Аннотация

Автоматическая векторизация является важной компиляторной техникой для ускорения работы программ. В связи с распространением использования SIMD (Single Instruction Multiple Data) расширений в современных процессорах, автоматическая векторизация стала горячо обсуждаемой темой в исследованиях компиляторных технологий.

Данная работа посвящена рассмотрению механизма автоматической векторизации в компиляторах, изучению аппарата контекстно-свободных (КС-) грамматик, разработке методики детектирования векторизуемого кода на основе КС-достижимости, выявлению и построению грамматик для паттернов интересующих конструкций.

В рамках поставленной задачи были выявлены паттерны невекторизуемых, векторизуемых и потенциально векторизуемых циклов для языка программирования C, для которых были написаны КС-грамматики. Данные грамматики были протестированы на специально разработанных программах.

На основе проведенной работы были сделаны выводы об приложении данного подхода в детектировании векторизуемых циклов, а также в общем о его применимости к статическому анализу. Вместе с тем были выявлены как сильные, так и слабые аспекты данного подхода и определены векторы дальнейшего развития.

Содержание

Введение **3**

1 Постановка задачи **5**

1.1 Актуальность 5

1.2 Цель работы 5

1.3 Формулировка задачи 6

1.4 Этапы решения 6

2 Обзор существующих решений **8**

2.1 Ограничения на векторизуемость 8

2.2 Обзор анализатора векторизуемости GCC 10

3 Исследование и построение решения задачи **13**

3.1 Теоретические основы 13

3.2 Приведение программы к графо-структурированному виду 14

3.3 Трансформация циклов 16

4 Описание практической части **20**

4.1 Разметка 20

4.2 Детектирование 21

4.2.1 Невекторизуемые конструкции 21

4.2.2 Векторизуемые конструкции 23

4.2.3 Потенциально векторизуемые конструкции 28

4.2.4 Бутылочное горлышко 32

Заключение **33**

Список литературы **34**

Введение

Современные высокопроизводительные компьютеры представляют собой мультиархитектурные системы и реализуют несколько уровней параллелизма: уровень процессов (PLP - Process Level Parallelism), уровень потоков выполнения (TLP - Thread Level Parallelism), уровень команд (ILP - Instruction Level Parallelism) и уровень данных (DLP - Data Level Parallelism). В работе будет рассмотрен уровень параллелизма по данным, а именно применение SIMD подхода (Single Instruction, Multiple Data), в котором, в отличие от обычных скалярных инструкций, обрабатываются векторы данных одной инструкцией процессора.

В настоящее время производители уделяют большое внимание развитию технологий процессоров для работы с векторными инструкциями (Intel SSE/AVX, ARM NEON/SVE, RISC-V) [1]. Векторные регистры доступны почти на всех современных процессорах, исключая, быть может, узкоспециализированные области такие, как интернет вещей (IoT).

К тому же данный подход крайне легковесный для выполнения параллельных расчетов, в то время как для использования нескольких потоков или же использования графических карт для вычислений требуются некоторые затраты на подготовку как данных, так и среды исполнения.

Векторизация может дать ощутимый вклад в области 3D графики, а также мультимедийных приложений. Для примера можно привести обработку фотографий, в частности, изменения яркости изображения, где применяется одна и та же операция для трех каналов.

Для того, чтобы использовать векторные возможности процессоров, у программистов имеется несколько основных способов:

- Ассемблерные вставки - наиболее эффективный с точки зрения использования ресурсов подход, но в то же время архитектурно-зависимый, что приводит к возрастанию трудозатрат на поддержание кроссплатформенности оптимизируемого приложения.
- Интринсики - множество типов данных компилятора и встроенных функций, для которых в компиляторах есть специальная особая обработка.
- SIMD-директивы (Fortran, C/C++).

- Языковые расширения (Intel Array Notation, Intel ISPC, Apple Swift SIMD), а также библиотеки (C++17, SIMD Types, Boot.SIMD, SIMD.js)
- Автоматическая векторизация при помощи применения специального оптимизационного прохода компилятора.

Написание, отладка и поддержка параллельных программ с использованием более высоких из списка подходов является трудоемкой задачей и требует определенного опыта от программиста, так как приходится иметь дело с низкоуровневыми нюансами, связанными с затратой значительных временных ресурсов на разработку и отладку.

Из изложенных соображений становится ясно, что большую популярность представляет собой последний подход, а именно возложение ответственности векторизации на компилятор. Однако, каким бы продвинутым ни был компилятор, до сих пор существует множество случаев, когда он бессилен что-либо векторизовать без дополнительных данных или подсказок.

Основное время на выполнения программы зачастую расходуется при итерации в циклических конструкциях. Именно поэтому в данной области активно проводится анализ и разработка способов векторизации. Одно из последних исследований [2] показало, что процент векторизованных циклов на искусственом тесте ETSVC (Extended Test Suite for Vectorizing Compiler) [3] для таких компиляторов, как ICC, GCC и LLVM составил 64,9%, 43,7% и 32,4% соответственно.

Несмотря на давность, другое исследование [4] дает понять, что на реальных примерах ситуация с векторизацией циклов обстоит еще хуже. Результаты были получены при анализе векторизации мультимедийных приложений, таких как кодеры и декодеры. Процент векторизованных циклов составил лишь 18%-30%.

В связи с этим поставлен вопрос о разработке иного подхода к детектированию векторизуемых циклов, а также потенциально векторизуемых циклов, которые игнорируются компиляторами, но после небольших изменений со стороны программиста могли бы быть векторизованны.

Для решения поставленной задачи будет рассмотрен аппарат КС-грамматик и его применение для выявления векторизуемых конструкций. Будут представлены определенные паттерны кода, выявленные в ходе работы, для поиска невекторизуемых, векторизуемых и потенциально векторизуемых циклов, а также приведены результаты тестирования на разработанных примерах.

1. Постановка задачи

1.1. Актуальность

С учетом развивающихся векторных возможностей процессоров потребность в генерации векторизованного кода компилятором как никогда высока. На данный момент как синтетические тесты, так и тесты на реальных программах показывают, что компиляторы не достигли предела возможностей в данной области. Связано это с консервативностью компиляторов, а также с существованием большого списка ограничений, накладываемых на циклы для их успешной векторизации.

Для проверки соблюдения всех условий проводится обширный анализ, состоящий из множества этапов, который не может привести к 100% векторизации. Про последовательность действий одного из компиляторов (GCC) будет описано далее в работе, в блоке "Обзор существующих решений".

Более того, для использования данного механизма необходима полная компиляция программы, что занимает немалое количество времени. Именно поэтому данная задача вынесена в статический анализ для ускорения, а также для реализации более удобного и продвинутого анализатора векторизуемости.

Исследование [4] выяснило, что при некоторой помощи со стороны программиста можно добится более выскоих результатов в получении оптимизированного кода. Однако на поиск потенциально векторизуемых циклов может уйти большое количество времени.

Применение иного подхода, состоящего в поиске паттернов на графо-структурированных промежуточных представлениях программы теоретически может помочь выявить большее количество циклов, которые могут быть векторизованы.

1.2. Цель работы

Цель данной работы заключается в разработке и проверке методики детектирования циклов, попадающих под одну из следующих категорий:

- Невекторизуемые циклы.
- Векторизуемые циклы (детектируются компиляторами).
- Потенциально векторизуемые циклы (игнорируются компиляторами при оптимизации, но при небольшом рефакторинге могут быть сведены к векторизуемым).

Основой будет служить решение задачи КС-достижимости на графо-структурированных промежуточных представлениях программ.

1.3. Формулировка задачи

В рамках определенной цели были поставлены следующие задачи:

- Разработать методику детектирования невекторизуемых, векторизуемых и потенциально векторизуемых циклов на основе КС-достижимости;
- Составить КС-грамматики для найденных и разработанных паттернов;
- Протестировать составленные грамматики на специально разработанных программах для проверки корректности, а также для выявления сильных и слабых сторон данного подхода.

В качестве рассматриваемого языка программирования был взят C. Дело в том, что C - один из самых распространенных, хорошо стандартизированных, а также синтаксически простых языков.

1.4. Этапы решения

Теоретическая часть

1. Обзор эффективности существующих подходов в компиляторах.

   Выявить слабые и сильные стороны компиляторов в области векторизации.

2. Рассмотрение различных техник оптимизаций и трансформаций циклов.

   Довольно часто компилятор не в состоянии векторизовать циклы в связи со своей консервативностью в трансформациях исходного кода на сложных входных конструкциях. Поэтому небольшой рефакторинг кода со стороны программиста может потенциально повысить шансы векторизации кода.

3. Изучение элементов теории формальных языков.

   Освоить методы применения КС-грамматик в статическом анализе и научиться применять данные техники в детектировании заданных сентенций.

4. Поиск и выявление паттернов для циклов.

   Выявить конструкции циклов, которые однозначно не векторизуемы, могут быть векторизованы, а также потенциально векторизуемы.

Практическая часть

1. Разметка вершин.

   Присвоить каждой вершине графа потока управления программы метку для последующей разметки ребер.

2. Разметка ребер.

   Разметить ребра графа терминалами, исходя из меток инцидентных ему вершин.

3. Разработка грамматик для циклов.

   Составить КС-грамматики в ослабленной форме Хомского, для детекции циклов.

4. Разработка программ для тестирования.

   Написать программы, содержащие как векторизуемые , так и невекторизуемые циклы для определения корректности разработанных грамматик.

5. Тестирование выявленных грамматик.

   Рассмотреть применимость подхода на разработанных примерах, и в целом убедиться в работоспособности метода. Обнаружить слабые стороны данного подхода и пути дальнейшего развития метода.

2. Обзор существующих решений

2.1. Ограничения на векторизуемость

В области детектирования векторизуемых циклов главную роль на данный момент занимают компиляторы, поэтому необходимо рассмотреть подходы и техники, используемые в них. Для начала рассмотрим некоторые условия, ограничивающие возможности автоматической векторизации циклов в современных компиляторах:

- В теле цикла не должно быть переходов вида *goto*, не должно быть ветвлений, образованных оператором *switch*. Однако использование масок допустимо, включая конструкции вида $if - then - else$, которые могут быть представлены в масочном виде.

- Количество итераций должно быть известно до начала выполнения цикла. Таким образом, не должно быть зависимых от данных выходов из цикла вида *break*.

- В цикле не должно быть зависимостей, препятствующих векторизации. Например, циклу не должно требоваться вычисленное выражение 2 из первой итерации для того, чтобы вычислить выражение 1 из второй итерации для корректных результатов, как это происходит в примере (Листинг 1). Это позволяет одновременно выполнять последовательные итерации исходного цикла в одной итерации развернутого векторизованного цикла. В тоже время в цикле (Листинг 2) есть зависимость.

```
for (i = 1; i < MAX; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] - a[i-1];
}
```

Листинг 1: Векторизуемая конструкция. a[i-1] всегда вычислено перед использованием.

```
for (i = 1; i < MAX; i++) {
    d[i] = e[i] - a[i-1];
    a[i] = b[i] + c[i];
}
```

Листинг 2: Невекторизуемая конструкция. a[i-1] требуется до того, как будет посчитано.

- Цикл не должен содержать специальных операторов и никаких вызовов функций, если они не являются inline-функциями (вызов функции заменяется телом функции во время компиляции) или же SIMD функциями. Математические интринсики, такие как $sin()$, $log()$, $fmax()$ и другие могут быть использованы в теле цикла, так как они имеют SIMD представление в стандартной библиотеке.
- В цикле не должно быть массивов с различными типоами данных (char, int, float).
- Векторизуется исключительно внутренний цикл, если присутствует вложенность. Исключением может быть, если оригинальный внешний цикл преобразован во внутренний цикл в результате некоторых иных оптимизаций.
- Данные должны быть выровнены в памяти.
- Векторизованный цикл должен быть эффективнее своего предшественника (подсчет различных эвристик).

Выше перечислены не все ограничения (остальные перечислены здесь[5]), проверяемые на этапе компиляции. Консервативность компилятора, а также довольно жесткие требования приводят к тому, что огромное количество циклов остается нетронутыми, в скалярном представлении.

В вышеприведенных статьях по исследованию оценки возможностей векторизации выясняется, что у каждого компилятора имеется свой подход, поэтому в различных компиляторах векторизуются различные циклы. И даже при объединении всех векторизованных циклов в синтетических тестах результат будет далек от 100%.

Кроме прочего, тестовые циклы довольно искусственны, так как предоставляют компилятору гору информации, которая порой недоступна на практике. В одном из исследований [6] была проведена оценка эффективности наиболее ходовых компиляторов, в процессе которой менялась информация, предоставляемая компилятору. В большинстве случаев результаты оказались хуже при недостатке информации, хотя есть и исключения.

Основной сложностью при SIMD подходе является то, что архитектура памяти позволяет поддерживать лишь доступ к последовательным элементам, которые к тому

же должны быть выровнены. SIMD предлагает механизмы для таких ситуаций, но появляются дополнительные накладые расходы для организации их работы. В связи с этим, часто требуются дополнительные трансформации кода для применения векторизации, которые компилятор в силу консервативности попросту не производит.

2.2. Обзор анализатора векторизуемости GCC

Для примера механизма анализа возможностей векторизации циклов рассмотрим вкратце анализатор векторизуемости компилятора GCC [7]. Схематически порядок векторизации представлен на рисунке 1.

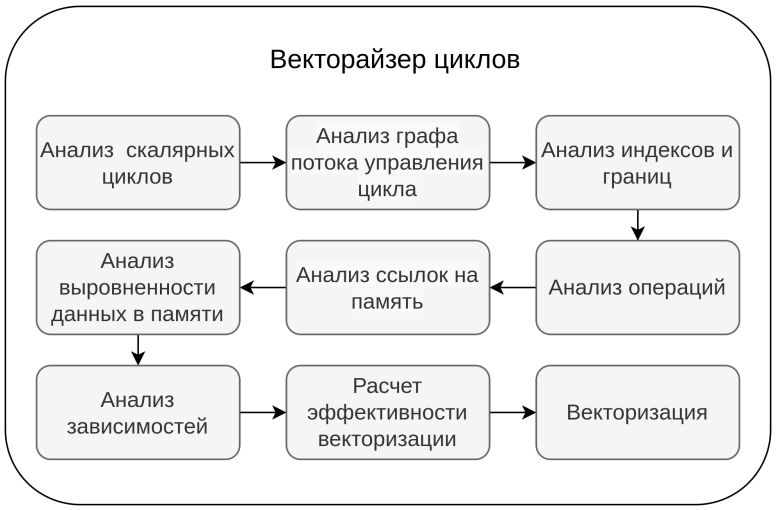

Рис. 1: Схема устройства анализатора векторизуемости GCC.

В ходе анализа производится сбор информации, которая позволяет определить возможность векторизации цикла без изменения семантики программы. Выделяют девять основных этапов (приведены на рисунке 1), семь из которых соответствуют анализу самого цикла, один этап для принятия решения о векторизации на основе оценки эффективности путем подсчета различных эвристик, завершающий этап отвечает за трансформацию скалярного кода в векторизованный.

Рассмотрим вкратце каждую из стадий:

1. **Анализ идиом (idiom recognition)**

   Под идиомами (скалярными циклами) подразумеваются циклы, на входе которых имеется вектор (векторы), а на выходе получается один скаляр. К таким циклам можно отнести поиск элементов в массиве, суммирование и т.п. Для данных

конструкций невозможно применить прямое сопоставление с векторными операциями, а требуется абсолютно иной подход и, как следствие, иной алгоритм.

2. **Анализ графа потока управления цикла (loop CFG analysis)**

   Проверка свойства потока управления цикла: количество базовых блоков, вложенность, единственность входа и выхода. Также рассматривается возможность применения if-conversion для приведения выражений вида if-then-else к масочному виду.

3. **Анализ индексов и границ (loop index and bound analysis)**

   Исследование границ цикла, то есть рассмотрение условия остановки цикла, а также связь индексов при итерировании.

4. **Анализ операций (loop statement analysis)**

   Проверка отсутствия операций, которые не поддерживаются векторными инструкциями, будь то вызов функции, выражения, которые не имеют аналогов в векторном виде и т.д.

5. **Анализ ссылок на память (assess pattern analysis)**

   Архитектура обычно разрешает лишь ограниченный доступ к памяти. Одним из таких ограничений является то, что доступ к памяти должен быть последовательным. Иной порядок обращения к памяти нуждается в дополнительной реорганизации памяти, упаковки элементов в нужный порядок. На этом этапе происходит анализ всех ссылок на память и их классификация по заранее определенным шаблонам доступа.

6. **Анализ выравненности данных по памяти (alignment analysis)**

   Анализ выравненности данных по памяти. Каждое смещение памяти для выравнивания записывается, если существует возможность исправления на этапе компиляции.

7. **Анализ зависимостей (loop carried dependence analysis)**

   Именно данная стадия является основным фокусом при исследовании на векторизуемость. Здесь выясняется, есть ли какие-нибудь зависимости у цикла между итерациями, которые могли бы препятствовать векторизации, а также такие, от

которых можно избавиться после некоторых оптимизаций и превратить код в векторизуемый.

Главной сложностью в этой области являются указатели, которые могут вносить эффекты алиасинга (aliasing). На практике довольно часто указатели задействованы при итерировании по циклам. Так, на данном этапе отдельно строится дерево зависимостей, разрешить которое помогает SSA представление. Зависимости строятся после применения к циклам специальных тестов, которые направлены на анализ использования индексов массивов.

8. **Оценка вероятности векторизации (vectorization probability estimation)**

   По достижении данного этапа, вопрос о возможности векторизации уже решен и исход положительный. Однако требуется исследовать, эффективнее ли векторизованное предстадставление цикла скалярного. Для этого применяются различные эвристики, которые не всегда хорошо учитывают возможности векторизации в силу их некоторого упрощения. Даже если бы была возможность четко рассчитать ускорение, это могло бы занять большое количество времени при компиляции, поэтому прибегают к более общим оценкам, которые зачастую отвергают векторизацию.

9. **Векторизация (vectorization)**

   Преобразование циклов из скалярного представления в векторное со всеми дополнительными трансформациями и оптимизациями, полагаясь на информацию, извлеченную из предыдущих этапов.

Как видно, поиск векторизуемых циклов является комплексной задачей. Компиляторы пока не в силах в полной мере предоставить векторизацию без некоторой помощи со стороны программиста. Более того, в связи с консервативностью, многие циклы, нуждающиеся в предварительной трансформации, игнорируются.

В силу того, что анализ векторизации проводится на этапе компиляции, возникают такие проблемы, как трата ресурсов устройства на построение окружения исполнения и вычислительной инфраструктуры, а также архитектурная зависимость.

Исходя из всех вышеуказанных проблем, возникла идея проводить проверку на векторизуемость циклов при статическом анализе программы на базе решения задач достижимости на графах в КС-ограничениях. Это и будет рассмотрено далее в работе.

3. Исследование и построение решения задачи

3.1. Теоретические основы

В данной части будут приведены понятия, используемые в работе, а также изложены иные теоретические аспекты, представляющие ценность для исследования.

Для ясности и последовательности изложения начнем с базовых определений в теории формальных языков, в основном взятых из [8] и [9].

Определение: Алфавит - конечное непустое множество символов.

Под символами в данном случае можно понимать любой знак, рассматриваемый как нечто неделимое. Алфавиты в работе будут обозначаться буквой Σ.

Определение: Цепочка над алфавитом Σ - произвольная конечная последовательность символов из Σ.

Также в рассмотрение входят и пустые цепочки, не содержащие ни одного символа, и обозначающиеся буквой ε.

Определение: Языком в алфавите Σ называется произвольное множество цепочек конечной длины, составленных из символов алфавита Σ.

Будем обозначать язык над алфавитом буквой $L(\Sigma)$, либо же L, если алфавит понятен из контекста.

Наконец, можно перейти к определению порождающей грамматики.

Определение: Порождающей грамматикой называется четверка $G = (T, N, R, S)$:

- T - алфавит терминальных символов (терминалов);
- N - алфавит нетерминальных символов (нетерминалов), $T \cap N = \varnothing$;
- R - конечное подмножество множества $(T \cup N)^+ \times (T \cup N)^*$, где знаки * и $^+$ означают все цепочки конечной длины, включая и исключая пустую цепочку ε соответственно. Элемент (α, β) множества R называется правилом вывода и записывается в виде $\alpha \to \beta$;
- S - начальный символ грамматики, $S \in N$

Порождающие грамматики - это простой и мощный механизм, позволяющий задавать класс рекурсивно перечислимых языков.

Определение: Сентенцией называется слово, выводимое из цели грамматики.

В работе как раз и предстоит поиск сентенций, составленных из меток ребер.

Грамматики классифицируются по тому, какой вид имеют правила вывода. В данной работе будет использована классификация грамматик по Хомскому, в частности, будет рассматриваться КС-грамматика.

Определение: Грамматика $G = (T, N, R, S)$ называется контекстно-свободной, если каждое правило из P имеет вид $A \to a$, где $A \in N$, a - цепочка любой длины, состоящая из символов $a_i \in (T \cup N)^+$.

Пример:

Язык $L(G) = \{(ac)^n(cb)^n \mid n > 0\}$ можно задать КС-грамматикой с правилами следующего вида:

- $S \to aAb \mid accb$
- $A \to cSc$

Для корректной работы алгоритма поиска путей на графо-структурированном представлении программы, анализатору требуется представление грамматики в нормальной форме Хомского:

Определение: Грамматикой в нормальной форме Хомского называется КС-грамматика, имеющая только правила вывода следующего вида:

- $S \to \varepsilon$
- $A \to BC$
- $A \to a$

где пустая строчка ε не может содержаться в правых частях правил вывода, кроме, быть может, стартового правила.

В силу компактности в работе будут приведены примеры в ослабленной нормальной форме Хомского, в которой допускаются правила вида $A \to \varepsilon$, где $A \in N$. В то же время любую КС-грамматику можно привести к нормальной форме Хомского, алгоритм приведения изложен в [10].

3.2. Приведение программы к графо-структурированному виду

Итак, перейдем непосредственно к задаче. Представим исходную программу в виде графа $GP = (V, E, L)$, где V, E, L - вершины, ребра и метки ребер соответственно, $E \subseteq V \times L \times V$. Такое представление задает язык $L(GP) = \{w | w = (v_0 l_0 v_1, v_1 l_1 v_2), v_i l_j v_k \in E\}$, состоящий из слов, являющихся конкатенацией меток ребер и представляющих собой пути в данном графовом представлении. При таком представлении программы можно проводить поиск паттернов, то есть поиск всех путей в графе, содержащих слова из некоторого рассматриваемого языка $L'(G) \subseteq L(GP)$.

Для того, чтобы привести программу к требуемому для анализа виду, использовался инструмент [11]. Рассмотрим последовательность шагов преобразования:

1. В начале производится частичная компиляция программы для получения промежуточного представления GCC GIMPLE [12] (либо LLVM IR [13]).

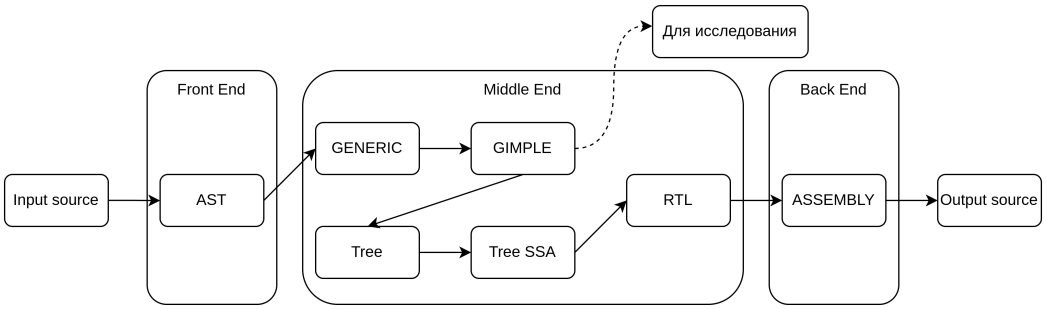

Рис. 2: Схема механизма компиляции GCC.

2. После этого происходит разбиение полученных базовых блоков представления GIMPLE на покомандные блоки для того, чтобы выявлять по их содержимому необходимые паттерны. Также на этом шаге все вызовы функций встраиваются в общий граф программы для возможности проведения межпроцедурного анализа[1].

3. Затем происходит стадия специализации графа, на которой происходят дополнительные преобразования текущего представления в более удобные для анализа форматы. Для примера, добавление в граф ребер потока данных, добавления SSA-представления. На данном этапе не происходит удаления уже существующих вершин и ребер, но лишь их добавление, сохраняется слабая связность. Построение обратных ребер не допустимо.

4. Далее требуется разметить ребра. Для этого происходит полный обход графа по вершинам, содержимое которых анализируется, а затем для каждой присваивается метка из заданного множества. После этого каждому ребру присваивается терминал, который определяется по меткам инцидентных вершин.

5. На конечной стадии готовое представление вместе с грамматикой используется для решения задачи КС-достижимости на графах (CFL-R).

[1]Встраивание вызовов при этом ограничено одной единицей трансляции, а рекурсия строится как циклический путь в графе

Для решения задачи КС-достижимости (CFL-R) в работе используется классический подход (Melski and Reps [14], Y.Hellings [15], CYK-подобный), основанный на динамическом программировании: применяется восходящий анализ, при котором заполняется матрица достижимости от терминалов до стартового символа грамматики. После этого по содержимому матрицы можно восстановить дерево вывода.

Также в последовательности шагов было упомянуто SSA представление, которое позволяет выполнять анализ ссылок на память и применять различные оптимизации к исходному коду. Поэтому следует описать данную концепцию.

Определение: SSA (Static Single Assignment form) - промежуточное представление, используемое компиляторами, в котором каждой переменной значение присваивается лишь единожды.

Такое представление также вносит свои коррективы и появляется неоднозначность в выборе переменной при ветвлении кода. Специально для этого введены PHI-функции, которые создают новую версию переменной, значение которой будет установлено в зависимости от того, по какой ветви было получено управление (рисунок 3).

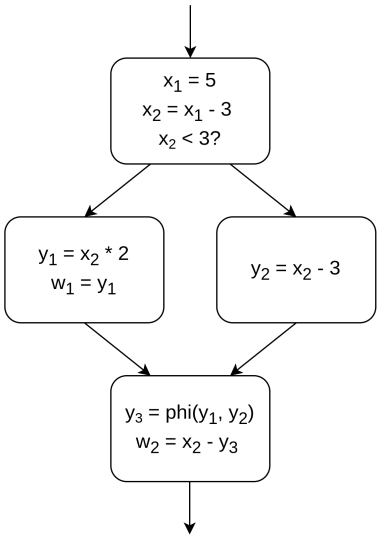

Рис. 3: Принцип работы SSA.

Одним из приложений данного представления программ может быть детекция векторизуемого кода. Именно об этой задаче и пойдет дальнейшее повествование.

3.3. Трансформация циклов

Трансформация циклов играет большую роль при оптимизации программ. Такие трансформации могут приводить как напрямую к ускорению исполнения циклов, так и

косвенно, то есть так, что после данной трансформации к циклу можно будет применить иные оптимизации. Именно последние трансформации довольно часто не применяются при компиляции в связи с консервативностью.

Существует довольно много различных техник оптимизации и трансформации циклов. С основным списком можно ознакомиться например здесь [16]. Я рассмотрю только те из них, которые могут быть полезны в случае векторизации.

Расщепление тела цикла

Расщепление тела цикла является довольно эффективной техникой трансформации циклов. С помощью нее можно избавиться от зависимостей в теле цикла, предотвращающих векторизацию. Рассмотрим использование данной техники (Листинг 3).

```
for (int i = 1; i < n; i++) {
    d[i] = e[i] - a[i-1];
    a[i] = b[i] + c[i];
}
```

Листинг 3: Разрешимая зависимость данных, препятствующая векторизации.

Данный цикл уже приводился в пример, когда оговаривались условия, при которых цикл может быть векторизован. Здесь возникает зависимость вида чтение после записи (RAW - Read-After-Write). Однако в данном примере есть один нюанс, а именно то, что разделения его на два отдельных тела, от зависимости можно избавиться, поэтому данный цикл можно будет векторизовать. После преобразования будет иметь вид (Листинг 4).

```
for (int i = 1; i < n; i++) {
    a[i] = b[i] + c[i];
}

for (int i = 1; i < n; i++) {
    d[i] = e[i] - a[i-1];
}
```

Листинг 4: Разрешение зависимостей предыдущего примера.

Изменение порядка итерирования

С учетом того, что векторизация применяется лишь ко внутреннему циклу при наличии вложенности, изменение порядка итерирования может помочь вынести зави-

симости во внешний цикл так, что векторизация станет доступной. Рассмотрим пример (Листинг 5).

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; ++j) {
        a[i][j+1] = a[i][j] + 5;
    }
}
```

Листинг 5: Зависимость во внутреннем цикле.

В данном случае существует зависимость между итерациями внутреннего цикла, поэтому векторизация невозможна. Однако после преобразования ограничения пропадают (Листинг 6).

```
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; ++i) {
        a[i][j+1] = a[i][j] + 5;
    }
}
```

Листинг 6: Зависимость во внешнем цикле.

Для применения SIMD инструкций требуется последовательный доступ к памяти. Данная техника может изменить порядок доступа к элементам массива и сделать цикл пригодным к векторизации. В основном 2D и большей размерности массивы в $C/C++$ хранятся в памяти построчно. Рассмотрим пример (Листинг 7).

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; ++j) {
        a[j][i] = a[j][i] + 5;
    }
}
```

Листинг 7: Обращение к элементам массива по столбцам.

В данном случае доступ к памяти будет происходить с шагом n, что приведет к игнорированию компилятором данного цикла при векторизации. Однако после преобразования проблемы с доступом к памяти будут разрешены (Листинг 8).

```
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; ++i) {
        a[j][i] = a[j][i] + 5;
    }
}
```

Листинг 8: Обращение к элементам массива по строкам.

Разделение диапазона итерации

Порой в циклах лишь первые или последние несколько итераций могут содержать зависимости и препятствовать векторизации. В данном случае может помочь техника разделения диапазона итерации. Рассмотрим пример (Листинг 9).

```
for (int i = 0; i < n; i++) {
    a[i] = a[i] + a[0]
}
```

Листинг 9: Зависимость от первого элемента.

В данном примере компилятор заметит зависимость цикла от первого элемента и может проигнорировать возможность векторизации. После разделения же, цикл будет векторизован без проблем (Листинг 10).

```
a[0] = a[0] + a[0]
for (int i = 1; i < n; i++) {
    a[i] = a[i] + a[0]
}
```

Листинг 10: Подсчет первого элемента вынесен - зависимости нет.

4. Описание практической части

Перейдем к практической части работы, а именно к рассмотрению выявленных шаблонов и разработанных КС-грамматик для их детекции. Помимо паттернов, которые могут быть векторизваны, в работе также рассматривались антипаттерны, которые определенно останутся в скалярной форме.

4.1. Разметка

Для начала работы требуется разметить вершины. Для этого используется механизм регулярных выражений, который позволяет изъять всю необходимую информацию о текущем узле для того, чтобы затем присвоить ему определенную метку. В таблице 2 представлены метки вершин, используемые в работе:

Таблица 1: Разметка вершин.

Тип	Пример	Семантика
assign_const	D.420 = 0;	Присв. константы
assign_var	_1 = j;	Присв. значения переменной
assign_function_call	res = foo();	Вызов функции с присв.
assign_aryphmetic_op	_1 = i * i;	Арифм. операция с присв.
if_cond	if (value > 0)	Условный оператор
goto	goto <bb 5>; [INV]	Безусловный переход
assigt_struct	a[i_1].x = 5;	Присв. структуре
switch_block	switch (check_12) <...>;	Switch блок
assign_address	a = &a1;	Присв. указателю адреса
assign_modified_address	a = &a1 + 4;	Присв. указателю мод. адреса
assign_phi	i_1 = PHI<...>;	Присв. PHI-функции
return_val	return;	Выражение возврата
_exit	EXIT	exit выражение
_entry	ENTRY	entry выражение
any	a = {v} {CLOBBER};	Любое другое выражение

После того, как всем вершинам присвоены метки, требуется приступить к разметке ребер. Каждое ребро помечается терминалом для будущих грамматик, исходя

из типов инцидентных вершин. В следующей таблице представлены введенные метки ребер:

Таблица 2: Разметка ребер.

Тип	Связь
a	any → assign_function_call
b	any → assign_phi
c	any → if_cond
d	assign_address → assign_modified_address
e	assign_function_call → any
f	assign_modified_address → assign_address
g	assign_phi → any
h	assign_phi → assign_phi
i	assign_phi → if_cond
j	assign_struct → any
k	if_cond → any
l	if_cond → assign_struct
m	return_val → exit
z	any → any

4.2. Детектирование

Собственно, подошли к самой важной части данной работы, а именно к детектированию следующих циклов: невекторизуемые, векторизуемые, потенциально векторизуемые. Далее будет обзор разработанных КС-грамматик, их тестирование на упрощенных примерах, а также визуализация.

4.2.1 Невекторизуемые конструкции

Для начала рассмотрим конструкцию, которая не подлежит векторизации.

Эффект алиасинга

Алиасинг - это эффект, при котором на один и тот же участок памяти ссылаются несколько разных указателей. Уже из определения понятно, что именно указатели вносят порой непреодолимую для разрешения сложность при решении о векторизации. Для разбора подобных случаев требуется проводить анализ указателей (pointer

analysis), который позволяет вычислять при статическом анализе, куда ссылаются указатели и разрешать конфликты алиасинга.

Пример № 1

В данной работе представлен простой пример алиасинга, который удалось выявить. Помимо всего прочего, в этом примере показывается одно из основных преимуществ данного подхода, а именно то, что можно проводить межпроцедурный анализ.

Сентенция: $dz^*aez^*bikz^*b$

Грамматика:

- $S \to AB$
- $A \to d$
- $B \to CD$
- $C \to \varepsilon \mid CC \mid z$
- $D \to EF$
- $E \to a$
- $F \to GH$
- $G \to e$
- $H \to CI$
- $I \to JK$
- $J \to b$
- $K \to LM$
- $L \to i$
- $M \to NO$
- $N \to k$
- $O \to CJ$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 11, а визуализация на рисунке 4.

```
void copyElems64AndAdd1(int* a, int* b) {
    for (long unsigned int i = 0; i < 64; ++i) {
        a[i] = b[i] + 1;
    }
}

void main() {
    int a[512];
    int *a1 = a;
    int *a2 = a + 1;
    copyElems64AndAdd1(a1, a2);
}
```

Листинг 11: Межпроцедурный анализ с алиасингом.

Как видно из кода выше, в данном случае возникает зависимость RAW. Данную зависимость невозможно устранить, поэтому цикл не подлежит векторизации.

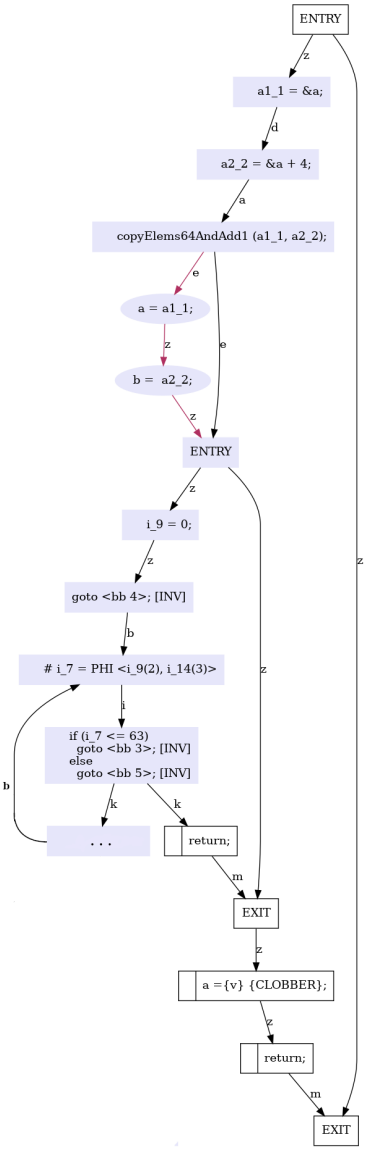

Рис. 4: Визуализация детектирования межпроцедурного алиасинга.

4.2.2 Векторизуемые конструкции

Теперь рассмотрим примеры циклов, которые могут быть векторизованы компилятором.

Эффект алиасинга

В некоторых ситуациях эффект алиасинга может не влиять на векторизацию. Рассмотрим один из таких примеров.

Пример № 2

Сентенция: fz^*bikz^*b

Грамматика:

- $S \to AB$
- $C \to \varepsilon \mid CC \mid z$
- $F \to GH$
- $I \to k$
- $A \to f$
- $D \to EF$
- $G \to i$
- $J \to CE$
- $B \to CD$
- $E \to b$
- $H \to IJ$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 12, а визуализация на рисунке 5.

```
void main() {
    int a[128];
    int *a2 = 1 + a;
    int *a1 = a;
    for (long unsigned int  i = 0; i < 64; ++i) {
        a2[i] = a1[i] + 1;
    }
}
```

Листинг 12: Алиасинг.

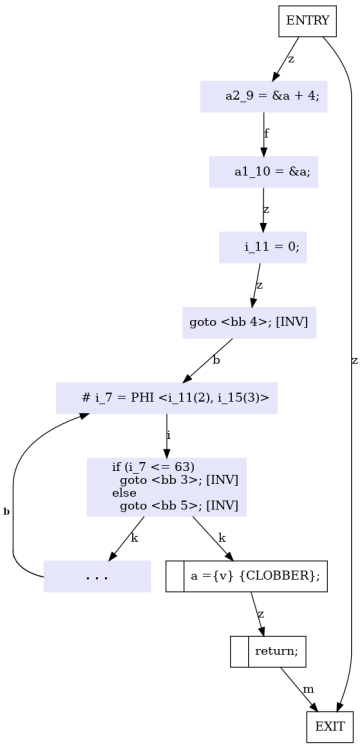

Рис. 5: Визуализация детектирования алиасинга.

Идиомы

Идиомы или же скалярные циклы не могут быть векторизованы напрямую сопо-

ставлением скалярных выражение векторным. О написании векторных аналогов идиом можно почитать здесь [17].

Пример № 3

Скалярных циклов существует немалое количество (суммирование, индукция, поиск элементов и т.п.). Их детекция не составляет сложности. Рассмотрим на примере суммирования.

Сентенция: $kzzbhi$

Грамматика:

- $S \to AB$
- $B \to CD$
- $D \to CE$
- $F \to b$
- $H \to h$
- $A \to k$
- $C \to z$
- $E \to FG$
- $G \to HI$
- $I \to i$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 13, а визуализация на рисунке 6.

```
void main() {
    int a[128];
    int sum = 0;
    for (int i = 0; i < 128; ++i) {
        sum += a[i];
    }
}
```

Листинг 13: Идиоматическая конструкция.

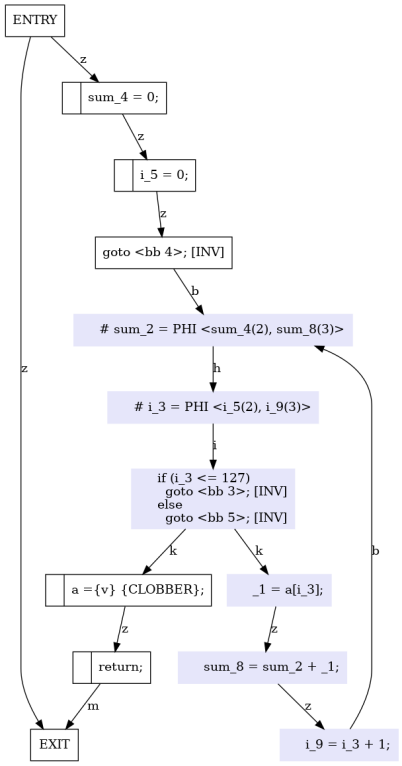

Рис. 6: Визуализация детектирования суммы элементов массива.

Условное ветвление

Компилятор способен выявить небольшое количество циклов, содержащих в себе условные конструкции, в частности, лишь самые простые конструкции. Обычной стратегией компилятора при таком подходе является:

1. представление условия в масочном виде, т.е. создание массива, состоящего из 1 и 0, где значения отвечают выполнению и невыполнению условия соответственно;
2. расчет выражений по обеим ветвям;
3. применение маски к результатам, полученным из двух ветвей.

Пример № 4

Сентенция: kz^*ckz^*b

Грамматика:

- $S \to AB$
- $A \to k$
- $B \to CD$
- $C \to \varepsilon \mid CC \mid z$
- $D \to EF$
- $E \to c$
- $F \to GH$
- $G \to k$
- $H \to CI$
- $I \to b$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 14, а визуализация на рисунке 7.

```
void main() {
    int a[128];
    for (int i = 0; i < 128; i++) {
        if (a[i] > 0) {
            a[i] = 6;
        } else {
            a[i] = 9;
        }
    }
}
```

Листинг 14: Простое ветвление в цикле.

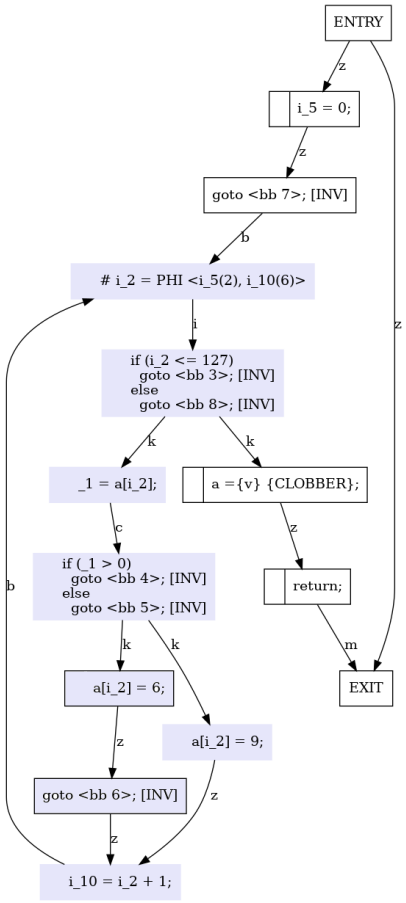

Рис. 7: Визуализация детектирования цикла с ветвлением.

4.2.3 Потенциально векторизуемые конструкции

И наконец перейдем к самым интересным и значительным паттернам в данной работе, а именно к циклам, которые игнорируются компилятором по тем или иным причинам при векторизации, хотя возможность таковая имеется.

Вызов функции в теле цикла

Для начала разберемся, почему компилятор отказывается векторизовать циклы, в которых присутствуют вызовы функций. Если говорить кратко, то дело в смене контекста.

Механизм вызова функций включает в себя взаимодействие с программным стеком. При очередном вызове все регистры процессора очищаются, а передаваемые аргументы кладуться в стэк. Уже из описания этой процедуры понятно, почему компилятор отказывается векторизовать циклы с вызовами функций, в силу больших накладных расходов.

Однако, если функция не такая объемная, то программист мог бы пересмотреть свою архитектуру кода, и удалить ненужный вызов или же сделать функцию inline. Это приведет к разрастанию кода, однако в то же время может сэкономить время, если конструкцию удастся векторизовать. В связи с этим, данный паттерн можно отнести к потенциально векторизуемым.

Пример № 5

Сентенция: $kz^*ez^*mz^*bi$

Грамматика:

- $S \to AB$
- $A \to k$
- $B \to CD$
- $C \to \varepsilon \mid CC \mid z$
- $D \to EF$
- $E \to e$
- $F \to CG$
- $G \to HI$
- $H \to m$
- $I \to CJ$
- $J \to KL$
- $K \to b$
- $L \to i$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 15, а визуализация на рисунке 8.

```
void foo() {}
```

```
void main() {
    int a[128];
    for (int i = 0; i < 128; ++i) {
        foo();
    }
}
```

Листинг 15: Цикл с вызовом функции в теле.

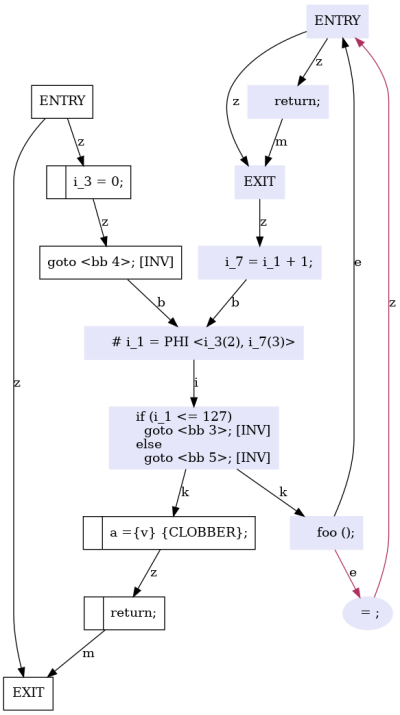

Рис. 8: Визуализация детектирования цикла с вызовом функции в теле.

Неизвестные границы цикла

Хотя разработчики компилятора LLVM утверждают, что циклы с неопределенными границами векторизуются, на практике такие циклы чаще игнорируются. Сложность заключается в том, что для выполнения векторных операций данные загружаются последовательно в специальные регистры процессора группой. Но что если условие невыполнимо для какого-либо из элементов загруженных данных?

Данную проблему можно решить путем рассмотрения выполнения условий заранее, а затем на основе полученных результатов принимать решение о векторизации по части массива, что удовлетворяет условию.

Пример № 6

Сентенция: kz^*bgc

Грамматика:

- $S \to AB$
- $C \to \varepsilon \mid CC \mid z$
- $F \to GH$
- $A \to k$
- $D \to EF$
- $G \to g$
- $B \to CD$
- $E \to b$
- $H \to c$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 16, а визуализация на рисунке 9.

```
void main() {
    int a[128];
    int i = 0;
    while (a[i] > 0) {
        a[i] = 5;
        ++i;
    }
}
```

Листинг 16: Цикл с неизвестными границами.

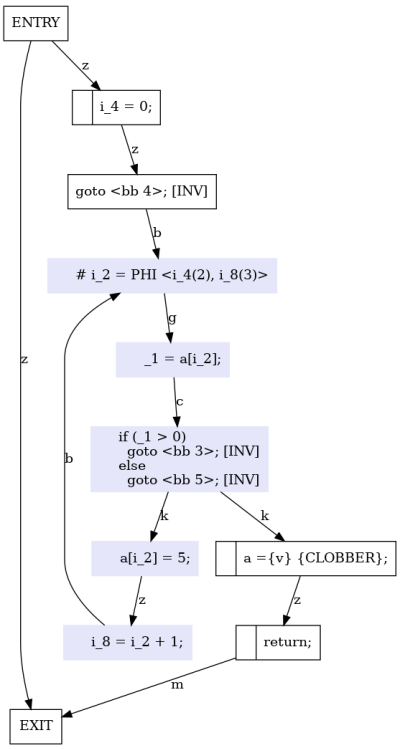

Рис. 9: Визуализация детектирования цикла с неопределенными границами

Работа со структурами

Довольно важная вещь, которая встречается во многих прикладных задачах,

особенно при ООП подходе. Как уже отмечалось ранее, для использования SIMD инструкций компилятору необходим последовательный доступ к элементам. Однако при итерировании по массиву, состоящему из структур, доступ получается не последовательный, а с определенным шагом, зависящим от устройства структуры. Именно поэтому компиляторы не рассматривают данные конструкции при векторизации.

Однако существует решение, которое способно повысить эффективность - требуется перейти от массива структур к структуре массивов. Да, это не всегда представляется возможным, но если воспользоваться данным переходом, векторизация будет выполнена.

Рассмотрим детектирование одного из таких случаев.

Пример № 7

Сентенция: $lgbi$

Грамматика:

- $S \to AB$
- $A \to l$
- $B \to CD$
- $C \to g$
- $D \to EF$
- $E \to b$
- $F \to i$

Сниппет кода для тестирования разработанной грамматики представлен в Листинге 17, а визуализация на рисунке 10.

```
struct point {
    int x, y;
};

void main() {
    struct point arr1[128];
    for (int i = 0; i < 128; i++) {
        arr1[i].x = 5;
    }
}
```

Листинг 17: Цикл с операциями над массивом структур.

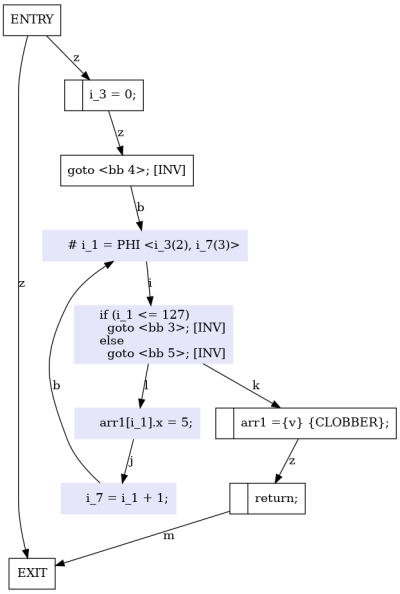

Рис. 10: Визуализация детектирования цикла с операциями над структурой в теле

4.2.4 Бутылочное горлышко

К сожалению, циклы, рассмотренные секции 3.3, в работе задетектировать не удалось. Обычно, Такие циклы отбрасываются компилятором в силу консервативности. Для успешного выявления таких случаев необходимо вводить специальный анализ зависимостей в цикле, что потенциально намного увеличит количество детектируемых векторизуемых и невекторизуемых циклов.

Заключение

В ходе данной работы был произведен обзор существующих решений в области векторизации циклов, рассмотрен метод детектирования невекторизуемых, векторизуемых и потенциально векторизуемых циклов на графо-структурированном представлении программы, а также были разработаны КС-грамматики для их детекции. Данные грамматики были протестированы на специально разработанных программах. Результаты работы выложены на github [18].

Применяемая методика позволяет детектировать циклы, которые могут быть векторизованы, но игнорируемые компилятором в силу различных обстоятельств.

Применяемая методика позволяет детектировать циклы, которые могут быть векторизованы, однако игнорируемые компилятором в силу консервативности.

В ходе дальнейшей исследовательской работы поставлены следующие задачи:

- Добавление анализа указателей (pointer analysis), который может сыграть ключевую роль в определении типов зависимостей на практических примерах.
- Добавление анализа зависимостей по данным в цикле. По большей части набор синтетических тестов ETSVC состоит из циклов, в которых компилятору требуется разрешать различные зависимости и связи. Поэтому добавление данной проверки может значительно увеличить точность детектируемых конструкций.
- Рассмотрение и разработка большего числа паттернов.
- Сравнение разработанного метода с реализациями в современных компиляторах (GCC, Clang). Проверку можно провести на наборе тестов ETSVC, а также на приложениях связанных с мультимедией.
- Реализация обратной аннотации исходного кода, то есть добавление комментариев в код с подсказками программисту, например, о необходимости трансформации цикла. Визуализация на графе даёт представление о результатах детектирования, но сильно усложняет восприятие при переходе к большим программам.

Список литературы

[1] Introduction to Intel® Advanced Vector Extensions. https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf.

[2] *Feng, Jing Ge.* Evaluation of Compilers' Capability of Automatic Vectorization Based on Source Code Analysis / Jing Ge Feng, Ye Ping He, Qiu Ming Tao // *Hindawi.* — 2021. — Vol. 2021.

[3] Extended Test Suite for Vectorizing Compilers. http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz.

[4] An Evaluation of Vectorizing Compilers / Saeed Maleki, Yaoqing Gao, Maria J. Garzaran et al. // *IJERA.* — 2013. — Vol. 3.

[5] A Guide to Vectorization with Intel® C++ Compilers. https://www.intel.com/content/dam/www/public/us/en/documents/guides/compiler-auto-vectorization-guide.pdf.

[6] *Siso, Sergi.* Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information / Sergi Siso, Wes Armour, Jeyarajan Thiyagalingam // *ACM Transactions on Architecture and Code Optimization.* — 2019. — Vol. 16.

[7] GCC подход к автовекторизации. https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[8] *И.А., Волкова.* Формальные грамматики и языки. Элементы теории трансляции / Волкова И.А., Руденко Т.В. — 1999.

[9] *Свердлов, Сергей.* Конструирование компиляторов / Сергей Свердлов. — 2015.

[10] Нормальная форма Хомского. https://en.wikipedia.org/wiki/Chomsky_normal_form.

[11] Статический анализатор, на основе которого строилось решение. https://github.com/nefanov/pathfinder.

[12] GCC Gimple. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[13] LLVM. https://llvm.org/docs/LangRef.html.

[14] *Chaudhuri, Swarat.* Subcubic algorithms for recursive state machines / Swarat Chaudhuri // *ACM SIGPLAN Notices.* — 2008. — Vol. 43.

[15] *Hellings, Jelle.* Conjunctive Context-Free Path Queries / Jelle Hellings // *ICDT.* — 2014.

[16] Loop optimization. `https://en.wikipedia.org/wiki/Loop_optimization`.

[17] Векторизация идиом. `https://www.mkurnosov.net/teaching/docs/ddp-simd-lec3.pdf`.

[18] Результаты исследования. `https://github.com/zigal0/pathfinder`.

[19] *Moldovanova, Olga.* Automatic SIMD Vectorization of Loops: Issues, Energy Efficiency and Performance on Intel Processors / Olga Moldovanova, Mikhail Kurnosov // *Communications in Computer and Information Science.* — 2017.

[20] *Naishlos, Dorit.* Autovectorization in GCC / Dorit Naishlos // *GCC Developers' Summit.* — 2004.

[21] Compilers: Principles, Techniques, and Tools / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. — 2006.