ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Николюкин Михаил Николаевич

ОЦЕНКА ПРИМЕНИМОСТИ ДИСТИЛЛЯЦИИ ДЛЯ ОПТИМИЗАЦИИ ПРОЦЕДУР РАЗРЕЖЕННОЙ ЛИНЕЙНОЙ АЛГЕБРЫ, ВЫПОЛНЯЕМЫХ НА ПРОЦЕССОРЕ РЕДУЦЕРОН

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
О.В. Медведев

Руководитель
канд.физ.-мат.наук
А.В. Подкопаев

Консультант
канд.физ.-мат.наук
С.В. Григорьев

Оглавление

Аннотация

Эффективная обработка разреженной линейной алгебры является важной задачей во многих областях. Во многих работах было показано, что обработка сжатых данных на CPU неэффективна, ввиду частых промахов кэша. Существуют два основных направления исследований, пытающихся решить данную проблему: автоматические оптимизации и создания специализированного оборудования. Возникает идея объединить оба подхода. В сфере функциональных языков программирования давно развивается идея переписывания программ с целью оптимизации, и одна самых сильных техник носит название дистилляция. С другой стороны для эффективного исполнения функционального кода разработано множество специализированного оборудования, но сильнее всего выделяется процессор под названием Редуцерон. В ходе данной работы Редуцерон и Дистиллятор были доработаны и объединены в единый пайплайн для обработки разреженной линейной алгебры. Проведено множество экспериментов, которые раскрыли возможности дистилляции для значительного уменьшения количества операций с памятью, при работе со специализированны процессором, а также продемонстрирован потенциал развития данной идеи.

Ключевые слова: разреженная линейная алгебра, Редуцерон, дистилляция, функциональное программирование, ПЛИС.

Efficient processing of sparse linear algebra is an important task in many fields. It has been shown that processing compressed data on the CPU is inefficient, due to frequent cache misses. There are two main areas of research trying to solve this problem: automatic optimizations and the creation of specialized equipment. The idea arises to combine both approaches. In the field of functional programming languages, the idea of rewriting programs for optimization has been developing for a long time, and one of the most powerful techniques is called Distillation. On the other hand, a lot of specialized equipment has been developed for the efficient execution of functional code, but the processor called Reduceron stands out the most. In the course of this work, the Reduceron and Distiller were refined and combined into a single pipeline for processing sparse linear algebra. A lot of experiments have been conducted that have revealed the possibilities of Distillation to significantly reduce the number of memory operations when working with a specialized processor, and the potential for the development of this idea has also been demonstrated.

Keywords: sparse linear algebra, Reduceron, distillation, functional programmin FPGA.

Введение

Линейная алгебра – фундаментальная наука, которая нашла применение во многих сферах, включая рекомендательные системы [1], компьютерное зрение [2] и анализ графов [3]. Данные, представленные в виде матриц и векторов, могут быть легко проанализированы и преобразованы с помощью высокооптимизированных процедур. Однако, на практике часто встречаются массивы данных с большим количеством нулей. Например матрица представляющая связность социальной сети YouTube содержит только 2.31% не нулевых значений [4]. Такая высокая разреженность данных приводит к неэффективности классических методов вычислений и хранения данных, так как значительная часть выделенной памяти заполнена нулями, а большинство вычислений являются тривиальными операциями.

Основным подходом для эффективной обработки разреженных данных является сжатие, которое сохраняет информацию только о ненулевых элементах. Такие библиотеки как GraphBLAS, SciPy, ALGLIB, Armadillo и EJML реализуют множество различных форматов сжатия и операций над ними и широко используется во многих областях связанных с разреженными данными: глубокие нейронные сети [5], анализ сетей [6] и вычислительная биология [7].

Основной проблемой высокоуровневых решений (библиотек и фреймворков) на данный момент является слабая утилизация оборудования [4,8,9]. Современные процессоры при чтении элемента из памяти также кэширует небольшую область вокруг него, но после сжатия индексация элементов становится неупорядоченной, а значит часто из всего кэша нам нужен только один элемент, что приводит к низкой утилизации кэша и увеличению количество запросов в память. Таким образом большую часть времени вместо полезных вычислений процессор ожидает данные из памяти.

Альтернативный подход – использование интегральных схем специального назначения, в частности программируемых логических интегральных схем (ПЛИС). Специальное оборудование хорошо зарекомен-

довало себя в задачах, где существующие процессоры общего назначения неэффективны. Такой подход получил распространение в машинном обучении [10], медицине [11] и обработке изображений [12]. Ряд работ посвящены аппаратном ускорению операций разреженной линейной алгебры [8,13–15]. Несмотря на впечатляющее результаты данные работы все еще остаются экспериментальными и не доходят до реализации на реальном оборудовании. Более того, такие решения часто слишком замкнуты в себе, и не поддаются высокоуровневым оптимизациям.

Другим возможным подходом для ускорения обработки сжатых данных является оптимизации и уменьшение интенсивности обращений к памяти. Существуют разные виды оптимизаций, но в этой работе будет рассмотрена только оптимизация под названием fusion. Данная оптимизация подразумевает объединение нескольких последовательных операций в одну с целью убрать все промежуточные структуры и вычисления. В общем случае подобные оптимизации очень сложны, поэтому часто на практике представлены в виде обработки частных случаев, как это сделано в GraphBLAS [16].

Автоматические оптимизации используются во многих сферах, но одной из самых обширных стали функциональные языки программирования. С конца прошлого века изучалась возможность автоматического переписывания программы с целью уменьшения вычислительной сложности и количества промежуточных структур [17, 18]. Обобщением данной идеи стала суперкомпиляция [19], однако даже она имеет множество ограничений. Так суперкомпиляция может проводить лишь линейные оптимизации [20]. Более мощным обобщением стала Дистилляция [21], которая использует идеи суперкомпиляции, но способна проводить сверхлиненые оптимизации, а также имеет больший потенциал для устранения промежуточных объектов [22].

Одной из фундаментальных проблем функциональных языков программирования является не оптимальное выполнения на стандартных GPU и CPU, ввиду различной идеологий лямбда-исчислений и машины Тьюринга. Множество работ посвящены проектированию специального аппаратного обеспечения для функциональных языков [23–25], но в

данной работе будет рассмотрен Редуцерон [26]. Это процессор, специально разработанный для эффективного выполнения редукции. Одной из главных преимуществ данной архитектуры является низкоуровневый параллелизм и широкая память, позволяющие выполнять несколько шагов редукции за один такт. Реализован процессор на ПЛИС, что позволяет переносить реализацию на различные чипы и проводить тесты в специализированных программах для симуляции. Но при этом Редуцерон является экспериментальным процессором и имеет множество технических ограничений.

Цели и задачи

Целью данной работы является оценка применимости дистилляции для оптимизации процедур разреженной линейной алгебры, выполняемых на специализированном процессоре Редуцерон.

Для достижения цели ставятся следующие задачи.

- Объединить дистилляцию и исполнение на Редуцероне в общий пайплайн.
- Провести серию экспериментов с использованием пайплайна на эмуляторе Редуцерона и с помощью симуляции ПЛИС.
- Проанализировать результаты и сравнить с другими существующими решениями.

Обзор последующих глав

В Главе 1 описаны все инструменты, которые будут использованы в данной работе.

Глава 2 содержит технические подробности объединения всех инструментов в единый пайплайн.

Глава 3 содержит описание всех сделанных экспериментов вместе с результатами и анализом.

В заключительной главе подведены итоги, а также описаны все результаты.

Обзор предметной области

В этой главе будет дан краткий обзор существующих инструментов для работы с разреженной линейной алгеброй. Будут описаны современные подходы и исследования в области эффективной обработки разреженных данных, а также представлены инструменты и технологии, используемые в данной работе. Узкоспециализированные и ручные оптимизации не важны для этого исследования, поэтому описаны не будут.

Разреженная линейная алгебра

Разреженная данные естественным образом появляются во многих областях. Например при анализе графа знакомств в социальной сети матрица смежности будет заполнена большим количеством нулей, ввиду того, что большинство людей незнакомы друг с другом [6]. Хранить такие данные в явном виде бессмысленно, так как это требует выделение памяти для сохранения нулей, и при применений операций большинство вычислений будут тривиальной обработкой нулей.

Решением этой проблемы является сжатие, при котором сохраняется информация лишь о ненулевых значениях. Для примера рассмотрим формат под названием *список координат* (COO – Coordinate list). Это один из самых простых форматов в котором каждое ненулевое значение сохраняется в виде (строка, столбец, значение). Преимуществами данного формата является небольшой размер, зависящий только от количества полезной информации, и простота, позволяющая легко конвертировать его в другие форматы. С другой стороны неупорядоченная индексация делает все арифметические операции малоэффективны из-за необходимости чтения данных из разных мест памяти, что приводит к неэффективному использованию кэша процессора и большим задержкам при частом обращении к памяти. Существуют упорядоченные форматы (CSR, CSC) позволяющие выполнять часть операций эффективно, но для большинства алгоритмов этого недостаточно [27]. Таким образом на практике сжатый формат данных становится узким местом

для алгоритмов, из-за задержек операций с памятью и неэффективной утилизации оборудования [4, 8, 28].

Большинство библиотек (SciPy, ALGLIB, Armadillo, EJML) для работы с разреженной линейной алгеброй предоставляют реализацию различный форматов сжатия и операций над ними, оставляя на пользователе ручные оптимизации и выбор форматов. Более продвинутым инструментом является стандарт GraphBLAS [16]. Он спроектирован так, чтобы реализации имели возможность проводить автоматические оптимизации, например выбора оптимального формата сжатия. Наиболее распространенной реализацией на CPU является *SuiteSparse* [29], которая вошла в стандарт MATLAB и используется в реализации модуля *RedisGraph* [30]. Так же существуют экспериментальные реализации на GPU [31]. Несмотря на выдающиеся результаты, текущие реализации всё еще страдают от общих проблем при обработке разреженных данных, а именно частых промахов кэша и слабой утилизация оборудования.

Fusion

Fusion не имеет строгого определения. Обычно когда говорят о данной оптимизации подразумевают объединение некоторых сущностной в одну. Это приводит к уменьшению промежуточных структур и лишних вычислений, что в случае разреженной алгебры особенно важно. Например, при наивном вычислении последовательного сложения матриц, каждый оператор создаст новую промежуточную матрицу, что очень неэффективно. После оптимизации должна остаться только явная формула для ответа, без ненужных вычислений, лишнего выделения памяти и, самое главное, меньшим количеством обращений к памяти.

Большинство высокоуровневых решений для разреженной линейной алгебры используют fusion в некотором виде. Например интерфейс GraphBLAS подразумевает возможность объединения операций в цепочки для дальнейшей оптимизации, но текущие реализации [29, 31] могут оптимизировать только некоторые частные случаи и не реализу-

ют полностью автоматическое слияние.

В императивных языках программирования fusion часто ассоциируется с заменой нескольких циклов одним. Это увеличивает локальность памяти, что повышает эффективность кэша и помогает избежать накладных расходов на структуры управления циклом. Этот тип оптимизации обычно основывается на анализе информационных зависимостей [32], но этот метод не справляется с не аффинным индексированием и вложенными циклами, что характерно для разреженных структур данных.

В функциональных языках fusion представлен как набор правил для переписывания программы для упрощения композиции функций. Например на языке Haskell выражение **map f . map g** можно переписать в упрощенный вариант **map (f . g)**. Обобщением таких правил является суперкомпиляция [19]. Суперкомпиляция объединяет в себе множество техник для оптимизации функциональных языков, такие как частичные вычисления [17] и дефористация [18]. Однако суперкомпиляция ограничена лишь линейными оптимизациями [20]. Существуют работы [33] пытающиеся применить подобный подход для эффективной работы с памятью, но оптимизация оказывается неэффективна при неравномерной индексации, как у сжатых разреженных данных.

Но у суперкомпиляции существует более сильная альтернатива под названием Дистиляция [21]. Это обобщение суперкомпиляции, которое позволяет автоматически проводить сверхлинейные оптимизации и имеет больший потенциал для fusion оптимизации [22]. Главная проблема дистиляции в том, что ввиду своей сложности существует лишь частичная реализация, которая тем не менее способна проводить нетривиальные преобразования, которые, возможно, могут помочь решить проблему автоматической оптимизации процедур разреженной линейной алгебры.

Аппаратное обеспечение специального назначения

Аппаратное обеспечение специального назначения (АОСН) – интегральная схема спроектированная для выполнения конкретной задачи. По сравнению с аппаратным обеспечением общего назначения, такого как CPU и GPU, часто предлагает лучшую производительность и меньшее потребление энергии, но требует сложной разработки и дорогого изготовления. Более универсальной альтернативой является программируемые логические интегральные схемы (ПЛИС), главной отличительной чертой которых является возможность конфигурации после изготовления, что очень удобно для проектирования и экспериментов. В данной работе разница между АОСН и ПЛИС несущественна, поэтому далее они будут рассмотрены вместе как специализированное оборудование.

Часто специализированного оборудования используют в случаях, когда невозможность эффективно использовать стандартные решения [10–12]. Это мотивировало множество исследований по проектированию интегральных схем для быстрых операций разреженной алгебры [8, 13–15]. Во всех работах упор сделан на реализацию небольшого количества ключевых функций, таких как умножение матриц или матрицы с вектором, и оптимизацию работы с памятью и индексированием, что дает заметное ускорение. Например умножения матриц с *SpArch* [8] в симуляции показывает выдающиеся результаты превосходя CPU (Intel MKL) и GPU (cuSPARSE) как по скорости, так и по энергопотреблению. Однако подобные решения имеют ряд проблем, ограничивающие их применимость. Во-первых маленький набор возможных операций делает невозможным полную реализацию большинства алгоритмов, например поиск максимального независимого множества в графе, а значит будет требоваться частое взаимодействие с главным процессором. Во-вторых такие решения замкнуты в себе и не могут быть использованы вместе с оптимизациями описанными выше.

Другой задачей, для которой оправданно использование специализированного оборудования является эффективное исполнение функ-

циональных языков программирования. С 80-х годов идет дискуссия о разработки специализированной машины редукции [23, 34], так как классическая архитектура процессоров плохо подходит для редукции. Одной из последних и наиболее продвинутых работ является процессор под названием Редуцерон [26]. Данная архитектура направленна на увлечение пропускной способности между процессором и памятью и была успешно реализована на ПЛИС. В дальнейшей работе [35] в архитектуру добавили низкоуровневый параллелизм, позволявший выполнять несколько шагов редукции параллельно, а также добавили динамический анализ для уменьшения числа тривиальных редукций. Как и большинство описанных выше работ, несмотря на возможность запуска на настоящей ПЛИС, Редуцерон остается экспериментальным процессором с большим количеством технических ограничений. Например отсутствует возможность загружать программу на ПЛИС из памяти, поэтому для запуска требуется каждый раз переконфигурировать ПЛИС.

Выводы

Задача эффективной работы с разреженной линейной алгебры является важной во многих областях. Уже существует множество библиотек и фреймворков для эффективной работы как на CPU, так и GPU. Однако исследования показывают, что текущие реализации не используют всю мощность современного оборудования из-за частых обращений к памяти, что подталкивает к двум направлениям исследований: автоматизация fusion для уменьшения числа промежуточных структуры и использование специализированного оборудования.

Возникает идея объединить оба направления. Дистилляция – потенциально сильный инструмент автоматической оптимизации, а для того, чтобы максимально раскрыть возможности оптимизации над функциональными языками программирования хорошо подходит специальный процессор Редуцерон. Исследование данной пары позволит оценить возможности функционального программирования для построения эффективного пайплайна выполнения процедур разреженной линейной

алгебры

1. Описание инструментов

Пайплайн состоит из трех частей: дистилляция, генерация описания аппаратуры для Редуцерона и набор утилиты для работы с ПЛИС. Написание реализаций этих этапов с нуля выходит за рамки данной работы, поэтому в этой главе будет дан краткий обзор всех используемых решений и приведены ссылки на исходный код.

1.1. Дистиллятор

Дистиллятор на данный момент имеет несколько реализаций. Решение [1], предложенное автором статьи [21] не полное и не детерминируется в некоторых случаях. Поэтому был выбран более стабильный форк[2] оригинального дистиллятора, который лучше покрыт тестами и активно развивается.

Дистиллятор работает с собственным языком `.pot`, который является простейшим функциональным языком программирования. В нем присутствуют лишь функции и алгебраические типы данных, отсутствует типизация и такие примитивные типы, как целые числа.

Также данный репозиторий уже содержит небольшую библиотеку на языке `.pot` для разреженной линейной алгебры и набор матриц из SuiteSparse Matrix Collection[3]. В данной реализации используется дерево квадрантов как формат сжатия разреженных матриц. Данный формат рекурсивно разбивает матрицу на 4 подматрицы до тех пор, пока либо размер матриц не 1x1, либо матрица не пуста. Данный формат легко реализуется в терминах функционального программирования, позволяет использовать подход "Разделяй и властвуй", не требует явного индексирования и при этом не сильно проигрывает в коэффициенте сжатия классическим фарматам, например сжатому хранению строк [36].

Одной из проблем языка `.pot` это отсутствие примитивов, поэтому

[1] `https://github.com/poitin/Distiller`

[2] `https://github.com/YaccConstructor/Distiller`

[3] `https://sparse.tamu.edu`

натуральные числа выражены через алгебраические типы. Так например число 3 будет представляться как 3 раза применненый конструктор `Succ` к значению `Zero` . Данный подход крайне неэффективен, поэтому для экспериментов будут рассматриваться только операции с булевыми матрицами.

1.2. Редуцерон

Изначально Редуцерон разрабатывала команда из Йоркского университета [26], выкладывая исходники на отдельный сайт[4]. После прекращения работы авторов, Редуцерон развивался сообществом как open-source проект[5] и именно этот форк будет использован в данной работе как основной.

Используемый репозиторий состоит из 4х больших частей.

- Red Lava – библиотека для описания логических интегральных схем на языке Haskell. Форк библиотеки York Lava, добавляющий функциональность необходимую для реализации Редуцерона
- Компилятор небольшого функционального языка F-lite в байт-код Редуцерона. Имеет минималистичный синтаксис, поддерживает алгебраические типы данных и натуральные числа, не имеет типизации.
- Реализация процессора, написанная с помощью библиотеки Red Lava, которая способна генерировать конфигурацию для ПЛИС.
- Эмулятор Редуцерона, написанный на C, способный быстро выполнить байт-код Редуцерона и подсчитать богатую статистику по многим параметрам.

Отдельно стоит выделить то, что Редуцерон обязан возвращать ровно одно натуральное 15-битное число как ответ. Поэтому ко всем процедурам, возвращающим матрицы будет применяться свертка, считающая сумму значений для проверки корректности.

[4] `https://www.cs.york.ac.uk/fp/reduceron/`

[5] `https://github.com/tommythorn/Reduceron`

1.3. Инструменты для работы с ПЛИС

Изначально Редуцерон разрабатывался под ПЛИС семейства Xilinx, поэтому для работы был выбран набор инструментов под названием Xilinx Vivado[6]. Он представляет из себя различные подсистемы, для разработки аппаратного обеспечения, объединенных единым графическим интерфейсом. В этой работе Vivado будет использоваться для сборки всех файлов, генерируемых реализацией Редуцерона, в один проект для дальнейшей конфигурации ПЛИС, а также проведения потактовых симуляций.

1.4. Выводы и результаты по главе

В этой главе приведено описание инструментов, которые необходимы для проведения экспериментов. Также дано обоснование выбора конкретных реализаций и описаны важные детали, необходимые для понимания переведенных далее улучшений.

[6] https://www.xilinx.com/products/design-tools/vivado.html

2. Пайплайн

Для обьединения всех инструментов в единый пайплайн, способный выполнять тяжелые операции разреженной линейной алгебры, потребовалось множество доработок. В этой главе будут подробно описаны все новые модули, а также расширения существующих, выполненные в рамках данной работы.

2.1. Транслятор `.pot` в F-lite

Первой подзадачей стало написание транслятора с языка дистиллятора в язык Редуцерона. Оба языка имеют схожие возможности, но сильно различаются в деталях синтаксиса. Для простоты было решено сделать транслятор модулем дистиллятора написанным на Haskell. Таким образом в дистилляторе появлялась возможность вывода в альтернативном формате, а реализация транслятора требовала только преобразование уже готового абстрактное синтаксического дерева (АСД) `.pot` языка в текст F-lite.

Для возможности ручной отладки и проверки корректности трансляции дополнительным условием было человекочитаемость результата. Поэтому основой нового модуля стала библиотека pretty[7], позволяющая описывать программу как набор элементарного текста объединенный комбинаторами в сложный документ, который затем можно записать в файла передав необходимые настройки форматирования.

Также в данный модуль вошли вспомогательные функции для модификации исходной программы. Была добавлены конвертация из натуральных чисел `.pot` языка в примитивы F-lite, а так же набор сверток, позволяющих трансформировать `.pot` программы возвращающие матрицы в корректные F-lite программы, возвращающие одно натурально число.

[7] `https://hackage.haskell.org/package/pretty`

2.2. Генерация датасета

Для проведения экспериментатор требовалось собрать датасет на языке F-lite из различных программ до и после дистилляции с различными матрицами на вход. Для генерации такого датасета был написан еще один модуль для дистиллятора.

Основой стал фреймворк testy[8] для тестирования программ на Haskell. Генерацию было решено сделать через по двум причинам. Во-первых для тестирования дистиллятора уже были написаны некторые вспомогательные функции, которые можно переиспользовать. Во-вторых это позволяло не перегружать существующий интерфейс REPL реализованный в Main.hs, а вместо этого запускать генерацию одним вызовом из командной строки.

Основная функция генерации `genAndWright` принимает 5 параметров:

- `progModifier :: (Prog -> Prog)` – модификация АСД программы. Необходим для трансформации программы перед трансляцией в F-lite, в частности добавлению свертки матриц.
- `fileToDistill :: String` – `.pot` файл с основной логикой, которую хотим протестировать до и после дистилляции
- `importsForDistill :: String` – директория с файлами, где лежат все, что использует `fileToDistill`, в частности файл LinearAlgebra с реализацией всех операций с разреженной линейной алгеброй
- `bindingsInfo :: [[(String, FilePath)]]` – набор списков входных аргументов. Представляет собой список списков пар "имя переменной - путь до файла в котором лежит значение"
- `outputDir :: String` – имя папки в которую будут сохранены все новые файлы

[8] `https://hackage.haskell.org/package/tasty`

Данная функция для каждого списка параметров `args` внутри `bindingsInfo` генерирует четверку файлов в папке `outputDir`: пару `.pot` и пару F-lite файлов до и после дистилляции с подставленными `args`.

Также для удобства дальнейшего анализа все генерируемые файлы имеют общий суффикс вида:
"{название}?{аргумент_1}={значение}&{аргумент_2}=..."
Таким образом по названию файла можно полностью восстановить все аргументы генерации, что полезно при отладке и автоматической обработке.

Важно отметить, что все импорты подставляются в файл до оптимизации, а все входные матрицы после. Это позволяет дистиллятору проводить сложные оптимизации над кодом из библиотеки, но при этом он не может использовать знания о конкретных входных значениях. Например если подставить все значения до дистилляции, то в некоторых случаях оптимизатор сможет вывести сразу ответ, что не поможет оценить применимость дистилляции для общих оптимизаций.

2.3. Модификация компилятора F-lite

Редуцерон накладывает множество ограничений на байт-код, который может исполнять, например максимальное количество аппликаций в теле функции. Большинство таких ограничений компилятор F-lite скрывает от пользователя преобразовывая байт-код таким образом, чтобы соответствовать всем параметрам реализации. Одно из последних заметных для пользователя условий, накладываемых на код – это ограничение в максимум 7 аргументов у функции. Хоть при ручном написании кода функции с большим числом аргументов встречаются редко, то при дистилляции количество аргументов часто может превышать заданный лимит. Поэтому было решено расширить возможности компилятора F-lite, добавив шаг автоматического уменьшения арности функций.

Упрошено байт-код Редуцерона можно представить как набор функций. Каждая функция хранит в себе свою арность, главную аппли-

цию и список вложенных аппликаций. Если в ходе выполнения программы на Редуцероне возникла необходимость вызвать определенную функцию, то последовательно выполняются следующие шаги:

1. Выбирается нужная функция из байт-кода
2. Со стека снимется нужное число аргументов и проставляется во все аппликации
3. Вложенные аппликации добавляются в конец кучи
4. Главная аппликация записывается наверх стека

Можно заметить, что раз мы не можем считать много аргументов со стека за раз, то можно было бы считать часть, сохранить в куче, а затем уже доставать значения из кучи по необходимости. Чтобы добиться такого поведения был реализован следующий алгоритм разделения функций в байт-коде:

- Пусть мы нашли функцию `fun` арности n и хотим уменьшить её на t
- Введем 2 новые функции: `fun'` и `fun2`
- fun' будет иметь вид `(t, fun2, [arg_1, arg_2, ..., arg_t])`
- fun2 будет почти полной копией оригинальной функции, но все ссылки на первые t аргументах будут заменены на ссылки на кучу с правильным смещением, а все остальные номера аргументов сдвинуты на t.
- Далее при необходимости можно рекурсивно запустить данный алгоритм на `fun2`

К сожалению, данный алгоритм не работает вместе с оптимизацией, добавляющей отдельные регистры для ускоренной работе с примитивами [35]. Ращение проблемы не было найдено, поэтому данную оптимизацию в дальнейших экспериментах не использовалась. Стоит однако

заметить, что в связи с отсутствием примитивов в `.pot` все матрицы в экспериментах булевы, а значит отсутствие регистров не повлияет на оценку работы дистиллятора.

2.4. Модификация Редуцерона

Как было упомянуто выше, реализация Редуцерона написана на Haskell с помощью библиотеки Red-Lava. Эта библиотека, кроме описания схемы, так же отвечает за генерацию аппаратного описания вместе с инициализацией всей необходимой памяти. Оказалось, что часть с генерацией устарела и не может быть обработана современными инструментами для работы с ПЛИС. Для исправления этого, было добавлено несколько мелких изменений.

- Поправлена логика генерации названий для блоков ram, так чтобы имена модулей памяти совпадали с соответствующими названиями в VHDL описании Редуцерона
- Добавлено описание нового чипа ПЛИС `xc7a200tfbg676-3`, для возможности выбора современного чипа в качестве целевой платформы для Редуцерона
- Изменено расширение файла с `.txt` на `.coe` для файлов инициализации памяти, для корректного распознавания инструментами Xilinx

В ходе экспериментов оказалось что узким местом текущей реализации Редуцерона является размер памяти для хранения байт-кода. Даже небольшие программы с разреженными матрицами 64x64 не помещались в текущее ограничения. Проблема находится в архитектуре процессора, где адреса функций представляют из себя 10-битные указатели. При детальном изучении текущей реализации указателей выяснилось, что один бит, выделенный на инструкцию, никак не используется. Добавление этого бита к адресам функций позволило без значительных изменений в архитектуре расширить адресное пространство в 2 раза.

Этого все еще недостаточно для экспериментов с большими матрицами, но это позволило для некоторых нетривиальных примеров проводить честную симуляцию исполнения на ПЛИС.

2.5. Модификация эмулятора Редуцерона

Редуцерон имеет эмулятор написанный на языке C, для быстрого выполнения байт-кода и сбора подробной статистики. В отличие от потактовой симуляции ПЛИС, эмулятор может показать сколько раз выполнялся сопоставление с образцом, максимальную утилизацию разных видов памяти и многое другое. Вся статистика выводилась в терминал в человекочитаемом виде, поэтому для автоматизации сбора результатов была добавлена альтернативная запись в `json` файл.

Другой важной особенностью эмулятора является возможность легкого увеличения лимитов памяти. Эмулятор воспроизводит только логику работы Редуцерон, а не полную реализацию, поэтому явно не зависит от размеров адресных пространств и инструкций. Все ограничения проверяются простым сравнением с заранее заданной константой. Таким образом на эмуляторе возможно запустить тесты с большими матрицами, внося минимальные изменения в код.

2.6. Автоматизация

Для автоматизации процесса тестирования и сбора результатов был разработан набор оберток и скриптов на языке Python.

- Программные обертки над Дистиллятором и Редуцероном. Инициализируются путями до корня проекта. Скрывают вызов подпроцессов для сборки и запуска проектов за удобным интерфейсов
- Скрипт для запуска всего паплайна, от генерации тестовых программ, до распределенной эмуляции.
- Скрипт для сборки всех данных в один `csv` файл
- Набор вспомогательных скриптов для анализа данных

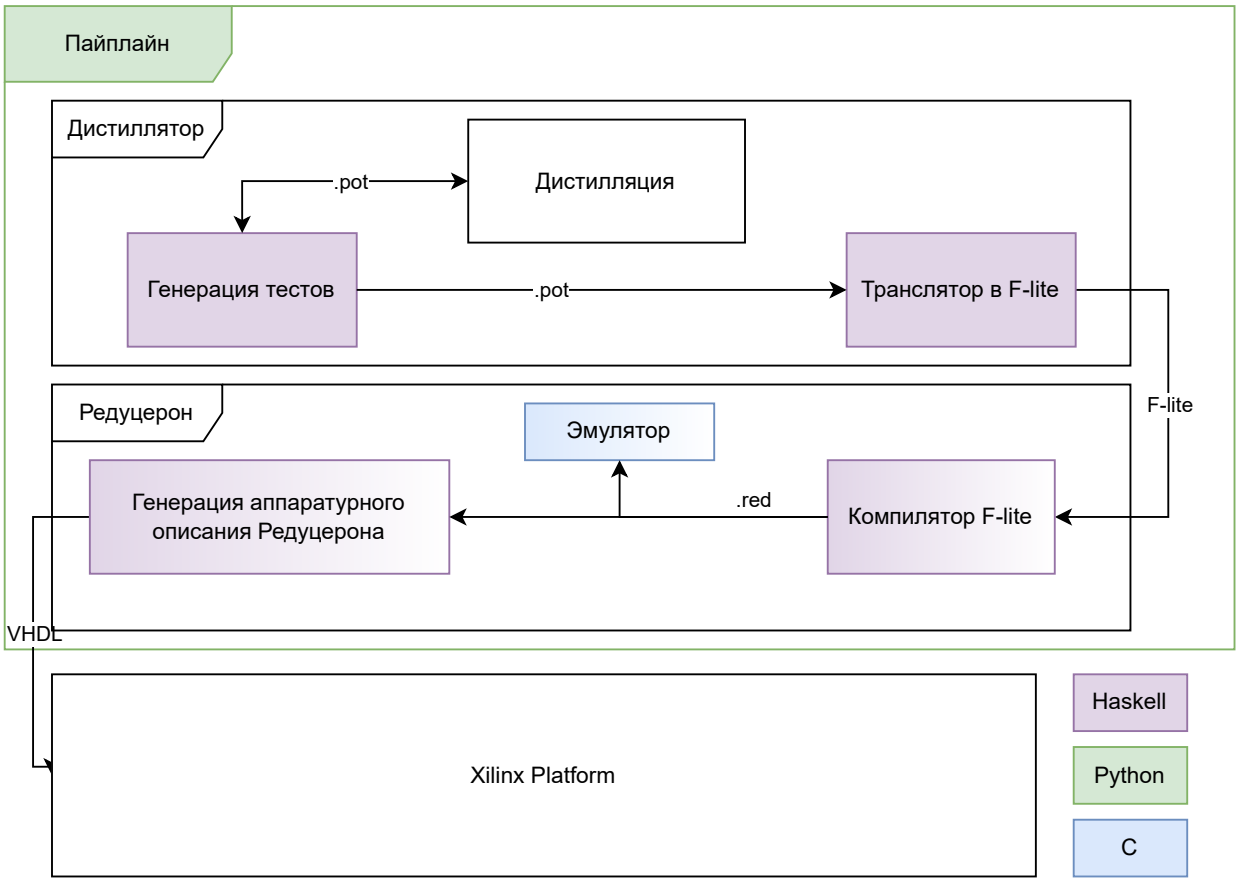

Рис. 1: Схема всего пайплайна. Цветами выделен части написанные с нуля или модифицированные в рамках данной работы

2.7. Выводы и результаты по главе

Успешно удалось объединить два совершенно разных инструмента в единый пайплайн, модифицировать Редуцерон для возможности работы с разреженными данными, а так же автоматизировать весь процесс с помощью Python скриптов. Весь пайплан целиком изображен на Рис. 1.

3. Эксперименты

Для оценки применимости дистилляции необходимо провести серию экспериментов с различными тестовыми программами с различными входными данными до и после дистилляции, а так же сравнить с выполнение на Редуцероне с исполнением на CPU.

Для экспериментов использовались матрицы из SuiteSparse matrix collection. Матрицы, размер которых не равен степени двойки, были дополнены нулями, а затем сохранены в виде дерева квадрантов. Как было описано выше, язык `.pot` не имеет примитивов, поэтому все представленные матрицы будут либо булевыми, либо с числами 0/1.

Тестовые программы подбирались таким образом, чтобы состоять из набора элементарных операций над разреженных матрицах, для того, чтобы оценить насколько хорошо дистиллятор справляются с fuison оптимизацией. При этом часть операций не использовалась из-за проблем с технической несовершенностью текущей реализации дистиллятора. Например он плохо справляется с умножением матриц, что теоретически можно исправить, но это выходит за рамки данной работы.

Список всех используемых программ:

- `addAdd` – последовательное сложение трех матриц
- `mapAdd` – сложение двух матриц и отображение в натуральные числа
- `mapKron` – произведение Кронекера двух матриц и отображение в натуральные числа
- `maskAdd` – сложение двух матриц и взятие маски

3.1. Потактовая симуляция ПЛИС

Для проведения тестирования был выбран чип `xc7a200tfbg676-3`, так как давал большой запас по ресурсам относительно того, на что был рассчитан Редуцерон. Это во-первых позволяло во время работы

экспериментировать с архитектурой Редуцерона, а во-вторых делает именно Редуцерон узким местом текущей реализации.

Чтобы вычислить производительность Редуцерона на данном чипе, были выполненные шаги синтеза и имплементации, при различных частотах устройства. Оптимальное значение оказалось равно 14 наносекундам на один такт, то есть примерно 71 МГц.

К сожалению большинство программ не соответствуют ограничениям Редуцерона, поэтому во всех экспериментах с симуляцией будет использована только 3 программы с небольшими матрицами 64x64. Для каждой программы запускалось как минимум 3 пары тестов с разными набором параметров.

Для получения количества тактов необходимых для выполнения тестовых программ запускалась потактовая симуляция поведения логической схемы, до завершения вычислений. Также Редуцерон умеет выводит текущий размер кучи на отдельные пины, что позволяет также посчитать фактический расход памяти во время симуляции.

В таблице **??** продемонстрированное уменьшение необходимого числа тактов и памяти после дистилляции. Видно, что дистилляция дает стабильный прирост производительности, как по памяти так и по скорости вплоть до 1.7 раз.

	Соотношение	
Программа	ticks	heap
addAdd	1.42	1.69
mapAdd	1.45	1.46
mapKron	1.76	1.72

Таблица 1: Ускорение при дистилляции. Показывает усредненное соотношение $\frac{V_{not\_distilled}}{V_{distilled}}$

3.2. Эмуляция

В отличие от симуляции эмулятор Редуцерона не строит полной логической схемы, а лишь реализует все этапы редукции, которые происходят в Редуцероне, на `C`. Из-за этого статистика собираемая эмуля-

тором может быть неточной и отличаться от реального выполнения. Поэтому первой подзадачей стала оценка погрешности эмулятора. Для этого все измерения, проведенные в симуляции, были повторены с помощью эмулятора и посчитали относительную погрешность по формуле $\frac{|V_{sim}-V_{emu}|}{V_{sim}}$. В таблице 2 можно проноблюдать, что хоть результаты эмулятора не полностью совпадают с симуляцией, но имеют низкую относительную погрешность в оценки тактов $< 1\%$ и $< 5\%$ в оценки памяти.

	Относительная погрешность	
Программа	ticks	heap
addAdd	0.002	0.043
mapAdd	0.001	0.029
mapKron	0.008	0.0003

Таблица 2: Относительная погрешность измерений на эмуляторе. Показывает усредненное значение $\frac{|V_{sim}-V_{emu}|}{V_{sim}}$

Так как на эмуляторе возможно легко изменять параметры Редуцерона, то на нем можно посчитать увеличенную выборку, добавить матрицы большего размера, а также посчитать статистику по количеству сопоставлений с шаблоном в ходе выполнения (`cases`). В таблице 3 продемонстрировано ускорение, даваемое дистиллятором. Видно, что дистилляция во всех случаях заметное улучшение по всем показателям. Особенно интересно уменьшение количества сопоставлений с шаблоном во время выполнения. Это свидетельствует об уменьшении количества промежуточных структур, то есть дистиллятор действительно выполнят fusion.

	64x64			128x128			256x256		
	ticks	heap	cases	ticks	heap	cases	ticks	heap	cases
addAdd	1.55	1.61	1.13	1.47	1.48	1.11	1.44	1.43	1.12
mapAdd	1.56	1.41	1.42	1.47	1.20	1.53	1.58	1.35	1.49
mapKron	1.76	1.71	1.68	1.74	1.72	1.67	1.77	1.71	1.70
maskAdd	1.42	1.34	1.46	1.40	1.29	1.46	1.31	1.19	1.45

Таблица 3: Ускорение при дистилляции посчитанное на эмуляторе

	Haskell	Haskell (дист.)	Редуцерон	Редуцерон (дист.)
размерность	время (us)	время (us)	время (us)	время (us)
64	29	20	461	295
128	94	79	1176	782
256	123	103	2329	1557

Таблица 4: addAdd

	Haskell	Haskell (дист.)	Редуцерон	Редуцерон (дист.)
размерность	время (us)	время (us)	время (us)	время (us)
64	45	37	387	248
128	162	105	1437	901
256	312	216	1782	1186

Таблица 5: mapAdd

	Haskell	Haskell (дист.)	Редуцерон	Редуцерон (дист.)
размерность	время (us)	время (us)	время (us)	время (us)
64	64	36	486	276
128	137	68	1006	575
256	364	126	2928	1638

Таблица 6: mapKron

	Haskell	Haskell (дист.)	Редуцерон	Редуцерон (дист.)
размерность	время (us)	время (us)	время (us)	время (us)
64	10	7	234	164
128	38	30	502	351
256	48	42	1038	782

Таблица 7: maskAdd

3.3. Сравнение с ghc

Для сравнения исполнения на Редуцерона с исполнением на CPU был выбран Haskell с компилятором `ghc`. Данные о средней скорости выполнения программ на Haskell взяты из другой работы[9], но так как датасеты совпадают, их можно переиспользовать. Для вычисления времени работы на Редуцероне количество тактов делилось на частоту с которой он может работать на данном чипе.

Таблицы 4-7, показывают ожидаемый проигрыш по скорости в абсолютном времени, так как Современные процессоры работают на гораздо большей частоте, чем Редуцерон. Но как было показано выше, дистилляция значительно уменьшает количество обращений к памяти, что дает потенциал к заметному ускорению на огромных матрицах при добавлении поддержки внешней памяти.

[9] `https://github.com/sedwards-lab/fhw/tree/sparse-linear-algebra-distillation/examples/QTreeBenchmarks/diploma/verilog-bool-no-nnz-inlined`

3.4. Выводы и результаты по главе

Были успешно проведены эксперименты с помощью потактовой симуляции ПЛИС, а так же показанно что эмулятор Редуцерона может с большой точностью собирать расширенную статистику. Собранные с помощью эмулятора данные подтверждают гипотезу о том, что дистилляция может быть эффективной автоматической оптимизацией для операций разреженной алгебры, в том числе, потому что способна проводить fusion оптимизации и уменьшать количество взаимодействий с памятью. В конце было показано, что пока данный метод далек от практического применения, но имеет потенциал дальнейшего развития.

Заключение

В ходе работы следующие задачи были выполнены:

- Дистиллятор и Редуцерон были успешно объединены в единый пайплайн выполнения процедур разреженной линейной алгебры. Для этого были написаны новые модули для Дистиллятора, расширенны возможности Редуцерона, а также написаны код для автоматизации проведений экспериментов и анализа результатов.

- Успешно проведены как эксперименты с использованием потактовой симуляцией ПЛИС, так и с использованием упрошенной эмуляции. Для экспериментов использовались различные программы представляющие собой комбинации функций над матрицами с различными входными данными.

- Дистилляция показала выдающиеся результаты не только по ускорению выполнения тестовых программ, но и значительно уменьшила количество обращений к памяти. Было продемонстрировано, что исполнение на Редуцероне уступает в скорости обычному исполнению на CPU, но имеет потенциал превзойти стандартные решения при работе с внешней памятью.

Список литературы

[1] Gupta Udit, Wu Carole-Jean, Wang Xiaodong et al. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. — 2020. — 1906.03109.

[2] He Kaiming, Zhang Xiangyu, Ren Shaoqing, Sun Jian. Deep residual learning for image recognition // Proceedings of the IEEE conference on computer vision and pattern recognition. — 2016. — P. 770–778.

[3] Brin Sergey, Page Lawrence. The anatomy of a large-scale hypertextual Web search engine // Computer Networks and ISDN Systems. — 1998. — Vol. 30, no. 1. — P. 107 – 117. — Proceedings of the Seventh International World Wide Web Conference. URL: `http://www.sciencedirect.com/science/article/pii/S016975529800110X`.

[4] Leskovec Jure, Sosič Rok. Snap: A general-purpose network analysis and graph-mining library // ACM Transactions on Intelligent Systems and Technology (TIST). — 2016. — Vol. 8, no. 1. — P. 1–20.

[5] Wen Wei, Wu Chunpeng, Wang Yandan et al. Learning structured sparsity in deep neural networks // Advances in neural information processing systems. — 2016. — Vol. 29.

[6] Ching Avery, Edunov Sergey, Kabiljo Maja et al. One trillion edges: Graph processing at facebook-scale // Proceedings of the VLDB Endowment. — 2015. — Vol. 8, no. 12. — P. 1804–1815.

[7] Selvitopi Oguz, Ekanayake Saliya, Guidi Giulia et al. Distributed Many-to-Many Protein Sequence Alignment using Sparse Matrices. — 2020. — 2009.14467.

[8] Zhang Zhekai, Wang Hanrui, Han Song, Dally William J. SpArch: Efficient architecture for sparse matrix multiplication // 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) / IEEE. — 2020. — P. 261–274.

[9] Song William S., Gleyzer Vitaliy, Lomakin Alexei, Kepner Jeremy. Novel graph processor architecture, prototype system, and results // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — 2016. — Sep. — URL: http://dx.doi.org/10.1109/HPEC.2016.7761635.

[10] Cass S. Taking AI to the edge: Google's TPU now comes in a maker-friendly package // IEEE Spectrum. — 2019. — Vol. 56, no. 5. — P. 16–17.

[11] Chabchoub Souhir, Mansouri Sofienne, Salah Ridha Ben. Biomedical monitoring system using LabVIEW FPGA // 2015 World Congress on Information Technology and Computer Applications (WCITCA) / IEEE. — 2015. — P. 1–5.

[12] Redgrave Jason, Meixner Albert, Goulding-Hotta Nathan et al. Pixel Visual Core: Google's Fully Programmable Image Vision and AI Processor For Mobile Devices // Proc. IEEE Hot Chips Symp.(HCS). — 2018. — P. 1–18.

[13] He Xin, Pal Subhankar, Amarnath Aporva et al. Sparse-TPU: Adapting systolic arrays for sparse matrices // Proceedings of the 34th ACM International Conference on Supercomputing. — 2020. — P. 1–12.

[14] Kanellopoulos Konstantinos, Vijaykumar Nandita, Giannoula Christina et al. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations // Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture. — 2019. — P. 600–614.

[15] Lin Colin Yu, Wong Ngai, So Hayden Kwok-Hay. Design space exploration for sparse matrix-matrix multiplication on FPGAs // International Journal of Circuit Theory and Applications. — 2013. — Vol. 41, no. 2. — P. 205–219.

[16] Buluc Aydin, Mattson Timothy, McMillan Scott et al. The

GraphBLAS C API Specification. — https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf. — 2019.

[17] Jones Neil D., Gomard Carsten K., Sestoft Peter. Partial Evaluation and Automatic Program Generation. — USA : Prentice-Hall, Inc., 1993. — ISBN: 0130202495.

[18] Wadler Philip. Deforestation: transforming programs to eliminate trees // Theoretical Computer Science. — 1990. — Vol. 73, no. 2. — P. 231–248. — URL: https://www.sciencedirect.com/science/article/pii/030439759090147A.

[19] Turchin Valentin F. The Concept of a Supercompiler // ACM Trans. Program. Lang. Syst. — 1986. — Vol. 8, no. 3. — P. 292–325. — URL: https://doi.org/10.1145/5956.5957.

[20] Sørensen Morten Heine. Turchin's Supercompiler Revisited - An operational theory of positive information propagation. — 1996.

[21] Hamilton G. W. Distillation: Extracting the Essence of Programs // Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. — PEPM '07. — New York, NY, USA : Association for Computing Machinery, 2007. — P. 61–70. — URL: https://doi.org/10.1145/1244381.1244391.

[22] Hamilton Geoffrey. The Next 700 Program Transformers // CoRR. — 2021. — Vol. abs/2108.11347. — arXiv : 2108.11347.

[23] Scheevel Mark. NORMA: a graph reduction processor // Proceedings of the 1986 ACM conference on LISP and functional programming. — 1986. — P. 212–219.

[24] Melo Cecil Accetti R. A., Liu Peilin, Ying Rendong. A Platform for Full-Stack Functional Programming // 2020 IEEE International Symposium on Circuits and Systems (ISCAS). — 2020. — P. 1–5.

[25] Burrows Emma. A Combinator Processor.

[26] Naylor Matthew, Runciman Colin. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA // Symposium on Implementation and Application of Functional Languages / Springer. — 2007. — P. 129–146.

[27] Saad Yousef. Iterative methods for sparse linear systems. — SIAM, 2003.

[28] Song William S., Gleyzer Vitaliy, Lomakin Alexei, Kepner Jeremy. Novel Graph Processor Architecture, Prototype System, and Results // CoRR. — 2016. — Vol. abs/1607.06541. — arXiv : 1607.06541.

[29] Davis Timothy A. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss // 2018 IEEE High Performance extreme Computing Conference (HPEC) / IEEE. — 2018. — P. 1–6.

[30] Cailliau Pieter, Davis Tim, Gadepally Vijay et al. RedisGraph GraphBLAS Enabled Graph Database // 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2019. — P. 285–286.

[31] Yang Carl, Buluç Aydın, Owens John D. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU // ACM Transactions on Mathematical Software (TOMS). — 2022. — Vol. 48, no. 1. — P. 1–51.

[32] Toubia Olivier, Hauser John R., Simester Duncan I. Polyhedral Methods for Adaptive Choice-Based Conjoint Analysis // Journal of Marketing Research. — 2004. — Vol. 41, no. 1. — P. 116–131. — https://doi.org/10.1509/jmkr.41.1.116.25082.

[33] Henriksen Troels, Serup Niels GW, Elsman Martin et al. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates // Proceedings of the 38th ACM SIGPLAN

Conference on Programming Language Design and Implementation. — 2017. — P. 556–571.

[34] Proc. of a Workshop on Graph Reduction. — Berlin, Heidelberg : Springer-Verlag, 1987.

[35] Naylor Matthew, Runciman Colin. The reduceron reconfigured // Proceedings of the 15th ACM SIGPLAN international conference on Functional Programming. — 2010. — P. 75–86.

[36] Kepner Jeremy, Gilbert John. Graph Algorithms in the Language of Linear Algebra. — USA : Society for Industrial and Applied Mathematics, 2011. — ISBN: 0898719909.