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.

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 задачи

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

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

Искуственная 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.

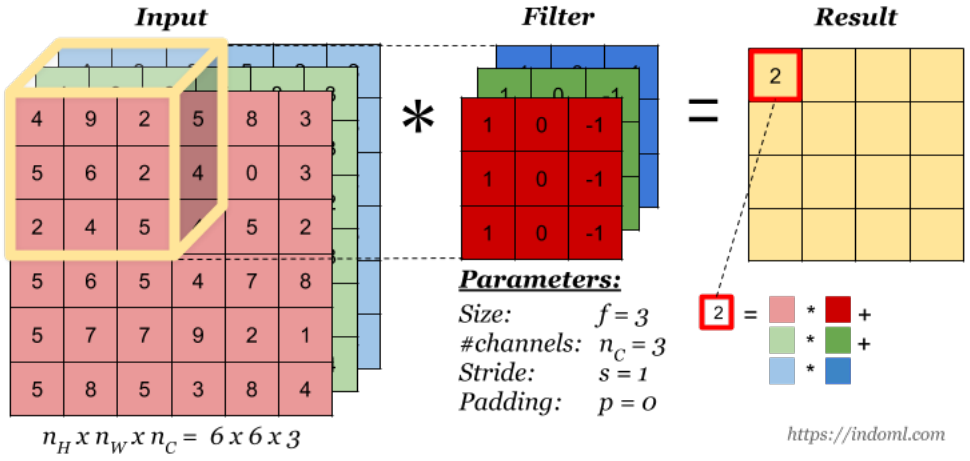

Рис. 1: Операция свёркти для изображения с тремя цветами

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

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

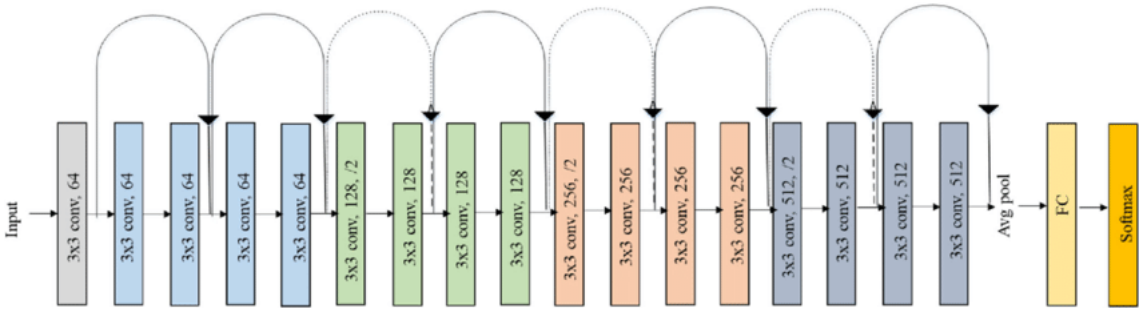

Рис. 2: Модель ResNet18

4 Умножение матриц и свёртка с точки зрения процессора и компилятора

4.1 Умножение матриц

Как было упомянуто ранее, нейропроцессоры в своей системе команд имеют операцию умножения матриц. Но, в силу особенностей разработки и применения процессоров, они имеют некоторые ограничения.

Во-первых, данные для умножения берутся из локального кэша, размер которого сильно ограничен (характерный размер — 64 КБ). Это означает, что в подавляющем количестве случаев необходимо производить перемножение по кусочкам. Удобный математический аппарат для этого — блочное перемножение матриц. Сама же техника называется *tiling* или *slicing*.

Во-вторых, сам процессор для удобства может требовать блочное расположение матриц. Например, в целевой архитектуре вся матрица должна быть разбита на блоки 16×16. Расположение элементов внутри блоков и блоков относительно друг друга может быть 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. Отметим, что в целевой архитектуре при умножении $C = A \times B$ матрица A должна быть заранее быть записана в формате `Zz`, матрица B — в формате `Zn`, а выходная матрица будет `Nz`.

В-третьих, в целях экономии целевой процессор поддерживает только умножение матриц у коротких типов. Для чисел с плавающей точкой это `float 16`, для целочисленных вычислений — `int 8`.

В связи с перечисленными выше причинами процедура перевода исходной крупноблочной операции в команды процессора (будем называть эту процедуру *lowering-ом*) нетривиальной. Можно выделить несколько стадий lowering-а:

1. Перевод исходных данных в сооответствующий блочный формат.
2. Копирование данных из оперативной памяти в локальный кэш.
3. Умножение матриц.
4. Повторение п. 2-3 необходимое количество раз.
5. Изменение формата хранения выходных данных (при необходимости).

Отметим, что п. 1 и 5 выходят за рамки исследования данной работы. Но, зачастую, они необходимы только на первом и последнем слоях нейронной сети, так как промежуточные данные используются только самим процессором.

4.2 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 —

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 Обзор существующих компиляторов и процессоров нейронных сетей

6 Инфраструктура LLVM MLIR

Как можно заметить из предыдущего параграфа, компиляторы из предыдущего параграфа 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.

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.

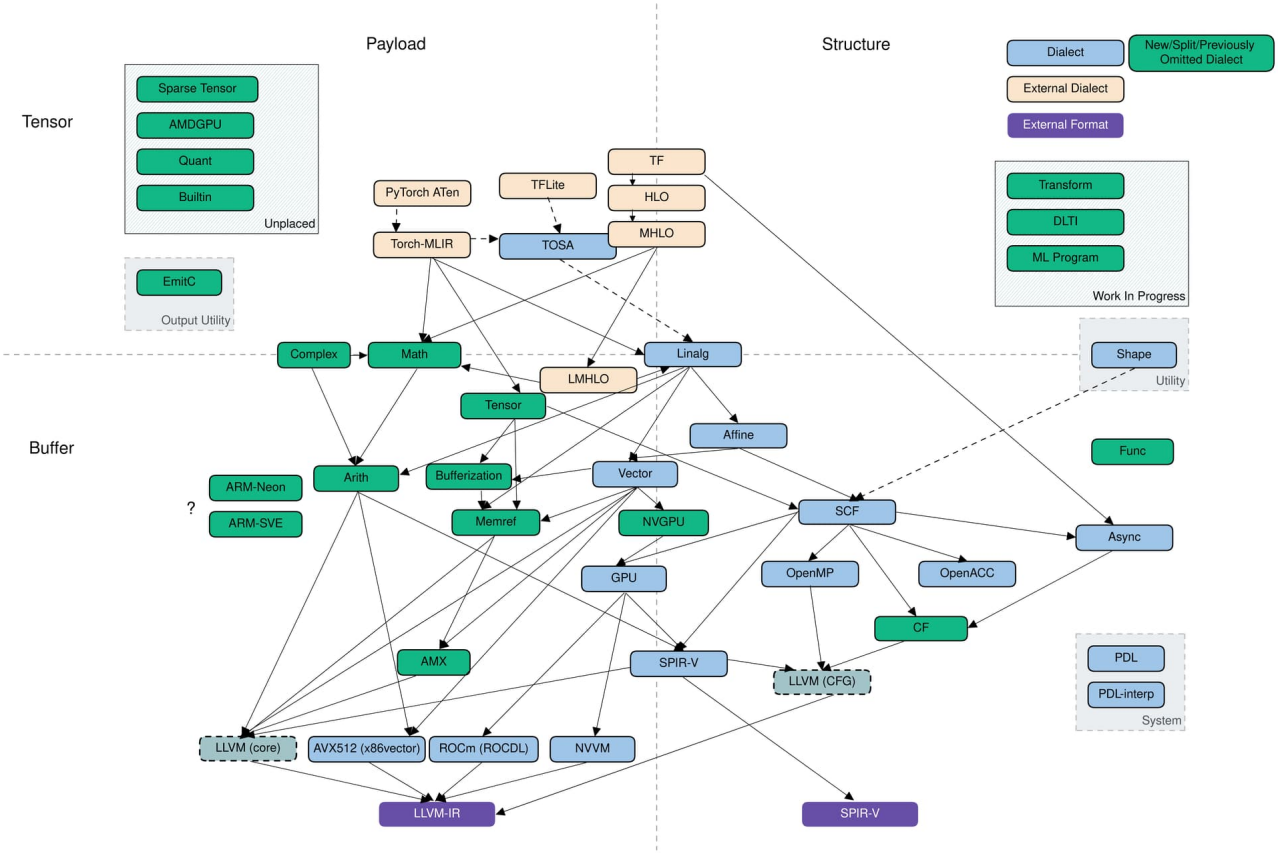

Рис. 3: Структура проекта MLIR и соотношения диалектов в них

7 Обзор архитектуры DaVinci

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). Схема архитектуры представлена на рисунке. Рассмотрим её основные особенности.

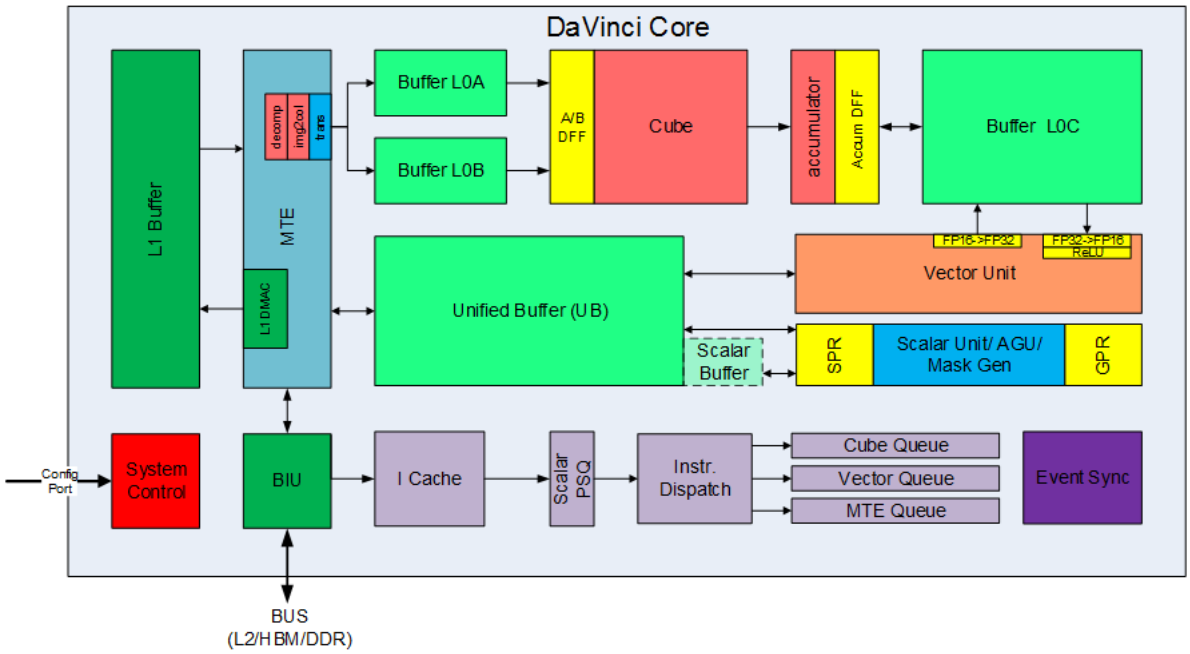

Рис. 4: Архитектура DaVinci

В ядре есть три вычислительных юнита: матричный, векторный и скалярный, которые используются для соответствующих вычилений. Как было сказано ранее, 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`. Элементы в матрице должны быть расположены в особом порядке (для матриц A, B, и C `Zz`, `Zn Nz` соответственно), более подробно это описывалось в главе, посвященной умножению матриц и операции свёртки.

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.

Во-первых, 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 lowering-у.

Отдельно стоит рассмотреть устройство матричного юнита, который представляет из себя систолический массив. Систолический массив — однородная сеть тесно связанных блоков обработки данных. Его схему для архитектуры DaVinci можно увидеть на картинке ниже.

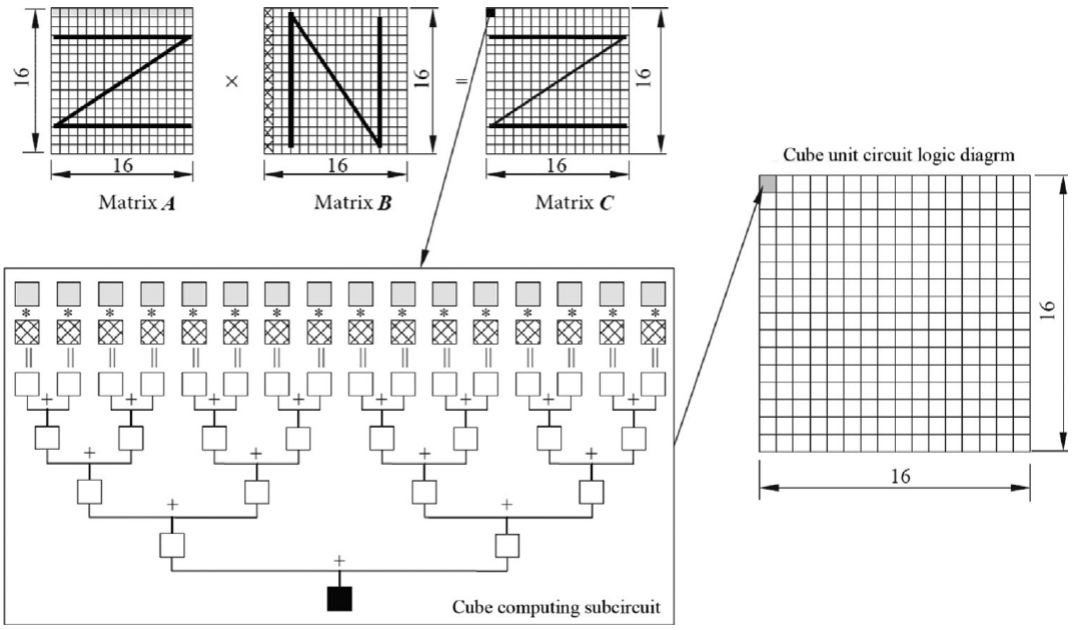

Рис. 5: Схема вычисления в матричном юните

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.

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

FIXME

8 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.

Пайплайн (последовательность действий какого-то процесса) компиляции выглядит следующим образом (FIXME: картинка?, пайплайн может поменяться): HLO -> ascend + tensor -> cce + tensor + affine + func -> hivm + memref -> llvm Подробнее описывать этапы, трансформации внутри каждого и переходы между ними не станем. Отметим лишь, что именно в процессе ловеринга из ascend в cce должно формироваться расписание операций для умножения матриц и свёртки, поэтому в дальнейшем только этот переход будет рассматриваться в данной работе.

9 Lowering операций и возможные стратегии

Изучив особенности целевой архитектуры, можно перейти к рассмотрению конкретных стратегий lowering-а 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, скорее всего, 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.

10 Реализация бенчмарка и стратегий lowering-а

WIP
Бенчмарк: реализованы стратегии, оптимальная синхронизация.
Реализация: ???

11 Результаты

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

12 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.