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 2)

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 без использования горизонтальных операций. Показано, что для матриц различного размера различные подходы характеризуются различной эффективностью, что позволяет достичь реальную производительность на уровне 7,5–18,5 GFLOP/s для однопоточной SSE-ориентированной реализации операции на современных процессорах Intel Core, что в 2,7–3 раза превосходит аналогичные результаты скалярных высокоуровневых реализаций без поддержки векторизации. Сформулированы рекомендации по эффективности применения различных программных реализаций для умножения матриц различного размера.

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

***

Данная работа является логическим продолжением работ [1, 2] в области исследования эффективности однопоточных программных реализаций операции умножения плотных квадратных матриц различного размера с использованием высокоуровневой программной реализации и реализации, ориентированной на использование расширения SSE системы команд процессора с использование intrinsic'ов.

Одним из недостатков программных реализаций [2] является то, что векторизация производится отталкиваясь от базового алгоритма «наивного», буферизованного или блочного умножения [1] путем векторизации внутреннего цикла. Данный подход не требует изменения логики работы объемлющих циклов и, как уже было отмечено в [2], может быть реализован автоматически «умным» компилятором, однако завершающим действием для каждого из элементов c_{ij} результирующей матрицы C является горизонтальное сложение компонент вектора

$$(s_1, s_2, s_3, s_4) \to (-, -, s_1 + s_3, s_2 + s_4) \to (-, -, -, s_1 + s_2 + s_3 + s_4)$$

с последующей записью одного значения с плавающей точкой одинарной точности в память, что требует использования ассемблерных команд MOVHLPS, HADDPS и MOVSS, имеющих RAW-зависимость по результату и, следовательно, выполняемых последовательно (частично этот недостаток нивелируется в ходе раскрутки цикла). Первые две из них не являются быстрыми (латентность MOVHLPS – 1–6 тактов в зависимости от модели CPU; латентность HADDPS – 5–13 тактов; для сравнения, латентность векторного сложения ADDPS – 3–5 тактов [3]), а третья создает большую

нагрузку на кэш-память и шину между процессором и оперативной памятью из-за большого числа запросов на запись данных (поток скалярных записей с использованием MOVSS размером 4 байта каждая вместо в 4 раза меньшего количества векторных записей с использованием MOVAPS размером 16 байт каждая).

Указанный недостаток может быть исправлен путем изменения логики работы используемых алгоритмов таким образом, чтобы в процессе вычисления были использованы только векторные команды, что в том числе изменяет паттерн обращений в память. С целью апробации целесообразности рассмотренной выше оптимизации была разработана модифицированная SSE-ориентированная версия рассмотренного в [1] «наивного» умножения.

```
for (int i=0; i<N; i++)
      for (int j=0; j<N; j+=4)  // Шаг цикла изменен с 1 на 4!
      {
            __m128 s;
            s = _mm_xor_ps(s, s);

            for (int k=0; k<N; k++)
            {
                  __m128 a = _mm_load_ss(&A[i][k]);
                  a = _mm_shuffle_ps(a, a, _MM_SHUFFLE(0, 0, 0, 0));

                  __m128 b = _mm_load_ps(&B[k][j]);

                  __m128 p = _mm_mul_ps(a, b);

                  s = _mm_add_ps(s, p);
            }

            _mm_store_ps(&C[i][j], s);
      }
```

Она отличается от реализации-прототипа изменением паттерна обращений в память (рис. 1), изменением приращения среднего (по *j*) цикла (с 1 на 4) и искомым отсутствием горизонтальных операций. Результаты измерения реальной производительности приведены на рис. 2.

MOVAPS

C_{11}-C_{14}

ADDPS

+=

MULPS

*

MOVAPS

SHUFPS

A

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

B

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Рис. 1. Параллельная векторная обработка с использованием расширения SSE: изменен шаг среднего цикла, векторизация внутреннего цикла, реализация без использования горизонтальных операций

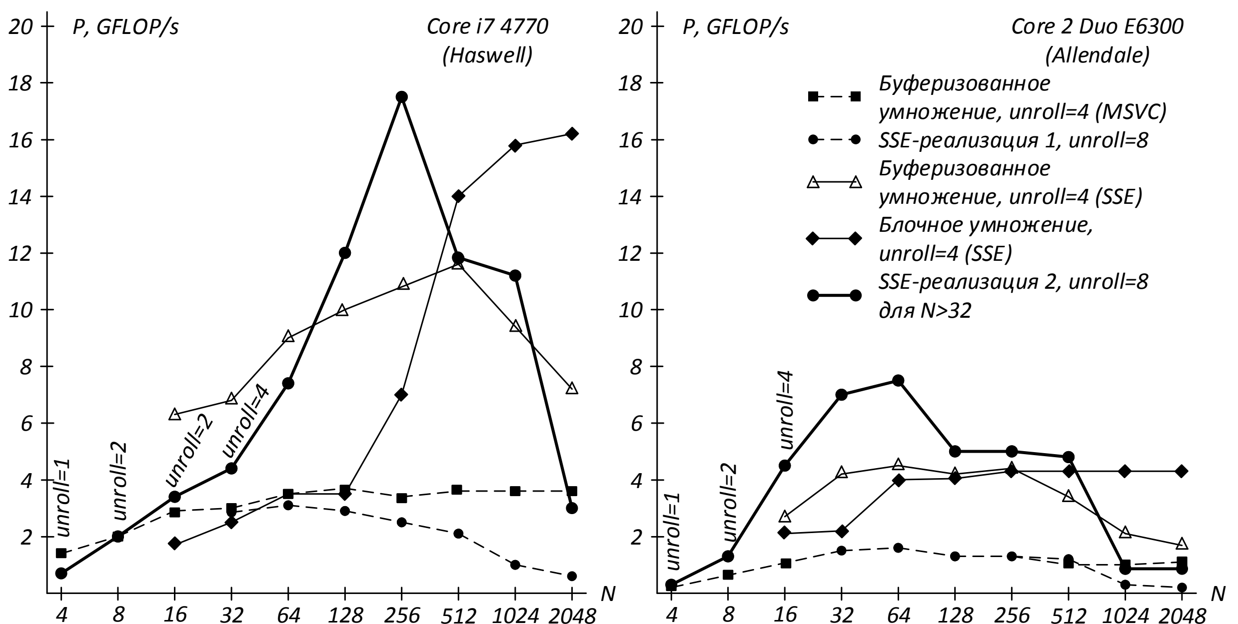

Рис. 2. Зависимость реальной производительности *P* от размера матриц *N* для лучших вариантов высокоуровневой реализации [1], лучших SSE-реализаций буферизованного и блочного умножения с использованием горизонтальных команд [2] и SSE-реализации наивного алгоритма без использования горизонтальных команд

Замечание. Следуя статье [2], 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.

Полученные зависимости доказывают эффективность применения векторных команд взамен горизонтальных, демонстрируя превосходство данной реализации (SSE-реализация 2) как над предыдущей (SSE-реализация 1), так и над векторизованным блочным и буферизованным умножением для некоторых значений *N*. Несмотря на высокую эффективность для матриц малого и среднего размера, с ростом *N* происходит исчерпание емкости кэш-памяти и реальная производительность существенно падает, что, как уже было отмечено в [1, 2], может быть исправлено путем использования буферизованного или блочного умножения.

Реализация буферизованного умножения также может быть разработана без использования горизонтальных операций.

```
__declspec(align(16)) float SseBuf[4*N];

for (int j=0; j<N; j+=4)  // Шаг цикла изменен с 1 на 4!
{
      for (int k=0; k<N; k++)
      {
                  __m128 t = _mm_load_ps(&B[k][j]);
                  _mm_store_ps(&SseBuf[k*4], t);
      }

      for (int i=0; i<N; i++)
      {
            __m128 s;
            s = _mm_xor_ps(s, s);

            for (int k=0; k<N; k++)
            {
                  __m128 a = _mm_load_ss(&A[i][k]);
                  a = _mm_shuffle_ps(a, a, _MM_SHUFFLE(0, 0, 0, 0));

                  __m128 b = _mm_load_ps(&SseBuf[k*4]);

                  __m128 p = _mm_mul_ps(a, b);

                  s = _mm_add_ps(s, p);
            }

            _mm_store_ps(&C[i][j], s);
      }
}
```

Ее отличительными особенностями по сравнению с [2] являются поэлементная загрузка компонент матрицы A, буферизация 4 смежных столбцов матрицы B вместо 1 и изменение приращения счетчика внешнего цикла с 1 на 4. Соответствующий паттерн обращений в память приведен на рис. 3, а результаты измерения реальной производительности – на рис. 4.

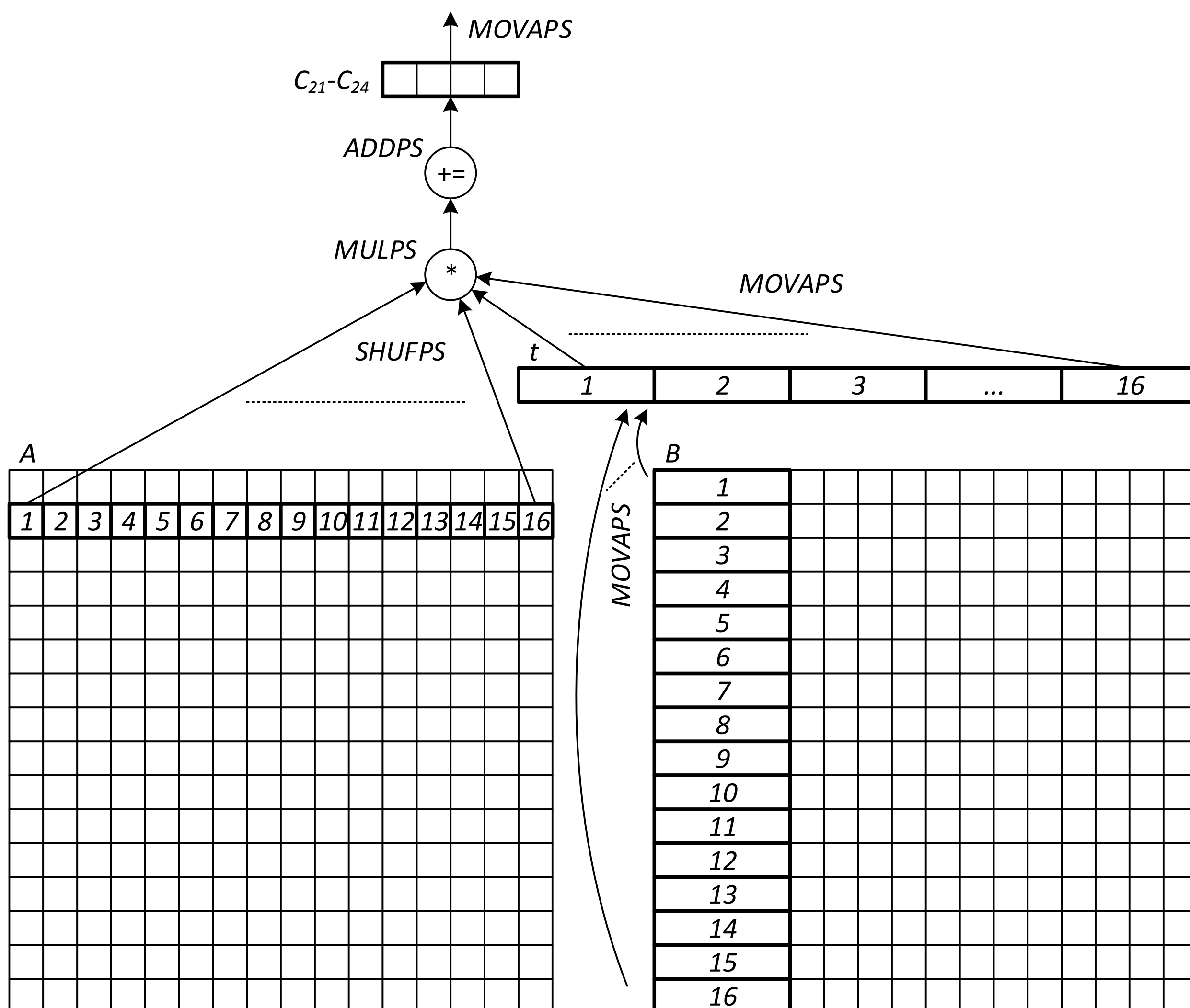

Рис. 3. Параллельная векторная обработка с использованием расширения SSE: изменен порядок циклов, буферизация четырех столбцов, векторизация внутреннего цикла, реализация без использования горизонтальных операций

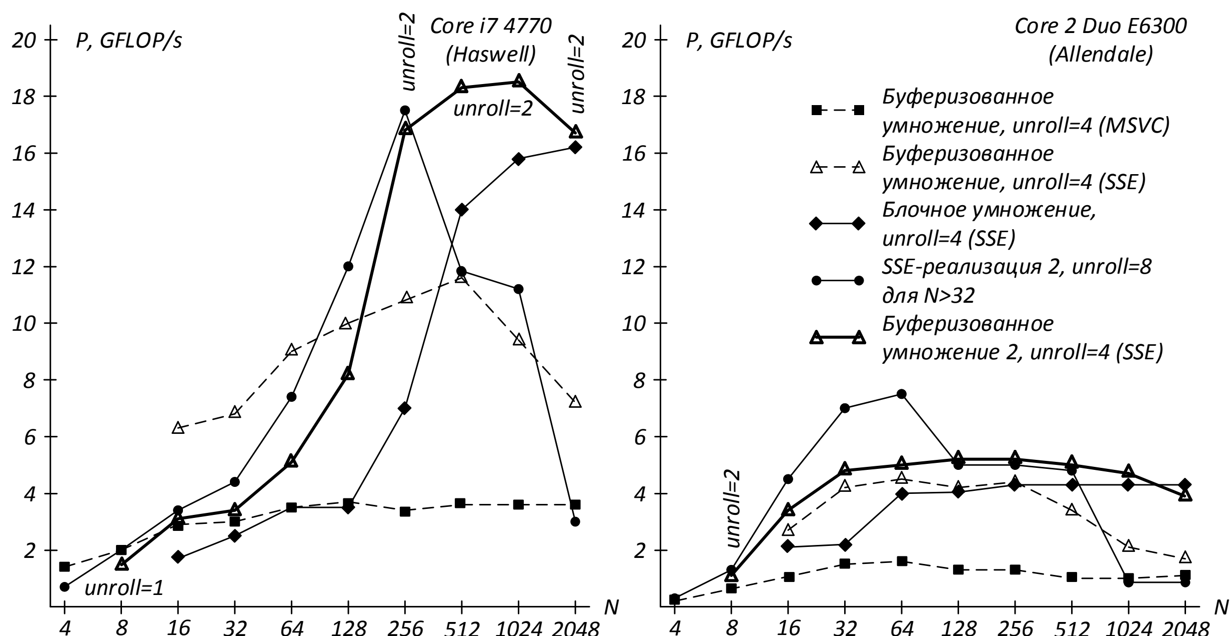

Рис. 4. Зависимость реальной производительности P от размера матриц N для лучших вариантов высокоуровневой реализации [1], лучших SSE-реализаций буферизованного и блочного умножения с использованием горизонтальных команд [2] и SSE-реализации буферизованного умножения без использования горизонтальных команд

Приведенные на рис. 4 зависимости демонстрируют высокую эффективность буферизованного умножения без горизонтальных операций при умножении матриц размера $N > 128 \div 256$, уступая на матрицах меньшего размера предыдущей SSE-реализации без горизонтальных команд. При этом для матриц разного размера эффективными являются реализации с различной степенью раскрутки цикла, что может быть использовано при адаптивной настройке программной реализации под конкретный процессор.

Код программной реализации блочного умножения без команд горизонтальных сложений приведен ниже.

```
const int BLOCK_SIZE = 64;

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

// Перебор блоков (x0 и y0 — координаты начала 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
			// 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];
					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+=4)
				{
					__m128 s;
					s = _mm_xor_ps(s, s);

					for (int k = 0; k < BLOCK_SIZE; k++)
					{
						__m128 a = _mm_load_ss(&Az[x][k]);
						a = _mm_shuffle_ps(a, a,
							_MM_SHUFFLE(0, 0, 0, 0));

						__m128 b = _mm_load_ps(&Bz[k][y]);

						__m128 p = _mm_mul_ps(a, b);

						s = _mm_add_ps(s, p);
					}

					__m128 tmp_s = _mm_load_ps(&sum[x][y]);
					tmp_s = _mm_add_ps(tmp_s, s);

					_mm_store_ps(&sum[x][y], tmp_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];
	}
```

Его отличительными особенностями являются загрузка компонент текущей подматрицы матрицы A по 1 на цикл, загрузка 4 компонент смежных столбцов текущей подматрицы матрицы B (при этом, как видно из рис. 5, не требуется транспонирование подматрицы B') и изменение шага вложенного цикла с 1 на 4.

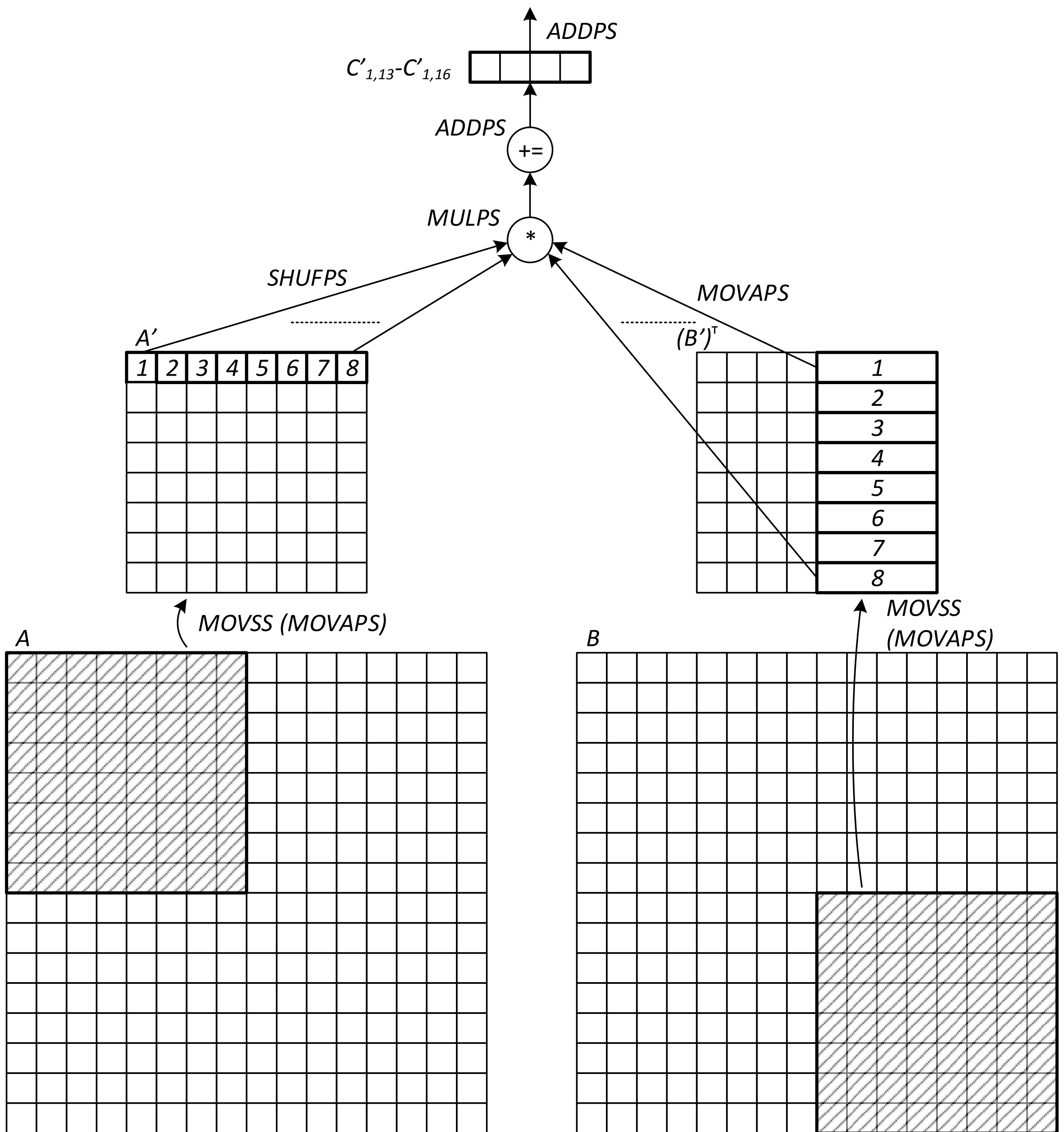

Рис. 5. Рис. 3. Параллельная векторная обработка с использованием расширения SSE: блочное умножение, векторизация внутреннего цикла при умножении блоков, реализация без использования горизонтальных операций

Достигаемая при использовании приведенного кода реальная производительность приведена на рис. 6.

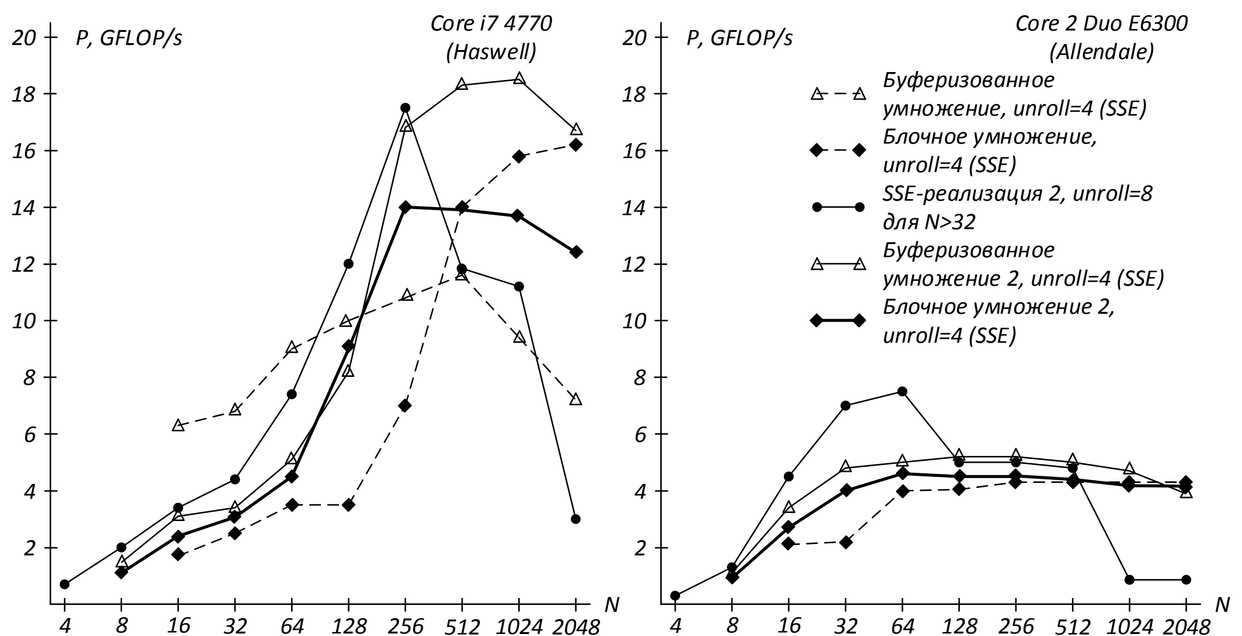

Рис. 6. Зависимость реальной производительности P от размера матриц N для лучших SSE-реализаций буферизованного и блочного умножения с использованием горизонтальных команд [2] и SSE-реализации блочного умножения без использования горизонтальных команд

Анализ полученных результатов позволяет сделать вывод о том, что вариант реализации блочного умножения без использования горизонтальных операций лучше рассмотренного в [2] варианта с горизонтальным сложением, однако он уступает «наивному» и буферизованному SSE-умножению без горизонтальных операций.

Анализ приведенных в [2] и на рис. 2, 4 и 6 зависимостей позволяет сделать вывод о том, что использование горизонтальных операций приводит к деградации реальной производительности. Программные реализации без их использования позволяют получить в рассматриваемой задаче производительность 18,5 GFLOP/s на процессоре Intel Core i7 4770 (в работе [2] максимальная достигнутая производительность на данном процессоре – 16,2 GFLOP/s) и 7,5 GFLOP/s на процессоре Intel Core 2 Duo E6300 (в работе [2] – 4,3 GFLOP/s), что характеризуется выигрышем в 2,7–3 раза по сравнению со скалярной реализацией матричного умножения [1]. При этом реальная производительность современных GPU, составляющая величину 100–570 GFLOP/s [4–5], является существенно большей.

Для конкретного размера умножаемых матриц N может быть подобрана оптимальная программная реализация, специфичная для конкретного процессора, что, как уже было отмечено выше, может быть использовано при адаптивной настройке используемой программной реализации под целевой процессор (при этом для матриц малой размерности с $N < 8 \div 16$ могут быть разработаны специальные упрощенные программные реализации без циклов, позволяющие выполнение умножения с максимальной производительностью):

- $N < 8$ – специализированная программная SSE-реализация без циклов и горизонтальных операций;
- $8 < N < 256$ (Intel Core i7 4770), $8 < N < 64$ (Intel Core 2 Duo E6300) – SSE-реализация «наивного» умножения без горизонтальных операций с максимально возможной степенью раскрутки внутреннего цикла для выбранного N, но не более 8;
- $N > 256$ (Intel Core i7 4770), $N > 64$ (Intel Core 2 Duo E6300) – буферизованное SSE-умножение без горизонтальных операций с раскруткой внутреннего цикла на 2–4 итерации.

Для матриц, размер которых превышает емкость кэш-памяти процессора, наряду с буферизованным эффективной следует считать SSE-реализацию блочного умножения без горизонтальных операций. Указанные особенности могут быть учтены при разработке высокоэффективной однопоточной CPU-реализации матричного умножения, ориентированной на выполнение в гетерогенных вычислительных средах (например, в гридах) с существенно различным составом аппаратного обеспечения.

Библиографический список

1. 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.
2. Ватутин Э.И., Титов В.С. Оценка реальной производительности современных процессоров в задаче умножения матриц для однопоточной программной реализации с использованием расширения SSE (часть 1) // Известия Юго-Западного государственного университета. 2015. Т. 1. № 4 (61). С. 26–35.
3. Intel 64 and IA-32 Architectures Optimization Reference Manual. Order number 248966-028. 2013.
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. 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.