электронное научно-техническое издание
НАУКА и ОБРАЗОВАНИЕ
Эл № ФС 77 - 30569. Государственная регистрация №0421100025. ISSN 1994-0408

Параллельные алгоритмы реляционного соединения на графическом процессоре

77-30569/234879

10, октябрь 2011

Коробицын В. В., Лыфарь Д. А.

УДК 004.657

Омский государственный университет им. Ф.М. Достоевского
korobits@gmail.com
dlyfar@gmail.com

Введение

Потенциал графических процессоров (GPU) в задачах обработки больших массивов данных привел к появлению множества работ, показывающих высокую производительность в сравнении с центральным процессором (CPU). Наиболее близкие по тематике работы [1]–[5] посвящены применению GPU для обработки баз данных. В [1] авторы описали реализацию СУБД GDB на основе примитивов, заимствованных из функциональных языков. Реализации операций выборки и соединения посвящены работы [3]–[5], операции группировки — [1].

В СУБД GDB любой исполняемый запрос является последовательностью примитивов из набора: map, reduce, scatter, gather, scan. Реализована возможность выполнения операции упорядочивания, группировки и соединения над данными. GDB поддерживает несколько способов доступа к данным, как-то: полное сканирование, доступ по хэшу и дереву. Однако, в GDB отсутствует поддержка SQL, а запросы задаются вручную, так как не реализовано автоматическое преобразование запроса в набор примитивов. Однако GDB отличается от аналогичных работ наличием реализации операции реляционного соединения, которая, как известно, одна из самых трудоемких операций для процессора. В традиционных СУБД известно несколько способов реализации операции реляционного соединения: посредством вложенных циклов (с использованием индекса и без него), основанное на сортировке и с использованием хэширования [6]. В GDB рассмотрены реализации всех четырех алгоритмов и приведены результаты тестирования соединения двух таблиц, которые можно обобщить на произвольное их количество.

Для сохранения результата соединения в оперативную память компьютера используется трехпроходный алгоритм:

1. Каждый поток подсчитывает в своей локальной памяти число кортежей, попадающих в результат.
2. Вычисляется префиксная сумма по всем счетчикам, что дает смещение в глобальной памяти для записи результата конкретного потока и общее число кортежей.
3. Хост выделяет необходимое число памяти для записи результата и, если, он не помещается в память устройства полностью, запись результата ведется в несколько потоков.

Недостатком этой схемы записи результата является то, что условие соединения вычисляется дважды: первый раз, когда подсчитывается число кортежей в результате и второй раз, когда мы ведем запись результата в глобальную память, однако доля вычислений по сравнению с операциями над памятью на GPU в таких задачах относительно мала. Достоинством по сравнению с другими подходами является отсутствие необходимости поддержки оборудованием атомарных операций с памятью.

В данной статье мы рассматриваем два алгоритма реляционного соединения на основании вложенных циклов и их адаптацию для открытой СУБД MySQL. Приведены, также, результаты оценки производительности для различных типов данных (на данный момент поддерживаются целочисленные, вещественные и строковые типы). В отличие от GDB мы обобщили алгоритм на случай, когда невозможно данные и результат полностью поместить в память видеоадаптера. Кроме того, реализована поддержка соединения по строковым типам данных.

1. Неиндексированные вложенные циклы

В [6] рассмотрены два последовательных алгоритма соединения таблиц по не индексированным столбцам. Первый случай, когда обе таблицы помещаются в оперативную память и второй, обобщенный, когда ни одна из таблиц не размещается в оперативной памяти полностью. Мы рассмотрим оба этих алгоритма с соответствующей адаптацией к параллельной архитектуре GPU. Введем дополнительные обозначения: $B(S)$, $B(R)$ — число блоков, необходимых для представления таблиц в памяти для второго случая, поскольку обрабатывать данные блочно в этом случае эффективнее. M — число блоков, которыми может быть представлена доступная оперативная память, $B(T)$ — место, отведенное под хранение промежуточного результата. В представленных алгоритмах мы будем использовать

вспомогательную процедуру `CheckJoinCondition`, сравнивающую атрибуты обеих таблиц, участвующих в соединении. Во всех алгоритмах мы использовали параллельный алгоритм префиксной суммы для подсчета количества места, необходимого для хранения результата текущей итерации соединения. Подробный анализ этого алгоритма приведен в [7].

Алгоритм 1 (Псевдокод параллельного алгоритма соединения на основе вложенных циклов. Частный случай, когда $B(S)+B(R)+T\leq M$.)

Вход: последовательность n кортежей отношения R, последовательность m кортежей отношения S имеющие общий атрибут A.

Выход: последовательность k кортежей отношения K с заголовком, полученным из объединения заголовков R и S, имеющих одинаковое значение атрибута A.

```
1    tIdx=blockDim.x*blockIdx.x+threadIdx.x;
2    offset=gridDim.x*blockDim.x;
3    count=0;
4    shared S_buff[];
5    constant C[];
6    matched=0;
7    foreach R r do in parallel
8     for (i=tIdx; i<|S| and count*sizeof(S0)<sizeof(S_buff)/blockDim.x;
9          i+=offset) do in parallel
10              S_buff[threadIdx.x+count]=S_i;
11              count++;
12         end
13    for S_buffs do in parallel
14         cond=True;
15         foreach C c do
16              if !(cond=C_i(s)) then break;
17         end
18    end
19    if cond then matched++;
20   end
21   syncthreads();
22   gPrefix[tIdx]=matched;
23   prefixsum(gPrefix);
24   res_offset=gPrefix[tIdx];
25   for S _ buffs do in parallel
26        if C_i(s) then result[res_offset++]=s;
27   end
```

В нашей реализации поддерживается тот же набор типов, что и в операции выборки. Существует несколько вариантов реализации соединения на GPU. Мы реализовали аналоги последовательных алгоритмов из MySQL: неиндексированные и индексированные вложенные циклы. Оба этих алгоритма могут быть исполнены на GPU практически без изменений.

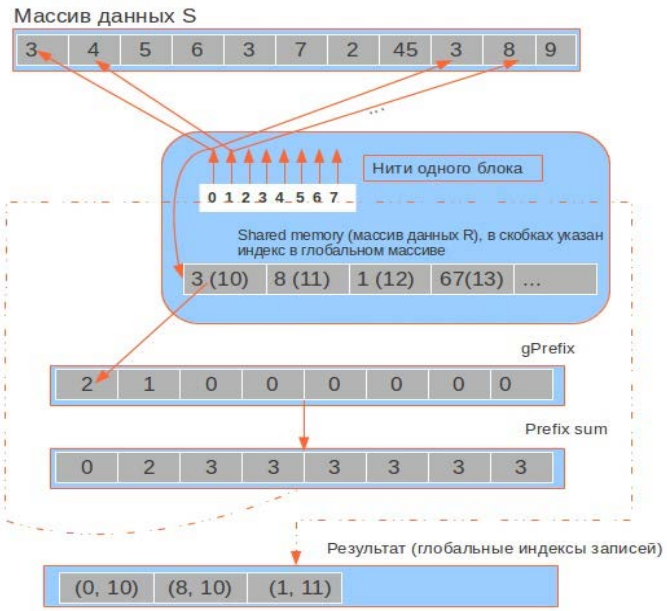


Рис. 1. Диаграмма работы параллельного алгоритма соединения по не индексированным данным

Рассмотрим частный случай операции соединения для двух отношений R и S. Как показано на рис. 1, каждая группа потоков (блок) B вычисляет операцию соединения над своей порцией данных, обозначенных R' и S' соответственно. Каждый поток из группы отвечает за вычисление соединения одного кортежа с каждым кортежем из R'. Таким образом, данные отношения S разбиты на группы с числом кортежей, равных числу потоков в блоке. Здесь мы должны учесть размер локальной памяти каждого из потоков. Для того чтобы каждый раз не обращаться за данными к отношению R, хранимому в глобальной памяти, мы копируем R' в разделяемую память (shared memory). Таким образом, размер R' в байтах не должен превышать объем доступной потоку разделяемой памяти. Результатом соединения является массив пар индексов строк из отношений R и S. Далее полученный массив копируется обратно в оперативную память, и соединяемые строки возвращаются клиенту.

2. Соединение по не индексированным столбцам, когда B(S)+B(R)≥M

Более эффективным методом является блочная обработка данных. Предположим, что $B(S) \leq B(R)$, тогда, почти наверняка, оптимизатор выберет таблицу *S* для внешнего цикла. В отличие от последовательного алгоритма для традиционной архитектуры, здесь необходимо учитывать размер результата. В случае если результат не помещается в память, он считывается в несколько проходов. Положим, что $B(S)=M-2$, $B(R)=B(S)-M-1$ и $B(T)=B(R)$, а так же, что мы заранее выделили необходимый объем памяти для хранения результатов префиксной суммы $P=K$.

Стоит обратить внимание на то, как алгоритм сохраняет результаты (строки 21-27). Это фаза алгоритма, в которой участвуют не все потоки, а только те, которые в соответствие со своим индексом могут записать результат, т.е. одновременно исполняются максимум $K/|T|$ потоков, остальные вынуждены простаивать, пока полученный результат не будет скопирован в оперативную память. Происходит следующее: каждый поток считывает запись в соответствие со своим индексом из разделяемой памяти, далее происходит либо запись в глобальную память *result* по смещению, либо поток имеет полученный после prefixsum индекс больший, чем количество элементов, которые *result* способна в себя вместить, поэтому вынужден ждать до тех пор, пока его корректирующийся в строке 25 индекс не примет значение, удовлетворяющее этому условию.

3. Индексированные вложенные циклы

Несмотря на превосходство GPU в алгоритме 1, соединение по неиндексированным данным имеет небольшое практическое значение, т.к. таблицы проектируются таким образом, чтобы соединения выполнялись как можно быстрее. Как правило, индексы в традиционных СУБД хранятся в одной из форм B-дерева, уменьшая количество операций ввода-вывода. В нашем случае доступ к данным дерева по ключу на GPU будет неоптимальным, поэтому, как правило, применяется одна из техник хранения дерева в виде непрерывного массива CSS-дерево (CSS-tree), описанная в [8].

CSS-дерево хранит B+-дерево в виде массива, доступ к узлам, которого осуществляется при помощи адресной арифметики, что является хорошим компромиссом между используемым объемом памяти нужной для хранения дерева и количеством вычислений для доступа к определенному узлу. CSS-дерево конструируется непосредственно на GPU из отсортированного массива [3], однако в нашем случае структура дерева скрыта от нас за интерфейсом хранилища, и мы просто имеем гарантию того, что данный массив отсортирован по ключу.

Еще одной причиной отказа от CSS-дерева послужило то, что данные могут не помещаться в память полностью. Поэтому эффективнее их считывать, используя API MySQL, в уже отсортированном виде и обрабатывать порциями. Соединение с использованием индекса (здесь мы также рассматриваем соединения между двумя отношениями *R* и *S*) осуществляется за два шага: поиск в индексе первого вхождения обрабатываемого значения и считывание всех последующих равных значений из него. Таким образом, мы одновременно ищем вхождения для нескольких ключей в индексе, что позволяет полностью использовать параллельность платформы.

Для поиска по массиву можно использовать последовательный O(n) или бинарный $O(\log_2 n)$ поиск. В случае бинарного поиска мы имеем меньшее число сравнений, чем достигается преимущество на последовательной версии алгоритма, однако для GPU это может быть невыгодным вариантом из-за сильного ветвления и вызвать последовательное исполнения большого числа групп потоков. Количество блоков, предполагаемых для хранения обеих таблиц и результата положим таким же, как и в предыдущем разделе и рассмотрим общий случай.

Алгоритм 2. (Псевдокод параллельного алгоритма реляционного соединения для $B(S)+B(R)>M$. Индексированные циклы.)

```
1    register tIdx=blockDim.x*blockIdx.x+threadIdx.x;
2    register offset=gridDim.x*blockDim.x;
3    register count=0;
4    shared S_buff[];
5    constant C[];
6    register matched=0;
7    for (r=0; r<B(R); r+=M-2) do
8        copy_to_device(s, s+M-2);
9        for (s=0; s<B(S); s+=1) do
10         copy_to_device(r, r+1);
11         for (i=tIdx; i<|R|;i+=offset, count+=1) do in parallel
12           if (S_value==BinarySearch(R_i, S)) then
13             S_buff[threadIdx.x+count]=(R_i, S_value);
14             matched++;
15           end
16         end
17       end
18   end
19   syncthreads();
20   gPrefix[tIdx]=matched;
```

```
21  prefixsum(gPrefix);
22  res_offset=gPrefix[tIdx];
23  while(res_offset>=0) do in parallel
24    for (i=threadIdx.x; i<threadIdx.x+matched and
25                        res_offset<sizeof result; ++i) do in parallel
26      result[res_offset++] = S_buff[i];
27    end
28    res_offset-=sizeof result;
29    copy_to_host(result);
30  end
```

В приведенном описании алгоритма существование индекса по соединяемому атрибуту таблицы *S*. Описанный способ соединения для каждого кортежа таблицы *R* выполняет бинарный поиск в данных *S*. Перед выполнением соединения индексные атрибуты (или их часть) *S* загружается в глобальную память так же, как и обычные атрибуты, но в заранее упорядоченном виде.

4. Анализ производительности

В тестах производительности операции соединения использовались с несколькими различными наборами данных, для того чтобы промоделировать различные сценарии, как-то:

- Результат соединения полностью помещается в память GPU.
- Результат соединения не помещается в память GPU и считывается за несколько проходов. Также было оценено изменение ускорения обработки на GPU относительно CPU с ростом объема данных.
- Результат соединения для различных типов данных: целочисленные и строковые.
- Соединение выполняется по индексируемым полям, либо нет.

Предварительно было выполнено несколько запусков, чтобы подтвердить, что расчетное значение числа потоков в блоке является оптимальным. Это означает, что мультипроцессоры полностью загружены во время выполнения. В качестве тестовых данных использовались 2 соединяемых таблицы MySQL. К таблице Domains со структурой:

```
Domains(id INT, domain_name CHAR(64), ip INT UNSIGNED, PRIMARY KEY (id))
```

добавлялась еще одна, имеющей следующую структуру:

```
Clients(client_id INT, income_ip UNSIGNED INT, PRIMARY KEY (client_id))
```

Здесь `income_ip` – это IP адрес клиента, который запросил домен `Domains.domain`. А также таблицу, объединяющую идентификаторы доменов и IP адресов клиентов

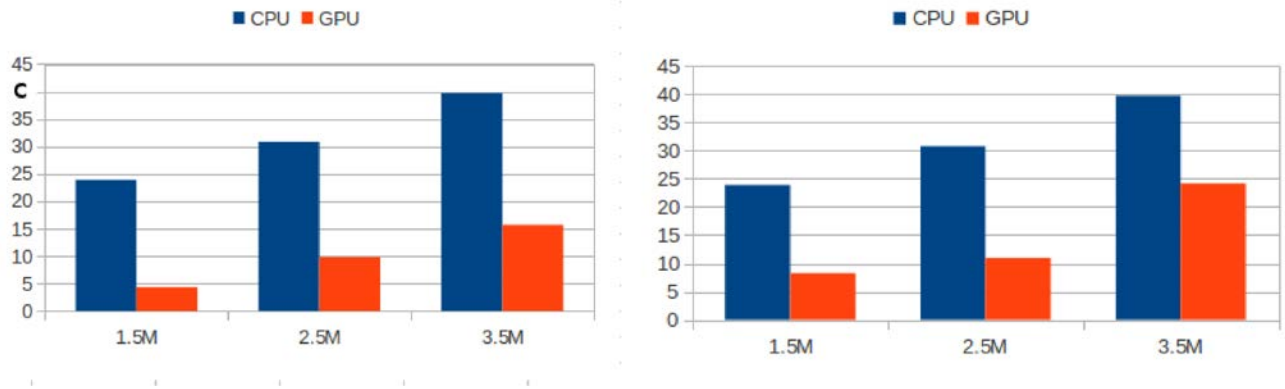
```
Log(id INT, client_id INT)
```

Здесь поля `id` и `client_id` – это внешние ключи на таблицы `Domains` и `Clients` соответственно. Общий объем соединяемых данных занимает около 1 ГБ, каждая таблица имеет по 3.5 миллиона строк. Основываясь на приведенном выше списке возможных сценариев, мы составили несколько SQL запросов, покрывающих этот список:

```
SELECT COUNT(*) FROM Domains d LEFT JOIN Clients c ON (d.id=c.id)
SELECT l.client_id FROM Domains d LEFT JOIN Log l ON (d.id=l.id) LIMIT 100
SELECT l.client_id FROM Domains d LEFT JOIN Log l ON (d.id=l.id) LIMIT 1000
SELECT l.id FROM Clients c LEFT JOIN Log l ON (c.client_id=l.client_id)
SELECT d.domain_name FROM Domains d LEFT JOIN Log l ON
      (d.domain_name='nvidia.com')
```

Некоторые из приведенных запросов не имеют смысла, т.е. являются «синтетическими», чтобы обеспечить тестирование конкретных мест реализации. Результаты производительности усреднены по запросам для каждого из обрабатываемых объемов данных.

Из-за временной оценки $O(n^2)$ неиндексированные вложенные циклы являются самым неэффективным алгоритмом соединения, что и подтверждается замерами производительности – этот алгоритм является самым медленным. Однако превосходит последовательную реализацию, работающую на CPU (в тестировании этого алгоритма, индексы по колонкам были отключены). Результаты производительности неиндексированного алгоритма вложенных циклов приведены на рис. 2.



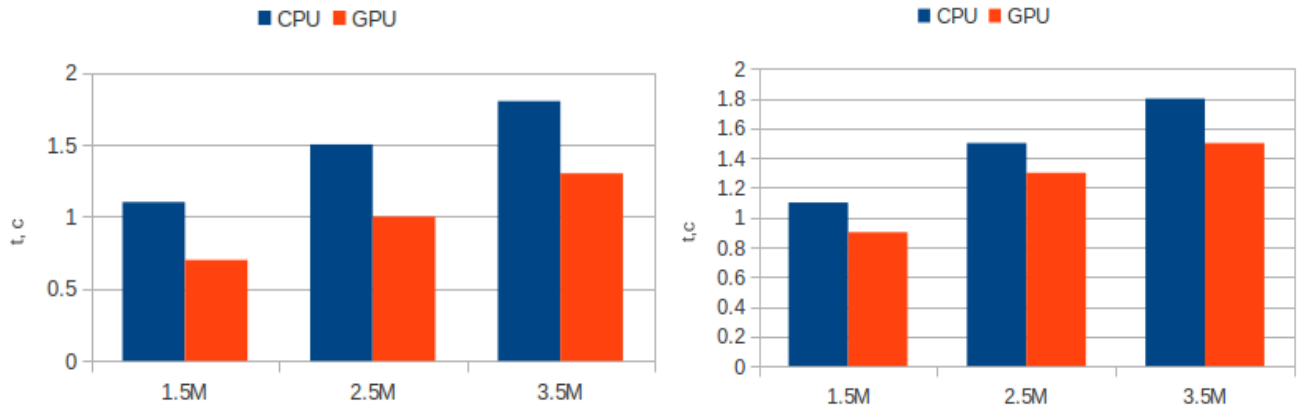
а) *б*)

Рис. 2. Производительность алгоритма неиндексированных вложенных циклов:
а) случай, когда результат полностью помещается в память GPU;
б) случай, когда результат не помещается в память GPU и вычисляется в 5 проходов

Для того чтобы понаблюдать, как изменяется поведение алгоритма с ростом объема данных, мы запускали алгоритм на разных объемах одних и тех же таблиц (1.5, 2.5 и 3.5 миллиона строк на каждую из таблиц, участвовавших в соединении). Необходимо обратить

внимание на рост времени исполнения на GPU из-за нескольких проходов в случае (б). На рисунке показан частный случай, где исполнителю запросов пришлось выполнить 5 проходов для копирования результатов в глобальную память.

Аналогичные тесты производительности были проведены для второго алгоритма, который использует бинарный поиск во внутреннем цикле. На рис. 3 приведены результаты тестирования, где видно, что в случае соединения по индексируемым данным GPU имеет не такое большое преимущество, в среднем быстрее CPU в 1.4 раза.



а) *б*)

Рис. 3. Оценка производительности алгоритма индексированных вложенных циклов:

а) случай, когда результат полностью помещается в память GPU;

б) случай, когда результат не помещается в память GPU и вычисляется в 5 проходов.

Заключение

Мы рассмотрели два параллельных алгоритма соединения и предложили обобщенное решение для случая, когда данные или результат не могут быть обработаны за один проход. Была реализована поддержка целочисленных, вещественных и строковых типов данных. В случае с алгоритмом вложенных циклов для неиндексированных данных получено в среднем пятикратное ускорение, в случае с индексированными данными GPU превосходит CPU не более чем в 1.4 раза. Полученные результаты дают основание полагать, что использование GPU совместно с CPU в задачах вычисления реляционного соединения в СУБД является целесообразным и позволяет увеличить общее число запросов, исполняемых в единицу времени.

Литература

1. He B., Yang K., Fang R.. et al. Relational joins on graphics processors // Proceedings of the 2008 ACM SIGMOD international conference on Management of data. 2008. 14 pp.
2. He B., Lu M., Yang K. et al. Relational query coprocessing on graphics processors // ACM Transactions on Database Systems. Vol. 34, N 4. Article 21. 2009. 39 pp.
3. He B., Govindaraju N.K., Luo Q., Smith B. Efficient gather and scatter operations on graphics processors // Proceedings of the 2007 ACM/IEEE conference on Supercomputing. 2007. 12 pp.
4. He B., Fang W., Luo Q., et al. Mars: a MapReduce framework on graphics processors // Proceedings of PACT 2008. P. 260–269.
5. Lieberman M. D., Sankaranarayanan J., Samet H. A fast similarity join algorithm using graphics processing units // Proceedings of the IEEE 24th International Conference on Data Engineering, 2008. P. 1111–1120.
6. Гарсиа-Молина Г., Ульман Д., Уидом Д. Системы баз данных. Полный курс. М.: Вильямс, 2004.
7. Harris M., Sengupta S., Owens J. D . Parallel prefix sum (scan) with CUDA // GPU Gems 3. Chapter 39. 2007. P. 851–876.
8. Rao J., Ross K.A. Cache Conscious Indexing for Decision-Support in Main Memory. Columbia University Technical Report. 1998. 18 pp.

electronic scientific and technical periodical
SCIENCE and EDUCATION
EL № FS 77 - 30569. №0421100025. ISSN 1994-0408

Parallel algorithms for relational connection on graphics processors

77-30569/234879

 # 10, October 2011

Korobicyn V.V., Lyfar' D.A.

Omsk F.M. Dostoevsky State University
korobits@gmail.com
dlyfar@gmail.com

We present two parallel algorithms for relational connection on graphics processing units (GPU). The article considers the cases when the connection result is located in GPU memory and is returned by portions. Performance tests show five-fold advantage of GPU over CPU for non-indexed nested loops and 1.4-fold advantage for indexed nested loops.

Publications with keywords: join, parallel algorithm, graphics processors, relational database
Publications with words: join, parallel algorithm, graphics processors, relational database

Reference

1. He B., Yang K., Fang R., et. al., Relational joins on graphics processors, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008, 14 pp.
2. He B., Lu M., Yang K. ., et. al., Relational query coprocessing on graphics processors, ACM Transactions on Database Systems, 34 (4) (2009) Article 21, 39 pp.
3. He B., Govindaraju N.K., Luo Q., Smith B., Efficient gather and scatter operations on graphics processors, in: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007, 12 pp.
4. He B., Fang W., Luo Q., et al., Mars: a MapReduce framework on graphics processors, in: Proceedings of PACT, 2008, pp. 260–269.
5. Lieberman M. D., Sankaranarayanan J., Samet H., A fast similarity join algorithm using graphics processing units, in: Proceedings of the IEEE 24th International Conference on Data Engineering, 2008, pp. 1111–1120.
6. Garsia-Molina G., Ul'man D., Uidom D., Database systems. A full course of, Moscow, Vil'iams, 2004.
7. Harris M., Sengupta S., Owens J. D., Parallel prefix sum (scan) with CUDA , in: GPU Gems 3, Chapter 39, 2007, pp. 851–876.
8. Rao J., Ross K.A., Cache Conscious Indexing for Decision-Support in Main Memory, Columbia University Technical Report, 1998, 18 pp.