Наука и Образование
МГТУ им. Н.Э. Баумана
Сетевое научное издание
ISSN 1994-0408

Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. № 10. С. 271–287.

DOI: **10.7463/1015.0820521**

Представлена в редакцию: 10.09.2015
Исправлена: 26.09.2015

УДК 004.051+519.171

Оптимизация вычислений над связными множествами в CUDA

профессор Иванова Г. С.[1,*], *gsivanova@gmail.com
Головков А. А.[1]

[1]МГТУ им. Н.Э. Баумана, Москва, Россия

В статье рассмотрены программно-аппаратные особенности графических процессоров CUDA с целью поиска методов оптимизации вычислений при реализации параллельных алгоритмов операций преобразования графов. Анализ модели исполнения CUDA-программы и организации работы с памятью позволил определить общие требования и правила для разработки параллельных алгоритмов и структур данных для представления графов. На основе полученных результатов были предложены несколько вариантов параллельной реализации операции декомпозиции вершины графа. Анализ экспериментальных данных позволил выявить набор факторов, существенно влияющих на ускорение параллельных алгоритмов. Результаты исследований предоставят возможность разрабатывать эффективные алгоритмы с учетом специфики программирования на графических процессорах.

Ключевые слова: операция над графом, параллельный алгоритм, CUDA, оптимизация, поток, мьютекс, структура данных

Введение

Многие алгоритмы решения задач структурного анализа и синтеза на графовых моделях [1, 2, 5-8, 11, 14, 22] имеют большой запас внутреннего параллелизма, так как включают однотипные операции, выполняемые над множествами вершин, ребер и/или их отображениями. При этом с теоретической точки зрения ускорение зависит только от размерности входных данных задачи при реализации на параллельной системе с достаточным количеством процессоров. Так в [8] показано, что оценка ускорения параллельных алгоритмов для основных операций преобразования графов прямо пропорциональна количеству вершин n и ребер m графа.

Как правило, любые теоретические оценки основываются на следующих допущениях:

- количество одновременно выполняющихся потоков (нитей) равно необходимому количеству потоков для алгоритма;

- все потоки равноправно работают с памятью, конфликты при одновременном чтении/записи исключены;
- структуры данных не требуют выделения памяти при работе с ними.

При реализации алгоритмов в конкретной параллельной вычислительной системе коэффициент ускорения может значительно измениться как в худшую, так и в лучшую сторону. Чтобы на практике получить существенное ускорение, необходимо учитывать архитектурные особенности систем параллельной обработки, преимущества и недостатки различных моделей и технологий параллельного программирования с целью использования лучших подходов, приемов и методов.

Целью настоящей работы является выявление факторов, не позволяющих достигнуть расчетных коэффициентов ускорения параллельных алгоритмов операций над графовыми моделями [3, 5-10] при использовании параллельной системы GPGPU CUDA [12, 13, 23], и выработка рекомендаций, следование которым приведет к снижению непроизводительных временных затрат.

1. Особенности модели исполнения программ в графических процессорах CUDA

Согласно [13] CUDA-программа, написанная на языках C++, C или Fortran, состоит из последовательных частей (host-код), выполняемых на обычном центральном процессоре (CPU), и параллельных частей (device-код), выполняемых на графических процессорах (GPU). Параллельная часть выполняется многими потоками, количество которых явно задается перед началом вычислений и может быть много больше числа потоков N, физически выполняющихся параллельно. В случае, если количество реальных потоком меньше заданного, потоки выполняются в порядке очереди. В соответствии с этим теоретически ускорение любого алгоритма, реализованного на CUDA, не может быть больше N. На современных процессорах GPU архитектуры Maxwell это число может достигать десятков тысяч, что позволяет говорить о возможности достижения высокого реального ускорения особенно при реализации операций над графами больших размерностей.

Все потоки CUDA выполняются варпами (англ. warp) [12] – группами по 32 потока, при этом потоки в варпе выполняются синхронно по принципу «одна команда – множество данных» (SIMD – Single Instruction Multiple Data). В связи с этим при разработке программ могут возникать определенные сложности, допустим, стандартная реализация мьютекса, необходимая для синхронизации потоков при работе со сложной структурой данных:

```
1   //функция выполняется параллельно на многих CUDA-потоках
2   void kernelFunction() {
3       while (atomicCAS(mutex, 0, 1) == 1); //захват мьютекса
4       //критическая секция
5       mutex = 0; //освобождение мьютекса
6   }
```

не будет правильно работать. Поскольку потоки в варпе будут выполнять код синхронно, только один поток выполнит код в строке 2, захватив мьютекс, и будет ожидать остальных, но они не смогут выполнить захват мьютекса, т.к. первый поток никогда его не освободит.

В случае ветвлений в варпе возникает ситуация дивергенции потоков (англ. thread divergence) – ветвление логики исполнения [12]. В силу синхронного выполнения кода, все потоки последовательно выполняют все возможные пути ветвлений, что негативно сказывается на производительности. Дивергенция особенно проявляется при работе многих потоков со сложными структурами данных.

Все CUDA-потоки выполняются блоками, максимальное количество потоков в блоке ограничено 1024 или 512 для разных версий CUDA. На уровне блоков потоки могут взаимодействовать посредством shared-памяти и синхронизации. Из-за ограничений на число потоков в блоке в общем случае нельзя утверждать, что алгоритм может эффективно использовать эту особенность.

Для синхронизации потоков CUDA предоставляет функции барьерной синхронизации в пределах блока. Синхронизация всех блоков может быть обеспечена только при условии их одновременного выполнения, когда количество блоков меньше либо равно количеству мультипроцессоров GPU [21].

2. Работа с CUDA-потоками

Количество необходимых потоков в алгоритмах может превышать не только число реально работающих потоков N, но и логическое количество потоков, которые можно инициализировать при вызове параллельной функции. В таком случае возможно итеративно исполнять параллельную функцию (см. рис. 1).

Рис. 1. Итеративное выполнение параллельной функции

При этом быстродействие может существенно уменьшиться, поскольку инициализация потоков, инициализация GPU, передача управления, копирование переменных в момент вызова параллельной функции займет значительное время.

Для эффективной реализации можно использовать итеративное выполнение кода внутри параллельной функции. В этом случае один CUDA-поток будет выполнять код нескольких логических потоков, необходимых для работы алгоритма (см. рис. 2).

Такая организация вычислений сокращает количество инициализаций потоков до 1 раза. Оба решения хорошо масштабируются на любое число потоков и могут применяться для любых GPU при любых значениях N.

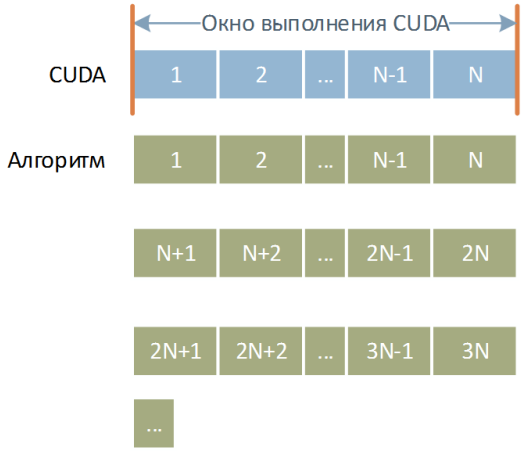

Рис. 2. Итеративное выполнение кода внутри параллельной функции

3. Особенности работы с памятью

Наряду с локальной и разделяемой на уровне блоков памятью (shared-memory) потоки CUDA могут взаимодействовать с глобальной памятью GPU, доступной на чтение и запись всем потокам. Существует несколько приемов работы с глобальной памятью.

В общем случае необходимый объем глобальной памяти выделяется из host-кода программы, после этого память становится доступна из device-кода. Также в host-коде происходит копирование данных из глобальной памяти GPU в оперативную память CPU. Скорость копирования данных по PCI шине на порядки меньше скорости исполнения инструкций потоками, поэтому эффективность реализуемого алгоритма во многом зависит от работы с памятью.

В CUDA-программе может быть выделена page-locked память, в этом случае для нее не применяется своппинг – механизм подкачки страниц, что позволяет использовать DMA для быстрого асинхронного копирования из CPU в GPU. Память такого типа может быть отображена в адресное пространство GPU, при этом формируются 2 указателя (CPU и GPU). Унифицированная модель, позволяет использовать один указатель, при этом копирование памяти происходит автоматически в момент доступа к ней, что значительно упрощает код программы. В целом данная технология за счет увеличения скорости способствует более эффективной работе с памятью.

Выделение глобальной памяти может осуществляться в device-коде. Для этого в начале программы необходимо задать максимальный размер памяти (англ. heap-size), который может быть выделен в процессе выполнения программы функциями `malloc`. Память, выделенная таким образом, не может быть отображена на адресное пространство памяти CPU, следовательно, исключаются все преимущества, описанные выше. Также в силу особенностей CUDA к ней нельзя получить доступ из host-кода, что особенно затрудняет программирование.

Чтобы сравнить время выполнения host- и device-функций выделения памяти был проведен эксперимент на графическом процессоре GeForce GTX 650 Ti, в котором оценивалось время выполнения программы по параллельному заполнению 1024*1024

массивов целыми числами. Размерность каждого массива менялась от 1 до 128. Полученные результаты – зависимости времени выполнения от размера памяти приведены на рисунке 3. В первом случае ($T1$ – красный график) память для массива выделялась на стороне CPU, во втором ($T2$ – синий график) – непосредственно в параллельных потоках CUDA. На графике ось T – время выполнения в секундах, x – общий размер выделяемой памяти в МБ.

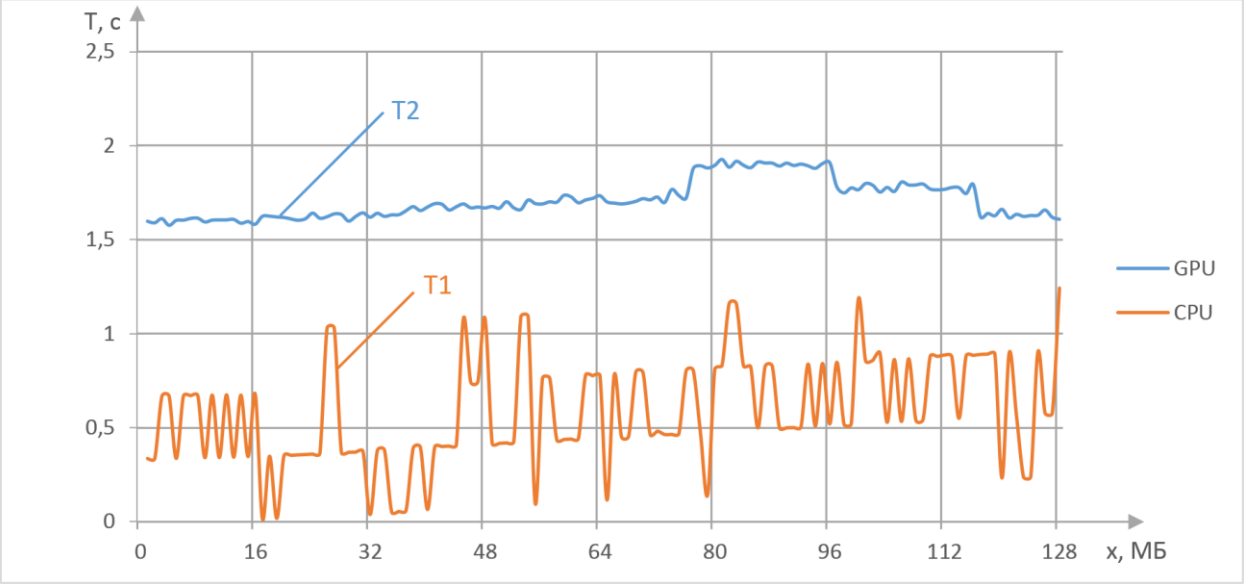

Рис. 3. Зависимость времени выполнения от размера выделяемой памяти

Максимальное значение отношения $T2$ к $T1$ равно 108.243, что говорит о крайней неэффективности использования динамического выделения памяти внутри device-кода, однако такая операция необходима для реализации алгоритмов, где нельзя определить нужный размер памяти до выполнения CUDA-потоков.

При работе с памятью могут возникать ситуации, когда нескольким потокам необходимо получить доступ к одной области памяти, для разрешения таких конфликтов в CUDA предусмотрены различные атомарные функции.

4. Организация структур данных

GPU CUDA – система с общей памятью, организация доступа к ней, кэширование, чтение и запись многими потоками реализована крайне эффективными алгоритмами, однако для разработки эффективных структур данных необходимо придерживаться определенных правил.

Множественный доступ к памяти реализован посредством механизма коалесинга (англ. coalesing) – объединение операций чтения/записи для 16 потоков (половины варпа) с памятью с помощью 32, 64 или 128-байтовых транзакций, при этом данные должны располагаться последовательно, участок должен быть выровнен по размеру линии кэша, используемого для кэширования global памяти GPU (размер линии кэша L2 для compute

capability 3.x равен 32 байта). В соответствии с этим структура данных, во-первых, должна быть максимально плоской – содержать минимум указателей, во-вторых, элементы структуры должны быть размещены максимально последовательно в памяти, что даст значительный прирост ускорения в случае доступа по индексу.

Поскольку при одновременной работе многих потоков с одной структурой данных могут возникать конфликты, необходимо предусмотреть надежный механизм синхронизации и обеспечения атомарности операций структуры данных. Целесообразно использовать lock-free [16, 18, 19] структуры данных, основанные на неблокирующей синхронизации, однако принципы lock-free подходят не для всех типов структур. В этом случае возможно использовать модифицированные для CUDA механизмы синхронизации. Предложим реализацию функции с критической секцией, в которой отсутствует взаимная блокировка потоков:

```
1   //функция выполняется параллельно на многих CUDA-потоках
2   void kernelFunction () {
3       bool isSet = false; // флаг завершения критической секции
4       do {
5           if (isSet = atomicCAS(mutex, 0, 1) == 0) { //захват мьютекса
6               // критическая секция
7           }
8           if (isSet) {
9               mutex = 0; //освобожнение мьютекса
10          }
11      } while (!isSet);
12  }
```

5. Эффективные вычисления в CUDA

На основе выполненного анализа модели исполнения CUDA-программ, работы с памятью и структурами данных определим набор рекомендаций, позволяющих эффективно использовать возможности параллелизации в CUDA.

1. Количество потоков, выделяемых для работы, должно быть меньше либо равно числу потоков, реально работающих в системе. Максимальная загрузка графического процессора и исключение времени на инициализацию дополнительных потоков позволит максимально эффективно использовать GPU.
2. Если количество необходимых потоков превышает число одновременно работающих физических потоков, рекомендуется использовать итеративное выполнение кода внутри параллельной функции (см. раздел 2).
3. Необходимо избегать сильной дивергенции потоков. Если возможно, уменьшить использование атомарных операций.
4. Потоки в разных блоках должны быть максимально независимы по данным. В противном случае для синхронизации конфликтующих блоков необходимо обеспечить их одновременное выполнение. Эффективное взаимодействие потоков в пределах блоках может быть реализовано посредством стандартных процедур и функций CUDA.

5. Необходимо минимизировать передачу данных из CPU в GPU и обратно. Типовая структура CUDA-программы предполагает копирование исходных данных из CPU в GPU, вычисления на GPU и копирование результатов обратно в CPU.
6. При выделении памяти со стороны CPU рекомендуется использовать унифицированную модель работы с памятью, отображенной на GPU. При выделении памяти со стороны GPU необходимо минимально использовать функции malloc в CUDA-потоках.
7. Для снижения фрагментации рекомендуется выделять максимально большие объемы памяти.
8. Структуры данных должны содержать минимум указателей. Элементы структуры должны быть размещены максимально последовательно в памяти, чтобы обеспечить работу механизма коалесинга.
9. Для синхронизации потоков при работе со структурами данных возможно использовать механизмы lock-free, модифицированные мьютексы и семафоры или стандартные функции синхронизации CUDA.

Далее в рамках реализации параллельных алгоритмов операций над графами экспериментально докажем применимость сформированных правил, ограничений и рекомендаций.

6. Структуры данных для представления графов

Перед непосредственной реализацией параллельных алгоритмов необходимо определить структуры данных, необходимые для представления графов в памяти GPU. Множества графа $X, U, \Gamma X, \Gamma U, FX, FU$ и т.д., используемые для описания любых графовых моделей [1, 2], могут быть представлены в матричной форме или аналитически множествами X, U и множествами множеств образов графа по отношениям инцидентности $\Gamma X, \Gamma U$ и смежности FX, FU. Будем основываться на аналитическом представлении, учитывая преимущества и недостатки обоих методов описания [3].

Определим $X, U, \Gamma X, \Gamma U$ как минимальный набор множеств, определяющих граф. Прямые и обратные отношения смежности FX, FU и $F^{-1}X, F^{-1}U$ могут быть определены через эти множества. В общем случае все связи вершин и ребер с точки зрения структур данных симметричны относительно друг друга. С точки зрения ООП связность вершины или ребра графа – составная часть описания объекта вершины или ребра. Нет необходимости четко разделять вершину или ребро от их ориентированных или неориентированных связей с ребрами или вершинами соответственно, поэтому будем представлять общую структуру графа множествами X и U, при этом в каждой вершине и ребре i содержится соответствующее множество ΓX_i и ΓU_i [9].

Реализация операций над графами предусматривает отсутствие ограничений на размерность, тип, ориентированность графа, а также какую-либо его модификацию. Рассматривая граф, как совокупность ранее определенного минимального набора множеств, можно утверждать, что множества также должны соответствовать этому

набору ограничений. В таком случае в качестве множества как единицы построения графа можно взять любую структуру данных, описанную в [3, 4, 9], однако для эффективной реализации необходимо разработать структуру на основе особенностей архитектуры и программирования в CUDA. Оптимальный выбор структур данных, основанный на особенностях алгоритмов решения конкретных задач, а также специфике представления графов в этих задачах [15, 17, 20, 22, 24], заслуживает отдельного исследования.

В качестве динамической структуры данных, удовлетворяющей всем требованиям и ограничениям, предложим динамический массив указателей. Основные операции модификации структуры: удаление, добавление и вставка элемента должны быть реализованы атомарно, поскольку может возникнуть конфликтная ситуация, когда часть работы должен атомарно совершить один поток, например, при добавлении элемента при выходе за границы массива необходимо выделить новую область памяти и копировать все элементы из старой области в новую. Для синхронизации потоков в таком случае будем использовать ранее предложенный модифицированный мьютекс.

Для уменьшения избыточности данных и исключения операций копирования вершины и ребра, все их атрибуты, а также множества $X, U, \Gamma X, \Gamma U$ должны существовать в единственном экземпляре в контексте одного графа полностью в памяти GPU. При этом все вершины и ребра должны иметь уникальный идентификатор для их однозначного определения.

7. Реализация параллельного алгоритма операции декомпозиции вершины графа

В качестве примера для оценки эффективности применения CUDA в обработке графов с учетом сформированных рекомендаций рассмотрим параллельный алгоритм операции декомпозиции вершины графа. Входными данными для операции являются:

- идентификатор разбиваемой вершины;
- новые вершины x_r и x_t, а также идентификаторы входящих и исходящих ребер;
- вводимое ребро u_f, идентификаторы входящих и исходящих вершин.

Алгоритм операции, подробно описанный в [7], представляет собой совокупность последовательно выполняющихся параллельных операций: удаление вершины, добавления и удаления вершин и ребер [8].

Предложим реализацию параллельного алгоритма удаления вершины:

```
1   // функция удаляет вершину с идентификатором id
2   void deleteVertex(ID id) {
3         // удаление вершины с идентификатором id из множеств исходящих вершин всех
4         ребер графа, выполняется на EdgesCount потоках – количество ребер графа
5         deleteVertexConnections <EdgesCount> (id);
6
7         // удаление вершины с идентификатором id из множества X графа, освобождение
8         // памяти
9         deleteVertex(id);
10  }
```

```
// функция исполняется параллельно на N CUDA-потоках, в переменной threadID
// содержится уникальный идентификатор потока от 0 до N
void deleteVertexConnections(ID id) {

      // удаление вершины с идентификатором id из ребра графа с номером threadID,
      // если такая вершина была найдена
      deleteVertexIfExist(Edges[threadID], id);
}
```

Функция `deleteVertexConnections` инициализирует количество потоков, равное числу ребер графа. Каждый поток, взаимодействуя с одним ребром, производит поиск по множеству исходящих вершин. Если в множестве была найдена вершина с идентификатором `id`, то поток удаляет эту вершину. После этого функция `deleteVertex` удаляет вершину из множества X.

Реализация параллельного алгоритма добавления вершины:

```
// функция добавляет вершину x в граф, inEdgesId – массив идентификаторов входящих
// ребер, outEdgesId – массив идентификаторов исходящих ребер
void addVertex(Vertex x, ID inEdgesId[], ID outEdgesId[]) {

      // инициализация вершины x, добавление в множество X графа
      initVertex(x);

      // добавление входящих и исходящих ребер, выполняется на inEdgesId.size +
      // + outEdgesId.size потоках
      addEdges <inEdgesId.size + outEdgesId.size> (x, inEdgesId, outEdgesId);
}

// функция исполняется параллельно на N CUDA-потоках, в переменной threadID
// содержится уникальный идентификатор потока от 0 до N
void addEdges(Vertex x, ID inEdgesId[], ID outEdgesId[]) {

      // если id потока меньше, чем размерность массива outEdgesId
      if (threadID < outEdgesId.size) {

          // связать вершину x с идентификатором id и исходящее ребро с
          // идентификатором outEdgesId[threadID]
          addConnectedEdge(x.id, outEdgesId[threadID]);
      }
      else {

          // связать входящее ребро с идентификатором inEdgesId[threadID –
          // - outEdgesId.size] с вершиной x с идентификатором id
          addConnectedVertex(inEdgesId[threadID - outEdgesId.size], x.id);
      }
}
```

Функция `addEdges` инициализирует количество потоков, количество которых определяется суммой числа входящих и исходящих ребер добавляемой вершины. Каждый поток в зависимости от его номера связывает входящее или исходящее ребро с вершиной. Параллельный алгоритм добавления ребра аналогичен приведенному.

Для получения практических результатов – зависимости ускорения алгоритма от количества входных данных – для операции декомпозиции вершины проведем измерение ускорения на случайном ориентированном гиперграфе с 50000 вершинами и 50000 ребрами при средней степени связности вершины или ребра равной 10 для 3 вариантов реализаций:

- без использования критической секции в потоках с итеративным выполнением параллельной функции;
- без использования критической секции в потоках с итеративным выполнением кода внутри параллельной функции;
- с использованием критической секции в потоках с итеративным выполнением кода внутри параллельной функции.

Экспериментальные результаты приведены на рисунке 4. На графике по оси K – ускорение, по x – сумма входящих и исходящих ребер для новых вершин x_r и x_t и входящих и исходящих вершин для вводимого ребра u_f.

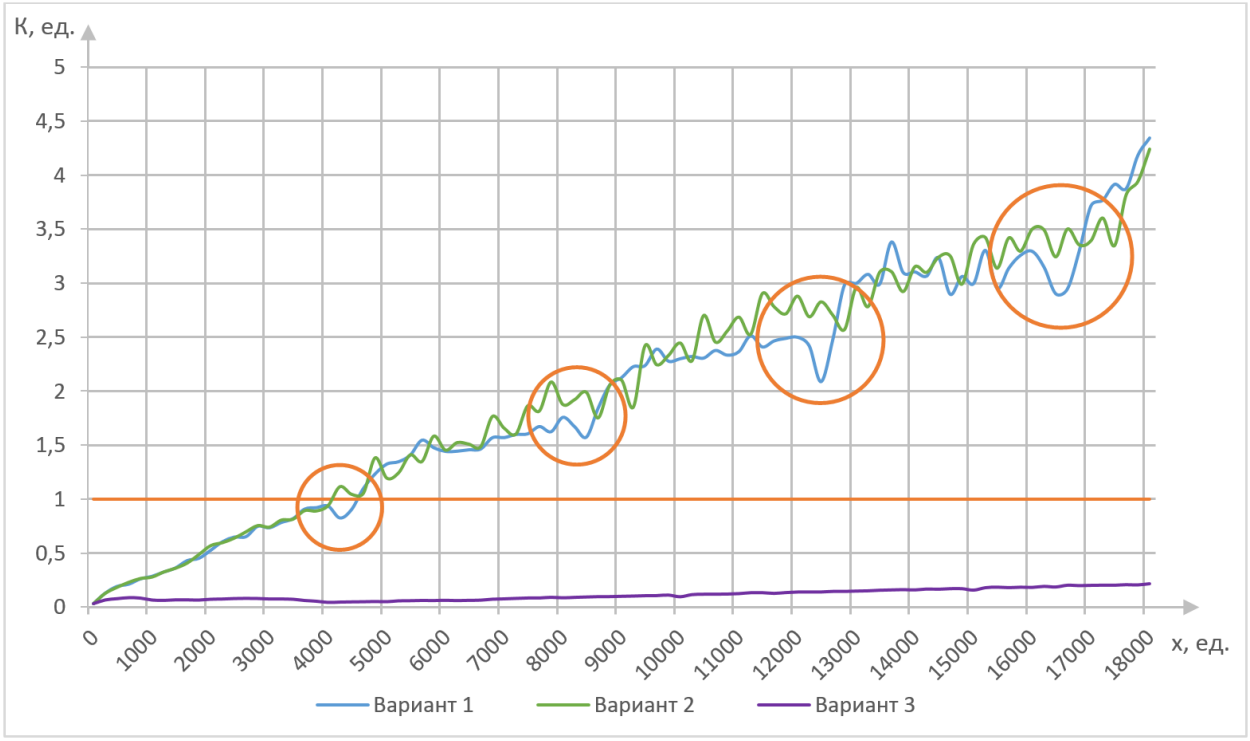

Рис. 4. Экспериментальные результаты

За счет значительной дивергенции потоков вариант 3 имеет наименьшее ускорение. Варианты 1 и 2 практически идентичны, однако вторая реализация более эффективна, так как в первом варианте, в силу затрат на инициализацию потоков на границах окон выполнения CUDA (см. рис. 1), присутствуют провалы, выделенные оранжевым на рисунке. Полученное ускорение для реализаций 1 и 2 больше единицы при сумме входящих и исходящих ребер более 4200.

Заключение

Полученные результаты позволяют говорить о применимости выработанных рекомендаций и высокой эффективности использования графических процессоров CUDA при реализации операций над графами в задачах больших размерностей. Предложенные методы организации параллельных вычислений дадут возможность разрабатывать эффективные параллельные алгоритмы и структуры данных при решении задач анализа и синтеза структур сложных систем.

Список литературы

1. Овчинников В.А. Графы в задачах анализа и синтеза структур сложных систем. М.: МГТУ им. Н.Э. Баумана, 2014. 423 с.
2. Овчинников В.А. Алгоритмизация комбинаторно-оптимизационных задач при проектировании ЭВМ и систем. М.: МГТУ им. Н.Э. Баумана, 2001. 288 с.
3. Иванова Г.С. Методология и средства разработки алгоритмов решения задач анализа и синтеза структур программного обеспечения и устройств вычислительной техники: дис. … докт. техн. наук. М., 2007. 416 с.
4. Овчинников В.А., Иванова Г.С., Ничушкина Т.Н. Выбор структур данных для представления графов при решении комбинаторно-оптимизационных задач // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2001. № 2 (43). С. 39-51.
5. Овчинников В.А. Операции над ультра и гиперграфами для реализации процедур анализа и синтеза структур сложных систем // Наука и образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2009. № 10. Режим доступа: http://technomag.bmstu.ru/doc/132769.html (дата обращения 07.10.2015).
6. Овчинников В.А. Операции над ультра и гиперграфами для реализации процедур анализа и синтеза структур сложных систем (часть 2) // Наука и образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2009. № 11. Режим доступа: http://technomag.bmstu.ru/doc/133223.html (дата обращения 07.10.2015).
7. Овчинников В.А. Операции над ультра и гиперграфами для реализации процедур анализа и синтеза структур сложных систем (часть 3) // Наука и образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2009. № 12. Режим доступа: http://technomag.bmstu.ru/doc/134335.html (дата обращения 07.10.2015).
8. Иванова Г.С., Головков А.А. Оценка эффективности параллельных алгоритмов операций преобразования графовой модели // Наука и образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2014. № 11. С. 535-554. DOI: 10.7463/1114.0741563
9. Головков А.А., Иванова Г.С. Структура данных для представления графов на параллельных вычислительных системах и параллельные алгоритмы операций над графами // Инженерный вестник. МГТУ им. Н.Э. Баумана. Электрон. журн. 2014. №

11. С. 625-632. Режим доступа: http://engbul.bmstu.ru/doc/751611.html (дата обращения 07.10.2015).
10. Головков А.А. Представление графовых моделей в системах параллельной обработки // Молодежный научно-технический вестник. МГТУ им. Н.Э. Баумана. Электрон. журн. 2014. № 7. Режим доступа: http://sntbul.bmstu.ru/doc/727773.html (дата обращения 07.10.2015).
11. Гергель В.П. Теория и практика параллельных вычислений: учеб. пособие. М.: Интернет-университет Информационных технологий; БИНОМ. Лаборатория знаний, 2013. 423 с.
12. Wilt N. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley, 2013. 512 p.
13. Farber R. CUDA Application Design and Development. Waltham, MA, USA: Morgan Kaufmann, 2011. 336 p.
14. Merrill D., Garland M., Grimshaw A. Scalable GPU Graph Traversal // Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12). ACM New York, NY, USA, 2012. P. 117-128. DOI: 10.1145/2145816.2145832
15. Hussein M., Varshney A., Davis L. On Implementing Graph Cuts on CUDA // First Workshop on General Purpose Processing on Graphics Processing Units, 2007. Available at: http://www-lb.cs.umd.edu/gvil/papers/hussein_GPGPU07.pdf, accessed 07.10.2015.
16. Dechev D., Pirkelbauer P., Stroustrup B. Lock-Free Dynamically Resizable Arrays // Principles of Distributed Systems. Springer-Verlag Berlin Heidelberg, 2006. P. 142-156. (Ser. Lecture Notes in Computer Science; vol. 4305). DOI: 10.1007/11945529_11
17. Luo L., Wong M., Wen-mei Hwu. An Effective GPU Implementation of Breadth-First Search // Proceedings of the 47th Design Automation Conference (DAC '10). ACM New York, NY, USA, 2010. P. 52-55. DOI: 10.1145/1837274.1837289
18. Misra P., Chaudhuri M. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs // Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems. IEEE Publ., 2012. P. 53-60. DOI: 10.1109/ICPADS.2012.18
19. Cederman D., Gidenstam A., Ha P., Sundell H., Papatriantafilou M., Tsigas P. Lock-free Concurrent Data Structures // Programming Multi-Core and Many-Core Computing Systems / ed. by S. Pllana, F. Xhafa. Wiley-Blackwell, 2013. (Wiley Series on Parallel and Distributed). Available at: http://arxiv.org/pdf/1302.2757.pdf, accessed 07.10.2015.
20. Schaeffer S.E. Graph clustering // Computer Science Review. 2007. Vol. 1, iss. 1. P. 27-64. DOI: 10.1016/j.cosrev.2007.05.001
21. Shucai Xiao, Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization // 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE Publ., 2010. P. 1-12. DOI: 10.1109/IPDPS.2010.5470477

22. Якобовский М.В. Введение в параллельные методы решения задач: учеб. пособие. М.: Изд-во Московского ун-та, 2013. 328 с.
23. Линев А.В., Богопелов Д.К., Бастраков С.И. Технологии параллельного программирования для процессоров новых архитектур: учебник. М.: Изд-во Московского ун-та, 2010. 160 с.
24. Pawan Harish, Vibhav Vineet and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report no. IIIT/TR/2009/74. Centre for Visual Information Technology, International Institute of Information Technology, Hyderabad - 500 032, INDIA, February 2009. Available at: http://cvit.iiit.ac.in/papers/pawan09GraphAlgorithms.pdf, accessed 07.10.2015.

Science & Education of the Bauman MSTU
Electronic journal
ISSN 1994-0408

Science and Education of the Bauman MSTU, 2015, no. 10, pp. 271–287.

DOI: **10.7463/1015.0820521**

Received: 10.09.2015
Revised: 26.09.2015

Optimization of Computations on Related Sets in CUDA

G.S. Ivanova[1,*], A.A. Golovkov[1]

*gsivanova@gmail.com

[1]Bauman Moscow State Technical University, Moscow, Russia

Keywords: operation on graph, parallel algorithm, CUDA, optimization, thread, mutex, data structure

Many algorithms for solving problems of analysis and synthesis of complex system structures have a large margin of internal parallelism. However, in their implementation in a specific parallel computing system an acceleration factor can be ultra low. First of all, such behaviour is because of hardware and software features of the computer system. This article focuses on identifying the factors that leave less room for achieving the calculated coefficients of acceleration of parallel algorithms for operations on the graph models when using GPU CUDA, and in the development of recommendations, which, if to follow them, result in downtime.

This article considers a model program implementation in CUDA, defines the nature of the flow implementation, and offers 2 methods of planning the flows: the iterative execution of parallel function and iterative code execution in a parallel function. In order to develop effective data structures was carried out analysis of the features of memory in CUDA. We studied different memory allocation algorithms and developed CUDA-implemented critical section, which does not cause a deadlock of flows. Based on the results recommendations have been formulated for the effective use of the parallelization potential in CUDA.

Within the framework of practical test of the proposed methods and rules, has been developed a data structure to represent graphs, which is based on the CUDA principles, as well as have been offered several options to implement the parallel algorithm of the vertex decomposition operation: without using a critical section in the flows of iterative execution of a parallel function; without using a critical section in the flow of iterative execution of code within the parallel functions; using a critical section in the flows of iterative execution of code within the parallel function. Based on the analysis of experimental results it was concluded that the developed recommendations turned to be useful and using the CUDA when implementing operations on the graphs in large-scale problems was highly efficient.

References

1. Ovchinnikov V.A. *Grafy v zadachakh analiza i sinteza struktur slozhnykh system* [Graphs in problems of analysis and synthesis of complex system structures]. Moscow, Bauman MSTU Publ., 2014. 423 p. (in Russian).
2. Ovchinnikov V.A. *Algoritmizatsiya kombinatorno-optimizatsionnykh zadach pri proektirovanii EVM i system* [Algorithmization of combinatorial optimization problems in computers and systems design]. Moscow, Bauman MSTU Publ., 2001. 288 p. (in Russian).
3. Ivanova G.S. *Metodologiya i sredstva razrabotki algoritmov resheniya zadach analiza i sinteza struktur programmnogo obespecheniya i ustroistv vychislitel'noi tekhniki. Doct. diss.* [The methodology and tools for algorithms development for solving problems of software and hardware structures analysis and synthesis. Dr. diss.]. Moscow, 2007. 416 p. (in Russian).
4. Ovchinnikov V.A., Ivanova G.S., Nichushkina T.N. Selection of Data Structures for Graph Representation while Solving Combinatorial and Optimizational Problems. *Vestnik MGTU im. N.E. Baumana. Ser. Priborostroenie = Herald of the Bauman Moscow State Technical University. Ser. Instrument Engineering*, 2001, no. 2, pp. 39-51. (in Russian).
5. Ovchinnikov V.A. Operations over ultra- and hypercolumns for realisation of procedures of the analysis and synthesis of structures of difficult systems. *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2009, no. 10. Available at: http://technomag.bmstu.ru/doc/132769.html , accessed 07.10.2015. (in Russian).
6. Ovchinnikov V.A. Operations over ultra- and hypercolumns for realisation of procedures of the analysis and synthesis of structures of difficult systems (Part 2). *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2009, no. 11. Available at: http://technomag.bmstu.ru/doc/133223.html , accessed 07.10.2015. (in Russian).
7. Ovchinnikov V.A. Operations over ultra- and hypercolumns for realisation of procedures of the analysis and synthesis of structures of difficult systems (Part 3). *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2009, no. 12. Available at: http://technomag.bmstu.ru/doc/134335.html , accessed 07.10.2015. (in Russian).
8. Ivanova G.S., Golovkov A.A. Evaluating Efficiency of Parallel Algorithms of Transformation Operations with Graph Model. *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2014, no. 11, pp. 535-554. DOI: 10.7463/1114.0741563 (in Russian).
9. Golovkov A.A., Ivanova G.S. Data structure for graph representation on parallel computing systems and parallel algorithms of operations on graphs. *Inzhenernyi vestnik MGTU im. N.E. Baumana = Engineering Herald of the Bauman MSTU*, 2014, no. 11, pp. 625-632. Available at: http://engbul.bmstu.ru/doc/751611.html , accessed 07.10.2015. (in Russian).

10. Golovkov A.A. Graph Models Representation on Parallel Computing Systems. *Molodezhnyi nauchno-tekhnicheskii vestnik MGTU im. N.E. Baumana = Youth Science and Technology Herald of the Bauman MSTU*, 2014, no. 7. Available at: http://sntbul.bmstu.ru/doc/727773.html , accessed 07.10.2015. (in Russian).
11. Gergel' V.P. *Teoriya i praktika parallel'nykh vychislenii* [Theory and practice of parallel computing]. Moscow, BINOM Publ., 2013. 423 p. (in Russian).
12. Wilt N. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013. 512 p.
13. Farber R. *CUDA Application Design and Development*. Waltham, MA, USA, Morgan Kaufmann, 2011. 336 p.
14. Merrill D., Garland M., Grimshaw A. Scalable GPU Graph Traversal. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM New York, NY, USA, 2012, pp. 117-128. DOI: 10.1145/2145816.2145832
15. Hussein M., Varshney A., Davis L. On Implementing Graph Cuts on CUDA. *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007. Available at: http://www-lb.cs.umd.edu/gvil/papers/hussein_GPGPU07.pdf, accessed 07.10.2015.
16. Dechev D., Pirkelbauer P., Stroustrup B. Lock-Free Dynamically Resizable Arrays. In: *Principles of Distributed Systems*. Springer-Verlag Berlin Heidelberg, 2006, pp. 142-156. (Ser. *Lecture Notes in Computer Science*; vol. 4305). DOI: 10.1007/11945529_11
17. Luo L., Wong M., Wen-mei Hwu. An Effective GPU Implementation of Breadth-First Search. *Proceedings of the 47th Design Automation Conference (DAC '10)*. ACM New York, NY, USA, 2010, pp. 52-55. DOI: 10.1145/1837274.1837289
18. Misra P., Chaudhuri M. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems*. IEEE Publ., 2012, pp. 53-60. DOI: 10.1109/ICPADS.2012.18
19. Cederman D., Gidenstam A., Ha P., Sundell H., Papatriantafilou M., Tsigas P. Lock-free Concurrent Data Structures. In: Pllana S., Xhafa F., eds. *Programming Multi-Core and Many-Core Computing Systems*. Wiley-Blackwell, 2013. (*Wiley Series on Parallel and Distributed*). Available at: http://arxiv.org/pdf/1302.2757.pdf, accessed 07.10.2015.
20. Schaeffer S.E. Graph clustering. *Computer Science Review*, 2007, vol. 1, iss. 1, pp. 27-64. DOI: 10.1016/j.cosrev.2007.05.001
21. Shucai Xiao, Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE Publ., 2010, pp. 1-12. DOI: 10.1109/IPDPS.2010.5470477
22. Yakobovskii M.V. *Vvedenie v parallel'nye metody resheniya zadach* [Introduction to the Parallel Methods of Problem Solving]. Moscow, MSU Publ., 2013. 328 p. (in Russian).

23. Linev A.V., Bogopelov D.K., Bastrakov S.I. *Tekhnologii parallel'nogo programmirovaniya dlya protsessorov novykh arkhitektur* [Parallel programming for new processor architectures]. Moscow, MSU Publ., 2010. 160 p. (in Russian).
24. Pawan Harish, Vibhav Vineet and P. J. Narayanan. *Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report no. IIIT/TR/2009/74*. Centre for Visual Information Technology, International Institute of Information Technology, Hyderabad - 500 032, INDIA, February 2009. Available at: http://cvit.iiit.ac.in/papers/pawan09GraphAlgorithms.pdf, accessed 07.10.2015.