Реализация алгоритма умножения матриц по Винограду на языке Haskell

Анисимов Н.С., МГТУ им. Баумана
anisimoff.nikita@gmail.com
Строганов Ю.В., МГТУ им. Баумана
stroganovyv@bmstu.ru

Аннотация

В данной работе рассматриваются реализации алгоритма умножения матриц на языке программирования Haskell. Рассматривается наивная реализация алгоритма умножения матриц, алгоритм Винограда и его улучшения с помощью использования средств компилятора, уменьшения числа вызовов функций, а также переход от ленивых вычислений к строгим.

1 Введение

Умножение матриц является основным инструментом линейной алгебры и имеет многочисленные применения в математике, физике, программировании. Одним из самых эффективных по времени алгоритмов умножения матриц является алгоритм Винограда, имеющий асимптотическую сложность $O(n^{2.3755})$[1]. Также существуют улучшения этого алгоритма [2] [3].

Haskell – это функциональный язык. Он воплощает понятие чистоты, модель его вычислений основана на концепции “лени”, обладает параметрическим полиморфизмом [4].

Все функции в Haskell представляют собой выражения, подобные математическим. Они не имеют побочных эффектов (за некоторыми исключениями): об этих выражениях принято говорить, что они чистые. Это означает отсутствие каких-либо переменных, что непременно ведет к отсутствию циклов, которые так активно используются в алгоритмах умножения матриц. Данный недостаток компенсируется множеством встроенных функций.

Haskell реализует ленивую модель вычислений. 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.

Параметрический полиморфизм позволяет давать участку кода обобщенный тип, используя переменные вместо настоящих типов, а затем конкретизировать, замещая переменны типами. Параметрические определения однородны: все экземпляры данного фрагмента кода ведут себя одинаково [5].

2 Описание алгоритма

Наивная реализация умножение матриц на языке Haskell представлена в листинге 1.

```
1  mult :: Num a => [[a]] ->
                    [[a]] -> [[a]]
2  mult m1 m2 = result
3    where
4      result = [[sum $ zipWith (*)
     a' b'| b' <- (transpose m2)] |
     a' <- m1]
```

Листинг 1. Наивная реализация умножение матриц

В данном фрагменте `Num a` – это ограничение типа, сообщающее о том, что в качестве типа, может быть использован исключительно числовой тип.

Плата за столь короткую реализацию алгоритма является высокое время выполнения. Для матриц размером 500x500 время выполнения уже выше трех секунд. Одним из способов уменьшения времени выполнения является использование более эффективного алгоритма умножения матриц – алгоритма Винограда.

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

Рассмотрим два вектора:

$$A = (a_1, \dots, a_n) \quad (1)$$

$$B = (b_1, \dots, b_n) \quad (2)$$

Их скалярное произведение равно:

$$A \times B = \sum_1^n a_i\, b_i \quad (3)$$

$$A \times B = \sum_1^2 (a_{2i} + b_{2i+1})(a_{2i+1} + b_{2i}) + t \quad (4)$$

$$t = \sum_1^{n/2} a_i\, a_{i+1} + \sum_1^{n/2} b_i\, b_{i+1} \quad (5)$$

В выражении (4) и (5) требуется большее число вычислений, чем в первом, но оно позволяет произвести предварительную обработку. Выражения $v_i v_{i+1}$ и $w_i w_{i+1}$ для $i \in \overline{0,2..n}$ можно вычислить заранее для каждой соответствующей строки и столбца. Это позволяет уменьшить число умножений [6].

Схема алгоритма для императивной реализации алгоритма представлена на рисунке 1.

Рис. 1. Схема алгоритма Винограда для императивного языка

Для реализации алгоритма Винограда использован тип `Data.Matrix` из пакета `matrix` [7]. Данный тип предлагает обширный интерфейс для работы с матрицами, и он основан на типе `Data.Vector` [8], который является стандартным типом для работы с массивами с целочисленной индексацией.

В реализации алгоритма умножения матриц в модуле `Data.Matrix`, использована директива `SPECIALIZE` [9] для типов `Int`, `Double`, `Rational`. Она явно специализирует функцию под данные типы, чтобы процессорное время не тратилось на обеспечение работы полиморфизма. Так как реализация представлена в виде отдельного модуля, для ускорения работы необходимо специализировать разработанную функцию под некоторые типы, как это сделано в модуле `Data.Matrix`.

Наиболее эффективной функцией умножения матриц, представленной модулем `Data.Matrix` является `multStrassenMixed` [7]. Время ее выполнения взято за эталонное и демонстрировано на рисунке 2.

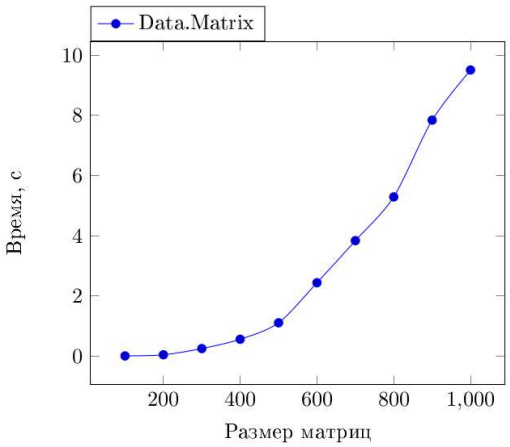

Рис. 2. График времени выполнения multStrassenStd из Data.Matrix

Шаг 1. На листинге 2 представлен самый очевидный способ реализации данного алгоритма.

```
1   winograd :: (Num a) =>
    Matrix a -> Matrix a -> Matrix a
2   winograd a b = if n' == n then c else
    error "error"
3   where
4     m = M.nrows a
5     n = M.ncols a
6     n' = M.nrows b
7     p = M.ncols b
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    group v = foldl (group' v) 0 [0, 2
    .. V.length v  - 2]
11    group' v acc i =
12    acc  -  V.unsafeIndex  v  i  *  V.un
    safeIndex v (i+1)
13    c = M.matrix m p $ \(i,j) ->
14      V.unsafeIndex rows (i-1) +
15      V.unsafeIndex cols (j-1) +
16      helper (M.getRow i a) (M.getCol j
    b) +
17      if odd n then M.unsafeGet i n a *
    M.unsafeGet n j b
18      else 0

19    helper r c = foldl (helper' r c) 0
    [0, 2 .. V.length r - 2]
20    helper' r c acc i = let
21      y1 = V.unsafeIndex c (i)
22      y2 = V.unsafeIndex c (i+1)
23      x1 = V.unsafeIndex r (i)
24      x2 = V.unsafeIndex r (i+1)
25    in
26      acc +(x1+y2)*(x2+y1)
```

Листинг 2. Первая версия реализации алгоритма Винограда

Рассмотрим эту реализацию детальнее. В строках (3) - (6) идет получение размеров матриц. Строки (8) - (13) выполняют предва-

рительное вычисление, предусмотренное алгоритмом Винограда. В строках (13) - (18) выполняется вычисление результирующей матрицы `c`.

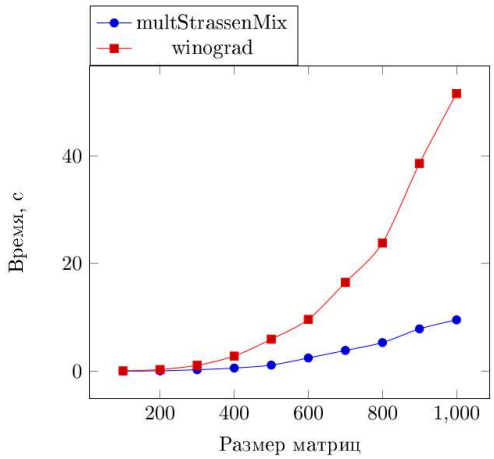

Рис. 3. Время выполнения реализации алгоритма Винограда

Как видно из рисунка 3, она 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 $\theta(1)$, то вызов функции `getCol` является затратным – $\theta(n)$, где n высота матрицы [7]. Более того, функция `getRow` по своей сути является взятием среза из `Data.Vector`, на котором основана матрица, а `Data.Vector` реализует эту операцию эффективно [8]. Функция `getCol`, в свою очередь, требует создания нового вектора, и использует индексирование матрицы. Взятие элемента по индексу в матрице сопровождается большим числом вычислений.

Шаг 2. Решить данную проблему можно заранее создав два вектора строк и столбцов соответствующих матриц, как показано на листинге 3.

На рисунке 4 видно, что результат лучше предыдущего, но все еще далек от эталонного.

```
1   a' = V.generate m $ \i -> M.getRow
    (i+1) a
2   b' = V.generate p $ \j -> M.getCol
    (j+1) b
3   rows = V.generate m $ \i -> group $
    V.unsafeIndex a' i
4   cols = V.generate p $ \j -> group $
    V.unsafeIndex b' j
5   group v =foldl (group' v) 0 [0, 2 ..
    V.length v - 2]
6   group' v acc i = acc - V.unsafeIndex
    v i * V.unsafeIndex v (i+1)

7   c = M.matrix m p $ \(i,j) ->
8     V.unsafeIndex rows (i-1) +
9     V.unsafeIndex cols (j-1) +
10    helper  (V.unsafeIndex  a'  (i-1))
    (V.unsafeIndex b' (j-1)) +
11    if odd n then M.unsafeGet i n a *
    M.unsafeGet n j b
12    else 0
```

Листинг 3. Предварительное вычисление столбцов и строк.

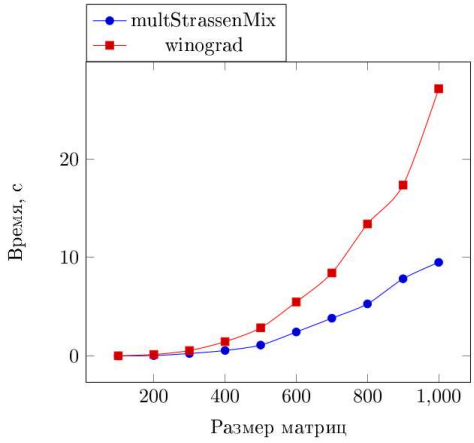

Рис. 4. Время выполнения реализации алгоритма Винограда

Шаг 3. 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, как видно на рисунке 5.

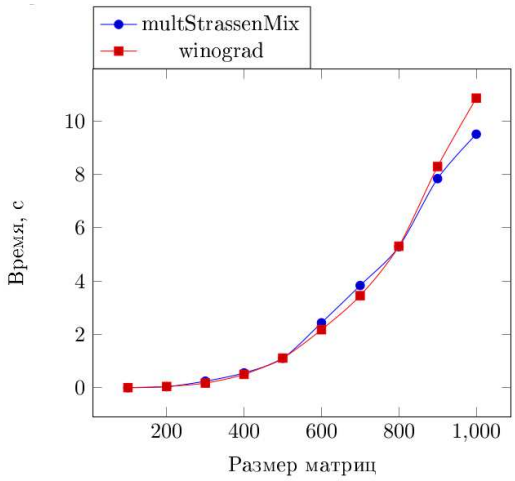

Рис. 5. Время выполнения реализации алгоритма Винограда

Шаг 4. Далее можно заметить, что в функцию `matrix` передается лямбда-функция, в которой есть конструкция `if-then-else`. Данная функция будет выполнена `m*p` раз, и, следовательно, будет выполнено столько же операций сравнения. Это условие может быть вынесено за пределы лямбда-функции.

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

Листинг 4. Перенос конструкции if-then-else за пределы лямбда-выражения.

Рисунок 6 показывает, что улучшение производительности получено, пусть и не высокое.

Шаг 5. 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”.

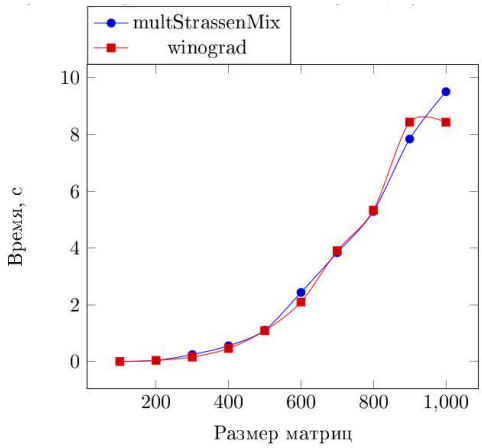

Рис. 6. Время выполнения реализации алгоритма Винограда

```
1   c = a `deepseq` b `deepseq`
2       if odd n then
3         M.matrix m p $ \(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 $ \(i,j) ->
10          V.unsafeIndex rows (i-1) +
11          V.unsafeIndex cols (j-1) +
12          helper (V.unsafeIndex a' (i-
    1)) (V.unsafeIndex b' (j-1))
```

Листинг 5. Предварительное вычисление a’ и b’.

Функция `deepseq` содержится в модуле `Control.DeepSeq`, и производит полное вычисление своего первого аргумента и после этого возвращает второй аргумент. Ее работу можно продемонстрировать на следующем примере:

```
1   Prelude  Control.DeepSeq>  let  a  =
    [1..10] :: [Int]
2   Prelude Control.DeepSeq> :sprint a
3   a = _
4   Prelude Control.DeepSeq> deepseq a 0
5   0
6   Prelude Control.DeepSeq> :sprint a
7   a = [1,2,3,4,5,6,7,8,9,10]
```

Листинг 6. Фрагмент вывода GHCI. Демонстрация работы deepseq.

В первой строке создается "обещание" вычислить список в будущем. `:sprint a` выводит значение без его вычисления. Из результата видно, что оно действительно не вычислено. `deepseq` производит полное вычисление a. Далее снова используется

`:sprint` для вывода значение, и теперь выведено полностью вычисленное выражение.

Для работы функции `deepseq` необходимо изменить сигнатуру:

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

`NFData a` ограничивает тип значения `a` только теми типами, которые могут быть вычислены полностью.

На рисунке 7 видно, что время выполнение получается еще меньше эталонного.

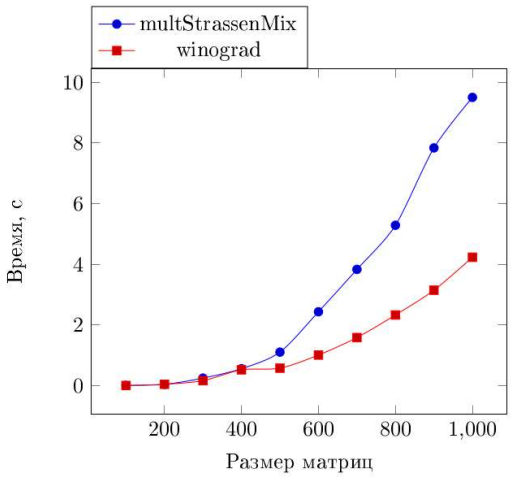

Рис. 7. Время выполнения реализации алгоритма Винограда

Шаг 6. 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, о чем свидетельствует рисунок 8.

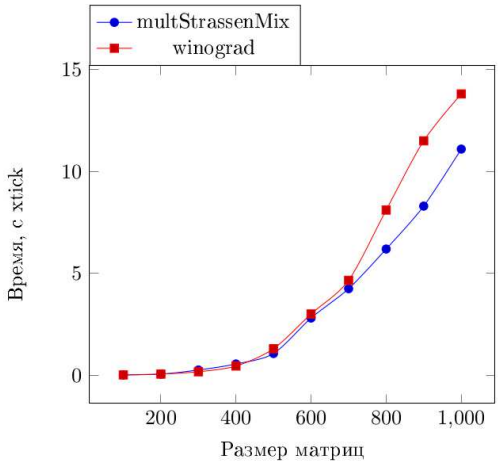

Рис. 8. Время выполнения реализации алгоритма Винограда

Это объясняется тем, что функция индексирования матрицы, требует много вычислений [7]. Более того, уже производилось обращение к соответствующим столбцам и строкам. Поэтому ускорить вычисления можно следующим образом:

```
1   c = a' `deepseq` b' `deepseq`
2       if odd n then
3         M.matrix m p $ \(i,j) ->
4           let
5             v1 = V.unsafeIndex a' (i-1)
6             v2 = V.unsafeIndex b' (j-1)
7           in
8             V.unsafeIndex rows (i-1) +
9             V.unsafeIndex cols (j-1) +
10            helper v1 v2 +
11            V.last v1 * V.last v2
12      else
13        M.matrix m p $ \(i,j) ->
14          let
15            v1 = V.unsafeIndex a' (i-1)
16            v2 = V.unsafeIndex b' (j-1)
17          in
18            V.unsafeIndex rows (i-1) +
19            V.unsafeIndex cols (j-1) +
20            helper v1 v2
```

Листинг 7. Ускорение вычислений для матриц с нечетной общей размерностью.

На рисунке 9 видно, что полученное время выполнения, соизмеримо со временем выполнения этой функции для матриц с четной размерностью.

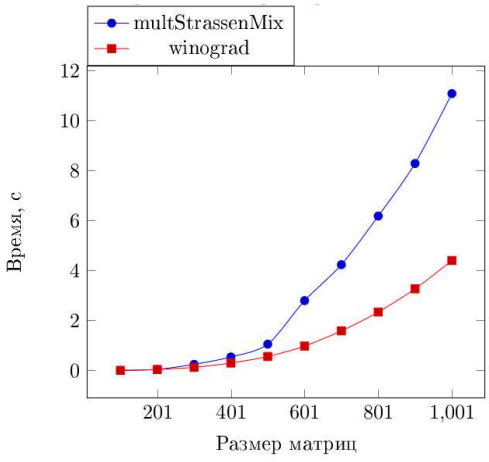

Рис. 9. Время выполнения реализации алгоритма Винограда

3 Заключение

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:

1. Предварительное получение строк и столбцов соответствующих матриц;
2. Замена вызова функции `length` на заранее вычисленное значение;
3. Вынос конструкции `if-then-else` за пределы лямбда-функции;
4. Избавление от лишних ленивых вычислений;
5. Ускорение работы для матриц с нечетной общей размерностью.

Также произведено сравнение с наиболее эффективным алгоритмом умножения матриц, входящих в состав модуля `Data.Matrix`. По результатам этого сравнения было выяснено, что реализованный алгоритм превосходит алгоритм из `Data.Matrix` для размеров матриц до 1000x1000.

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

Список литературы

1. Kakaradov B. Ultra-Fast Matrix Multiplication: An Empirical Analysis of Highly Optimized Vector Algorithms [Электронный ресурс] // cs.stanford.edu: [сайт]. [2004]. URL: https://cs.stanford.edu/people/boyko/pubs/MatrixMult_SURJ_2004.pdf
2. Stothers A.J. On the Complexity of Matrix [Электронный ресурс] // era.lib.ed.ac.uk: [сайт]. [2010]. URL: https://www.era.lib.ed.ac.uk/bitstream/handle/1842/4734/Stothers2010.pdf
3. Williams V.V. Multiplying matrices [Электронный ресурс] // http://theory.stanford.edu: [сайт]. [2014]. URL: http://theory.stanford.edu/~virgi/matrixmult-f.pdf
4. Мена А.С. Изучаем Haskell. Санкт-Петербург: Питер, 2015. 464 pp.
5. Пирс Б. Типы в языках программирования. 655 pp.
6. Алгоритм Копперсмита — Винограда [Электронный ресурс] // ru.math.wikia.com/: [сайт]. URL: http://ru.math.wikia.com/wiki/Алгоритм_Копперсмита_—_Винограда (дата обращения: 23.02.2018).
7. Data.Matrix [Электронный ресурс] // hackage.haskell.org: [сайт]. URL: https://hackage.haskell.org/package/matrix-0.3.5.0/docs/Data-Matrix.html (дата обращения: 23.02.2018).
8. Data.Vector [Электронный ресурс] // haskell.hackage.org/: [сайт]. URL: https://hackage.haskell.org/package/vector-0.12.0.1/docs/Data-Vector.html (дата обращения: 23.02.2018).
9. Chapter 7. GHC Language Features [Электронный ресурс] // Haskell.org/: [сайт]. URL: https://downloads.haskell.org/~ghc/7.0.2/docs/html/users_guide/pragmas.html (дата обращения: 23.02.2018).