Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление подготовки: 09.03.01 – “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

К защите допустить:
Заведующий кафедрой
д. т. н., профессор_____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: Анализ эффективности векторизации кода в современных компиляторах

Студент _____ А. Н. Багиев

Руководитель
к. т. н., доцент _____ А. А. Пазников

Консультант к.э.н. _____ О. С. Артамонова

Консультант от кафедры
к. т. н., доцент, с. н. с. _____ И. С. Зуев

Санкт-Петербург
2023 г.

Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление: 09.03.01 – “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___”_____ 2023 г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент А Н. Багиев Группа № 9307

1. **Тема:** анализ эффективности векторизации кода в современных компиляторах.
   (утверждена приказом № _____ от _____)
   Место выполнения ВКР: кафедра ВТ
2. **Объект исследования:** векторизующий компилятор
3. **Предмет исследования:** автоматическая векторизация
4. **Цель:** анализ эффективности векторизации кода в современных компиляторах
5. **Исходные данные:** учебная литература, периодические издания.
6. **Содержание:** актуальность темы, тестовый набор циклов ETSVC, выбор компиляторов для теста, вычислительная система для тестов, Анализ реализации векторизации кода в компиляторах, тестирование компиляторов.
7. **Дополнительные разделы:** обеспечение качества разработки, продукции, программного продукта.
8. **Результаты:** приложение, текст ВКР, иллюстративный материал

Дата выдачи задания «_____» _____ 2023 г.

Дата представления ВКР к защите «_____» _____ 2023 г.

Руководитель
к. т. н., доцент _____ А. А. Пазников

Студент _____ А. Н. Багиев

Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление: 09.03.01 – “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2023 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема Анализ эффективности векторизации кода в современных компиляторах

Студент А. Н. Багиев Группа № 9307

№ этапа	Наименование работ	Срок выполнения
1	Аналитический обзор предметной области	14.04.2023–17.04.2023
2	Изучение реализации векторизации в LLVM/clang	18.04.2023–24.04.2023
3	Изучение работы «VPlan»	25.04.2023–30.04.2023
4	Изучение реализации векторизации в gcc	01.05.2023–10.05.2023
6	Описание экономического обоснования	10.05.2023–15.05.2023
7	Оформление материалов ВКР	16.05.2023–25.05.2023
8	Предварительное рассмотрение работы	13.06.2023
9	Представление работы к защите	16.06.2023

Руководитель
к. т. н., доцент _____ А. А. Пазников

Студент _____ А. Н. Багиев

РЕФЕРАТ

Пояснительная записка 57 стр., 16 рис., 4 табл., 6 прил.

Объектом исследования является компилятор, который транслирует код, написанный на языке программирования в машинный.

Предметом исследования в данной работе будет автоматическая реализация, которая представлена в большинстве современных компиляторов.

Цель работы заключается в анализе реализации автоматической векторизации в современных компиляторах.

Отличительной особенностью данной работы является изучение автоматической векторизации в известных компиляторах, которое может быть применено на практике для улучшения скорости работы программ.

Большинство современных процессоров включают в себя векторные блоки, которые были разработаны для ускорения работы однопоточных программ. Хотя векторные инструкции могут обеспечить высокую производительность, написание векторного кода на языке ассемблера или использование встроенных функций на языках высокого уровня является трудоемкой и подверженной ошибкам задачей. Альтернативой является автоматизация процесса векторизации с помощью векторизующих компиляторов.

ABSTRACT

The object of the study is a compiler that translates code written in a programming language into a machine one.

The subject of research in this paper will be the automatic implementation, which is presented in most modern compilers.

The purpose of the work is to analyze the implementation of automatic vectorization in modern compilers.

A distinctive feature of this work is the study of automatic vectorization in well-known compilers, which can be applied in practice to improve the speed of programs.

Most modern processors include vector blocks, which were designed to speed up the work of single-threaded programs. Although vector instructions can provide high performance, writing vector code in assembly language or using built-in functions in high-level languages is a time-consuming and error-prone task. An alternative is to automate the vectorization process using vectorizing compilers.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

SSA (static single assignment form) – свойство промежуточного представления (IR), которое требует, чтобы каждая переменная присваивалась ровно один раз и определялась перед ее использованием.

SIMD (single instuction , multiple data) – принцип вычислений в компьютерах, позволяющий обеспечить параллелизм на уровне данных.

АЛУ (Арифметико-логическое устройство) – блок процессора, который служит для выполнения арифметических и логический преобразований над данными.

VF (vector factor) – коэффициент векторизации в LLVM/Clang.

UN (unrolling factor) – коэффициент разворачивания цикла в LLVM/Clang.

LNO (loop nest optimization) – в проектировании компиляторов оптимизации вложенных циклов, который использует набор преобразований циклов с целью оптимизации вложенных циклов.

SCEV (scalar evolutions) – скалярные эволюции используются для представления результатов анализа индукционных переменных в GIMPLE.

IV (induction variable) – индуктивная переменная, переменная в циклах, последовательные значения которой образуют арифметическую прогрессию.

CFG (control flow graf) – граф потока управления, множество всех возможных путей исполнения программы, представленное в виде графа.

LHT (loop hierarchy tree) – дерево иерархии циклов, основано на графе потока управления.

SCC (strongly connected components) – сильно связанные компоненты.

ВВЕДЕНИЕ

Оптимизация в наше время приобретает все большее значение. Векторизация кода дает возможность ускорить работу программы в несколько раз, что позволяет производить сложные вычисления за разумное время.

Актуальность данной работы обусловлена следующим: сейчас, в эпоху, когда нужна высокая скорость вычислений, производители процессоров решили использовать сложную архитектуру в процессорах и начали внедрять различные аппаратные средства, которые одновременно выполняют векторные операции (обрабатываются несколько операндов одновременно). Поэтому разработчики компиляторов внедряют множество способов автоматической векторизации. Есть несколько методов автоматической векторизации:

- Автоматическая векторизация на уровне цикла;
- автоматическая векторизация на уровне базового блока;
- автоматическая вектроризация потока управления;
- сокращение накладных расходов на векторизацию при наличии потока управления.

Целью данной работы является анализ автоматической векторизации кода в современных компиляторах, а конкретно, методов, которые реализованы в современных компиляторах.

Объектом данной работы является компилятор, выполняющий автоматическую векторизацию. Векторизация – это вид оптимизации, который изменяет однопоточные приложения, которые выполняют по одной операции в момент времени, на те, которые выполняют несколько одиночных операций одновременно.

Предметом данной работы является автоматическая векторизация. Автоматическая векторизация подразумевает, что код, который был написан с использованием скалярных инструкций, будет изменен на векторизованный при компиляции и без вмешательства со стороны человека.

Основными задачами данной работы является:

- Анализ предметной области;

- выбор компиляторов для анализа;
- анализ реализованных в компиляторах методов автоматической векторизации;
- тестирование выбранных компиляторов;
- рекомендации по использованию автоматической векторизации в компиляторах.

В первом разделе работы производится анализ предметной области, выбор компиляторов для анализа, описание методов оптимизации и выбор тестовых данных. Во втором разделе будут рассмотрены компиляторы, которые были выбраны в первом разделе. В третьем разделе будет произведено тестирование компиляторов и даны рекомендации по их использованию. В дополнительном разделе рассмотрено обеспечение качества разработки продукции, программного продукта.

1 Предпосылки и появление автоматической векторизации. Представление методов оптимизации. Выбор тестовых наборов

В данном разделе описываются основные предпосылки и развитие технологий позволивших появиться векторизации в целом. анализируются компиляторы, которые будут в последствии изучены и выбирается тестовый набор, на котором будет происходить тестирование.

1.1 Развитие процессоров и появление возможности векторизации

На данный момент в производстве современных процессоров перестала расти мощность отдельных ядер.

Как можно наблюдать на рисунке 1, представленном выше, до 2004 года закон Мура и закон масштабирования Деннарда работали, но причина по которой тактовая частота больше не растет заключается в удельной мощности, как можно видеть ниже, на рисунке 2, предполагалось, что температура процессора может достигнуть температур ядерного реактора и выше.

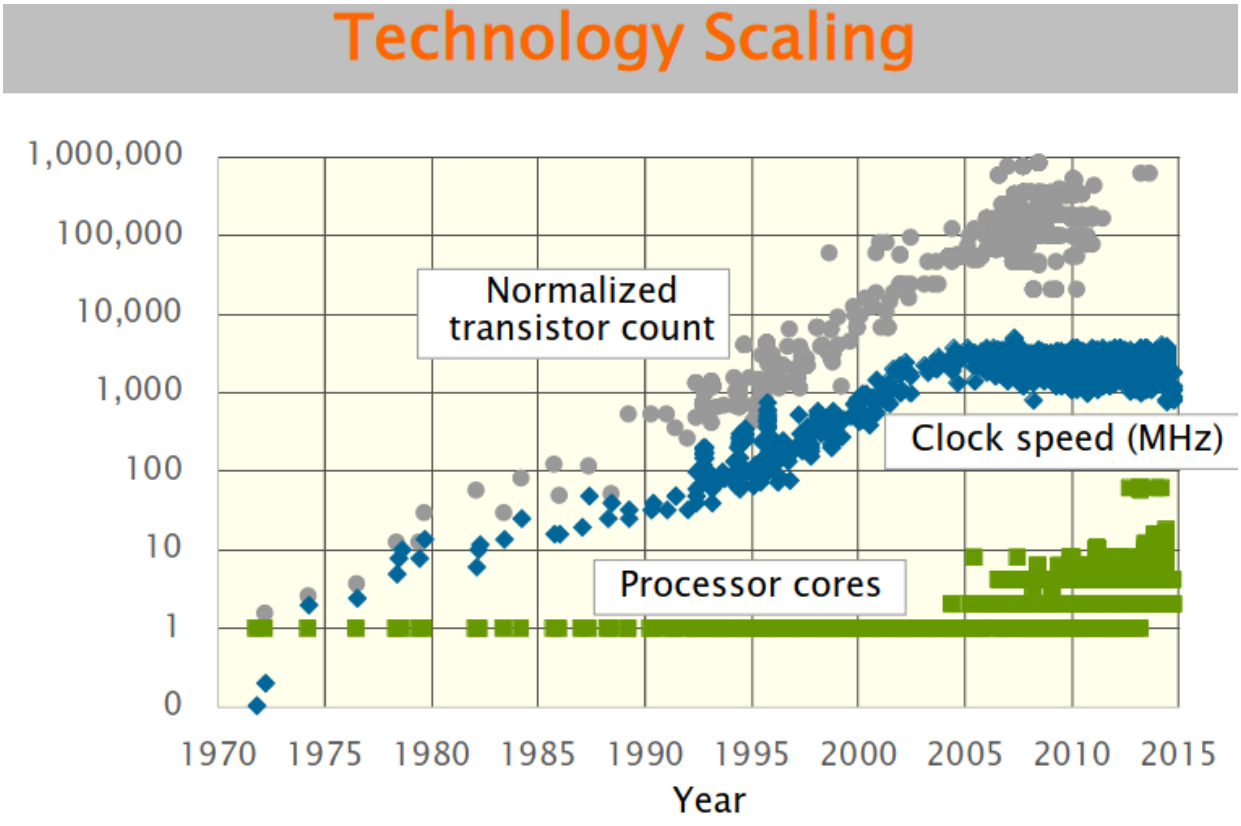


Рисунок 1 – Данные процессора из базы данных процессоров Стэнфорда [1]

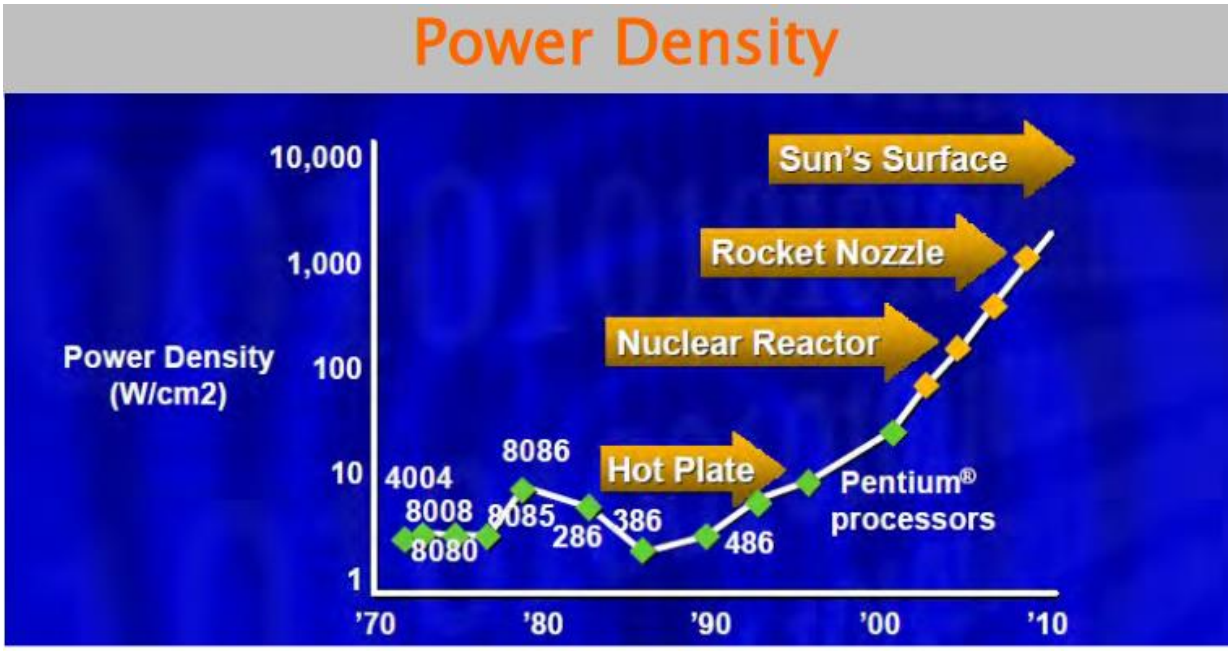


Рисунок 2 – Прогнозы роста удельной мощности [1]

Из-за этого производители процессоров ввели параллелизм в виде многоядерных процессоров (в чип помещается более одного процессорного ядра), а что бы масштабировать производительность, нужно иметь несколько ядер (пример на рисунке 3), и теперь каждое поколение закона Мура потенциально удваивает количество ядер, что наблюдается на рисунке 1. Поэтому теперь, чтобы увеличить производительность, нужно заниматься параллельным программированием.

Современный многоядерный настольный процессор содержит ядра параллельной обработки, векторные блоки, кэши, средства предварительной выборки, графические процессоры, гиперпоточность, динамическое масштабирование частоты и т. д..

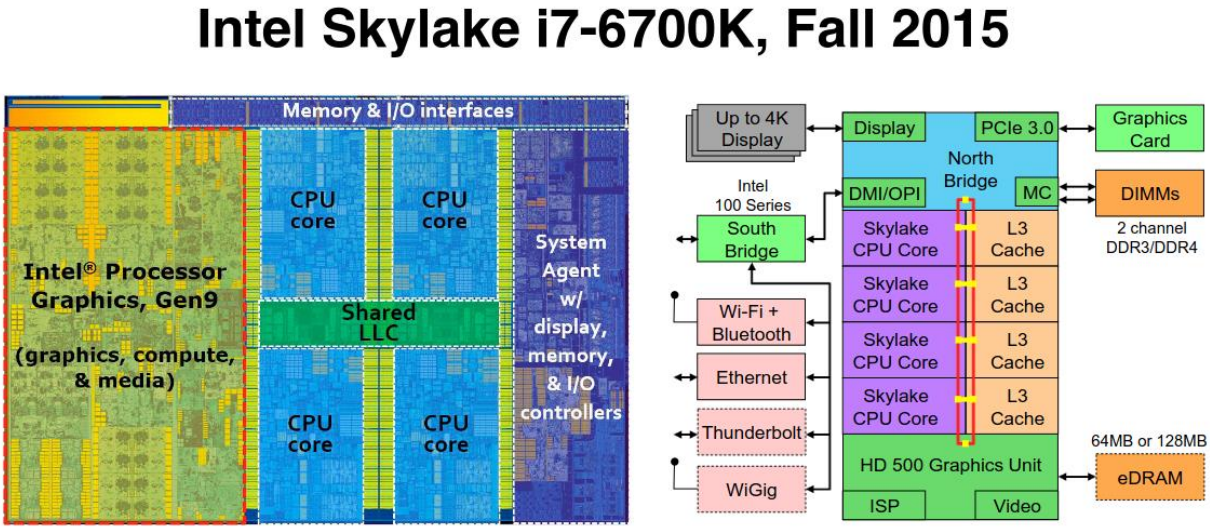


Рисунок 3 – Пример архитектуры процессора

Благодаря нескольким ядрам современные процессоры предоставляют параллелизм на уровне вычислительных ядер (многопоточное программирование), отдельные ядра реализуют суперскалярную архитектуру с конвейерным принципом выполнения команд (в таких ядрах содержится множество одновременно функционирующих АЛУ, реализующих параллелизм команд), а также вычислительные ядра реализуют параллелизм уровня данных векторными АЛУ (SSE, AVX, AltiVec). Высока производительность процессора достижима только в том случае, если в нем реализованы различные возможности по ускорению вычислений, например, концепции внеочередного выполнения команд, переименование регистров, предсказание условных переходов, обнаружения программных циклов и т. д [2].

Векторизация является важной оптимизацией для максимального повышения производительности приложений в современных системах. Ключевым принципом векторизации является определение возможностей для выполнения аналогичного набора операций над несколькими элементами данных с помощью одной инструкции. Архитектуры, поддерживающие такую форму параллелизма, часто называют системами с несколькими данными с одной инструкцией (SIMD). Современные компиляторы способны в определенной сте-

пени автоматизировать генерацию векторизованного кода. Однако точная результирующая производительность зависит от ряда факторов, включая: тип приложения, качество кода, длину векторных регистров, типы операций и количество доступных векторных блоков. В зависимости от выбранного языка программирования потенциальными кандидатами для векторизации являются гнезда циклов, абстрактные операторы, работающие с массивами, и обращения к памяти. В идеале компиляторы должны идентифицировать всех возможных кандидатов и векторизовать их таким образом, чтобы инструкции SIMD использовались для выполнения как можно большего количества операций параллельно.

Хотя современные компиляторы обеспечивают автоматическую оптимизацию векторизации, возможности компиляторов по полной автоматической векторизации заданного фрагмента кода часто ограничены. В первую очередь это связано с отсутствием контекстной информации, которую компиляторы могут извлечь или воспринять из кода. Фактически, значения многих переменных становятся известны только во время выполнения, что уменьшает эффективность векторизации во время компиляции. Это отсутствие или недостаток информации во время компиляции вынуждает компиляторы генерировать неоптимальный векторизованный код. Таким образом, важно оценить, способны ли компиляторы создавать векторизованный код, и измерить их эффективность с точки зрения возможностей целевой архитектуры.

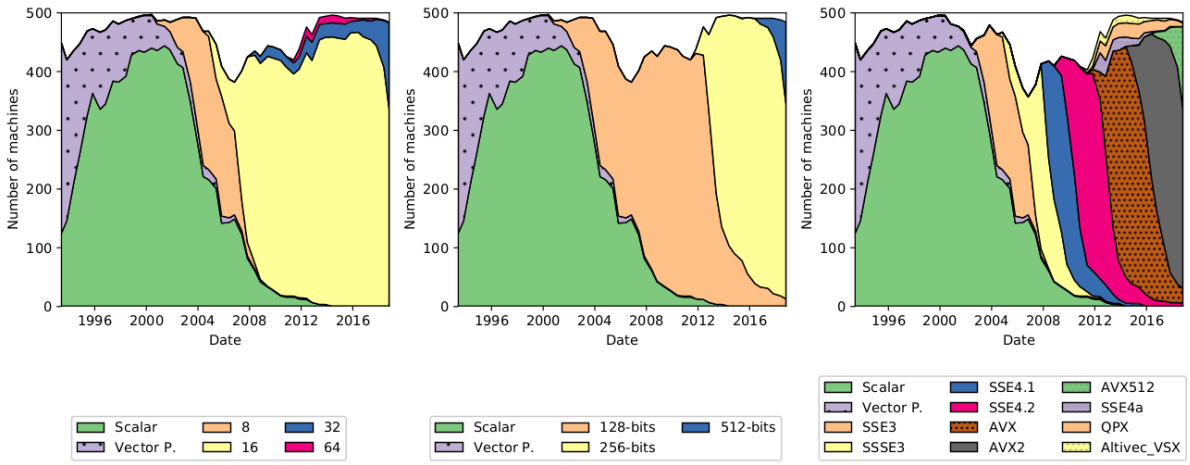
Набор тестов для векторизации компиляторов (TSVC) и расширений TSVC, предоставленный Maleki et al., которые смогли создать прецедент для оценки возможностей векторизации и до сих пор широко используются. TSVC и подобные тестовые наборы состоят из наборов циклов с определенной реализацией и фиксированным и обычно щедрым объемом контекстной информации, предоставляемой компилятору. Хотя циклы предназначены для использования возможностей векторизации, контекстная информация дает наилучший возможный результат. Благодаря контекстной информации,

предоставляющей компилятору все возможные данные, необходимые для векторизации, компиляторы обеспечивают наилучшую возможную производительность. Другими словами, этот подход дает общую картину производительности компиляторов при конкретном сценарии, который является почти идеальным случаем. Хотя, это полезно для того, чтобы знать наилучшие возможности векторизации, которые может предоставить компилятор, на самом деле этот подход не позволяет охватить весь спектр поведения компилятора при наличии ограниченного объема информации. В действительности, реалистичным научным приложениям не хватает значительного объема информации во время компиляции, что напрямую противоречит подходу к проектированию, принятому TSVC.

Более того, с момента создания TSVC конструкции как компилятора, так и процессора изменились довольно кардинально. Например, ряд процессоров, поддерживающих векторные возможности, существенно эволюционировали. Заметным и хорошо консолидированным ресурсом, подтверждающим это, является Топ-500 суперкомпьютеров в мире [3].

Путем ручного анализа и компиляции списка Top-500, соответствующих руководств по системам и соответствующих руководств по процессорам для систем Top-500 в период с 1993 по 2018 год можно наблюдать, как изменились возможности векторной обработки за 25 лет на рисунке 4. Этот анализ фокусируется исключительно на главном процессоре и не принимает во внимание аппаратное обеспечение ускорителя, доступное в системах. Тем не менее, из рисунка можно сделать ряд выводов. Во-первых, систем без возможностей SIMD-вектора больше нет. Во-вторых, за последние годы длина векторных регистров новейших машин увеличилась в четыре раза (со 128 бит до 512 бит), и внедрение каждого нового набора команд происходит довольно быстро. Наконец, как показано на рисунке 4, системы с наборами команд AVX-2 доминируют в списке Top-500, при этом они так же быстро набирают популярность.

По причине того, что отсутствует информация о реализуемых методах компиляции, анализ будет выполняться по методу “черного ящика” – с помощью набора тестовых циклов.



(a) Number of Vector Registers. (b) Length of Vector Registers. (c) Vector ISAs.

Рисунок 4 – Изменение различных векторных возможностей в системах, входящих в Топ-500 за последние 20 лет

1.2 Выбор компиляторов для теста

Современные процессоры x86-64 представляют собой очень сложные машины с архитектурой CISC. Современные векторные расширения для архитектуры x86-64, такие как AVX2 и AVX-512, содержат инструкции, разработанные для работы с обычными вычислительными ядрами. Например, объединенная команда умножения-сложения используется для повышения производительности и точности в плотной линейной алгебре (плотные матрицы), команда collision detection подходит для операции объединения в статистические вычисления, а команды с битовой маской предназначены для обработки ветвей в векторных вычислениях. Однако рабочие нагрузки со сложными схемами доступа к памяти и нестандартными ядрами требуют значительной работы как от программиста, так и от компилятора для достижения наивысшей производительности.

В то же время современные языковые стандарты прилагают все усилия, чтобы абстрагироваться от деталей базового оборудования и структур данных и генерировать общий код, который больше соответствует логике и математике, чем инструкциям и ячейкам памяти. Новые языковые стандарты уделяют больше внимания конструкциям, которые позволяют программистам выражать свои намерения. Современные стандарты языка C++ движутся в направлении большей выразительности и абстракции. Язык программирования Python популярен благодаря своей удобочитаемости и выразительности, даже ценой снижения скорости выполнения. Удобочитаемые, выразительные языки позволяют создавать код, не содержащий ошибок и поддерживаемый в дальнейшем.

Следствием увеличения выразительности является возросшая нагрузка на компилятор по созданию хорошего ассемблерного кода из сложных высокоуровневых конструкций, написанных программистами. Компиляторы должны быть “умнее”, чтобы получить из кода максимальную производительность. Не все компиляторы равны, и некоторые из них лучше других справляются с одним и тем же фрагментом кода и производят эффективную сборку.

Нужно протестировать наиболее распространенные компиляторы C/C++. Чтобы рассматриваться в качестве кандидата для нашего теста, компилятор должен:

- Быть с открытым исходным кодом
- Компилироваться для архитектуры x86-x64,
- Быть доступным для платформы Linux,
- Находиться в стадии активной разработки;

Следующие 2 компилятора соответствуют нашим критериям:

- GNU Compiler Collection/GCC
- LLVM/Clang

Все протестированные компиляторы выдают инструкции AVX-512.

Чтобы протестировать производительность скомпилированного HPC-кода (high performance computing), мы предлагаем компиляторам три вычислительные программы:

- Простая реализация LU-декомпозиции без поворота с простыми структурами данных и схемой доступа к памяти и без какой-либо ручной настройки. Тестируем неоптимизированную версию программы, чтобы определить, как каждый компилятор обрабатывает "наивный" исходный код со сложными шаблонами векторизации и потоковой обработкой, скрытыми внутри,
- Высоко абстрактная объектно-ориентированная реализация метода Якоби для задачи Пуассона. Эта программа проверяет, насколько хорошо компиляторы могут выполнять сложный кросс-процедурный анализ кода для обнаружения общих параллельных шаблонов
- Тщательно настроенная реализация вычисления структурных функций. Мы полностью оптимизируем это ядро до такой степени, что оно привязано к вычислениям, то есть ограничено возможностями арифметической производительности центрального процессора. Цель этого теста - увидеть, насколько эффективен результирующий двоичный файл, когда исходный код хорошо осведомлен о базовой архитектуре [4].

Исходя из вышеприведенных исследований принято решение использовать следующие компиляторы:

- GNU Compiler Collection/GCC
- LLVM/Clang

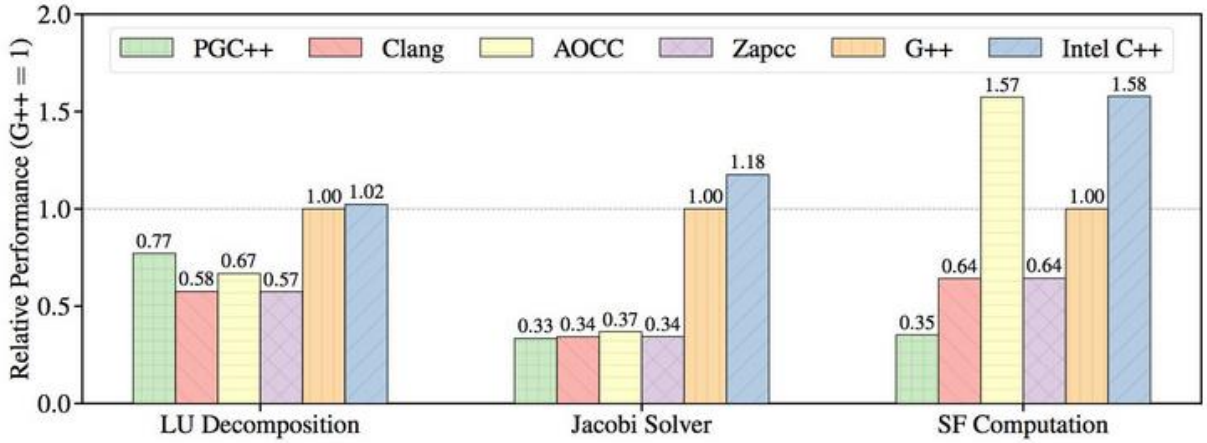


Рисунок 5 – Относительная производительность каждого ядра, скомпилированного различными компиляторами. (однопоточный, чем выше, тем лучше)

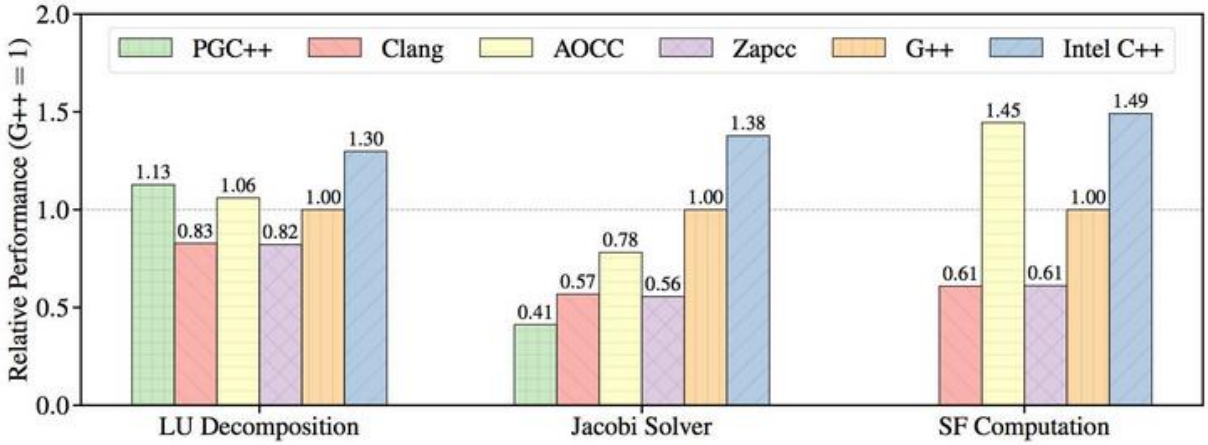


Рисунок 6 – Относительная производительность каждого ядра, скомпилированного различными компиляторами. (многопоточный, чем выше, тем лучше)

Вывод: после проведенных экспериментов были выбраны gcc и clang, хоть и можно наблюдать, что некоторые компиляторы лучше в некоторых задачах, остается проблема того ,что они коммерческие и изучить их проблематично, либо не предоставляется возможным.

1.3 Тестовый набор циклов ETSVC

Целью набора тестов является тестирование четырех основных областей векторизующего компилятора: анализ зависимостей, векторизация. распознавание идиом и полнота языка. Компилятор с автоматической векторизацией — это тот, который берет код, написанный на последовательном языке

(C\C++), и преобразует его в векторные инструкции. Векторные инструкции могут быть специфичными для конкретной машины или в исходной форме, или в виде вызовов подпрограмм для векторной библиотеки. Циклы, которые присутствуют в наборе тестов, были написаны людьми, участвовавшими в разработке векторизующих компиляторов. Эти циклы отражают конструкции, векторизация которых варьируется от простой до сложной и чрезвычайно сложной.

Установлено, что на архитектуре Intel 64 известные алгоритмы способны векторизовать от 34% до 52% циклов из данного пакета.

Ниже представлен пример кода цикла из набора TSVC:

```
real_t s000(struct args_t * func_args)
{

//    linear dependence testing
//    no dependence - vectorizable

   initialise_arrays(__func__);
   gettimeofday(&func_args->t1, NULL);


   for (int nl = 0; nl < 2*iterations; nl++) {
      for (int i = 0; i < LEN_1D; i++) {
         a[i] = b[i] + 1;
      }
      dummy((real_t*)a, (real_t*)b, (real_t*)c, (real_t*)d, (real_t*)e, aa, bb, cc, 0.);
   }


   gettimeofday(&func_args->t2, NULL);
   return calc_checksum(__func__);
}
```

1.4 Оптимизация кода

Целевой код, сгенерированный компиляторами, все еще не так хорош, как код, созданный опытным программистом на языке ассемблера

Компиляторам необходимо выполнять преобразования, которые будут улучшать код, чтобы сокращать время выполнения целевого кода. Эти преобразования называют оптимизацией, хотя редко код бывает оптимальным в

формально м смысле этого слова. Преобразования, которые выполняет оптимизирующий компилятор, должны соответствовать следующим свойствам:

- Преобразования должны сохранять смысл (семантику) программ;
- преобразование, в среднем, должно ускорять выполнение программ;
- размер программ должен оставаться разумным;
- объем усилий компилятора, затрачиваемый на оптимизацию кода, должен быть пропорционален:
    - количеству раз, когда код будет запущен;
    - ко всей программе.

Общие проблемы оптимизации кода:

- являются NP – полными, и поэтому мы не можем ожидать, что найдем глобальный оптимум за разумный промежуток времени;
- стратегия основана, в основном на эвристике, то есть определить горячие точки и вложить много ресурсов в их оптимизацию.

Несколько примеров горячих точек:

- циклы или вложенные циклы;
- оптимизация или исключение вызовов процедур;
- распознавание общих подвыражений;
- распространение констант;
- и многое другое.

1.4.1 Оптимизация трехадресного кода

Трехадресный код (thee-address code, TAC) – это промежуточное представление исходной программы, которая выглядит как последовательность операторов вида x := y op z, где x, y, z – имена, константы или сгенерированные компилятором временные объекты. op – это двуместная операция. Смысл данного представления в том, что каждый оператор имеет три адреса: два для операндов и один для результата. Оптимизация трехадресного кода опирается на несколько простых концепций:

- Базовый блок (Basic block, BB)

- Графы потока управления (control flow graph, CFG)

Базовый блок – это последовательность TAC, в которой поток управления входит в начале и выходит в конце. Пример:

```
L3:   t5 := a * b
      t6 := t5 + c
      d := t2 * t2
      if d = 0 goto L4
```

Метод распознавания базовых блоков:

1) Операторы, с которых начинается базовый блок, идентифицируются путем поиска:
   a) Первого оператора любой функции
   b) Любое помеченное утверждение, являющееся целью перехода (условного или безусловного)
   c) Любое утверждение, следующее за переходом
2) Для каждого оператора, запускающего базовый блок, блок состоит из всех операторов, вплоть до (но исключая) начала следующего базового блока или конца программы.

Манипулирование TAC в базовом блоке:

Подразумевает, что нам нужно беспокоиться только о последствиях, вызванных значениями переменных в начале блока и в его конце, обычно называется локальной оптимизацией

Граф потока управления

Базовые блоки могут быть организованы в ориентированный граф: граф потока управления (control flow graph, CFG), чьи узлы являются базовыми блоками и таковы, что существует ребро, идущее от базового блока A к базовому блоку B, если управление может передаться от A к B. Пример:

```
      i := 20
      s := 20
L1:   if i=0 goto L2
      s := s + i
      i := i - 1
      goto L1
L2:
```

Анализ переменных в реальном времени.

Первый алгоритм вычисляет временные переменные блока B, на которые никогда не ссылаются, для вычисления правильных значений действующей переменной. Определение времени жизни и использования переменных в программе называется анализом в реальном времени. Можно сказать базовый блок вычисляет значения переменных, на момент выхода из блока.

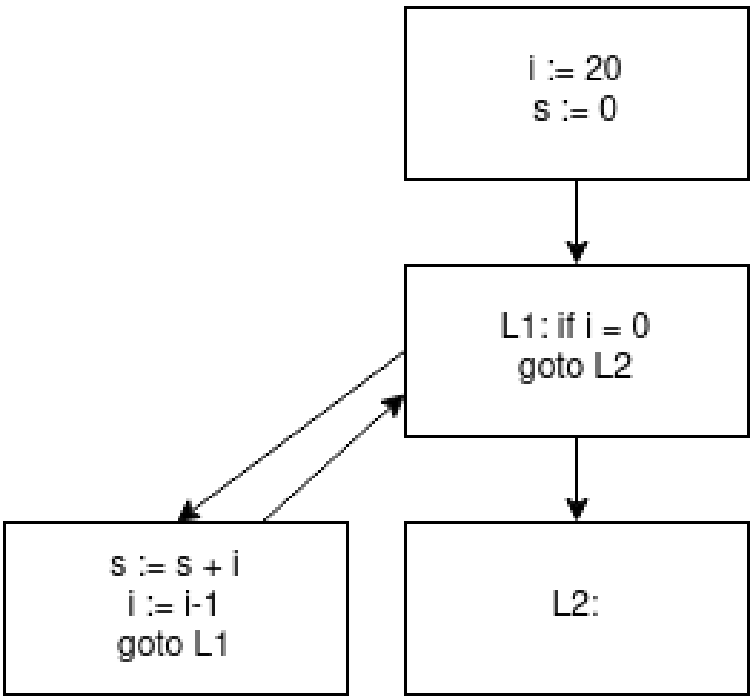


Рисунок 7 – Граф потока управления

1.4.2 Определение бесполезных имен для базового блока

В следующем алгоритме базового блока B, предполагается, что lives(B) – это набор переменных, появляющихся в B, на которые ссылаются в базовом блоке B’, следующем за B в CFG.

Input: A basic block B consisting of TAC statements $s_1, \ldots, s_\ell$ in this order.

Output: The set of the useless names in B.

$D := \emptyset$
$L := \mathsf{lives}(B)$
for $i := \ell..1$ **step** -1 **repeat**
  let s_i be `x := y op z`
  if `x` $\notin L$
    then $D := D \bigcup \{$ `x` $\}$
    else $L := L \bigcup \{$ `y, z` $\}$
return $D \setminus L$

Рисунок 8 – Первый алгоритм

1.4.3 Устранение мертвого кода

- Удаление любого оператора TAC x := y po z таки образом, что бы на x больше не ссылались в программе;
- замена любого перехода к неработающей строке кода переходом ко всему, что следует за этой строкой.

Мертвый код может возникать по множеству причин, самой частой причиной бывает распространение копий.

1.4.4 Устранение обычного подвыражения

Рассмотрим следующий TAC:

```
t1 := 4 - 2
t2 := t1
t3 := a * t2
t4 := t3 * t1
t5 := t4 + b
t6 := t3 * t1
t7 := t6 + b
c := t5 * t7
```

В данном примере видно, что выражения для t4 и t6 одинаковы поэтому строку с t6 можно изменить:

t6 := t3 * t1 => t6 := t4

1.4.5 Распространение копий

После утверждения x := y мы знаем что x и y имеют одинаковое значение, мы можем заменить все вхождения x на y между тем присваиванием и следующим определением y. Распространение копий делает возможным дальнейшее устранение подвыражений.

1.5 Вычислительная система для тестов

Таблица 1 – Вычислительная система.

Processor	Intel Core
Model	i5-11400H
Architecture	X86(64)
Number of Cores	6
Core Frequency	2.7 Ghz
Vector ISA	AVX2, AVX-512
Vector Length	256 and 512 bits
Memory	128GB
L1 d Cache	96 Kb
L2 Cache	1.25 Mb
L3 Cache	12 Mb
OS	Ubuntu
Compilers Used	gcc clang

1.6 Вывод

В данном разделе была рассмотрена актуальность и историческое развитие темы. Рассмотрена архитектура современного процессора и модулей, которые в нем присутствуют, а также современных методов векторизации и инструкций. Также представлены некоторые проблемы, которые присутствуют при попытке автоматизировать процессы векторизации в данном направлении и была представлена система, на которой будут проводиться тесты.

2 Анализ реализации векторизации кода в компиляторах

В данном разделе будет произведен небольшой обзор некоторых компиляторов и анализ реализованного в них функционала автоматической векторизации кода.

2.1 Обзор LLVM

Проект LLVM представляет собой набор модульных и повторно используемых компиляторов и технологий цепочки инструментов. Несмотря на свое название, LLVM имеет мало общего с традиционными виртуальными машинами.

LLVM начинался как исследовательский проект в Университете Иллинойса с целью создания современной стратегии компиляции на основе SSA, способной поддерживать как статическую, так и динамическую компиляцию произвольных языков программирования. С тех пор LLVM превратился в зонтичный проект, состоящий из ряда подпроектов, многие из которых используются в производстве широким спектром коммерческих проектов и проектов с открытым исходным кодом, а также широко используется в академических исследованиях.

Само ядро проекта содержит все инструменты, библиотеки и заголовочные файлы, необходимые для обработки промежуточных представлений и преобразований их в объектные файлы. Инструменты включают ассемблер, дизассемблер, анализатор битового кода и оптимизатор байт-кода. Он также содержит базовые регрессионные тесты.

Языки, подобные C, используют интерфейс Clang. Этот компонент компилирует код C,C++,Objtctive C и Objective C++ в битовый код LLVM, а оттуда в объектные файлы, используя LLVM.

Основными подпроектами LLVM являются:

1. Библиотеки ядра LLVM предоставляют современный оптимизатор, независимый от исходного кода и цели, а также поддержку генерации коа для многих популярных процессоров. Данные библиотеки, построенные

на представлении кода, известного как промежуточное представление LLVM (IR).

Большинство оптимизаций компилятора выполняется в промежуточном представлении компилятора (IR), хотя и не все из них.

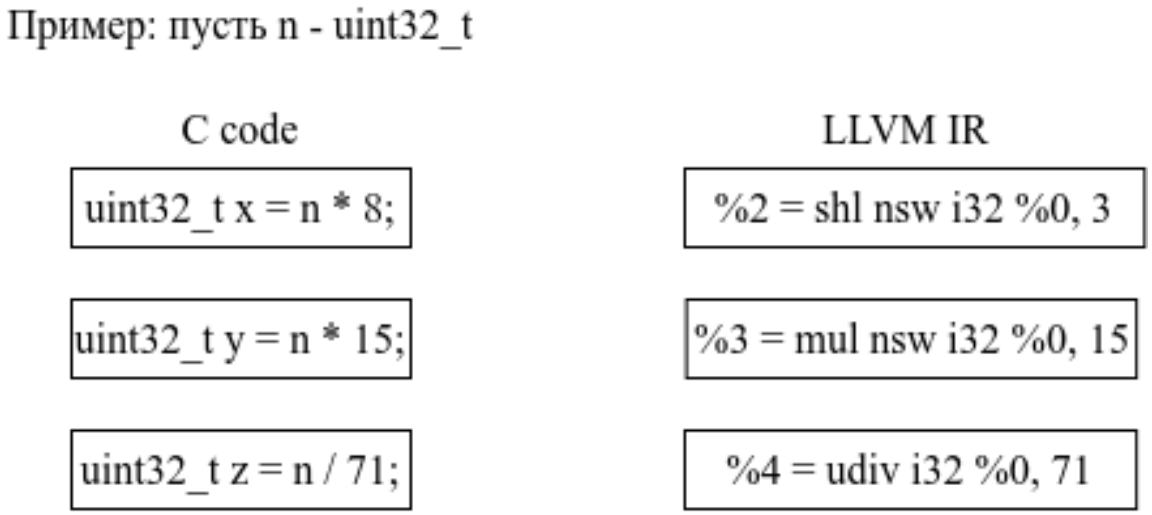


Рисунок 9 – Промежуточное представление LLVM

2. Clang – это компилятор C/C++/Objective-C, целью которого является обеспечение быстрой компиляции, полезными сообщениями об ошибках и предупреждениями.
3. Подпроект OpenMP предоставляет среду выполнения OpenMP для использования с реализацией OpenMP в Clang.
4. Подпроект MLIR — это подход к созданию повторно используемой и расширяемой инфраструктуры компилятора. Цель MLIR - устранить фрагментацию программного обеспечения, улучшить компиляцию для разнородного аппаратного обеспечения, значительно снизить стоимость создания компиляторов, специфичных для предметной области, и помочь в объединении существующих компиляторов.

   И некоторые другие модули.

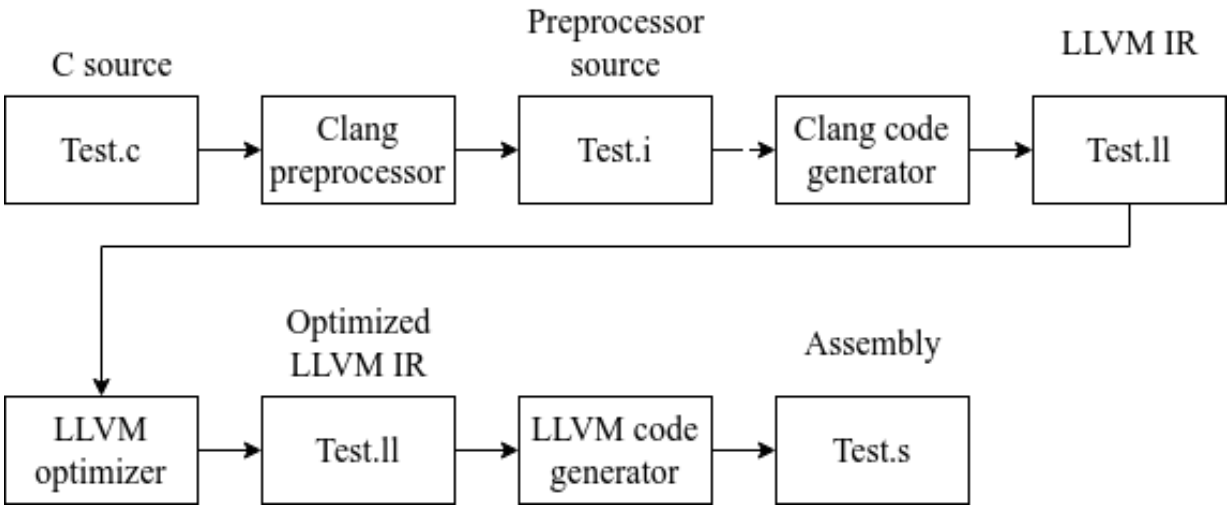


Рисунок 10 – Конвейер Clang/LLVM

2.2 Реализация векторизации в LLVM/Clang

В этой части будет произведен анализ векторизации кода компилятора LLVM. LLVM векторизатор работает по плану векторизации (VPlan). План векторизации — это явная модель для описания кандидатов на векторизацию Он служит как для оптимизации кандидатов, включая достоверную оценку их стоимости, так и для выполнения окончательного перевода в IR (промежуточный язык). Это облегчает работу с несколькими кандидатами на векторизацию. Векторизация на основе «VPlan» включает в себя три основных этапа, используя «сценарный подход» к планированию векторизации:

1. Шаг проверки: проверяется возможность эффективно векторизовать цикл, если такая возможность существует, то кодируются ограничения.
2. Этап планирования:
    a. Создается первоначальные планы векторизации (разные способы векторизации) в соответствии с ограничениями и решениями, принятыми на шаге 1, и происходит расчет стоимости
    b. Применяется оптимизация к планам векторизации, возможно, создав дополнительные планы. Отсекаются неоптимальные планы, имеющие относительно высокую стоимость.
3. Шаг выполнения: реализуется наилучший план векторизации. Это единственный шаг, который меняет IR.

Рассмотрим работу плана векторизации подробнее. В дальнейшем термин «входной IR» будет относиться к коду, который вводится в векторизатор, а «выходной IR» относится к коду, который был получен после векторизации. Выходной IR содержит код, который был векторизован или «расширен» в соответствии с коэффициентом векторизации цикла (VF) и/или циклически разворачивался и заканчивался в соответствии с коэффициентом разворачивания (UF). Дизайн плана векторизации соответствует нескольким принципам высокого уровня:

1. Анализ: построение «VPlan» и манипулирование ими не должны изменять входной IR. В частности, если наилучшим вариантом является вообще не выполнять векторизацию, процесс векторизации завершается до достижения шага 3, и компиляция продолжается, как если бы «VPlans» не были созданы.
2. Согласование затрат и выполнение: каждый план векторизации поддерживает как оценку затрат, так и генерацию выходного IR кода.
3. Поддержка векторизации дополнительных конструкций:
    a. Векторизация по внешнему контуру. В частности, план векторизации способен моделировать поток управления выходным IR, который может включать в себя несколько базовых блоков и вложенных циклов. Это нужно в случае, если количество итераций в самом внутреннем цикле невелико. В этом случае векторизация по внутреннему циклу невыгодна. Однако, если внешний цикл содержит больше работы, комбинация функций с поддержкой SIMD, может принудительно выполнить векторизацию на внешнем выгодном уровне.
    b. Векторизация SLP
    c. Комбинации из вышеперечисленного, включая вложенную векторизацию: векторизация как внутреннего цикла, так и внешнего цикла одновременно (каждый со своими собственными VF и UF), смешанная векторизация: векторизация цикла с шаблонами SLP

внутри (повторная) векторизация входного IR, содержащего векторный код.

d. Векторизация функции:

Clang FE (front end) анализирует код и генерирует имена, включающие в себя префиксы в качестве векторных подписей. Эти названия-префиксы записываются как атрибуты функции LLVM. Может быть несколько имен, которые соответствуют разным векторизованным версиям. Например:

```
  #pragma omp delcare simd uniform(a) linear(k)
  float dowork(float *a, int k) {
    a[k] = sinf(a[k]) + 9.8f;
}
define __stdcall f32 @_dowork(f32* %a, i32 %k) #0
  attributes #0 = { nounwind uwtable "_ZGVbM4ul_" "_ZGVbN4ul_" ...}
```

Для генерации векторной функции введен новый этаg генерации вариантов векторизации из исходной функции, основанных на VectorABI (Vector Function Application Binary Interface). VectorABI предоставляет интерфейс для векторных функций, генерируемых компилятором, который поддерживает SIMD-конструкции OpenMP.

```
define __stdcall <4 x f32> @_ZGVbN4ul_dowork(f32* %a, i32 %k) #0 {
    #pragma clang loop vectorize(enable)
    for (int %t = k; %t < %k + 4; %t++) {
      %a[t] = sinf(%a[t]) + 9.8f;
    }
    vec_load xmm0, %a[k:VL]
    return xmm0;
}
```

Тело функции заключено в цикл, имеющий VL итераций, которые соответствуют длине вектора. LLVM LoopVectorizer векторизует сгенерированный цикл, как ожидается для получения следующего векторизованного кода:

```
define __stdcall <4 x f32> @_ZGVbN4ul_dowork(f32* %a, i32 %k) #0 {
    vec_load xmm1,  %a[k: VL]
    xmm2 = call __svml_sinf(xmm1)
    xmm0 = vec_add  xmm2, [9,8f, 9.8f, 9.8f, 9.8f]
    store %a[k:VL], xmm0
    return xmm0;
  }
```

4. Эффективное поддержание нескольких вариантов векторизации. В частности, аналогичные варианты, относящиеся к целому ряду возможных VF и UF.
5. Поддержка векторизации идиом, таких как чередующиеся группы ступенчатых загрузок или хранилищ.
6. Инкапсуляции областей с одним входом и выходом (SESE).
7. Поддержка анализа и преобразования на уровне инструкций.

Векторизатор циклов и SLP – векторизатор. Эти векторизаторы фокусируются на различных возможностях оптимизации и используют различные методы. Векторизатор SLP объединяет несколько скаляров, найденных в коде, в векторы, а векторизатор циклов расширяет инструкции в циклах для выполнения нескольких последовательных итераций. Оба векторизатора включены по умолчанию. Векторизатор циклов также выполняет две функции: векторизует внутренние циклы, разворачивает циклы для ILP.

2.3 Векторизатор внутренних циклов LLVM/Clang

Во время прохода по циклу изменяет «векторизуемые» циклы и генерирует независимый от цели LLVM-IR (промежуточный код). Векторизатор использует анализ, для оценки затрат на инструкции, чтобы оценить рентабельность векторизации. Векторизатор объединяет последовательные итерации цикла в одну «широкую» итерацию. После этого преобразования индекс увеличивается на ширину вектора SIMD. Этот проход состоит из трех частей:

1. Проход основного контура, который приводит в движение различные детали
2. LoopVectorizationLegality – модуль, который проверяет допустимость векторизации
3. InnerLoopVectorizer – модуль, который выполняет фактическое расширение инструкций. Он векторизует только один базовый блок, с заданным коэффициентом векторизации (VF – vector factor). Этот класс выполняет расширение скаляров или скаляры в векторы.

   Этот класс также реализует следующие функции:

a. Он вставляет цикл эпилога для обработки циклов, у которых нет количества итераций, которое кратно коэффициенту векторизации.
b. Он обрабатывает генерацию кода для сокращенных переменных.
c. Скаляризация (реализация с использование скаляров) не векторизуемых инструкций.

Этот модуль не выполняет никаких проверок и полагается на вызывающую программу для проверки различных аспектов допустимости векторизации.

4. LoopVectorizationCostModel - модуль, который оценивает ожидаемое ускорение за счет векторизации. Он определяет оптимальную ширину вектора, которая может быть равна единице, если векторизация не выгодна.

Трансформации:

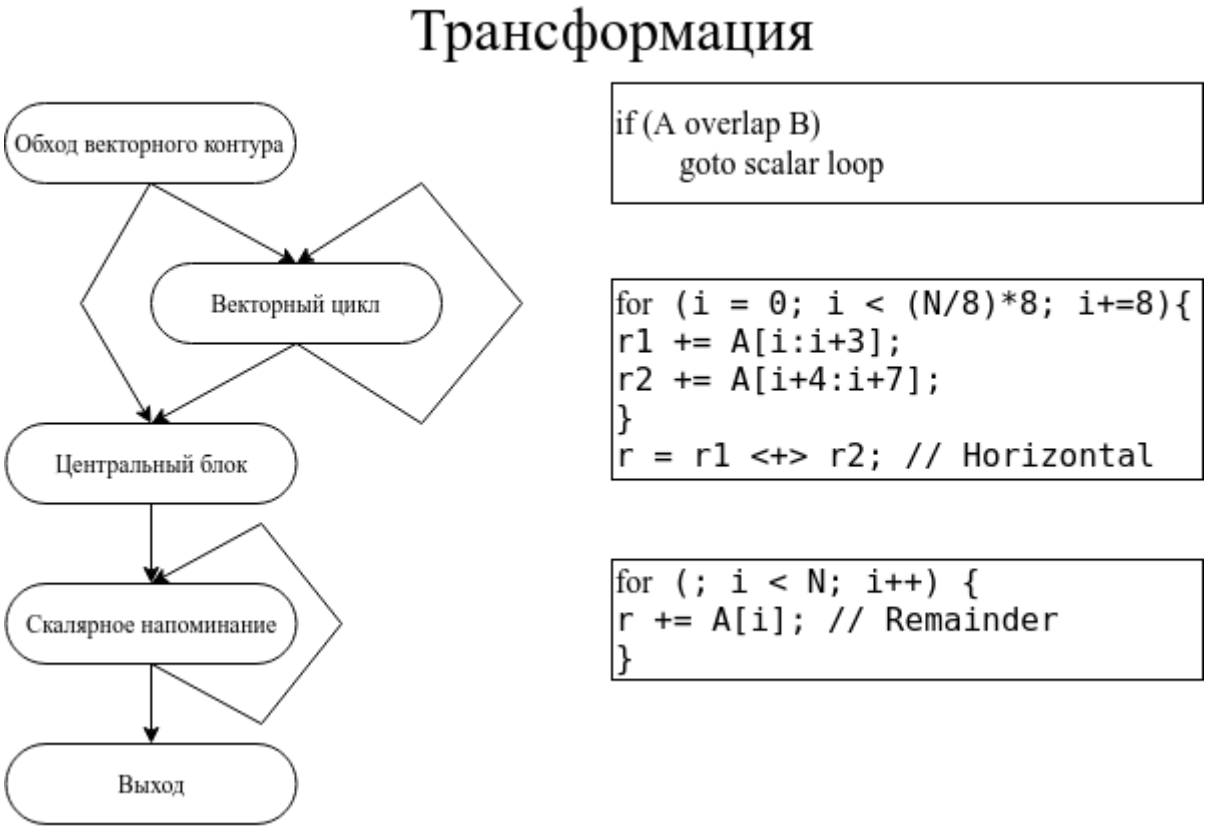


Рисунок 11 – Этап трансформации

Разворот для ILP (Instruction-level parallelism):

- Современные процессоры могут выполнять множество инструкций одновременно;
- сокращения приводят к потере данных (вычисление зависит от предыдущей итерации).

Разворот цикла можно рассмотреть на примере: возьмем простой цикл, в котором на каждой итерации суммируется элементы массива.

```
for (i = 0; i < N; ++i)
        sum += A[i];
```

Для того что бы развернуть цикл нужно учитывать то, что текущий результат зависит от предыдущей итерации, поэтому, если мы хотим уменьшить количество итераций в цикле, нужна еще одна переменная, в которую будут записываться сумма на итерациях, который получилось сократить.

```
for (i = 0; i < n; i+=2) {
        sum0 += A[i];
        sum1 += A[i+1];
}
```

Разворот цикла происходит в векторизаторе, так как это тот же вид анализа, который выполняет векторизатор (например, сокращения и последние итерации цикла). Так же циклический векторизатор часто разворачивает и сохраняет скалярный код.

2.4 SLP – векторизатор LLVM/Clang

Этапы автовекторизации базового блока:

- Идентификация потенциальных сочетаний команд;
- идентификация связанных пар;
- выбор пары;
- слияние пар.
- повторение всей процедуры (метод простой итерации)

Алгоритм автовекторизации базового блока:

Этап первый:

```
Для каждого (инструкция в базовом блоке) {
        Если (инструкция может быть векторизирована)
```

```
            продолжить;
        Для каждого (последующая инструкция в базовом блоке) {
            Если (две инструкции могут быть сопряжены)
                Записать команды в качестве кандидата на векторизацию;
        }
}
```

Какие инструкции могут быть сопряжены:

- Загрузка и сохранение (только простые)
- Двоичные операторы
- Встроенные функции (sqrt, pow, powi, sin, cos, log, los2, log10, exp, exp2, fma)
- Приведения (для типов, не являющихся указателями)
- Операции вставки и извлечения элементов

Этап второй:

```
Для каждой (пары команд-кандидатов) {
        Для каждой (пара команд кандидатов преемников) {
            Если (обе инструкции во второй паре используют некоторый
                результат из первой пары)
                Записать парное соединение;
        }
}
```

Пара кандидатов преемников – это пара, в которой первая инструкция во второй паре является преемницей первой инструкции в первой паре.

Этап третий:

```
Для каждого (сопоставимая инструкция, которая является частью оставшейся
пары кандидатов) {
        Лучшее дерево = null;
        Для каждого (пары кандидатов, членом которой является данная
            инструкция) {
            Если (эта пара кандидат конфликтует с уже выбранной парой) {
                Продолжить;
            }
            Построение и обрезка дерева с этой парой корней (и, если есть
            возможность, сделать его лучшим деревом);
```

}

Если (лучшее дерево имеет необходимый размер и глубину) {

Удалить из пар кандидатов все пары, не входящие в лучшее дерево, которые разделяют инструкции с парами в лучшем дереве;

Добавить все пары в лучшем дереве в список выбранных пар;

}

}

Строится и обрезается дерево, используя эту пару в качестве корня: {

Строится дерево из всех пар, подключенных к этой паре (транзитивное замыкание);

Обрезается дерево, удалив конфликтующие пары (отдавая предпочтение парам с самыми глубокими дочерними элементами);

Если (дерево имеет требуемую глубину и больше пар, чем лучшее дерево) {

Лучшее дерево = это дерево;

}

}

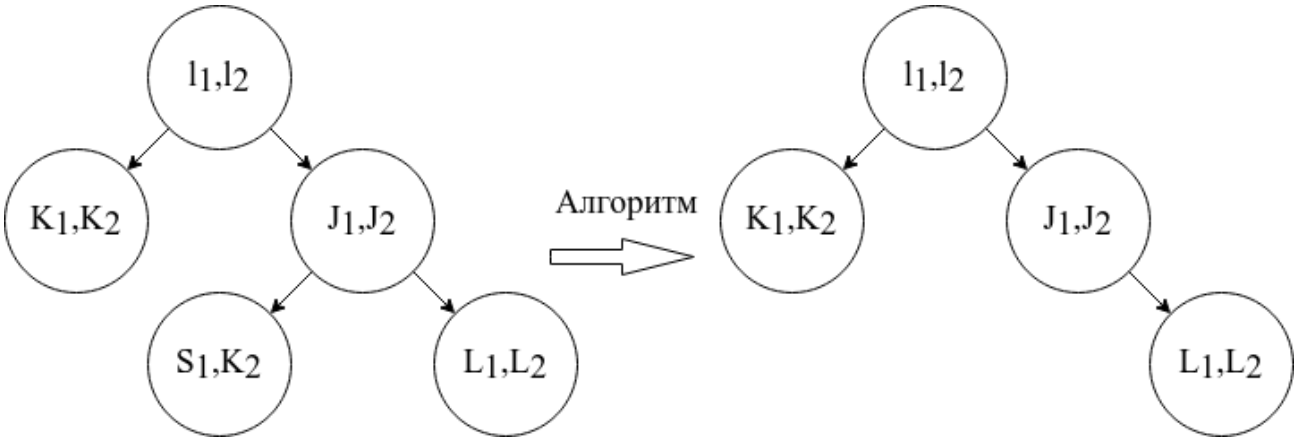


Рисунок 12 – Пример работы алгоритма.

Этап четвертый:

Для каждого (инструкция в оставшейся выбранной паре) {

Формируются входные операнды (обычно используя вставку элемента или перемешивание векторов);

Копируется первая инструкция, изменяется ее тип и заменяются ее операнды;

Формируются выходные данные замены (обычно используются изъятие элемента или перемешивание векторов);

Перемещаются все варианты использования первой инструкции после второй;

Вставляется новая векторная инструкция после второй инструкции;

Заменяются оригинальные инструкций полученными;

Удаление исходных инструкций;

Удаление пары команд из списка оставшихся выбранных пар;

}

Примечание: если мы векторизуем вычисления адресов, то анализ указателя может возвращать разные значения по мере продолжения процесса объединения. В результате все необходимые запросы анализа указателя должны быть кэшированы до начала объединения команд.

Что поддерживает SLP – векторизация:

- Параллелизм на уровне суперслов, определяется как короткий SIMD – параллелизм, при котором исходный и результирующий операнды упаковываются в хранилище.
- Объединение нескольких скалярных операций в одну векторную
- Уменьшение размера кода и использования регистров

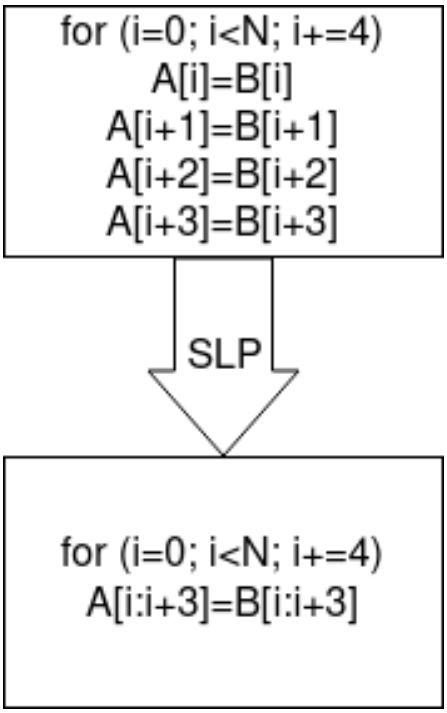


Рисунок 13 – Пример SLP векторизации.

SLP-векторизатор выполняет автоматическую векторизацию прямолинейного кода. Он работает путем сканирования кода в поисках начальных скалярных инструкций, которые можно сгруппировать, а затем замены каждой группы ее векторизованной формой. Инструкции (хранилища), обращающиеся к соседним ячейкам памяти или инструкции, вводимые в дерево сокращений. Инструкции, которые обращаются к смежной памяти, являются одними из наиболее многообещающих исходных данных, поэтому большинство компиляторов ищут их первыми [6]. Когда инструкции найдены, они группируются и помечаются как кандидаты для векторизации. Данная группа становиться корнем графа, в котором находятся все группы-кандидаты скалярных инструкций. Каждый узел графа SLP, представляет собой группу векторизуемых скалярных инструкций. Затем SLP продвигается вверх по цепочкам, пытаясь сформировать больше групп изоморфных инструкций. Данный процесс происходит до тех пор, пока не достигнет неизоморфных инструкций или пока инструкции не станут недопустимыми для векторизации. Каждый узел, помимо скалярных инструкций, содержит в себе некоторые дополнительные данные, такие как стоимость группы. Как только алгоритм сталкивается со скалярными инструкциями, которые не получается сформировать, векторизуемая группа формирует конечную не векторизуемую группу, которая содержит затраты на сбор данных из скаляров и вставку их в вектор. Во время следующего шага происходит оценка, улучшит ли производительность преобразование полученных групп графов SLP в векторный инструкции. Этот расчет учитывает накладные расходы на вставку данных в векторные регистры и на обратную операцию. Если это выгодно, компилятор планирует код и обновляет промежуточное представление (IR), заменяя скалярные инструкции на векторные и выдавая все необходимые инструкции для передачи любых требуемых данных между векторами и скалярами вне графика. Если преобразование невыгодно, то код остается прежним. Ниже приведен пример построения дерева векторизации.

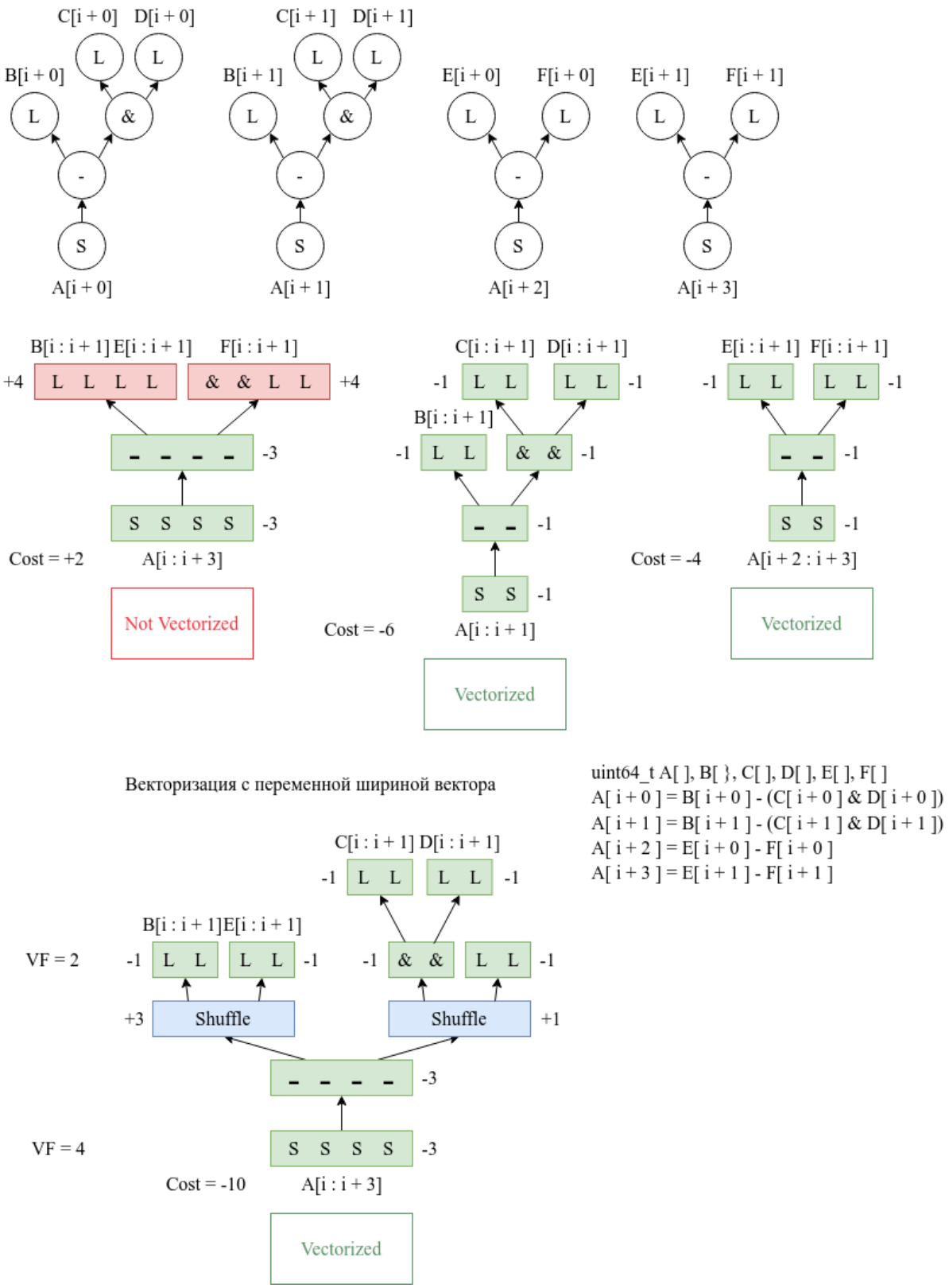


Рисунок 14 – Пример работы SLP – векторизатора

2.5 Обзор GCC

GCC является частью проекта GNU, который направлен на улучшение компилятора, который используется в системе GNU (включая GNU/Linux). Работа по разработке GCC ведется открыто и ее поддерживает множество других платформ. GCC работает на множестве архитектур и в различных средах.

Структура GCC:

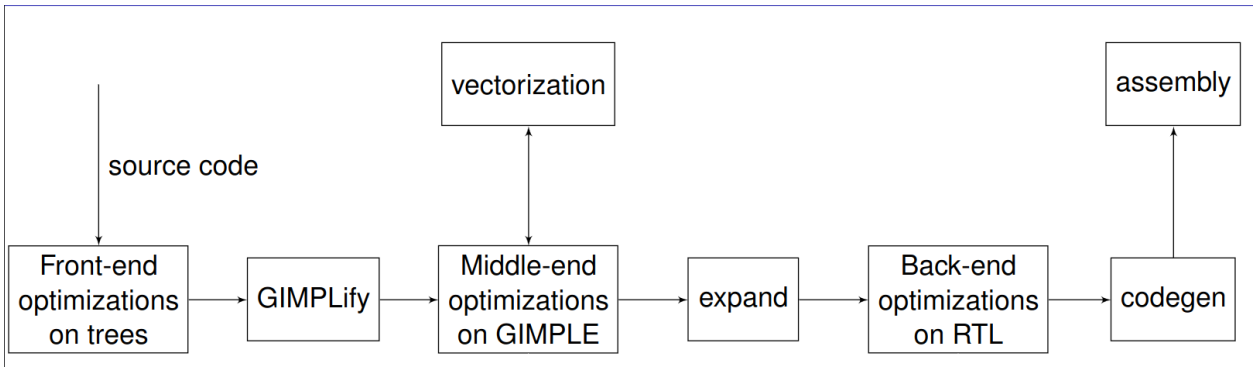


Рисунок 15 – Структура работы GCC [5]

2.6 Векторизация в GCC

Этап векторизаци представлен в оптимизации вложенных циклов (LNO), на IR уровне деревьев GIMPLE, обработанных SSA. Вызывается анализатор скалярных эволюций, который возвращает SCEV. SCEV позволяет представлять переменный со сложным поведением простыми и непротиворечивым способом. SCEV анализа кеширует результаты для экономии времени и памяти, однако этот кэш становится недействительным в результате большинства преобразований цикла, включая удаление кода.

2.7 Дизайн векторизатора GCC.

Схема прохождения векторизации представлена на рисунке 16. Векторизатор применяет набор анализов к каждому циклу, за которым следует фактическое векторное преобразование циклов, которые прошли проверку.

2.8 Анализ перед векторизацией GCC

Начальная фаза анализа, analyze_loop_form(), проверяет условие выхода из цикла и количество итераций, а также некоторые атрибуты потока управления, такие как количество базовых блоков и уровень вложенности.

Алгоритмы потока управления

Важным аспектом при изучении анализатора потока данных является базовый язык, который представляет инструкции, которые либо изменяют данные, либо определяют поток управления программой. В общем, существует два основных представления потока управления, на основе которых строятся анализаторы потока данных:

- Граф потока управления (CFG) состоит из базовых блоков, которые содержат инструкции, выполняемые от первого до последнего. Они связаны ребрами, которые представляют поток управления;
- дерево иерархии циклов (LHT) основано на графе потока управления и хранит, для каждого цикла набор базовых блоков, составляющих тело цикла, а также отношение включения над циклами, которое связывает цикл с его отцом (т. е. заключающим циклом) в гнездовом цикле.

Промежуточное представление LHT содержит результаты анализа потока управления, который обнаруживает сильно связанные компоненты (SCC) на CFG (эти SCC известны как естественные циклы) [7, 8]. Представление LHT хранит связь между циклами CFG, но при создании LHT структура не содержит другой информации, относящейся к данным. Другие анализаторы должны выводить такую информацию.

Одно из основных ограничений, накладываемых на цикл для применимости векторизации, заключается в том, что цикл является счетным – т. е. выражение, вычисляющее границу цикла, может быть построено и оценено во время компиляции или во время выполнения. Например, цикл:

```
while (*p != NULL){
*p++ = X;
}
```

```
vect_analyze_loop (struct loop *loop) {
  loop_vec_info loopinfo;
  loop_vinfo = vect_analyze_loop_form (loop);
  if (!loop_vinfo)
    FAIL;
  if (!analyze_data_refs (loopinfo))
    FAIL;
  if (!analyze_scalar_cycles (loopinfo))
    FAIL;
  if (!analyze_data_ref_dependences (loopinfo))
    FAIL;
  if (!analyze_data_ref_accesses (loopinfo))
    FAIL;
  if (!analyze_data_refs_alignment (loopinfo))
    FAIL;
  if (!analyze_operations (loopinfo))
    FAIL;
  LOOP_VINFO_VECTORIZABLE_P (loopinfo) = 1;
  return loopinfo;
}

vect_transform_loop (struct loop *loop) {
  FOR_ALL_STMTS_IN_LOOP(loop, stmt)
    vect_transform_stmt (stmt);
  vect_transform_loop_bound (loop);
}
```

Рисунок 16 – Цикл анализа [6]

Не является счетным. Анализ привязки к циклу выполняется с помощью скалярного эволюционного анализатора. Векторизатор также проверяет, что цикл является самым внутренним циклом и состоит из одного базового блока. Конструкции с несколькими базовыми блоками, такие как if, then, else, сворачиваются в условные операции, если это возможно, путем преобразований if пред векторизацией. Далее analyze_data_refs(), анализатор находит все ссылки

на память в цикле и проверяет, являются ли они анализируемыми – т. е. можно создать функцию доступа, которая описывает модификацию в цикле. Это нужно для анализа зависимости от данных. Также анализатор analyze_scalar_cycles(), проверяет «скалярные циклы» (циклы зависимости, в которых присутствуют только скалярные переменные) и проверяет, что любой скалярный цикл может быть обработан так, чтобы нарушить зависимость от перекрестных итераций. Один из видов таких «разрушаемых» скалярных циклов являются те, которые представляют собой сокращения. Операция сокращения вычисляет результат из набора элементов данных [6]. Например цикл:

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

В данном цикле вычисляется сумма набора элементов массива в скалярную итоговую сумму. Операции сокращения могут быть векторизованы, путем параллельного вычисления нескольких частичных результатов и объединения их в конце. Скалярные циклы так же могут быть созданы с помощью индукционных переменных (IV). Некоторые IV используются для управления циклом и для вычисления адреса, они обрабатываются как часть векторизации. Заключительная фаза анализа analyze_operations(), сканирует все операции в цикле и определяет коэффициент векторизации (VF), который представляет количество элементов в данных, которые будут упакованы вместе в вектор, а также является коэффициентом извлечения данных из цикла. Он выбирается в зависимости от типов данных, с которыми работает цикл, и длиной векторов, которая поддерживает целевая платформа [6].

2.9 Преобразования во время векторизации GCC

Во время этапа анализа векторизатор записывает информацию на трех уровнях

- На уровне цикла;
- на уровне инструкций;

- на уровне памяти (ссылка на память)

Эти структуры данные используются на этапе преобразования цикла. Преобразование происходит следующим образом: удаление с помощью VF и замена один к одному, что подразумевает, что каждая скалярная операция будет заменена на векторную. На этапе преобразования цикла сканируются все операторы цикла сверху вниз, вставляя векторный оператор VS в цикл для каждого скалярного оператора S, и записывает прикрепленный к S указатель на VS. Он нужен чтобы определять местоположение скалярной операции. После того, как все инструкции будут векторизованы, исходные скалярные инструкции могут быть удалены. Для других вычислений требуется специальный код эпилога после цикла, с помощью него преобразуется конец цикла. Конец цикла преобразуется, чтобы отразить новое число итераций, и при необходимости создает скалярный цикл эпилога, для обработки пограничных случаев, когда количество итераций не делится на коэффициент векторизации (VF) [6].

2.10 Обработка ссылок на память GCC

При работе со ссылками на память требуется повышенное внимание при векторизации. Векторизатор рассматривает две формы ссылок на данные – одномерные массивы и обращения к указателям. Как только функция доступа была вычислена (для индекса массива или указателя), векторизатор приступает к применению набора анализов ссылок на данные, которые описаны ниже.

2.11 Зависимости и наложение псевдонимов GCC

Одно из основных ограничений, которое нужно соблюдать для безопасной векторизации, является отсутствие циклов зависимостей (отсутствие связи между двумя или более модулями, которые прямо или косвенно зависят друг от друга). Применяется упрощенная анализа зависимостей с использование простых тестов зависимости. Обращение к указателю требует дополни-

тельного анализа псевдонимов (проверка того, что разные переменные обращаются к одному и тому же распределению памяти), чтобы понять могут ли два обращения к указателю в цикле быть псевдонимами (указывать на одно и тоже распределение памяти).

2.12 Схема доступа GCC

Когда данные размещены в памяти как это нужно для вычислений, их можно векторизовать, используя простую схему векторизации «один к одному». Но не всегда вычисления обращаются к элементам так, как они организованы в памяти. Шаблоны непоследовательного доступа обычно требуют специальных манипуляций с данными, чтобы повторно упорядочить элементы данных и упаковать их в векторы. Это нужно, потому что архитектура памяти ограничивает доступ к векторным данным последовательными элементами векторного размера. Расширения SIMD обычно предоставляют механизмы для упаковки данных из двух векторов в один, при этом возможна перестановка элементов данных. Реорганизация данных определяет, можно ли применить векторизацию и по какой цене. Эти манипуляции с данными нужно применять на каждой итерации цикла и требуют значительных накладных расходов. Поэтому некоторые шаблоны доступа (например, косвенный доступ) не могут быть эффективно векторизованы на большинстве SIMD/векторных архитектур. Анализатор проверяет, что шаблон доступа ко всем ссылкам на данные в цикле поддерживается векторизатором.

2.13 Выравнивание данных GCC

Доступ к блоку памяти, в котором данные не выравненны по границе размера вектора, часто запрещен или приводит к значительному снижению производительности. Данную проблему решают с помощью механизмов переупорядочения данных. Такие механизмы обычно являются дорогостоящими, чтобы избежать этого, используются такие методы, как очистка цикла и определение статического и динамического выравнивания. Таким образом обработка выравнивания состоит из трех уровней:

1. Статический анализ выравнивания.

2. Преобразования для принудительного выравнивания.

3. Эффективная векторизация оставшихся несогласованных обращений.

2.14 Вывод

В данном разделе были рассмотрены методы автоматической векторизации в современных компиляторах. Рассмотрен цикл векторизации в компиляторе LLVM/Clang и нововведение в виде плана векторизации. Также был рассмотрен цикл векторизации в GCC.

3 Тестирование автоматической векторизации

В данном разделе будет произведено тестирование автоматической векторизации в компиляторах LLVM/Clang и GCC.

3.1 Результаты тестирования

Тестирование проводилось на тестовом наборе ETSVC.

Таблица 2 – Тестирование векторизации в LLVM/Clang

Работа цикла (без векторизации)			Работа цикла (векторизация)		
Loop	Time(sec)	Checksum	Loop	Time(sec)	Checksum
s000	1,329	512066944	s000	0,49	512066944
s111	1,069	32000,41016	s111	0,836	32000,41016
s1111	2,942	16005,82227	s1111	0,97	16005,82227
s112	2,93	84644,89844	s112	2,908	84644,89844
s1112	2,057	32001,64063	s1112	0,757	32001,64063
s113	3,044	32000,64063	s113	1,347	32000,64063
s233	9,558	504920,1875	s233	9,483	504920,1875
s2233	5,144	337652,7188	s2233	4,569	337652,7188
s235	8,934	160024,0469	s235	8,922	160024,0469
s241	4,068	64000	s241	4,066	64000
s242	2,861	1535966208	s242	2,852	1535966208
s243	2,491	810659,4375	s243	0,89	810659,4375
s311	28,551	10,950721	s311	28,47	10,950721
s31111	1,039	10,950721	s31111	1,039	10,950721
s312	28,573	1,030518	s312	28,467	1,030518
s313	14,311	1,644725	s313	14,244	1,644725
s314	14,268	1	s314	14,232	1
s315	2,859	54857	s315	2,852	54857
s421	3,999	32009,03125	s421	1,614	32009,02344
s1421	4,064	16000	s1421	1,653	16000
s422	12,052	257,701355	s422	12,503	257,660736
s423	5,921	439,68158	s423	6,276	439,621765
s424	4,027	822,355896	s424	4,135	822,364014

Продолжение таблицы 2

s431	9,927	1674250,25	s431	4,176	1674287,625
va	8,171	1,644725	va	2,603	1,644725
vag	2,332	1,644725	vag	2,336	1,644725
vas	2,452	1,644725	vas	3,115	1,644725
vif	1,155	1,644725	vif	1,196	1,644725
vpv	11,228	1642244,625	vpv	3,864	1642244,625
vtv	11,137	32000	vtv	3,837	32000

В таблице выше представлены результаты тестирования автоматической векторизации с флагами –O3 и –fvectorize, –fslp-vectorize. Флаг –O3 данный флаг разрешает умеренный уровень оптимизации, который позволяет выполнять большинство оптимизаций, а также он позволяет выполнять оптимизацию, выполнение которой занимает больше времени или которая может генерировать код большего размера (в попытке ускорить работу программы). –fvectorize – включает циклические проходы векторизации. –fslp-vectorize – включает проходы векторизации параллелизма на уровне суперслов.

```
real_t s1111(struct args_t * func_args)

{
//    no dependence - vectorizable
//    jump in data access
    initialise_arrays(__func__);
    gettimeofday(&func_args->t1, NULL);
    for (int nl = 0; nl < 2*iterations; nl++) {
        for (int i = 0; i < LEN_1D/2; i++) {
            a[2*i] = c[i] * b[i] + d[i] * b[i] + c[i] * c[i] + d[i] * b[i] + d[i] * c[i];
        }
        dummy(a, b, c, d, e, aa, bb, cc, 0.);
    }
    gettimeofday(&func_args->t2, NULL);
    return calc_checksum(__func__);
}
```

Автовекторизация смогла векторизовать данный цикл таким образом, что время уменьшилось больше чем в два раза. Данный цикл легко был векторизован из-за того, что отсутствует зависимость, а также происходит скачок в доступе к данным. Рассмотрим следующий цикл:

```
real_t s233(struct args_t * func_args)
```

```
{
//   loop interchange
//   interchanging with one of two inner loops
   initialise_arrays(__func__);
   gettimeofday(&func_args->t1, NULL);
   for (int nl = 0; nl < 100*(iterations/LEN_2D); nl++) {
      for (int i = 1; i < LEN_2D; i++) {
         for (int j = 1; j < LEN_2D; j++) {
            aa[j][i] = aa[j-1][i] + cc[j][i];
         }
         for (int j = 1; j < LEN_2D; j++) {
            bb[j][i] = bb[j][i-1] + cc[j][i];
         }
      }
      dummy(a, b, c, d, e, aa, bb, cc, 0.);
   }
   gettimeofday(&func_args->t2, NULL);
   return calc_checksum(__func__);
}
```

Данный цикл не был векторизован, так как векторизатор не смог заменить один из двух внутренних циклов. Рассмотрим следующий цикл:

```
real_t s311(struct args_t * func_args)
{
//   reductions
//   sum reduction
   initialise_arrays(__func__);
   gettimeofday(&func_args->t1, NULL);
   real_t sum;
   for (int nl = 0; nl < iterations*10; nl++) {
      sum = (real_t)0.;
      for (int i = 0; i < LEN_1D; i++) {
         sum += a[i];
      }
      dummy(a, b, c, d, e, aa, bb, cc, sum);
   }
   gettimeofday(&func_args->t2, NULL);
   return calc_checksum(__func__);
}
```

После векторизации удалось сократить время работы на 0.1 секунду. В следующем цикле:

```
real_t s421(struct args_t * func_args)
{
//   storage classes and equivalencing
//   equivalence- no overlap
   initialise_arrays(__func__);
   gettimeofday(&func_args->t1, NULL);
   xx = flat_2d_array;
   for (int nl = 0; nl < 4*iterations; nl++) {
      yy = xx;
```

```
        for (int i = 0; i < LEN_1D - 1; i++) {
            xx[i] = yy[i+1] + a[i];
        }
        dummy(a, b, c, d, e, aa, bb, cc, 1.);
    }
    gettimeofday(&func_args->t2, NULL);
    return calc_checksum(__func__);
}
```

Данный цикл был векторизован, векторизатор смог определить лишние конструкции присваивания и оптимизировать их, за счет чего получилось сократить время работы почти в 3 раза.

Таблица 3 – Тестирование векторизации в GCC

Работа цикла (без векторизации)			Работа цикла (векторизация)		
Loop	Time(sec)	Checksum	Loop	Time(sec)	Checksum
s000	1,982	512066944	s000	0,575	512066944
s111	1,159	32000,41016	s111	0,87	32000,41016
s1111	2,915	16005,82227	s1111	0,992	16005,82227
s112	3,4	84644,89844	s112	1,511	84644,89844
s1112	2,994	32001,64063	s1112	1,242	32001,64063
s113	4,005	32000,64063	s113	1,339	32000,64063
s233	9,412	504920,1875	s233	9,391	504920,1875
s2233	5,163	337652,7188	s2233	4,556	337652,7188
s235	8,896	160024,0469	s235	2,516	160024,0469
s241	4,149	64000	s241	4,171	64000
s242	2,853	1535966208	s242	2,852	1535966208
s243	2,486	810659,4375	s243	2,487	810659,4375
s311	28,483	10,950721	s311	28,475	10,950721
s31111	0,992	10,950721	s31111	1,023	10,950721
s312	28,482	1,030518	s312	28,519	1,030518
s313	14,253	1,644725	s313	14,238	1,644725
s314	14,245	1	s314	14,241	1
s315	2,853	54857	s315	2,853	54857
s422	8,28	257,660736	s422	3,016	257,660736
s423	4,09	439,621765	s423	1,595	439,621765
s424	4,064	822,364014	s424	1,557	822,364014
s431	11,19	1674287,625	s431	3,858	1674287,625
s441	2,202	196500,2656	s441	2,172	196500,2656
s442	1,021	114244,6328	s442	0,975	114244,6328
va	2,005	1,644725	va	2,003	1,644725
vag	2,317	1,644725	vag	2,323	1,644725

Продолжение таблицы 3

vas	2,41	1,644725	vas	2,677	1,644725
vif	0,977	1,644725	vif	0,944	1,644725
vpv	9,989	1642244,625	vpv	3,977	1642244,625
vtv	9,963	32000	vtv	3,945	32000

3.1 Вывод

Если сравнить результаты, получившиеся после тестов, то можно сделать вывод, что автоматические векторизаторы используют общие идеи оптимизации кода. Так и есть, ведь разработчики LLVM/Clang разрабатывая автоматический векторизатор, ориентировались на GCC. Самыми проблемными оказались циклы с условными и безусловными переходами, а также рекурентные 1 и 2 порядка.

4 Обеспечение качества разработки продукции, программного продукта

В данном разделе будет проводиться обеспечение качества разработки продукции, программного продукта данной работы.

4.1 Определение групп потребителей

Векторизация (или автовекторизация) является оптимизацией, которая позволяет автоматически использовать векторные инструкции процессора для эффективного выполнения циклов и других операций в программе. В компиляторах C/C++ функцию векторизации могут использовать разработчики программного обеспечения, которые хотят оптимизировать свой код для более эффективного использования аппаратных ресурсов.

Одним из возможных потребителей могут быть программисты научных вычислений. Эта группа включает исследователей, ученых и инженеров, которые занимаются численными методами, моделированием и анализом данных. Они используют векторизацию кода для оптимизации производительности своих вычислительных алгоритмов, которые часто включают матричные операции, решение дифференциальных уравнений и другие сложные вычисления.

Также к потребителям можно отнести разработчиков графических приложений. Данная группа разработчиков работает над компьютерными играми, компьютерной графикой, виртуальной реальностью и другими приложениями, связанными с обработкой изображений и графикой. Векторизация кода используется для оптимизации алгоритмов рендеринга, обработки изображения, физической симуляции и других графических операций.

4.2 Функции анализа эффективности векторизации кода в современных компиляторах.

Диагностика векторизации: компиляторы могут выводить диагностические сообщения, которые указывают, была ли применена векторизация для конкретного цикла или операции, и какие причины препятствовали векторизации.

Анализ зависимостей данных: компиляторы проводят анализ зависимостей данных в циклах и операциях, чтобы определить, возможна ли векторизация без нарушения правильности вычислений. Если есть зависимости, которые не могут быть разрешены, компилятор может отказаться от векторизации или предложить варианты для их устранения.

Расчет эффективности векторизации: некоторые компиляторы предоставляют информацию о том, насколько эффективно была векторизованная определенная часть кода. Это может включать информацию о количестве векторных инструкций, использовании регистров SIMD, проценте загрузки функциональных блоков процессора и других метриках производительности.

Оптимизационные флаги: компиляторы предоставляют различные оптимизационные флаги, которые могут включать или выключать векторизацию в целом или для конкретных оптимизаций. Разработчики могут использовать эти флаги для настройки векторизации в соответствии с требованиями.

4.3 Качество и характеристики продукта.

Таблица 4 – Анализ функциональных требований к разработке.

Функции по группам	**Источник**	**Требования**
Эффективность	ГОСТ Р ИСО/МЭК 25010-2015	С помощью анализа и замеров времени получение информации о том, насколько эффективно была векторизованная определенная часть кода.
Результативность	ГОСТ Р ИСО/МЭК 25010-2015	Компиляторы должны выводить диагностические сообщения, которые указывают, была ли применена векторизация для конкретного цикла или операции, и какие причины препятствовали векторизации.
Полноценность	ГОСТ Р ИСО/МЭК 25010-2015	Компиляторы должны предоставлять различные оптимизационные флаги, которые могут включать или выключать векторизацию в целом или для конкретных оптимизаций.
Надежность	ГОСТ Р ИСО/МЭК 25010-2015	Компиляторы должны проводить анализ зависимостей данных в циклах и операциях.

4.4 Измерение характеристик качества. Операциональное определение.

На основе функциональных требований необходимо выделить характеристики для оценки качества программного продукта. Должны быть сформулированы точные способы измерения данных характеристик, для обеспечения возможности однозначно оценить соответствие продукта требованиям.

Операциональное определение используется для точного измерения и описания явлений, процессов или объектов путем определения конкретных операций или процедур. Такой подход обеспечивает ясность, согласованность и возможность сравнения в различных условиях и контекстах, что в свою очередь важно для контроля и улучшения качества продуктов и услуг.

4.5 Предложения по улучшению

Предложения по улучшению продукта представлены в таблице 4.

В процессе исследования были выявлены требования потребителей к разработке, преобразованы данные требования в характеристики качества разработки и установлены целевые значения характеристик качества.

Исходя из того, что все перечисленные выше критерии почти выполняются в проведенном исследовании, можно сделать вывод о том, что компиляторы могут использоваться на практике с автоматической оптимизацией, но за их работой все еще нужно следить и, иногда вносить изменения или исправлять ошибки. А если компилятору предоставить контекстную информацию, то его можно будет сильно улучшить.

Таблица 5 – Операционное определение.

Требование	**Тест**		**Решение (правило)**		
	Характеристика	**Метод измерения**	**Соответствие**	**Частичное соответствие**	**Несоответствие**
С помощью анализа и замеров времени получение информации о том, насколько эффективно была векторизованная определенная часть кода.	Время работы определенного участка кода	Замер времени в секундах на работу оптимизированной функции	Время работы 1.5 секунды и меньше.	Время работы 2 секунды и меньше.	Время работы больше 2-ух секунд.
Компиляторы должны выводить диагностические сообщения, которые указывают, была ли применена векторизация для конкретного цикла или операции, и какие причины препятствовали векторизации.	Наличие сообщений о применении векторизации	Проверка подробного диагностического сообщения	Все характеристики нужные при оптимизации представлены	Отсутствуют некоторые характеристики	Диагностическое сообщение отсутствует
Компиляторы должны предоставлять различные оптимизационные флаги, которые могут включать или выключать векторизацию в целом или для конкретных оптимизаций.	Наличие оптимизационных флагов	Проверка кода после включения флагов	Флаги оптимизации векторизуют код в соответствии с заданной оптимизацией	Флаги оптимизации векторизуют код только в определенных случаях	Флаги оптимизации не влияют на векторизацию
Компиляторы должны проводить анализ зависимостей данных в циклах, чтобы определить, возможна ли векторизация без нарушения правильности вычислений.	Количество ошибок работы после оптимизации	Проверка точности вычислений после работы программы	Вычисления выполнены с высокой точностью	Вычисления выполнены с точностью до 2-ух знаков после запятой	Вычисления выполнены с ошибками

Таблица 6 – Предложения по улучшению продукта.

Требование	**Анализ текущего состояния**	**Возможные причины невыполнения критерия**	**Предложения по улучшению**
С помощью анализа и замеров времени получение информации о том, насколько эффективно была векторизованная определенная часть кода.	В настоящий момент компиляторы не могут оптимизировать или оптимизируют не максимально эффективно сложные участки кода	В первую очередь это связано с отсутствием контекстной информации, которую компиляторы могут извлечь или воспринять из кода.	Предоставлять компиляторам контекстную информацию.

ЗАКЛЮЧЕНИЕ

Произведен анализ векторизации в компиляторах LLVM/Clang и GCC. Рассмотрены эвристики, которые используются для векторизации базовых блоков (SLP – векторизация), векторизация внутренних циклов и также разобран алгоритм автоматической векторизации базового блока. Был рассмотрен так называемый «VPlan», который находится на данный момент в процессе разработки. Также был рассмотрен цикл анализа кода в компиляторе GCC, с реализованными методами векторизации. В разделе тестирования можно пронаблюдать, то, что в компиляторах используются схожие идеи, которые используются во время векторизации кода. Установлено, что компилятор способен векторизовать от 40 до 70% циклов в представленном наборе ETSVC. Самыми проблемными циклами оказались

Но несмотря на все описанное выше, рассмотрев циклы в разделе тестов, можно сделать вывод, что компиляторы могут векторизовать не все циклы. Самыми проблемными оказались циклы с условными и безусловными переходами, а также рекуррентности 1 и 2 порядка. В случаях, когда компилятор не может решить будет ли эффективен код он генерирует несколько версий (например, скалярную и векторную). После этого происходит решение о том какая версия лучше – это происходит во время выполнения кода.

Направление последующих исследований будет связанно с анализом реализаций других оптимизаций в компиляторах, их тестированию и рекомендациям, которые будут улучшать их работу.

СПИСОК ЛИТЕРАТУРЫ

1. MIT 6.172 Performance Engineering of Software Systems, Fall 2018

2. Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11). IEEE Computer Society, Washington, 372–382.

3. Callahan D., Dongarra J., Levine D. Vectorizing Compilers: A Test Suite and Results // Proc. Of the ACM/IEEE conference on Supercomputing (Supercomputing'88), 1988. Pp. 98–105.

4. [Электронный ресурс] A performance-Based Compraison of C/C++ Compilers: https://colfaxresearch.com/compiler-comparison/

5. A. Zaks and D. Nuzman. Autovectorization in GCC-two years later.

6. Autovectorization in GCC. Dorit Naishlos IBM Research Lab in Haifa

7. The First Annual GCC Developers' Summit. http://www.gccsummit.org/2003/

8. A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques and Tools. AddisonWesley, 1986.

9. Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2013. The LINPACK Benchmark: past, present and future. Concurrency and Computation: Practice and Experience 15, 9 (2013), 803–820.

10. Fog, Agner. — The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers [Text], 2010. — http://www.agner.org/optimize/microarchitecture.pdf.

11. I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware SLP in GCC," in GCC Developers Summit, 2007.

12. Jibaja I., Jensen P., Hu N., Haghighat M., McCutchan J., Gohman D., Blackburn S., McKinley K. Vector Parallelism in JavaS- cript: Language and Compiler Support for SIMD // Proc. Of the International Conference on Parallel Architecture and Compila- tion (PACT-2015). 2015. Pp. 407–418.

13. Vektorizatsiya programm: teoriya, metody, realizatsiya. Sb. statei: Per. s angl. i nem. [Program Vectorization: Theory, Methods, Implementation. Coll. Works: Transl. from Eng. And Germ]. Moscow, Mir Publ., 1991, 275 p.