Инженерный вестник Дона, №4 (2024)
ivdon.ru/ru/magazine/archive/n4y2024/9173

Оптимизация процедуры умножения плотных матриц для систем с общей памятью

В.А. Егунов, В.А. Шабаловский, Д.М. Дудкин

Волгоградский государственный технический университет

Аннотация: Исследование представляет обширный анализ методов низкоуровневой оптимизации алгоритма умножения матриц для вычислительных систем с общей памятью. Опираясь на сравнение различных подходов, включая блочную оптимизацию, параллельное выполнение с OpenMP, векторизацию с AVX и использование библиотеки Intel MKL, выявляются значительные улучшения в производительности полученных программных реализаций. В частности, блочная оптимизация сокращает количество кеш-промахов, параллелизм эффективно задействует многоядерность, а векторизация и Intel MKL демонстрируют максимальное ускорение за счет более эффективных программных оптимизаций. Полученные результаты подчеркивают важность тщательного выбора оптимизационных методов и их соответствия архитектуре вычислительной системы для достижения требуемых параметров эффективности проектируемого программного обеспечения.
Ключевые слова: низкоуровневая оптимизация, блочная оптимизация, параллельное выполнение, OpenMP, векторизация, AVX, Intel MKL, производительность, бенчмаркинг, умножение матриц,

Введение. В последние годы, на фоне растущего спроса на более эффективные и высокопроизводительные вычислительные решения, внимание к низкоуровневой оптимизации приложений значительно усилилось. Эта тенденция подчеркивает важность поиска и реализации методов оптимизации, которые могли бы повысить эффективность и производительность компьютерных систем [1]. В рамках этого исследования, мы фокусируемся на возможностях оптимизации программной реализации операции умножения плотных матриц GEMM (General Matrix Multiply) для систем с общей памятью. Операция GEMM является одной из ключевых в различных реализациях BLAS (Basic Linear Algebra Subprograms). Используя GEMM в качестве основы, мы сравниваем и анализируем производительность различных методов оптимизации: от блочной оптимизации и параллельной реализации с использованием OpenMP до векторизованной реализации с помощью AVX [2]. Кроме того, анализ

производительности библиотеки Intel MKL предоставляет ценное представление о том, как оптимизированные реализации могут улучшить производительность приложений [3].

Оптимизация вычислительных алгоритмов на системах с общей памятью может быть достигнута за счет использования параллелизма, векторизации и улучшения локальности доступа к данным [4]. Рассмотрим кеш-оптимизацию алгоритма умножения матриц, которая применяется для снижения частоты кеш-промахов, а также использование параллелизма и векторизации. Замер производительности осуществляется с помощью библиотеки Google Benchmark [5].

Базовая реализация алгоритма. Код реализации базового алгоритма умножения матриц приведен ниже.

```
void naive_ijk(double *const A, double *const B, double *const C, const int
     N) {
  for (int i = 0; i < N; i++) {
     for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
        C[i * N + j] += A[i * N + k] * B[k * N + j];
        }}}
}
```

Здесь и далее опускаются операции, связанные с предварительным обнулением элементов результирующей матрицы. Большое влияние на эффективность получаемого приложения оказывает порядок циклов. Здесь возможны варианты ijk, ikj, kij, kji, jki, jik. Различные варианты порядка следования циклов обеспечивают различный порядок обхода матриц, а, следовательно, и различную частоту кеш-промахов. Все замеры времени производятся на выполнении операций GEMM с матрицами размерностью 1024x1024, которые генерируются случайным образом, используется тип double. На рис.1 приведены результаты выполнения бенчмарка.

Как видно из рисунка, функции вида naïve_**i имеют самые плохие показатели. Это связано с тем, что переменная i является индексом строки

при обращении к массиву A. Таким образом, на каждой итерации внутреннего цикла происходят промахи кеш-памяти.

В свою очередь, наилучшие результаты имеют функции, в которых циклы с i и k находятся на первых двух местах. Это связано с тем, что эти переменные являются индексами строк в двух циклах, они меняются реже, а следовательно, и строки соответствующих матриц остаются к кеше дольше. Для дальнейшего исследования и оптимизации используется алгоритм ijk.

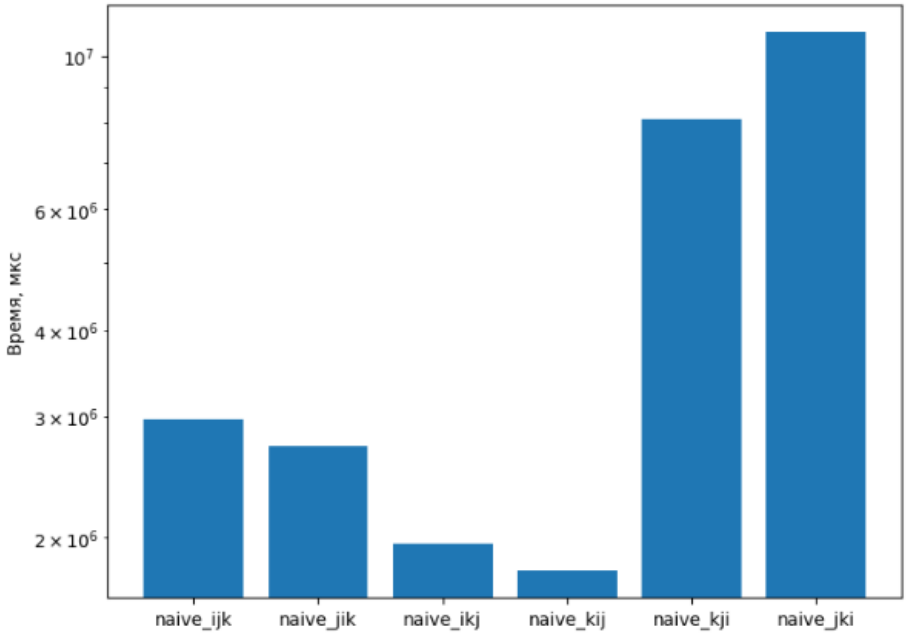

Рис. 1/ – Результат выполнения бенчмарка для базовой реализации

Блочная оптимизация базового алгоритма. Суть метода заключается в разбиении большой матрицу на блоки (подматрицы). Если блоки достаточно малы, чтобы поместиться в кеш (например, L1), то не нужно заботиться о том, будут ли циклы переупорядочены или нет [6]. Конечно, поскольку каждая координата x, y в выходной матрице C должна быть результатом умножения всего вектора строки i из матрицы A на весь вектор столбца j из матрицы B, извлечение маленьких квадратных блоков из матрицы из обеих матриц даст только частичные результаты для каждой выходной координаты, поэтому придется суммировать произведения, полученные в результате умножения отдельных блоков. Например, чтобы

вычислить синий блок в матрице C на рис. 2, нужно взять два частичных результата от умножения блоков.

Ниже представлена программная реализация блочного алгоритма.

```
template<int T>
void tiled(double* const A, double* const B, double* const C, const int N) {
    for (int i = 0; i < N; i += T) {
        for (int j = 0; j < N; j += T) {
            for (int k = 0; k < N; k += T) {
                const int minMt = std::min(i + T, N);
                const int minNt = std::min(j + T, N);
                const int minKt = std::min(k + T, N);
                for (int i_t = i; i_t < minMt; i_t++) {
                    for (int j_t = j; j_t < minNt; j_t++) {
                        for (int k_t = k; k_t < minKt; k_t++) {
                            C[i_t * N + j_t] += A[i_t * N + k_t] * B[k_t * N + j_t];
                        }}}
            }}}
}
```

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$+ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

Рис. 2. – Графическое пояснение блочной оптимизации

На рис. 3 показаны результаты выполнения бенчмарка для блочной реализации.

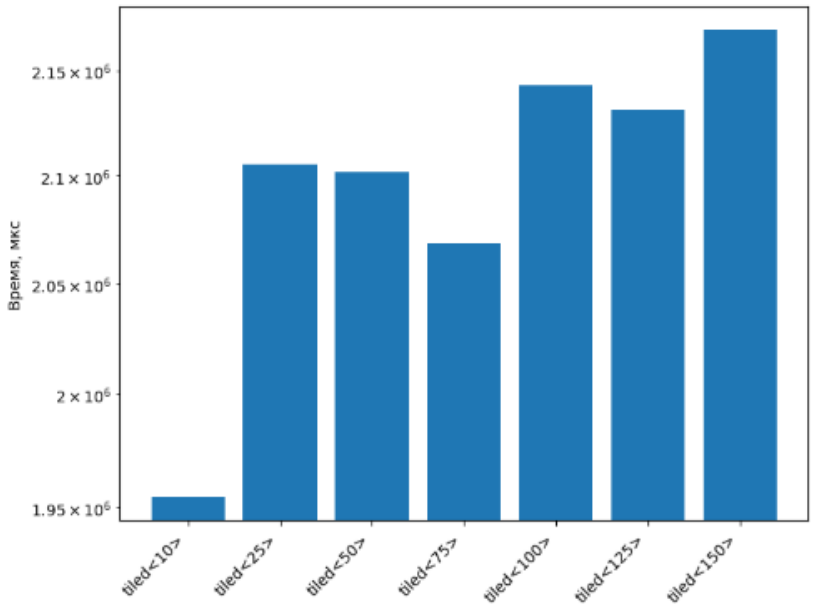

Рис. 3. – Результаты выполнения бенчмарка для блочной реализации

Таким образом, из рис. 1 видно, для расчета функции naive_ijk затрачивается 2,97 с., а в блочном варианте ijk лучшей является реализация tiled<10> с размером блока 10, которая выполняется за 1,95 с. Из этого следует, что блочная реализация выполняется эффективнее.

Параллельная реализация алгоритма с помощью OpenMP. Для реализации параллельной версии алгоритма используется версия базового алгоритма, а именно naive_ijk [7].

```
void parallel_ijk(double *const A, double *const B, double *const C, const int N){
#pragma omp parallel for
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      for (int k = 0; k < N; k++) {
        C[i * N + j] += A[i * N + k] * B[k * N + j];
      }}}
}
```

Вычислительные эксперименты проводились на процессоре Intel Core i7 12700, который имеет 8 производительных и 4 эффективных ядер, что в

сумме с гиперпоточностью даёт 20 потоков. Результаты экспериментов приведены в таблице 1.

Векторизация алгоритма. SIMD (Single Instruction, Multiple Data) представляет собой способ распараллеливания вычислений, используемый для одновременной обработки нескольких элементов данных с помощью одной инструкции. Это позволяет значительно повышать производительность по сравнению с традиционной последовательной обработкой [8,9].

Одной из реализаций SIMD для процессоров Intel является AVX (Advanced Vector Extensions), представляющая собой 256-битное расширение набора инструкций для архитектуры x86. AVX предоставляет более широкие векторы и богатый набор инструкций для данных с плавающей точкой и целых чисел. Данная технология реализует более высокую производительность для приложений, требующих больших вычислений и способных эффективно использовать векторы шириной 256 бит, обеспечивает эффективное использование конвейера процессора и кеш-памяти, что ведет к дельнейшему росту производительности [10].

Приведем реализацию векторизованной версии алгоритма.

```
void parallel_ijk_simd(double *const A, double *const B, double *const C,
                       const int N) {
#pragma omp parallel for
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j += 4) {
         auto sum = _mm256_setzero_pd();
         for (int k = 0; k < N; k++) {
            auto a = _mm256_set1_pd(A[i * N + k]);
            auto b = _mm256_loadu_pd(&B[k * N + j]);
            auto prod = _mm256_mul_pd(a, b);
            sum = _mm256_add_pd(sum, prod);
         }
         _mm256_storeu_pd(&C[i * N + j], sum);
      }}
}
```

Цикл с индексом j использует шаг (stride) в 4, что означает увеличение k на 4 при каждой итерации. Это позволяет выполнять векторные операции сразу над 4 последовательными элементами, обеспечивая параллельную обработку.

Функция инициирует процесс с установкой начального значения суммы в ноль, используя интринсик _mm256_setzero_pd. Этот шаг критически важен для корректной агрегации результатов умножения. Далее происходит вычисление произведения элементов матрицы A на соответствующие элементы строк матрицы B, с последующим накоплением сумм этих произведений. Для умножения и добавления используются интринсики _mm256_mul_pd и _mm256_add_pd соответственно.

По завершении вычислений накопленная сумма произведений сохраняется в соответствующей позиции результирующей матрицы C с помощью интринсика _mm256_storeu_pd.

Использование Intel MKL. Intel Math Kernel Library (Intel MKL) представляет собой математическую библиотеку, разработанную корпорацией Intel. Она предоставляет высоко оптимизированные многопоточные процедуры для таких математических функций, как линейная алгебра (BLAS), быстрое преобразование Фурье (БПФ) и ряд других.

Исследуем функцию cblas_dgemm. Это функция библиотеки Intel Math Kernel Library (Intel MKL) реализует операцию GEMM для значений типа double. Функция оптимизирована для процессоров Intel и демонстрирует высокую производительность.

В таблице 1 показано сравнение приведенных выше методов умножения плотных матриц. Из приведенных результатов видно, что с каждой новой оптимизацией время выполнения преобразования сокращается, растет полученное ускорение. Максимальное ускорение достигается при использовании функции библиотеки Intel MKL.

Таблица 1

Сравнение различных вариантов оптимизации операции GEMM

Реализация алгоритма	512x512		1024x1024	
	Время (мкс)	Ускорение	Время (мкс)	Ускорение
naive_ijk	396541	–	2973466	–
block_ijk	235154	1.7x	1954248	1.5x
parallel_ijk	41747	9.5x	437076	6.8x
simd_ijk	27595	14.4x	260636	11.4x
cblas_dgemm	1007	394x	8115	366x

Заключение. В работе рассмотрены различные методы низкоуровневой оптимизации алгоритма умножения матриц, ключевой операции во многих областях вычислений. Приведены различные подходы к оптимизации, включая базовую реализацию, блочную оптимизацию, параллельное выполнение с использованием OpenMP, векторизацию с использованием AVX и работа с библиотекой Intel MKL. Результаты показывают, что векторизация и использование специальных библиотек, таких как Intel MKL, могут значительно повысить производительность по сравнению с базовой реализацией алгоритма.

Сравнение различных подходов демонстрирует, что важно не только выбирать правильный метод оптимизации, но и учитывать специфику архитектуры исполняющей системы. Например, блочная оптимизация показала значительное улучшение производительности за счет уменьшения промахов кеша, а параллельное выполнение с помощью OpenMP позволило эффективно использовать многоядерные процессоры. Векторизация с AVX предоставила дальнейшие улучшения, позволяя обрабатывать несколько данных одной операцией, что еще более ускорило выполнение алгоритма. Однако наибольшее ускорение было достигнуто при использовании Intel MKL, что подчеркивает значимость оптимизированных библиотек в

достижении максимальной производительности.

Исследование показывает, что интенсивное использование ресурсов вычислительных систем через продуманные оптимизации является ключом к повышению эффективности и производительности в вычислительных задачах. Такие оптимизации не только улучшают время выполнения программ, но и могут значительно сократить потребление энергии, что критически важно для современных вычислительных центров и приложений.

Литература

1. Стратегии эффективной оптимизации алгоритмов // Zeba Academy. URL: open.zeba.academy/optimizatsiya-algoritma/ (дата обращения: 30.03.2024).

2. Функции OpenMP // Microsoft. URL: learn.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-functions?view=msvc-170 (дата обращения: 30.03.2024).

3. Developer Reference for Intel® oneAPI Math Kernel Library - C // Intel. URL: intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-0/overview.html (дата обращения: 30.03.2024).

4. Кожин А.С., Недбайло Ю.А. Методы оптимизации времени доступа в общий кэш многоядерного микропроцессора // Вопросы радиоэлектроники. 2017. № 3. С. 27–32.

5. Benchmark // Google Source. URL: fuchsia.googlesource.com/third_party/benchmark/+/21f1eb3fe269ea43eba862bf6b699cde46587ade/README.md (дата обращения: 30.03.2024).

6. Юрушкин М.В., Семионов С.Г. Переразмещение матриц к блочному виду с минимизацией использования дополнительной памяти // Известия вузов. Северо-Кавказский регион. Серия: Технические науки. 2017. №3 (195). С. 81-88.

7. Аль-Саиди А.А.,Темкин И.О., Алтай В.И., Алмунтафеки А.Ф., Мохмедхуссин А.Н. Повышение эффективности алгоритма Дейкстры с помощью технологий параллельных вычислений с библиотекой OpenMP // Инженерный вестник Дона, 2023 №8. URL: ivdon.ru/ru/magazine/archive/n8y2023/8595.

8. Егунов В.А., Андреев А.Е. Векторизация алгоритмов выполнения собственного и сингулярного разложений матриц с использованием преобразования Хаусхолдера // Прикаспийский журнал: управление и высокие технологии. 2020. №2 (50). С. 71-85.

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

10. Медакин П.О., Никулин Р.Н., Авдеюк О.А., Королева И.Ю., Павлова Е.С., Лемешкина И.Г. Векторизация и распараллеливание метода «частица-частица» // Инженерный вестник Дона, 2021, №1. URL: ivdon.ru/uploads/article/pdf/IVD_50__1_Medakin.pdf_57963b288b.pdf.

References

1. Strategii e`ffektivnoj optimizacii algoritmov [Strategies for effective optimization of algorithms]. Zeba Academy. URL: open.zeba.academy/optimizatsiya-algoritma/.

2. OpenMP functions. Microsoft. URL: learn.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-functions?view=msvc-170.

3. Developer Reference for Intel® oneAPI Math Kernel Library – C. Intel URL: intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-0/overview.html.

4. Kozhin A.S., Nedbajlo Yu.A. Voprosy` radioe`lektroniki. 2017. №3. pp.

27–32.

5. Benchmark. Google Source. URL: uchsia.googlesource.com/third_party/benchmark/+/21f1eb3fe269ea43eba862bf6b699cde46587ade/README.md.

6. Yurushkin M.V., Semionov S.G. Izvestiya vuzov. Severo-Kavkazskij region. Seriya: Texnicheskie nauki. 2017. №3 (195). pp. 81-88.

7. Al`-Saidi A.A.,Temkin I.O., Altaj V.I., Almuntafeki A.F., Moxmedxussin A.N. Inzhenernyj vestnik Dona, 2023, №8. URL: ivdon.ru/ru/magazine/archive/n8y2023/8595.

8. Egunov V.A., Andreev A.E. Prikaspijskij zhurnal: upravlenie i vy`sokie texnologii. 2020. №2 (50). pp. 71-85.

9. Egunov V.A. Prikaspijskij zhurnal: upravlenie i vy`sokie texnologii. 2019. №1 (45). pp. 154-163.

10. Medakin P.O., Nikulin R.N., Avdeyuk O.A., Koroleva I.Yu., Pavlova E.S., Lemeshkina I.G. Inzhenernyj vestnik Dona, 2021, №1. URL: ivdon.ru/uploads/article/pdf/IVD_50__1_Medakin.pdf_57963b288b.pdf.

Дата поступления: 11.03.2024

Дата публикации: 19.04.2024