САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Направление: 02.03.02 «Фундаментальная информатика и информационные технологии»

ООП: Программирование и информационные технологии

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Тема: Использование современных Web-технологий для виртуализации платформы вычислений на основе GPU

Выполнил: Матюхин Александр Юрьевич 20.Б13-пу

ФИО *номер группы*

Научный руководитель: Дегтярев Александр Борисович, доктор технических наук, профессор кафедры компьютерного моделирования и многопроцессорных систем

ФИО, ученая степень, должность

Санкт-Петербург

2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сегодня можно уверенно утверждать, что Web-браузер является незаменимой компонентой практически любого персонального устройства — от компьютера до смартфона. Действительно, найти его можно практически где угодно, и, открыв его, можно увидеть, что за последние десятилетия Web-технологии совершили ощутимый прогресс: из простого средства передачи текстовых документов и информационных ресурсов они превратились в полноценную платформу для разработки самых разнообразных приложений: от простых калькуляторов до 3D-игр и редакторов изображений и видео. Это объясняется в первую очередь удобством и универсальностью Web-платформы — приложения на ней не нужно устанавливать и обновлять, они доступны по запросу и за счёт возможностей в плане виртуализации (в смысле абстракции от аппаратных ресурсов) такие приложения пишутся в одном экземпляре и сразу доступны практически на любой операционной системе. Так, можно сказать, что Web-браузер представляет собой ещё одну абстракцию — полноценную платформу, работающую на базе другой, нативной платформы в виде операционной системы.

В то же время в данный момент активно развивается такое направление, как GPGPU — **G**eneral-**P**urpose computing on **G**raphics-**P**rocessing **U**nits, или же вычисления при помощи GPU, которое позволяет нам использовать графические процессоры скорее не для непосредственно графики, а для решения более общих вычислительных задач. Такой подход применяется, когда задача предполагает выполнение однотипных операций над однородным, обычно большим объёмом данных (в отличие от вычислений на CPU, где предполагаются гетерогенные операции над различными и разнородными данными). В частности, GPGPU активно применяется в таких областях, как криптография и решение задач машинного обучения и искусственного интеллекта.

Как и с другими направлениями, существует множество решений для осуществления вычислений на GPU, из самых известных можно отметить CUDA [1] и OpenCL [2]. Вместе с тем, начиная работу с ними и, в частности, обучаясь процессу вычисления на GPU как направлению можно столкнуться с тем, что они довольно сложны как в плане установки и настройки (часто приходится действовать в соответствии с длинными и запутанными инструкциями для конкретной операционной системы), так и в плане взаимодействия. К тому же, иногда требуется использование специального

оборудования, которое может быть не у всех: так, та же CUDA, разрабатываемая компанией NVIDIA, предназначена только для использования на GPU, произведённых самой NVIDIA.

Следовательно, хотелось бы иметь такую виртуализированную платформу, на которой можно было бы практиковаться работе с GPGPU или же демонстрировать такие вычисления в аудитории, не прибегая к установке и настройке больших программных продуктов, желательно на как можно более обширном круге устройств. Возникает резонный вопрос: можно ли для этих целей использовать уже виртуализированную платформу Web-браузера? Ведь, если посмотреть со стороны, это кажется вполне логично — характеристики Web-платформы, описанные выше, как раз подходят под те требования, которые мы поставили для нашей платформы. И, на самом деле, такая возможность существует, и в ходе данной выпускной квалификационной работы подобная платформа действительно была реализована.

ПОСТАНОВКА ЗАДАЧИ

1. Предварительные объяснения

Перед тем, как перейти к постановке задачи, хотелось бы общими словами объяснить отличия GPU вычислений от вычислений на CPU и заметить несколько моментов, связанных с GPU вычислениями в нашем проекте.

Так, GPGPU предполагает параллельное исполнение большого числа потоков (например, на личном компьютере на выделенной видеокарте у меня имеется 2048 физических ядер для них), которые собраны в группы равных размеров (в частности, в нашем проекте такие группы называются workgroups, или же рабочие группы).

Эти группы размещаются в специальной трёхмерной сетке, соответственно каждая группа (ровно как и каждый поток) может быть однозначно идентифицирована при помощи координат её расположения в этой сетке.

Основная память разделена между потоками, то есть они используют одно и то же пространство памяти (далее, при создании проекта, мы будем оперировать такими терминами, как входные и выходные буферы — по сути, это и есть основная память, отличия лишь в том, что входные буферы используются для чтения данных и задаются заранее, а выходные буферы — для записи данных пользовательской программой).

Сами потоки исполняют одни и те же команды (наборы инструкций), только с разными параметрами, которыми являются как раз описанные ранее координаты групп или самих потоков.

CPU, в свою очередь, может параллельно исполнять гораздо меньшее число потоков (например, на том же компьютере, с которого пишется эта работа, физических ядер всего восемь, то есть на несколько порядков меньше, чем в GPU), но эти потоки могут иметь разные наборы инструкций. Основная память совместная, как в GPU, и для синхронизации работы с ней обычно используют механизмы наподобие семафоров, мьютексов и других.

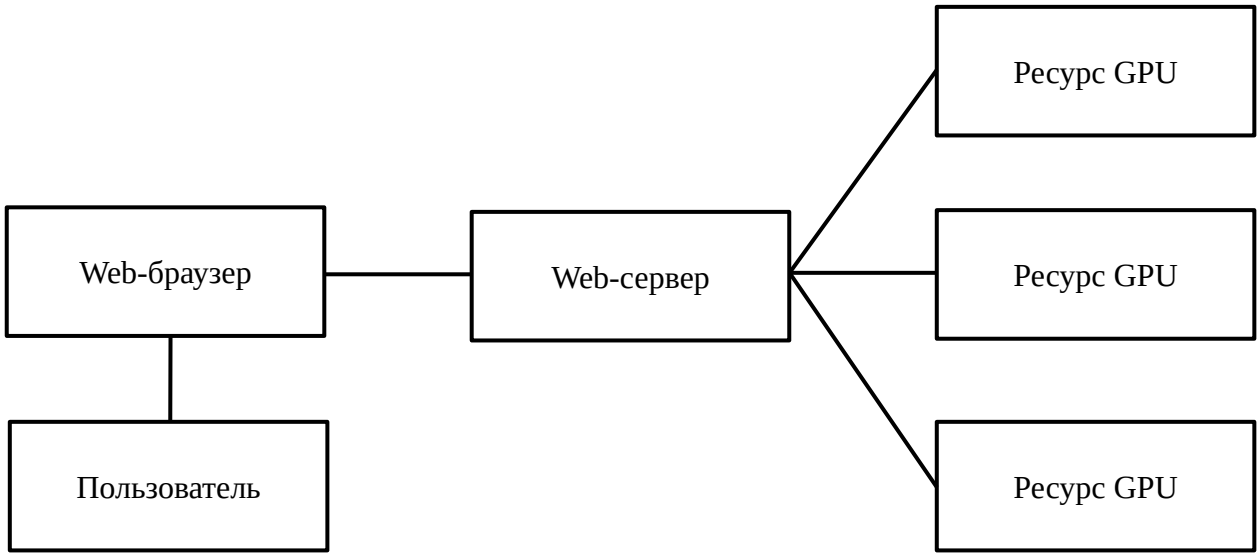
Нередко вычисления на CPU происходят в рамках всего одного потока, например когда накладные расходы на создание дополнительных потоков слишком велики или же когда алгоритм просто не предусматривает возможности «распараллеливания».

Таким образом, GPU практично использовать для узкого круга алгоритмов, в отличии от CPU, в частности для тех из них, которые хорошо «распараллеливаются» и предполагают в каждом потоке одни и те же операции (таким является, например, перемножение двух матриц, обильно рассматриваемое в работе) — для подобных алгоритмов GPU позволяет получить куда более высокое значение ускорения, чем CPU.

2. Постановка задачи

Начать хотелось бы с того, что для использования GPU в Web-браузере можно применить как минимум два разных подхода. Первый из них включает в себя использование окна Web-браузера просто в качестве интерфейса для взаимодействия с сервером (или несколькими серверами), которые в свою очередь и производит (производят) вычисления с использованием своих ресурсов GPU. Общая схема такого подхода предоставлена на Рисунке 1. В качестве примера можно привести платформу WebGPU [3] (отмечу, что не стоит путать её с упоминаемой далее Web-технологией WebGPU — они просто имеют одинаковое название), которая преследует цели обучения и, так же, как и мы, практики работы с GPU.

Рисунок 1 — Общая схема работы кластера GPU ресурсов, состоящего из непосредственно вычислительных машин и основного Web-сервера



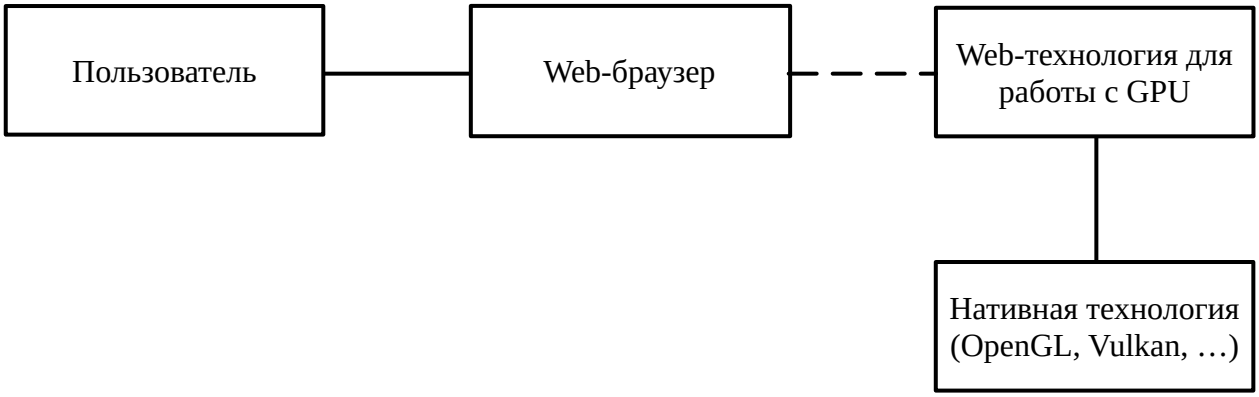
В качестве главного и, в свою очередь, решающего недостатка можно отметить то, что подобные платформы обычно закрыты и сильно ограничены — понятно, что для работы такой платформы требуется какой-то централизованный кластер GPU ресурсов,

обеспечение работы которого, очевидно, стоит больших денег. Так что такой подход нам не подходит, и он сразу отбрасывается.

Остановимся на втором подходе: использовании возможностей Web-платформы для утилизации непосредственно GPU пользователя, то есть Web-браузер перестаёт быть простым интерфейсом и становится уже полноценной средой для всего приложения.

На самом деле, за счёт стандартизированных Web-технологий возможности GPU уже давно используются в Web-браузере для определённых приложений. Общая схема того, как это устроено, предоставлена на Рисунке 2. Так, в качестве примера можно привести такие приложения, как симуляции импульсной нейронной P системы [4] и цунами [5]. Эти два примера показывают, что применение GPU в Web-браузере возможно как непосредственно для визуализации, так и для расчётов.

Рисунок 2 — Общая схема использования Web-платформы для GPU вычислений в Web-браузере



Стоит отметить, что оба предыдущих примера так или иначе утилизировали такую Web-технологию, как WebGL [6] — стандартизованный низкоуровневый API, основанный на нативной технологии OpenGL ES и разрабатываемый Khronos Group. Также важно, что WebGL (что видно и из аббревиатуры) создан в основном для графики, и для того, чтобы производить через него вычисления, приходится применять определённые усилия.

В то же время на данный момент активно разрабатывается и даже внедряется новая стандартизованная Web-технология, поддерживающая как вычисления, так и графику, под названием WebGPU [7].

Несмотря на то, что разработка этого Web-стандарта ещё не завершена, он довольно перспективен для решения задач, требующих GPU вычисления, на широком классе устройств. Так, например, существует исследование, в котором успешно применили WebGPU для стандарта шифрования AES, тем самым показав его перспективность [8].

Отметим, что также существуют исследования, в которых WebGL и WebGPU сравнивались в плане производительности [9], и оказалось, что WebGPU где-то на 7.5% лучше в этом плане, чем WebGL. Также хотелось бы отметить исследование с использованием WebGPU на кластере из Web-браузеров [10], которое показало, что, хоть WebGPU, естественно, хуже в плане производительности, чем CUDA, он показывает себя вполне сравнимо.

Итак, более лучшая производительность WebGPU, относительная низкоуровневость WebGL и цель WebGL как API для в первую очередь графики однозначно определяют наш выбор в пользу WebGPU для нашего приложения. Несмотря на то, что выбор мы осуществили, перед непосредственно разработкой мы проведём небольшое собственное исследование производительности WebGPU на одной конкретной машине (то есть не в рамках кластера, как это было сделано в статье, указанной выше).

Говоря про общую задачу, нам предстоит создать проект, который позволил бы практиковать и демонстрировать, например, в аудитории вычисления на GPU непосредственно в среде Web-браузера, на настоящем GPU (что важно, ведь, как я описывал ранее, вычисления на GPU отличаются от вычислений на CPU, и возможностей в общем однопоточной архитектуры страниц в Web-браузере (и в частности языка JavaScript) просто не хватает даже для простейших GPGPU задач — и это будет показано результатами первого исследования производительности WebGPU).

Также будет справедливо предъявить к проекту требование относительного удобства использования и доступности для людей с ограниченными возможностями, ведь наша цель — не только предоставить людям удобную площадку для практики, но и в целом дать им возможность определить, есть ли у них интерес к этому направлению. В случае отрицательного ответа на данный вопрос человек может просто закрыть вкладку Web-браузера, а в случае положительного — уже задуматься об установке более продвинутого программного решения.

ГЛАВА 1: ПРОИЗВОДИТЕЛЬНОСТЬ WEBGPU

1. Постановка задачи исследования

Итак, как я упоминал ранее, перед началом разработки нам хотелось бы провести собственное исследование производительности WebGPU на определённой машине. Сравнивать мы будем WebGPU и другую платформу GPGPU. Также рассмотрим соответствующие последовательные вычисления, в частности выполненные при помощи однопоточного языка JavaScript непосредственно в браузере. В качестве «соперника» я выбрал OpenCL как наиболее универсальную и открытую сейчас GPGPU платформу.

В качестве исследования был выбран именно практический эксперимент, так как теоретическую модель довольно сложно вывести и она, скорее всего, окажется не точна, а реальные результаты нам наиболее интересны.

2. Проведение исследования

В качестве алгоритма для эксперимента было выбрано перемножение двух квадратных матриц одного размера. Оно хорошо адаптируется к вычислению на GPU, просто в реализации, как раз представляет собой однотипные операции над довольно большим объёмом данных и часто используется почти во всех областях информационных технологий.

Произведение матриц на GPU можно считать довольно эффективно. Будем учитывать, что обычно произведение двух матриц $n \times n$ считается по следующей формуле [11]:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj},\ i = 1, \ldots, n,\ j = 1, \ldots, n \quad (1)$$

где c_{ij} — элементы матрицы-результата размера $n \times n$, a_{ik} — элементы первой матрицы и b_{kj} — элементы второй матрицы. Тогда в сетке потоков GPU можно каждому потоку дать задачу вычислять по формуле одно значение матрицы-результата на соответствующей позиции в сетке.

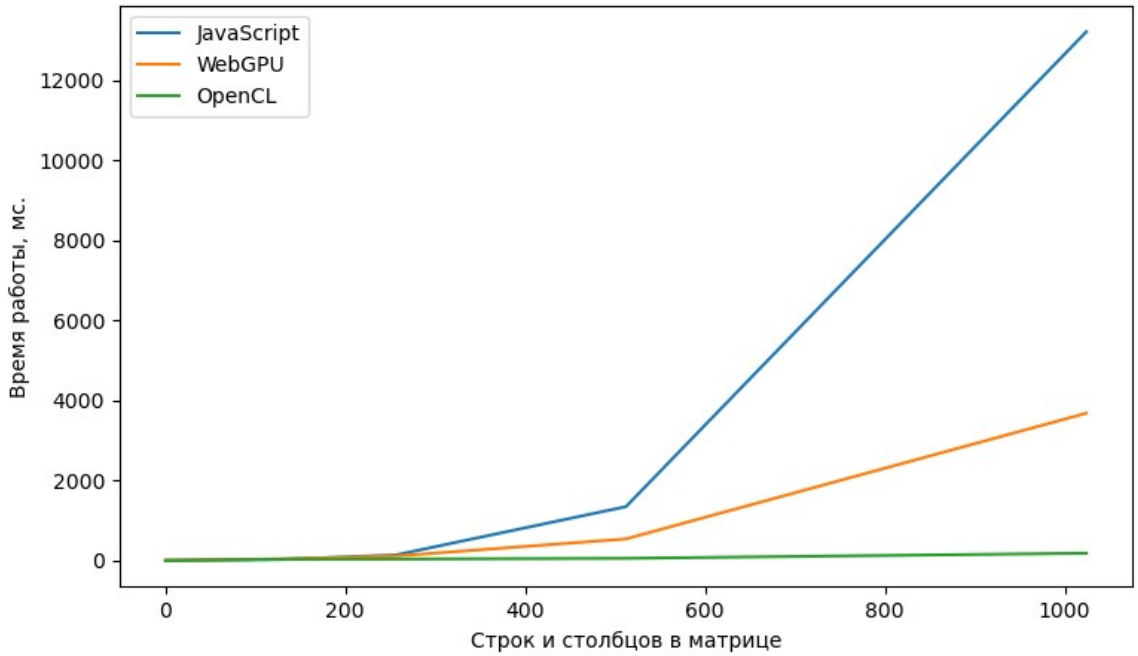
Для измерения производительности был поставлен практический эксперимент. Для этого эксперимента был написан код на HTML, JavaScript, Python и OpenCL, предоставленный в Приложениях А, Б и В.

В результате работы были произведены замеры на персональном компьютере (характеристики: Intel Core i7-11370H (CPU и GPU), 16GB RAM) для четырёх различных размеров матриц: 128×128, 256×256, 512×512 и 1024×1024 (матрицы из чисел с плавающими точками были сгенерированы случайно при помощи Python-скрипта, который предоставлен в Приложении Г). Результаты представлены ниже в Таблице 1 и более наглядно на рисунке 3:

Таблица 1 — Результаты экспериментов

	Время на вычисления в чистом JavaScript, мс.	Время на вычисления в WebGPU, мс.	Время на вычисления в OpenCL, мс.
128×128	25.3	36.7	31.5
256×256	136.6	110.5	35.1
512×512	1344.5	538.3	53.5
1024×1024	13214.2	3680.5	181.1

Рисунок 3 — Результаты экспериментов



3. Анализ результатов исследования

Посчитаем ускорение S для алгоритмов на WebGPU и OpenCL по сравнению с последовательным алгоритмом на JavaScript. Эффективность E считать не будем, так как значение p — количество процессоров — нам неизвестно, и скорее всего оно динамично. Считать ускорение будем по следующей формуле:

$$S=\frac{T_s}{T_p} \quad (2)$$

где T_s — время работы последовательного алгоритма, T_p — время работы параллельного алгоритма. В результате получаем следующие значения, предоставленные в Таблице 2:

Таблица 2 — Значения ускорений

	Для WebGPU	Для OpenCL
128×128	0,69	0,80
256×256	1,24	3,89
512×512	2,50	25,13
1024×1024	3,59	72,97

Как можно видеть из Таблиц 1 и 2 и Рисунка 3, WebGPU действительно придаёт ускорение по сравнению с обычной имплементацией, то есть сами по себе GPGPU вычисления в браузере возможны и работают уже сегодня.

Конечно, пока по сравнению с OpenCL показатели оставляют желать лучшего, но я считаю, что на данный момент них особо ориентироваться не стоит — достаточно знать, что WebGPU работает, и вспомнить, что он всё ещё активно разрабатывается, и скорее всего через несколько лет результаты будут гораздо более оптимистичные.

В принципе, ожидалось, что показатели будут хуже, так как в случае с WebGPU имеет место виртуализация, или абстракция от нативной платформы, соответственно применяется меньше оптимизаций и больше вычислительных мощностей уходит на накладные расходы.

Если подводить итог по поводу оправданности использования WebGPU как основы для нашей платформы, то можно сказать, что это вполне может стать «инвестицией» в будущее — для многих практических и демонстрационных задач и сегодняшних показателей будет достаточно, а учитывая достоинства самой Web-платформы и быстрые темпы развития технологий, связанных с ней, уже через какое-то время она сможет «побороться» и с такими проектами, как OpenCL и CUDA.

ГЛАВА 2: РАСШИРЕНИЕ ВИРТУАЛИЗАЦИИ?

1. Постановка задачи исследования

Во время исследования производительности WebGPU вполне резонно возникает вопрос, можно ли расширить то понятие виртуализации, которое мы себе поставили, с абстракции от аппаратных ресурсов до эмуляции какой-либо существующей GPGPU платформы непосредственно в среде Web-браузера. В частности, интересно, насколько это легко осуществимо сегодня и с использованием существующих инструментов.

В качестве рассматриваемой платформы мы опять выберем OpenCL, по тем же причинам, что и выше — его открытость и универсальность для различного аппаратного обеспечения.

Попытки реализовать это существуют уже очень давно, в том числе и сама The Khronos Group в 2014 году опубликовала стандарт WebCL, представляющий из себя платформу для гетерогенных параллельных вычислений в Web-браузерах, основанную на OpenCL [12]. К сожалению, 10 лет спустя, данный стандарт не реализован ни в одном популярном браузере и его развитие на данный момент остановилось на этой ступени.

Итак, в качестве задачи на данную главу нам необходимо рассмотреть, какие инструменты существуют для помощи в реализации этой задачи и насколько возможно их применение в нашем случае.

Исследование, опять же, будет именно практическое, так как теоретический подход в данном случае просто невозможен — так, нельзя аналитически понять, подходят нам те или иные инструменты или нет.

Хочу отметить, что нечто подобное уже предлагалось сделать в одной из статей ранее [8], но, к сожалению, возможно из-за трудностей перевода, я не обнаружил какой-то конкретики в нём по этому вопросу, что подталкивает меня на проведение самостоятельного исследования и приведение как можно более подробного описания действий, совершаемых мной в нём.

2. Проведение исследования

WebGPU как на данный момент, так и в целом поддерживает только один язык вычислительных шейдеров — WGSL, который создавался именно как универсальный язык шейдеров для WebGPU, предназначенный как для вычислений, так и для 3D-графики [13]. То есть, нам нужен инструмент, который может преобразовать код для вычислений на OpenCL в WGSL.

Так как прямых инструментов для подобного преобразования я не нашёл, необходимо воспользоваться промежуточным языком шейдеров — в качестве такого стоит использовать SPIR-V, который разрабатывается The Khronos Group, как и OpenCL, и как раз создан в качестве промежуточного языка для преобразования в среде 3D-графики и параллельных вычислений [14].

Для первого этапа преобразования OpenCL в SPIR-V можно рассмотреть следующий набор инструментов: Clang [15] в связке с LLVM-SPIRV [16] и Clspv [17]. При помощи следующих команд в терминале произведём все возможные комбинации SPIR-V для них [18]:

```
clang -c -o ./mulmat-1st.bc -emit-llvm -target spir ./mulmat.cl

clspv -o ./mulmat-2nd.bc --cl-std=CL1.0 --output-format=bc --spv-
version=1.0 ./mulmat.cl

llvm-spirv -o ./mulmat-1st.spv --spirv-max-version=1.0 ./mulmat-1st.bc

llvm-spirv -o ./mulmat-2nd.spv --spirv-max-version=1.0 ./mulmat-2nd.bc

clspv -o ./mulmat-3rd.spv --cl-std=CL1.0 --output-format=spv --spv-
version=1.0 ./mulmat.cl
```

В результате на выходе у нас есть 3 варианта SPIR-V файлов, которые потенциально можно преобразовать в WGSL. Попробуем воспользоваться следующими двумя инструментами для преобразования полученного кода SPIR-V в WGSL: Tint [19] и naga (часть wgpu) [20]. Запустим следующие команды в терминале:

```
naga --entry-point mulmat ./mulmat-1st.spv ./mulmat-1st.wgsl

naga --entry-point mulmat ./mulmat-2nd.spv ./mulmat-2nd.wgsl

naga --entry-point mulmat ./mulmat-3rd.spv ./mulmat-3rd.wgsl

tint -o ./mulmat-4th.wgsl --entry-point mulmat ./mulmat-1st.spv

tint -o ./mulmat-5th.wgsl --entry-point mulmat ./mulmat-2nd.spv
```

```
tint -o ./mulmat-6th.wgsl --entry-point mulmat ./mulmat-3rd.spv
```

К сожалению, ни одна из предыдущих команд успешно не сработала (упомяну, что я так же испытывал множество других вариаций этих команд, но тщетно). Все они так или иначе не срабатывают из-за проблем совместимости WGSL и OpenCL — так, WGSL сегодня обладает меньших функционалом, чем OpenCL, их спецификации не соответствуют друг другу, а инструменты ещё не «созрели», так как сам WGSL ещё очень нов. Таким образом, отображение OpenCL в WGSL на данный момент нам недоступно.

3. Анализ результатов исследования

Так как текущие инструменты не позволяют нам осуществить конвертацию из OpenCL в WGSL, я не вижу смысла на данный момент продолжать как-то рассматривать данный вопрос (например пытаться реализовать OpenCL C/C++ API через WebAssembly [21] в браузере). Также, с отсылкой на предыдущее исследование, стоит учитывать то, что если производить эмуляцию без GPU, то все операции, осуществляемые в Web-браузере на данный момент, будут однопоточны (либо же реализация многопоточности будет использовать столько накладных расходов, что потеряет всякий смысл).

Стоит также отметить, что даже если такая возможность и существует в настоящее время, и я просто не рассмотрел какой-то инструмент или подход, то она скорее всего не будет универсальной и в лучшем случае будет работать в частных случаях, по причинам, описанным в предыдущем параграфе.

ГЛАВА 3: СОЗДАНИЕ ПЛАТФОРМЫ

1. Постановка конкретных задач

Наконец, остановившись окончательно на WebGPU, приступим к созданию платформы для практики и демонстрации GPGPU в браузере (теперь уже точно утвердившись, что сами вычисления будут оформляться на языке шейдеров WGSL, сопутствующем WebGPU и поддерживающем вычисления на уровне языка в той же степени, что и графику).

Перед переходом к работе хотелось бы конкретно утвердить, что в целом должно быть сделано для того, чтобы как сам проект, так и вся выпускная квалификационная работа в целом считались бы успешными. Итак, в качестве основных требований можно выделить следующее:

- Редактор текста с подсветкой, чтобы можно было бы вводить сам текст кода WGSL, который требовалось бы запустить.

- Контрольная панель, с которой можно было бы задать входные и выходные буферы данных и соответствующие им данные, а также запустить сам предоставленный пользователем код с заданием некоторых параметров.

- Возможность использовать JavaScript для создания входных буферов данных, а также осуществления работы с выходными данными для удобства и отсутствия необходимости покидать окно Web-браузера в течение всего цикла.

- Вывод всех происходящих ошибок не только в консоль браузера, но и непосредственно на экран, чтобы была возможность их отслеживать и исправлять, не полагаясь на необходимость проверять упомянутую консоль.

- Некоторые элементы удобства пользователя, такие как поддержка нескольких языков и возможность выбора ночной темы, по довольно очевидным причинам.

- Доступность для людей с ограниченными возможностями (упомяну, Web-браузер уже на уровне платформы предоставляет мощные инструменты для этого, разработчику требуется лишь следовать установленным практикам и отслеживать через инструменты разработчика, что всё в порядке).

Данные требования, по моему мнению, вполне справедливы для поставленных перед проектом задач и его цели, а также помогут нам не отставать от общего курса в процессе разработки.

2. Выбор технологий

На данный момент существует множество технологий для создания Web-приложений. Основываясь на собственном опыте и на тех навыках, которые у меня есть на данный момент, я остановился на следующем выборе основных из них:

- Yarn [22] как менеджер пакетов. У меня есть опыт работы с ним, он функциональный, быстрый и поддерживает множество возможностей, если они потребуются в ходе разработки, такой например является применение патчей к внешним пакетам.

- NextJS [23] как основной фреймворк для разработки. Он мне знаком (хоть и в меньшей степени, чем другие технологии) и позволит сократить время на разработку многих прикладных к проекту задач, таких как разработка механизма рендеринга страниц для сопутствия дедупликации кода, создание сервера для приложения и других.

- JestJS [24] для unit-тестирования приложения. Unit-тестирование — это тестирование отдельных компонентов приложения, направленное на снижение количества ошибок в них и, как следствие, избегание неприятного опыта для пользователей. Забегая вперёд, скажу, что в ходе разработки тесты действительно не раз меня выручали, когда я совершал какие-либо оплошности в ходе написания кода.

- highlight.js [25] для подсветки текста WGSL и JavaScript кода. Подсветка кода важна при его написании, так как она помогает визуально выявлять ошибки и в целом упрощает ориентацию в коде.

- TypeScript [26] — язык программирования, компилирующийся в JavaScript и добавляющий систему типов как надстройку. Эта система существенно упрощает написание порой хаотичного JavaScript кода и предоставляет подсказки при разработке в помощь разработчику.

- Eslint [27] и Prettier [28] — помогают искать недочёты в коде, а также приводят его к единому стилю. В частности, мне они помогали не так сильно акцентировать внимание на оформлении кода и в целом писать его правильно.

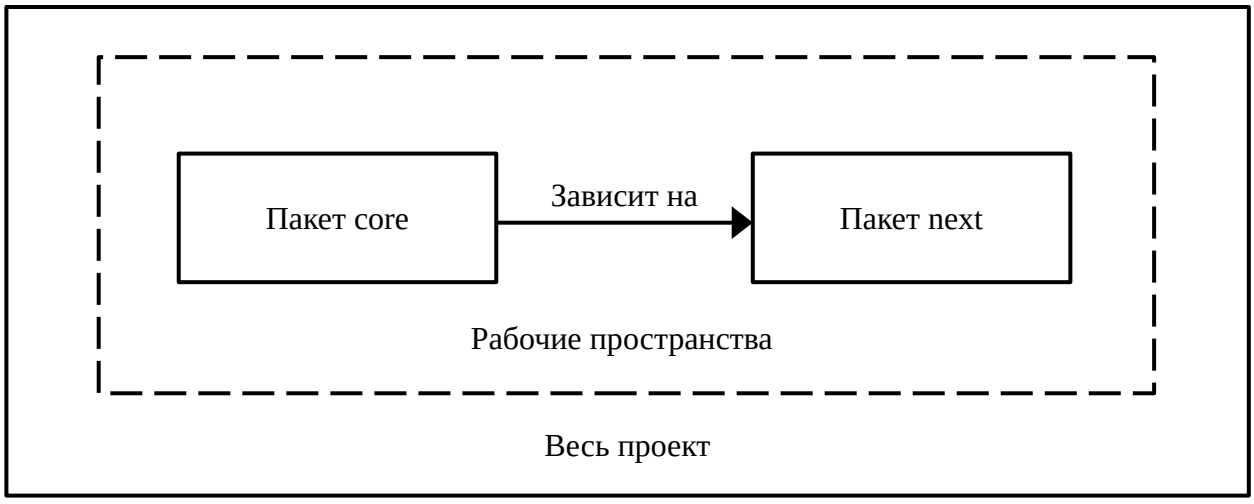
Теперь, когда инструменты выбраны, можно перейти к продумыванию основной архитектуры приложения, а впоследствии и её реализации.

3. Продумывание архитектуры

Понятно, что некоторые аспекты архитектуры уже задаются выбором инструментов — нашей задачей сейчас является использование их в полной возможности, а также более детальная обработка нюансов.

Одной из основных деталей архитектуры моего проекта я считаю разделение на два рабочих пространства, которые называются core и next. Схематично это представлено на следующем Рисунке 4:

Рисунок 4 — Общая схема архитектуры создаваемой платформы.



Как видно из схемы, такой подход отделяет пакет core, в котором находятся основная «бизнес» логика, связанная с WebGPU и общие утилиты, от пакета next, который напрямую завязан на NextJS и в котором находится всё, связанное с представлением пользователю.

Так, это даже в рамках одной платформы позволяет разделить обязанности в коде, что не только позволяет легче его разрабатывать и поддерживать, но и даёт возможность

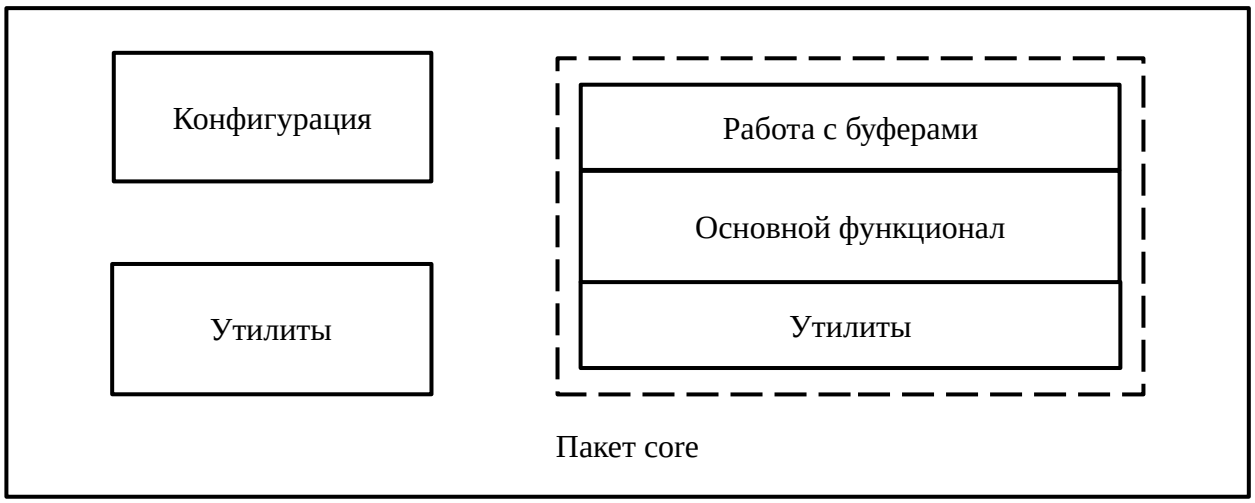
не без больших усилий, но всё же легче заменить NextJS и пакет next каким-нибудь другим проектом, при этом не трогая уже проверенный бизнес-код. Это важно, ведь в мире Web-разработки всё динамично, и предугадать дальнейшие её повороты просто невозможно.

Также, такая архитектура позволяет создать сразу несколько альтернативных «фронтендов» при появлении такой необходимости в будущем (для этого достаточно создать новый пакет для каждого «фронтенда» и поставить им зависимость от пакета core).

Архитектура пакета next задаётся конвенциями NextJS, которые подробно описаны на его сайте (следование им — одно из условий его использования, иначе сам инструмент будет мешать разработке, нежели чем помогать с ней).

Архитектура пакета core, на самом деле, очень проста — схематично её упрощенная версия (без мало-релевантных деталей) представлена на следующем рисунке 5:

Рисунок 5 — Общая схема пакета core



Как видно, отдельно вынесены общая конфигурация и утилиты, которые могут использоваться где угодно. В рамку из чёрточек выделен подпакет WebGPU — это и есть основная логика всего приложения, непосредственно инкапсулирующая в себе вычисления на GPU.

На данный момент мы обозначили основную архитектуру приложения и в целом распланировали, как мы будем его разрабатывать. Но как известно, ничто никогда не идёт точно по плану, и этот проект не станет исключением. Далее мы рассмотрим некоторые моменты из разработки проекта, на которые хотелось бы обратить внимание, обозначим возникшие трудности и то, как из этих ситуаций пришлось выходить.

4. Работа над требованием: редактор кода

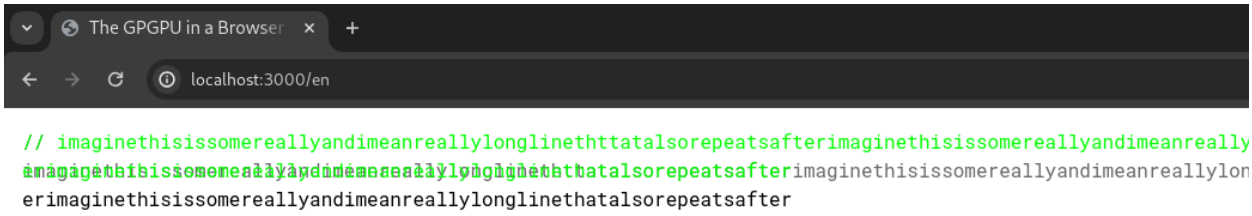
После создания «каркаса» проекта на основе описанной выше архитектуры, я приступил к созданию «мини-редактора», необходимого для ввода WGSL и JavaScript кода. Перед этим я пробовал найти готовое решение, но к сожалению тех, которые бы поддерживали язык WGSL, я не нашёл (стоит помнить, что сам WGSL ещё очень новый). К тому же, полноценный редактор значительно бы расширил размер скачиваемых пользователем ресурсов, а большинство из этих ресурсов никогда бы толком не использовались.

Так как библиотеки для подсветки текста, и, в частности, highlight.js, не поддерживают ввод текста, этот момент мне придётся реализовать самому. К счастью, я сумел найти Web-статью, которая подсказала мне идею, как это сделать [29] (отмечу, что изначально я как раз использовал react-syntax-highlighter, указанный в статье, но потом я счёл его не таким полезным для конкретно своего приложения и перешёл на эквивалентный подход, не предполагающий использования внешних библиотек).

Идея, в общих чертах, состоит в наложении полностью прозрачного (кроме мигающего курсора) окна ввода текста на непосредственно подсвеченный код, и синхронизация их за счёт средств библиотеки ReactJS [30] (которая является частью NextJS, так что в нашем случае её использование уже предполагается).

Во время реализации идеи, к сожалению, всё пошло не так, как хотелось бы. Несмотря на то, что содержимое синхронизировалось, иногда пропадали, например, новые линии при их вводе после основного текста, что может придавать пользователю определённый дискомфорт при вводе текста своей программы. К тому же, из-за неправильного подбора стилей иногда невидимый и видимый текст «разъезжались», что, определённо, вредило функционалу приложения. Увидеть этот «эффект» можно на Рисунке 6 ниже (для наглядности, прозрачный текст ввода был сделан ярко-зелёным):

Рисунок 6 — Демонстрация проблем, одно время происходивших с редактором текста (реконструкция)



Как можно видеть, вводить текст так становиться невозможным. Но, несмотря на трудности, мне удалось устранить эти проблемы, и теперь текст и имеет подсветку, и вводится стабильно хорошо, причём всё это сделано с минимально затраченными ресурсами. Считаю, что это требование было успешно выполнено, что показано далее на Рисунке 7:

Рисунок 7 — Ввод пользователем текста программы на языке WGSL

5. Работа над требованиями: контрольная панель, элементы удобства и разработка для людей с ограниченными возможностями

При помощи контрольной панели пользователь взаимодействует со всем приложением, поэтому очевидно, что она должна быть эргономичной и в свою очередь простой в использовании. Я считаю, что мне удалось это осуществить, сделав её к тому же довольно приятной на взгляд.

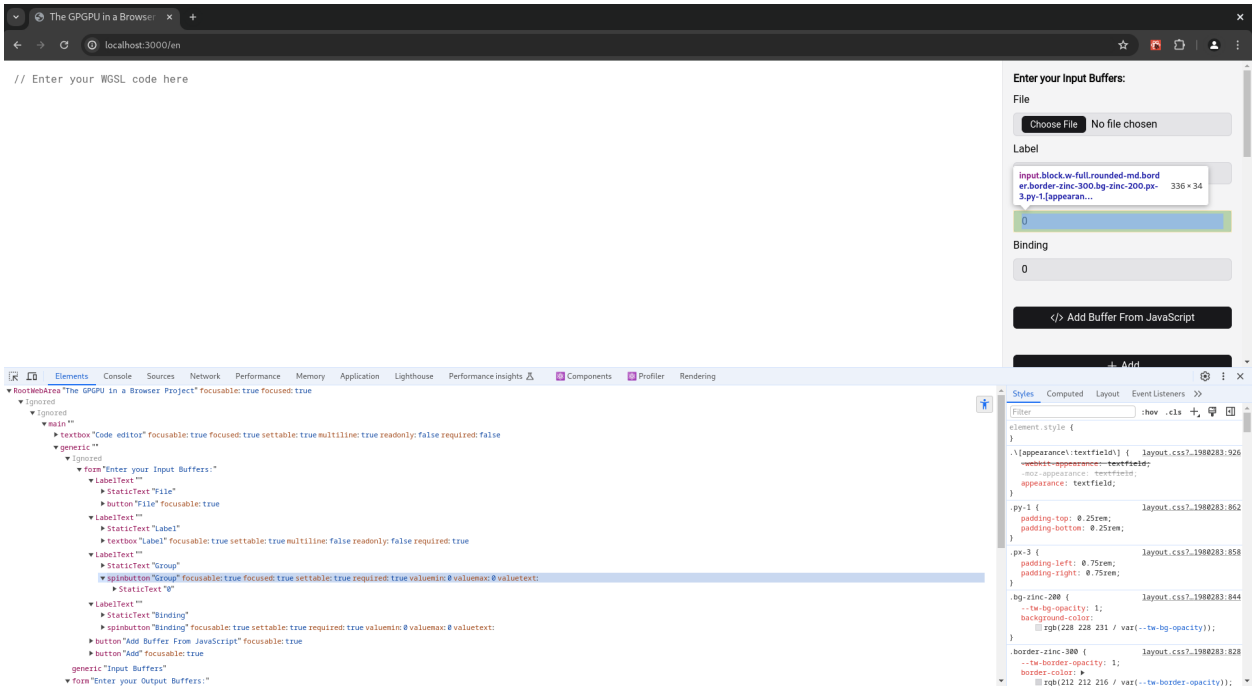
Из особых моментов здесь сложно что-то выделить, так что сейчас хотелось бы затронуть ту тему, которая была выделена как отдельное требование и, как мне кажется, наиболее релевантна как раз для контрольной панели: разработка приложения так, чтобы оно было доступно для людей с ограниченными возможностями.

Ни для кого не секрет, что проблема доступности сайтов велика в Web-среде даже в настоящее время. Многие сайты, как крупные, так и малые, не уделяют должного внимания этому, нанося ущерб своим пользователям. Так, в исследованиях говорится, что «где-то 15% людей в мире имеют какие-либо ограничения, и ещё больше могут страдать временными недугами» [31]. В том же исследовании указано, что «на каждую проверенную Web-страницу приходится в среднем по 30 ошибок в плане доступности, а на каждый Web-сайт — по 521 ошибок», что подтверждает сказанное выше.

Чтобы не допустить подобного в моём приложении, я использовал соответствующие стандарты и инструменты. Среди стандартов можно выделить HTML5 [32] и WAI-ARIA [33], которые поддерживаются абсолютно всеми современными популярными браузерами.

В качестве инструментов первым рассмотрим встроенный в средства разработчика современных браузеров «Accessibility Tree». Это — некая теневая копия DOM в виде дерева, представляющего из себя элементы Web-страницы и информацию, которая обычно тем или иным образом используется средствами для людей с ограниченными возможностями. На Рисунке 8 показано, как оно выглядит при работе над моим приложением и что оно в целом из себя представляет:

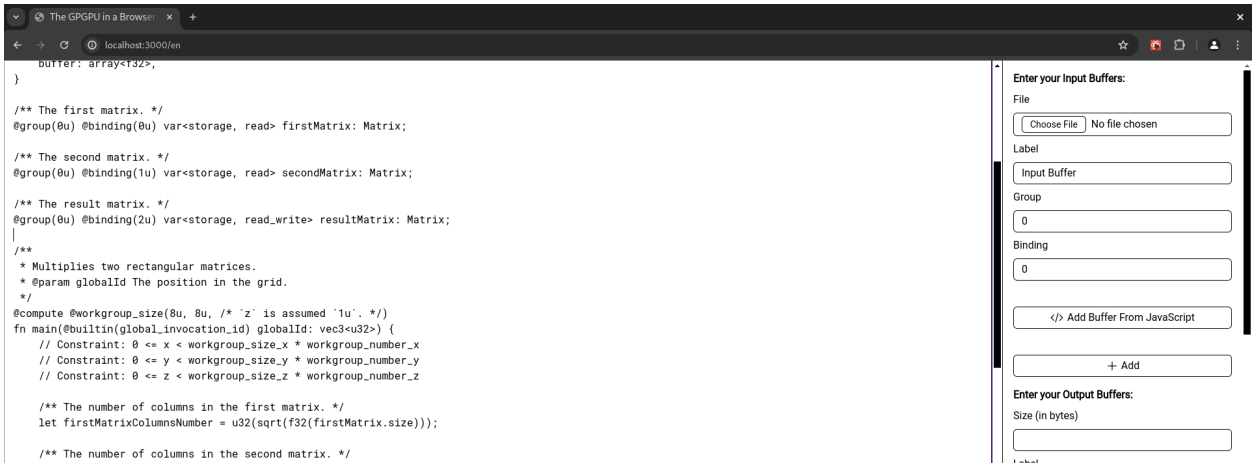
Рисунок 8 — Демонстрация Accessibility Tree



В своём приложении я его использовал, чтобы убедиться, что мой проект действительно является доступным, а если находил какие-то недочёты, то исправлял их с использованием стандартов, указанных выше.

К тому же, стоит упомянуть такую настройку стиля в современных браузерах, которая обычно интегрирована с операционной системой, как forced-colors, которая принудительно переключает палитру контента на заранее заданную, чтобы пользователю было легче ознакомиться с содержимым. Для её правильной работы мне пришлось подправить некоторые стили, но я это осуществил так, что без этой настройки это совершенно незаметно. На Рисунке 9 показано, как выглядит приложение, если включить эту настройку:

Рисунок 9 — Вид приложения при включённой настройке forced-colors



Помимо этого, при работе со стилями, я применял лучшие практики, как, в качестве примера, использование outline, или окружающей линии для пометки активного на данный момент элемента Web-страницы.

В рамках разговора про контрольную панель упомянем элементы удобства. Как мы помним, в качестве них были заявлены поддержка нескольких языков и возможность выбора темы. Действительно, в рамках проекта они были реализованы (упомяну разве, что перевод на русский язык далеко не полный, что, однако, легко исправимо и служит наглядной демонстрацией того, что в случаях отсутствия перевода какого-либо элемента используется язык по умолчанию). Эти элементы продемонстрированы ниже на Рисунках 10 и 11:

Рисунок 10 — Переключатели языка и темы на контрольной панели и вид приложения при включённой «ночной» теме

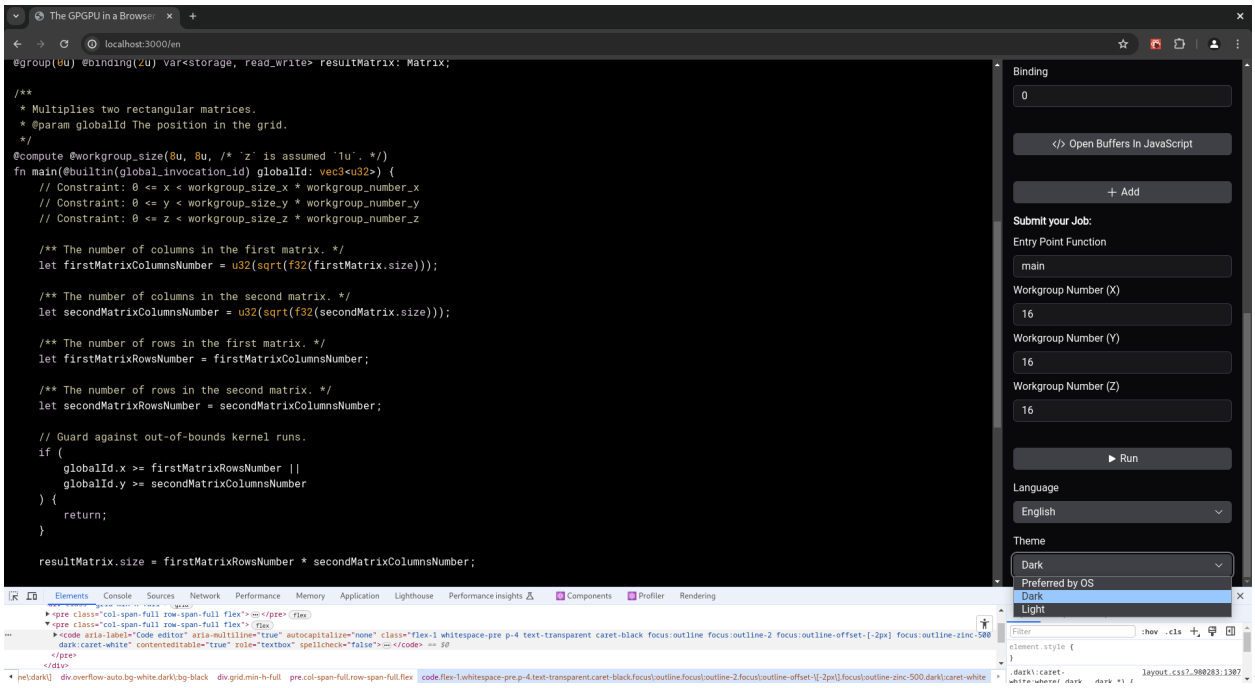


Рисунок 11 — Переведённый на русский язык текст кода по умолчанию и не переведённые элементы контрольной панели, использующие язык по умолчанию

Подытоживая, я могу с уверенностью сказать, что все поставленные требования и пожелания по контрольной панели, элементам удобства и доступности сайта для людей с ограниченными возможностями были выполнены.

6. Разработка основного ядра взаимодействия с WebGPU

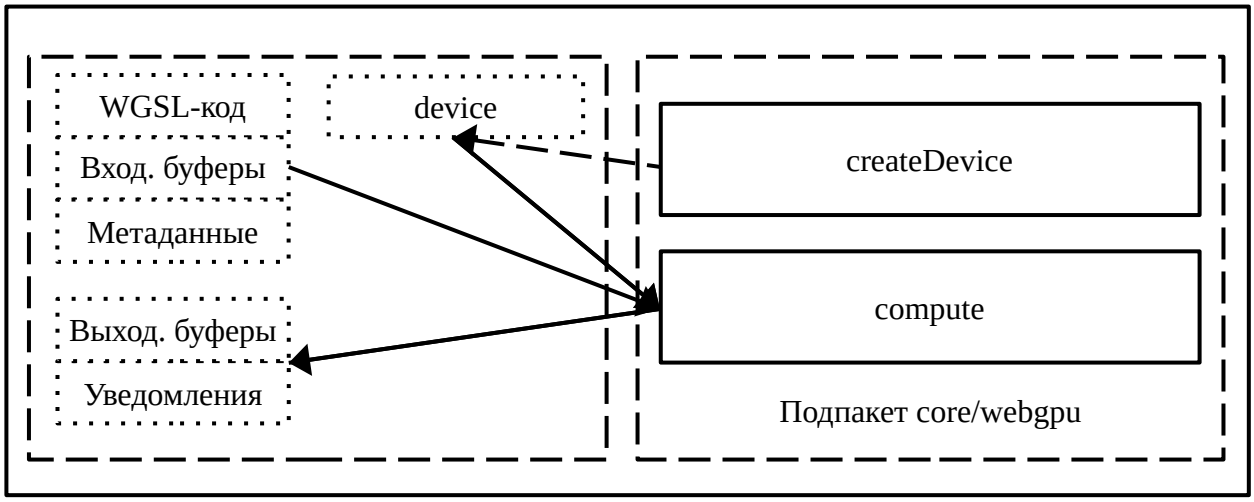
Как было показано в Рисунке 5, подпакет WebGPU в пакете core, который является основным звеном логики в нашем приложении, состоит из трёх основных частей: работа с буферами, основная часть и утилиты.

Исполнение пользовательского кода начинается с вызова функции createDevice из утилит. Этот вызов происходит во «фронтенде», проверяет присутствие WebGPU API в браузере и создаёт устройство WebGPU, которое затем может быть несколько раз использовано для исполнения WGSL кода, предоставленного пользователем.

Позднее, опять же из «фронтенда» вызывается функция compute, которой предоставляется код пользователя на WGSL, WebGPU устройство, входные буферы и их метаданные, метаданные выходных буферов и некоторые другие метаданные. Взамен compute возвращает уведомления, которые были созданы во время исполнения кода, и выходные буферы, данные в которых получены с GPU теми вычислениями, которые задал пользователь.

Наглядно всё это представлено ниже на Рисунке 12 (стрелкой с чёрточками представлен процесс, который происходит одноразово, а обычными — процессы, происходящие на каждой итерации запуска пользователем своего кода):

Рисунок 12 — Схематичное представление процесса запуска кода



Далее рассмотрим непосредственно процессы, происходящие внутри core и связанные с вызовом compute.

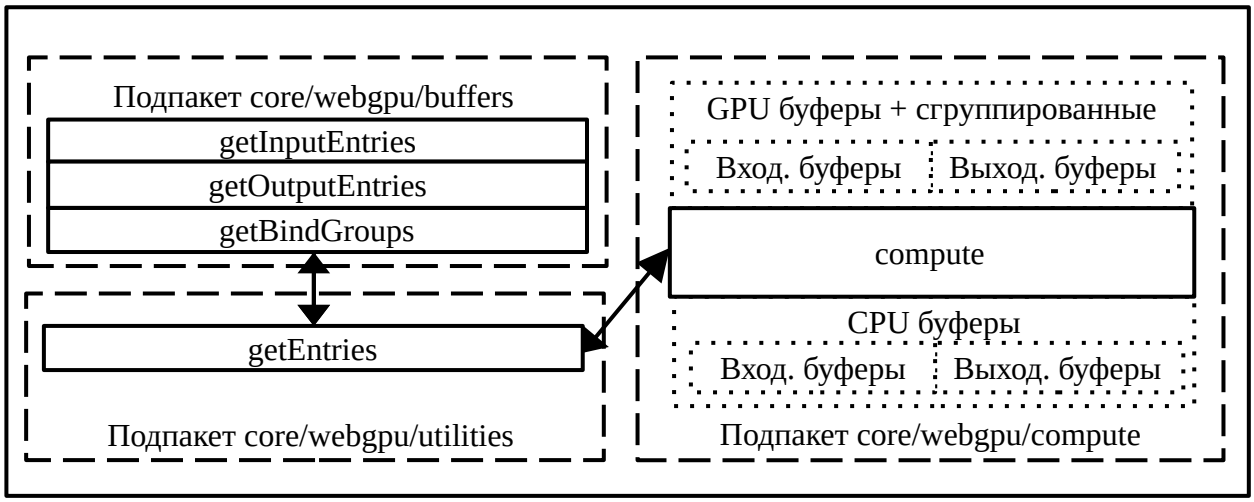
Сначала скажу, что память GPU и CPU всегда разделена, в том числе и в случае с WebGPU: CPU не может напрямую взаимодействовать с памятью GPU, и наоборот. Сейчас те входные буферы (в виде ArrayBuffer, то есть просто бинарного массива данных), которые были предоставлены нам, находятся в ведении JavaScript, а соответственно CPU.

Чтобы передать входной буфер GPU, compute через утилиту getEntries вызывает getInputBindGroupEntries, который, используя WebGPU API (в частности вызов device.createBuffer), создаёт буфер в памяти GPU, помещает туда контент и возвращает его с соответствующей мета-информацией, необходимой GPU для определения и идентификации буфера.

Затем аналогичная процедура, только без передачи информации из буферов, под неожиданным названием getOutputBindGroupEntries проводится для выходных буферов (ведь они тоже должны быть известны GPU уже при прогоне программы). Также эти буферы специальным образом группируются при помощи вызова getBindGroups, который создаёт пригодные для передачи в WebGPU API группы буферов с ссылками на сами GPU буферы.

Проиллюстрируем всё сказанное выше на Рисунке 13:

Рисунок 13 — Схематичное представление создания GPU буферов в процессе compute



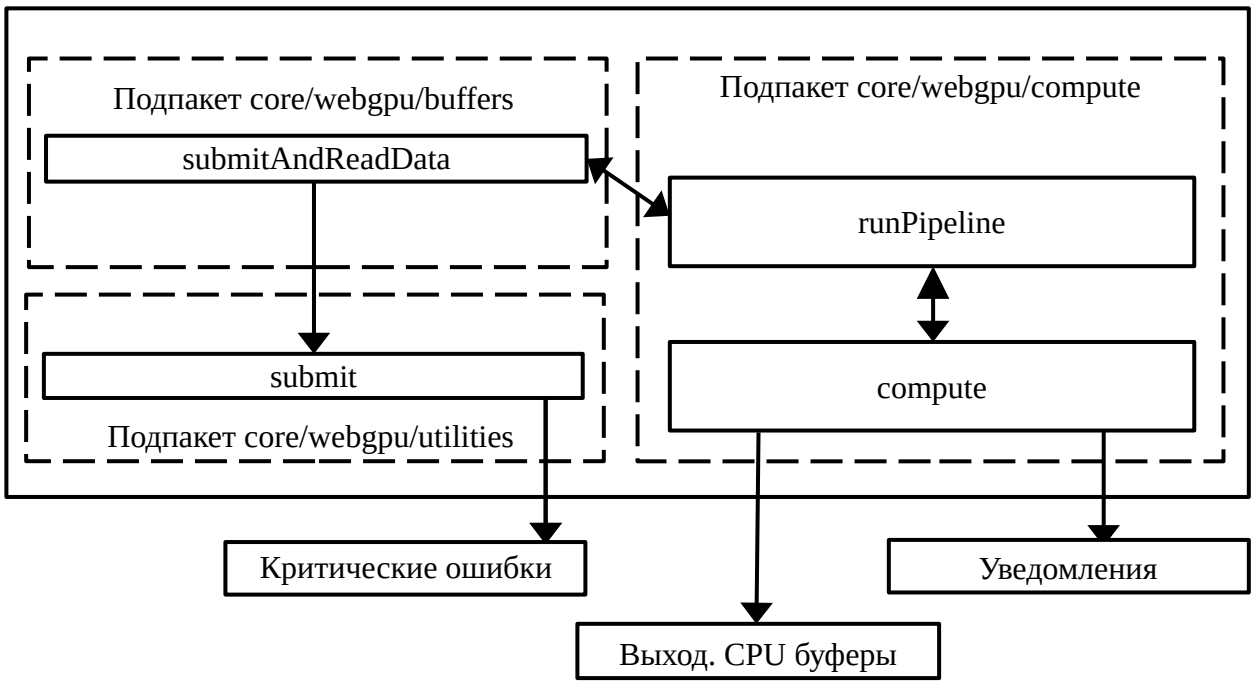
После того, как все GPU буферы созданы, compute компилирует WGSL код пользователя и создаёт «линию» для прогонки кода. Сама compute потом передаёт контроль соседней функции runPipeline, которая прогон осуществляет, и вернёт её результат.

runPipeline задаёт параметры прогонки и завершает её цикл — остаётся только отправить эту прогонку на GPU и непосредственно совершить её. Для этого делается вызов к «буферной» функции submitAndReadData, которая всё отправляет, а также читает результаты из GPU и создаёт для них буферы на CPU, которые в итоге и отправятся назад к пользователю.

Помимо этого, как уже упоминалось, из compute возвращаются уведомления, которые были созданы в ходе исполнения WGSL программы [34], но с критическими ошибками дела обстоят иначе: в ходе вызова submitAndReadData вызывается функция submit, которая, помимо непосредственно отправки всех команд на GPU, также при помощи механизма исключений JavaScript «выкидывает» критическую ошибку, если та произошла. Подразумевается, что «фронтенд» обработает данную ошибку, и в случае с пакетом next именно это и происходит.

Опишем сказанные выше процессы визуально, при помощи Рисунка 14:

Рисунок 14 — Схема прогонки, отправки данных пользователю и «выброса» ошибок:



Можно сказать, что на этом основной цикл завершён — теперь пакет next дополнит записи выходных буферов соответствующими значениями, откуда они уже будут доступны пользователю, и выведет все сообщения и ошибки, появившиеся в ходе цикла. Впоследствии пользователь может запустить программу снова, что-либо изменив, и цикл повторится снова.

Отмечу, что никакой ручной чисти памяти не требуется: все GPU буферы, использованные в ходе цикла, автоматически удаляются, в то время как CPU буферы находятся в ведении JavaScript, который автоматически помечает данные как ненужные в тот момент, когда все ссылки на них исчезают из кода, и регулярно удаляет такие данные из памяти (это называется garbage collection). Так, в моём коде все ссылки на устаревшие и более не нужные буферы действительно удаляются, что означает, что проблем из-за этого при долгосрочном использовании приложения не возникнет.

В целом, подход в плане архитектуры бизнес-логики относительно простой, но я считаю, что в данном случае он таким и должен быть: нет смысла придумывать что-то более запутанное просто ради того, чтобы устроить путаницу. Текущая архитектура вполне расширяема и её можно быстро понять внешнему разработчику, если это потребуется, что, учитывая то, что мы привязываемся не просто к какой-то библиотеке, а к принятому стандарту, не вызывает каких-либо других очевидных вопросов.

Теперь, когда мы знаем реализацию ядра проекта, мы опять рассмотрим подтемы, которые относятся к части с «фронтендом», в частности по реализации механизма вывода уведомлений и ошибок пользователю (которые были либо «выкинуты», либо возвращены вместе с выходными буферами), а также ввод входных буферов и работа с выходными непосредственно в окне браузера с нашей платформой.

7. Работа над требованием: вывод ошибок и уведомлений

Как уже было сказано ранее, вызов compute может как вернуть некие сгенерированные в ходе обработки и исполнения WGSL кода уведомления, так и выкинуть критические ошибки, тем самым прервав нормальный ход программы и ничего не вернув.

И ошибки, и уведомления пользователю необходимо показать не просто в консоли, а непосредственно в окне Web-браузера. Во-первых, пользователь может просто не догадаться открыть средства разработчика браузера и, без какой-либо видимой обратной связи, не суметь нормально воспользоваться платформой или узнать, что что-то пошло не

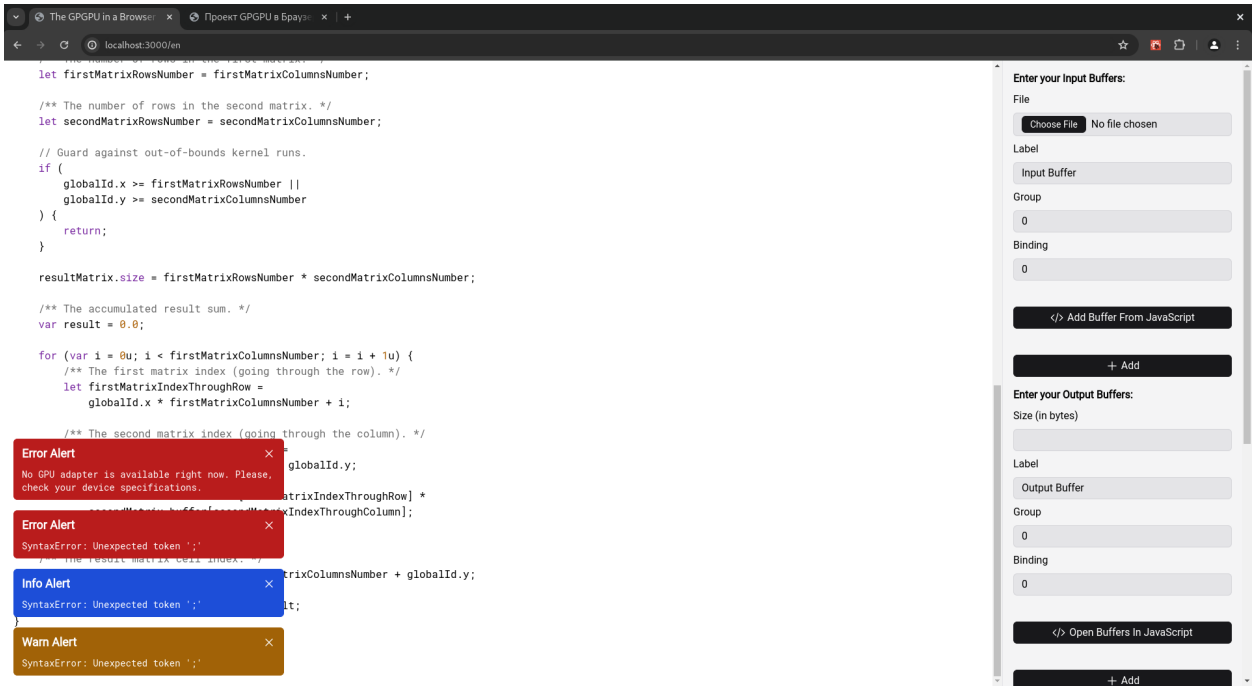
так. Во-вторых, сообщения в консоли могут затеряться, если браузер в свою очередь выведет туда какие-либо собственные уведомления, в то время как сообщения на экране будут контролироваться приложением и только приложением, что автоматически исключает данный недостаток.

Так, как я уже упоминал, ошибки из compute отлавливаются и помещаются в специальную единую очередь сообщений, которая выводится пользователю (и дублируется в консоль, если потребуется дополнительная информация в виде stack trace). То же самое происходит и с уведомлениями: те, которые возвращаются, также помещаются в эту очередь.

«Фронтенд», за счёт возможностей ReactJS, автоматически заново рендерит страницу, если эта очередь изменяется. Таким образом, пользователь всегда видит актуальную информацию у себя на экране.

Общий внешний вид ошибок и уведомлений в качестве примера представлен на Рисунке 15 ниже:

Рисунок 15 — Пример и внешний вид ошибок, выводимых пользователю на экран



При желании, пользователь может закрыть сообщение, нажав соответствующую иконку «крестик». Я решил не имплементировать автоматическое закрытие сообщений после выдержки времени, потому что это может навредить, если пользователь не успеет

прочитать сообщение, в то время как ему всегда предоставляется возможность самостоятельно и легко закрыть ненужные сообщения.

Отмечу, что некоторые ошибки, например об отсутствии WebGPU API, мы выбрасываем сами, а значит имеем возможность их перевести. Так, наши собственные ошибки обладают специальным именем-ключом, по которому они определяются в ходе обработки и переводятся на язык, выбранный пользователем.

Увы, без использования внешних сервисов мы не сможем что-то сделать в рамках нашей платформы с ошибками, создаваемыми браузерами, поэтому мы выводим их как есть. Но это не является какой-то значительной проблемой, ведь обычно сам браузер уже переводит эти ошибки на язык, выставленный в нём, то есть можно спокойно предположить, что пользователь их поймёт.

8. Работа над требованием: использование JavaScript для ввода и вывода данных

Возможность ввода данных во входных буферах и вывода данных из выходных буферов с использованием языка JavaScript изначально в проекте не предполагалась, но в итоге стала одним из основных требований, так как создание бинарных буферов вне приложения — довольно трудоёмкое занятие, требующее использование дополнительных инструментов. Стоит отметить, что такая возможность сохраняется (то есть пользователь всё ещё может загрузить бинарный файл с компьютера и загрузить созданный выходной файл прямо из приложения).

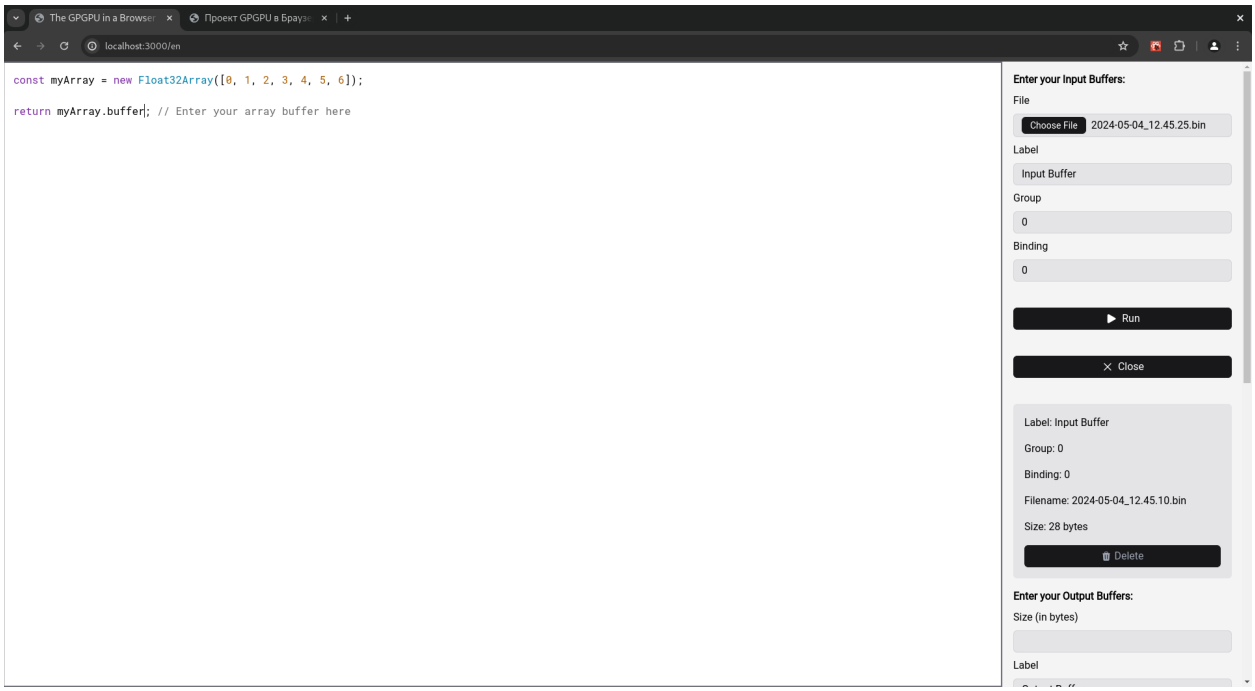
JavaScript был выбран для ввода буферов вместо создания какого-либо интерфейса по той причине, что входные данные могут принимать самую разнообразную форму, иметь разный тип данных или даже комбинировать несколько типов данных в один буфер, так что подобный интерфейс был бы слишком громоздок. К тому же, стоит учитывать, что пользователь может не просто захотеть ввести данные, а как-либо сгенерировать их, что JavaScript делает возможным.

Аналогично, с выходными данными перед тем, как их вывести в консоль, пользователь может захотеть произвести какие-либо последующие действия, например если типов данных несколько, то вывести их раздельно, так что логично, что для вывода данных также используется интерфейс в виде JavaScript.

В приложении этот интерфейс использует тот же редактор, что использовался и для ввода WGSL кода, то есть необходимости в каких-либо новых методах для данных целей просто нет. Само исполнение JavaScript кода использует встроенную в JavaScript функцию Function, которая в сочетании с использованием HTML тега iframe позволяет изолировать код пользователя от основного кода программы, чтобы ход исполнения самого приложения не мог быть случайно прерван.

В качестве примера, проиллюстрируем возможность ввода данных на Рисунке 16:

Рисунок 16 — Демонстрация возможности ввода данных во входной буфер при помощи JavaScript



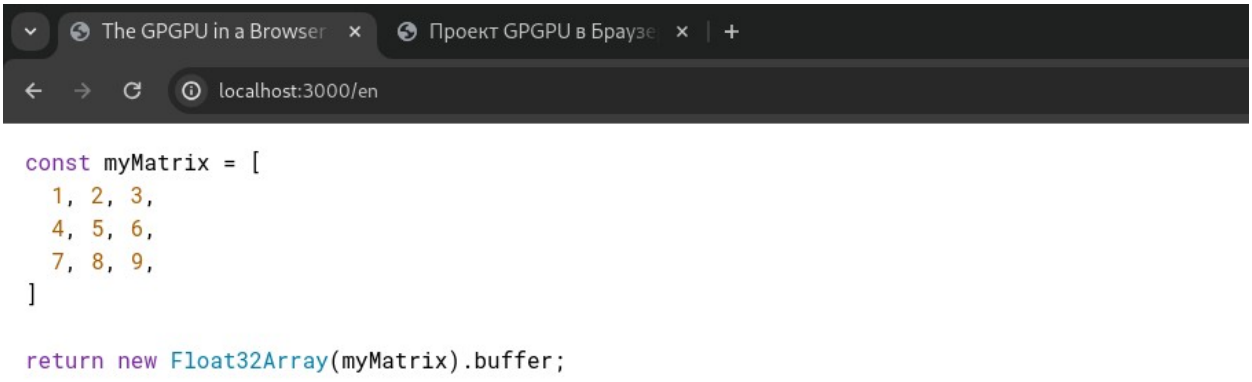
На этом обзор процесса и деталей создания приложения можно считать завершённым. В следующей главе будет продемонстрирована работа в программе от лица пользователя, чтобы показать наглядно простоту всех действий в ней.

ГЛАВА 4: ДЕМОНСТРАЦИЯ РАБОТЫ ПЛАТФОРМЫ

1. Демонстрация работы в платформе

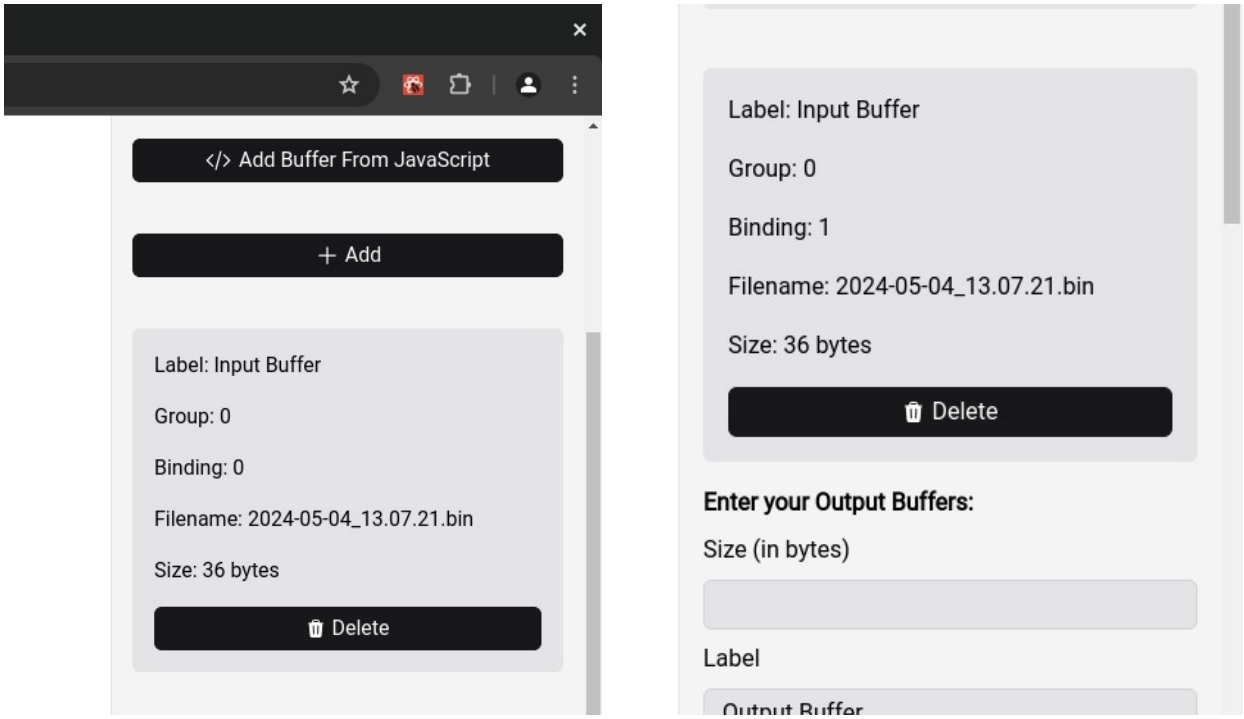
Представим, что мы — пользователь, который хотел бы попрактиковаться в направлении GPGPU или же, например, продемонстрировать студентам GPGPU вычисления. Мы решили, что в качестве нашей учебной задачи мы бы хотели перемножить две квадратные матрицы размера три на три, но обязательно с использованием GPU. Переходим на сайт и с использованием JavaScript создадим нашу матрицу, как показано на Рисунке 17:

Рисунок 17 — Демонстрация ввода матрицы во входной буфер при помощи JavaScript



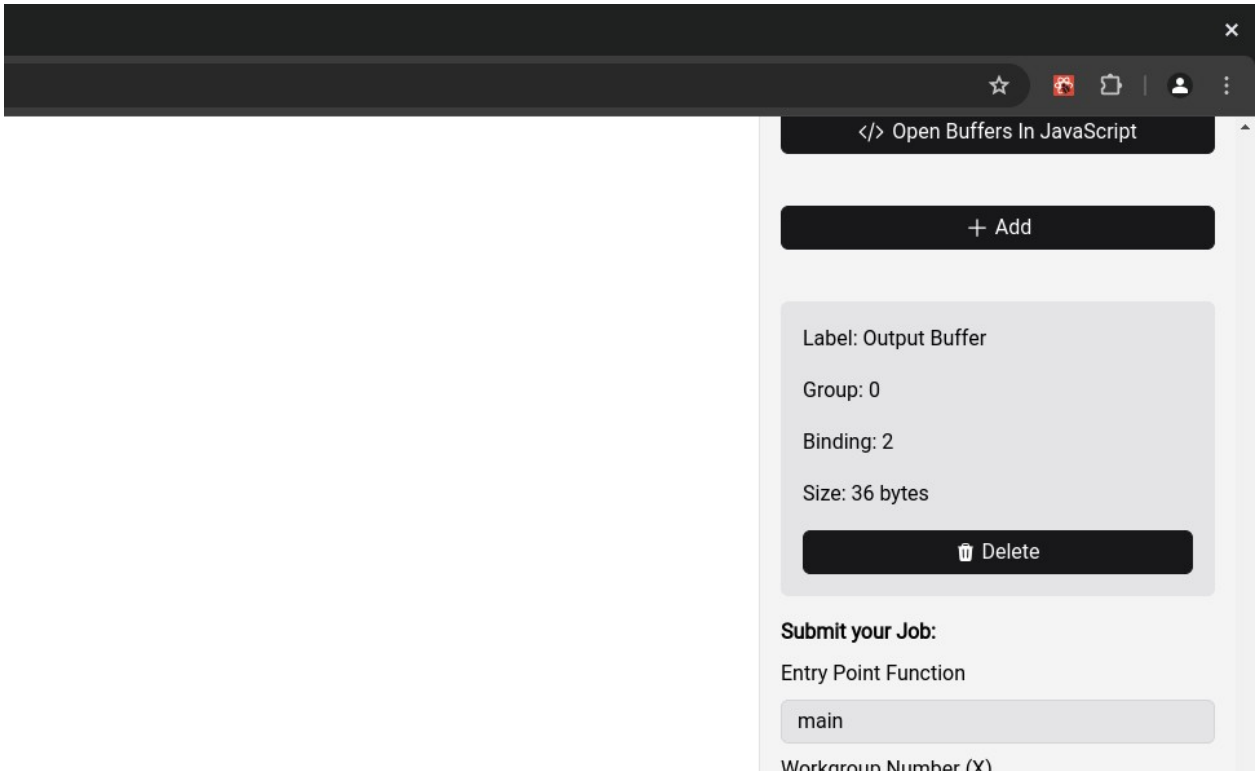
Добавим этот буфер как нулевое и первое связывание, что будет потом использовано в WGSL коде. Получим следующие входные буферы, показанные на Рисунке 18:

Рисунок 18 — Демонстрация входных буферов для примера



Далее, создадим выходной буфер размера 36 байт (так как на выходе получим массив из девяти элементов типа float32), и поставим ему в соответствие второе связывание. В итоге получим буфер с Рисунка 19:

Рисунок 19 — Демонстрация выходного буфера для примера

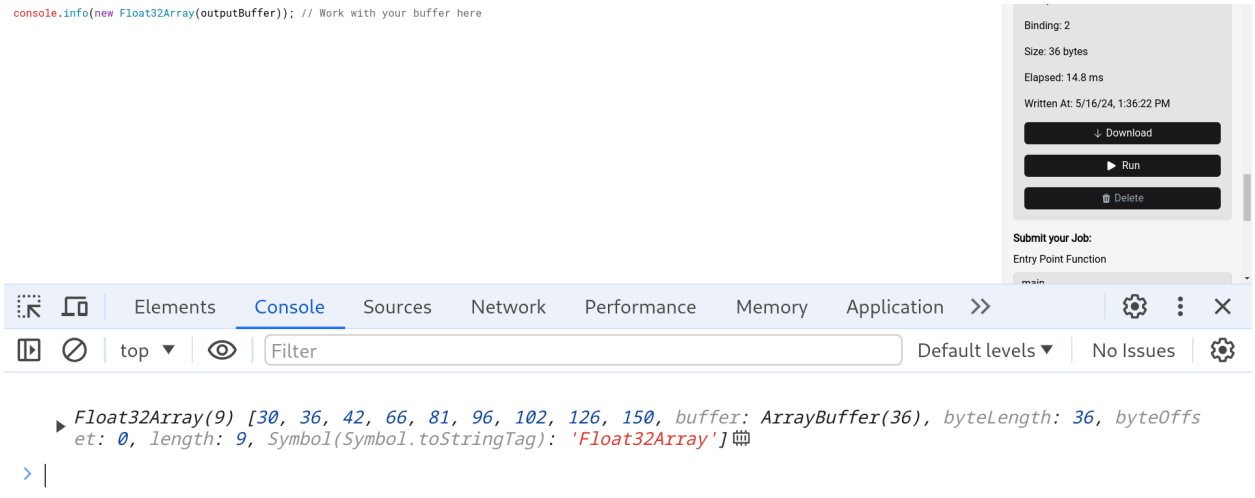


Теперь осталось ввести код WGSL программы и запустить её. Сделаем это, и увидим, что после запуска у выходного буфера появилась возможность загрузить его и время его записи, что продемонстрировано на Рисунке 20:

Рисунок 20 — Демонстрация приложения после запуска WGSL программы

Но мы не хотим загружать буфер, нам было бы достаточно увидеть результат. Что мы и сделаем, опять с использованием JavaScript. Вывод буфера в консоль представлен на Рисунке 21:

Рисунок 21 — Демонстрация результата работы приложения



Как можно видеть, результат действительно правдив, и он был получен целиком и полностью с использованием GPU. Всё легко и просто!

2. Выводы по работе в платформе

Приложение действительно выполняет те цели, которые были перед ним поставлены. Можно смело полагать, что начать работать в нём и практиковать или демонстрировать GPGPU с ним легче, чем с традиционными OpenCL и CUDA:

- Начать работу с ним можно сразу же, зайдя в Web-браузер. В отличии от него, те же OpenCL и CUDA требуют установки громоздких программных пакетов, на которую уйдёт не только от двух часов времени (в зависимости от скорости Интернета и чтения пользовательских соглашений и так далее), но и много впустую потраченных усилий и места на жёстком диске. Особенно это важно для демонстраций GPGPU, ведь она может происходить из сред, где установка дополнительного программного обеспечения запрещена.

- После установки соответствующего программного обеспечения, на изучение основ OpenCL, например, по собственному опыту, уйдёт два-три часа; к тому же, придётся разобраться с сопутствующим программным обеспечением (в моём случае это был пакет pyopencl), на что уйдёт примерно час. Стоит помнить, что это время прикладывается ко времени, необходимому на установку и настройку, если у пользователя вообще останется мотивация что-то изучать после них.

- В свою очередь, на изучение WGSL с нуля, по моему собственному опыту, будет вполне достаточно тех же двух-трёх часов; например, я начинал свой путь как раз со статьи про перемножение матриц [35], к которой я не раз в своём коде оставляю ссылки. Она простая и понятная, особенно для тех, у которых есть базовые навыки работы с JavaScript, а учитывая, что большинство «склеивающего» кода приложение берёт на себя, получить результаты в течении указанного срока вполне реально.

НЕДОСТАТКИ ПЛАТФОРМЫ

Безусловно, всё не может быть абсолютно идеально, и с созданным приложением это не исключение. Отмечу, что следующая критика приложения — не повод «забраковать» его, а скорее мотивация на доработку недочётов.

Так, одним из ключевых недостатков, которые я вижу, является необходимость использования JavaScript для ввода и вывода буферов в окне приложения. В современных Web-браузерах поддерживается такая Web-технология, как WebAssembly [21], которая позволяет исполнять код на других языках программирования, помимо JavaScript, в среде Web-браузера, за счёт предварительной компиляции такого кода в специальный, похожий на Assembler язык. Следовательно, этот момент вполне можно рассмотреть и принять решение, реалистична ли мульти-языковая система на данный момент.

Также важным моментом является то, что WebGPU ещё поддерживается далеко не везде. Безусловно, это новая технология, которая ещё толком не «прижилась» в Web-браузерах и находится в активной разработке, но всё же сегодня поддержка довольно слабая, что однозначно сказывается на приложении. Так, это не идеально для демонстрации GPGPU — никто не хочет перед аудиторией увидеть сообщение о том, что WebGPU не поддерживается их Web-браузером. К тому же, в текущей версии приложения уведомление об отсутствии поддержки возникает только после первого запуска программы, что не идеально для пользователя, ведь он уже мог подготовить WGSL код для запуска, который, при отсутствии другого Web-браузера с поддержкой WebGPU, опять возвращает нас к проблеме установки и настройки.

Как небольшое и довольно легко исправимое замечание можно отметить отсутствие возможности сохранения изменений в выходном буфере из JavaScript, то есть без специальных знаний о загрузке файлов с использованием JavaScript выходной буфер может быть только выведен в консоль.

Также, хотелось бы, чтобы приложение в целом было более активно в плане действий пользователя. Так, не хватает уведомлений (возможно со звуком) об успешном завершении исполнения WGSL кода и успешном запуске JavaScript кода. Без них в настоящее время пользователю может быть не понятно, успешно ли прошла та или иная операция, так как отсутствие уведомлений не всегда говорит об обратном.

Чтобы подытожить приложение в плане его недостатков, хотелось бы ещё раз сделать акцент на том, что перечисленные недостатки либо исправимы собственными силами, либо будут снижаться в значимости со временем (так, поддержка WebGPU будет только расти); следовательно, цели самого приложения этими недочётами не затрагивается, и они всё ещё остаются перспективными.

ЗАКЛЮЧЕНИЕ

В ходе выпускной квалификационной работы мы поставили себе цель создать виртуализированную (в плане абстракции от аппаратного обеспечения) за счёт средств Web-браузера платформу, на которой можно было бы практиковаться работе с GPGPU или же продемонстрировать работу с GPGPU, не прибегая к установке и настройке больших программных продуктов.

Можно смело утверждать, что с задачей мы справились успешно: в результате работы было сделано обладающее ключевым функционалом и готовое к расширению приложение, которое выполняет все предъявленные требования. Также можно отметить, что приложение обладает автоматическими unit-тестами и внутренней документации в виде комментариев, по которым можно сгенерировать внешнюю документацию в виде Web-страниц — это может помочь сторонним разработчикам быстрее включиться в работу с приложением, а также на протяжении всего периода разработки улучшать его качество.

Безусловно, имеются направления, которых работа может развиваться далее. Так, в будущем, когда инструменты для работы с WebGPU будут более развитыми (или обладая достаточными ресурсами для создания подобных инструментов самостоятельно) можно вновь вернуться к вопросу о расширении понятия виртуализации, чтобы платформа поддерживала не только WGSL, а, скажем, также и OpenCL — это было бы ещё более полезно с практической точки зрения, так как такие платформы куда более популярны, чем тот же WGSL.

Также можно доработать платформу до полноценной платформы GPGPU, а не только ограничившись теми целями, которые мы перед собой поставили. Для этого нужно, например, сделать поддержку настройки пределов использования памяти, поддержку нескольких WGSL программ и создания связей между ними, их полноценное сохранение. Это определённо потребует времени и, скорее всего, значительной переработки хотя бы интерфейса приложения.

Можно также предложить развивать подобные платформы для каких-нибудь специфичных GPGPU-задач, например для популярной сегодня области машинного обучения, то есть вместо абстрактного интерфейса придумать, как оптимизировать его под конкретную область, и предоставить API для удобства работы в этой области.

Отмечу, что всем этим направлениям ещё предстоит пережить пик интереса к ним, скорее всего это произойдёт в тот момент, когда WebGPU и WGSL стабилизируются и окончательно станут частью любого современного Web-браузера (то есть где-то через 5-10 лет).

Основа для этих направлений деятельности, я надеюсь, была создана уже сегодня, вместе с этой выпускной квалификационной работой. При желании, с текущей версией проекта всегда можно будет ознакомиться по следующей ссылке: https://github.com/gpu-browser-project/gpu-browser-project. Надеюсь, что он окажется полезен для тех, кто хотел бы развивать данную тему в будущем. Также код, предоставленный в Приложениях, может оказаться полезен для повторения проведённых экспериментов в будущем с целью обзора улучшений WebGPU и WGSL в плане производительности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 NVIDIA Corporation. CUDA Toolkit — Free Tools and Training, 2024. — URL: https://developer.nvidia.com/cuda-toolkit (дата обращения 2024-05-18).

2 The Khronos Group. OpenCL — The Open Standard for Parallel Programming of Heterogeneous Systems, 2024. — URL: https://khronos.org/opencl (дата обращения 2024-05-18).

3 Dakkak A., Pearson C., Hwu W.-M. WebGPU: A Scalable Online Development Platform for GPU Programming Courses, 2016 // 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, P. 942-949. doi:10.1109/IPDPSW.2016.63

4 Valdez, A.A.M., Wee, F., Odasco, A.N.L. et al. GPU simulations of spiking neural P systems on modern web browsers, 2023 // Nat Comput 22, P. 171–180. doi:10.1007/s11047-022-09914-1

5 J. Galaz, R. Cienfuegos, A. Echeverría, S. Pereira, C. Bertín, G. Prato, J.C. Karich. Integrating tsunami simulations in web applications using BROWNI, an open source client-side GPU-powered tsunami simulation library, 2022 // Computers & Geosciences, Volume 159, 104976. doi:10.1016/j.cageo.2021.104976

6 The Khronos Group. WebGL Overview, 2024. — URL: https://khronos.org/webgl (дата обращения 2024-05-18).

7 World Wide Web Consortium. WebGPU, 2024. — URL: https://w3.org/TR/webgpu (дата обращения 2024-05-18).

8 Nam, Hyunwoo and Park, Neungsoo. Accelerating AES Algorithm using WebGPU, 2022 // The Transactions of The Korean Institute of Electrical Engineers 71(7):1008-1014. doi:10.5370/KIEE.2022.71.7.1008

9 Satyadhyan Chickerur, Sankalp Balannavar, Pranali Hongekar, Aditi Prerna, Soumya Jituri. WebGL vs. WebGPU: A Performance Analysis for Web 3.0, 2024 // Procedia Computer Science, Volume 233, P. 919-928. doi:10.1016/j.procs.2024.03.281

10 Aldahir Abdulsalam. Evaluation of the performance of WebGPU in a cluster of web-browsers for scientific computing, 2022. — URL: https://umu.diva-portal.org/smash/record.jsf?pid=diva2%3A1674447 (дата обращения 2023-12-24).

11 Калукова Ольга, Кошелева Наталья, Никитина Марина, Павлова Елена. Курс лекций по высшей математике, 2006. — URL: https://edu.tltsu.ru/er/book_view.php?book_id=1e2&page_id=499 (дата обращения 2023-12-24).

12 The Khronos Group. WebCL, 2023. — URL: https://khronos.org/webcl (дата обращения 2023-12-24).
13 World Wide Web Consortium. WebGPU Shading Language, 2023. — URL: https://w3.org/TR/WGSL (дата обращения 2023-12-24).
14 The Khronos Group. SPIR, 2023. — URL: https://khronos.org/spir (дата обращения 2023-12-24).
15 The Clang Authors. Clang: a C language family frontend for LLVM, 2023. — URL: https://clang.llvm.org/ (дата обращения 2023-12-24).
16 The Khronos Group. LLVM/SPIR-V Bi-Directional Translator, 2023. — URL: https://github.com/KhronosGroup/SPIRV-LLVM-Translator (дата обращения 2023-12-24).
17 The Clspv Authors. Clspv, 2023. — URL: https://github.com/google/clspv (дата обращения 2023-12-24).
18 The Khronos Group. Offline Compilation of OpenCL Kernels into SPIR-V Using Open Source Tooling, 2020. — URL: https://khronos.org/blog/offline-compilation-of-opencl-kernels-into-spir-v-using-open-source-tooling (дата обращения 2023-12-24).
19 The Tint Team. Tint, 2023. — URL: https://dawn.googlesource.com/tint (дата обращения 2023-12-24).
20 The gfx-rs Developers. wgpu, 2023. — URL: https://github.com/gfx-rs/wgpu (дата обращения 2023-12-24).
21 W3C Community Group. WebAssembly Specification, 2024. — URL: https://webassembly.github.io/spec/core (дата обращения 2024-05-18).
22 The Yarn Contributors. Yarn, 2024. — URL: https://yarnpkg.com (дата обращения 2024-05-18).
23 Vercel. Next.js by Vercel — The React Framework, 2024. — URL: https://nextjs.org (дата обращения 2024-05-18).
24 Contributors to the Jest project. Jest · Delightful JavaScript Testing, 2024. — URL: https://jestjs.io (дата обращения 2024-05-18).
25 The highlight.js Team. highlight.js, 2024. — URL: https://highlightjs.org (дата обращения 2024-05-18).
26 Microsoft. TypeScript: JavaScript With Syntax For Types, 2024. — URL: https://typescriptlang.org (дата обращения 2024-05-18).
27 OpenJS Foundation and Other Contributors. ESLint - Find and fix problems in your JavaScript code, 2024. — URL: https://eslint.org (дата обращения 2024-05-18).

28 James Long and the Prettier Contributors. Prettier · Opinionated Code Formatter, 2024. — URL: https://prettier.io (дата обращения 2024-05-18).

29 Hayden Bleasel. Making react-syntax-highlighter "editable", 2023. — URL: https://haydenbleasel.com/blog/making-react-syntax-highlighter-editable (дата обращения 2024-05-18).

30 The React Contributors. React, 2024. — URL: https://react.dev (дата обращения 2024-05-18).

31 Martins, Beatriz and Duarte, Carlos. A large-scale web accessibility analysis considering technology adoption, 2023 // Universal Access in the Information Society. 1-16. 10.1007/s10209-023-01010-0

32 World Wide Web Consortium. HTML5, 2011. — URL: https://w3.org/TR/2011/WD-html5-20110405 (дата обращения 2024-05-18).

33 World Wide Web Consortium. Accessible Rich Internet Applications (WAI-ARIA) 1.2, 2023. — URL: https://w3.org/TR/wai-aria (дата обращения 2024-05-18).

34 Brandon Jones. WebGPU Error Handling best practices, 2024. — URL: https://toji.dev/webgpu-best-practices/error-handling.html (дата обращения 2024-05-18).

35 François Beaufort. Get started with GPU Compute on the web, 2019. — URL: https://web.dev/gpu-compute (дата обращения 2024-05-18).

Приложения

1. Приложение А: HTML и JavaScript-код для эксперимента

```
<!-- Following the article: https://web.dev/gpu-compute. -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>WebGPU Matrix Multiplication Benchmark</title>
    <link rel="icon" href="data:;base64,iVBORw0KGgo=" />
  </head>
  <body>
    <form id="matrices" method="post">
      <label>
        The generated matrices file:
        <div>
          <input accept=".bin" name="data" type="file" />
        </div>
      </label>
      <div>
        <button>Submit</button>
      </div>
    </form>
    <script>
      /** The form where matrices are gotten from. */
      const matrices = document.getElementById("matrices");

      /**
       * Loads the matrices from `file`.
       * @param file The file to load from.
       * @returns The result matrices.
       */
      const loadMatricesFromFile = async (file) => {
        /** The buffer with the file contents. */
        const buffer = await file.arrayBuffer();

        /** The data view object for reading data. */
        const dataView = new DataView(buffer);

        /** The number of matrix rows and columns. */
        const size = dataView.getUint32(0, true);

        /** The first matrix values. */
        const firstValues = [];

        /** The second matrix values. */
        const secondValues = [];

        /** The index of the data view "pointer". */
        let index = Uint32Array.BYTES_PER_ELEMENT;

        // Get the result values as per the format.
        while (
          index <
          Uint32Array.BYTES_PER_ELEMENT +
            Float64Array.BYTES_PER_ELEMENT * 1 * size ** 2
```

```
  ) {
    firstValues.push(dataView.getFloat64(index, true));
    index += Float64Array.BYTES_PER_ELEMENT;
  }

  while (
    index <
    Uint32Array.BYTES_PER_ELEMENT +
      Float64Array.BYTES_PER_ELEMENT * 2 * size ** 2
  ) {
    secondValues.push(dataView.getFloat64(index, true));
    index += Float64Array.BYTES_PER_ELEMENT;
  }

  // Return the result in a normal `js` types.
  return { size, firstValues, secondValues };
};

/** Multiplies two matrices. */
const multiplyMatrices = async ({
  benchmark,
  firstValues,
  secondValues,
  size,
}) => {
  /** The index of a number of rows in a buffer. */
  const rowIndex = 0;

  /** The index of a number of columns in a buffer. */
  const colIndex = 1;

  /** The workgroup `x` dimension size. */
  const workgroupSizeX = 8;

  /** The workgroup `y` dimension size. */
  const workgroupSizeY = 8;

  const firstMatrix = new Float32Array([size, size, ...firstValues]);
  const secondMatrix = new Float32Array([size, size, ...secondValues]);

  if (!"gpu" in navigator) {
    throw new Error("no WebGPU API available");
  }

  /** The graphics card, integrated or discrete. */
  const adapter = await navigator.gpu.requestAdapter({
    powerPreference: "high-performance",
  });

  if (!adapter) {
    throw new Error("no GPU adapter available");
  }

  /** The device that is used for computations. */
  const device = await adapter.requestDevice({
    requiredFeatures: [],
    requiredLimits: {},
  });

  const resultMatrixBufferSize =
    Float32Array.BYTES_PER_ELEMENT *
```

```
          (firstMatrix[rowIndex] * secondMatrix[colIndex] + 2);

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

        const secondMatrixGPUBuffer = device.createBuffer({
          mappedAtCreation: true,
          size: secondMatrix.byteLength,
          usage: GPUBufferUsage.STORAGE,
        });

        const resultMatrixGPUBuffer = device.createBuffer({
          size: resultMatrixBufferSize,
          usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC,
        });

        // Write the matrices to the buffers.
        const firstMatrixArrayBuffer = firstMatrixGPUBuffer.getMappedRange();
        const secondMatrixArrayBuffer =
secondMatrixGPUBuffer.getMappedRange();

        new Float32Array(firstMatrixArrayBuffer).set(firstMatrix);
        new Float32Array(secondMatrixArrayBuffer).set(secondMatrix);

        // Give the buffers to the GPU.
        firstMatrixGPUBuffer.unmap();
        secondMatrixGPUBuffer.unmap();

        /** I/O interface expected by the shader. Concept specific to WebGPU.
*/
        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" },
            },
          ],
        });

        /** Actual I/O for the shader. Concept specific to WebGPU. */
        const bindGroup = device.createBindGroup({
          layout: bindGroupLayout,
          entries: [
            {
              binding: 0,
              resource: { buffer: firstMatrixGPUBuffer },
            },
```

```
    {
      binding: 1,
      resource: { buffer: secondMatrixGPUBuffer },
    },
    {
      binding: 2,
      resource: { buffer: resultMatrixGPUBuffer },
    },
  ],
});

const shaderModule = device.createShaderModule({
  // Shader that is written in WGSL, a standard
  // shader language for WebGPU.
  code: `
    /** The matrix. */
    struct Matrix {
      /** The numbers of rows and columns. */
      size: vec2<f32>,

      /** The floating point number buffer. */
      buffer: array<f32>,
    }

    /** The first matrix. */
    @group(0)
    @binding(0)
    var<storage, read> firstMatrix: Matrix;

    /** The second matrix. */
    @group(0)
    @binding(1)
    var<storage, read> secondMatrix: Matrix;

    /** The result matrix. */
    @group(0)
    @binding(2)
    var<storage, read_write> resultMatrix: Matrix;

    /**
     * Multiplies two rectangular matrices.
     * @param globalId The position in the grid.
     */
    @compute
    @workgroup_size(
      ${workgroupSizeX},
      ${workgroupSizeY},
      // \`z\` is assumed \`1\`.
    )
    fn mulmat(@builtin(global_invocation_id) globalId : vec3<u32>) {
      // 0 <= x, y, z < workgroup_size_{coord} * group_count_{coord}

      // Guard against out-of-bounds kernel runs.
      if (
        globalId.x >= u32(firstMatrix.size.x) ||
        globalId.y >= u32(secondMatrix.size.y)
      ) {
        return;
      }

      resultMatrix.size = vec2(
```

```
        firstMatrix.size.x,
        secondMatrix.size.y,
      );

      /** The accumulated result sum. */
      var result = 0.0;

      for (var i = 0u; i < u32(firstMatrix.size.y); i = i + 1u) {
        /** The first matrix index (going through the row). */
        let firstMatrixIndexThroughRow =
          globalId.x * u32(firstMatrix.size.y) + i;

        /** The second matrix index (going through the column). */
        let secondMatrixIndexThroughCol =
          globalId.y + i * u32(secondMatrix.size.y);

        result +=
          firstMatrix.buffer[firstMatrixIndexThroughRow] *
          secondMatrix.buffer[secondMatrixIndexThroughCol];
      }

      /** The result matrix cell index. */
      let resultIndex = globalId.y +
        globalId.x * u32(secondMatrix.size.y);

      resultMatrix.buffer[resultIndex] = result;
    }
  `,
});

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

const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginComputePass();

passEncoder.setPipeline(computePipeline);
passEncoder.setBindGroup(0, bindGroup);

/** `x` dimension of the grid of workgroups to dispatch. */
const workgroupCountX = Math.ceil(
  firstMatrix[rowIndex] / workgroupSizeX
);

/** `y` dimension of the grid of workgroups to dispatch. */
const workgroupCountY = Math.ceil(
  secondMatrix[colIndex] / workgroupSizeY
);

passEncoder.dispatchWorkgroups(
  workgroupCountX,
  workgroupCountY
  // `z` is assumed `1`.
);
```

```
  passEncoder.end();

  const readingCopyGPUBuffer = device.createBuffer({
    size: resultMatrixBufferSize,
    usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.MAP_READ,
  });

  commandEncoder.copyBufferToBuffer(
    // Source buffer.
    resultMatrixGPUBuffer,
    // Source offset.
    0,
    // Destination buffer.
    readingCopyGPUBuffer,
    // Destination offset.
    0,
    // Copy size.
    resultMatrixBufferSize
  );

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

  await readingCopyGPUBuffer.mapAsync(GPUMapMode.READ);
  const readingCopyArrayBuffer = readingCopyGPUBuffer.getMappedRange();

  console.info(`result: ${new Float32Array(readingCopyArrayBuffer)}`);
  console.info(`benchmark: ${performance.now() - benchmark}ms`);
};

const multiplySequential = async ({
  benchmark,
  firstValues,
  secondValues,
  size,
}) => {
  const result = [...firstValues];

  for (let i = 0; i < size; ++i) {
    for (let j = 0; j < size; ++j) {
      result[i * size + j] = 0;

      for (let k = 0; k < size; ++k) {
        result[i * size + j] +=
          firstValues[i * size + k] * secondValues[k * size + j];
      }
    }
  }

  console.info(`result: ${result}`);
  console.info(`benchmark: ${performance.now() - benchmark}ms`);
};

// Recieve a file with our matrices.
matrices.addEventListener("submit", async (event) => {
  event.preventDefault();

  /** The file from the form. */
  const file = new FormData(event.target).get("data");
```

```
        // Do not continue without any data.
        if (file.size === 0) {
          console.error("No file has been sent");
          return;
        }

        const recieved = await loadMatricesFromFile(file);

        await multiplySequential({
          ...recieved,
          benchmark: performance.now(),
        });

        await multiplyMatrices({
          ...recieved,
          benchmark: performance.now(),
        });
      });
    </script>
  </body>
</html>
```

2. Приложение Б: OpenCL-код для эксперимента

```
// Following the article: https://web.dev/gpu-compute.

/**
 * Multiplies two rectangular matrices.
 * @param firstMatrixRows  The first matrix number of rows.
 * @param firstMatrixCols  The first matrix number of columns.
 * @param secondMatrixRows The second matrix number of columns.
 * @param firstMatrix      The first matrix floating point number buffer.
 * @param secondMatrix     The second matrix floating point number buffer.
 * @param resultMatrix     The result matrix floating point number buffer.
 */
kernel void mulmat(const uint firstMatrixRows,
                   const uint firstMatrixCols,
                   const uint secondMatrixRows,
                   global const float *firstMatrix,
                   global const float *secondMatrix,
                   global float *resultMatrix) {
  /** The `x` position in the grid. */
  size_t globalIdX = get_global_id(0u);

  /** The `y` position in the grid. */
  size_t globalIdY = get_global_id(1u);

  /** The accumulated cell result. */
  float result = 0.0f;

  // Guard against out-of-bounds kernel runs.
  if (globalIdX >= firstMatrixRows || globalIdY >= secondMatrixRows) {
    return;
  }

  for (size_t i = 0u; i < firstMatrixCols; ++i) {
    /** The first matrix index (going through the row). */
    size_t firstMatrixIndexThroughRow = globalIdX * firstMatrixCols + i;

    /** The second matrix index (going through the column). */
```

```
    size_t secondMatrixIndexThroughCol = globalIdY + i * secondMatrixRows;

    result += firstMatrix[firstMatrixIndexThroughRow] *
              secondMatrix[secondMatrixIndexThroughCol];
  }

  resultMatrix[globalIdX * secondMatrixRows + globalIdY] = result;
}
```

3. Приложение B: Связующий Python-код для эксперимента

```
#!/usr/bin/env python3

# 06: 1024, 07: 512, 08: 256, 09: 128

import numpy as np
import pathlib
import pyopencl as cl
import sys
import time

if __name__ == "__main__":
  with open(
    file=(sys.argv[1:] + ["generated-matrices.bin"])[0],
    mode="rb"
  ) as file:
    size = int.from_bytes(
      bytes=file.read(4),
      byteorder="little",
    )

    first_values = np.fromfile(
      file=file,
      count=size ** 2,
      dtype=np.float64,
    ).astype(dtype=np.float32).reshape((size, size))

    second_values = np.fromfile(
      file=file,
      count=size ** 2,
      dtype=np.float64,
    ).astype(dtype=np.float32).reshape((size, size))

    result_values = np.zeros(
      shape=(size, size),
      dtype=np.float32,
    )

  benchmark = time.time()
  platforms = cl.get_platforms()

  context = cl.Context(
    dev_type=cl.device_type.GPU,
    properties=[(cl.context_properties.PLATFORM, platforms[1])],
  )

  queue = cl.CommandQueue(
    context=context,
  )
```

```
  first_matrix = cl.Buffer(
    context=context,
    flags=cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR,
    hostbuf=first_values,
  )

  second_matrix = cl.Buffer(
    context=context,
    flags=cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR,
    hostbuf=second_values,
  )

  result_matrix = cl.Buffer(
    context=context,
    flags=cl.mem_flags.WRITE_ONLY,
    size=result_values.nbytes,
  )

  program = cl.Program(
    context,
    pathlib.Path("mulmat.cl").read_text(),
  ).build()

  program.mulmat(
    queue,
    (size, size),
    None,
    cl.cltypes.uint(size),
    cl.cltypes.uint(size),
    cl.cltypes.uint(size),
    first_matrix,
    second_matrix,
    result_matrix,
  )

  cl.enqueue_copy(
    queue=queue,
    dest=result_values,
    src=result_matrix,
  )

  print(f"devices: {context.devices}")
  print(f"result: {result_values}")
  print(f"benchmark: {time.time() - benchmark}s")
```

4. Приложение Г: Python-код для генерации матриц

```
#!/usr/bin/env python3
"""The generator of square matrices for multiplication.

The result matrices are put in the file `generated-matrices.bin` in the
same location as the script. It is binary because it makes it easier to
parse and allows to represent 64-bit floating-points exactly rather than
rounding them up as if when they are represented as strings.

The format is as follows: the first four bytes is a 32-bit integer `N`
which represents a number of rows and columns in a matrix, after that a
first matrix goes (`N ^ 2` 64-bit floating-point values), then a second
matrix goes (`N ^ 2` 64-bit floating-point values).
"""
```

```
import os
import numpy as np

# The path to the directory where this script is located.
SCRIPT_DIR = os.path.dirname(os.path.realpath(__file__))

# The size of a square matrix (`N` by `N`).
SIZE = 1024

# The range in which matrix values will lie.
RANGE = (-10.0, 10.0)

# The file in which newly generated matrices are placed.
PATH = os.path.join(SCRIPT_DIR, "generated-matrices.bin")

if __name__ == "__main__":
  rand = np.random.default_rng().uniform

  # First, we generate our first matrix values.
  first_values = rand(*RANGE, (SIZE, SIZE))

  # Then, we generate our second matrix values.
  second_values = rand(*RANGE, (SIZE, SIZE))

  # Then, we write everything to the file.
  with open(PATH, "wb") as file:
    file.write(SIZE.to_bytes(4, "little"))
    file.write(first_values.tobytes())
    file.write(second_values.tobytes())
```