УСКОРЕНИЕ ПОИСКА В ШИРИНУ НА GPU-АРХИТЕКТУРЕ С ПОМОЩЬЮ РАСПРЕДЕЛЕНИЯ НАГРУЗКИ

М.А. Черноскутов[1,2]

[1] *Институт математики и механики УрО РАН*
[2] *Уральский Федеральный Университет*

Введение. Поиск в ширину на графе является одним из наиболее известных алгоритмов на графах. Его приложения встречаются в различных областях науки, промышленности и бизнеса. Например, в криптографии, биоинформатике, анализе социальных сетей, медицине и проч.

Алгоритм поиска в ширину на графе обладает большим потенциалом к распараллеливанию: теоретически, каждому потоку можно назначить обработку одной вершины (или диапазона вершин), что позволяет сделать предположение об эффективности программно-аппаратной модели графических ускорителей вычислений для обработки крупных графов, т. к. она позволяет работать с большим числом потоков.

Подходы к распараллеливанию поиска в ширину на графе. Чаще всего, для распараллеливания алгоритма поиска в ширину на графе применяются алгоритмы, параллельно обрабатывающие вершины, лежащие на одинаковом расстоянии от корневой вершины (такие алгоритмы называются синхронизированными по уровням). Разница между ними заключается лишь в механизме выбора вершин для обработки. На данный момент распространение получили два типа таких алгоритмов: на основе параллельной обработки очереди вершин для текущей итерации алгоритма, а также на основе полного обхода всех вершин в графе, независимо от номера итерации. Первый подход, несмотря на линейную сложность, обладает одним существенным недостатком: работа с очередью требует использования атомарных операций добавления и удаления вершин в очередь, что на практике может быть источником большого количества накладных расходов, значительно снижающих эффективность использования параллельной архитектуры. Второй подход, хоть и обладает квадратичной сложностью, больше подходит для распараллеливания алгоритма поиска в ширину на графе, т. к. позволяет большому количеству параллельных потоков работать независимо, почти исключая использование атомарных операций для данных, локализованных в одних и тех же или близких участках оперативной памяти. Реализации и модификации обоих подходов рассмотрены в работах [1-4].

В работе [1] описывается реализация простейшего алгоритма на основе обхода вершин. Данная реализация хорошо зарекомендовала себя при обработке графов, у которых вершины имеют примерно одинаковые степени, но заметно уступала при работе с графами, имеющими неравномерное распределение степеней вершин.

В работе [2] содержится адаптированная для GPU версия алгоритма на основе очередей. Однако авторы указывают, что их алгоритм показывает хорошие результаты только для графов, содержащих небольшие степени у всех вершин.

В работе [3] содержится оптимизированная версия алгоритма на основе полного обхода вершин, содержащая многие оптимизационные техники, а также позволяющая более аккуратно распределять обработку вершин по «warp'ам».

В работе [4] демонстрируются самые высокие показатели скорости обхода графов на одном GPU на сегодняшний день. Авторы используют оригинальные идеи устранения обработки повторяющихся ребер при выполнении текущей итерации и балансировки нагрузки между потоками.

Метод балансировки нагрузки. Помимо механизма распараллеливания, при реализации параллельного алгоритма поиска в ширину на графе существует еще одна существенная проблема – дисбаланс вычислительной нагрузки по потокам. Чаще всего такая проблема встречается при обработке графов с неравномерным распределением степеней вершин (в таких графах, как правило, имеется небольшое число вершин с большим числом входящих и исходящих ребер и много вершин с малым числом ребер). Проблема еще более усугубляется при реализации параллельного алгоритма для архитектуры графических ускорителей вычислений. Причиной этому служит то, что АЛУ в GPU устроены проще, чем АЛУ в CPU и работают на более низкой частоте. Это приводит к тому, что отдельный вычислительный поток в GPU менее производителен, чем поток в CPU. В результате, обработка отдельных «сложных» вершин затягивает выполнение всей итерации алгоритма. А если такие вершины будут встречаться на каждой итерации алгоритма, то это приведет к значительному замедлению работы всего алгоритма.

Входные данные: массив 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 ← current + max_edge_count
9              index++
```

Рис. 1. Алгоритм заполнения массива SV

Для устранения этого недостатка предлагается метод балансировки нагрузки. Суть метода заключается в разделении массива ребер (для обработки графов наиболее часто используется формат CSR, состоящий из массива вершин и массива ребер) на равные участки и просмотре каждого такого участка отдельным потоком. Отличие от стандартного подхода к распараллеливанию заключается в том, что в нем предлагается распределение по параллельным потокам элементов массива вершин. При этом каждый поток обязан рассмотреть все входящие и исходящие из вершины ребра. В предлагаемом подходе ребра, принадлежащие одной вершине могут быть обработаны несколькими потоками и, наоборот, ребра от нескольких вершин могут быть обработаны одним потоком.

Входные данные: массив вершин `V`, массив стартовых вершин `SV`, корневая вершина `s`, параметр `max_edge_count`
Выходные данные: массив расстояний `dist`, содержащий значения дистанций от корневой до всех остальных вершин, массив `pred`, содержащий номера предшествующих вершин
Функции: `check_end()`, возвращающая `1` если текущая итерация была последней и `0` в других случаях

```
1   parallel for 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 ϵ [first_edge;last_edge)
11                    if neighbors of curr_vert in [first_edge;last_edge)
12                          if dist[curr_vert] = level
13                                for all k in neighbors of curr_vert
14                                      if dist[k] = -1
15                                            dist[k] ← level + 1
16                                            pred[k] ← curr_vert
17                    curr_vert++
18        level++
19  while(!check_end())
```

Рис. 2. Алгоритм поиска в ширину с балансировкой нагрузки (top-down подход)

Для реализации метода необходимо по номеру элемента в массиве ребер определить номер вершины, которой он принадлежит. На рис. 1 показано как это можно сделать путем введения дополнительного массива SV (start vertices), содержащего номер вершины, которой принадлежит начальный элемент из участка массива ребер. Параметр max_edge_count описывает размер участка из массива ребер.

Таким образом, после работы алгоритма на рис. 1 весь массив ребер оказывается разбит на равные участки, которые могут быть обработаны каждым потоком независимо. Затем, опираясь на алгоритм, представленный на рис. 2 можно выполнять поиск в ширину на графе (стандартная top-down реализация). Опираясь на работу [5], в которой описано сочетание подходов top-down и bottom-up при выполнении параллельного алгоритма поиска в ширину на графе были разработан алгоритм, представленный на рис. 3, реализующий bottom-up подход к реализации метода балансировки нагрузки.

Входные данные: массив вершин `V`, массив стартовых вершин `SV`, корневая вершина `s`, параметр `max_edge_count`
Выходные данные: массив расстояний `dist`, содержащий значения дистанций от корневой до всех остальных вершин, массив `pred`, содержащий номера предшествующих вершин
Функции: `check_end()`, возвращающая `1` если текущая итерация была последней и `0` в других случаях

```
1  parallel for 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 ϵ [first_edge;last_edge)
11                   if neighbors of curr_vert in [first_edge;last_edge)
12                         if dist[curr_vert] = -1
13                               for all k in neighbors of curr_vert
14                                     if dist[k] = level
15                                           dist[curr_vert] ← level + 1
16                                           pred[curr_vert] ← k
17                                           break
18                   curr_vert++
19       level++
20 while(!check_end())
```

Рис. 3. Алгоритм поиска в ширину с балансировкой нагрузки (bottom-up подход)

Результаты. Тестирование проводилось с использованием GPU NVidia Tesla M2050 на «scale-free» графах, имеющих неравномерное распределение степеней вершин. Графы такого типа используются в тесте производительности вычислительных систем Graph500. Описание используемых графов приведено в табл. (количество вершин в графе: 2^{Scale}; количество ребер в графе: $2^{Scale+1}$ *Edgefactor).

Таблица 1. Используемые графы

Scale	Количество вершин, шт.	Количество ребер, шт.		
		Edgefactor		
		16	32	48
19	524 288	16 777 216	33 554 432	50 331 648
20	1 048 576	33 554 432	67 108 864	100 663 296
21	2 097 152	67 108 864	134 217 728	201 326 592

На рис. 4 приведено сравнение времени выполнения каждой итерации алгоритма балансировки нагрузки для подходов top-down и bottom-up (обрабатывался граф с параметром Scale = 20).

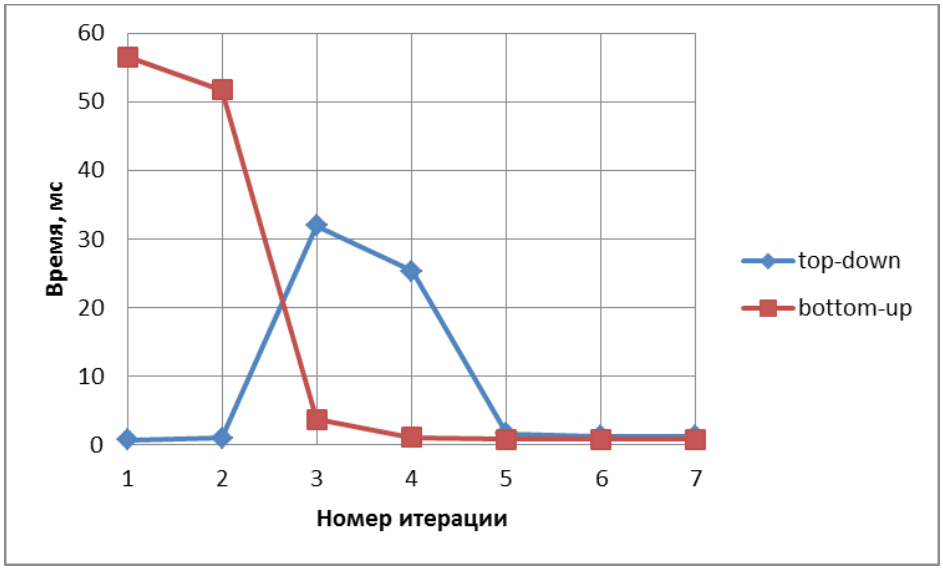

Рис. 4. Сравнение времени выполнения каждой итерации в подходах top-down и bottom-up

На рис. 4 видно, что в первых двух итерациях bottom-up подход заметно уступает традиционному top-down подходу, однако затем ситуация меняется на противоположную. Гибридная реализация алгоритма, реализующего метод балансировки нагрузки показывает наилучшую производительность, если первые две итерации выполняются по схеме top-down, а остальные – по схеме bottom-up. Замеры производительности гибридной реализации приведены на рис. 5 (результаты усреднены по 64 запускам). Метрикой производительности служит количество пройденных в секунду ребер (TEPS – Traversed Edge Per Second). Данная метрика применяется также и в тесте производительности Graph500.

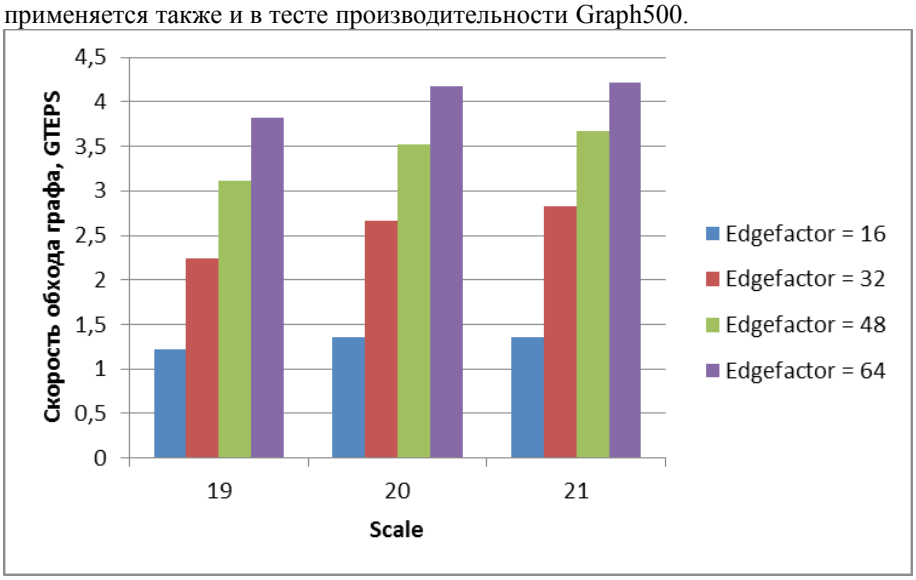

Рис. 5. Результаты измерений скорости поиска в ширину на графе

Сравнение результатов с мировым уровнем. Наибольшая скорость поиска в ширину на графе для GPU-реализаций была представлена в работе [4]. Для графа с параметром Scale = 20 и Edgefactor = 48 авторы получили скорость обхода в 2.5 GTEPS (заполнение массива pred, в котором хранятся номера предшествующих вершин) и 3.1 GTEPS (заполнение массива dist, в котором хранятся расстояния до корневой вершины). Реализация, описанная в данной работе, позволяет достичь скорости 3.5 GTEPS. Причем в ней производится заполнение как массива pred, так и массива dist.

Заключение. Стандартные параллельные алгоритмы поиска в ширину на графе плохо подходят для реализации на программно-аппаратной платформе графических ускорителей вычислений из-за дисбаланса нагрузки между вычислительными потоками (при обработке графов с неравномерным распределением степеней). В данной работе описан метод балансировки нагрузки, позволяющий повысить эффективность GPU-реализации поиска в ширину на графе и достичь скорости поиска в ширину более 4 GTEPS.

В качестве дальнейшего направления исследований предполагается адаптация метода балансировки нагрузки к другим алгоритмам на графах.

Благодарности. Работа поддержана грантами РФФИ 14-07-00435, УрО РАН РЦП-14-П14. При проведении работ был использован суперкомпьютер «Уран» ИММ УрО РАН.

ЛИТЕРАТУРА:

1. 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.
2. 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.
3. S. Hong, S.K. Kim, T. Oguntebi et al. "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.
4. D. Merril, M. Garland, A. Grimshaw "High performance and scalable graph traversal" // Technical report CS-2011-05, Nvidia, 2011, pp. 1–15.
5. S. Beamer, K. Asanovic, D. Patterson "Direction-optimizing breadth-first search" // in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, Utah, USA, 2012, pp. 12–22.