Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем
Допустить к защите
зав. кафедрой д.т.н. доцент
_____ Курносов М.Г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Применение векторизации в задачах поиска в таблицах

Пояснительная записка

Студент: Тимошкин В.Н.

Факультет ИВТ Группа ИС-341

Руководитель к.т.н. доцент Молдованова О.В.

Новосибирск - 2017

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

КАФЕДРА
вычислительных систем

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

СТУДЕНТУ Тимошкину В.Н. ГРУППЫ ИС-341

«УТВЕРЖДАЮ»
«_____» _____
зав. кафедрой ВС
д.т.н. доцент
_____ Курносов М.Г.

Новосибирск, 2017 г.

1. Тема выпускной квалификационной работы бакалавра: «Название бакалаврской работы» утверждена приказом СибГУТИ от «30» января 2017 г. № 4/85о-17

2. Срок сдачи студентом законченной работы: 16 июня 2017 г.

3. Исходные данные к работе

1 Специальная литература

Параллельные вычисления/Под ред. Г. Родрига: Пер. с англ./Под ред.

Ю.Г. Дадаева. – М.: Наука. Гл. ред. Физ.-мат.лит., 1986.-376с.

2 Материалы сети интернет

Курносов Михаил Георгиевич, Лекция 4 Векторизация кода (code vectorization:

SSE/AVX), 2015.

URL: http://www.mkurnosov.net/teaching/uploads/HPC/hpcs- fall2015-lec4.pdf

(дата обращения: 15.03.2017)

General Matrix Multiply Sample, 2013.

URL: https://software.intel.com/sites/products/vcsource/files/GEMM.pdf

(дата обращения: 16.03.2017)

4. Содержание пояснительной записки (перечень подлежащих разработке вопросов)	**Сроки выполнения по разделам**
Изучение возможностей векторных процессоров	14.02.17-20.02.17
Анализ возможностей векторизации различных вариантов алгоритмов	21.02.17-27.02.17
Изучение методики измерения времени выполнения алгоритмов	28.02.17-04.03.17
Изучение возможностей компилятора GNU CC	05.03.17-16.03.17
Изучение возможностей компилятора Intel ICC	17.03.17-27.03.17
Реализация последовательного метода поиска делением интервала поиска пополам	28.03.17-05.04.17
Исследование производительности поиска методом поиска делением интервала поиска пополам	06.04.17-18.04.17
Исследование возможностей автовекторизации компиляторов GNU GCC C/C++, Intel ICC на примере метода поиска делением интервала поиска пополам. Сравнение результатов	19.04.17-05.05.17
Реализация поиска методом линейного векторного поиска	6.05.17-10.05.17

Анализ результатов векторизованного поиска метода поиска делением интервала поиска пополам	11.05.17-20.05.17
Исследование возможностей автовекторизации компиляторов GNU GCC C/C++, Intel ICC на примере метода линейного векторного поиска. Сравнение результатов	21.05.17-24.05.17
Исследование производительности поиска методом линейного векторного поиска	25.05.17-26.05.17
Реализация поиска делением интервала поиска на M частей	27.05.17-31.05.17
Анализ результатов векторизованного поиска делением интервала поиска на M частей	01.06.17-02.06.17
Исследование возможностей автовекторизации компиляторов GNU GCC C/C++, Intel ICC на примере поиска делением интервала поиска на M частей. Сравнение результатов	03.06.17-04.06.17
Исследование производительности метода поиска делением интервала поиска на M частей	05.06.17-06.06.17

Дата выдачи задания: «_____» _____

Руководитель _____ Молдованова О.В.

Задание принял к исполнению «_____» _____

Студент _____ Тимошкин В.Н.

АННОТАЦИЯ

Выпускная квалификационная работа Тимошкина В.Н.
по теме «Применение векторизации в задачах поиска в таблицах»

Объём работы 73 страницы, на которых размещены 15 рисунков и 17 таблиц. При написании работы использовалось 24 источника.

Ключевые слова: вычислительная система, трансляционные обмены.

Работа выполнена на кафедре ВС СибГУТИ.
Руководитель –к.т.н. доцент Молдованова О.В.,

Целью исследования являлось изучение возможностей векторизации алгоритмов поиска в одномерных таблицах.

В рамках бакалаврской работы были реализованы скалярные и векторизованные версии следующих методов поиска в таблице:

- Метод поиска делением интервала поиска пополам
- Метод линейного векторного поиска
- Метод поиска делением интервала поиска на M частей

Векторизованные версии были реализованы на языке Си с использованием функций-интринсиков. Также был проведен ряд экспериментов для анализа ускорения при векторизации методов поиска в одномерных таблицах. Исследована возможность автоматической векторизации с помощью компиляторов Intel® C++ Compiler, GNU GCC C/C++, LLVM/Clang.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

ОТЗЫВ

на выпускную квалификационную работу студента
группы ИС-341 Тимошкина В.Н.
по теме «Применение векторизации в задачах поиска в таблицах»

Поиск в таблицах является распространенной задачей. Под операцией поиска в таблице понимают отыскание места вхождения заданного значения в упорядоченный массив, или таблицу.

При разработке алгоритмов для векторного процессора учитываются возможности ускоренного выполнения привычных математических операций, но табличный поиск сохраняет все трудности реализации на обычном процессоре. Суть проблемы состоит в том, что табличный поиск связан с принятием решения о требуемом действии, а не с выполнением действия над элементами массива.

В ходе выполнения бакалаврской работы студент Тимошкин В.Н. реализовал последовательные и векторизованные версии алгоритмов поиска в таблицах на языке Си с использованием функций-интринсиков, провел эксперименты и проанализировал ускорение при векторизации методов поиска в одномерных таблицах. Им также были исследованы возможности автоматической векторизации методов поиска при помощи компиляторов Intel C/C++ Compiler, GNU GCC C/C++, LLVM/Clang.

Считаю, что бакалаврская работа студента Тимошкина В.Н. заслуживает оценки «**отлично**», а сам Тимошкин В.Н. – присвоения квалификации «бакалавр» по направлению 02.03.02 «Фундаментальная информатика и информационные технологии».

Оценка уровней сформированности общекультурных и профессиональных компетенций обучающегося:

Компетенции		Уровень сформированности компетенций		
		Высокий	Средний	Низкий
Общекультурные	ОК-7 - способностью к самоорганизации и самообразованию	+		

Общепрофессиональные	ОПК-3 - способностью к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей, созданию информационных ресурсов глобальных сетей, образовательного контента, прикладных баз данных, тестов и средств тестирования систем и средств на соответствие стандартам и исходным требованиям	+		
	ОПК-4 - способностью решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учетом основных требований информационной безопасности	+		
Профессиональные	ПК-1 - способностью собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям	+		
	ПК-2 - способностью понимать, совершенствовать и применять современный математический аппарат, фундаментальные концепции и системные методологии, международные и профессиональные стандарты в области информационных технологий	+		
	ПК-3 - способностью использовать современные инструментальные и вычислительные средства	+		
	ПК-4 - способностью решать задачи профессиональной деятельности в	+		

	составе научно-исследовательского и производственного коллектива			
	ПК-5 - способностью критически переосмысливать накопленный опыт, изменять при необходимости вид и характер своей профессиональной деятельности	+		

Работа имеет практическую ценность	☒	Тема предложена предприятием	☐
Работа внедрена	☐	Тема предложена студентом	☐
Рекомендую работу к внедрению	☐	Тема является фундаментальной	☐
Рекомендую работу к опубликованию	☐	Рекомендую студента в магистратуру	☒
Работа выполнена с применением ЭВМ	☒	Рекомендую студента в аспирантуру	☐

Доцент кафедры вычислительных систем СибГУТИ
к.т.н. доцент _____ Молдованова О.В.
(Молдованова Ольга Владимировна)
«_____» _____

Содержание

1 ВВЕДЕНИЕ

Целью исследования являлось изучение возможностей векторизации алгоритмов поиска в одномерных таблицах.

При решении на векторном процессоре задачи отыскания требуемого элемента в упорядоченной таблице складывается впечатление, что векторный процессор позволяет эффективно выполнять любые задачи, но только не табличный поиск. При разработке алгоритмов для векторного процессора учитываются возможности ускоренного выполнения привычных математических операций, но табличный поиск сохраняет все трудности реализации на обычном процессоре. Суть проблемы состоит в том, что табличный поиск связан с принятием решения о требуемом действии, а не с выполнением действия над элементами массива.

Поиск в таблице является чрезвычайно распространенной операцией в алгоритмах задач современной вычислительной физики. Под операцией поиска в таблице понимают отыскание места вхождения заданного значения в упорядоченный массив, или таблицу. При вычислении кусочно-заданной или табулированной на конечном множестве значений функции почти всегда применяется операция поиска в таблице.

Попытки реализовать архитектурные механизмы для использования параллелизма данных для ускорения вычислений предпринимались еще с появления самых первых суперкомпьютеров. Часто в вычислительных программах одна и та же операция применяется к целому набору данных, обычно это секция массива или массив. В этом случае существует некоторая повторяющаяся модель доступа к данным, а также операции, которые могут выполняться параллельно над элементами из этого набора данных.

Первые конвейерные векторные процессоры поддерживали наборы инструкций, которые выполнялись над векторами данных. Эти данные располагались прямо в памяти. Затем были введены векторные регистры, что сильно ускорило выполнение инструкций, поскольку векторные регистры могли сохранять промежуточные результаты вычислений. Использование векторных архитектурных механизмов может программироваться напрямую. Ну а в том случае, если уже имеется метод поиска, написанный с использованием последовательно выполняющихся вычислений, необходимо модифицировать для использования параллелизма данных.

2 ПОСТАНОВКА ЗАДАЧИ

В данной бакалаврской работе требуется выполнить скалярную и векторизованную реализацию следующих методов поиска в таблице:

- метод поиска делением интервала поиска пополам;
- метод линейного векторного поиска;
- метод поиска делением интервала поиска на M частей.

Векторизация должна быть выполнена вручную на языке Си с использованием функций-интринсиков и автоматически с помощью компиляторов Intel® C++ Compiler, GNU GCC C/C++, LLVM/Clang. Также необходимо провести ряд экспериментов и проанализировать ускорение при векторизации методов поиска в одномерных таблицах.

3 ХАРАКТЕРИСТИКА ПРЕДМЕТНОЙ ОБЛАСТИ

Потребность в решении сложных прикладных задач с большим объемом вычислений и принципиальная ограниченность ресурсов и максимальной производительности «классических» – по схеме фон Неймана – привели к появлению многопроцессорных вычислительных систем (МВС). Широкое распространение параллельные вычисления приобрели с переходом компьютерной индустрии на массовый выпуск многоядерных процессоров с векторными расширениями. В настоящие время практически все устройства оснащены многоядерными процессорами, поддерживающими векторные расширения. Так, при написании последовательной программы, не используя распараллеливания между разными ядрами центрального процессора и не проведя векторизацию, не задействуется большая часть вычислительного потенциала процессора.

Векторизация — вид распараллеливания, при котором однопоточные приложения, выполняющие одну операцию в каждый момент времени, модифицируются для выполнения нескольких однотипных операций одновременно, векторных операций. Одна и та же операция применяется к целому набору данных, например, к массиву или его части.

По одной инструкции векторный процессор обрабатывает сразу некоторый массив(вектор) значений полностью, а не единственное значение, в отличие от скалярного процессора. Для наглядной демонстрации возьмем три массива Arr1, Arr2 и Res, имеющие одинаковую размерность и одинаковую длину, и имеется оператор Res = Arr1 * Arr2.

Векторный процессор за один цикл выполнения команды выполнит попарное умножение элементов массивов Arr1 и Arr2, и присвоит полученные значения соответствующим элементам массива Res. Каждый операнд при этом хранится в векторном регистре. Векторный процессор выполняет лишь одну команду, в отличие от скалярного процессора, которому пришлось бы циклично складывать элементы по очереди.

За счет векторизации возможно получить высокую производительность. Кроме того, векторным ЭВМ присущи и другие интересные особенности. Количество команд, необходимых для выполнения одной и той же программы, использующей векторные инструкции, намного меньше в случае векторного процессора, чем обычного, скалярного. Так уменьшение потока команд позволяет снизить требования к устройствам коммуникации, в том числе между процессором и оперативной памятью компьютера. Другой момент заключается в том, что при соответствующей организации оперативной памяти данные в процессор будут передаваться на каждом такте, что дает значительный выигрыш в производительности компьютера. На рисунке 3.1 приведен пример перемножения двух векторов.

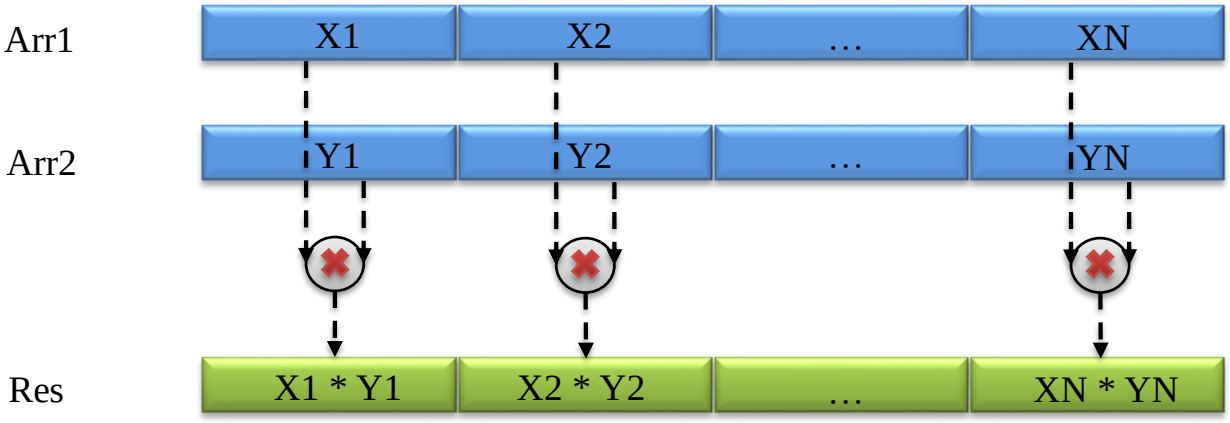

Рисунок 3.1 – Пример перемножения двух векторов

Максимальное ускорение (Speedup) линейно зависит от числа элементов находящихся в векторном регистре. Использование векторных инструкций может привести к сокращению количества команд в программе, а это может обеспечить более эффективное использование кеш-памяти.

Рассмотрим в листинге 3.1 фрагмент программы, который поэлементно перемножает два массива.

Листинг 3.1 – Псевдокод последовательного перемножения двух векторов

```
for (i = 0; i < 1024; i++)
    Res[i] = Arr1[i] * Arr2[i];
```

Данный цикл может быть векторизован, его SIMD версия представлена в листинге 3.2.

Листинг 3.2 – Псевдокод перемножения двух векторов с помощью векторных инструкций

```
for (i = 0; i < 1024; i += 4)
   Res[i:i+3] = Arr1[i:i+3] * Arr2[i:i+3];
```

Запись Res[i:i + 3] означает вектор из 4 элементов — от Res[i] до Res[i + 3] включительно, а под * понимается операция поэлементного умножения векторов. Векторный процессор в данном примере сможет выполнить 4 скалярные операции при помощи одной векторной инструкции за время, близкое к выполнению скалярной операции. Таким образом, векторных операций потребуется в 4 раза меньше, и программа исполнится быстрее.

Преимущество векторизации в том, что за время выполнения одной векторной инструкции операция проводится над массивом данных, но времени затрачивается столько же сколько и на одну скалярную инструкцию. Одна векторная команда распознаётся, декодируется и выполняется быстрее нескольких скалярных, выполняющих тот же набора инструкций. Векторизация позволяет не

только добиться значительного ускорения выполнения кода, но также и уменьшить его объем.

Существует ряд вопросов, который возникает при использовании векторизации и требует решения:

1. Переносимость реализованного кода. Какие из существующих векторных расширений использовать и рассмотреть возможность создания кроссплатформенного кода.
2. Выравнивание доступа к данным, это означает, что адрес первого элемента должен нацело делится на заданное число, например, тогда загрузка данных из памяти будет происходить быстрее.
3. Доказательство независимости операций, обязательно необходимо чтобы операции могли выполняться параллельно.
4. Поиск однотипных операций в программе над различными данными или приведение к однотипным операциям.
5. Оценка затрат на сборку и разборку векторов. Рассмотрение зависимости выполнения векторной инструкции и затрат на сборку и разборку векторов.

Одной важной особенностью, которая в значительной мере может облегчить написание кода является возможность компиляторов автоматически векторизовывать последовательный код.

Почти все процессоры, на данный момент, позволяют выполнять код с использованием векторных инструкций.

Еще в начале 70х годов были применены решения, позволяющие архитектурно решить задачу ускорения вычислений с помощью векторной обработки данных. Первые векторные процессоры поддерживали набор инструкций, которые выполнялись над векторами данных и располагались прямо в памяти. Например, в процессор ЭВМ Cray-1 были добавлены векторные регистры, что сильно ускорило выполнение инструкций. Процессор 8086, выпускавшийся компанией Intel с 1976 года, работал вместе с математическим сопроцессором, задачей которого были вычисления для чисел с плавающей точкой. Этот сопроцессор работал с регистровым стеком и поддерживал набор команд x87. С появлением процессора Intel 486DX математический сопроцессор был интегрирован в процессор. Это позволило добавить восемь новых 80-битных регистров данных для хранения чисел с плавающей точкой, но они никак не использовались при целочисленных вычислениях. Тогда помощью данного модуля, для ускорения вычислений, стали обрабатывать целочисленные массивы данных, векторы. В процессоре Intel Pentium MMX впервые дебютировало векторное расширение под названием MMX. Одним из нововведений этого расширения стало добавление новых регистров (8 64-битных регистров MM0 и MM7), которые адресовались к регистрам устройства обработки чисел с плавающей точкой. Добавлен новый набор инструкций, работающих с этими регистрами и выполняющими над ними целочисленные операции. Эти инструкции еще называются SIMD-инструкциями (сокращение от Single Instruction, Multiple Data). Каждый из этих регистров мог хранить 2 32-битных целых числа, 4 16-битных или 8 8-битных чисел. SIMD-инструкции позволяли над этими наборами чисел проводить операции одновременно. Но работать с векторами и

вещественными числами одновременно было невозможно. Для этого приходилось переключать процессор в специальный режим, что занимало большое количество времени.

Расширение SSE было логическим продолжением MMX, оно впервые было реализовано в Pentium 3. Важной особенностью расширения стал увеличенный размер векторных регистров до 128 бит. Это позволило решить проблему одновременной работы с упакованными целыми и вещественными данными. Упакованные целые числа стали обрабатываться с помощью MMX инструкций, в то же время вещественные вычисления проводились с помощью SSE инструкций и векторных регистров. Помимо векторных регистров, SSE добавил в вычислительную систему 32-битный регистр флагов и операции с этим регистром, а также дополнительно расширил набор SIMD-операций над целыми. Добавлены инструкции явной предвыборки данных, контроля кэширования данных и контроля порядка операции сохранений. Расширение SSE2 стало обновлением SSE, которое добавило новые инструкции в SSE с целью полного вытеснения технологии MMX, и обработку упакованных данных с плавающей точкой с двойной точностью. Далее последовало усовершенствование SSE инструкций, SSE3, SSE4, ну и SSE4.1, где основными нововведениями стало добавление инструкции для взаимодействия с векторными регистрами.

3.1 Intel Advanced Vector Extensions

Новой ступенью векторных технологий стало расширение Intel Advanced Vector Extensions. Оно предоставляет возможности обработки большего количества данных за одну инструкцию. Эти инструкции расширяют возможности расширений MMX и SSE, добавляя следующие новые функции:

1. 128-битные SIMD-регистры были расширены до 256 бит. Intel AVX предназначен для расширения поддержки до 512 или 1024 бит в будущем.
2. Добавлена возможность использования трех операндов, если раньше приходилось использовать два операнда A = A + B, в результате чего перезаписывался исходный операнд; Теперь новые операнды могут выполнять такие операции, как A = B + C, оставляя исходные операнды исходного текста неизменными.
3. В нескольких инструкциях используются операнды с четырьмя регистрами, позволяющие выполнять меньший и быстрый код, удаляя ненужные инструкции
4. Требования к выравниванию памяти для операндов были ослаблены.
5. Новая схема кодирования расширения (VEX) была разработана для облегчения будущих дополнений, а также для того, чтобы кодирование инструкций было меньше и быстрее выполнялось.

Тесно связанные с этими достижениями новые инструкции Fused-Multiply-Add (VMA), которые позволяют выполнять более быстрые и более точные специализированные операции, такие как одна команда A = A * B + C. Команды FMA доступны во втором поколении Intel Core. Другие функции включают в себя новые инструкции для работы с шифрованием и расшифровкой Advanced Encryption Standard, операцию умножения на переносимую операцию переноса, полезную для определенных примитивов шифрования, и некоторые

зарезервированные слоты для будущих инструкций, таких как генератор случайных чисел оборудования.

3.2 Обзор набора AVX инструкций

Новые инструкции кодируются с использованием того, что Intel называет префикс VEX, который представляет собой двух- или трехбайтовый префикс, предназначенный для устранения сложности текущей и будущей кодировки команд x86 / Intel 64. Два новых префикса VEX формируются из двух устаревших 32-битных инструкций – Load Pointer Using DS и Load Pointer с использованием ES, которые загружают регистры сегментов DS и ES в 32-битном режиме. В 64-битном режиме коды операций LDS и LES генерируют исключение недействительного кода операции, но в Intel AVX эти коды операций перераспределяются для кодирования новых инструктивных префиксов. В результате инструкции VEX могут использоваться только при работе в 64-битном режиме. Префиксы позволяют кодировать больше регистров, чем предыдущие инструкции x86, и необходимы для доступа к новым 256-битным SIMD-регистрам или использования синтаксиса трех и четырех операндов.

Аппаратное обеспечение, поддерживающее Intel AVX (и FMA), состоит из 16 256-битных регистров YMM YMM0- YMM15 и 32-битного регистра управления / состояния, называемого MXCSR. Регистры YMM сглажены в более старых 128-битных регистрах XMM, используемых для Intel SSE, обрабатывая регистры XMM как нижнюю половину соответствующего регистра YMM, как показано на рисунке 3.2 и рисунке 3.3.

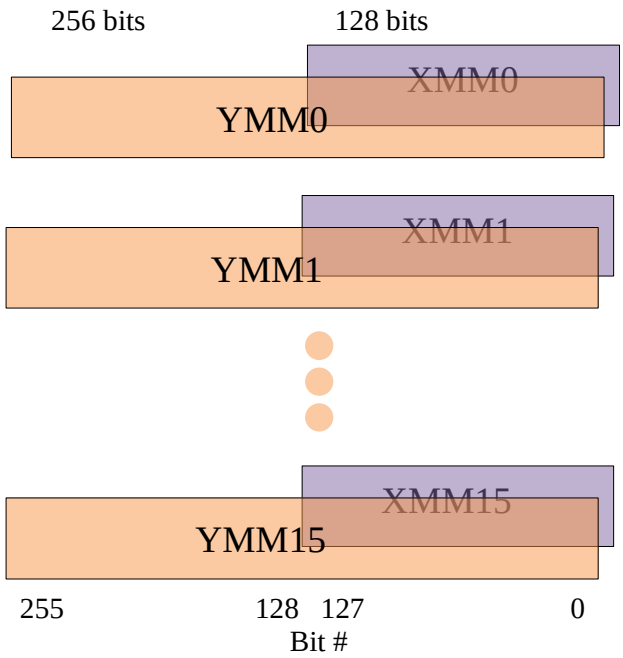

Рисунок 3.2 – Структура YMM и XMM регистров

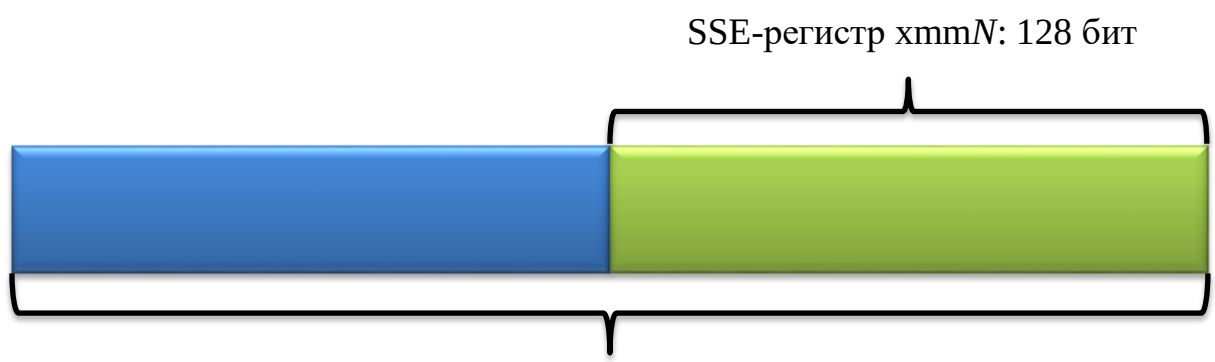

Рисунок 3.3 – Структура векторного AVX регистра

На рисунке 3.4 показаны типы данных, используемые в инструкциях Intel SSE и Intel AVX. Для Intel AVX допускается использование нескольких кратных 32-битных или 64-битных типов с плавающей запятой, добавляющих до 128 или 256 бит, а также кратные любому целочисленному типу, который добавляет 128 бит.

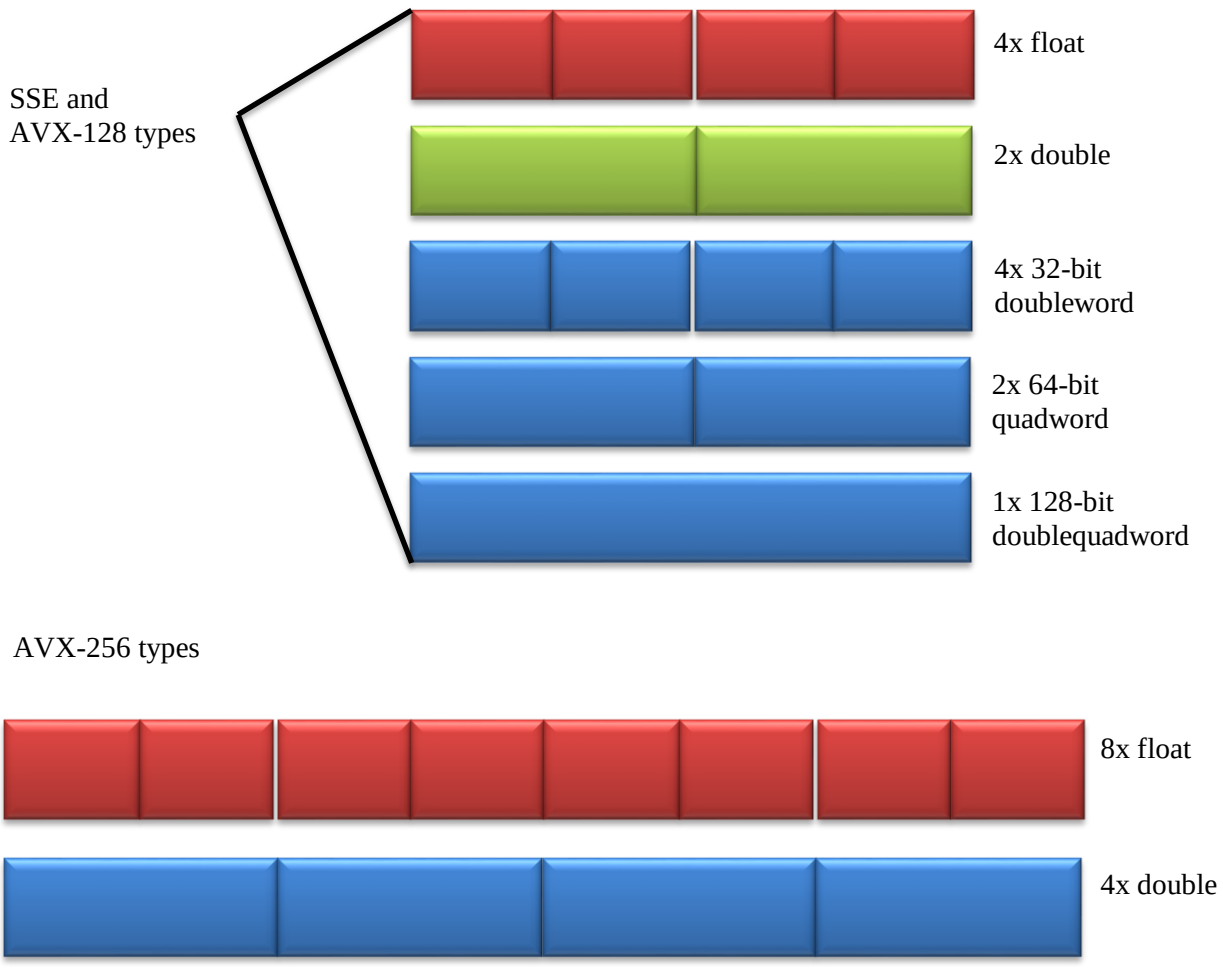

Рисунок 3.4 – Типы данных Intel AVX и Intel SSE

3.3 Инструменты векторизации

Для написания векторного кода существует множество способов, начиная от ассемблера и до автоматического векторизатора. На рисунке 3.5 показаны возможные варианты получения векторизованного кода.

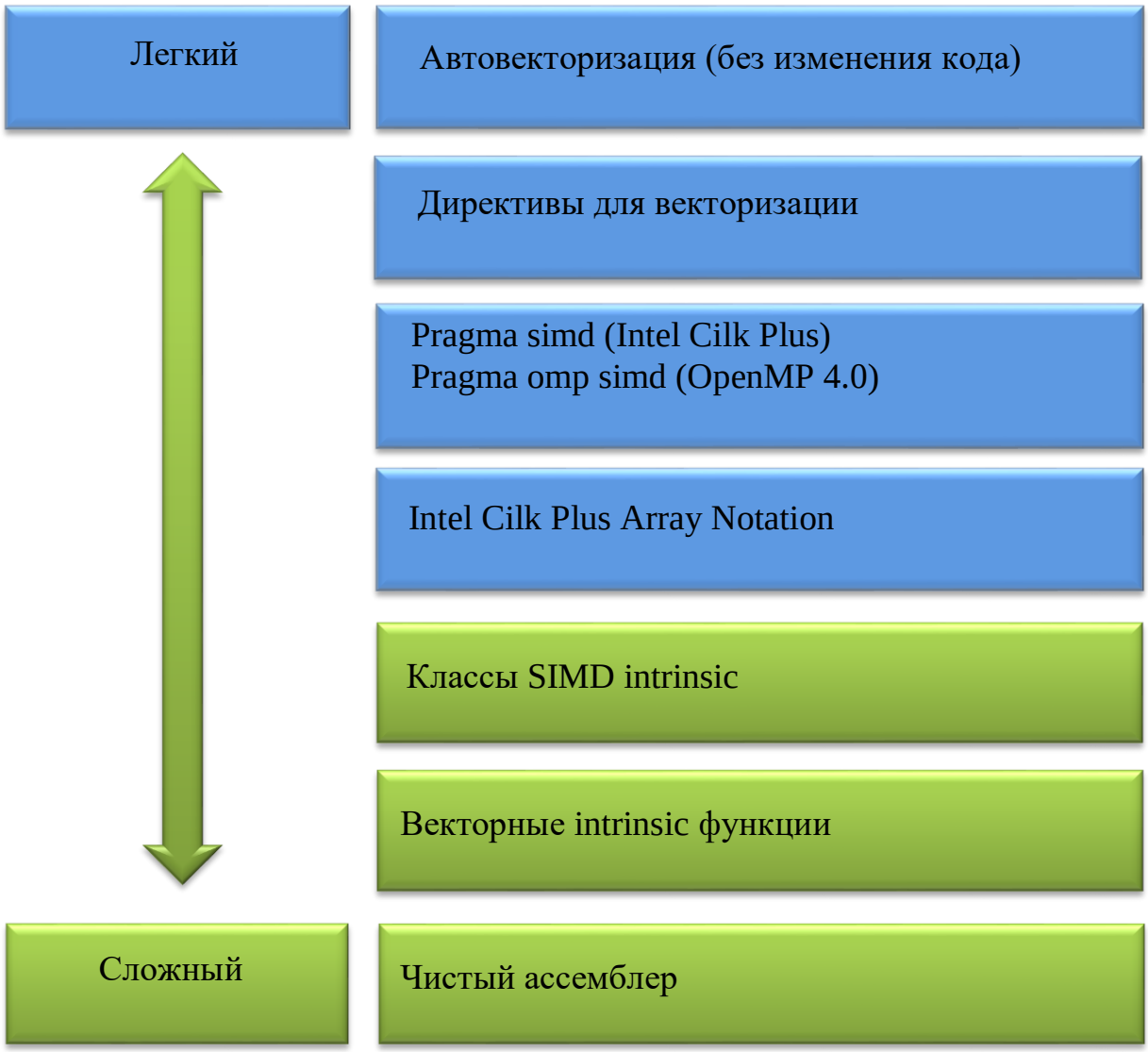

Рисунок 3.5 – Методы векторизации кода в порядке возрастания сложности

У ассемблера есть несколько преимуществ, код можно писать с использованием нужных инструкций и использовать все возможности процессора. Но данный метод содержит один серьёзный недостаток, это платформозависимость, и для каждой новой версии аппаратного обеспечения потребуется адаптация или, что еще хуже, полностью новая реализация.

Далее идут функции-интринсики, более удобный аналог ассемблера, но тоже не лишенный вышеупомянутого недостатка, времени для переписывания кода под новую платформу потребуется немало. В листинге 3.3 приведен пример реализации перемножения двух массивов.

Листинг 3.3 – Перемножение векторов с помощью функций-интринсиков на языке Си

```
#include <immintrin.h>

double Arr1[200], Arr2[200], Res[200];
for (int j = 0; j < 200; j = j + 4) {
          __m256d m1 = _mm256_load_pd(&Arr1[j]);
          __m256d m2 = _mm256_load_pd(&Arr1[j]);
          __m256d r = _mm256_mul_pd(m1, m2);
          _mm256_store_pd(&Res[i], r);
}
```

Ограничение на зависимость кода от используемой платформы будет всегда, пока необходимые инструкции явно используются в коде, и не важно будет это ассемблер или функции-интринсики. Данная зависимость сохраняется даже при использовании SIMD intrinsic классов, которые являются следующим уровнем абстракции, хотя в этом случае разработчику не нужно знать какие функции использовать, а просто создать данные нужного класса.

В современном мире термин «экономическая эффективность» играет важную роль. Поэтому новые команды появляются в процессорах не просто так, а чтобы сделать выпускаемый продукт более привлекательным для покупателей и мотивировать их на замену более совершенными архитектурами. Но если модификация и оптимизация существующих инструкций и улучшение архитектуры сразу же могут положительно сказаться на производительности, то с новые инструкции нужно поддерживать, включить их поддержку производителями компиляторов. Поэтому новые версии компиляторов выходят одновременно с появлением на рынке знаковых новых архитектур.

Наиболее простым способом является возможность векторизации кода автоматически с помощью компилятора, т.е. компилятор должен уметь сам осуществлять преобразование скалярного кода в векторный. Автоматический векторизатор имеет также большую самостоятельную ценность, как важный компонент оптимизирующего компилятора, позволяющий сравнительно легко улучшать производительность вычислительных программ. Достоинство автоматического векторизатора в том, что достаточно добавить опции при компиляции, и компилятор сам выполнит векторизацию тех циклов, которые выгодно векторизовать.

Но в таком случае возможности векторизации ограничены и компилятор не всегда может векторизовать код без дополнительных данных или подсказок.

4 ТАБЛИЦЫ ПОИСКА

Под таблицей поиска (look-up table, LUT) понимается структура данных, обычно ассоциативный массив, который заменяет задачу вычисления необходимого элемента на простую операцию поиска вхождения необходимого элемента в таблицу. Экономия машинного времени представляется значительной, т.к. извлечение элемента из памяти происходит заметно быстрее чем вычисление нового значения или операции ввода/вывода. Таблица поиска может заимствоваться или будет заново вычислена единожды при запуске программы.

Одним из наиболее ярких примеров можно считать вычисление синуса. Вместо того что бы каждый раз тратить ресурсы на вычисление нового значения рассчитывается определенное количество значений, например, для целых градусов. Затем, когда программе понадобится значение, она использует таблицу поиска. Большинство компьютеров выполняют только основные арифметические операции и не могут напрямую вычислять синус заданного значения. Вместо этого они используют алгоритм CORDIC или сложные формулы, основывающиеся на рядах Тейлора для вычисления синуса с высокой степенью точности:

$$\sin(x) \approx x - \frac{x^3}{6} + \frac{x^3}{120} - \frac{x^7}{5040} + \cdots$$

Такой способ вычисления оказывается сильно затратным, это довольно заметно на медленных процессорах, в то же время есть множество приложений, особенно в трехмерной графике, которые должны вычислять тысячи значений синуса в секунду. Наилучшим решением является изначальное вычисление множества значений синуса, равномерно распределенных, а затем для нахождения синуса *x* мы выбираем синус значения, ближайшего к *x*.

Так же таблицы поиска используются в процессорах, например, в модуле операций с плавающей точкой используется LUT, основываясь на данных которой процессор рассчитывает значения.

4.1 Поиск в одномерной таблице поиска

Пусть дано некоторое значение X и массив, или таблица, возрастающих значений $t_1 < t_2 < \ldots < t_n$. Задача поиска в одномерной таблице состоит в нахождении наименьшего целого числа $i = i(x)$, удовлетворяющего условию $x \le t_i$. Такой поиск обычно приходится выполнять, когда заданы пары значений (t_i, f_i) $(i = 1, N)$, представляющие таблично заданную функцию f, т.е. $f_i = f(t_i)$, и требуется вычислить значение функции f в некоторой точке x. Для этого независимо от используемой схемы интерполирования необходимо определить значение $i(x)$. Очень часто функция f представляет собой физические данные , соответствующие различным значениям временных или пространственных координат. Например в задаче исследования атмосферы величина t_i может представлять собой высоту , а функция f_i – некоторый параметр атмосферы на этой высоте, например скорость звука. В реальной атмосфере скорость звука изменяется с высотой по сложному закону. Поэтому используется табличное представление функции с выбором ее значений на определённых высотах; промежуточные значения функции вычисляются путем интерполирования.

5 РЕАЛИЗАЦИЯ МЕТОДОВ ПОИСКА

Кусочно-заданная функция f одной скалярной переменной записывается в виде $f(x) = f_i(x), \quad t_{i-1} < x \leq t_i, \quad$ где $t_1 < t_2 < \cdots < t_M, \quad t_1 < x \leq t_M.$ В практических задачах значения *M* невелики. Предположим, что *x* – скалярная величина и ничего не известно об $i = i(x)$. Интервалы (t_{i-1}, t_i) часто называют ячейками, а число $i(x)$ называют индексом x[2].

Таким образом, вычисление функции производится за два шага:

1) Поиск в таблице для определения $i(x)$.
2) Вычисление значения $f_{i(x)}(x)$.

Методы поиска должны удовлетворять следующему определению функции:

FUNCTION LUF (*X, T, N*) принимает значение:

1) 1, если $N \leq 0$;

2) наименьшее *J* такое, что $X \leq T(J)$;

3) $N + 1$, если $X > T(N)$, где $T(1) < T(2) < \ldots < T(N)$ – возрастающая последовательность чисел

При оценке алгоритмов реализации функции LUF предполагается что попадание *X* в любой интервал равновероятно.

5.1 Метод поиска делением интервала поиска пополам

Методом поиска делением интервала поиска пополам примерно за $\log_2 N$ итераций определяется число $f(x)$; в процессе поиска производится сравнение *x* с некоторым «средним» элементом, лежащим между текущими нижней и верхней границами интервала поиска [2]. Если есть таблица, содержащая упорядоченную последовательность данных, то метод деления интервала поиска пополам очень эффективен.

На каждой итерации должны быть выполнены следующие шаги:

1) определение следующего значения *J*;
2) считывание *T*(*J*);
3) сравнение *T*(*J*) с *X*;
4) определение того, какая граница — нижняя или верхняя — должна быть изменена;
5) выполнение перечисленных действий до тех пор, пока верхняя и нижняя границы различаются больше чем на 1.

При осуществлении последовательных итераций шаг 5 может выполняться параллельно с шагами 1 и 2, однако шаги 2–4 должны выполняться только последовательно.

Метод деления интервала поиска пополам – эффективный метод. Если, например, длина массива равна 1023, после первого сравнения область сужается до 511 элементов, а после второй –-до 255. Для поиска в массиве из 1023 элементов достаточно 10 сравнений.

Ниже представлен листинг 6.1 последовательной версии.

Листинг 6.1 – последовательная версия метода поиска делением интервала поиска пополам

```
int search_l(int a, float mass[], float n)
{
    int low, high, middle;
    low = 0;
    high = n - 1;
    while (low <= high)
    {
        middle = (low + high) / 2;
        if (a < mass[middle])
            high = middle - 1;
        else if (a > mass[middle])
            low = middle + 1;
        else
            return middle;
    }
    return -1;
}
```

Одна возможная оптимизация этого алгоритма, использует тот факт, что размер массива равен степени двух. Код может быть проще, поскольку разделение на два никогда не приводит к ошибкам округления. Поэтому данная техника была использована для упрощения двоичного поиска. Реализация оптимизированного метода поиска делением интервала поиска пополам представлена на листинге 6.2.

Листинг 6.2 - Реализация оптимизированного метода поиска делением интервала поиска пополам

```
int BSearch_improved(int *data, int N, int key)
{
     double powerOfTwo = pow(2, floor(log2(N - 1)));
     int splitIndex = N - powerOfTwo;
     double splitValue = data[splitIndex];
     int p = splitIndex * (key >= splitValue);
     for (int i = powerOfTwo / 2; i >= 1; i = i / 2)
          if (key >= data[p + i])
               p = p + i;
     return p;
}
```

Финальная оптимизация заключается в векторизации алгоритма с использованием AVX инструкций. Элементы таблицы массива будут сравниваться с ключом не по одному, а по четыре элемента одновременно. Векторизованная версия метода поиска делением интервала поиска пополам представлена в листинге 6.3.

Листинг 6.3 – Векторизованная версия метода поиска делением интервала поиска пополам

```
int BSearch_AVX(int *data, int N, int key)
{
     double powerOfTwo = pow(2, floor(log2(N - 1)));
     int splitIndex = N - powerOfTwo;
     double splitValue = data[splitIndex];
     int p = splitIndex * (key >= splitValue);
     double *store_p = new double;
     *store_p = p;
     if (data[N - 1] < key) return -1;
     for (int j = powerOfTwo / 2; j >= 1; j /= 2) {
          __m256d xm_jvec = _mm256_set_pd(j + j / 2 + j /
4 + j / 8,
                                  j + j / 2 + j / 4, j + j /
2, j);
          __m256d xm_idxvec = _mm256_set1_pd(p);
          int cmpval0 = data[p + j];
          int cmpval1 = data[p + j + j / 2];
          int cmpval2 = data[p + j + j / 2 + j / 4];
          int cmpval3 = data[p+j+j / 2 + j / 4 + j / 8];
          __m256d xm_cmpvalvec =
                                 _mm256_set_pd(cmpval3,
                                 cmpval2, cmpval1, cmpval0);
          __m256d xm_valvec = _mm256_set1_pd(key);
          xm_idxvec = _mm256_add_pd(
                    xm_idxvec, _mm256_andnot_pd(
                         _mm256_cmp_pd(xm_valvec,
xm_cmpvalvec,
                                 _CMP_LT_OQ), xm_jvec));
          _mm256_store_pd((store_p+0), xm_idxvec);
          for (int i = 0; i < 4; i++) {
               if (key >= store_p[i]){
                    p = (int)store_p[i];
                    break;
               }
          }
     }
     if (key != data[p]) return -1;
     return p;
}
```

Количество итераций K в худшем случае (когда элемент не найден, или находится на последнем шаге) легко оценить, если количество элементов в массиве $n = 2^m - 1$. После первого повторения цикла, если A не был найден, размер области поиска становится равным $\frac{n-1}{2}$, то есть опять является числом того же вида – степень двойки минус единица. После i повторений, если A всё ещё не найден, в области поиска останется 2^{m-i-1} элементов. Значит, после $m-1$ повторения останется один элемент. После этого цикл повторится ещё один раз, и работа будет закончена. Таким образом, общее число повторений цикла равно $K = m =$

$log_2(n + 1)$. Нетрудно показать, что в общем случае (когда n произвольное) число повторений будет равно ближайшему целому сверху к числу $log_2(n + 1)$, то есть $K = \lceil log_2(n + 1) \rceil$.

В среднем случае сложность можно оценить, используя ту же схему рассмотрения, но находя среднее число итераций по всем элементам массива. Для вида $n = 2^m - 1$:

$K = \frac{1}{n}\sum_{i=0}^{log_2(n+1)-1} i2^i = \lceil log_2(n + 1) \rceil + \frac{2}{n} - 2$, в общем виде отличается от формулы не более, чем на 0,5. В лучшем же случае (когда элемент сразу найден) — $K = 1$.

Таким образом, алгоритм обладает логарифмической временной сложностью по данным, и даже последовательная реализация достаточно быстра и, в принципе, не нуждается в распараллеливании.

5.2 Метод линейного векторного поиска

Линейный алгоритм является простейшим алгоритмом поиска и имеет тривиальную реализацию. Поиск значения осуществляется простым сравнением очередного рассматриваемого значения и, если значения совпадают, то поиск считается завершённым. Асимптотическая сложность алгоритма – $O(n)$.

Для векторизации алгоритма была подобрана длина вектора T – 32 * 4. По условию $T(1)$ - $X < 0$ может быть сформирована маска. Число единиц в маске определяет искомый индекс; если все элементы маски имеют значение 1, то $X > T$ (32 * 4). В последнем случае должны быть считаны следующие 32 * 4 элемента исходного массива.

Число итераций в среднем равно (1 + N / 32 * 4) / 2, так как после обнаружения искомого места вхождения числа X в таблицу итерации прекращаются. Таким образом, общее число тактов на выполнение этого алгоритма в среднем равно 50 * (1 + N / 32 * 4). Метод линейного векторного поиска, имеет лучшие характеристики по времени поиска по сравнению с методом поиска делением интервала поиска пополам для интересующих нас значений N.

В листинге 6.4 представлен метод линейного поиска.

Листинг 6.4 – Метод линейного поиска

```
int linear_search_ref(const int *A, int key, int n) {
     int result = -1;
     int i;
     for (i = 0; i < n; ++i)  {
          if (A[i] >= key){
               result = i;
               break;
          }
     }
     return result;
}
```

В листинге 6.5 представлен метод линейного векторного поиска.

Листинг 6.5 – Метод линейного векторного поиска

```
int linear_search(const int *A, int key, int n)
{
#define VEC_INT_ELEMS 4
#define BLOCK_SIZE (VEC_INT_ELEMS * 32)
#define CODE_OF_NOT_EQ 15
     const __m256d vkey = _mm256_set1_pd(key);
     int vresult = -1;
     int result = -1;
     int i, j;

     for (i = 0; i <= n - BLOCK_SIZE; i += BLOCK_SIZE){
       __m256d vmask0 = _mm256_set1_pd(-1);
       __m256d vmask1 = _mm256_set1_pd(-1);
       int mask0, mask1;

       for (j = 0; j < BLOCK_SIZE; j += VEC_INT_ELEMS *
2)   {
                __m256d vA0 = _mm256_set1_pd(A[i + j]);
                __m256d vA1 = _mm256_set1_pd(A[i + j +
VEC_INT_ELEMS]);
                __m256d vcmp0 = _mm256_cmp_pd(vkey, vA0,
_CMP_GE_OS);
                __m256d vcmp1 = _mm256_cmp_pd(vkey, vA1,
_CMP_GE_OS);
                vmask0 = _mm256_and_pd(vmask0, vcmp0);
                vmask1 = _mm256_and_pd(vmask1, vcmp1);
           }
           mask0 = _mm256_movemask_pd(vmask0);
           mask1 = _mm256_movemask_pd(vmask1);
           if ((mask0 & mask1) != CODE_OF_NOT_EQ) {
                vresult = i;
                break;
           }
     }
     if (vresult > -1) {
          result = vresult +
linear_search_ref(&A[vresult], key, BLOCK_SIZE);
     }
     else if (i < n)
     {
       result = i + linear_search_ref(&A[i], key, n - i);
     }
     return result;
#undef BLOCK_SIZE
#undef VEC_INT_ELEMS
}
```

5.3 Метод поиска делением интервала поиска на *M* частей

Метод поиска делением интервала поиска на *M* частей является векторным аналогом метода поиска делением интервала поиска пополам. В методе линейного векторного поиска формируется вектор разностей $T(I) - X$, где $I = 1, 2, \ldots, N$. В алгоритме поиска с разбиением интервала на *M* частей вычисляются разности $T(I) - X$ для $I = 1, 1 + L, \ldots, 1 + (M - 1)L$. При этом должны выполняться условия: $M \le N$, $1 + (M - 1)L \le N$. Производится выбор *M* элементов из массива *T*, расположенных в интервале поиска, и определение того отрезка, в котором находится число *X*. Процесс деления продолжается до тех пор, пока величина *L* не станет равной 1, т.е. требуется выполнить примерно $\log_M N$ разбиений.

Время выполнения каждой итерации складывается из времени, равного *M*-чайму, и некоторой значительной доли общего времени, которая не зависит от величины *M*.

В векторизованной версии, рассмотренной в листинге 6.6, основное внимание уделялось адаптированию алгоритма к исследуемым архитектурам с использованием AVX инструкций и функций-интринсиков.

Листинг 6.6 – Векторизованный метод поиска делением интервала поиска на M частей.

```
int LUF_AVX(double X, double* T, int NBIG)
{
  int M = 33;
  int N = NBIG;
  int LUF = 0;
  double *storeX = (double*)malloc(4 * sizeof(double));

  if (X > T[NBIG - 1] || N <= 0) {
       free(storeX);
       return -1;
  }
  do {
       int L = (N - 1) / M;

       if (L == 0)
       {
            int K = 0;
            double tmpX[2];
            double tmpT[2];
            for (int i = LUF; i < NBIG - 1; i += 2)
            {
                 __m256d mT;
                 __m256d mX;
                 tmpT[0] = T[i];
                 tmpT[1] = T[i + 1];
                 mT = _mm256_load_pd(tmpT);
                 tmpX[0] = X;
                 tmpX[1] = X;
                 mX = _mm256_load_pd(tmpX);
                 __m256d cm = _mm256_max_pd(mT, mX);
                 _mm256_store_pd(storeX, cm);
                 if (storeX[0] == X) {
```

```
                    K = K + 1;
               }
               if (storeX[1] == X) {
                    K = K + 1;
               }

          }

          LUF = LUF + K;
          return LUF-1;
     }
     int K = 0;
     double tmpX[2];
     double tmpT[2];

     for (int i = LUF; i < LUF + N - 1; i += L * 2)
     {
          __m256d mT;
          __m256d mX;

          tmpT[0] = T[i];
          if (i + L < NBIG)
               tmpT[1] = T[i + L];
          else tmpT[1] = T[i];

          mT = _mm256_load_pd(tmpT);

          tmpX[0] = X;
          tmpX[1] = X;
          mX = _mm256_load_pd(tmpX);
          __m256d cm = _mm256_max_pd(mT, mX);

          _mm256_store_pd(storeX+0, cm);
          if (storeX[0] == X) {
               K = K + 1;
          }
          if (storeX[1] == X && i + L < NBIG) {
               K = K + 1;
          }
     }
     int J = 1 + (K - 1)*L;
     if (K == 0)
          return LUF-1;
     else
          LUF = LUF + J;
     if (K == M)
          N = N - J;
     else
          N = L;
  } while (N > 1);
  free(storeX);
  return LUF-1;
}
```

Ниже представлен листинг 6.7 метода поиска делением на M частей.

```
int LUF(double X, double T[], int NBIG)
{
     int M = 33;
     int N = NBIG;
     int LUF = 0;
     if (X > T[NBIG - 1] || N <= 0) return -1;
     do {
          int L = (N - 1) / M;
          if (L == 0){
               int K = 0;
               for (int i = LUF; i < NBIG; i++){
                    if (T[i] <= X) {
                         K = K + 1;
                    }
                    else break;
               }
               LUF = LUF + K;
               return LUF-1;
          }
          int K = 0;
          for (int i = LUF; i < NBIG; i += L){
               if (T[i] <= X) {
                    K = K + 1;
               }
               else break;
          }
          int J = 1 + (K - 1)*L;
          if (K == 0)
               return LUF-1;
          else
               LUF = LUF + J;
          if (K == M)
               N = N - J;
          else
               N = L;
     } while (N >= 1);
     return LUF-1;
}
```

6 МЕТОДИКА ТЕСТИРОВАНИЯ РАЗРАБОТАННОГО ПО

6.1 Конфигурация тестовых вычислительных систем

Конфигурация кластерной ВС Oak [2]

Кластер Oak укомплектован 6 вычислительными узлами, управляющим узлом, вычислительной и сервисной сетями связи, а также системой бесперебойного электропитания.

Узлы построены на базе серверной платформы Intel S5520UR. На каждом узле размещено два процессора Intel Xeon E5620 с частотой 2,4 GHz. Пиковая производительность кластера – 460 GFLOPS. В таблице 5.1 представлена конфигурация вычислительного узла. В таблице 5.2 представлена конфигурация управляющего узла. В таблице 5.3 представлена представлена конфигурация сети управляющего узла.

Таблица 5.1 - Конфигурация вычислительного узла

Системная плата	Intel S5520UR
Процессор	2 x Intel Xeon E5620 (2,4 GHz; Intel-64)
Оперативная память	24 GB (6 x 4GB DDR3 1067 MHz)
Жесткий диск	SATAII 250GB (Seagate Barracuda ST3250318AS)
Сетевая карта	1 x Mellanox MT26428 InfiniBand QDR 2 x Intel Gigabit Ethernet (Intel 82575EB Gigabit Ethernet Controller)
Корпус	Rack mount 2U

Таблица 5.2 - Конфигурация управляющего узла

Системная плата	Intel S5520UR
Процессор	2 x Intel Xeon E5620 (2,4 GHz; Intel-64)
Оперативная память	24 GB (6 x 4GB DDR3 1067 MHz)
Жесткий диск	4 x SATAII 500 GB (Seagate Barracuda ST3500514NS)
Сетевая карта	1 x Mellanox MT26428 InfiniBand QDR 2 x Intel Gigabit Ethernet (Intel 82575EB Gigabit Ethernet Controller)
Корпус	Rack mount 2U

Таблица 5.3 – Конфигурация сети управляющего узла

Сервисная сеть	Коммутатор Gigabit Ethernet (D-Link DGS-1224T)
Вычислительная сеть	Коммутатор Infiniband QDR (Mellanox InfiniScale IV IS5030 QDR 36-Port InfiniBand Switch)

Кластер размещен в 19’’ телекоммуникационном шкафу. Система бесперебойного электропитания имеет мощность 12kVA и построена на базе источников бесперебойного питания APC Smart-UPS XL 3000VA. Структура коммуникационной сети кластера Oak представлена на рисунке 5.1.

В таблице 5.4 представлена конфигурация локальной вычислительной системы.

Таблица 5.4 - Конфигурация локальной ВС

Системная плата	Asus P8Z77-V LK
Процессор	QuadCore Intel Core i7-3770K, 4700 MHz (47 x 100)
Оперативная память	16Gb (2 x 8Gb DDR3 1866 MHz KHX2133C11D3/8GX)
SSD диск	Диск #1 - Samsung SSD 850 EVO 250GB (232 ГБ)
Жесткий диск	Диск #2 - ST2000DM001-1CH164 (1863 ГБ)
Сетевая карта	Realtek RTL8168/8111 Gigabit Ethernet

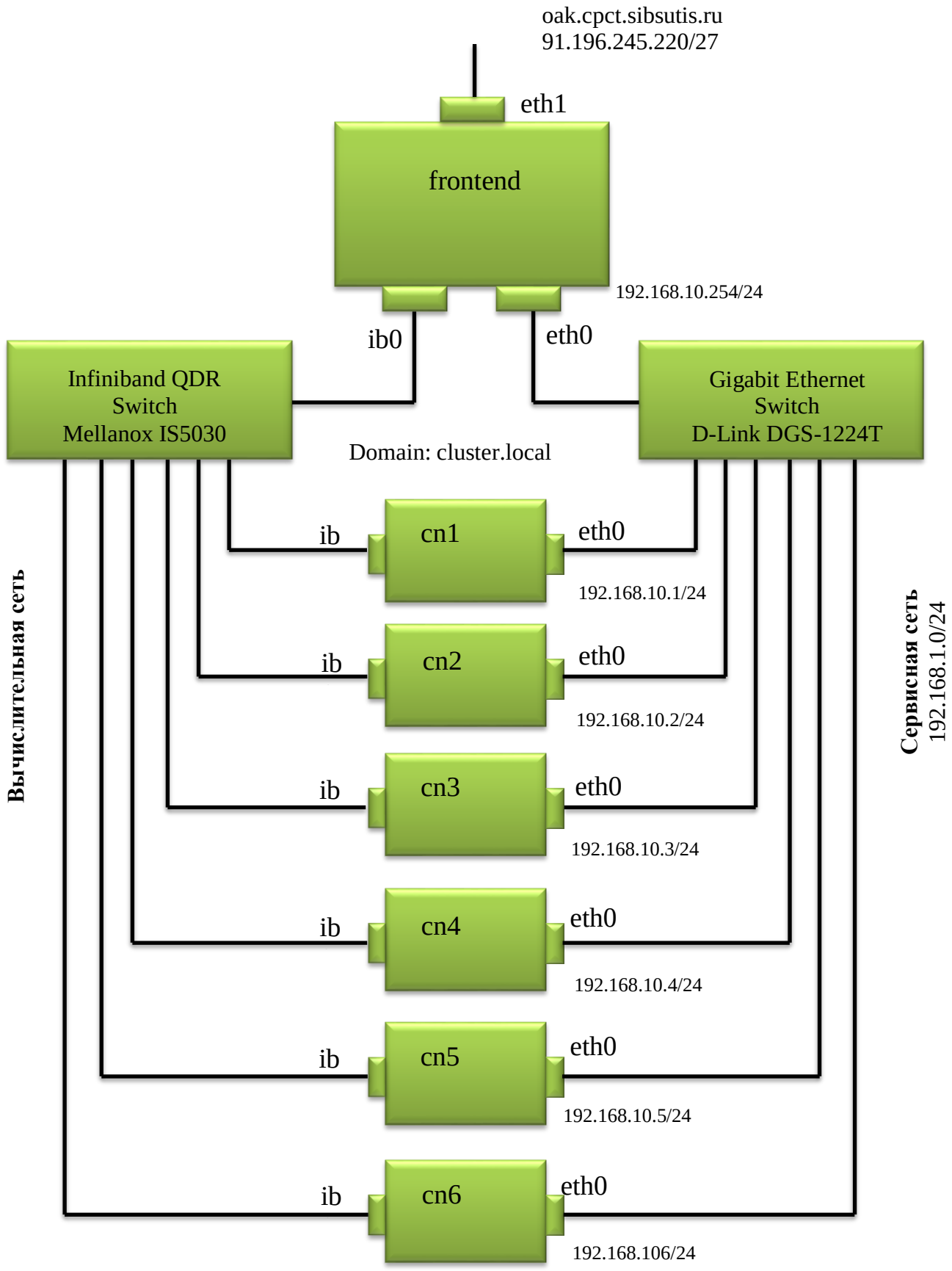

Рисунок 5.1 – Структура коммуникационной сети кластера Oak

6.2 Компиляция программ

Компиляция программ осуществлялась с помощью компиляторов GNU GCC C/C++, Intel C/C++ Compiler, LLVM/Clang. В таблицах 5.5, 5.6 и 5.7 представлены ключи компиляции программ для компиляторов GNU GCC C/C++, Intel C/C++ Compiler, LLVM/Clang.

Таблица 5.5 - Опции компиляции GNU GCC C/C++

Скалярная версия	-O3 -ffast-math -fivopts -march=native -fno-tree-vectorize
Автоматически векторизованная версия	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed
Векторизованная версия	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed -fno-tree-vectorize

Значения опций компилятора GNU GCC C/C++:

- **-c** - создать только объектный файл (source.o) из исходного
- **-o** - создать загрузочный файл с именем file (по умолчанию создается файл с именем a.out)
- **-Wall** - вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы
- **-O** - включить оптимизацию
- **-O0** - этот уровень отключает оптимизацию полностью и является уровнем по умолчанию, если никакого уровня с префиксом -O не указано в переменных CFLAGS или CXXFLAGS. Это сокращает время компиляции и может улучшить данные для отладки, но некоторые приложения не будут работать должным образом без оптимизации. Эта опция не рекомендуется, за исключением использования в целях отладки
- **-O1** - это наиболее простой уровень оптимизации. Компилятор попытается сгенерировать быстрый, занимающий меньше объема код, без затрачивания наибольшего времени компиляции
- **-O2** - рекомендуемый уровень оптимизации, до тех пор. -O2 активирует несколько дополнительных флагов вдобавок к флагам, активированных -O1. С параметром -O2, компилятор попытается увеличить производительность кода без нарушения размера, и без затрачивания большого количества времени компиляции
- **-O3** - наибольший возможный уровень оптимизации. Включает оптимизации, являющейся дорогостоящей с точки зрения времени компиляции и потребления памяти. Компиляция с -O3 не является гарантированным способом повышения производительности, и на самом деле во многих случаях может привести к замедлению системы из-за больших двоичных файлов и увеличения потребления памяти
- **-mavx** – включение поддержки инструкций AVX
- **-ffast-math** - этот параметр не включается никакой опцией -O, кроме -Ofast, поскольку это может привести к некорректному выходу для программ, которые зависят от точной реализации спецификаций IEEE или ISO для математических функций. Однако он может дать более

быстрый код для программ, для которых не требуются гарантии этих спецификаций

- **-fivopts** - влияет только на компиляцию
- **-march=native** - сообщает компилятору, какой код генерировать для архитектуры процессора; он сообщает GCC компилятору, что тот должен генерировать код для определенного типа CPU. Разные типы CPU имеют разные возможности, поддерживают различные наборы команд и обладают разными способами исполнения кода. Флаг -march проинструктирует компилятор генерировать специфичный код для архитектуры CPU, со всеми доступными возможностями, особенностями, наборами команд, интересными функциями и так далее. Если тип процессора все еще нее определен, либо не знаете какую настройку выбрать, то можно воспользоваться параметром -march=native. Когда используется этот флаг, GCC попытается распознать процессор и автоматически установит для него подходящие флаги
- **-fopt-info-vec-missed** - печатает информацию о пропущенных возможностях оптимизации от векторизации на stderr, -fopt-info-vec-missed эквивалентен -fopt-info-missed-vec. Порядок имен групп оптимизации и типов сообщений, перечисленных после -fopt-info, не имеет значения
- **-fopt-info-vec** – выводит информацию о проделанной векторизации на stderr

Таблица 5.6 - Опции компиляции Intel C/C++ Compiler

Скалярная версия	-O3 -xHost -no-vec
Автоматически векторизованная версия	-O3 -xHost -qopt-report3 -qopt-report-phase=vec,loop -qopt-report-embed
Векторизованная версия	-O3 -xHost -qopt-report3 -qopt-report-phase=vec,loop -qopt-report-embed

Значения опций компилятора Intel C/C++ Compiler:

- **-xHost** – флаг означает оптимизацию для того процессора, на котором запущен компилятор
- **-qopt** – сообщает компилятору генерировать отчет об оптимизации
- **-qopt-report3** – (n = 3 необязательно) указывает уровень детализации в отчете в диапазоне от 0 до 5. Если указать ноль, отчет не будет сгенерирован. Для уровней n = 1 - n = 5 каждый уровень включает всю информацию предыдущего уровня, а также дополнительную информацию. Уровень 5 дает наибольший уровень детализации. Если не указать n, по умолчанию используется уровень 2
- **-qopt-report-embed** – определяет, будут ли внедрены специальные аннотации информации цикла в файл объекта и / или файл сборки при его создании

Таблица 5.7 - Опции компиляции LLVM/Clang

Скалярная версия	-O3 -ffast-math -fno-vectorize
Автоматически векторизованная версия	-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize
Векторизованная версия	-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize

- **-Rpass=loop-vectorize** - идентифицирует циклы, которые были успешно векторизованы.
- **-Rpass-missed=loop-vectorize** – идентифицирует циклы, которые не были успешно векторизованы.

6.3 Методика измерения времени

Измерение времени выполнения кода проводилось методом вычисления разницы между начальным временем и конечным. То есть, есть начальное значение времени, после которого объявлен фрагмент кода, время выполнения которого необходимо измерить. После фрагмента кода фиксируется ещё одно, конечное, значение времени. После чего, из конечного значения времени вычитаем начальное время и получим время, за которое выполнился измеряемый фрагмент кода или вся программа. Каждый алгоритм выполнялся N раз, за итоговое время бралось среднее время выполнения из количества запусков.

Рекомендации к тестированию кода

1. Следует выбирать самый точный таймер, имеющийся в системе:RTC, HPET, TSC, …
2. Необходимо учитывать эффекты кеш-памяти и возможно игнорировать первый запуск теста – не включая его в итоговый результат
3. Измерения необходимо проводить несколько раз
4. Запуск тестов осуществлять, по возможности, в одних и тех же условиях – набор запущенных программ, аппаратная конфигурация…

Методика измерения времени выполнения кода[3]:

1. Готовим систему к проведению измерений
   - настраиваем аппаратные подсистемы (настройки BIOS)
   - параметры операционной системы и процесса, в котором будут осуществляться измерения
2. Выполняем разогревочный вызов измеряемого кода (Warmup)
   - Регистрируем время выполнения первого вызова и не учитываем его в общей статистике (при первом вызове может осуществляться отложенная инициализация и пр.)
3. Выполняем многократные запуски и собираем статистику о времени выполнения
   - Каждый запуск должен осуществляться в одних и тех же условиях (входные массивы заполнены одними и теми же данными, входные данные отсутствуют/присутствуют в кеш-памяти процессора, …)
4. Проводим статистическую обработку результатов измерений
   - Вычисляем оценку математического ожидания времени выполнения (mean)

Листинг 5.1 – Листинг измерения времени выполнения кода

```
#include <sys/time.h>

double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int main ()
{

double t;
…
    t = wtime();
        /* Measured code */
    t = wtime() - t;

    printf("%.12f\n", t);

    return 0;
}
```

6.4 Обработка результатов измерений

Для обработки результатов измерений использовались следующие функции:

- Математическое ожидание времени выполнения:

$$\bar{t} = \frac{t_1 + t_2 + \cdots + t_n}{n}$$

- Стандартное отклонение

$$s = \sqrt{s^2}$$

7 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

7.1 Метод поиска делением интервала поиска пополам

Метод поиска делением интервала поиска пополам является наиболее быстрым среди рассматриваемых нами методов, то для измерения времени использовалось среднее время дух тысяч вызовов функции поиска, не учитывая времени первого запуска. Размер выборки составлял от $60 * 10^7$ до $160 * 10^7$ элементов типа double. Объем используемой оперативной памяти составил 11.92Gb для $160 * 10^7$ элементов. Результаты тестирования метода поиска делением интервала поиска пополам для компиляторов GNU GCC C/C++, Intel C/C++ Compiler и LLVM/Clang представлены в таблицах 7.1, 7.2, 7.3.

Таблица 7.1 – Результат тестирования метода поиска делением интервала поиска пополам с помощью компилятора GNU GCC C/C++

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,000008756	0,000010454
80	0,000008153	0,000010440
100	0,000009245	0,000010386
120	0,000008727	0,000010475
140	0,000008171	0,000010691
160	0,000009619	0,000010777

Таблица 7.2 – Результат тестирования метода поиска делением интервала поиска пополам с помощью компилятора Intel C/C++ Compiler

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,000008756	0,000010454
80	0,000008153	0,000010440
100	0,000009245	0,000010386
120	0,000008727	0,000010475
140	0,000008171	0,000010691
160	0,000009619	0,000010777

Таблица 7.3 – Результат тестирования метода поиска делением интервала поиска пополам с помощью компилятора LLVM/Clang

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,000008756	0,000010248
80	0,000008153	0,000010277
100	0,000009943	0,000010579
120	0,000009901	0,000010626
140	0,000008171	0,000011213
160	0,000009619	0,000010767

Проанализировав результаты можно увидеть, что векторизованный метод поиска оказался хуже последовательного, учитывая не высокую вычислительную

сложность, ведь для выборки 10^9 элементов придется сделать всего лишь 30 сравнений, можно считать, что векторизованная версия для столь простого алгоритма является актуальной так как алгоритм обладает логарифмической временной сложностью по данным и даже последовательная реализация достаточно быстра и, в принципе, не нуждается в распараллеливании. В приложении Г, на графиках Г.1, Г.2, Г.3 показаны результаты тестирования метода поиска делением интервала поиска пополам.

7.2 Метод линейного векторного поиска

В результате тестирования метода линейного векторного поиска были получены интересные результаты, при помощи векторизации удалось добиться увеличения производительности. Так для компилятора GNU GCC C/C++ процент ускорения в среднем был 15,71%. Для компилятора Intel C/C++ Compiler прирост производительности оказался равным ~50% за счет того, что последовательная версия метода выполнялась намного дольше чем при использовании других компиляторов. В таблице 7.4 представлены результаты ускорения выполнения метода линейного векторного поиска для компиляторов GNU GCC C/C++, Intel C/C++ Compiler и LLVM/Clang.

Таблица 7.4 – Увеличение производительности векторизованного алгоритма, %

Компилятор	Увеличение производительности векторизованной версии алгоритма
GNU GCC C/C++	15.71904%
Intel C++ compiler	49.11915%
LLVM/Clang	14.55971%

Для измерения времени использовалось среднее время пятисот вызовов функции поиска, не учитывая времени первого запуска. Размер выборки составлял от 60 * 10^7 до 160 * 10^7 элементов типа double. Объем используемой оперативной памяти составил 11.92Gb для 160 * 10^7 элементов. Результаты тестирования метода линейного векторного поиска для компиляторов GNU GCC C/C++, Intel C++ compiler и LLVM/Clang представлены в таблицах 7.5, 7.6, 7.7.

Таблица 7.5 – Результат тестирования метода линейного векторного поиска с помощью компилятора GNU GCC C/C++

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,025194774	0,021733050
80	0,033549092	0,028936962
100	0,042528656	0,036523064
120	0,050197440	0,043541452
140	0,058579700	0,050794316
160	0,066909238	0,057985202

Таблица 7.6 – Результат тестирования метода линейного векторного поиска с помощью компилятора Intel C/C++ Compiler

Размер выборки 10_7 элементов	Последовательный	Векторизованный

60	0,032560576	0,021884044
80	0,043350212	0,029108068
100	0,054318944	0,036521128
120	0,065096694	0,043804000
140	0,075970244	0,050466952
160	0,086745036	0,058170138

Таблица 7.7 – Результат тестирования метода линейного векторного поиска с помощью компилятора LLVM/Clang

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,025310766	0,022168286
80	0,033803982	0,029253954
100	0,042088980	0,036770256
120	0,050530678	0,044247138
140	0,058971884	0,051545350
160	0,067396022	0,058832572

В приложении Г, на графиках Г.4, Г.4, Г.6 показаны результаты тестирования метода линейного векторного поиска.

7.3 Метод поиска делением интервала поиска на M частей

Метод поиска делением интервала поиска на M частей показал хорошее время в последовательной версии на выборке любого размера в отличие от векторизованной версии алгоритма, время выполнения которой растет линейно в зависимости от размера выборки.

Для измерения времени использовалось среднее время пятидесяти вызовов функции поиска, не учитывая времени первого запуска. Размер выборки составлял от $60 * 10^7$ до $160 * 10^7$ элементов типа double. Объем используемой оперативной памяти составил 11.92Gb для $160 * 10^7$ элементов. Результаты тестирования метода линейного векторного поиска для компиляторов GNU GCC C/C++, Intel C++ compiler и LLVM/Clang представлены в таблицах 7.8, 7.9, 7.10.

Таблица 7.8 – Результат тестирования метода поиска делением интервала поиска на M частей с помощью компилятора GNU GCC C/C++

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,000006242	4,845002799
80	0,000007758	6,462312179
100	0,000006661	8,078637342
120	0,000007420	9,684090581
140	0,000006561	11,308964438
160	0,000006561	12,938406520

Таблица 7.9 – Результат тестирования метода поиска делением интервала поиска на M частей с помощью компилятора Intel C++ compiler

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,000006242	5,149195962

80	0,000008158	6,857318301
100	0,000006661	8,571525183
120	0,000007420	10,301561561
140	0,000007420	12,012726171
160	0,000006561	13,729856539

Таблица 7.10 – Результат тестирования метода поиска делением интервала поиска на M частей с помощью компилятора LLVM/Clang

Размер выборки 10_7 элементов	Последовательный	Векторизованный
60	0,000006842	4,785742799
80	0,000008158	6,282152179
100	0,000007461	7,919593425
120	0,000007420	9,599890581
140	0,000006761	11,230964438
160	0,000006961	12,994006520

В приложении Г, на графиках Г.7, Г.8, Г.9 показаны результаты тестирования метода поиска делением интервала поиска на M частей.

7.4 Автоматическая векторизация GNU GCC C/C++

GNU Compiler Collection (GCC) – набор компиляторов, способных производить оптимизацию и векторизацию кода.

Кроме C/C++ GCC поддерживает компиляцию кода, написанного на языках:

- Objective-C;
- Fortran;

GNU GCC является свободным программным обеспечением, поддерживается и распространяется фондом свободного программного обеспечения (Free Software Foundation) на условиях, предусмотренных лицензиями GNU GPL и GNU LGPL. GCC является основным стандартным компилятором для свободно распространяемых UNIX-подобных операционных систем [4].

Автоматическая векторизация кода в GCC происходит при следующих условиях:

- рассматриваемый цикл – внутренний;
- цикл должен быть определённой структуры:
  - итератор цикла – целочисленный;
  - одна точка входа и выхода из цикла;
- тело цикла должно быть без циклов и сложных конструкций;
- итерация цикла должна быть независима от внутреннего состояния;
- типы данных внутри цикла должны быть векторизуемы;
- векторизация должны быть выгодна.

Далее рассмотрим возможности автоматической векторизации реализованных методов поиска с помощью компилятора GNU GCC C/C++. На листинге 7.1 представлено отчет компилятора о возможности автоматической векторизации метода поиска делением интервала поиска пополам.

Листинг 7.1 – Отчет компилятора о возможности автоматической векторизации метода поиска делением интервала поиска пополам.

```
bin_search_lin.c:20:5: note: not vectorized: not enough
data-refs in basic block.
bin_search_lin.c:35:23: note: not vectorized: control flow
in loop.
bin_search_lin.c:35:23: note: bad loop form.
bin_search_lin.c:32:14: note: not vectorized: not enough
data-refs in basic block.
bin_search_lin.c:24:5: note: not vectorized: not enough
data-refs in basic block.
bin_search_lin.c:35:23: note: not vectorized: not enough
data-refs in basic block.
bin_search_lin.c:35:23: note: not vectorized: not enough
data-refs in basic block.
…
bin_search_lin.c:66:5: note: no optab.
bin_search_lin.c:66:5: note: not vectorized: relevant stmt
not supported: _21 = (double) i_52;
bin_search_lin.c:66:5: note: bad operation or unsupported
loop bound.
bin_search_lin.c:50:12: note: not consecutive access x =
0.0;
bin_search_lin.c:50:12: note: not consecutive access MAX =
1000000000;
bin_search_lin.c:50:12: note: Failed to SLP the basic block.
bin_search_lin.c:50:12: note: not vectorized: failed to find
SLP opportunities in basic block.
bin_search_lin.c:50:12: note: not consecutive access x =
0.0;
bin_search_lin.c:50:12: note: not consecutive access MAX =
1000000000;
bin_search_lin.c:50:12: note: Failed to SLP the basic block.
bin_search_lin.c:50:12: note: not vectorized: failed to find
SLP opportunities in basic block.
/usr/include/x86_64-linux-gnu/bits/stdio2.h:104:10: note:
not vectorized: not enough data-refs in basic block.
```

Как видно из отчета компилятор не смог выполнить автоматическую векторизацию последовательной версии метода поиска делением интервала поиска пополам.

Рассмотрим возможности автоматической векторизации метода линейного векторного поиска с помощью компилятора GNU GCC C/C++ в листинге 7.2.

Листинг 7.2 – Отчет компилятора о возможности автоматической векторизации метода линейного векторного поиска.

```
alg2_lv.c:11:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:20:28: note: not vectorized: control flow in loop.
alg2_lv.c:20:28: note: bad loop form.
alg2_lv.c:20:9: note: not vectorized: not enough data-refs
in basic block.
```

```
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:15:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:15:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:15:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:20:28: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:28:16: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: control flow in loop.
alg2_lv.c:22:22: note: bad loop form.
alg2_lv.c:20:28: note: not vectorized: control flow in loop.
…
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:22:22: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:79:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:79:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:79:5: note: not vectorized: not enough data-refs
in basic block.
alg2_lv.c:20:28: note: not vectorized: not enough data-refs
in basic block.
/usr/include/x86_64-linux-gnu/bits/stdio2.h:104:10:   note:
not vectorized: not enough data-refs in basic block.
alg2_lv.c:123:9: note: not vectorized: not enough data-refs
in basic block.
```

Как видно из отчета компилятор не смог векторизовать алгоритм.

Рассмотрим возможности автоматической векторизации метода поиска делением интервала поиска на M частей с помощью компилятора GNU GCC C/C++ в листинге 7.3.

Листинг 7.3 – Отчет компилятора о возможности автоматической векторизации метода поиска делением интервала поиска на M частей.

```
alg_3.c:20:5: note: not vectorized: not enough data-refs in
basic block.
alg_3.c:35:23: note: not vectorized: control flow in loop.
alg_3.c:35:23: note: bad loop form.
```

```
alg_3.c:32:14: note: not vectorized: not enough data-refs in
basic block.
alg_3.c:24:5: note: not vectorized: not enough data-refs in
basic block.
alg_3.c:35:23: note: not vectorized: not enough data-refs in
basic block.
alg_3.c:35:23: note: not vectorized: not enough data-refs in
basic block.
…
alg_3.c:66:5: note: no optab.
alg_3.c:66:5: note: not vectorized: relevant stmt not
supported: _21 = (double) i_52;
alg_3.c:66:5: note: bad operation or unsupported loop bound.
alg_3.c:50:12: note: not consecutive access x = 0.0;
alg_3.c:50:12: note: not consecutive access MAX =
1000000000;
alg_3.c:50:12: note: Failed to SLP the basic block.
alg_3.c:50:12: note: not vectorized: failed to find SLP
opportunities in basic block.
alg_3.c:50:12: note: not consecutive access x = 0.0;
alg_3.c:50:12: note: not consecutive access MAX =
1000000000;
alg_3.c:50:12: note: Failed to SLP the basic block.
alg_3.c:50:12: note: not vectorized: failed to find SLP
opportunities in basic block.
/usr/include/x86_64-linux-gnu/bits/stdio2.h:104:10: note:
not vectorized: not enough data-refs in basic block.
```

Компилятор не смог векторизовать алгоритм.

7.5 Автоматическая векторизация Intel C++ compiler

Intel C/C++ Compiler – проприетарный оптимизирующий компилятор, поддерживаемый и разрабатываемый компанией Intel. Его основным преимуществом являются целевые и высокоуровневые оптимизации, специализированные под процессоры Intel [5].

Основные возможности:

- высокоуровневая оптимизация;
- межпроцедурная оптимизация;
- автоматическое распараллеливание кода;
- векторизация для SSE, SSE2, SSE3, SSE4, AVX;

В данном компиляторе автоматическая векторизация кода включается при ключе -O2 и выше.

Чтобы данный компилятор смог векторизовать циклы, они должны удовлетворять следующим критериям:

- счётчик цикла должен быть известен при входе в цикл, во время выполнения и не меняться;
- из цикла должен быть один выход.
- внутри цикла результат вычисления должен помещаться в одни и те же переменные, даже при наличии ветвления;
- цикл не может быть внешним;
- внутри цикла нет вызовов других функций.

Далее рассмотрим возможности автовекторизации реализованных методов поиска с помощью компилятора Intel C++ compiler. На листинге 7.4 представлено отчет компилятора о возможности автовекторизации метода поиска делением интервала поиска пополам.

Листинг 7.4 – Отчет компилятора о возможности автовекторизации метода поиска делением интервала поиска пополам.

```
      Intel(R) Advisor can now assist with vectorization and show
optimization
      Compiler options: -O3 -xHost -qopt-report3 -qopt-report-
phase=vec -o algicc
      Begin optimization report for: main()
          Report from: Vector optimizations [vec]
      LOOP BEGIN at alg.c(111,9)
      <Peeled loop for vectorization>
      LOOP END
      LOOP BEGIN at alg.c(115,9)
         remark #15542: loop was not vectorized: inner loop was already
vectorized

         LOOP BEGIN at alg.c(50,9) inlined into alg.c(118,35)

      remark #15520: loop was not vectorized: loop with multiple exits
cannot be vectorized unless it meets search loop idiom criteria   [
alg.c(50,42) ]
         LOOP BEGIN at alg.c(56,17) inlined into alg.c(118,35)
                  remark #15344: loop was not vectorized: vector
         dependence prevents vectorization. First dependence is shown
         below. Use level 5 report for details
                  remark #15346: vector dependence: assumed FLOW
         dependence between vmask1 (63:34) and vmask1 (63:34)
               LOOP END
            LOOP END
         LOOP BEGIN at alg.c(28,9) inlined into alg.c(80,30)
            remark #15300: LOOP WAS VECTORIZED
            remark #15442: entire loop may be executed in remainder
            remark #15450: unmasked unaligned unit stride loads: 1
            remark #15475: --- begin vector cost summary ---
            remark #15476: scalar cost: 11
            remark #15477: vector cost: 5.250
            remark #15478: estimated potential speedup: 2.050
            remark #15488: --- end vector cost summary ---
         LOOP END
         LOOP BEGIN at alg.c(28,9) inlined into alg.c(80,30)
         <Remainder loop for vectorization>
         LOOP END
```

Из отчета видно, что компилятор не смог выполнить автоматическую векторизацию последовательной версии метода поиска делением интервала поиска пополам.

Рассмотрим возможности автовекторизации метода линейного векторного поиска с помощью компилятора Intel C++ compiler в листинге 7.5.

Листинг 7.5 – Отчет компилятора о возможности автовекторизации метода линейного векторного поиска.

```
Intel(R) Advisor can now assist with vectorization and show
optimization
Intel(R) C Intel(R) 64 Compiler for applications running on
Intel(R) 64, Version 17.0.0.098 Build 20160721
Compiler options: -O3 -xHost -qopt-report3 -qopt-report-
phase=vec -o alg2icc2

Begin optimization report for: main()

    Report from: Vector optimizations [vec]
LOOP BEGIN at alg2.c(111,9)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at alg2.c(111,9)
   remark #15410: vectorization support: conversion from int
to float will be emulated   [ alg2.c(112,25) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 5
   remark #15477: vector cost: 3.250
   remark #15478: estimated potential speedup: 1.480
   remark #15487: type converts: 1
   remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at alg2.c(111,9)
<Remainder loop for vectorization>
LOOP END

   LOOP BEGIN at alg2.c(28,9) inlined into alg2.c(130,34)
      remark #15300: LOOP WAS VECTORIZED
      remark #15442: entire loop may be executed in remainder
      remark #15450: unmasked unaligned unit stride loads: 1
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 11
      remark #15477: vector cost: 5.250
      remark #15478: estimated potential speedup: 2.050
      remark #15488: --- end vector cost summary ---
   LOOP END

   LOOP BEGIN at alg2.c(28,9) inlined into alg2.c(130,34)
   <Remainder loop for vectorization>
   LOOP END
LOOP END

Begin optimization report for: linear_search_ref(const double
*, double, long long)

    Report from: Vector optimizations [vec]
```

```
LOOP BEGIN at alg2.c(28,9)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at alg2.c(28,9)
   remark #15300: LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
remark #15450: unmasked unaligned unit stride loads: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 11
   remark #15477: vector cost: 5.250
   remark #15478: estimated potential speedup: 2.050
   remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at alg2.c(28,9)
<Remainder loop for vectorization>
LOOP END
```

Intel C++ compiler смог выполнить автоматическую векторизацию последовательной версии метода линейного векторного поиска.

Рассмотрим возможности автовекторизации метода поиска делением интервала поиска на M частей с помощью компилятора Intel C++ compiler в листинге 7.6.

Листинг 7.6 – Отчет компилятора о возможности автовекторизации метода поиска делением интервала поиска на M частей.

```
Intel(R) Advisor can now assist with vectorization and show
optimization report messages with your source code.
Compiler options: -O3 -xHost -qopt-report3 -qopt-report-
phase=vec -o alg3icc2

Begin optimization report for: main()
    Report from: Vector optimizations [vec]
LOOP BEGIN at alg3.c(182,9)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at alg3.c(182,9)
<Remainder loop for vectorization>
LOOP END
LOOP BEGIN at alg3.c(125,17) inlined into alg3.c(197,40)
   remark #15344: loop was not vectorized: vector dependence
prevents vectorization. First dependence is shown below. Use
level 5 report for details
   remark #15346: vector dependence: assumed FLOW dependence
between tmpT[0] (130:25) and tmpX[0] (142:38)
LOOP END

LOOP BEGIN at alg3.c(87,25) inlined into alg3.c(197,40)
```

```
   remark #15344: loop was not vectorized: vector dependence
prevents vectorization. First dependence is shown below. Use
level 5 report for details
   remark #15346: vector dependence: assumed FLOW dependence
between tmpT[0] (92:33) and tmpX[0] (101:46)
LOOP END
Non-optimizable loops:
LOOP BEGIN at alg3.c(195,9)
   remark  #15536:  loop  was  not  vectorized:  inner  loop
throttling prevents vectorization of this outer loop. Refer to
inner loop message for more details.   [ alg3.c(165,22) ]

   LOOP BEGIN at alg3.c(165,22) inlined into alg3.c(197,40)
      remark #15522: loop was not vectorized: loop control
flow is too complex. Try using canonical loop form from OpenMP
specification
   LOOP END
LOOP END
Begin optimization report for: LUF(double, double *, int)

    Report from: Vector optimizations [vec]
Begin optimization report for: LUF_AVX(double, double *, long
long)
    Report from: Vector optimizations [vec]
LOOP BEGIN at alg3.c(125,17)
   remark #15344: loop was not vectorized: vector dependence
prevents vectorization. First dependence is shown below. Use
level 5 report for details
   remark #15346: vector dependence: assumed FLOW dependence
between tmpT[0] (130:25) and tmpX[0] (142:38)
LOOP END

LOOP BEGIN at alg3.c(87,25)
   remark #15344: loop was not vectorized: vector dependence
prevents vectorization. First dependence is shown below. Use
level 5 report for details
   remark #15346: vector dependence: assumed FLOW dependence
between tmpT[0] (92:33) and tmpX[0] (101:46)
LOOP END
Non-optimizable loops:
LOOP BEGIN at alg3.c(165,22)
   remark #15522: loop was not vectorized: loop control flow
is too complex. Try using canonical loop form from OpenMP
specification
LOOP END
```

Из листинга видно, что компилятор не выполнил автоматическую векторизацию последовательной версии метода поиска делением интервала поиска на M частей.

7.6 Автоматическая векторизация LLVM/Clang

LLVM (Low Level Virtual Machine) представляет собой универсальную систему анализа, трансформации и оптимизации программ. Он был разработан в Университете Иллинойса группой разработчиков, возглавляемой Крисом

Латтенером. В настоящее время LLVM используется и разрабатывается ведущими компаниями, такими как Apple и Adobe.

В основе LLVM лежит промежуточное представление кода, над которым можно производить изменения во время компиляции, линковки и выполнения.

Clang – представляет из себя транслятор в байт-код LLVM для языков высокого уровня, таких как Си и C++

После трансляции кода фреймворк LLVM производит последующие оптимизацию, векторизацию и кодогенерацию.

Целью разработки данного транслятора и фреймворка является замена компилятора GCC. В отличие от GCC, связка фреймворка LLVM и транслятора Clang стремится предоставить программисту и среде разработки универсальный фреймворк для парсинга, анализа и компиляции кода[6].

На данный момент LLVM/Clang поддерживает векторизацию в следующих случаях:

- циклы с неизменным количеством итераций;
- циклы с редукцией;
- циклы с ветвлением;
- циклы с обратной итерацией;
- циклы с разными типами данных;
- векторизация вызванных внутри цикла функций;
- циклы с частичной развёрткой (unrolling)

Далее будут рассмотрены возможности автовекторизации реализованных методов поиска с помощью компилятора LLVM/Clang. На листинге 7.7 представлено отчет компилятора о возможности автовекторизации метода поиска делением интервала поиска пополам.

Листинг 7.7 – Отчет компилятора о возможности автовекторизации метода поиска делением интервала поиска пополам.

```
alg.cpp:30:9: remark: loop not vectorized: could not determine
 number of loop iterations [-Rpass-analysis=loop-vectorize]
while (l + 1 != u)
alg.cpp:30:9: remark: loop not vectorized: use -Rpass-
 analysis=loop-vectorize for more info [-Rpass-missed=loop-
 vectorize]
alg.cpp:52:9: remark: loop not vectorized: loop control flow is
 not understood by vectorizer [-Rpass-analysis=loop-vectorize]
for (long long j = powerOfTwo / 2; j >= 1; j /= 2)
alg.cpp:52:9: remark: loop not vectorized: use -Rpass-
 analysis=loop-vectorize for more info [-Rpass-missed=loop-
 vectorize]
alg.cpp:94:9: remark: vectorized loop (vectorization width: 4,
 interleaved count:4) [-Rpass=loop-vectorize]
for (int i = 0; i < N; i++)
alg.cpp:30:9: remark: loop not vectorized: could not determine
 number of loop iterations [-Rpass-analysis=loop-vectorize]
while (l + 1 != u)
alg.cpp:30:9: remark: loop not vectorized: use -Rpass-
 analysis=loop-vectorize for more info [-Rpass-missed=loop-
 vectorize]
```

```
alg.cpp:52:9: remark: loop not vectorized: loop control flow is
 not understood by vectorizer [-Rpass-analysis=loop-vectorize]
for (long long j = powerOfTwo / 2; j >= 1; j /= 2)
```

Из листинга можно заметить, что компилятор не смог векторизовать алгоритм.

Рассмотрим возможности автовекторизации метода линейного векторного поиска с помощью компилятора LLVM/Clang в листинге 7.8.

Листинг 7.8 – Отчет компилятора о возможности автовекторизации метода линейного векторного поиска.

```
alg2.c:28:9: remark: loop not vectorized: loop control flow is
not understood by vectorizer [-Rpass-analysis=loop-vectorize]
        for (i = 0; i < n; ++i)
alg2.c:39:5: remark: loop not vectorized: loop control flow is
not understood by vectorizer [-Rpass-analysis=loop-vectorize]
int linear_search(const double *A, double key, long long n)

alg2.c:28:9: remark: loop not vectorized: loop control flow is
not understood by  vectorizer [-Rpass-analysis=loop-vectorize]
        for (i = 0; i < n; ++i)
alg2.c:28:9: remark: loop not vectorized: loop control flow is
not understood by  vectorizer [-Rpass-analysis=loop-vectorize]
alg2.c:111:9:   remark:   the   cost-model   indicates   that
vectorization   is   not   beneficial   [-Rpass-analysis=loop-
vectorize]
        for ( i = 0; i < MAX; i++)
alg2.c:111:9:   remark:   the   cost-model   indicates   that
interleaving   is   not   beneficial   [-Rpass-analysis=loop-
vectorize]
alg2.c:28:9: remark: loop not vectorized: loop control flow is
not understood by vectorizer [-Rpass-analysis=loop-vectorize]
        for (i = 0; i < n; ++i)
alg2.c:28:9: remark: loop not vectorized: loop control flow is
not understood by vectorizer [-Rpass-analysis=loop-vectorize]
alg2.c:118:35: remark: loop not vectorized: loop control flow
is   not   understood   by   vectorizer   [-Rpass-analysis=loop-
vectorize]
           printf("Vec  -  %lli\n",  linear_search(t_data,  x,
MAX));
alg2.c:28:9: remark: loop not vectorized: loop control flow is
not understood by vectorizer [-Rpass-analysis=loop-vectorize]
        for (i = 0; i < n; ++i)
```

Компилятор не смог векторизовать алгоритм.

Рассмотрим возможности автовекторизации метода поиска делением интервала поиска на M частей с помощью компилятора LLVM/Clang в листинге 7.9.

Листинг 7.9 – Отчет компилятора о возможности автовекторизации метода поиска делением интервала поиска на M частей.

```
alg3.c:46:17: remark: loop not vectorized: loop control flow
is not understood by vectorizer [-Rpass-analysis=loop-
vectorize]
                for (int i = LUF; i < NBIG; i += L)
alg3.c:46:17: remark: loop not vectorized: use -Rpass-
analysis=loop-vectorize for more info [-Rpass-missed=loop-
vectorize]
alg3.c:35:25: remark: loop not vectorized: loop control flow
is not understood by vectorizer [-Rpass-analysis=loop-
vectorize]
for (int i = LUF; i < NBIG; i++)
alg3.c:35:25: remark: loop not vectorized: use -Rpass-
analysis=loop-vectorize for more info [-Rpass-missed=loop-
vectorize]
alg3.c:132:33: remark: loop not vectorized: control flow
cannot be substituted for a select [-Rpass-analysis=loop-
vectorize]
tmpT[1] = T[i + L];

alg3.c:125:17: remark: loop not vectorized: use -Rpass-
analysis=loop-vectorize for more info [-Rpass-missed=loop-
vectorize]
      for (long long i = LUF; i < LUF + N - 1; i += L * 2)
alg3.c:66:5: remark: loop not vectorized: loop control flow
is not understood by vectorizer [-Rpass-analysis=loop-
vectorize]
int LUF_AVX(double X, double* T, long long NBIG)
alg3.c:87:25: remark: loop not vectorized: use -Rpass-
analysis=loop-vectorize for more info [-Rpass-missed=loop-
vectorize]
for (long long i = LUF; i < NBIG - 1; i += 2)
alg3.c:182:9: remark: the cost-model indicates that
vectorization is not beneficial [-Rpass-analysis=loop-
vectorize]
for(long long i = 0; i < N; i++)
alg3.c:182:9: remark: the cost-model indicates that
interleaving is not beneficial [-Rpass-analysis=loop-
vectorize]
printf("LUF_AVX - %lli\n", LUF_AVX(key, arr, N));
alg3.c:87:25: remark: loop not vectorized: use -Rpass-
analysis=loop-vectorize for more info [-Rpass-missed=loop-
vectorize]
for (long long i = LUF; i < NBIG - 1; i += 2)
alg3.c:132:33: remark: loop not vectorized: control flow
cannot be substituted for a select [-Rpass-analysis=loop-
vectorize]
tmpT[1] = T[i + L];
alg3.c:125:17: remark: loop not vectorized: use -Rpass-
analysis=loop-vectorize for more info [-Rpass-missed=loop-
vectorize]
```

Компилятор не смог векторизовать алгоритм.

8 ЗАКЛЮЧЕНИЕ

В результате проделанной работы были реализованы следующие методы поиска в таблице:

- Метод поиска делением интервала поиска пополам
- Метод линейного векторного поиска
- Метод поиска делением интервала поиска на M частей

Реализованы последовательные и векторизованные версии алгоритмов на языке Си с использованием функций-интринсиков, проведен ряд экспериментов, проанализированы возможности ускорения при векторизации методов поиска в одномерных таблицах. Исследованы возможности автоматической векторизации методов поиска при помощи компиляторов Intel C/C++ Compiler, GNU GCC C/C++, LLVM/CLANG.

ПРИЛОЖЕНИЕ А

(Библиография)

1 Параллельные вычисления/Под ред. Г. Родрига: Пер. с англ./Под ред. Ю.Г. Дадаева. – М.: Наука. Гл. ред. Физ.-мат.лит., 1986.-376с.

2 Вычислительный кластер F (Oak). Конфигурация кластерной ВС Oak. URL: http://cpct.sibsutis.ru/index.php/Main/Oak. (дата обращения 10.06.2017)

3 Курносов Михаил Георгиевич, Измерение времени выполнения кода. URL: https://www.mkurnosov.net/teaching/index.php/DSA/CodeSnippetWtime. (дата обращения: 23.05.2017)

4 Auto-vectorization in GCC. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html. (дата обращения 07.06.2017)

5 A Guide to Vectorization with Intel® C++ Compilers. URL: https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf. (дата обращения 11.06.2017)

6 Auto-Vectorization in LLVM. URL: http://llvm.org/docs/Vectorizers.html. (дата обращения 13.06.2017)

7 Хорошевский, В.Г. Архитектура вычислительных систем :Учеб.пособие / В.Г. Хорошевский . – 2-е изд., перераб. и доп. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2008. – 520 с. – (Информатика в техническом университете, ISBN 987-5-7038-3175-5)

8 А.В.Комолкин, С.А.Немнюгин, Программирование для высокопроизводительных ЭВМ, 2.2.3 Векторная обработка данных. URL: https://www2.sscc.ru/Links/Litera/hpc/g2.2.3.html. (дата обращения: 7.05.2017)

9 Курносов Михаил Георгиевич, Лекция 4 Векторизация кода (code vectorization: SSE/AVX), 2015. URL: http://www.mkurnosov.net/teaching/uploads/HPC/hpcs-fall2015-lec4.pdf. (дата обращения: 15.03.2017)

10 Блог компании Intel, andrei_an, Некоторые простейшие принципы автовекторизации, 18 марта 2013. URL: https://habrahabr.ru/company/intel/blog/171439. (дата обращения: 17.05.2017)

11 Курносов Михаил Георгиевич, Лекция Архитектурно-ориентированная оптимизация программного обеспечения (software optimization introduction), 2015. URL: https://www.mkurnosov.net/teaching/uploads/HPC/hpcs-fall2015-lec1.pdf. (Дата обращения: 24.04.2017)

12 Сайт компании Intel. Intel intrinsics Guide. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (дата обращения 07.03.2017)

13 Блог компании Intel, @ivorobts, Разработчик на распутье: как векторизовать?!, 16 декабря 2013. URL: https://habrahabr.ru/company/intel/blog/171439. (дата обращения: 17.05.2017)

14 Сайт компании Intel, Chris Lomont. Introduction to Intel® Advanced Vector Extensions. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf. (дата обращения 17.03.2017)

15 Сайт компании Intel, Copyright © 2010, Intel Corporation. A Guide to Vectorization with Intel® C++ Compilers. URL:

https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf. (дата обращения 29.04.2017)
16 Database Architects. Trying to speed up Binary Search, September 12, 2015. URL: http://databasearchitects.blogspot.ru/2015/09/trying-to-speed-up-binary-search.html. (дата обращения 14.05.2017)
17 Сайт компании Intel. Optimizing Performance with Intel Advanced Vector Extensions, September 2014. URL: https://computing.llnl.gov/tutorials/linux_clusters/intelAVXperformanceWhitePaper.pdf. (дата обращения 20.05.2017)
18 Intel Developer Zone. Intrinsics for Intel® C++ Compilers. URL: https://software.intel.com/ru-ru/node/682974. (дата обращения 02.06.2017)
19 Dubois P.F. Inverse equation of state look-up. Rep UCRL-52563. – Livermore: Lawrence Livermore National Laboratory, 1978.
20 Dubois P.F., Kohn J.R. Equation of state table look-up: A case study in vectorization. Rep. UCRL-81184. – Livermore: Lawrence Livermore National Laboratory, 1978.
21 Dubois P.F., Rodrigue G.H. – In: High speed Computer and Algorithm Organization/Eds. D.J. Kuck, D.H. Lawrie, A.H. Sameh. New York: Academic Press, 1977.
22 Fong K., Jordan T.L. Some linear algebraic algorithms and their performance on Cray-1. Rep LA-6774. – Los Almos: Los Almos National Laboratory, 1977.
23 Knuth D.E. Art of Computer Programming. V.3. – Reading: Addison-Wesley, 1968.
24 McMahon F.H., Sloan L.J., Long G.A. Stacklibe, a vector funcktion library of optimum stackloops for the CDC7600. Rep UCID-30083. – Livermore: Lawrence Livermore National Laboratory, 1972.

ПРИЛОЖЕНИЕ Б

Наиболее употребляемые текстовые сокращения

ВС – вычислительная система
ЭВМ – Электронно-вычислительная машина
ЦП – центральный процессор
SSE – Streaming SIMD Extensions
AVX – Advanced Vector Extensions
GNU – GNU’s Not UNIX
GCC – GNU Compiler Collection
LLVM – Low Level Virtual Machine
AES – Advanced Encryption Standard
МВС – многопроцессорных вычислительных систем
MMX – Multimedia Extensions
SIMD – Single Instruction, Multiple Data
AMD – Advanced Micro Devices
LUT – lookup table
CORDIC - COordinate Rotation DIgital Computer
FMA – Fused-Multiply-Add

ПРИЛОЖЕНИЕ В

Листинги реализованного ПО

Листинг В.1 – Последовательный метод поиска делением интервала поиска пополам

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdbool.h>
#include <sys/time.h>
#include <immintrin.h>

double wtime()
{
  struct timeval t;
  gettimeofday(&t, NULL);
  return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int BSearch(long long a, double mass[], double n)
{
  long long low, high, middle;
  low = 0;
  high = n - 1;
  while (low <= high)
  {
    middle = (low + high) / 2;
    if (a < mass[middle])
      high = middle - 1;
    else if (a > mass[middle])
      low = middle + 1;
    else
      return middle;
  }
  return -1;
}
int main ()
{
  double x = 0;
  long long MAX = 1000000000;
  long long i = 0, j = 0;
  double t;
  double *t_data;

  printf("Enter X: ");
  scanf("%lf", &x);
  printf("X = %lf\n\n", x);

  printf("Enter MAX table size: ");
  scanf("%lli", &MAX);
  printf("MAX = %lli\n\n", MAX);
```

```
  t_data = (double*)malloc(MAX * sizeof(double));

  for ( i = 0; i < MAX; i++)
    t_data[i] = i;

  for ( int k = 0; k < 10; k++ ) {
    t = wtime();
    j = BSearch( x, t_data, MAX);
    printf("Test %i:\n", k);
    printf("%lli\n\n", j);
    t = wtime() - t;
    printf("Binary search(linear): %.15f\n\n", t);
   }
  free(t_data);
  return 0;
}
```

Листинг В.2 – Векторизованный метод поиска делением интервала поиска пополам

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdbool.h>
#include <sys/time.h>
#include <immintrin.h>

int BSearch_AVX(int *data, int N, int key)
{
  double powerOfTwo = pow(2, floor(log2(N - 1)));
  int splitIndex = N - powerOfTwo;
  double splitValue = data[splitIndex];
  int p = splitIndex * (key >= splitValue);
  double *store_p = (double*)malloc(7 * sizeof(double));
  *store_p = p;
  if (data[N - 1] < key)
          return -1;
  for (int j = powerOfTwo / 2; j >= 1; j /= 2) {
     __m256d xm_jvec = _mm256_set_pd(j + j / 2 + j / 4 + j / 8,
j + j / 2 + j / 4, j + j / 2, j);
     __m256d xm_idxvec = _mm256_set1_pd(p);
     int cmpval0 = data[p + j];
     int cmpval1 = data[p + j + j / 2];
     int cmpval2 = data[p + j + j / 2 + j / 4];
     int cmpval3 = data[p + j + j / 2 + j / 4 + j / 8];

     __m256d xm_cmpvalvec = _mm256_set_pd(cmpval3, cmpval2,
cmpval1, cmpval0);
     __m256d xm_valvec = _mm256_set1_pd(key);
     xm_idxvec = _mm256_add_pd(
               xm_idxvec, _mm256_andnot_pd(
                   _mm256_cmp_pd(xm_valvec, xm_cmpvalvec,
_CMP_LT_OQ),
                     xm_jvec));
```

```
    _mm256_store_pd(store_p+0, xm_idxvec);

    for (int i = 0; i < 4; i++) {
      if (key >= store_p[i])
      {
              p = (int)store_p[i];
              break;
      }
    }
  }
  return p;
}

int main()
{
int key = 123000;
int MAX = 1000000;

int *data = (int*)malloc(MAX * sizeof(int));
for (int i = 0; i < MAX; i++) {
      data[i] = i;
}
printf("%d\n", BSearch_AVX(data, MAX, key));
free(data);
return 0;
}
```

Листинг В.3 – Метод линейного векторного поиска (последовательная и векторизованная версия)

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdbool.h>
#include <sys/time.h>
#include <immintrin.h>

double wtime()
{
struct timeval t;
gettimeofday(&t, NULL);
return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int linear_search_ref(const double *A, double key, long long n)
{
    long long result = -1;
    long long i;

    for (i = 0; i < n; ++i)
    {
      if (A[i] >= key)
      {
        result = i;
        break;
      }
    }
    return result;
}

int linear_search(const double *A, double key, long long n)
{
#define VEC_INT_ELEMS 4
#define BLOCK_SIZE (VEC_INT_ELEMS * 32)
#define CODE_OF_NOT_EQ 15
    const __m256d vkey = _mm256_set1_pd(key);
    long long vresult = -1;
    long long result = -1;
    long long i, j;
    bool breakf = false;

    for (i = 0; i <= n - BLOCK_SIZE; i += BLOCK_SIZE)
    {
      __m256d vmask0 = _mm256_set1_pd(-1);
      __m256d vmask1 = _mm256_set1_pd(-1);
      int mask0, mask1;

      for (j = 0; j < BLOCK_SIZE; j += VEC_INT_ELEMS * 2)
      {
        __m256d vA0 = _mm256_set1_pd(A[i + j]);
        __m256d vA1 = _mm256_set1_pd(A[i + j + VEC_INT_ELEMS]);
```

```
        __m256d vcmp0 = _mm256_cmp_pd(vkey, vA0, _CMP_GE_OS);
        __m256d vcmp1 = _mm256_cmp_pd(vkey, vA1, _CMP_GE_OS);
        vmask0 = _mm256_and_pd(vmask0, vcmp0);
        vmask1 = _mm256_and_pd(vmask1, vcmp1);

      }
      mask0 = _mm256_movemask_pd(vmask0);
      mask1 = _mm256_movemask_pd(vmask1);
      if ((mask0 & mask1) != CODE_OF_NOT_EQ)
      {
              vresult = i;
              break;
      }
    }
    if (vresult > -1)
    {
      result = vresult + linear_search_ref(&A[vresult], key,
BLOCK_SIZE);
    }
    else if (i < n)
    {
        result = i + linear_search_ref(&A[i], key, n - i);
    }
    return result;
#undef BLOCK_SIZE
#undef VEC_INT_ELEMS
}

int main()
{
    double x = 0;
    long long MAX = 1000000000;
    long long i = 0;

    double t = 0, t1 = 0;
    double *t_data;

    t_data = (double*)malloc(MAX * sizeof(double));

    printf("Enter X: ");
    scanf("%lf", &x);
    printf("X = %lf\n\n", x);

    printf("Enter MAX table size: ");
    scanf("%lli", &MAX);
    printf("MAX = %lli\n\n", MAX);



    for ( i = 0; i < MAX; i++)
        t_data[i] = i;

    for ( int j = 0; j < 5; j++ )
    {
```

```
        printf("Test %i:\n", j);
        t = wtime();
        printf("X = %d\n", linear_search(t_data, x, MAX));
        t = wtime() - t;
        printf("Linear search: %.12f\n\n", t);

        t1 = 0;
        t1 = wtime();
        printf("X = %d\n", linear_search_ref(t_data, x, MAX));
        t1 = wtime() - t1;
        printf("%.12f\n\n", t1);
       }


    free(t_data);

return 0;
}
```

Листинг В.4 - Метод поиска делением интервала поиска на M частей (последовательная и векторизованная версия)

```
#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <stdbool.h>

#include <sys/time.h>

#include <immintrin.h>

double wtime()

{

  struct timeval t;

  gettimeofday(&t, NULL);

  return (double)t.tv_sec + (double)t.tv_usec * 1E-6;

}

int LUF_AVX(double X, double* T, long long NBIG)

{

  long long M = 33;

  long long N = NBIG;

  long long LUF = 0;
```

```
if (N <= 0) return -1;
do {
  long long L = (N - 1) / (M - 1);
  if (L == 0)
  {
    long long K = 0;
    double *tmpX = (double*)malloc(2 * sizeof(double));
    double *tmpT = (double*)malloc(2 * sizeof(double));
    for (long long i = LUF; i < NBIG-1; i+=2)
    {
        __m256d mT;
        __m256d mX;
        tmpT[0] = T[i];
        tmpT[1] = T[i + 1];
        mT = _mm256_load_pd(tmpT);
        tmpX[0] = X;
        tmpX[1] = X;
        mX = _mm256_load_pd(tmpX);
        __m256d cm = _mm256_max_pd(mT, mX);
        _mm256_store_pd(tmpX, cm);
        if (tmpX[0] == X) {
                K = K + 1;
        }
        if (tmpX[1] == X) {
                K = K + 1;
        }
    }
    free(tmpX);
    free(tmpT);
```

```
  LUF = LUF + K;
  return LUF;
}
long long K = 0;
double *tmpX = (double*)malloc(2 * sizeof(double));
double *tmpT = (double*)malloc(2 * sizeof(double));
for (long long i = LUF; i < LUF+N-1; i += L*2)
{
        __m256d mT;
        __m256d mX;
        tmpT[0] = T[i];
        if(i + L < NBIG)
                tmpT[1] = T[i + L];
        else tmpT[1] = T[i];
        mT = _mm256_load_pd(tmpT);
        tmpX[0] = X;
        tmpX[1] = X;
        mX = _mm256_load_pd(tmpX);
        __m256d cm = _mm256_max_pd(mT, mX);
        _mm256_store_pd(tmpX, cm);
        if (tmpX[0] == X) {
                K = K + 1;
        }
        if (tmpX[1] == X && i+L < NBIG) {
                K = K + 1;
        }
}
free(tmpX);
```

```
    free(tmpT);
    long long J = 1 + (K - 1)*L;
    if (K == 0)
            return LUF;
    else
            LUF = LUF + J;
    if (K == M)
            N = N - J;
    else
            N = L - 1;
  } while (N > 1);
  return LUF;
}
int LUF(double X, double T[], long long NBIG)
{
      long long M = 33;
      long long N = NBIG;
      long long LUF = 0;
      if (N <= 0) return -1;
      do {
              long long L = (N - 1)/(M - 1);
              if (L == 0)
              {
                      long long K = 0;
                      for (long long i = LUF; i < NBIG; i++)
                      {
                              if (T[i] <= X) {
                                      K = K + 1;
                              }
```

```
                        else break;
                }
                LUF = LUF + K;
                return LUF;
        }
        long long K = 0;
        for (long long i = LUF; i < LUF+N; i += L)
        {
                if (T[i] <= X) {
                        K = K + 1;
                }
                else break;
        }
        long long J = 1 + (K - 1)*L;
        if (K == 0)
                return LUF;
        else
                LUF = LUF + J;
        if (K == M)
                N = N - J;
        else
                N = L - 1;
    } while (N > 1);
    return LUF;
}
int main()
{
    double x = 7500;
    long long i = 0;
```

```
        long long MAX = 8000;
        double *arr = (double*)malloc(MAX * sizeof(double));

        for ( i = 0; i < MAX; i++ )
        {
                arr[i] = i;
        }
        double t = 0;
        t = wtime();
        printf("LUF_AVX - %d\n", LUF_AVX(x, arr, MAX));
        t = wtime() - t;
        printf("%.12f\n", t);
        t = 0;
        t = wtime();
        printf("LUF - %d\n", LUF(x, arr, MAX));
        t = wtime() - t;
        printf("%.12f\n", t);
        free(arr);
  return 0;
}
```

ПРИЛОЖЕНИЕ Г

Результаты тестирования методов поиска

Результат тестирования метода поиска делением интервала поиска пополам при помощи компилятора GNU GCC C/C++ для последовательной и векторизованной версии представлен на рисунке Г.1.

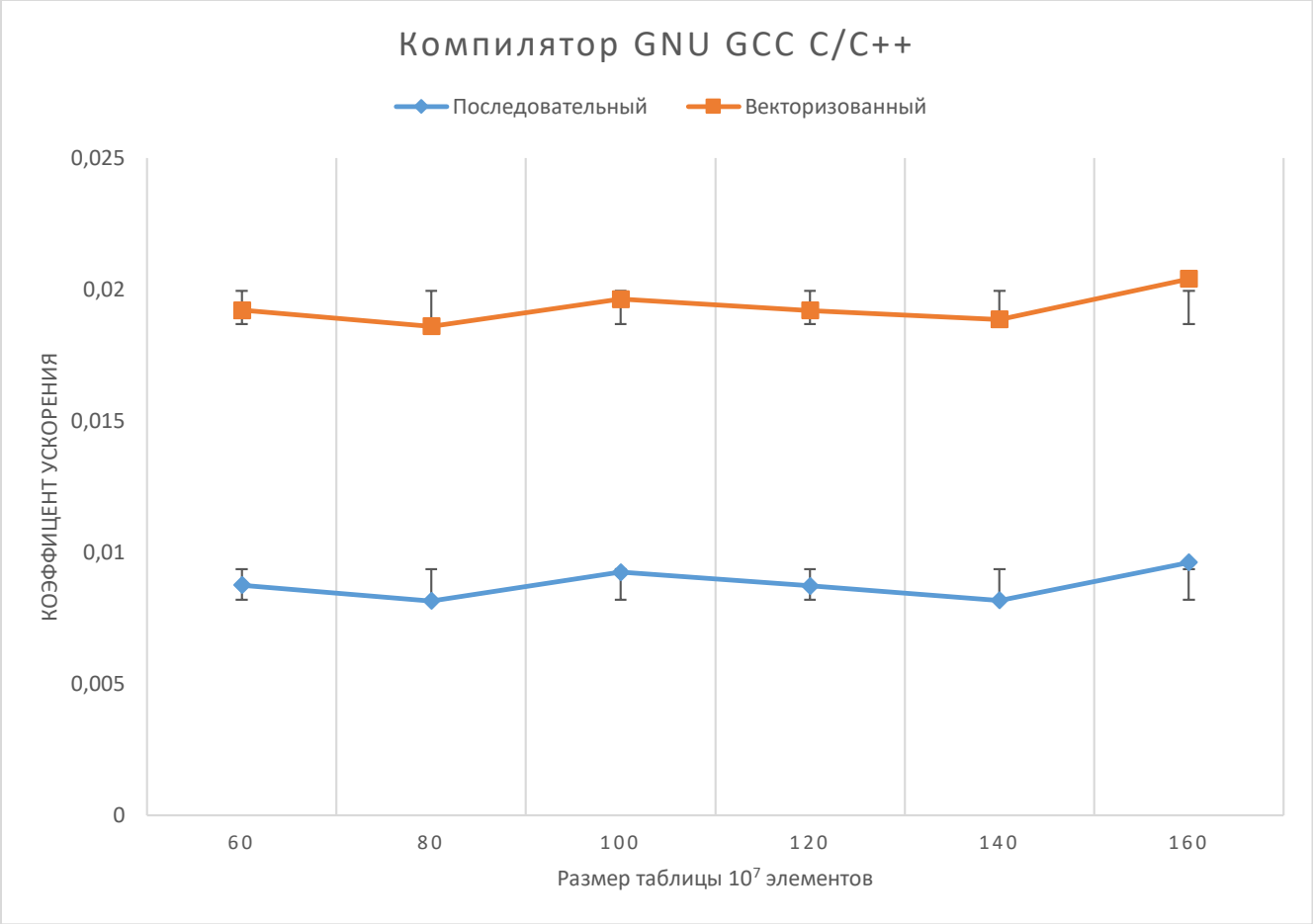

Рисунок Г.1 - тестирование метода поиска делением интервала поиска пополам при помощи компилятора GNU GCC C/C++

Результат тестирования метода поиска делением интервала поиска пополам при помощи компилятора Intel® C++ Compiler для последовательной и векторизованной версии представлен на рисунке Г.2.

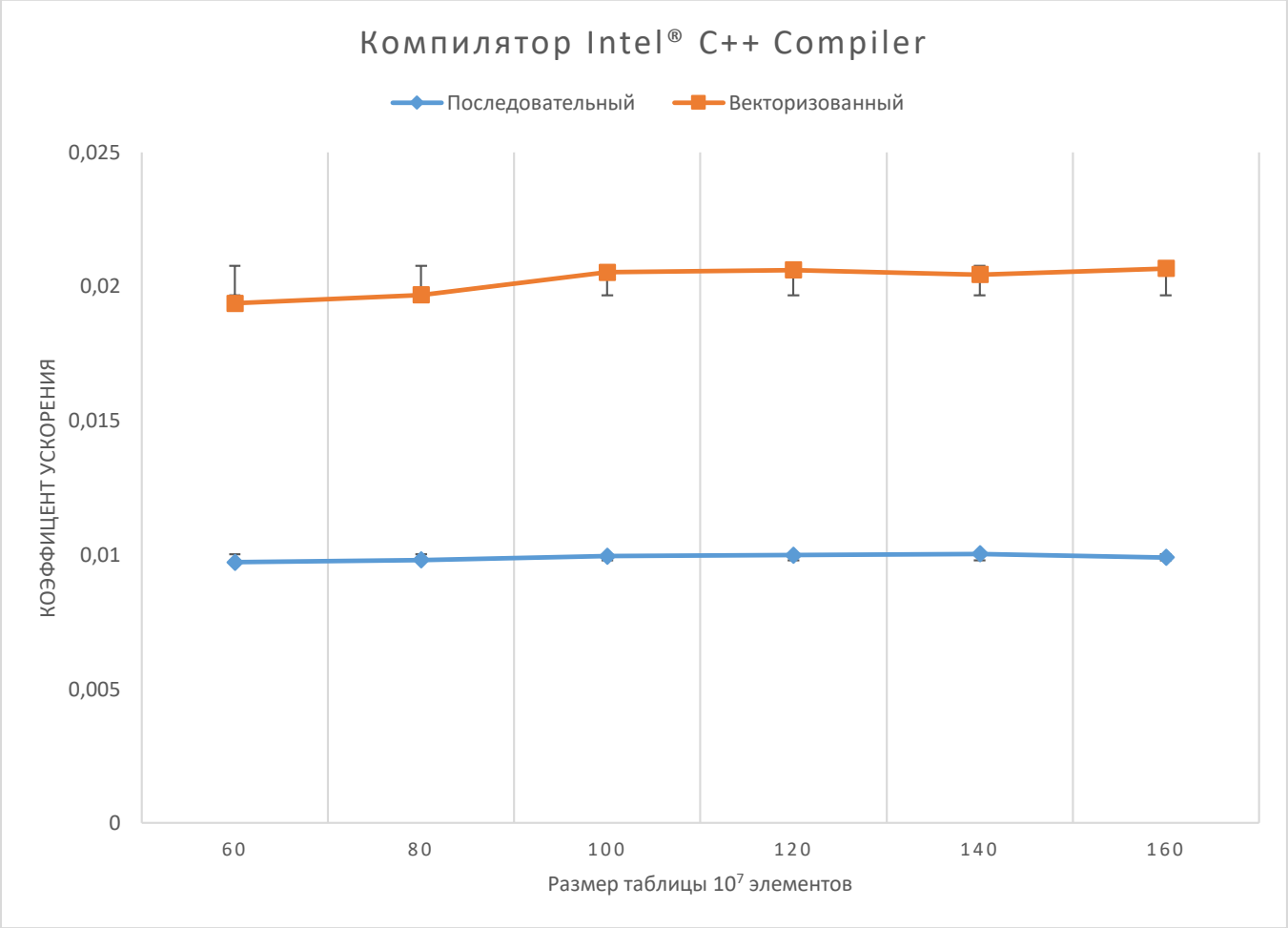

Рисунок Г.2 - тестирование метода поиска делением интервала поиска пополам при помощи компилятора Intel® C++ Compiler

Результат тестирования метода поиска делением интервала поиска пополам при помощи компилятора LLVM/CLANG для последовательной и векторизованной версии представлен на рисунке Г.3.

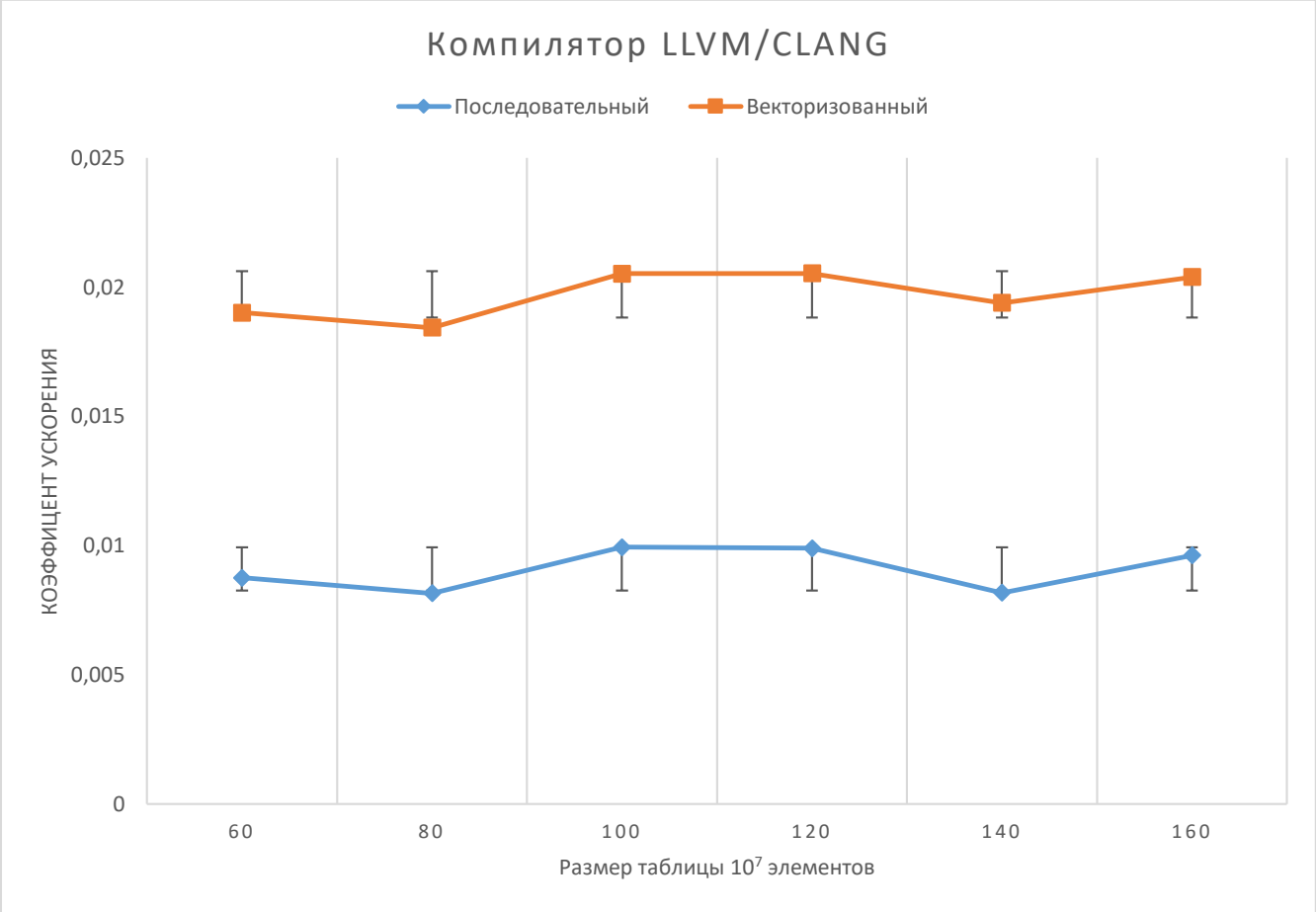

Рисунок Г.3 - тестирование метода поиска делением интервала поиска пополам при помощи компилятора LLVM/CLANG

Результат тестирования метода линейного векторного поиска при помощи компилятора GNU GCC C/C++ для последовательной и векторизованной версии представлен на рисунке Г.4.

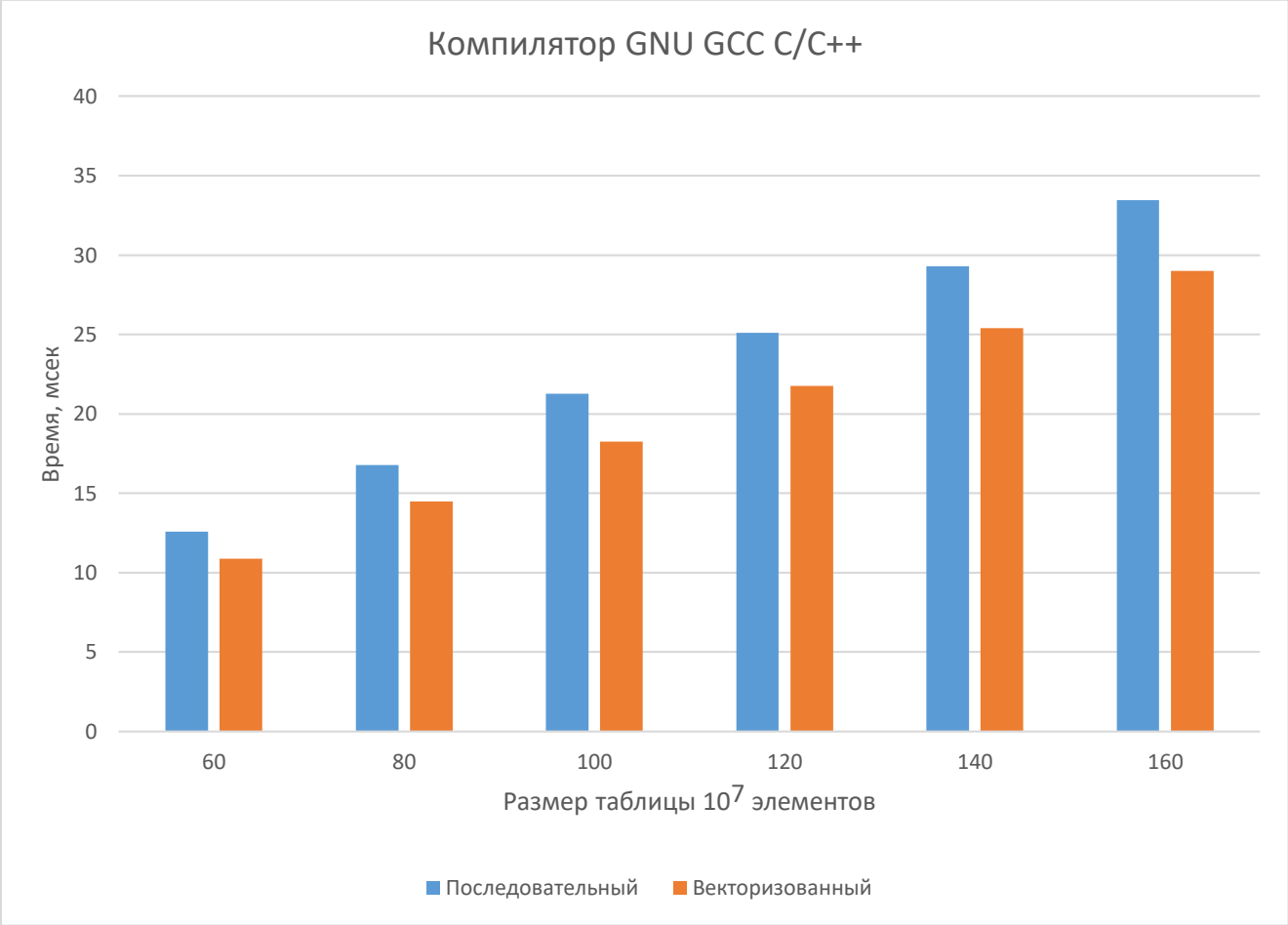

Рисунок Г.4 - тестирование метода линейного векторного поиска при помощи компилятора GNU GCC C/C++

Результат тестирования метода линейного векторного поиска при помощи компилятора Intel® C++ Compiler для последовательной и векторизованной версии представлен на рисунке Г.5.

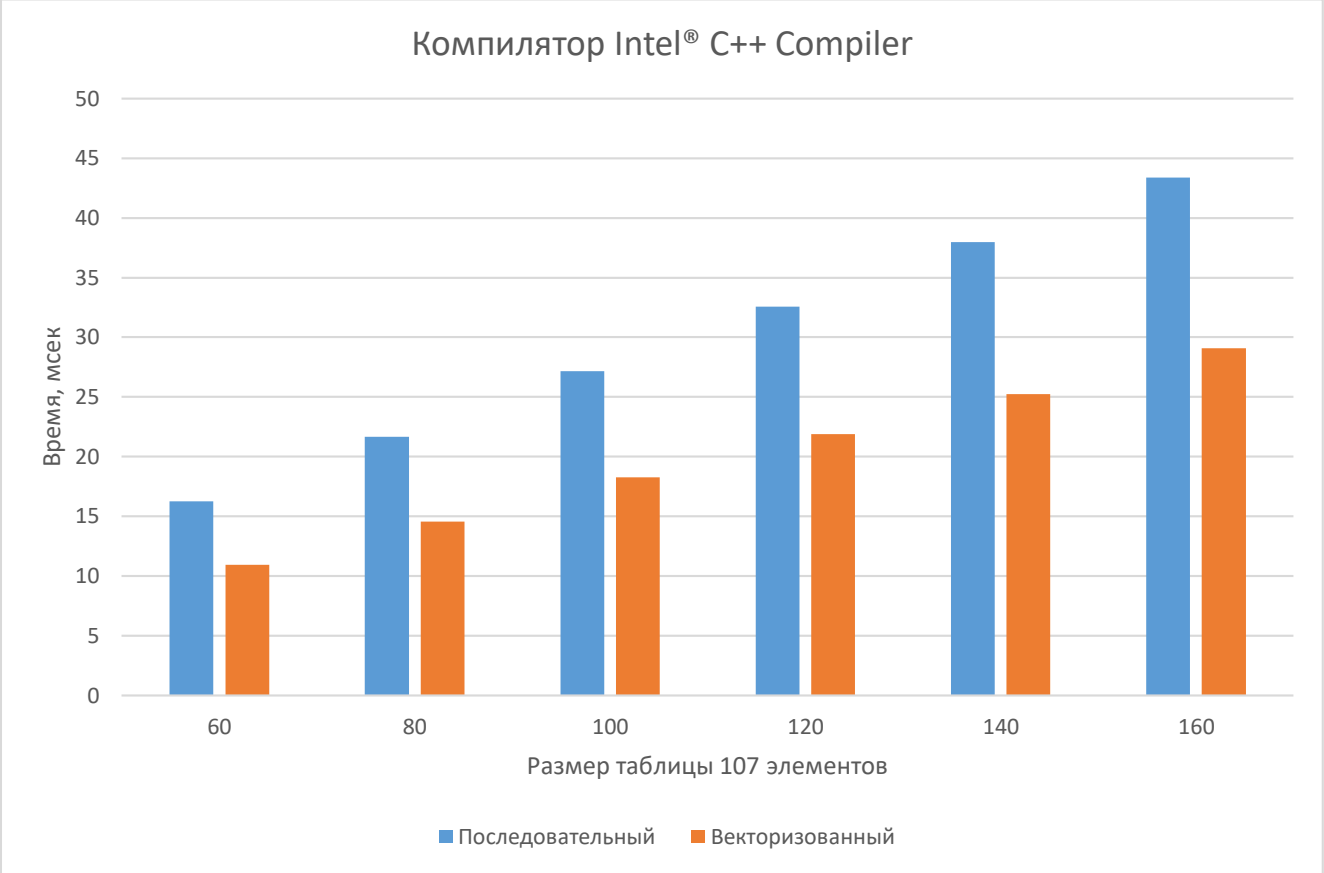

Рисунок Г.5 - тестирование метода линейного векторного поиска при помощи компилятора Intel® C++ Compiler

Результат тестирования метода линейного векторного поиска при помощи компилятора LLVM/CLANG для последовательной и векторизованной версии представлен на рисунке Г.6.

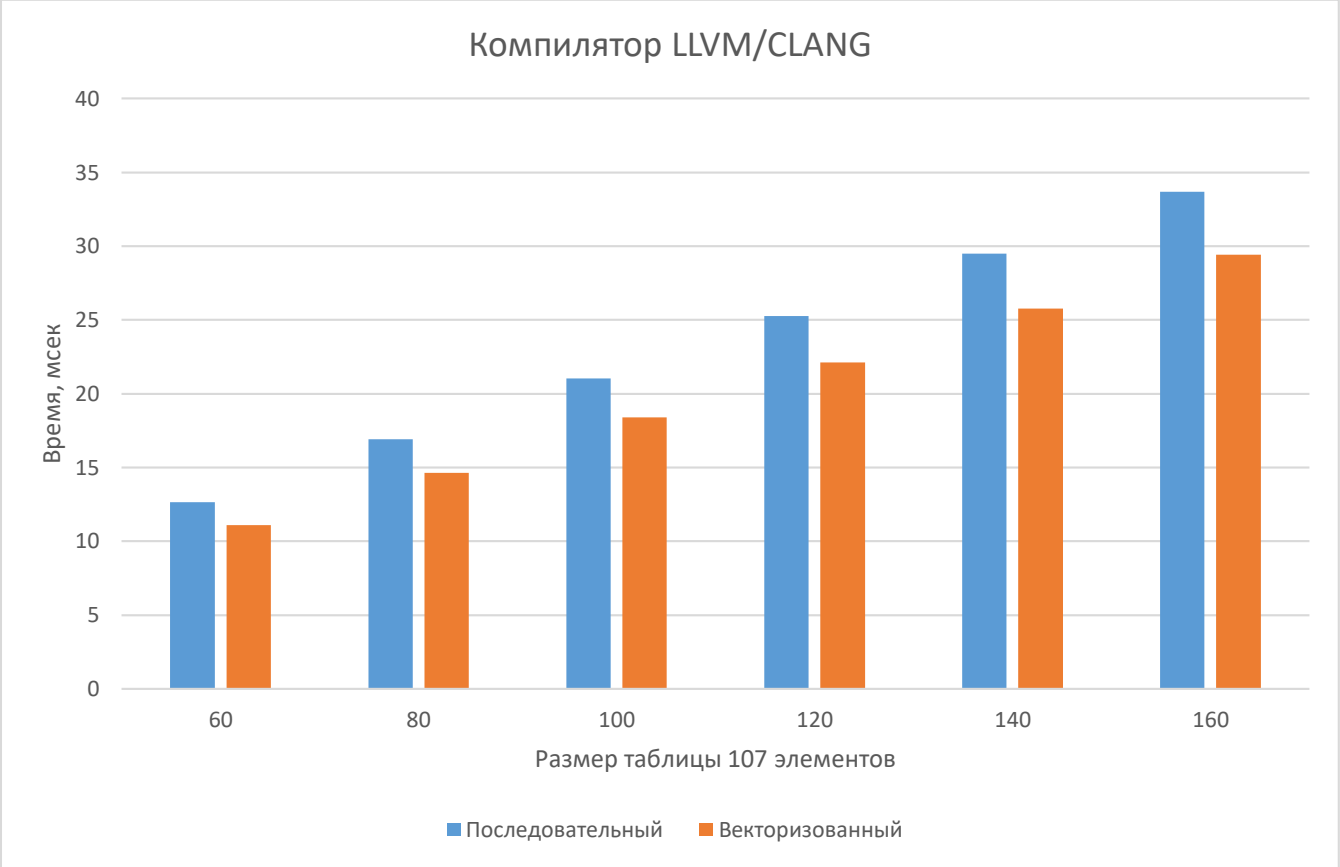

Рисунок Г.6 - тестирование метода линейного векторного поиска при помощи компилятора LLVM/CLANG

Результат тестирования метода поиска делением интервала поиска на M частей при помощи компилятора GNU GCC C/C++ для последовательной и векторизованной версии представлен на рисунке Г.7.

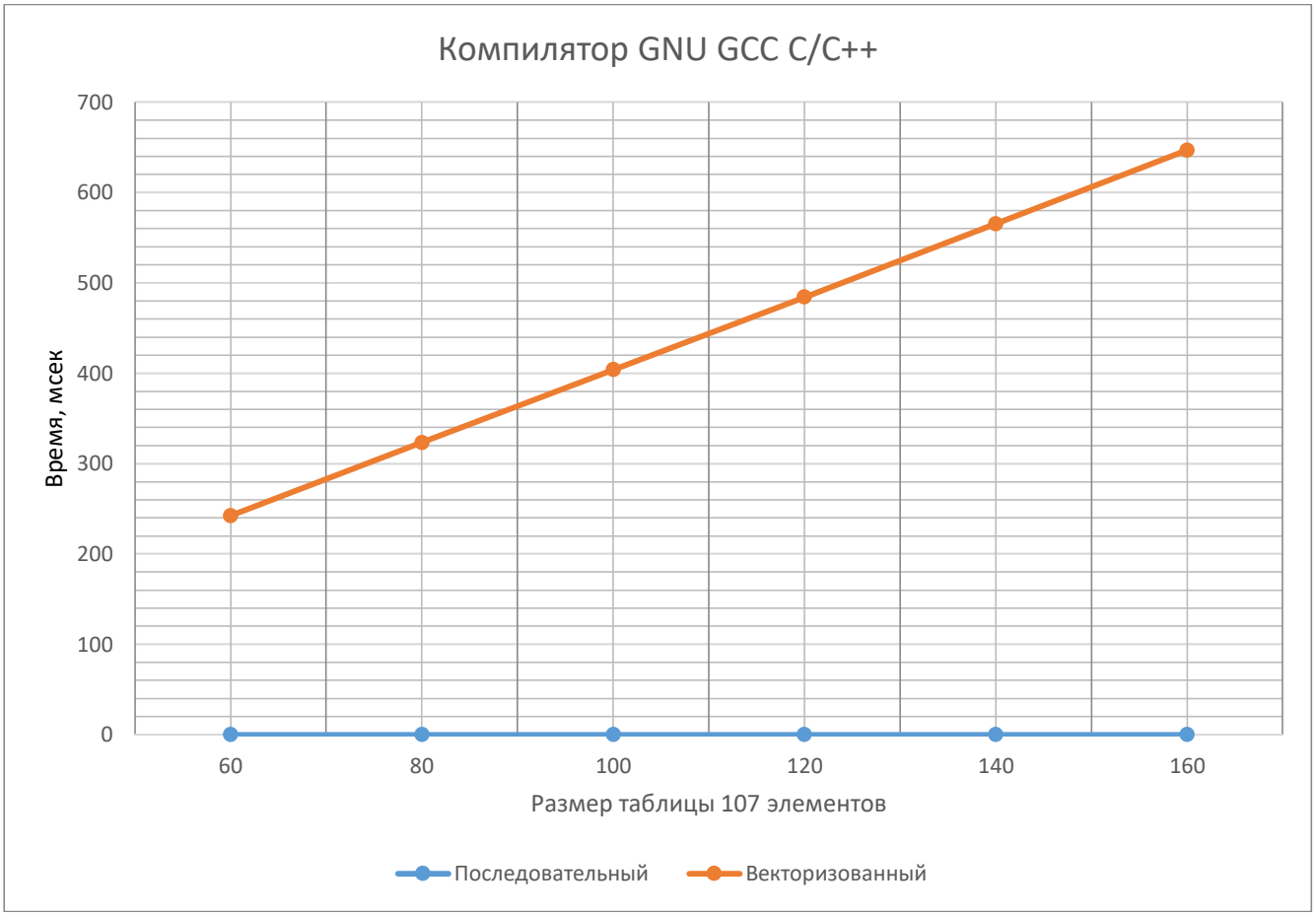

Рисунок Г.7 - тестирование метода поиска делением интервала поиска на M частей при помощи компилятора GNU GCC C/C++

Результат тестирования метода поиска делением интервала поиска на M частей при помощи компилятора Intel® C++ Compiler для последовательной и векторизованной версии представлен на рисунке Г.8.

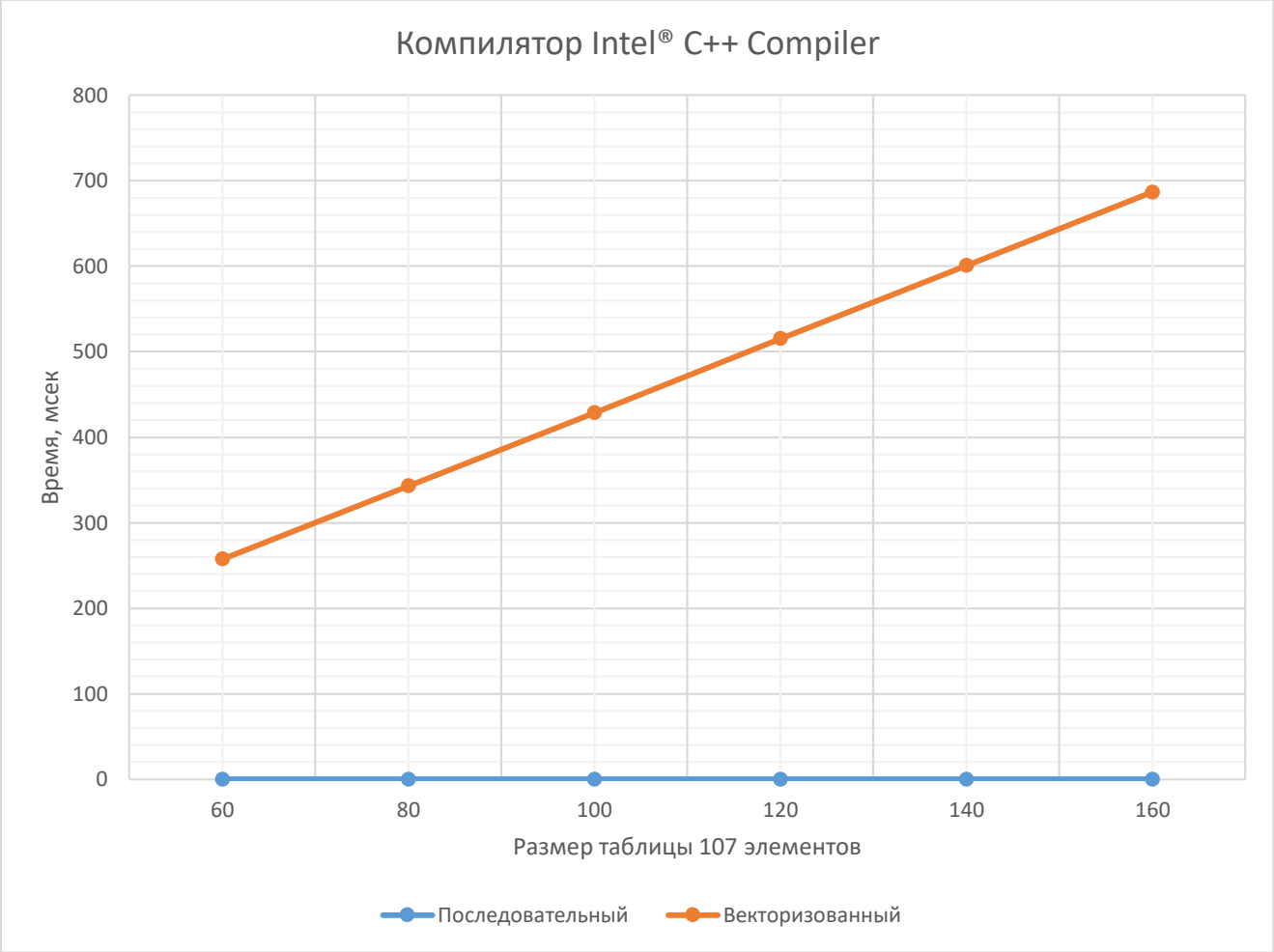

Рисунок Г.8 - тестирование метода поиска делением интервала поиска на M частей при помощи компилятора Intel® C++ Compiler

Результат тестирования метода поиска делением интервала поиска на M частей при помощи компилятора LLVM/CLANG для последовательной и векторизованной версии представлен на рисунке Г.9.

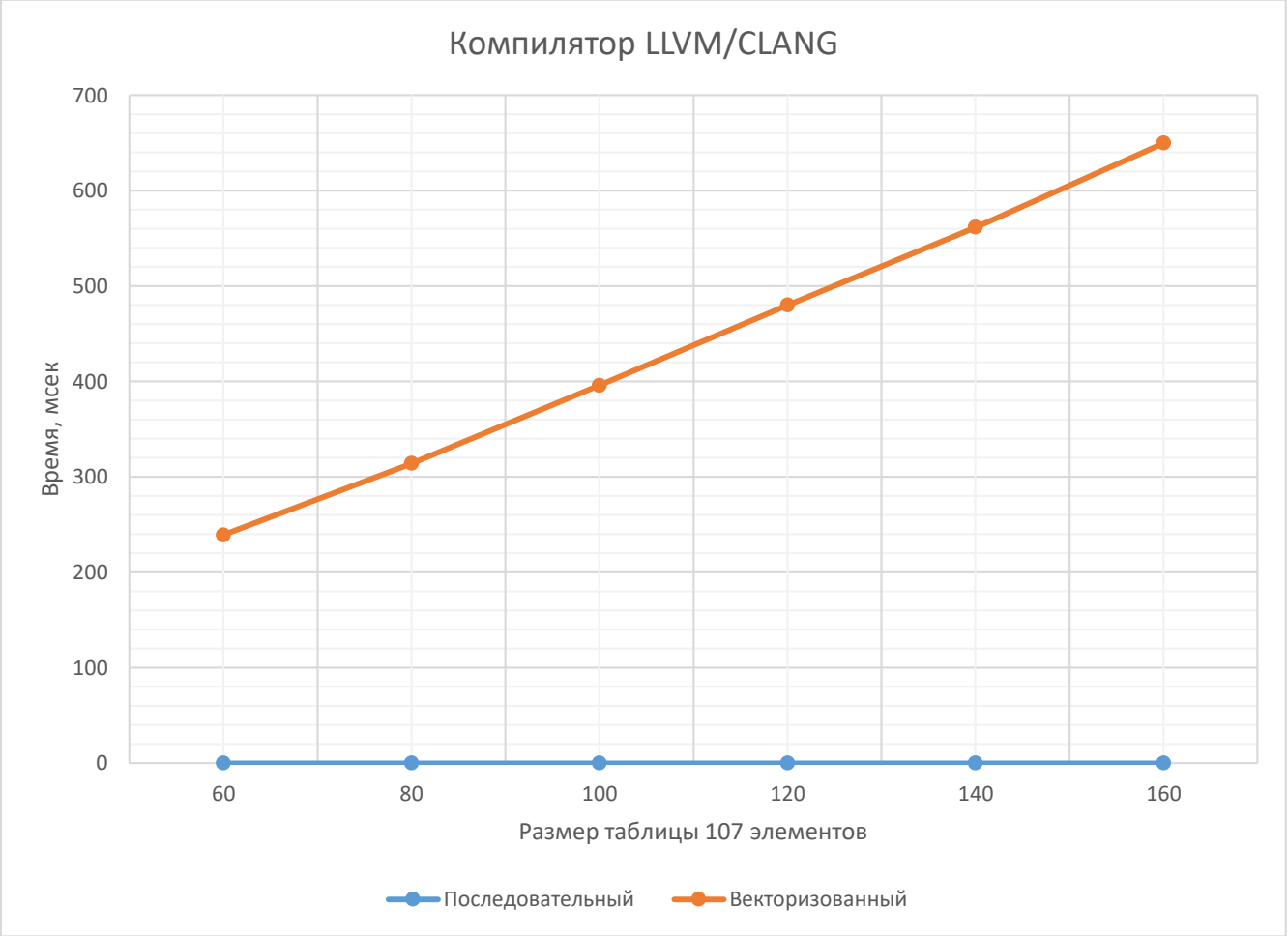

Рисунок Г.9 - тестирование метода поиска делением интервала поиска на M частей при помощи компилятора LLVM/CLANG