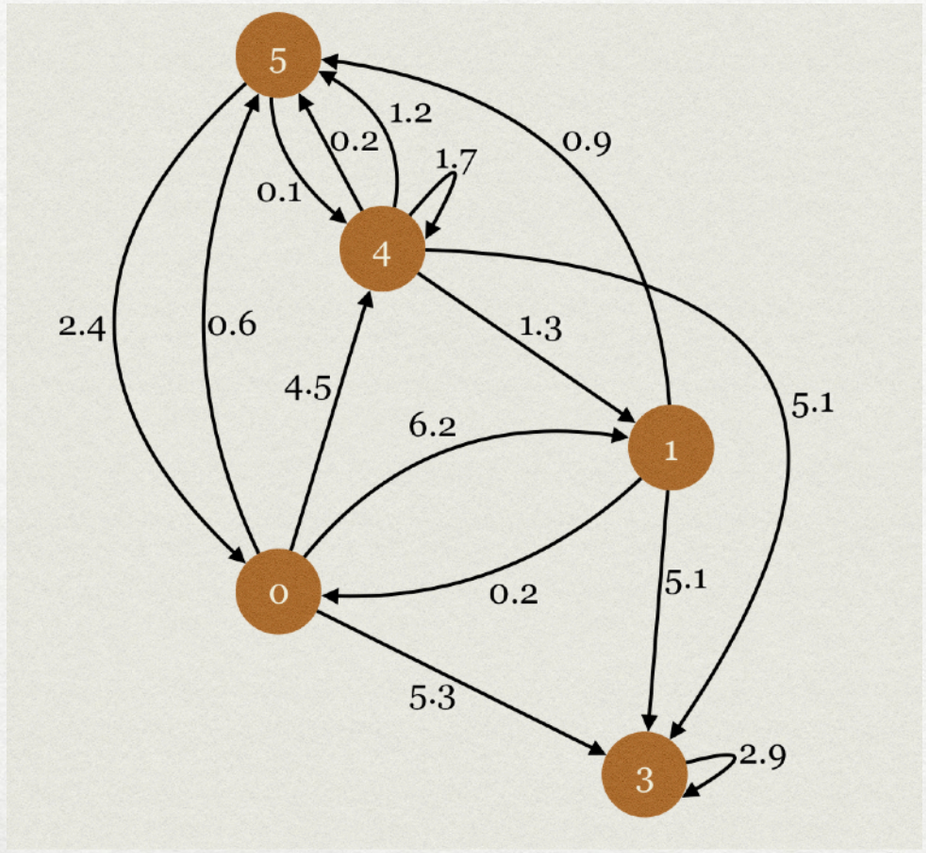
КОНКУРС!
РЕАЛИЗАЦИЯ АЛГОРИТМА ПОИСКА В ШИРИНУ (BFS) В ГРАФЕ

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

ГРАФЫ

Граф – это совокупность двух множеств: множества точек, которые называются вершинами, и множества ребер A. Каждый элемент есть упорядоченная пара элементов множества , вершины и называются концевыми точками или концами ребра a.

Пример графа



Графы бывают ориентированные и нет (мы будем работать с НЕ ориентированными)

Графы бывают взвешенными и нет (мы будем работать со НЕ взвешенными графами)

ГРАФЫ И GPU

- Обработка графов большого размера крайне актуальна в наши дни (обработка социальных сетей, веб-графов, дорожные карты, и др.)
- Графовые алгоритмы относятся к классу data-intensive задач, обычно имеют значительное число нерегулярных доступов к памяти
- В графовых задачах преобладает целочисленная арифметика
- GPU имеют значительный потенциал для обработки графов, так как:
  - Многие задачи имеют значительный потенциал параллелизма
  - GPU имеет сравнительно большую пропускную способность памяти в сравнении с другими платформами
  - Зачастую копирования данных занимают значительно меньшее время в сравнении с вычислениями

ФОРМАТЫ ХРАНЕНИЯ ГРАФОВ

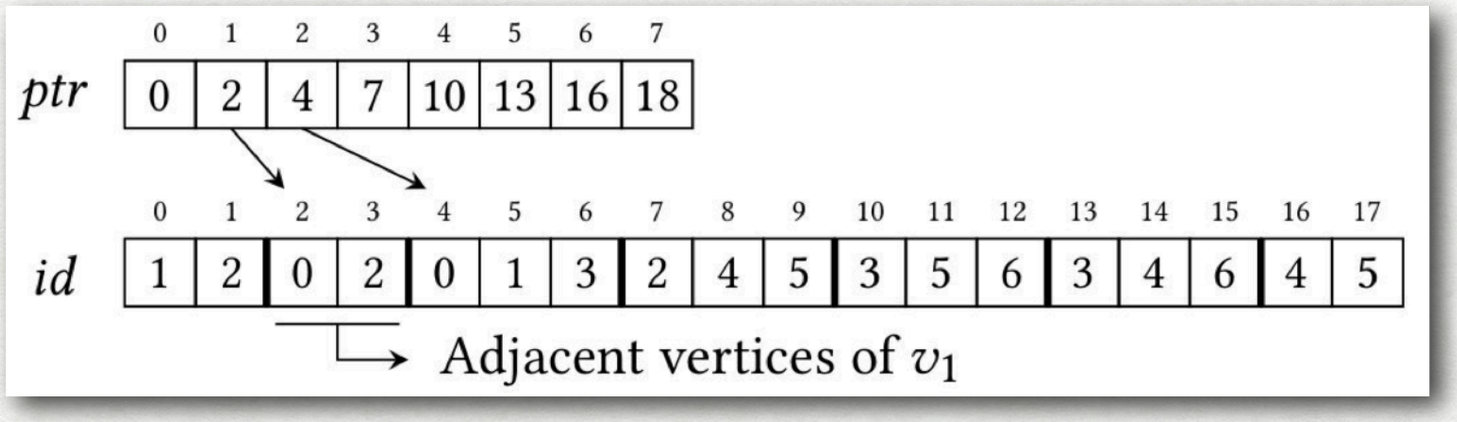
Пример формата хранения графа (edges list):

0	**1**	**4**	**3**	**4**	**5**	**4**	**7**	**8**	**9**	**10**	**11**	**14**	**13**	**14**
5	1	0	4	1	3	0	4	0	4	4	0	4	5	1
4	0	5	4	5	3	3	5	1	1	5	4	3	0	3
0.1	0.2	0.6	1.7	0.9	2.9	5.3	1.2	6.2	1.3	0.2	4.5	5.1	2.4	2.1

Преимущества списка ребер:
– тривиальная параллельная обработка (1 нить - 1 ребро)
– эффективный шаблон доступа к памяти (без необходимости оптимизаций)

Преимущества списка смежности:
– требует меньше памяти
– позволяет обрабатывать выбранный набор вершины (напр. 1-ую и 10-ую)

Список смежности (CSR):



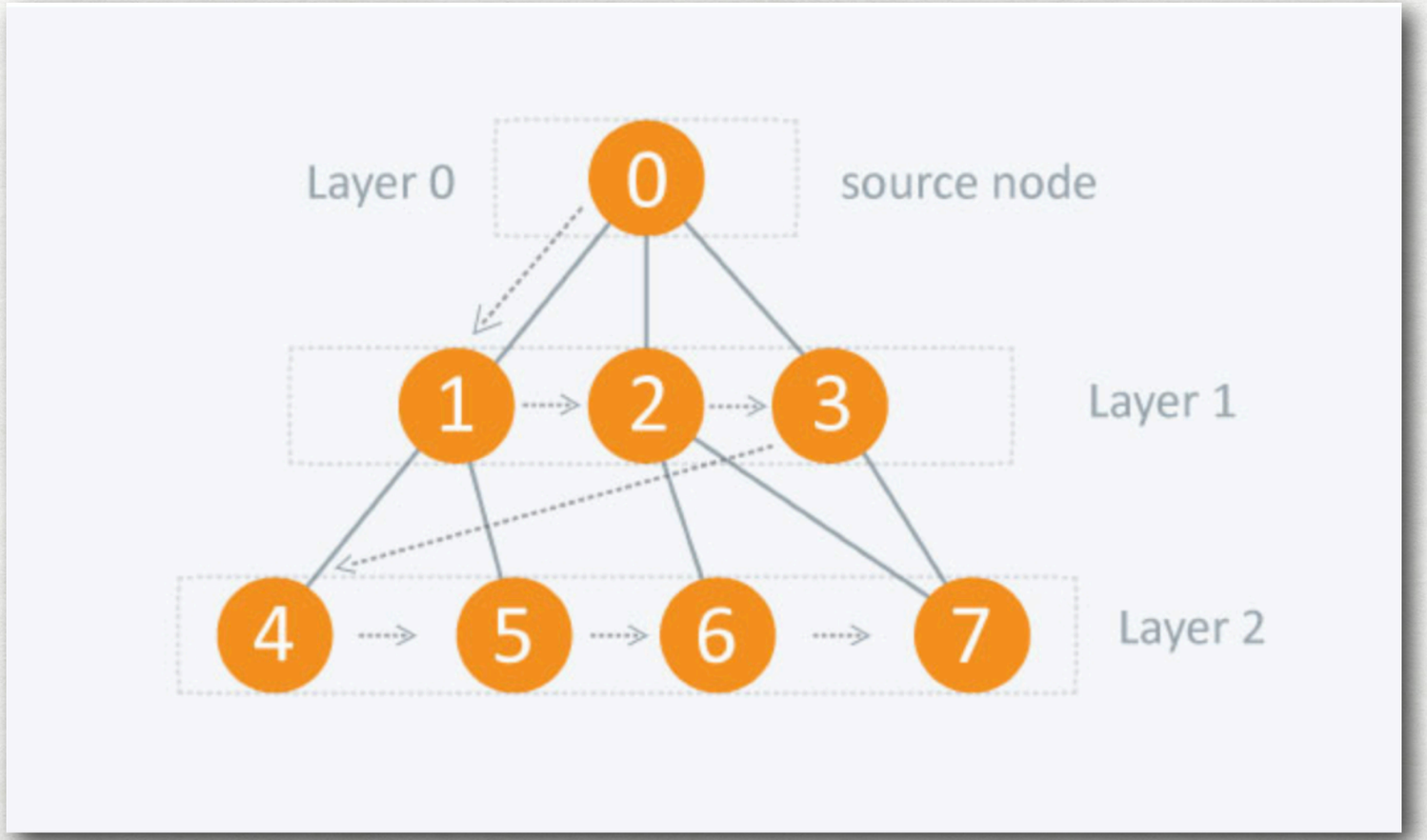
ЗАДАЧА ПОИСКА В ШИРИНУ

Алгоритм поиска в ширину (англ. breadth-first search, BFS) позволяет найти кратчайшие пути из одной вершины графа до всех остальных вершин.

кратчайший путь – содержащий наименьшее число ребер.

обзор графа «по слоям»

значительный ресурс параллелизма – каждая вершина слоя (напр 1, 2 и 3) могут быть обработаны **параллельно**



АЛГОРИТМ TOP-DOWN ПОИСКА В ШИРИНУ

- Алгоритм Top-Down - простейший алгоритм поиска в ширину в графе, его последовательная реализация приведена в файле bfs.hpp

```
while(!queue.empty())
{
    // Dequeue a vertex from queue and print it
    int s = queue.front();
    queue.pop_front();

    const long long edge_start = outgoing_ptrs[s];
    const int connections_count = outgoing_ptrs[s + 1] - outgoing_ptrs[s];

    for(int edge_pos = 0; edge_pos < connections_count; edge_pos++)
    {
        long long int global_edge_pos = edge_start + edge_pos;
        int v = outgoing_ids[global_edge_pos];
        if (_levels[v] == UNVISITED_VERTEX)
        {
            _levels[v] = _levels[s] + 1;
            queue.push_back(v);
        }
    }
}
```

А КАК АЛГОРИТМ TOP-DOWN РЕАЛИЗОВАТЬ НА GPU?

- Запустим vertices_count CUDA-нитей

```
register const int src_id = blockIdx.x * blockDim.x + threadIdx.x;

if (src_id < _vertices_count) // для всех графовых вершин выполнить следующее
{
    if(_levels[src_id] == _current_level) // если графовая вершина принадлежит текущему (ранее посещенному уровню)
    {
        const long long edge_start = _outgoing_ptrs[src_id]; // получаем положение первого ребра вершины
        const int connections_count = _outgoing_ptrs[src_id + 1] - _outgoing_ptrs[src_id]; // получаем число смежных ребер вершины

        for(int edge_pos = 0; edge_pos < connections_count; edge_pos++) // для каждого смежного ребра делаем:
        {
            int dst_id = _outgoing_ids[edge_start + edge_pos]; // загружаем ID напарвляющей вершины ребра

            if (_levels[dst_id] == UNVISITED_VERTEX) // если направляющая вершина - не посещенная
            {
                _levels[dst_id] = _current_level + 1; // то помечаем её следующим уровнем
            }
        }
    }
}
```

- Какие вы можете назвать недостатки данного подхода?

1. «ПРОСТОЙ» ВЫЧИСЛИТЕЛЬНЫХ НИТЕЙ

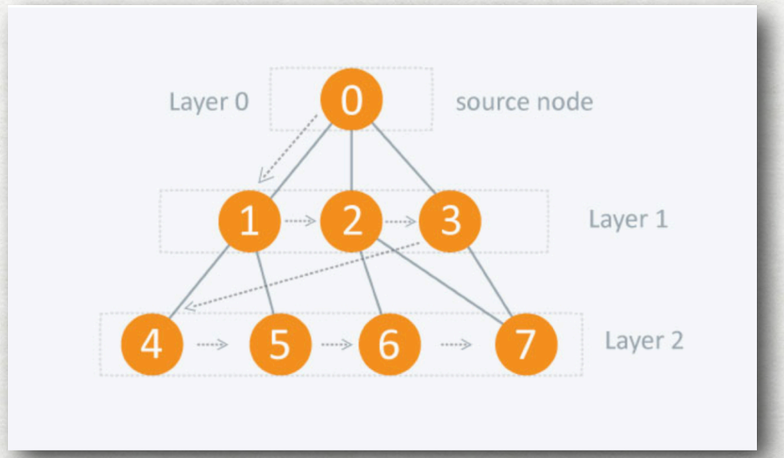
- Нити, вершины которых не принадлежат к текущему уровню – ничего не делают (простаивают)

```
register const int src_id = blockIdx.x * blockDim.x + threadIdx.x;

if (src_id < _vertices_count) // для всех графовых вершин выполнить следующее
{
    if(_levels[src_id] == _current_level)
  {
      const long long edge_start = _outgoing_ptrs[src_id]; // получаем положение первого ребра вершины
      const int connections_count = _outgoing_ptrs[src_id + 1] - _outgoing_ptrs[src_id]; // получаем число смежных ребер вершины

      for(int edge_pos = 0; edge_pos < connections_count; edge_pos++) // для каждого смежного ребра делаем:
      {
          ……
      }
   }
}
```

- Решение – создавать списки вершин, которые необходимо обрабатывать на следующем уровне (thrust copy_if)



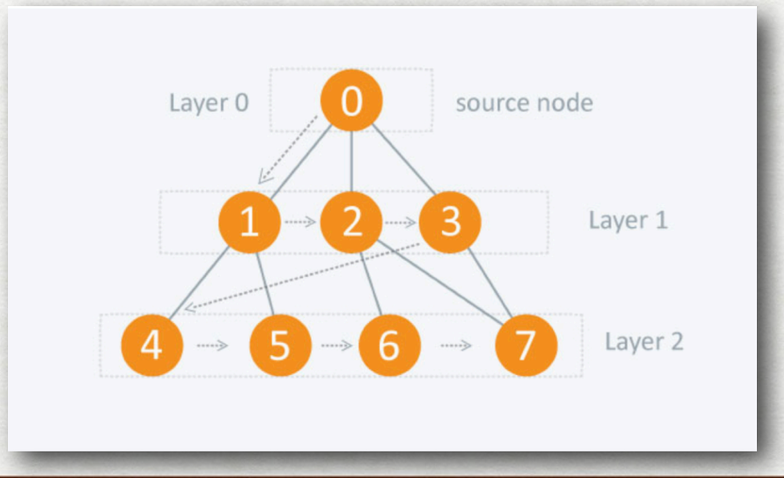
2. НЕСБАЛАНСИРОВАННОСТЬ ВЫЧИСЛИТЕЛЬНОЙ НАГРУЗКИ

- Пусть есть 2 нити, принадлежащие одному варпу – 21-ая и 22-ая
- connections_count[21] = 100000, connections_count[22] = 10 – частая ситуация в графах реального мира
- 22-ая нить будет ничего не делать, пока 21-ая не закончит обработку всех её смежных ребер

```
....
const int connections_count = _outgoing_ptrs[src_id + 1] - _outgoing_ptrs[src_id]; // получаем число смежных ребер вершины

for(int edge_pos = 0; edge_pos < connections_count; edge_pos++)
{
  ....
}
```

- Решение – создавать отдельные списки вершин с различной степенью связанности: [1-32], [32-1024], [1024+]
- Выделять на обработку вершин первой группы – 1 нить
- Выделять на обработку вершин второй группы - 32 нити
- Выделять на обработку вершин третьей группы - 1024 нити
- можно (и нужно!) использовать другие значения



3. СЛУЧАЙНЫЕ ОБРАЩЕНИЯ К ПАМЯТИ

- Обращения к массиву о вершинах (levels) – разбросаны по памяти => кэши используются неэффективно

```
register const int src_id = blockIdx.x * blockDim.x + threadIdx.x;

if (src_id < _vertices_count) // для всех графовых вершин выполнить следующее
{
        ....

        if (_levels[dst_id] == UNVISITED_VERTEX) // если направляющая вершина - не посещенна
        {
            _levels[dst_id] = _current_level + 1; // то помечаем её следующим уровнем
        }
    }
  }
}
```

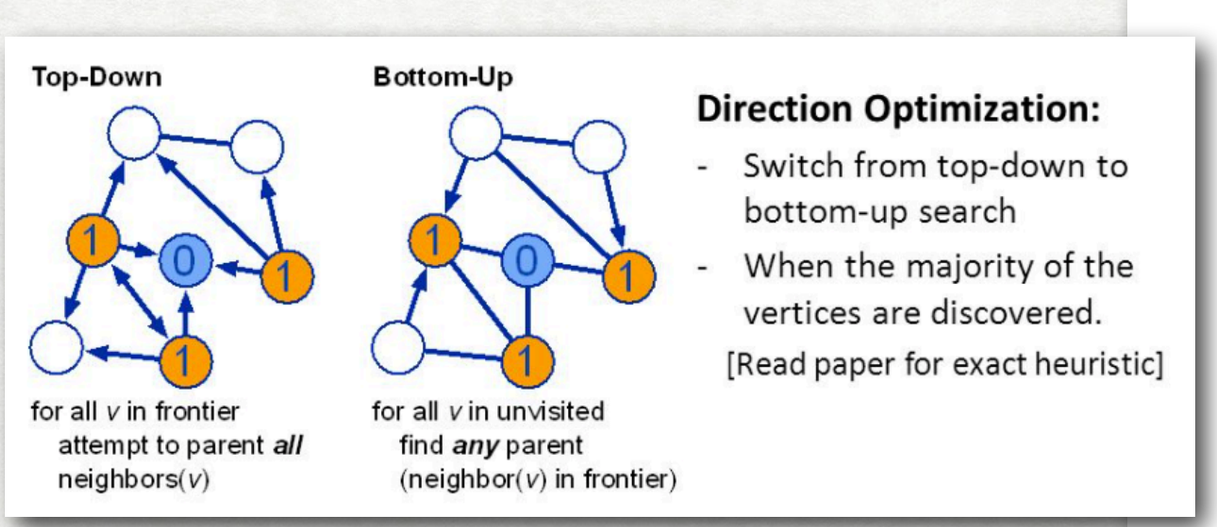
- Решение – локализация обращений к наиболее часто запрашиваемым вершинам (с высокой степенью связанности) – сортировка вершин по убыванию степени связанности

- использования texture кэша (__ldg/const restrict)

- использование разделяемой памяти для хранения данных вершин

4. ИСПОЛЬЗОВАНИЕ ДРУГОГО АЛГОРИТМА (BOTTOM-UP ПОИСКА В ШИРИНУ)

- Алгоритм Bottom-up поиска в ширину – использует обработное направление обхода
- Для всех вершин графа проверяем, есть ли для неё смежная еще не посещенная




```
for v ∈ vertices do
    if parents[v] = -1 then
        for n ∈ neighbors[v] do
            if n ∈ frontier then
                parents[v] ← n
                next ← next ∪ {v}
                break
            end if
        end for
    end if
end for
```

- Можно для каждого уровня чередовать направления, выбирая более эффективное!

С ЧЕГО НАЧАТЬ?

- В начале конкурса всем участникам будет представлено:
  - Генератор случайных графов
  - CPU версия алгоритма с пояснительными комментариями, принимающая на параметры генерируемого графа
  - Неоптимизированная GPU версия алгоритма (описанная ранее)
  - Верификатор полученных результатов
  - Python-скрипт для отправки результата в таблицу

ВОЗМОЖНЫЕ ДРУГИЕ ПОДХОДЫ К ОПТИМИЗАЦИИ:

- Оптимизация грида (размер блоков, occupancy)
- Оптимизация обращений к памяти (использование текстурного кэша, улучшение локальности данных в графе)
- Изменение формата хранения графа (edges list, CSR)
- Использование cudaStreams (для обработки вершин с разной степенью связанности)
- Оптимизация пересылок данных с хоста на девайс
- Использование multi-GPU (разбиение входного графа на 2 части, разные итерации BFS на разных GPU)
- Использование гетерогенных вычислений (CPU + GPU)
- Использование альтернативных алгоритмов...

БОЛЕЕ ПОДРОБНО ПРО ПОСЫЛКУ РЕЗУЛЬТАТОВ

- Скрипт submit_result.py прилагается к базовым исходным кодам и служит для отправки результатов в итоговую таблицу
- Необходимо установить права на запуск: chmod 777 submit_result.py
- Скрипт очень прост и бесхитростен: он подает на вход программе определенный набор сгенерированных графов, проверяет правильность ответа и отправляет результат работы в таблицу на сервер
- Вычисление производительности базируется на парсинге вывода программы. **Пожалуйста, не меняйте формат вывода!**
- Да, обмануть скрипт не сложно. **Пожалуйста, не жульничайте!**

ПРИМЕР СБОРКИ И ТЕСОВЫХ ЗАПУСКОВ

- Давайте все вместе сделаем первую посылку!
- cp -r **/home/afanasyevi/graph_for_contest/*** ~/cuda_contest/bin/
- cd ./my_dir
- make -f Makefile.polus
- cd bin
- ./bfs
- ./bfs -load ru_20_16.el_graph -check -it 5
- mpisubmit.pl -g ./bfs -- -load ru_20_16.el_graph // это просто тестирование программы (для отладки)!!! в табличку эта команда не отправит ничего!

ПРИМЕР ОТПРАВКИ РЕЗУЛЬТАТА В ТАБЛИЦУ

- Отредактируйте скрипт если хотите более симпатичное имя (в противному случае будет имя вашего пользователя на Polus)
- *vim submit_result.py*
- username = str(getpass.getuser()) -> username = "my_cool_name"

ls

bfs submit_result.py ***//должны находиться в одной папке!!!***

//!! в той же папке должны лежать входные графы!!!

скопировать графы можно из папки /home/afanasyevi/graph_for_contest

mpisubmit.pl -g python –– ./submit_result.py

Job <3832> is submitted to default queue <normal>.