УДК 681.3
Э.И. Ватутин, канд. техн. наук, доцент, кафедра вычислительной техники, ЮЗГУ (e-mail: evatutin@rambler.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 С ИСПОЛЬЗОВАНИЕМ РАСШИРЕНИЯ SSE (ЧАСТЬ 1)

Приведено описание подходов к выполнению операции умножения матриц, базирующихся на известных однопоточных программных реализациях и отличающихся наличием векторизации кода с использованием intrinsic'ов SSE-расширения системы команд процессоров x86. Показано, что для матриц различного размера различные подходы характеризуются различной эффективностью, что позволяет достичь реальную производительность на уровне 4,3–16,2 GFLOP/s для однопоточной SSE-ориентированной реализации операции на современных процессорах Intel Core (Haswell), что в 1,7–2,4 раза превосходит аналогичные результаты скалярных высокоуровневых реализаций без использования векторизации.

Ключевые слова: умножение матриц, программная оптимизация, SSE, параллельное программирование, векторизация кода

***

Одной из задач, имеющей важное значение для ряда научно-технических направлений (томография, компьютерная графика, проектирование роботизированных средств, транзитивное замыкание бинарных отношений [1] и др.), является задача умножения матриц. Время ее решения во многих случаях является бутылочным горлышком (англ. bottleneck), 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, систолических или реконфигурируемых вычислительных структур [2]). Несмотря на кажущуюся простоту и тривиальность решения, рассматриваемая задача характеризуется рядом особенностей, к которым в первую очередь можно отнести высокую степень параллелизма и однородности выполняемых операций, а также сильную зависимость времени вычисления от темпа поступления данных из памяти при умножении достаточно больших матриц.

В работе [3] описан ряд оптимизаций, оказывающих значительное влияние на время выполнения умножения для однопоточной программной реализации, ориентированной на выполнение на CPU, путем алгоритмической оптимизации, направленной на повышение эффективности работы кэш-памяти. При этом реальная производительность современных процессоров Intel Core в указанной задаче достигает величины 2,5–6,8 GFLOP/s в зависимости от модели.

В работе [4] приведено описание ряда алгоритмических подходов и деталей их программной оптимизации, ориентированных на решение задачи общего вида (умножение «плотных» квадратных матриц размера $N \times N$) с использованием видеокарт с поддержкой технологии CUDA [5] в рамках концепции GPGPU. При этом реальная производительность вычислительной системы, рассчитанная с учетом времени передачи матриц между оперативной памятью и глобальной памятью видеокарты с использованием интерфейса PCI Express [6], составляет величину 100–300 GFLOP/s в зависимости от аппаратной конфигурации компьютера (процессор, память, видеокарта, материнская плата).

Сопоставление полученных цифр реальной производительности CPU и GPU в рассматриваемой задаче позволяет сделать вывод о приблизительно 40–50-кратном выигрыше во времени обработки в пользу GPU. Учитывая однородность расположения данных в памяти, данный разрыв может быть сокращен путем использования векторных расширений системы команд процессора в соответствии с принципом SIMD [7]. С учетом того, что в исходной постановке [3–4] рассматривалась задача умножения матриц вещественных чисел одинарной точности, возможно использование расширения SSE с применением некоторых команд, входящих в состав расширения SSE3 (например, команды горизонтального сложения HADDPS). При этом одной командой производится параллельное выполнение 4 действий, что теоретически должно повысить производительность CPU в 4 раза. Однако в реальности данная цифра недостижима, т.к. имеет место ряд особенностей, подробно рассматриваемых ниже.

Простейший вариант векторного кода может быть получен отталкиваясь от реализации «в лоб» путем раскрутки внутреннего цикла на 4 с его последующей векторизацией (данная оптимизация не требует алгоритмических преобразований и может быть реализована автоматически «умным» компилятором с поддержкой генерации кода под SIMD-расширения системы команд процессора). Соответствующий ей код, написанный с использованием intrinsic’ов ассемблерных SSE-команд, приведен ниже.

```
for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
      {
            __m128 s;

            // Буфер для загрузки компонент вектора матрицы B
            __declspec(align(16)) float t[4];

            s = _mm_xor_ps(s, s);   // s = (0.0; 0.0; 0.0; 0.0)

            for (int k=0; k<N; k+=4)
            {
                  __m128 a = _mm_load_ps(&A[i][k]);
                  // a = (A[i][k]; A[i][k+1]; A[i][k+2]; A[i][k+3]);

                  t[0] = B[k][j];
```

```
            t[1] = B[k+1][j];
            t[2] = B[k+2][j];
            t[3] = B[k+3][j];

            __m128 b = _mm_load_ps(&t[0]);
            // b = (B[k][j]; B[k+1][j]; B[k+2][j]; B[k+3][j]);

            __m128 p = _mm_mul_ps(a, b);
            // p = a*b

            s = _mm_add_ps(s, p);
        }

        // s = (a; b; c; d)

        __m128 tmp_s = _mm_movehl_ps(s, s);
        // tmp_s = (a; b; a; b)

        s = _mm_add_ps(s, tmp_s);
        // s = (-; -; a+c; b+d)

        s = _mm_hadd_ps(s, s);
        // s = (-; -; -; a+b+c+d)

        C[i][j] = s.m128_f32[0];
    }
```

Результаты исследования реальной производительности CPU для данной реализации и ее модификации с раскруткой внутреннего цикла на 8 приведены на рис. 1.

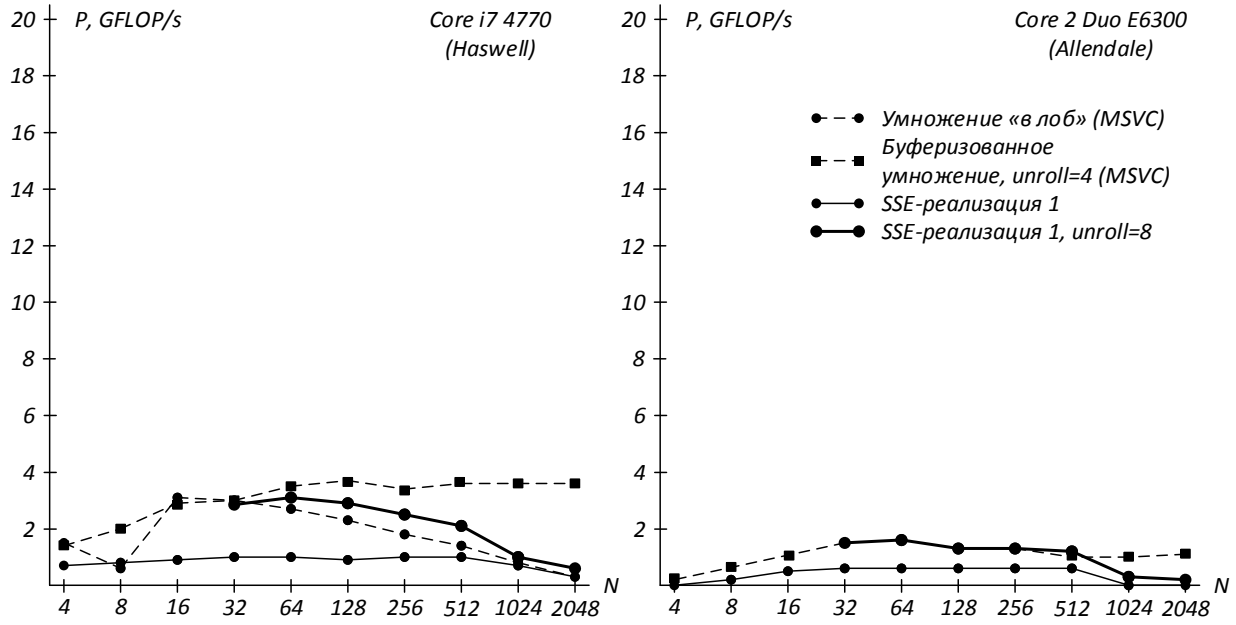

Рис. 1. Зависимость реальной производительности *P* от размера матриц *N* для различных вариантов высокоуровневой реализации [2] и SSE-реализации с векторизацией внутреннего цикла

Замечание. Все данные производительности получены для 32-битной программной реализации с использованием компилятора языка C++,

входящего в состав IDE Microsoft Visual Studio 2012, в конфигурации Release с ключами компилятора по умолчанию.

Изучение полученных результатов позволяет сделать вывод о том, что предложенный вариант SSE-реализации проигрывает в ряде случаев как исходному высокоуровневому коду, так и разработанному ранее буферизованному умножению [3], в особенности на матрицах большого размера. Чтобы разобраться в причинах, необходимо изучить паттерн обращений в память (рис. 2).

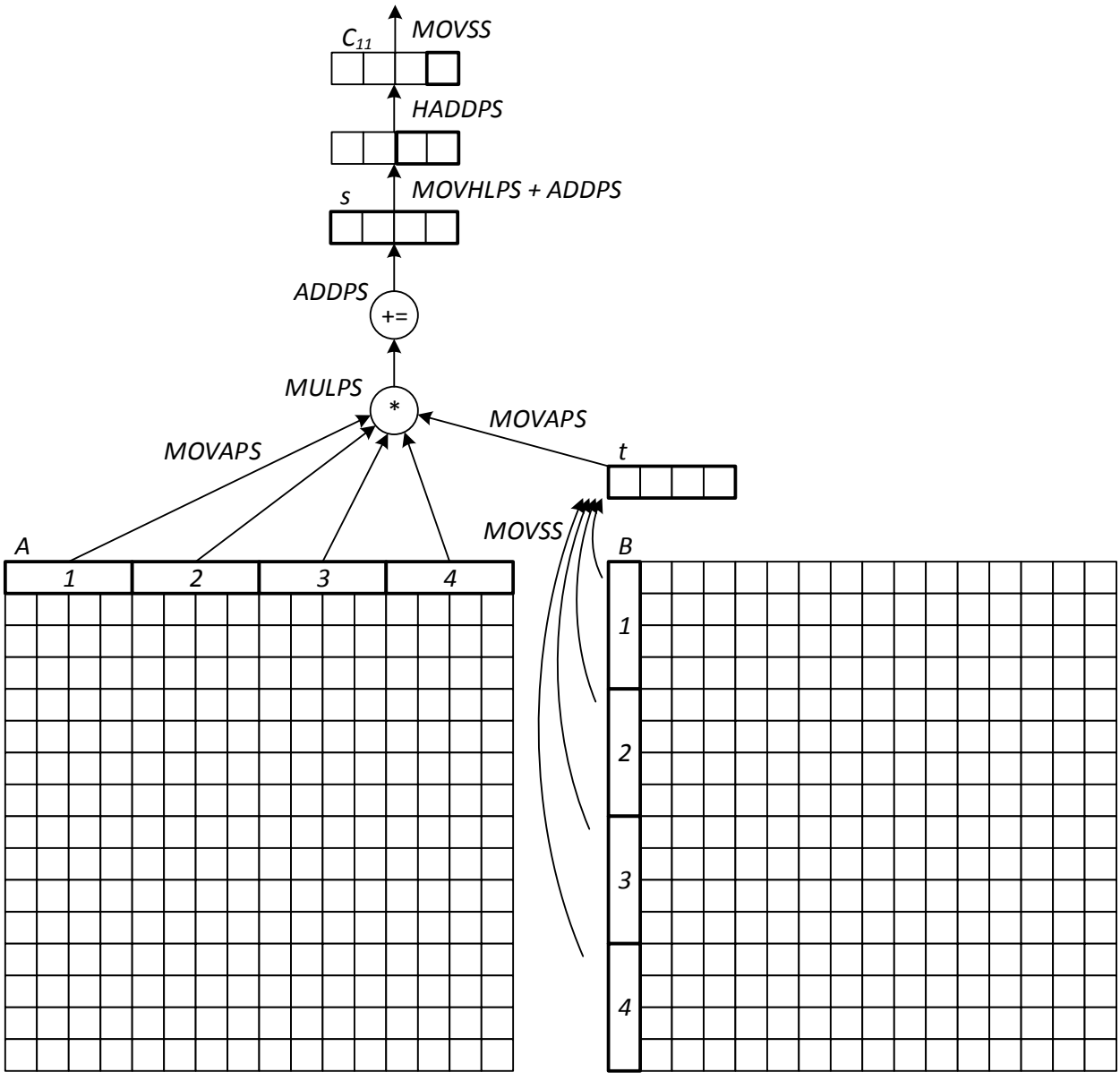

Рис. 2. Параллельная векторная обработка с использованием расширения SSE: векторизация внутреннего цикла

Первым существенным недостатком, свойственным исходной «наивной» реализации умножения, взятой за основу при векторизации кода,

являются обращения к *j*-му столбцу матрицы *B*, которые производятся прыжками и приводят к неэффективной работы механизма аппаратной предвыборки (англ. hardware prefetch) и кэш-памяти, что уже было отмечено в [3]. Кроме того, для формирования вектора $\left(b_{k,j}, b_{k+1,j}, b_{k+2,j}, b_{k+3,j}\right)$ в XMM-регистре необходимо дополнительное промежуточное копирование указанных значений в буфер *t* (4 скалярные команды MOVSS) с последующей загрузкой в регистр командой MOVAPS (при этом формирование вектора $\left(a_{i,k}, a_{i,k+1}, a_{i,k+2}, a_{i,k+3}\right)$ эффективно производится одной командой MOVAPS).

Избавиться от указанных недостатков можно путем следования стратегии буферизованного умножения [3], программная реализация которой приведена ниже.

```
__declspec(align(16)) 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++)
      {
            __m128 s;
            s = _mm_xor_ps(s, s);

            for (int k=0; k<N; k+=4)
            {
                  __m128 a = _mm_load_ps(&A[i][k]);
                  __m128 b = _mm_load_ps(&buf[k]);

                  __m128 p = _mm_mul_ps(a, b);

                  s = _mm_add_ps(s, p);
            }

            __m128 tmp_s = _mm_movehl_ps(s, s);
            s = _mm_add_ps(s, tmp_s);
            s = _mm_hadd_ps(s, s);

            C[i][j] = s.m128_f32[0];
      }
}
```

Данный вариант реализации, как и его скалярный прототип [3], позволяет более эффективное использование кэш-памяти, учитывает алгоритмы работы подсистемы аппаратной предвыборки данных при последовательном обходе области памяти и характеризуется возможностью эффективного формирования вектора $\left(b_{k,j}, b_{k+1,j}, b_{k+2,j}, b_{k+3,j}\right)$ с использованием одной команды MOVAPS. Зависимость реальной производительности от размера обрабатываемых матриц приведена на рис. 3, а соответствующий паттерн обращений в память – на рис. 4.

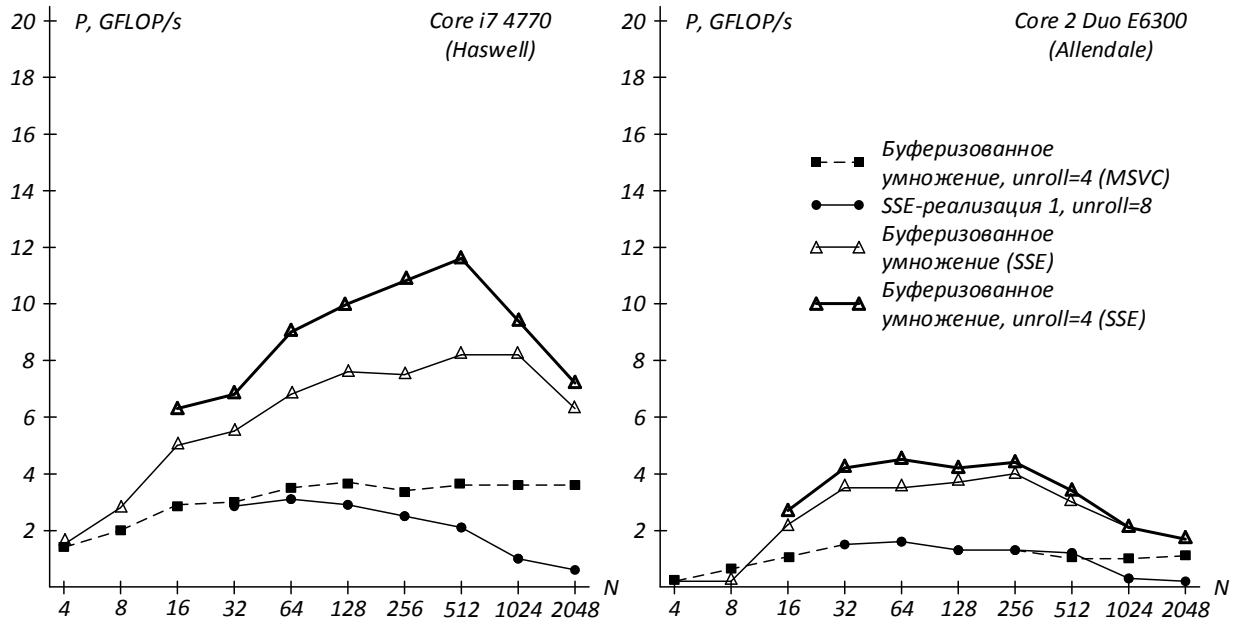

Рис. 3. Зависимость реальной производительности P от размера матриц N для различных вариантов высокоуровневой реализации [3] и SSE-реализации с буферизацией j-го столбца матрицы B

Приведенные зависимости показывают высокую эффективность данной программной реализации (с раскруткой внутреннего цикла на 4), превосходящей скалярную реализацию буферизованного умножения [3] по реальной производительности в 2–3 раза. На матрицах малого размера накладные расходы, связанные с буферизацией столбца, являются существенными по сравнению с вычислительными операциями умножения и сложения, что снижает достигнутое значение реальной производительности по сравнению с умножением матриц среднего размера. При обработке матриц размером $N > 256 \div 512$ наблюдается деградация производительности, по видимому связанная с ограниченным темпом поступления данных из оперативной памяти.

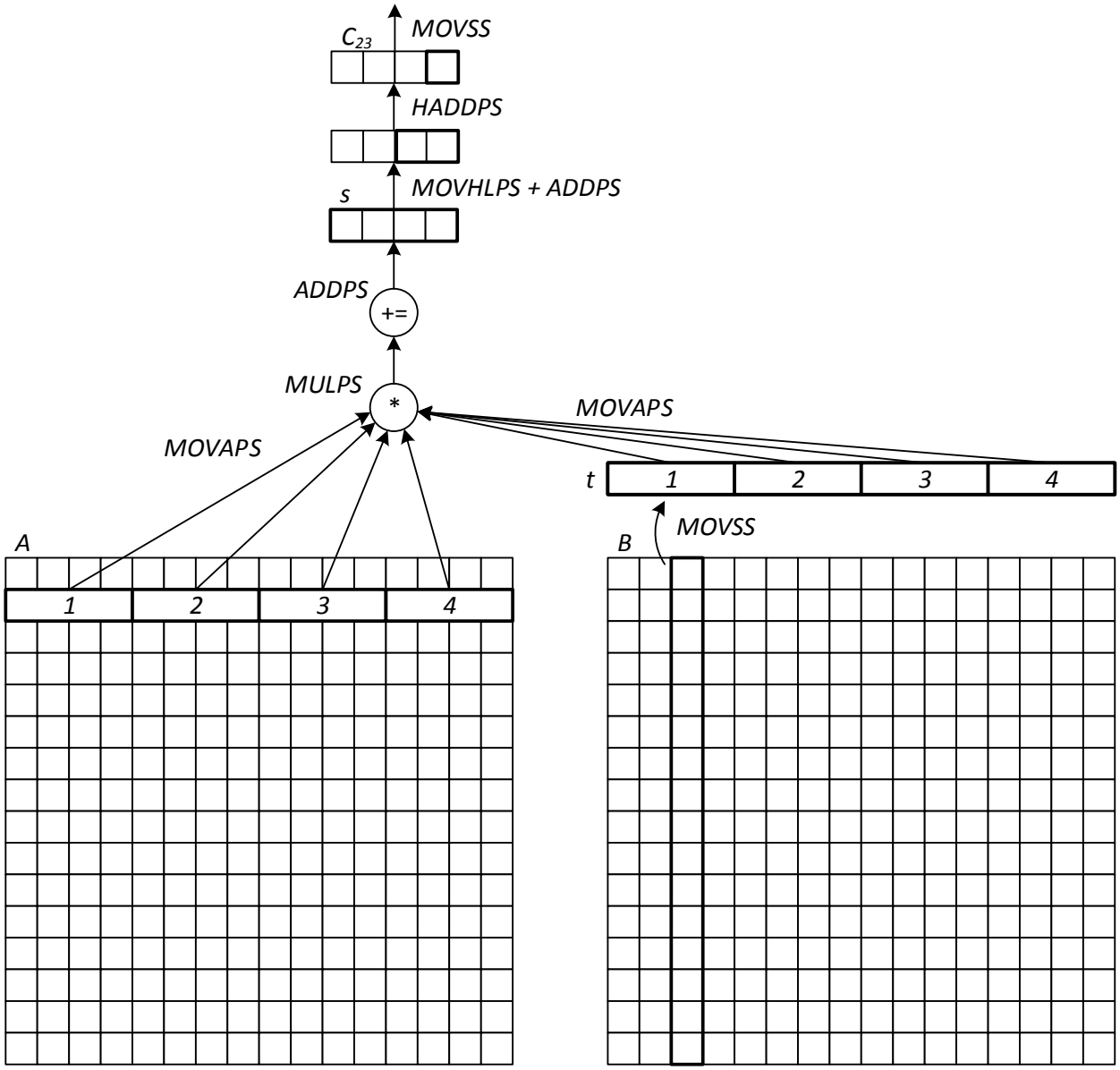

Рис. 4. Параллельная векторная обработка с использованием расширения SSE: буферизация *j*-го столбца матрицы *B*, изменение вложенности циклов, векторизация внутреннего цикла

Еще одним вариантом реализации, допускающим эффективную векторизацию, является блочное умножение [3]. Особенностью его векторной реализации является возможность транспонирования «на лету» текущей подматрицы матрицы *B* во время ее загрузки в кэш (см. рис. 5), что впоследствии позволяет использовать команды MOVAPS при формировании вектора $\left(b'_{k,y}, b'_{k+1,y}, b'_{k+2,y}, b'_{k+3,y}\right)$. Соответствующая программная реализация приведена ниже.

```
const int BLOCK_SIZE = 64;

__declspec(align(16)) float Az[BLOCK_SIZE][BLOCK_SIZE];
__declspec(align(16)) float Bz[BLOCK_SIZE][BLOCK_SIZE];
```

```
// x0 и y0 — координаты верхнего левого угла блока
for (int x0 = 0; x0 < N; x0 += BLOCK_SIZE)
      for (int y0 = 0; y0 < N; y0 += BLOCK_SIZE)
      {
            __declspec(align(16)) 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
                  // x и y — координаты смещения в пределах блока
                  for (int x = 0; x < BLOCK_SIZE; x++)
                        for (int y = 0; y < BLOCK_SIZE; y++)
                        {
                              Az[x][y] = A[x0+x][zb+y];

                              // При загрузке в кэш подматрица Bz
                              // транспонируется на лету
                              Bz[y][x] = 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++)
                        {
                              __m128 s;
                              s = _mm_xor_ps(s, s);

                              for (int k = 0; k < BLOCK_SIZE; k+=4)
                              {
                                    //s += Az[x][k] * Bz[y][k];
                                    __m128 a = _mm_load_ps(&Az[x][k]);
                                    __m128 b = _mm_load_ps(&Bz[y][k]);

                                    __m128 p = _mm_mul_ps(a, b);

                                    s = _mm_add_ps(s, p);
                              }

                              __m128 tmp_s = _mm_movehl_ps(s, s);
                              s = _mm_add_ps(s, tmp_s);
                              s = _mm_hadd_ps(s, s);

                              sum[x][y] += s.m128_f32[0];
                        }
            }

            // 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];
      }
```

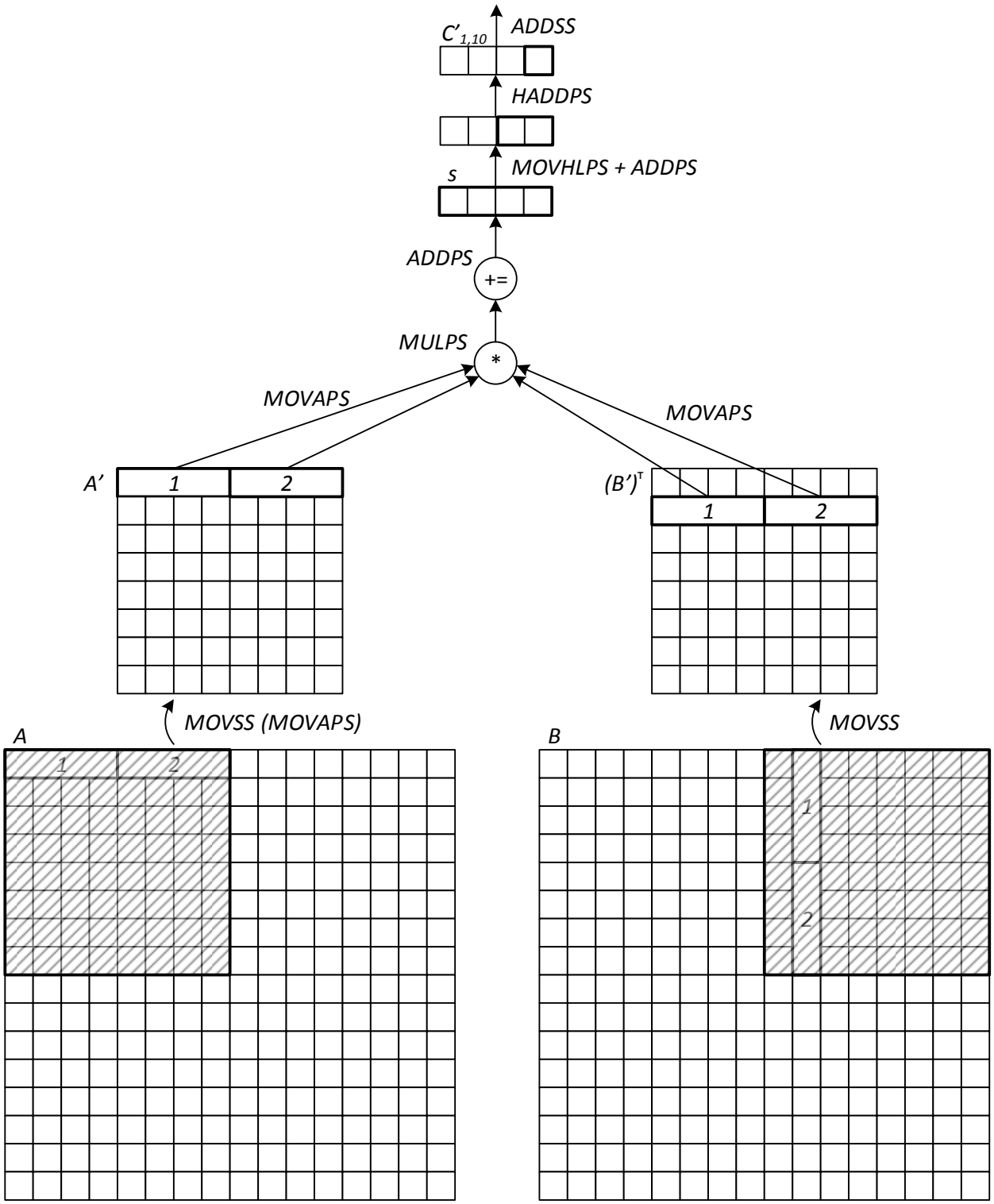

Рис. 5. Параллельная векторная обработка с использованием расширения SSE: блочное умножение, транспонирование подматрицы B' «на лету», векторизация внутреннего цикла при умножении блоков

Зависимости достигнутой реальной производительности от размера умножаемых матриц приведены на рис. 6.

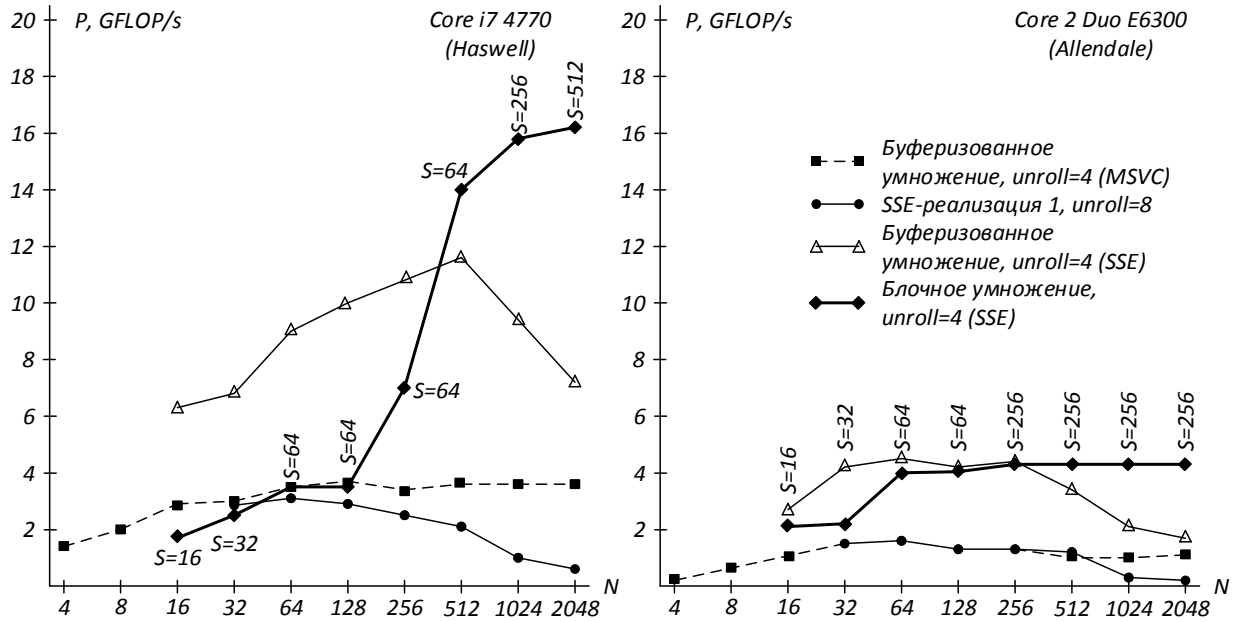

Рис. 6. Зависимость реальной производительности P от размера матриц N для различных вариантов высокоуровневой реализации [3] и SSE-реализаций буферизованного и блочного умножения с размером блока S, обеспечивающим минимальное время обработки

В ходе анализа полученных результатов можно заметить, что для умножения матриц малого и среднего размера блочный подход уступает буферизованному умножению, в то время как при $N \geq 512$ он является наиболее эффективным и позволяет достичь реальной производительности 16,2 GFLOP/s для процессора Intel Core i7 4770 и 4,3 GFLOP/s для процессора Intel Core 2 Duo E6300, что превосходит приведенные в [3] результаты в 2,4 и 1,7 раза соответственно (при теоретическом максимальном выигрыше в 4 раза). При этом производительность современных GPU в рассматриваемой задаче, составляющая 100–570 GFLOP/s [4, 8], для данного набора векторных CPU-реализаций программного кода является недостижимой. Приведенные зависимости показывают, что для матриц различного размера N имеет место различный оптимальный размер блока S, что может быть учтено путем адаптивной подстройки размера блока под конкретные условия применения (размер умножаемых матриц и целевой процессор).

Библиографический список

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. Кун С.Ю. Матричные процессоры на СБИС. М.: Мир, 1991. 672 с.
3. Ватутин Э.И., Мартынов И.А., Титов В.С. Оценка реальной производительности современных процессоров в задаче умножения матриц для однопоточной программной реализации // Известия Юго-Западного государственного университета. Серия: Управление,

вычислительная техника, информатика. Медицинское приборостроение. 2013. № 4. С. 11–20.
4. Ватутин Э.И., Мартынов И.А., Титов В.С. Оценка реальной производительности современных видеокарт с поддержкой технологии CUDA в задаче умножения матриц // Известия Юго-Западного государственного университета. Серия: Управление, вычислительная техника, информатика. Медицинское приборостроение. 2014. № 2. С. 8–17.
5. Параллельные вычисления на GPU. Архитектура и программная модель CUDA / Боресков А.В., Харламов А.А. Марковский Н.Д. и др. М.: изд-во Московского университета, 2012. 336 с.
6. Мартынов И.А., Ватутин Э.И. Измерение реальной пропускной способности шины PCI Express с использованием видеокарт с поддержкой технологии CUDA в качестве периферийных устройств // Оптико-электронные приборы и устройства в системах распознавания образов, обработки изображений и символьной информации (Распознавание – 2015). Курск, 2015. С. 242–244.
7. Intel 64 and IA-32 Architectures Software Developer’s Manual. Volume 1: Basic Architecture. Order number 253665-021. 2013.
8. Ватутин Э.И., Мартынов И.А., Титов В.С. Оценка реальной производительности современных процессоров и видеокарт с поддержкой технологии CUDA в задаче умножения матриц // CUDA альманах (май 2015). 2015. С. 9–10.