МАТЕМАТИЧЕСКОЕ МОДЕЛИРОВАНИЕ,
ЧИСЛЕННЫЕ МЕТОДЫ И КОМПЛЕКСЫ ПРОГРАММ

УДК 004.42, 004.021

ВЕКТОРИЗАЦИЯ АЛГОРИТМОВ ВЫПОЛНЕНИЯ СОБСТВЕННОГО И СИНГУЛЯРНОГО РАЗЛОЖЕНИЙ МАТРИЦ С ИСПОЛЬЗОВАНИЕМ ПРЕОБРАЗОВАНИЯ ХАУСХОЛДЕРА

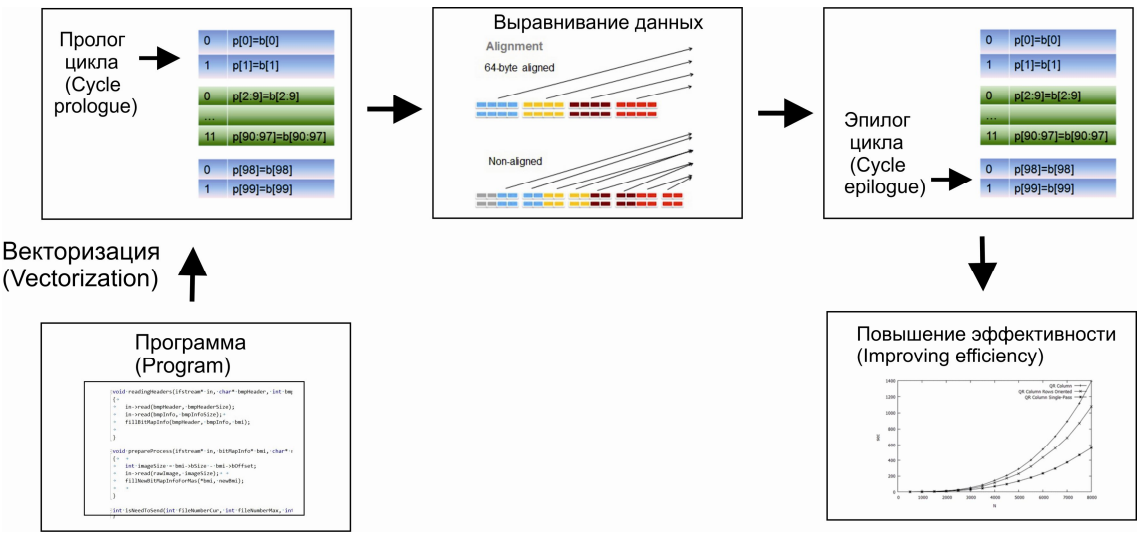
Статья получена редакцией 03.03.2020, в окончательном варианте – 05.05.2020.

Егунов Виталий Алексеевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
кандидат технических наук, доцент, e-mail: vegunov@mail.ru
Андреев Андрей Евгеньевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
кандидат технических наук, заведующий кафедрой ЭВМ и систем, e-mail: andan2005@yandex.ru

Рассматривается векторизация алгоритмов собственного и сингулярного разложений матриц общего вида, в качестве базового преобразования используется преобразование отражения Хаусхолдера. Векторизация вычислений, являющаяся одним из видов их распараллеливания, является эффективным средством повышения быстродействия программ, в рамках которого однопоточные приложения могут выполнять несколько однотипных операций одновременно. Современные компиляторы могут выполнять автоматическую векторизацию вычислений, т.е. преобразовывать программы из скалярного представления к векторной реализации. В статье анализируется эффективность автоматической векторизации, выполненной современными компиляторами, рассматриваются проблемы, присущие автоматической векторизации. Предлагается авторский алгоритм векторизации вычислений для алгоритмов собственного и сингулярного разложений квадратных матриц общего вида. Предложенные решения позволяют значительно повысить скорость программной реализации указанных преобразований. Приводятся особенности процесса векторизации, результаты вычислительных экспериментов.

Ключевые слова: квадратные матрицы, собственное разложение, сингулярное разложение, преобразование отражения, преобразование Хаусхолдера, эффективность программ, ускорение работы программ, векторизация вычислений, автоматическая векторизация, оптимизирующие компиляторы

Графическая аннотация (Graphical annotation)



VECTORIZATION OF ALGORITHMS FOR SOLVING OF EIGENVALUES AND SINGULAR VALUE PROLEMS USING THE HOUSEHOLDER TRANSFORMATIONS

The article was received by the editorial board on 03.03.2020, in the final version – 05.05.2020.

Egunov Vitaly A., Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation,

Cand. Sci. (Engineering), Associate Professor, e-mail: vegunov@mail.ru

Andreev Andrey E., Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation,

Cand. Sci. (Engineering), Head of the Department of Computers and Systems, e-mail: andan2005@yandex.ru

The vectorization of algorithms for proper and singular expansions of General-type matrices, and use the householder reflection transform as the base transformation are considered. Vectorization of calculations, which is a type of parallelization, is an effective tool for improving the performance of programs, in which single-threaded applications can perform several similar operations simultaneously. Modern compilers can perform automatic vectorization of calculations, i.e. convert programs from a scalar representation to a vector implementation. The article analyzes the effectiveness of automatic vectorization performed by modern compilers, and considers the problems inherent in automatic vectorization. The author's algorithm for vectorization of calculations for algorithms of proper and singular expansions of General square matrices is proposed. The proposed solutions allow significantly increasing the speed of software implementation of these transformations. The paper presents the features of the vectorization process and the results of computational experiments.

Keywords: square matrices, eigenvalues, eigenvectors, singular values, reflection transformation, Householder transformation, program efficiency, program acceleration, vectorization, automatic vectorization, optimizing compilers

Введение. Поиск собственных чисел и системы собственных векторов квадратных матриц общего вида, выполнение сингулярного разложения данных матриц широко используются при решении целого ряда задач научного и прикладного характера. В качестве примера можно привести задачи моделирования различных динамических систем, описываемых дифференциальными уравнениями, например, при поиске собственных частот колебаний динамической системы [6, 7]; задачу определения энергетического спектра квантовых систем [8]; обработку изображений [10] и пр. Данная статья посвящена разработке алгоритма векторизации вычислений при выполнении собственного и сингулярного разложений квадратных матриц общего вида. В частности, предлагаются алгоритмы векторизации процесса приведения матрицы общего вида к хессенберговой и двухдиагональной формам с целью дальнейшего вычисления ее собственных или сингулярных значений, о чем в дальнейшем будет сказано подробнее. Данная работа продолжает исследования, отраженные в работах [3–5, 11], направленные на повышение эффективности программной реализации указанных задач на параллельных вычислительных системах с общей памятью.

Постановка задачи. Рассматриваемые задачи имеют высокую вычислительную сложность, что делает их важным объектом исследований для математиков и специалистов в области разработки программного обеспечения. Сами задачи хорошо известны. Задача поиска собственного разложения заключается в поиске векторов и чисел, удовлетворяющих условию (1) [2]:

$$\mathbf{Av} = \lambda\mathbf{v}. \quad (1)$$

Вектор **v** в данном случае называется собственным вектором линейного оператора **A**; число λ – собственным значением. λ и **v** могут быть комплексными даже в случае вещественного **A**. Алгоритмы решения задачи (1) можно разделить на: алгоритмы поиска всех собственных значений и, возможно, системы собственных векторов; алгоритмы поиска нескольких собственных чисел, возможно одного. Наибольшее распространение получили итеративные алгоритмы, заключающиеся в построении последовательности преобразований, сходящейся к собственным значениям, а также в построении последовательности векторов, сходящихся к собственным векторам линейного оператора.

$$\mathbf{A_{k+1}} = \mathbf{U_k A_k U_k^*}, \quad (2)$$

где $\mathbf{U_k}$ – унитарные матрицы; $\mathbf{U_k^*}$ – матрицы, эрмитово-сопряженные к $\mathbf{U_k}$. В вещественном случае используются ортогональные и транспонированные матрицы. Все матрицы $\mathbf{A_i}$ являются подобными, т.е. их собственные значения совпадают. Не существует относительно простых прямых алгоритмов поиска собственных чисел и системы собственных векторов, однако такие алгоритмы известны для матриц специального вида. В качестве примера можно привести треугольные матрицы, у которых собственные числа располагаются на главной диагонали.

Задача поиска сингулярного разложения заключается в поиске векторов и чисел, удовлетворяющих условию (3):

$$\mathbf{A^*u} = \lambda\mathbf{v}. \quad (3)$$

Такие векторы **u** и **v** называются левым и правым сингулярным вектором, соответствующим сингулярному числу λ; **A*** – матрица, эрмитово-сопряженная к **A**. Сингулярным разложением матрицы является разложение вида (4):

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{*}, \quad (4)$$

где**A** – исходная матрица; **U** и **V** – унитарные матрицы; **Σ** – матрица, у которой элементы, лежащие на главной диагонали, являются сингулярными числами. В вещественном случае также используются ортогональные и транспонированные матрицы.

Большинство современных вычислительных систем имеют параллельную архитектуру. Вследствие этого в настоящее время проводится множество исследований, посвященных адаптации известных алгоритмов к использованию на параллельных вычислительных системах, а также исследованию эффективности данных алгоритмов при использовании на параллельных системах с различными характеристиками, например [16, 21]. Помимо распараллеливания вычислений большое внимание уделяется их векторизации. Современные процессоры поддерживают векторные операции [20] в пределах одного вычислительного ядра, что дает дополнительные возможности для ускорения расчетов. В данной работе описан процесс векторизации известных алгоритмов решения задач поиска собственных и сингулярных чисел, в результате применения векторизации сокращается время решения данных задач. Выбор объекта исследований обусловлен востребованностью данных задач при решении различных проблем научного и прикладного характера. Во многих случаях указанные преобразования повторяются многократно, а вычисления продолжаются долго. Поэтому сокращение времени вычислений в данном случае представляется актуальной задачей. В качестве еще одного побудительного мотива к написанию данной работы можно назвать тот факт, что авторы в течение ряда лет занимаются исследованием вопросов, связанных с реализацией матричных преобразований на неоднородных вычислительных системах, в том числе указанных выше преобразований [3–5, 11–12, 17–19].

Используемые методы. Одним из широко известных приемов ускорения процесса вычисления собственных чисел является приведение исходной матрицы общего вида к «почти треугольной» форме, в которой кроме элементов треугольной матрицы сохраняется диагональ под или над главной диагональю. Матрицы такого вида называются верхней или нижней матрицей Хессенберга соответственно. Существует конечная последовательность преобразований, приводящая матрицу произвольного вида к хессенберговой форме с сохранением ее собственных значений. Подобные матрицы, наряду с трехдиагональными матрицами, являются исходными для многих алгоритмов поиска собственных чисел, таких как, например, QR – алгоритм или метод Якоби. Соответственно, можно сделать вывод, что приведение матрицы произвольного вида к хессенберговой, а симметричной матрицы – к трехдиагональной форме, является неотъемлемой частью большого числа алгоритмов поиска собственных чисел линейного оператора. Повышение эффективности методов решения данной задачи приводит к уменьшению времени решения задачи поиска собственных чисел в целом [2].

В случае сингулярного разложения матриц общего вида одним из наиболее эффективных методов является приведение матрицы к двухдиагональному виду с помощью преобразований (4), причем в данном случае **Σ** является двухдиагональной матрицей. В дальнейшем применяются эффективные алгоритмы вычисления сингулярного разложения двухдиагональной матрицы. Соответственно, повышение эффективности методов приведения матриц к двухдиагональному виду приводит к уменьшению времени выполнения сингулярного разложения в целом.

И в (2), и в (4) в качестве операторов **U** и **V** используются унитарные, а в вещественном случае – ортогональные матрицы. В данной работе будет рассмотрена задача приведения матрицы общего вида к хессенберговой и двухдиагональной форме с целью дальнейшего вычисления ее собственных или сингулярных значений.

В качестве базового метода будем использовать метод отражения Хаусхолдера:

$$\mathbf{A} = \mathbf{QRQ}^{\mathrm{T}}, \quad (5)$$

где **A** – исходная матрица; **R** – хессенбергова или двухдиагональная матрица; **Q** – ортогональная матрица, представляющая собой произведение матриц отражения Хаусхолдера, каждая из которых однозначно определяется вектором отражения.

Рассмотрим приведение матрицы размерности $n \times n$ к верхней хессенберговой форме. Для этого необходимо выполнить (n–2) шага, в рамках каждого необходимо:

- вычислить элементы вектора отражения по элементам очередного столбца исходной матрицы;
- умножить исходную матрицу на матрицу отражения справа;
- умножить исходную матрицу на матрицу отражения слева.

Алгоритм выполнения *k*-го шага преобразования можно записать следующим образом (6):

$$\begin{gathered} s_k = -sign(a_{k+1,k})\left(\sum_{i=k+1}^{n} a_{ik}^2\right)^{\frac{1}{2}}, \\ \varphi_k = \left(s_k^2 - s_k a_{k+1,k}\right)^{-1} \\ \boldsymbol{u}_k^T = \left(0, \dots, 0, a_{k+1,k} - s_k, a_{k+2,k}, \dots, a_{nk}\right) \\ \left.\begin{gathered} \lambda_j = \varphi_k \boldsymbol{u}_k^T \boldsymbol{a}_j \\ \boldsymbol{a}_j = \boldsymbol{a}_j - \lambda_j \boldsymbol{u}_k \end{gathered}\right| j = \overline{k,n} \\ \left.\begin{gathered} \lambda_j' = \varphi_k \boldsymbol{b}_j \boldsymbol{u}_k \\ \boldsymbol{b}_j = \boldsymbol{b}_j - \lambda_j' \boldsymbol{u}_k^T \end{gathered}\right| j = \overline{k,n} \end{gathered} \tag{6}$$

Здесь $\boldsymbol{a}$ – столбцы исходной матрицы; $\boldsymbol{b}$ – ее строки; выражение $j = \overline{k,n}$ указывает на диапазон изменения индексов при обработке строк и столбцов матрицы на текущем шаге преобразования. Последовательность шагов алгоритма в данном случае следующая:

- определяется вектор отражения из значений элементов *k*-го столбца матрицы, находящихся под главной диагональю;
- осуществляется умножение на матрицу умножения справа, в результате которого обнуляются элементы *k*-го столбца, изменяются (*n*–1) элементов всех остальных столбцов, начиная с (*k*+1); при этом столбцы матрицы можно обрабатывать параллельно;
- осуществляется умножение на матрицу умножения слева, в результате чего изменяются (*n*–1) элементов всех строк матрицы, начиная с *k*; при этом строки матрицы можно обрабатывать параллельно.

Для приведения матрицы к двухдиагональной форме необходимо выполнить бо́льшее число операций. Это связано с тем, что элементы вектора отражения вычисляются на обоих этапах преобразования. Алгоритм приведения матрицы к верхней двухдиагональной форме можно записать следующим образом (7):

$$\begin{gathered} s_k = -sign(a_{k,k})\left(\sum_{i=k}^{n} a_{ik}^2\right)^{\frac{1}{2}}, \\ \varphi_k = \left(s_k^2 - s_k a_{k,k}\right)^{-1} \\ \boldsymbol{u}_k^T = \left(0, \dots, 0, a_{k,k} - s_k, a_{k+1,k}, \dots, a_{nk}\right) \\ \left.\begin{gathered} \lambda_j = \varphi_k \boldsymbol{u}_k^T \boldsymbol{a}_j \\ \boldsymbol{a}_j = \boldsymbol{a}_j - \lambda_j \boldsymbol{u}_k \end{gathered}\right| j = \overline{k,n} \\ \left.\begin{gathered} s_k' = -sign(a_{k,k+1})(\textstyle\sum_{i=k+1}^{n} a_{ki}^2)^{\frac{1}{2}}, \\ \varphi_k' = \left(s_k'^2 - s_k' a_{k,k+1}\right)^{-1} \\ \boldsymbol{u'}_k^T = \left(0, \dots, 0, a_{k,k+1} - s_k', a_{k,k+2}, \dots, a_{kn}\right) \\ \lambda_j' = \varphi_k \boldsymbol{b}_j \boldsymbol{u}_k' \\ \boldsymbol{b}_j = \boldsymbol{b}_j - \lambda_j' \boldsymbol{u}_k'^T \end{gathered}\right| j = \overline{k,n} \end{gathered} \Bigg\| \; k < n-1 \tag{7}$$

Последовательность действий в целом похожа на последовательность действий при приведении к верхней хессенберговой форме, за исключением:

- при умножении справа для вычисления столбцов $\boldsymbol{a}_j$ при формировании вектора отражения используются элементы столбца под главной диагональю вместе с элементом главной диагонали;
- при умножении слева формируется новый вектор отражения на основе элементов строки, лежащих справа от главной диагонали;
- общее число шагов в данном случае составляет (*n*–1);
- при умножении слева выполняется меньшее число шагов (*n*–2), обозначение $k < n - 1$ в данном выражении указывает на номера шагов, в рамках которых необходимо выполнять данные преобразования.

На рисунке 1 изображены блок-схемы алгоритмов приведения матриц к хессенберговой и двухдиагональной формам.

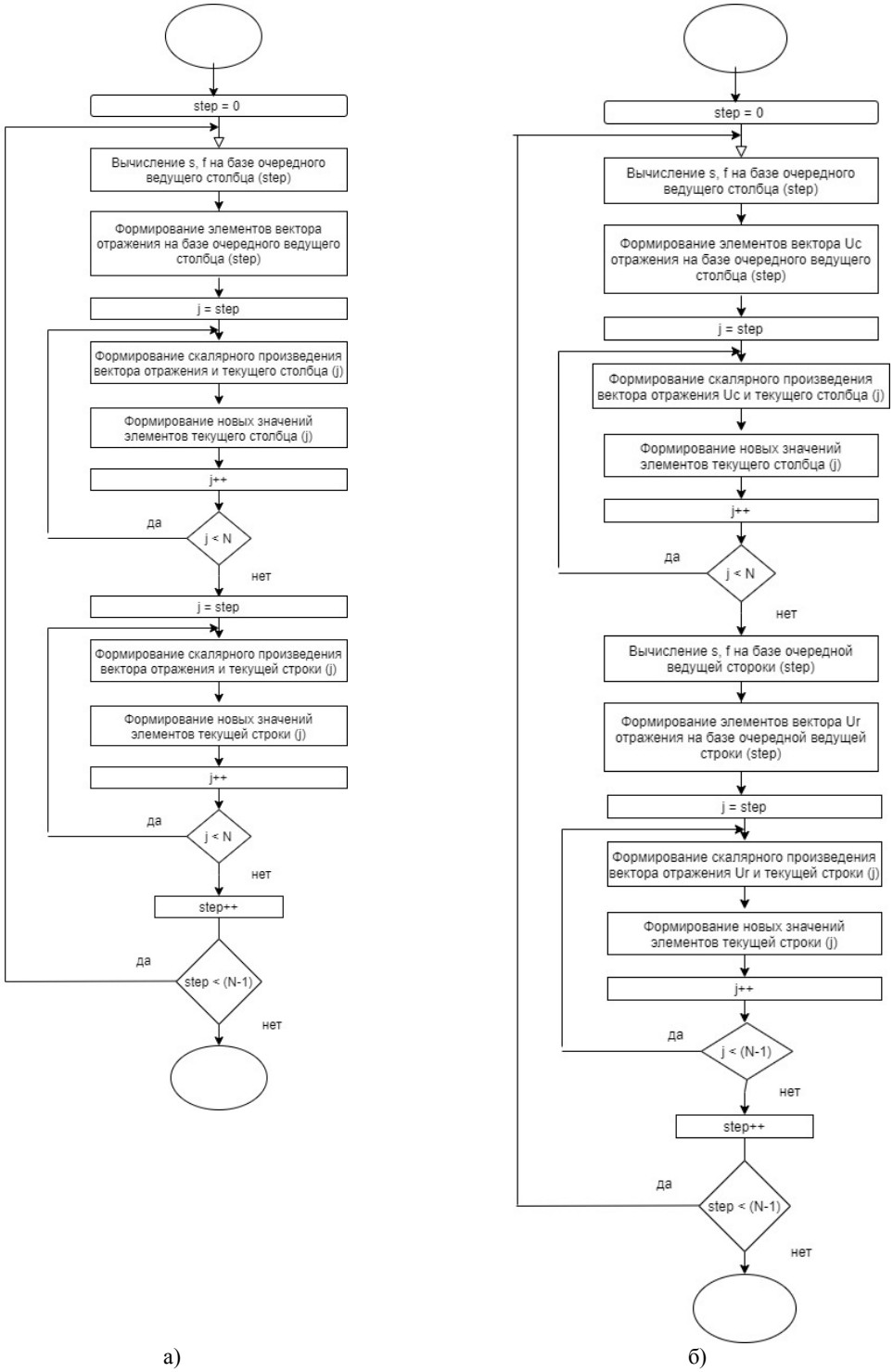


Рисунок 1 – Блок-схемы алгоритмов приведения матриц к хессенберговой (а) и двухдиагональной (б) формам

Как видно из рисунка 1, оба преобразования в целом очень похожи. Существенным отличием является тот факт, что в случае приведения матрицы к двухдиагональной форме на каждом шаге преобразования вектор отражения вычисляется дважды: при преобразовании столбцов на базе элементов ведущего столбца $\boldsymbol{U_C}$ и при преобразовании строк на базе элементов ведущей строки $\boldsymbol{U_R}$. Так как преобразования в целом похожи, сосредоточимся на приведении матрицы к хессенберговой форме. В дальнейшем распространим полученные выводы на алгоритм приведения матрицы к двухдиагональной форме.

Для приведения матрицы к нижней хессенберговой форме на каждом шаге преобразования необходимо выполнить следующие действия (8):

$$
\begin{aligned}
& s_k = -sign(a_{k,k+1})\left(\sum_{i=k+1}^{n} a_{ki}^2\right)^{\frac{1}{2}}, \\
& \varphi_k = \left(s_k^2 - s_k a_{k,k+1}\right)^{-1} \\
& \boldsymbol{u}_k^T = \left(0, \dots, 0, a_{k,k+1} - s_k, a_{k,k+2}, \dots, a_{kn}\right) \\
& \left.\begin{aligned} \lambda_j' &= \varphi_k \boldsymbol{b_j u_k} \\ \boldsymbol{b_j} &= \boldsymbol{b_j} - \lambda_j' \boldsymbol{u_k^T} \end{aligned}\right| j = \overline{k,n} \\
& \left.\begin{aligned} \lambda_j &= \varphi_k \boldsymbol{u_k^T a_j} \\ \boldsymbol{a_j} &= \boldsymbol{a_j} - \lambda_j \boldsymbol{u_k} \end{aligned}\right| j = \overline{k,n}
\end{aligned} \qquad (8)
$$

В целом выполняются те же действия с учетом следующих замечаний:

- вектора отражения формируются на базе элементов строк матриц;
- сначала преобразуются строки, затем столбцы матриц.

Во всех трех приведенных преобразованиях (6)–(8) для написания параллельных программ используется естественный параллелизм по данным, присущим данным методам:

- при умножении справа столбцы матрицы можно обрабатывать параллельно;
- при умножении слева строки матрицы можно обрабатывать параллельно.

Анализу данных преобразований посвящено значительное число публикаций. Большое внимание исследователей привлекает решение таких задач: выполнение собственного разложения для матриц специального вида, например [13–14, 22, 24]; обработка матриц специального вида в целом [13, 21]; использование сингулярного разложения при решении различных технических задач. Исследуется преобразование хессенберговых матриц, в том числе для получения собственного и сингулярного разложений [23, 25]. Традиционно пристальное внимание исследователей привлекает преобразование отражения Хаусхолдера [15, 16, 21, 23]. В основном публикации посвящены применению данного преобразования к решению различных конкретных задач, часто на вычислительных системах определенного типа. В данной статье предлагаются алгоритмы векторизации преобразования отражения Хаусхолдера, осуществляется сравнение полученных результатов с автоматической векторизацией кода современными оптимизирующими компиляторами.

Векторизация вычислений. Под векторными командами процессора понимается применение некоторой операции к набору данных, например, секции массива. Во многих алгоритмах можно выделить некоторую повторяющуюся модель доступа к данным, а также операции, которые могут выполняться над элементами из этого набора. С математической точки зрения данные операции представляют собой операции над некоторыми векторами. Использование таких векторных операций позволяет существенно повысить вычислительную эффективность целого ряда программ. Для реализации векторных операций микропроцессор должен иметь специальные «широкие» или SIMD (Single Instruction Multiply Data) регистры и набор соответствующих инструкций. Речь идет прежде всего об инструкциях наборов SSE и AVX. Каждый из SIMD-регистров может хранить несколько значений в зависимости от используемого типа данных. На рисунке 2 приведен пример упаковки данных в SIMD-регистры для вещественных типов данных языка программирования C++.

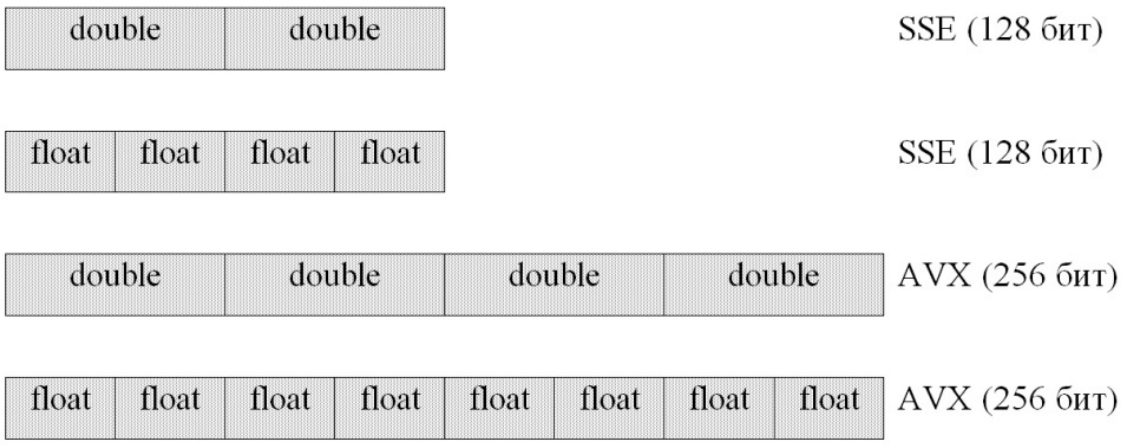


Рисунок 2 – Упаковка данных в SSE- и AVX-регистры

В последних поколениях процессоров Intel SIMD-регистры имеют размер 512 бит. Таким образом, на каждом ядре такого процессора можно одновременно производить действия над шестнадцатью числами с плавающей запятой одинарной точности (float) или с восемью числами с плавающей запятой двойной точности (double), шестнадцатью 32-битными целыми числами или восемью 64-битными целыми числами и т.д.

Операндами векторных инструкций могут быть как сами SIMD-регистры, так и вектора чисел, расположенные в памяти. Для эффективной загрузки этих векторов в SIMD-регистры их адреса в памяти должны удовлетворять определенным условиям: для 128-битных операций адрес должен быть кратен 16, для 256 – кратен 32, для 512 – кратен 64. Если адреса удовлетворяют этим условиям, то говорят, что они являются выровненными. Существует возможность работы и с невыровненными данными, однако это приводит к существенным потерям производительности.

Рассмотрим векторизацию алгоритма приведения матрицы к верхней хессенберговой форме (6). В рамках одного шага преобразования можно выделить четыре цикла:

1) вычисление нормы ведущего столбца;
2) вычисление элементов вектора отражения;
3) обработка столбцов;
4) обработка строк.

Циклы (1) и (2) имеют меньшую вычислительную сложность по сравнению с (3) и (4). При этом циклы (1)–(3) выполняются вдоль столбцов матрицы, цикл (4) – вдоль строк. Считаем, что матрицы хранятся в памяти по строкам, поэтому цикл обработки строк (4) имеет больший потенциал в контексте векторизации. Связано это с тем, что данные, используемые в цикле (4), расположены в памяти непрерывно. Поэтому они могут загружаться в SIMD-регистры операциями векторной загрузки. В циклах (1)–(3) данные не расположены непрерывно. Поэтому они при векторной обработке будут загружаться в SIMD-регистры поэлементно, что отрицательно влияет на производительность. Кроме всего прочего, цикл (4) имеет высокую вычислительную сложность.

В связи с этим основное внимание уделим векторизации данного цикла, полученные решения частично могут быть применены и к остальным циклам. Рассмотрим проблемы векторизации данного цикла на примере матрицы с нечетной размерностью, например, 11×11. Цветом выделим элементы вектора отражения и строк матрицы, попадающих в один регистр AVX (256 бит). При этом считаем, что сами массивы **V** и **A** выровнены по границе 32 байта.

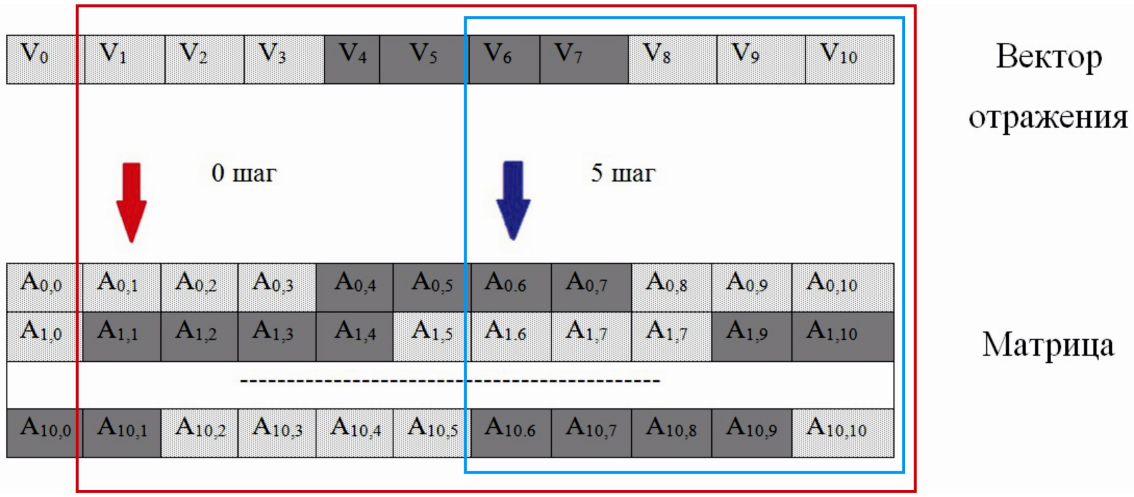


Рисунок 3 – Выполнение очередного шага преобразования

Рассмотрим выполнение нулевого шага преобразования (k = 0 в (6)). На нулевом шаге обработка начинается с первого элемента вектора отражения ($\mathbf{V}_1$), также с первых элементов строк матрицы ($A_{0,1}$, $A_{1,1}$ и т.д.). Проанализируем обработку строк:

- первая строка. Элементы строки и вектора отражения выровнены одинаково, обработка начинается с первого элемента. Необходим пролог цикла для обработки первых трех элементов. Далее векторизованная часть (в данном случае 4 элемента). Затем – эпилог цикла для обработки последних трех элементов;
- вторая строка. Элементы строки и вектора отражения выровнены по-разному, в зависимости от способа обработки начала цикла. Может быть сформирован пролог цикла, далее следует векторизованная часть. Затем – эпилог цикла.

Необходимо пояснить смысл терминов «пролог цикла» и «эпилог цикла». Они относятся к анализу выровненности данных, к которым производится обращение. В общем случае данные находятся в памяти по невыровненным адресам, что потенциально снижает производительность работы программы. Избежать данных потерь позволяет разбиение цикла на три части: пролог, выровненная основная часть, эпилог цикла. Пролог представляет собой несколько первых элементов, которые предшествуют первому элементу, расположенному по выровненному адресу. В свою очередь эпилог представляет собой последние несколько элементов цикла, следующие за последним вектором, расположенным по выровненному адресу.

В качестве иллюстрации обратимся к рисунку 3 и рассмотрим преобразование первой строки. Элементы векторов обозначены одним цветом (по четыре элемента на вектор), причем первый элемент матрицы расположен по выровненному адресу. Тогда при обработке первой строки необходимо сформировать пролог, содержащий три элемента ($A_{0,1}$, $A_{0,2}$, $A_{0,3}$). Далее следует основной цикл, содержащий в данном случае один выровненный вектор ($A_{0,4}$, $A_{0,5}$, $A_{0,6}$, $A_{0,7}$), и эпилог, содержащий три элемента ($A_{0,8}$, $A_{0,9}$, $A_{0,10}$).

При выполнении 5-го шага преобразования наблюдается такая же картина, т.е. необходимо формирование пролога и эпилога цикла, присутствует различное выравнивание элементов вектора отражения и строк матрицы.

Современные оптимизирующие компиляторы осуществляют автоматическую векторизацию кода программ. При этом схема векторизации циклов в общем случае выглядит примерно следующим образом:

- формируется невекторизованный пролог цикла, после выполнения которого текущие элементы обрабатываемых массивов находятся по выровненным адресам;
- векторизуется средняя часть цикла;
- формируется невекторизованный эпилог, обрабатывающий последние элементы, число которых меньше ширины SIMD-регистра;
- если в цикле используются несколько массивов и выровнены они по-разному, то компилятор выбирает один из них. Обработка данного массива будет осуществляться с использованием выровненных считываний из памяти, а обработка остальных – с использованием невыровненных обращений. Пролог и эпилог цикла формируются на основе выбранного массива.

Компиляция исходной программы, реализующей приведение матрицы к верхней хессенберговой форме, осуществлялась компилятором Intel C++ Compiler. В результате автоматической векторизации был получен значительный прирост производительности, о чем будет сказано ниже.

Однако компилятор не может преодолеть описанные выше ограничения, заключающиеся в необходимости формирования невекторизованных прологов и эпилогов циклов, а также в необходимости использования невыровненных массивов в том случае, если вектор отражения и строка матрицы выровнены по-разному. Повысить эффективность векторизации можно, решив проблемы с формированием неэффективных пролога и эпилога цикла, а также выполнив взаимное выравнивание элементов вектора отражения и элементов строк матрицы. Здесь необходимо рассмотреть три случая:

- размерность матрицы кратна ширине SIMD-регистра; например, в случае использования типа данных double и технологии AVX – кратна 4, при использовании AVX и float – кратна 8 и т.д.;
- размерность матриц четная, но не кратна ширине SIMD-регистра, например, 10 для AVX и double;
- размерность матриц нечетная.

Для каждого из этих случаев будет использоваться свой алгоритм векторизации. Проанализируем каждый из этих случаев по отдельности.

На рисунке 4 приведено выполнение шага преобразования для кратной размерности матрицы. Для примера используется матрица 12×12.

Анализируя рисунок 4, можно сделать следующие выводы:

- элементы вектора отражения и строк исходной матрицы выровнены одинаково;
- эпилог цикла отсутствует в силу того, что длина строки кратна ширине SIMD-регистра (в данном случае 4).

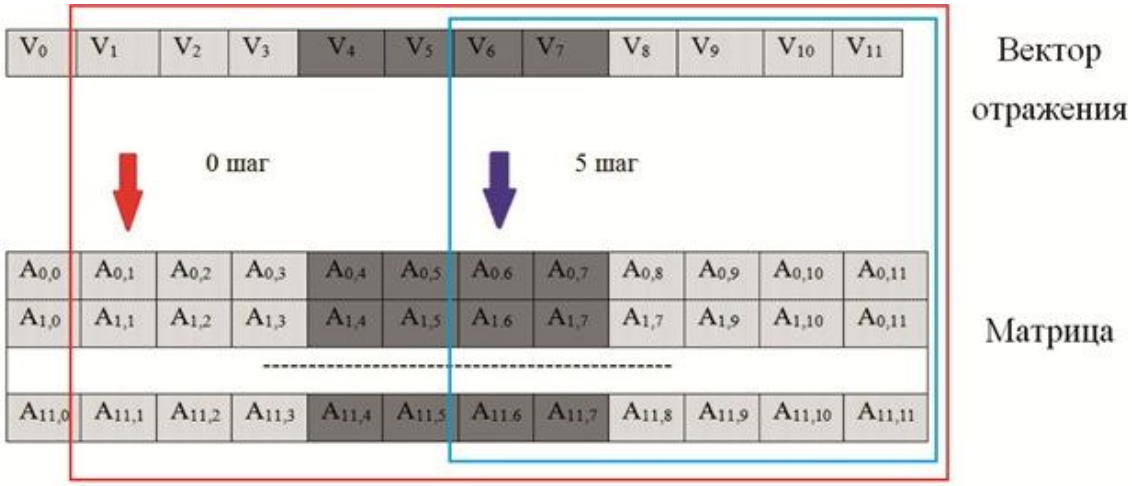


Рисунок 4 – Выполнение шага преобразования для матрицы с размерностью, кратной ширине SIMD-регистра

Несколько повысить эффективность реализации данного цикла можно, избавившись от пролога цикла.

Рассмотрим исходный невекторизованный код обработки строк.

```
for(int row = step; row < N; row++)
{
double scalar = 0;
for(int i = (step + shift); i < N; i++)
scalar += matr[row * N + i] * vect[i];
scalar *= gamma;
for(int i = (step + shift); i < N; i++)
matr[row * N + i] -= scalar * vect[i];
}
```

Для каждой строки матрицы сначала вычисляется скалярное произведение с вектором отражения, затем вычисляются новые значения элементов строки. В строке обрабатываются элементы, начиная с (step + shift), где step – номер шага преобразования, shift = 1 для приведения матрицы к хессенберговой форме. Очевидно, что обработка в общем случае начинается с невыровненного адреса. Однако если учесть, что все элементы вектора отражения с индексами, меньшими (step + shift), в соответствии с алгоритмом (6) равны нулю и данные нулевые значения нужно записывать в необходимые элементы V при его формировании, то обработку можно начинать с ближайшего меньшего выровненного индекса. В случае нулевого шага обработку можно начинать с нулевого элемента, на пятом шаге – с четвертого. Код с использованием соответствующих директив компилятора в данном случае будет выглядеть следующим образом.

Формирование скалярного произведения:

```
for(int row = step; row < N; row++)
{
// Скалярное произведение
double scalar = 0;
#pragma vector aligned
for(int i = stepShift - alignVect; i < N; i++)
scalar += matr[row * N + i] * vect[i];
scalar *= gamma;
#pragma vector aligned
for(int i = stepShift - alignVect; i < N; i++)
matr[row * N + i] -= scalar * vect[i];
}
```

Рассмотрим матрицы с четной размерностью, величина которой не кратна ширине SIMD-регистров. Для примера рассмотрим матрицу 10×10.

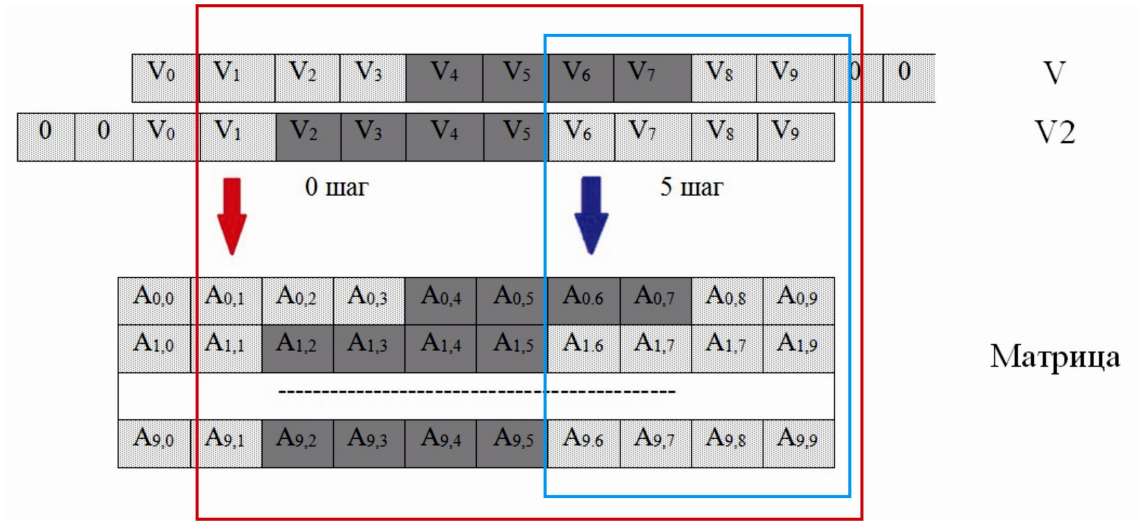


Рисунок 5 – Выполнение шага преобразования для матрицы с четной размерностью, не кратной ширине SIMD-регистра

Анализируя рисунок 5, можно сделать следующие выводы:

- элементы вектора отражения и строк исходной матрицы в общем случае выровнены по-разному;
- эпилог цикла необходим (например, элементы $A_{0,8}$ и $A_{0,9}$ первой строки).

Избавиться от пролога цикла можно так же, как и в предыдущем случае, т.е. путем смещения первого обрабатываемого элемента в сторону первого меньшего выровненного индекса. Однако такой подход сработает только для строк с четными номерами (0, 2 и т.д.), так как они выровнены так же, как и вектор отражения. Для строк нечетными номерами необходимо сформировать еще один вектор отражения **V2**, как показано на рисунке 5. В этом случае, например, обработка элемента $A_{1,1}$ будет осуществляться совместно с $A_{0,8}$, $A_{0,9}$ и $A_{1,0}$. Но за счет наличия ведущих нулей в векторе **V2** результат будет корректным.

В строках с четными номерами необходимо формировать эпилог, однако за счет добавления нулей в конец вектора **V** результат также будет корректен. Цикл обработки строк в данном случае трансформируется в два цикла – обработки строк с четными и нечетными номерами.

```
alignVect = stepShift % 4;
for(int row = rowEven; row < N; row += 2){
double scalar = 0;
#pragma vector aligned
for(int i = stepShift - alignVect; i < (N + 2); i++)
scalar += matr[row * N + i] * vect[i];
scalar *= gamma;
#pragma vector aligned
for(int i = stepShift - alignVect; i < (N + 2); i++)
matr[row * N + i] -= scalar * vect[i];}
alignMatr = (rowNotEven * N + stepShift) % stride;
for(int row = rowNotEven; row < N; row += 2){
double scalar = 0;
#pragma vector aligned
for(int i = stepShift - alignMatr; i < N; i++)
scalar += matr[row * N + i] * vect2[i + 2];
scalar *= gamma;
#pragma vector aligned
for(int i = stepShift - alignMatr; i < N; i++)
matr[row * N + i] -= scalar * vect2[i + 2];}
```

Для матриц с нечетной размерностью возможны два алгоритма. Они представлены на рисунках 6 и 7.

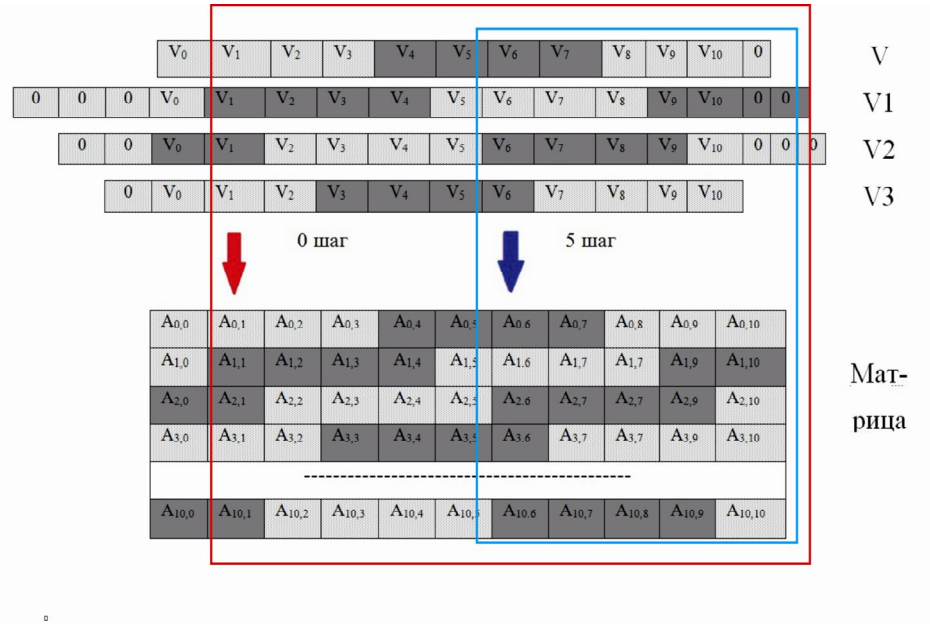


Рисунок 6 – Выполнение шага преобразования для матрицы с нечетной размерностью

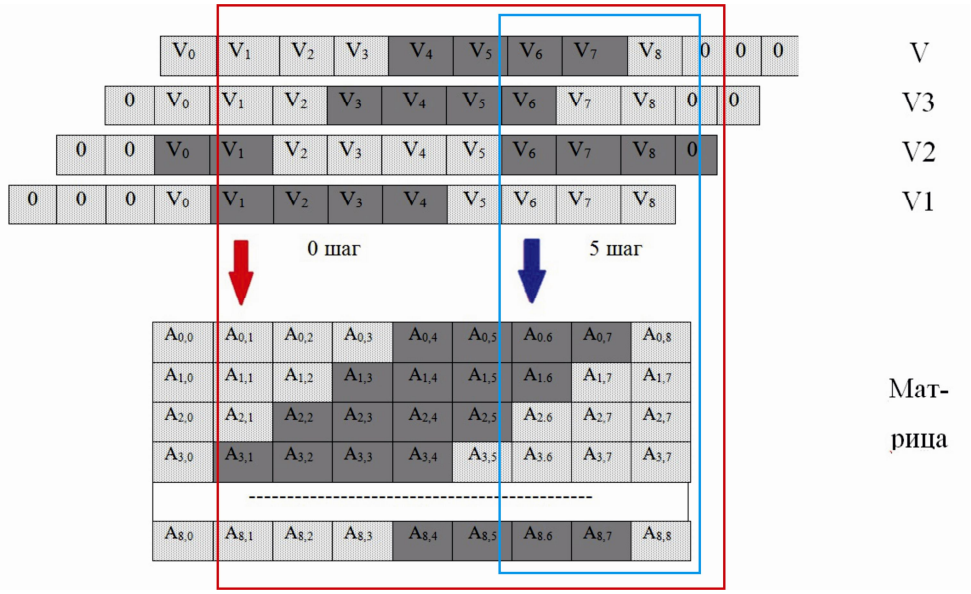


Рисунок 7 – Выполнение шага преобразования для матрицы с нечетной размерностью (вариант 2)

В данном случае необходимо формировать четыре цикла в рамках цикла обработки строк. При этом в каждом цикле используется свой вектор отражения.

Результаты вычислительных экспериментов. В качестве объекта исследований была выбрана параллельная вычислительная система с общей памятью, построенная на базе многоядерных микропроцессоров Xeon E5-2650v3(x2) 2.3 GHz. Эта система входит в состав вычислительного кластера Волгоградского государственного технического университета. В качестве компилятора использовался Intel C++ Compiler version 18.

Эксперименты проводились следующим образом. Отдельно для матриц четной кратной, четной некратной и нечетной размерности:

- осуществлялась компиляция исходного кода без векторизации (ключ компилятора no-vec);
- осуществлялась компиляция исходного кода с автоматической векторизацией (ключ компилятора xAVX);
- осуществлялась компиляция кода, векторизованного с использованием описанных выше алгоритмов;
- осуществлялся замер времени выполнения программ на ряде значений размерностей;

- определялись ускорения полученных программ, векторизованных автоматически и с использованием предложенных алгоритмов.

Отдельно определялось ускорение для векторизуемого цикла и для всего преобразования в целом. Были получены следующие результаты (табл. 1, 2).

Таблица 1 – Полученные значения ускорения для векторизуемого цикла

Размерность матриц	**Автоматическая векторизация**	**Ручная векторизация**
N = 100	2,0	2,7
N = 200	1,9	2,3
N = 500	1,85	2,1
N = 1000	1,8	2,0
N = 1500	1,75	1,9
N = 2000	1,7	1,8
N = 3000	1,43	1,5
N = 4000	1,36	1,4
N = 5000	1,33	1,34
N = 6000	1,30	1,32

Таблица 2 – Полученные значения ускорения для приведения матриц к хессенберговой форме в целом

Размерность матриц	**Автоматическая векторизация**	**Ручная векторизация**
N = 100	1,55	1,7
N = 200	1,5	1,55
N = 500	1,4	1,45
N = 1000	1,17	1,2
N = 1500	1,15	1,18
N = 2000	1,13	1,15
N = 3000	1,1	1,12
N = 4000	1,08	1,1
N = 5000	1,07	1,09
N = 6000	1,06	1,08

Анализируя полученные данные, можно сделать следующие выводы:

- для векторизуемого цикла максимальное значение ускорения, полученное при ручной векторизации в соответствии с предложенным алгоритмом, достигает 2,7 при асимптотическом значении 4 (соответствует ширине SIMD-регистра). Это на 70 % превышает максимальное значение ускорения, полученное при автоматической векторизации;
- для преобразования в целом значения получено меньшее ускорение. Это связано с необходимостью выполнения ряда других циклов, гораздо менее перспективных с точки зрения векторизации. Например, циклов обработки столбцов матрицы.

Следует также отметить, что при увеличении размерности матрицы значение ускорения значительно снижается. При использовании матриц, размерность которых превышает 4000, ускорение от векторизации не превышает значения 10 %. Снижение производительности связано с тем, что матрицы подобных размерностей не помещаются целиком в кэш-память процессора и при обращении к элементам данных матриц генерируется большое число кэш-промахов (кэш-промах происходит в том случае, когда микропроцессор пытается обратиться к данным, которые отсутствуют в кэш-памяти). В этом случае данные необходимо подгружать из основной памяти. Если же необходимые данные находятся в кэш-памяти, то они быстро извлекаются из нее для обработки. Такое событие называется кэш-попаданием. Повышение быстродействия вычислительной системы достигается в том случае, когда кэш-попадания реализуются намного чаще, чем кэш-промахи. Данный эффект подробно рассмотрен в работе [4]. В данном случае потери производительности от большого числа кэш-промахов начинают превалировать над полученным ускорением от векторизации, и общая производительность программы снижается.

Заключение. Эксперименты подтвердили эффективность предложенных алгоритмов векторизации. Данные алгоритмы позволяют увеличить скорость работы программы, полученной в результате автоматической векторизации на величину до 15 %. При этом увеличение скорости программы по отношению к невекторизованному коду составляет до 70 %. Выполнение векторизованного цикла ускорилось в 2,7 раза, выигрыш по отношению к автоматической векторизации в данном случае составил до 70 %. Следует также отметить, что при больших размерностях, когда возникает необходимость частых обращений к оперативной памяти, в процессе которых генерируются кэш-промахи, эффект от векторизации уменьшается.

Библиографический список

1. Брумштейн Ю. М. Анализ возможных подходов к методам компьютерного моделирования некоторых специальных задач геофильтрации / Ю. М. Брумштейн // Известия ВолГТУ. Сер. Актуальные проблемы управления, вычислительной техники и информатики в технических системах. – 2014. – № 25 (152). – С. 5–11.

2. Голуб Дж. Матричные вычисления / Дж. Голуб, Ч. Ван Лоун. – Москва : Мир, 1999.

3. Егунов В. А. Кэш-оптимизация процесса вычисления собственных значений на параллельных вычислительных системах / В. А. Егунов // Прикаспийский журнал: управление и высокие технологии. – 2019. – № 1 (45). – С. 154–163.

4. Егунов В. А. О влиянии кэш-памяти на эффективность программной реализации базовых операций линейной алгебры / В. А. Егунов // Прикаспийский журнал: управление и высокие технологии. – 2018. – № 3. – С. 88–96.

5. Егунов В.А. Оценка эффективности программной реализации QR-разложения на многоядерных архитектурах. / В. А. Егунов, С. И. Кирносенко, П. Д. Кравченя, О. О. Шумейко // Известия ВолгГТУ. Сер. Актуальные проблемы управления, вычислительной техники и информатики в технических системах. – 2017. – № 1 (196). – С. 56–59.

6. Забелин А. В. Анализ корреляции собственных частот и векторов колебаний конечно-элементных моделей пролетных строений мостов / А. В. Забелин, А. А. Пыхалов // Известия вузов. Инвестиции. Строительство. Недвижимость. – 2018. – Т. 8, № 1 (24). – С. 143–159.

7. Клаф Р. Динамика сооружений / Р. Клаф, Дж. Пензиен. – Москва : Стройиздат, 1979. – 320 с.

8. Кон В. Нобелевские лекции по химии – 1998 / В. Кон, Дж. А. Попл // УФН. – 2002. – Т. 172. – С. 335.

9. Кочура А. Е. Разработка алгоритма решения систем линейных уравнений с варьируемыми параметрами, использующего разреженность матрицы / А. Е. Кочура, Л. В. Подкользина, Я. А. Ивакин, И. И. Нидзиев // Прикаспийский журнал: управление и высокие технологии. – 2014. – № 2. – С. 10–115.

10. Aase S. O. A critique of SVD-based image coding systems / S. O. Aase, J. H. Husoy, P. Waldemar // Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI. – Piscataway : IEEE Press, NJ, 1999. – Vol. 4. – P. 13–16.

11. Andreev A. E. Solving of Eigenvalue and Singular Value Problems via Modified Householder Transformations on Shared Memory Parallel Computing Systems / A. E. Andreev, V. A. Egunov // Supercomputing: RuSCDays : 5th Russian Supercomputing Days (Moscow, Russia, September 23–24, 2019) : Conference proceedings. – Cham (Switzerland) : Springer Nature Switzerland AG, 2019. – P. 131–151. (Book Ser.: Communications in Computer and Information Science (CCIS); Vol. 1129). – Режим доступа: https://link.springer.com/book/10.1007%2F978-3-030-36592-9, свободный. – Заглавие с экрана. – Яз. англ.

12. Andreev A., Doukhnitch E., Egunov V., Zharikov D., Shapovalov O., Artuh S. Evaluation of Hardware Implementations of CORDIC-Like Algorithms in FPGA Using OpenCL Kernels // Knowledge-Based Software Engineering : Proceedings of 11th Joint Conference, JCKBSE 2014 (Volgograd, Russia, September 17–20, 2014). – Springer International Publishing, 2014. – P. 228–242. – (Series: Communications in Computer and Information Science ; Vol. 466).

13. Baker C. G. Anasazi software for the numerical solution of large-scale eigenvalue problems / C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, H. K. Thornquist // ACM Transactions on Mathematical Software. – 2009. – Vol. 36 (3), art. no. 13. – DOI: 10.1145/1527286.1527287.

14. Bogoya J. M. Extreme individual eigenvalues for a class of large hessenberg toeplitz matrices / J. M. Bogoya, S. M. Grudsky, I. S. Malysheva // Operator Theory: Advances and Applications. – 2018. – Vol. 271. – P. 119–143.

15. Bujanovic Z. A householder-based algorithm for hessenberg-triangular reduction / Z. Bujanovic, L. Karlsson, D. Kressner // SIAM Journal on Matrix Analysis and Applications. – 2018. – Vol. 39 (3). – P. 1270–1294.

16. Dominguez T. Fast Blocking of Householder Reflectors on Graphics Processors / T. Dominguez, A. E. Quintana, E. S.Orti // 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. – PDP, 2018. – P. 385–393.

17. Egunov V. A. Implementation of QR and LQ decompositions on shared memory parallel computing systems / V. A. Egunov, A. E. Andreev // 2016 2nd International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) (Chelyabinsk, Russia, 19–20 May 2016). – IEEE, 2016. – 5 p. – DOI: 10.1109/ICIEAM.2016.7911607.

18. Getmanskiy V. Optimization and Parallelization of CAE Software Stress-Strain Solver for Heterogeneous Computing Hardware / V. Getmanskiy , Andreev, S.Alekseev, A. S. Gorobtsov, V. Egunov, E. Kharkov // Creativity in Intelligent Technologies and Data Science. Second Conference, CIT&DS 2017 (Volgograd, Russia, September 12–14, 2017) : Proceedings. – Germany : Springer International Publishing AG, 2017. – P. 562–674. – (Ser. Communications in Computer and Information Science ; Vol. 754).

19. Glinsky B. The Co-design of Astrophysical Code for Massively Parallel Supercomputers / B. Glinsky, I. Kulikov, I. Chernykh, D. Weins, A. Snytnikov, V. Nenashev, A. Andreev, V. Egunov, E. Kharkov // Algorithms and Architectures for Parallel Processing. ICA3PP 2016 Collocated Workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS (Granada, Spain, December 14–16, 2016) : Proceedings. – Springer International Publishing, 2016. – P. 342–353. – (Ser. Lecture Notes in Computer Science ; Vol. 10049).

20. Harris D. Digital Design and Computer Architecture / D. Harris, S. Harris. – 2nd ed. – NY, Morgan Kaufmann, 2012. – 712 p.

21. Merchant F. Efficient realization of householder transform through algorithm-architecture co-design for acceleration of QR Factorization / F. Merchant, T. Vatwani , A. Chattopadhyay, S. K. Nandy, R. Narayan // IEEE Transactions on Parallel and Distributed Systems. – 2018. – Vol. 29 (8). – P. 1707–1720.

22. Polizzi E. Density-matrix-based algorithm for solving eigenvalue problems / E. Polizzi // Physical Review B – Condensed Matter and Materials Physics. – 2009. – Vol. 79 (11), art. no. 115112. – DOI: 10.1103/PhysRevB.79.115112.

23. Salam A. An upper J-Hessenberg reduction of a matrix through symplectic Householder transformations / A. Salam , H. B. Kahla // Computers and Mathematics with Applications. – 2019.

24. Tian Y. Some results on the eigenvalue problem for a fractional elliptic equation / Y. Tian // Boundary Value Problems. – 2019. – no. 1. – P. 13.

25. Vatankhah S. Large-scale Inversion of Magnetic Data Using Golub-Kahan Bidiagonalization with Truncated Generalized Cross Validation for Regularization Parameter Estimation / S. Vatankhah // Journal of the Earth and Space Physics. – 2019. – Vol. 44 (4). – P. 29–39.

References

1. Brumshteyn Yu. M. Analiz vozmozhnykh podkhodov k metodam kompyuternogo modelirovaniya nekotorykh specialnykh zadach geofiltratsii [Analysis of possible approaches to computer modeling methods for some special geofiltration problems]. *Izvestiya VolgGTU. Seriya. Aktualnye problemy upravleniya, vychislitelnoy tekhniki i informatiki v tekhnicheskikh sistemakh* [News of Volgograd State Technical University. Series. Actual problems of control, computer engineering and Informatics in technical systems], 2014, no. 25 (152), pp. 5–11

2. Golub Dzh., Van Loun Ch. *Matrichnye vychisleniya* [Matrix calculations]. Moscow, Mir Publ., 1999.

3. Egunov V. A. Kesh-optimizatsiya protsessa vychisleniya sobstvennykh znacheniy na parallelnykh vychislitelnykh sistemakh [Cache-optimization of the eigenvalue calculation process on parallel computing systems]. *Prikaspiyskiy zhurnal: upravleniye i vysokiye tekhnologii* [Caspian Journal: Control and High Technologies], 2019, no. 1 (45), pp. 154–163.

4. Egunov V. A. O vliyanii kesh-pamyati na effektivnost programmnoy realizatsii bazovykh operatsiy lineynoy algebry [On the influence of cache memory on the effectiveness of software implementation of basic operations of linear algebra]. *Prikaspiyskiy zhurnal: upravleniye i vysokiye tekhnologii* [Caspian Journal: Control and High Technologies], 2018, no. 3, pp. 88–96.

5. Egunov V. A., Kirnosenko S. I., Kravchenya P. D., Shumeyko O. O. Otsenka effektivnosti programmnoy realizatsii QR-razlozheniya na mnogoyadernykh arkhitekturakh [Evaluating the effectiveness of software implementation of QR-decomposition on multicore architectures]. *Izvestiya VolgGTU. Seriya. Aktualnye problemy upravleniya, vychislitelnoy tekhniki i informatiki v tekhnicheskikh sistemakh* [News of Volgograd State Technical University. Series. Actual problems of control, computer engineering and Informatics in technical systems], 2017, no. 1 (196), pp. 56–59.

6. Zabelin A. V., Pykhalov A. A. Analiz korrelyatsii sobstvennykh chastot i vektorov kolebaniy konechnoelementnykh modeley proletnykh stroeniy mostov [Analysis of correlation of natural frequencies and vibration vectors of finite element models of bridge spans]. *Izvestiya vuzov. Investitsii. Stroitelstvo. Nedvizhimost* [University news. Investment. Construction. Realty], 2018, vol. 8, no. 1 (24), pp. 143–159.

7. Klaf R., Penzien Dzh. *Dinamika sooruzheniy* [Dynamics of structures]. Moscow, Stroyizdat Publ., 1979. 320 p.

8. Kon V., Popl Dzh. A. *Nobelevskie lektsii po khimii – 1998* [Nobel lectures in chemistry – 1998]. *Uspekhi phizicheskhikh nauk* [Physics-Uspekhi (Advances in Physical Sciences], 2002, vol. 172, p. 335. (

9. Kochura A. E., Podkolzina L. V., Ivakin Ya. A., Nidziev I. I. Razrabotka algoritma resheniya sistem lineynykh uravneniy s variruemymi parametrami, ispolzuyushchego razrezhennost matritsy [Development of an algorithm for solving systems of linear equations with variable parameters using sparsity of the matrix]. *Prikaspiyskiy zhurnal: upravleniye i vysokiye tekhnologii* [Caspian Journal: Control and High Technologies], 2014, no. 2, pp. 101–115.

10. Aase S. O.; Husoy J. H. Waldemar P. A critique of SVD-based image coding systems. *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI*. Piscataway, IEEE Press, NJ, 1999, vol. 4, pp. 13–16.

11. Andreev A. E., Egunov V. A. Solving of Eigenvalue and Singular Value Problems via Modified Householder Transformations on Shared Memory Parallel Computing Systems. *Supercomputing : RuSCDays: 5th Russian Supercomputing Days (Moscow, Russia, September 23–24, 2019) : Conference proceedings*. Cham (Switzerland), Springer Nature Switzerland AG, 2019, pp. 131–151. Available at: https://link.springer.com/book/10.1007%2F978-3-030-36592-9. (Book Ser.: Communications in Computer and Information Science (CCIS) ; Vol. 1129).

12. Andreev A., Doukhnitch E., Egunov V., Zharikov D., Shapovalov O., Artuh S. Evaluation of Hardware Implementations of CORDIC-Like Algorithms in FPGA Using OpenCL Kernels. *Knowledge-Based Software Engineering : Proceedings of 11th Joint Conference, JCKBSE 2014 (Volgograd, Russia, September 17–20, 2014)*. Springer International Publishing, 2014, pp. 228–242. (Series: Communications in Computer and Information Science ; Vol. 466).

13. Baker C. G., Hetmaniuk U. L., Lehoucq R. B., Thornquist, H. K. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software*, 2009, vol. 36 (3), art. no. 13. DOI: 10.1145/1527286.1527287.

14. Bogoya J. M., Grudsky S. M., Malysheva I. S. Extreme individual eigenvalues for a class of large hessenberg toeplitz matrices. *Operator Theory: Advances and Applications*, 2018, vol. 271, pp. 119–143.

15. Bujanovic Z., Karlsson L., Kressner D. A householder-based algorithm for hessenberg-triangular reduction. *SIAM Journal on Matrix Analysis and Applications*, 2018, vol. 39 (3), pp. 1270–1294.

16. Dominguez T., Quintana A. E., Orti E. S. Fast Blocking of Householder Reflectors on Graphics Processors. *26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. PDP, 2018, pp. 385–393.

17. Egunov V. A., Andreev A. E. Implementation of QR and LQ decompositions on shared memory parallel computing systems. *2016 2nd International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) (Chelyabinsk, Russia, 19–20 May 2016)*. IEEE, 2016. 5 p. DOI: 10.1109/ICIEAM.2016.7911607.

18. Getmanskiy V., Andreev A. E., Alekseev S., Gorobtsov A. S., Egunov V., Kharkov E. Optimization and Parallelization of CAE Software Stress-Strain Solver for Heterogeneous Computing Hardware. *Creativity*

in Intelligent Technologies and Data Science. Second Conference, CIT&DS 2017 (Volgograd, Russia, September 12–14, 2017) : Proceedings. Germany, Springer International Publishing AG, 2017, pp. 562–674. (Ser. Communications in Computer and Information Science ; Vol. 754).

19. Glinsky B., Kulikov I., Chernykh I., Weins D., Snytnikov A., Nenashev V., Andreev A, Egunov V., Kharkov E. The Co-design of Astrophysical Code for Massively Parallel Supercomputers. *Algorithms and Architectures for Parallel Processing. ICA3PP 2016 Collocated Workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS (Granada, Spain, December 14–16, 2016) : Proceedings*. Springer International Publishing, 2016, pp. 342–353. (Ser. Lecture Notes in Computer Science ; Vol. 10049).

20. Harris D., Harris S. *Digital Design and Computer Architecture.* 2nd ed. NY, Morgan Kaufmann, 2012. 712 p.

21. Merchant F., Vatwani T.,Chattopadhyay A., Nandy S. K., Narayan R. Efficient realization of householder transform through algorithm-architecture co-design for acceleration of QR Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 2018, vol. 29 (8), pp. 1707–1720.

22. Polizzi E. Density-matrix-based algorithm for solving eigenvalue problems. *Physical Review B – Condensed Matter and Materials Physics*, 2009, vol. 79 (11), art. no. 115112. DOI: 10.1103/PhysRevB.79.115112.

23. 22. Salam A., Kahla H. B., An upper J-Hessenberg reduction of a matrix through symplectic Householder transformations. *Computers and Mathematics with Applications*, 2019.

24. Tian Y. Some results on the eigenvalue problem for a fractional elliptic equation. *Boundary Value Problems*, 2019, no. 1, p. 13.

25. Vatankhah S. Large-scale Inversion of Magnetic Data Using Golub-Kahan Bidiagonalization with Truncated Generalized Cross Validation for Regularization Parameter Estimation. *Journal of the Earth and Space Physics*, 2019, vol. 44 (4), pp. 29–39.

DOI 10.21672/2074-1707.2020.50.2.096-110
УДК 004.4:004.89:004.94

МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ АГЕНТОВ ДЛЯ РЕАЛИЗАЦИИ ИГРОВОГО ИСКУССТВЕННОГО ИНТЕЛЛЕКТА[1*]

Статья поступила в редакцию 19.03.2020, в окончательном варианте – 21.04.2020.

Анохин Александр Олегович, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. Ленина, 28,
магистрант, e-mail: alex.anokhin.st@gmail.com

Садовникова Наталья Петровна, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. Ленина, 28,
доктор технических наук, профессор, ORCID 0000-0002-7214-9432 e-mail: npsn1@ya.ru, elibrary: https://elibrary.ru/author_profile.asp?authorid=427120

Катаев Александр Вадимович, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. Ленина, 28,
кандидат технических наук, доцент, e-mail: alexander.kataev@gmail.com

Парыгин Данила Сергеевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. Ленина, 28,
кандидат технических наук, доцент, ORCID 0000-0001-8834-5748, e-mail: dparygin@gmail.com, elibrary: https://elibrary.ru/author_profile.asp?authorid=842349

В статье рассматриваются некоторые вопросы эффективности применения методов машинного обучения к моделированию поведения интеллектуальных агентов в компьютерных играх и других программных разработках. Основное внимание уделено анализу методов построения моделей поведения интеллектуальных агентов. Проведен обзор существующих программных решений, реализующих данные методы в игровых приложениях. Рассмотрены их особенности, достоинства и недостатки. С учетом результатов анализа разработана новая программная платформа для моделирования поведения интеллектуальных агентов. В качестве способа описания поведения агента выбран автоматный подход. Тестирование разработанной модели поведения агента показало, что ее можно использовать для создания игрового искусственного интеллекта, который будет демонстрировать достаточно реалистичное поведение игровых персонажей в различных ситуациях. Платформа реализована на языке C++, для визуализации использовалась библиотека SFML.

Ключевые слова: игровой искусственный интеллект, обучения с подкреплением, интеллектуальный агент, машинное обучение, модель на правилах, дерево поведения, конечно-автоматная модель

[1*] Исследование выполнено при финансовой поддержке РФФИ в рамках научных проектов № 18-07-01308_а и № 18-37-20066 “мол_а_вед”