = document.querySelector("#gpuCanvas");
const context = null;
context = canvas.getContext("webgpu");
const canvasConfig = {
  device: this.device,
  format: navigator.gpu.getPreferredCanvasFormat(),
  alphaMode: "opaque",
};
context.configure(canvasConfig);
```

Рисунок 3 Налаштування HTMLCanvasElement

4.1.2 Адаптери та логічні пристрої

Одними з найголовніших абстракцій, які використовуються у WebGPU, є адаптери та логічні пристрої.

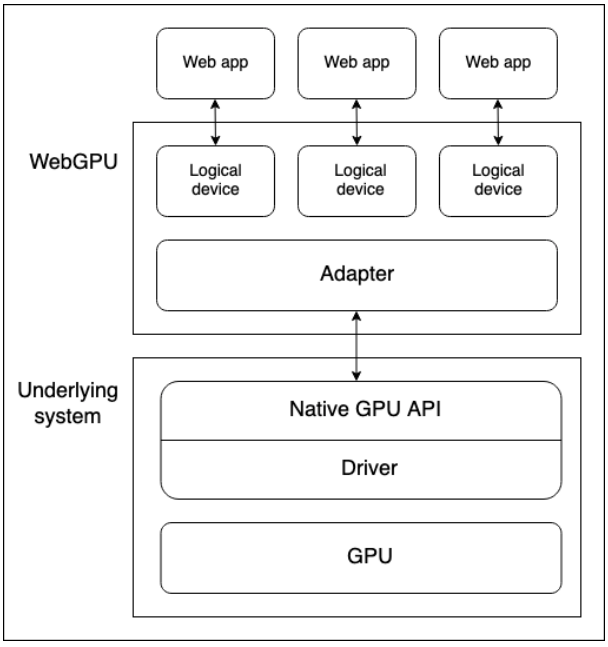

Рисунок 4 Рівні абстракції від фізичних графічних процесорів до логічних пристроїв.

Фізичні пристрої — це самі графічні процесори, які поділяють на вбудовані та дискретні графічні процесори. Зазвичай будь-який пристрій має один графічний

процесор, але також нерідко буває, що він може мати два або навіть більше. Наприклад, на пристрої, на якому запускалися застосунки, створені в рамках цієї кваліфікаційної роботи, в наявності були малопотужний вбудований графічний процесор і високопродуктивний дискретний графічний процесор, між якими операційна система перемикається за потребою.

Драйвер, створений виробником графічного процесора, дає операційній системі можливість використовувати графічний процесор у спосіб, який розуміє та очікує ця ОС. Операційна система, у свою чергу, зможе надавати доступ до GPU іншим застосункам, використовуючи графічні API, які зроблені конкретно під неї, наприклад DirectX 12 або Metal.

Графічний процесор є спільним ресурсом у системі. Він не тільки використовується багатьма програмами одночасно, але також контролює те, що користувач бачить на екрані.

Адаптери, у свою чергу, є перехідним рівнем від нативного графічного API операційної системи до WebGPU. Вони описують фізичні властивості певного графічного процесора, наприклад його ім’я, характеристики тощо. Щоб отримати доступ до адаптера, необхідно викликати navigator.gpu.requestAdapter(). На момент написання роботи функція requestAdapter() має дуже небагато доступних параметрів, які дозволяють обрати режим енергозбереження або режим високої продуктивності. Приклад виклику адаптера:

```
let adapter = null;
adapter = await navigator.gpu.requestAdapter();
if (!adapter) {
  throw Error("Couldn't request WebGPU adapter.");
}
```

Рисунок 5 Отримання адаптера

Оскільки веббраузер є єдиною програмою на рівні ОС, яка може запускати кілька вебзастосунків, виникає потреба в мультиплексуванні (передачі даних з багатьох каналів через один), щоб кожен вебдодаток відчував себе так, ніби він повністю контролює графічний процесор і може повноцінно й, що найголовніше, ізольовано від інших використовувати його ресурси. Це моделюється у WebGPU за допомогою концепції логічних пристроїв. Логічні пристрої необхідні для того, щоб отримувати доступ до ядра прикладного програмного інтерфейсу WebGPU і створювати потрібні структури даних. Приклад виклику логічного пристрою з адаптеру, який успішно повернула функція requestAdapter():

```
let device = null;
device = await adapter.requestDevice();
if (!device) {
  throw Error("Couldn’t request WebGPU logical device.");
}
```

Рисунок 6 Отримання логічного пристрою

Важливо зазначити, якщо у функцію requestDevice() не передавати жодних параметрів, то вона поверне пристрій, який не обов’язково повністю відповідає можливостям фізичного пристрою, а радше те, що команда розробників WebGPU вважає найменшим спільним знаменником більшості GPU (детальніше про це описано у стандарті WebGPU на відповідному ресурсі в інтернеті). Наприклад, незважаючи на те, що графічний процесор, на якому запускалися застосунки, створені в ході цієї роботи, легко обробляє буфери даних розміром до 8 ГБ, повернутий функцією requestDevice() пристрій допускатиме лише буфери даних розміром до 1 ГБ і відхилятиме будь-які буфери даних, які є більшими за розміром. На перший погляд це може здатися непотрібним і зайвим обмеженням, однак, насправді, воно є досить корисним. Якщо застосунок із застосуванням технології WebGPU за замовчуванням буде використовувати саме таку мінімальну за характеристиками конфігурацію пристрою, то він гарантовано працюватиме на

переважній більшості користувацьких пристроїв. Якщо ж все-таки необхідно використовувати максимальну потужність GPU, то існує можливість перевірити реальні обмеження фізичного графічного процесора за допомогою adapter.limits і запросити логічний пристрій із підвищеними характеристиками, передавши об’єкт параметрів у requestDevice().

4.1.3 Шейдери у WebGPU

У WebGPU існує 3 основних типи шейдерів: вершинний (англ. vertex), фрагментний (англ. fragment) і обчислювальний (англ. compute).

У даному контексті вершина — це точка в просторі. Ці вершини об’єднуються в групи по дві вершини для формування ліній та/або три вершини для формування трикутників. Сучасні засоби рендерингу використовують трикутники для створення будь-яких форм, від простих (наприклад, кубів) до складних (таких як люди). Ці трикутники зберігаються як вершини, які є точками, що утворюють кути трикутників. Вершинний шейдер використовується для того, щоб маніпулювати вершинами, трансформуючи об’єкти найрізноманітнішим чином. Потім вершини перетворюються на фрагменти. Кожен піксель у фінальному зображенні отримує принаймні один фрагмент. Обчислювальний шейдер використовується для обчислень загального призначення на графічному процесорі.

В загальному процес відображення графіки виглядає наступним чином: програма завантажує буфер даних у графічний процесор і вказує йому, яким чином інтерпретувати ці дані як серію трикутників. Кожна вершина займає частину цього буфера даних, описуючи позицію цієї вершини в просторі, а також допоміжні дані, такі як колір, ідентифікатори текстур, нормалі та інші параметри. Кожна вершина в списку обробляється графічним процесором, запускаючи вершинний шейдер для кожної з них, який застосує переміщення, поворот або зміну перспективи. Після

цього графічний процесор растеризує трикутники, тобто він визначає, які пікселі покриває кожен трикутник на екрані. І, нарешті, кожен піксель обробляється фрагментним шейдером, який має доступ до координат пікселя, а також до допоміжних даних, щоб, скажімо, вирішити, якого кольору цей піксель має бути.

У WebGPU є своя мова шейдерів – WebGPU Shading Language (WGSL). WebGPU Shading Language подібна до інших мов, таких як Rust, Metal Shading Language (MSL) і DirectX High Level Shading Language, з декораторами стилю як у JavaScript, наприклад “@location(0)”, форматуванням коду в стилі Rust із функціями/членами із “snake_case” записом, а структурами “CamelCase” записом, і оголошенням функцій на зразок “fn my_func() -> i32”. Веббраузер компілює WGSL код в той, який очікує система. Тобто це буде HLSL для DirectX 12, MSL для Metal або SPIR-V для Vulkan.

Приклад вершинного шейдеру:

```
struct VSOut {
  @builtin(position) nds_position: vec4<f32>,
  @location(0) color: vec3<f32>,
};

@vertex
fn main(@location(0) in_pos: vec3<f32>,
        @location(1) in_color: vec3<f32>) -> VSOut {
  var vs_out: VSOut;
  vs_out.nds_position = vec4<f32>(in_pos, 1.0);
  vs_out.color = inColor;
  return vsOut;
}
```

Рисунок 7 Вершинний шейдер

Приклад фрагментного шейдеру:

```
@fragment
fn main(@location(0) in_color: vec3<f32>) -> @location(0) vec4<f32> {
    return vec4<f32>(in_color, 1.0);
}
```

Рисунок 8 Фрагментний шейдер

Хорошою практикою вважається запаковувати шейдери в модулі для зручнішого використання. Приклад створення шейдерних модулів:

```
import vertShaderCode from './shaders/triangle.vert.wgsl';
import fragShaderCode from './shaders/triangle.frag.wgsl';

let vertModule = null;
let fragModule = null;

const vsmDesc = { code: vertShaderCode };
vertModule = device.createShaderModule(vsmDesc);

const fsmDesc = { code: fragShaderCode };
fragModule = device.createShaderModule(fsmDesc);
```

Рисунок 9 Шейдерні модулі

4.1.4 Конвеєр

Конвеєр — це логічна структура, що містить програмовані етапи, які послідовно відпрацьовують для виконання роботи програми. Наприклад, система передачі даних у вершинний шейдер, потім у фрагментний шейдер і подальше виведення оброблених даних безпосередньо на екран називається конвеєром, і у WebGPU необхідно явно визначити свій конвеєр. Наразі WebGPU дозволяє створювати два типи конвеєрів: конвеєр візуалізації (англ. Render Pipeline) та конвеєр обчислень (англ. Compute Pipeline). Як випливає з назви, конвеєр візуалізації щось рендерить, тобто створює зображення. Це зображення не обов’язково повинно бути на екрані, воно може зберігатися в спеціальному місці у пам’яті, яке називається буфером кадрів (англ. Framebuffer). Конвеєр обчислень є більш загальним, оскільки він повертає буфер, який може містити будь-які дані. Обчислювальний конвеєр був побудований як узагальнення над спеціально створеним конвеєром візуалізації. У майбутньому до WebGPU буде додано більше типів конвеєрів, можливо навіть конвеєр трасування променів (Raytracing Pipeline). У WebGPU конвеєр складається з одного (або кількох) програмованих етапів, де

кожен етап визначається шейдером і точкою входу. Обчислювальний конвеєр має один етап з обчислювальним шейдером, тоді як конвеєр візуалізації має етап вершинного та фрагментного шейдерів.

Приклад обчислювального конвеєру:

```
const module = device.createShaderModule({
    code: `
      @compute @workgroup_size(64)
      fn main() {
      }
    `,
});

const pipeline = device.createComputePipeline({
    compute: {
        module,
        entryPoint: "main",
    },
});
```

Рисунок 10 Обчислювальний конвеєр

У цьому прикладі шейдерний модуль містить функцію під назвою main, і вона ж позначається тут як точку входу за допомогою параметру entryPoint у конвеєрі.

4.1.5 Буфер

Буфером називають блок пам'яті, який призначений для зберігання даних, які братимуть участь в операціях на GPU. Буфери зазвичай використовуються для зберігання структур чи масивів, але вони можуть містити в собі і складніші структури даних, наприклад графові (такі як дерева), однак, лише за умови, що всі вузли зберігаються разом і не посилаються ні на що поза буфером.

Найчастіше у буфері зберігаються такі масиви даних, як-от координати елемента у просторі, його колір, індекс тощо. Для рендерингу трикутників за допомогою конвеєру візуалізації знадобиться один або більше буферів для зберігання даних, які стосуються вершин. Їх ще називають вершинні буферні об’єкти (англ. Vertex Buffer Objects або VBOs). Також необхідним є один буфер з

індексами, які відповідатимуть кожній вершині трикутника, відомий як індексний буферний об’єкт (англ. Index Buffer Object або IBO).

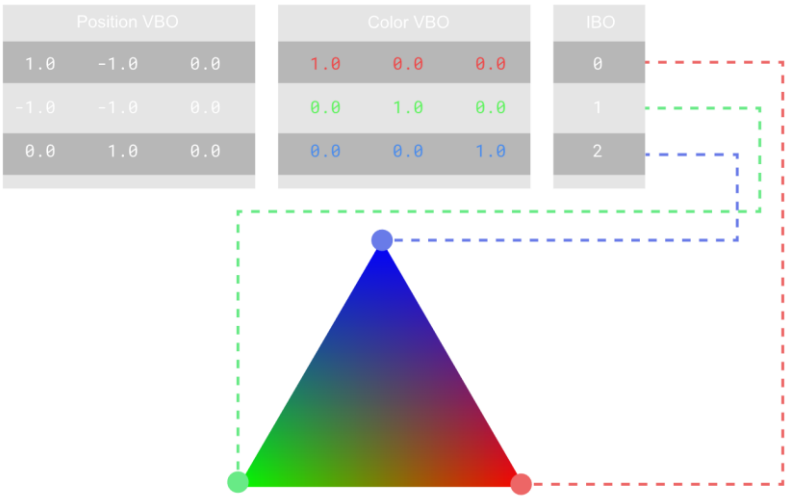

Рисунок 11 Буфери, які необхідні для візуалізації кольорового трикутника

4.1.6 Паралелізм

Як відомо, графічні процесори оптимізовані для шаленої пропускної здатності, жертвуючи при цьому часом затримки. Графічні процесори мають велику кількість ядер, які дозволяють виконувати безліч процесів паралельно. Інтегровані графічні процесори зазвичай мають від 170 до 700 ядер, у той же час як дискретні легко можуть мати понад 1500 ядер. Однак ці ядра не такі незалежні, як, наприклад, у багатоядерного центрального процесора. По-перше, ядра GPU згруповані ієрархічно. Варто зазначити, що термінологія для різних рівнів ієрархії не є узгодженою між різними постачальниками графічних процесорів та API. Незважаючи на те, що точна архітектура графічних процесорів є приватною інформацією, захищеною NDA, у компанії Intel є хороша документація, яка надає загальний огляд їхньої архітектури, і можна з упевненістю припустити, що інші графічні процесори працюють принаймні схожим чином.

У випадку Intel, найнижчим рівнем в ієрархії є виконавчий блок (англ. Execution Unit або EU), який має кілька ядер SIMT (Single instruction, multiple threads). Це означає, що він має ядра, які працюють узгоджено й завжди виконують однакові інструкції. Проте кожне ядро має власний набір регістрів і покажчик стека. Таким чином, хоча їм доводиться виконувати ту саму операцію, вони можуть виконувати її з різними даними. Їх узгодженість є причиною, чому експерти з продуктивності графічних процесорів уникають розгалужень (як-от if/else або циклів): коли виконавчий блок стикається з if/else, усі ядра повинні виконувати обидві гілки, якщо тільки всі вони не підуть по одній і тій же гілці. Те ж саме стосується і циклів. Якщо одне ядро завершує свій цикл раніше за інші, йому доведеться робити вигляд, що воно виконує тіло циклу, доки всі ядра не завершать роботу. Кожному ядру можна наказати ігнорувати інструкції, які йому надходять, але це, очевидно, марнує дорогоцінний час, який можна витратити на обчислення.

Незважаючи на частоту ядра, отримання даних з пам'яті (або пікселів з текстур) все ще займає відносно багато часу. Цей вільний час очікування не проходить дарма, а витрачається на обчислення: для цього кожний виконавчий блок сильно перевантажений (англ. oversubscribed) роботою. Кожного разу, коли у виконавчого блоку з’являється вікно бездіяльності (наприклад, при очікуванні надходження якихось даних з пам’яті), він перемикається на інший робочий процес і повертається до попереднього лише тоді, коли новий робочий процес потребує якогось очікування. Саме таким чином графічні процесори оптимізують пропускну здатність, жертвуючи затримкою: окремі робочі процеси можуть зайняти більше часу, оскільки перемикання на інший робочий процес може призупинити виконання на довше, ніж очікувалося, але при цьому це компенсується більшою загальною кількістю оброблюваних процесів у певний період часу, що у свою чергу призводить до великої пропускної здатності. Графічний процесор завжди

намагається по максимуму завантажувати чергу виконання робочими процесами, які надсилаються до виконавчих блоків, щоб вони постійно були зайняті.

Однак виконавчі блоки є лише найнижчим рівнем в ієрархії. Кілька таких блоків групуються в те, що Intel називає підсегмент (англ. SubSlice). Усі виконавчі блоки у підсегменті мають доступ до невеликої кількості спільної локальної пам’яті (англ. Shared Local Memory або SLM). Якщо програма, яку потрібно запустити, має будь-які команди синхронізації, її слід виконувати в межах того самого підсегменту, оскільки лише вони мають спільну пам’ять для синхронізації.

На найвищому рівні в ієрархії знаходяться сегменти (англ. Slice), які складаються з декількох згрупованих підсегментів. Саме сегменти є основними елементами у структурі графічного процесора.

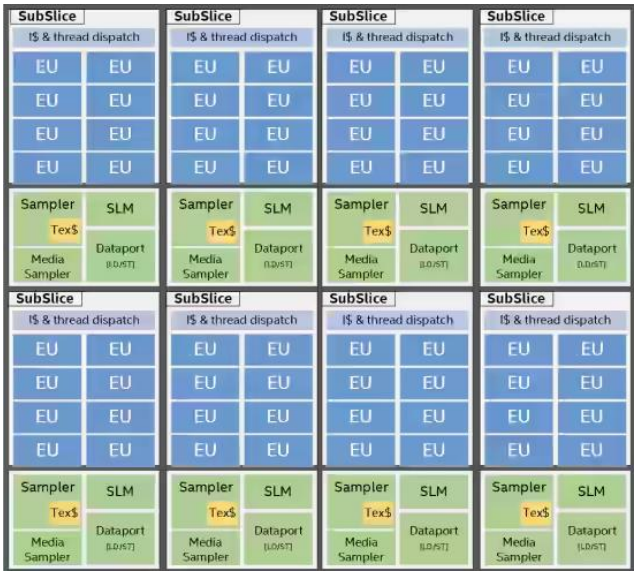

Рисунок 12 Архітектура чіпа Intel Iris Xe Graphics. Виконавчий блок має 7 SIMT ядер. Підсегменти складаються із 8 виконавчих блоків. 8 підсегментів утворюють сегмент.

Найменування структурних одиниць, використаних у даній кваліфікаційній роботі щодо тих чи інших елементів графічного процесора, були взяті із офіційної документації від компанії Intel, але цілком ймовірно, що інші виробники GPU

використовують інші назви, хоча й загальна архітектура в кожному графічному процесорі є подібною.

Щоб повністю використати переваги цієї архітектури, програми мають бути написані таким чином, щоб користуватися усіма її можливостями. Як наслідок, графічні API надають так звану модель потоків (англ. threading model), яка зручним чином дозволяє працювати з GPU. У WebGPU важливим примітивом, пов’язаним з цим, є робоча група (англ. workgroup).

4.1.7 Команди

Після написання шейдерів і налаштування конвеєрів все, що залишається зробити – це фактично звернутися до графічного процесора і наказати йому виконати усе написане. Оскільки графічний процесор може бути у складі окремої відеокарти з власним чіпом пам’яті, змога керувати ним реалізовується за допомогою так званого командного буфера (англ. Command Buffer) або черги команд (англ. Command Queue). Черга команд — це частина пам’яті, яка містить закодовані команди для виконання графічним процесором. Кодування є дуже специфічним і різниться в залежності від виробника графічного процесора, а забезпечується драйвером. WebGPU надає CommandEncoder для використання цих команд.

```
let commandEncoder = null;
let passEncoder = null;

commandEncoder = device.createCommandEncoder();
passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(pipeline);
passEncoder.dispatchWorkgroups(1);
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

Рисунок 13 Приклад роботи з командами

CommandEncoder має кілька методів, які дозволяють копіювати дані з одного буфера GPU в інший і маніпулювати текстурами. Він також дозволяє створити PassEncoder, який налаштовує конвеєри та запускає їх. У лістингу вище використовується обчислювальний конвеєр, тому необхідно створити саме ComputePassEncoder, налаштувати його на попередньо оголошений конвеєр та, нарешті, викликати dispatchWorkgroups(w_x, w_y, w_z), щоб повідомити графічному процесору, скільки робочих груп створити в кожному вимірі. Робочі групи (англ. workgroups) – це окремі порції від усіх робочих процесів, які розбиваються із загальної сукупності усіх робочих елементів, яку ще називають робочим навантаженням. PassEncoder, до речі, є абстракцією WebGPU, яка створювалася задля того, щоб уникнути внутрішнього глобального об’єкта стану, який присутній у WebGL. Усі дані, необхідні для роботи конвеєра GPU, явно передаються через PassEncoder.

Командний буфер також є гачком (англ. hook) для драйвера або операційної системи, щоб дозволити декільком програмам використовувати графічний процесор, не заважаючи при цьому одна одній. Коли команди ставляться у чергу, елементи, що знаходяться на нижчих рівнях абстракції, поставлять у чергу додаткові команди для збереження попереднього стану програми, щоб відновити його пізніше у випадку переривання. Саме таким чином і створюється відчуття, що ніхто інший не використовує графічний процесор у даний момент.

5 РОЗРОБКА І ОГЛЯД ДЕМОНСТРАЦІЙНИХ ЗАСТОСУНКІВ

У наступних розділах будуть оглянуті застосунки, які створювалися із використанням технології WebGPU.

5.1 Множення матриць

Для демонстрації роботи з обчислювальним конвеєром та обчислювальними шейдерами був створений застосунок для множення двох матриць. Деталі та особливості реалізації будуть викладені нижче. План етапів розробки застосунку виглядав наступним чином:

1. Створити три буфери: два для матриць, які безпосередньо приймають участь у множенні, та один для результуючої матриці.
2. Описати вхідні та вихідні дані для обчислювального шейдера.
3. Скомпілювати код обчислювального шейдера.
4. Налаштувати обчислювальний конвеєр.
5. Надіслати необхідні команди для виконання бажаних операцій на графічному процесорі.
6. Зчитати дані результуючої матриці з буфера.

5.1.1 Початкове налаштування

Перш за все необхідно доступитися до адаптера та отримати логічний пристрій:

```
const adapter = await navigator.gpu.requestAdapter();
if (!adapter) {
    console.log("Failed to get GPU adapter.");
    return;
}
const device = await adapter.requestDevice();
```

Рисунок 14 Множення матриць. Початкове налаштування

5.1.2 Робота з буферами

У попередніх розділах цієї кваліфікаційної роботи йшлося про використання буферів для обміну даними між програмою і графічним процесором, зараз же це буде продемонстровано на практиці.

У наведеному нижче прикладі показано, як записати чотири байти в буферну пам’ять, доступну з GPU. У функцію device.createBuffer() для створення буфера передається його бажаний розмір та тип використання. Незважаючи на те, що прапорець GPUBufferUsage.MAP_WRITE, який вказує тип використання, не потрібен для цього конкретного виклику, не буде зайвим показати у явний спосіб, що даний буфер створюється для запису даних в нього.

```
// Створення відображеного буфера
const gpuBuffer = device.createBuffer({
    mappedAtCreation: true,
    size: 4,
    usage: GPUBufferUsage.MAP_WRITE
});
// Отримання доступу до буфера
const arrayBuffer = gpuBuffer.getMappedRange();

// Запис байтів у буфер
new Uint8Array(arrayBuffer).set([0, 1, 2, 3]);
```

Рисунок 15 Множення матриць. Приклад створення відображеного буфера

На цьому етапі цей буфер є відображеним (англ. mapped), тобто він належить центральному процесору і доступний для читання/запису з JavaScript. Щоб графічний процесор міг отримати до нього доступ, буфер потрібно перетворити на

невідображений (англ. unmapped) за допомогою виклику функції gpuBuffer.unmap(). Концепція відображеного/невідображеного буфера потрібна, щоб запобігти ситуаціям, коли GPU і CPU намагаються отримати доступ до нього одночасно.

Важливою складовою роботи з буферами є можливість передавати дані з одного буфера в інший. У випадку, коли у програмі на одному з етапів необхідно буде скопіювати дані, при створенні буфера, з якого будуть братися дані, необхідно додатково вказати прапорець GPUBufferUsage.COPY_SRC, а при створенні буфера, в який дані будуть копіюватися – GPUBufferUsage.COPY_DST та GPUBufferUsage.MAP_READ.

```
// Створення відображеного буфера для запису та копіювання даних
const gpuWriteBuffer = device.createBuffer({
    mappedAtCreation: true,
    size: 4,
    usage: GPUBufferUsage.MAP_WRITE | GPUBufferUsage.COPY_SRC
});
// Отримання доступу до буфера
const arrayBuffer = gpuWriteBuffer.getMappedRange();

// Запис байтів у буфер
new Uint8Array(arrayBuffer).set([0, 1, 2, 3]);

// Перетворення буфера на невідображений
gpuWriteBuffer.unmap();

// Створення невідображеного буфера для читання
const gpuReadBuffer = device.createBuffer({
    size: 4,
    usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.MAP_READ
});
```

Рисунок 16 Множення матриць. Приклад створення різних буферів

Варто пам’ятати, що усі команди графічного процесора виконуються асинхронно. У WebGPU шифрувальник команд, який повертає функція createCommandEncoder(), являє собою JavaScript об’єкт, що створює набір буферизованих команд (вони зберігатимуться у командному буфері), які в певний

момент будуть надіслані на виконання до GPU. Методи ж класу GPUBuffer, який необхідний для створення звичайних буферів, є небуферизованими, тобто вони відпрацьовують атомарно відразу у момент їх виклику.

Після створення шифрувальника команд, необхідно викликати функцію copyBufferToBuffer(), щоб додати необхідну команду для копіювання буфера до черги команд для подальшого виконання. Нарешті, щоб завершити кодування команд, треба скористатися методом шифрувальника finish(). Після цього слід надіслати їх до черги для виклику команд на графічному процесорі. Зробити це можна за допомогою функції submit(), передавши в неї бажані команди в якості аргументів. Коли вона надішле команди на GPU, вони будуть почергово виконані у тому порядку, в якому вони зберігалися у масиві команд.

```
// Шифрування команди для копіювання даних з одного буфера в інший
const copyEncoder = device.createCommandEncoder();
copyEncoder.copyBufferToBuffer(
    gpuWriteBuffer /* первинний буфер */,
    0 /* вихідний зсув */,
    gpuReadBuffer /* буфер, в який будуть скопійовані дані */,
    0 /* зсув у об'єкті призначення */,
    4 /* розмір */
);

// Надсилання команди на копіювання
const copyCommands = copyEncoder.finish();
device.queue.submit([copyCommands]);
```

Рисунок 17 Множення матриць. Приклад копіювання буферів

Щоб зчитати дані з буфера, у який вони були скопійовані, необхідно скористатися методом mapAsync(), передавши в нього прапорець GPUMapMode.READ. Це поверне об'єкт типу Promise, який буде вирішено (англ. resolve), коли буфер стане відображеним. Потім треба викликати getMappedRange(), щоб отримати відображений діапазон, який міститиме скопійовані дані після відпрацювання відповідних команд з черги у графічному процесорі.

```
// Зчитування скопійованих даних з буфера
await gpuReadBuffer.mapAsync(GPUMapMode.READ);
const copyArrayBuffer = gpuReadBuffer.getMappedRange();
// Вивід отриманих даних у консоль
console.log(new Uint8Array(copyArrayBuffer));
```

Рисунок 18 Множення матриць. Приклад зчитування даних із буфера

Отож, працюючи із буферами у WebGPU, варто пам’ятати наступне:

1. Буфери, які надсилаються до графічного процесора, мають бути невідображеними, щоб він міг із ними працювати.
2. Тільки відображені буфери можуть використовуватися для читання/запису із JavaScript програми.
3. Для перетворення невідображеного буфера у відображений слід використовувати функцію mapAsync(), щоб досягти зворотного ефекту – метод unmap().

У WebGPU саме буферами є входи (англ. inputs) та виходи (англ. outputs) шейдерів.

Повернемося до плану з розробки застосунку. Матриці у ньому будуть представлені у вигляді списку чисел, де перший елемент – це число рядків, другий – число стовпчиків, а решта – фактичні значення комірок матриці. Тепер створимо необхідні буфери, використавши код із попередніх прикладів і замінивши деякі його частини:

```
// Перша матриця
const firstMatrix = new Float32Array([
    5 /* рядки */, 12 /* стовпчики */,
    8, 2, 7, 4, 5, 0, 5, 5, 3, 7, 2, 4,
    3, 0, 5, 6, 8, 6, 4, 3, 3, 9, 6, 4,
    4, 1, 9, 8, 7, 6, 3, 8, 1, 2, 2, 7,
    7, 7, 1, 9, 7, 9, 0, 1, 1, 0, 2, 8,
    8, 3, 0, 0, 4, 0, 4, 6, 8, 2, 5, 7,
]);

const gpuBufferFirstMatrix = device.createBuffer({
    mappedAtCreation: true,
    size: firstMatrix.byteLength,
    usage: GPUBufferUsage.STORAGE
});
const arrayBufferFirstMatrix = gpuBufferFirstMatrix.getMappedRange();

new Float32Array(arrayBufferFirstMatrix).set(firstMatrix);
gpuBufferFirstMatrix.unmap();

// Друга матриця
const secondMatrix = new Float32Array([
    12 /* рядки */, 7 /* стовпчики */,
    6, 2, 4, 8, 5, 3, 8,
    0, 9, 6, 0, 9, 6, 7,
    5, 1, 3, 9, 3, 9, 9,
    0, 2, 5, 8, 1, 9, 0,
    3, 1, 3, 3, 9, 5, 9,
    4, 2, 2, 8, 4, 4, 7,
    6, 2, 3, 3, 5, 9, 8,
    4, 0, 1, 6, 5, 1, 7,
    1, 2, 0, 0, 0, 2, 2,
    5, 0, 7, 5, 8, 1, 3,
    9, 5, 8, 7, 1, 6, 8,
    9, 2, 3, 8, 2, 3, 0
]);

const gpuBufferSecondMatrix = device.createBuffer({
    mappedAtCreation: true,
    size: secondMatrix.byteLength,
    usage: GPUBufferUsage.STORAGE
});
const arrayBufferSecondMatrix = gpuBufferSecondMatrix.getMappedRange();
new Float32Array(arrayBufferSecondMatrix).set(secondMatrix);
gpuBufferSecondMatrix.unmap();

// Результуюча матриця
const resultMatrixBufferSize = Float32Array.BYTES_PER_ELEMENT * (2 + firstMatrix[0] * secondMatrix[1]);
const resultMatrixBuffer = device.createBuffer({
    size: resultMatrixBufferSize,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC
});
```

Рисунок 19 Множення матриць. Створення необхідних буферів

5.1.3 Група зв’язування

Концепція групи зв’язування (англ. Bind Group) та структури групи зв’язування (англ. Bind Group Layout) у WebGPU є особливо важливими. Групою зв’язування називають набір об’єктів (буфери пам’яті, текстури тощо), які використовуються при виконанні обчислень на графічному процесорі і стають доступними у момент виконання конвеєра. Структура групи зв’язування необхідна для того, щоб попередньо визначити типи даних, цілі та використання цих об’єктів, що дозволяє графічному процесору завчасно визначити, як найбільш ефективно виконати заданий конвеєр.

У конкретному прикладному випадку можна сказати, що структура групи зв’язування визначає інтерфейс для входів та виходів шейдера, а група зв’язування представляє собою фактичні дані, якими оперуватиме шейдер.

У наведеному нижче прикладі структура групи зв’язування визначає наявність буферів необхідних для обчислювального шейдеру: двох, призначених лише для читання та зберігання інформації, та одного буфера для зберігання. Група ж зв’язування, визначена для цієї структури, зіставляє ті оголошені сутності із реальними даними. У даному випадку gpuBufferFirstMatrix до прив’язки (англ. binding) 0, gpuBufferSecondMatrix до прив’язки 1 і resultMatrixBuffer до прив’язки 2.

```
const bindGroupLayout = device.createBindGroupLayout({
    entries: [
        {
            binding: 0,
            visibility: GPUShaderStage.COMPUTE,
            buffer: {
                type: "read-only-storage"
            }
        },
        {
            binding: 1,
            visibility: GPUShaderStage.COMPUTE,
            buffer: {
                type: "read-only-storage"
            }
        },
        {
            binding: 2,
            visibility: GPUShaderStage.COMPUTE,
            buffer: {
                type: "storage"
            }
        }
    ]
});
const bindGroup = device.createBindGroup({
    layout: bindGroupLayout,
    entries: [
        {
            binding: 0,
            resource: {
                buffer: gpuBufferFirstMatrix
            }
        },
        {
            binding: 1,
            resource: {
                buffer: gpuBufferSecondMatrix
            }
        },
        {
            binding: 2,
            resource: {
                buffer: resultMatrixBuffer
            }
        }
    ]
});
```

Рисунок 20 Множення матриць. Створення групи зв’язування та її структури

5.1.4 Створення шейдерів

Як зазначалося раніше, для написання шейдерів WebGPU надає власну мову шейдерів WGSL, яка легко транслюється у SPIR-V. Нижче наведений шейдер, який використовує буфери, передані раніше у якості вхідних даних як частина групи зв'язування, і реалізовує алгоритм множення двох матриць. Варто зазначити, що доступ до різних буферів відбувається за допомогою використання декоратора "@binding" з індексом елемента, використаним на етапі прив'язки під час створення структури групи зв'язування.

```
const shaderModule = device.createShaderModule({
    code: `
      struct Matrix {
        size : vec2f,
        numbers: array<f32>,
      }

      @group(0) @binding(0) var<storage, read> firstMatrix : Matrix;
      @group(0) @binding(1) var<storage, read> secondMatrix : Matrix;
      @group(0) @binding(2) var<storage, read_write> resultMatrix : Matrix;

      @compute @workgroup_size(64)
      fn main(@builtin(global_invocation_id) global_id : vec3u) {
        if (global_id.x >= u32(firstMatrix.size.x) || global_id.y >= u32(secondMatrix.size.y)) {
          return;
        }

        resultMatrix.size = vec2(firstMatrix.size.x, secondMatrix.size.y);

        let resultCell = vec2(global_id.x, global_id.y);
        var result = 0.0;
        for (var i = 0u; i < u32(firstMatrix.size.y); i = i + 1u) {
          let a = i + resultCell.x * u32(firstMatrix.size.y);
          let b = resultCell.y + i * u32(secondMatrix.size.y);
          result = result + firstMatrix.numbers[a] * secondMatrix.numbers[b];
        }

        let index = resultCell.y + resultCell.x * u32(secondMatrix.size.y);
        resultMatrix.numbers[index] = result;
      }
    `
});
```

Рисунок 21 Множення матриць. Створення шейдерів

5.1.5 Налаштування конвеєра

Обчислювальний конвеєр — це об'єкт, який фактично описує обчислювальні операції. Щоб створити його, слід викликати метод createComputePipeline(). Він приймає два аргументи: структуру групи зв'язування і етап обчислення (англ.

compute stage), у якому вказується точка входу обчислювального шейдера (зазвичай нею виступає функція “main”) та сам обчислювальний шейдер.

```
const computePipeline = device.createComputePipeline({
    layout: device.createPipelineLayout({
        bindGroupLayouts: [bindGroupLayout]
    }),
    compute: {
        module: shaderModule,
        entryPoint: "main"
    }
});
```

Рисунок 22 Множення матриць. Створення конвеєра

5.1.6 Передача команд до GPU

Після створення групи зв’язування з трьома буферами та налаштування обчислювального конвеєра настав час їх використати. Спочатку треба запустити кодувальник команд за допомогою функції beginComputePass(), потім передати в нього конвеєр через setPipeline(computePipeline) і групу зв’язування, викликавши метод setBindGroup(0, bindGroup). Індекс 0 тут відповідає індексу групи “group(0)” у коді шейдеру.

```
const commandEncoder = device.createCommandEncoder();

const passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(computePipeline);
passEncoder.setBindGroup(0, bindGroup);
const workgroupCountX = Math.ceil(firstMatrix[0]);
const workgroupCountY = Math.ceil(secondMatrix[1]);
passEncoder.dispatchWorkgroups(workgroupCountX, workgroupCountY);
passEncoder.end();
```

Рисунок 23 Множення матриць. Запис команд

Після цього треба створити буфер, у який буде скопійована результуюча матриця, завершити кодування команд і відправити їх на обробку до графічного процесора.

```
const gpuReadBuffer = device.createBuffer({
    size: resultMatrixBufferSize,
    usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.MAP_READ
});
commandEncoder.copyBufferToBuffer(
    resultMatrixBuffer,
    0,
    gpuReadBuffer,
    0,
    resultMatrixBufferSize
);

const gpuCommands = commandEncoder.finish();
device.queue.submit([gpuCommands]);
```

Рисунок 24 Множення матриць. Відправлення на виконання до GPU

І, нарешті, отримавши результати множення, можемо вивести отриману матрицю на екран:

```
await gpuReadBuffer.mapAsync(GPUMapMode.READ);
const arrayBuffer = gpuReadBuffer.getMappedRange();
const result = new Float32Array(arrayBuffer)

let formattedResult = ''
result.forEach((item, index) => {
    if (index > 1) {
        if ((index - 2) % result[1] === 0) {
            formattedResult += `<br>`
        }
        formattedResult += `${item} `
    }
})
const outputElement = document.getElementById('result');
outputElement.innerHTML = formattedResult
```

Рисунок 25 Множення матриць. Відображення отриманого результату

Перемножуючи дві матриці, які задавалися у коді в якості тестових даних раніше, отримуємо наступний результат, правильність якого була перевірена:

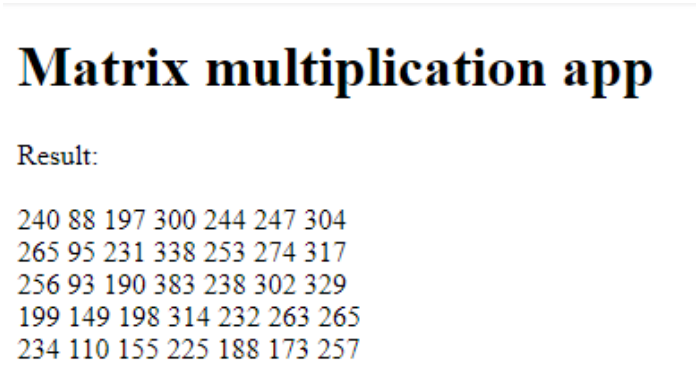

Рисунок 26 Результат роботи програми

5.1.7 Аналіз результатів

Як згадувалося раніше, єдиним прямим конкурентом WebGPU є WebGL, тому логічним буде порівняти продуктивність і швидкодію програм для множення матриць, які побудовані з використанням цих технологій. Теоретичні відмінності роботи з обчислювальними шейдерами цих двох API були описані у попередніх розділах цієї кваліфікаційної роботи, тому, щоб не повторюватися, нижче будуть проілюстровані лише схеми.

WebGL:

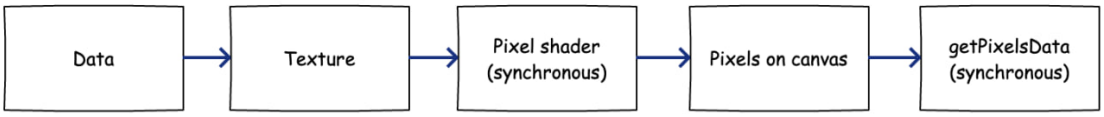

Рисунок 27 Робота з обчислювальними шейдерами у WebGL

WebGPU:

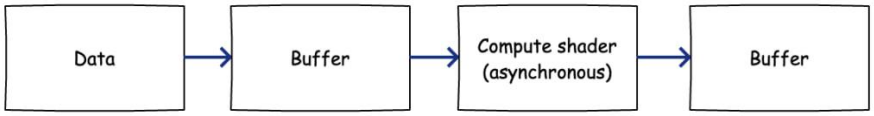

Рисунок 28 Робота з обчислювальними шейдерами у WebGPU

Можна побачити, що WebGPU має беззаперечну перевагу, адже:

1. У WebGPU дані можуть надсилатися до графічного процесора у вигляді буферів, їх не потрібно додатково конвертувати, як це вимагається у WebGL.
2. У WebGPU обчислення відбуваються асинхронно, не блокуючи основний потік виконання JavaScript (це особливо помітно при роботі зі складними фізичними симуляціями і сучасною 3-D графікою).
3. Для програм з використанням обчислювальних конвеєрів і обчислювальних шейдерів у WebGPU не є необхідною наявність елемента HTMLCanvasElement на сторінці.
4. Для даних, які повертає графічний процесор після обробки, не потрібно викликати дороговартісну (з точки зору використання системних ресурсів) синхронну функцію getPixelsData.
5. Відсутня необхідність конвертувати дані, які повертає GPU, з пікселів у необхідний формат для продовження роботи з ними у JavaScript програмі.

Для порівняння продуктивності і швидкодії програма, яка була написана у попередніх розділах на WebGPU, зазнала модифікації для можливості множення матриць довільних розмірностей (з урахуванням перевірки на їх коректність). Для досягнення кращої демонстрації, окрім двох програм із використанням WebGL та WebGPU, була написана третя програма, яка виконуватиме обчислення на центральному процесорі. Були отримані наступні результати:

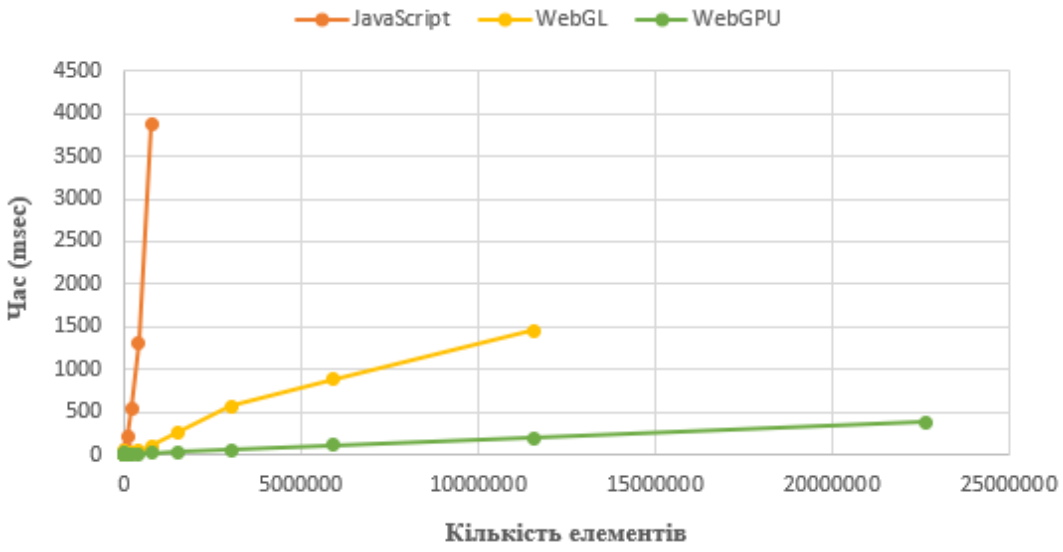

Рисунок 29 Порівняння продуктивності і швидкодії різних програм для множення матриць

Як бачимо, програма, написана із використанням технології WebGPU, значно перевершує інші програми в усіх показниках.

5.2 Робота з рендерингом графіки

Ще однією з проблем, які вирішує WebGPU, є полегшення написання програм для відображення сучасної графіки та надзвичайну оптимізацію процесів і використання системних ресурсів, необхідних для роботи з нею, на боці графічного процесора.

5.2.1 Відображення геометричних фігур

Простоту й зручність роботи із використанням WebGPU для візуалізації графіки буде продемонстровано на прикладі створення програми для відображення тривіальної геометричної фігури. Усі базові практичні підходи й концепції, які

будуть описані у ході розробки застосунку, легко застосовуються й до складніших проектів.

5.2.1.1 Налаштування середовища

Єдиною відмінністю із застосунком для множення матриць у цьому випадку буде необхідність наявності елементу HTMLCanvasElement на сторінці. Отож його необхідно створити та сконфігурувати контекст, деталі того, як це зробити, було описано у одному з попередніх розділів.

5.2.1.2 Створення буферів

Далі визначимо дані вершин, де об’єднаємо інформацію, яка стосується їх координат і кольорів, в один уніфікований буфер. Для цього спочатку ініціалізуємо масив цих значень:

```
const vertices = new Float32Array([
    -1.0, -1.0, 0, 1, 1, 0, 0, 1, // перша вершина
    -0.0, 1.0, 0, 1, 0, 1, 0, 1, // друга
    1.0, -1.0, 0, 1, 0, 0, 1, 1, // третя
]);
```

Рисунок 30 Робота з рендерингом графіки. Ініціалізація даних вершин

Кожна вершина подана у форматі (p_x,p_y,p_z,p_w,r,g,b,a), де перші чотири значення – координати, інші – колір у форматі rgba. Після цього необхідно створити дескриптори для вершинних буферів, де будуть визначені правила, за якими графічний процесор декодуватиме уніфікований буфер.

```
const vertexBuffersDescriptors = [
    {
        attributes: [
            {
                shaderLocation: 0,
                offset: 0,
                format: "float32x4",
            },//позиція
            {
                shaderLocation: 1,
                offset: 16,
                format: "float32x4",
            },//колір
        ],
        arrayStride: 32,
        stepMode: "vertex",
    },
];
```

Рисунок 31 Робота з рендерингом графіки. Створення дескриптора вершинного буфера

І нарешті створюємо необхідний буфер:

```
const vertexBuffer = device.createBuffer({
    size: vertices.byteLength,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
    mappedAtCreation: true,
});
new Float32Array(vertexBuffer.getMappedRange()).set(vertices);
vertexBuffer.unmap();
```

Рисунок 32 Робота з рендерингом графіки. Створення уніфікованого буфера

Також для більшості програм із динамічними анімаціями задається дескриптор для рендерингу, в контексті цієї програми він не є необхідним, але буде застосовуватися у наступних, більш складних, тому наведемо приклад:

```
const renderPassDescriptor = {
    colorAttachments: [
        {
            loadOp: "clear", // Очищувати зображення на кожне завантаження
            clearValue: { r: 0.0, g: 0.0, b: 0.0, a: 1.0 }, // Колір, яким очищувати зображення
            storeOp: "store", // Виводити результат на екран
        },
    ],
};
```

Рисунок 33 Робота з рендерингом графіки. Створення дескриптора рендерингу

5.2.1.3 Написання шейдерів

Для нашого трикутника вершинний шейдер запускатиметься тричі, а фрагментний – для кожного пікселя, який розташований між заданими вершинами, за один кадр (англ. frame). Код вершинного і фрагментного шейдерів представлений нижче:

```
const shaderModule = device.createShaderModule({
    code: `
    struct VertexOut {
        @builtin(position) position : vec4<f32>,
        @location(0) color : vec4<f32>,
    };
    @vertex
    fn vertex_main(@location(0) position: vec4<f32>,
                   @location(1) color: vec4<f32>) -> VertexOut
    {
        var output : VertexOut;
        output.position = position;
        output.color = color;
        return output;
    }
    @fragment
    fn fragment_main(fragData: VertexOut) -> @location(0) vec4<f32>
    {
        return fragData.color;
    }
`,
})
```

Рисунок 34 Робота з рендерингом графіки. Створення шейдерів

5.2.1.4 Налаштування конвеєра

Після написання шейдерів і створення необхідних буферів, залишилося налаштувати конвеєр візуалізації:

```
const pipeline = device.createRenderPipeline({
    layout: "auto",
    vertex: {
        module: shaderModule,
        entryPoint: "vertex_main",
        buffers: vertexBuffersDescriptors,
    },
    fragment: {
        module: shaderModule,
        entryPoint: "fragment_main",
        targets: [
            {
                format: presentationFormat,
            },
        ],
    },
    primitive: {
        topology: "triangle-list",
    },
});
```

Рисунок 35 Робота з рендерингом графіки. Налаштування конвеєра

5.2.1.5 Передача команд до GPU

Тепер треба лише відправити команди на виконання до графічного процесора та відобразити зображення на екрані:

```
function frame() {
    renderPassDescriptor.colorAttachments[0].view = context
        .getCurrentTexture()
        .createView();

    const commandEncoder = device.createCommandEncoder();

    const passEncoder =
        commandEncoder.beginRenderPass(renderPassDescriptor);

    passEncoder.setPipeline(pipeline);
    passEncoder.setVertexBuffer(0, vertexBuffer);

    passEncoder.draw(3);
    passEncoder.end();
    device.queue.submit([commandEncoder.finish()]);

    requestAnimationFrame(frame);
}

requestAnimationFrame(frame);
```

Рисунок 36 Робота з рендерингом графіки. Відправлення команд на виконання

Результат виконання програми:

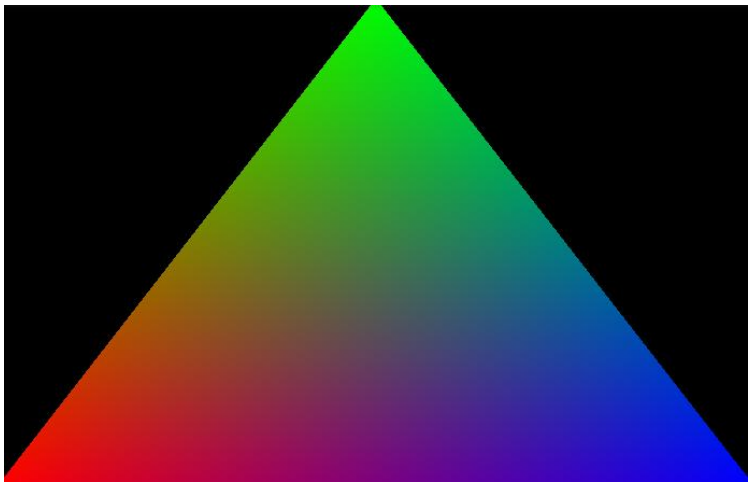

Рисунок 37 Результат виконання програми з відображення простої геометричної фігури

5.2.1.6 Ускладнення програми

Маючи всі необхідні знання по роботі з конвеєром візуалізації, буферами та графічними шейдерами у WebGPU, можна створювати програми із графікою будь-якої складності.

Відображення різнокольорового куба, який обертається навколо своєї осі:

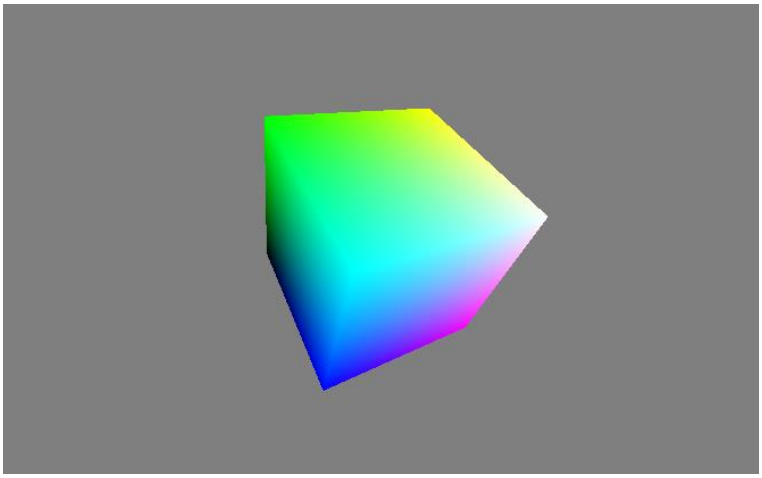

Рисунок 38 Приклад програми по роботі з графікою у WebGPU. Куб.

Декілька таких кубів:

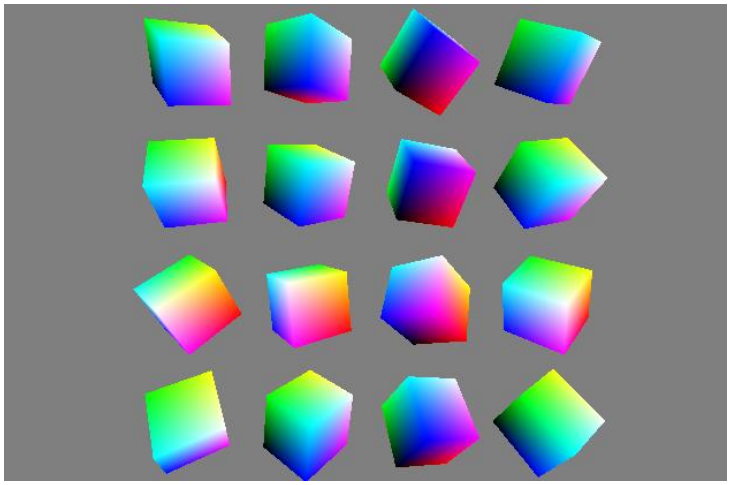

Рисунок 39 Приклад програми по роботі з графікою у WebGPU. Набір кубів

Накладання сторонньої текстури:

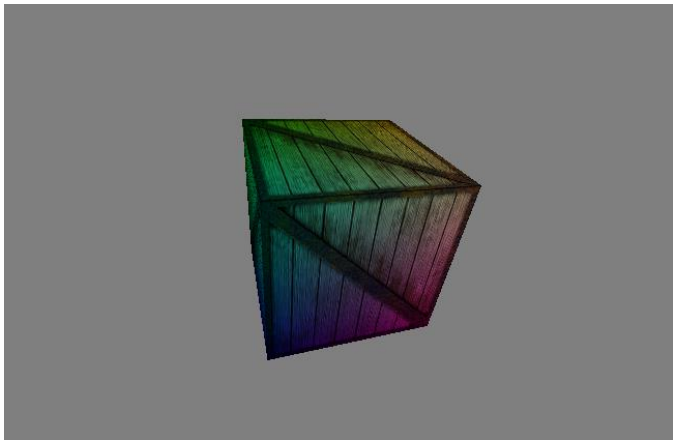

Рисунок 40 Приклад програми по роботі з графікою у WebGPU. Куб з накладанням текстури.

Усі ці програми були створені на основі попередньо описаної й по кількості написаного коду не перевищують навіть трьохсот стрічок, що ще раз доводить простоту й компактність WebGPU.

5.2.2 Огляд існуючих бібліотек

Незважаючи на те, що WebGPU дуже молода і все ще експериментальна технологія, все більше великих компаній почали звертати на неї увагу. Так, наприклад, підтримка WebGPU уже з'явилася у Babylon.js. Babylon.js – це безкоштовний рушій (англ. engine) для роботи із сучасною 3-D графікою у веббраузері. Він надає надзвичайно широкий інструментарій для роботи зі складними 3D-сценами із високоякісною графікою, аудіо та фізикою. Багато функцій можна розширити за допомогою плагінів та додаткових модулів.

На основі Babylon.js у поєднанні з WebGPU уже створені повноцінні 3-D відеоігри, які працюють у веббраузері, не поступаючись у своїй продуктивності традиційним, розробленим для встановлення на комп'ютер. Більше того, оскільки раніше Babylon.js масово використовував WebGL через відсутність альтернативи, його розробники попіклувалися про легкий перехід з WebGL на WebGPU, забезпечивши зворотну сумісність (англ. Backwards Compatibility). Для цього у коді програми, яка використовує цей рушій, слід всього лише замінити його ініціалізацію на:

```
const engine = new BABYLON.WebGPUEngine(canvas);
await engine.initAsync();
```

Рисунок 41 Ініціалізація рушія Babylon.js із використанням WebGPU

Ще одним яскравим прикладом переваги WebGPU над WebGL є тест продуктивності та швидкодії цих двох API, запропонований самими розробниками Babylon.js. Варто зазначити, що пристрій, який використовувався під час написання цієї кваліфікаційної роботи, має відеокарту NVIDIA GeForce GTX 1050 Ti.

У цьому тесті, при використанні WebGL, частота кадрів трималася на рівні 7-13 кадрів в секунду (англ. Frames Per Second або FPS).

Рисунок 42 Тест продуктивності WebGL

WebGPU ж показав значно кращі результати, продемонструвавши стабільні 42-45 кадрів в секунду:

Рисунок 43 Тест продуктивності WebGPU

Окрім Babylon.js, є й інші рушії, бібліотеки та фреймворки, які починають підтримувати й активно використовувати технологію WebGPU, як-от: Three.js, Dawn, wgpu, Use.GPU та багато інших. Їх кількість невпинно зростає.

5.3 Використання у сфері машинного навчання

Протягом останніх років сфера машинного навчання зазнала небаченого розвитку й нарешті отримала заслужену увагу публіки й всесвітнє захоплення. Отож використання WebGPU саме у цій сфері є одним із найбільш перспективних векторів розвитку. Одними з перших над цим замислилися розробники Web Stable Diffusion.

Оригінальна Stable Diffusion — це модель глибокого навчання для перетворення тексту в зображення, випущена в 2022 році. Вона використовується для створення детальних зображень на основі текстових описів. Подібні моделі зазвичай дуже великі і потребують важких обчислень, а це означає, що при розробці, скажімо, власних вебзастосунків, взаємодія з цими моделями реалізовуватиметься шляхом надсилання запитів до віддаленого сервера, на чиєму графічному процесорі і відбуватимуться всі необхідні обчислення. Крім того, більшість робочих навантажень мають виконуватися на певному типі графічних процесорів, де вже доступні популярні фреймворки для машинного навчання.

Web Stable Diffusion радикально це змінює. Цей проект переносить Stable Diffusion моделі до веббраузерів. Його ідея полягає в тому, що все працює всередині браузера без будь-якого стороннього сервера. Розробники цього застосунку переносять більшість складних обчислень на бік клієнта. Це дозволяє істотно економити на утримуванні серверної інфраструктури, а також покращує персоналізацію та рівень захисту конфіденційності клієнта. Враховуючи

підвищення потужностей персональних пристроїв користувачів, яке відбувається мало не експоненційно, подібний розподіл обрахунків із серверної на клієнтську частину має неабиякий сенс. Одним із способів реалізувати це міг би бути клієнтський настільний застосунок, але чи не краще було би просто відкрити вкладку браузера і почати працювати із моделлю? Саме це і реалізували розробники Web Stable Diffusion, використавши WebGPU. Уже зараз доступна демоверсія цього проекту за посиланням https://mlc.ai/web-stable-diffusion/. Картинка за текстовим запитом “Kremlin burning in flames” була згенерована за 82 секунди:

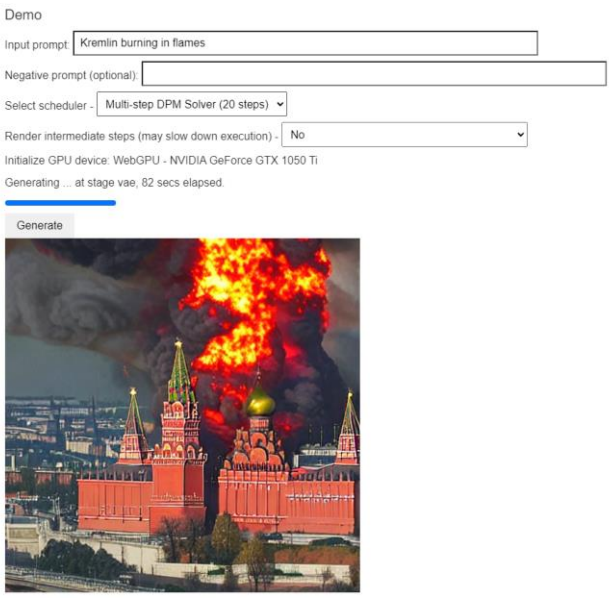

Рисунок 44 Приклад роботи Web Stable Diffusion

ВИСНОВКИ

У ході цієї кваліфікаційної роботи було досліджено WebGPU, новітню і перспективну вебтехнологію для використання потужностей графічного процесору у веббраузері, описано й проаналізовано основні особливості, перспективи і переваги цього прикладного програмного інтерфейсу, а також створено демонстраційні застосунки, де показано на реальних прикладах зручність й ефективність WebGPU у рендерингу графіки та обчисленнях загального призначення на графічних процесорах.

У порівнянні зі своїм прямим конкурентом, WebGL, WebGPU є простішим, легшим для розуміння та значно ефективнішим. На відміну від WebGL, він підтримує усі новітні функції, представлені в основних сучасних графічних API, наприклад останніх версіях DirectX, Vulkan чи Metal, та приносить усіх їхні найкращі практики та підходи до роботи з графічним процесором у царину вебтехнологій. Більше того, окрім звичного рендерингу графіки, WebGPU надає неймовірні можливості для організації обчислень загального призначення на GPU, по максимуму використовуючи високопаралельну природу цих пристроїв.

Беручи до уваги зручність застосування, шалену продуктивність і швидкодію, можна сміливо стверджувати, що WebGPU неодмінно стане новим вебстандартом та спричинить справжню революцію в індустрії.

СПИСОК ЛІТЕРАТУРИ

1. Xu J. Practical WebGPU Graphics: Creating Advanced Graphics on Web Using WebGPU - the Next-Generation Graphics API. 2021. 444 с.
2. Xu J. WebGPU by Examples: Learn and Explore Next-Generation Web Graphics and Compute API. 2023. 522 с.
3. W3C. WebGPU Explainer. URL: https://gpuweb.github.io/gpuweb/explainer/.
4. W3C. WebGPU. URL: https://www.w3.org/TR/webgpu/
5. W3C. WebGPU Shading Language. URL: https://www.w3.org/TR/WGSL/
6. Mozilla. WebGPU API - Web APIs | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API/
7. Bailey M., Cunningham S. Graphics Shaders: Theory and Practice, Second Edition. CRC Press LLC, 2016. 518 с.
8. Barlas G. Multicore and GPU Programming: An Integrated Approach. Elsevier Science & Technology Books, 2014. 698 с.
9. Cai Y., See S. GPU Computing and Applications. Springer, 2016. 300 с.