Анализ эффективности использования графических процессоров для ускорения обработки SQL-запросов

А.Г.Савельев[a], И.С. Вершинин[a], Р.Ф. Гибадуллин[a]

[a] *КНИТУ-КАИ, 420111, ул. Карла Маркса, 10, Казань, Россия*

Аннотация

В работе представляются результаты исследования эффективности использования вычислительной мощности графического процессора для выполнения задач СУБД. Используя графический процессор для решения подобного рода задач, можно освободить ресурсы на центральном процессоре для других задач и тем самым обеспечить более быстрое выполнение.

Ключевые слова: базы данных; графический процессор; СУБД; параллельные вычисления; высокопроизводительные вычисления

1. Введение

Благодаря огромной вычислительной мощности современных графических процессоров, открываются новые возможности для решения задач СУБД в связке с центральным процессором. Тем не менее, многие задачи в СУБД по-прежнему связаны с памятью, и не могут извлечь выгоду из более мощного сопроцессора, особенно с узким местом PCIe [1]. Кроме того не каждая задача подходит для ее выполнения на графическом процессоре. Обработка строк и задачи с интенсивной обработкой данных, как правило, не выигрывают в скорости из-за узкого места передачи данных. В остальных случаях нагрузка должна быть достаточной, чтобы в полной мере использовать параллельную архитектуру и оправдать затраты на запуск ядра.

На основании имеющихся исследований [2] и проведенных экспериментов были сделаны выводы о том, как можно ускорить СУБД с помощью графического процессора. В результаты в СУБД были найдены четыре различные категории задач, изображенные на рис.1, в которых выигрыш в производительности будет иметь значительное влияние на всю систему, и сократит время ожидания ответов на запросы.

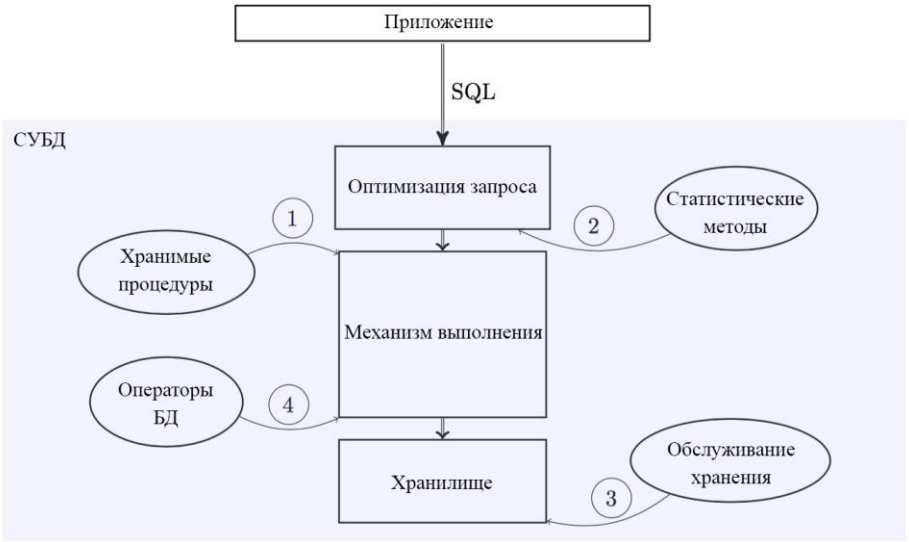

Рис. 1. Возможные задачи совместной обработки в СУБД.

Прикладная логика (1). Большинство СУБД предоставляют возможность интегрировать прикладную логику в виде хранимых процедур. Таким образом, СУБД может выполнять логику, написанную на другом языке, отличном от SQL. C++ или Java могут быть проще, и предоставляют больше возможностей или более высокую производительность в некоторых случаях. Таким образом, вопрос заключается в том, как можно использовать сопроцессор для извлечения выгоды из задач этой первой категории.

Оптимизация запросов (2). Пользователи и приложения, как правило, получают доступ или вносят изменения в базу данных с помощью SQL, который является декларативным языком, то есть, он не указывает на то, как запрос должен быть выполнен. Следовательно, СУБД сначала должна построить план выполнения запроса (ПВЗ), который сообщает исполнительному механизму, какой оператор и в каком порядке должен выполнятся. Производительность очень сильно зависит от этого плана. Так что вторая категория состоит из всех задач, необходимых для построения и оптимизации ПВЗ.

Обслуживание хранения (3). Еще одним важным фактором, влияющим на производительность выполнения запроса, является доступность хранимых данных. В целом, оптимизация хранения преследует три цели: уменьшенный объем потребляемой памяти, быстрое чтение, и быстрые модификации с данными [3]. Оптимизация для всех трех не представляется возможной. Индексы, например, ускорят чтение и поиск данных, но необходима память для их хранения и при изменении данных они должны быть скорректированы. Сжатие уменьшает объем потребляемой памяти, но изменение сжатых данных требует дополнительных усилий. Обычно СУБД разделяет данные так, что часто изменяемые данные хранятся в форме отличной от тех данных, которые нужны только для чтения. Данные, доступ к которым нужен редко, находятся в архиве. Система автоматически перемещает данные на нужный раздел на основе использования, нагрузки и требований системы. Чем быстрее это обслуживание может быть сделано, тем чаще оно может использоваться. Кроме того, если части этого обслуживания могут быть переложена на сопроцессор, ресурсы на CPU освободятся для других операций.

Выполнение запроса (4). Последней и, возможно, наиболее важной категорией является исполнение самого ПВЗ. Типичные операторы баз данных, такие как соединение, выборка и агрегация имеют различные характеристики, и не каждый из них подходит для исполнения на GPU. Кроме того, существуют различные способы, передачи данных от одного оператора к другому.

2. Интеграция GPU для выполнения статических задач в СУБД

ЦП представляет собой программируемый универсальный блок обработки в наших компьютерах. Он может обрабатывать изображения, фильтровать большие объемы данных по определенным критериям, расчет научных формул на различных представлениях чисел (с фиксированной точкой и с плавающей), анализировать тексты, и все остальное, что делается с помощью современных компьютеров. Поскольку назначение центрального процессора носит настолько общий характер, он не может быть эффективно оптимизирован для одной задачи. Сопроцессора в противоположность этому оптимизируется для подмножества задач, например, графическая обработка в случае с GPU. Но он не очень хорошо подходит для операций, которые не соответствуют шаблону SIMD, таких как обработка строк. Следовательно, мы не ставим своей целью построить СУБД, которая работает полностью на GPU, но можно определить задачи в СУБД, подходящей для обработки на GPU.

Есть целый ряд задач в СУБД, которые можно считать независимыми от остальной части системы. Таким образом, можно разгружать CPU, выполняя эти задачи на GPU без концептуальных изменений в СУБД. Общая модель распределения задач в параллельной архитектуре сопроцессора изображена на рис.2. Этапы один и пять необходимы, потому что GPU не может получить доступ к памяти процессора, и они представляют собой узкое место передачи. Этапы два и четыре являются частью необходимых накладных расходов для параллелизации, соотв. распределение работы между разными ядрами. Этапы этого шаблона могут перекрываться с другими этапами, например, разделение может быть частью передачи.

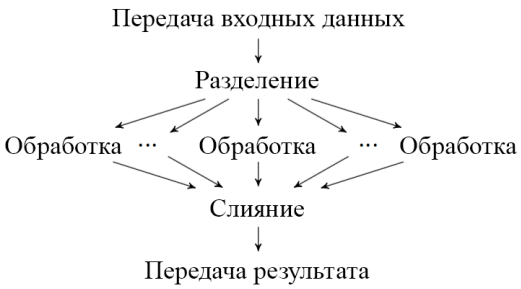

Рис. 2. Общая модель выполнения задачи на GPU.

Нет никакой гарантии, что на GPU задача выполнится быстрее. Основное преимущество заключается в том, что интенсивные вычислительные операции могут извлечь выгоду из разгрузки, т.е. узкое место передачи не будет являться проблемой, так как большое количество обработки делается на небольшом количестве данных. Тем не менее, существуют такие алгоритмы перемалывания чисел, которые не подходят к модели программирования GPU [4]. Даже несмотря на то, что передача данных уже не является проблемой, центральный процессор может решить данную задачу быстрее и эффективнее. С другой стороны, существуют задачи с интенсивным обменом данными, которые могут извлечь выгоду из характеристик графического процессора (в основном с высокой пропускной способностью на устройстве). Как показывают эксперименты [4], не так легко предсказать производительность алгоритма. Но если даны определенные критерии, есть шанс, что мы окажемся в выигрыше от использования GPU.

Критерии, при наличии которых может быть прирост производительности при использовании GPU:

1. Существует алгоритм для выполнения этой задачи, которая работает параллельно.
2. Существует алгоритм для выполнения этой задачи, которая работает с массовым параллелизмом в модели SIMD.
3. Обработка данных на процессоре занимает больше времени, чем передача его на карточку.
4. Нет никакой потребности в дополнительных передачах (подкачивающий к оперативной памяти) на третьем этапе.
5. Передача в GPU и фактические вычисления могут перекрываться.

3. Проектирование динамических задач для сопроцессора

Задачи, относящиеся к категории (1), (2) и (3) имеют очень важную общую характеристику. Нагрузку можно назвать статической, если во входных данных меняется только в размере данных и параметрах, которые влияют на количество необходимых вычислений на байт. Сами задачи являются статическими, когда они построены путем комбинирования примитивов, например, путем последовательного выполнения математических операций над матрицами. Эти операции обычно выполняются в определенном порядке и не зависит от размера всей рабочей нагрузки, и время выполнения для каждого примитива в сравнении с остальными можно предсказать [5]. При попытке оптимизировать такие статические задачи, разработчик ищет в операциях «горячие участки», где тратится больше всего времени, и оптимизирует их. При использовании сопроцессора, во многих случаях выгодно просто выполнить эти операции на сопроцессоре и оставить незначительные операции в ЦП.

Другой метод оптимизации задачи заключается в объединении примитивов таким образом, что нет никаких границ соотв. точкам синхронизации между ними. Крайне важно выполнить как можно больше работы в один вызов ядра. Может быть даже следует объединить маленькие операции в одну большую, которая извлечет выгоду из разгрузки. Конечно, это уничтожает пригодность обслуживания и модульный принцип самой программы и возможно только в случае, когда мы знаем порядок выполнения примитивов.

Динамические задачи состоят из модульных примитивов, которые выполняются согласно плану, построенному во время исполнения. Без дополнительного анализа мы не сможем предсказать, какие из этих примитивов являются горячими участками. GPU значительно медленнее при выполнении умножений небольших матриц. Если мы заранее не знаем размер входных данных для операций, как эта, и они выполняются несколько раз, решение всей задачи может быть также гораздо медленнее. Так как примитивом может быть только ядро, на первый взгляд, нет возможности объединить их. Таким образом, планировщик задач должен уметь предсказать промежуточные размеры результата и распределить работы соответствующим образом. Также он должен уметь делать низкоуровневую оптимизацию. В то время как процессор сам оптимизирует выполнение команд, блоки обработки на GPU не делают почти никакой оптимизации. Это характерно для сопроцессоров: архитектура точно предназначена для решения задач определенной области, и каждый транзистор должен обеспечить как можно больше вычислительной мощности. Таким образом, операции должны быть разработаны и оптимизированы на уровне команд. В случае с графическими процессорами имеется дополнительная проблема иерархии памяти и планирования доступа к ней для корректности вычислений. Опять же, с ядрами, как примитивами, мы можем использовать эту иерархию памяти только лишь внутри. Даже если результаты были достаточно маленькими, чтобы сохранить их в разделяемой памяти, ими нельзя обменяться без передачи в оперативную память устройства.

Все это доказывает, что подход, использования модульных ядер в качестве строительных блоков не подходит для динамических задач. Поэтому вместо этого, предлагается генерировать и компилировать ядра во время выполнения. Таким образом, мы можем использовать иерархию памяти через примитивы и объединять их, чтобы сократить количество точек синхронизации во время выполнения. Сам компилятор может сделать оптимизацию низкого уровня.

4. Заключение

В то время как есть задачи, которые могут извлечь выгоду из GPU в качестве блока обработки, большинство из них работает хорошо только на определенных входных данных или определенных параметрах. С одной стороны, если данные слишком малы, то не все ядра графического процессора будут задействованы и накладные расходы на запуск ядра и распараллеливание выполнения преобладают над временем выполнения. С другой стороны, если входные данные слишком велики, чтобы поместиться в память GPU, появляется необходимость передачи входных данных и промежуточных результатов несколько раз из оперативной памяти в память устройства и обратно. В этих случаях для расчетов должен использоваться ЦП. Точки безубыточности изменяются для каждой задачи и каждой комбинации аппаратных средств. Особенно, когда время выполнения зависит от многократных параметров, таких как тип данных ввода или число итераций, определение этой точки вручную не представляется возможным. Кроме того, каждый раз может возникать необходимость повторной калибровки при изменении аппаратных средств. Таким образом, необходима платформа, которая бы автоматически изучала точки безубыточности.

Современные СУБД все чаще хранят данные в памяти во время обработки, некоторые даже хранят там всю базу данных для быстрого доступа. В такой системе большинство алгоритмов ограничено мощностью процессора, потому что узкое место ввода/вывода в/из диска исчезло. Одновременно с этим графические процессоры (GPU) превзошли центральный процессор (CPU) в плане вычислительной мощности. Проведенные исследования показывают, что они могут использоваться не только для обработки графики, но также могут решать задачи общего назначения. Однако не каждый алгоритм может быть портирован на архитектуру GPU с выигрышем производительности. Во-первых, алгоритмы должны быть адаптированы, чтобы допускать параллельную обработку в стиле один поток команд, много потоков данных (SIMD). Во-вторых, осталось «узкое место» передачи, потому что высокоэффективные графические процессоры соединены через шину PCI Express (PCIe).

Литература

[1] Джейсон Сандерс, Эдвард Кэндрот. Технология CUDA в примерах. Введение в программирование графических процессоров: Пер. с англ. Слинкина А.А. – Москва: Изд-во ДМК Пресс, 2011. – 232 с.

[2] Peter Bakkum and Kevin Skadron, Accelerating SQL Database Operations on a GPU with CUDA //Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. 2010 P.94-103
[3] Гибадуллин, Р.Ф., Новиков, А.А., Хевронин, Н.В., Перухин, М.Ю. Разработка параллельного модуля генерации защищенной картографической базы данных // Вестник Казан. технол. ун-та. – 2016. – № 10. – С.102-105.
[4] Гибадуллин, Р.Ф., Савельев, А.Г., Перухин, М.Ю. Ускорение обработки SQL-запросов к базам данных на GPU посредством аппаратно-программной платформы NVIDIA CUDA // Вестник Казан. технол. ун-та. – 2016. – № 20. – С.110-116.
[5] Вершинин, И.С., Гибадуллин, Р.Ф., Пыстогов, С.В., Перухин, М.Ю. Импорт/экспорт ассоциативно защищенных картографических данных с их обработкой в системе Security Map Cluster // Вестник Казан. технол. ун-та, 2015. – № 10. – С. 174-180.