УДК 681.3
Э.И. Ватутин, канд. техн. наук, доцент, кафедра вычислительной техники, ЮЗГУ (e-mail: evatutin@rambler.ru)
И.А. Мартынов, аспирант кафедры вычислительной техники, ЮЗГУ (e-mail: morzee@inbox.ru)
В.С. Титов, д-р техн. наук, профессор, зав. кафедрой вычислительной техники, ЮЗГУ (e-mail: titov-kstu@rambler.ru)

ОЦЕНКА РЕАЛЬНОЙ ПРОИЗВОДИТЕЛЬНОСТИ СОВРЕМЕННЫХ ПРОЦЕССОРОВ В ЗАДАЧЕ УМНОЖЕНИЯ МАТРИЦ ДЛЯ ОДНОПОТОЧНОЙ ПРОГРАММНОЙ РЕАЛИЗАЦИИ

Приведено описание подходов к выполнению операции умножения матриц, показано, что «наивный» подход без оптимизации работы с памятью характеризуется низкой производительностью обработки, в то время как подходы с буферизацией столбца и блочного умножения позволяют более эффективно использовать кэш-память, что в совокупности с применением раскрутки циклов обеспечивает реальную производительность на уровне 2,5–6,8 GFLOP/s для однопоточной реализации операции для современных процессоров Intel Core.

Ключевые слова: умножение матриц, алгоритмическая оптимизация

***

Одной из частных подзадач, находящих широкое применение при решении задач широкого спектра (томография, компьютерная графика, проектирование роботизированных средств, классификация бинарных отношений [1] и др.), является задача умножения матриц. Время ее решения во многих случаях является бутылочным горлышком (англ. bottleneck) и напрямую влияет на время решения поставленной задачи, поэтому существует большое количество различных подходов, связанных с оптимизацией и распараллеливанием выполняемых действий. Существуют целый спектр программно-алгоритмического обеспечения для выполнения действий над матрицами в ряде частных случаев (например, для разряженных или ленточных матриц), а также базирующееся на нем аппаратно-алгоритмическое обеспечение (например, с использованием транспьютерных сетей или систолических вычислительных структур). Несмотря на кажущуюся простоту, задача характеризуется рядом особенностей, к которым в первую очередь можно отнести высокую степень параллелизма для выполняемых операций и сильную зависимость времени вычисления от темпа поступления данных из памяти. В данной статье рассмотрены вопросы анализа эффективности различных подходов к решению задачи общего вида (умножение «плотных» квадратных матриц размера $N \times N$) для однопоточной высокоуровневой программной реализации, ориентированной на использование процессоров семейства x86, с оптимизацией работы кэш-памяти.

Как известно, результатом умножения квадратных матриц A и B размера $N \times N$ является квадратная матрица C размером $N \times N$, элементы c_{ij}, $i, j = \overline{1, N}$ которой определяются по формуле

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} . \quad (1)$$

«Наивная» программная реализация, соответствующая формуле (1) и часто используемая в процессе обучения основам программирования, представлена ниже.

```
for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
      {
            float s = 0.0f;

            for (int k=0; k<N; k++)
                  s += A[i][k]*B[k][j];

            C[i][j] = s;
      }
```

Для последующих оценок производительности различных программных реализаций заметим, что в составе внутреннего цикла присутствуют две операции с плавающей точкой (сложение и умножение), что формально позволяет оценить необходимый объем вычислений $V = 2N^3$ FLOP. Зная время t, которое затрачивает вычислительная система на выполнение программы, можно оценить полученную реальную производительность $P = \frac{V}{t}$. При этом, в отличие от оценки абстрактной пиковой производительности вычислительной системы (в простейшем случае – процессора), учитывается ряд особенностей архитектуры современных вычислительных средств (суперскалярность, конвейерное и внеочередное исполнение команд, наличие кэш-памяти) и их влияние на реальную производительность конкретной программной реализации.

Для приведенной выше «наивной» программной реализации время выполнения и соответствующая реальная производительность для матриц различного размера приведены в таблице 1 (здесь и далее тип элементов матриц имеет одинарную точность).

Таблица 1. Оценка времени выполнения и реальной производительности для «наивной» программной реализации, компилятор Microsoft Visual Studio 2012, без оптимизаций компилятора

Размерность задачи	**Объем обрабатываемых данных**	**Объем вычислений**	**Время выполнения, производительность**	
			Intel Core 2 Duo E6300, 1,86 ГГц, 2 МБ L2 (Allendale, 2006)	**Intel Core i7 4770, 3,4 (3,9) ГГц, 8 МБ L3 (Haswell, 2013)**
256×256	2×256 = 512 КБ	32 MFLOP	0,135 с 0,24 GFLOP/s	0,045 с 0,71 GFLOP/s

512×512	2×1 = 2 МБ	256 MFLOP	1,1 с 0,23 GFLOP/s	0,41 с 0,62 MFLOP/s
1024×1024	2×4 = 8 МБ	2 GFLOP	40,8 с 0,05 GFLOP/s	3,3 с 0,61 GFLOP/s
2048×2048	2×16 = 32 МБ	16 GFLOP	340 с (5 мин 40 с) 0,05 GFLOP/s	103 с (1 мин 43 с) 0,16 GFLOP/s

Анализ полученных результатов позволяет сделать ряд выводов. Так производительность обработки на использованной в тестировании паре процессоров (первый и последний на данный момент представители семейства Intel Core) отличается приблизительно в 3 раза за счет разницы в тактовых частотах ($3{,}9 / 1{,}86 = 2{,}1$ раза) и микроархитектурных улучшений ядра Haswell по сравнению с Allendale. Из общей тенденции выбивается тест для размерности задачи 1024×1024 (разница в производительности $0{,}61 / 0{,}05 = 12{,}2$ раза), объяснением чему служит исчерпание объема доступной кэш памяти (2 МБ для ядра Allendale при объеме обрабатываемых данных 8 МБ). По той же причине наблюдается падение производительности при переходе 512×512→1024×1024 (в $0{,}23 / 0{,}05 = 4{,}6$ раза) для ядра Allendale и 1024×1024→2048×2048 (в $0{,}61 / 0{,}16 = 3{,}8$ раза) для ядра Haswell. Таким образом, скорость поступления исходных данных существенно лимитирует реальную производительность обработки и для матриц большой размерности (превышающих объем доступной кэш-памяти) определяется пропускной способностью оперативной памяти, кэш-память процессора при этом используется неэффективно.

Включение оптимизаций в составе опций компилятора (опция «/O2») сокращает время обработки приблизительно в 2 раза (с 103 до 68 с для умножения матриц 2048×2048 на ядре Haswell), позволяя достичь производительности 0,24 PFLOP/s и не меняя общей картины.

При выполнении «наивной» программной реализации производится $2N^3$ обращений в оперативную память, при этом обращения к элементам матрицы A производятся по смежным адресам $\left(a + iN, a + iN + 1, a + iN + 2, ...\right)$, что делает эффективной работу механизма аппаратной предвыборки данных (англ. hardware prefetch), в то время как обращения к элементам матрицы B производятся «прыжками» (см. рис. 1) через N элементов $\left(b + j, b + N + j, b + 2N + j, ...\right)$, что затрудняет как работу кэш-памяти, так и механизма аппаратной предвыборки данных, приводя к большому числу кэш-промахов (англ. cache miss) [2].

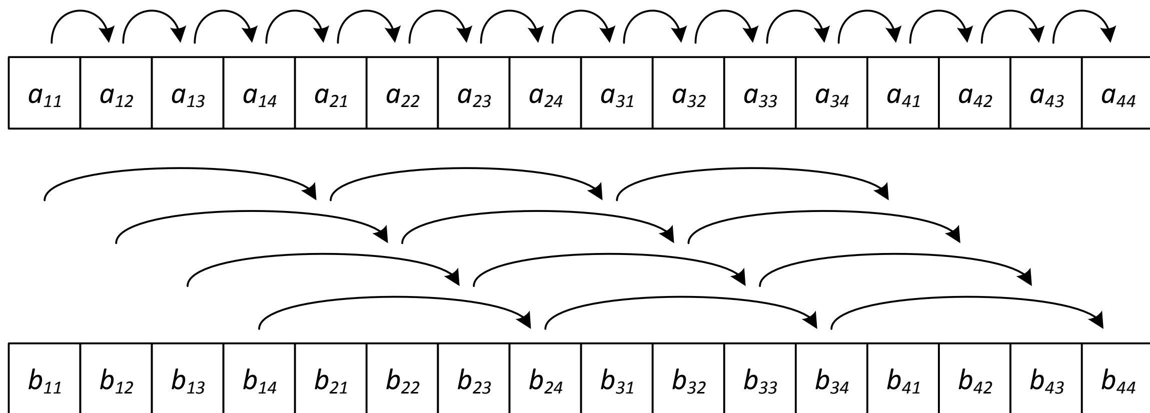

Рис. 1. Схема обращений к памяти при «наивной» реализации умножения матриц

Снизить число «неудобных» обращений в память можно путем буферизации значения *j*-го столбца матрицы B, что, в свою очередь, требует изменения порядка следования циклов в программе. При реализации данной стратегии обращения к элементам матрицы B «прыжками» через N элементов заменяются обращениями к временному массиву-буферу $T = \left[t_1, t_2, ..., t_N\right]$ размером N элементов, $t_k = b_{kj}$, $k = \overline{1, N}$, которые располагаются в кэш-памяти и обходятся подряд. При этом умножение матриц выполняется по формуле

$$c_{ij} = \sum_{k=1}^{N} a_{ik} t_k , \tag{2}$$

соответствующая программная реализация приведена ниже.

```
float buf[N];     // Буфер для хранения j-го столбца

for (int j=0; j<N; j++)
{
      // Буферизация столбца
      for (int k=0; k<N; k++)
            buf[k] = B[k][j];

      // Умножение
      for (int i=0; i<N; i++)
      {
            float s = 0.0f;

            for (int k=0; k<N; k++)
                  s += A[i][k]*buf[k];

            C[i][j] = s;
      }
}
```

При подобной буферизации в память производится $N^3 + N^2$ обращений, остальные N^3 обращения к массиву-буферу T производятся в кэш. Результаты оценки необходимого времени и соответствующая реальная производительность приведены в таблице 2.

Таблица 2. Оценка времени выполнения и реальной производительности для программной реализации с буферизацией столбца, компилятор Microsoft Visual Studio 2012, с оптимизациями компилятора

Размерность задачи	**Время выполнения, производительность, выигрыш по сравнению с «наивной» реализацией**	
	Intel Core 2 Duo E6300, 1,86 ГГц, 2 МБ L2 (Allendale, 2006)	**Intel Core i7 4770, 3,4 (3,9) ГГц, 8 МБ L3 (Haswell, 2013)**
256×256	27 мс 1,3 GFLOP/s 5,4x	14 мс 2,4 GFLOP/s 3,4x
512×512	278 мс 1,0 GFLOP/s 4,3x	104 мс 2,6 GFLOP/s 4,2x
1024×1024	2,1 с 1,0 GFLOP/s 20x	850 мс 2,5 GFLOP/s 4,1x
2048×2048	16 с 1,1 GFLOP/s 22x	7 с 2,5 GFLOP/s 15,6x

Приведенные в таблице данные позволяет заметить, что при использовании буферизации столбца не происходит существенной деградации реальной производительности с ростом размерности задачи, что подтверждает эффективность использования кэш-памяти. При этом по сравнению с «наивной» реализацией наблюдается существенный выигрыш во времени обработки (от 3 до 20 раз), в особенности в ситуации, когда объем обрабатываемых данных превышает объем доступной кэш-памяти.

Основным действием, лимитирующим время выполнения умножения, является действия накапливающего сложения во внутреннем цикле. Действия сводятся к двум обращениям к (кэш)памяти (для чтения значений a_{ik} и t_k), которые не зависят друг от друга по данным и теоретически могут выполняться параллельно во времени при отсутствии аппаратных ограничений, одному умножению $a_{ik} \times t_k$, зависящему по данным от значений a_{ik} и t_k, и одному сложению $s + a_{ik}t_k$, зависящему по данным от результата умножения. Таким образом, тело цикла в основном состоит из зависящих друг от друга действий (RAW-зависимости), что приводит к низкому параллелизму на уровне команд (англ. Instruction Level Parallelism, ILP) и низкой загрузке исполнительных устройств процессора. Повысить степень загрузки исполнительных устройств можно путем раскрутки внутреннего цикла программы (англ. loop unrolling) [2]. Пример раскрутки на 4 приведен ниже.

```
float buf[N];

for (int j=0; j<N; j++)
{
	for (int k=0; k<N; k++)
		buf[k] = B[k][j];

	for (int i=0; i<N; i++)
	{

		// Раскрутка на 4
		float s1 = 0.0f;
		float s2 = 0.0f;
		float s3 = 0.0f;
		float s4 = 0.0f;

		for (int k=0; k<N; k += 4)
		{
			s1 += A[i][k]*buf[k];
			s2 += A[i][k+1]*buf[k+1];
			s3 += A[i][k+2]*buf[k+2];
			s4 += A[i][k+3]*buf[k+3];
		}

		C[i][j] = s1+s2+s3+s4;
	}
}
```

Результаты влияния указанной оптимизации на времени выполнения приведены в таблицах 3 и 4.

Таблица 3. Оценка влияния раскрутки внутреннего цикла на время обработки, реальную производительность и выигрыш по сравнению с реализацией без раскрутки для программной реализации с буферизацией столбца, компилятор Microsoft Visual Studio 2012, с оптимизациями компилятора, ядро Haswell

<table>
<tr><th rowspan="3">Размер-ность задачи</th><th colspan="4">Время выполнения, производительность, выигрыш</th></tr>
<tr><th rowspan="2">Без раскрутки</th><th colspan="3">С раскруткой</th></tr>
<tr><th>на 2 итерации</th><th>на 4 итерации</th><th>на 8 итераций</th></tr>
<tr><td>256×256</td><td>14 мс
2,4 GFLOP/s</td><td>7,9 мс
4,2 GFLOP/s
1,8x</td><td>5,9 мс
5,7 GFLOP/s
2,4x</td><td>6,2 мс
5,4 GFLOP/s
2,3x</td></tr>
<tr><td>512×512</td><td>104 мс
2,6 GFLOP/s</td><td>55 мс
4,9 GFLOP/s
1,9x</td><td>40 мс
6,7 GFLOP/s
2,6x</td><td>44 мс
6,2 GFLOP/s
2,4x</td></tr>
<tr><td>1024×1024</td><td>850 мс
2,5 GFLOP/s</td><td>442 мс
4,9 GFLOP/s
1,9x</td><td>318 мс
6,8 GFLOP/s
2,7x</td><td>350 мс
6,1 GFLOP/s
2,4x</td></tr>
</table>

2048×2048	7 с 2,5 GFLOP/s	3,7 с 4,6 GFLOP/s 1,8x	2,9 с 5,9 GFLOP/s 2,4x	3,1 с 5,6 GFLOP/s 2,2x

Таблица 4. Оценка влияния раскрутки внутреннего цикла на время обработки, реальную производительность и выигрыш по сравнению с реализацией без раскрутки для программной реализации с буферизацией столбца, компилятор Microsoft Visual Studio 2012, с оптимизациями компилятора, ядро Allendale

Размер-ность задачи	**Время выполнения, производительность, выигрыш**			
	Без раскрутки	**С раскруткой**		
		на 2 итерации	**на 4 итерации**	**на 8 итераций**
256×256	27 мс 1,3 GFLOP/s	21 мс 1,6 GFLOP/s +23%	20 мс 1,7 GFLOP/s +31%	21 мс 1,6 GFLOP/s +23%
512×512	278 мс 1,0 GFLOP/s	175 мс 1,5 GFLOP/s +50%	171 мс 1,6 GFLOP/s +60%	199 мс 1,3 GFLOP/s +33%
1024×1024	2,1 с 1,0 GFLOP/s	1,5 с 1,4 GFLOP/s +40%	1,7 с 1,3 GFLOP/s +30%	1,6 с 1,4 GFLOP/s +40%
2048×2048	16 с 1,1 GFLOP/s	12 с 1,4 GFLOP/s +27%	12 с 1,5 GFLOP/s +36%	14 с 1,2 GFLOP/s +9%

Анализ приведенных результатов показывает, что для данной задачи достаточно раскрутки внутреннего цикла на 4 итерации, что обеспечивает дополнительную прибавку в производительности на 30-60% для ядра Allendale и в 2,4 раза для ядра Haswell за счет повышения степени загрузки исполнительных устройств, раскрутка на большее число итераций приводит к падению производительности и нецелесообразна.

Еще одним известным способом, позволяющим повысить локальность обрабатываемых данных, является блочное умножение. При этом результирующая матрица *C* получается не поэлементно, как в предыдущих примерах, а поблочно, квадратными блоками размером $S \times S$ элементов. При этом для вычисления значений выбранного блока требуется обращение к группе из *S* строк матрицы *A* и *S* столбцов матрицы *B*, т.е. для текущего блока требуется обращение к $2 \times S \times N$ элементам матриц, которые могут быть размещены в кэше процессора. Если разбить процесс нахождения

результирующих значений на $z = \frac{N}{S}$ стадий (см. рис. 2), то требование к необходимым для выполнения текущей группы операции элементам может быть дополнительно снижено до $2S^2$.

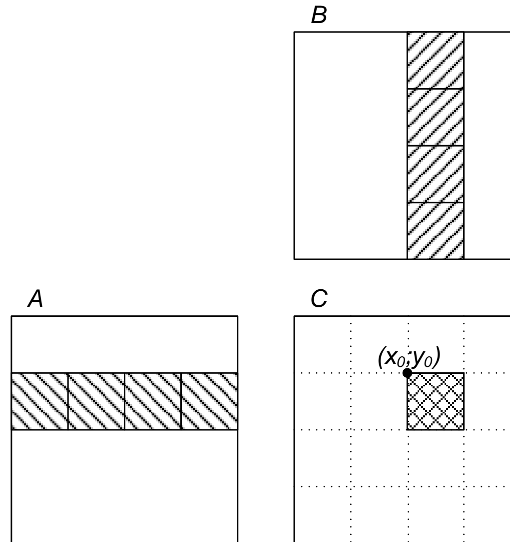

Рис. 2. Иллюстрация в блочному умножению матриц

При этом само умножение фактически выполняется как

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} = \underbrace{\sum_{k=1}^{S} a_{ik} b_{kj} + \sum_{k=S+1}^{2S} a_{ik} b_{kj} + \sum_{k=SB+1}^{3S} a_{ik} b_{kj} + \ldots + \sum_{k=N-S+1}^{N} a_{ik} b_{kj}}_{z=\frac{N}{S} \text{ сумм}} = \tag{3}$$

$$= \sum_{z=0}^{N/S} \sum_{k=(z-1)S+1}^{zS} a_{ik} b_{kj}.$$

Программная реализация, соответствующая умножению по формуле (3), приведена ниже.

```
float Az[BLOCK_SIZE][BLOCK_SIZE];
float Bz[BLOCK_SIZE][BLOCK_SIZE];

// Перебор блоков результирующей матрицы
for (int x0 = 0; x0 < N; x0 += BLOCK_SIZE)
      for (int y0 = 0; y0 < N; y0 += BLOCK_SIZE)
      {
            float sum[BLOCK_SIZE][BLOCK_SIZE];

            for (int x = 0; x < BLOCK_SIZE; x++)
                  for (int y = 0; y < BLOCK_SIZE; y++)
                        sum[x][y] = 0.0f;

            // Цикл по подматрицам (z)
            for (int z = 0; z < N/BLOCK_SIZE; z++)
            {
                  int zb = z*BLOCK_SIZE;

                  // Копирование подматриц Az и Bz
                  for (int x = 0; x < BLOCK_SIZE; x++)
```

```
                for (int y = 0; y < BLOCK_SIZE; y++)
                {
                    Az[x][y] = A[x0+x][zb+y];
                    Bz[x][y] = B[zb+x][y0+y];
                }

            // Умножение подматриц Az*Bz
            for (int x = 0; x < BLOCK_SIZE; x++)
                for (int y = 0; y < BLOCK_SIZE; y++)
                {
                    float s = 0.0f;

                    for (int k = 0; k < BLOCK_SIZE; k++)
                        s += Az[x][k] * Bz[k][y];

                    sum[x][y] += s;
                }
        }

        // Запись результирующей подматрицы
        for (int x = 0; x < BLOCK_SIZE; x++)
            for (int y = 0; y < BLOCK_SIZE; y++)
                C[x0+x][y0+y] = sum[x][y];
    }
```

При использовании блочного подхода производится $\frac{2N^3}{S}$ обращений в память и $2N^3$ обращений в кэш. При этом производительность обработки зависит от выбранного размера блока S (табл. 5 и 6).

Таблица 5. Оценка влияния размера блока S на время обработки (в секундах) по сравнению с программной реализацией с буферизацией столбца, компилятор Microsoft Visual Studio 2012, с оптимизациями компилятора, ядро Haswell

Размер-ность задачи	**Размер блока S**							
	2	4	8	16	32	64	128	256
256×256	0,010	0,008	0,007	0,008	0,011	0,011	0,013	0,014
512×512	0,117	0,079	0,057	0,062	0,089	0,086	0,098	0,127
1024×1024	0,971	0,665	0,463	0,504	0,726	0,691	0,784	0,999
2048×2048	28,836	13,628	6,026	4,816	6,207	5,622	6,297	7,952

Таблица 6. Оценка влияния размера блока S на время обработки (в секундах) по сравнению с программной реализацией с буферизацией столбца, компилятор Microsoft Visual Studio 2012, с оптимизациями компилятора, ядро Allendale

Размер-ность задачи	**Размер блока S**							
	2	4	8	16	32	64	128	256

256×256	0,031	0,026	0,021	0,025	0,026	0,028	0,049	0,049
512×512	0,329	0,230	0,172	0,207	0,209	0,205	0,391	0,391
1024×1024	9,908	4,997	2,365	2,049	1,764	1,652	3,151	3,106
2048×2048	78,719	41,142	18,822	16,411	14,111	13,226	25,254	24,963

Анализ приведенных данных показывает, что в каждом конкретном случае (размерность задачи, процессор) имеет место вполне определенной оптимальный размер блока *S*, при котором время обработки минимально. По-видимому, данный эффект связан с микроархитектурными особенностями реализации кэш-памяти для каждого процессора (разбиение на кэши 1, 2 и 3 уровня, их латентность, ассоциативность и пропускная способность, пропускная способность связывающих их шин и т.д.).

Дополнительного снижения времени выполнения умножения можно добиться путем раскрутки внутреннего цикла. При этом, например, для матриц размером 2048×2048 время обработки для выбранных процессоров сокращается с 5,622 с до 3,090 с (ядро Haswell, оптимальный размер блока *S* изменился с 16 на 64, $P = 5{,}6$ GFLOP/s) и с 13,226 с до 6,962 с (ядро Allendale, оптимальный размер блока $S = 64$, $P = 2{,}5$ GFLOP/s).

Сравнение блочного умножения и умножения с буферизацией столбца позволяет сделать вывод о том, что для ядра Haswell оба варианта приводят к достижению сопоставимой производительности, в то время как для ядра Allendale блочное умножение является предпочтительным. Процессоры семейства Intel Core демонстрируют в поставленной задаче реальную производительность на уровне 2,5 – 6,8 GFLOP/s при однопоточной обработке данных, что в десятки раз превосходит производительность «наивного» варианта умножения без оптимизации работы с памятью.

Список литературы

1. Ватутин Э.И., Зотов И.В. Построение матрицы отношений в задаче оптимального разбиения параллельных управляющих алгоритмов // Известия курского государственного технического университета. Курск, 2004. № 2. С. 85–89.
2. Intel 64 and IA-32 Architectures Software Developer’s Manual. Volume 1: Basic Architecture. Order number 253665-021.