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

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления

Выпускная квалификационная работа бакалавра

Векторизация ленивых вычислений

Автор:
Студент Б01-008 группы
Егоров Иван Максимович

Научный руководитель:
научная степень
Добров Андрей Дмитриевич

Научный консультант:
научная степень
Черноног Вячеслав Викторович

Москва 2024

Аннотация

Векторизация ленивых вычислений
Егоров Иван Максимович

В данной работе рассмотрены алгоритмы векторизации, представленные в большинстве современных компиляторах, а также условия применимости этих алгоритмов.
Было выявлено, что ленивые вычисления, поддерживаемые стандартами языков C/C++, не могу быть векторизованы современными компиляторами. Вследствие этого был предложен и разработан алгоритм векторизации таких вычислений.
Для реализации оптимизации был выбран набор компиляторов GCC. В качестве результатов и демонстрации ценности проделанной работы приведена статистика работы реальных тестовых данных из пакетов CPUBench и SPEC CPU 2017.

Abstract

Vectorization of lazy evaluation
This paper examines the vectorization algorithms presented in most modern compilers, as well as the conditions for the applicability of these algorithms.
It has been established that lazy evaluations exposed to C/C++ language standards cannot be vectorized by modern compilers. As a result, an algorithm for vectorizing such evaluations was proposed and developed.
To implement the optimization, the GCC compiler set was chosen. Statistics of the performance of real test data from the CPUBench and SPEC CPU 2017 packages are presented as the results of the work performed.

Содержание

1 Введение

В данной главе вводятся базовые понятия, необходимые для постановки задачи.

1.1 Компиляция

Компиляция — это процесс преобразования исходного кода, написанного на языке программирования высокого уровня (например, C, C++, Java), в машинный код или код, исполняемый целевой машиной.

Компиляция включает в себя несколько этапов:

1. *Лексический анализ (лексинг)*: Разбиение исходного кода на лексемы — минимальные синтаксически значимые единицы.

2. *Синтаксический анализ (парсинг)*: Проверка синтаксической правильности кода и его преобразование в дерево синтаксического разбора (AST).

3. *Семантический анализ*: Проверка корректности использования переменных, типов данных и других языковых конструкций.

4. *Генерация промежуточного кода*: Создание кода на промежуточном языке, который проще анализировать и оптимизировать.

5. *Оптимизация*: Улучшение промежуточного кода для повышения производительности и уменьшения размера.

6. *Генерация машинного кода*: Преобразование промежуточного кода в машинный код.

7. *Линковка (связывание)*: Объединение скомпилированных файлов и библиотек в один исполняемый файл.

Компилятор — это программа, которая выполняет процесс компиляции. Она принимает исходный код и производит на его основе машинный код. Компиляторы могут быть разными для разных языков программирования и целевых платформ. Например, GCC (GNU Compiler Collection) — это компилятор для языков C, C++, и Fortran.

В данной работе основное внимание будет уделено пункту 5, а имменно разработке алгоритма оптимизации на базе компилятора GCC. GCC является современным оптимизирующим компилятором, который широко используется в различных областях разработки программного обеспечения.

Основные виды компиляторов:

1. *Однопроходный компилятор*: Выполняет компиляцию за один проход, что обычно быстрее, но может ограничивать возможности оптимизации.

2. *Многопроходный компилятор*: Выполняет несколько проходов по исходному коду, позволяя более тщательно анализировать и оптимизировать код.

3. *Кросс-компилятор*: Компилирует код на одной платформе для выполнения на другой.

4. *Just-in-time (JIT) компилятор*: Компилирует код во время выполнения программы, улучшая производительность за счёт оптимизаций, которые невозможны при статической компиляции.

Компиляторы играют важную роль в процессе разработки программного обеспечения, позволяя программистам писать код на высокоуровневых языках и затем преобразовывать его в эффективный исполняемый машинный код. Разработка оптимизаций для повышения производительности исполнения кода является основополагающей задачей в современных многопроходных компиляторах.

1.2 Промежуточное представление программы

Промежуточное представление (IR - Intermediate Representation) — это форма представления программы, используемая в компиляторах между этапами анализа исходного кода и генерации машинного кода. Оно играет важную роль в компиляции и позволяет упростить и улучшить процесс преобразования исходного кода в машинный код. Промежуточный код упрощает выполнение различных оптимизаций, поскольку он обычно более абстрактен, чем исходный код или машинный код. На рис. 1 приводится структура компилятора по Aho и др. 2006. TODO: ссылка

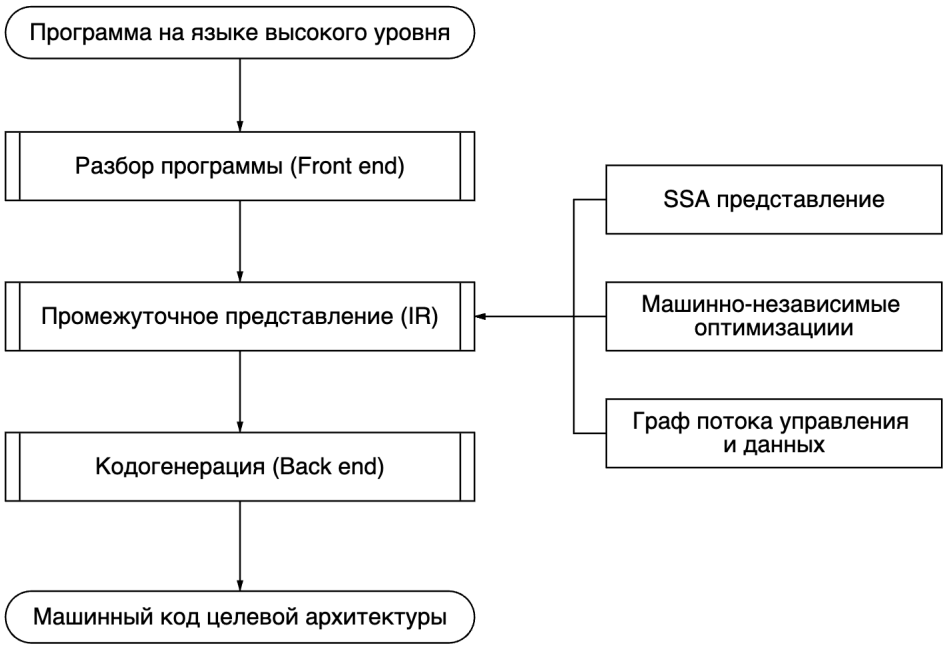

Рис. 1: Структура компилятора

В GCC используются несколько видов промежуточных представлений, среди которых ключевыми являются GIMPLE и RTL:

GIMPLE — это высокоуровневое промежуточное представление, используемое в GCC. Оно является упрощенной версией исходного кода, предназначенной для анализа и оптимизации.

RTL — это более низкоуровневое промежуточное представление, используемое в GCC для представления программы ближе к машинному коду.

GIMPLE используется в основном для машинно-независимых оптимизаций на более высоком уровне абстракции, таких как межпроцедурные оптимизации и преобразования графов потока управления и данных (более подробно в главе TODO: номер). RTL используется для оптимизаций, специфичных для целевой архитектуры, таких как распределение регистров и генерация инструкций. В данной работе для реализации алгоритма векторизации ленивых вычислений будет использоваться первый вид промежуточного представления.

Основные характеристики GIMPLE:

1. *Трехадресный код*: GIMPLE представляется в виде трехадресного кода, где каждая инструкция имеет не более трех операндов.

2. *Упрощение сложных выражений*: Сложные выражения разбиваются на более простые, что упрощает последующую обработку.

3. *SSA-форма (Static Single Assignment)*: GIMPLE преобразуется в SSA-форму, в которой каждая переменная присваивается единожды.(более подробно в главе TODO: номер)

TODO: Убого выглядит
Пример. Исходный код на C:

```
int sum(int a, int b) {
    return a + b;
}
```

GIMPLE представление:

```
D.2098 = a + b;
return D.2098;
```

Введем понятия, относящиеся к промежуточному представлению программы.

1.2.1 Трехадресный код

Трехадресный код (Three-Address Code, TAC) и его реализация в виде GIMPLE в компиляторе GCC представляет собой промежуточное представление, где каждая инструкция имеет не более трех операндов. Трехадресный код состоит из инструкций, каждая из которых может быть представлена в следующем общем виде:

$$(res, op, x, y) \tag{1}$$

где

- op - код оператора(операции)
- x и y - аргументы, подающиеся на вход оператора
- res - результат, вырабатываемый оператором(операцией)

Например *присваивание* в SSA-форме (подробнее в главе TODO: глава) будет иметь следующий вид:

$$res = (op, x, y) \tag{2}$$

Следует учесть, что некоторые операторы могут иметь только один аргумент или не иметь их вовсе, тогда соответствующие элементы оставляются пустыми.

1.2.2 Граф потока управления

Граф потока управления (Control Flow Graph, CFG) является важным инструментом в компиляторах, включая такие, как GCC (GNU Compiler Collection). Он используется для представления возможных путей выполнения программы и помогает оптимизировать и анализировать код.

CFG состоит из следующих основных компонентов:

- *Узлы (Nodes)*: Каждый узел представляет собой базовый блок (basic block) — последовательность инструкций, выполняемых последовательно без переходов.

- *Дуги (Edges)*: Дуги соединяют узлы и обозначают возможные переходы управления между ними. Они могут быть условными (в случае ветвлений) и безусловными (например, вызовы функций или переходы).

Прежде чем дать определелеление базового блока, следует сказать, что операции по их влиянию на ход исполнение программмы относят к двум видам: изменяющие ход исполнения и неизменяющие. Первые это обычно операторы условного или безусловного перехода, а также вызова функций. Главное свойство этих операторов - изменение линейного потока управления программы. Вторые это операторы, результат выполнения которых никак не влияет на линейное исполнение программы.

Базовым блоком назовем совокупность операций, неизменяющих ход исполнения программы со следующими свойствами:

1. Управление базовому блоку может передаваться только через первую инструкцию блока.

2. Управление передается от базового блока только через последнюю инструкцию блока

Соотвественно, передачи управления от одного базового блока к другому — *дугами* графа. На основании анализа CFG компилятор может проводить различные оптимизации, такие как удаление мертвого кода, инлайнинг функций, развертку циклов и другие.

1.2.3 Граф потока данных

Граф потока данных (Data Flow Graph, DFG) — это структурное представление программ, которое используется для анализа потоков данных в компиляторах. В отличие от графа потока управления (CFG), который фокусируется на последовательности выполнения команд, DFG концентрируется на зависимости данных между различными частями программы.

DFG состоит из следующих компонентов:

- *Узлы (Nodes)*: Каждый узел представляет собой операцию или вычисление, выполняемое в программе, например, арифметическую операцию или присваивание.

- *Рёбра (Edges)*: Рёбра обозначают потоки данных между узлами. Они показывают, как результаты одной операции используются в качестве входных данных для другой.

DFG позволяет выявлять независимые операции, которые могут быть выполнены параллельно. Это особенно важно для современных многоядерных процессоров и архитектур с поддержкой параллельных вычислений. Также DFG позволяет проводить так называемый *def-use* анализ, который в дальнейшем будет использоваться в данной работе.

Def-use анализ (анализ определения и использования) — это метод анализа данных, используемый в компиляторах для отслеживания, где в программе значения переменных определяются (присваиваются) и где они затем используются (читаются).

Процесс *def-use* анализа выглядит следующим образом.

1. *Выявление определений и использований*:

   - Определения (def) — это инструкции, которые присваивают значение переменной (например, $x = 5$;)
   - Использования (use) — это инструкции, которые читают значение переменной (например, $y = x + 1$;)

2. *Построение def-use цепочек*:

   Def-use цепочка связывает каждое определение переменной с ее использованными значениями. Например, если переменная x определяется в одной инструкции и используется в нескольких других, создается связь между этими инструкциями.

3. *Анализ потоков управления:*

   Для корректного анализа def-use необходимо учитывать возможные пути выполнения программы. Это обычно делается с помощью графа потока управления (CFG), где для каждого базового блока анализируются определения и использования переменных.

В компиляторах, таких как GCC, оба графа могут использоваться совместно для комплексного анализа программ. На основе CFG можно построить DFG для каждого базового блока, чтобы детально проанализировать потоки данных внутри блока. Информация из DFG может быть использована для улучшения глобальных оптимизаций на уровне CFG, таких как перемещение инвариантов цикла или оптимизация ветвлений.

1.2.4 SSA-представление

Статическое однократное присваивание (Static Single Assignment, SSA) — это форма промежуточного представления программы, используемая в компиляторах для упрощения анализа и оптимизации кода. В SSA каждая переменная присваивается только один раз, а все использованные переменные явно указывают на свое определение. Это достигается путем введения новых переменных при каждом присваивании.

Основные понятия SSA:

- *Присваивание*: Каждое присваивание переменной получает уникальное имя, что устраняет неоднозначность в определении переменных.
- *Функции φ (phi)*: Используются для объединения значений переменных из различных путей потока управления. Эти функции помогают сохранить единственное присваивание переменной, даже если она может принимать разные значения в зависимости от пути выполнения программы.

Рис. 2: Пример SSA-формы

Чтобы перевести код в форму статического единственного присваивания, всем объектам, хранящим в себе результат какой-либо операции присваивается индивидуальный

номер. В точках схождения нескольких определений для одного объекта вводится специальная операция - φ-функция, которая по своей сути объединяет два определения одного объекта в один, так как при реальном исполнении произойдет только одно из определений.

1.2.5 Машинно-независимые оптимизации

(необязательно)

1.2.6 Ленивые вычисления

Ленивые вычисления, также известные как *ленивое вычисление* или *отложенное вычисление(lazy evaluation)*, это техника оптимизации, при которой вычисление выражения откладывается до тех пор, пока его значение действительно не понадобится. В языках программирования C и C++ ленивые вычисления поддерживаются через различные механизмы, стандартизованные самим языком.

```
... some code ...
if (arr[len] != cst1 || arr[len + 1] != cst2) {
  /* do smth.  */
}
... some code ...
```

Рис. 3: Пример ленивых вычислений

В C и C++ логические операторы && (логическое И) и || (логическое ИЛИ) реализованы с использованием механизма *краткого замыкания (Short-Circuit Evaluation)*. Это означает, что вычисление второго операнда происходит только в том случае, если значение первого операнда недостаточно для определения результата выражения. Общее назначение ленивых вычислений заключается в экономии времени на проведении вычислений, результаты которых заведомо не будут использованы программой в дальнейшем. Соответственно, за счет снижения объемов вычислений повышается производительность.

1.2.7 Product-Sum Form (PSF)

Product-Sum Form (PSF) представляет из себя сумму произведений. Каждое произведение содержит в себе последовательность объектов, которые должны быть перемножены, а также постоянный множитель. Помимо произведений, сумма, по аналогии с первыми, имеет константное слагаемое. В общем случае, PSF имеет следующий вид:

$$PSF = c + \sum_i (p_i \cdot \prod_k x_{ik}), \tag{3}$$

где x_{ik} — перемножаемые объекты i-го слагаемого, p_i — его константный множитель, c — константное слагаемое всей формы.

PSF-формы могут быть использованы для проведения оптимизаций над выражениями, а также для анализа зависимостей данных.

1.3 Векторизация

Векторизация — это процесс преобразования обычного (скалярного) кода в код, который может выполняться на векторных или SIMD (Single Instruction, Multiple Data) процессорах. Основная цель векторизации — увеличить производительность программ за счёт параллельного выполнения одной и той же операции над несколькими данными одновременно.

1.3.1 Методы векторизации

Методы векторизации, представленные в современных компиляторах, обычно делятся на циклическую векторизацию и векторизацию линейного кода. Более подробно это будет рассмотрено в 3 главе. Сам же процесс векторизации выглядит примерно следующим образом:

1. *Анализ кода*: Компилятор анализирует исходный код программы для поиска циклов и других конструкций, которые могут быть векторизованы. Это часто означает поиск независимых операций, которые можно выполнять параллельно.

2. *Преобразование кода*: Обнаружив подходящие конструкции, компилятор преобразует их в векторные инструкции. Например, если в цикле происходит сложение элементов двух массивов, компилятор может заменить этот цикл одной или несколькими векторными инструкциями, которые выполняют это сложение над несколькими элементами массивов одновременно.

3. *Генерация векторных инструкций*: Компилятор генерирует соответствующие векторные инструкции для целевой архитектуры процессора. Различные процессоры поддерживают разные наборы векторных инструкций, такие как SSE, AVX для процессоров Intel, NEON для ARM и другие.

1.3.2 Таксономия Флинна

По признаку наличия параллелизма в потоках данных и инструкций Майклом Флинном в 1996 году была предложена следующая классификация:

- **ОКОД(SISD) -** Система с одиночными потоками комманд и данный (single instruction, single data). В данном классе отсутствует какой-либо параллелизм. Такая система исполняет инструкции последовательно, работая только с одним потоком данных.

- **ОКМД(SIMD) -** Система с одиночным потоком комманд и множественным потоком данных (single instruction, multiple data). Машины этого класса загружают набор данных и одну инструкцию для исполнения. Этот класс является особенно важным для данной работу и будет описан детальнее в следующей главе.

- **МКОД(MISD) -** Система с множественным потоком команд и одиночным потоком данных(multiple instruction, single data). Отказоустойчивый тип архитектуры, не получивший широкого распространения.

- **МКМД(MIMD) -** Система с множественными потоками данных и комманд (multiple instruction,multiple data). К этому классу обычно относят многопроцессерные системы и компьютерные кластеры.

1.3.3 Векторизация без использования векторных инструкций

TODO: стоит перенести в 4 часть

В данной работе было решено ограничиться использованием скалярных операций, поскольку векторизация ленивых вычислений накладывает некоторые ограничения в виде потребности векторного сравнения и т.п. Чтобы решить данную проблему было предложено использовать скалярные регистры, данные в которые будут загружаться по секциям с помощью битовых *сдвигов и масок*. Пример такого механизма представлен на рисунке TODO: рисунок

рисунок

В данном случае удается эмулировать параллельное выполнение за счет уменьшения количества обращений в память с использованием обычных регистров. Однако максимальный размер скалярного регистра ограничивает область применения данного подхода и позволяет в основном получать выигрыш засчет загрузки данных, обладающих маленьким размером типа. На рисунке TODO: рисунок представлен пример загрузки 4 элементов массива типа *char* в один элемент размера *int*. Несмотря на подобные ограничение, такой подход все еще остается выигрышным. Тем не менее, в будущем планируется избавиться от данного ограничения путем использования векторного сравнения и улучшения самого алгоритма векторизации.

2 Постановка задачи

Целью данной работы является разработка и реализация алгоритма векторизации линейного кода, содержащего ленивые вычисления

Для достижения данной цели были поставлены следующие задачи:

- Провести анализ частоты встречаемости ленивых вычислений, которые можно было бы векторизовать
- Исследовать подходы к векторизации в современных компиляторах
- Изучить работу компилятора с памятью при векторизации данных и инструкций
- Реализовать поиск подходящих шаблонов, содержащих ленивые вычисления, и отбор потенциальных кандидатов для векторизации
- Реализовать версионирование кода в местах применения трансформации
- Интегрировать оптимизацию в качестве отдельного прохода в компиляторе GCC
- Провести замеры и добиться повышения производительности на реальных тестовых данных

Результат работы можно считать успешным, если удастся показать повышение производительности на реальных тестовых данных.

TODO: решить

Актуальность работы заключается в том, что прежде всего разработка оптимизаций является основополагающей задачей в современных оптимизирующих компиляторах и создание новых алгоритмов векторизации, повышающих производительность, представляет научный и экономический интерес.

или

Актуальность работы состоит в том, что метод векторизации ленивых вычислений, представляемый в работе, позволяет обойти проблему векторизации условных выражений, расширить функционал современных компиляторов и повысить производительность.

или

Оптмизации -> GCC -> ARM -> ARM хайп

3 Обзор существующих решений

В современных компиляторах по типу GCC или LLVM (Low Level Virtual Machine) представлены несколько методов векторизации, которые направлены на оптимизацию выполнения программ за счет использования SIMD (Single Instruction, Multiple Data) инструкций. Поскольку принципиального различия в логике работы векторизаторов LLVM, GCC или других компиляторов нет, то обзор существующих решений можно провести на примере GCC.

3.1 Циклическая векторизация

Циклическая векторизация (Loop Vectorization) в GCC — это процесс преобразования циклов в код, использующий SIMD инструкции, для выполнения нескольких итераций цикла одновременно. Это позволяет значительно улучшить производительность за счет параллельного выполнения операций.

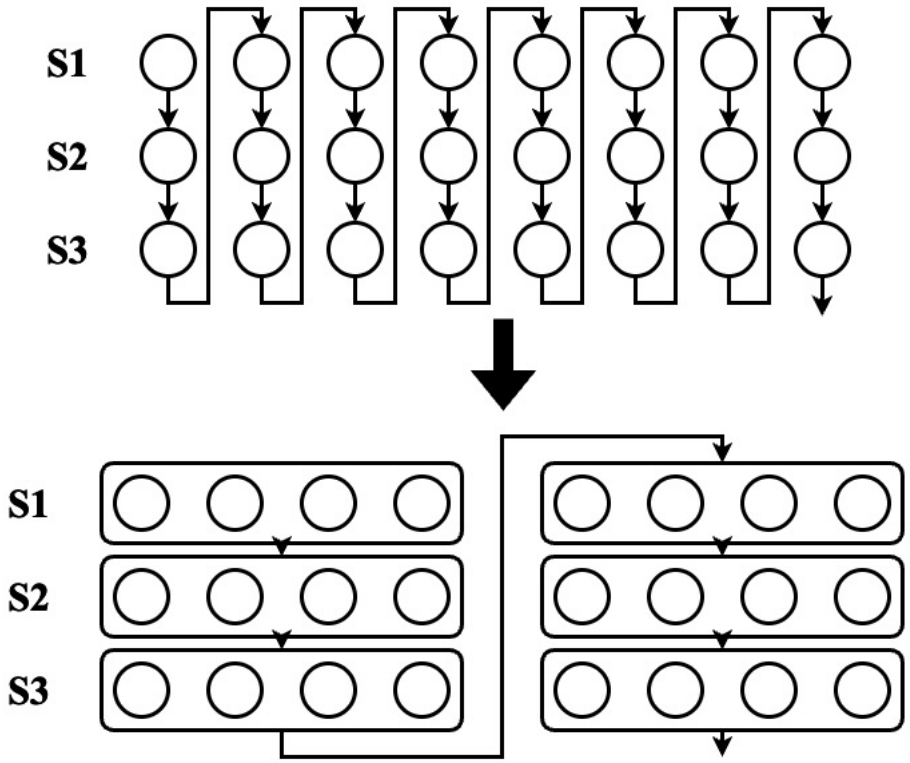

Рис. 4: Циклическая векторизация в GCC

Классический подход к векторизации базируется на теории анализа зависимостей данных. Первый шаг в этом анализе заключается в построении *графа зависимости данных* (Data Dependence Graph, DDG). Анализ начинается с нахождения *сильно связных компонент* (Strongly Connected Components, SCCs). Сильно связные компоненты ориентированного графа представляют собой его максимальные по включению сильно связные подграфы. Ориентированный граф называется *сильно связным*, если любые две его вершины сильно связаны, то есть существует ориентированный путь от первой вершины ко второй и обратно.

SCC, состоящие из одного узла, представляют собой операторы, которые могут выполняться параллельно на данном уровне цикла. SCC, состоящие из нескольких узлов, обозначают операторы, участвующие в цикле зависимости, что может препятствовать

векторизации цикла, если цикл не может быть разорван. Различают три типа зависимостей данных:

- Запись после записи(Write after Write)
- Чтение после записи(Read after Write)
- Запись после чтения(Write after Read)

Пример всех трех зависимостей представлен на рисунке ниже:

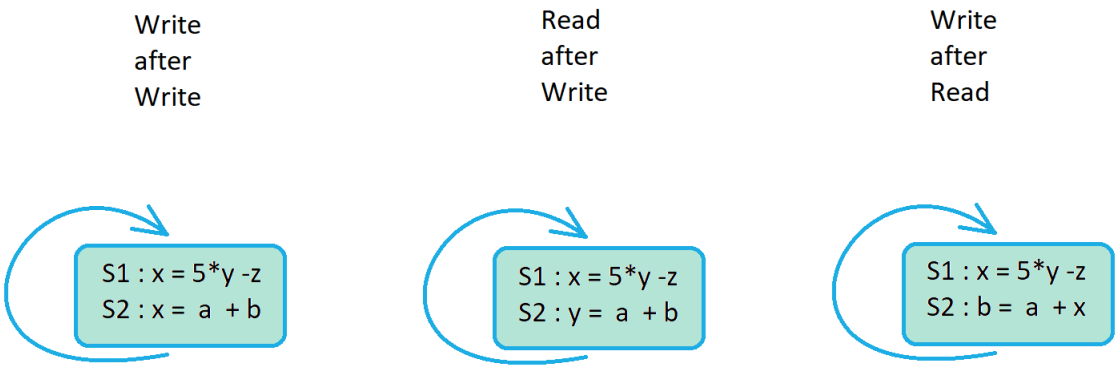

Рис. 5: Пример зависимостей по данным

Помимо этого необходимо отсутсвие сложного потока управления и невекторизуемых типов данных. Если условия векторизации выполнены, компилятор преобразует последовательные операции в векторные инструкции. Например, вместо выполнения сложения для каждого элемента массива последовательно, компилятор может создать векторную операцию, которая выполняет сложение нескольких элементов одновременно.

3.2 SLP векторизация

SLP (Superword Level Parallelism) – это метод векторизации, который фокусируется на объединении отдельных операций, выполняющихся над независимыми данными, в векторные операции. Этот метод необходим для векторизации последовательностей независимых инструкций в линейном коде, которые могут быть выполнены параллельно.

Существующий в GCC алгоритм может обрабатывать линейные участки кода в любом месте программы и, хотя он не ориентирован на циклы, SLP может векторизовать и код внутри циклов, где циклическая векторизация дает сбой. Для применения SLP векторизации в том или ином месте должен быть выполнен ряд условий, главными из которых также являются независимость операций по данным и отсутсвие сложного потока управления. Например, следующая функция (рис. TODO: номер) выполняет очень похожие операции со своими входными данными (a1, b1) и (a2, b2). Линейный векторизатор может объединить их в векторные операции.

```
void foo(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
}
```

Рис. 6: Пример ситуации, подходящей для линейной векторизации

3.3 Векторизация потока управления

В работе *Control Flow Vectorization for ARM NEON* (TODO: ссылка) были рассмотрены некоторые алгоритмы векторизации потока управления, а также был предложен способ векторизации операций загрузок из памяти на основе знания о страничной организации памяти для архитектур, которые не поддерживают *маскированные* операции чтения и записи. Авторы работы, утверждают, что если на архитектуре выполнены следующие свойства,

- аллоцированная память является непрерывной, т.е. массив хранится в непрерывной области памяти без пробелов и дырок
- доступ к памяти может быть выражен в виде линейного аффинного выражения, т.е. адреса строго возрастают или убывают

то код, содержащий операции ветвления, можно безопасно векторизовать, если во время исполнения будет происходить проверка на то, что элементы массива лежат и доступны в памяти. На рисунке TODO: рисунок представлен пример трансформации кода, позволяющий векторизовать блок кода, содержащий ветвления.

здесь рисунок

Стоит обратить внимание, что здесь авторы используют идею *маскированных* загрузок из памяти, где в качестве маски выступает массив $cond[i]$, содержащийся в условии, который по сути содержит информацию о том, какие значения векторизуемого массива будут использованы. Рисунок TODO: рисунок показывает пример условной операции чтения из памяти, где условное выражение используется для *маскирования* доступа в память.

3.4 Остальные методы векторизации

Для того, чтобы предоставить GCC свободу в выборе и применении векторизации существует автовекторизация, которая позволяет компилятору самому выполнить векторизацию тех циклов, которые выгодно векторизовать и применить SLP там, где это необходимо.

Помимо всего этого, пользователю доступен некоторый функционал для *ручной векторизации*. GCC поддерживает использование встроенных функций (intrinsics) для конкретных SIMD наборов инструкций, таких как SSE, AVX и других. Программист может явно использовать эти функции для управления векторизацией.

В GCC присутствуют директивы *#pragma*, которые могут быть использованы для указания компилятору векторизовать определенные участки кода. Например, *#pragma GCC ivdep* указывает компилятору игнорировать некоторые зависимости и векторизовать цикл. Также GCC поддерживает OpenMP. OpenMP (Open Multi-Processing) — это API, который позволяет реализовать многопоточность в программах на C, C++ и

Fortran, предоставляя набор директив, функций и переменных среды для управления потоками.

3.5 Проблема существующих решений

Выше обсуждалось, что для применения векторизации в том или ином месте необходимо удовлетворить некоторому набору условий, одним из которых является *сложный поток управления*. Сложным потоком управления можно считать последовательность выполнения операций или инструкций в программе, когда эта последовательность имеет множество ветвлений, циклов, вызовов подпрограмм и параллельных процессов. Такой поток управления может быть сложен для понимания и анализа, потому что в нем часто присутствуют следующие элементы:

- *Условные операторы*: Ветвления, такие как if, else, switch, которые приводят к выполнению различных блоков кода в зависимости от условий.
- *Циклы*: Конструкции, такие как for, while, do-while, которые повторяют выполнение блока кода до тех пор, пока выполняется определенное условие.
- *Рекурсия*: Функции, которые вызывают сами себя, что может усложнить анализ последовательности выполнения кода.
- *Обработчики исключений*: Конструкции, такие как try, catch, которые управляют обработкой ошибок и исключений.
- *Асинхронность и параллелизм*: Конструкции для работы с многопоточностью и асинхронным выполнением.
- *Вложенные функции и лямбда-выражения*: Использование функций, определяемых внутри других функций, и анонимных функций.

Ленивые вычисления, обсуждаемые в главе TODO: глава, являются частным случаем сложного потока управления. Различные современные векторизаторы не способны объединять такие операции. Хоть авторы в работе TODO: работа сильно продвинулись в этом направлении, их подход все еще не решает проблему векторизации загрузок из памяти, которые могли бы находится в самом условном выражении, а не внутри блока if-else конструкции. К тому же данный подход накладывает некоторые ограничения на диапазон ситуаций, подходящих для векторизации, в виде наличия массива, выполняющего роль *маскирования*. Тем не менее, их подход к векторизации с помощью знания о страничной организации памяти будет использован в дальнейшем в работе.

Не смотря на очевидные преимущества ленивых вычислений, в некоторых случаях строгое следование этому правилу может препятствовать векторизации инструкций, что в свою очередь могло бы дать прирост производительности. В данной работе рассматривается алгоритм, которые позволяет обойти данное ограничение.

4 Исследование и построение решения задачи

Требуется разбить большую задачу, описанную в постановке, на более мелкие подзадачи. Процесс декомпозиции следует продолжать до тех пор, пока подзадачи не станут достаточно простыми для решения непосредственно. Это может быть достигнуто, например, путем проведения эксперимента, доказательства теоремы или поиска готового решения.

мб рассказать про проблему с памятью в ленивых вычислениях и предложить метод ее решения

5 Описание практической части

5.1 Построение шаблона

Для выполнения поставленных задач, прежде всего требуется оценить насколько часто конструкции, содержащие в себе ленивые вычисления, могут встречаться в коде. Для этого необходимо выбрать шаблон, по которому можно определить ленивые вычисления, требуемый для поиска потенциальных кандидатов для векторизации. Следует также определить стартовую точку для построения этого шаблона и реализовать отбор среди кандидатов на предмет наличия тех или иных зависимостей.

Определимся каким условиям должен удовлетворять шаблон, требуемый для задачи.

Первым условием является поиск стартовой точки, с которой будет начинаться построение шаблона. *Стартовая точка* - определение начала участка кода, который будет подвергаться векторизации. В работе за стартовые точки векторизации были приняты операции чтения из памяти, поскольку в последующем они будут объединяться в одну операцию. Так как целью работы является векторизация ленивых вычислений, то все инструкции чтения из памяти не подходят. Необходимо найти такие инструкции обращения в память, которые находятся внутри условия if-else конструкции. Пример представлен на рисунке TODO: number.

```
... some code ...
if (arr[len] != cst1 || arr[len + 1] != cst2) {
  /* do smth.  */
}
... some code ...
```

Рис. 7: Пример шаблона ленивых вычислений, требуемый для поиска и векторизацииTODO: добавить 2 и более условия - переделать рисунок

Соотвественно, *вторым условием* для построения шаблона является наличие инструкций ветвления. Поскольку оптимизация проводится на уровне IR, то непосредственный интерес представляют базовые блоки, конечными инструкциями которых являются условные переходы. Ввиду того, что для векторизации необходимо два или более обращений в память и каждое из условий if-else конструкции представлено отдельным базовым блоком из-за специфики самого IR и определения базового блока, то требуется найти цепочку из базовых блоков, связанных между собой условными переходами и удовлетворяющих условию выше.

Приступая к *третьему условию*, необходимо напомнить, что базовые блоки соединены между собой дугами (подробнее в TODO: глава). Базовые блоки из набранной цепочки имеют две выходящих дуги, так как оканчиваются на условные переходы. Одна из дуг каждого базового блока должна быть входящей дугой для следующего, а вторая дуга должна соединять с одним и тем же базовым блоком, который по сути является продолжением потока управления в случае *выполнения (оператор ИЛИ)* или же *невыполения (оператор И)* условия. Пример подходящего шаблона представлен на рис. TODO: рисунок

Самой сложной частью является определение *соседства* операций чтения из памяти. Под *соседством* в данном случае подразумевается, что данные в памяти лежат рядом друг с другом. В случае двух соседних элеметов массива это гарантируется линейной моделью памяти, упоминаемой в пункте 3.3 в работе (TODO: работа). В коде

же данное свойство представлено несколькими обращениями в память, отличающимися лишь смещением на константу, равную размеру типа. Пример такой ситуации представлен так же на рис. TODO: номер.

Таким образом, *четвертым условием* является поиск таких обращений в память, которые являются *соседними*. Поскольку промежуточное представление использует SSA-форму (рассмотрено в главе TODO: глава), то задача поиска *соседних* операций загрузки из памяти представляет собою задачу построения PSF-формы (подробнее в главе TODO: глава). В SSA-форме каждый объект имеет ровно одно *определение*, поэтому на момент сравнения двух обращений в память вовсе непонятно являются ли они *соседними*, так как имеют разные определения. Помимо этого, в инструкции загрузки из памяти зачастую теряется информация о смещении (относительно начала указателя).

Чтобы решить данную проблему, введем понятие терма. *Терм* - это операнд (*использование*) или результат операции (*определение*), не являющийся константой, который удовлетворяет одному из следующих условий:

- Базовый блок, в котором находится *терм*, доминирует любой базовый блок из найденной цепочки
- Инструкция, содержащая *терм*, не является присваиванием в SSA, то есть не удовлетворяет виду (TODO: ссылка на формулу)

Теперь можно провести следующий анализ. Для всех операций чтения из памяти, представленных в найденной цепочке и содержащихся внутри условий if-else конструкции, строится дерево, состоящее из *термов* данной инструкции. Построение дерева происходит рекурсивно. Берется инструкция, для операндов которой находятся их *определения*, далее в этих *определениях* снова берутся операнды, для которых так же находятся *определения* и так пока не будут найдены все термы. Если же некоторые операнды оказываются константой, то они суммируются и сохраняются для последующего анализа. Теперь можно утверждать следующее:

- Если деревья *термов* всех операций чтения из памяти являются одинаковыми, то есть все *термы* совпадают друг с другом, и посчитанные суммы констант для каждого дерева в отдельности образуют числовую последовательность с шагом равным размеру типа обращения в память, то такие обращения можно считать *соседними*.

Рисунок TODO: рисунок дерева и примера гимпла иллюстрирующий поиск вверх иллюстрирует работу вышеописанного алгоритма на примере GIMPLE IR (а) и последующее построение дерева (б).

TODO: чекнуть код че я там еще делал, заменить на курсив соседние и остальную хрень

После завершения построения шаблона было определено его количество встречаемости в коде. Для сбора информации использовались задачи из пакета CPUBench и SPEC CPU 2017. 8088 подобных шаблонов было обнаружено в наборе задач CPUBench и 5680 в SPEC CPU 2017.

5.2 Отбор кандидатов

сайд эффекты из юз аутсайд еще че то

5.3 Версионирование кода

сплит бб условие на одну страницу 4095 и создание версионного бб картинка результата может про ренейм мапу (необязательно)

5.4 Ограничения предлагаемого подхода

6 Заключение

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

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.