Параллельная реализация алгоритма поиска минимальных остовных деревьев с использованием центрального и графического процессоров

А.С. Колганов[1,2]

МГУ им. М.В. Ломоносова[1], ИПМ им. М.В. Келдыша РАН[2]

Решение задачи поиска минимальных остовных деревьев является распространенной в различных областях исследований: распознавание различных объектов, компьютерное зрение, анализ и построение сетей (например, телефонных, электрических, компьютерных, дорожных и т.д.), химия и биология и многие другие. Существует, по крайней мере, три известных алгоритма, решающих данную задачу: Борувки, Крускала и Прима. Обработка больших графов – достаточно трудоемкая задача для центрального процессора (CPU) и является востребованной в данное время. Все более широкое распространение для решения задач общего назначения получают графические ускорители (GPU), имеющие большую вычислительную мощность, чем CPU. Но данная задача, как и многие задачи по обработке графов, плохо ложится на архитектуру GPU. В данной статье будет рассмотрена гибридная реализация данного алгоритма.

Ключевые слова: MST, CUDA, NVidia, Graphs, Boruvka

1. Введение

В данной статье рассматривается задача построения минимального остовного дерева (MST – minimum spanning tree) для неориентированного взвешенного графа. Остовное дерево – такое дерево, которое является максимальным по включению ребер подграфом, не имеющее циклов, и в котором сумма весов ребер – минимальна. Если исходный граф связный, то будет построено остовное дерево, если же в исходном графе несколько несвязных компонент, то результатом будет остовный лес.

Минимальные остовные деревья имеют широкое практическое применение. Они используются при построении различных сетей, например, коммуникационных, компьютерных или транспортных. Также они используются в кластеризации, сегментации при обработке изображений, распознавании рукописного ввода. Вычисление остовных деревьев на графах является нерегулярным алгоритмом обработки данных. Наименьшая вычислительная сложность последовательного алгоритма MST была получена автором данной статьи [1] и равна $O(E\alpha(E,V))$, где E – количество ребер графа, V – количество вершин, α – функция Аккермана [2].

Алгоритм Борувки дает оценку $O(Elog(V))$. Также существуют некоторые параллельные реализации данного алгоритма: со сложностью $O(log(V))$ и временной сложностью $O(Vlog(V))$, на симметричном мультипроцессоре со сложностью $O((V+E)/p)$, где p – количество ядер в процессоре, на графическом процессоре [3–5]. Но все рассмотренные результаты были получены на достаточно старых и разных графических процессорах, что не позволяет корректно сравнивать такие реализации между собой.

В данной работе описывается гибридный параллельный алгоритм Борувки, использующий и графический и центральный процессоры. В разделах 2 и 3 приведено описание обрабатываемых графов. В разделе 4 описаны алгоритмы преобразования входных графов. В разделе 5 дано описание параллельной реализации алгоритма Борувки на графическом процессоре. Разделы 6 и 7 описывают принципы построения гибридной реализации алгоритма и результаты полученной производительности.

2. Описание формата представления графов

Рассмотрим структуру хранения неориентированного взвешенного графа, так как в дальнейшем она будет часто упоминаться и преобразовываться. Граф задается в сжатом CSR (Compressed Sparse Row) [6,7] формате. Данный формат получил широкое распространение для хранения разреженных матриц и графов. Для взвешенного неориентированного графа с N вершинами и M ребрами необходимо три массива: X (массив указателей на смежные вершины), A (массив списка смежных вершин) и W (массив весов ребер, соответствующие списку смежных вершин). Массив X размера N + 1, остальные два – 2 * M, так как в неориентированном графе для любой пары вершин необходимо хранить прямую и обратную дуги. В массиве X хранятся начало и конец списка соседей, находящиеся в массиве A, то есть весь список соседей вершины J находится в массиве A с индекса X[J] до X[J+1], не включая его. По аналогичным индексам хранятся веса каждого ребра из вершины J. Для иллюстрации на рис. 1 слева показан граф из 4 вершин, записанный с помощью матрицы смежности, а справа – в формате CSR (для упрощения, вес каждого ребра считается одинаковым и не указан на рисунке).

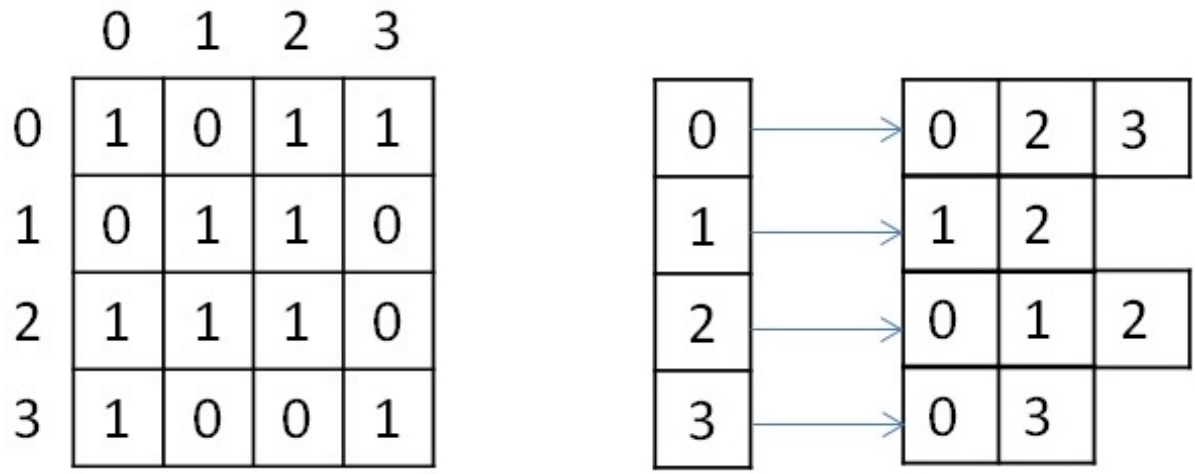

Рис. 1. Представление графа в разных форматах

3. Тестируемые графы

В данном разделе описываются графы, на которых происходило тестирование реализованного алгоритма. Для описания алгоритмов преобразования графа и параллельного алгоритма MST требуется представление структуры рассматриваемых графов. Для оценки производительности реализации используются два вида синтетических графов: R-MAT-графы (A Recursive Model for Graph Mining) [8] и SSCA2-графы (Scalable Synthetic Compact Applications) [9].

R-MAT-графы хорошо моделируют реальные графы из социальных сетей, Интернета. В данном случае рассматриваются R-MAT-графы со средней степенью связности вершины 32, а количество вершин является степенью двойки. В таком R-MAT графе имеется одна большая связная компонента и некоторое количество небольших связных компонент или висящих вершин.

SSCA2-граф представляет собой большой набор независимых компонент, соединенных ребрами друг с другом. SSCA2-граф генерируется таким образом, чтобы средняя степень связности вершины была близка к 32, количество вершин в данном графе также является степенью двойки. Таким образом, рассматриваются два совершенно разных по структуре графа с одинаковым количеством вершин.

4. Преобразование входных данных

Так как тестирование алгоритма будет производиться на графах R-MAT и SSCA2, которые получаются с помощью генератора, то для улучшения производительности реализованного алгоритма необходимо проделать некоторые преобразования. Все преобразования не будут учитываться в подсчете производительности.

4.1. Локальная сортировка списка вершин

Для каждой вершины выполним сортировку ее списка соседей по весу в порядке возрастания. Это позволит частично упростить выбор минимального ребра на каждой итерации алгоритма. Данная сортировка является локальной, следовательно, она не дает полное решение задачи.

4.2. Перенумерация всех вершин графа

Занумеруем вершины графа таким образом, чтобы наиболее связные вершины имели наиболее близкие номера. В результате данной операции в каждой связной компоненте разница между максимальным и минимальным номером вершины будет наименьшей, что позволит лучшим образом использовать маленький кэш графического процесса. Стоит отметить, что для R-MAT-графов данная перенумерация не дает существенного эффекта, потому что в данном графе присутствует очень большая компонента, которая не помещается в кэш даже после применения данной оптимизации. Для SSCA2-графов эффект от данного преобразования заметен больше, так как в данном графе большое количество компонент, содержащих относительно небольшое количество вершин.

4.3. Отображение весов графа в целые числа

В данной задаче не требуется производить каких-либо операций над весами графа. Необходимо уметь сравнивать веса двух ребер. Для этих целей можно использовать целые числа, вместо чисел двойной точности, так как скорость обработки чисел одинарной точности на GPU намного выше, чем двойной. Данное преобразование можно выполнить для графов, у которых количество уникальных ребер не превосходит 2^{32} (максимальное количество различных чисел, помещающихся в unsigned int языка Си). Если средняя степень связности каждой вершины равна 32, то самый большой граф, который можно обработать с применением данного преобразования, будет иметь 2^{28} вершин и будет занимать в памяти 64 ГБ. На 2015 год наибольшее количество памяти присутствует в ускорителях NVidia Tesla k40 / NVidia Titan X и AMD FirePro w9100 и составляет 12ГБ и 16ГБ соответственно. Поэтому на одном GPU с применением данного преобразования можно обработать достаточно большие графы.

4.4. Сжатие информации о вершинах

Данное преобразование применимо только к SSCA2-графам из-за их структуры. В данной задаче большую роль играет производительность памятей всех уровней: начиная от глобальной памяти и заканчивая кэшем первого уровня. Для снижения трафика между глобальной памятью и L2-кэшем, можно хранить информацию о вершинах в сжатом виде. Изначально информация о вершинах представлена в виде двух массивов: массива X, в котором хранятся начало и конец списка соседей в массиве A.

Рассмотрим пример на рис. 2. У вершины J есть 10 вершин-соседей, и если номер каждого соседа хранится с использованием типа unsigned int, то для хранения списка соседей вершины J потребуется 10 * sizeof(unsigned int) байт, а для всего графа – 2 * M * sizeof(unsigned int) байт. Будем считать, что sizeof(unsigned int) = 4 байта, sizeof(unsigned

short) = 2 байта, sizeof(unsigned char) = 1 байт. Тогда для данной вершины необходимо 40 байт для хранения списка соседей. Нетрудно заметить, что разница между максимальным

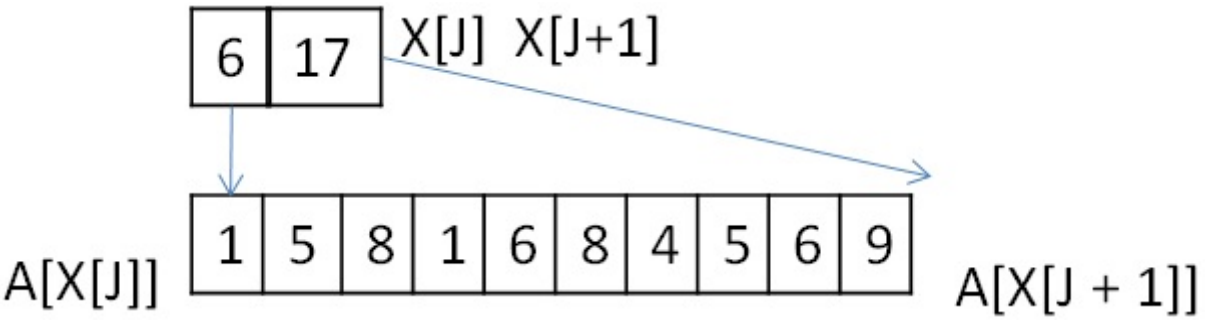

Рис. 2. Смежные 10 вершин

и минимальным номером вершины в этом списке равна 8, причем для хранения данного числа необходимо всего 4 бита. Исходя из тех соображений, что разница между максимальным и минимальным номером вершины может быть меньше, чем unsigned int, можно представить номер каждой вершины следующим образом:

$$base\_J + 256 * k + short\_endV,$$

где <base_J> – например, минимальный номер вершины из всего списка соседей. В данном примере на рис. 2 это будет 1. Данная переменная будет иметь тип unsigned int, и таких переменных будет столько, сколько вершин в графе. Далее посчитаем разницу между номером вершины и выбранной базой. Так как в качестве базы мы выбрали наименьшую вершину, то данная разница будет всегда положительной. Для SSCA2-графа данная разница будет помещаться в unsigned short. <short_endV> – это остаток от деления на 256. Для хранения данной переменной будем использовать тип unsigned char; а <k> – есть целая часть от деления на 256. Для k выделим 2 бита (то есть k лежит в пределах от 0 до 3). Выбранное представление является достаточным для рассматриваемого графа. В битовом представление данное сжатие проиллюстрировано на рис. 3. Тем самым для хранения спис-

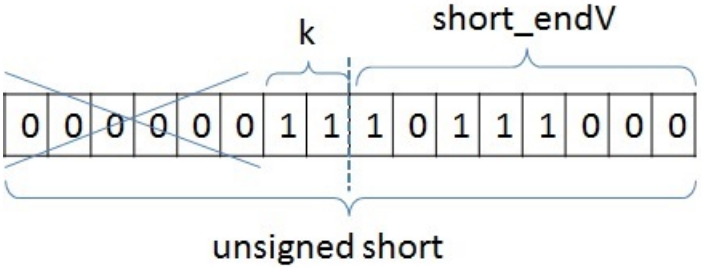

Рис. 3. Сжатия номера одной смежной вершины

ка вершин требуется (1 + 0,25) * 10 + 4 = 16,5 байт для данного примера, вместо 40 байт, а для всего графа: (2 * M + 4 * N + 2 * M / 4) вместо 2 * M * 4. Если N = 2 * M / 32, то общий объем уменьшится в (8 * M) / (2 * M + 8 * M / 32 + 2 * M / 4) = 2.9 раз.

5. Общее описание алгоритма и этапы его реализации

Для реализации алгоритма MST был выбран алгоритма Борувки. Данный алгоритм имеет минимальную временную сложность и большой потенциал для распараллеливания

по сравнению, например, с алгоритмом Прима или Крускала. Базовое описание алгоритма Борувки и иллюстрация его итераций хорошо представлена [10]. Согласно алгоритму, все вершины изначально включены в минимальное дерево – каждая вершина в своем дереве. Далее необходимо выполнить следующие шаги:

1. Найти минимальные ребра между всеми деревьями для их последующего объединения. Если на данном шаге не выбрано ни одно ребро, то ответ задачи получен;
2. Выполнить объединение соответствующих деревьев. Данный шаг разбивается на два этапа: удаление циклов, так как два дерева могут в качестве кандидата на объединение указать друг друга, и этап объединения, когда выбирается номер дерева, в которое входят объединяемые поддеревья. Для определенности будем выбирать минимальный номер такого дерева. Если в ходе объединения осталось лишь одно дерево, то ответ задачи получен;
3. Выполнить перенумерацию полученных деревьев для перехода на первый шаг (чтобы все деревья имели номера от 0 до k).

Схематично реализованный алгоритм показан на рис. 4. Выход из всего алгоритма проис-

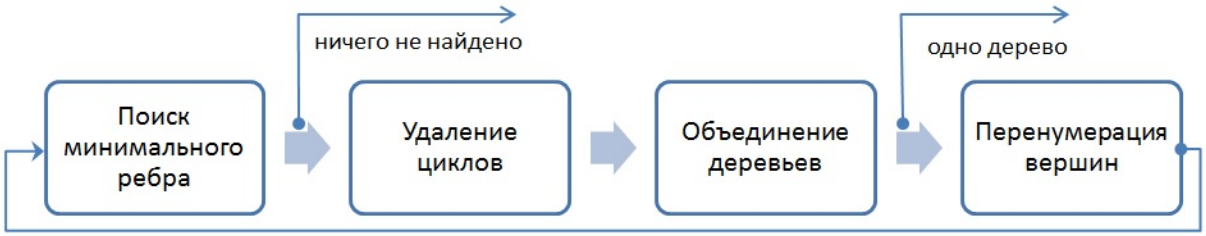

Рис. 4. Схема работы алгоритма Борувки

ходит в двух случаях:

- Если все вершины после N итераций объединены в одно дерево;
- Если невозможно найти минимальное ребро из каждого дерева (в таком случае минимальные остовные деревья найдены).

5.1. Поиск минимального ребра.

Сначала каждая вершина графа помещается в отдельное дерево. Далее происходит итеративный процесс объединения деревьев, состоящий из четырех рассмотренных выше процедур. Процедура поиска минимального ребра позволяет выбрать именно те ребра, которые будут входить в минимальное остовное дерево. Как было описано выше, на входе у данной процедуры преобразованный граф, хранящийся в формате CSR. Так как для списка соседей была выполнена частичная сортировка ребер по весу, то выбор минимальной вершины сводится к просмотру списка соседей и выбора первой вершины, которая принадлежит другому дереву. Если предположить, что в графе нет петель, то на первом шаге алгоритма выбор минимальной вершины сводится к выбору первой вершины из списка соседей для каждой рассматриваемой вершины, потому что список соседних вершин (которые составляют вместе с рассматриваемой вершиной ребра графа), отсортированы по возрастанию веса ребра и каждая вершина входит в отдельное дерево. На любом другом шаге необходимо просмотреть список всех соседних вершин по порядку и выбрать ту вершину, которая принадлежит другому дереву.

Почему же нельзя выбрать вторую вершину из списка соседних вершин и положить данное ребро минимальным? После процедуры объединения деревьев (которая будет рассмотрена далее) может возникнуть ситуация, что некоторые вершины из списка соседних

могут оказаться в том же дереве, что и рассматриваемая, тем самым данное ребро будет являться петлей для данного дерева, а по условию алгоритма необходимо выбирать минимальное ребро до других деревьев.

Для реализации обработки вершин и выполнения процедуры поиска, объединения и слияния списков хорошо подходит структура Union Find [11]. К сожалению, не все структуры оптимально обрабатываются на GPU. Наиболее выгодно в данной задаче (как и в большинстве других) использовать непрерывные массивы в памяти GPU, вместо связных списков. Ниже будут рассмотрены похожие алгоритмы Union-Find для поиска минимального ребра, объединения сегментов, удаления циклов в графе. Рассмотрим алгоритм поиска минимального ребра. Его можно представить в виде двух шагов:

- Выбор исходящего из каждой вершины минимального ребра (которое входит в какой-то сегмент) рассматриваемого графа;
- Выбор ребра минимального веса для каждого дерева.

Для того, чтобы не перемещать информацию о вершинах, записанную в формате CSR, будем использовать два вспомогательных массива, которые будут хранить индекс начала и конца массива A списка соседей. Два данных массива будут обозначать сегменты списков вершин, принадлежащих одному дереву. Например, на первом шаге массив начал или нижних значений будет иметь значения X[0]...X[N], а массив концов или верхних значений будет иметь значения X[1]...X[N+1]. После процедуры объединения деревьев, данные сегменты перемешаются, но массив соседей A не будет перемещен в памяти.

Оба шага могут быть выполнены параллельно. Для выполнения первого шага необходимо просмотреть список соседей каждой вершины (или каждого сегмента) и выбрать первое ребро, принадлежащее другому дереву. Можно выделить один warp – набор из 32х нитей GPU, работающие физически параллельно и синхронно – для просмотра списка соседей каждой вершины. На рис. 5 красным выделены сегменты, принадлежащие дереву 0, а зеленым – дереву 1.

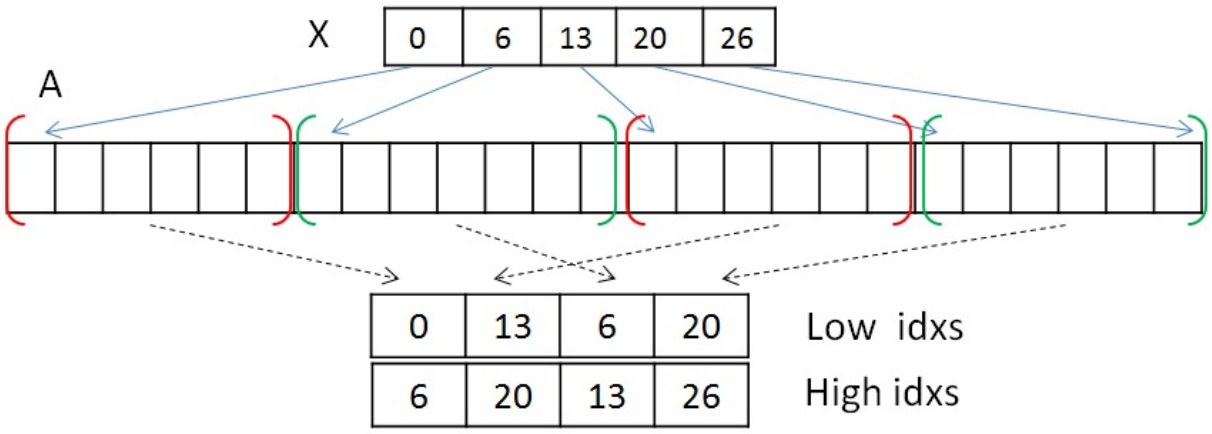

Рис. 5. Расположение сегментов в памяти

В силу того, что каждый сегмент списка соседей отсортирован, то не обязательно просматривать все вершины. Так как один warp состоит из 32 нитей, то просмотр будет осуществляться порциями по 32 вершины. После того, как просмотрены 32 вершины, необходимо объединить результат и если ничего не найдено, то просмотреть следующие 32 вершины. Для объединения результата можно воспользоваться префиксной суммой. В результате данного шага для каждого сегмента будет записана следующая информация: номер вершины в массиве A, входящее в ребро минимального веса и вес самого ребра. Если ничего не найдено, то в номер вершины можно записать, например, число N + 2.

Второй шаг необходим для редуцирования найденной информации для выбора ребра с минимальным весом для каждого из деревьев. Данный шаг делается из-за того, что сегменты, принадлежащие одному и тому же дереву, просматриваются параллельно и независимо, и для каждого из сегментов выбирается ребро минимального веса. В данном шаге один warp может редуцировать информацию по каждому дереву (по нескольким сегментам). После выполнения данного шага будет известно с каким деревом каждое из деревьев соединено минимальным ребром (если оно существует). Для записи данной информации введем еще два вспомогательных массива, в одном из которых будем хранить номера деревьев, до которых есть минимальное ребро, во втором – номер вершины в исходном графе, которая является корнем входящих в дерево вершин. Результат работы данного шага показан на рис. 6.

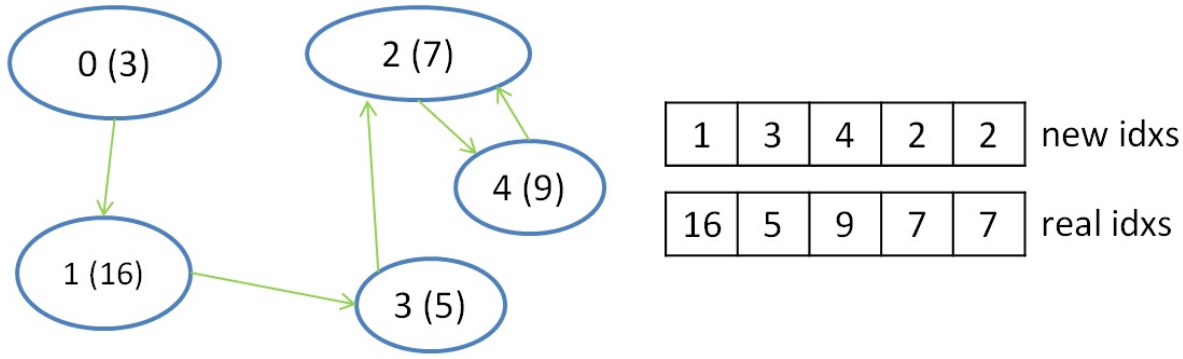

Рис. 6. Результат работы процедуры поиска минимального ребра

Для работы с индексами необходимо еще два массива, которые помогают конвертировать первоначальные индексы в новые индексы и получать по новому индексу первоначальный. Рассмотренные таблицы переконвертации индексов обновляются с каждой итерацией алгоритма. Таблица получения нового индекса по первоначальному индексу имеет размер N – количества вершин в графе, а таблица получения первоначального индекса по новому сокращается с каждой итерацией и имеет размер, равный количеству деревьев на какой-либо выбранной итерации алгоритма (на первой итерации алгоритма эта таблица имеет также размер N элементов).

5.2. Удаление циклов

Данная процедура необходима для удаления циклов между двумя деревьями. Такая ситуация возникает тогда, когда у дерева N1 минимальное ребро до дерева N2, а у дерева N2 минимальное ребро до дерева N1. На рис. 6 присутствует цикл только между двумя деревьями с номерами 2 и 4. Так как деревьев на каждой итерации становится меньше, то будем выбирать минимальный номер из двух деревьев, составляющих цикл. В данном случае, дерево с номером 2 станет указывать на само себя, а дерево с номером 4 продолжит указывать на дерево с номером 2. С помощью проверок, код которых показан на рис. 7, можно определить такой цикл и устранить его в пользу минимального номера.

Данная процедура может быть выполнена параллельно, так как каждая вершина может быть обработана независимо и записи в новый массив вершин без циклов не пересекаются.

5.3. Объединение деревьев

Данная процедура производит объединение деревьев в более крупные деревья. Процедура удаления циклов между двумя деревьями, по сути, является предобработкой перед данной процедурой. Она позволяет избежать зацикливания при объединении деревьев.

```
unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
unsigned local_f = cF[i];
if (cF[local_f] == i)
{
    if (i < local_f)
        F[i] = i;
}
```

Рис. 7. Реализация алгоритма удаления циклов на GPU

Объединение деревьев представляет собой процесс выбора нового корня путем изменения ссылок. Если, допустим, дерево с номером 0 указывало на дерево с номером 1, а в свою очередь дерево с номером 1 указывало на дерево с номером 3, то можно сменить ссылку дерева 0 с дерева 1 на дерево 3. Описанное изменение ссылки стоит производить, если оно не приводит к появлению цикла между двумя деревьями. Применим описанный алгоритм к рис. 6. После процедур удаления циклов и объединения деревьев останется только одно дерево с номером 2. Процесс объединения представлен на рис. 8. Структура графа и прин-

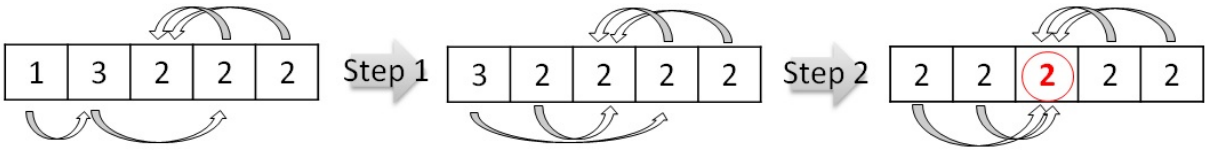

Рис. 8. Итерационный процесс объединения деревьев

цип его обработки таков, что не возникнет ситуации, когда будет происходить зацикливание процедуры и она также может быть выполнена параллельно.

5.4. Перенумерация вершин (деревьев)

После выполнения процедуры объединения деревьев необходимо перенумеровать полученные деревья так, чтобы их номера шли подряд от 0 до P. По построению новые номера должны получить элементы массива, удовлетворяющие условию F[i] == i (для рассмотренного примера на рис. 6, данному условию удовлетворяет только элемент с индексом 2). Тем самым, с помощью атомарных операций можно разметить весь массив новыми значениями от 1 ... (P+1). Далее выполнить заполнение таблиц получения нового индекса по первоначальному и первоначального индекса по новому. Пример такой перенумерации представлен на рис. 9.

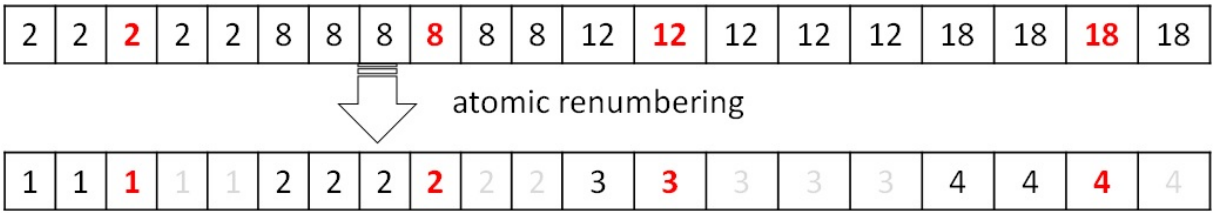

Рис. 9. Процесс перенумерации деревьев

Работа с данными таблицами описана в процедуре поиска минимального ребра. Следующая итерация не может корректно выполняться без обновления данных таблиц. Все описанные операции выполняются параллельно и на GPU.

6. Гибридная реализация процедуры поиска минимального ребра.

Описанный алгоритм в разделе 5, показывает хорошие ускорения на одном GPU. Решение задачи MST организовано таким образом, что можно распределить нагрузку процедуры поиска минимального ребра между центральными процессором (CPU) и GPU. Данное распределение можно сделать только на общей памяти. Для этого был использован стандарт OpenMP [12] для параллельного счета на CPU и передача данных между CPU и GPU по шине PCIe с помощью технологии CUDA [13]. Выполнение всех процедур на одной итерации на линии времени при использовании одного GPU представлено на рис. 10. Изначально все

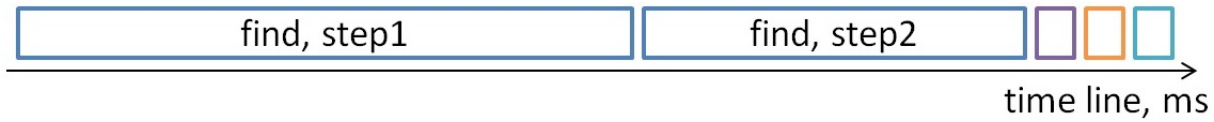

Рис. 10. Время выполнения процедур на одной итерации алгоритма на GPU

данные о графе хранятся как на CPU, так и на GPU. Для того, чтобы CPU мог считать, необходимо передать информацию о перемещенных во время объединения деревьев сегментах. Также для того, чтобы GPU продолжил итерацию алгоритма, необходимо вернуть подсчитанные на CPU данные. Логичным было бы использование асинхронного копирования между центральным процессором и ускорителем (см рис. 11). Алгоритм на CPU повто-

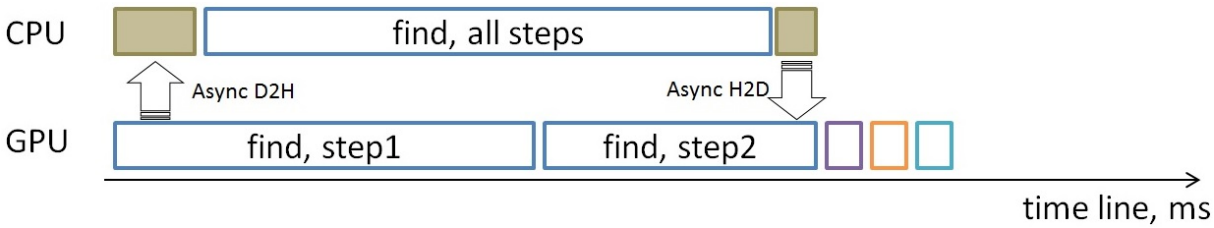

Рис. 11. Время выполнения процедур на одной итерации алгоритма на GPU и CPU

ряет алгоритм, используемый на GPU, только для распараллеливания цикла используется OpenMP. Как и стоило ожидать, CPU вычисляет медленнее, чем GPU. Чтобы CPU успевал посчитать свою часть, данные между CPU и GPU надо делить в отношении 1 : 5. Остальные процедуры невыгодно считать на обоих устройствах, так как они занимают очень мало времени, а накладные расходы и медленная скорость CPU только увеличат время алгоритма. Также очень важна скорость копирования между CPU и GPU. На тестируемой платформе поддерживался PCIe 3.0, который позволял достигать 15 ГБ/с. В итоге, использование CPU позволило получить около 15% прироста производительности.

На сегодняшний день количество оперативной памяти на GPU и CPU существенно отличается в пользу последнего. На тестовой платформе на GPU доступно 6ГБ GDDR5, в то время как на CPU доступно 40ГБ. Ограничения по памяти на GPU не позволяют обсчитывать большие графы.

С версии CUDA Toolkit 6.0 доступна для использования технология Unified Memory [14], которая позволяет обращаться с GPU в память CPU. Так как информация о графе необходима только в процедуре поиска минимального ребра, то для больших графов можно сделать следующее: сначала поместить все вспомогательные массивы в памяти GPU, а далее расположить часть массивов графа (массив соседей A, массив X и массив весов W) в памяти GPU, а то что не уместилось – в памяти CPU. Далее, во время счета, можно делить данные так, чтобы на CPU, по возможности, обрабатывалась та часть, которая не поместилась на GPU, а GPU минимально использовал доступ в память CPU (так как доступ в

память CPU с графического ускорителя осуществляется через шину PCIe на скорости не более 15 ГБ/с).

Используя такой подход можно обработать графы, которые изначально не помещаются на GPU даже при использовании описанных алгоритмов сжатия, но с меньшей скоростью, так как пропускная способность PCIe значительно меньше, чем память ускорителя.

7. Результаты тестирования и обзор существующих решений

Тестирование производилось на GPU NVidia GTX Titan и на 6 ядерном процессоре Intel Xeon E5 v1660 с частотой 3.7 ГГц. Использовались графы в масштабе от 2^{16}(0.023ГБ) до 2^{26}(25.2ГБ). Граф масштаба 2^{16} достаточно мал (порядка 25 МБ) и даже без преобразований легко помещается в кэш одного современного процессора Intel Xeon. Однако, большие графы требуют достаточного количества памяти и граф 2^{24} масштаба уже не помещается в память тестируемого GPU без сжатия. Производительность измеряется в количестве обработанных ребер в секунду (traversed edges per second – TEPS). На рис. 12 показаны

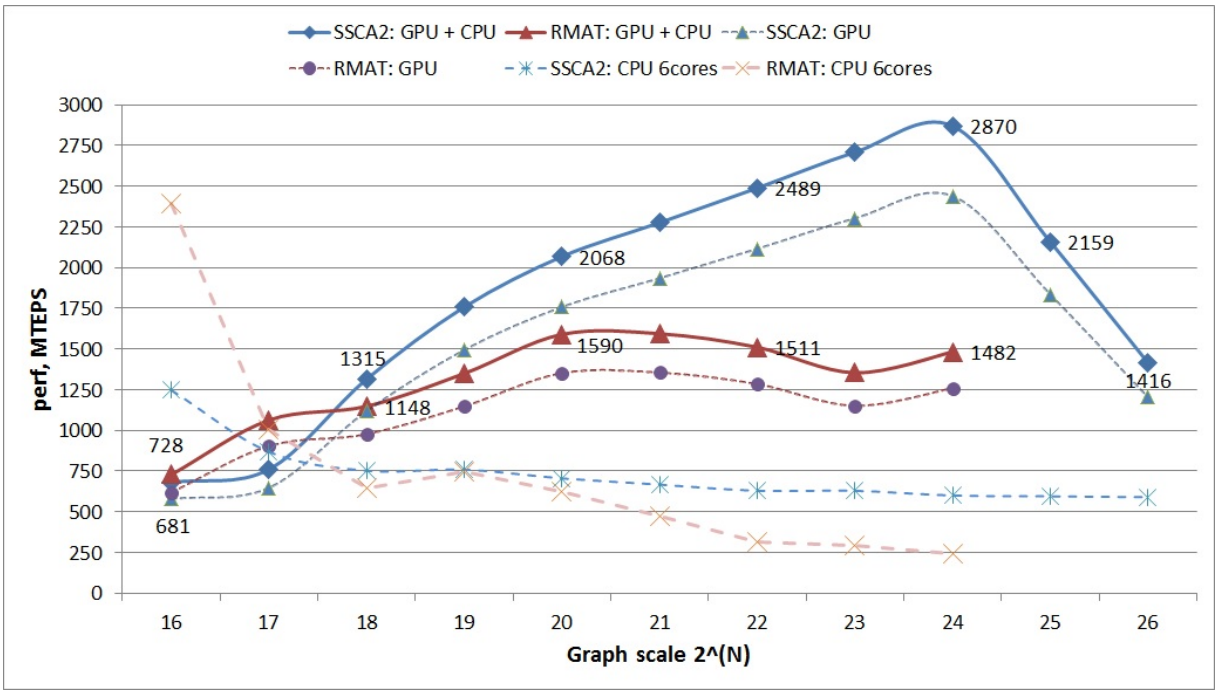

Рис. 12. Производительность гибридного параллельного и параллельного на CPU алгоритмов Борувки

производительности следующих алгоритмов: параллельного алгоритма на GPU, параллельного алгоритма на CPU, параллельного алгоритма на CPU и GPU. Из построенных графиков производительности видно, что с использованием всех оптимизаций на SSCA2-графах гибридный алгоритм показывают хорошую эффективность: чем больше граф, тем лучше производительность. Данный рост сохраняется до тех пор, пока все данные помещаются в память GPU. На 2^{25} и 2^{26} масштабах был использован механизм Unified Memory, что приводит к ухудшению производительность из-за более медленного доступа к памяти CPU.

Во многих зарубежных статьях, описывающих реализацию алгоритма MST с использованием GPU (например, [3–5]), не использовались какие-либо предобработки входного графа. В статье [5] приводятся результаты производительности на графах различных штатов Америки. Данные графы имеют среднюю степень связности вершин 2.5, количество вершин – от 264тыс до 6млн, а полученная производительность варьируется от 16 MTEPS до 36 MTEPS в зависимости от размера графа. Авторами данной статьи получены ускоре-

ния по сравнению с последовательной версией более, чем в 30 раз. В статье [4] алгоритм MST используется для кластеризации.

В статье [15] был реализован алгоритм Прима для поиска минимального остовного дерева. Авторы данной статьи использовали некоторые описанные ваше преобразования графа, например, сортировка ребер по возрастанию веса. В статье приводятся результаты производительности не только на графах различных штатов Америки, но и на R-MAT-графах и SSAC2-графах. Для генерации графов авторы статьи использовали инструмент "Georgia Tech graph generator suite". Полученная производительность варьируется от 20 MTEPS до 40 MTEPS на всех протестированных графах. RMAT-граф генерировался со средней степенью связности вершины 3, 6 и 9.

В статье [16] и других статьях, описываются похожие техники реализации алгоритма MST. Все описанные реализации демонстрируют невысокую производительность на GPU, по сравнению с реализованным гибридным алгоритмом.

8. Заключение

В результате проделанной работы был реализован гибридный параллельный алгоритм Борувки, использующий ядра центрального процесса и графического процессор. Данный алгоритм показывает высокие ускорение и эффективность на SSCA2-графах (максимальная производительность около 3000 MTEPS) и хорошие ускорения для RMAT-графов (максимальная производительность около 1500 MTEPS). Были получены достаточно высокие результаты на двух графах различной структуры по сравнению с последовательной версией алгоритма Борувки, а также сравнительно высокие результаты относительно существующих параллельных реализаций алгоритма Борувки на GPU.

Литература

1. Chazelle B.A., Minimum spanning tree algorithm with inverse-ackermann type complexity. Journal of the ACM. 2000. Vol. 47, No. 6. P. 1028–1047.

2. Pettie S. An inverse-Ackermann style lower bound for the online minimum spanning tree verification problem. Foundations of Computer Science. 2002. P. 155–163.

3. Wei W., Shaozhong G., Fan Y., Jianxun C. GPU-Based Fast Minimum Spanning Tree Using Data Parallel Primitives. Information Engineering and Computer Science (ICIECS), 2nd International Conferenc. 2010. P. 1–4.

4. Arefin A.S., Riveros C., Berretta R., Moscato P. kNN-MST-Agglomerative: A fast and scalable graph-based data clustering approach on GPU. Computer Science & Education (ICCSE), 7th International Conference. 2012. P. 585–590.

5. Vineet V., Harish P., Patidar S., Narayanan P.J. Fast Minimum Spanning Tree for Large Graphs on the GPU. HPG '09 Proceedings of the Conference on High Performance Graphics. 2009. P. 167–171.

6. Gibbs Norman E., Poole W.G., Stockmeyer P.K. A comparison of several bandwidth and profile reduction algorithms. ACM Transactions on Mathematical Software. 1976. Vol. 2, No. 4. P. 322-–330.

7. Gilbert J.R., Moler C, Schreiber R. Sparse matrices in MATLAB: Design and Implementation. SIAM Journal on Matrix Analysis and Applications. 1992. Vol. 13, No. 1. P. 333—356.

8. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A Recursive Model for Graph Mining. In Proceedings of 4th International Conference on Data Mining, 2004. P. 442–446.

9. Bader D.A.,Madduri K. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. Technical report. 2005.

10. Описание алгоритма Борувки. URL: https://goo.gl/8gG7oD (дата обращения: 30.11.2015)

11. Описание алгоритмов для структуры Union-Find. URL: https://www.cs.princeton.edu/ rs/AlgsDS07/01UnionFind.pdf (дата обращения: 30.11.2015)

12. Описание стандарта OpenMP. URL: http://openmp.org/wp/ (дата обращения: 30.11.2015)

13. Compute Unified Device Architecture. URL: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html (дата обращения: 30.11.2015)

14. Unified Memory in CUDA 6. URL: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ (дата обращения: 30.11.2015)

15. Nobari S., Cao T., Karras P., Bressan S. Scalable Parallel Minimum Spanning Forest Computation. PPoPP'12 Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. 2012. P. 205–214.

16. Wei W.,Shaozhong G.,Fan Y.,Jianxun C. Design and Implementation of GPU-Based Prim's Algorithm. Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference. 2010. P. 1–4.

Parallel implementation of minimum spanning tree algorithm on CPU and GPU

A.S. Kolganov[1,2]

Lomonosov Moscow State University[1], Keldysh Institute of Applied Mathematics RAS[2]

Solution of the finding a minimum spanning tree problem is common in various areas of research: recognition of different objects, computer vision, analysis and construction of networks (eg, telephone, electrical, computer, travel, etc.), chemistry and biology, and many others. There are at least three well-known algorithms for solving this problem: Boruvka, Kruskal and Prim. Processing large graphs is a quite time-consuming task for the central processor (CPU), and in high demand at the present moment. The usage of Graphics processing units (GPUs) as a mean to solve general-purpose problems grows every day, because GPUs have more computing power than CPUs. But the minimum spanning tree (MST) computation on a general graph is an irregular algorithm. So it suits poorly the GPU architecture. This article examins a hybrid implementation of this algorithm on GPU and CPU.

Keywords: MST, CUDA, NVidia, Graphs, Boruvka

References

1. Chazelle B.A., Minimum spanning tree algorithm with inverse-ackermann type complexity. Journal of the ACM. 2000. Vol. 47, No. 6. P. 1028–1047.

2. Pettie S. An inverse-Ackermann style lower bound for the online minimum spanning tree verification problem. Foundations of Computer Science. 2002. P. 155–163.

3. Wei W., Shaozhong G., Fan Y., Jianxun C. GPU-Based Fast Minimum Spanning Tree Using Data Parallel Primitives. Information Engineering and Computer Science (ICIECS), 2nd International Conferenc. 2010. P. 1–4.

4. Arefin A.S., Riveros C., Berretta R., Moscato P. kNN-MST-Agglomerative: A fast and scalable graph-based data clustering approach on GPU. Computer Science & Education (ICCSE), 7th International Conference. 2012. P. 585–590.

5. Vineet V., Harish P., Patidar S., Narayanan P.J. Fast Minimum Spanning Tree for Large Graphs on the GPU. HPG '09 Proceedings of the Conference on High Performance Graphics. 2009. P. 167–171.

6. Gibbs Norman E., Poole W.G., Stockmeyer P.K. A comparison of several bandwidth and profile reduction algorithms. ACM Transactions on Mathematical Software. 1976. Vol. 2, No. 4. P. 322—330.

7. Gilbert J.R., Moler C, Schreiber R. Sparse matrices in MATLAB: Design and Implementation. SIAM Journal on Matrix Analysis and Applications. 1992. Vol. 13, No. 1. P. 333—356.

8. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A Recursive Model for Graph Mining. In Proceedings of 4th International Conference on Data Mining. 2004. P. 442–446.

9. Bader D.A.,Madduri K. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. Technical report. 2005.

10. Opisanie algoritma Boruvki [Description of the Boruvka algorithm]. URL: https://goo.gl/8gG7oD (accessed: 30.11.2015)

11. Opisanie algoritmov dlya struktury Union-Find [Union-Find algorithms]. URL: https://www.cs.princeton.edu/ rs/AlgsDS07/01UnionFind.pdf (accessed: 30.11.2015)

12. Opisanie strandarta OpenMP [OpenMP standard]. URL: http://openmp.org/wp/ (accessed: 30.11.2015)

13. Compute Unified Device Architecture. URL: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html (accessed: 30.11.2015)

14. Unified Memory in CUDA 6. URL: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ (accessed: 30.11.2015)

15. Nobari S., Cao T., Karras P., Bressan S. Scalable Parallel Minimum Spanning Forest Computation. PPoPP'12 Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. 2012. P. 205–214.

16. Wei W.,Shaozhong G.,Fan Y.,Jianxun C. Design and Implementation of GPU-Based Prim's Algorithm. Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference. 2010. P. 1–4.