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

Физтех-школа радиотехники и компьютерных технологий
Кафедра системного программирования ИСП РАН
Отдел «Компиляторные технологии»

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

Современные нейронные сети увеличиваются и требуют большее количество вычислительных ресурсов. По этой причине разрабатываются специализированное аппаратное обеспечение, например нейронные процессоры, и оптимизирующие компиляторы для него. В рамках дипловной работы реализован компилятор нейронных сетей для процессора с архитектурой DaVinci. Компилятор разработан на базе инфраструктуры LLVM MLIR. В работе рассмотрены основные особенности архитектуры и исследованы способы их эффективого использования. Реализованный компилятор содержит три промежуточных уровня представления и выполняет трансляцию основных операций нейронных сетей в целевой набор инструкций архитектуры DaVinci с учётом её особенностей, таких как: неоднородная организация памяти, наличие матричных и векторных инструкций. Корректность работы компилятора была экспериментально исследована на синтетическом наборе тестов, включающих в себя отдельные крупноблочные операции, а также на реальной нейронной сети BERT, содержащей 478 операторов.

Abstract

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

Вязовцев Андрей Викторович

Neural networks are a good and effective way to process any data. For this reason, they are used in various areas of life and are becoming increasingly popular. To increase the performance of neural networks, various methods are used, such as compilers and special computing devices.

This paper examines various methods for optimizing neural networks and describes the implementation of the compiler. The goal of the work is to develop a neural network compiler for Ascend processors based on the DaVinci architecture using the LLVM MLIR infrastructure and ensure the generation of efficient machine code in it. In particular, individual large-block operations of neural networks and the process of their translation into instructions of the target architecture are considered.

Содержание

1 Введение

Нейронные сети в последниее время активно развиваются и находят применение в большом количестве различных областей, в том числе очень популярными стали средства для обработки естественной речи. Количество операций нейронных сетей и их параметров непрерывно увеличивается, поэтому для их выполнения требуются существенные вычислительные мощности. Для выполнения современных нейронных сетей разработчики аппаратного обеспечения развивают специализированные процессорные архитектуры, которые могут быть использованы в серверных или мобильных решениях. Наиболее известными из них являются процессор Ascend с архитектурой DaVinci компании Huawei, NeuroMorphic Processor компании LG, Google TPU, а также российская разработка — векторно-матричная архитектура NeuroMatrix.

Задача эффективного использования возможностей архитектуры нейропроцессоров полностью делегируется компилятору и среде времени выполнения. Значительная часть существующих компиляторов с открытым исходным кодом, как правило, ориентированы на генерацию эффективного кода для моделей вычислений, используемых графическими процессорами или процессорами общего назначения с векторными расширениями. Одними из наиболее популярных проектов, позволяющих создавать компиляторы нейронных сетей являются проекты TVM и Halide. Данные проекты предоставляют примитивы для описания стратегии выполнения операций линейной алгебры. Однако использование инфраструктуры этих проектов для генерации эффективого кода специализированных нейропроцессоров может оказаться затруднительным. На сегодняшний день перспективным проектом для создания компиляторов является LLVM MLIR [1, 2, 3, 4].

Инфраструктура MLIR предоставляет широкий набор примитивов для проектирования собственных промежуточных представлений, трансформирующих преобразований над ними, а также трансляции между ними. Описание инструкций промежуточного представления выполняется на декларативном языке, а сами инструкции могут обладать произвольной семантикой: скалярной, векторной, матричной и другими.

В данной работе исследованы особенности архитектуры DaVinci и представлены способы генерации эффективого машинного кода с использовани-

ем инфраструктуры MLIR путём трансляции крупноблочных операторов в операторы, соответствующие целевой архитектуре.

2 Постановка цели и задач

Цель исследования: разработать компилятор нейронных сетей для процессоров Ascend, основанных на архитектуре DaVinci [5], с использованием инфраструктуры LLVM MLIR [1] 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 устройтва 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 опера-

ций в ней занимают умножения матриц и поэлементные бинарные операции, такие как сложение и умножение. Эти операции были выбраны для дальнейшего рассмотрения.

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.

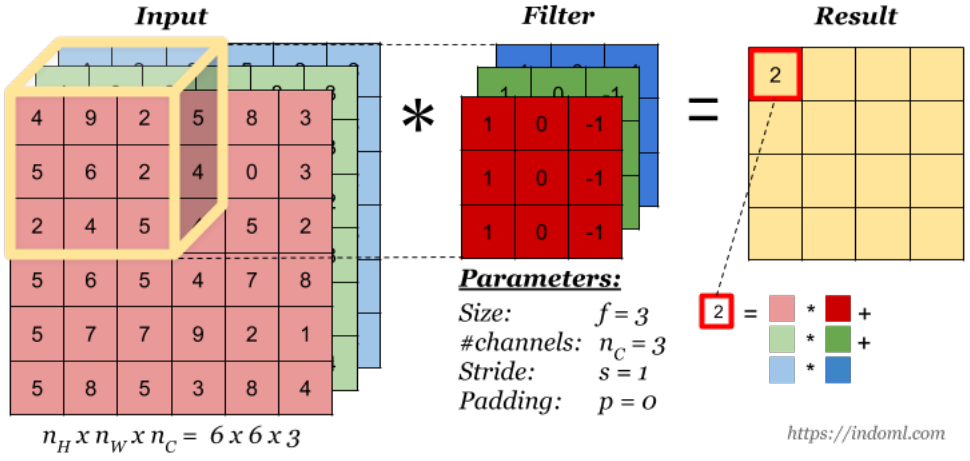


Рис. 1: Операция свёркти для изображения с тремя цветами

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*, вычисляющейся по формуле (1).

$$Relu(x) = \begin{cases} x, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \tag{1}$$

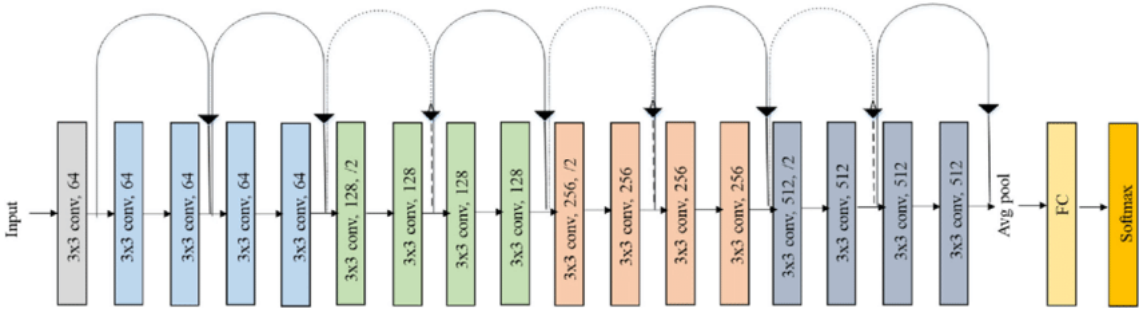


Рис. 2: Модель ResNet18

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* [6] и *TensorFlow* [7]. 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* [8], *TOSA* [9] или *StableHLO* [10]. 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* [11], *MindSpore* [12], *ONNX runtime* [13]. Но все они также нацелены на исполнение на самых распространённых архитектурах (см. главу 5). 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 соот-

ветствуют преобразованиям цикла, изменениям порядка обхода элементов.

Рассмотрим некоторые проекты, использующие этот подход, а именно *TVM* [14], *Halide* [15] и *Polly* [16]. 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)* [17], 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*, *central processing unit*) и графических процессоров (*GPU*, *graphics processing unit*). Рассмотрим технологии, которые они при этом используют.

В случае CPU современным стандартом является технология *AVX* [18] (*advanced vector extentions*), которые позволяют использовать векторные инструкции с длиной вектора 128, 256 или 512 бит. Большинство операций в нейросетях являются векторными или матричными, 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* [18] у компании Intel, *SME* [19] в архитектуре ARM), начиная с 2023 года в процессорах компаний Intel, AMD и Qualomm начинают появляться нейропроцессоры (*NPU*, *neural processing unit*), специализированные 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* [20] у Nvidia и *ROCm* [21] у AMD. PyTorch и TensorFlow поддерживают компиляцию с использованием этих технологий, время исполнения при этом уменьшается в несколько десятков раз. Также отметим, что недавно в графические ускорители 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 них. Помимо модулей в процессорах, упомянутых выше, существуют и другие решения. Таковыми являются архитектура *DaVinci* и процессоры *Ascend* на её основе, которые будут рассмотрены далее, *LG NeuroMorphic Processor*, *Google TPU*, а также российская разработка *NeuroMatrix* от НТЦ «Модуль».

5.2 Обзор архитектуры DaVinci

5.2.1 Общее описание

Архитектура DaVinci [5] — нейропроцессор,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 DaVinci 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.

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

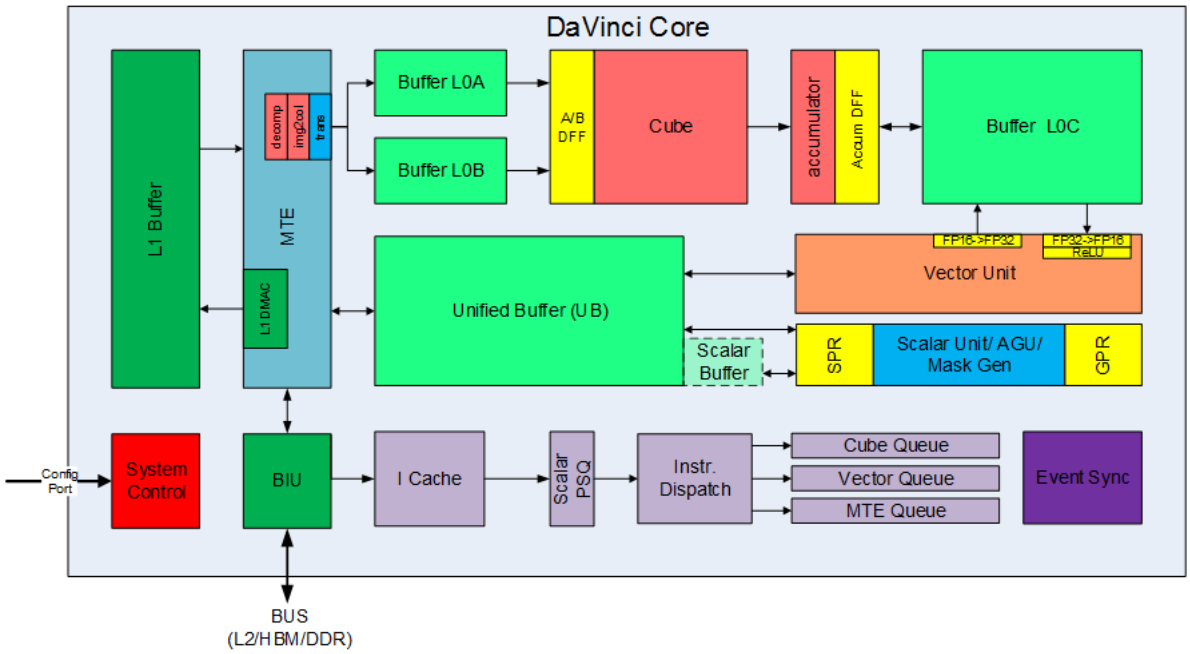


Рис. 3: Архитектура DaVinci

копирования из разных буфферов друг в друга (о них речь пойдёт ниже). Для синхронизации очередей используются команды `set_flag`, `wait_flag` и `pipe_barrier`, которые по своей сути представляют систему событий. Первая команда сигнализирует, что событие произошло, вторая запускает ожидание события, а третья создаёт барьер, который означает приостановку всех очередей до завершения какой-либо очереди. 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: про 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 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.

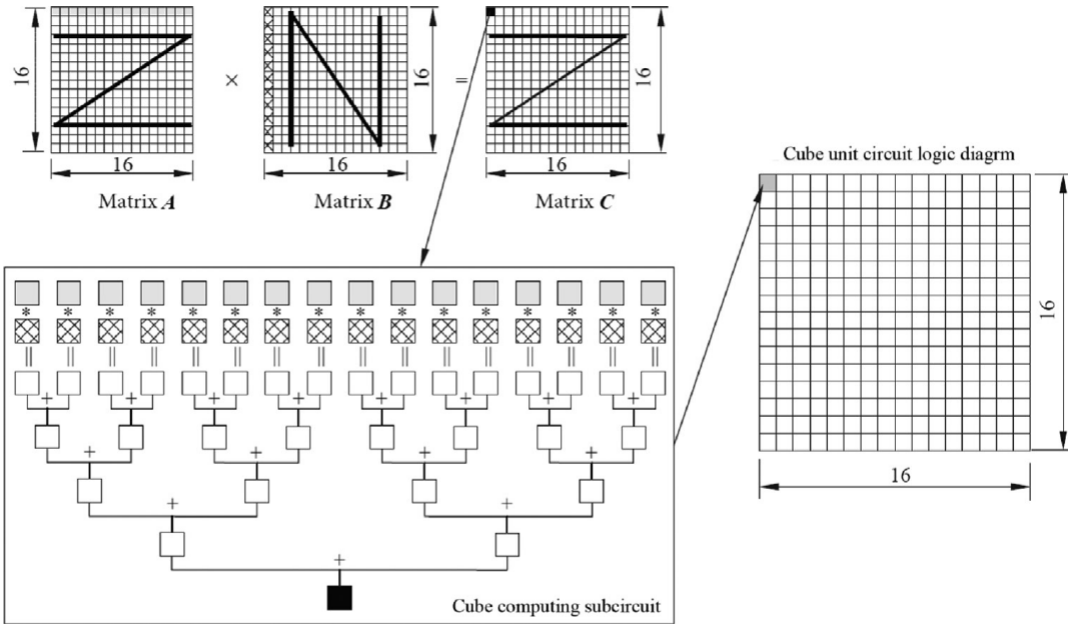


Рис. 4: Схема вычисления в матричном юните

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 или 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 будет значительно влиять на построение стратегий эффективного исполнения операций. Также буферы являются *гранулярными*, в L1 и UB загрузка и выгрузка происходит блоками по 32 байта, а в L0A, L0B и L0C — по 512.

5.2.5 Свёртка

Реализация свёртки на нейроматричных процессоров несколько сложнее, чем умножения. На некоторых архитектурах [22] 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_iW_iC, а ядро — FH_kW_kC. Выходная карта признаков, имеет структуру, схожую со входной: H_oW_oF, 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 $H_oW_o \times H_kW_kC$), а ядро — в матрицу K размеров $H_kW_kC \times F$, 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 5 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 вычилений. Поэтому компилятор должен генерировать отдельно код выполнения для хоста, и отдельно для девайса. Первый должен поддержить граф вычислений, аллоцировать память, планировать исполнение различных функций девайса. Далее в данной работе будет рассматриваться только компиляция и оптимизация кода девайса.

2. *Типы данных.* Зачастую 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 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 па-

раллельно. 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 было сказано в главе 4, компиляторы решали сходные задачи, но они отличались некоторыми деталями, из-за чего приходилось создавать новый компилятор и пересоздавать большое количество компонентов. В связи с этим сообщество разработчиков LLVM придумали и реализовали переиспользуемую и расширяемую инфраструктуру MLIR [1, 2, 3, 4].

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.

10. *bultin* — базовый диалект, предоставляющий базовые типы данных и некоторые служебные объекты (например, аттрибуты).

11. *arith* — диалект операций над элементарными типами.

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.

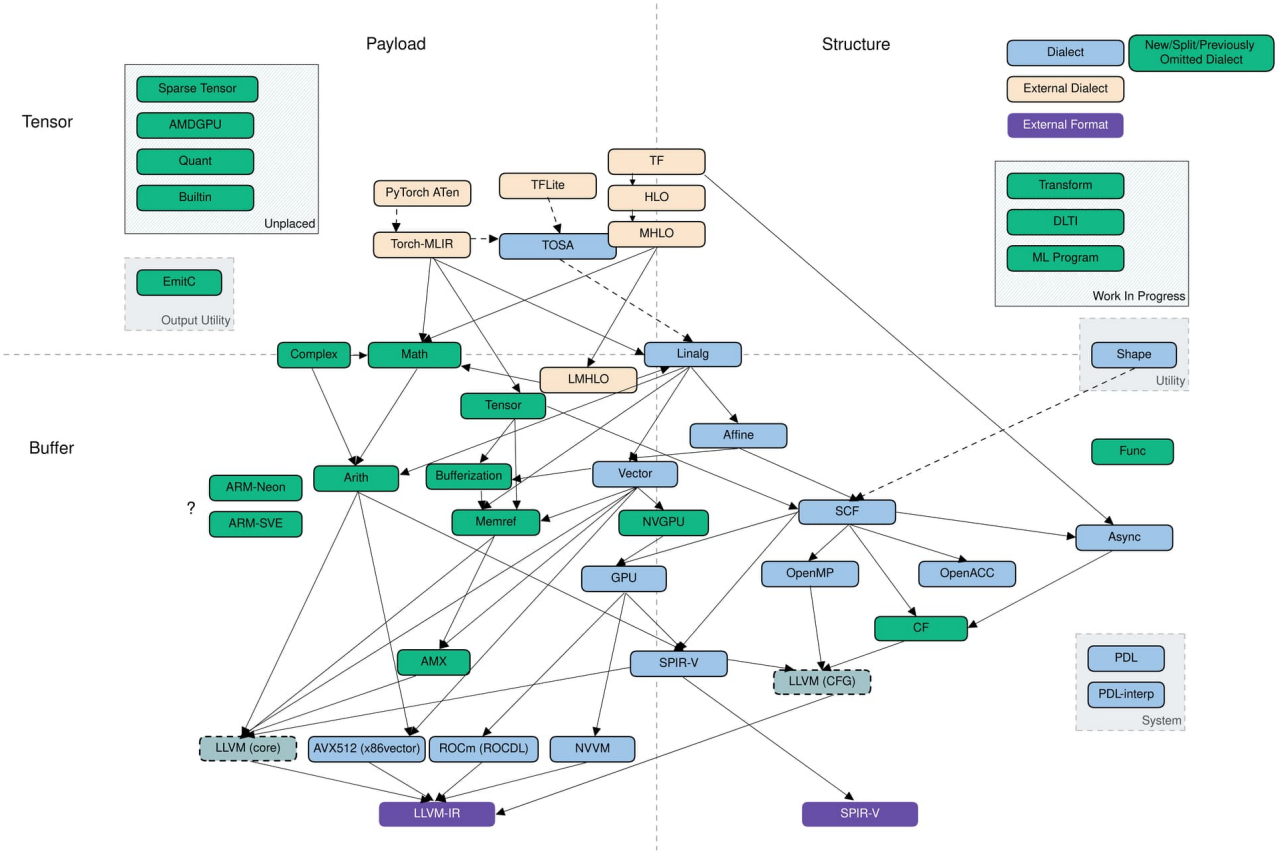


Рис. 5: Структура проекта MLIR и соотношения диалектов в них

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 ловеринга 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 многомерных массивов, 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:

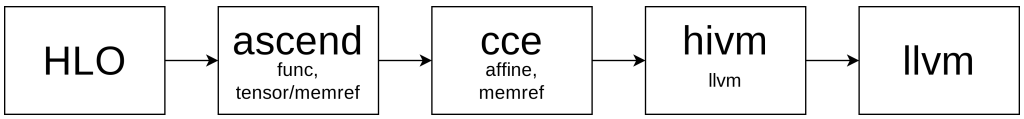


Рис. 6: Пайплайн компиляции для девайса

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

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

Теоретическая модель

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 [22], 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 исполнение, дающее максимальную производительность. Для теоритического предсказания оптимального исполнения нами была предложена метрика, соответствующая количеству загруженной памяти. Опишем её более формально. Пусть матрица A имеет размеры $M \times N$, а матрица B —

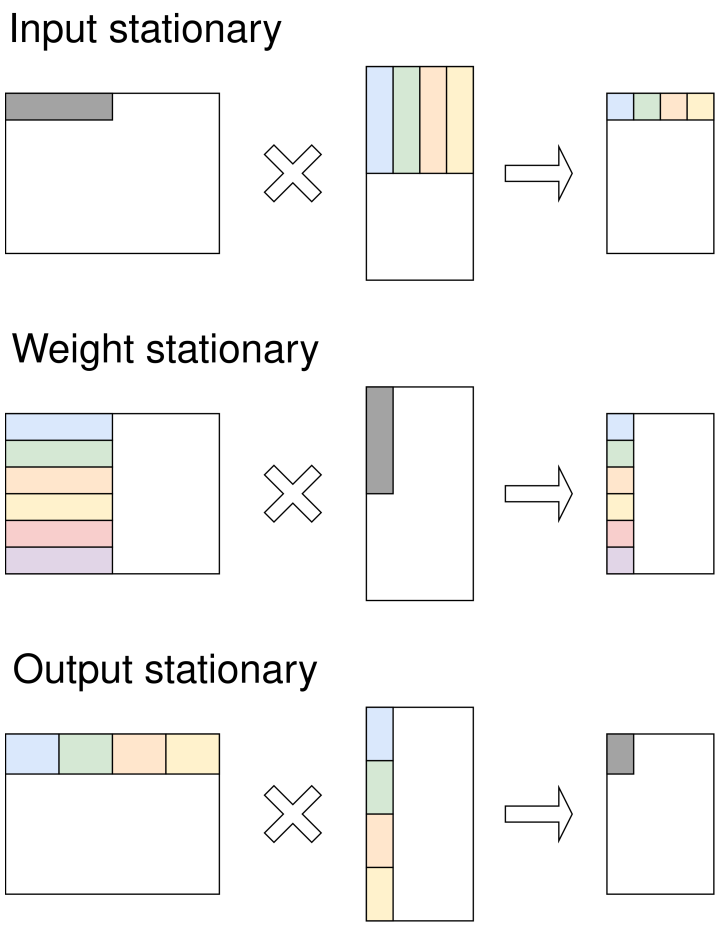


Рис. 7: Стратегии для умножения матриц

$N \times K$, 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 $m \times n$ соответствует $\frac{K}{k}$ 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 других 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:

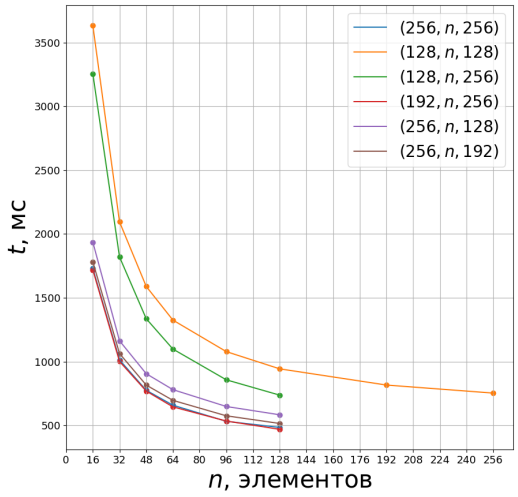
$$\begin{cases} mn \to \max \\ nk \to \max \\ mk \to \max \end{cases}$$

Экспериментальная модель

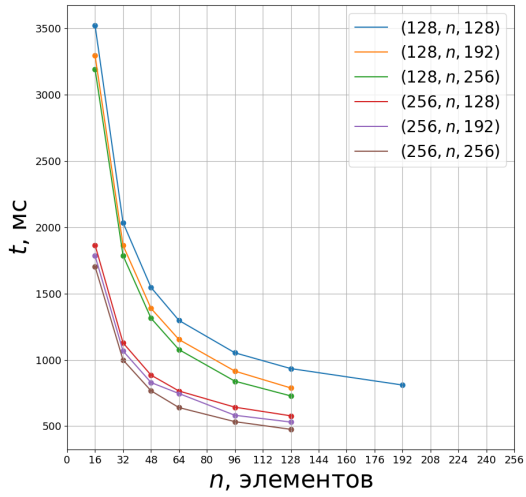
Для проверки гипотезы был разработан тестовый бенчмарк, замеряющий время исполнения при различных конфигурацях и оптимизациях (см. главу 5.2.7). Напомним, что исполнение задаётся стратегией и двумя тройками (M, N, K) и (m, n, k), обозначающими размеры целой матрицы и блоков разбиения. Рассмотрим пример, типичный для нейросети BERT: $(M, N, K) = (512, 768, 768)$. Путём перебора был найден минимум метрики $\Sigma = 2359296$ в случае OS стратегии при $m = k$. Рассмотрим различные m и k, проварьируем n и рассмотрим возможные (подходящие под ограничения) OS-исполнения. Соответствующие графики представлены ниже.

Также сравним наиболее оптимальные исполнения IS — $(32, 768, 32)$, WS — $(32, 768, 32)$ и OS — $(256, 128, 256)$. Диаграмма для случая без оптимизаций представлена ниже.

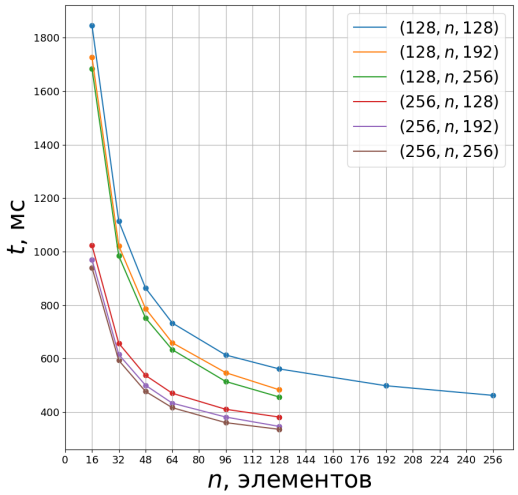
Гипотеза, изложенная выше, частично подтвердилась. Если рассматривать типичные размеры умножаемых матриц в нейросетях (например,



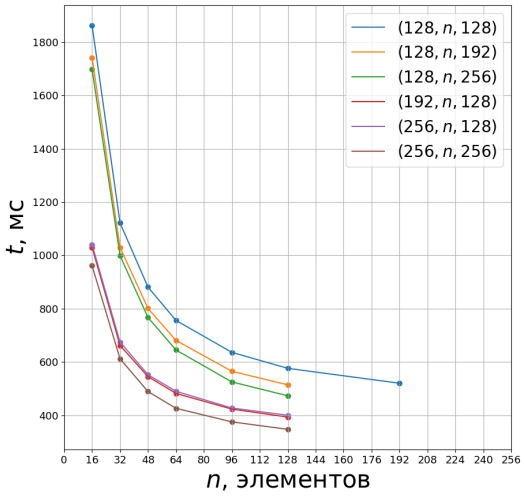
(a) Без оптимизаций



(b) С двойной буферизацией



(a) С распараллеливанием



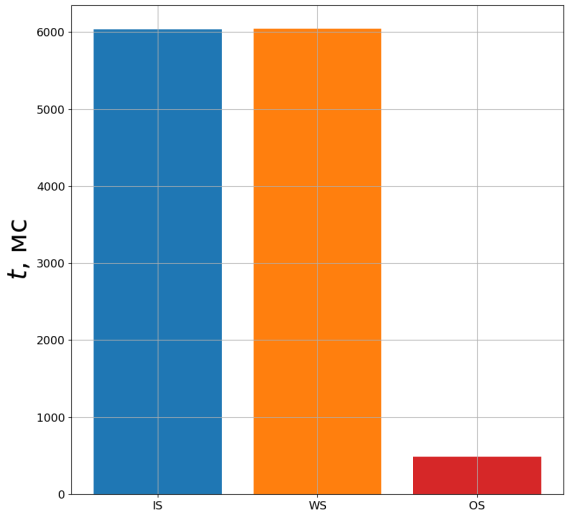
(b) С двойной буферизацией и распараллеливанием

BERT) — $(512, 768, 768)$, то оказывается, что OS — наиболее оптимальная стратегия, а $(256, 128, 256)$ — оптимальная конфигурация. Конфигурация $(128, 256, 128)$ не является оптимальной, несмотря на текущую теоритическую модель, т. к. в ней не учитываются выгрузки памяти.

Реализация

OS стратегия была реализована в компиляторе на инфраструктуре MLIR в виде понижения оператора `ascend.matmul` до цикла на диалектах scf, 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 Поэлементные операции

На момент написания текста были реализованы некоторые бинарные операции (сложение, вычитание, умножение, деление, максимум, минимум), преобразование типов и операция «зажима»: $clamp(x, y, z) = \min(z, \max(x, y))$. Все они работают по общему приниципу: если массив данных многомерный, он преобразуется к одномерному. Далее массив разбивается на максималь-

но возможные порции, при которых в UB помещаются входные и выходные данные. Данные загружаются из GM порциями, обрабатываются соответствующими векторными операциями и ответ выгружается. Отметим, что в силу того, что количество элементов в порции может быть некратным количеству элементов, обрабатываемых за итерацию инструкции (128 в случае `float 16` и 64 в случае `float 32` или `int 32`), конец порции необходимо обработать отдельно, предварительно изменив маску.

Также может возникнуть проблема, связанная с обработкой последней порции и гранулярностью UB. Напомним, что в UB возможна заргузка данных, размер которых кратен 32 байтам. Это означает, что если размер всего массива данных не кратен 32, то с загрузкой последней порции могут возникнуть случае. Решением этой проблемы является загрузка «с наложением»: загрузка некоторую часть предыдущей порции с целью выровнять размер. При этом это «наложение» обрабатывается и выгружается повторно. Отметим, что такая схема не влияет на общую производительность, т. к. размер «наложения» пренебрежимо мал по сравнению с размерами всего массива.

Пример псевдокода для операции cast представлен ниже:

```
half src[N];
float dst[N];

size_t n = pick_max_available(N);
half src_local[n];
float dst_local[n];

for (i = [0, N / n))
    copy(src + i * n, src_local, n);
    cast_f16_f32(
        src = src_local, dst = dst_local,
        repeat = n / 64, mask = full_mask,
        src_stride = 4 blocks, // 64 elems
        dst_stride = 8 blocks // 64 elems
    );
    copy(dst_local, dst + i * n, n);

// tail handling with overlapping
size_t block_size = 16;
size_t tail = (N % n + block_size - 1) / block_size * block_size;

copy(src + N - tail, src_local, tail);
cast_f16_f32(
    src = src_local, dst = dst_local,
    repeat = 1, mask = partial_mask,
    src_stride = 0 blocks, // only one repeat
```

```
    dst_stride = 0 blocks // only one repeat
);
copy(dst_local, dst + N - tail, tail);
```

6.5 Транспонирование

Операция транспонирования в моделях нейронных сетей работает с многомерными тензорами, по этой причине данную операцию корректнее называть «перестановкой измерений». В каком-то смысле эта операция является изменением формата хранения данных, но более общим случаем. На данный момент поддержаны операции из BERT, их можно представить в двух формах:

Первая форма выражается формулой $dst_{ijk} = src_{ikj}$, т. е. является транспонированием нескольких матриц. Решается задача аналогично поэлементным операциям за тем лишь исключением, что в данном случае используется инструкция `scatter_vnchwconv`, которая работает с матрицами 16×16, что является гранулярностью в данном случае. Проблема, связанные с гранулярностью, решается аналогично: алгоритм использует схему «с наложением».

Вторая форма выражается формулой $dst_{ijkl} = src_{ikjl}$. В данном случае алгоритм осуществляет поблочное копирование из преположения, что размер младшего измерения кратен 32 байтам.

7 Заключение и дальнейшая работа

В рамках данной работы были решены следующие задачи:

1. Исследованы архитектуры нейронных сетей BERT и ResNet, основными целями для реализации были выбраны умножение матриц и векторные операции.
2. Исследованы различные инфраструктуры для создания и компиляции нейронных сетей (например, PyTorch, Tensorflow), и используемые ими аппаратные возможности для увеличения производительности.
3. Разработан набор операторов для целевой архитектуры DaVinci в инфраструктуре MLIR (диалекты CCE и HIVM).
4. Разработан набор тестов для оценки эффективности стратегий трансляции операций нейронных сетей.
5. Реализованы методы генерации целевых инструкций процессора для операций умножения матриц, фрактализации, векторных операций.

Корректность трансляции каждой крупноблочной операции из главы 6.4 была проверена на синтетическом наборе тестов. Результат работы оператора сравнивался с эталонным результатом, полученным при исполнении аналогичного кода на языке Python. Корректность работы всего компилятора экспериментально исследована и подтверждена на реальной нейронной сети BERT, содержащей 478 операторов.

Но, несмотря на это, планируются дальнейшие исследования. На данный момент компилятор находится в стадии активной разработки, поэтому отметим проблемы, которые были выяснены в процессе исследования и будут решены в будущем:

1. Процессоры Ascend содержат несколько ядер, по этой причине разделение крупноблочных операций на несколько независимых частей является одним из наиболее простых способов увеличить производительность (см. главу 5.2.7).
2. Замеры производительности показали, что двойная буферизация (см. главу 5.2.7) повышают производительность.

3. Инструменты синхронизации в текущем компиляторе используются неоптимально, улучшение алгоритмов синхронизации ведёт к снижению времени исполнения.

4. Методы полиэдральной компиляции (см. главу 4.2) могут быть использованы для оптимизации гнёзд циклов. Необходимы дополнительные исследования для изучения этих возможностей.

5. В операциях нейронных сетей многомерные массивы могут иметь динамические размеры (т. е. неизвестные на этапе компиляции), на данный момент их поддержка отсутствует.

Список литературы

[1] Документация к LLVM MLIR, 2024. — https://mlir.llvm.org/docs/.

[2] *Lattner, Chris*. MLIR: A Compiler Infrastructure for the End of Moore's Law. — 2020.

[3] *Vasilache, Nicolas*. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. — 2022.

[4] *Li, Mingzhen*. The Deep Learning Compiler: A Comprehensive Survey. — IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 03, pp. 708-727, 2021. — 2020.

[5] *Liang, Xiaoyao*. Ascend AI Processor Architecture and Programming / Xiaoyao Liang // Principles and Applications of CANN. — Elsevier, 2020.

[6] Документация к PyTorch, 2024. — https://pytorch.org/docs/stable/index.html.

[7] Документация к TensorFlow, 2024. — https://www.tensorflow.org/api_docs.

[8] Документация к ONNX, 2024. — https://onnx.ai/onnx/index.html.

[9] Спецификация TOSA, 2024. — https://www.mlplatform.org/tosa/tosa_spec.html.

[10] Спецификация Stable HLO, 2024. — https://openxla.org/stablehlo/spec.

[11] Документация к XLA, 2024. — https://openxla.org/xla.

[12] Документация к MindSpore, 2024. — https://gitee.com/mindspore/docs.

[13] Документация к ONNX Runtime, 2024. — https://onnxruntime.ai/docs/.

[14] Документация к TVM, 2024. — https://tvm.apache.org/docs/.

[15] Документация к Halide, 2024. — https://halide-lang.org/.

[16] Документация к Polly, 2024. — https://polly.llvm.org/docs/.

[17] Репозиторий AKG, 2024. — https://gitee.com/mindspore/akg.

[18] Intel 64 and IA-32 Architectures Software Developer's Manual. — 2024. — https://www.intel.com/content/www/us/en/content-details/819723/intel-64-and-ia-32-architectures-software-developer-s-manual-combi html.

[19] Arm A-profile A64 Instruction Set Architecture. — 2024. — https://developer.arm.com/documentation/ddi0602/latest/.

[20] Документация к Nvidia CUDA, 2024. — https://docs.nvidia.com/cuda/doc/index.html.

[21] Документация к AMD ROCm, 2024. — https://rocm.docs.amd.com/en/latest/.

[22] Tensor Slicing and Optimization for Multicore NPUs / Rafael Sousa, Marcio Pereira, Yongin Kwon et al. // *arXiv*. — 2023.