

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:

научная степень

Денисов Денис Денисович

Научный консультант:

научная степень

Сергеев Сергей Сергеевич



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	4
2	Постановка задачи	5
3	Обзор современных нейронных сетей	6
3.1	Общие соображения	6
3.2	Обзор модели BERT	6
3.3	Обзор модели ResNet	6
4	Умножение матриц и свёртка с точки зрения процессора и компилятора	7
4.1	Умножение матриц	7
4.2	Свёртка	7
5	Обзор существующих компиляторов нейронных сетей	9
6	Инфраструктура LLVM MLIR	10
7	Обзор архитектуры DaVinci	12
8	Функциональная структура компилятора	14
9	Lowering операций и возможные стратегии	15
10	Реализация бенчмарка и стратегий lowering-a	16
11	Результаты	17
12	Заключение и дальнейшая работа	18

1 Введение

Нейронные сети в последнее время испытывают большой подъём. Это происходит, прежде всего, благодаря успехам Chat GPT, которая показала новые возможности для обработки естественной речи. Стоит отметить, что развитие этой сферы происходит не только за счёт совершенствования точности ответов нейронных сетей. Например, разрабатываются 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, распространения 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 будут исследованы особенности целевой архитектуры и представлены способы генерации эффективного машинного кода.

2 Постановка задачи

При проведении исследования были поставлены следующие цели:

1. Исследовать архитектуру современных популярных нейронных сетей и типичные для них операции.
2. Исследовать существующие компиляторы, узнать их особенности и используемые в них оптимизации.
3. Изучить инфраструктуру LLVM MLIR и предоставляемые ею возможности для написания собственного компилятора.
4. Исследовать целевую архитектуру, принцип работы нейроматричного процессора и её язык ассемблера.
5. Исследовать и предложить методы генерации оптимального машинного кода для некоторых типичных операций нейронных сетей.

FIXME: delete it

Необходимо формально изложить суть задачи в данной секции, предоставив такие ясные и точные описания, которые позволят в последующем оценить, насколько разработанное решение соответствует поставленной задаче. Текст главы должен следовать структуре технического задания, включая как описание самой задачи, так и набор требований к ее решению.

3 Обзор современных нейронных сетей

3.1 Общие соображения

Основные применения: распознавание изображений и обработка естественного языка.

Основы устройства: слои и операции в них, обратное распространение ошибки (?)

3.2 Обзор модели BERT

Применяется для обработки естественного языка. Часто повторяющиеся операции: умножение матриц, сложение, `relu(?)`, мб ещё что-то.

3.3 Обзор модели ResNet

Применяется для распознавания изображений. Часто повторяющиеся операции: свёртка, сложение, `relu`.

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: по 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 крупно-блочной операции в команды процессора (будем называть эту процедуру *lowering-ом*) нетривиальной. Можно выделить несколько стадий lowering-a:

1. Перевод исходных данных в соответствующий 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 нейроматричных процессорах несколько сложнее, чем умножения. На некоторых архитектурах (FIXME: ссылка) она поддерживается 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:

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

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 соответствовать 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 нейронных сетей

Если в рамках работы писался какой-то код, здесь должно быть его описание: выбранный язык и библиотеки и мотивы выбора, архитектура, схема функционирования, теоретическая сложность алгоритма, характеристики функционирования (скорость/-память).

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, имплементация 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, представляющий 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 представляется 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-a 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.

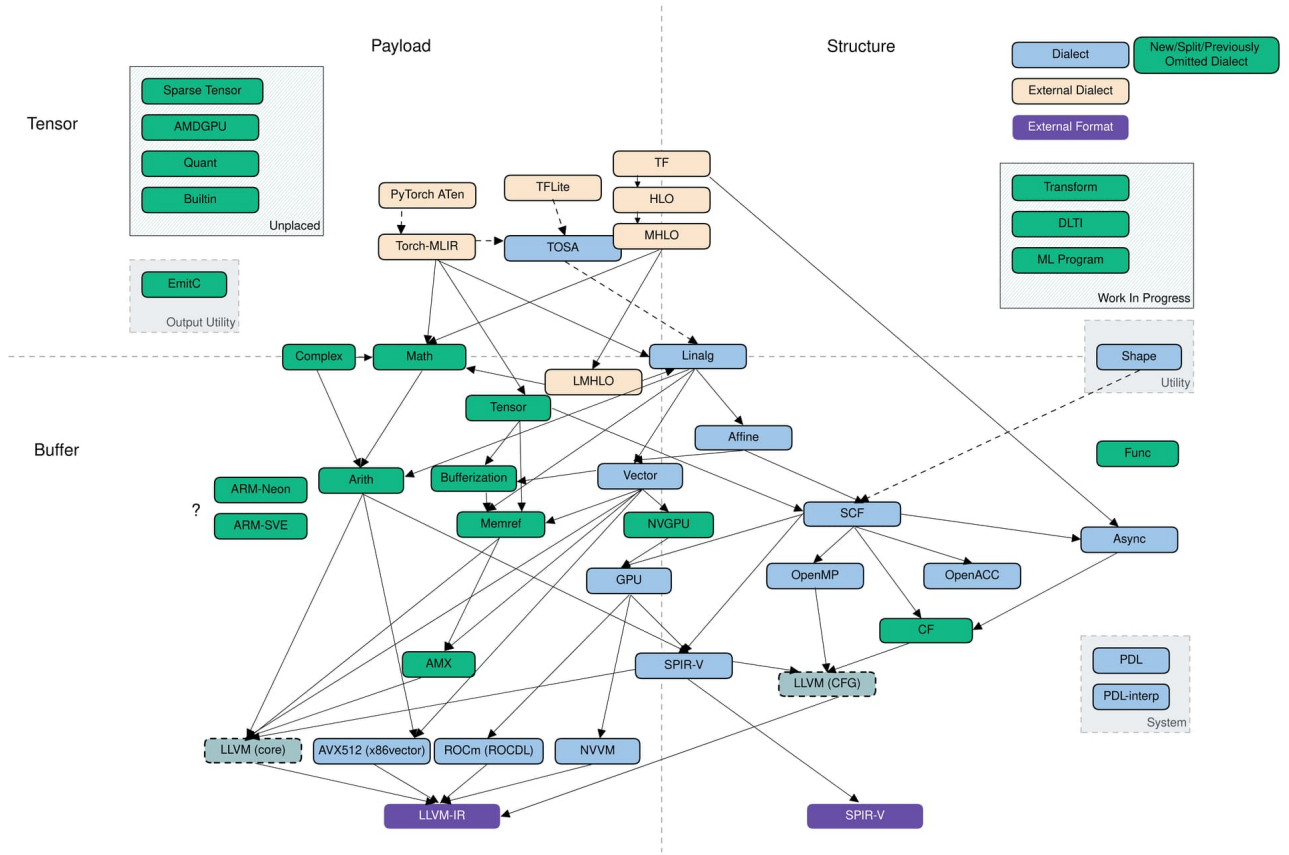


Рис. 1: Структура проекта MLIR и соотношения диалектов в них

7 Обзор архитектуры DaVinci

Архитектура DaVinci — нейропроцессор (NPU, neural processing unit), разработанный компанией 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 вычисления). Схема архитектуры представлена на рисунке. Рассмотрим её основные особенности.

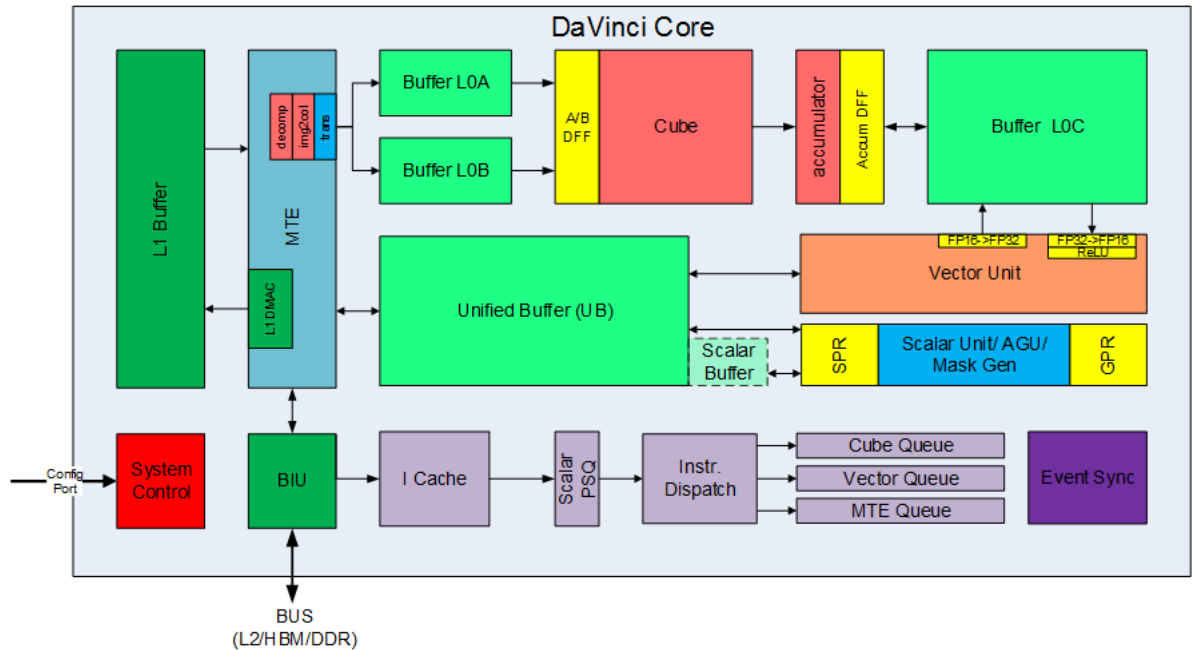


Рис. 2: Архитектура DaVinci

В ядре есть три вычислительных юнита: матричный, векторный и скалярный, которые используются для соответствующих вычислений. 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 Z_z , Z_n N_z 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 буфферов: 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-y.

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.

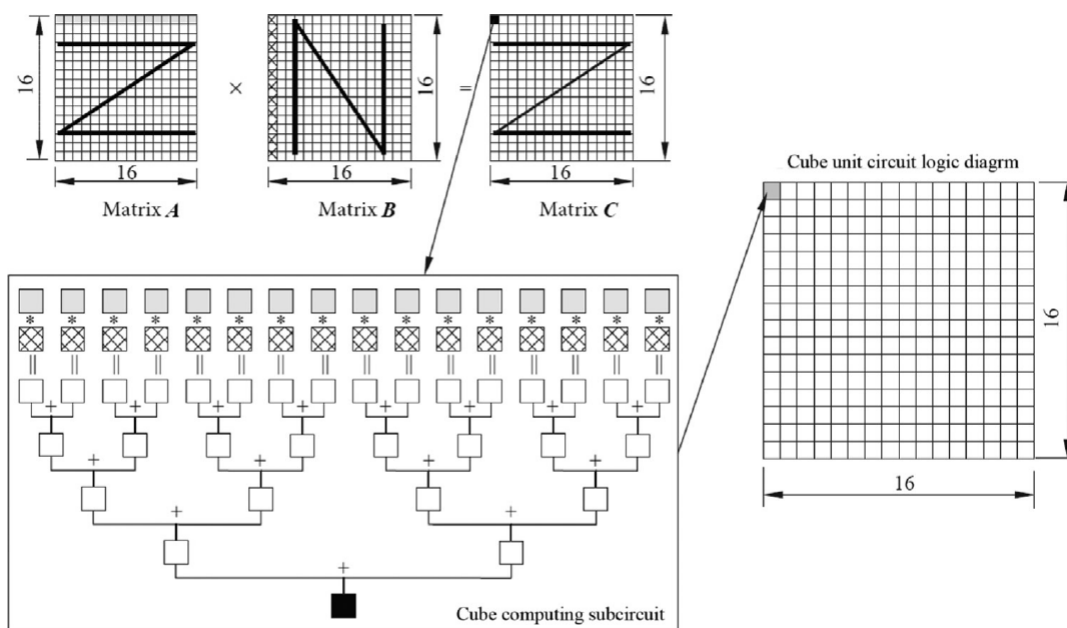


Рис. 3: Схема вычисления в матричном юните

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 по блоку, 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-a 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. `cse` — диалект операций, схожих с ассемблерными инструкциями. Основная его особенность 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 ловеинг в `llvm`.

Пайплайн (последовательность действий какого-то процесса) компиляции выглядит следующим образом (FIXME: картинка?, пайплайн может поменяться): `HLO -> ascend + tensor -> cse + tensor + affine + func -> hivm + memref -> llvm` Подробнее описывать этапы, трансформации внутри каждого и переходы между ними не станем. Отметим лишь, что именно в процессе ловеинга из `ascend` в `cse` 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 памятью 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-a

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, Долгонос М. С., Красовский В. Л.* // Девятая ежегодная конференция «Физика плазмы в Солнечной системе». — 2014.