Санкт-Петербургский государственный университет

Вологин Илья Олегович

Выпускная квалификационная работа

Поддержка исполнения моделей машинного обучения в виртуальных машинах JavaScript для платформы KInference

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2018 «Математическое обеспечение и администрирование информационных систем»*

Профиль *Системное программирование*

Научный руководитель:
доцент кафедры системного программирования, к. т. н., Т. А. Брыксин

Консультант:
программист ООО «Интеллиджей Лабс», В. Д. Танков

Рецензент:
программист ООО «Интеллиджей Лабс», В. И. Бибаев

Санкт-Петербург
2022

Saint Petersburg State University

Ilya Vologin

Bachelor's Thesis

Supporting the inference of machine learning models for JavaScript virtual machines in KInference

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2018 "Software and Administration of Information Systems"*

Profile: *System Programming*

Scientific supervisor:
C.Sc., docent, T. A. Bryksin

Consultant:
Software developer at "IntelliJ Labs Co Ltd", V. D. Tankov

Reviewer:
Software developer at "IntelliJ Labs Co Ltd", V. I. Bibaev

Saint Petersburg
2022

Оглавление

Введение

В последние годы в индустрии программного обеспечения всё большее внимание уделяется теме машинного обучения и, в частности, современным методам глубинного обучения, основанным на нейронных сетях. Модели машинного обучения находят применение во всем спектре разрабатываемых приложений, не исключая и сверхпопулярный сектор веб-приложений.

На данный момент существует два концептуальных подхода к интеграции моделей машинного обучения в веб-приложения. Первый подход предполагает удаленное исполнение моделей машинного обучения: необходимые данные из веб-приложения на удалённый сервер отправляются, далее вычисляется результат работы модели, и результат пересылается обратно пользователю. Данный способ имеет ряд недостатков: зависимость от скорости интернета пользователя, необходимость обеспечения конфиденциальности данных и затраты на содержание серверов для запуска моделей.

Второй подход заключается в исполнении модели машинного обучения напрямую в среде исполнения веб-приложения, то есть в браузере пользователя. Данный подход исключает все недостатки предыдущего, но при нем производительность модели зависит от вычислительных мощностей устройства пользователя. Тем не менее, для широкого спектра “простых” моделей, то есть моделей с небольшим числом операторов, этот вариант вполне подходит.

Вариант с локальным исполнением моделей машинного обучения неизбежно приводит к необходимости выбора системы описания и исполнения модели, способной работать в браузере. Существует большое количество библиотек для создания моделей машинного обучения, реализующих разные форматы их представления, и задача нахождения единой системы запуска для всех них далеко не тривиальна. К тому же необходимо, чтобы система исполнения была эффективна и обеспечивала приемлемую задержку, так как многие задачи требуют быстрого отклика.

Для решения проблемы поддержки многих форматов моделей был разработан универсальный формат Open Neural Network Exchange (ONNX)[1], в который можно конвертировать модели из различных форматов.

В настоящее время на языке Kotlin разрабатывается библиотека KInference[2]. Её основная цель — запуск моделей машинного обучения в формате ONNX. На данный момент KInference поддерживает исключительно JVM как среду исполнения, поддержка JS виртуальных машин — безусловно важная задача для развития данной библиотеки.

Постановка задачи

Целью данной работы является поддержка исполнения моделей машинного обучения в виртуальных машинах JavaScript для платформы KInference.

Для достижения поставленной цели требуется:

- реализовать возможность запуска платформы KInference в виртуальной машине JavaScript;
- провести обзор оптимизаций математических операций, доступных в браузере;
- провести апробацию оптимизаций математических операций, доступных в браузере;
- выбрать и реализовать наиболее оптимальные варианты оптимизаций;
- провести апробацию конечного решения.

[1]Официальный сайт ONNX – https://onnx.ai/

[2]Репозиторий KInference – https://github.com/JetBrains-Research/kinference

1. Обзор

Для того чтобы лучше понимать, как именно для платформы KInference должно быть поддержано исполнение в виртуальных машинах JavaScript, далее рассмотрим необходимый контекст:

- формат ONNX;
- существующие решения, поддерживающие ONNX;
- существующие решения для запуска моделей в браузерах;
- библиотека KInference;
- технология Kotlin Multiplatform;
- существующие браузерерные оптимизации.

1.1. ONNX

Open Neural Network Exchange (ONNX) — универсальный формат представления моделей машинного обучения, призванный обеспечить переносимость моделей, обученных с помощью различных библиотек (например, PyTorch [14], TensorFlow [17], Keras[3] и т.п.), а также облегчить запуск данных моделей. Для ONNX разработано множество утилит, которые позволяют преобразовать модель из формата библиотеки для обучения в модель формата ONNX.

Модель в формате ONNX представляется в виде графа вычислений, где вершины — это операторы, исполняющие вычисления, а рёбра определяют последовательность передачи данных между операторами. Данный граф должен быть топологически отсортирован и не должен иметь циклов.

Так как формат ONNX разрабатывался для запуска моделей, полученных с помощью одной из платформ для обучения, то при преобразовании в него из базового формата конкретной платформы обучения

[3]Официальный сайт Keras – https://keras.io/

удаляются данные, которые были необходимы на этапе обучения, но не нужны для запуска модели. Это позволяет уменьшить объем потребляемых моделью ресурсов, включая объём занимаемой ею дисковой и оперативной памяти.

В настоящее время активно разрабатываются инструменты для запуска моделей машинного обучения, описанных с помощью формата ONNX.

Рассмотрим несколько таких инструментов.

- Библиотека ONNX Runtime[4]. Данная библиотека разрабатывается компанией Microsoft на языке C++ и предоставляет API для языков Python, C, C#, Java и JavaScript. Разработчики библиотеки предоставляют на выбор большое количество платформ, на которых может использоваться библиотека, а также выбор наиболее предпочтительного аппаратного ускорения.

- Библиотека NCNN[5]. Эта библиотека разрабатывается компанией Tencent на языке C++. Данная библиотека ориентирована для работы на мобильных устройствах: оптимизирована для работы с ARM процессорами, поддерживает мобильные графические устройства, а также не имеет дополнительных зависимостей, благодаря чему является легковесной.

- Библиотека MNN [11]. Данная библиотека разрабатывается компанией Alibaba на языке C++. Данная библиотека также ориентирована для работы на мобильных устройствах и имеет все те же преимущества, что и библиотека NCNN.

В проектах, запускаемых с помощью виртуальной машины JavaScript, можно использовать только библиотеку ONNX Runtime. Для запуска реализации на C++ в браузере используется технология WebAssembly[6] [3], она поддерживает все существующие операторы

[4]Репозиторий ONNXRuntime – https://github.com/microsoft/onnxruntime
[5]Репозиторий NCNN – https://github.com/Tencent/ncnn
[6]Бинарный формат инструкций, предназначенный для запуска в браузере

ONNX, но требует существенного объема дискового пространства и проблематично встраивается в веб-приложения. Также существует реализация, поддерживающая технологию WebGL[7] для вычисления модели на графическом процессоре пользователя, но она поддерживает малое количество операторов ONNX и находится в активной разработке.

1.2. TensorFlow.js

Одной из альтернатив для запуска моделей машинного обучения в браузерах является библиотека TensorFlow.js (TFJS) [18].

Данная библиотека предоставляет свои собственные форматы представления моделей: Layers Model и Graph Model:

- Модели в формате Layers Model можно создавать, обучать и запускать прямо в TFJS. В данный формат можно конвертировать модели из форматов, которые предоставляет библиотека для обучения Keras, а также можно конвертировать LayersModel в форматы Keras.

- Модели в формате GraphModel нельзя создавать в TFJS, в данный формат можно только конвертировать модель из форматов, которые предоставляет библиотека для обучения TensorFlow [17], и форматов Keras. Модели в данном формате можно только запускать в TFJS, при этом при конвертации происходят оптимизации графа вычислений, описанные в [8], что позволяет ускорить запуск модели. Однако формат Graph Model может не поддерживать некоторые возможности Layers Model, например рекуррентные нейронные сети.

Помимо вышеперечисленных форматов в данный момент ведётся активная разработка поддержки формата TFLite [16]. В данной работе

[7]WebGL – технология для выполнения вычислений на графическом процессоре в браузере. Ссылка на официальный сайт WebGL – https://www.khronos.org/webgl/

TFJS не рассматривается в качестве аналога для библиотек, запускающих модели в формате ONNX, так как не поддерживает формат ONNX.

TFJS поддерживает несколько реализаций для работы с моделями на различных устройствах в браузере:

- Стандартная реализация для работы с CPU на JavaScript;
- WebAssembly [3] реализация для работы с CPU;
- WebGL реализация для работы с GPU.

На данный момент активно разрабатывается WebGPU[8] реализация, но она ещё не имеет стабильной версии.

1.3. KInference

KInference — библиотека для запуска моделей машинного обучения, разрабатываемая в лаборатории методов машинного обучения в программной инженерии JetBrains Research. Данная библиотека написана на языке Kotlin и поддерживает универсальный формат моделей машинного обучения ONNX, а также имеет реализацию большого количества операторов. Также данная библиотека изначально разрабатывалась как легковесная, то есть имеющая малое число зависимостей и небольшой объем бинарного файла.

С помощью KInference можно запускать модели, построенные на архитектуре Transformer [1]: Generative Pre-trained Transformer (GPT) [7], Bidirectional Encoder Representations from Transformers (BERT) [2], и подобные; а также рекуррентные нейронные сети: Recurrent neural network (RNN), Long-short term memory (LSTM) [6] и Gated Recurrent Units (GRU) [4].

На данный момент библиотека KInference может использоваться только в JVM проектах, но уже в следующей главе мы обсудим возможность использования Kotlin Multiplatform для реализации поддержки JS.

[8]WebGPU – технология для выполнения вычислений на графическом процессоре в браузере. Официальный сайт спецификаций WebGPU – https://gpuweb.github.io/gpuweb/

В KInference применяются различные оптимизации, которые позволяют значительно ускорить запуск моделей. Основой этих оптимизаций является представление многомерных массивов в виде блоков подмассивов размера 512. При таком представлении виртуальная машина JVM достаточно хорошо оптимизирует работу с кэш-памятью процессора во время исполнения тензорных операций, что снижает количество обращений к оперативной памяти. В силу того, что данная оптимизация основывается на общих соображениях об устройстве современных процессоров и виртуальных машин, она может быть актуальна и для виртуальных машин JS.

1.4. Kotlin Multiplatform

Kotlin Multiplatform [13] — это возможность языка Kotlin, позволяющая переиспользовать один и тот же код для компиляции под разные платформы, например: Java Virtual Machine (JVM), JavaScript и Native (LLVM [9]).

Данная возможность основана на фундаментальном свойстве языка Kotlin — встроенной поддержке диалектов. Благодаря ей существует как Kotlin/Common — язык, который может быть скомпилирован под любую платформу, — так и, например, Kotlin/JS — надмножество Kotlin/Common, включающее в себя ключевое слово `dynamic`, доступное только в данном диалекте.

Kotlin Multiplatform — это технология, реализованная поверх языка Kotlin и его диалектов и позволяющая, в частности, объединять код на разных диалектах, написанный под разные платформы в рамках набора модулей. К примеру, в Kotlin Multiplatform можно создавать как общий код, подходящий для всех платформ, так и специфичный код под конкретную платформу. Например, так можно реализовать код для работы с файловой системой, доступной в Kotlin/JVM.

В Kotlin Multiplatform весь код делится на несколько модулей: основной модуль с общим кодом (commonMain) и модули с кодом для каждой платформы (например, для JavaScript это `jsMain`, а для JVM

это `jvmMain` и т.д.). Также в Kotlin Multiplatform поддерживается возможность создания иерархии из модулей и объединения общего кода для определённых платформ в один модуль. Это может быть удобно для объединения компиляций для одной платформы под разные архитектуры, например, на Рис. 1 модули `iosArm64Main` и `iosX64Main` объединены в модуль `iosMain`, так как они имеют общее API.

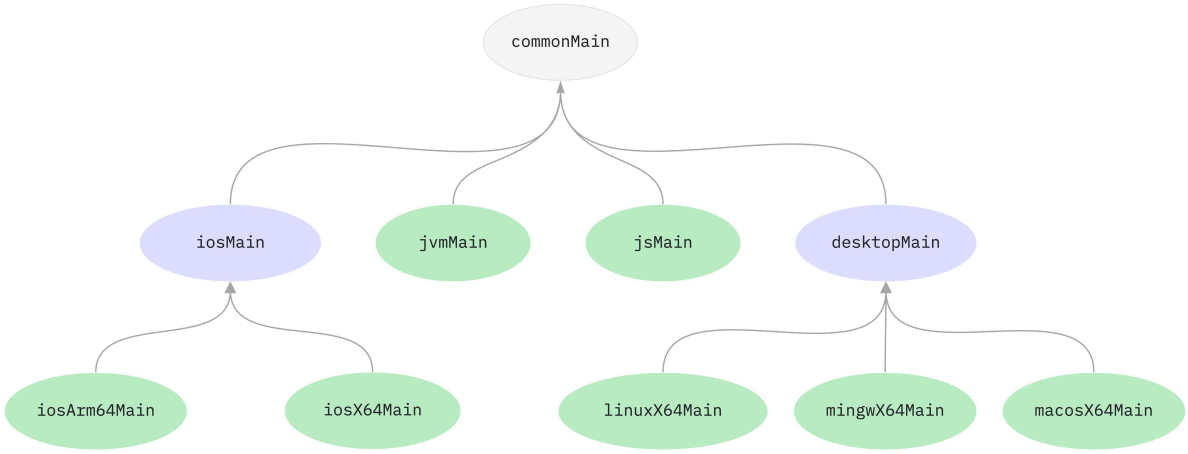

Рис. 1: Пример иерархической структуры в Kotlin Multiplatform (рисунок взят из [13])

Для использования API платформ в common модуле доступна expect/actual нотация. Принцип работы данной нотации довольно прост (см. Рис. 2). Изначально в основном модуле определяется функция или класс с ключевым словом expect, но не реализуются, далее для каждой платформы описывается реализация с ключевым словом actual.

1.5. Браузерные оптимизации

Существующие виртуальные машины JavaScript в большинстве своём не являются ориентированными на выполнение большого количества математических операций, необходимых для запуска моделей машинного обучения. С развитием браузеров появляется всё больше различных возможностей для увеличения производительности виртуальных машин JavaScript в области математических операций. Рассмотрим некоторые из них.

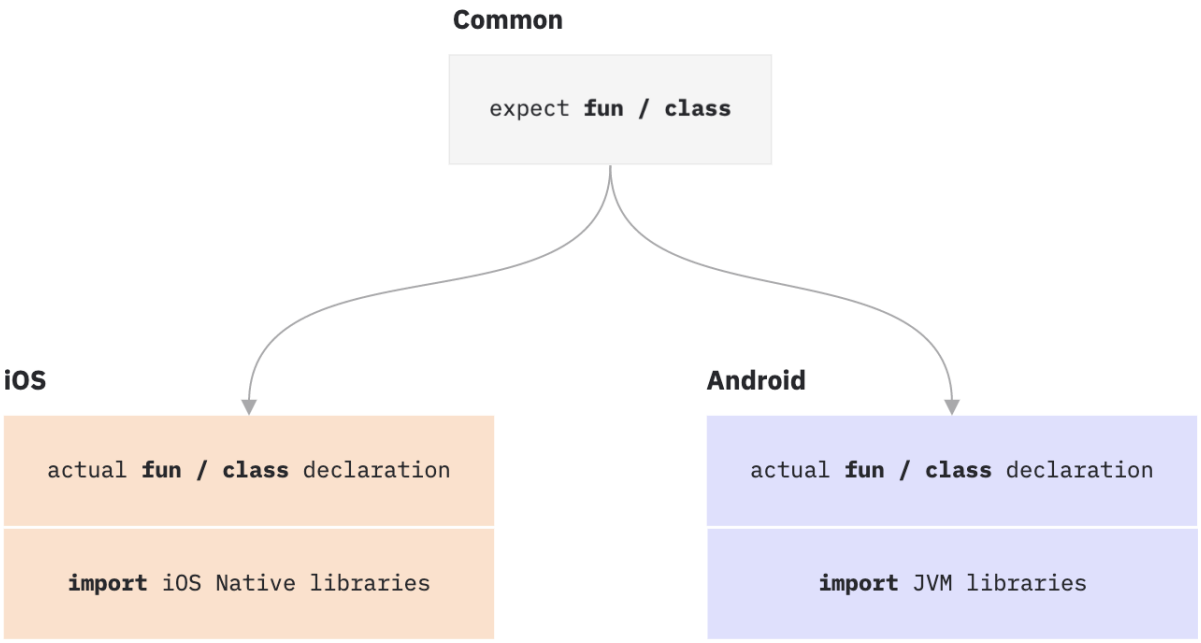

Рис. 2: Принцип работы expect/actual нотации (рисунок взят из [13]).

1.5.1. Веб-воркеры

Для ускорения программ зачастую применяется распараллеливание с помощью потоков или процессов. Оно позволяет сократить время выполнения программы за счёт задействования нескольких ядер процессора, способных выполнять работу параллельно. Однако JavaScript при исполнении в браузере является однопоточным языком программирования, выполняемым в рамках единственного процесса виртуальной машины, и в нём невозможно действительно параллельное исполнение. Для решения этой проблемы были придуманы веб-воркеры.

Веб-воркеры[9] (воркеры) — это потоки браузера, которые могут использоваться для фонового выполнения некоторой задачи. Воркеры создаются из основного потока браузера, а в конструктор воркера передается путь к файлу с JavaScript, скриптом с которого он и начнет исполнение кодом воркера.

Поскольку веб-воркеры — это отдельные потоки, а весь JavaScript однопоточный, работа с ними ведется в рамках акторной модели с помощью передачи сообщений. Это обеспечивает потоко-безопасность и,

[9]Официальный сайт спецификакции Web Workers — https://html.spec.whatwg.org/multipage/workers.html

в силу текущего устройства API JavaScript, воспринимается программистами как знакомая концепция.

Веб-воркеры могут успешно использоваться для распараллеливания, однако стоит учитывать, что при создании веб-воркеров могут создаваться потоки с помощью вызовов к операционной системе, что может занимать дополнительно время и не позволять использовать веб-воркеры для выполнения небольших задач.

1.5.2. WebAssembly

WebAssembly (WASM) — это бинарный формат описания инструкций (своего рода байт-код), исполняемый низкоуровневой виртуальной машиной и обеспечивающий крайне высокую производительность. На данный момент WASM поддерживается многими современными браузерами.

Одной из основных целей создания WASM была задача создания виртуальной машины, способной обеспечить сверх-высокую производительность в браузере. Учитывая интеграцию с LLVM и простоту инструкций WASM, он стал популярной платформой для генерации кода с различных компилируемых языков программирования. На данный момент WASM поддерживает большое количество языков программирования: C, C++, C#, Rust, Kotlin и многие другие.

Байт-код WebAssembly представляет из себя наборы инструкций, схожие с инструкциями процессора. Благодаря данному представлению байт-кода, скорость выполнения WASM программ значительно выше, чем в JavaScript и близка к машинному коду.

Более того, WebAssembly поддерживает SIMD [5] инструкции (SSE [15] и AVX [10] для процессоров на x86 архитектуре и NEON[10] для процессоров на ARM архитектуре), что, в свою очередь, позволяет повысить скорость выполнения программ. Стоит отметить, что векторизация обеспечивает кратный прирост производительности на математических задачах, что крайне актуально в свете цели данной работы.

[10]Ссылка на официальный сайт технологии NEON – https://www.arm.com/technologies/neon

Стоит отметить, что WASM программы не могут напрямую взаимодействовать с веб-API (такими как DOM[11], WebGL, IndexedDB[12] и т.д.). Несмотря на это, современные браузеры поддерживают импортирование и вызов JS функций в WASM программах, но данный подход накладывает некоторые ограничения на производительность WASM программ в связи с тем, что приходится вызывать интерпретатор JS. К тому же современные WASM компиляторы предоставляют различные возможности для использования JS функций в WASM программах и автоматически генерируют необходимый код на обоих языках.

1.5.3. GPU

Многие математические операции, необходимые для запуска моделей машинного обучения, выполняются быстрее на графических процессорах, которые есть практически на любом современном устройстве. Прирост в скорости получается за счёт большего числа вычислительных ядер, чем в центральных процессорах, и распараллеливания на уровне графического процессора.

Но для достаточно малых данных скорость выполнения на графическом процессоре может быть сопоставима со скоростью выполнения на центральном процессоре, а в некоторых случаях и медленнее. Это связано с тем, что операция загрузки данных из оперативной памяти в память графического процессора занимает сравнительно много времени. К тому же скорость работы может быть довольно низкой при попытке использовать графический процессор для вычислений, которые нельзя выполнять параллельно.

В современных браузерах есть возможность использования графического процессора при помощи технологии WebGL. Web Graphics Library (WebGL) это программная библиотека для языка JavaScript, предназначенная для визуализации графики в браузерах.

Для работы с WebGL необходимо написать программу на языке программирования шейдеров GLSL [12], которая в дальнейшем будет

[11]Официальный сайт спецификации DOM – https://dom.spec.whatwg.org/

[12]Официальный сайт спецификации Indexed DB – https://www.w3.org/TR/IndexedDB/

загружаться и исполняться графическим процессором. Описание математических операций, необходимых для запуска моделей машинного обучения, на языке шейдеров GLSL является весьма трудоемким процессом. Основная проблема заключается в том, что данный язык ориентирован на описание узко-специальных математических преобразований над графическими примитивами, и описание математических операций, используемых в моделях машинного обучения, в данном языке является нетривиальной задачей.

В настоящее время активно развивается новая технология для работы с графическими процессорами в браузерах — WebGPU. Для работы с WebGPU необходимо написать программу на языке программирования шейдеров WGSL[13], которая в дальнейшем будет загружаться и исполнятся графическим процессором.

Основная цель данной технологии — обеспечить доступ к более продвинутым функциям графического процессора. Основное отличие WebGPU от WebGL состоит в том, что WebGPU предоставляет API не только для графических преобразований, но и для общих вычислений на графическом процессоре, в том числе и математических операций, необходимых для запуска моделей машинного обучения.

[13]Официальный сайт спецификакции языка WGSL – https://www.w3.org/TR/WGSL/

2. Реализация

В рамках реализации поддержки JavaScript виртуальных машин было опробовано несколько подходов и проведена оценка их эффективности.

2.1. Переход на Kotlin Multiplatform

Первоначально была применена технология Kotlin Multiplatform и поддержана компиляция библиотеки в JavaScript. Для этого код существующего приложения пришлось “коммонизировать” — то есть выделить часть, выразимую в ограничениях диалекта Kotlin/Common, а оставшуюся вынести в модуль Kotlin/JVM через expect/actual аннотации. На данном этапе существенных затруднений не возникло, так как библиотека KInference изначально проектировалась с мыслью о будущей возможной её “коммонизации”.

Была проведена апробация полученного решения для ранее разработанных операторов и поддерживаемых моделей. Для этого пришлось перевести всю тестовую инфраструктуру также на Kotlin Multiplatform. Преодолев незначительные сложности с загрузкой ресурсов и асинхронной структурой системы JavaScript тестирования, удалось запустить существовашие тесты — все они выполнились корректно и для JVM, и для JS.

Далее было проведено два эксперимента, в которых измерялось время запуска моделей, основанных на архитектуре Transformer [1], для JVM и JS версий библиотеки. В первом эксперименте рассматривается модель для автодополнения текста на английском языке, а во втором эксперименте модель для исправления грамматических ошибок в текстах на английском языке. Ранее KInference использовался для запуска данных моделей в Grazie Platform[14] — платформе для работы с текстами на естественном языке, разрабатываемой в JetBrains.

Все замеры проводились на стенде со следующими техническими

[14]Ссылка на официальный сайт Grazie Platform – https://grazie.ai/

характеристиками:

- операционная система macOS 12.3.1;
- 8-ядерный процессор Intel Core i9 с тактовой частотой 2,3 ГГц;
- 32 Гб оперативной памяти DDR4 с тактовой частотой 2667 МГц;
- для замеров скорости JS версии библиотеки использовался браузер Google Chrome версии 100.0.4896.127.

2.1.1. Эксперимент с моделью для автодополнения текста

Данная модель основана на архитектуре GPT-2 [7] и предоставляет возможность автодополнения текста на английском языке в Grazie Platform.

Первоначально текст, который необходимо дополнить, при помощи внешних утилит преобразуется в токены и извлекается N последних токенов. Данные токены составляют контекст, относительно которого модель будет предсказывать слова для автодополнения, а параметр N определяет размер контекста и является настраиваемым параметром. Далее полученные токены преобразуются в численные векторы и передаются в модель, которая вычисляет выходной токен. В конце данный токен декодируется в текст на английском языке.

Замеры проводились для различного размера контекста. В Таблице 1 представлены средние значения для 100 запусков и разброс значений для данных замеров.

2.1.2. Эксперимент с моделью для исправления грамматических ошибок

Данная модель основана на архитектуре BERT и предоставляет возможность исправления грамматических ошибок в текстах на английском языке в Grazie Platform.

Первоначально текст, который необходимо проверить на наличие грамматических ошибок, при помощи внешних утилит преобразуется

Длина контекста, количество токенов	JVM версия библиотеки, мс	JS версия библиотеки, мс
50	72 ± 1	1022 ± 5
100	151 ± 1	1203 ± 4
200	325 ± 1	4287 ± 20
256	435 ± 2	5582 ± 20

Таблица 1: Результаты замеров для модели GPT-2

в токены, а они в свою очередь передаются в модель. В ответ модель возвращает список токенов, которые необходимо исправить, а также несколько токенов с вариантами исправления для обнаруженных ошибок.

Замеры проводились для различного количества входных токенов. В Таблице 2 представлены средние значения для 100 запусков и разброс значений для данных замеров.

Количество токенов	JVM версия библиотеки, мс	JS версия библиотеки, мс
32	29 ± 1	313 ± 7
64	58 ± 1	725 ± 9
128	126 ± 1	1193 ± 3
256	287 ± 2	2586 ± 4
512	729 ± 3	7185 ± 25

Таблица 2: Результаты замеров для модели BERT

В итоге обоих экспериментов было установлено, что JVM версия библиотеки значительно эффективнее JS версии библиотеки. Это обусловлено как тем, что виртуальные машины JS не ориентированы на выполнение большого количества математических вычислений, так и тем, что оптимизации в KInference выбирались и апробировались для

JVM.

В связи с низкой производительностью JS версии библиотеки было принято решение исследовать существующие браузерные оптимизации и реализовать наиболее подходящие для JS версии библиотеки KInference.

2.2. Сравнение браузерных оптимизаций

Для проведения сравнения браузерных оптимизаций был проведён эксперимент[15], в котором замерялось время выполнения функции матричного умножения в браузере. Данная функция была выбрана в связи с тем, что она является одной из наиболее часто используемых функций в современных моделях глубинного обучения.

В ходе эксперимента замерялось время умножения матрицы $N \times K$ на матрицу $K \times M$. Замеры производились с помощью встроенных функций языка JS.

Эксперимент проводился с использованием следующих реализаций функции матричного умножения:

- реализация TFJS с использованием WebGL;
- реализация TFJS с использованием WASM с использованием SIMD и параллелизации;
- реализация ONNXRuntime с использованием WebGL;
- реализация ONNXRuntime с использованием WASM с использованием SIMD и параллелизации.

В Таблице 3 представлены средние результаты для 100 запусков и разброс значений, полученные в ходе проведения данного эксперимента. Все замеры производились на стенде со следующими техническими характеристиками:

- операционная система macOS 12.3.1;

[15]Ссылка на репозиторий с исходным кодом эксперимента – https://github.com/cupertank/browser-benchmarks

- 8-ядерный процессор Intel Core i9 с тактовой частотой 2,3 ГГц;
- графический процессор AMD Radeon Pro 5500M с 4 Гб оперативной памяти GDDR6;
- 32 Гб оперативной памяти DDR4 с тактовой частотой 2667 МГц;
- браузер Google Chrome версии 100.0.4896.127.

Параметры матриц			TFJS WebGL, мс	TFJS WASM, мс	ONNXRuntime WebGL, мс	ONNXRuntime WASM, мс
N	K	M				
2048	1024	2048	46 ± 1	100 ± 1	610 ± 2	134 ± 3
1024	2048	2048	50 ± 1	125 ± 1	362 ± 6	131 ± 3
2048	2048	2048	74 ± 1	213 ± 1	694 ± 9	229 ± 3
3072	3072	3072	175 ± 2	446 ± 3	1476 ± 10	1922 ± 5

Таблица 3: Результаты замеров времени работы функции матричного умножения для различных реализаций

В итоге данного эксперимента было установлено, что реализация TFJS с использованием технологии WebGL показала наилучшие результаты. Это обусловлено тем, что графические процессоры более оптимизированы для выполнения большого количества математических операций.

По итогам эксперимента было решено подключить тензорные операции с использованием технологии WebGL из библиотеки TFJS к библиотеке KInference, и с их помощью реализовать операторы ONNX, необходимые для запуска моделей GPT-2 и BERT.

2.3. Подключение TFJS к KInference

Рассматривая возможные варианты интеграции тензорных операций из библиотеки TFJS в KInference, было принято решение сохранить полученную в главе 2.1 реализацию KInference (далее она будет называться CPU реализацией KInference) и создать новую реализацию с использованием библиотеки TFJS (далее она будет называться GPU реализацией KInference). Данный подход предоставит пользователям

KInference выбор устройства, на котором будет запускаться модель в браузере: на центральном процессоре с помощью CPU реализации или на графическом процессоре с помощью GPU реализации.

Первостепенной задачей являлось сохранение общих интерфейсов между обеими реализациями и переиспользование уже существующего кода, поэтому было принято решение разделить существующую архитектуру KInference на несколько частей.

2.3.1. Разделение архитектуры KInference

В первую очередь был создан API модуль, в котором были собраны интерфейсы и абстрактные классы, определяющие реализацию KInference. На Рис. 3 изображена архитектура данного модуля.

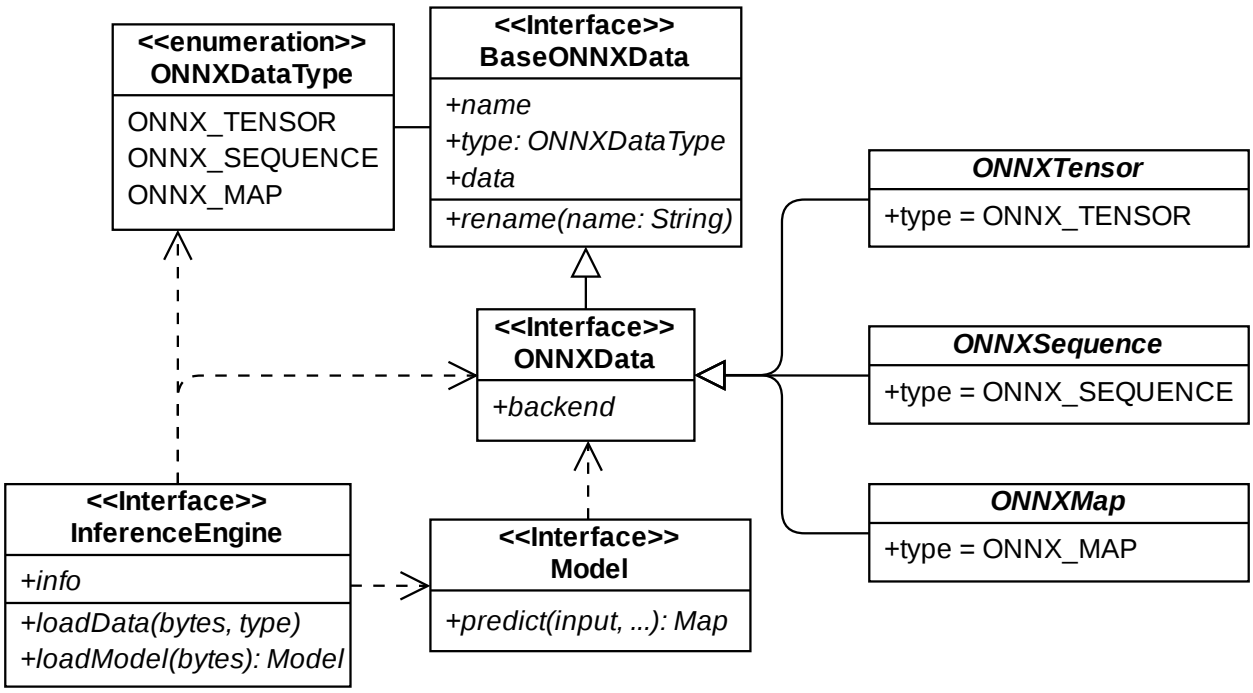

Рис. 3: Архитектура API модуля

Формат ONNX поддерживает три типа данных, которые могут использоваться в модели: **тензор**, **последовательность** и **словарь**. Последовательность является типизированным списком, который может в себе содержать либо тензоры, либо словари, либо другие последовательности. Словарь является отображением строк или целых положительных чисел в тензоры, списки или другие словари. Например, словарь

может отображать строки в тензоры или числа в другие словари и т.д.

Для данных типов были созданы три соответствующих абстрактных класса, их необходимо реализовать для описания их внутреннего представления и функций над ними в каждой из реализаций KInference, так как в разных реализациях они могут быть специфичными (например, в CPU реализации это собственное блочное представление, а в GPU реализации это будут многомерные массивы из библиотеки TFJS).

Поскольку в каждой реализации неизбежно придётся реализовывать класс для описания модели и её запуска, в данный модуль был вынесен общий интерфейс модели, который необходимо реализовывать в каждой реализации KInference. Данный интерфейс содержит одну абстрактную функцию `predict`, реализация которой должна принимать на вход список входных данных модели, запускать вычисление результатов модели и возвращать их.

Формат ONNX предоставляет модели машинного обучения в виде файла в формате Protobuf [19]. Рано или поздно при создании реализаций KInference встанет вопрос о процедуре десериализации моделей и типов данных ONNX, а в каждой реализации она может протекать различными способами в зависимости от платформы и внутренних представлений типов ONNX. Для обобщения данной процедуры был создан интерфейс `InferenceEngine`, который имеет две функции `loadModel` и `loadData`, реализации которых должны по набору байтов модели ONNX или набору байтов типов ONNX возвращать экземпляр класса модели ONNX или экземпляр типа данных ONNX соответственно.

Данного набора интерфейсов достаточно для обеспечения основных классов и функций в каждой из реализаций KInference, благодаря чему сохраняется целостность платформы KInference и удобство её использования.

Следующим шагом была вынесена обработка исполнения графа вычислений ONNX, ранее используемая в CPU реализации KInference, в отдельный “execution” модуль. Данный модуль будет полезен при создании GPU реализации KInference, так как стратегия исполнения графа вычислений ONNX имеет общий характер и не привязана к конкрет-

ной реализации KInference. На Рис. 4 изображена архитектура данного модуля.

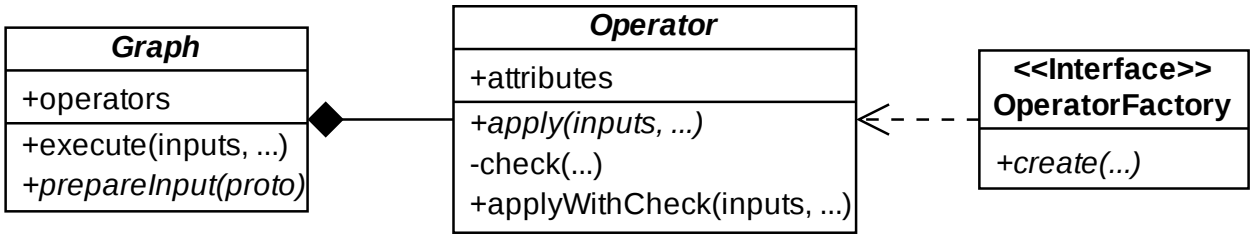

Рис. 4: Архитектура execution модуля с исполнением графа ONNX.

Основным классом в execution модуле является абстрактный класс `Operator`. Данный класс является представлением операторов ONNX в KInference, он содержит множество реализованных функций для работы с атрибутами операторов и функции для проверки корректности входов оператора. Для реализации конкретного оператора необходимо отнаследоваться от данного класса и реализовать ключевую функцию `apply`, она должна получать список входов оператора, выполнять необходимые вычисления и возвращать ответ оператора. В данном модуле не представлены реализации операторов ONNX, так как они должны быть созданы в модуле с конкретной реализацией KInference с учётом внутреннего представления типов ONNX в конкретной реализации KInference.

Для сопоставления десериализованных операторов ONNX с конкретными классами операторов ONNX необходима фабрика операторов, но так как в различных реализациях KInference может быть представлен различный набор реализованных операторов, то данная фабрика не может быть реализована одним экземпляром. В связи с этим в execution модуле был создан интерфейс для фабрики операторов ONNX, который должен быть реализован в каждой реализации KInference, которая использует данный модуль. В данном интерфейсе определена одна ключевая функция `create`, которая должна сопостав-

лять имя десериализованного оператора ONNX с соответствующим ему классом в конкретной реализации KInference.

Ключевым классом в execution модуле является класс графа. В данном классе реализован разбор графа вычислений ONNX, его топологическая сортировка и создание необходимых операторов с помощью фабрики (экземпляр которой передаётся в конструктор класса). Также в данном классе уже реализована основная функция `execute`, которая обеспечивает корректное исполнение данного графа вычислений. В конкретных реализациях необходимо отнаследоваться от данного класса и реализовать функцию `prepareInput`, которая используется при считывании инициализаторов графа (инициализаторами являются предобученные параметры модели, которые в формате ONNX представляются тензорами). Данную функцию надо реализовывать в каждой конкретной реализации KInference так как внутреннее представление тензоров в разных реализациях может быть различным.

Данного набора классов достаточно для описания корректного исполнения графа вычислений и операторов ONNX.

В конце была модифицирована и дополнена CPU реализация KInference с учётом новых модулей. На Рис. 5 изображена обновлённая архитектура CPU реализации KInference.

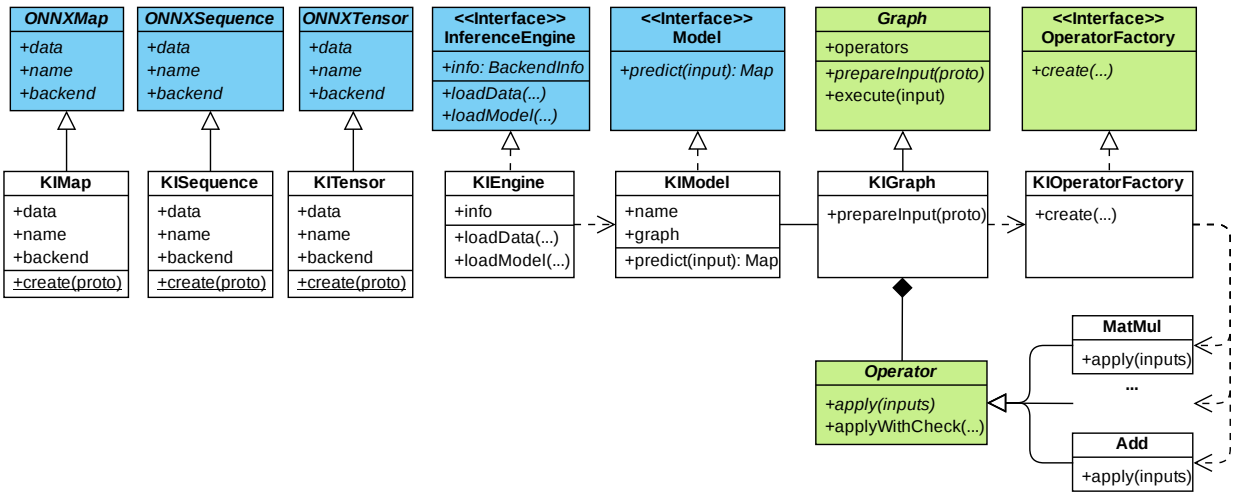

Рис. 5: Обновлённая архитектура основной реализации (синим цветом выделены объекты из API модуля, зелёным объекты из execution модуля).

В первую очередь были модифицированы классы, реализующие типы ONNX, и добавлено наследование от абстрактных классов типов ONNX из API модуля.

В связи с тем, что обработка графа вычислений ONNX была вынесена в отдельный модуль, были модифицированы и дополнены классы всех ранее реализованных операторов, фабрики операторов и графа и добавлено наследование соответствующих классов в CPU реализации KInference.

Далее был модифицирован класс модели с учётом обновлённого класса графа, а также добавлено наследование от интерфейса модели из API модуля.

В конце был создан класс `KIEngine` с реализацией интерфейса `InferenceEngine` из API модуля. Для десериализации моделей ONNX из формата Protobuf в данной реализации используется собственная библиотека, ранее реализованная в KInference, с чтением в блочный массив примитивов.

Во время интеграции новых модулей с CPU реализацией KInference не возникло каких-либо проблем, так как большинство интерфейсов и функций были позаимствованы и вынесены из данной реализации в отдельные модули.

После успешного разделения архитектуры началось создание GPU реализации KInference с использованием тензорных операций из библиотеки TFJS.

2.3.2. Создание новой реализации для KInference

В связи с тем, что GPU реализация зависит от библиотеки TFJS, которая имеет только JS реализацию, то для неё была настроена компиляция только для JS.

В первую очередь к данной реализации была подключена библиотека `tfjs-core`. В данной библиотеке содержится основной API для тензоров и их функций в библиотеке TFJS. Так как TFJS является JS библиотекой, а KInference Kotlin библиотекой, необходимо было провести линковку основных классов и функций из библиотеки TFJS к

KInference с помощью ключевого слова `external`[16]. Таким образом, был подключен основной класс тензоров `NDArrayTFJS` из библиотеки TFJS и 34 функции для него.

Следующим шагом к GPU реализации была подключена библиотека `tfjs-backend-webgl`. Данная библиотека содержит в себе реализации внутреннего представления тензоров и их функций с использованием технологии WebGL. Данная библиотека содержит в себе ключевой класс `MathBackendWebGL`, экземпляр которого необходимо передать встроенной функции библиотеки `tfjs-core` для обеспечения работоспособности тензоров и их операций из библиотеки TFJS с помощью технологии WebGL. Данный класс был аналогично подключен к GPU реализации KInference с помощью ключевого слова `external`.

Ключевой особенностью реализации тензоров в библиотеке TFJS является то, что код шейдеров генерируется динамически во время вызова конкретной тензорной операции с учётом переданных параметров. Далее данный код компилируется и кэшируется, для дальнейшего переиспользования и ускорения выполнения данной операции в будущих запусках.

Таким образом, были подключены тензоры и их операции из библиотеки TFJS с использованием технологии WebGL к GPU реализации KInference. Следующим шагом необходимо было реализовать исполнение модели ONNX.

Первоначально к GPU реализации KInference были подключены API модуль и модуль с исполнением графа ONNX и реализованы типы ONNX, наследуясь от соответствующих классов из API модуля. В качестве внутреннего представления тензоров ONNX в данной реализации используются тензоры из библиотеки TFJS.

Следующим шагом были реализованы операторы ONNX, необходимые для запуска моделей GPT-2 [7] и BERT [2]: `LayerNormalization`, `MatMul`, `Attention` и т.д, с использованием тензорных операций из библиотеки TFJS. Суммарно было реализовано **22** оператора для GPU ре-

[16]Ссылка на описание работы ключевого слова `external` в языке Kotlin – https://kotlinlang.org/docs/js-interop.html#external-modifier

ализации KInference.

Далее была создана фабрика для реализованных операторов, и реализован класс графа. На данном этапе не возникло никаких трудностей, так как из-за своей архитектуры они очень схожы с классами из CPU реализации KInference.

В конце был реализован класс модели и класс `TFJSEngine`. Ключевое отличие класса модели от аналогичного из CPU реализации заключается в том, что при создании экземпляра класса модели с помощью встроенных функций библиотеки TFJS производится проверка на то, что выбрана реализация библиотеки TFJS с использованием WebGL, и если это не так, то создаётся экземпляр класса `MathBackendWebGL` и передаётся во встроенную функцию TFJS. Для десериализации моделей ONNX в GPU реализации KInference используется библиотека, ранее реализованная в KInference, с чтением в стандартный массив примитивов.

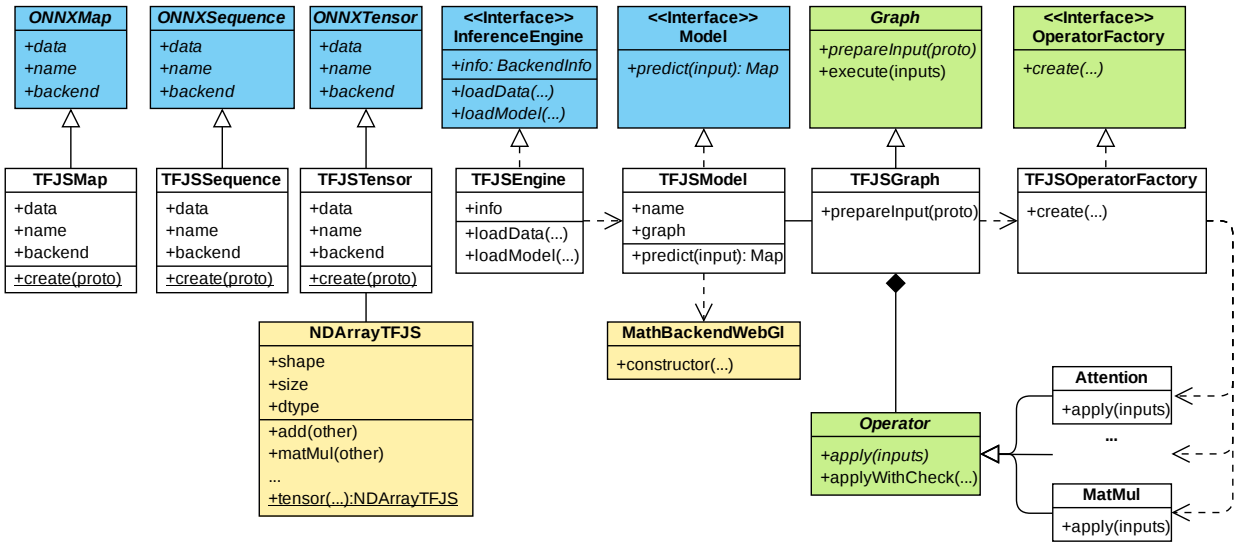

Рис. 6: Полученная архитектура GPU реализации KInference (синим цветом выделены объекты из API модуля, зелёным объекты из execution модуля, жёлтым объекты из внешних библиотек TFJS).

В итоге всех выполненных действий была получена новая GPU реализация KInference для JavaScript при помощи тензорных операций из библиотеки TFJS с использованием технологии WebGL. Следующим шагом была проведена апробация полученного решения.

3. Апробация

В рамках апробации ставилась задача как проверить корректность работы полученных реализаций, так и оценить влияние GPU реализации на производительность по сравнению с наивным решением — простым использованием Kotlin Multiplatform.

В первую очередь было проведено тестирование реализованных операторов на корректность и соответствие спецификации ONNX в обеих реализациях. Для данной проверки использовались модели, состоящие из тестируемого оператора, которые ранее применялись в KInference. Данные модели были созданы с помощью библиотеки ONNX на языке Python. Для каждой модели были сгенерированы соответствующие входы, получены заведомо корректные выходы с помощью библиотеки ONNXRuntime и экспортированы в формат ONNX. Данные тесты были подключены к обеим реализациям KInference. К тому же было проведено тестирование на корректное исполнение моделей GPT-2 и BERT. Для данных моделей аналогично были сгенерированы входы и заведомо корректные выходы и также подключены к обеим реализациям KInference.

Все операторы успешно прошли тестирование, и модели выполняются корректно в обеих реализациях.

Дальнейшим шагом было проведено два эксперимента, в которых измерялось время запуска моделей GPT-2 и BERT для CPU и GPU реализаций KInference для JS.

Все замеры проводились на стенде со следующими техническими характеристиками:

- операционная система macOS 12.3.1;
- 8-ядерный процессор Intel Core i9 с тактовой частотой 2,3 ГГц;
- графический процессор AMD Radeon Pro 5500M с 4 Гб оперативной памяти GDDR6;
- 32 Гб оперативной памяти DDR4 с тактовой частотой 2667 МГц;

- браузер Google Chrome версии 100.0.4896.127.

3.1. Эксперимент с моделью GPT-2

Данный эксперимент проводился с использованием модели для автодополнения текста на английском языке из Grazie Platform. Эксперимент проводился с теми же данными и аналогично эксперименту из главы 2.1.1.

Замеры проводились для различной длины контекста. В Таблице 4 представлены средние значения для 100 запусков и разброс значений для данных размеров.

Длина контекста, количество токенов	CPU реализация Kinference, мс	GPU реализация Kinference, мс
50	1022 ± 5	67 ± 1
100	1203 ± 4	74 ± 2
200	4287 ± 20	83 ± 1
256	5582 ± 20	86 ± 2

Таблица 4: Результаты замеров на модели GPT-2

В итоге данного эксперимента было установлено, что GPU реализация KInference кратно эффективнее старой. Таким образом, подтвердилось предположение о том, что реализация с использованием библиотеки TFJS даст существенный прирост производительности за счет задействования GPU.

3.2. Эксперимент с моделью BERT

Данный эксперимент проводился с использованием модели для исправления грамматических ошибок в текстах на английском языке из Grazie Platform. Эксперимент проводился с теми же данными и аналогично эксперименту из главы 2.1.2.

Замеры проводились для различного количества входных токенов. В Таблице 5 представлены средние значения для 100 запусков и разброс

данных замеров.

Количество токенов	CPU реализация Kinference, мс	GPU реализация Kinference, мс
32	313 ± 7	11 ± 1
64	725 ± 9	12 ± 1
128	1193 ± 3	12 ± 1
256	2586 ± 4	13 ± 1
512	7185 ± 25	17 ± 1

Таблица 5: Результаты замеров для модели BERT

В итоге данного эксперимента было установлено, что GPU реализация KInference кратно эффективнее старой. Таким образом, аналогично ситуации с GPT-2 было подтверждено предположение о том, что реализация с использованием библиотеки TFJS дает существенный прирост производительности за счет использования GPU.

В результате экспериментов удалось показать кратное увеличение эффективности исполнения моделей. Что еще более важно, достигнутые показатели менее 300 мс — барьера времени отклика пользователю, указанного представителями индустрии, после которого время работы модели становится заметным глазу на задаче автодополнения.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

- реализована возможность использования KInference в проектах на JS;
- проведён обзор оптимизаций математических операций, доступных в браузере;
- проведена апробация оптимизаций математических операций, доступных в браузере;
- разработана и интегрирована новая реализация KInference для JS с использованием тензорных операций из библиотеки TFJS и технологии WebGL;
- проведена апробация конечного решения.

В рамках дальнейшей работы планируется провести эксперименты с WebGPU и расширить поддержку операторов в GPU реализации KInference.

Результаты, полученные в ходе данной работы, будут использованы в проекте Grazie Platform компании JetBrains для осуществления возможностей автодополнения и исправления грамматических ошибок в текстах на английском языке локально в веб-редакторе Grazie и расширении для браузеров Google Chrome — Grazie Chrome Extension.

Ссылка на репозиторий с исходным кодом: https://github.com/JetBrains-Research/kinference, разработка велась под аккаунтом cupertank.

Список литературы

[1] Attention is All you Need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // Advances in Neural Information Processing Systems / Ed. by I. Guyon, U. Von Luxburg, S. Bengio et al. — Vol. 30. — Curran Associates, Inc., 2017. — URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[2] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding / Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova // arXiv preprint arXiv:1810.04805. — 2018. — URL: https://arxiv.org/pdf/1810.04805.pdf.

[3] Bringing the Web up to Speed with WebAssembly / Andreas Haas, Andreas Rossberg, Derek L. Schuff et al. // SIGPLAN Not. — 2017. — jun. — Vol. 52, no. 6. — P. 185–200. — URL: https://doi.org/10.1145/3140587.3062363.

[4] Empirical evaluation of gated recurrent neural networks on sequence modeling / Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio // arXiv preprint arXiv:1412.3555. — 2014.

[5] Flynn Michael J. Some Computer Organizations and Their Effectiveness // IEEE Transactions on Computers. — 1972. — Vol. C-21, no. 9. — P. 948–960.

[6] Hochreiter Sepp, Schmidhuber Jürgen. Long short-term memory // Neural computation. — 1997. — Vol. 9, no. 8. — P. 1735–1780.

[7] Language Models are Unsupervised Multitask Learners / Alec Radford, Jeff Wu, Rewon Child et al. — 2019. — URL: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[8] Larsen Rasmus Munk, Shpeisman Tatiana. Tensorflow graph optimizations. — 2019.

[9] Lattner C., Adve V. LLVM: a compilation framework for lifelong program analysis amp; transformation // International Symposium on Code Generation and Optimization, 2004. CGO 2004. — 2004. — P. 75–86.

[10] Lomont Chris. Introduction to intel advanced vector extensions // Intel white paper. — 2011. — Vol. 23.

[11] MNN: A Universal and Efficient Inference Engine / Xiaotang Jiang, Huan Wang, Yiliu Chen et al. // Proceedings of Machine Learning and Systems / Ed. by I. Dhillon, D. Papailiopoulos, V. Sze. — Vol. 2. — 2020. — P. 1–13. — URL: https://proceedings.mlsys.org/paper/2020/file/8f14e45fceea167a5a36dedd4bea2543-Paper.pdf.

[12] Marroquim Ricardo, Maximo André. Introduction to GPU Programming with GLSL // 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing. — 2009. — P. 3–16.

[13] JetBrains. — Multiplatform Programming. — URL: https://kotlinlang.org/docs/multiplatform.html (дата обращения: 2021-12-15).

[14] PyTorch: An Imperative Style, High-Performance Deep Learning Library / Adam Paszke, Sam Gross, Francisco Massa et al. // Advances in Neural Information Processing Systems / Ed. by H. Wallach, H. Larochelle, A. Beygelzimer et al. — Vol. 32. — Curran Associates, Inc., 2019. — URL: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[15] Raman S.K., Pentkovski V., Keshava J. Implementing streaming SIMD extensions on the Pentium III processor // IEEE Micro. — 2000. — Vol. 20, no. 4. — P. 47–57.

[16] Shuangfeng Li. TensorFlow Lite: On-Device Machine Learning Framework // Journal of Computer Research and Development. — 2020. — Vol. 57, no. 9. — P. 1839. — URL: https://crad.ict.ac.cn/EN/abstract/article_4251.shtml.

[17] Tensorflow: Large-scale machine learning on heterogeneous distributed systems / Martín Abadi, Ashish Agarwal, Paul Barham et al. // arXiv preprint arXiv:1603.04467. — 2016.

[18] Tensorflow. js: Machine learning for the web and beyond / Daniel Smilkov, Nikhil Thorat, Yannick Assogba et al. // Proceedings of Machine Learning and Systems. — 2019. — Vol. 1. — P. 309–321.

[19] Protocol Buffers: Google's Data Interchange Format : Rep. / Google ; Executor: Kenton Varda : 2008. — 6. — URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.