Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления

Выпускная квалификационная работа бакалавра

Профилирование и аннотация времени выполнения в MLIR как инструмент для оптимизации программ машинного обучения

Автор:
Студент 109 группы
Алексеев Алексей Алексеевич

Научный руководитель:
Черноног Вячеслав Викторович

Научный консультант:
Голенев Александр Дмитриевич

Москва 2025

Аннотация

Профилирование и аннотация времени выполнения в MLIR как инструмент для оптимизации программ машинного обучения

Алексеев Алексей Алексеевич

Современные модели машинного обучения представляют собой важнейшую область как для фундаментальных исследований, так и для решения прикладных задач в различных областях. Одной из актуальных задач в индустрии является создание инфраструктуры, которая позволит эффективно запускать и исполнять эти модели на устройствах с ограниченными ресурсами, на неоднородных и специализированных архитектурах.

Моя работа направлена на решение проблемы формализации подхода к поиску возможных мест для оптимизаций моделей во время компиляции. Этот подход учитывает специфику выполнения программ на различных устройствах. Для достижения этой цели используются инструменты многоуровневого промежуточного представления (MLIR), а также динамические профили исполнения программ.

В рамках работы была разработана система автоматического аннотирования промежуточного представления. Эти аннотации могут служить основой для применения дальнейших трансформаций и анализа, что позволит значительно ускорить работу исскустевнного интелекта.

Содержание

1 Введение

1.1 Особенности задач машинного обучения

За последние несколько лет методы машинного обучения (ML, от Machine Learning) продемонстрировали стремительный рост как в области применения, так и в качестве получаемых результатов. В большинстве случаев этот прогресс стал возможен благодаря увеличению вычислительных мощностей используемых исполняющих сред. Чтобы в полной мере оценить масштаб произошедших изменений, необходимо рассмотреть вычислительные требования, предъявляемые современными моделями, а также проанализировать особенности архитектур, на которых они исполняются. Каждая программа исскустевнного интелекта (AI, от Artificial Intelligence) условно делится на два этапа: обучение и использование (инференс). Этап обучения заключается в подаче большего количества разнообразных примеров, на основе которых модель адаптирует свои параметры, чтобы повысить способность к обобщению и точности предсказаний на новых, ранее не виденных данных. Этот процесс достигается с помощью математических методов регрессии и оптимизации, таких как поиск локального минимума в многомерном параметрическом пространстве. Целью является построение модели, приближённой к оптимальной для заданного распределения входных данных. Характерной чертой большинства таких задач является возможность их формализации с использованием примитивов линейной алгебры. В упрощённом виде модель можно представить как последовательность алгебраических операций над тензором параметров X и входными данными F, где F представляет собой пространство признаков, а Y — соответствующее пространство ответов. Результатом обучения является набор параметров (весов) X, таких, что выполняется следующее условие:

$$|XF - Y|_{\text{norm}} \to \min \tag{1}$$

На этапе инференса модель применяется к новым данным. Это означает сохранение последовательности алгебраических операций (часто представляемых в виде графа) и загрузку ранее полученных значений параметров X на конечное устройство. В отличие от обучения, инференс не включает в себя процесс оптимизации — он лишь выполняет предсказание на основе уже обученных весов, что делает его значительно менее ресурсоёмким. Тем не менее, в последние годы наблюдается постоянное увеличение размера моделей, выражающееся как в количестве параметров, так и в сложности вычислений. Это ставит под вопрос возможность стабильной и быстрой работы моделей на конечных устройствах без потери качества или увеличения времени отклика. На рисунке ниже представлена экспоненциальная тенденция роста количества операций при обучении современных моделей. Этот рост напрямую коррелирует с вычислительной нагрузкой при их инференсе. Наблюдаемая зависимость в некотором смысле напоминает закон Мура, с той разницей, что вместо количества транзисторов речь идёт о росте размеров тензоров и значений их элементов. Но за счёт чего физически

компенсируется постоянно растущая сложность современных AI-моделей?

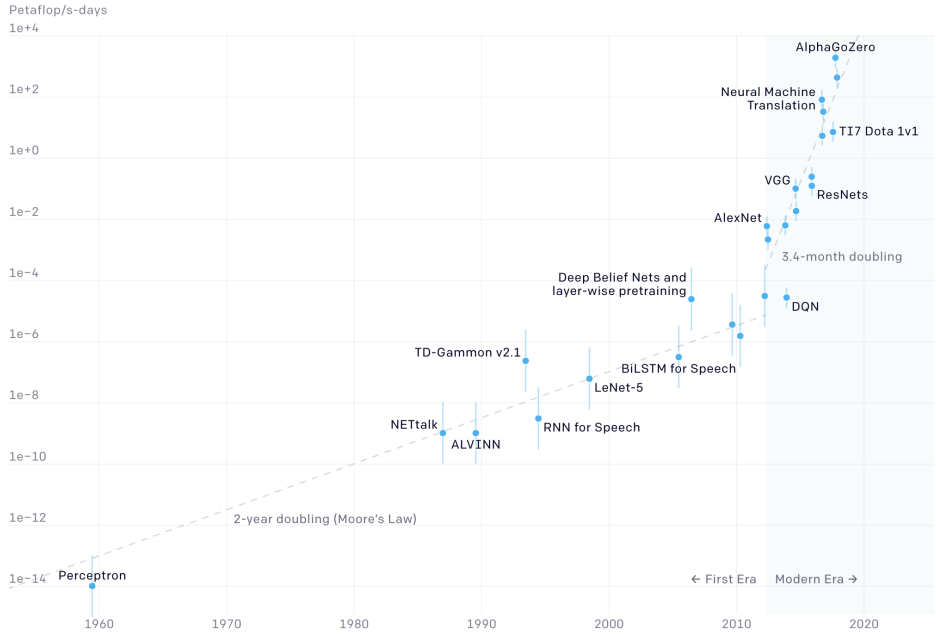

Рис. 1: Рост вычислительных требований при обучении современных моделей.

Вычислительно, процесс получения предсказания модели можно представить как последовательность вложенных циклов for, где каждая последующая строка соответствует проекции i-й координатной оси на итерационное пространство. Конструкция вложенных циклов `for` представляет собой основную проблему при распараллеливании программ. Это связано с множественными зависимостями данных, на основе которых можно построить **граф потока данных** (от англ. data flow graph). Вершинами такого графа являются данные некоторой ячейки многомерного массива:

$$A[i][j][k]\dots[l][m]$$

Ребро, проведённое из вершины $(a_1, b_1, c_1, d_1, \dots)$ в вершину $(a_2, b_2, c_2, d_2, \dots)$, обозначает необходимость значения первой ячейки для вычисления второй. Обычно при построении графа потока данных выделяют следующие типы **зависимостей** данных:

1. **Внутрициклические зависимости** - между итерациями одного цикла. Если текущая итерация зависит от предыдущей, это создает последовательную цепочку в графе.

2. **Межциклические зависимости** - между разными уровнями вложенности. Внешний цикл может влиять на все итерации внутреннего цикла.

3. **Диагональные зависимости** - когда элемент зависит от элементов на предыдущих итерациях обоих циклов одновременно.

Data-flow граф широко применяется при разработке алгоритмов распараллеливания тензорных вычислений. Тензорные вычисления являются основой современных программ искусственного интеллекта, что обусловливает необходимость их строгого и структурированного описания в рамках выполняемой задачи. К числу типичных тензорных операций относятся, в частности, матричное умножение, свёртка тензоров, а также редукция по заданным осям тензорных структур.

Применение графа вычислительного потока при описании операций матричного умножения позволяет выявить повторяемость типов зависимостей между операциями и их регулярную структуру. Так, каждый элемент результирующей матрицы $C[i][j]$ зависит от всех элементов i-й строки матрицы A и j-го столбца матрицы B, что отражает характер тензорной зависимости и даёт основу для эффективного распараллеливания вычислений.

Подобная схема требует максимальной загрузки вычислительных ресурсов (CPU, GPU, NPU — подробнее о них ниже), что влечёт за собой значительные накладные расходы, включая потребление электроэнергии. Вычислительные устройства можно условно расположить по шкале "приспособленности"к выполнению таких задач — от наименее до наиболее эффективных. Тогда иерархия будет выглядеть следующим образом:

$$CPU < GPU < NPU$$

Такое ранжирование обусловлено наличием аппаратной поддержки параллельных вычислений. Преимущество одного типа устройств над другим определяется степенью их специализации для конкретного класса задач. Если отличия между CPU и GPU относительно хорошо понятны, то NPU (Neural Processing Unit, или ИИ-ускоритель, AI accelerator) представляет собой новый архитектурный подход к выполнению матричных операций.

В отличие от CPU и GPU, где требуется программная реализация эффективного параллелизма, NPU изначально сконструированы именно для этой задачи и реализуют регулярность зависимостей данных на уровне аппаратуры. Примером такой микроархитектуры служит TPU (Google Tensor Processing Unit), в которой логические элементы располагаются так, что результат вычислений как бы «течёт» по микросхеме, распространяясь к её выходу. Это обеспечивает аппаратный уровень конвейерной обработки и высокую степень параллелизма.

В рамках данной работы различия в микроархитектуре перечисленных типов устройств не будут подробно рассматриваться. Они приведены здесь лишь для иллюстрации того, как развивается аппаратная часть и какие решения предлагаются в ответ на растущие вычислительные требования современных моделей. Важно подчеркнуть, что аппаратные средства активно эволюционируют, открывая всё больше возможностей для эффективного исполнения усложняющихся AI-моделей.

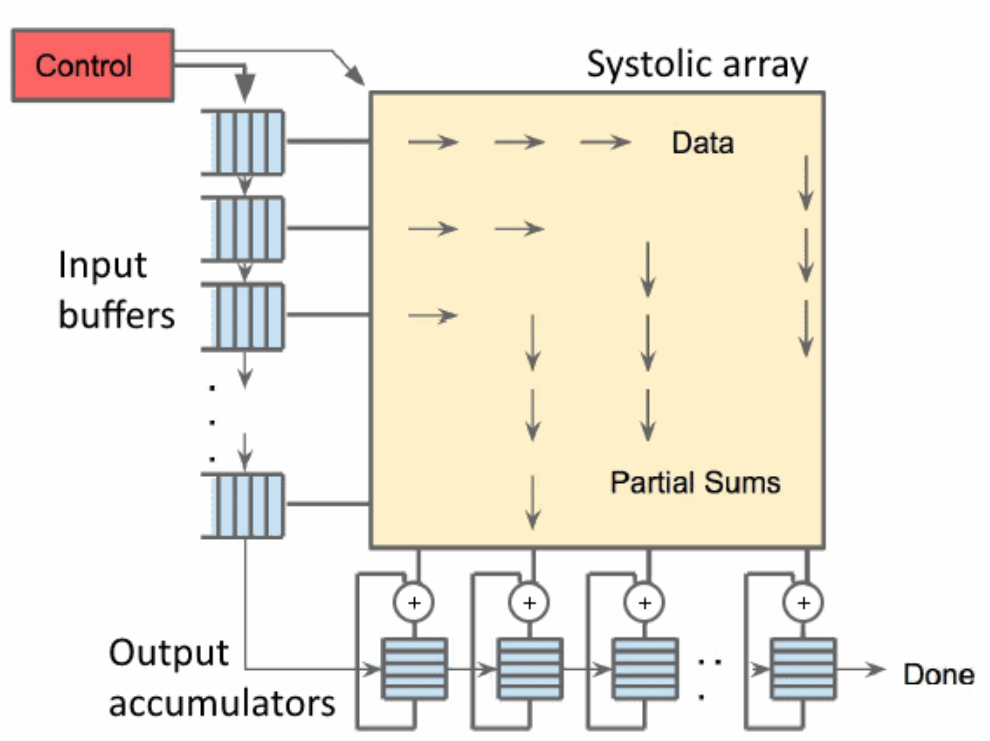

Рис. 2: Схема распространения вычислений в TPU.

1.2 Средства разработки ML-моделей

Помимо аппаратных средств ускорения, существуют также программные подходы к оптимизации.

Основная сложность в данной области исследований заключается в разработке полноценной среды программирования и предоставлении библиотек, реализующих базовые математические абстракции. Такие среды, или фреймворки, делают акцент на удобстве использования, простоте построения моделей, а также предоставлении инструментов для сжатия, профилирования и нативного исполнения моделей.

Появление этих фреймворков существенно упростило процесс разработки, снизив технический порог входа в область машинного обучения и способствовав стремительному росту объёма доступного кода. В результате сформировалась масштабная кодовая база с широким спектром моделей, способных решать множество прикладных задач.

К наиболее популярным и широко применяемым фреймворкам сегодня относятся **TensorFlow**, **PyTorch** и **ONNX**.

Однако там, где достигается удобство, нередко приходится жертвовать производительностью — именно это изначально наблюдалось в перечисленных фреймворках. Высокоуровневые математические конструкции, такие как тензорные операции, векторы, а также операции скалярного и векторного произведения, зачастую транслируются в исполняемый код довольно прямолинейно. Под «прямолинейной» трансляцией подразумевается процесс, при котором тип и порядок высокоуровневых операций не учитываются при выборе возможных оптимизаций. Оптимизации — такие как свёртка констант, планирование инструкций и машинно-зависимые преоб-

разования — выполняются на уровне элементарных арифметических операций, что ограничивает общий прирост производительности.

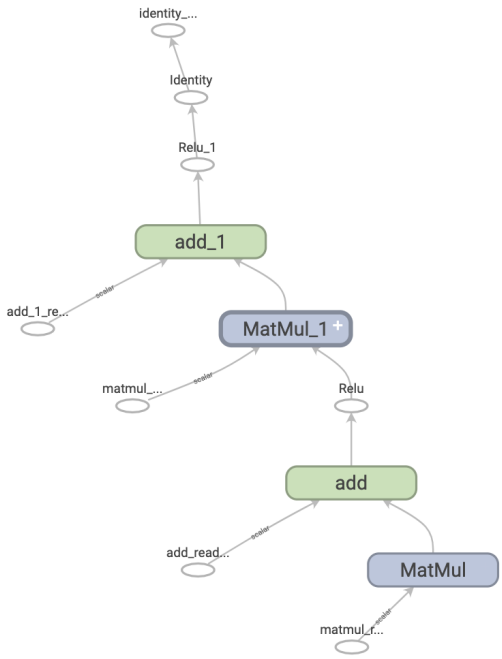

Рис. 3: Пример графа исполнения **Tensorflow**.

Становится очевидным, что разрыв между уровнями абстракции — от описания модели до низкоуровневой реализации — представляет собой незаполненную нишу, которую можно использовать для создания новых оптимизаций программ искусственного интелекта.

Одним из наиболее успешных и широко применяемых проектов в данной области является **ONNX Runtime** [1], разработанный компанией Microsoft. Он предоставляет расширенные возможности для оптимизации и выполнения моделей, представленных в формате ONNX. В рамках данного проекта было разработано собственное графовое промежуточное представление, на базе которого реализована многоуровневая система графовых оптимизаций. Эта система позволяет эффективно преобразовывать вычислительные графы с целью повышения производительности и сокращения времени исполнения. Кроме того, **ONNX Runtime** поддерживает механизмы оптимизаций, ориентированных на профиль исполнения моделей машинного обучения, что обеспечивает адаптацию вычислений под реальные характеристики нагрузки.

Процесс компиляции ONNX-моделей может быть схематично представлен в виде последовательности представлений:

$$\text{ONNX} \rightarrow \text{GraphIR} \rightarrow \text{Execution Providers} \rightarrow \text{Execution binary}$$

Еще одиним удачным примером подобного подхода можно считать проект LLVM [2], получивший широкое распространение и продолжающий активно развиваться. Ключевым преимуществом LLVM является высокая модульность и гибкость его архитектуры. Это достигается благодаря введению промежуточного представления (англ. Intermediate Representation, IR), которое используется как основа для множества этапов компиляции и оптимизации.

LLVM IR [2] сохраняет семантику исходной программы, написанной на высокоуровневом языке, но одновременно обладает структурой, близкой к ассемблерному коду. Такое сочетание делает IR мощным инструментом для анализа и преобразования программ на этапе, независимом от целевой архитектуры. Одним из главных достоинств LLVM IR является возможность реализации архитектурно-независимых оптимизаций, применяемых до генерации финального машинного кода. К таким оптимизациям можно отнести:

- распространение констант (constant propagation),
- разворачивание циклов (loop unrolling),
- инлайнинг функций,
- переупорядочивание инструкций и устранение избыточных операций,
- **и многие другие.**

Важно отметить, что LLVM предоставляет расширяемую инфраструктуру, позволяя добавлять собственные типы, инструкции, метаинформацию и реализовывать пользовательские проходы оптимизации (passes). Кроме того, LLVM IR активно используется в реализации оптимизаций, направляемых профилем выполнения (Profile-Guided Optimizations, PGO), где информация о поведении программы в реальных условиях позволяет проводить более эффективные преобразования.

Тем не менее, несмотря на выразительность и гибкость LLVM IR, он остается ориентированным на императивные языки общего назначения и ближе к низкоуровневой модели вычислений. В условиях растущей сложности моделей машинного обучения, необходимости в высокоуровневых тензорных операциях, графовых представлениях и специфике разнообразных аппаратных ускорителей (GPU, NPU), возникает потребность в промежуточном представлении, способном отразить более абстрактные вычисления, сохраняя при этом поддержку всех преимуществ компиляторной инфраструктуры.

1.3 Уровни абстракций и MLIR

Именно по этой причине в апреле 2019 года был представлен новый проект от разработчиков LLVM, направленный на решение проблемы разрыва между уровнями абстракций в представлениях программ — MLIR.

MLIR (от англ. Multi-Level Intermediate Representation) [3] представляет собой принципиально новый подход к описанию и трансформации высокоуровневых операций. Само название — многоуровневое промежуточное представление — точно отражает основную идею проекта. Используя модульную архитектуру LLVM, разработчики предложили расширяемую инфраструктуру, в рамках которой стало возможным создавать собственные уровни представления — так называемые диалекты (dialects). Каждый диалект описывает специфический набор операций, типов и правил обработки, соответствующий определённому уровню абстракции. Теперь, благодаря механизму lowering (понижение уровня представления), стало возможно последовательно преобразовывать программу с высокого уровня до низкоуровневого представления LLVM IR, контролируя и оптимизируя каждый этап компиляции.

Для современных фреймворков машинного обучения типичный процесс перехода по уровням абстракций выглядит следующим образом:

- **TF** или **TFLite** - диалект уровня исходного кода моделей,
- **MHLO** или **TOSA** - диалект математических операций ML (присутствуют тензоры, батчи и др.),
- **LINALG** - диалект алгебры линейных операций (использует буферы вместо тензоров),
- **VECTOR** - диплект низкоуровневого векторного представления.
- **LLVM IR**

Целью данной работы является создание вспомогательного инструмента для **PGO** (Profile-Guided Optimization) в инфраструктуре MLIR на основе диалекта TensorFLow [4], позволяющего проводить анализ участков исполнения программы, требующих непосредственной оптимизации. Предлагается использовать существующие профилировщики программ машинного обучения и разработать расширение диалекта верхнего уровня за счёт добавления полей метаданных. Такое отображение профиля исполнения на граф операций позволит принимать решения об оптимизации высокоуровневых операций на основе фактического поведения программы, при этом сохранив совместимость с широким набором существующих моделей.

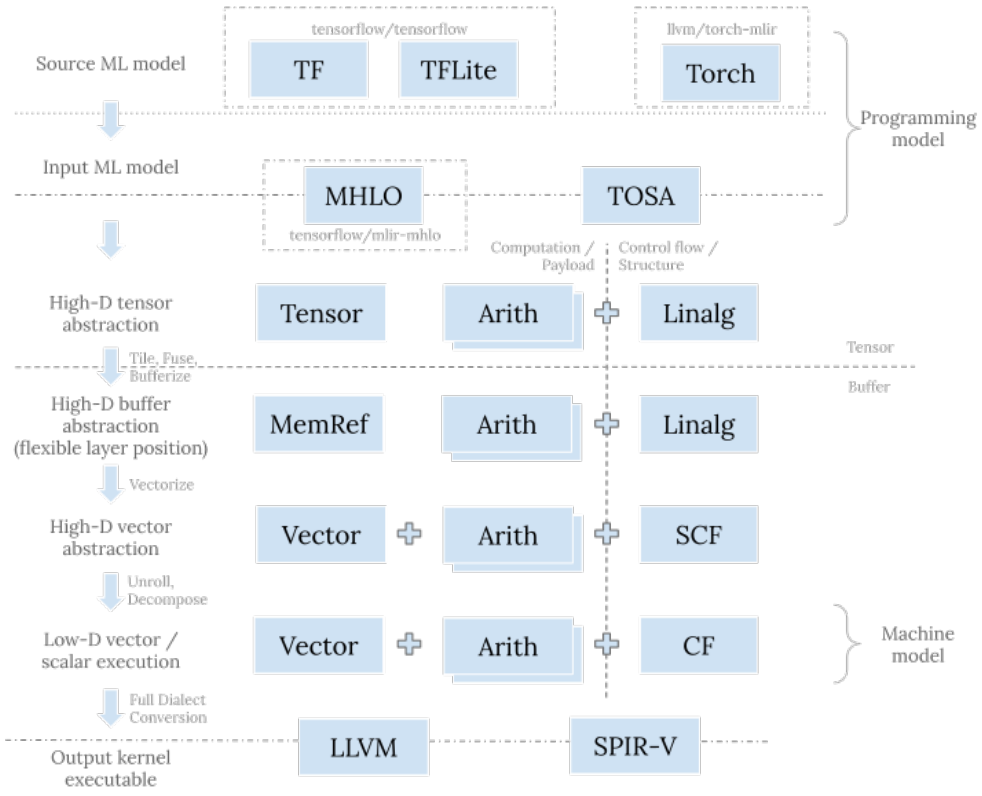

Рис. 4: Схема многоуровневого представления MLIR.

2 Постановка задачи

Необходимость вышеописанного инструмента для проведениня PGO в сущности очевидна, его наличие позволило бы детально инспектировать программы искусственного интелекта и быстро находить места, требующие оптимизаций. В особенности с появлением новых вычислительных ускорителей и альтернативных архитектур очень актуальным становится вопрос об оптимизациях для исполнения на конкретных устройствах. Это позволило бы значительно ускорить предкомпилированные модели, использующие оптимизации основанные на их же профиле исполнения.

Чтобы достичь этой цели, сформулируем задачу решаемую в рамках данной дипломной работы:

> *Разработка средства автоматического аннотирования промежуточного представления программ на основе профиля исполнения в рамках инфраструктуры MLIR*

Поставленная задача требует дополнительных пояснений, так как не может быть решена исходя из начальной формулировки. Постараемся разбить задачу на более мелкие части и зададимся целью подробного описания их взаимосвязи. Будем ранжировать также подзадачи в порядке убывающей важности, так чтобы сохранять акцент на поставленной задаче. Рассмотрим каждую из сформулированных подзадач в отдельности:

2.1 Расширение существующего диалекта/ов

MLIR предоставляет широкие возможности как для создания собственных диалектов, так и для расширения уже существующих. Однако, в отличие от классических объектно-ориентированных подходов, в MLIR отсутствует механизм прямого наследования диалектов, поскольку архитектура системы строится на принципах композиции.

Цель данной работы — получение расширенного промежуточного представления, способного не только сохранять профиль исполнения, но и обеспечивать доступ к этим данным на любом уровне абстракции после понижения (lowering).

Рассматривался подход создания обёрточного (proxy) диалекта, однако он оказался неприемлемым по ряду причин. Такой диалект создает дополнительный уровень представления, где каждый узел графа должен содержать информацию о соответствии операциям различных существующих диалектов. Это потребовало бы ручного или автоматического "протягивания"метаданных через весь процесс lowering'а, что существенно усложняет архитектуру и увеличивает техническую сложность без значимых преимуществ. По этой причине такой путь был отвергнут.

В данной работе основным и выбранным для реализации способом является создание интерфейса для основных операций. Этот подход не привязан к конкретному диалекту, а лишь служит инструментом в рамках контекста выбранных для профилирования операций. Небольшой участок кода поможет лучше понять структуру предлагаемого подхода:

```
1
2 class ProfiledOpInterface : public OpInterface<ProfiledOpInterface> {
3     public:
4     void attachProfileData(ProfileData data);
5     ProfileData getProfileData();
6     bool hasProfileData();
7 };
8
```

Листинг 1: Структура реализуемого интерфейса

Важно подчеркнуть, что данный подход не накладывает жестких ограничений на целевой (инспектируемый) диалект и может быть реализован на произвольном уровне абстракции, соответствующем интересам анализа. Кроме того, описанный паттерн органично интегрируется в модульную архитектуру MLIR, не нарушая ее принципов и не внося дополнительных межмодульных зависимостей. Подробнее о конкретной реализации интерфейса и служебных структур в контексте аннотируемых метаданных можно ознакомться в секции Описание практической части.

2.2 Получение формата данных профилирования

Следующей и не менее важной подзадачей является получение унифицированного формата данных после работы профилировщика. В этой ра-

боте предлагается расширить возможности использования привычного для PGO оптимизаций в LLVM профилировщика perf. На сегодняшний день интересующие нас фреймворки **Tensorflow**, **ONNX** и **PyTorch** предлагают готовые решения для сбора различной статистики выполнения высокоуровневых операций.

Типичный кусок кода написанный для сбора профиля исполнения модели:

```
# Profiling options set
options = tf.profiler.experimental.ProfilerOptions(
    ...
)
# Profiling start
tf.profiler.experimental.start(log_dir, options=options)

try:
    # Model run code
            ...
finally:
    # Profiling stop
    tf.profiler.experimental.stop()
```

Листинг 2: Получение профиля с помощью утилиты фреймворка **Tensorflow**

Согласно документации tensorflow-profile-plugin [см источник [5]] в настоящее время поддерживается следующий список устройств, в соответствующей конфигурации:

Profiling API	Local	Remote	Multiple workers	Hardware Platforms
TensorBoard Keras Callback	Supported	Not Supported	Not Supported	CPU, GPU
`tf.profiler.experimental` start/stop API	Supported	Not Supported	Not Supported	CPU, GPU
`tf.profiler.experimental` client.trace API	Supported	Supported	Supported	CPU, GPU, TPU
Context manager API	Supported	Not supported	Not Supported	CPU, GPU

Рис. 5: Список поддерживаемых устройств и режимов работы профилировщика **TensorFlow**.

В рамках данной работы в качестве целевой платформы предлагается использовать **CPU**. Важно отметить, что такой выбор не накладывает ограничений на область применимости разрабатываемого инструмента профилирования.

С появлением поддержки новых устройств в существующих профилировщиках, либо при использовании альтернативных решений, разработанных для отдельных устройств (например, Huawei NPU Ascend), потребуется изменить лишь этап предобработки входных данных, но не саму архитектуру обработки и анализа. Такой уровень абстракции достигается за счёт использования промежуточного формата представления данных профиля в виде сериализованного файла `.json`.

Преимущества этого подхода, а также его реализация, будут подробно рассмотрены в секции Описание практической части.

Последним замечанием к описанию подзадачи получения формата данных профиля будет представлен список возможностей **Tensorflow**.

- Анализатор конвейера данных - анализирует эффективность загрузки и предобработки данных, помогает выявить узкие места в подаче данных в модель
- Статистика TensorFlow - собирает и отображает статистику выполнения операций, показывает время работы каждой операции и использование ресурсов
- Просмотрщик трассировки - визуализирует временную шкалу выполнения операций на процессоре и GPU, позволяет увидеть параллельность и простои
- Инструмент профилирования памяти - отслеживает использование памяти GPU и RAM, помогает оптимизировать потребление памяти и избежать переполнения

Каждый инструмент решает конкретные задачи оптимизации: от анализа загрузки данных до мониторинга распределенного обучения. Второй и третий пункты из списка возможностей будут активно использованы в рамках выполнения практической части дипломной работы.

Подробнее о возможностях `TensorflowProfiler` и формате выходных данных будет рассказано в секции Обзор существующих решений.

2.3 Визуализация и анализ

После получения формата профилированных данных и создания интерфейса взаимодействия с ними в рамках операций MLIR, формируется подзадача визуализации результатов профилирования на итоговом графе исполнения. Данный этап является значимым, поскольку качество визуального представления профиля критически важно по следующим причинам:

- обоснование аналитических выводов;
- обеспечение наглядности и интерпретируемости результатов;
- возможность последующего применения визуализации при разработке оптимизаций.

Примером возможных преобразований могут являться трансформации графа, основанные на длительности выполнения операций. Графическое представление последовательности вычислений в виде ориентированного графа с аннотациями, содержащими данные профиля, позволяет локализовать узлы с наибольшей нагрузкой.

В качестве иллюстрации представлен условный граф, демонстрирующий целевой формат визуализации:

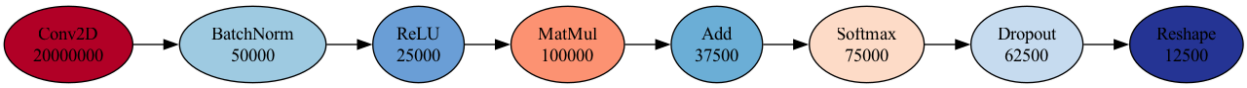

Рис. 6: Условный пример визуализации профиля исполнения.

Представленные в приложениях визуализации (см. секцию Результаты) позволяют выполнить оценку полноты и корректности реализованного инструмента. Кроме того, на основе профиля реальной модели выделяются участки, требующие приоритетного применения оптимизаций.

3 Обзор существующих решений

3.1 Введение в раздел

В условиях стремительного развития технологий машинного обучения особую актуальность приобретает задача эффективного выполнения сложных моделей на разнообразных аппаратных платформах. Одним из ключевых компонентов современной инфраструктуры стал промежуточный язык представления MLIR (Multi-Level Intermediate Representation), который служит унифицирующим звеном между высокоуровневыми описаниями моделей и низкоуровневым оптимизированным кодом. Тем не менее, существующая реализация MLIR демонстрирует ограниченные возможности для интеграции данных о производительности, что существенно затрудняет процесс комплексного анализа и оптимизации промежуточного представления. Целью данного обзора является систематический анализ современных подходов к профилированию машинного обучения, инструментам анализа и манипуляции MLIR, а также методам визуализации производительности. В рамках исследования рассматриваются профилировщики ML-моделей, инструменты анализа и модификации промежуточного представления, системы метаданных и подходы к визуализации результатов профилирования. Особое внимание уделяется выявлению возможностей автоматического аннотирования MLIR на основе данных профиля выполнения. Методология анализа базируется на следующих критериях оценки:

- **Функциональность**: степень охвата задач профилирования и аннотирования промежуточного представления.
- **Архитектурная совместимость**: уровень интеграции с инфраструктурой MLIR/LLVM.
- **Масштабируемость**: способность эффективно работать с моделями большого размера.
- **Расширяемость**: возможность кастомизации и добавления новых метрик.
- **Применимость**: соответствие целевой платформе (CPU) и экосистеме TensorFlow.

Анализ проводится по категориям инструментов, после чего осуществляется сравнительный анализ, направленный на выявление ключевых пробелов и перспектив интеграции различных решений.

3.2 Профилировщики машинного обучения

3.2.1 TensorFlow Profiler

TensorFlow Profiler [5] представляет собой комплексный инструмент для анализа производительности, интегрированный непосредственно в экосистему TensorFlow. Архитектура профилировщика основана на тесной интеграции с TensorFlow Runtime, что обеспечивает сбор метрик на уровне

отдельных операций и ядер вычислений при поддержке различных вычислительных бэкендов, включая CPU, GPU и TPU. Выходные данные профилировщика формируются в формате trace events (JSON), содержащем структурированную информацию о времени выполнения, использовании памяти и утилизации вычислительных ресурсов. Такой формат позволяет проводить детальный анализ производительности на уровне отдельных операций модели. Несмотря на широкие возможности, TensorFlow Profiler имеет ряд ограничений, существенных в контексте интеграции с MLIR: жесткая привязка к TensorFlow Runtime, отсутствие прямой поддержки промежуточного представления MLIR и сложность извлечения метрик для отдельных операций MLIR. Таким образом, применение данного инструмента ограничено рамками экосистемы TensorFlow.

3.2.2 PyTorch Profiler

PyTorch Profiler [6] предлагает альтернативный подход к профилированию, основанный на использовании Autograd profiler для отслеживания прямого и обратного распространения, а также Kineto backend для низкоуровневого анализа производительности. Интеграция с TensorBoard обеспечивает интерактивную визуализацию результатов профилирования, включая поддержку распределённого профилирования для многоузловых конфигураций. Однако применимость PyTorch Profiler к задачам, связанным с MLIR, ограничена различиями в архитектуре промежуточного представления и ориентацией на специфичные для PyTorch структуры данных.

3.2.3 XLA Profiler и связь с MLIR

Особый интерес представляет XLA (Accelerated Linear Algebra) Profiler, который демонстрирует тесную связь с MLIR в процессе генерации кода. XLA использует MLIR в качестве промежуточного представления, что создаёт основу для интеграции профильных данных. Промежуточное представление XLA HLO (High-Level Optimizer) может быть транслировано в MLIR, обеспечивая прослеживаемость от исходного графа TensorFlow до низкоуровневого кода. XLA Profiler предоставляет возможности анализа на уровне операций HLO, исследования паттернов fusion и различных оптимизаций. Высокую актуальность данного инструмента для пайплайнов, основанных на MLIR, определяет общность инфраструктуры и возможность переноса подходов профилирования.

3.2.4 Системные профилировщики

Системные профилировщики, такие как Intel VTune Profiler, Linux perf и LLVM XRay, обеспечивают микроархитектурный анализ и низкоуровневое профилирование. Intel VTune Profiler позволяет выявлять узкие места (hotspot analysis), анализировать паттерны доступа к памяти и интегрироваться с ML-фреймворками через API. Linux perf использует sampling-

based профилирование с генерацией flame graphs для визуализации, однако имеет ограничения при анализе высокоуровневых ML-операций. LLVM XRay предоставляет функцию трассировки на уровне функций и демонстрирует потенциал для инструментирования кода, сгенерированного из MLIR, что особенно важно для связывания профильных данных с промежуточным представлением.

3.3 Инструменты анализа и манипуляции MLIR

3.3.1 Базовые инструменты MLIR

MLIR [3] представляет собой гибридное промежуточное представление, поддерживающее множество требований в унифицированной инфраструктуре. Базовые инструменты MLIR включают `mlir-opt` для трансформации промежуточного представления с возможностью добавления пользовательских проходов (pass'ов), а также `mlir-translate` для конвертации между различными форматами, включая импорт TensorFlow SavedModel в MLIR. `mlir-opt` обеспечивает расширяемую архитектуру для добавления новых оптимизационных проходов, что создаёт предпосылки для интеграции проходов, работающих с профильными данными. Тем не менее, существующие возможности `mlir-opt` ограничены в контексте интеграции динамических данных профилирования.

3.3.2 Визуализация MLIR

Инструмент `mlir-to-dot` предназначен для генерации представления MLIR-кода в формате GraphViz, обеспечивая статическую визуализацию структуры промежуточного представления. Основными ограничениями данного инструмента являются отсутствие поддержки метрик времени выполнения и цветового кодирования производительности, что существенно снижает его применимость для анализа производительности. Альтернативные подходы включают разработку пользовательских проходов для генерации аннотированных графов и интеграцию с внешними инструментами визуализации, что требует значительных дополнительных усилий по разработке.

3.3.3 Проекты на базе MLIR

IREE (Intermediate Representation Execution Environment) [7] [8] представляет собой MLIR-based end-to-end компилятор и среду выполнения, который компилирует MLIR в исполняемый код со встроенными возможностями профилирования runtime. IREE обеспечивает инструментирование MLIR-проходов для анализа времени компиляции, что демонстрирует потенциал интеграции профилирования в инфраструктуру MLIR. Основные ограничения IREE связаны с фокусом на задачах инференса, а не на детальном анализе производительности промежуточного представления. В сообществе MLIR обсуждается необходимость обеспечения прослеживаемости между компонентами сгенерированного кода и операци-

ями входной спецификации. XLA-MLIR обеспечивает интеграцию XLA-компилятора [9] с MLIR, создавая предпосылки для переноса возможностей XLA-профилирования в контекст MLIR. ByteIR демонстрирует промышленное применение MLIR с опытом интеграции профилирования и оптимизаций.

3.4 Метаданные и аннотации в MLIR

3.4.1 Системы метаданных MLIR

MLIR предоставляет развитую систему метаданных, включающую типизированные атрибуты операций, информацию о местоположении (location) для отслеживания источника операций, а также traits и interfaces для определения свойств операций. Атрибуты представляют собой статические метаданные, которые могут быть расширены для включения профильных данных, однако их статическая природа ограничивает возможности работы с динамическими runtime-метриками. Информация о местоположении в MLIR создаёт потенциал для связывания профильных данных с конкретными операциями промежуточного представления. Traits и interfaces обеспечивают возможности для определения профилируемых операций и расширения функциональности существующих диалектов.

3.4.2 Опыт LLVM в Profile-Guided Optimization (PGO)

LLVM предоставляет развитую инфраструктуру для оптимизации на основе профиля (PGO) с workflow `-fprofile-generate` / `-fprofile-use` [10] и специализированным форматом профильных данных. Метаданные профиля включают branch weights, block frequencies и интеграцию с оптимизационными проходами. Применимость LLVM PGO к MLIR ограничена различиями в уровне абстракции и необходимостью адаптации для ML-специфичных метрик, таких как время выполнения тензорных операций и характеристики использования памяти.

3.4.3 Ограничения текущих подходов

Анализ существующих подходов выявляет отсутствие стандартизированного формата профильных аннотаций в MLIR, ограниченную поддержку динамических метрик и необходимость ручной интеграции с профилировщиками, что создаёт значительные барьеры для практического применения. Традиционные методы визуализации производительности включают flame graphs для иерархического представления времени выполнения и heatmaps для цветового кодирования метрик производительности. Flame graphs имеют ограничения при представлении графов операций ML из-за различий в структуре данных, тогда как heatmaps демонстрируют применимость к визуализации вычислительных графов. TensorBoard Graph Visualization обеспечивает интерактивную визуализацию графов TensorFlow

с интеграцией профильных данных через overlays, однако ограничена привязкой к формату TensorFlow. Netron служит универсальным визуализатором нейронных сетей, но не предоставляет интеграции с профильными данными. Специфические требования к визуализации MLIR включают поддержку иерархической структуры (модули, функции, блоки), цветовое кодирование «горячих» операций, интерактивность для детального анализа метрик и масштабируемость для больших графов.

3.4.4 Анализ пробелов

Анализ выявляет критические пробелы в существующей инфраструктуре:

- Отсутствуют готовые решения для связки профилировщиков ML с промежуточным представлением MLIR.
- Существующие инструменты не поддерживают цветовое кодирование производительности на уровне операций MLIR.
- Метаданные MLIR не предназначены для работы с динамическими runtime-метриками.
- Необходимость использования множества разрозненных инструментов для полного workflow.

3.4.5 Научная и практическая значимость

Разработка средства автоматического аннотирования MLIR на основе профиля выполнения обладает значимостью в следующих аспектах:

- Создание моста между профилированием ML-моделей и анализом промежуточного представления MLIR.
- Предоставление данных для оптимизаций на основе профиля на уровне MLIR.
- Визуализация производительности для быстрого выявления узких мест.

3.4.6 Ожидаемые результаты

Реализация предлагаемого средства обеспечит:

- Повышение эффективности анализа производительности ML-моделей.
- Создание основы для будущих оптимизаций на основе профильных данных.
- Расширение инструментария MLIR для практических задач анализа производительности.

4 Описание практической части

4.1 Профилирование и сериализация

Одной из первых и ключевых задач практической части работы стало унифицирование формата полученного профиля выполнения. Ввиду разнообразия существующих решений для профилирования и в целях соответствия поставленным задачам, наиболее подходящим инструментом трассировки исполняемых операций был выбран **TensorFlow Profiler (TF Profiler)**. Он предоставляет средства для отслеживания времени выполнения высокоуровневых операций, что позволяет с относительной лёгкостью сопоставить полученные профили с операциями графа MLIR.

Промежуточным этапом на пути к этому стало создание собственного представления трассы операций в виде направленного ациклического графа .

4.1.1 Что было сделано

В рамках выполнения данной задачи был разработан универсальный подход к промежуточной сериализации, предполагающий сохранение событий в формате направленного ациклического графа. Основной целью разработки являлось восстановление зависимостей между операциями на основе анализа их временных характеристик — времени начала и длительности исполнения. Это позволило определить последовательность, а также возможный параллелизм выполнения операций.

Дополнительно было проведено детальное изучение функциональных возможностей инструмента TensorFlow Profiler. Исследованы особенности его применения, реализован вызов API, а также представлен и описан формат выходных данных, используемых на этапе сериализации.

4.1.2 Выходной формат профиля и возможности профилировщика

Для получения профиля была разработана простая программа на языке Python, выполняющая обучение модели. Структура и логика самой модели не являются предметом данного исследования и далее рассматриваться не будут. Ключевым аспектом является формат сохраняемых данных.

Профилировщик TensorFlow предоставляет возможность настройки через специальную структуру параметров — **ProfilerOptions**:

```
options = tf.profiler.experimental.ProfilerOptions(
    host_tracer_level=2,
    python_tracer_level=1,
    device_tracer_level=1
)
```

Листинг 3: Опции профилирования **TF Profiler**

Значения уровней профилирования соответствуют степени детализации: чем выше значение, тем более подробные данные собирает профили-

ровщик. В рамках данной работы были задействованы следующие классы событий:

- Трассировщик хоста — события, связанные с планировщиком операций и потоками исполнения на уровне хост-системы.
- Трассировщик Python — вызовы функций на языке Python и высокоуровневые операции TensorFlow.
- Трассировщик устройства — данные об использовании памяти, загрузке CPU и других характеристиках вычислительных устройств.

Результатом работы профилировщика является двоичный формат на основе технологии Protocol Buffers, разработанной компанией Google. TensorFlow предоставляет средства для преобразования файлов `.pb` в человекочитаемый формат `.json`. Полученный файл представляет собой последовательность событий, каждое из которых включает как временные характеристики, так и ссылку на метаданные.

```
{
"metadata_id": "1327",
"offset_ps": "120000",
"duration_ps": "20000000"
},
```

Листинг 4: Данные события **TF Profiler**

```
{
"id": "1327",
"name": "$tensorflow.python.
    _pywrap_tfe
    TFE_Py_TapeSetRecordOperation"
},
```

Листинг 5: Метаданные события **TF Profiler**

Путём варьирования уровней детализации удалось добиться включения в профиль названий интересующих операций, что впоследствии использовалось для аннотирования. Следующим этапом стало связывание событий с метаданными, а также построение DAG на основе временных характеристик (времени начала и продолжительности операций).

4.1.3 DAG: построение и сериализация

Поскольку каждое событие содержит время начала и длительность, оно может быть представлено как временной отрезок. При добавлении события в граф осуществляется проверка на наличие перекрытия с уже добавленными отрезками. Такие перекрытия определяют уровни графа: события, попавшие на один уровень, интерпретируются как потенциально параллельные, тогда как переход на следующий уровень отражает зависимость во времени исполнения операций.

Для представления операций в виде графа был разработан собственный класс **MlirGraph**, вершины которого содержат агрегированную информацию об операциях. Логика построения графа реализована в модуле **traceReader.py**, предоставляющем также интерфейс командной строки (CLI) для удобства использования.

Пример вызова утилиты:

```
python3 traceReader.py --path-to-trace trace.pb(.json) --store-output
    mlir_graph.json
```

Листинг 6: Пример использования программы traceReader.py

Результатом работы является сериализованный граф, содержащий следующую информацию:

```
{
"name": "$tensorflow.python.
    _pywrap_tfe TFE_DeleteExecutor",
"ts": 37501000000,
"duration": 2000000,
"id": 483,
"adj": [484, 485, 486, 487]
},
```

Листинг 7: Формат вершины MLIR-графа

```
{
    "edgeFrom": 483,
    "edgeTo": 484
},
{
    "edgeFrom": 483,
    "edgeTo": 485
}
```

Листинг 8: Формат ребра MLIR-графа

Помимо времени выполнения и идентификатора, каждая вершина включает имя операции и список смежных узлов (смежность указывает на логические или временные зависимости). Такой граф может быть повторно считан из файла, а при необходимости — преобразован в формат **NetworkX** для последующего анализа и разработки алгоритмов трансформации графов.

4.2 Аннотирование операций

Классовая организация, высокая модульность и гибкая архитектура MLIR делают возможным простое расширение функциональности существующих диалектов. В рамках данной работы была реализована система аннотирования операций диалекта `tf` (TensorFlow) в соответствии с архитектурными принципами MLIR.

4.2.1 Что было сделано

В рамках работы была выполнена модификация диалекта `TensorFlow:tf`, заключающаяся в добавлении интерфейса для работы с аннотациями. Также был реализован и зарегистрирован специализированный pipeline, сопровождаемый набором опций, позволяющим задать путь к сериализованному файлу формата `JSON` посредством аргумента командной строки. Кроме того, обеспечена поддержка добавления пользовательских проходов (`passes`), осуществляющих анализ и трансформацию на основе ранее внедрённых аннотаций.

4.2.2 Интерфейс для работы с аннотациями

В качестве основного подхода была выбрана реализация интерфейса непосредственно внутри операций. Такой метод не привязан к конкретному

диалекту и, при необходимости, может быть легко адаптирован для других диалектов в будущем.

Фреймворк **TensorFlow** использует систему **TableGen** для генерации заголовочных файлов, определения классов, описания операций и трейтов. TableGen позволяет избежать дублирования C++-кода, предоставляя шаблонный, компактный способ описания, на основе которого во время компиляции автоматически создаются соответствующие заголовочные файлы.

Ниже представлено описание разработанного интерфейса аннотирования на языке TableGen:

```
def TF_ProfilerAnnotationsInterface : OpInterface<"ProfilerAnnotationsInterface
    "> {
let description = [{Methods to get/attach profiler data as annotations}];
let methods = [
    InterfaceMethod<
        /*retTy=*/"void",
        /*methodName=*/"AttachProfilerData",
        /*args*/(ins "const ProfilerData&":$data)
    >,
    ...
];
}
```

Листинг 9: Описание интерфейса аннотирования на языке TableGen

Для краткости изложения методы `GetProfilerData` и `HasProfilerData` в данном фрагменте опущены, однако они также были реализованы.

В качестве аннотаций было решено использовать временные характеристики операций: время начала и длительность исполнения.

4.2.3 Трейты: назначение и применение

Трейты в MLIR представляют собой механизм, позволяющий присваивать операциям определённые свойства. Эти свойства могут использоваться, например, для проверки типов операндов: если операнды не удовлетворяют требованиям трейта, они не могут быть переданы соответствующей операции. Трейты также обеспечивают связь между операциями и интерфейсами, позволяя гарантировать их совместимость на этапе компиляции.

В рассматриваемой работе трейты используются для реализации интерфейса аннотирования вне тела самих операций. Это позволяет значительно сократить количество зависимостей в коде и способствует модульной повторной используемости компонентов, аналогично паттерну *dependency injection*, хорошо известному по экосистеме Java (например, в Spring Framework через механизмы Beans).

Трейты реализуются на основе паттерна **CRTP** (Curiously Recurring Template Pattern), обеспечивающего статический полиморфизм. Базовый шаблонный класс принимает в качестве параметра дочерний класс, что позволяет ему вызывать методы, определённые в этом дочернем классе. Ниже приведён минимальный пример реализации данного паттерна:

```
template <class T>
struct Base {
    void interface() {
        // ...
        static_cast<T*>(this)->implementation();
        // ...
    }

    static void static_func() {
        // ...
        T::static_sub_func();
        // ...
    }
};

struct Derived : public Base<Derived> {
    void implementation();
    static void static_sub_func();
};
```

Листинг 10: Демонстрация принципа CRTP

На основе вышеуказанного подхода был реализован собственный трейт, обеспечивающий поддержку интерфейса аннотирования:

```
template <typename ConcreteType>
class ProfileAnnotation : public TraitBase<ConcreteType, ProfileAnnotation> {
    public:
        // Implements methods required for TF_ProfilerAnnotationsInterface
        void AttachProfilerData(const ProfilerData& data);

        ProfilerData GetProfilerData();

        bool HasProfilerData();
};
```

Листинг 11: Трейт аннотирования операций MLIR

В качестве результата разработки, ниже приведена реализация метода `AttachProfilerData`, предназначенного для присвоения аннотаций операции:

```
void AttachProfilerData(const ProfilerData& data) {
    Operation* op = this->getOperation();
    MLIRContext* context = op->getContext();

    Builder builder(context);

    auto tsAttr = builder.getI64IntegerAttr(data.timestamp);
    auto durationAttr = builder.getI64IntegerAttr(data.duration);

    NamedAttribute attrs[] = {
        builder.getNamedAttr("ts", tsAttr),
        builder.getNamedAttr("dur", durationAttr)
```

```
    };

    auto dictAttr = DictionaryAttr::get(context, attrs);
    op->setAttr("profiler_data", dictAttr);
}
```

Листинг 12: Реализация метода AttachProfilerData

По средствам описанного паттерна был реализован расширяемый интерфейс аннотирования операций. В дальнейшем, трейт может быть использован с любой операцией TensorFlow, представляющей интерес для анализа трансформаций графа.

4.2.4 Регистрация проходов и создание pipeline для аннотирования операций

Для того чтобы аннотации, добавленные в операции, появились в генерируемом промежуточном представлении (IR), необходимо зарегистрировать собственный проход компиляции (далее — pass) в менеджере проходов (pass manager). Этот механизм позволяет эффективно обрабатывать и изменять IR на различных этапах компиляции.

В рамках данной работы было принято решение вынести изменения IR, касающиеся профилирования и оптимизации на основе профиля исполнения (PGO, Profile-Guided Optimization), в отдельную последовательность проходов, которая будет называться pipeline. Это решение имеет несколько ключевых преимуществ. Во-первых, оно помогает организовать оптимизации в логическую структуру, разделяя их по категориям. Во-вторых, такой подход обеспечивает гибкость в добавлении новых проходов и оптимизаций, которые могут быть реализованы в будущем для конкретных задач машинного обучения.

Создание и регистрация pass-ов и pipeline в MLIR схожи с подходом, используемым в LLVM, что делает систему гибкой и хорошо документированной. Разработка и настройка этих механизмов является важной частью работы по интеграции аннотирования в процесс компиляции.

В рамках задачи по созданию pass-ов и pipeline были выполнены следующие шаги:

- Реализация pass-а для рекурсивного прохода по операциям и добавления аннотаций.
- Регистрация pipeline, включающего на данный момент только проход для аннотирования операций.

Ниже представлено объяснение каждого из этих шагов в контексте выполнения задач.

Pass — это ключевая концепция в MLIR и LLVM, которая представляет собой отдельный шаг обработки, который может изменять, оптимизировать или анализировать промежуточное представление (IR). В данном

случае, основной задачей pass-а является рекурсивный обход операций и добавление аннотированных данных о времени выполнения.

Процесс рекурсивного прохода по операциям включает следующие этапы:

1. Программы, описанные в виде IR, представляют собой граф операций, где каждая операция может содержать множество подопераций (дочерних операций).
2. Pass рекурсивно проходит по всем операциям и их подоперациям, извлекая и присваивая аннотированные данные (например, время начала и продолжительности выполнения).
3. На каждой операции добавляются метаданные, которые затем сохраняются в IR и могут быть использованы для анализа.

Пример кода для реализации такого pass-а может выглядеть следующим образом:

```
void AnnotateOperationsProfilePass::runOnOperation() {
  ModuleOp op = getOperation();

  op.walk([&](Operation* nestedOp) {
    if (nestedOp->getDialect() &&
        nestedOp->getDialect()->getNamespace() == "tf") {
        if (nestedOp->hasTrait<ProfileAnnotation>()) {
            ProfilerData data(0, 0);
            readProfilerData(&data, nestedOp);
            nestedOp->AttachProfilerData(data);
        }
    }
  });
}
```

Листинг 13: Проход аннотирования операций

Вторым важным шагом является создание и регистрация **pipeline**, который будет включать все проходы (pass-ы), необходимые для обработки программы. **Pipeline** — это последовательность проходов, которая выполняется в заданном порядке, позволяя эффективно обрабатывать код. В данной работе pipeline был сконфигурирован для включения только одного pass-а, отвечающего за аннотирование операций. Это позволяет организовать обработку профиля таким образом, что на следующем этапе можно легко добавить новые проходы, например, для анализа или оптимизации на основе собранных данных.

Пример кода для регистрации pipeline:

```
void CreateTFProfileGuidedPipeline(OpPassManager &pm,
                                   const ProfileGuidedPipelineOptions &options) {
    if (options.enable_profile) {
        OpPassManager &module_op_pm = pm.nest<ModuleOp>();
        module_op_pm.addPass(TF::CreateAnnotateOperationsProfilePass(options.
    path_to_profile));
        // perform here profile guided transformations
    }
}
```

Листинг 14: Регистрация PGO pipeline-а

Таким образом, была реализована модульная и расширяемая система аннотирования операций диалекта TensorFlow, использующая архитектурные возможности MLIR: интерфейсы, трейты и шаблонное программирование на C++.

4.3 Визуализация DAG

4.3.1 Что было сделано

В рамках данной работы был разработан инструмент для визуального анализа сериализованного направленного ациклического графа (DAG, от англ. Directed Acyclic Graph), реализованный в виде модуля **plotGraph.py**. Данный модуль принимает на вход сериализованный файл в формате JSON, содержащий описание DAG, который является результатом работы компонента `traceReader.py`. Кроме того, в коде было реализовано разделение вершин графа с использованием цветовой кодировки в зависимости от времени выполнения операций, связанных с функциями-вершинами DAG. В возможности модуля входит также сохранение графа в следующих форматах: **.graphml**, **.dot**, **.svg**.

Перекрашивание вершин графа в различные цвета, соответствующие времени выполнения операций, позволяет визуально выделить наиболее ресурсоёмкие участки («красные» зоны). Такой подход способствует выявлению подграфов, требующих детального анализа и последующей трансформации. Иными словами, разрабатываемый инструмент предоставляет возможность автоматического определения входных точек для применения оптимизаций, основанных на профиле исполнения программ искусственного интеллекта.

4.3.2 Пример использования

Основным преимуществом предложенного решения является возможность визуализации и детального анализа результатов работы TensorFlow Profiler. Иными словами, в зависимости от выбора уровня профилирования, возможно как изменять, так и расширять область визуального анализа.

К примеру, при настройке уровней профилирования следующим образом:

```
options = tf.profiler.experimental.ProfilerOptions(
    host_tracer_level=0,
    python_tracer_level=1,
    device_tracer_level=0
)
```

Листинг 15: Регистрация PGO pipeline-а

можно получить граф вызовов функций, реализованных на языке Python. Пример вызова утилиты:

```
python3 plotGraph.py (-s) --path-to-mlir-graph mlir-dag.json -o out.svg
```

Листинг 16: Пример использования программы plotGraph.py

Пример визуализации, полученной в результате работы утилиты:

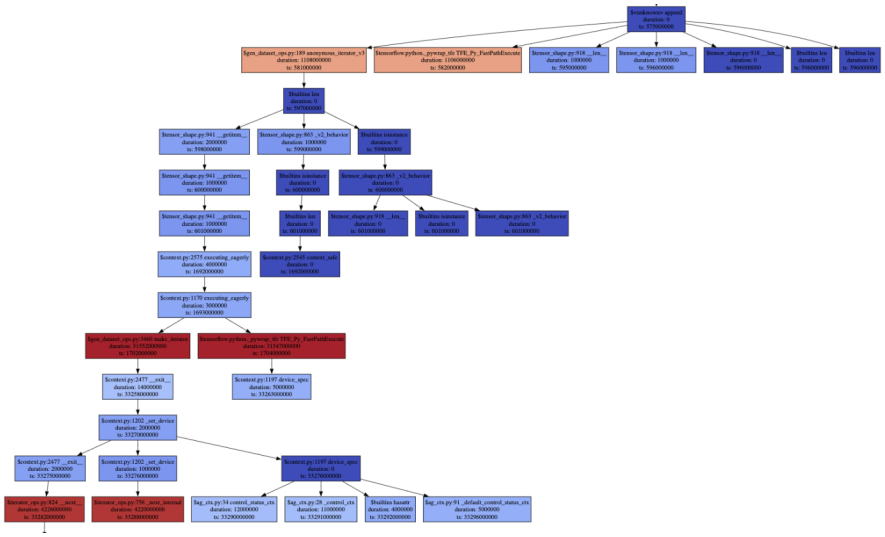

Рис. 7: Пример результата работы программы plotGraph.py. Граф вызовов функций Python

5 Заключение

В рамках данной работы было подробно рассмотрено современное решение для разработки компиляторов программ машинного обучения — MLIR. Благодаря модульной и расширяемой архитектуре удалось интегрировать собственный пайплайн, который добавляет аннотации времени выполнения высокоуровневых операций. Кроме того, в ходе работы были получены данные о выполнении операций на центральном процессоре с использованием TensorFlow Profiler, которые затем были сохранены в унифицированном формате (DAG). Преимущество данного подхода было продемонстрировано на примере разработанной утилиты для визуального анализа последовательности произвольных событий, связанных с временем исполнения. Все разработанные модули были объединены в единый инструмент, который предоставляет возможность автоматического аннотирования высокоуровневых операций собранной информацией о профилировании. Сравнивая полученные результаты с перечнем поставленных целей, можно утверждать, что все цели работы были успешно достигнуты. Разработанное решение может быть использовано сторонними специалистами на практике для анализа и оптимизации программ машинного обучения с использованием PGO.

6 Результаты

Предоставить все полученные результаты по итогу проделанных работ

1. аннотированный mlir
2. граф операций с обозначением узких мест
3. ссылку на гитхаб с подробным README.md с описанием цельного запуска (full pipeline)

Список литературы

[1] *ONNX Runtime:*. Accelerated Machine Learning. — https://opensource.microsoft.com/blog/2019/10/30/announcing-onnx-runtime-1-0/. — 2019.

[2] *LLVM Project:*. Intermediate Representation. — https://llvm.org/docs/. — 2003.

[3] *MLIR Project*. MLIR: Multi-Level Intermediate Representation. — https://mlir.llvm.org/. — 2025.

[4] *TensorFlow Team*. MLIR: A new intermediate representation and compiler framework. — https://blog.tensorflow.org/2019/04/mlir-new-intermediate-representation.html. — 2019.

[5] *TensorFlow Team*. Introducing the new TensorFlow Profiler. — https://blog.tensorflow.org/2020/04/introducing-new-tensorflow-profiler.html. — 2020.

[6] *PyTorch Team*. PyTorch Profiler. — https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html. — 2025.

[7] *IREE Project*. IREE: Intermediate Representation Execution Environment. — https://iree.dev/. — 2025.

[8] *IREE Project*. Profiling with Tracy. — https://iree.dev/developers/performance/profiling-with-tracy/. — 2025.

[9] *XLA Project:*. ML programs acceleration with XLA compiler. — https://openxla.org. — 2025.

[10] *LLVM Project:*. Experience in PGO with LLVM. — https://llvm.org/devmtg/2020-09/slides/PGO_Instrumentation.pdf. — 2020.