Лекция 3: Использование графических процессоров для ускорения вычислений. Базовые понятия

Н.Д. Смирнова

Санкт-Петербургский государственный Политехнический университет

6.10.2011

Содержание

Содержание

Терминология

знакомые понятия

- поток инструкций = программа
- поток данных = данные для выполнения программы

вводим новые базовые понятия

- kernel (ядро [1]) - программа обработки потока данных
- stream (поток) - данные для выполнения программы

в компьютерной графике

- kernel = vertex shader, fragment (pixel) shader
- stream = vertex data, fragment data: position, texture coordinates, color,...
- global data = глобальная информация: матрицы транформаций, положение источников света, время,...

[1] не путаем с "core" - ядром графической платы

Обработка потоков (stream processing)

Kernel [2]

- функция, применяемая к каждому элементу потока данных (stream'a)
  - трансформация,...
- подразумевает низкую зависимость между элементами потока данных
  - способствует высокой *арифметической интенсивности* (arithmetic intensity = operations / words transfered)

Stream

- набор блоков данных, требующих сходной обработки
  - позиции вершин, ячейки трехмерной сетки, ячейки метода конечных элементов,...
- подходит для data-parallelism'a

[2] General-Purpose Computation on Graphics Hardware. Introduction. David Luebke, NVIDIA, SUPERCOMPUTING 2006.

Содержание

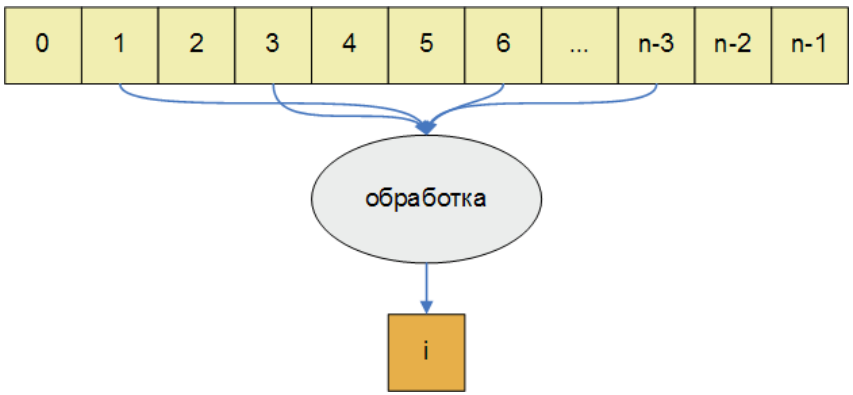
Аналогии между GPU и CPU

GPU	CPU
текстуры (textures)	массивы (arrays)
фрагментный шейдер (fragment shader)	”внутренность” цикла
render-to-texture	feedback
отрисовка (geometry rasterization)	запуск вычислений
текстурные координаты	область данных, необходимых для вычислений (computational domain)
вершинные координаты	область записи результатов вычислений (computational range)

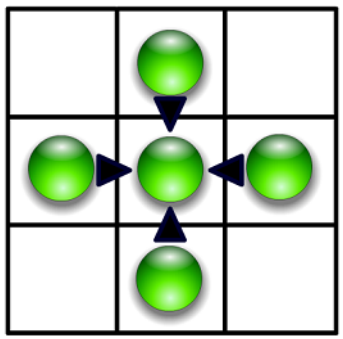
Содержание

Содержание

Gather - собирание данных



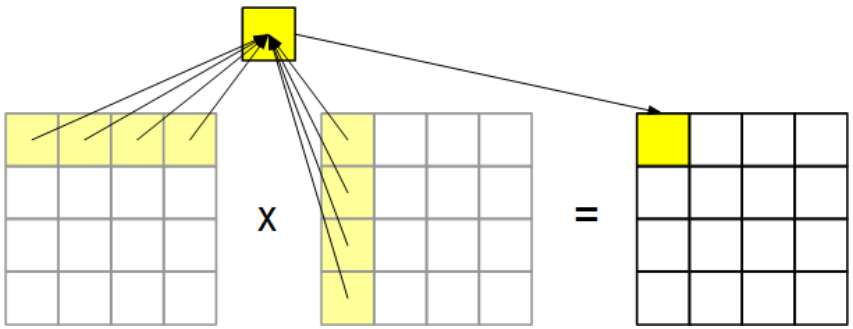
- прекрасно подходит для GPGPU (обработка пикселя (фрагментный шейдер), вершины (вершинный шейдер))
- используется выборка из текстуры



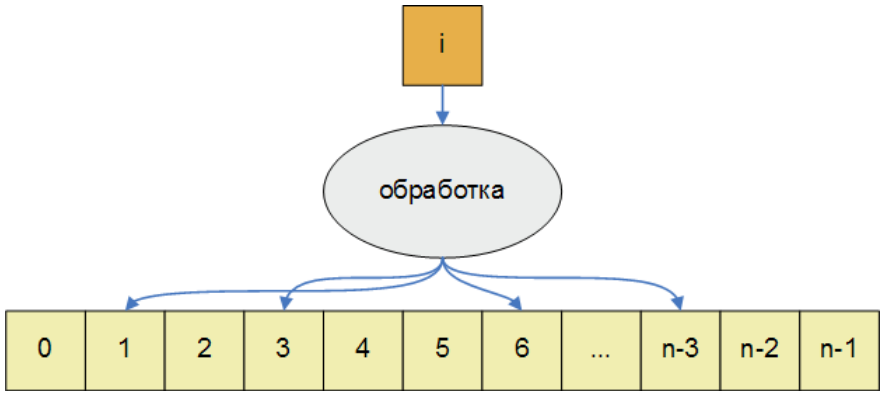
Gather - пример

- фильтрация изображения: для расчета каждого пикселя используются данные и некоторой его окрестности (соседние пикслели)
  - пример: blur (размывание)

- перемножение матриц

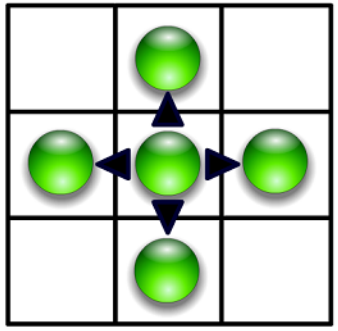


Scatter - рассеивание данных



для GPGPU **все уже не так радужно**:

1. за 1 шаг через обработку вершин (т.к. только позицию вершины можно менять)
2. либо в 2 шага:
   1. запись *(данные + адрес)*
   2. перестановка данных по адресам



отсутствие полноценного scatter во фрагментных шейдерах сильно влияет на алгоритмы

Содержание

Map - отображение

- Дано:
  - массив или поток данных A
  - функция $f(x)$
- $map(A, f)$ = применить f к каждому элементу данных $a_i \in A$
- реализация на GPU - элементарна
  - A - текстура, a_i - тексели текстуры
  - $f(x)$ - фрагментный шейдер
  - надо нарисовать квадрат с количеством пикселей, равным количеству текселей в A
  - результат сохранить в другой текстуре B
  - получить данные из текстуры B
- получили $A \xrightarrow{f} B$

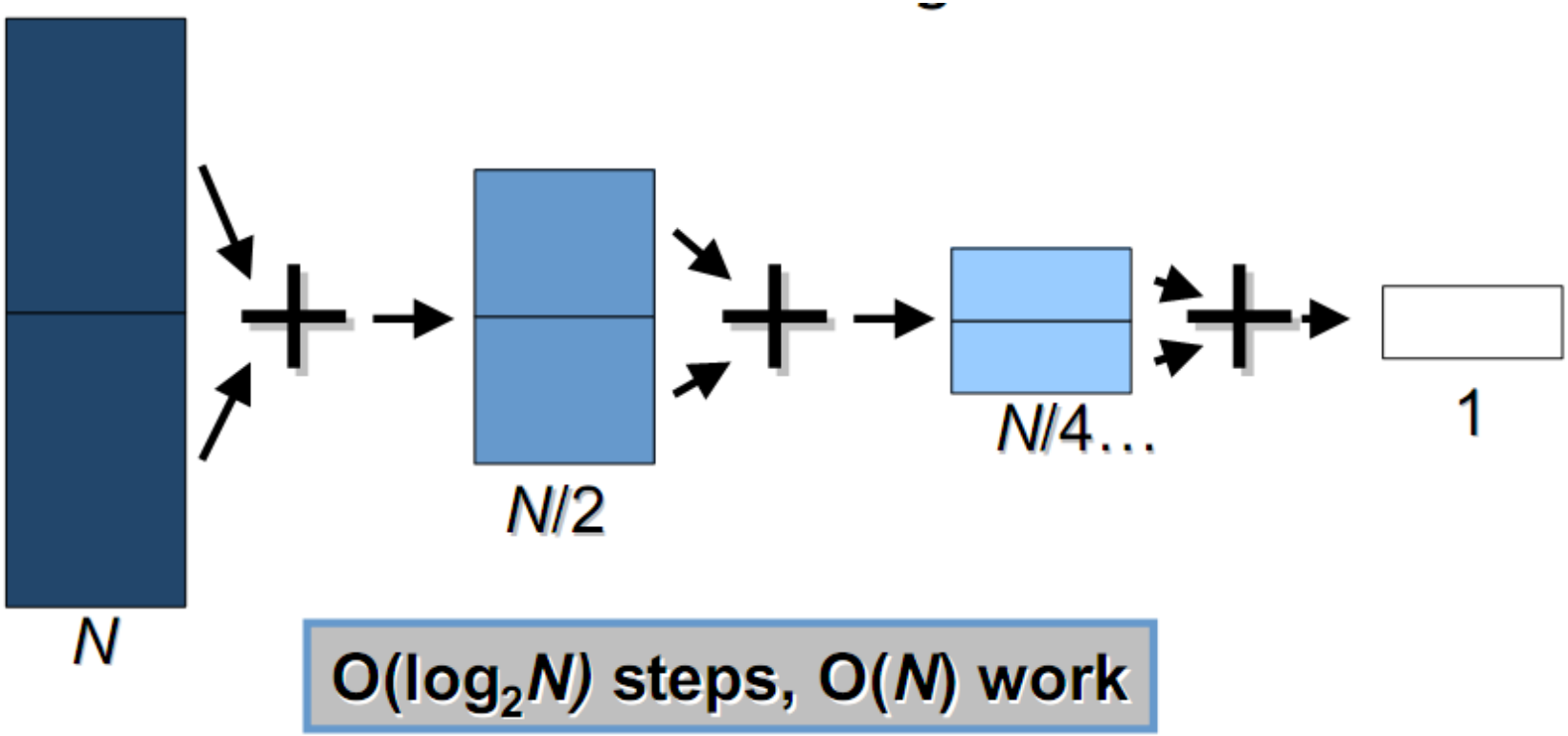
Содержание

Reduce - уменьшение потока данных

- Дано:
  - поток упорядоченных данных $A = \{a_i\}$
  - бинарный ассоциативный оператор $\oplus$ с единицей
- $reduce(\oplus, A) = a_0 \oplus a_1 \oplus a_2 \oplus ... \oplus a_{n-1})$ применяется к данным, чтобы уменьшить их количество (часто до 1 элемента)
- пример: $reduce(+, 3, 1, 7, 0, 4, 1, 6, 3) = 25$
- обычно используется для вычисления: $+, \times, \min, \max$
- получили $A \xrightarrow{\oplus} b$
- операции с плавающей запятой - **псевдоассоциативны**

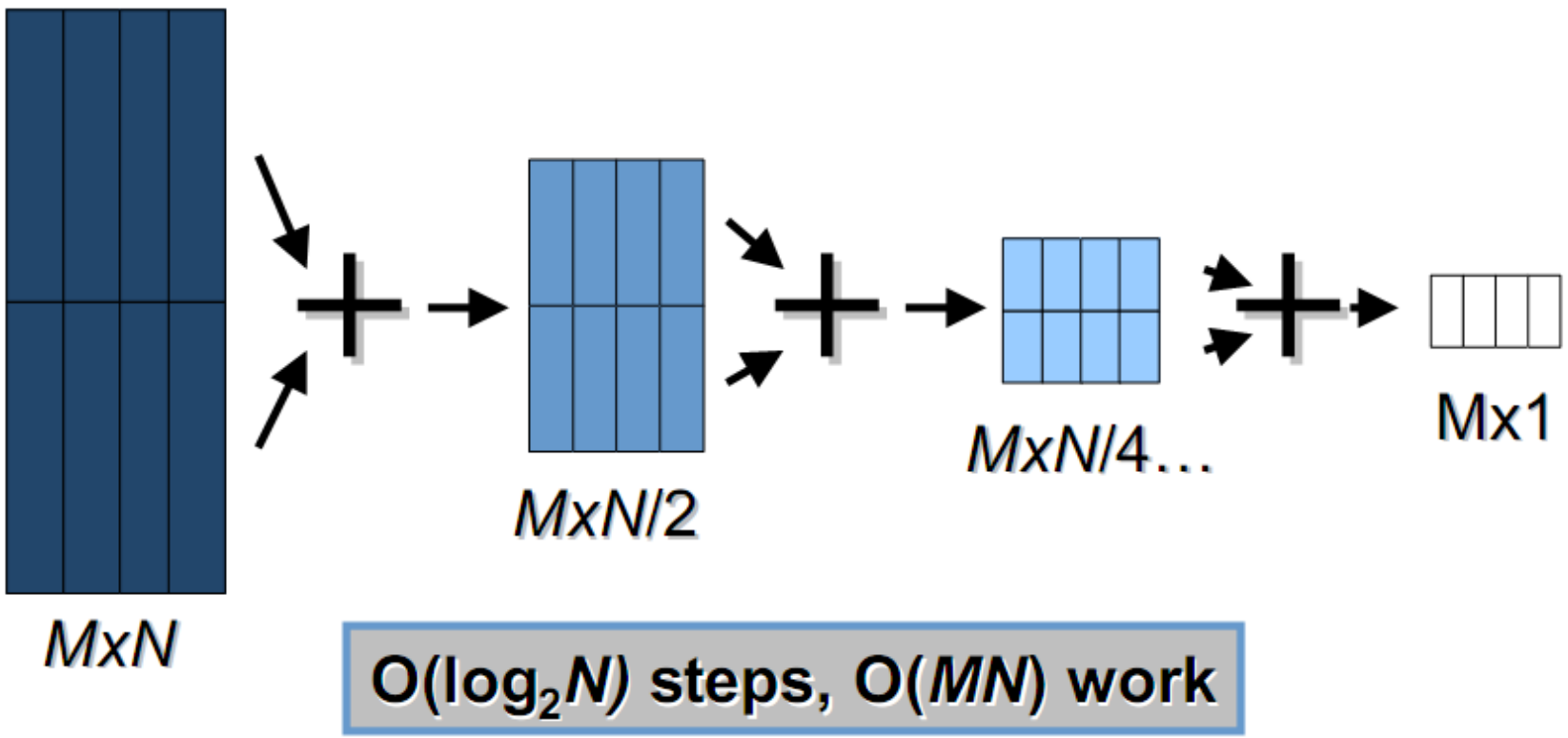
Пример: 1D-reduction

- складываем 2 половины данных поэлементно
- повторяем
- до тех пор пока на останется 1 элемент



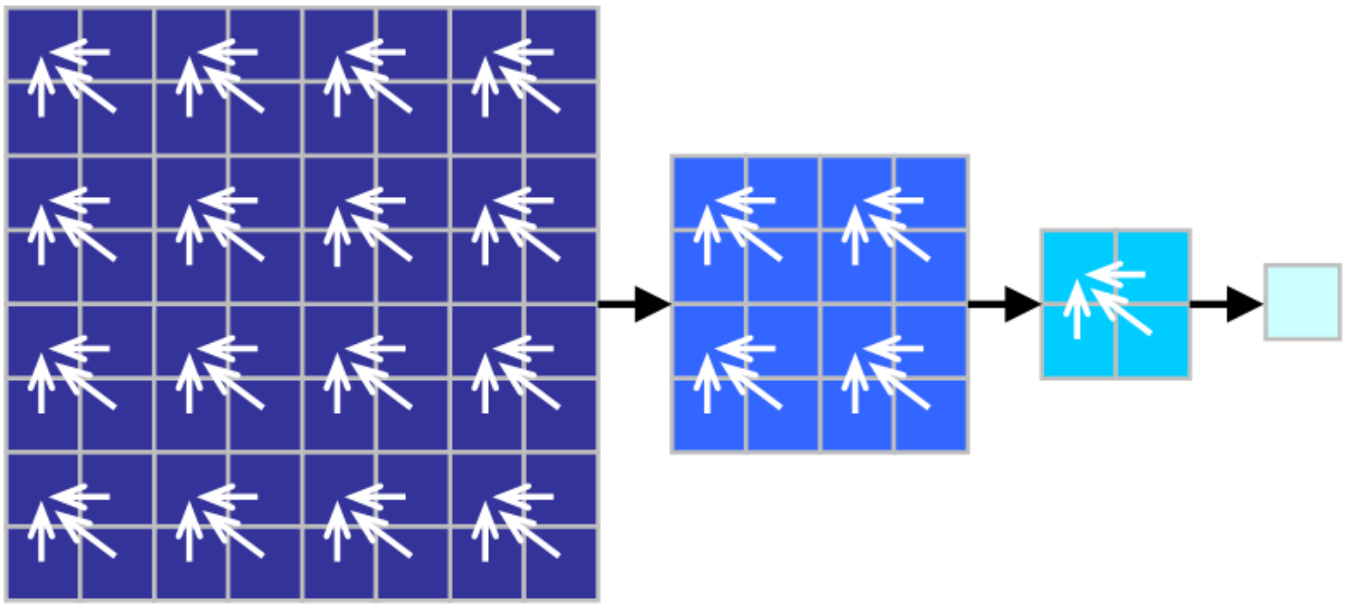
Пример: MX1D-reduction

- можно параллельно сделать M редукций
- 2D текстуру уменьшаем до 1D текстуры



Пример: 2D-reduction

- тоже самое, что и 1D, только редукция проводится одновременно по двум направлениям



- можно обрабатывать больше чем 2x2 пикселей
  - K - количество обрабатываемых элементов за шаг
  - шагов $O(\log_K^N)$
  - выбор K завичит от возможностей железа

Содержание

Binary search - бинарный поиск

- дан поток данных $A = \{a_i\}$ с возможностью сравнения
- необходимо найти элемент
- один конкретный **поиск не распараллеливается**
- но можно выполнять **несколько поисков параллельно** [3]

[3] "A Toolkit for Computation on GPUs". Ian Buck and Tim Purcell. In GPU Gems. Randy Fernando, ed. 2004

Содержание

Scan (all-prefix-sums) - построение всех префиксных сумм

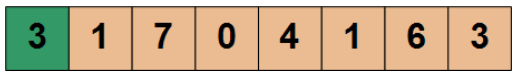
- Дано:
  - поток упорядоченных данных $A = \{a_i\}$
  - бинарный ассоциативный оператор $\oplus$ с единицей
- $scan(\oplus, A) = \{a_0, (a_0 \oplus a_1), ..., (a_0 \oplus ... \oplus a_{n-1})\}$ - inclusive scan [4]
- пример:
  $scan(+, 3, 1, 7, 0, 4, 1, 6, 3) = \{3, 4, 11, 11, 15, 16, 22, 25\}$
- получили $A \xrightarrow{\oplus} B = \{b_i\}$
- оператор выглядит абсолютно последовательным

[4] Parallel Prefix Sum(Scan) for CUDA, Mark Harris, 2007

Применение

- Radix sort, Quicksort
- String comparison
- Lexical analysis
- Histograms
- Tree operations

Наивный алгоритм

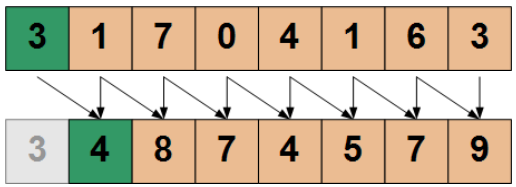


эффективен по шагам, но не по количеству операций [5]

- $O(\log n)$ шагов, $O(n \log n)$ сложений
- последовательный алгоритм - $O(n)$ сложений
- $log(n)$ дает замедление в 20 раз на 10^6 элементах

[5] Horn, Daniel. Stream reduction operations for GPGPU applications. GPU Gems 2.http://developer.nvidia.com/node/17

Наивный алгоритм

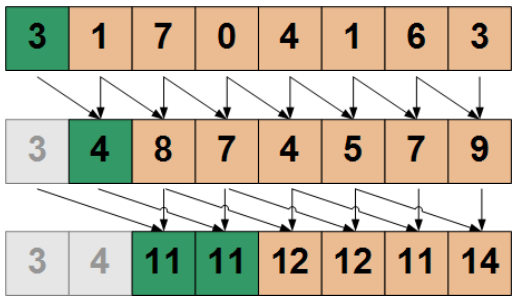


эффективен по шагам, но не по количеству операций [5]

- $O(\log n)$ шагов, $O(n \log n)$ сложений
- последовательный алгоритм - $O(n)$ сложений
- $log(n)$ дает замедление в 20 раз на 10^6 элементах

[5] Horn, Daniel. Stream reduction operations for GPGPU applications. GPU Gems 2.http://developer.nvidia.com/node/17

Наивный алгоритм

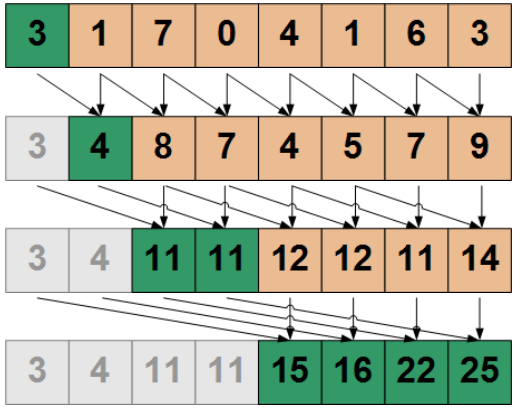


эффективен по шагам, но не по количеству операций [5]

- $O(\log n)$ шагов, $O(n \log n)$ сложений
- последовательный алгоритм - $O(n)$ сложений
- $log(n)$ дает замедление в 20 раз на 10^6 элементах

[5] Horn, Daniel. Stream reduction operations for GPGPU applications. GPU Gems 2.http://developer.nvidia.com/node/17

Наивный алгоритм



эффективен по шагам, но не по количеству операций [5]

- $O(\log n)$ шагов, $O(n \log n)$ сложений
- последовательный алгоритм - $O(n)$ сложений
- $log(n)$ дает замедление в 20 раз на 10^6 элементах

[5] Horn, Daniel. Stream reduction operations for GPGPU applications. GPU Gems 2.http://developer.nvidia.com/node/17

Использование "сбалансированных деревьев"

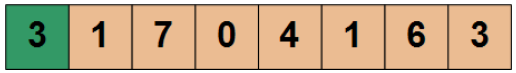
Деревья часто используются в параллельных алгоритмах

- обычная практика: построить дерево и, заметанием вверх, а потом вниз, сделать необходимые вычисления
- дерево - концептуальная структура, а не реальная

Для скана:

- проходом от листьев к корню посчитать суммы в промежуточных узлах
- обратным проходом, используя суммы в узлах, посчитать недостающие суммы

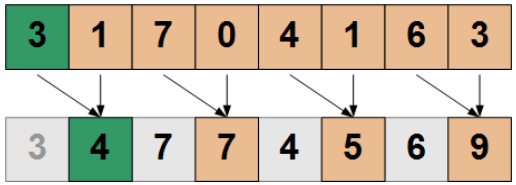
Использование "сбалансированных деревьев"



не очень эффективен по шагам, зато эффективен по количеству операций

- $O(\log n)$ шагов, $O(\log n)$ сложений
- теряет производительность на тех этапах, где обрабатывается мало элементов

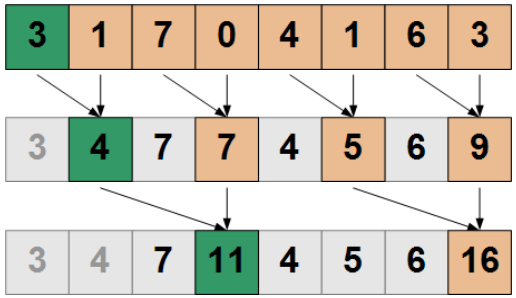
Использование "сбалансированных деревьев"



не очень эффективен по шагам, зато эффективен по количеству операций

- $O(\log n)$ шагов, $O(\log n)$ сложений
- теряет производительность на тех этапах, где обрабатывается мало элементов

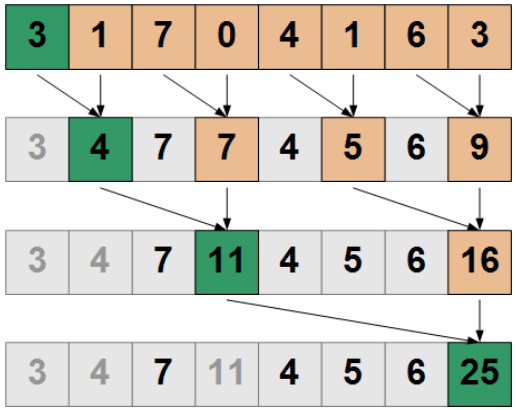
Использование "сбалансированных деревьев"



не очень эффективен по шагам, зато эффективен по количеству операций

- $O(\log n)$ шагов, $O(\log n)$ сложений
- теряет производительность на тех этапах, где обрабатывается мало элементов

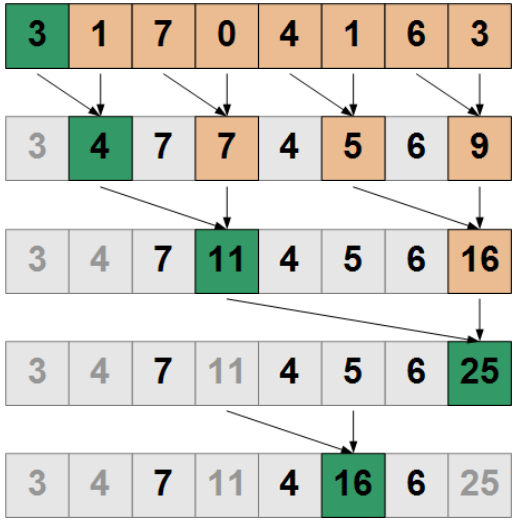
Использование "сбалансированных деревьев"



не очень эффективен по шагам, зато эффективен по количеству операций

- $O(\log n)$ шагов, $O(\log n)$ сложений
- теряет производительность на тех этапах, где обрабатывается мало элементов

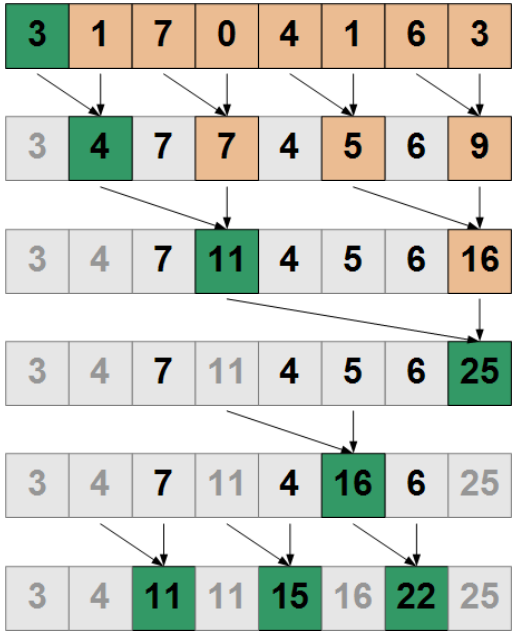
Использование "сбалансированных деревьев"



не очень эффективен по шагам, зато эффективен по количеству операций

- $O(\log n)$ шагов, $O(\log n)$ сложений
- теряет производительность на тех этапах, где обрабатывается мало элементов

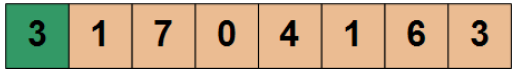
Использование "сбалансированных деревьев"



не очень эффективен по шагам, зато эффективен по количеству операций

- $O(\log n)$ шагов, $O(\log n)$ сложений
- теряет производительность на тех этапах, где обрабатывается мало элементов

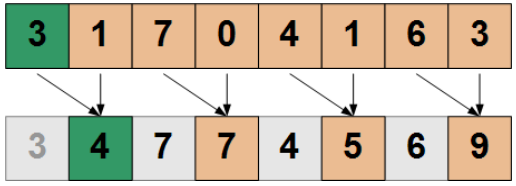
Гибридный метод



переключается на наивный метод для шагов с малым количеством сложений [6]

[6] YA Work-Efficient Step-Efficient Prefix Sum Algorithm¥. Shubhabrata Sengupta, Aaron E. Lefohn, John D. Owens. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures

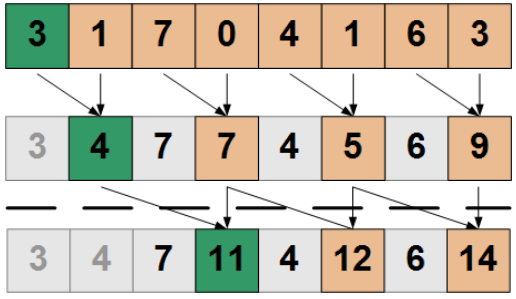
Гибридный метод



переключается на наивный метод для шагов с малым количеством сложений [6]

[6] YA Work-Efficient Step-Efficient Prefix Sum Algorithm¥. Shubhabrata Sengupta, Aaron E. Lefohn, John D. Owens. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures

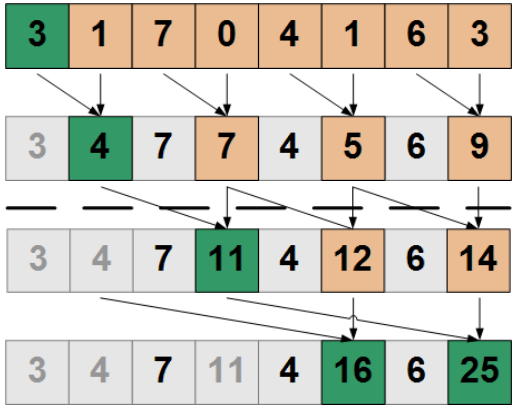
Гибридный метод



переключается на наивный метод для шагов с малым количеством сложений [6]

[6] YA Work-Efficient Step-Efficient Prefix Sum Algorithm¥. Shubhabrata Sengupta, Aaron E. Lefohn, John D. Owens. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures

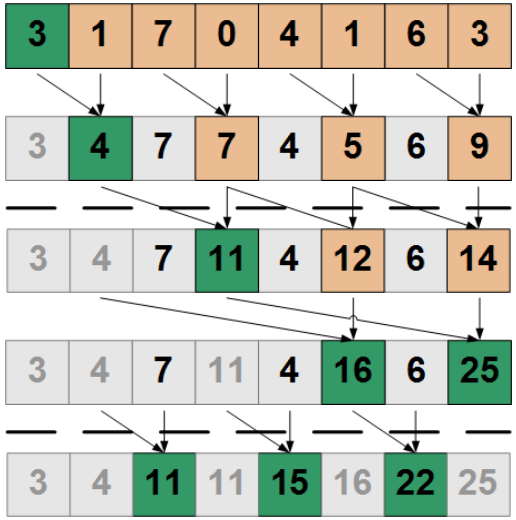
Гибридный метод



переключается на наивный метод для шагов с малым количеством сложений [6]

[6] YA Work-Efficient Step-Efficient Prefix Sum Algorithm¥. Shubhabrata Sengupta, Aaron E. Lefohn, John D. Owens. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures

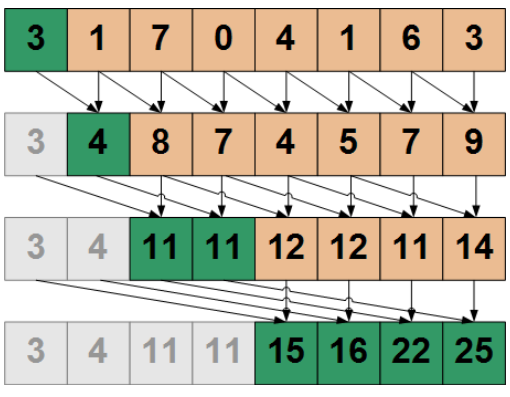
Гибридный метод



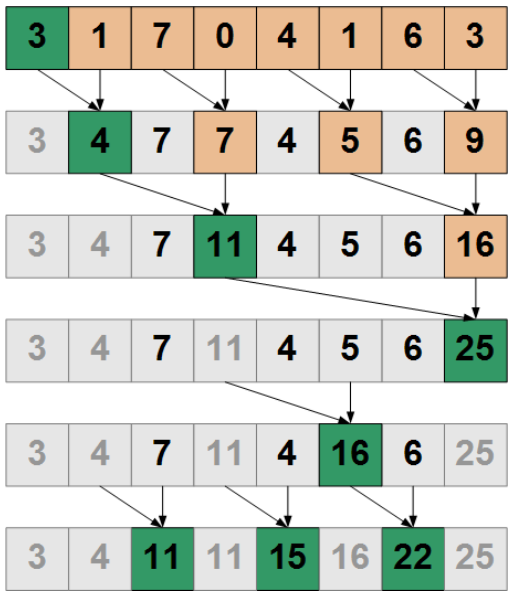
переключается на наивный метод для шагов с малым количеством сложений [6]

[6] YA Work-Efficient Step-Efficient Prefix Sum Algorithm¥. Shubhabrata Sengupta, Aaron E. Lefohn, John D. Owens. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures

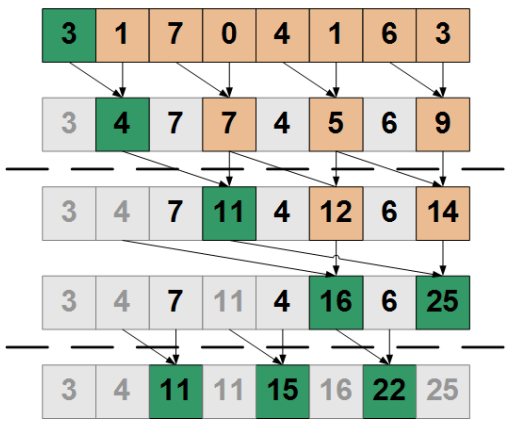
Сравнение



- 3 шага
- 17 сложений



- 5 шагов
- 11 сложений



- 4 шага
- 12 сложений

Содержание

Sort - сортировка

- Дано:
  - поток неупорядоченных данных $A = \{a_i\}$
  - **kernel = {compare, swap}** - сравнивает и переставляет элементы
- $sort(kernel, A) = \{a_2 \overset{\leq}{,} a_1 \overset{\leq}{,} ... \overset{\leq}{,} a_{n-1}\}$
- пример: $scan(\leq, 3, 1, 7, 0, 4, 1, 6, 3) = \{0, 1, 1, 3, 3, 4, 6, 7\}$
- bitonic merge sort [7] (пройдем чуть позже)
- periodic balanced sorting networks [8]

[7] http://en.wikipedia.org/wiki/Bitonic_sorter
[8] http://en.wikipedia.org/wiki/Sorting_networks

Использованные материалы

- Introduction to GPGPU Programming . David Luebke. NVIDIA. Workshop. SUPERCOMPUTING'06
- Data-Parallel Algorithms on GPUs. Mark Harris. NVIDIA. Workshop. SUPERCOMPUTING'06
- Horn, Daniel. Stream reduction operations for GPGPU applications. GPU Gems2. http://developer.nvidia.com/node/17

ПРОДОЛЖЕНИЕ СЛЕДУЕТ...