Отчет по заданию
По курсу "Введение в суперкомпьютерное моделирование"

Шкляр Эрнест. БПМ-161

Апрель 2020 г.

Даны матрицы A и B. Написать программу для расчета матрицы $C = A^T B$. Числа считать комплесными (способ представления выбрать самостоятельно).

Последовательная версия

Оптимизация алгоритма

Реализация последовательного алгоритма приведена на *листинге 1*. Используется реализация комплесной арифметики стандартной библиотеки языка C. Явного транспонирования матрицы в данной реализации не производится, матрица **A** здесь - исходная. Из очевидных недостатков: обращение к матрицам **A** и **B** происходит не последовательно (во внутреннем цикле на каждой итерации происходит обращение к разным строкам матрицы), следовательно кэш используется неэффективно. Отсюда стремительный рост времени исполнения программы на сравнительно небольших размерах задачи.

```
int i, j, k;

for (i = 0; i < result->rows; ++i)
    for(j = 0; j < result->cols; ++j) {
        for (k = 0; k < a->rows; ++k) {
            result->data[i][j] += conj(a->data[k][i]) * b->data[k][j];
        }
    }
```

Листинг 1: Последовательный алгоритм

Поменяем цикл **ijk** на **ikj** и заметим, что **A[k][i]** не меняется во внутреннем цикле. Поэтому вынесем обращение к этому элементу на уровень выше. Однако, доступ к массиву A все ещё не является последовательным. Попробуем это исправить. Для этого выделим память под массив $conjT$, в котором будем хранить строку матрицы A и добавим ещё один цикл, в котором будем в нужном порядке записывать сопряженные элементы A. Эта операция потребует меньше времени, чем явное транспонирование матрицы, но даст прирост производительности в основном цикле за счет повышения эффективности работы с кэшем (*листинг 2*).

```
int i, j, k, ii;
double complex *conjT = (double complex *) emalloc(a->rows * sizeof(double
    complex));

for (i = 0; i < result->rows; ++i)
    for (k = 0; k < a->rows; ++k) {
        for (ii = 0; ii < result->rows; ++ii)
            conjT[ii] = conj(a->data[k][ii]);
        for (j = 0; j < result->cols; ++j) {
            result->data[i][j] += conjT[i] * b->data[k][j];
        }
    }
```

Листинг 2: Усовершенствованный алгоритм

На *рис.1* видно, что время исполнения для усовершенствованной версии алгоритма растет медленнее с увеличением размера задачи, чем для "наивного"алгоритма. Самыми дорогими являются операции чтения и записи в память, и величина времени исполнения главным образом зависит главным от пропукной способности контроллера памяти. Резкие "скачки"на графиках объясняются использованием уровней кэш-памяти процессора с разной пропускной способностью.

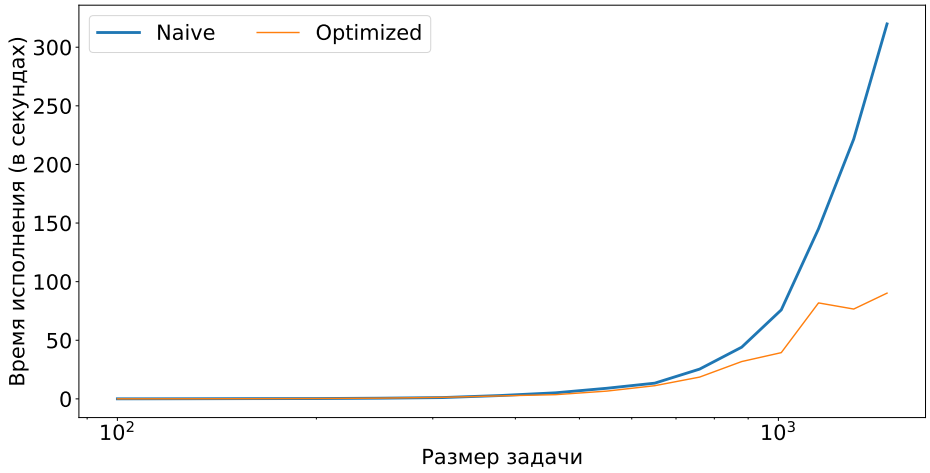

Рис. 1: Сравнение последовательных алгоритмов

Для тестов используются квадратные матрицы, заполненные случайными комплексными числами. Корректность всех алгоритмов была проверена на тестах из папки *test*.

Оптимизация компиляции GCC

Сравним время работы и производительность программы при компиляции с различными флагами оптимизации (*рис.2, рис.3*). Автовекторизация, задействованная на уровне оптимизации **-O3** с флагом **-mtune=native** показала самый хороший результат (*рис.3*).

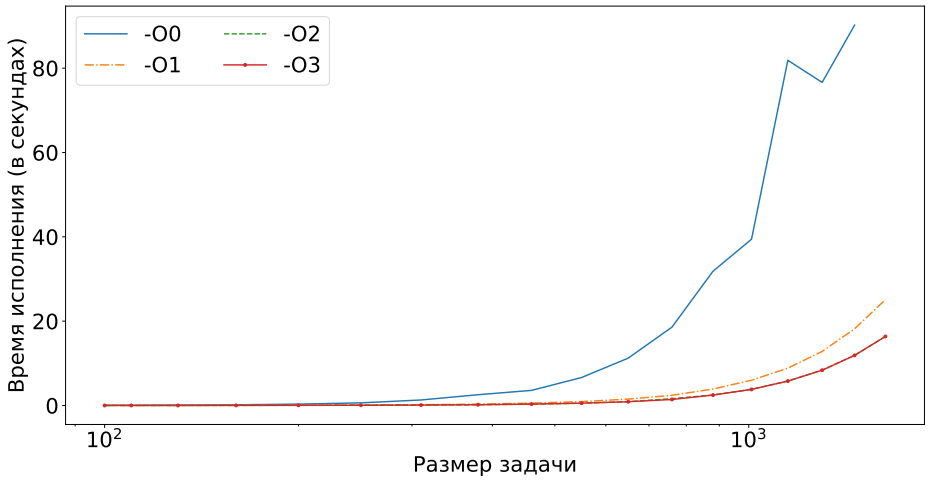

Рис. 2: Время работы последовательного алгоритма

Автоматическая векторизация выполнялась с помошью флагов компилятора **--O3** и **-mtune=native**. Автоматическая векторизация выполнялась с помошью флагов компилятора **--O3** и **-mtune=native**. Основной цикл в обоих случаях векторизован успешно (*листинг 3*)
Основной цикл в обоих случаях векторизован успешно (листинг 3)

```
sequential.c:24:13: optimized: loop vectorized using 16 byte vectors
sequential.c:24:13: optimized:  loop versioned for vectorization because of
    possible aliasing
sequential.c:26:13: optimized: basic block part vectorized using 16 byte
    vectors
```

Листинг 3: Отчет о векторизации

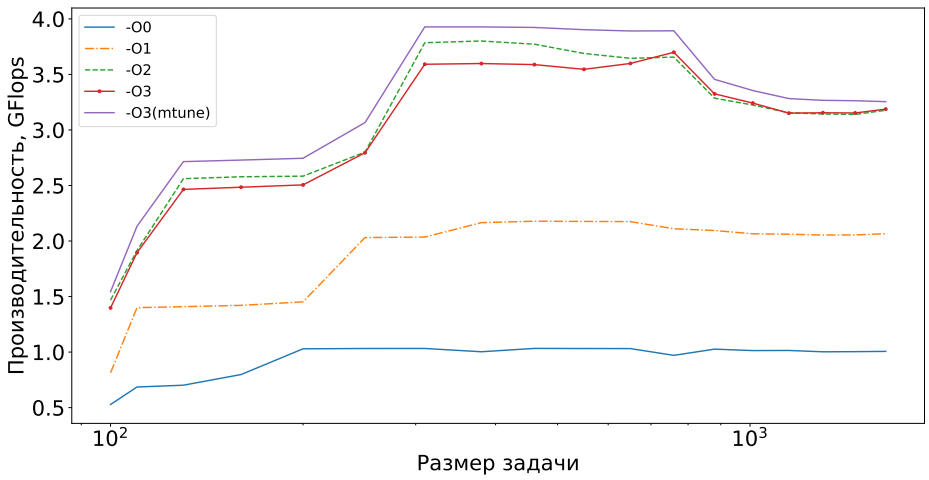

Рис. 3: Производительность последовательного алгоритма

Число операций с плавающей точкой определялось из следующих соображений:

1. Произведение: 8 операций [*glibc*].
2. Сумма: 4 операции.

Сравнение производительности всех версий с максимально возможной представлено на *рис.4*.

Версия	GFlops	% от max
-O0	0.94	11.2
-O1	1.87	22.2
-O2	3.07	36.6
-O3	3.02	35.9
-O3(mtune)	3.23	38.5

Рис. 4: Сравнение последовательных версий

Параллельная версия

OpenMP

Реализация OpenMP версии тривиальна, см. *листинг 4*.

```
int i, j, k, ii;
unsigned int row_size = a->rows;
double complex *conjTN = (double complex *) emalloc(num_threads*row_size*
    sizeof(double complex));
double complex *conjT = NULL;

omp_set_num_threads(num_threads);

#pragma omp parallel for private(i, j, k, ii, conjT)
for (i = 0; i < result->rows; ++i)
    for (k = 0; k < a->rows; ++k) {
        conjT = conjTN + row_size*omp_get_thread_num();
        for (ii = 0; ii < result->rows; ++ii)
            conjT[ii] = conj(a->data[k][ii]);
        for (j = 0; j < result->cols; ++j) {
            result->data[i][j] += conjT[i] * b->data[k][j];
        }
    }
free(conjTN);
```

Листинг 4: OpenMP

OpenMPI

Реализацию OpenMPI можно найти в *репозитории*.