БАЛАНСИРОВКА НАГРУЗКИ В ГПУ-РЕАЛИЗАЦИИ ПОИСКА В ШИРИНУ НА ГРАФЕ

М.А. Черноскутов

1. Введение. Интересной особенностью алгоритма поиска в ширину является тот факт, что его последовательная версия из-за отсутствия лишних накладных расходов часто может опережать по производительности свои параллельные аналоги. К накладным расходам могут быть отнесены затраты на синхронизацию данных между итерациями, потребность в массивном параллелизме, а также нерегулярность алгоритма. При этом использование таких аппаратных особенностей ГПУ, как широкий канал доступа в DRAM-память и эффективный механизм управления большим количеством потоков, может значительно ускорить выполнение параллельного алгоритма поиска в ширину по сравнению с ЦПУ. Однако если граф имеет неравномерное распределение степеней вершин, то при этом между потоками возникает значительный дисбаланс нагрузки, из-за которого может заметно увеличиться время выполнения каждой итерации алгоритма, что в итоге сведет на нет все преимущества, предоставляемые архитектурой ГПУ. Данная работа посвящена описанию метода балансировки вычислительной нагрузки для ГПУ-реализации параллельного алгоритма поиска в ширину на графе. Использование данного метода позволяет достичь ускорения в пять и более раз по сравнению с последовательной реализацией стандартного алгоритма на ЦПУ.

2. Параллельные алгоритмы. Распараллеливание алгоритмов поиска в ширину строится по следующей схеме: на итерации номер N параллельно обрабатываются те вершины, которые находятся на расстоянии N ребер от корневой. При этом сами итерации выполняются последовательно. Алгоритмы такого типа называются синхронизированными по уровням.

В данный момент имеется два основных типа синхронизированных по уровням алгоритмов:

- на основе использования очередей;
- на основе полного обхода всех вершин на каждой итерации.

Псевдокод алгоритма, основанного на использовании очередей, показан на рис.1.

Входные данные: множество вершин V, очередь для текущего уровня C, очередь для следующего уровня N, корневая вершина s

Выходные данные: массив расстояний dist, содержащий значения дистанций от корневой до всех остальных вершин

Функции: push(val,Q), вставляющая val в конец очереди Q

```
1           for all u in dist
2                 dist[u] == -1
3           dist[s] = 0
4           push(s,C)
5           while C ≠ ø
6                 parallel for all i in C
7                       if dist[i] == level
8                             for all k є neighbors of i
9                                   if dist[k] == -1
10                                        dist[k] = level + 1
11                                        push(k,N)
12                for level++
13                C = N
```

Рис.1 Алгоритм 1

Алгоритм 1 имеет линейную сложность O(V+E), где V и E – количество вершин и ребер, соответственно. При обработке очереди можно назначить каждому потоку одну или несколько вершин и провести их обработку в параллельном режиме. Данный алгоритм плохо подходит для распараллеливания на ГПУ из-за накладных расходов, возникающих при работе с очередью. Фактически, обновление состояний начала и конца очереди (таких как выталкивание вершин из очереди в строке 6 и добавление вершин в очередь в строке 11) должно выполняться с помощью атомарных операций, что на практике приводит к превращению параллельного алгоритма в последовательный.

Псевдокод алгоритмов, основанных на полном обходе всех вершин на каждой итерации, показан на рис.2.

Входные данные: множество вершин V, корневая вершина s

Выходные данные: массив расстояний dist, содержащий значения дистанций от корневой до всех остальных вершин

Функции: check_end(), возвращающая 1 если текущая итерация была

последней и 0 в других случаях

```
1          for all u in dist
2                dist[u] == -1
3          dist[s] = 0
4          level = 0
5          do
6                parallel for i in V
7                      if dist[i] == level
8                            for all k ϵ neighbors of i
9                                  if dist[k] == -1
10                                       dist[k] = level + 1
11               level++
12         while(!check_end())
```

Рис.2 Алгоритм 2

Алгоритм 2 имеет квадратичную сложность $O(V^2+E)$. Каждому вычислительному элементу назначается фиксированный диапазон вершин, которые необходимо обработать (строка 6). На первый взгляд, тот факт, что в данном алгоритме на каждой итерации необходимо выполнять много лишней работы, может служить доводом в пользу непригодности алгоритма. Однако такая схема распараллеливания хорошо подходит для архитектуры ГПУ из-за отсутствия атомарных операций и накладных расходов на обработку очереди. Это подтверждают результаты измерений, показанные на рис.3.

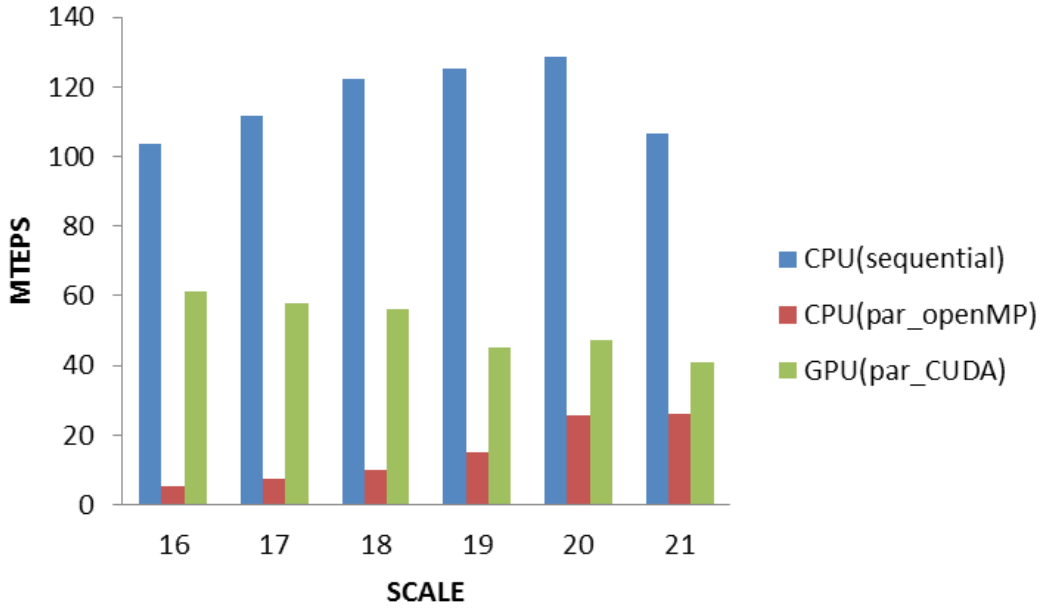

Рис.3. Сравнение скоростей обхода графа

По оси абсцисс отложен размер обрабатываемого графа (двоичный логарифм от числа вершин в графе), а по оси ординат — скорость обработки графа, измеряемая в количестве пройденных в секунду ребер (TEPS: Traversed Edges Per Second). Сравнивались три различных реализации алгоритма: стандартная последовательная на основе очередей (более подробно о данной версии алгоритма можно прочитать в [1]), реализация алгоритма 2 на ЦПУ (Intel Xeon X5675) с помощью технологии OpenMP, а также реализация алгоритма 2 на ГПУ (Nvidia Tesla C2075) с помощью технологии CUDA. Здесь и далее будем ссылаться на данные реализации как на “sequential”, “par_openMP” и “par_CUDA” соответственно. Из рис.3 видно, что последовательная версия обгоняет все параллельные реализации. Как будет видно далее, это происходит из-за дисбаланса нагрузки на каждой итерации алгоритма. При этом реализация “par_openMP” заметно уступает “par_CUDA” в большинстве экспериментов, что позволяет сделать вывод о необходимости дальнейшей оптимизации ГПУ-версии данного алгоритма с целью увеличения скорости обработки графов.

С другими исследованиями поведения графовых алгоритмов на ГПУ можно ознакомиться в работах [2–5]. Однако детальное сравнение производительности между реализациями, представленными в [2–5] и реализациями, представленными в данной работе затруднено ввиду использования иного аппаратного обеспечения, арифметики, а также выходных данных используемых алгоритмов.

3. Используемые графы. Для тестирования используются безмасштабные (“scale-free”) графы. Для таких графов характерно неравномерное распределение степеней по вершинам. Например, в графе может быть

несколько вершин, имеющих большие степени и множество вершин, имеющих маленькие степени. Данные графы являются моделью реальных графов, полученных в результате решения прикладных или фундаментальных задач.

В работе использованы графы из [6], созданные с помощью генератора графов Kronecker [7], который используется в тесте Graph500 [8]. Основными параметрами генератора являются:

- Scale – двоичный логарифм от числа вершин в графе;
- Edgefactor – отношение количества ребер к количеству вершин в графе.

В табл.1 приведены данные обо всех используемых в данной работе графах:

Таблица 1

Используемые графы			
Scale	Edgefactor	Количество вершин	Количество ребер
16	48	65536	6289992
17	48	131072	12581183
18	48	262144	25163641
19	48	524288	50328927
20	48	1048576	100659854
21	48	2097152	201322399

Все графы представлены во внутренней памяти системы в формате Compressed Sparse Row (CSR).

Одной из характерных особенностей алгоритма обхода в ширину является наличие пика нагрузки (количества обрабатываемых вершин) в одной из средних итераций и практически полное отсутствие нагрузки в начале и в конце выполнения алгоритма (см. рис.4). На рисунке видно, что практически вся работа по обходу графа выполняется на пятой, шестой и седьмой итерациях (распределение приведено для графа с параметром Scale, равным 20).

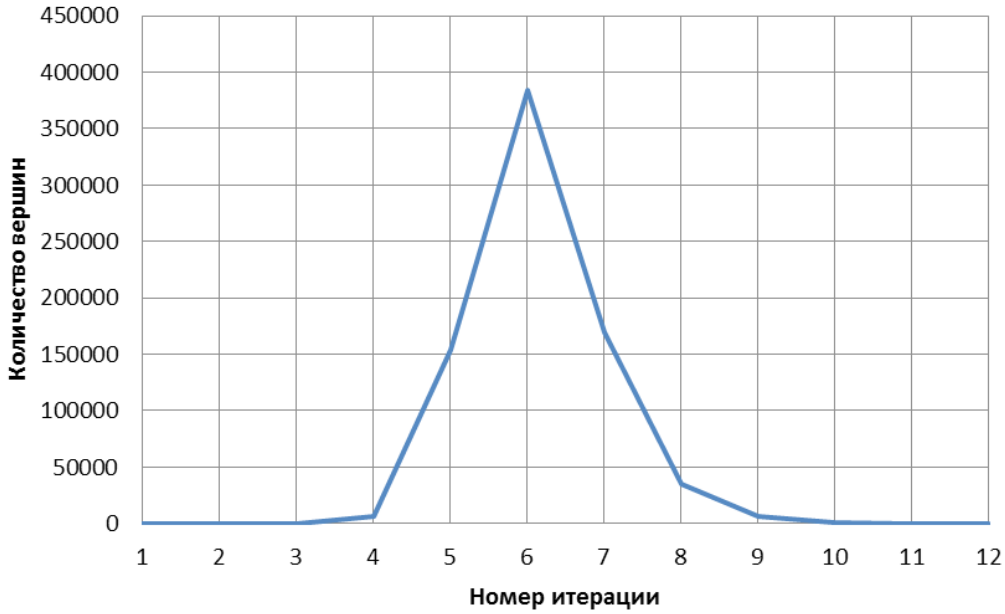

Рис.4. Распределение вычислительной нагрузки по итерациям

4. Описание метода балансировки нагрузки. Как упоминалось выше, при использовании параллельных синхронизированных по уровням алгоритмов, обработка каждого следующего уровня начинается только после того, как закончилась обработка предыдущего. При этом для безмасштабных графов характерна ситуация, когда все вершины имеют разные степени. Таким образом, если на текущем уровне встретится хотя бы одна вершина с большим количеством инцидентных ей ребер, то время работы всей итерации алгоритма будет определяться временем обработки самой “сложной” вершины. Кроме того, постоянно возникают различные накладные расходы, такие как доступ в память и управление большим количеством потоков. Все это в сумме, как видно на рис.3, значительно замедляет работу параллельного алгоритма.

Таким образом, чтобы снизить время выполнения каждой из итераций алгоритма, необходимо избавиться от дисбаланса нагрузки, вносимого неравномерным распределением степеней в графе. С этой целью предложен метод, основанный на ограничении количества ребер, которые может обработать один поток на каждой итерации алгоритма. При этом общее количество потоков должно быть увеличено. Данный метод основан на том, что архитектура ГПУ более приспособлена для работы с большим числом слабонагруженных

потоков, а не с малым числом потоков, каждому из которых приходится выполнять большой объем вычислений. Псевдокод алгоритма обхода графа, реализующий данный метод, представлен на рис.5.

Входные данные: множество вершин V, множество стартовых вершин SV, корневая вершина s, параметр max_edge_count

Выходные данные: массив расстояний dist, содержащий значения дистанций от корневой до всех остальных вершин

Функции: check_end(), возвращающая 1 если текущая итерация была последней и 0 в других случаях

```
1          for all u in dist
2                dist[u] = -1
3          dist[s] = 0
4          level = 0
5          do
6                parallel for i in SV
7                      first_edge = i*max_edge_count
8                      last_edge = (i+1)*max_edge_count
9                      curr_vert = SV[i]
10                           for edge є [first_edge;last_edge)
11                                 if neighbors of curr_vert є
                                   [first_edge;last_edge)
12                                       if dist[curr_vert] == level
13                                             for all k є neighbors of
                                               curr_vert
14                                                   if dist[k] == -1
15                                                   dist[k] = level + 1
16                                 curr_vert++
17               level++
18         while(!check_end())
```

Рис.5. Алгоритм 3

Данный алгоритм делит все множество ребер на равные доли, размер каждой из которых определяется параметром max_edge_count (строки 7 и 8). Затем каждый поток начинает обработку только тех вершин, ребра которых попадают в выделенный в строке 10 интервал. При этом, если количество ребер у вершины выходит за границу данного интервала, то текущим потоком они уже не обрабатываются. В массиве SV хранятся номера вершин, с которых необходимо начинать обход графа в каждом потоке. Количество элементов в массиве SV определяется частным от деления общего количества ребер на max_edge_count (если возникает ненулевой остаток от деления, то к значению неполного частного добавляется единица). В случае безмасштабных графов некоторые вершины (с большим количеством инцидентных ребер) могут встретиться в массиве по несколько раз, а некоторые могут не встретиться вообще (обычно это те вершины, чьи степени меньше значения константы max_edge_count). Алгоритм заполнения массива SV приведен на рис.6.

Входные данные: множество вершин V, корневая вершина s, параметр max_edge_count

Выходные данные: множество стартовых вершин SV

Функции: round_up(res), округляющая res до ближайшего целого сверху

```
1          parallel for i in V
2                first = V[i]
3                last = V[i+1]
4                index = round_up(first/max_edge_count)
5                current = index*max_edge_count
6                while(current < last)
7                      SV[index] = i
8                      current += max_edge_count
9                      index++
```

Рис.6. Алгоритм 4

В большинстве случаев для безмасштабных графов количество элементов в массиве SV превышает количество элементов в массиве V. Поэтому некоторые потоки могут заполнять сразу несколько ячеек массива SV (см. цикл в строках 6 – 9).

Таким образом, в отличие от алгоритма на рис.2, где один поток обрабатывает все инцидентные вершине ребра независимо от их количества, в алгоритмах на рис. 5, 6:

- обработка “сложных” вершин распределяется между несколькими потоками;
- один поток может обработать несколько “простых” вершин (если сумма их степеней не превышает max_edge_count).

Таким образом, значительно снижаются накладные расходы на выполнение каждой итерации алгоритма.

5. Результаты. На рис.7 показаны результаты реализации вышеупомянутого метода (здесь и далее будем ссылаться на его реализацию как “par_bal_CUDA”) балансировки нагрузки. Тестирование проводилось на системе с ЦПУ Intel Xeon X5675 (6 ядер) и ГПУ Nvidia Tesla C2075 (448 ядер, ECC отключено). В качестве сравнения приведены показатели скорости обхода графа для реализаций “sequential” и “par_CUDA” (они аналогичны тем, что приведены на рис.3), а также для ЦПУ реализации метода балансировки нагрузки (“par_bal_openMP”).

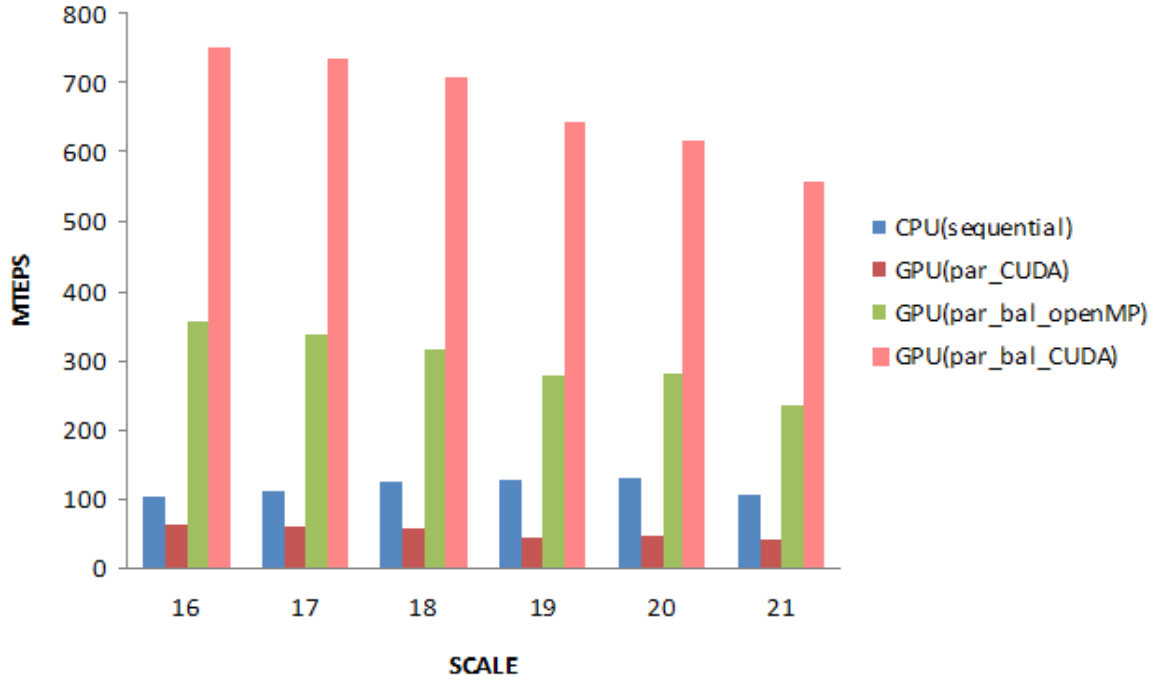

Рис.7. Результаты измерений скорости обхода графов в ширину

Как видно, реализация данного метода на ГПУ позволяет ускорить обход графов в ширину в два и более раз по сравнению с аналогичной реализацией на ЦПУ, в пять и более раз по сравнению с последовательным алгоритмом на ЦПУ, в десять и более раз для неоптимизированной реализации на ГПУ. На рис.7 результаты приведены для значения параметра max_edge_count, равного 16. Данное значение позволяет добиться наилучших показателей производительности для графов, использующихся в тестировании, благодаря балансу между затратами времени на конструирование массива SV и выполнение основного цикла в алгоритме 3. На

рис. 8 показана зависимость скорости обхода одного из графов (Scale = 20) от значения параметра max_edge_count.

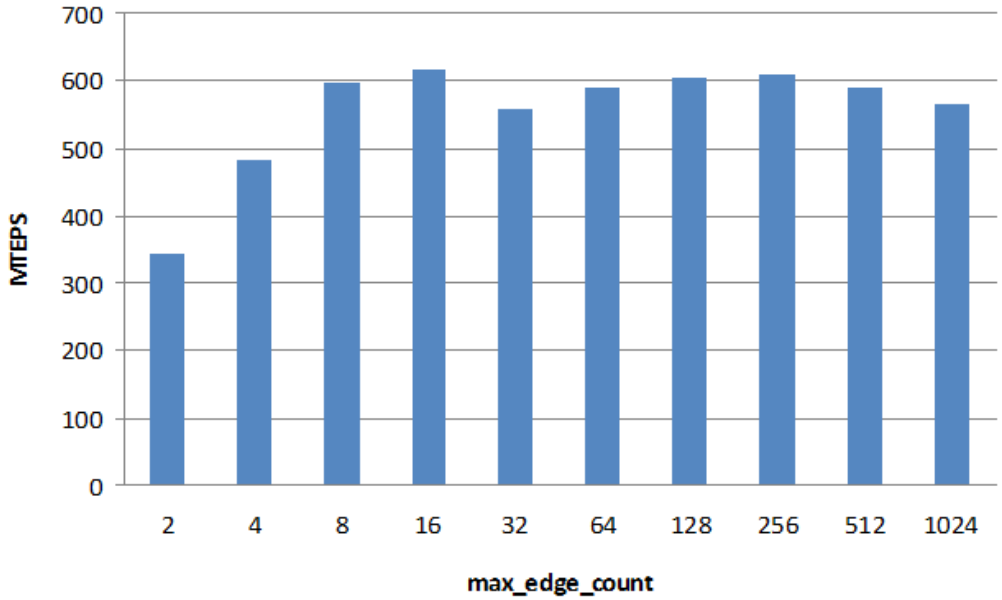

Рис.8. Результаты измерений скорости обхода графа в ширину для разных значений параметра max_edge_count

На рис.9 показаны замеры времени выполнения каждой итерации алгоритма (Scale = 20). Результаты приведены для реализаций “par_CUDA” и “par_bal_CUDA”.

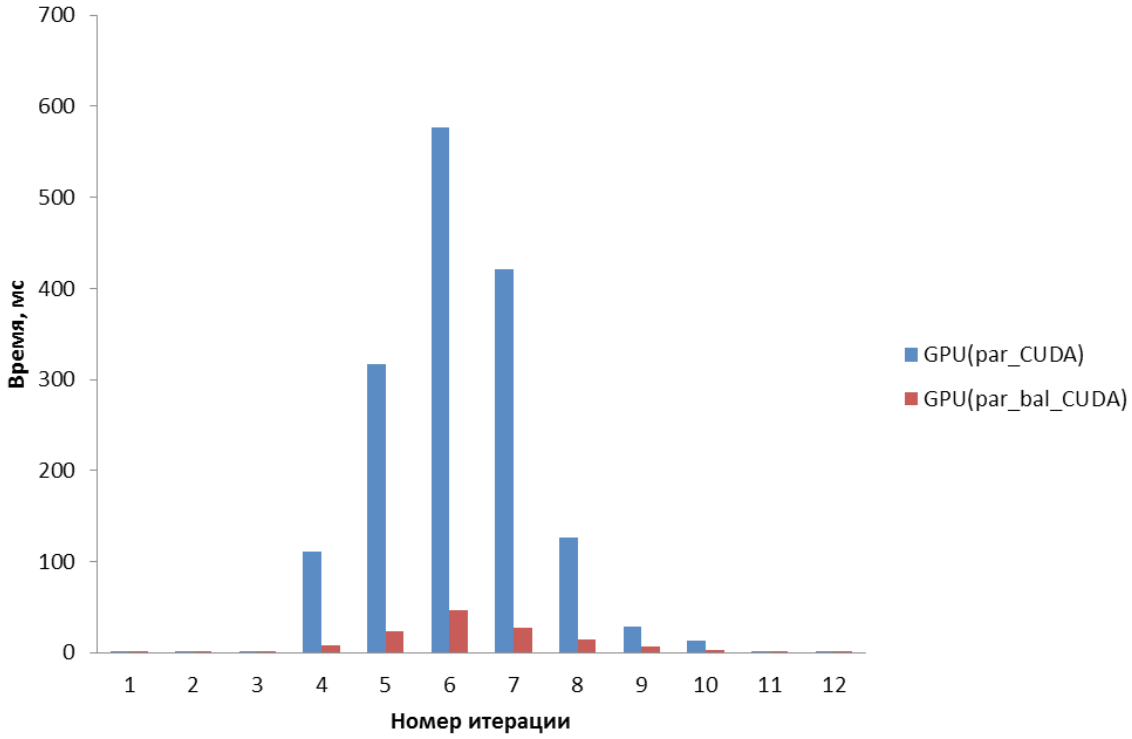

Рис.9. Время выполнения итераций алгоритма обхода графа в ширину

Как видно из рисунка, использование метода балансировки нагрузки помогает значительно снизить накладные расходы при выполнении наиболее загруженных итераций (таких как 4 – 8). Но, если обратить внимание на итерации 1 – 3, 11, 12, то можно заметить, что реализация “par_bal_CUDA” уступает по производительности реализации “par_CUDA”. Данный негативный эффект связан прежде всего с тем, что в реализации “par_bal_CUDA” приходится работать с большим числом потоков, каждый из которых исполняет множество условных операций, что негативно отражается на скорости исполнения CUDA-приложений. Однако, по сравнению со временем, затраченным на исполнение других итераций, данный негативный эффект довольно мал.

Несмотря на то, что реализация “par_bal_CUDA” является самой высокопроизводительной из всех рассмотренных в данной статье, в ней тоже присутствуют накладные расходы, такие как:

- затраты на создание и заполнение массива SV;
- затраты на определение финальной итерации алгоритма обхода в ширину (функция check_end()).

Сначала рассмотрим затраты на создание и заполнение массива SV. На рис.10 приведена доля затрат от общего времени выполнения обхода графа (Scale = 20) в ширину, в зависимости от размера графа.

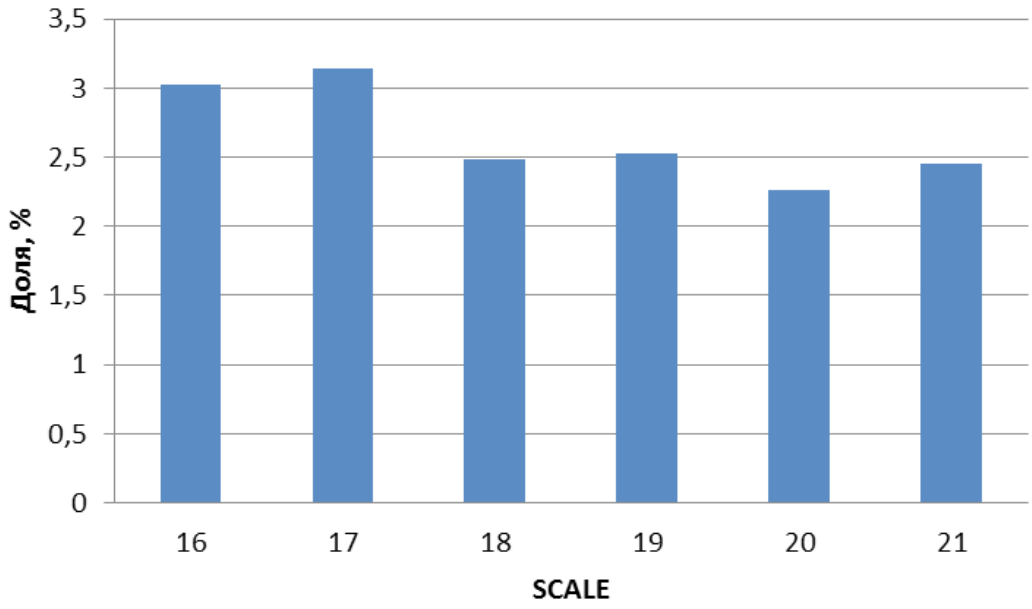

Рис.10 Затраты на создание и заполнение массива SV

Как мы можем наблюдать из рисунка, данные затраты держатся примерно на одном и том же уровне для графов различной величины.

Еще одна разновидность накладных расходов связана с выполнением функции check_end(). После каждой итерации алгоритма поиска в ширину функция должна определить, закончен ли обход графа. Выполняется это путем параллельного перебора всех вершин. Функция завершает свою работу, как только встречает вершину, которая должна быть обработана на следующей итерации. При отсутствии таких вершин алгоритм поиска в ширину завершает свою работу. На рис.11 представлена доля затрат на выполнение функции check_end() на каждой итерации алгоритма (Scale = 20).

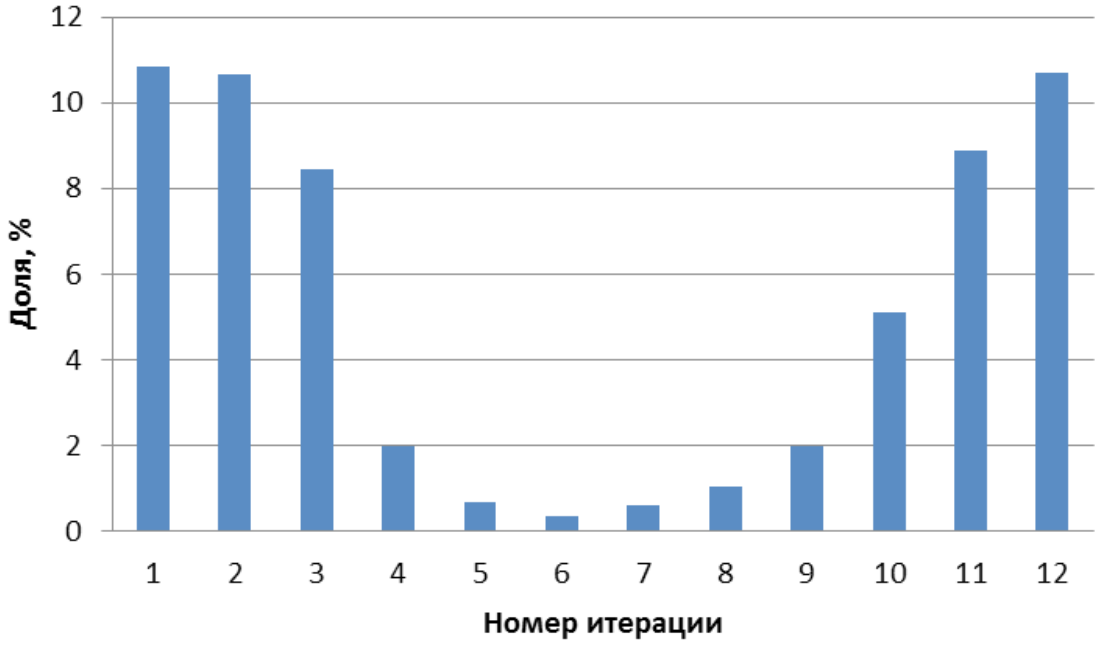

Рис.11. Затраты на выполнение функции определения заключительной итерации

В целом, затраты на выполнение функции check_end() не превышает 11%, что является приемлемым показателем для графов с небольшим диаметром (таких как безмасштабные). Снижение доли затрат в 4 – 9 итерациях объясняется тем, что функция имеет сложность O(1), вследствие чего время ее выполнения примерно одинаковое на всех итерациях. С другой стороны, время, затрачиваемое на выполнение алгоритма обхода графа в ширину, на итерациях 4 – 9 значительно возрастает (см. рис.9), что и приводит к снижению доли функции check_end().

6. Выводы. В работе представлен метод балансировки нагрузки, позволяющий значительно снизить накладные расходы, возникающие в стандартном синхронизированном по уровням параллельном алгоритме поиска в ширину на графе. Реализация данного метода на ГПУ позволяет достичь ускорения в два и более раз по сравнению с аналогичной реализацией на ЦПУ, а также в пять и более раз по сравнению с последовательной версией алгоритма для ЦПУ.

На основании результатов, полученных в данной работе, выбраны направления дальнейших исследований:

- совершенствование разработанной реализации алгоритма поиска в ширину:
  - снижение накладных расходов на заполнение массива SV и выполнение функции check_end();
  - уменьшение количества ветвлений в алгоритме на рис.5;
- реализация оптимизированного алгоритма поиска в ширину на мультиГПУ системах;
- анализ применимости разработанного метода балансировки нагрузки для других алгоритмов на графах.

Работа поддержана грантами РФФИ-13-01-12021, УрО РАН 12-П-1-1029, РЦП-13-П18. При проведении работ был использован суперкомпьютер “Уран” ИММ УрО РАН.

ЛИТЕРАТУРА:

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C.Stein “Introduction to Algorithms”, Second. Cambridge, MA: MIT Press, 2001
2. D. Merril, M. Garland “High performance and scalable graph traversal”, Technical report CS-2011-05, Nvidia, 2011
3. L. Luo, M. Wong, W.-mei Hwu “An effective GPU implementation of breadth-first search”, in Proceedings of the 47th Design Automation Conference, New York, NY, USA, 2010, pp. 52–55
4. P. Harish, P.J. Narayanan “Accelerating large graph algorithms on the GPU using CUDA”, in Proceedings of the 14th international conference on High performance computing, Berlin, Heidelberg, 2007, pp. 197–208
5. S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun “Accelerating CUDA graph algorithms at maximum warp”, in Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, New York, NY, USA, 2011, pp. 267–276
6. 10th DIMACS Implementation Challenge. [Электронный ресурс]. URL: http://www.cc.gatech.edu/dimacs10/index.shtml. [Дата обращения: 01.04.2013]
7. J. Leskovec, D. Chakrabarti, J.M. Kleinberg, C. Faloutsos “Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication”, in PKDD, 2005, pp. 133–145
8. R. Murphy, K. Wheeler, B. Barrett, J. Ang “Introducing the Graph 500”, Cray User's Group (CUG), May 5, 2010