Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

(программный проект)

на тему

“Оптимизация ClickHouse под современный набор инструкций CPU”

Выполнил студент группы 175, 3 курса,
Ковальков Дмитрий Андреевич

Руководитель КР:
Кандидат технических наук, доцент, Сухорослов Олег Викторович

Соруководитель КР:
Руководитель группы разработки ClickHouse, Миловидов Алексей Николаевич

Оглавление

Аннотация

В настоящее время производители компьютерных компонентов уже практически достигли предела по скорости процессоров. Количество инструкци, которое современные ядра процессоров могут выполнить за единицу времени, не сильно возросло за последние несколько лет. Чтобы увеличить вычислительную мощность процессоров такие производители, как Intel, AMD и ARM вынуждены использовать другие подходы, нежели простое увеличение скорости ядра.

Первый подход заключается в увеличении количества вычислительных ядер процессора. Благодаря данному методу процессор может выполнять сразу несколько процессов одновременно, тем самым кратно увеличивая количество вычислений за единицу времени.

Второй подход увеличения вычислительной мощности процессоров без повышения тактовой частоты — разработка более совершенных инструкций процессора, позволяющих производить сразу несколько операций за одно инструкцию. Такие инструкции получили название SIMD (Single Instruction Multiple Data). Однако использования таких инструкций затруднительно для разработчиков, так как количество различных наборов таких инструкций уже исчисляется десятками и под каждый набор нужен отдельный код.

Целью данной работы является реализация механизмов в SQL-базе данных ClickHouse, позволяющих использовать SIMD инструкции для реализации высоконагруженных частей проекта.

Ключевые слова

Расширенные наборы инструкций, векторизованные вычисления, динамическая диспетчеризация, SIMD, SSE, AVX, NEON.

Abstract

Nowadays, manufacturers of computer components have almost reached the limit on the speed of processors. The number of instructions that modern processor cores can execute per second has not increased much over the past few years. Companies such as Intel, AMD, and ARM develop other approaches to increase computational capability.

The first approach is to increase the number of processor cores. With this method the processor can perform several processes simultaneously, thereby multiplying the number of calculations per second.

The second approach to increasing the processing power of processors is to develop more advanced processor instructions that can perform several operations per one instruction. These instructions are called SIMD (Single Instruction Multiple Data). However, using such instructions is difficult for developers, since the number of different instructions sets is already in the tens, and separate code is required for each set.

The purpose of this work is to implement mechanisms in the ClickHouse SQL database that help to use new SIMD instructions for implementing highly loaded parts of the project.

Основные определения, термины и сокращения

Набор инструкций — множество инструкций поддерживаемых процессором.

Расширенный набор инструкций — набор инструкций процессора, не являющийся стандартным. Такие наборы инструкций доступны не на всех процессорах.

Single Instruction Multiple Data (SIMD) — общее название инструкций, которые производят обрабатывают несколько операций за раз.

Streaming SIMD Extension (SSE) — семейство расширенных наборов инструкций от компании Intel, позволяющих оперировать регистрами размера до 128 бит. Включает в себя наборы инструкций SSE, SSE2, SSE3, SSE4.1, SSE4.2.

Advanced Vector Extensions (AVX) — семейство расширенных наборов инструкций от компании Intel, добавляющих поддержку регистров длины 256 и 512 бит, с возможность расширения в будущем до 1024. В настоящий момент наиболее распространены AVX, AVX2 и AVX512F

Dynamic Dispatch (динамическая диспетчеризация) — механизм, позволяющий определять реализацию функции во время работы программы.

Введение

SQL база данных ClickHouse была спроектирована компанией Yandex, а исходный код был выложен в свободный доступ в 2016 году. С тех пор проект развивается не только сотрудниками Yandex, но и сотнями независимых разработчиков со всего мира. В настоящее время ClickHouse используется в большом количестве компаний, таких как Cloudflare, Bloomberg, ВКонтакте, Rambler и других. Проект стал популярен благодаря своей скорости работы. ClickHouse изначально проектировался под высоконагруженные вычисления для нужд аналитики, и поэтому в большинстве тестов значительно обгоняет другие СУБД по скорости обработки данных. Чтобы достичь таких результатов в проект постоянно добавляются все новые оптимизации и техники ускорения. Однако оптимизации с помощью самых современных SIMD инструкций на сегодняшний момент не реализованы, что делает данный проект более значимым и важным для выполнения.

В рамках данного проекта планируется реализация различных высоконагруженных частей системы с использованием современных наборов инструкций. В процессе выполнения работы и реализации данных оптимизации придется столкнуться с целом рядом проблем, вот некоторые из них:

1. При наличии нескольких реализаций одной и той же функции необходимо во время работы программы выбирать, какую запустить. Для этих целей планируется разработать механизм динамического выбора (диспетчеризации), который исходя из характеристик доступных реализаций будет выбирать наиболее быструю. На данный механизм накладываются требования максимальной простоты в использовании для других разработчиков, так как его планируется применять повсеместно.

2. В настоящее время во многих компаниях все еще остаются сервера, которые поддерживаю не все современные наборы инструкций. Это означает, что написанный под современные архитектуры код не будет работать на некоторых устаревших серверах. Чтобы обойти данное ограничение в рамках работы планируется использования механизмов автоматического определения архитектуры процессора. Для данных целей на процессорах семейства x86 существует инструкция CPUID, позволяющая определять доступные наборы инструкций во время выполнения программы. Данный механизм необходимо внедрить в механизм диспетчеризации, описанный в первом пункте.

3. Уже существует большое количество различных наборов инструкций. Реализация кода под каждый набор инструкций требует дополнительной работы. Чтобы облегчить разработку таких реализаций планируется

реализовать механизмы, позволяющие генерировать несколько реализаций функций под разные платформы из одного исходного кода.

4. Время работы одних и тех же инструкций может отличаться на разных процессорах. Это означает что для разных процессоров оптимальные реализации будут отличаться. Для решения данной проблемы можно использовать механизм сбора статистик времени выполнения различных реализаций. После сбора достаточного количества статистик выбор реализации в механизме диспетчеризации можно производить более оптимально.

Работа структурирована следующим образом: в обзоре существующих решений рассматриваются похожие работы и решения с объяснением почему они не подходят в ClickHouse. В главе 1 описывается разработанная библиотека для написания платформо-зависимого кода. В главе 2 описывается механизм, который позволяет объединять реализации, написанные с помощью библиотеки из главы 1. В главах 3 и 4 приводятся примеры применения разработанных механизмов оптимизации, а также сравнение производительности до и после их применения. Выводы о работе даны в заключении.

Обзор существующих решений

В настоящий момент в ClickHouse уже используются оптимизации под некоторые расширенные наборы инструкций, такие как SSE4.2. Данные инструкции используются всегда и любая архитектура, на которой запущен ClickHouse, должна поддерживать данный набор инструкций. В настоящий момент существуют более современные инструкции, которые способны совершать за то же время больший объем работы, использование которых и планируется в данной работе.

Среди сторонних проектов с реализованными оптимизациями под самые современные наборы инструкций можно выделить библиотеку SIMDxorshift [1]. Данная библиотека позволяет вычислять последовательности псевдослучайных чисел с использованием алгоритма xorshift и инструкциями из набора AVX2. Все реализации, приведенные в библиотеке, написаны в ручную и не могут быть переиспользованы для других алгоритмов. Библиотека значительно ускоряет вычисление последовательностей псевдослучайных чисел, однако не содержит никаких механизмов автоматического выбора среди представленных реализаций, что затрудняет ее использование в больших проектах. В ClickHouse тоже есть функция генерации псевдослучайных чисел, но на нее накладывается меньше требований по качеству распределения чисел, что дает возможность реализовать ее более эффективно, чем SIMDxorshift.

Другим ярким примером является библиотека Turbo-Base64 [2]. Эта библиотека в том или ином виде использует сразу несколько подходов решения задачи, а также имеет свой механизм динамической диспетчеризации. Однако версия динамической диспетчеризации основывается только на поддерживаемых наборах инструкций на текущей платформе и никак не использую измерения производительности реализаций, что важно, так как инструкции могут работать по-разному на разных платформах. Кроме того, этот механизм спрятан внутри библиотеки, и использовать его в других проектах не представляется возможным.

Компиляция одного и того же кода с разными опциями в библиотеке Turbo-Base64 тоже выполнена очень специфично. Компилируется не только определенные части кода, а все файлы целиком. Такой подход ведет к проблеме что в бинарных файлах могут появиться одинаковые символы, но использующие разные наборы инструкций. В таком случае компоновщик программы на этапе сборки может оставить одну произвольную реализацию и использовать ее везде. В зависимости от выбранной компоновщиком реализации может получиться так, что новые инструкции появятся в функциях, не предназначенных для выполнения на современных процессорах. В таком

случае бинарная программа не будет работать на некоторых процессорах. Чтобы справиться с этой проблемой в библиотеке используются различные макросы, меняющие названия функций в зависимости от опций компиляции, что создает сложно читаемый код, а кроме того такой подход невозможно использовать там, где используются какие-то другие библиотеки.

Переиспользовать реализации алгоритмов из библиотеки Turbo-Base64, которые написаны вручную под каждую платформу, тоже не представляется возможным, так как для каждого алгоритма такая реализация специфичная.

Подход автоматической генерации кода под разные платформы описан в работе “Toward Hardware-Sensitive Database Operations” [3]. Авторы статьи предлагают обобщенный подход к проблеме, разбивают задачу на этапы (подзадачи), и дают советы по проектирования подобных систем. Однако цель данной работы не теоретический подход, а конкретная реализация для проекта ClickHouse, в которой учитываются все тонкости используемых компиляторов и систем сборки.

Глава 1. Компиляция платформо-зависимого кода

Компиляция части кода с платформо-зависимыми опциями является достаточно тяжелой задачей, так как никаких средств для такой компиляции в стандарте языка не предусмотрено, а каждый компилятор предоставляет свои несовместимые с другими инструменты.

В данной работе был разработан интерфейс для разработчиков, позволяющий легко включать платформо-зависимый код в проект, генерировать несколько реализаций под разные платформы и определять поддерживаемые наборы инструкций.

Весь код, предназначенный для какой-то конкретной платформы ***Arch***, должен быть написан внутри макросов *DECLARE_ARCH_SPECIFIC_CODE*. Данный макрос определен по-разному для разных компиляторов и предоставляет единый удобный интерфейс для обозначения платформо-зависимого кода. Данный макрос включает все необходимые опции компиляции и оборачивает весь код в специальное пространство имен TargetSpecific::Arch. Данный подход имеет следующие преимущества:

1. В одном файле можно писать реализации функции сразу под все поддерживаемые платформы.
2. Внутри каждого макроса можно использовать специфичные для архитектуры инструкции. Без данных макросов компиляторы будут выдавать предупреждение об использовании неподдерживаемых инструкций.
3. Так как весь код находится в специальных пространствах имен, то названия функций / классов / переменных не конфликтуют друг с другом. Это также сильно уменьшает вероятность ошибочного использования кода, предназначенного для другой платформы, так как необходимо явно писать название платформы при вызове функций из других пространств имен.
4. Если код компилируется под архитектуру, отличную от x86, то код внутри данных макросов не имеет никакого смысла. В таких случаях он полностью удаляется из исполняемого файла и даже не компилируется.

Для генерации нескольких реализаций под разные платформы был предусмотрен макрос *DECLARE_MULTITARGET_CODE*. Данный макрос автоматически создает несколько копий кода, размещает этот код в соответствующих платформо-зависимых пространствах имен, указывает для

каждой копии специфичные опции компиляции, а также определяет некоторые платформо-зависимые константы и функции.

Для определения какие реализации возможно исполнить на текущей машине была разработана функция IsArchSupported. Данная функция автоматически проверяет поддерживаемые наборы инструкций на процессоре с помощью инструкции CPUID, и возвращает результат. Разработчикам, использующим данный интерфейс для написания платформо-зависимого когда, необходимо вызывать данную функцию перед каждым исполнением кода из платформо-зависимых пространств имен, чтобы избежать ошибки “illegal instruction”.

Глава 2. Объединение реализаций

После создания нескольких реализаций функции нужен механизм выбора, который будет выбирать какую из реализаций функции вызывать. В рамках данной работы такой механизм был реализован в виде класса ImplementationSelector, речь о котором пойдет в этой главе.

Данный класс имеет метод registerImplementation, в который через параметры передается реализация функции, необходимые наборы инструкций для работы, а также произвольное количество аргументов, необходимых для создания объекта функции. Данный метод автоматически проверяет доступны ли запрошенные наборы инструкций на текущем процессоре, не отключена ли данная реализация функции конфигурацией пользователя, и так далее. Если все проверки были пройдены, то данную реализацию можно безопасно использовать, поэтому она добавляется в список доступных.

Класс ImplementationSelector также имеет метод selectAndExecute. При вызове данного метода класс запускает алгоритм выбора реализации из доступных, зарегистрированных с помощью registerImplementation. После выбора реализации она исполняется, а ImplementationSelector собирает статистики, такие как время работы реализации, количество обработанных строк, количество обработанных байт, и так далее. Данные статистики можно будет использовать в алгоритме выбора реализации при последующих запусках функции.

Скорость работы конкретной реализации зависит от множества разных причин. Скорость одних и тех же инструкций на разных процессорах может сильно отличаться, скорость и задержки чтения оперативной памяти тоже отличаются для каждой планки. Кроме того, на скорость могут влиять независимые от машины факторы, такие как конкурирующие процессы или температура окружающей среды. Поэтому определить достоверно какая реализация будет работать быстрее на конкретной машине не представляется

возможности. Чтобы решить какую реализацию запускать используется алгоритм Байесовских бандитов. В данном алгоритме изначально все реализации считаются равными и запускается случайная. Предполагается, что производительность каждого алгоритма является случайной величиной с нормальным распределением. После того как в алгоритме накопится статистически значимое количество данных о времени работы каждой реализации, выбор происходит на основе оценки распределений каждой реализации. Чем выше скорость и ниже дисперсия - тем вероятнее реализация будет исполнена в следующий раз. Таким образом алгоритм обучается на основе статистики и в конце концов сходится к какой-то оптимальной реализации. Аналогичный механизм уже используется в ClickHouse для сжатия блоков данных алгоритмом LZ4 [4], но для использования с другими функциями необходимо его обобщить.

В настоящее время в ClickHouse используется два интерфейса для представления функции, это IFunction и IExecutableFunctionImpl. Первый интерфейс считается устаревшим, тем не менее он до сих пор используется в подавляющем большинстве функций. ImplementationSelector поддерживает оба данных интерфейса с помощью перегрузки по шаблонному параметру, где необходимо передать желаемый интерфейс функции.

Глава 3. Авто-векторизация

Авто-векторизация кода является очень важной оптимизацией компиляторов, так как требует наименьшего времени программиста на разработку и поддержку такого кода. Однако компиляторы не всегда одинаково хорошо выполняют эту задачу. Наиболее простыми элементами для векторизации компилятором являются циклы с простыми арифметическими операциями, применяемыми ко всем элементам одинаково.

Рассмотрим такие оптимизации на примере функций intHash32 и intHash64 в ClickHouse. Данные функции считают некриптографические хеш-значения для каждого элемента независимо, используя последовательность простых математических операций, таких как умножение, сложение, битовые сдвиги и исключающее или. С помощью описанного выше макроса *DECLARE_MULTITARGET_CODE* были сгенерированы реализации под разные платформы, объединенные в одну функцию с помощью класса ImplementationSelector.

Так как ClickHouse это СУБД, то нас интересует не скорость выполнения какого-то алгоритма, а скорость выполнения запроса целиком. Поэтому для измерения производительности использовался SQL-запрос, который считает хеш-значения для всех чисел от 1 до 100’000’000. Чтобы получить

воспроизводимые результаты и исключить некоторые факторы случайности все запросы выполнялись используя один поток.

Ниже приведены результаты измерения скорости работы запроса на разных машинах для обычного и векторизованного компилятором под набор инструкций AVX2 кода. Каждый запрос повторялся множество раз в течение нескольких минут, в таблице представлено усредненное время работы по всем запускам. В левом столбце написана запускаемая функция, компилятор, которым был собран бинарный файл, а так же машина, на которой запускался код (remote - тестовый сервер команды ClickHouse, local - мой рабочий компьютер).

	Время, default	Время, AVX2	Относительное ускорение
intHash32, gcc, remote	0.1837 с	0.1357 с	26.2%
intHash32, gcc, local	0.3885 с	0.2739 с	29.5%
intHash32, clang, local	0.4610 с	0.3028 с	34.3%
intHash64, gcc, remote	0.1557 с	0.1285 с	17.5%
intHash64, gcc, local	0.3802 с	0.3095 с	18.6%
intHash64, clang, local	0.3225 с	0.2903 с	10.0%

Таблица 3.1 Сравнение времени работы после оптимизаций.

Как можно видеть из таблицы, время работы и относительное ускорение сильно зависят от машины и от компилятора. Причем нету компилятора который лучше во всех случаях, на разных функциях они производят оптимизации разной степени успешности. Так как мой рабочий компьютер значительно слабее сервера, то сравнивать между разными машинами можно только относительное ускорение векторизированного кода, но не само время выполнения запросов

Тем не менее, среднее ускорение для функции intHash32 на всех запусках составило порядка 30%, а для функции intHash64 порядка 15%. Это очень хороший результат, учитывая что данная оптимизация была проделана с добавлением всего пары десятков строк кода, используя разработанное в работе API.

Глава 4. Ручная векторизация

Автоматическая векторизация проста в использовании, но ввиду ограничений по времени компиляции и гарантий языка программирования компиляторы не всегда могут генерировать оптимальный код. В таких случаях необходимо писать векторные вычисления вручную.

Рассмотрим функцию rand, которая генерирует последовательность псевдослучайных чисел с помощью линейного конгруэнтного метода [5]. Метод, описанный в предыдущей главе, ни только не дал никакого прироста производительности, но и замедлил исполнение некоторых запросов. Существует несколько причин по которым в данном случае компилятор не смог сгенерировать оптимальный код:

1. В реализации алгоритма используется 64-битное умножение целых неотрицательных чисел. По историческим причинам в векторных наборах инструкций SSE (всех версий), AVX и AVX2 отсутствуют инструкции умножения векторов 64-битных чисел.

2. Компилятор может делать только эквивалентные преобразования, не меняющие результата выполнения функции. Однако, поскольку мы генерируем последовательность случайных байтов, то мы можем позволить преобразования, не меняющих итогового распределения чисел. С помощью таких преобразований можно проделать гораздо больше оптимизаций.

Первая проблема была решена с помощью эмулирования умножения 64-битных чисел через умножения 32-битных. Одно такое умножение требует 3 инструкции умножения 32-битных чисел и несколько инструкций битового сдвига и сложения. Однако за счет того, что мы можем обрабатывать таким способом сразу по 4 числа за инструкцию, это преобразование даже дало небольшой прирост производительности.

В используемой вариации линейного конгруэнтного генератора для генерации случайных чисел используются только биты с 15 по 48 позицию. Поскольку компилятор должен сохранить результат выполнения исходного кода, то он генерировал множество инструкций для извлечения данных битов из середины состояния каждого генератора. Чтобы избавиться от лишних инструкций по извлечению случайных данных был добавлен второй вектор с другими состояниями генераторов, которые считались одновременно. Получить из двух векторов состояний один со случайными байтами можно с помощью всего одной инструкции shuffle и одной инструкции blend, что значительно уменьшило количество используемых инструкций. Данное преобразование не

сохраняет порядок байт внутри регистра, но поскольку мы генерируем случайные числа, то порядок байтов не важен.

Описанный выше алгоритм был объединен со старым скалярным с помощью класса ImplementationSelector.

Ниже приведены результаты измерения производительности запроса, использующего функцию rand. Все эксперименты были проведены по правилам, описанным в предыдущей главе.

	Время, default	Время, AVX2	Относительное ускорение
gcc, remote	0.0504 с	0.0390 с	22.7%
gcc, local	0.1235 с	0.1110 с	10.1%
clang, local	0.1205 с	0.1073 с	10.9%

Таблица 4.1

Как можно видеть из таблицы, время работы алгоритма практически не зависит от используемого компилятора, так как оптимизации были выполнены вручную. При этом относительное ускорение времени работы запроса достаточно сильно зависит от машины, на которой запускался код. Это показывает необходимость адаптивного метода выбора реализации в зависимости от времени выполнения на предыдущих запусках.

Данные результаты показывают, что даже когда код плохо приспособлен для векторизации, его можно переписать на немного видоизмененный алгоритм и получить прирост в производительности порядка 10-15% с использованием векторизированных инструкций.

Заключение

В результате выполнения работы были достигнуты поставленные цели и разработаны механизмы и библиотеки, значительно облегчающие разработку платформо-зависимого кода. С помощью данных механизмов были прооптимизированны некоторые функции ClickHouse, получив увеличение производительности некоторых запросов на 15-30% на платформах, поддерживающих современные векторные наборы инструкций.

Данный подход к решению поставленной задачи показал себя хорошо и в дальнейшем планируется работать в данном направлении. Прежде всего, можно расширить количество поддерживаемых платформ, генерируя больше платформо-зависимого кода. Во-вторых, можно улучшать механизм динамической диспетчеризации, чтобы определение наиболее производительной реализации занимало меньше времени. Этого можно пытаться достичь как с помощью разработки нового алгоритма сбора и обработки статистик, который асимптотически быстрее сходится к оптимуму, так и с помощью сохранения и переиспользования статистик между различными запросами. В-третьих, можно увеличить количество функций, в которых используется платформо-зависимый код.

Список источников

[1] Реализация библиотеки Tubo-Base 64.
https://github.com/powturbo/Turbo-Base64
[2] Реализация библиотеки SIMDxorshift.
https://github.com/lemire/SIMDxorshift
[3] David Broneske, Sebastian Breß, Max Heimel: Toward Hardware-Sensitive Database Operations // 10.5441/002/edbt.2014.22
[4] Использование вероятностных алгоритмов для выбора реализации сжатия данных
https://github.com/ClickHouse/ClickHouse/blob/master/src/Compression/
[5] Joe Bolte, “Linear Congruential Generators”, Wolfram Demonstrations Project
https://demonstrations.wolfram.com/LinearCongruentialGenerators/