ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ BFS НА GPU В ТЕРМИНАХ ЛИНЕЙНОЙ АЛГЕБРЫ

ЕГОР ОРАЧЕВ,
RENDER PROGRAMMER,
SABER INTERACTIVE

5 ИЮНЯ, 2024

Алгоритм обхода в ширину

- Классический поиск в ширину / обход в ширину
- Начинаем из выделенной вершины
- Обходим все достижимые вершины из выделенной
- Считаем расстояние до достигнутых вершин

```
 1  procedure BFS(G, root) is
 2      let Q be a queue
 3      label root as explored
 4      Q.enqueue(root)
 5      while Q is not empty do
 6          v := Q.dequeue()
 7          if v is the goal then
 8              return v
 9          for all edges from v to w in G.adjacentEdges(v) do
10              if w is not labeled as explored then
11                  label w as explored
12                  w.parent := v
13                  Q.enqueue(w)
```

Особенности алгоритма

- В худшем случае необходимо посетить все вершины графа
- Необходимо помечать посещенные вершины
- Требуется очередь для хранения активного фронта обхода
- Требуется массив чтобы сохранять дистанции до каждой вершины
- Требуется структура данных для графа чтобы оптимально итерироваться по смежным вершинам с любой выбранной вершиной
- Опционально массив чтобы хранить родителя каждой посещенной вершины

Возможности для оптимизаций

Идея: все вершины в текущем фронте обхода можно обработать независимо

Проблемы:

- Необходимо синхронизировать доступ к очереди
- Необходимо безопасно помечать уже посещенные
- Необходимо вовремя понять когда остановиться

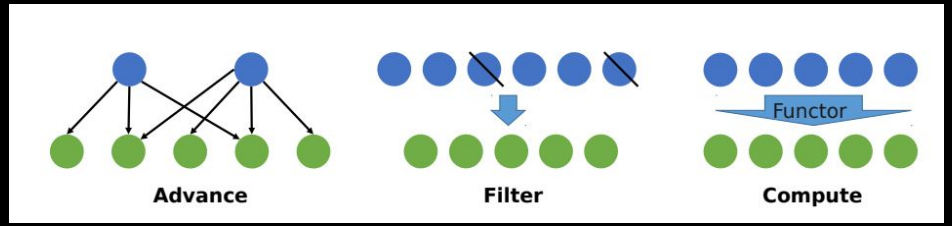
Реализация на GPU?

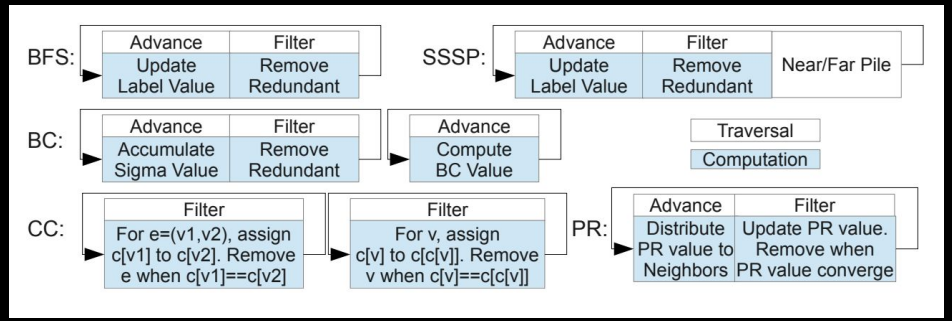
В точности реализация вышеизложенных идей

- Граф в виде списка смежности
  - Буфер смещений
  - Буфер с индексами смежных вершин
- Очередь вершин
  - Буфер с индексами вершин для посещения
  - Атомарные операции для добавления индексов в очередь
- Множество посещенных
  - Буфер с булевыми флагами для каждой вершины
- Ядро
  - Потоки достают вершины из очереди
  - Каждый поток обрабатывает свою вершину
  - Добавляем в очередь еще непосещенные вершины
- Повторяем ядро пока в очереди есть вершины

Gunrock

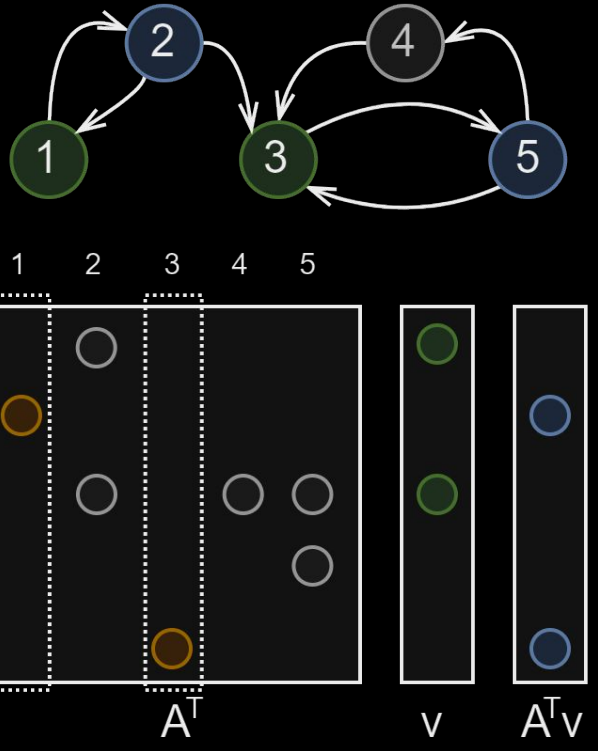
- Практически в точности реализация алгоритма выше
- CUDA для GPU
- Нетривиальная система балансировки нагрузки
- Нетривиальная фильтрация фронта обхода
- Автоматизация рутины для работы с графом





Реализация в терминах линейной алгебры

- Более высокий уровень абстракции
- Возможность использовать существующий математический аппарат
- Существующие структуры данных
- Существующие алгоритмы для эффективной реализации операций
- Потенциальные незначительные накладные расходы
- Простота использования для пользователя
- Высокая сложность реализации для разработчика



GraphBLAS

- Граф в виде матрицы смежности
- Булево / целочисленное кольцо для вычислений
- **MXV** для обновления фронта обхода
- **ASSIGN** для обновления дистанций
- **REDUCE** для вычисления условий сходимости

1: **procedure** GrB_BFS(Vector $\mathbf{v}$, Graph $\mathbf{A}$, Source s)
2: Initialize $d \leftarrow 1$
3: Initialize $\mathbf{f}(i) \leftarrow \begin{cases} 1, & \text{if } i = s \\ 0, & \text{if } i \neq s \end{cases}$ ▷ GrB_Vector_new
4: Initialize $\mathbf{v} \leftarrow [0, 0, \ldots, 0]$ ▷ GrB_Vector_new
5: Initialize $c \leftarrow 1$
6: **while** $c > 0$ **do**
7: Update $\mathbf{v} \leftarrow \mathbf{f} \times d + \mathbf{v}$ ▷ GrB_assign
8: Update $\mathbf{f} \leftarrow \mathbf{A}^{\mathrm{T}}\mathbf{f} \,.*\, \neg\mathbf{v}$ ▷ GrB_mxv
9: Compute $c \leftarrow \sum_{i=0}^{n} \mathbf{f}(i)$ ▷ GrB_reduce
10: Update $d \leftarrow d + 1$
11: **end while**
12: **end procedure**

GraphBLAST

- Вдохновлено GraphBLAS стандартом
- C++ API, видоизмененный
- CUDA для вычислений на GPU
- Референсный CPU бэкенд для прототипирования

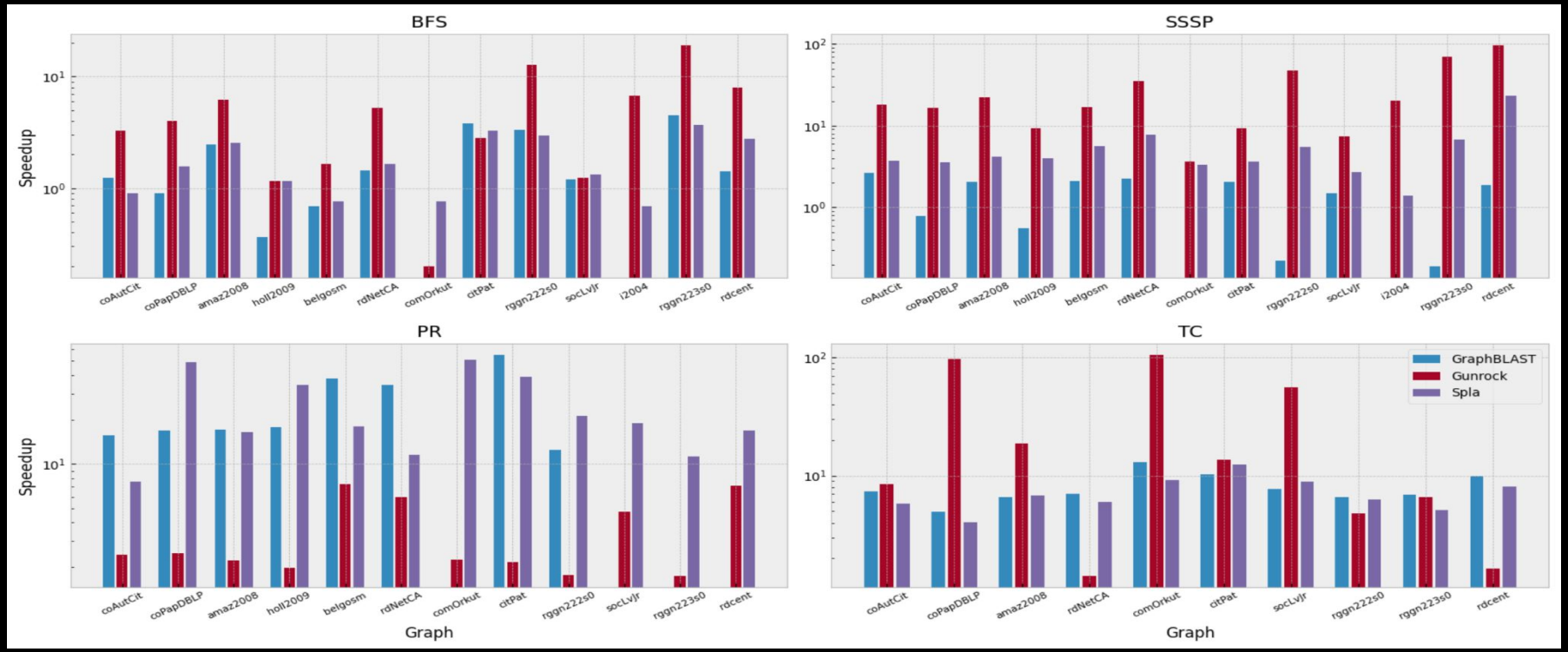
```cpp
#include <graphblas/graphblas.hpp>

void bfs(Vector<float>*       v,
         const Matrix<float>* A,
         Index                s) {
  Descriptor desc;
  Index A_nrows;
  A->nrows(&A_nrows);
  float d = 1.f;

  Vector<float> f1(A_nrows);
  Vector<float> f2(A_nrows);
  std::vector<Index> indices(1, s);
  std::vector<float> values(1, 1.f);
  f1.build(&indices, &values, 1, GrB_NULL);

  v->fill(0.f);
  float c = 1.f;
  while (c > 0) {
    // Assign level d at indices f1 to visited vector v
    graphblas::assign(v, &f1, GrB_NULL, d, GrB_ALL, A_nrows, &desc);
    // Set mask to use structural complement (negation)
    desc->toggle(GrB_MASK);
    // Multiply frontier f1 by transpose of matrix A using visited vector v as mask
    // Semiring: Boolean semiring (see Table 4)
    graphblas::vxm(&f2, v, GrB_NULL, LogicalOrAndSemiring<float>(), &f1, A, &desc);
    // Set mask to not use structural complement (negation)
    desc->toggle(GrB_MASK);
    f2.swap(&f1);
    // Check how many vertices of frontier f1 are active, stop when number reaches 0
    // Monoid: Standard addition (see Table 4)
    graphblas::reduce(&c, GrB_NULL, PlusMonoid<float>(), &f1, &desc);
    d++;
  }
}
```

Spla

- Вдохновлено GraphBLAS стандартом
- C++ API, Python API
- OpenCL для вычислений на GPU
- Референсный CPU бэкенд для прототипирования

```
SPLA_API Status spla::bfs(const ref_ptr<Vector>&     v,
                          const ref_ptr<Matrix>&     A,
                          uint                       s,
                          const ref_ptr<Descriptor>& desc) {
    const auto N = v->get_n_rows();

    ref_ptr<Vector> front_prev     = make_vector(N, INT);
    ref_ptr<Vector> front          = make_vector(N, INT);
    ref_ptr<Scalar> front_size     = make_int(1);
    ref_ptr<Scalar> depth          = make_int(1);
    ref_ptr<Scalar> zero           = make_int(0);
    int             current_level  = 1;

    front_prev->set_int(s, 1);

    while (!(front_size->as_int() == 0)) {
        depth->set_int(current_level);

        exec_v_assign_masked(v, front_prev, depth, SECOND_INT, NQZERO_INT, desc);
        exec_vxm_masked(front, v, front_prev, A, BAND_INT, BOR_INT, EQZERO_INT, zero, desc);
        exec_v_reduce(front_size, zero, front, PLUS_INT, desc);

        current_level += 1;

        std::swap(front_prev, front);
    }

    return Status::Ok;
}
```

Производительность



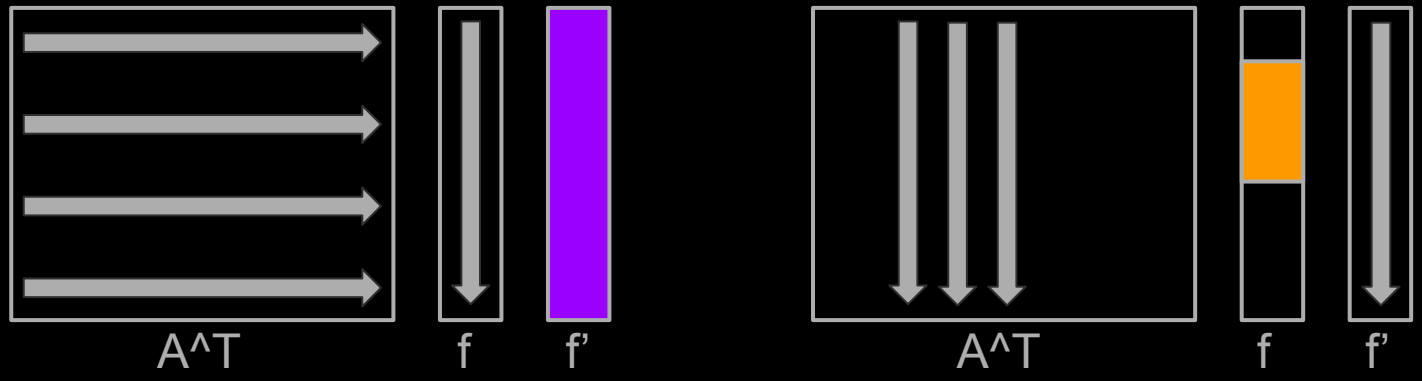
Ускорение относительно **SuiteSparse::GraphBLAS** реализации

Проблемы

- Наивная реализация не состоятельна
- Много итераций поиска (если у графа большой диаметр)
- Маленький фронт обхода дает слабую загрузку девайса
- Стоимость доступа к памяти превышает сложность вычислений
- Время работы не пропорционально получаемому результату
- Использование плотных векторов на графах ~1M+ вершин и диаметре ~10+ итераций нивелирует весь прирост производительности

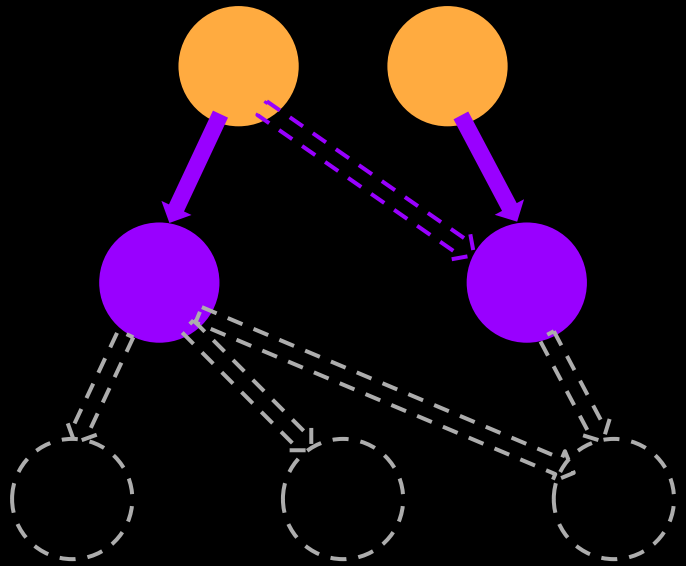
Оптимизации: направление обхода

- Change of direction / push and pull
- Менять направление BFS при изменении плотности фронта
- Стратегия:
  - Push: фронт маленький, много непосещенных
  - Pull: фронт большой, много-средне непосещенных
  - Push: фронт маленький, мало непосещенных

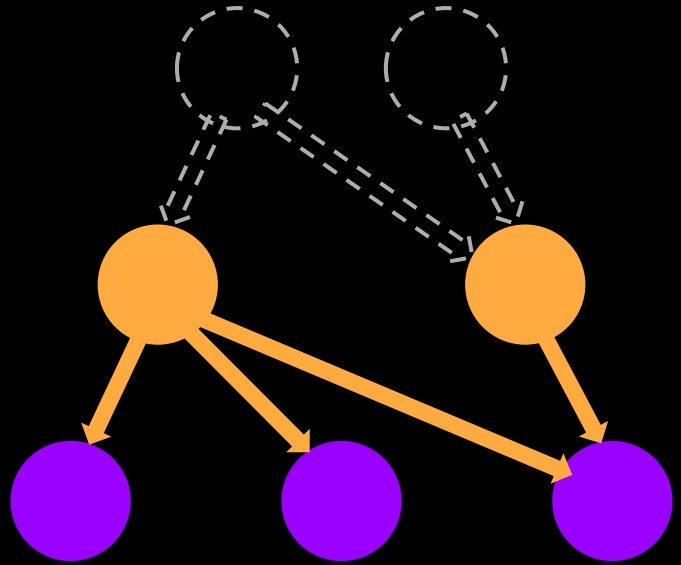


Push and Pull

Pull, старт из непосещенных

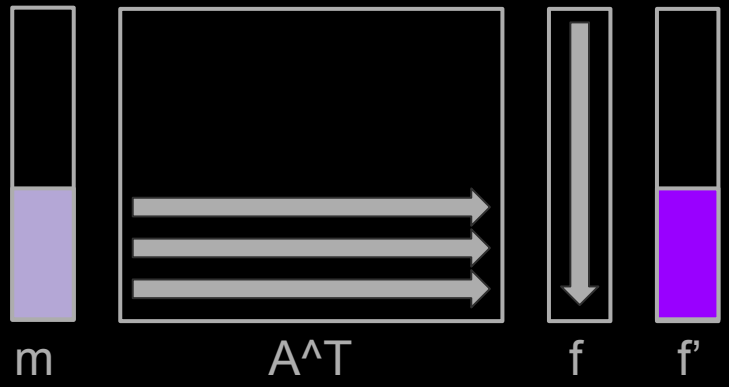


Push, старт из фронта



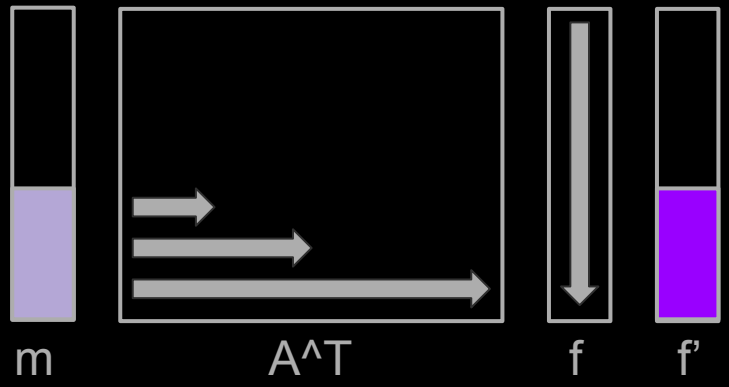
Оптимизации: использование маски

- Msking
- Контролирует кол-во строк матрицы, которые необходимо обработать на pull итерации (умножение на плотный вектор)
- Чем плотнее маска, тем меньше элементов обрабатывать



Оптимизации: ранний выход

- Early-exit
- Актуально для pull-направления
- Мотивация: для выделенной вершины достаточно первого ребра, которое смежно с фронтом обхода, чтобы пометить вершину достижимой
- В булевом полукольце выглядит как: f’[i] = g[i, 0] && f[0] || g[i, 1] && f[1] || …



Оптимизации: переиспользование операндов

- Operand reuse
- На этапе перехода от push к pull
- Исключает необходимость конвертирования плотного фронта к разреженному
- Заменить **f` = $A^T$ f.*-v на f` = A^T v.*-v** тк f это всегда подмножество v

Оптимизации: только структура

- Structure-only
- Актуально для push-направления / частично для pull
- В разреженных форматах наличия элемента уже достаточно
- Значения не интересуют (гарантированно true)
- Можно сократить кол-во операций, используя только массивы индексов

Требуемые структуры и операции

- Матрица в CSR формате (CPU + GPU)
- Вектор в разреженном формате COO
- Вектор в плотном формате
- Конвертация вектора плотный ↔ разреженный
- Обновление плотного вектора по разреженной маске
- Обновление плотного вектора по плотной маске
- Редукция плотного вектора
- Редукция разреженного вектора
- Умножение разреженной матрицы на разреженный вектор
- Умножение разреженной матрицы на плотный вектор с маской

Нюансы

- Эвристики перехода от pull к push и обратно
- Различные форматы хранения данных
- Накладные расходы на множество обращений к драйверу (OpenCL)
- Матрично-векторное умножение (matvec)
  - Умножение разреженной матрицы на плотный вектор (SpMV)
  - Умножение разреженной матрицы на разреженный вектор (SpMSpV)

SpMxSpV

- Column-based ***matvec***
- Матрица в CSR формате
- Вектор в COO формате
- Алгоритм:
  - Вычислить кол-во ненулевых произведений
  - Вычислить промежуточные произведения каждой колонки матрицы на элемент вектора
  - Выполнить сортировку по ключам
  - Выполнить редукцию по ключам чтобы исключить дубликаты

```
1:  procedure COL_MASKED_MXV(Vector v, Graph A^T, MaskVector m,
    MaskIdentity identity, Boolean scmp, Boolean accum)
2:      for each thread i in parallel do
3:          length(i) ← row_ptr(i+1)-row_ptr(i) for all i such that v(i) ≠ 𝕀
4:      end for
5:      scan ← prefix-sum length
6:      addr(i) ← IntervalGather(scan, v)
7:      col ← col_ind(j) such that A^T(j, i) ≠ 𝕀 from addr(i)
8:      val ← A^T(j, i) from addr(i)
9:      val ← val ⊗ v(i)
10:     write (col, val) to global memory
11:     Barrier synchronization
12:     key-value sort (col, val)
13:     (optional: structure only opt. turns this into a key-only sort)
14:     Barrier synchronization
15:     segmented-reduction using (⊕, 𝕀) produces w'
16:     Barrier synchronization
17:     for each thread i in parallel do
18:         ind ← ind such that w'(ind) ≠ 𝕀
19:         if m(ind) ≠ identity XOR scmp then
20:             w''(i) = w'(i)
21:         else
22:             w''(i) = 𝕀
23:         end if
24:     end for
25:     if accum then
26:         w ← w + w''
27:     else
28:         w ← w''
29:     end if
30: return w
31: end procedure
```

Литература и ссылки

- **Основной источник об алгоритме BFS**
  Carl Yang, Aydın Buluç, John D. Owens. Implementing Push-Pull Efficiently in GraphBLAS. In Proceedings of the International Conference on Parallel Processing, ICPP, pages 89:1-89:11, August 2018.

- **Эффективная реализация SpMSpV matvec**
  Carl Yang, Yangzihao Wang, and John D. Owens. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. In Graph Algorithms Building Blocks, In Graph Algorithm Building Blocks, GABB, pages 841–847, May 2015.

- **Библиотека GraphBLAST**
  Carl Yang, Aydin Buluc, John D. Owens, GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU, ACM Transactions on Mathematical Software, Volume 48, Issue 1, Article No.: 1pp 1–51