Математические структуры и моделирование
2011, вып. 22, с. 86–94

УДК 004.655.3

ОБРАБОТКА РЕЛЯЦИОННЫХ БАЗ ДАННЫХ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

Д. А. Лыфарь

Review of existing technologies which can be used to process relational databases on graphical processor units (GPU). Short description of GPU primitives like prefixsum, Map-Reduce, parallel sort and solutions which are built on them are provided.

1. Введение

Поиск и обнаружение информации в базах данных является одной из важнейших задач в информационных технологиях сегодня. Академические центры таких компаний, как Oracle, Microsoft, SAP, заинтересованы поиском масштабируемых решений на базе графических процессоров (GPU). Вычисления на графических процессорах получили серьёзный толчок в развитии лишь недавно и по возможностям, которые предоставляют текущие API для GPU (OpenCL/CUDA) приближаются к центральным процессорам, что способствует появлению направления GPGPU (Generic Programming on GPU). Эта область относительно молода, однако ряд исследований показывает, что с помощью GPU возможно эффективно решать задачи общего назначения, в частности, такие как: data mining и обработка запросов к большим объёмам данных.

Основным препятствием в обработке баз данных на графическом процессоре является ограниченный объем памяти GPU. Традиционно производительность баз данных упирается в подсистему ввода–вывода, т. е. центральный процессор не является узким местом. Хотя диски существенно дешевле оперативной памяти и имеют высокую ёмкость, они гораздо медленнее оперативной памяти. С распространённой скоростью вращения шпинделя, равной 7200 об/мин, оборот требует примерно 8,33 мс, что почти на 5 порядков превышает время обращения к оперативной памяти (которое составляет 100 нс). Однако сейчас с ростом объёма оперативной памяти серверов, которые обычно обслуживают СУБД, скорость обработки становится немаловажным фактором, так как зачастую вся таблица(ы) помещается в оперативной памяти. На рынке существует такое понятие, как *commodity hardware*. В русской интерпретации это означает недорогое аппаратное обеспечение, в противовес дорогим серверным системам,

Омский государственный университет им. Ф. М. Достоевского.
E-mail: dlyfar@gmail.com

из которых можно построить вычислительный комплекс под нужды бизнеса. В данный момент commodity hardware можно назвать компьютер с 4–8 ядрами, 8–16 Гб оперативной памяти и дисками в 15K об/мин. Так как подобное оборудование имеет большое превосходство по объёму памяти над графическим процессором, то область, где мы можем получить прирост производительности, — это непосредственно сама обработка данных (полагаясь на возможность параллельного исполнения задач на векторной архитектуре GPU). В этой статье мы сделаем обзор существующих решений, предназначенных для обработки больших объёмов данных на GPU, рассмотрим детали реализации и компоненты системы управления баз данных на GPU.

Прежде всего стоит рассмотреть архитектуру существующих баз данных. Практически в основе любой базы данных в качестве структуры данных для хранения лежит B-дерево. Оно представляет собой сбалансированное дерево поиска, созданное специально для быстрой работы с дисковой памятью. B-деревья отличаются от красно-чёрных деревьев тем, что узлы B-дерева могут иметь много дочерних узлов — до тысяч, так, что степень ветвления B-дерева может быть очень большой (хотя она обычно определяется характеристиками используемых дисков. B-деревья схожи с красно-чёрными деревьями в том, что все B-деревья с n узлами имеют высоту $O(\lg n)$, хотя само значение высоты в этом дереве существенно меньше, чем у красно-чёрного, за счёт более сильного ветвления. Таким образом, B-деревья также могут использоваться для реализации многих операций над динамическими множествами за время $O(\lg n)$.

Мы указывали, что узким местом системы обработки баз данных является производительность дисковой подсистемы, по этой причине в алгоритмах обработки баз данных делают акцент на:

- количество обращений к дисковой подсистеме,
- вычислительное время (время процессора).

Как уже было указано выше, ранее производительность обработки данных упиралась в дисковую подсистему, в настоящий момент можно отметить появление новых исследований на эту тему, в частности обработки на GPU (исследование компании Oracle [10]). Существует множество аргументов в пользу поиска масштабируемых решений на графическом процессоре. С течением времени это становится наиболее экономически эффективным. На момент написания статьи одна из самых быстрых видеокарт от NVIDIA (GTX 285) соразмерна по цене с четырёхядерным процессором (имея мощность около 200 ватт). Скорость математических операций для подобного видеоадаптера порядка 1 триллион в секунду (что в 20 раз быстрее самого быстрого CPU). Так как видеоадаптеры можно объединить в одном сервере, то четыре GPU обеспечивают 80-кратное превосходство над скоростью одного CPU при увеличении мощности лишь в 6 раз, что ведёт к существенной экономии потребляемой мощности и затрат на обслуживание серверов (их становится существенно меньше). Появление кластерных решений для GPU способствует тому, что в ряде задач по обработке данных GPU будут иметь приоритет перед CPU.

2. Обзор существующих решений

2.1. Исполнение табличных SELECT запросов средствами GPU

В области обработки баз данных на GPU уже существует ряд исследований ([6,7] и др.). Рассмотрим одно из последних и наиболее близких к исследуемому вопросу в работе [1]. Под обработкой баз данных в этой статье будем понимать выполнение операций CRUD (это сокращённое наименование четырёх базовых функций управления данными: создание (C), чтение (R), редактирование (U) и удаление (D)) над множеством данных с помощью языка запросов. Цель указанных исследований — показать, насколько GPU превосходят обычные процессоры на определённом множестве операций.

Среди исследований, которые также можно отнести к обработке баз данных, можно выделить: поиск подстроки, выявление зависимостей, быструю сортировку ([5,8,9]), являющиеся базовыми блоками при реализации CRUD.

Так же как и в этом исследовании, был выбран язык SQL для описания запросов. SQL — это признанный декларативный язык в индустрии для манипулирования данными. Он является промежуточным слоем между логикой программы и набором данных. Реализация SQL на графическом процессоре позволит ускорить программу без изменений в исходном коде. Несмотря на очевидное преимущество этого подхода, пока не существует законченных реализаций SQL на GPU.

Архитектура СУБД включает в себя множество компонентов, поэтому, чтобы исключить ненужную работу по реализации парсера, виртуальной машины, тестов, была выбрана существующая СУБД SQLite. Она широко используется в ряде проектов и поддерживается большим числом известных компаний. Виртуальная машина SQLite отвечает за преобразование декларативного SQL в набор команд, реализующих логику запроса (всего около 128 команд). Каждая из команд виртуальной машины может принимать до пяти аргументов (команды машины достаточно высокоуровневые: открытие таблицы, загрузка значения, математические операции, операции перехода и т. п.). Самый простой SELECT состоит из инициализации таблицы, цикла по всем строкам и очистки ресурсов. Решение использует практически все виды памяти, предоставляемые программной моделью CUDA. Регистровая память используется для хранения смещений в блоке данных и результатов, разделяемая (shared) память — для хранения результатов каждого потока. В константной памяти хранится программа виртуальной машины, которую исполняет каждый из потоков, а также служебная информация, например, размерность обрабатываемых типов. Глобальная память хранит обрабатываемые данные. В этом исследовании реализована поддержка операторов выборки вида:

```
SELECT id, uniformi, normali5 FROM test WHERE
       uniformi > 60 AND normali5 < 0
SELECT id, uniformf, normalf5, normalf20 FROM test
       WHERE NOT uniformf OR NOT normalf5
       OR NOT normalf20
```

Существует ряд препятствий для использования графических процессоров при реализации запросов на основе архитектуры SQLite:

- Внутренняя структура хранения данных (B-tree).
- Ограничения архитектуры GPU.
- Вопросы хранения данных (исходных и результатов).

Множество обрабатываемых данных было ограничено столбцами с типами целое (int) и число одинарной точности с плавающей точкой (float). Тестовые таблицы представляют собой набор кортежей, в каждом из которых — пять элементов. В дополнение к реализации функции SELECT авторы реализовали набор примитивных операций, таких как: SUM, MIN, MAX, COUNT, AVG. Данные одной таблицы постоянно хранятся в памяти GPU для того, чтобы снизить размер передаваемых данных через шину для каждого запроса. Для запуска выборки над множеством данных необходимо преобразование из B-дерева в структуру, аналогичную табличному представлению. Поэтому необходимо преобразование из B-дерева к табличной структуре. Тесты проводились на Tesla C1060 GPU, имеющем 4 Гб оперативной памяти, что и является ограничением размера таблицы (для того, чтобы избежать подкачки данных). Множество результатов тоже требует определённого количества памяти (заранее неизвестное). Этот размер выбирается эмпирически, в будущем возможен поиск эвристики, позволяющей свести размер множества результатов к минимуму. Для метаданных, таких как размер блока, число колонок в блоке, размер каждой колонки, предусмотрена отдельная область памяти.

Существует ряд факторов, которые необходимо учесть при сравнении результатов, полученных на CPU и GPU:

- Мы загружаем данные на GPU полностью, в то время как SQLite на CPU запрашивает их с диска по необходимости. Для того чтобы устранить очевидное преимущество GPU, в этом случае в SQLite использовалась таблица, которая хранит содержимое таблицы полностью в памяти.
- Чтобы сделать традиционное исполнение SQLite быстрее, результаты, возвращаемые ей, игнорировались, а также она была скомпилирована с опцией хранения всех временных файлов в памяти.
- Не использовалась страничная память (pinned или paged-memory), которая позволяет примерно в два раза увеличить скорость обмена данными между host и device. Это означает, что результаты, в которых учитывается время передачи данных на GPU, могут быть лучше, если провести ряд оптимизаций, основанных на этом типе памяти.

На скорость исполнения влияет такое явление, как расхождение потоков (thread divergence) из-за присутствия условных переходов. Оно заключается в том, что GPU исполняет код условного перехода по разу для каждого условия последовательно. То есть, имея обычную ветку if... then... else, этот код будет выполнен сначала для тех потоков, которые в первой части имеют истинное значение, и затем для тех, которые имеют ложное. Полученные результаты

в этом исследовании можно разделить на те, в которых учитывалось время передачи с хоста на устройство и не учитывалось. Без учёта времени загрузки данных на GPU запросы выполнялись в среднем в 50 раз быстрее, чем на обычном процессоре, если учитывать, что время передачи в среднем GPU быстрее в 36 раз (табл. 1).

Таблица 1

Ускорение исполнения запросов на GPU по отношению к CPU

Запросы (тип)	Ускорение на GPU	Ускорение на GPU (со временем передачи)
int	42,11	28,89
float	59,16	43,68
aggregation($AND, OR, etc.$)	36,22	36,19
average	50,85	36,20

2.2. Map-Reduce framework Mars на GPU

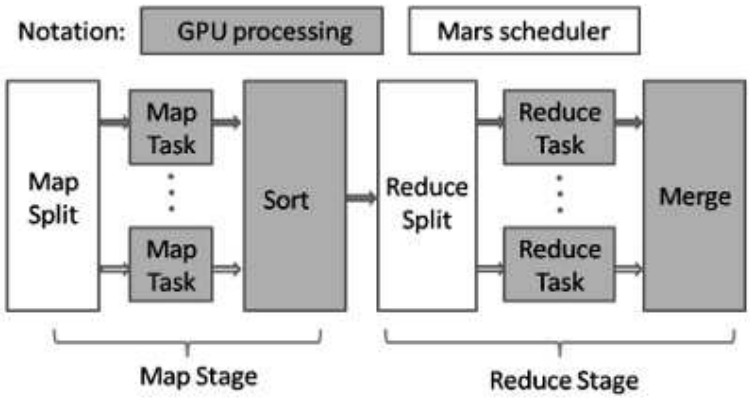

Рис. 1. Архитектура Mars

Компания Microsoft провела исследование, которое показало эффективность Map-Reduce алгоритмов на GPU [3]. В ходе исследований был реализован фреймворк, облегчающий реализацию задач, которые могут быть эффективно решены при переносе их на Map-Reduce фазы. Изначально подобный подход использовался в программировании кластеров с большим количеством компьютеров, чтобы получить вычислительную мощность, сравнимую с суперкомпьютерами на commodity hardware. MapReduce — это фреймворк для вычисления некоторых наборов распределённых задач с использованием большого количества компьютеров (называемых нодами), образующих кластер (ядер в нашем случае). Работа MapReduce состоит из двух шагов: Map и Reduce. На Map-шаге происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным узлом — master node) получает входные данные задачи, разделяет их на части и передаёт другим компьютерам (рабочим узлам — worker node) для предварительной обработки. Название данный

шаг получил от одноимённой функции высшего порядка. На Reduce-шаге происходит свёртка предварительно обработанных данных, главный узел получает ответы от рабочих узлов и на их основе формирует результат — решение задачи, которая изначально формулировалась.

Преимущество MapReduce заключается в том, что он позволяет распределённо производить операции предварительной обработки и свёртки. Операции предварительной обработки работают независимо друг от друга и могут производиться параллельно (хотя на практике это ограничено источником входных данных и/или количеством используемых процессоров). Аналогично множество рабочих узлов могут осуществлять свёртку — для этого необходимо только, чтобы все результаты предварительной обработки с одним конкретным значением ключа обрабатывались одним рабочим узлом в один момент времени. Хотя этот процесс может быть менее эффективным по сравнению с более последовательными алгоритмами, MapReduce может быть применён к большим объёмам данных, которые могут обрабатываться большим количеством серверов. Так, MapReduce может быть использован для сортировки петабайта данных, что займёт всего лишь несколько часов. Классический пример использования MapReduce — это подсчёт слов в документе:

```
// Функция, используемая рабочими нодами на Map-шаге
// для обработки пар ключ-значение из входного потока
map(String name, String document):
    // Входные данные:
    //   ключ --- название документа
    //   значение --- содержимое документа
    // Результат:
    //   ключ --- слово
    //   значение --- всегда 1
    for each word w in document:
        EmitIntermediate(w, "1");

// Функция, используемая рабочими нодами на Reduce-шаге
// для обрабоки пар ключ-значение, полученных на Map-шаге
reduce(String word, Iterator partialCounts):
    // Входные данные:
    //   ключ --- слово
    //   значения --- всегда 1. Количество записей в partialCounts и есть
    //     требуемое значение
    // Результат:
    //   общее количество вхождений слова word во все
    //     обработанные на Map-шаге документы
    int result = 0;
    for each v in partialCounts:
        result += parseInt(v);
    Emit(AsString(result));
```

Поскольку GPU пока ещё не поддерживает динамическое выделение памяти на устройстве во время исполнения кода на нём, были использованы заранее выделенные массивы для работы фреймворка. Входящие данные, промежуточные и конечные результаты хранятся в трёх различных типах массивов: массив ключей, массив значений и индекс. Под индексом понимается запись вида: <key offset, key size, value offset, value size> для каждой пары ключ-значение. Имея индексную запись для каждой пары ключ–значение, мы запрашиваем ключ или значение по соответствующему смещению в массиве ключей или значений.

Рис. 1 иллюстрирует принцип работы MapReduce фреймворка. Так же как MapReduce, фреймворк, исполняемый на обычном процессоре, имеет две фазы.

На стадии map оператор MapSplit разделяет пары входных значений на равные наборы таким образом, число наборов было равно числу потоков. Таким образом достигается балансировка на стадии map — каждый поток ответственен за один набор. После окончания фазы map промежуточные пары ключ–значение сортируются таким образом, чтобы пары с одинаковым ключом хранились последовательно.

На стадии reduce оператор ReduceSplit делит отсортированные промежуточные пары ключ–значение на наборы одинакового размера. Пары с одинаковыми ключами принадлежат одному набору. Число наборов равно числу потоков. Поток с наибольшим threadID отвечает за наборы с большими ключами. Это даёт гарантию, что вывод фазы reduce будет также отсортирован. На последнем шаге вывод со всех потоков записывается в один общий буфер.

Существует планировщик, который отвечает за некоторые стадии алгоритма и исполняется на CPU. В задачи планировщика входит: подготовить входные пары ключ–значение, вычислить необходимое число потоков, скопировать входные массивы на GPU и забрать результаты исполнения.

На GPU выделяется место как для входных данных, так и для результата до исполнения программы на GPU. Но размеры выходных данных из фаз map и reduce неизвестны заранее, более того, существуют конфликты по записи, когда много потоков записывают результаты в общий массив, так как в модели CUDA нет механизмов, обеспечивающих синхронизацию между разными блоками исполнения на аппаратном уровне. Это потребовало реализации собственного механизма записи результатов (который одинаков для обоих фаз).

Каждая из задач map имеет результат в виде трех чисел: число промежуточных результатов, размер ключей (в байтах) и размер значений (в байтах). Основываясь на суммарном объеме ключей (либо значений), планировщик вычисляет префиксную сумму [2] (которая может быть вычислена параллельно) и создаёт массив, необходимый для хранения промежуточных результатов. Далее так же вычисляется место для индекса. Таким образом планировщик выделяет нужное количество памяти на устройстве для хранения промежуточных результатов.

Далее каждая map задача выводит промежуточный набор ключ–значение в выходной массив и обновляет индекс. Так как каждая из задач map знает позиции, в которые она должна писать, то конфликты по записи исключены.

Эти две стадии существуют на GPU, так как он не поддерживает динамического выделения памяти с кода, исполняющегося на устройстве, как уже говорилось. Это стало причиной существования двух стадий в каждой из фаз (map или reduce). Первая стадия отвечает за подсчёт необходимого места для своих результатов, вторая делает всю работу и выводит результат в заранее выделенный массив. Для программиста это выглядит так (фаза map для подсчёта слов в документе):

```
MAP\_COUNT(key, val, keySize, valSize){
1: for each word w in key
2:    EMIT\_INTERMIDIATE\_COUNT(w.length, sizeof (int));
}

MAP (key, val, keySize, valSize) {
1: for each word w in key
2:      EMIT\_INTERMIDIATE (w, 1);
}
```

3. Заключение

Мы рассмотрели наиболее значимые на данный момент исследования в области применения GPU для обработки данных. Несмотря на существующие ограничения архитектуры GPU, результаты говорят о существенном преимуществе в области обработки больших объёмов данных. Мощность графических процессоров растёт быстрее, чем мощность центральных процессоров по закону Мура (который для CPU уже перестал выполняться), поэтому существенными ограничениями являются скорость шины и объем памяти. Частично вопрос нехватки памяти решается использованием pinned memory, т. е. проецированием памяти RAM и передачей порциями на GPU по запросу. Так как шина PCI-E является полнодуплексной, то вместе с запросами на чтение из pinned memory возможны одновременные запросы на запись. Усложнённая модель программирования в сравнении с CPU накладывает свои ограничения: отсутствие возможности динамического выделения памяти на GPU, невозможность синхронизации между исполняемыми блоками приводят к появлению конструкций, которые добавляют накладные расходы там, где на CPU их нет. Такие особенности архитектуры, как расхождение потоков, конфликты банков разделяемой памяти, особенности доступа к глобальной памяти, обусловливают необходимость адаптации существующих примитивов для параллельного программирования (сортировка, префиксная сумма) для GPU.

Литература

1. Bakkum P., Skadron K. Accelerating SQL Database Operations on a GPU with CUDA / Department of Computer Science University of Virginia.
2. Harris M. Parallel Prefix Sum (Scan) with CUDA / NVIDIA corp.
3. Mars: A MapReduce Framework on Graphics Processors / Bingsheng He, Wenbin Fang, Qiong Luo and other; Microsoft research.
4. NVIDIA CUDA Compute Unified Device Architecture / NVIDIA corp.
5. Parallel Data Mining on Graphics Processors / Wenbin Fang, Ka Keung Lau, Mian Lu and others; Microsoft Research Asia, Microsoft China Co.
6. Relational Joins on Graphics Processors / Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. ACM SIGMOD, 2008.
7. Relational Query Co-Processing on Graphics Processors / Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. TODS, 2009.
8. Schatz M. C., Trapnell C. Cmatch: Fast Exact String Matching on the GPU. URL: http://www.cbcb.umd.edu/software/cmatch/ (дата обращения: 20.10.2010).
9. Sintorn E., Assarsson U. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. URL: http://www.cse.chalmers.se/~uffe/hybridsort.pdf (дата обращения: 20.10.2010).
10. Why graphics processors will transform database processing / IEEE Spectrum. URL: http://www.spectrum.ieee.org/computing/software/data-monster/0 (дата обращения: 20.10.2010).