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

Физтех-школа радиотехники и компьютерных технологий
Кафедра системного программирования ИСП РАН
Лаборатория (laboratory name)

Выпускная квалификационная работа бакалавра

Разработка компилятора нейронных сетей на основе инфраструктуры MLIR для процессора с матричной архитектурой

Автор:
Студент Б01-009 группы
Вязовцев Андрей Викторович

Научный руководитель:
Кандидат технических наук
Маркин Юрий Витальевич

Научный консультант:
Кандидат технических наук
Кулагин Иван Иванович

Москва 2024

Аннотация

Разработка компилятора нейронных сетей на основе инфраструктуры MLIR для процессора с матричной архитектурой
Вязовцев Андрей Викторович

Краткое описание задачи и основных результатов, мотивирующее прочитать весь текст.
Пункт 5: разделить на два: 1. специализированные процессоры для выполнения (inference) нейронных сетей (мб GPU, LG и DaVinci, сюда перенести п. 4 и 7, рассказать про методы оптимизации (двойная буферизация)), сосредоточить внимание на юнитах, привести примеры кода. 2. компиляторы а. Tensorflow, XLA, Onnx, Pytorch, стандарты: TOSA, StableHLO, проблематика: модель -> план, трансляция в код б. Полиэдральная компиляция, планирование, преобразования циклов. Промышленное внедрение: TVM, Halide, poly, polygeist. Рассмотрение этих компиляторов, примеры (такое будет в диалекте transform). Есть возможность создавать свои операции, но это не удобно. в. Заключение: компиляторы повторяются, большой количество IR, появление LLVM MLIR.
Объединить п. 6, 8, 9, 10 в один и разделить на подпункты. Пайплайны хоста и асценда. Примеры? Соответствие операторов и ассемблерных инструкций. Проблемы: подготовка данных, эффективное использование памяти, параллелизм, синхронизация, расписание (стратегия выполнения цикла).
Заключение, дальнейшая работа.

Abstract

FIXME: English abstract?

Содержание

1 Введение

Нейронные сети в последниее время испытывают большой подъём. Это происходит, прежде всего, благодаря успехам Chat GPT, которая показала новые возможность для обработки естественной речи. Стоит отметить, что развитие этой сферы происходит не только за счёт совершенствования точности ответов нейронных сетей. Например, разбатываются процессоры с матричной архитектурой, которые могут быть встроены в смартфоны. Очевидно, что такие решения будут востребованы на мобильном рынке, который занимает крупную часть всего IT-рынка.

Для использования таких процессоров необходим широкий набор утилит, в том числе и компиляторы. Основной задачей любого компилятора является получение наиболее оптимального с точки зрения производительности машинного кода при сохранении всех свойств исходной программы. Заметим, что в таких компиляторах помимо традиционных техник оптимизации, таких как удаление мёртвого кода, распостранения констант, сокращения общих подвыражений и других, должны применяться другие техники, связанные с математическими свойствами тензоров и спецификой целевой архитектуры.

В силу описанных выше причин идут активные исследования в области компиляторов для нейронных сетей, в том числе и нашей лабораторией. В данной работе будет исследованы особенности целевой архитектуры и представлены способы генерации эффективого машинного кода.

2 Постановка цели и задач

Цель исследования: разработать компилятор нейронных сетей для процессоров Ascend, основанных на архитектуре DaVinci, с использованием инфраструктуры LLVM MLIR и обеспечить генерацию эффективного машинного кода в нём. Для достижения данной цели были поставлены следующие задачи:

1. Исследовать архитектуру современных популярных нейронных сетей и типичные для них операции.
2. Исследовать подходы к эффективному исполнению нейронных сетей, в том числе использование специальных процессоров (NPU), компиляторов с разными целевыми архитектурами (CPU, GPU, NPU), узнать их особенности и используемые в них оптимизации.
3. Изучить инфраструктуру LLVM MLIR и предоставляемые ею возможности для написания собственного компилятора.
4. Исследовать архитектуру DaVinci, принцип работы нейроматричного процессора и её язык ассемблера.
5. Разработать набор операторов для целевой архитектуры DaVinci в инфраструктуре MLIR.
6. Исследовать и предложить методы генерации оптимального машинного кода для некоторых типичных операций нейронных сетей.
7. Реализовать наиболее эффективные способы генерации машинного кода, исследовать их производительность.

3 Обзор современных нейронных сетей

3.1 Общие соображения

Искусственная нейронная сеть — математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации биологических нейронных сетей — сетей нервных клеток живого организма. Этот принцип отражается в её устройстве: нейронная сеть состоит из нескольких слоёв, каждый из которых принимает информацию с предыдущего, обрабатывает её каким-то образом, а затем передаёт её на следующий слой.

Благодаря новым исследованиям в этой области, нейронные сети нашли большое количество применений в разных сферах жизни. В медицине они позволяют проводить более точную диагностику заболеваний (например, онкологии), создавать портативные устройства для диагностики (например, для проведения ЭКГ). Они используются для обработки больших данных в разных исследовательских областях, таких как астрономия и геологоразведка. Также они упрощают жизнь в робототехнике и автоматизации производства.

Рассмотрим наиболее популярные архитектуры нейронных сетей, их основные особенности.

3.2 Обработка естественного языка

Обработка естественного языка — общее направление искусственного интеллекта и математической лингвистики. Оно изучает проблемы компьютерного анализа и синтеза текстов на естественных языках. Применительно к искусственному интеллекту анализ означает понимание языка, а синтез — генерацию грамотного текста. Одним из подходов к решению данной задачи стала архитектура трансформер, представленная компанией Google в 2017 году. Эта архитектура используется в переводчиках (например, от компаний Яндекс и Google) и в чат-ботах (например, Chat GPT). BERT, GPT-3, LLaMA — модели, основывающиеся на архитектуре трансформер.

Многие из таких моделей можно скачать, после чего изучить их внутреннее устройство. Нами была выбрана модель BERT. Не погружаясь в детали реализации можно заметить, что подавляющее большиство операций в ней занимают умножения матриц. По этой причине эта операция была выбрана для дальнейшего исследования.

3.3 Компьютерное зрение

Компьютерное зрение — теория и технология создания машин, которые могут производить обнаружение, отслеживание и классификацию объектов. Распознавание изображений может быть полезно в любой сфере, например, в сельском хозяйстве — для обнаружения болезни растений, в области безопасности — для обнаружения преступников и т.д.

Одно из наиболее популярных решений в этой области — свёрточные нейронные сети. Принцип их работы схож с работой зрительной коры головного мозга. Основываются они на операции свёртки (конволюции). В функциональном анализе она применяется к двум функциям и возвращает третью, соответствующую их взаимной корреляции. Проще говоря, их можно интерпретировать как «схожесть» двух функций.

В нейронных сетях свёртка применяется к изображениям, её схему можно увидеть ниже. На часть изображения «накладывается» ядро, т.е. эта часть скалярно умножается на ядро. Получившийся результат является каким-то признаком, он записывается в результирующую матрицу — матрицу выходных признаков (output feature map). Стоит

отметить, что общий случай свёркти несколько сложнее, более подробно этот вопрос будет рассмотрен в соответствующей главе.

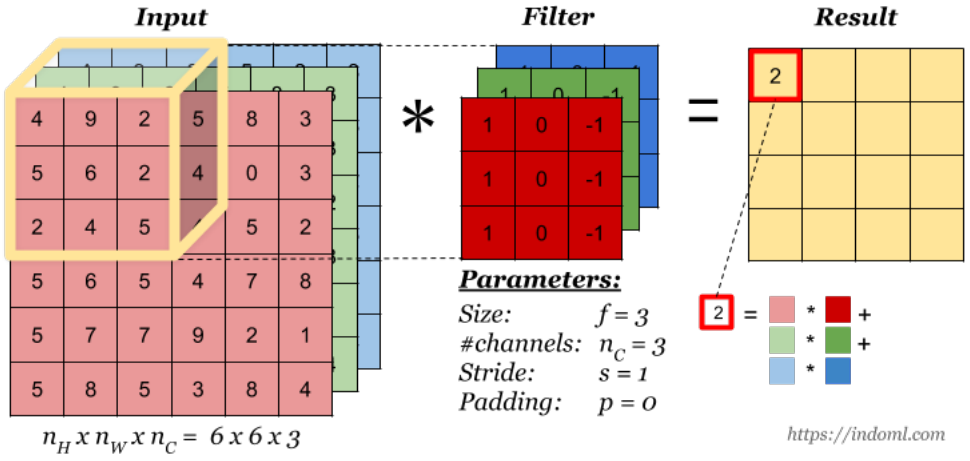

Рис. 1: Операция свёркти для изображения с тремя цветами

Существует большое количество свёрточных нейронных сетей: LeNet-5, AlexNet, VGG, GoogLeNet, ResNet, Inception. Нами была выбрана ResNet для дальнейшего исследования. Она представлена несколькими вариантами, которые отличаются количеством слоёв, а следовательно, точностью вычислений и размерами весов модели. Модель с 18 слоями представлена на рисунке ниже. Как видно из рисунка, в ней используются только операции свёркти. Но внутри них также есть операции сложения и *Relu*, где

$$Relu(x) = \begin{cases} x, & x \geqslant 0 \\ 0, & x < 0 \end{cases}$$

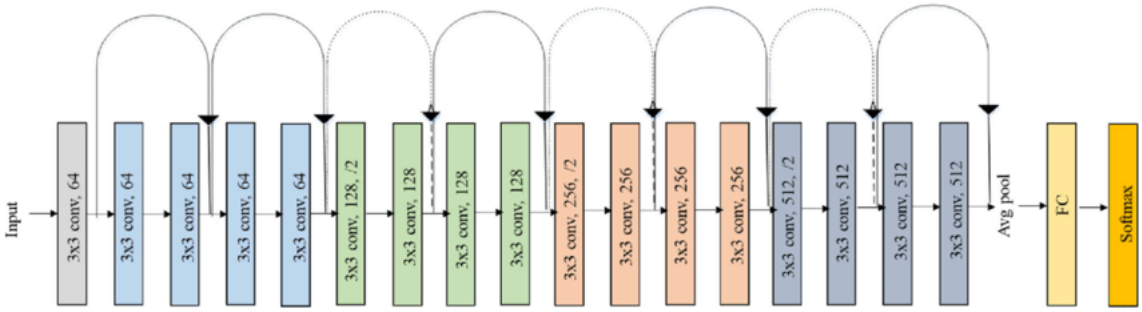

Рис. 2: Модель ResNet18

4 Обзор существующих компиляторов нейронных сетей

4.1 Инфраструктуры и стандарты

Для удобного проектирования, обучения и запуска нейронных сетей создаются фреймворки (т. е. библиотеки), поддерживающие большое количество операций нейросетей и содержащие некоторые заранее натренированные модели и открытые датасеты (наборы данных), использованные для их обучения. Наиболее популярными из таких являются *PyTorch* и *TensorFlow*. Обучение и исполнение моделей в этих фреймворках возможно как на CPU, так и на GPU.

Очевидно, что PyTorch и TensorFlow являются не единственными в своём роде. По этой причине добавление возможностей исполнения на специализированном процессоре в каждый фреймворк является нерациональным. Для унификации работы с фреймворками, создаются стандарты, такие как *ONNX*, *TOSA* или *StableHLO*. Они создают единую точку входа для специализированных компиляторов: сначала модель из любого фреймворка конвертируется в модель на языке стандарта, и именно эту модель принимает на вход компилятор для создания архитектурно-специфичного кода. Среди таких компиляторов можно выделить *XLA*, *MindSpore*, *ONNX runtime*. Но все они также нацелены на исполнение на самых распростаннённых архитектурах: x86-64 (в т. ч. с векторизацией при помощи AVX), ARM, GPU (в т. ч. с использованием технологий CUDA от Nvidia и ROCm от AMD). Их использование в качестве основы для создания компилятора для архитектуры DaVinci означало бы написание полного цикла компиляции из стандарта до ассемблерных инструкцию практически «с нуля» и не давало бы никаких преимуществ.

4.2 Оптимизации и полиэдральная компиляция

Как было упомянуто ранее, одними из основных операций в нейронных сетях являются умножения матриц и свёртки. Эти операции представляют из себя большое количество простых и однотипных арифметических операций (сложения и умножения). По этой причине их можно оптимизировать средствами векторизации и переупорядочивания обхода циклов. Для этих целей была создана математическая модель, основанная на алгебраическом представлении программ и их преобразований, — полиэдральная модель. В них цикл является полиэдром, т. е. многомерным многогранником в аффином пространстве. Аффинные преобразования этого пространства соответствуют преобразованиям цикла, изменениям порядка обхода элементов.

Рассмотрим некоторые проекты, использующие этот подход, а именно *TVM*, *Halide* и *Polly*. Polly является частью проекта LLVM и преобразует LLVM IR для генерации оптимального кода. Его использование привело бы к необходимости добавления новых инструкций в LLVM IR и не давало бы достаточных уровней абстракции. Инфраструктуры TVM и Halide позволяют создавать новые операции, но требуют, чтобы для них был задан способ исполнения. Такой подход является неудобным, т. к. это означает, что после преобразований код должен быть транслирован в ассемблерные инструкции по неким сложным шаблонам. Более того, в Halide это исполнение задаётся исключительно скалярном виде. К сожалению, единственный существующий компилятор для Ascend, преобразующий ONNX модели, — *Automatic kernel generator (AKG)*, являющийся частью MindSpore, пошёл по этому пути и использует Halide в качестве инфраструктуры.

4.3 Итог

Существует большое количество различных компиляторов для нейронных сетей. Они используют разлиные подходы для генерации кода, но во многом повторяют друг

друга. По этой причиной одно из крупнейших сообществ — LLVM — решило создать переиспользуемую инфраструктуру MLIR, собирающую в себя лучшее из различных подходов и техник, в т. ч. и полиэдральной компиляции. На данный момент проект активно развивается и является самой современной крупной инфраструктурой.

5 Обзор аппаратных решений

5.1 Оптимизации с помощью CPU, GPU и NPU

Как было рассказано в главе про компиляторы, большинство современных библиотек генерируют эффективный код для центральных процессоров (CPU) и графических процессоров (GPU). Рассмотрим технологии, которые они при этом используют.

В случае CPU современным стандартом является технология AVX (advanced vector extentions), которые позволяют использовать векторные инструкции с длиной вектора 256 бит (а с недавнего времени и 512 бит). Большинство операций в нейросетях являются векторными или матрицными, по этой причине использование их во время вычислений даёт прирост производительности в несколько раз по сравнению со скалярным кодом. Но существуют и другие подходы. Добавляются модули для матричных вычислений (AMX у компании Intel, SME в архитектуре ARM), начиная с 2023 года в процессорах компаний Intel, AMD и Qualomm начинают появляться NPU, специально предназначенные для ускорения нейронных сетей.

Другим классическим решением являются GPU, которые изначально предназначены для параллельных однородных вычислений. Производители видеокарт создают программные интерфейсы и их аппаратную поддержку для подобных целей. Таковыми являются CUDA у Nvidia и ROCm у AMD. PyTorch и TensorFlow поддерживают компиляцию с использованием этих технологий, время исполнения при этом уменьшается в несколько десятков раз. Также отметим, что недавно эти компании добавили модуль матричных вычислений (AMD matrix cores и NVIDIA Tensor Cores).

Распространение нейронных сетей во всех сферах жизни общества естественно привело к созданию акселераторов, специализированных под пих. Помимо модулей в процессорах, упомянутых выше, существуют и другие решения. Помимо архитектуры DaVinci и процессоров Ascend на её основе, которые будут рассмотрены далее, можно выделить LG NeuroMorphic Processor, Google TPU, а тааже российскую разработку NeuroMatrix от НТЦ «Модуль».

5.2 Обзор архитектуры DaVinci

5.2.1 Общее описание

Архитектура DaVinci — нейропроцессор (NPU, neural processing unit), разработанный компаней HiSilicon (подразделение Huawei). В отличие от обычных CPU и GPU, которые необходимы для вычислений общего назначения, и ASIC, предназначенной для конкретного алгоритма, архитектура Da Vinci предназначена для исполнения уже обученных нейронных сетей. Работа с NPU является обычной схемой гетерогенных вычислений, в ней CPU является хостом (главным устройством, которое запрашивает вычисления), а NPU — девайсом (подчинённым устройством, производящим вычиления). Взаимодействие между ними происходят по следующему алгоритму:

1. Хост производит инициализацию, необходимую для общения с девайсом.
2. Хост аллоцирует память (общую для него и девайса) и загружает данные в неё.
3. Хост загружает объектный файл девайса и регистрирует функцию для исполнения.
4. Хост передаёт указатели на аллоцированную память и даёт команду к исполнению.
5. Девайс исполняет выбранную функцию.

6. Хост копирует результат исполнения девайса к себе.

Процессоры, основанные на архитектуре DaVinci и их основные характеристики представлены в таблице ниже:

Процессор	Производительность для float 16/int 8	Мощность	Тех. процесс
Ascend 910	320/640 терафлопс	310 Вт	7 нм, N7+
Ascend 310	16/8 терафлопс	8 Вт	12 нм, FFC

Теперь перейдём к внутреннему устройствую чипа Ascend. Схема архитектуры представлена на рисунке. Рассмотрим её основные особенности.

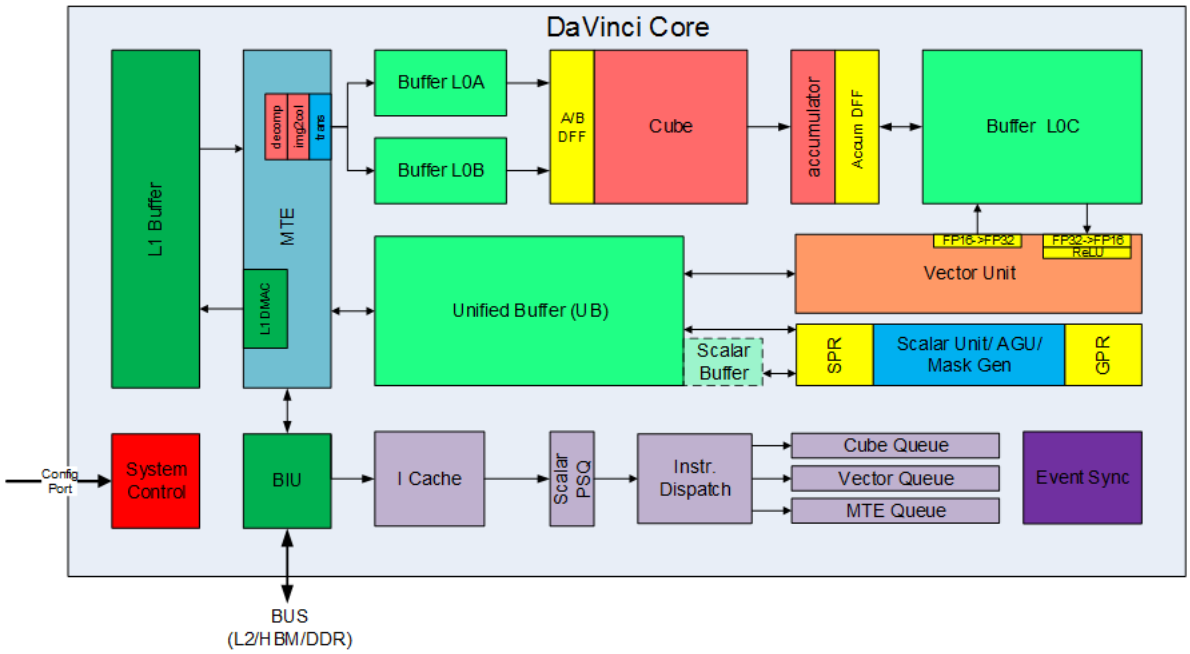

Рис. 3: Архитектура DaVinci

В ядре есть три вычислительных юнита: матричный, векторный и скалярный, которые используются для соответствующих вычилений. Исполнение на юнитах происходит параллельно, для каждого юнита существует отдельная, независимая очередь задач. Ещё три очереди предназначены для копирования из разных буфферов друг в друга (о них речь пойдёт ниже). Для синхронизации очередей используются команды `set_flag` и `wait_flag`, которые по своей сути представляют систему событий. Первая команда сигнализирует, что событие произошло, а вторая запускает ожидание события. Правильное использование механизмов синхронизации позволяет значительно увеличить загрузку всех юнитов и, следовательно, снизить общее время исполнения. В данной работе не будут рассматривать проблемы с расстановкой операций синхронизации и будет считаться, что они всегда расставлены наиболее оптимальным образом.

5.2.2 Матричный юнит

Матричный юнит на вход принимает матрицы с типом элементов `float 16` или `int 8`, на выходе же элементы имеют тип `float 16`, `float 32` или `int 32`. Умножение работает в двух режимах:

1. Обычное умножение: $C = A \times B$

2. Режим накопления: $C = A \times B + C$, т. е. результат текущего умножения прибавляется к предыдущему.

Каждая из входных матриц должна быть разбита на блоки 16×16 (в случае `float 16`) или 16×32 (в случае `int 8`). Расположение элементов внутри блоков и блоков относительно друг друга также различно. Существуют две стратегии размещения: по строкам (формат `Z`) и по столбцам (формат `N`). Примем обозначение: размещение внутри блока обозначается строчной буквой, а между блоками — заглавной. При умножении матрица A должна быть заранее быть записана в формате `Zz`, матрица B — в формате `Zn`, а выходная матрица C будет `Nz`.

Матричный юнит работает по принципу систолического массива. Систолический массив — однородная сеть тесно связанных блоков обработки данных. Его схему для архитектуры DaVinci можно увидеть на картинке ниже.

Принцип умножения довольно прост: за первый такт (FIXME: лучше не использовать слово такт в данном контексте) происходят все умножения, после чего за оставшиеся четыре такта произведения суммируются. Таким образом, за пять тактов можно перемножить две матрицы `16x16`. Матричный юнит, итерируясь по матрицам и перемножая их поблочно, быстро получает результат перемножения.

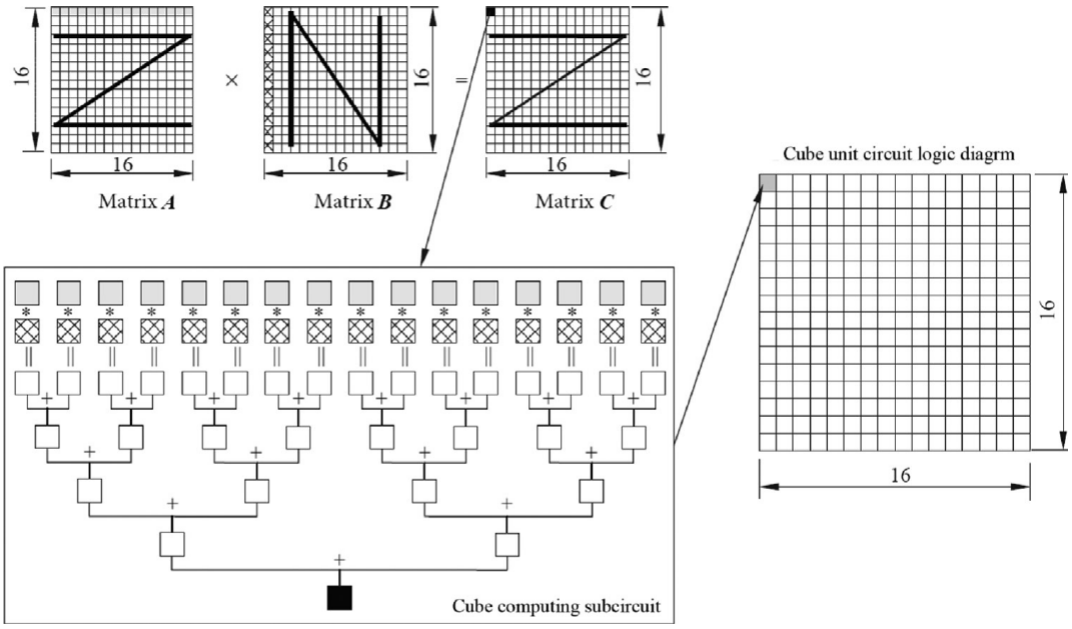

Рис. 4: Схема вычисления в матричном юните

5.2.3 Векторный юнит

Теперь рассмотрим работу векторного юнита. Существуют два режима работы: *common* и *VA*. В обоих случаях память делится на блоки по 32 байта, а векторная операция представляется в виде цикла, на каждой итерации которого обрабатывается 8 блоков. Таким образом, за одну итерацию может быть обработано 128 элементов типа `float 16` или 64 элемента 32-разрядных типов (`float 32`, `int 32`). Стоит отметить, что не обязательно обрабатывать такое количество элементов. Для выбора, с какими именно элементами будет работать операция необходимо задать маску. Векторный юнит обрабатывает только те элементы, на позиции которых в маске стоит бит 1. На каждой следующей итерации блоки сдигаются на заранее заданный шаг, после чего операция

повторяется. Отличаются режимы способом описания этих блоков. В случае *VA* режима блоки задаются в виде восьми указателей, а в случае *common* — указателем на первый блок и шагом между блоками. Для первого, второго аргумента и результата шаг может быть различным. Юнит поддерживает большое количество операций: математических, логических, приведения типов и т. д. Есть и спецефичные операции: *relu*, часто используемый в нейронных сетях, операция транспонирования.

5.2.4 Память

Память ядра неоднородна. В ядре существует 5 буферов: L1, L0A, L0B, L0C, UB. Также существует внешняя память (GM), через которую происходит общение с хостом. Опишем общую схему потока данных между этими кэшами. Данные из внешней памяти загружаются в L1 и UB. Данные в UB предназначены для обработки векторным и скалярным юнитами. Данные из L1 загружаются в L0A и L0B, которые соответствуют матрицам A и B матричного умножения. Результат после перемножения (которое, как было упомянуто раньше, выполняется матричным юнитом), попадает в буфер L0C, из которого происходит данные отправляются в UB. Выгрузка результата вычислений во внешнюю память возможна только из UB. Отметим, что описанные выше буферы имеют небольшой размер, что является одной из основных проблематик нашей работы. Более подробно этот вопрос будет рассмотрен в главе, посвященной ловерингу.

5.2.5 Свёртка

Реализация свёртки на нейроматрицных процессоров несколько сложнее, чем умножения. На некоторых архитектурах (FIXME: ссылка) она поддержана нативно. К сожалению, наша не является таковой. Но с помощью особого преобразования её можно свести к умножению матриц. Приведём некоторые общие соображения, которые позволят понять его.

Итак, пусть есть входное изображение (*image*) размеров $H_i \times W_i$, содержащее C цветов. Будем называть его *входной картой признаков* (*input feature map*). Ядро (*kernel*) свёртки представляет из себя небольшую матрицу размеров $H_k \times W_k$ (характерный размер — $3 - 5$). Ядро имеет такое же количество входных цветов C, но также имеет и F выходных цветов. Таким образом, изображение имеет формат $H_i W_i C$, а ядро — $F H_k W_k C$. Выходная карта признаков, имеет структуру, схожую со входной: $H_o W_o F$, где $H_o = H_i - H_k + 1$, $W_o = W_i - W_k + 1$ в простейшем случае. Если обозначить: a — входная карта, k — ядро, c — выходная, то свёрка выражается следующей формулой:

$$c_{ijf} = \sum_{h=0}^{H_k} \sum_{w=0}^{W_k} \sum_{c=0}^{C} a_{i+h,j+w,c} \cdot k_{fhwc}$$

Заметим, что операция чем-то схожа на скалярное умножение векторов (если цвета считать вектором) или матричное умножение. Если первый тензор преобразовать в матрицу A, где одной строке будет соответвовать одна такая сумма (т.е. размеры матрицы станут $H_o W_o \times H_k W_k C$), а ядро — в матрицу K размеров $H_k W_k C \times F$, то выходная матрица $C = A \times K$. Этот процесс преобразования входной карты признаков называется *img2col* (*image-to-column*), оно содержится в архитектуре команд целевого процессора.

Таким образом, свёрка есть композиция *img2col* и умножения матриц. Отметим, что в реальности свёртка имеет такие параметры, как *stride*, *dilation* и *pad*. Они усложняют приведённые формулы, но не меняют сути происходящего. Также в качестве обобщения

можно взять N изображений, форматы входной и выходной карт приобретают вид NH_iW_iC и NH_oW_oF соответственно.

5.2.6 Изменение формата хранения данных

Рассмотрим принцип работы инструкции *scatter_vnchwconv*. Она будет использоваться в операциях, где необходимо изменить формат хранения данных (например, транспонировать матрицу). За одну итерацию операция работает с двумя матрицами 16×16 типа `float 16`. Каждая матрица задаётся 16-ю указателями на строки, строки не обязаны «лежать подряд». Исходная матрица транспонируется и записывается по соответствующим указателям выходной матрицы. При переходе на следующую итерацию указатели сдвигаются на некоторый шаг (общий для всех 16-ти указателей, но различный для входной и выходной матриц).

5.2.7 Возможные оптимизации

Рассмотрим некоторые оптимизации, которые применимы на архитектуре DaVinci:

1. Разбиение задачи на несколько потоков. В процессорах Ascend несколько ядер на архитектуре DaVinci, поэтому задачу можно разбивать на несколько параллельных (например, в случае умножения матриц — воспользоваться блочным умножением).

2. Двойная буферизация. Как было описано выше, ядро процессора имеет несколько вычислительных юнитов, работающих параллельно. Чем больше юнитов работают одновременно, тем быстрее завершится исполнение задачи. Сделаем одновременным работу модуля памяти и какого-нибудь вычислительного модуля. Для этого разделим буфер на две части. Пока в первом будет происходить загрузка или выгрузка, в другом производятся вычисления.

6 Структура компилятора и его реализация

6.1 Общая схема компиляции и возможные проблемы

В главе FIXME было дано подробное описание особенностей работы процессоров Ascend. Для написания эффективного компилятора необходимо учесть их и отразить в создаваемой инфраструктуре. Кратко обозначим эти проблемы и пути их решения:

1. Гетерогенность системы. Работа с Ascend является примером гетерогенных вычислений. Поэтому компилятор должен генерировать отдельно код выполнения для хоста, и отдельно для девайса. Первый должен поддержить граф вычислений, аллоцировать память, планировать исполнение раличных функций девайса. Код девайса должен быть как можно более оптимальным, так как именно он занимает большую часть времени. Отметим, что далее в данной работе будет рассматриваться только компиляция кода девайса.

2. Типы данных. Очень часто входные данные имеют тип `float 32`, но матричные вычисления на Ascend возможны только с типами `float 16` или `int 8`. По этой причине данные конвертируются в тип для вычислений. После выполнения операции, вероятно, необходима обратная конвертация.

3. Форматы данных. Напомним, матрицы для умножения должны иметь блочный формат хранения (`Zz`, `Zn` или `Nz`). Входные данные, как правило, не соответствуют ему, поэтому перед умножением необходимо выполнять операцию *фрактализации*, т. е. изменения формата хранения. Аналогично предыдущему пункту, после вычислений используется *дефрактализация*.

4. Использование памяти. Размеры внутренних буферов крайне ограничены, по этой причине для исполнения одной операции, зачастую, делается несколько загрузок из внешней памяти, которые могут повторяться в силу свойств операции (например, в случае блочного умножения матриц). Количество загрузок, особенно повторных, должно быть минимальным, т. е. должна быть обеспечена пространственная и временная локальность.

5. Параллелизм и синхронизация. Как было отмечено, у процессоров с архитектурой DaVinci есть шесть юнитов, которые могут работать параллельно. По этой причине процент задействования каждого юнита (т. е. время работы юнита по отношению к общему времени работы) должно быть максимизировано, для этого синхронизация между операциями различных юнитов должна присутствовать только в случае наличия зависимости по данным или именам, а само количество зависимостей — минимизировано.

6. Стратегии и расписание выполнения. Стоит отметить, что в силу ограниченности памяти большинство операций будут представлены в виде цикла: загрузка-обработка-выгрузка. Очевидно, что загрузка может быть для разных по форме и размеру данных, а цикл может иметь различный порядок обхода. Таким образом, необходим выбор оптимальной стратегии (формы данных) и расписания (порядка обхода). Этот пункт включает в себя предыдущие два, но является более общим.

6.2 Инфраструктура LLVM MLIR

Как можно заметить из предыдущего параграфа, компиляторы из предыдущего параграфа решали сходные задачи, но они отличались некоторыми деталями, из-за че-

го приходилось создавать новый компилятор и пересоздавать большое количество компонентов. В связи с этим сообщество разработчиков LLVM придумали и реализовали переиспользуемую и расширяемую инфраструктуру MLIR.

Основная концепция MLIR — диалекты. Диалект объединяет в себе типы, операции и их преобразования на каком-либо уровне абстракции. В MLIR существует более 40 встроенных диалектов, имплентация собственных диалектов возможна с помощью декларативного языка *ODS* или на языке C++.

Рассмотрим некоторые диалекты, которые будут использованы в данной работе.

1. HLO — диалект, который позволяет представлять модели нейросетей, написанных на tensorflow, в представлении MLIR. Несмотря на то, что он не является стандартным и представлен в виде отдельного репозитория, пользуется популярностью благодаря широкой известности tensorflow.

2. tensor — диалект для представления тензоров и операций, позволяющих менять форму тензоров, изменять их размеры, «вырезать» и «вставлять» части из них. Стоит отметить, на данном уровне абстракции считается, что тензоры не имеют какого-то конкретного расположения в памяти. Этим они похожи на виртуальные регистры из теории компиляторов.

3. memref — диалект, который абстрагирует работу с многомерными массивами. Операции в этом диалекте схожи с операциями из диалекта tensor, но этих диалектов есть существенное отличие: memref является представлением реальных объектов.

4. affine — диалект, который предоставляет возможность работы с аффинными циклами и преобразованиями над ними, тем самым реализуя возможности для полиэдральной компиляции.

5. scf (structured control flow) — диалект, в котором представлен структурный поток исполнения (т.е. в виде системы вложенных блоков).

6. cf (control flow) — диалект, предсталяющий исполнение в виде графа потока управления.

7. func — диалект, реализующий концепцию функций, их вызова, передачи аргументов, возвращения значения.

8. transform — диалект, необходимый для реализации преобразований внутри одного диалекта. С его помощью операция предсталяется в виде одной или нескольких операций (зачастую, более эффективных по производительности, чем исходная), что позволяет подготовить код для дальнейшего lowering-а или оптимизировать его.

9. llvm — самый низкоуровневый диалект, реализующий семантику LLVM IR. Его можно перевести в LLVM IR непосредственно, после чего воспользоваться другими средствами LLVM для компиляции. Отметим, что получение кода именно в таком представлении является нашей непосредственной задачей.

Выше были перечислены лишь те диалекты, которые непосредственно будут использованы во время lowering-а из HLO в llvm. Помимо в них в MLIR существует большое количество других диалектов, например, для графических ускорителей (GPU), для векторных инструкций (AVX512), для распараллеливания исполнения программ (OpenMP) и другие. Общая диаграмма диалектов и их соотношения представлена на рисунке ниже.

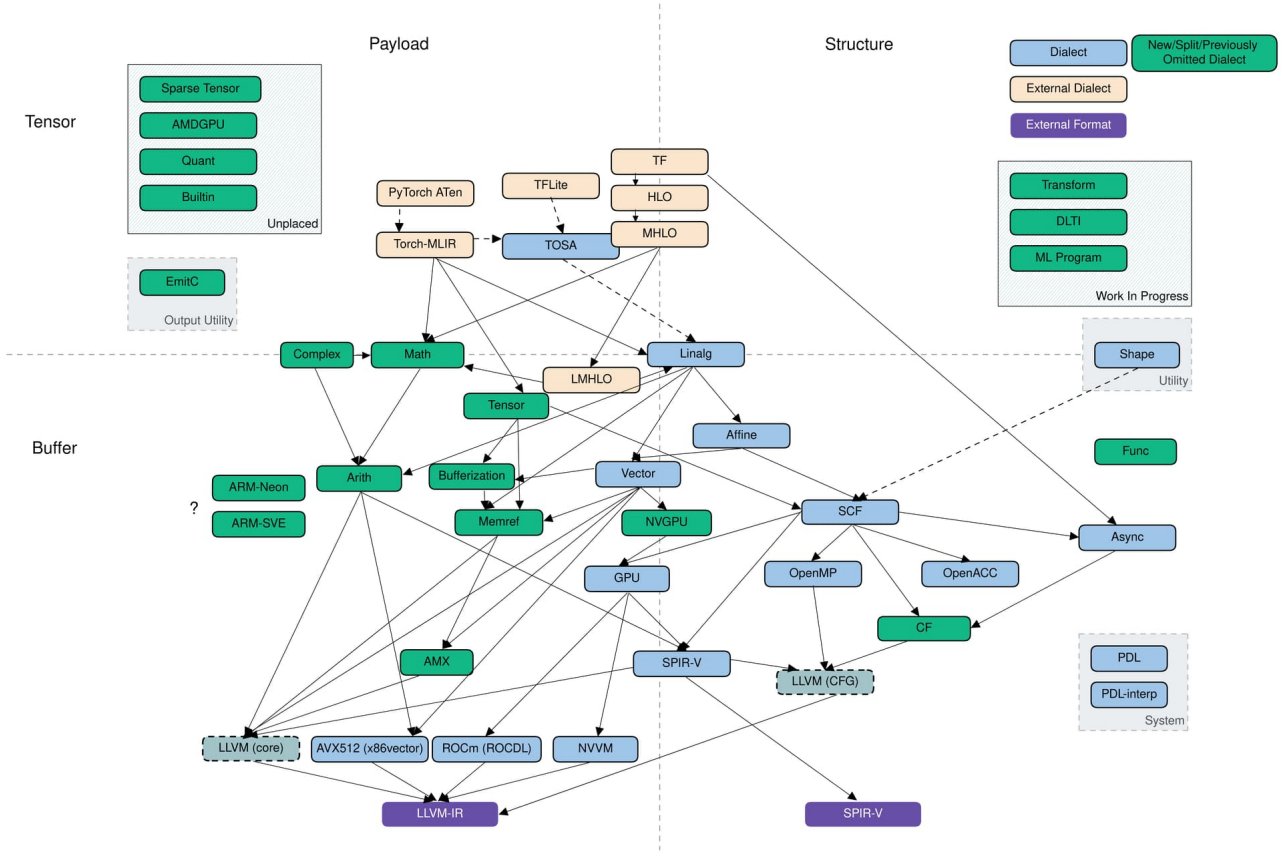

Рис. 5: Структура проекта MLIR и соотношения диалектов в них

6.3 Функциональная структура компилятора

Используя знания о MLIR и DaVinci, полученные в результате исследовательской работы, наша команда приступила к разработке нового компилятора. Было решено создать новые диалекты, которые на разных уровнях абстракции отражают особенности архитектуры DaVinci. Перечислим их и отметим основные особенности:

1. ascend — диалект крупноблочных операций. Является аналогом HLO, но операции в нём предъявляют требования к типам данных: матрицы должны быть расположены в блочном формате. Поэтому в процессе lowering-а из HLO в ascend для входных и выходных данных вставляются операции фрактализации, т.е. приведения матрицы к нужному виду. Для остальных диалектов требование на формат данных сохраняется, при этом считается, что оно выполняется благодаря корректности представления графа исполнения в диалекте ascend.

2. cce — диалект операций, схожих с ассемблерными инструкциями. Основная его особеннность заключается в сохранении семантики тензоров, что позволяет упрощать процесс генерации таких операций и их верификации (проверки корректности).

3. hivm — диалект непосредственных ассемблерных инструкций. Он в точности повторяет их семантику, что упрощает его ловеринг в llvm.

Конвейер (пайплайн) компиляции для девайса выглядит следующим образом:

В данной работе рассматривается ловеринг от диалекта ascend к диалекту cce, что соответствует переходу от крупноблочных операторов к операторам, схожих с инструкциями процессора Ascend.

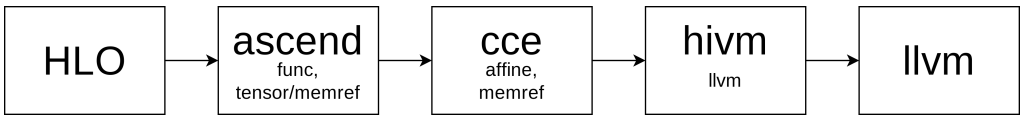

Рис. 6: Пайплайн компиляции для девайса

6.4 Ловеринг операций и возможные стратегии

Изучив особенности целевой архитектуры, перейдём к рассмотрению конкретных стратегий ловеринга и их классификации. В связи с тем, что именно работа с внешней память занимает большую часть времени, будем пытаться оптимизировать её. Как было упомянуто в одной из предыдущих глав, из-за малого объёма внутренних кэшей данные приходится загружать частями, при этом каждая часть, возможно, будет загружена несколько раз. В связи с этим, уменьшение количества повторных загрузок — самый простой способ оптимизации, а стратегия разбиения, при которой достигается наименьшее количество повторных загрузок, будет считаться нами наиболее оптимальной.

6.4.1 Умножение матриц

Умножение матриц в диалекте `ascend` представлено оператором `matmul`:

```
func.func @kernel_func(
    %matrixA: memref<48x32x16x16xf16, 1>,
    %matrixB: memref<48x48x16x16xf16, 1>,
    %matrixC: memref<48x32x16x16xf16, 1>) {
    ascend.matmul(%matrixA, %matrixB, %matrixC) : memref<48x32x16x16xf16, 1>,
        memref<48x48x16x16xf16, 1>, memref<48x32x16x16xf16, 1>
    return
}
```

На вход оператор примает три многомерных массива, соответствующих матрицам A, B, C матричного умножения $C = A \times B$. Все три матрицы хранятся в формате `Nz`, это сделано для удобства, т. к. в реальных нейронных сетях несколько умножений могут идти подряд, при этом результат предыдущего передаётся на вход следующего. Привести кусок матрицы к необходимому формату (`Zz` или `Zn`) не является проблемой при правильном использовании операций копирования из GM в L1 и загрузки из L1 в L0. Число 1 в конце каждого `memref` означает, что массив находится в первом адресвом пространстве, что соответствует памяти GM.

Теперь перейдём к тому, каким образом можно реализовать ловеринг и оптимизировать количество копирований. Согласно статье FIXME, можно выделить три основные стратегии: *input stationary (IS)*, *weight stationary (WS)* и *output stationary (OS)*. Отметим, что в данной статье рассматривается операция свёртки, но всё перечисленное в ней верно и для умножения матриц. Все три стратегии имеют общую идею: умножение производится блочно, блок одной из матриц «фиксируется» (A для IS, B для WS и C для OS), после чего перебираются всевозможные блоки других матриц. Наглядное объяснение этого процесса можно увидеть на картинке. После перебора выбирается другой блок и операция повторяется.

Перечисленные стратегии описывают порядок обхода матриц по их блокам, но ничего не говорят о размерах самих блоков. Пара стратегия + разбиение задаёт конкретное исполнение, задача же состоит в том, чтобы выбрать конкретное исполнение, дающее максимальную производительность. Для теоритического предсказания оптимального исполнения нами была предложена метрика, соответствующая количеству

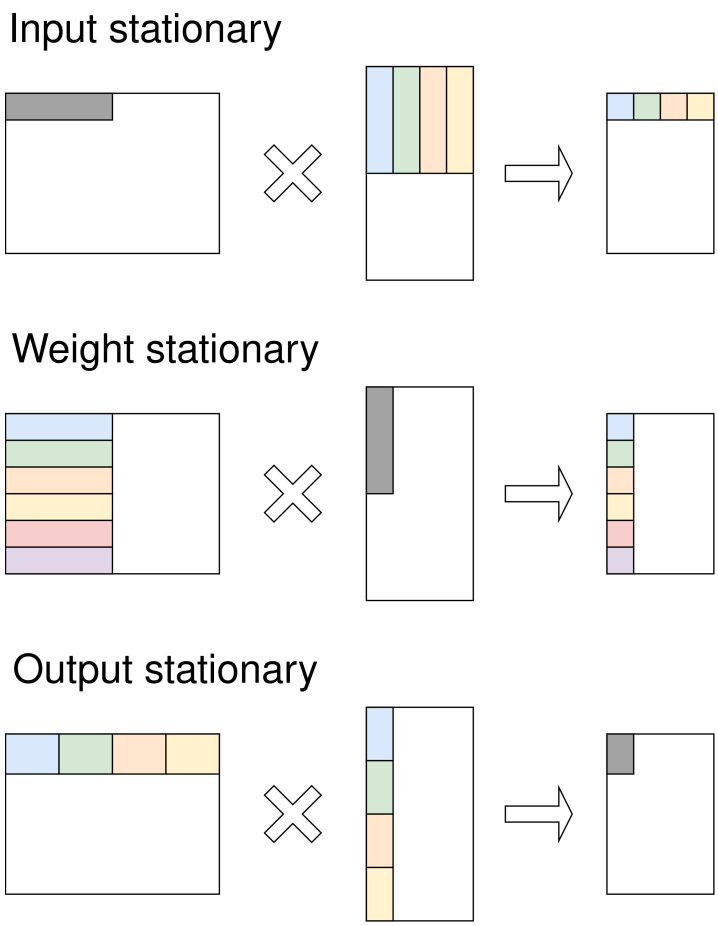

Рис. 7: Стратегии для умножения матриц

загруженной памяти. Опишем её более формально. Пусть матрица A имеет размеры $M \times N$, а матрица B — $N \times K$, тогда C — матрица $M \times K$. Таким образом, матрицы описываются тройкой (M, N, K). Аналогично, разбиение на блоки описывается тройкой (m, n, k). Будем рассматривать только такие конфигурации, при которых блочное умножение можно выполнить за одну инструкцию, а выполнение перебора блоков матриц при фиксации одного из них не требует выгрузки и загрузки промежуточных результатов. В силу этих ограничений можно считать, что для IS $k = K$ или $n = N$, а для WS — $m = M$ или $n = N$.

Рассмотрим IS стратегию. Каждой загрузке матрицы $m \times n$ соответстует $\frac{K}{k}$ загрузок матриц $n \times k$. Повторяется это действие $\frac{MN}{mn}$ раз. Таким образом, общее количество загруженной памяти составляет:

$$\Sigma_{IS} = \left(mn + nk\,\frac{K}{k}\right)\frac{MN}{mn} = MN + \frac{MNK}{m}$$

Аналогичным образом можно получить метрики для дугих стратегий:

$$\Sigma_{WS} = KN + \frac{MNK}{k}$$

$$\Sigma_{OS} = MNK\,\frac{m+k}{mk}$$

На тройку (m, n, k) должны быть наложены ограничения, связанные с размерами буферов. Если считать, что элементы входных матриц имеют тип `float 16`, а выходной — `float 32` (это соответствует вычислениям на Ascend с повышенной точностью) тогда:

$$\begin{cases} M, N, K, m, n, k \vdots 16 \\ mn \leqslant 2^{15} \\ nk \leqslant 2^{15} \\ mk \leqslant 2^{16} \end{cases}$$

Итак, загружаемая память должна быть минимизирована, т. е. $\Sigma \to \min$ Из эмпирических наблюдений (речь о которых пойдёт далее), при прочих равных используемая память должна быть максимизирована:

$$\begin{cases} mn \to \max \\ nk \to \max \\ mk \to \max \end{cases}$$

Для проверки гипотезы был разработан тестовый бенчмарк, замеряющий время исполнения при различных конфигурацях. Гипотеза, изложенная выше, подтвердилась. Если рассматривать типичные размеры умножаемых матриц в нейросетях (например, BERT) — $(512, 768, 768)$, то оказывается, что OS — наиболее оптимальная стратегия, а $(256, 128, 256)$ — оптимальная конфигурация.

FIXME графики

OS стратегия была реализована в компиляторе на инфраструктуре MLIR в виде понижения оператора `ascend.matmul` до цикла на диалектах `cce`, `affine` и `memref`. Ниже приведён псевдокод алгоритма:

```
func ascend.matmul(matrixA, matrixB, matrixC) {
    for (x = [0, M / m)) {
        for (y = [0, K / k)) {
            c = empty_buf(m, k)
            for (sum_part = [0, N / n)) {
                a = empty_buf(m, n)
                b = empty_buf(n, k)
                load_part(a, matrixA, sum_part, y)
                load_part(b, matrixB, x, sum_part)
                cce.mad(a, b, c)
            }
            store_part(matrixC, c, x, y)
        }
    }
}
```

Подчеркнём, что в реальности алгоритм имеет несколько особенностей, не отражённых в псевдокоде. Во-первых, учитывается фрактализация матриц (их формат хранения). Несмотря на то, что в L0A и L0B матрицы должны лежать в формате Zz и Zn соответственно, в памяти GM они хранятся в формате Nz (это сделано для выполнения нескольких умножений подряд без выполнения промежуточных фрактализаций). Изменение формата происходит в ходе загрузки матриц во внутренни буферы. Во-вторых, операции, названные в псевдокоде `load_part` для A и B и `store_part` для C, состоят из двух этапов, для каждой матрицы по-разному: $A : GM \to L1 \to L0A$, $B : GM \to L1 \to L0B$, $C : L0C \to UB \to GM$. В-третьих, поддержана операция `ascend.batch_matmul`, которая производит умножение нескольких пар матриц.

6.4.2 Фрактализация и дефрактализация

Как было написано ранее, перемножение матриц требует особый формат хранения, при этом было решено передавать на вход оператора умножения в формате Nz. По этой причине необходимо матрицы, поступающие на вход нейронной сети, заранее фрактализовать (т.е. поменять формат хранения). Обратное действие (дефрактализация) необходимо для матрицы на выходе. Отметим, что далее речь будет идти только о фрактализации, т.к. все преобразования линейны и дефрактализация будет представлять собой действия фрактализации, выполненные в обратном порядке. Рассмотрим фрактализацию на примере матрицы (512×768). В формате Nz она будет иметь вид $48 \times 32 \times 16 \times 16$. В скалярном коде фрактализация имеет следующий вид:

```
half src[512][768];
half dst[48][32][16][16];

for (i = [0, 48))
  for (j = [0, 32))
    for (k = [0, 16))
      for (l = [0, 16))
        dst[i][j][k][l] = src[j * 16 + k][i * 16 + l];
```

Исполнение скалярного кода является медленным, по этой причине будем использовать инструкцию scatter_vnchwconv. Было реализовано два алгоритма.

Наивный алгоритм работает циклом с частями 16×768, первый scatter собирает матрицы в блоки 16×16, а второй транспонирует блоки. Для преобразования всей матрицы происходит перебор этих кусочков, а копирование в GM происходит по блокам с шагом 32 блока.

```
half src[16][768];
half tmp[48][16][16]
half dst[48][16][16];

scatter_vnchwconv(
    src = src, dst = tmp,
    src_ptrs = {{0, 0}, {1, 0}, {2, 0}, ...},
    dst_ptrs = {{0, 0, 0}, {0, 1, 0}, {0, 2, 0}, ...},
    src_stride = {0, 16},
    dst_stride = {1, 0, 0},
    repeat = 48
);

scatter_vnchwconv(
    src = tmp, dst = dst,
    src_ptrs = {{0, 0, 0}, {0, 1, 0}, {0, 2, 0}, ...},
    dst_ptrs = {{0, 0, 0}, {0, 1, 0}, {0, 2, 0}, ...},
    src_stride = {1, 0, 0},
    dst_stride = {1, 0, 0},
    repeat = 48
);
```

Улучшенный алгоритм позволяет задействовать всю память UB:

```
half src[80][768];
half tmp[5][48][16][16]
half dst[48][16][5][16];
```

```
half src_reshape[16][5][48][16] = reshape(src)

scatter_vnchwconv(
    src = src_reshape, dst = tmp,
    src_ptrs = {{0, 0, 0, 0}, {1, 0, 0, 0}, {2, 0, 0, 0}, ...},
    dst_ptrs = {{0, 0, 0, 0}, {0, 0, 1, 0}, {0, 0, 2, 0}, ...},
    src_stride = {0, 0, 1, 0},
    dst_stride = {0, 1, 0, 0},
    repeat = 240
);

for (i = [0, 5))
    scatter_vnchwconv(
        src = tmp, dst = src,
        src_ptrs = {{i, 0, 0, 0}, {i, 0, 1, 0}, {i, 0, 2, 0}, ...},
        dst_ptrs = {{0, 0, i, 0}, {0, 1, i, 0}, {0, 2, i, 0}, ...},
        src_stride = {0, 1, 0, 0},
        dst_stride = {1, 0, 0, 0},
        repeat = 48
    );

half dst_logical_shape[48][5][16][16] = reshape(dst)
```

С его помощью обрабатывается часть 80×768. После разбиения матрицы на части остаётся «хвост» 32×768, который обрабатывается аналогично. Загрузка результата в GM происходит по 5 блоков. Отметим, что операция reshape является исключительно логической и не изменяет порядок хранения элементов.

6.4.3 Преобразование типов

FIXME

6.5 Бинарные операции

FIXME

6.6 Транспонирование

FIXME

7 Результаты

WIP
Output Stationary — самая эффективная стратегия? Графики (или в аппендикс?)?

8 Заключение и дальнейшая работа

WIP
Работа не закончена...

Список литературы

[1] *Mott-Smith, H.* The theory of collectors in gaseous discharges / H. Mott-Smith, I. Langmuir // *Phys. Rev.* — 1926. — Vol. 28.

[2] *Морз, Р.* Бесстолкновительный PIC-метод / Р. Морз // Вычислительные методы в физике плазмы / Ed. by Б. Олдера, С. Фернбаха, М. Ротенберга. — М.: Мир, 1974.

[3] *Киселёв, А. А.* Численное моделирование захвата ионов бесстолкновительной плазмы электрическим полем поглощающей сферы / А. А. Киселёв, Долгоносов М. С., Красовский В. Л. // Девятая ежегодная конференция «Физика плазмы в Солнечной системе». — 2014.