Оглавление

Введение

Обработка изображений — это совокупность алгоритмов и методов обработки информации, которые получают на вход некие изображения и выдают на выходе другие изображения. Обработка изображений широко используется в компьютерной графике, алгоритмах компьютерного зрения (извлечения информации из изображений, например распознавания текста), создании панорам, видео и других областях.

Функции обработки изображений делятся на несколько основных типов:

- Конвертация изображений между цветовыми моделями и пространствами (RGB, HSL, HSV, Lab), применение цветовых профилей.
- Общие цветовые преобразования, наложение фильтров, изменение цветовых характеристик (контрастности, яркости, насыщенности и т. д.)
- Кодирование и декодирование, изменение изображений с целью уменьшения их размера в памяти с минимальной потерей информации.
- Изменение локальных характеристик: повышение и уменьшение резкости, уменьшение шумов и т. д.
- Первичное выделение информации: выделение границ, движущихся объектов на серии изображений.

Большинство перечисленных функций объединяет несколько характерных свойств:

- *Чистота* (в терминах функционального программирования) — при одинаковом входе (одно или несколько изображений) всегда выдается одинаковый результат.
- Локальность — цвет пикселя в результирующем изображении зависит только от цвета одного соответствующего пикселя входного изображения, либо от небольшой окрестности пикселей.
- У «локальных» функций правила преобразования пикселей не зависят от координат пикселя на изображении, т. е. зависят только непосредственно от цветов и их сочетаний.

Это свидетельствует о том что обработка изображений достаточно хорошо укладывается в парадигму функционального программирования.

Создание библиотеки для обработки изображений на языке Haskell представляет интерес по следующим причинам:

- Так как многие функции обработки изображений естественно описываются в функциональных терминах, при использовании такой библиотеки будет получаться очень краткий и почти на 100% безошибочный код. Эффективность программиста при этом будет очень высокой.
- Некоторые особенности компилятора GHC (Glasgow Haskell Compiler) позволяют автоматически генерировать очень быстрый исполняемый код. Т. е. использование такой библиотеки эффективно не только с точки зрения человека, но и машины, может пригодиться при массовой обработке изображений.
- При использовании такой библиотеки легко взаимодействовать с существующими алгоритмами и библиотеками искусственного интеллекта, нейронных сетей и т. д. на Хаскеле, которых довольно много. Можно с удобством писать алгоритмы машинного зрения на одном языке.

Цель работы — написать быструю библиотеку обработки изображений общего назначения на языке Haskell и доказать ее работоспособность на примерах.

Требования к библиотеке

Базовые функции библиотеки:

- Попиксельное (покомпонентное в формате) преобразование (`map`).
- Попиксельное (покомпонентное в формате) «наложение» 2 или более изображений (`zipWith`). Используется при подавлении шумов, выделении границ, движущихся объектов и в других алгоритмах.
- Применение сверток (`convolution`), т. е. функций от небольшой окрестности входного изображения. Полностью статических, по ядру (например, используется при сглаживании) — stencil convolutions, так и с некоторой логикой (требуется в большинстве алгоритмов более-менее нетривиального локального преобразования).
- «Проход» (правильный перевод «свертка» уже использован в свертках по ядру) по всем пикселям изображения (`fold`). Пример использования — расчет гистограммы, как первый этап выравнивания яркости.

Так как изображения бывают представлены в совершенно разных форматах (цветовых моделях) и нельзя сказать что один из них доминирует, общая библиотека не должна зависеть от какого-то одного формата.

В пределах же одного формата нужна независимость от типа данных для компонент, потому что в разных случаях эффективность их обработки сильно отличается. Ограниченность одним типом данных будет означать медленную работу в определенных случаях. Например, Компоненты формата RGB могут быть любого числового типа, от беззнакового байта (24-битный RGB) до числа с плавающей точкой с двойной точностью, `double`. Построить гистограмму быстрее всего по изображению в формате 24-bit RGB, применить какое-нибудь не дискретное преобразование — точнее и, скорее всего, быстрее к нормализованному RGB.

Таким образом, требование к библиотеке — возможность обрабатывать кортежи (общая абстракция большинства форматов) произвольной длины с произвольными числовыми типами для компонент.

Так как многие функции обработки изображений локальны, быстрая библиотека предполагает возможность распределить работу между несколькими вычислительными потоками, которые могут работать параллельно на многоядерных процессорах.

Чтобы оправдать 1-й пункт «интереса» к себе, библиотека должна давать возможность легко переиспользовать промежуточные результаты и предоставлять достаточно высокоуровневые функции покомпонентного преобразования.

Существующее решение

Существует только 1 библиотека на Хаскеле, которая в целом удовлетворяет представленным требованиям — Repa (Regular Parallel arrays). Это обобщенная библиотека для работы с массивами любой размерности. В ней реализованы все перечисленные базовые функции, возможна работа не только с кортежами чисел, но с данными вообще любого типа. Repa умеет автоматически распараллеливать работу. Она быстрая.

Однако, к сожалению, библиотека не удовлетворяет последнему требованию — в Repa нет удобных функций для покомпонентной обработки. Поэтому реализация даже некоторых простых алгоритмов выглядит довольно громоздко.

Тем не менее, библиотека все равно очень хороша. Реализованная в процессе данной работы библиотека примерно на половину основана на Repa. Основная идея дизайна взята из Repa, повторно реализованы хитрые оптимизации. Принципиально новые решения — обобщенная инфраструктура покомпонетных операций (включая покомпонентное распараллеливание), параметризованные оптимизации. В целом, написанная библиотека еще быстрее, чем Repa.

Дизайн библиотеки

Почти безальтернативным выбором принципа работы библиотеки, как по духу функциональных языков и Haskell в частности, так и по опыту существующих решений — является разделение двух *категорий представлений* изображений:

- *Явные* (manifest) представления инкапсулируют обычные двумерные массивы пикселей. Этот тип представлений является единственным во многих библиотеках обработки изображений на императивных языках.
- *Неявные* (delayed) представления инкапсулируют абстрактную функцию, которая принимает на вход координаты пикселя и возвращает цвет (компоненту формата).

Все базовые функции, возвращающие изображения (`map`, `zipWith`, `convolution`), принимают на вход изображения любого типа и выдают изображения неявного типа. При поддержке Хаскелем функций высших порядков и частично примененных функций это организуется очень просто. Автоматически решается задача возможности переиспользования промежуточных результатов, изображения неявных типов и есть эти результаты.

Естественно, при таком подходе нужна система *вычисляющих функций*, которые переводят изображения неявных типов в явные. Эти функции главным образом разделяются на последовательные (обычные) и параллельные, параметризуемые кол-вом потоков, между которыми будет разделен перевод изображения в явное представление.

Следует отметить, что представление изображений в виде функций от координат и их последовательная композиция и сочетание (при передаче на вход базовых функций неявных изображений) не сказывается на производительности, потому что GHC отлично справляется с редукцией абстрактного синтаксического дерева (deforestration) и генерирует промежуточный код, в котором в принципе отсутствуют частичное применение и композиция функций. (Это полезное «явление» называется fusion.)

Для поддержки пресловутых покомпонетных операций используются элементы так называемой dataflow архитектуры:

- Функция `slices`, которая принимает изображение и возвращает кортеж двумерных массивов компонент в неявном представлении.
- Функция `zip` принимает на вход кортеж массивов компонент и отдает на выходе неявный массив кортежей.

С использованием этих двух функций покомпонентные операции реализуются тривиально:

```
mapComponents f = zip ∘ map f ∘ slices
```

Реализация

Так как Хаскель позволяет легко абстрагировать очень многие аспекты программы, реализованная библиотека обобщена по любой размерности массивов и любому типу данных в качестве элементов (как и прообраз — Repa).

Основные классы и типы данных для массивов

Чтобы базовые функции библиотеки (а так же многие другие вспомогательные функции) могли принимать изображения (массивы) любого типа (разновидности явных и неявных категорий), нужен единый класс для массивов, ассоциированный с семейством типов. Вот как выглядит его определение:

```
class (NFData (UArray r l sh a), Shape sh) ⇒ Regular r l sh a where
    data UArray r l sh a -- Associated data type family

    extent ∷ UArray r l sh a → sh
    touchArray ∷ UArray r l sh a → IO ()
    force ∷ UArray r l sh a → IO ()
```

Класс (и семейство) параметризованы 4 типами: типом-индексом представления (`r`, representation), типом-индексом вычисления (`l`, load), размерностью массива (`sh`, shape) и типом элемента (`a`). Тип-индекс представления нужен как раз чтобы отличать разные представления массивов (реализации) — экземпляры семейства. Тип-индекс вычисления абстрагирует поведение конкретного представления при вычислении в явное представление (об этом ниже).

Условие `NFData (UArray r l sh a)` означает, что экземпляр семейства должен быть полностью вычислимой структурой (в терминах компилятора, а не библиотеки обработки изображений), это хак для обеспечения производительности (ниже). Смысл 2-го предиката очевиден — размерность действительно должна быть размерностью. `Shape` — это вспомогательный класс для размерностей (для линейных массивов, двумерных массив, трехмерных и т. д.), который тут не приведен.

Функция `extent` возвращает размер массива. Еще 2 функции не относятся к абстракции массива или изображения, это тоже оптимизационные хаки (ниже).

Класс Regular специализируется 2 более конкретными классами: USource и UTarget — для массивов, которые могут быть индексированы и для массивов, в которые может осуществляться запись (изменяемые), соответственно. Такое разделение необходимо, например, потому что неявные массивы неизменяемы. С другой стороны, в такой системе могут быть определены «неявные изменяемые» массивы, инкапсулирующие абстрактную записывающую функцию (write), которые могут оказаться полезными.

Классы USource и UTarget определены следующим образом:

```
class Regular r l sh a ⇒ USource r l sh a where
    index ∷ UArray r l sh a → sh → IO a
    index arr sh = linearIndex arr (toLinear (extent arr) sh)

    linearIndex ∷ UArray r l sh a → Int → IO a
    linearIndex arr i = index arr (fromLinear (extent arr) i)

class Regular tr tl sh a ⇒ UTarget tr tl sh a where
```

```
    write :: UArray tr tl sh a → sh → a → IO ()
    write tarr sh = linearWrite tarr $ toLinear (extent tarr) sh


    linearWrite :: UArray tr tl sh a → Int → a → IO ()
    linearWrite tarr i = write tarr $ fromLinear (extent tarr) i

-- Functions toLinear and fromLinear are defined in Shape class.
```

Варианты функций index и write очевидного назначения с префиксом linear- используют вместо «координат» (составного индекса нужной размерности) линейное представление, т. е. гипотетическое смещение адреса нужного элемента относительно адреса начала массива с учетом размера, если бы он располагался в памяти единым блоком. Смещение гипотетическое, т. к. при наличии неявных представлений в общем случае массив может вообще не располагаться в памяти. Эти функции не входят в абстракцию массивов, но удобны:

- Функции index и linearIndex (аналогично write и linearWrite) определены по умолчанию циклично друг через друга, поэтому конкретному представлению при реализации класса достаточно определить 1 любую из них. Для представлений на основе обычных массивов и указателей проще определить «линейную» функцию.
- В определенных случаях можно проще и эффективнее *вычислять* массив с итерацией не по «настоящему», а по линейному индексу (об этом ниже).

Основные операции библиотеки

Упомянутая выше функция slices, необходимая для реализации покомпонентных операций в форматах вроде RGB, определена в классе для массивов векторов (т. е. кортежей с одинаковым типом компонент, ниже они будут называться только «векторами») :

```
class (Regular r l sh (v e), Regular slr l sh e, Vector v e) ⇒
        VecRegular r slr l sh v e | r → slr where
    slices :: UArray r l sh (v e) → VecList (Dim v) (UArray slr l sh e)

class (VecRegular r slr l sh v e, USource r l sh (v e), USource slr l sh e) ⇒
        UVecSource r slr l sh v e

class (VecRegular tr tslr tl sh v e,
       UTarget tr tl sh (v e), UTarget tslr tl sh e) ⇒
        UVecTarget tr tslr tl sh v e
```

Дополнительный параметр slr (slice representation) — тип-нидекс представления массивов компонент. При этом *тип вычисления* у массивов векторов и компонент должен совпадать, размерность, очевидно, тоже.

Vector — это контейнерный класс для коротких векторов известного на этапе компиляции размера из библиотеки fixed-vector. Вообще, в реализации эта библиотека используется повсеместно. VecList — реализация класса Vector на основе стандартного списка, Dim — тип, указывающий на *арность* вектора.

Классы UVecSource (для индексируемых массивов векторов) и UVecTarget (для изменяемых массивов векторов) не определяют собственных функций, потому что вся их спецификация отражена в зависимостях.

Функция zip и базовые функции библиотеки определены в очередном классе с несколькими параметрами (упрощенное определение):

```
class Fusion r fr l | r → fr where
    map :: (USource r l sh a, USource fr l sh b)
        ⇒ (a → b)          -- Mapping function
        → UArray r l sh a  -- Source array
        → UArray fr l sh b -- Result delayed array

    zipWith :: (USource r l sh a, USource r l sh b, USource fr l sh c)
            ⇒ (a → b → c)      -- Zipping function
            → UArray r l sh a  -- 1st source array
            → UArray r l sh b  -- 2nd source array
            → UArray fr l sh c -- Result delayed array

    zip :: (USource r l sh a, USource fr l sh b, Arity n, n ~ S n0)
        ⇒ Fun n a b -- Injective wrapper of function
                    -- which accepts n arguments of type a (components)
                    -- and returns value of type b (vector)
        → VecList n (UArray r l sh a) -- Vector of arrays of components
        → UArray fr l sh b            -- Result delayed array of vectors
```

Класс Fusion объединяет пару представлений (с типами-индексами r и fr (fused representation)), которые переводятся друг в друга базовыми функциями. Выходное представление (UArray fr l sh a) — неявное.

Функциональная зависимость в заголовке класса означает, что исходное представление определяет выходное. Это не принципиально, но без такой зависимости при использовании библиотеки пришлось бы постоянно указывать точный тип промежуточных массивов, потому что компилятор не смог бы их вывести, что не удобно.

Как и в случае массивов векторов и связанных с ними массивами компонент, тип вычисления и размерность неявных массивов — результатов базовых операций должны соответствовать исходному представлению.

Тип-индекс вычисления вынесен в параметры класса (3-й параметр l), потому что в некоторых случаях реализация класса при одинаковых типах представления может от него зависеть.

Класс Arity (арность) — это представление неотрицательных чисел на уровне типов. Предикат n ~ S n0, буквально «n is a Successor of some arity n0», означает что функция zip принимает ненулевое кол-во массивов компонент.

Вычисление неявных массивов

Индексы-типы вычисления массивов, которые при большинстве операций бережно инъективно передаются от представления к представлению, позволяют автоматически вычислять неявные массивы в явные наиболее эффективным образом. Полиморфизм по паре типов вычисления (массива-источника и приемника вычисления) обеспечивается еще одним параметризованным классом (упрощенное определение):

```
type Fill i a
    = (i → IO a)      -- Indexing function
    → (i → a → IO ()) -- Writing function
    → i → i           -- Start, end
    → IO ()


Load l tl sh where
    type LoadIndex l tl sh -- Associated type family

    loadP :: Fill (LoadIndex l tl sh) a -- Filling (real worker) function
          → Int                         -- Number of threads
                                        -- to parallelize loading on
          → UArray r l sh a             -- Source array
          → UArray tr tl sh a           -- Target array
          → IO ()

    loadS :: Fill (LoadIndex l tl sh) a -- Filling (real worker) function
          → UArray r l sh a             -- Source array
          → UArray tr tl sh a           -- Target array
          → IO ()
```

Класс параметризован 2 типами вычисления, l и tl (target load index), и размерностью вычисляемых массивов.

Функция loadP (load in Parallel) распределяет вычисление между несколькими потоками, loadS (load Sequentially) вычисляет массив в одном потоке (последовательно).

Первый параметр обеих функций позволяет параметризовать разворачивание циклов, об этом ниже.

Класс Load и индексы-типы вычисления вероятно могут показаться жутким переусложнением, поэтому теперь о том, как именно все это используется. Есть 2 основных типа-индекса вычисления:

- L (linear). Линейный тип вычисления имеют представления массивов (в общем случае многомерных) на основе обычных, одномерных массивов языка Haskell и просто указателей на блок памяти (т. е. массивов из языка C). Такие представления переопределяют функции linearIndex и linearWrite в классах USource и UTarget соответственно.

  Если и неявный массив-источник, и явный массив – результат вычисления имеют тип L, удобнее и быстрее организовать вычисление по «линейному» индексу. Экземпляр класса Load выглядит примерно так:

  ```
  instance Shape sh ⇒ Load L L sh where
      type LoadIndex L L sh = Int -- Linear index is just Int
      loadS linearFill arr tarr =
          linearFill (indearIndex arr) (linearWrite tarr)
                     0 (size (extent arr))
          -- size is a function from Shape class.
      ...
  ```

- SH (Shaped). Обычный тип вычисления для представлений, которые принимают на вход композитный индекс (координаты пикселя на изображении). Такие

представления переопределяют функции index и write в классах USource и UTarget.

Если хотя бы одно из представлений при вычислении, входное или выходное, имеет такой тип вычисления, для того чтобы избежать дорогих операций перевода линейного представления в координатное с использованием размеров входного массива (включающие операции целочисленного деления, выполняющиеся десятки тактов даже на современных процессорах), массивы итерируются по композитному индексу (код упрощен):

```
instance Shape sh ⇒ Load SH SH sh where -- or SH L sh, or L SH sh
    type LoadIndex SH SH sh = sh -- Don't touch “original” index

    loadS fill arr tarr =
        fill (index arr) (write tarr) zero (extent arr)
    -- zero is a function from Shape class.
    ...
```

Конкретные представления массивов

Основные представления массивов (по типу-индексу представления, описания некоторых представлений упрощены):

- F (Foreign) — явное представление, инкапсулирующее указатель на блок памяти для элементов. «Foreign» означает, что содержимое этого блока памяти не управляется средой исполнения GHC. Тип вычисления — L. Элементами могут являться только типы данных, принадлежащие классу Storable (в том числе кортежи чисел, т. е. представления цветовых моделей).

  Хаскель — язык с автоматической сборкой мусора, поэтому в реализации GHC с каждым «сырым» указателем ассоциирован finalizer (как правило просто обертка для процедуры free из стандартной библиотеки языка C), который выполняется средой исполнения после последнего вызова специальной процедуры touchForeignPtr.

  В обычных случаях эта процедура исполняется при каждом обращении к указателю, что бессмысленно (и неэффективно) при вычислении массивов, ведь можно исполнить процедуру только 1 раз, после вычисления, с тем же эффектом. Именно для этого в базовом классе массивов (Regular) определена процедура touchArray. Она вызывается от входного и выходного массивов в конце всех операций вычисления (из класса Load).

  Так как внешние блоки памяти не управляются средой исполнения, их содержимое неконтролируемо изменяемо. Поэтому функции индексации от внешних массивов (и для совместимости во всей библиотеке, см. класс USource) возвращают не чистые значения элементов, а обернутые в монаду IO, т. е. зависящие от состояния «внешнего мира».

- B (Boxed) — явное представление на основе примитивных массивов в GHC. Тип вычисления — L. Тип данных элементов произволен (на самом деле примитивные массивы в GHC — это массивы ссылок на кучу).

- D (Delayed) — главное неявное представление, является оберткой для абстрактной индексной функции, принимающей композитный индекс (координаты) при типе вычисления SH или линейный индекс при типе вычисления L.

- SE (Separate) — это мета-представление (параметризованное другим представлением)

для массивов векторов, которое инкапсулирует вектор массивов компонент. Оно может быть явным или неявным в зависимости от категории исходного представления, также наследует тип вычисления. «Separate» означает, что в случае явного представления массивов компонент содержимое каждого вектора хранится в нескольких разных местах в памяти (раздельно).

- CV (Convoluted) — неявное представление, результат свертки (тип выхода функции convolution).

  При свертках по статическому ядру цвета близких к границам пикселей выходного изображения как бы зависят от несуществующих пикселей за границами исходного изображения. Есть несколько стратегий «доопределения» исходного изображения, но в любом случае при этом все координаты проверяются на невыход за исходные границы. Однако при вычислении «центральных» (на деле почти всех, за исключением тонкой границы ширины в 1-5 пикселей, при типичных линейных размерах изображений порядка 1000 пикселей) пикселей результирующего изображения этими проверками можно пренебречь.

  Поэтому «свернутое» представление хранит 2 индексные функции: «для границ», с проверками на невыход за границы и быструю, без проверок, «для центра», а также координаты верхнего левого и нижнего правого углов центра (с соотв. обобщением на все размерности).

  Чтобы вычислять границы отдельно от центра с использованием соответствующих функций, «свернутое» представление имеет отдельный тип вычисления CVL (convoluted load index).

Оптимизации

Борьба с недостатками GHC

По умолчанию GHC не встраивает (inline) почти ни какие функции, что плохо не только само по себе, так так вызовы ленивых функций в среде исполнения GHC приводят в машинном коде к записи/чтению примерно 10 параметров на стеке, помимо собственно вызова и возвращения из функции, что в сумме занимает около 50 тактов, но и препятствует многим другим оптимизациям.

Поэтому при использовании реализованной библиотеки фактически возможно 2 «режима»:

- Будут встроены *абсолютно все* функции библиотеки и пользователя, тогда производительность операций над массивами (изображениями) будет как минимум сопоставима с производительностью реализации тех же операций на низкоуровневых языках, например C.
- В противном случае программа будет работать примерно со скоростью интерпретации, на 2 порядка медленнее, чем могла бы.

Необходимость все встроить приводит к незначительным неудобствам при написании библиотеки и пользовательского кода, приходится снабжать все функции директивой INLINE, потому что в GHC нет флага вроде -inline-all-functions-by-default.

Более серьезная вытекающая проблема — дублирование кода. Объем генерируемого машинного кода растет экспоненциально с увеличением сложности программы. Полностью решить проблему нельзя, возможны следующие компромиссы:

- Очень аккуратно запрещать встраивание функций, которые порождают дублирование вблизи корня абстрактного синтаксического дерева.
- Заменять чистые функции монадическими, эмулировать изменение состояния. Теоретически это должно препятствовать GHC ветвить АСД, практически компилятор с трудом дает себя обмануть и пренебрегает искусственными побочными эффектами.
- Смириться с достаточно долгим временем компиляции, большим объемом потребляемой при компиляции памяти и большим размером конечного исполняемого файла.

Если GHC встроил все функции и выкинул (fused) все неявные представления, то тело цикла вычисления массива (изображения) содержит похожую последовательность действий:

1. обращение по одному или нескольким индексам к одному или нескольким первичным явным массивам-источникам,
2. некоторая логика над полученными значениями (цвета, компоненты), например несколько арифметических операций,
3. (покомпонентная) запись результирующего значения в выходной явный массив.

Последняя существенная неэффективность в том, что структуры представлений исходных массивов вычисляются непосредственно перед индексацией, на каждой итерации. Например, в самом лучшем случае, если GHC соптимизировал «внешнее» (Foreign) представление так, что в структуре данных хранится необернутый указатель, в теле цикла вычисления происходит двойная косвенная адресация, когда можно было бы обойтись обычной, один раз

переместив указатель из структуры в регистр за пределами цикла.

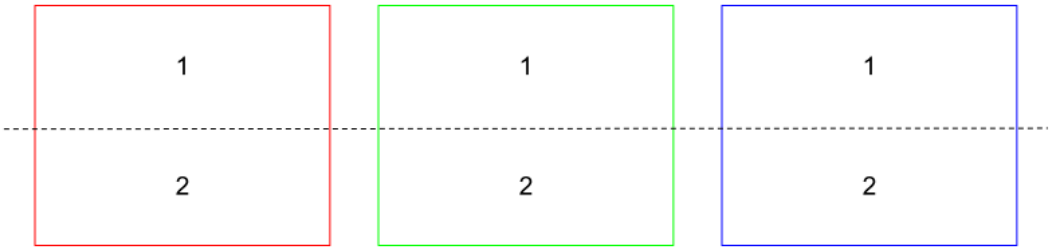
Если GHC видит «подсказку», вычисление структуры данных за пределами цикла, и вообще просто где-то выше по ACT, то все-таки убирает все повторные вычисления. Именно для этого экземпляры семейства типов UArray должны быть членами класса NFData для вычислимых структур данных.

Но вычисление структуры массивов неявных представлений ничего не дает, эти представления и так выкидываются компилятором. Функция force — своего рода «глубокое» вычисление структуры, для неявных представлений за ней скрыто последовательное вычисление структур всех реальных явных источников данного неявного массива. Для явных представлений force — обертка функции вычисления структуры из класса NFData.

Функция force вызывается от массива-источника и явного выходного массива в начале всех функций вычисления (из класса Load).

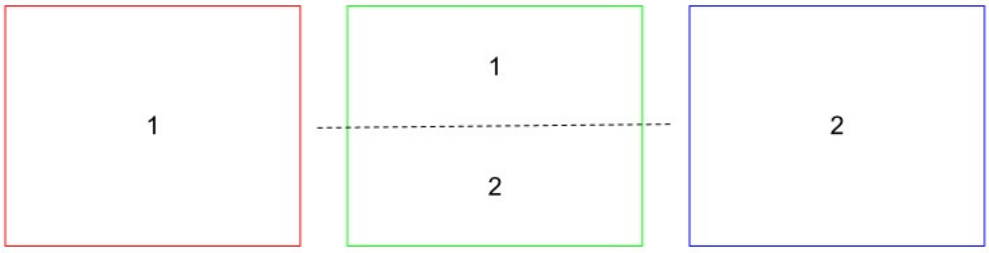
Покомпонентное вычисление массивов

Допустим, есть вычисление, реальный явный источник и результат которого — изображения в формате RGB в раздельном представлении, например, (SE F). Между 2 потоками обычно оно распределяется так:



Цветные прямоугольники означают двумерные массивы компонент, цифры — номера потоков, в которых будет вычисляться блок массива. В каждом потоке выполняется 1 цикл по соответствующим координатам, зато на каждой итерации индексируются и записываются все 3 цветовые компоненты.

В определенных случаях эффективнее распределять вычисление компонент между потоками по отдельности, в данном случае, так:



В пределах потока соответствующие части массивов компонент вычисляются одна за другой, в конкретный момент в конкретном потоке обрабатывается 1 массив компонент.

Преимущества покомпонетного вычисления:

- Лучшая локальность по памяти в пределах потока. Вообще, покомпонентно можно вычислять и явные массивы не раздельного представления, тогда это преимущество теряется.
- Если со значениями компонент выполняется сложная логика или в процессоре мало

отдельных вычислительных конвейеров или модулей чтения/записи, обработка 3-4 компонент одновременно может тормозиться ограниченностью вычислительного потенциала процессора. Покомпонентное вычисление вряд ли приведет к таким проблемам.

- При покомпонентном вычислении лучше работают (или в принципе получают шанс на осуществление) другие оптимизации, например векторизация цикла и GVN (об этом ниже).

Недостаток: в каждом потоке происходит больше итераций, чем при обычном вычислении. В примере выше — в 3 раза, итерируется полтора двумерных массива вместо половины.

В целом, покомпонентное вычисление применимо нечасто. Одна из немногих ситуаций, в которой оправдано покомпонентное вычисление — сглаживание цветного изображения.

Покомпонентное вычисление реализовано через отдельный класс для типов вычисления (определение сильно упрощено):

```
class Load l tl sh ⇒ VecLoad l tl sh where
    loadSlicesP
        :: Fill (LoadIndex l tl sh) e -- Filling function to work /on slices/
        → Int                         -- Number of threads
                                      -- to parallelize loading on
        → UArray r l sh (v e)         -- Source array of vectors
        → UArray tr tl sh (v2 e)      -- Target array of vectors
        → IO ()

    loadSlicesS
        :: Fill (LoadIndex l tl sh) e -- Filling function to work /on slices/
        → UArray r l sh (v e)         -- Source array of vectors
        → UArray tr tl sh (v2 e)      -- Target array of vectors
        → IO ()
```

Параметизованное разворачивание циклов и GVN

Библиотека позволяет разворачивать циклы загрузки на любое статически известное число итераций. В классе Shape определены 2 функции, которые могут передаваться 1-м параметром во все функции вычисления (в классах Load и VecLoad): одна напрямую, другая — после частичного применения:

```
class ... ⇒ Shape sh where
    ...

    fill :: Fill sh a -- Vanilla filling without loop unrolling

    unrolledFill :: ∀ uf a. Arity uf
                 ⇒ uf           -- Unroll factor
                 → (a → IO ()) -- Special “touch” function to suppress
                                -- GHC's excessive AST transformations
                 → Fill sh a    -- Curried result function
                                -- to be passed to loading functions
```

Реализация параметризованного разворачивания основывается на очень мощной редукции ACT, которую может проводить GHC. Создается вектор (не случайно тип параметра

разворачивания — арность из бибилотеки fixed-vector) *действий по вычислению*, который приводится компилятором к виду обычной развертки цикла.

Такое разворачивание применимо при вычислении как по линейному, так и по композитному, «настоящему» индексу.

Выгода от разворачивания циклов очевидна: лучшая утилизация возможностей процессоров с несколькими конвейерами, меньше операций над счетчиком цикла относительно «полезных».

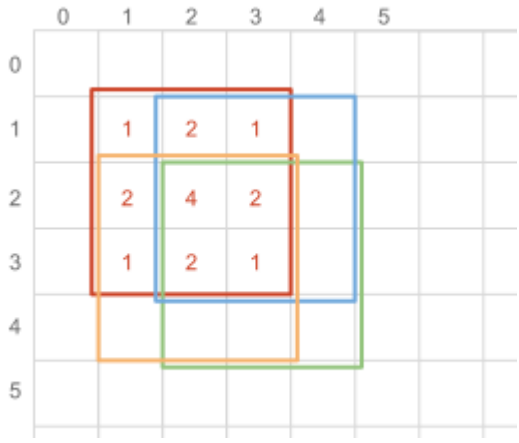
При разворачивании на слишком большое чисто итераций выгода не растет, зато узким местом может стать декодер инструкций.

При создании определенных условий очень большое ускорение приносит оптимизация GVN (Global Value Numbering) компилятора LLVM при вычислении сверток по небольшому ядру. Сам GHC не производит эту оптимизацию, но умеет генерировать код в промежуточном представлении, которое понимает LLVM, и использовать его в качестве бэкэнда.

Для примера возьмем задачу гауссова сглаживания с небольшим радиусом черно-белого изображения, в котором цвет пикселя кодируется беззнаковым байтом. Один из самых быстрых путей решения — свернуть изображение по ядру 3×3, потом для нормализации разделить значения на коэффициент:

$$Image * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16$$

Если подряд вычислить значения 4 соседних пикселей, LLVM может доказать, что при повторных чтениях исходные значения пикселей не могут поменяться (так как между ними нет операций записи) и переиспользовать последние:



Выше изображен исходный двумерный массив интенсивностей пикселей, красный, синий, желтый и зеленый прямоугольники — ядра для вычисления сглаженных значений пикселей по координатам (y=2, x=2), (2, 3), (3, 2) и (3, 3) соответственно. Множители в ядре написаны только для первого пикселя.

Если в процессоре хватает регистров общего назначения, LLVM компилирует такое вычисление с 16 операциями чтения вместо 36, что как раз есть результат срабатывания оптимизации GVN.

В примере используется разворачивание на 2 индекса по горизонтали и на 2 по вертикали. Конечно, это не всегда оптимально. В библиотеке реализована функция для параметризованного разворачивания вычисления двумерных массивов по обоим измерениям со следующей сигнатурой:

```
type Dim2 = (Int, Int)

dim2BlockFill :: ∀ bsx bsy a. (Arity bsx, Arity bsy)
              ⇒ bsx         -- Unroll block size by x
              → bsy         -- Unroll block size by y
              → (a → IO ()) -- Special “touch” function to suppress
                            -- GHC's excessive AST transformations
              → Fill Dim2 a -- Curried result function
                            -- to be passed to loading functions
```

Результаты тестирования разных параметров разворачивания при свертке по ядру [1 4 6 4 1] («горизонтальный» этап гауссова сглаживания при разложении):

Пример — детектор границ Кэнни

Реализация детектора границ Кэнни — удачный способ тестирования библиотеки, потому что в алгоритме используются многие функции библиотеки.

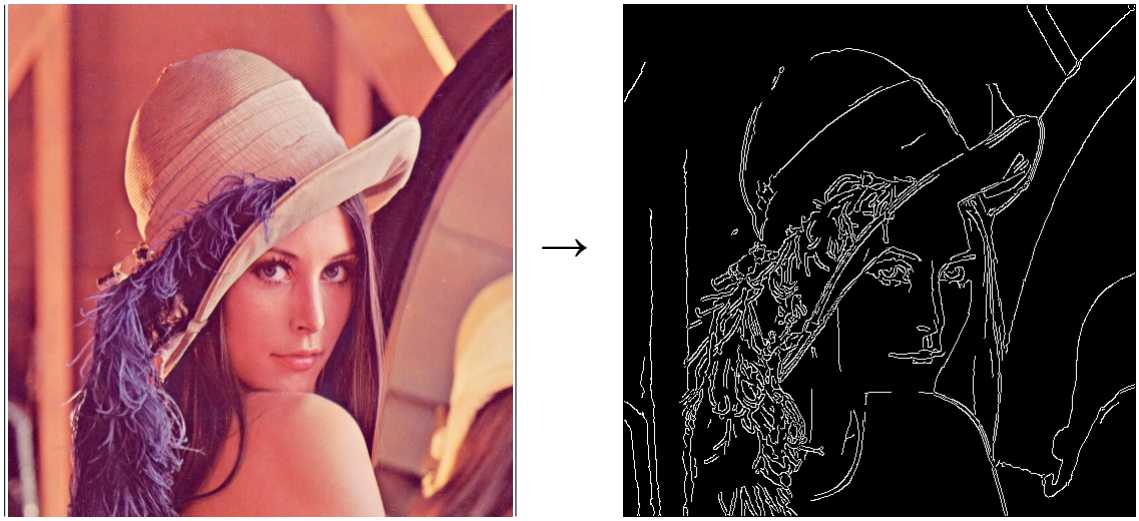
Этапы алгоритма:

1. Обесцвечивание изображения. Реализация через простое отображение.
2. Сглаживание черно-белого изображения. Реализация через последовательную свертку «линейными» ядрами (напр. по горизонтали — как в примере выше).
3. Применение к сглаженному изображению оператора Собеля. Реализация через одновременную свертку 2 разными ядрами размера 3×3, Sobel_X и Sobel_Y, с последующим наложением (zipWith).
4. Выявление максимальных градиентов в окрестностях 3×3. Реализация через свертку с логикой.
5. Распространение границ вокруг точек максимального градиента. Концептуально, реализуется через fold с состоянием, включающим стек необработанных точек.

Все этапы, кроме последнего, могут быть эффективно распараллелены.

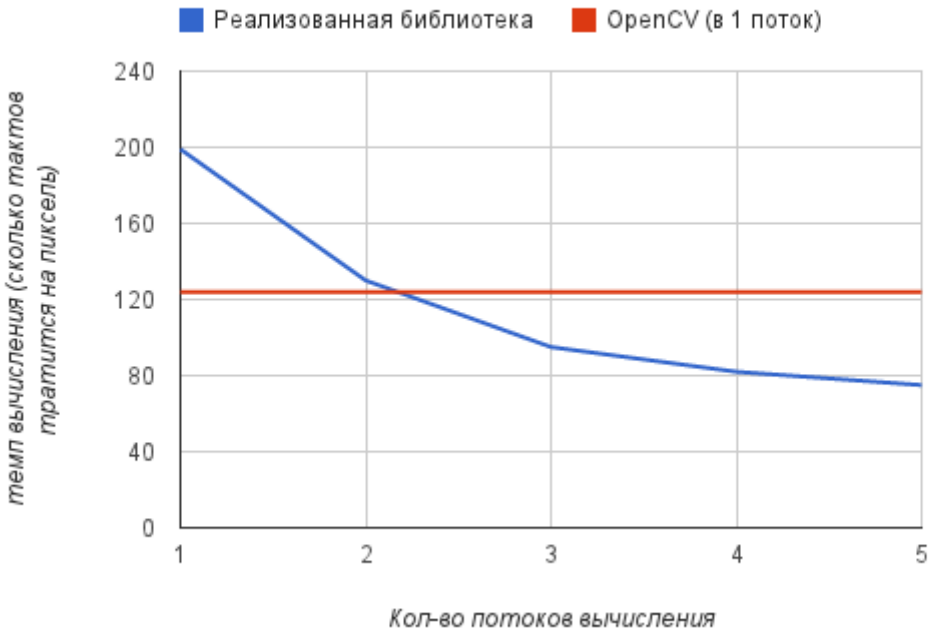
Полный код алгоритма: https://github.com/leventov/yarr/blob/master/tests/canny.hs

Результат работы детектора:



Ниже приведен график результатов сравнения производительности этой реализации детектора границ и реализации в библиотеке OpenCV — индустриальном стандарте в обработке изображений. Обрабатывалась фотография размера 2560×1600.

В последовательном режиме реализация на Хаскеле медленнее OpenCV примерно на 40%, но при распределении работы между 5 потоками она ускоряется почти в 3 раза.


Реализованная библиотека
OpenCV (в 1 поток)
240
200
160
120
80
40
0
1
2
3
4
5
темп вычисления (сколько тактов тратится на пиксель)
Кол-во потоков вычисления

Выводы

Главный вывод — создание быстрой библиотеки обработки изображений на Хаскеле с поддержкой покомпонентных операций возможно. Такая библиотека, собственно, и была написана.

Написать эту библиотеку с нуля было бы крайне сложно. Библиотека «состоит» из библиотеки Repa примерно на 50%, еще на 30% — из библиотеки fixed-vector. Себе я оставляю примерно 20% — работа по соединению этих 2 библиотек и придумывание системы типов. При этом над библиотекой Repa работает целая группа ученых с 2009 года. Библиотека fixed-vector была придумана в 2010 году и реализована только в ноябре 2012 года.

Порог входа в реализованную библиотеку тоже получился очень высокий.

Недостатки библиотеки

Не удалось заставить GHC до конца встраивать функции при покомпонентном вычислении с разворачиванием циклов. В данный момент покомпонентное вычисление работает быстро только при обычных (не развернутых) циклах.

Не производится векторизация циклов. Впрочем, это целиком компетенция компилятора LLVM. В данный момент последняя версия LLVM, с которой работает GHC — 3.1. Возможно, с версии 3.3 LLVM станет нормально векторизовать циклы, что сулит существенное ускорение всей бибилиотеки.

Ссылки

Полный код библиотеки, дополнительных модулей, тестов, результаты замеров: https://github.com/leventov/yarr.

Документация по библиотеке: http://hackage.haskell.org/package/yarr, http://hackage.haskell.org/package/yarr-image-io.

Guiding Parallel Array Fusion with Indexed Types: http://www.cse.unsw.edu.au/~benl/papers/guiding/guiding-Haskell2012-sub.pdf.

Efficient Parallel Stencil Convolution in Haskell: http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/stencil.pdf.

Dataflow-based Design and Implementation of Image Processing Applications: http://www.ece.umd.edu/DSPCAD/papers/shen2011x3.pdf.