Умножение матриц по Винограду на языке Haskell

Анисимов Н.С.

18 февраля 2018 г.

Содержание

1 Постановка задачи

Задача умножения матриц нередко ставится перед разработчиками программного обеспечения. Алгоритмов решения этой задачи достаточно большое количество. Одним из самых эффективных алгоритмов умножения квадратных матриц является алгоритм Винограда, имеющий асимптотическую сложность $O(n^{2,3755})$. Были разработаны также улучшения этого алгоритма.

Реализация этого алгоритма на императивном языке не представляет особых трудностей. В то же время его реализация на языке Haskell может составить программисту некоторые сложности. Дело в том, что Haskell – это чистый функциональный язык. Одной из отличительных особенностей функциональных языков является неизменность состояния. Это означает отсутствие каких-либо переменных, что непременно ведет к отсутствию циклов, которые так активно используются в алгоритмах умножения матриц. Этот недостаток компенсируется множеством встроенных функций работы с массивами и рекурсией.

Другое свойство языка, которое может доставить хлопот программисту при разработке – это ленивые вычисления. Ленивость языка означает, что вычисление не будет выполнено, пока его результат не будет необходим для выполнения следующего действия. Другими словами, выражение на языке Haskell – это лишь обещание того, что оно будет вычислено при необходимости. Одной из проблем ленивых вычислений является использование большого количества памяти, так как необходимо хранить целое выражение, для последующего вычисления.

2 Описание алгоритма

Каждый элемент результирующей матрицы представляет собой скалярное произведение соответствующих строки и столбца.

Рассмотрим два вектора

$$V = (v_1, v_2, v_3, v_4)$$

и

$$W = (w_1, w_2, w_3, w_4)$$

Их скалярное произведение равно:

$$V \times W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4.$$

В то же время:

$$V \times W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1v_2 - v_3v_4 - w_1w_2 - w_3w_4.$$

Во втором выражении требуется большее число вычислений, чем в первом, но оно позволяют произвести предварительную обработку. Выражения v_iv_{i+1} и w_iw_{i+1} для $i \in \overline{0, 2..n}$ можно вычислить заранее для каждой соответствующей строки и столбца. Это позволяет уменьшить число умножений.

3 Реализация

Алгоритм будет реализован с использованием типа `Data.Matrix`. Данный тип предлагает богатый интерфейс для работы с матрицами, и он основан на типе `Data.Vector`, который является стандартным типом для работы массивами с целочисленной индексацией.

В реализации алгоритма умножения матриц в модуле `Data.Matrix`, использована прагма `SPECIALIZE` для типов `Int`, `Double`, `Rational`. Она необходима, чтобы ускорить выполнение функции для данных типов. Надо заметить, что если полиморфная функция реализована внутри модуля `Main`, то использование данной прагмы не является обязательным для ускорения. Реализация алгоритма Винограда будет находиться в отдельном модуле, поэтому так же необходимо использовать эту возможность ускорения.

Наиболее эффективной функцией умножения матриц, представленной модулем `Data.Matrix`, согласно документации, является `multStrassenMixed`. Время ее выполнения и будет взято за эталонное.

Тут график.

Программа скомпилирована с флагом оптимизации `-O2`. Все последующие версии также используют этот флаг. Измерения производились при помощи модуля `Criterion`.

Сигнатура функции будет иметь ожидаемый вид:

```
winograd :: Num a => Matrix a -> Matrix a -> Matrix a
```

Далее представлен самый очевидный способ реализации данного алгоритма:

Рис. 1: Реализация

```
1  winograd1 a b = if n' == n then c else error "error"
2    where
3      m = M.nrows a
4      n = M.ncols a
5      n' = M.nrows b
6      p = M.ncols b
7
8      rows = V.generate m $ \i -> group $ M.getRow (i + 1) a
9      cols = V.generate p $ \j -> group $ M.getCol (j + 1) b
10
11     group v = foldl (group' v) 0 [0, 2 .. V.length v  - 2]
12     group' v acc i =
13       acc - V.unsafeIndex v i * V.unsafeIndex v (i+1)
14
15     c = M.matrix m p $ \(i,j) ->
16           V.unsafeIndex rows (i-1) +
17           V.unsafeIndex cols (j-1) +
18           helper (M.getRow i a) (M.getCol j b) +
19           if odd n then
20             M.unsafeGet i n a * M.unsafeGet n j b
21           else
22             0
23
24     helper r c =
25       foldl (helper' r c) 0 [0, 2 .. V.length r - 2]
26     helper' r c acc i = let
27                           y1 = V.unsafeIndex c (i)
28                           y2 = V.unsafeIndex c (i+1)
29                           x1 = V.unsafeIndex r (i)
30                           x2 = V.unsafeIndex r (i+1)
31                         in
32                           acc +(x1+y2)*(x2+y1)
```

Рассмотрим эту реализацию детальнее. В строках (3) - (6) идет получение размеров матриц для более удобной работы в дальнейшем. Строки (8) - (13) выполняют предварительное вычисление, предусмотренное алгоритмом Винограда. В строках (15) - (22) выполняется главная работа по вычислению матрицы.

Данная реализация не является эффективной. Время ее выполнения остав-

ляет желать лучшего.

Еще один график.

Как видно из графика, она очень сильно уступает функции `multStrassenMixed`. Одна из проблем заключается в лишних вызовах функций `getRow` и `getCol` в строках (8), (9) и (18). Если вызов функции `getRow` является быстрой операцией, согласно документации $\Theta(1)$, то вызов функции `getCol` является затратным – $\Theta(n)$, где n. Более того, функция `getRow` по своей сути является взятием среза из `Data.Vector`, на котором основана матрица, а `Data.Vector` реализует эту операцию эффективно. Функция `getCol`, в свою очередь, требует создания нового вектора, и использует индексирования матрицы. Взятие элемента по индексу в матрице сопровождается большим числом вычислений.

Решить данную проблему можно заранее создав два массива строк и столбцов соответствующих матриц.

Рис. 2: Предварительный вызов `getRow` и `getCol`

```
1       a' = V.generate m $ \i -> M.getRow (i+1) a
2       b' = V.generate p $ \j -> M.getCol (j+1) b
3
4       rows = V.generate m $ \i -> group $ V.unsafeIndex a' i
5       cols = V.generate p $ \j -> group $ V.unsafeIndex b' j
6
7       group v =foldl (group' v) 0 [0, 2 .. V.length v - 2]
8       group' v acc i = acc - V.unsafeIndex v i * V.unsafeIndex v (i+1)
9
10      c = M.matrix m p $
11              \(i,j) ->
12              V.unsafeIndex rows (i-1) +
13              V.unsafeIndex cols (j-1) +
14              helper (V.unsafeIndex a' (i-1)) (V.unsafeIndex b' (j-1)) +
15              if odd n then
16                M.unsafeGet i n a * M.unsafeGet n j b
17              else
18                0
```

Таким образом было получено следующее время:

+график

Результат гораздо лучше предыдущего, но все еще далек от стандартного.

Следующий шаг, который можно предпринять для уменьшения времени выполнения – это замена в функциях `group` и `helper` вызова функции `length` на значение `n`. Можно заметить, что в эти функции передаются только строки

матрицы `a` и столбцы матрицы `b`, а они имеют одинаковую размерность `n`.

Произведя данную замену, можно получить результат, который уже сопоставим с эталонным временем.

+график.

Далее можно заметить, что в функции `matrix` передается лямбда-функция, в которой есть конструкция `if-then-else`. Данная лямбда-функция будет выполнена $m * p$ раз, и следовательно будет выполнено столько же операций сравнения. Это условие можно вынести за пределы лямбда-функции.

Рис. 3: Вынесение `if-then-else` за пределы лямбда-функции.

```
1     c = if odd n then
2           M.matrix m p $
3           \(i,j) ->
4           V.unsafeIndex rows (i-1) +
5           V.unsafeIndex cols (j-1) +
6           helper (V.unsafeIndex a' (i-1)) (V.unsafeIndex b' (j-1)) +
7           M.unsafeGet i n a * M.unsafeGet n j b
8         else
9           M.matrix m p $
10          \(i,j) ->
11          V.unsafeIndex rows (i-1) +
12          V.unsafeIndex cols (j-1) +
13          helper (V.unsafeIndex a' (i-1)) (V.unsafeIndex b' (j-1))
```

После данного улучшение получено время выполнения меньше эталонного.

+график

Данный результат можно улучшить еще больше, избавившись от ленивых вычислений. Значения `a'` и `b'` вычисляются по ходу выполнения программы. Можно вычислить их заранее, тем самым сэкономить время, которое требовалось на поддержку лени.

Рис. 4: Предварительное вычисление `a'` и `b'`.

```
            M.matrix m p $
            \(i,j) ->
            V.unsafeIndex rows (i-1) +
            V.unsafeIndex cols (j-1) +
            helper (V.unsafeIndex a' (i-1)) (V.unsafeIndex b' (j-1)) +
            M.unsafeGet i n a * M.unsafeGet n j b
          else
            M.matrix m p $
            \(i,j) ->
            V.unsafeIndex rows (i-1) +
            V.unsafeIndex cols (j-1) +
            helper (V.unsafeIndex a' (i-1)) (V.unsafeIndex b' (j-1))
```

Для работы функции `deepseq` необходимо изменить сигнатуру:

```
winograd :: (Num a, NFData a) => Matrix a -> Matrix a -> Matrix a
```

Полученное время выполнение получается еще меньше эталонного.

+график

Все предыдущие тесты проводились для матриц с четной общей размерностью. В случае если общая размерность нечетная необходимо провести дополнительные вычисления. В результате время выполнения сильно ухудшается.

+график.

Дело в том, что функция индексирования матрицы, как было замечено раньше, требует много вычислений. Более того, мы уже обращались к соответствующим столбцам и строкам. Поэтому ускорить вычисления можно следующим образом:

Рис. 5: Ускорение вычислений для матриц с нечетной общей размерностью.

```
1     c = a' `deepseq` b' `deepseq` if odd n then
2           M.matrix m p $
3           \(i,j) ->
4           let
5             t1 = V.unsafeIndex a' (i-1)
6             t2 = V.unsafeIndex b' (j-1)
7           in
8             V.unsafeIndex rows (i-1) +
9             V.unsafeIndex cols (j-1) +
10            helper t1 t2 +
11            V.last t1 * V.last t2
12        else
13          M.matrix m p $
14          \(i,j) ->
15          let
16            t1 = V.unsafeIndex a' (i-1)
17            t2 = V.unsafeIndex b' (j-1)
18          in
19            V.unsafeIndex rows (i-1) +
20            V.unsafeIndex cols (j-1) +
21            helper t1 t2
```

В результате получено время выполнения, соизмеримое со времени выполнения этой функции для матриц с четной размерностью.

+график

4 Выводы

В данной работе был реализован алгоритма Винограда на языке программирования Haskell. Были рассмотрены трудности, с которыми может столкнуться программист, при разработке данного алгоритма и варианты их решения. Выполнено 5 итерации улучшения, за которые время выполнения алгоритма было улучшено более чем в десять раз:

- Предварительное получение строк и столбцов соответствующих матриц;
- Замена вызова функции `length` на заранее вычисленное значение;
- Вынос конструкции `if-then-else` за пределы лямбда-функции;
- Избавление от лишних ленивых вычислений;

- Ускорение работы для матриц с нечетной общей размерностью.

Также произведено сравнение с наиболее эффективным алгоритмом умножения матриц, входящих в состав модуля `Data.Matrix`.

В качестве дальнейших вариантов улучшения следуют рассмотреть параллелизацию данного алгоритма и использование других, более специализированных типов данных.

5 Сравнение

Сравнения производились на четырехъядерный процессоре.

Таблица 1:

Size	N=4	N=2	N=1
100	$4.5 \cdot 10^{-3}$	$4.519 \cdot 10^{-3}$	$4.938 \cdot 10^{-3}$
200	$2.178 \cdot 10^{-2}$	$3.021 \cdot 10^{-2}$	$3.194 \cdot 10^{-2}$
300	$6.469 \cdot 10^{-2}$	$9.439 \cdot 10^{-2}$	0.110
400	0.144	0.249	0.251
500	0.265	0.389	0.479
600	0.451	0.625	0.819
700	0.730	0.901	1.285
800	0.986	1.444	1.907
900	1.405	1.786	2.664
1,000	1.934	2.558	3.601