УДК 004.422.639 doi: 10.15827/0236-235X.142.398-413 2023. Т. 36. № 3. С. 398–413

Исследование свойств конкурентного двоичного дерева

В.Г. Грачев
А.С. Ушаков

Ссылка для цитирования
Грачев В.Г., Ушаков А.С. Исследование свойств конкурентного двоичного дерева // Программные продукты и системы. 2023. Т. 36. № 3. С. 398–413. doi: 10.15827/0236-235X.142.398-413
Информация о статье
Поступила в редакцию: 10.05.2023 После доработки: 05.06.2023 Принята к публикации: 30.06.2023

Аннотация. В разрабатываемом программном обеспечении навигационных тренажеров «Система моделирования морских операций» реализована технология автоматического построения трехмерной модели глобального рельефа на базе алгоритма рекурсивного деления с локально адаптивным управлением детализацией. Алгоритм реализован полностью на графическом адаптере (GPU) и использует возможности массированной параллельной обработки данных в вычислительных шейдерах. Алгоритм рекурсивного деления строится на использовании двоичного дерева, однако классические двоичные деревья, явно описанные с использованием указателей, неприменимы для GPU-реализаций из-за архитектурных особенностей. Для обеспечения возможности параллельного выполнения алгоритма применена специализированная параллельная структура данных – конкурентное двоичное дерево. В статье описаны предпосылки к созданию структуры конкурентного двоичного дерева и рассматриваются этапы его построения от использования неявного двоичного дерева до представления в виде бинарного поля с последующим дополнением его редуцированной суммой листовых узлов. Теоретически обоснован объем оперативной памяти, необходимый для размещения конкурентного двоичного дерева заданной глубины. Выполнен анализ алгоритмической сложности построения конкурентного двоичного дерева и итерации по его листовым узлам. Приведены и проанализированы результаты сравнительных синтетических тестов производительности конкурентного двоичного дерева, выполненные на CPU и GPU, а также результаты его практического применения в системе моделирования морских операций.

Ключевые слова: конкурентное двоичное дерево, параллельная структура данных, алгоритм рекурсивного деления, оптимизация объема памяти, массированный GPU-параллелизм, система моделирования морских операций

В разрабатываемом перспективном отечественном кроссплатформенном ПО навигационных тренажеров для практической подготовки специалистов ВМФ «*Система моделирования морских операций*» (СММО) [1] принята парадигма глобального района тренировки, которая подразумевает возможность проведения тренировок в любом районе земного шара [2]. При этом конкретный географический район может быть детализирован до необходимой степени визуального соответствия реальному. Основной составной частью любого района тренировки, в том числе и глобального, является массив суши, на котором располагаются другие объекты – строения, причальное оборудование, навигационные знаки и др. Если конкретные районы тренировок детализируются, как правило, вручную с использованием труда дизайнеров, то для глобального района, то есть для всего остального пространства земной поверхности вне конкретных районов, ввиду его обширной протяженности необходимо реализовать процесс автоматического или автоматизированного создания и отображения недетализированных массивов суши.

Эта задача решается в СММО при помощи алгоритма локально адаптивного управления детализацией протяженного рельефа на базе рекурсивного деления с использованием вычислительных возможностей современных графических адаптеров (GPU). Практическая реализация этого алгоритма требует использования специализированного представления данных, допускающих их параллельную обработку.

Структурирование данных для алгоритмов рекурсивного деления

Алгоритмы рекурсивного деления демонстрируют быстрый рост вычислительной сложности в зависимости от глубины рекурсии. Из-за фундаментально экспоненциального характера этих алгоритмов вычислительные затраты с увеличением глубины подразделения могут быстро стать ограничением. Один из путей амортизации возрастающей вычислительной сложности – выполнять рекурсивное деление не только адаптивно, но и параллельно. Хотя адаптивное подразделение легко реализовать последовательно, в общем случае совместить его с параллельной обработкой непросто. Решить эту проблему можно путем внедрения структуры данных, подходящей для параллельной обработки.

Канонический алгоритм рекурсивного деления строится на использовании двоичного дерева, количество узлов которого удваивается на каждом шаге рекурсии. Листовые узлы дерева формируют его изменяющуюся топологию. Если добиться возможности параллельной обработки листовых узлов двоичного дерева, то можно ускорить любой алгоритм подразделения.

В технической литературе описаны способы параллельного построения BVH-деревьев. В работе [3] представлен алгоритм построения иерархий ограничивающих объемов на многоядерных GPU. Авторы [4] устранили для этого алгоритма необходимость синхронизации данных между CPU и GPU, а автор [5] улучшил масштабируемость построения дерева в целом, тем самым максимизировав параллелизм. В работе [6] предложено повышение утилизации GPU путем кодирования листовых узлов Мортон-кодами для кривой Z-порядка. Однако специфические структуры данных BVH-деревьев неприменимы в алгоритмах рекурсивного деления, так как не обеспечивают изменение своей топологии с течением времени.

Классическая реализация двоичного дерева использует указатели для хранения связей между его узлами. Однако архитектура GPU не предусматривает работу с ними.

Известно, что *конкурентное двоичное дерево* (КДД) – это структура данных, предназначенная для построения и параллельного обновления топологии полного двоичного дерева, в том числе с использованием возможностей GPU [7, 8]. В оригинальной статье [7] КДД описаны постфактум без рассмотрения этапов создания и строгого математического анализа их свойств. В настоящей работе теоретически исследованы все этапы построения КДД и алгоритмы его использования.

Совершенные двоичные деревья как двоичная куча

Свойства и определения. Двоичное дерево, в котором все внутренние узлы имеют по два дочерних узла, а все листовые находятся на одном уровне, формирует так называемое совершенное двоичное дерево. В нем суммарно $2^{D+1} - 1$ узлов, где $D \geq 0$ – уровень листовых узлов (глубина дерева). На рисунке 1 представлена структура совершенного двоичного дерева глубиной $D = 4$.

Двоичное дерево тесно связано с таким понятием, как двоичная куча – полным двоичным деревом, в котором значение в любом узле не меньше, чем значения его потомков, а глубина всех узлов отличается не более чем на единицу. Топология совершенного двоичного дерева может быть представлена в виде двоичной кучи. Индексация двоичной кучи выполняется по следующему принципу: корневому узлу дерева присваивается индекс 1, и далее выполняется обход дерева в ширину с инкрементальным увеличением индексов узлов.

Алгебраические зависимости. Такой подход к индексации обладает следующими свой-

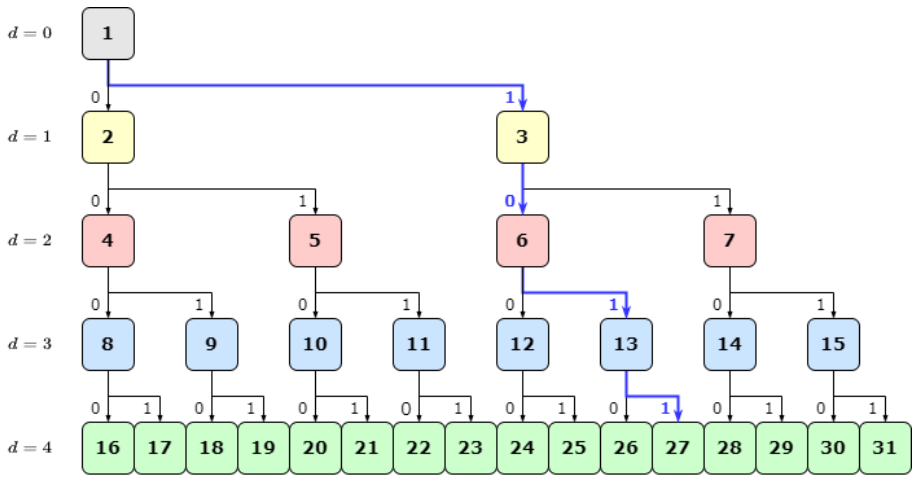


Рис. 1. Совершенное двоичное дерево глубиной D = 4 (в узлах дерева указаны их индексы) и неявная кодировка пути от корневого узла к узлу 27

Fig. 1. A perfect binary tree with a depth D = 4 (tree node indexes are shown) and implicit path encoding from a root node to node 27

ствами. Для узла дерева с индексом $k \geq 1$ индекс его родительского узла k_p, левого k_L и правого k_R потомков будут следующими:

$$k_P = \lfloor k/2 \rfloor,$$
$$k_L = 2k, \quad (1)$$
$$k_R = 2k+1.$$

Для узлов дерева на уровне $d \geq 0$ диапазон их индексов будет находиться в промежутке $[2^d, 2^{d+1} - 1]$.

Из индекса узла можно извлечь дополнительную информацию – номер уровня дерева, на котором узел находится:

$$d_k = \lfloor \log_2(k) \rfloor. \quad (2)$$

Для машинного двоичного представления положительного целого числа эта операция по результату эквивалентна нахождению в нем позиции старшего значащего бита:

$$\lfloor \log_2(k) \rfloor = FindMSB(k),$$

где *FindMSB* – функция, возвращающая позицию старшего значащего бита (справа налево, считая крайнюю правую позицию нулевой).

Неявная кодировка пути к узлу. Индекс узла двоичного дерева неявно кодирует путь к нему. Бинарное представление любого индекса двоичной кучи содержит весь путь от корневого узла до узла с этим индексом. При этом ноль обозначает переход от узла к его левому потомку, а единица – к правому. Путь предваряется установленным в единицу старшим битом, причем позиция бита равна уровню дерева, на котором находится этот узел (2). Например, узел с индексом 27 имеет бинарное представление индекса 11011. Смещение старшего бита $d_{27} = 4$, а путь к узлу 1011 или «направо-налево-направо-направо» от корневого узла (рис. 1).

Неявная двоичная куча. Двоичная куча может быть эффективно закодирована с использованием неявной структуры данных, в которой структурная информация неявно определяется способом хранения данных, а не прямым образом через указатели [9]. Для случая двоичной кучи это одномерный массив, содержащий значения узлов дерева. Размер этого массива фиксированный и равен 2^{D+1}. В первом элементе массива с индексом 0 хранится глубина дерева D. Оставшиеся элементы массива ассоциированы с каждым узлом дерева в соответствии с индексом. Корневой узел дерева хранится в элементе массива с индексом 1, за ним следуют два узла первого уровня, затем четыре узла второго уровня и т.д. На рисунке 2 совершенное двоичное дерево глубиной $D = 4$ закодировано неявным образом в одномерный массив.

Благодаря такому представлению двоичное дерево может быть размещено в непрерывной области оперативной памяти с тривиальной итерацией по его узлам. Отсутствие в используемой неявной структуре данных указателей позволяет реализовать алгоритмы работы с ней на GPU.

Фундаментальный недостаток двоичной кучи состоит в том, что ее применение ограничено совершенными двоичными деревьями. Несмотря на то, что можно изменять значения любого узла двоичного дерева, представленного в виде двоичной кучи, топология такого дерева остается неизменной.

КДД предоставляет возможность обойти это ограничение путем представления произвольного двоичного дерева без использования указателей таким образом, что топология этого дерева может изменяться после его первичного формирования.

Структура КДД

Представление бинарных деревьев в виде битового поля. ***Создание битового поля***. Двоичное дерево глубиной D может быть представлено в виде битового поля размером 2^D, содержащего по одному биту для каждого узла двоичного дерева на максимальном уровне $d = D$. На рисунке 3 представлены произвольное двоичное дерево глубиной $D = 4$ и соответствующее ему битовое поле.

В таком битовом поле каждый бит, установленный в единицу, обозначает листовой узел

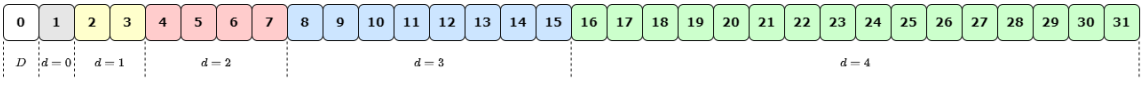


Рис. 2. Представление совершенного двоичного дерева глубиной D = 4 в виде одномерного массива (в ячейках массива указаны индексы узлов дерева)

Fig. 2. A perfect binary tree with a depth D = 4 as 1D-array (tree node indexes are shown in array elements)

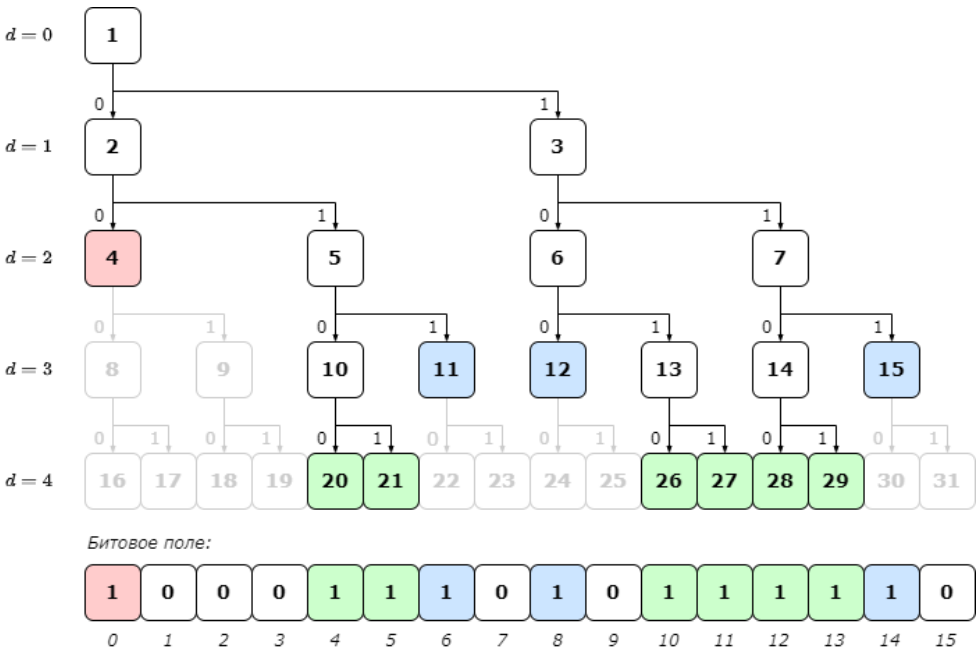


Рис. 3. Произвольное двоичное дерево глубиной D = 4 (в узлах дерева указаны их индексы) и его представление в виде битового поля (внизу)

Fig. 3. An arbitrary binary tree with a depth D = 4 (top, tree node indexes are shown) and its representation as binary field (bottom)

дерева. При этом уровень листового узла определяется количеством нулей, следующих за соответствующим ему битом.

Раскодирование узлов дерева из битового поля представляет собой процесс получения индексов этих узлов $k \in [1, 2^{D+1} - 1]$. Бинарное представление индекса каждого узла двоичного дерева неявно кодирует путь к нему – таким образом достигается доступ ко всей топологии двоичного дерева.

Индекс узла дерева из индекса бита. Существует однозначное преобразование индекса бита x в битовом поле, кодирующем листовой узел произвольного двоичного дерева глубиной D, в индекс этого узла k.

Рассмотрим бит, находящийся в битовом поле по индексу $x \in [0, 2^D - 1]$. Если бит имеет значение ноль, то соответствующий ему узел дерева не является листовым и не существует в топологии дерева. Когда значение бита равно единице, он может кодировать до N_x листовых узлов дерева.

Для узлов дерева на уровне d соответствующие им биты в битовом поле располагаются через каждые 2^{D-d} позиций. Следовательно, в зависимости от позиции бит может кодировать несколько узлов дерева. Если последовательно рассматривать битовое поле справа налево, то можно увидеть, что каждый бит кодирует узел уровня $d = D$, каждый второй бит дополнительно кодирует узел на уровне $d = D - 1$, каждый четвертый бит дополнительно к ним кодирует узел на уровне $d = D - 2$ и т.д. Таким образом, один и тот же бит x битового поля может кодировать до $N_x = D + 1$ узлов дерева.

Для определения значения N_x для бита с индексом x необходимо найти максимальное значение $n \in [0, D]$, при котором выполняется условие

$$(2^D - x) \bmod 2^n = 0, \quad (3)$$

то есть определить максимальную степень двойки, на которую без остатка делится дополнение индекса бита до размера битового поля. Тогда количество кодируемых битом узлов дерева будет

$$N_x = n + 1. \quad (4)$$

Учитывая, что размер битового поля представляет собой степень двойки, при нахождении остатка от деления индекса бита на ту или иную степень двойки можно в (3) вместо дополнения индекса до размера битового поля использовать непосредственно сам индекс:

$$x \bmod 2^n = 0. \quad (5)$$

Если рассматривать двоичное машинное представление положительного целочислен-

ного индекса бита x, то максимальное значение n в (5) будет соответствовать позиции младшего значащего бита. Тогда

$$N_x = \begin{cases} D+1, & x = 0, \\ FindLSB(x)+1, & x > 0, \end{cases} \quad (6)$$

где $FindLSB$ – функция, возвращающая позицию младшего значащего бита (справа налево, считая крайнюю правую позицию нулевой).

Индексы N_x узлов дерева, кодируемых битом x, определяются как

$$\{k_1,\ldots,k_{N_x}\} = (2^D + x)2^{-n_x}, \quad n_x \in [0, N_x - 1]. \quad (7)$$

Множество $\{k_1,\ldots,k_{N_x}\}$ содержит индексы узлов, находящихся в зависимости «потомок–родитель», начиная с узла на уровне $d_k = D$ до узла на уровне

$$d_k = D - (N_x - 1).$$

С учетом (4)

$$d_k = D - (n_x + 1 - 1) = D - n_x. \quad (8)$$

Для приведенного на рисунке 3 примера бит с индексом 4 имеет машинное представление 100. Наименьший значащий бит находится на позиции 2. Следовательно, в соответствии с (6) этот бит может кодировать три узла дерева, находящиеся на различных уровнях. Индексы узлов определяются по (7) и равны {20, 10, 5}. Аналогично биты с индексами 0 и 14 могут соответственно кодировать до пяти и до двух узлов с индексами {16, 8, 4, 2, 1} и {30, 15}.

Чтобы определить, какой именно узел дерева из допустимого диапазона индексов узлов кодирует бит, необходимо установить уровень дерева, на котором располагается кодируемый узел.

Если узел k на уровне дерева d_k является листовым и кодируется единичным битом с индексом x_k, то следующий единичный бит $x_{k'}$ в битовом поле будет находиться на позиции

$$x_{k'} = x_k + 2^{D-d_k}$$

Отсюда следует, что в битовом поле за единичным битом, кодирующим листовой узел на уровне дерева d_k, будут следовать нулевые биты в количестве, равном $N_0 = 2^{D-d_k} - 1$.

Тогда

$$N_0 + 1 = 2^{D-d},$$

$$\log_2(N_0 + 1) = D - d_k$$

и уровень кодируемого узла в двоичном дереве определяется как

$$d_k = D - \log_2(N_0 + 1). \quad (9)$$

Из (8) выразим

$$n_x = D - d_k.$$

С учетом этого выражение (7) преобразуется:

$$k_x = (2^D + x)2^{-(D-d_k)}. \quad (10)$$

Принимая во внимание (9), индекс узла дерева, кодируемый битом x, можно определить как

$$k_x = (2^D + x)2^{-\log_2(N_{0x}+1)}.$$

Так как каждый бит в битовом поле может кодировать только по одному узлу на каждом уровне дерева, битовое поле однозначно отображает индексы битов в индексы узлов двоичного дерева произвольной топологии.

Индекс бита из индекса узла дерева. Для любого узла дерева k можно определить кодирующий его бит, расположенный в битовом поле на позиции x_k.

Решая уравнение (10) относительно x с учетом известного из (2) d_k, получаем

$$x_k = k2^{D-d_k} - 2^D.$$

Смысл этого выражения в следующем. Так как бит x в битовом поле может кодировать N_x узлов, требуется привести индекс узла k к максимальному из значений множества индексов узлов, которые могут быть закодированы соответствующим битом. Для этого необходимо рекурсивно вычислять индекс левого потомка узла k вплоть до последнего уровня дерева $d = D$, на котором для кодирования узла требуется один бит. Это можно сделать, последовательно использовав выражение (1) $D - d_k$ раз. Затем полученный индекс следует уменьшить на индекс первого узла дерева на уровне D, то есть на 2^D.

Для приведенного на рисунке 3 примера узел с индексом 5 имеет самого глубокого потомка на уровне D с индексом 20. Этот индекс необходимо уменьшить на индекс первого узла на уровне D, то есть на 16. Соответствующий узлу бит будет иметь индекс 4.

Изменение топологии дерева. Представление двоичного дерева в виде битового поля позволяет реализовать операции разделения и слияния узлов дерева напрямую.

Разделение заданного узла двоичного дерева с индексом k на левый и правый потомки означает установку в единицу бита битового поля, соответствующего правому потомку этого узла с индексом $2k + 1$.

Правый и левый потомки одного родительского узла с индексом k могут быть объединены в родительский узел. Эта операция означает установку в ноль бита битового поля, со-

ответствующего правому потомку родительского узла с индексом $2k + 1$.

Пример изменения топологии двоичного дерева и его представления в виде битового поля в результате операций слияния и разделения листовых узлов показан на рисунке 4.

Этот метод инвариантен к типу обрабатываемого узла. Допускается разделение нелистового узла. В этом случае для бита, соответствующего правому потомку такого узла, будет повторно установлено значение единица. При слиянии несуществующих узлов для бита, соответствующего несуществующему узлу, будет повторно установлено значение ноль.

Итерация по листовым узлам. Для получения из битового поля всех индексов листовых узлов дерева необходимо последовательно проверить значения всех его битов. Сложность такого алгоритма составляет $O(2^D)$ [10]. Она не зависит от количества листовых узлов $L \in [1, 2^D]$ в закодированном двоичном дереве. Блок-схема алгоритма представлена на рисунке 5.

Проверка значений битов в битовом поле должна выполняться последовательно, начиная с нулевого индекса, что исключает возможность параллельного поиска индексов листовых узлов.

Тесты производительности декодирования битового поля. Для сравнительной оценки производительности структуры КДД на CPU и GPU была выполнена его программная реализация на языках C++ и GLSL [11].

Тесты выполнялись на ПК, оснащенном центральным процессором Intel Core i7-11700 и графическим адаптером NVIDIA 3080 RTX с 8704 универсальными процессорами, под управлением Windows 10.

Замеры производительности декодирования битового поля для КДД глубиной от 10 до 30 приведены на рисунке 6.

Архитектурные особенности GPU, выполнение алгоритма с использованием одного потока, а также эффективное задействование кеша центрального процессора и упреждающей выборки данных при последовательном обращении к линейному участку памяти битового поля приводят к существенному преимуществу CPU перед GPU [12]. Разница во времени декодирования составляет примерно 50 раз.

Полученные результаты существенно ограничивают практическое применение метода итерации по листовым узлам дерева путем декодирования битового поля.

Дополнительным ограничивающим фактором является необходимость отдельно от самого КДД хранить получаемые индексы декодированных листовых узлов для дальнейшего использования в алгоритме рекурсивного деления. Максимальное количество листовых узлов КДД равно 2^D, номер каждого узла на GPU хранится в переменной типа uint, занимающей в памяти четыре байта (32 бита), то есть для записи всех возможных индексов листовых узлов необходимо дополнительно выделить $4(2^D)$ байт, что в 32 раза больше, чем объем памяти самого битового поля. Это контрпродуктивно в условиях существующих ограниченных ресурсов видеопамяти.

Дополнение битового поля суммой листовых узлов. *Редуцированная сумма листовых*

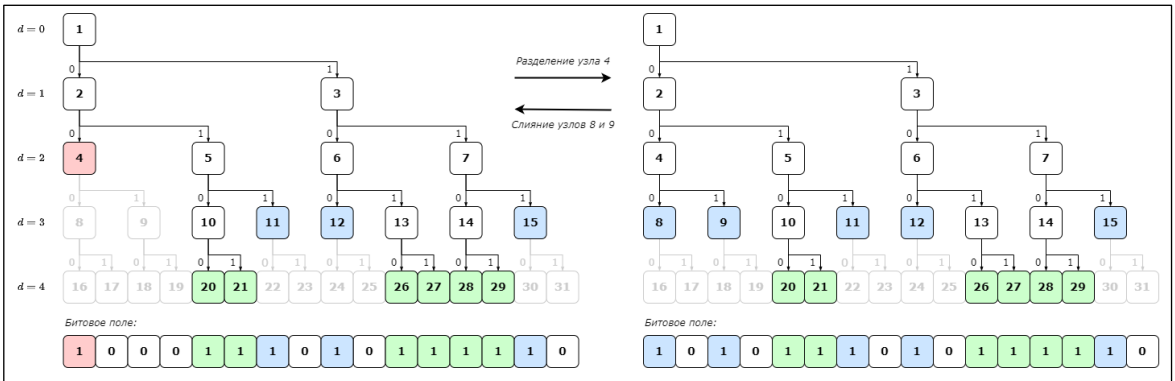


Рис. 4. Изменение битового поля при операциях разделения (слева направо) и слияния (справа налево) узлов двоичного дерева (в ячейках указаны значения редуцированной суммы листовых узлов, в верхнем левом углу ячеек – индексы узлов)

Fig. 4. Bitfield variation during binary tree nodes split (left-to-right) and merge (right-to-left) operations. Leaf nodes sum reduction values are shown in elements and node indexes are shown in elements left top corners

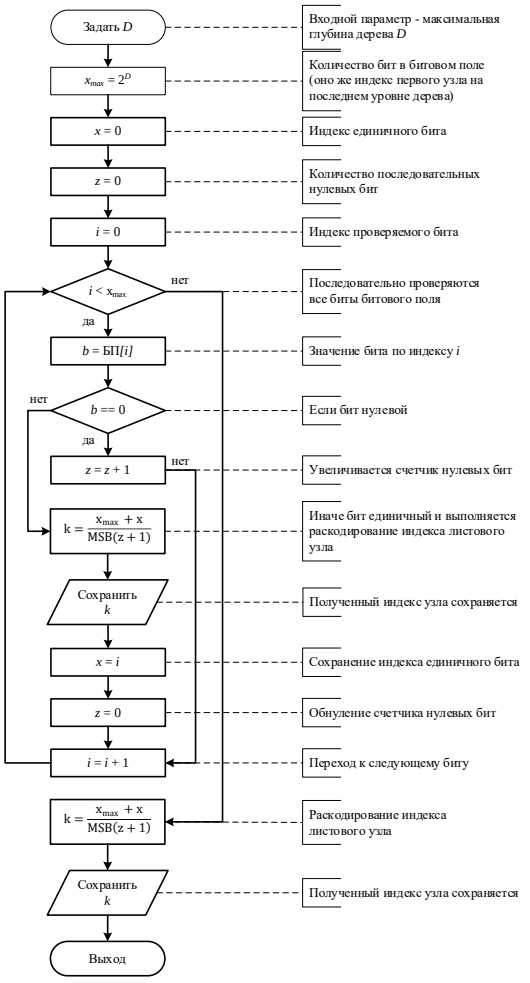
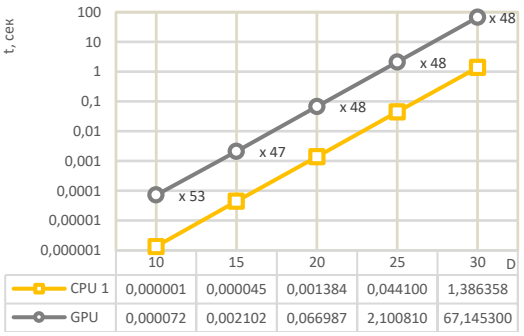


Рис. 5. Блок-схема алгоритма поиска индексов листовых узлов двоичного дерева путем декодирования битового поля

Fig. 5. A block diagram of a bitfield decoding algorithm for findumn binary tree leaf node indexes



	10	15	20	25	30
CPU 1	0,000001	0,000045	0,001384	0,044100	1,386358
GPU	0,000072	0,002102	0,066987	2,100810	67,145300

Рис. 6. Время декодирования битового поля КДД в зависимости от его глубины

Fig. 6. Concurrent binary tree bitfield decoding time depending on the tree depth

узлов. Представление двоичного дерева в виде битового поля позволяет полностью закодировать топологию дерева и выполнять операции разделения и слияния узлов. Тем не менее битовому полю недостает эффективных методов итерации по закодированным в нем узлам дерева. Эта возможность особенно важна для больших битовых полей, так как их размер экспоненциально растет с ростом глубины двоичного дерева и простой однопоточный последовательный перебор битов на GPU будет выполняться за время, недопустимо большое при практическом использовании.

Для обеспечения эффективной итерации по листовым узлам двоичного дерева представляющее это дерево битовое поле дополняется его редуцированной суммой. Этот процесс может быть рассмотрен как построение снизу вверх совершенного двоичного дерева глубиной D, в котором каждый листовой узел соответствует биту битового поля (рис. 7).

По значению редуцированной суммы левого и правого потомков узла такого дерева можно определить, какое количество листовых узлов содержится в левом и правом поддеревьях узла.

В свою очередь, такое совершенное дерево может быть представлено в виде двоичной кучи, в которой значения узлов хранятся в порядке обхода дерева в ширину начиная от корневого узла. Эта двоичная куча, включая битовое поле, являющееся его последним уровнем, представляет собой структуру данных КДД.

В элементе двоичной кучи с индексом 0 хранится глубина КДД D. Значение глубины КДД принимается в момент его построения и в дальнейшем не может быть изменено.

Второй элемент кучи с индексом 1 хранит общее количество листовых узлов дерева $L \in [1, 2^D]$, полученное в результате вычисления редукции суммы. Это свойство структуры КДД является отправным для выполнения итераций по листовым узлам дерева.

Индекс листового узла дерева кучи по его порядковому номеру. Индекс узла КДД равен индексу соответствующего ему элемента двоичной кучи. В КДД, содержащем L листовых узлов, для каждого l-го листового узла, где $l \in [0, L-1]$, можно определить индекс соответствующего ему элемента двоичной кучи.

Алгоритм поиска индекса узла представляет собой рекурсивный переход от корневого узла дерева к его левому или правому поддереву в зависимости о того, превышает ли по-

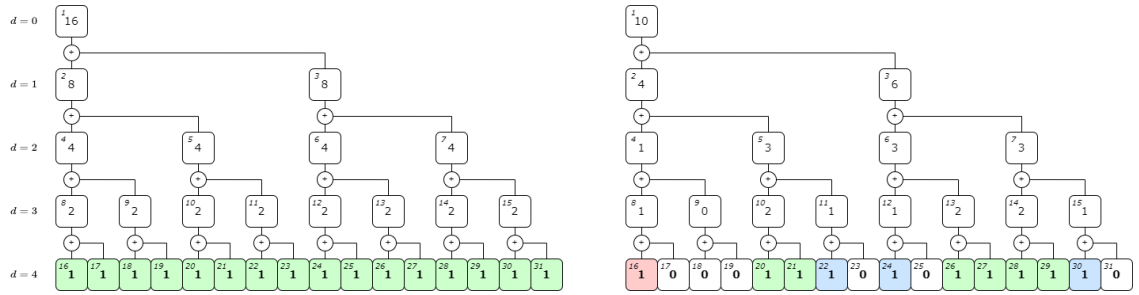


Рис. 7. Битовое поле совершенного (слева) и произвольного (справа) двоичного дерева глубиной D = 4, дополненное редуцированной суммой листовых узлов (в ячейках указаны значения редуцированной суммы листовых узлов, в верхнем левом углу ячеек – индексы узлов)

Fig. 7. A bitfield of a perfect (left) and arbitrary (right) binary tree with a depth D = 4 augmented with leaf nodes sum reduction. Leaf nodes sum reduction values are shown in elements and node indexes are shown in elements left top corners

рядковый номер листового узла количество листовых узлов в левом поддереве. Блок-схема алгоритма приведена на рисунке 8.

Итерация по листовым узлам. Сложность алгоритма поиска индекса одного листового узла КДД по его порядковому номеру в худшем случае составляет $O(D)$. При поиске всех листовых узлов совершенного дерева $L_D \in [1, 2^D]$ эта сложность возрастает до $O(DL_D) = O(D2^D)$.

Полученная оценка сложности алгоритма поиска всех листовых узлов КДД в D раз превосходит сложность поиска листовых узлов по бинарному дереву. Однако КДД является параллельной структурой данных и допускает использование до $P \in [1, L]$ потоков, одновременно выполняющих поиск индексов листовых узлов. С учетом этого сложность многопоточного алгоритма составит $O(D2^D / P)$. Для современных GPU $P >> D$.

Условием завершения алгоритма поиска индекса l-го листового узла является количество потомков k-го узла, меньшее или равное единице. В свою очередь, как было показано ранее, изменение топологии КДД приводит к изменению значения необходимого бита битового поля на ноль (в случае слияния узлов) либо на единицу (в случае разделения узла). При этом происходит неявное изменение значений сумм листовых узлов только на последнем уровне дерева $d = D$, где расположены биты битового поля, непосредственно кодирующие листовые узлы. Так как это значение для последнего уровня КДД не может превышать единицу, условие завершения алгоритма поиска листовых узлов продолжает выполняться. Это дает возможность одновременно с поиском листовых узлов выполнять их слияние или разделение, таким образом изменяя топологию дерева. К тому же отпадает необходимость промежуточного хранения индексов листовых узлов и выделения для этого дополнительного объема памяти.

Размещение структуры данных в памяти. *Потребный объем памяти КДД.* Определим минимальный размер памяти, необходимый для размещения КДД глубиной D. Для этого найдем верхнюю границу значения редуцированной суммы для узлов дерева на произвольном уровне d и, как следствие, минимальное количество бит для кодирования этого значения.

Любое значение узла КДД (редуцированная сумма его потомков, являющихся листовыми узлами) на глубине $d \in [0, D]$ требует не более $N_{b_d} = D - d + 1$ бит для записи.

На последнем уровне дерева в битовом поле признак листового узла кодируется одним битом. Каждые два смежных узла имеют общий родительский узел на предыдущем уровне, который может принять максимальное значение 2. Дальнейшая редукция суммы будет удваивать это значение для каждого предшествующего уровня двоичного дерева. Следовательно, максимальное значение редуцированной суммы листовых узлов на уровне d будет

$$S_{MAX_d} = 2^{D-d}.$$

Количество бит для записи значения S_{MAX_d} равно

$$\begin{aligned} N_{b_d} &= \left\lfloor \log_2 \left(S_{MAX_d} \right) \right\rfloor + 1, \\ N_{b_d} &= D - d + 1. \end{aligned} \qquad (11)$$

Количество узлов двоичного дерева на уровне $d \in [0, D]$ составляет

$$N_{l_d} = 2^d.$$

Общее количество памяти, необходимое для хранения узлов КДД глубиной D, будет равно сумме произведений количества узлов на каждом уровне на количество бит, необходимых для записи их значений:

$$N_b = \sum_{d=0}^{D} N_{l_d} N_{b_d}$$

или

$$N_b = \sum_{d=n}^{D} 2^d \left(D - d + 1\right). \quad (12)$$

Для определения этого значения найдем частную сумму числового ряда (12). Представим сумму ряда как

$$N_b = D\sum_{d=n}^{D} 2^d - \sum_{d=n}^{D} d2^d + \sum_{d=n}^{D} 2^d \ . \quad (13)$$

Обозначим входящие в это выражение частные суммы как

$$S_1 = \sum_{d=n}^{D} 2^d \ ,$$

$$S_2 = \sum_{d=n}^{D} d2^d$$

и последовательно определим их.

Частная сумма ряда S_1 от n до D равна разности между частной суммой этого ряда от нуля до D и частной суммой от нуля до $n - 1$:

$$S_1 = \sum_{d=0}^{D} 2^d - \sum_{d=0}^{n-1} 2^d. \quad (14)$$

Обозначим уменьшаемое в (14) как

$$S_{11} = \sum_{d=0}^{D} 2^d,$$

а вычитаемое как

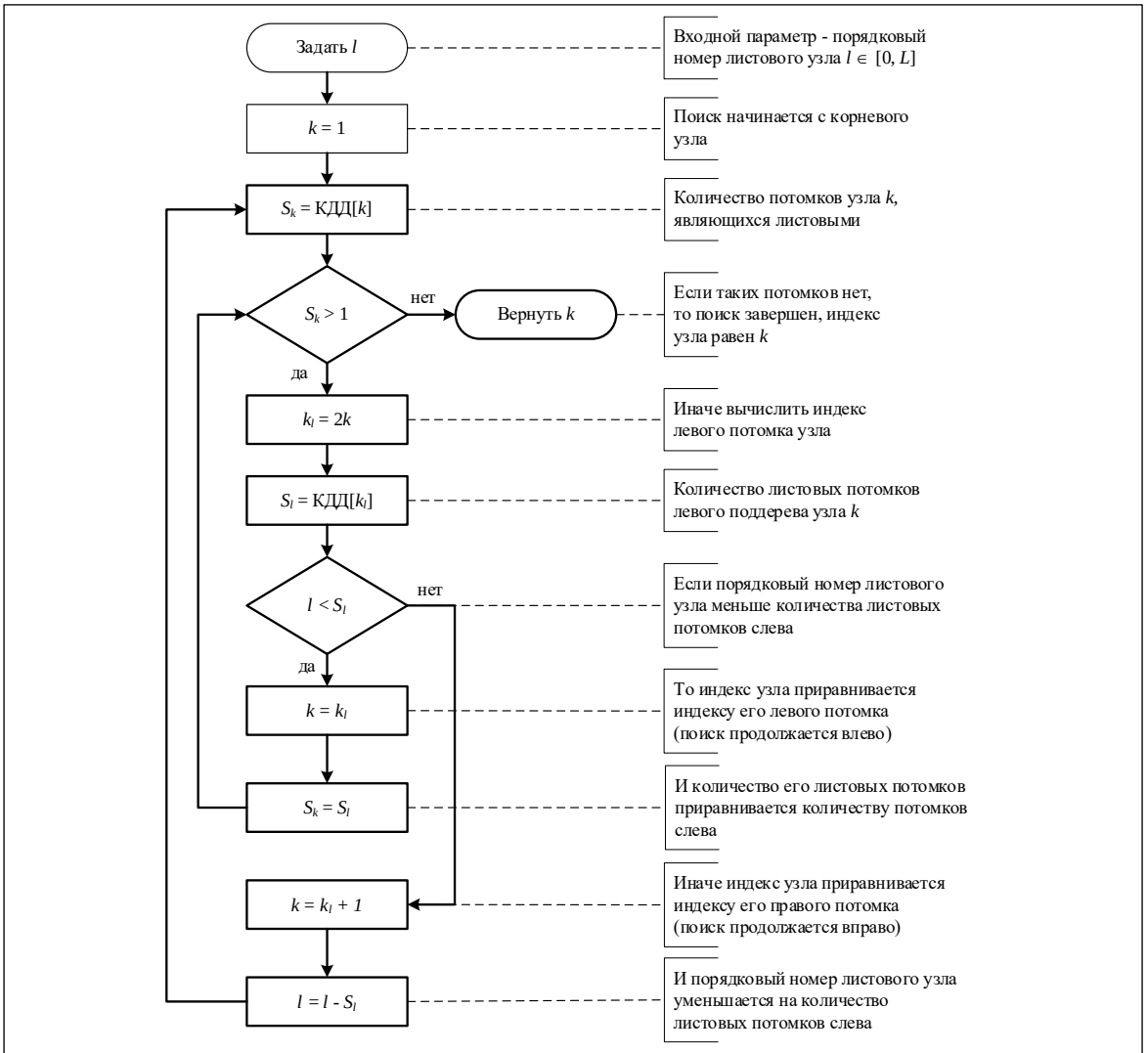


Рис. 8. Блок-схема алгоритма поиска индекса листового узла КДД по его порядковому номеру

Fig. 8. An algorithm of a leaf node index search by its sequence number

$$S_{12} = \sum_{d=0}^{n-1} 2^d.$$

Уменьшаемое S_{11} представляет собой геометрическую прогрессию со знаменателем $q = 2$ и первым членом $a_1 = 2^0 = 1$. Частная сумма ряда S_{11} будет равна сумме $D + 1$ членов этой геометрической прогрессии:

$$S_{11} = \frac{a_1\left(q^{D+1}-1\right)}{q-1} = \frac{1\left(2^{D+1}-1\right)}{1-2} = 2^{D+1}-1. \quad (15)$$

По аналогии с (15)

$$S_{12} = \frac{a_1\left(q^{n-1+1}-1\right)}{q-1} = \frac{1\left(2^{n}-1\right)}{1-2} = 2^{n}-1.$$

Тогда частная сумма ряда S_1 равна

$$S_1 = S_{11} - S_{12},$$

$$S_1 = 2^{D+1} - 2^n.$$

Частная сумма ряда S_2 от n до D также равна разности между частной суммой этого ряда от нуля до D и частной суммой от нуля до $n - 1$:

$$S_2 = \sum_{d=0}^{D} d2^d - \sum_{d=0}^{n-1} d2^d. \quad (16)$$

Обозначим уменьшаемое в (16) как

$$S_{21} = \sum_{d=0}^{D} d2^d, \quad (17)$$

а вычитаемое как

$$S_{22} = \sum_{d=0}^{n-1} d2^d.$$

Найдем S_{21}. Для этого домножим обе части равенства (17) на 2, а множитель d запишем как $d + 1 - 1$:

$$2S_{21} = \sum_{d=0}^{D} (d+1-1)2^{d+1}. \quad (18)$$

Представим (18) в виде разности сумм:

$$2S_{21} = \sum_{d=0}^{D} (d+1)2^{d+1} - \sum_{d=0}^{D} 2^{d+1} \quad (19)$$

или

$$2S_{21} = S_{211} - S_{212}, \quad (20)$$

где

$$S_{211} = \sum_{d=0}^{D} (d+1)2^{d+1},$$

$$S_{212} = \sum_{d=0}^{D} 2^{d+1}.$$

Запишем первую сумму ряда в (19):

$$S_{211} = 1(2^1) + 2(2^2) + 3(2^3) + \ldots + D(2^D) + (D+1)(2^{D+1}).$$

Все слагаемые в этом ряду, за исключением последнего, можно записать как

$$1(2^1) + 2(2^2) + 3(2^3) + \ldots + D(2^D) = \sum_{d=0}^{D} d2^d.$$

Тогда с учетом (17)

$$S_{211} = S_{21} + (D+1)2^{D+1}. \quad (21)$$

Вторая сумма ряда в (19) S_{212} равна сумме $D + 1$ членов геометрической прогрессии со знаменателем $q = 2$ и первым членом $a_1 = 2^1 = 2$:

$$S_{212} = \frac{a_1\left(q^{D+1}-1\right)}{q-1} = \frac{2\left(2^{D+1}-1\right)}{1-2} = 2(2^{D+1}) - 2 = 2^{D+2} - 2. \quad (22)$$

Подставляя в (20) выражения (21) и (22), получим

$$2S_{21} = S_{21} + (D+1)2^{D+1} - 2^{D+2} - 2.$$

Вычтем из обоих частей этого равенства S_{21}:

$$S_{21} = (D+1)2^{D+1} - 2^{D+2} - 2.$$

В результате упрощения

$$S_{21} = D(2^{D+1}) - 2^{D+1} - 2. \quad (23)$$

По аналогии с (23)

$$S_{22} = n(2^n) - 2^{n+1} + 2.$$

Таким образом, частная сумма ряда

$$S_2 = S_{21} - S_{22},$$

$$S_2 = D(2^{D+1}) - 2^{D+1} - n(2^n) + 2^{n+1}.$$

Подставляя S_1 и S_2 в (13), получаем

$$N_b = D\left(2^{D+1} - 2^n\right) - \left(D2^{D+1} - 2^{D+1} - n2^n + 2^{n+1}\right) + \left(2^{D+1} - 2^n\right).$$

В результате упрощения

$$N_b = 2^{D+2} - D(2^n) + n(2^n) - 3(2^n). \quad (24)$$

Для случая $n = 0$ выражение (24) принимает вид $N_b = 2^{D+2} - D - 3$.

На практике к этому количеству бит необходимо добавить $D + 1$ бит для записи глубины дерева в элементе с индексом 0 в виде 2^D. Еще два дополнительных бита необходимы для выравнивая их общего количества до границы восьмибитного байта. Таким образом, количество бит, необходимых для размещения КДД глубиной D, равно

$$N_b = 2^{D+2}.$$

На рисунке 9 представлен пример размещения КДД глубиной $D = 4$ в непрерывном участке памяти.

Несмотря на то, что рост количества необходимой для размещения КДД памяти растет экспоненциально с ростом глубины дерева, ее объем остается относительно небольшим для средних глубин. Так, КДД глубиной $D = 20$, имеющее 1 048 576 листовых узлов на последнем уровне, для хранения требует всего 512 килобайт памяти.

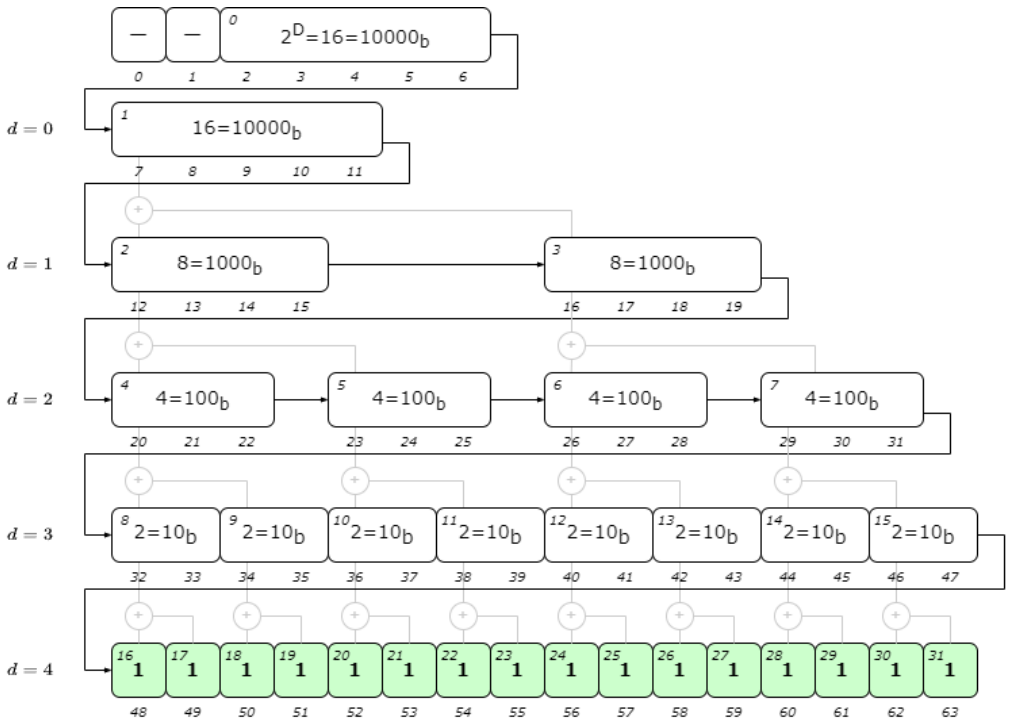


Рис. 9. Расположение структуры данных КДД глубиной D = 4 в непрерывном участке памяти (ширина каждого элемента пропорциональна количеству занимаемых бит)

Fig. 9. A concurrent binary tree with a depth D = 4 memory layout (the width of each block is proportional to its bit-size)

Доступ к элементам структуры КДД. Для доступа к k-му элементу КДД необходимо определить диапазон бит, в которых он хранится в непрерывном представлении КДД в памяти.

Первые два бита в структуре КДД не используются, размер последнего элемента равен одному биту, а полный размер структуры составляет 2^D бит. Следовательно, смещение k-го элемента от начала структуры КДД $x_k \in [2, 2^D–1]$.

Из предыдущего текста известно, что k-й элемент занимает $N_b = D – d_k – 1$ бит, где $d_k = \lfloor \log_2(k) \rfloor$.

Выведем выражение для определения индекса первого бита x_k для k-го элемента КДД.

Заметим, что размер всех элементов КДД в пределах любого уровня d одинаков и определяется по формуле (11). Тогда, зная смещение первого элемента уровня d относительно начала КДД и порядковый номер узла k на этом уровне, можно однозначно найти искомый x_k.

Определим индекс бита x_{d_k}, с которого начинается запись узлов КДД на уровне d_k. Для этого найдем разницу между полным размером КДД и количеством бит, необходимых для записи узлов уровней дерева, начиная с уровня d_k:

$$x_{d_k} = N_b - N_{b_{tail}}.$$

По аналогии с (12) количество бит $N_{b_{tail}}$ для кодирования узлов уровней от d_k до D равно

$$N_{b_{tail}} = \sum_{d=d_k}^{D} 2^d \left(D - d + 1\right).$$

Ранее эта сумма была найдена для общего случая. Решая (24) для случая $n = d_k$, получим

$$N_{b_{tail}} = 2^{D+2} - D2^{d_k} + d_k 2^{d_k} - 3(2^{d_k}).$$

Тогда смещение первого элемента уровня d_k будет

$$x_{d_k} = 2^{d_k}\left(D - d_k + 3\right). \quad (25)$$

Из (2) индекс первого узла на уровне d_k равен

$$k_d = 2^{d_k}. \quad (26)$$

Следовательно, выразим общее смещение первого бита узла k:

$$x_k = x_{d_k} + \left(k - k_d\right) N_{b_k}.$$

При этом каждый узел на уровне d_k занимает N_{b_k} бит. После подстановки (25), (26) и (11) получаем выражение

$$x_k = 2^{d_k}\left(D - d_k + 3\right) + \left(k - 2^{d_k}\right)\left(D - d_k + 1\right).$$

В результате упрощения

$$x_k = 2^{d_k+1} + k\left(D - d_k + 1\right).$$

Таким образом, *k*-й элемент КДД занимает в его представлении в памяти биты в интервале $\left[x_k, x_k + N_{b_k}\right)$. Например, первый бит битового поля КДД, которое кодирует его листовые узлы, имеет индекс $k = 2^D$ и занимает в памяти диапазон бит [3(2^D), 3(2^D + 1)). Для КДД, изображенного на рисунке 9, это диапазон [48, 49).

Процесс использования КДД

После первичной инициализации КДД его использование представляет собой циклический процесс параллельного обновления топологии путем разбиения или слияния узлов с последующей редукцией суммы (рис. 10).

Для инициализации КДД заданной глубины *D* выделяется непрерывный участок памяти размером $N_b = 2^D+2$ бит. Выделенная память инициализируется нулями. Затем выполняется первичное разделение дерева до выбранной глубины $d \in [0, D]$. Для этого определяется диапазон индексов узлов этого уровня $k_d \in [2^d, 2^{d+1} - 1]$ и для каждого из них в битовом поле устанавливается в единицу значение соответствующего ему бита с индексом $x_k = k2^{D-d_k} - 2^D$.

В результате в битовом поле установленным в единицу оказывается каждый 2^{D-d} бит. Результат инициализации КДД при $d = D = 4$ приведен на рисунке 7 (слева).

Редукция суммы КДД выполняется начиная с битового поля на уровне $d = D$ и продолжается до записи в корневой узел на уровне $d = 0$ количества всех листовых узлов дерева. Алгоритм требует $D - 1$ итераций. Начиная с $d = D$ на каждой итерации выполняется 2^{d-1} сложений. Общее количество сложений будет

$$\sum_{d=1}^{D} 2^{d-1} = 2^D - 1.$$

Следовательно, сложность алгоритма редукции двоичного дерева составляет $O(2^D)$.

Алгоритмы редукции двоичного дерева могут быть очень эффективно реализованы с использованием параллельных вычислений, в том числе с GPU [13, 14]. При вычислении редукции в *P* потоков сложность алгоритма составляет

$$O\left(\frac{2^D}{P} + \log_2 P\right).$$

Алгоритм обновления топологии КДД представляет собой простую итерацию по всем его листовым узлам, каждый из которых может быть объединен с соседним узлом (сестринским) или разделен на два дочерних. Условие слияния или разделения узлов зависит от конкретного практического применения КДД.

После завершения обновления топологии выполняется редукция суммы и процесс циклически повторяется.

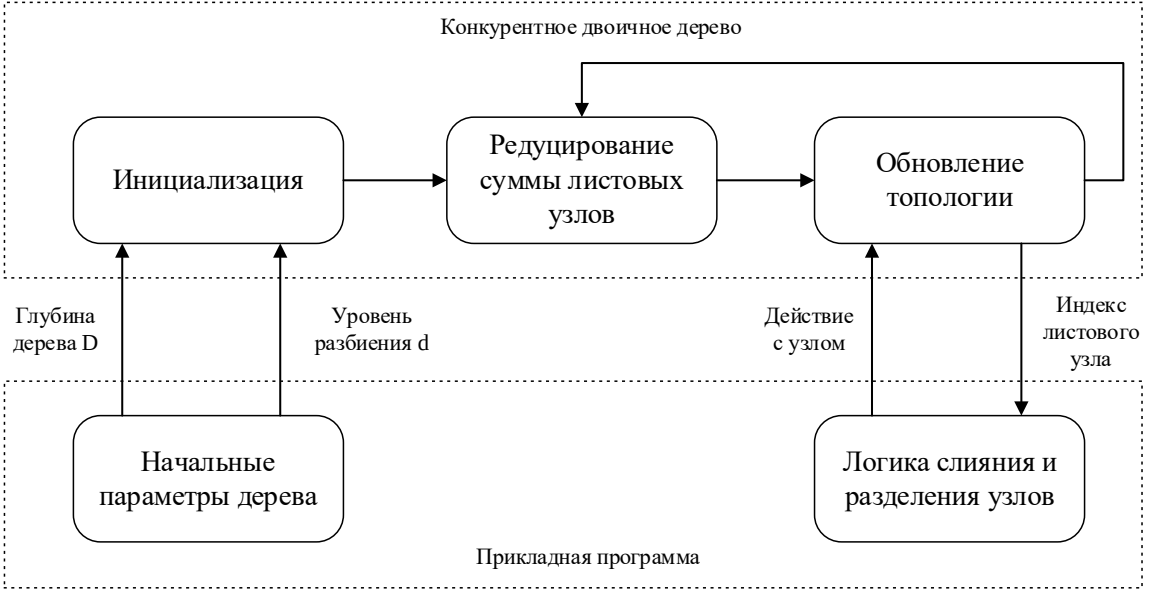


Рис. 10. Блок-схема процесса использования КДД

Fig. 10. A concurrent binary tree usage process diagram

Тесты производительности КДД

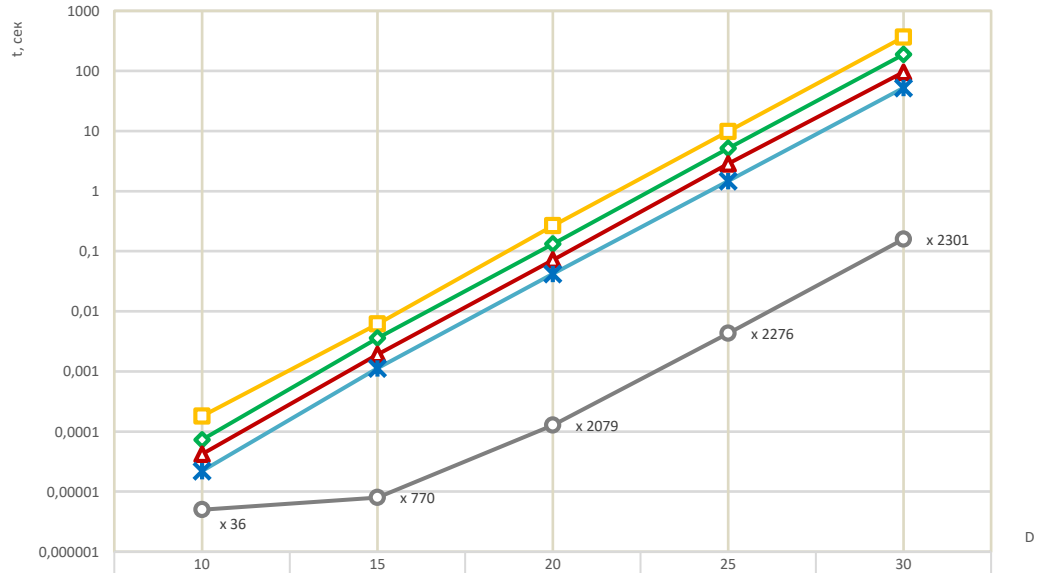
Синтетические тесты производительности процесса использования структуры КДД проводились с использованием его многопоточной программной реализации. Для многопоточной обработки КДД на центральном процессоре использована библиотека OpenMP. Многопоточная обработка на GPU реализована с использованием вычислительных шейдеров OpenGL [11].

Для хранения в памяти двоичной кучи КДД используются 32-битные переменные, причем количество бит для хранения суммы листовых узлов на каждом уровне различно. При многопоточной записи в двоичную кучу возможны ситуации, когда различные потоки будут одновременно обращаться к одной и той же переменной. Это может привести к повреждению данных в двоичной куче и нарушению корректной работы алгоритмов. Для исключения подобной ситуации на CPU применяется специальный тип std::atomic<uint_32t>, на аппаратном уровне гарантирующий атомарность операций с ним [15]. На GPU используются специальные атомарные операции над типом uint – atomicAnd и atomicOr. Запись нескольких битов в 32-битную переменную – единственное место в программной реализации КДД, требующее применения специальных приемов многопоточного программирования. Такая ситуация возникает при вычислении суммы листовых узлов дерева, а также при выполнении операций слияния и разделения узлов.

Сравнительные результаты замеров производительности редукции суммы листовых узлов КДД представлены по ссылке (http://www.swsys.ru/uploaded/image/2023-3/2023-3-dop/13.jpg), их декодирования – на рисунке 11.

Время редуцирования суммы листовых узлов КДД линейно уменьшается с увеличением количества потоков, выполняющих алгоритм редукции. Использование восьми потоков процессора уменьшает время выполнения алгоритма в пять–шесть раз. Редуцирование суммы с использованием нескольких тысяч шейдеров GPU снижает это время на два-три порядка.

Декодирование листовых узлов также эффективно выполняется несколькими потоками. Прирост скорости при выполнении итерации по листовым узлам на GPU составляет более двух тысяч раз.



	10	15	20	25	30
CPU 1	0,000180	0,006158	0,266135	9,848790	365,332272
CPU 2	0,000073	0,003586	0,131800	5,163936	187,841708
CPU 4	0,000042	0,001915	0,071508	2,845855	95,478423
CPU 8	0,000022	0,001127	0,042104	1,467655	52,341579
GPU	0,000005	0,000008	0,000128	0,004327	0,158764

Рис. 11. Время декодирования листовых узлов совершенного КДД в зависимости от его глубины

Fig. 11. Concurrent binary tree leaf nodes decoding time depending on tree depth

Необходимо обратить внимание, что на рисунке 11 представлены результаты декодирования максимально возможного количества листовых узлов совершенного КДД заданной глубины, то есть наиболее худший с точки зрения производительности случай. При практическом использовании КДД в СММО для реализации алгоритма рекурсивного деления количество его листовых узлов, как правило, не превышает 20 % от максимально возможного. Время полного перебора листовых узлов дерева пропорционально уменьшается в зависимости от степени его заполнения. Пример интегральной производительности КДД (вычисление редукции суммы плюс декодирование листовых узлов) с 25-процентным заполнением листовыми узлами приведен по ссылке (http://www.swsys.ru/uploaded/image/2023-3/2023-3-dop/14.jpg).

На практике нужно дополнительно учесть время, затрачиваемое прикладной программой на принятие решения о слиянии или разделении узлов дерева. Однако оно не повлияет на линейный характер масштабируемости производительности КДД.

Интегральная производительность одного цикла использования КДД на реальном примере использования в СММО отражена на рисунке 12.

В СММО глубина КДД составляет 17 при 25-процентном заполнении листовыми узлами. Общее время выполнения декодирования листовых узлов с последующим расчетом их редуцированной суммы составляет 23 микросекунды. Если декодирование листовых узлов не приводит к последующему изменению топологии дерева, то есть не происходит слияние или разделение узлов, это время уменьшается до 11 микросекунд за счет исключения из цикла шага редуцирования суммы. Такая ситуация в СММО встречается, когда априорно известно, что алгоритм рекурсивного деления не внесет изменений в топологию двоичного дерева, так как не изменились входные данные этого алгоритма.

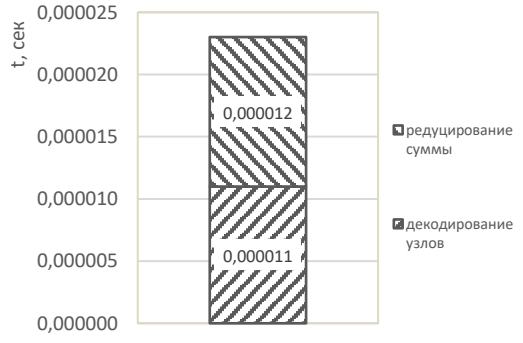


Рис. 12. Время выполнения одного цикла использования КДД глубиной 17 при 25-процентном заполнении листовыми узлами

Fig. 12. One cycle time of concurrent binary tree usage with a tree depth equals 17 and 25 % leaf node count

Заключение

Задача автоматического построения локально адаптивной трехмерной модели глобального рельефа может быть эффективно решена на GPU с использованием алгоритма рекурсивного деления, допускающего массированную параллельную обработку данных. Для обеспечения корректной параллельной работы такого алгоритма обрабатываемые данные необходимо представлять в виде специализированной структуры – КДД. Рассмотренные в статье теоретические свойства КДД и результаты тестов его производительности позволяют сделать следующие выводы.

КДД сочетает в себе возможность представления произвольной изменяемой топологии дерева и эффективной параллельной обработки его узлов.

Представление КДД в непрерывном участке памяти позволяет использовать его на GPU.

Несмотря на экспоненциальный характер роста потребного размера памяти КДД с увеличением его максимальной глубины ее объем остается в пределах, допускающих применение КДД в условиях ограниченного бюджета видеопамяти.

Параллельная итерация по листовым узлам КДД может быть совмещена с их слиянием и разделением без потери работоспособности структуры и без необходимости промежуточного хранения индексов листовых узлов с дополнительным выделением памяти.

Производительность алгоритмов обработки КДД на GPU позволяет эффективно использовать его в приложениях реального времени, таких как системы визуализации.

Применение КДД ограничено задачами, использующими алгоритмы рекурсивного деления, в которых возможно либо разделение узла на два потомка, либо слияние двух узлов в родительский узел и исключена ситуация возникновения у узла дерева единственного потомка.

Список литературы

1. Барков В.А., Грачев В.Г., Насыров Р.Р. и др. Система моделирования морских операций (СММО): Свид. о регистр. ПрЭВМ № 201866191. Рос. Федерация, 2018.
2. Барков В.А., Грачев В.Г., Насыров Р.Р. и др. СММО – Моделирующий комплекс: Свид. о регистр. ПрЭВМ № 2018661128. Рос. Федерация, 2018.
3. Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D. Fast BVH construction on GPUs. Comput. Graphics Forum, 2009, vol. 28, no. 2, pp. 375–384. doi: 10.1111/j.1467-8659.2009.01377.x.
4. Garanzha K., Pantaleoni J., McAllister D. Simpler and faster HLBVH with work queues. Proc. ACM SIGGRAPH Symposium on HPG, 2011, pp. 59–64. doi: 10.1145/2018323.2018333.
5. Karras T. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. Proc. ACM SIGGRAPH Symposium on HPG, 2012, pp. 33–37.
6. Воронцов Г.В., Преображенский А.П., Чопоров О.Н. Быстрое построение BVY дерева на GPGPU // Моделирование, оптимизация и информационные технологии. 2018. Т. 6. № 2. С. 25–34.
7. Dupuy J. Concurrent binary trees (with application to longest edge bisection). PACMCGIT, 2020, vol. 3, no. 2, pp. 1–20. doi: 10.1145/3406186.
8. Moir M., Shavit N. Concurrent Data Structures. Boca Raton, CRC Press, 2018, 31 p.
9. Munro J.I., Suwanda H. Implicit data structures for fast search and update. JCSS, 1980, vol. 21, no. 2, pp. 236–250.
10. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ; [пер. с англ.]. М.: Вильямс, 2013. 1328 с.
11. Kessenich J., Baldwin D., Rost R. The OpenGL Shading Language, Version 4.60.7. 234 p. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf (дата обращения: 03.05.2023).
12. Drepper U. What Every Programmer Should Know About Memory. 2007, 114 p. URL: https://www.akkadia.org/drepper/cpumemory.pdf (дата обращения: 03.05.2023).
13. Harris M. Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology. 38 p. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (дата обращения: 03.05.2023).
14. Mahardito A., Suhendra A., Hasta D.T. Optimizing Parallel Reduction in CUDA to Reach GPU Peak Performance. URL: https://core.ac.uk/reader/143963699 (дата обращения: 03.05.2023).
15. Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ; [пер. с англ.]. М.: ДМК Пресс, 2012. 672 с.

Software & Systems doi: 10.15827/0236-235X.142.398-413 2023, vol. 36, no. 3, pp. 398–413

A research on concurrent binary tree properties

Vsevolod G. Grachev
Aleksandr S. Ushakov

For citation

Grachev, V.G., Ushakov, A.S. (2023) 'A research on concurrent binary tree properties', *Software & Systems*, 36(3), pp. 398–413 (in Russ.). doi: 10.15827/0236-235X.142.398-413

Article info
Received: 10.05.2023 After revision: 05.06.2023 Accepted: 30.06.2023

Abstract. The navigation simulator software "Marine Operation Simulation System" (MOSS) implements the automatic construction technology for a global terrain 3D-model based on the recursive subdivision algorithm with a locally adaptive level of detail control. The algorithm is entirely implemented on the graphics adapter (GPU) and uses massive parallel data processing in compute shaders. The recursive subdivision algorithm is based on using a binary tree; however, classical binary trees explicitly described using pointers are not applicable to GPU implementations due to architectural features. To ensure the parallel execution of the algorithm, the authors use a specialized parallel data structure that is a concurrent binary tree (CBT). The article describes the prerequisites for creating a CBT structure and considers its construction stages from using an implicit binary tree up to representing a binary tree as a binary field augmented with a reduced sum of leaf nodes. The required RAM amount to accommodate a CBT of a given depth is theoretically substantiated. There is an analysis of the algorithmic complexity of constructing a CBT and iterating over its leaf nodes. The article presents and analyzes the results of comparative synthetic performance tests of a concurrent binary tree performed on a central processing unit (CPU) and GPU, as well as the results of the practical application of CBT in MOSS.

Keywords: concurrent binary tree, parallel data structure, recursive subdivision algorithm, memory optimization, massive GPU parallelism, marine operations simulation system

Reference List

1. Barkov, V.A., Grachev, V.G., Nasyrov, R.R. et al. (2018) *Marine Operations Simulation System (MOSS)*, Pat. RF, № 2018661914.
2. Barkov, V.A., Grachev, V.G., Nasyrov, R.R. et al. (2018) *MOSS – Modeling Complex*, Pat. RF, № 2018661128.
3. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D. (2009) 'Fast BVH construction on GPUs', *Comput. Graphics Forum*, 28(2), pp. 375–384. doi: 10.1111/j.1467-8659.2009.01377.x.
4. Garanzha, K., Pantaleoni, J., McAllister, D. (2011) 'Simpler and faster HLBVH with work queues', *Proc. ACM SIGGRAPH Symposium on HPG*, pp. 59–64. doi: 10.1145/2018323.2018333.
5. Karras, T. (2012) 'Maximizing parallelism in the construction of BVHs, octrees, and k-d trees', *Proc. ACM SIGGRAPH Symposium on HPG*, pp. 33–37.
6. Vorontsov, G.V., Preobrazhenskiy, A.P., Choporov, O.N. (2018) 'The algorithms of parallel radix sorting on GPGPU', *Modeling, Optimization and Inform. Tech.*, 6(2), pp. 25–34 (in Russ.).
7. Dupuy, J. (2020) 'Concurrent binary trees (with application to longest edge bisection)', *PACMCGIT*, 3(2), pp. 1–20. doi: 10.1145/3406186.
8. Moir, M., Shavit, N. (2018) *Concurrent Data Structures*. Boca Raton: CRC Press, 31 p.
9. Munro, J.I., Suwanda, H. (1980) 'Implicit data structures for fast search and update', *JCSS*, 21(2), pp. 236–250.
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2013) *Introduction to Algorithms*. Cambridge: The MIT Press, 1292 p. (Russ. ed.: (2013) Moscow, 1328 p.).
11. Kessenich, J., Baldwin, D., Rost, R. *The OpenGL Shading Language, Version 4.60.7*. 234 p., available at: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf (accessed May 03, 2023).
12. Drepper, U. (2007) *What Every Programmer Should Know About Memory*, 114 p., available at: https://www.akkadia.org/drepper/cpumemory.pdf (accessed May 03, 2023).
13. Harris, M. *Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology*, 38 p., available at: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (accessed May 03, 2023).
14. Mahardito, A. *Optimizing Parallel Reduction in CUDA to Reach GPU Peak Performance*, available at: https://core.ac.uk/reader/143963699 (accessed May 03, 2023).
15. Williams, A. (2012) *C++ Concurrency in Action: Practical Multithreading*, Shelter Island, NY: Manning Publ., 506 p. (Russ. ed.: (2012) Moscow, 672 p.).

Авторы
Грачев Всеволод Геннадиевич [1], к.т.н.,
ведущий научный сотрудник отдела комплексных технических средств обучения, grachev@cps.tver.ru
Ушаков Александр Сергеевич [2], к.ф.-м.н.,
начальник отдела программно-аппаратных средств и телекоммуникационных систем,
al.s.ushakov@yandex.ru

[1] НИИ «Центрпрограммсистем»,
г. Тверь, 170024, Россия
[2] ОАО «НИИЭС»,
г. Москва, 125124,
Россия

Authors
Vsevolod G. Grachev [1], Ph.D. (Engineering),
Leading Researcher,
grachev@cps.tver.ru
Aleksandr S. Ushakov [2],
Ph.D. (Physics and Mathematics),
Head of Department,
al.s.ushakov@yandex.ru

[1] R&D Institute Centerprogramsystem,
Tver, 170024, Russian Federation
[2] Open Joint Stock Company
"Scientific Testing Institute of Ergatic Systems",
Moscow, 125124, Russian Federation