Collision detection 2

Вычисления на видеокартах. Лекция 6

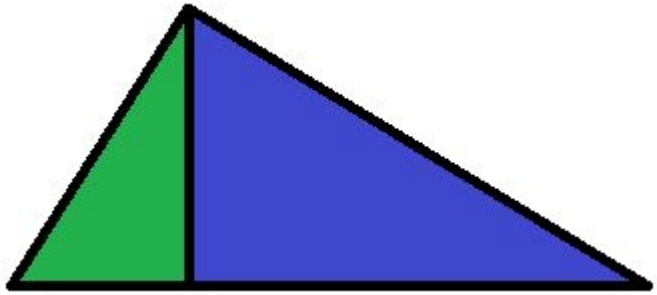
Bitonic sort, Radix sort
Bounding Volume Hierarchy
Z-Order Curve, LBVH

Полярный Николай

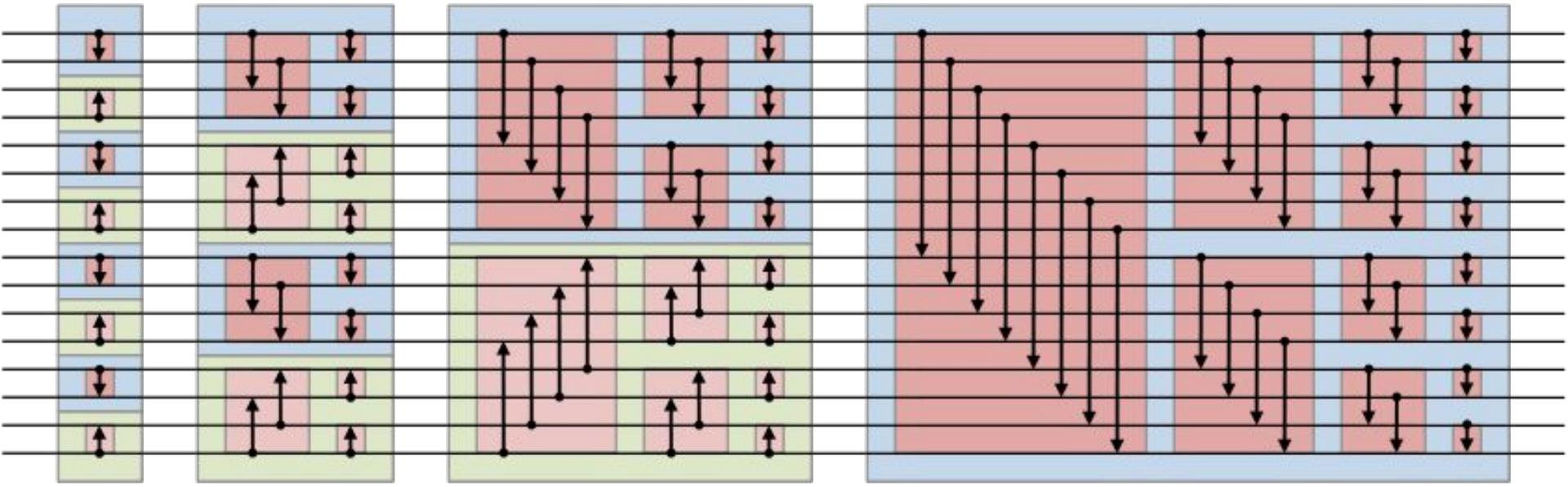
polarnick239@gmail.com

Bitonic sort

Битонная последовательность состоит из неубывающей последовательности элементов за которой следуют невозрастающая последовательность элементов.



Bitonic sort

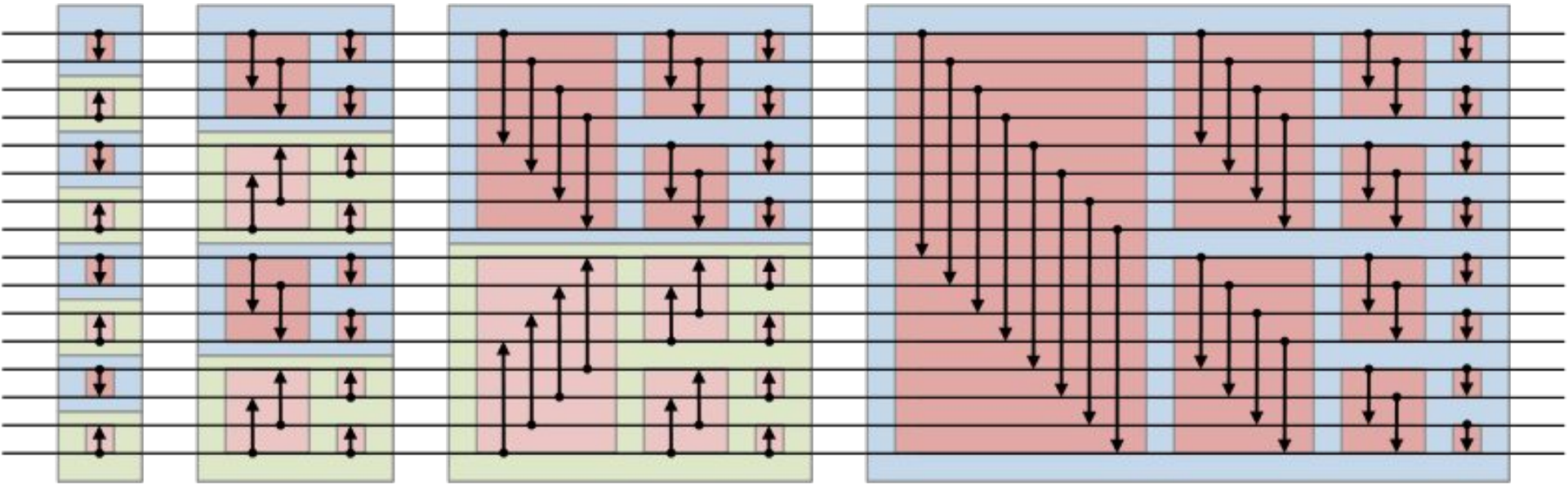


Стрелка сравнивает два числа и меняет их так, чтобы стрелка указывала на большее число.

Синие под-блоки выдают **возрастающие** последовательности.

Зеленые под-блоки выдают **убывающие** последовательности.

Bitonic sort

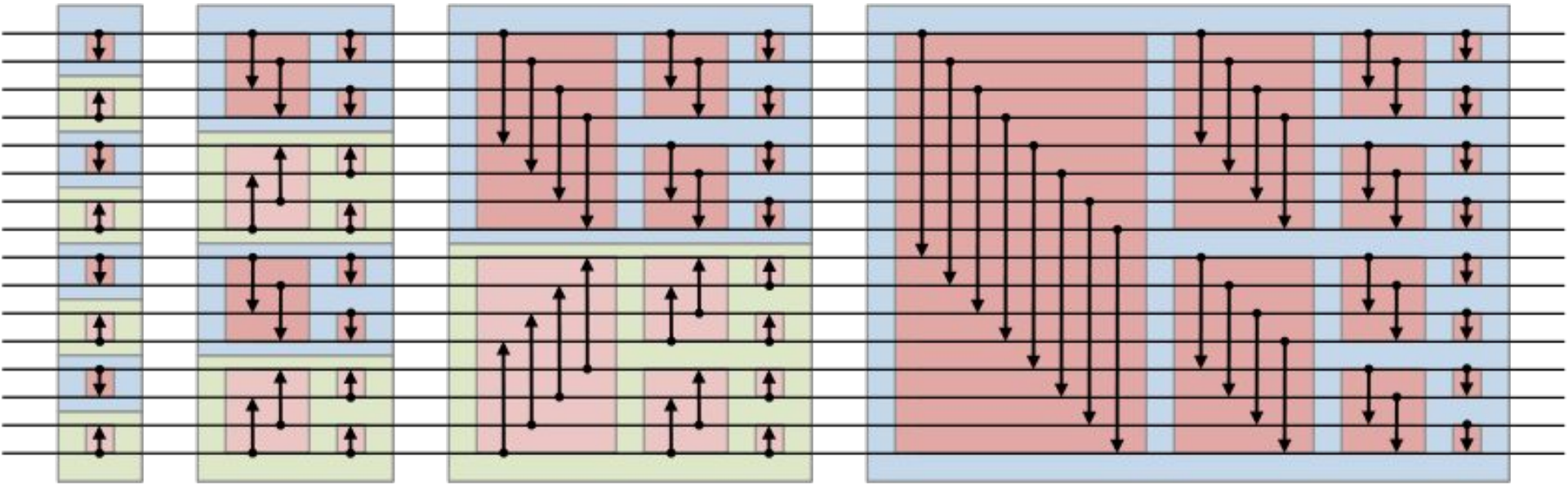


На вход каждому большому блоку подается **N** отсортированных подпоследовательностей каждая длины **K** (каждая вторая - убывающая).

На выход каждый блок выдает **N/2** отсортированных последовательностей размера **2*K**.

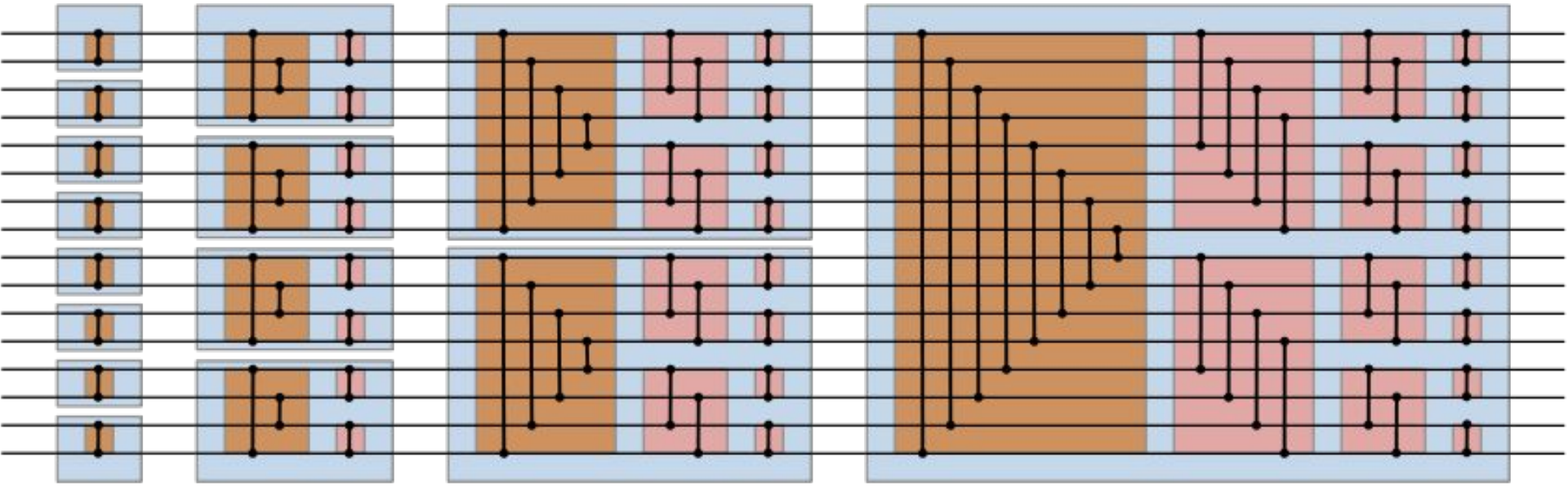
Сначала **N=16** и **K=1**, затем **N=8** и **K=2**, затем **N=4** и **K=4**, затем **N=2** и **K=8**.

Bitonic sort



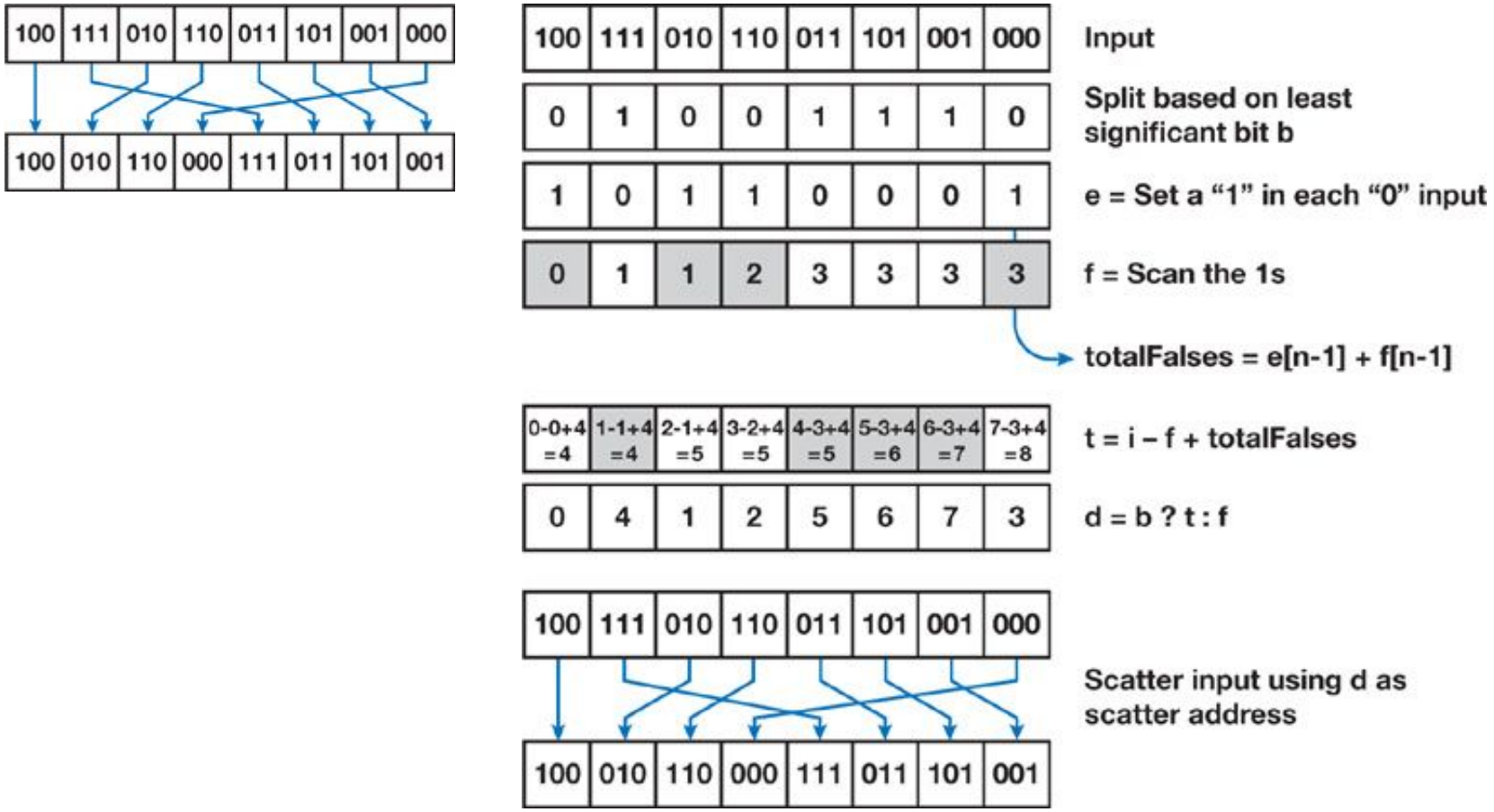
Т.е. каждый блок принимает на вход несколько **битонных последовательностей** и преобразует каждую из них в отсортированную последовательность, которая является половиной битонной последовательности подаваемой на вход следующему блоку.

Bitonic sort (альтернативная версия)

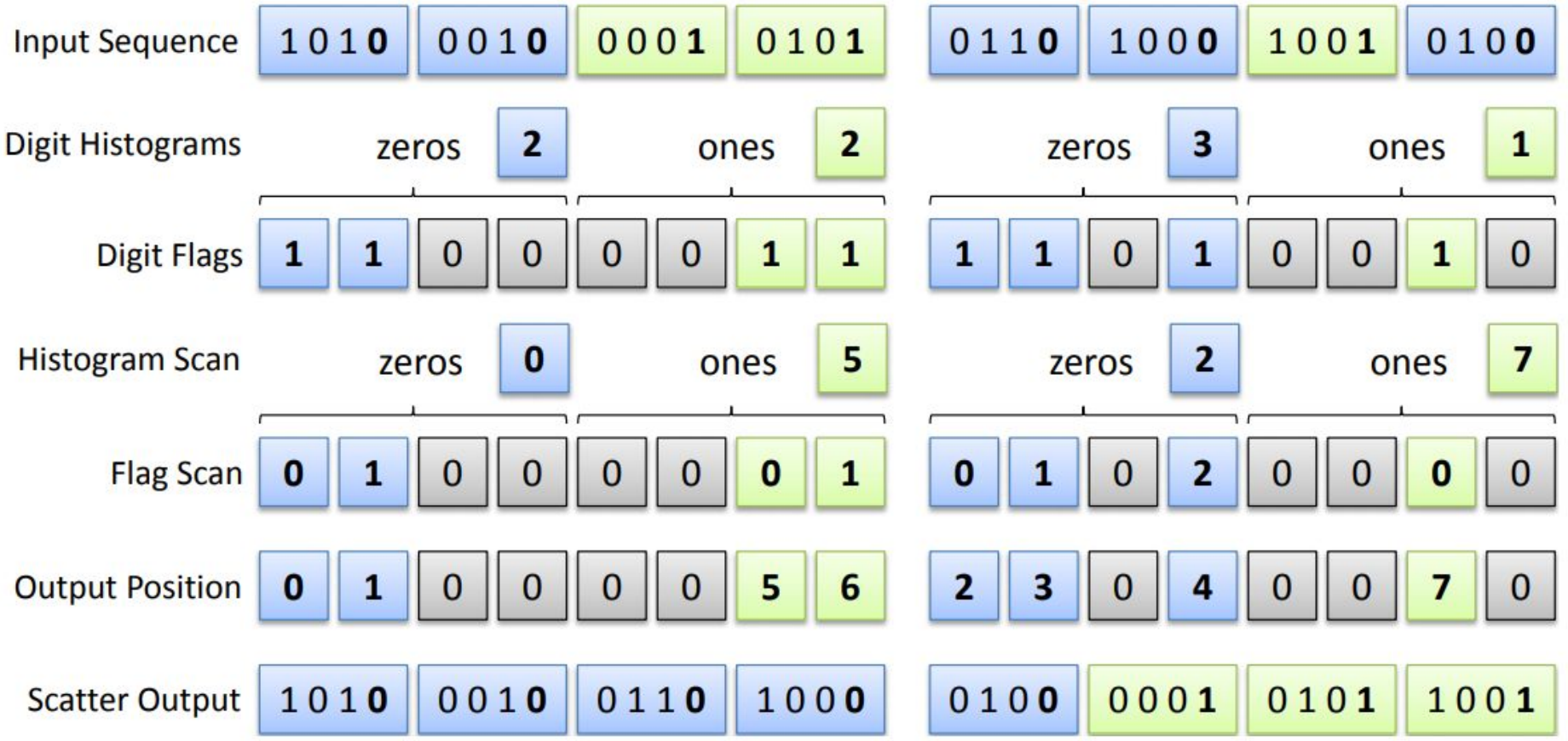


Из любых двух отсортированных последовательностей можно получить битонную последовательность перевернув порядок во второй отсортированной последовательности.

Radix sort: Local



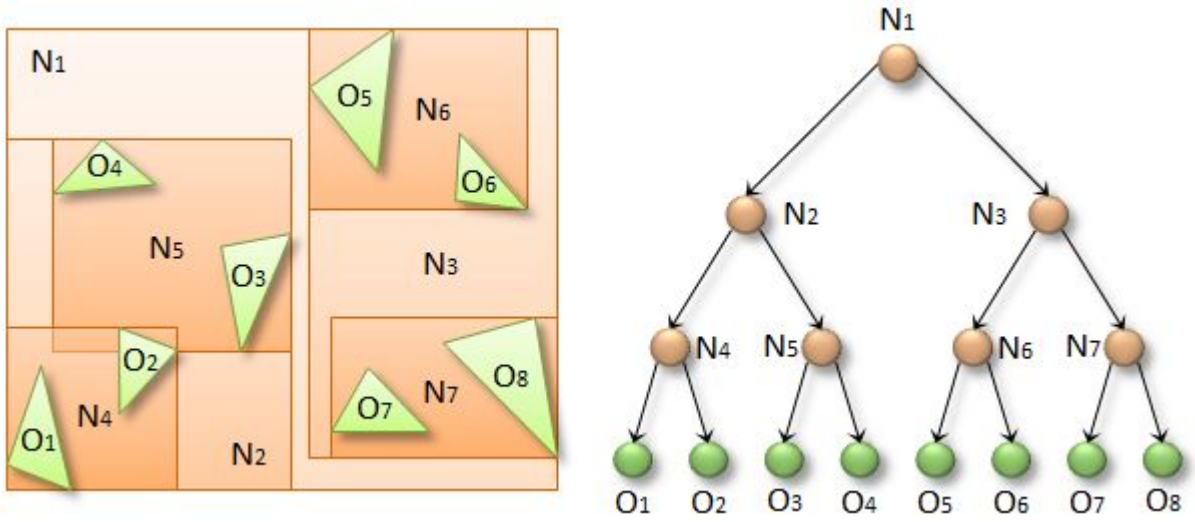
Radix sort: Local + Global



Bounding Volume Hierarchy

Листья - объекты.

Узлы - AABB содержащие все листья-объекты поддерева.



Рекурсивный подход

```
void traverseRecursive(AABB                        queryAABB,
                       int                         queryObjectIdx,
                       const __global BVHNode*     node,
                       __local CollisionList*      list)
{
    if (checkOverlap(node->getAABB(), queryAABB)) {
        if (node->isLeaf()) {
            list.add(queryObjectIdx, node->getObjectIdx());
        } else {
            const __global BVHNode childL = node->getLeftChild();
            const __global BVHNode childR = node->getRightChild();
            traverseRecursive(queryAABB, queryObjectIdx,
                              childL, list);
            traverseRecursive(queryAABB, queryObjectIdx,
                              childR, list);
        }
    }
}
```

Рекурсивный подход

Code path divergence.

Т.к. каждый поток самостоятельно принимает решение о том завершить ли обход текущего поддерева или спускаться в детей. И поэтому мало шансов что потоки согласованно будут углубляться в рекурсию.

Можно явно развернуть стек.

```
BVHNode* stack[MAX_STACK_SIZE];
BVHNode* node = bvhRoot;
do {
    BVHNode* childL = node->getLeftChild();
    BVHNode* childR = node->getRightChild();
    bool overlapL = checkOverlap(queryAABB, childL->getAABB());
    bool overlapR = checkOverlap(queryAABB, childR->getAABB());

    if (overlapL && childL->isLeaf())
        list.add(queryObjectIdx, childL->getObjectIdx());
    if (overlapR && childR->isLeaf())
        list.add(queryObjectIdx, childR->getObjectIdx());

    if (!traverseL && !traverseR) {
        node = stack.pop();
    } else {
        node = traverseL ? childL : childR;
        if (traverseL && traverseR)
            stack.push(childR);
    }
} while (node != NULL);
```

Подход с явным стеком

Code divergence присутствует в виде числа итераций цикла.

Data divergence увеличивает количество запрашиваемых кеш-линий.

Но в нашем случае объекты с которыми выполняется пересечение - сами являются листьями.

Вместо запуска потоков для обработки объектов в случайном порядке - можно запускать их в порядке обхода листьев.

Подход с явным стеком

1) Объекты будут проверять коллизии с самими собой.
2) Каждая пара проверяется дважды, т.е. делается в два раза больше работы.

Вводим порядок - пара проверяется только если первый элемент меньше второго.

И тогда можно сэкономить половину работы, сравнивая самый большой элемент в поддереве левого и правого ребенка с объектом-листом запроса:

```
if (node->getRightmostLeafInLeftSubtree() <= queryLeaf)
    overlapL = false;
if (node->getRightmostLeafInRightSubtree() <= queryLeaf)
    overlapR = false;
```

Создание BVH на GPU

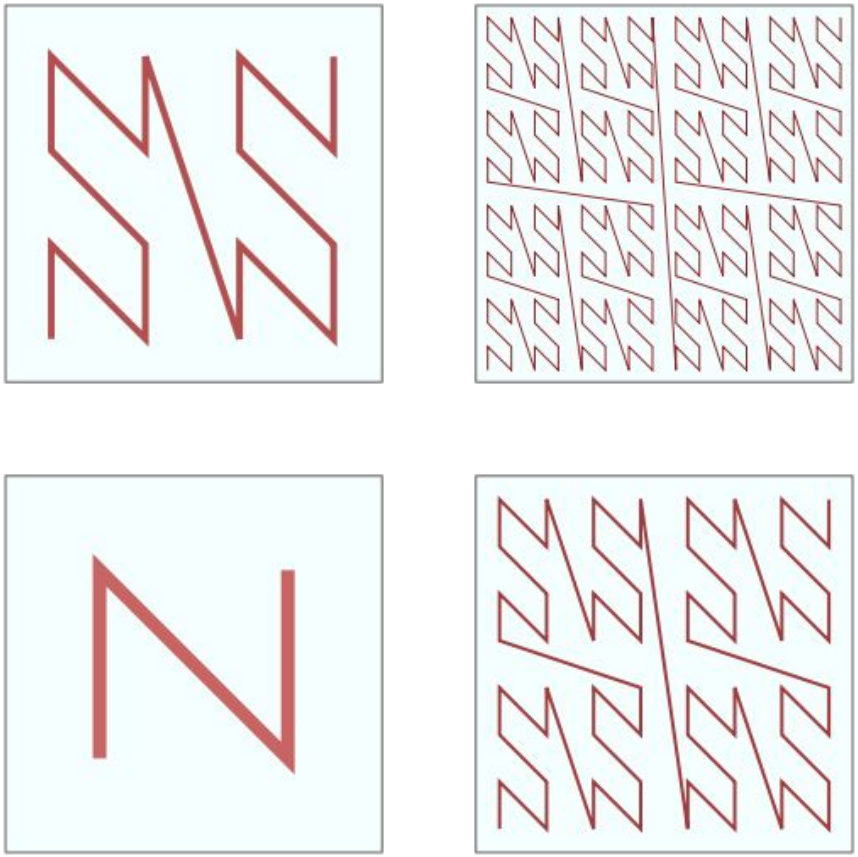
Объекты могут двигаться.

Можно сохранять структуру дерева, а обновлять лишь AABB во всех узлах. Но тогда в худшем случае все будет неконтролируемо плохо.

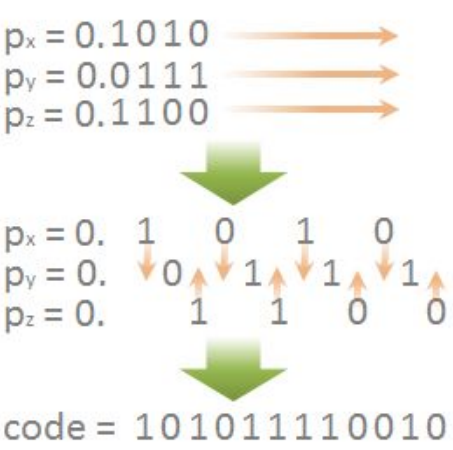
Поэтому хочется динамически строить BVH с чистого листа каждый момент времени.

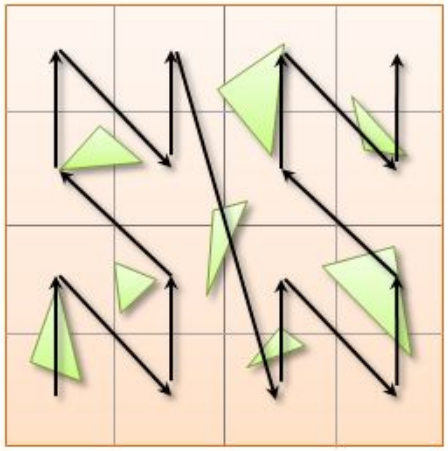
Linear BVH (LBVH)

Упорядочим все пространство в Z-порядок (кривой Мортона):



Morton code





```
// Дополняем 10-битное число до 30 бит
// вставкой двух нулей после каждого бита
unsigned int expandBits(unsigned int v) {
    v = (v * 0x00010001u) & 0xFF0000FFu;
    v = (v * 0x00000101u) & 0x0F00F00Fu;
    v = (v * 0x00000011u) & 0xC30C30C3u;
    v = (v * 0x00000005u) & 0x49249249u;
    return v;
}
// Вычисляем 30-битный код Мортона для
// данной 3D точки расположенной в единичном кубе
unsigned int morton3D(float x, float y, float z) {
    x = min(max(x * 1024.0f, 0.0f), 1023.0f);
    y = min(max(y * 1024.0f, 0.0f), 1023.0f);
    z = min(max(z * 1024.0f, 0.0f), 1023.0f);
    unsigned int xx = expandBits((unsigned int)x);
    unsigned int yy = expandBits((unsigned int)y);
    unsigned int zz = expandBits((unsigned int)z);
    return xx * 4 + yy * 2 + zz;
}
```

Собираем дерево из объектов

Сортируем все объекты через их коды Мортона через radix sort.

Теперь на будущих листьях дерева введен порядок.

А это значит что нам осталось лишь научиться по подотрезку элементов находить хороший индекс подразбиения на два подотрезка.

Собираем дерево из объектов

```
BVHNode generateHierarchy(
        const __global unsigned int* sortedMortonCodes,
        const __global unsigned int* sortedObjectIDs,
        unsigned int first, unsigned int last) {
    if (first == last)
        return LeafNode(sortedObjectIDs[first]);

    unsigned int split = findSplit(sortedMortonCodes, first, last);

    BVHNode childA = generateHierarchy(sortedMortonCodes, sortedObjectIDs,
                                       first, split);
    BVHNode childB = generateHierarchy(sortedMortonCodes, sortedObjectIDs,
                                       split + 1, last);
    return InternalNode(childA, childB);
}
```

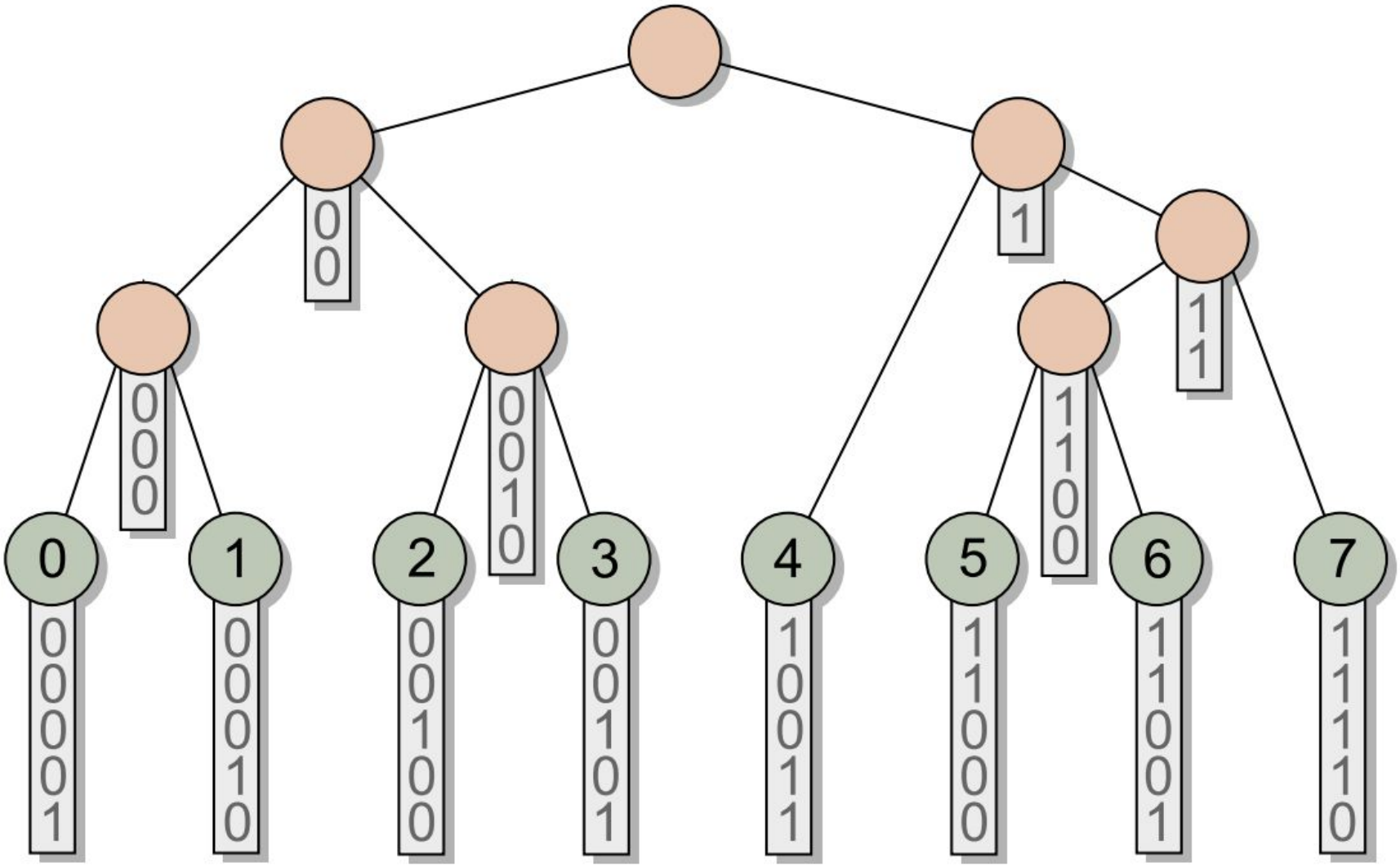
Как выбрать метод разбиения findSplit?

Разбивать по самому старшему не-одинаковому в подотрезке биту, т.е. чтобы в первом подотрезке этот бит был ноль, а во втором - единица.

Т.е. по сути рассечь множество axis aligned плоскостью.

Нам понадобиться считать количество лидирующих нулей (чтобы найти индекс первого различающегося бита). Для этого есть функция clz (count leading zeros).

Как выбрать метод разбиения findSplit?



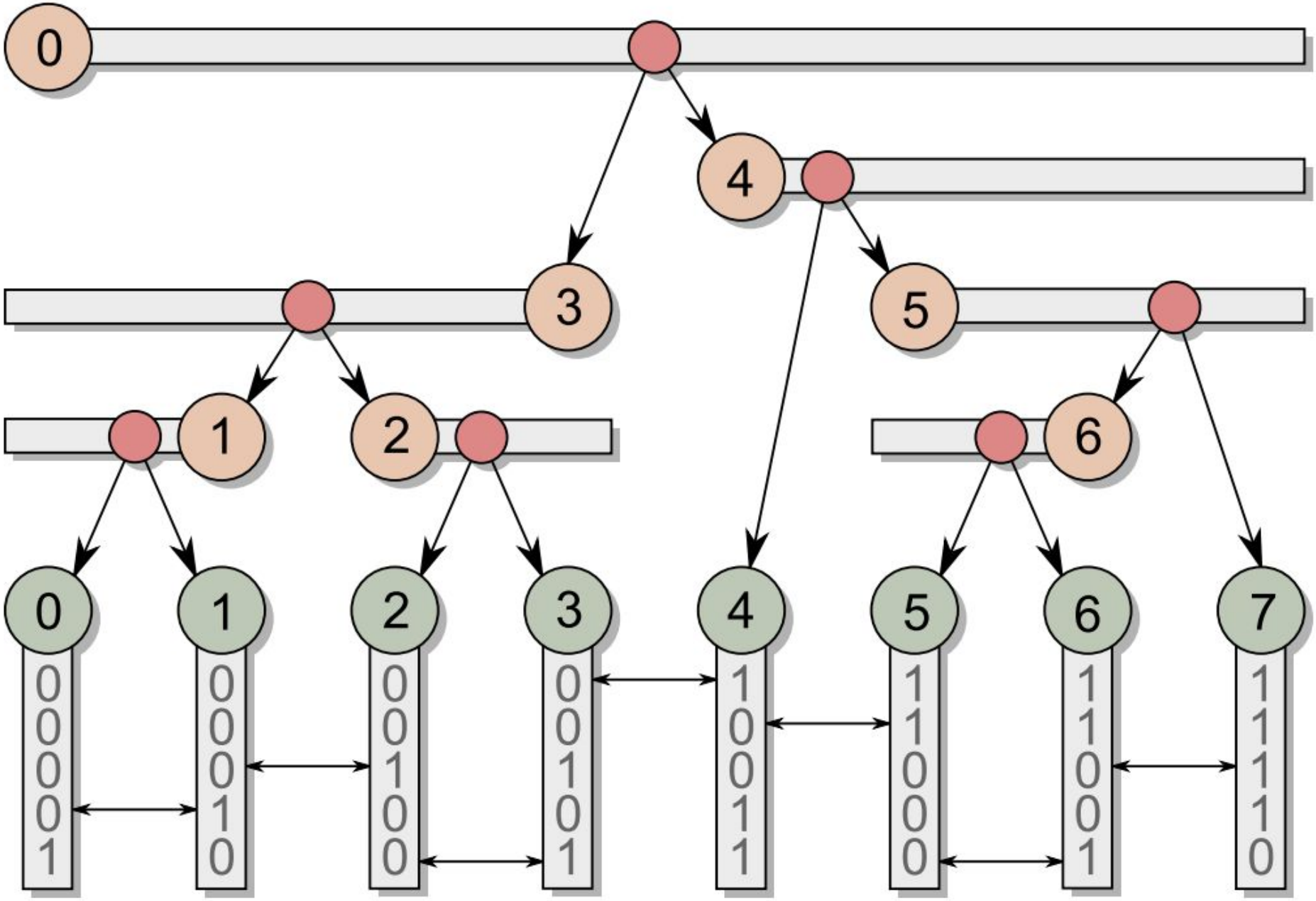
```
int findSplit(... sortedMortonCodes, first, last) {
    int firstCode = sortedMortonCodes[first];
    int lastCode  = sortedMortonCodes[last];
    // Количество старших бит одинаковых у всех
    int commonPrefix = clz(firstCode, lastCode);
    int split = first;
    int step = last - first;
    // Ищем наибольший элемент одинаковый с первым элементом
    // больше чем в commonPrefix битах
    do {
        step = (step + 1) / 2;
        int newSplit = split + step;
        if (newSplit < last) {
            int splitCode = sortedMortonCodes[newSplit];
            int splitPrefix = clz(firstCode ^ splitCode);
            if (splitPrefix > commonPrefix)
                split = newSplit;
        }
    } while (step > 1);
    return split;
}
```

Как эффективно распараллелить findSplit?

Но такой рекурсивный алгоритм плохо ложится на модель массового параллелизма, т.к. Последующие findSplit нельзя начать считать пока не станет известен их диапазон для работы. А на неглубоких уровнях дерева параллелизма практически нет.

Для массового параллелизма нужно избавиться от зависимости более глубоких уровней от диапазона подотрезка их родителей.

Т.е. для каждого узла хочется сразу угадать какой у него диапазон работы.


0
4
3
5
1
2
6
0
1
2
3
4
5
6
7
00001
00010
00100
00101
10011
11000
11001
11110

Как восстановить узлы?

1) Заметим что если листьев **N**, то внутренних узлов **N-1**.

2) Номер узла - индекс его самого левого/правого листа (который является точкой рассечения родителя узла).

3) Каждый лист (кроме крайних) является рассекателем не больше одного раза, а значит ровно один раз (см. первое и второе свойство).

Как восстановить узлы?

1) Направление **d** которое узел покрывает относительна листа рассекателя определяется по листу рассекателю и его соседям: **k(i-1), k(i), k(i+1)**.
(в ту сторону, в которой наибольший общий префикс)

2) Т.е. **k(i)** и **k(i+d)** лежат в конструирующемся узле, а **k(i-d)** в узле-соседнем ребенке.

3) Можно найти свою длину бинарным поиском в направлении **d**.
(максимальная длина равна)

Определяем направление и свою вторую границу

1: **for each** internal node with index $i \in [0, n-2]$ **in parallel**
2: // Determine direction of the range (+1 or -1)
3: $d \leftarrow \text{sign}(\delta(i, i+1) - \delta(i, i-1))$
4: // Compute upper bound for the length of the range
5: $\delta_{\text{min}} \leftarrow \delta(i, i-d)$
6: $l_{\text{max}} \leftarrow 2$
7: **while** $\delta(i, i + l_{\text{max}} \cdot d) > \delta_{\text{min}}$ **do**
8: $l_{\text{max}} \leftarrow l_{\text{max}} \cdot 2$
9: // Find the other end using binary search
10: $l \leftarrow 0$
11: **for** $t \leftarrow \{l_{\text{max}}/2, l_{\text{max}}/4, \ldots, 1\}$ **do**
12: **if** $\delta(i, i + (l+t) \cdot d) > \delta_{\text{min}}$ **then**
13: $l \leftarrow l + t$
14: $j \leftarrow i + l \cdot d$

Находим рассечение нашего подотрезка

15: // Find the split position using binary search
16: $\delta_{\text{node}} \leftarrow \delta(i, j)$
17: $s \leftarrow 0$
18: **for** $t \leftarrow \{\lceil l/2 \rceil, \lceil l/4 \rceil, \ldots, 1\}$ **do**
19: **if** $\delta(i, i + (s+t) \cdot d) > \delta_{\text{node}}$ **then**
20: $s \leftarrow s + t$
21: $\gamma \leftarrow i + s \cdot d + \min(d, 0)$
22: // Output child pointers
23: **if** $\min(i, j) = \gamma$ **then** left $\leftarrow L_\gamma$ **else** left $\leftarrow I_\gamma$
24: **if** $\max(i, j) = \gamma + 1$ **then** right $\leftarrow L_{\gamma+1}$ **else** right $\leftarrow I_{\gamma+1}$
25: $I_i \leftarrow (\text{left}, \text{right})$
26: **end for**

Ссылки

- https://en.wikipedia.org/wiki/Bitonic_sorter
- https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html
- https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html
- https://devblogs.nvidia.com/thinking-parallel-part-i-collision-detection-gpu/
- https://devblogs.nvidia.com/thinking-parallel-part-ii-tree-traversal-gpu/
- https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/
- Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees