УДК 519.242

ИССЛЕДОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ОПЕРАЦИЙ НАД МАТРИЦАМИ В РАЗЛИЧНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

М. А. Артемов, Д. Е. Кочкин, И. Б. Крыжко

Воронежский государственный университет

Рассмотрены возможные способы реализации операций над матрицами при решении различных математических проблем. Выполнено исследование производительности операции расчета матричного произведения, проведено сравнение производительности в языках программирования C++/C, Delphi/Pascal и C#.

ВВЕДЕНИЕ

При программной реализации различных математических задач и алгоритмов одной из часто реализуемых функций является работа с матрицами и векторами. В настоящее время разработано множество методов для выполнения различных матричных вычислений. Существует большое количество готовых библиотек операций над матрицами под различные платформы, как с открытым исходным кодом, так и с закрытым. Более того, наиболее часто выполняющиеся операции реализованы в цифровых сигнальных процессорах на аппаратном уровне. Помимо этого, существует ряд программ, специально предназначенных для математических вычислений (MATLAB, SciLab, Mathematica и др.). Таким образом, перед математиком, берущимся за практическую проверку разработанных теоретических гипотез, стоит проблема выбора: во-первых, языка программирования, во-вторых, платформы, реализующей язык, в-третьих, самого способа реализации матричных операций. Попытаемся в этой статье провести детальный анализ существующих вариантов решения этой проблемы, а также различных методов, позволяющих увеличить производительность приложений.

ПОСТАНОВКА ЗАДАЧИ

В качестве анализируемой задачи выбрана простейшая операция над матрицами: нахождение произведения двух матриц A и B заданных размеров, заполненных набором случайных чисел из интервала [0,1]. Для корректности операции матричного умножения размеры матриц согласованы (число строк матрицы B равно числу столбцов матрицы A). Матричное произведение определяется следующим образом:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\text{, для } i = 1..m, j = 1..r, \qquad (1)$$

где m и n — число строк и столбцов матрицы A; r — число столбцов матрицы B.

Таким образом, для нахождения одного элемента результирующей матрицы требуется выполнить n операций умножения и $n-1$ операцию сложения. Соответственно, для вычисления всей результирующей матрицы требуется выполнить $m \cdot n \cdot r$ операций умножения и $m \cdot (n-1) \cdot r$ операций сложения. Значит, операция нахождения матричного произведения имеет сложность, пропорциональную n^3: при увеличении числа строк и столбцов у исходных матриц вдвое, требуемое число операций увеличится в 8 раз.

Первым выбором, возникающим перед математиком при решении задачи, будет выбор операционной системы. Наибольшее распространение получили две операционные системы: Microsoft Windows и различные сборки Linux. Вопрос, который достаточно остро стоял 10—20 лет назад, к настоящему времени уже утратил свою актуальность: подавляющее большинство языков программирования, платформ разработки, вычислительных сред имеют реализации как для Windows, так и для Linux. В силу этого факта, мы не будем рассматривать в данной статье производительность приложения в различных операционных системах.

© Артемов М. А., Кочкин Д. Е., Крыжко И. Б., 2007

В качестве экспериментальной системы выбрана операционная система Microsoft Windows XP Professional SP2. Для тестирования производительности использовался компьютер с двухъядерным процессором Intel Pentium Core2Duo с тактовой частотой 1,83 ГГц и оперативной памятью 1 Гб.

Для тестирования были выбраны языки C++, Pascal/Delphi и C#. C++- и Delphi-приложения создавались в двух вариантах: под платформу Microsoft.NET и как обычное Win32-приложение. Для компиляции C++ и C#-приложений использовалась среда разработки Microsoft Visual Studio 2006. Для компиляции Delphi-приложений использовалось среда Turbo Delphi 2006.

Операции с матрицами инкапсулировались в классе Matrix/TMatrix. Все матрицы были вещественнозначными, для этого использовался наиболее часто использующийся математиками восьмибайтный тип double. Операция умножения при языковой возможности реализовывалась как перегруженный оператор умножения, при невозможности (Delphi под Win32) — как обычная функция, принимающая два параметра и возвращающая матрицу. Общее описание класса (без реализации) на языке C# выглядит следующим образом:

```
public class Matrix {
    //конструктор
    Matrix(int m, int n)
    //статический метод, создающий
    //экземпляр матрицы, размером m на n,
    //содержащий случайные числа
    //в интервале [0,1]
    public static Matrix Random(int m, int n)
    //функция, возвращающая число строк
    //матрицы
    int RowCount
    //функция, возвращающая число столбцов
    //матрицы
    int ColumnCount
    //Перегруженный оператор умножения
    public static Matrix operator *(Matrix
       m1, Matrix m2) }
```

В случае реализации на языке, не поддерживающем сборку мусора, добавлялась реализация деструктора.

Для замеров времени использовались следующие возможности:

- на платформах, поддерживающих .NET Framework версии 2.0 (C++.NET и C#) — таймер высокого разрешения, реализованный в виде класса Stopwatch пространства имен System.Diagnostics;
- на остальных платформах — WinAPI-функции QueryPerformanceCounter и QueryPerformanceFrequency, в совокупности обеспечивающие функции таймера высокого разрешения.

Для построения графиков использовалась следующая программа MATLAB:

```
datfiles = dir('*.dat');
fid = fopen(datfiles(1).name);
n = fread(fid, 1, 'int32');
x = fread(fid, n, 'int32', 4);
fclose(fid);
fc = size(datfiles, 1);
y = zeros(fc, n);
for i = 1:fc
    fid = fopen(datfiles(i).name);
    fread(fid, 2, 'int32');
    y(i, 1:n) = fread(fid, n, 'int32', 4);
    fclose(fid);
end
plot(x, y(1:fc, 1:n))
grid on
legend(datfiles(1:fc).name, 'Location',
      'NorthWest');
xlabel('Matrix size, elements')
ylabel('Time, ms')
title('Performance')
```

В силу того, что на производительность влияет много факторов, замеры проводились следующим образом: создавались две матрицы для вычисления произведения, брался эталонный отсчет времени, затем запускалась операция нахождения произведения матриц, после чего брался второй отсчет времени. По разности отсчетов определялось время выполнения операции. Далее для языков, не поддерживающих сборку мусора, вызывались деструкторы для всех трех матриц: двух исходных и результирующей. Результирующая матрица создавалась в процессе нахождения произведения матриц, что, конечно, влияло на результаты замеров производительности, но это добавочное время следует учитывать в оценке, поскольку при практическом использовании оно все равно будет включаться в общее время расчетов. Для ускорения работы с памятью в Delphi для Win32 использовался нестандартный менеджер памяти FastMM (http://fastmm.sourceforge.net), являющийся альтернативной заменой стандартного

менеджера памяти Delphi для ускорения работы с памятью и отслеживания утечек памяти.

В качестве языкового средства для хранения и обращения к элементам двумерного массива использовалось то, которое является наиболее часто применяемым вариантом реализации двумерных массивов для данного языка. В некоторых ситуациях удобнее хранить в памяти матрицу как линейный одномерный массив, выполняя обращение к паре [сегмент, смещение] вручную. Такой вариант тестировался на Win32-приложениях C++ и Delphi. Таким образом, использовались следующие языковые конструкции:

- C# — двумерные массивы double[,];
- Delphi для Win32 — array of array of double и array of double;
- Delphi для .NET — array of array of double (однако внутри эта структура реализована не так, как в Delphi для Win32);
- C++ — vector<vector<double>> и указатель double* на динамический массив, созданный с помощью new double[m*n];
- C++.NET — ссылочный массив в управляемой куче .NET array<double,2>^, созданный с помощью gcnew array<double,2>(m, n).

В случае платформы .NET, имеющей сборщик мусора, специально не выполнялось никаких действий по дополнительному управлению памятью, чтобы оценить влияние сборки мусора на работу программы.

Ни на одной из платформ не предпринималось никаких мер по оптимизации производительности приложения, однако в настройках соответствующих компиляторов были установлены такие настройки компилятора с максимальной оптимизацией. Первоначально обращения к элементам матрицы выполнялось через свойства класса, однако профилирование приложений показало, что при таком подходе около 50% времени занимает вызов метода получения значения свойства. Таким образом, можно сделать первую рекомендацию по реализации: следует обращаться к элементам массива в методах класса напрямую (однако пользователь вполне может иметь доступ к элементам и только через свойства, так как всю необходимую для работы функциональность должен обеспечивать матричный класс, а пользователь должен программировать только логику программу).

На рисунке приведены графики зависимости времени расчета матричного произведения от размеров матриц для матриц от 15×15 до 600×600 с шагом 15. По горизонтальной оси отложены размеры матриц, по вертикальной — время расчета.

Как следует из рисунка, дольше всего проработало приложение Delphi под Win32 с реализацией в виде линейного массива. Для матриц размером 600×600 время работы составило 6035 мс. Такие результаты вызывают удивление, т.к. самым быстрым оказался C++ с реализацией в виде линейного массива: расчет был выполнен всего за 2040 мс. Следующим по производительности идет приложение Delphi под Win32 с реализацией в виде массива массивов. Примерно на одном уровне с ним по времени выполнения оказался C++ с реализацией в виде контейнеров STL vector. Время расчета составило 3933 мс и 3793 мс для Delphi и C++ соответственно. Далее один в один идут приложение C#, приложение C++.NET и приложение Delphi.NET. Графики их производительности пересекаются и на небольших матрицах (до 400×400) быстрее оказываются компиляторы от Microsoft, а на больших (от 500×500) — компилятор Delphi. Время выполнения приложений составило около 3 с. И, наконец, абсолютным лидером по производительности стал язык C.

Проведем анализ полученных результатов. Дизассемблирование исходного кода на C++ и его сравнение с дизассемблированным кодом на Delphi показало, что в режиме отладки компиляторы генерируют примерно схожий код. Когда же код C++ собирается с максимальной оптимизацией, оптимизатор разворачивает несколько инструкций работы с числами с плавающей точкой из цикла в несколько подряд идущих команд, что, видимо, и приводит к такому скачку производительности. Помимо этого, динамические массивы Delphi реализованы с автоматическим подсчетом ссылок, что также снижает производительность вычислений, но, пожалуй, наиболее критичным оказывается тот факт, что динамический многомерный массив является массивом массивов — то есть в памяти сохраняется ссылка на каждую строчку массива и определение адреса нужной ячейки требует дополнительных затрат времени. Однако природа снижения производительности с линейным массивом в Delphi, видимо, может быть объяснена только неоптимальностью генерируемого кода. В случае с Delphi.NET динамические массивы реализованы схожим обра-

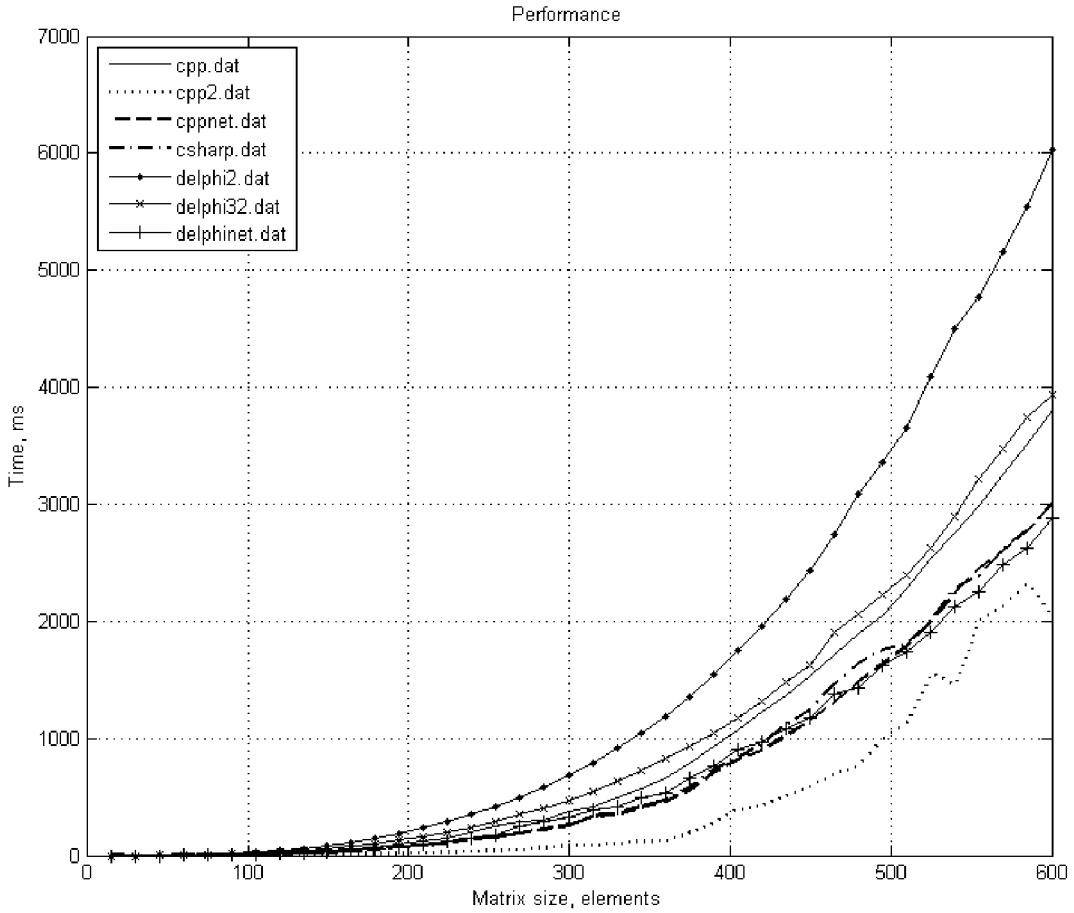

Рис. 1. Зависимости времени расчета матричного произведения от размера матриц в различных языках программирования

зом, в отличие от C#, где есть многомерные массивы ([,]), для которых задается два размера — ширина и высота, а есть массивы массивов ([][]), где размерность каждой строки можно задать вручную.

Не вполне понятной остается природа изгибов графика производительности для языка C++ в случае линейного массива при размерах матриц от 500×500 до 600×600. Видимо, разворачивание цикла выполняется таким образом, что максимальной эффективности оно достигает на некоторых конкретных значениях размеров массивов.

Код приложений .NET, как известно, компилируется JIT-компилятором в момент первого вызова. Следовательно, нет никаких оснований полагать, что он будет работать медленнее обычного неуправляемого кода, в результате и получаем, что приложения .NET не имеют потерь производительности из-за своей «компиляции на лету». Однако некоторые потери производительности вносит сборщик мусора — это можно заметить по «горбам» на рис. 1 для приложений .NET. То есть в определенный момент «просыпается» поток сборщика мусора и параллельно с основным рабочим потоком делает свое дело — проводит сбор мусора, что не может не сказаться на производительности главного потока (впрочем, как можно заметить, эти потери незначительны).

Следует отметить, что со специализированными системами типа MATLAB не может конкурировать даже C++. Так, например, произведение двух матриц 2000×2000 на C++ занимает 85 с, в бесплатной системе SciLab — около 20 с, а в коммерческой системе MATLAB — и вовсе 4 с!

В любом случае, главным фактором, влияющим на выбор средства разработки, остается область применения полученных результатов.

Если целью разработки является проверка гипотезы или работоспособности построенной модели, то наиболее быстрым вариантом (имеется в виду скорость разработки) будет написание программы в специализированный системе (MATLAB, SciLab). При этом все операции представляются в достаточно наглядной для

математика форме, имеют хорошую производительность и не требуют большого времени на освоение базовых функций.

Если требуется получить рабочее приложение, то все зависит от области его применения: при большом объеме вычислений (размеры матриц больше 500×500, требуемое время работы алгоритма на C++ — менее 1 с) в коммерческих приложениях даже вариант C++ окажется медленным и в этом случае можно предложить воспользоваться оптимизирующими компиляторами, которые генерируют код с поддержкой новых наборов инструкций — 3DNow!, SSE, SSE2, SSE3. Однако для приложений, не имеющих серьезных ограничений на время расчета, подойдет любой из перечисленных языков. Разработка под платформу .NET будет выполняться быстрее, чем на неуправляемых языках, за счет наличия технологии сборки мусора. Кроме того, Delphi и C#, в отличие от C++, являются языками более высокого уровня и поэтому с их помощью легче написать хорошо структурированное приложение. Если это приложение в дальнейшем планируется переписывать на C++ (например, с целью компиляции под определенный цифровой процессор), то прототип лучше всего писать на C#, т.к. этот язык близок к C++ и дальнейший перевод алгоритма займет немного времени. Вариант C++. NET является, пожалуй, самым неудобным вариантом — имея производительность, идентичную производительности C#, за счет введения дополнительных синтаксических конструкций для поддержки .NET, язык сильно проигрывает в удобочитаемости кода и возможностях внесения в него изменений.

На основании проведенных исследований можно сделать вывод, что самой быстродействующей реализацией является реализация на C с хранением матриц в виде линейного массива. На втором месте по скорости выполнения оказываются приложения на базе платформы . NET, независимо от языка программирования. Следом идут приложения Delphi, хранящие матрицу в массиве массивов и приложения C++, хранящие матрицу в векторе векторов STL. Наименее быстродействующим оказался вариант Delphi, хранящий матрицу в виде линейного массива.

ЛИТЕРАТУРА

1. *Страуструп Б.* Язык программирования C++. Специальное издание / Пер. с англ. — М.: Бином, 2006. — 1104 с.

2. *Тейксейра С., Пачеко К.* Delphi 5. Руководство разработчика. Т. 1. Основные методы и технологии программирования / Пер. с англ. — СПб.: Вильямс, 2000. — 832 с.

3. *Рихтер Дж.* CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер-класс / Пер. с англ. —СПб.: Питер, 2007. — 656 с.

Статья принята к опубликованию
25 декабря 2006 г.