

---

**Отчет по проделанной работе  
Архитектура и программное обеспечение  
высокопроизводительных вычислительных систем.**

**Студент**  
Кривохижин Даниил

**Преподаватель**  
к.ф.-м.н., ведущий научный сотрудник Института прикладной математики им  
М.В.Келдыша РАН, кафедра системного программирования, факультет  
вычислительной математики и кибернетики МГУ им. М.В. Ломоносова  
**Бахтин В.А.**

Астана, 2024

# Contents

<b>1 Постановка задачи 2mm</b>	<b>2</b>
1.1 Исходная задача и методы ее оптимизации . . . . .	2
1.1.1 Тонкие моменты при оптимизации (без распараллеливания) . . . . .	2
1.1.2 Первая идея . . . . .	3
1.2 Настоящая оптимизация и блочное умножение . . . . .	5
1.3 Небольшие промежуточные итоги . . . . .	8
<b>2 Распараллеливание с OPENMP</b>	<b>9</b>
2.1 Использование OpenMP для дальнейшего ускорения . . . . .	9
<b>3 Директива FOR</b>	<b>10</b>
3.1 Код реализованной программы . . . . .	10
3.2 Подробное описание реализации . . . . .	14
3.3 Результаты (Графики) . . . . .	15
3.3.1 EXTRALARGE DATASET . . . . .	15
3.3.2 LARGE DATASET . . . . .	17
3.3.3 MEDIUM DATASET . . . . .	19
3.3.4 SMALL DATASET . . . . .	21
3.3.5 MINI DATASET . . . . .	23
3.4 Анализ полученных результатов и заключение . . . . .	25
3.4.1 Общие закономерности . . . . .	25
3.4.2 Особенности поведения на разных размерах датасетов	25
3.4.3 Итоговые выводы . . . . .	25
<b>4 Директива Task</b>	<b>26</b>
4.1 Основная идея и структура кода . . . . .	26
4.2 Код реализации . . . . .	26
4.3 Пояснения к коду . . . . .	31
4.4 Заключение . . . . .	31
4.5 Результаты (Графики) . . . . .	32
4.5.1 EXTRALARGE DATASET . . . . .	32
4.5.2 LARGE DATASET . . . . .	34
4.5.3 MEDIUM DATASET . . . . .	36
4.5.4 SMALL DATASET . . . . .	38
4.5.5 MINI DATASET . . . . .	40
4.6 Анализ полученных результатов и заключение . . . . .	42
<b>5 Заключение</b>	<b>43</b>

# 1 Постановка задачи 2mm

## 1.1 Исходная задача и методы ее оптимизации

Задача 2mm (*two matrix multiplications*) предполагает следующее. Даны размеры:

$$N_i, \quad N_j, \quad N_k, \quad N_l,$$

а также четыре матрицы:

$$A \in \mathbb{R}^{N_i \times N_k}, \quad B \in \mathbb{R}^{N_k \times N_j}, \quad C \in \mathbb{R}^{N_j \times N_l}, \quad D \in \mathbb{R}^{N_i \times N_l},$$

и два скаляра  $\alpha$  и  $\beta$ .

Задача состоит в вычислении следующей композиции умножений:

$$\text{tmp} = \alpha \times A \times B,$$

$$D = \beta \times D + \text{tmp} \times C.$$

По сути, это можно записать *двумя* этапами:

1.  $\text{tmp}[i, j] = \sum_{k=0}^{N_k-1} \alpha A[i, k] B[k, j],$
2.  $D[i, j] = \beta D[i, j] + \sum_{k=0}^{N_j-1} \text{tmp}[i, k] C[k, j].$

### 1.1.1 Тонкие моменты при оптимизации (без распараллеливания)

Оптимизация этой задачи сводится к повышению:

- **Кэш-локальности**, то есть более эффективному использованию кэш-памяти (L1, L2, L3). При большом размере матриц перегрузка шины памяти становится узким местом.
- **Эффективного порядка обхода индексов**. Поскольку С хранит двумерные массивы в строковом формате, важно уметь переставлять циклы так, чтобы проходить элементы последовательно по строкам, особенно во внутренних циклах.
- **Частичной или полной блокировки**. Блочное умножение разбивает большие матрицы на подблоки, которые обрабатываются целиком, чтобы элементы оставались в кэше и многократно переиспользовались.
- **Минимизации арифметических накладных расходов**. Например, вынос умножения на константу  $\alpha$  или  $\beta$  за глубокие вложенные циклы, чтобы не множить эти операции на каждую итерацию.
- **Переупорядочивания операций**. Иногда помогают небольшие приёмы вроде сначала “обнулить” временную матрицу tmp отдельным циклом, а затем уже выполнить перемножение, чтобы избавиться от условных проверок или лишнего смешивания кода.

Пример «тонкого» момента — использование ключевого слова `restrict` в C99 (или аналогов в C++17), чтобы компилятор знал, что массивы  $A, B, C, D, \text{tmp}$  точно не пересекаются по адресам). Это может позволить компилятору агрессивно векторизовать умножения.

Другой типичный пример — *вынос умножения* на  $\beta$  из глубины цикла:

$$D[i, j] \leftarrow \beta D[i, j] + \dots$$

можно заменить на один проход по  $D$  (“ $D[i, j]* = \beta$ ”), а затем уже добавлять  $\text{tmp} \times C$  в результирующий  $D$ . Это, как правило, сокращает общее число операций при больших размерах.

## Сводка без распараллеливания

Таким образом, даже без использования OpenMP или других средств параллелизма, можно существенно улучшить скорость выполнения за счёт:

- **Перестановки циклов** для лучшего считывания строк.
- **Блокирования** — чтобы данные нескольких строк/столбцов лежали в кэше во время умножения.
- **Выделения независимых участков** (умножение на константу, обнуление матриц, и т. д.) в отдельные проходы.
- **Выноса** арифметических констант ( $\alpha, \beta$ ) во внешние циклы, локальные переменные для хранения промежуточных значений.

Все эти приёмы уменьшают объём «паразитных» действий и повышают пропускную способность вычислений, не затрагивая параллелизацию.

### 1.1.2 Первая идея

В исходном коде (листинг 3) матрицы  $A$ ,  $B$ ,  $C$ ,  $D$ ,  $\text{tmp}$  были объявлены как двумерные массивы

```
double A[Ni][Nk], double B[Nk][Nj], ...
```

и при обращении использовалось индексирование  $A[i][j]$ . Однако в языке С матрица  $A[N_i][N_k]$  фактически хранится в памяти в “row-major order”, то есть в одну строку, где элементы строки  $i$  идут подряд, а затем идут элементы следующей строки.

**Мотивация «хранить как одномерный массив»** Часто считают, что хранить матрицу как `double *A` (одномерный массив) и обращаться к элементам через

$$A[i \cdot (N_k) + j]$$

может:

- улучшить **локальность** (выстраивание доступа) — поскольку мы явно контролируем формулу вычисления адреса;
- упростить **векторизацию** и другие оптимизации (компилятор видит «плоский» массив).

Тем самым ожидалось, что при больших размерах ( $N_i, N_j, N_k, N_l$ ) может улучшиться кэширование и **ускориться** расчёт.

Практический результат На деле, при переписывании кода (листинг 2) с двумерных объявлений на «одномерные» указатели, выигрыш либо оказался *нулевым*, либо даже привёл к **замедлению**. С точки зрения производительности это может объясняться следующими причинами:

1. **Современные компиляторы C/C++** и так видят, что  $A[i][k]$  хранится непрерывно, и *автоматически* оптимизируют доступы.
2. **Alias-аналитика (aliasing)** — при переходе на указатели `double *A` компилятор иногда *не* может доказать, что массивы  $A$ ,  $B$ ,  $C$ ,  $D$ ,  $\text{tmp}$  не пересекаются по памяти (т. е. не имеют aliasing). Это препятствует агрессивной оптимизации и векторизации.
3. **Порядок циклов уже «правильный»** в исходном коде, и дополнительных улучшений кэш-локальности смена декларации  $\langle A[i][k] \rangle$  на  $\langle A[i \cdot N_k + k] \rangle$  не даёт.
4. **Дополнительные арифметические действия** (умножение  $i \times N_k + k$ ) могут быть несущественными, но при огромном числе итераций могут накопиться. Частично компиляторы их оптимизируют, но не всегда.

Листинг 1: 2mm\_og

```

1 //...
2
3
4 static void kernel_2mm(int ni, int nj, int nk, int nl,
5   double alpha,
6   double beta,
7   double tmp[ ni ][nj],
8   double A[ ni ][nk],
9   double B[ nk ][nj],
10  double C[ nj ][nl],
11  double D[ ni ][nl])
12 {
13   int i, j, k;
14   for (i = 0; i < ni; i++)
15     for (j = 0; j < nj; j++)
16     {
17       tmp[i][j] = 0.0;
18       for (k = 0; k < nk; ++k)
19         tmp[i][j] += alpha * A[i][k] * B[k][j];
20       }
21       for (i = 0; i < ni; i++)
22         for (j = 0; j < nl; j++)
23         {
24           D[i][j] *= beta;
25           for (k = 0; k < nj; ++k)
26             D[i][j] += tmp[i][k] * C[k][j];
27         }
28   }
29
30 //...

```

Листинг 2: 2mm\_new

```

1 //...
2
3
4 static void kernel_2mm(int ni, int nj, int nk, int nl,
5   double alpha,

```

```

6         double beta,
7         double *tmp,
8         double *A,
9         double *B,
10        double *C,
11        double *D)
12 {
13     int i, j, k;
14     // First matrix multiplication: tmp = alpha * A * B
15     for (i = 0; i < ni; i++)
16         for (j = 0; j < nj; j++) {
17             tmp[i * nj + j] = 0.0;
18             for (k = 0; k < nk; k++)
19                 tmp[i * nj + j] += alpha * A[i * nk + k] * B[k * nj + j];
20         }
21     // Second matrix multiplication: D = beta * D + tmp * C
22     for (i = 0; i < ni; i++)
23         for (j = 0; j < nl; j++) {
24             D[i * nl + j] *= beta;
25             for (k = 0; k < nj; k++)
26                 D[i * nl + j] += tmp[i * nj + k] * C[k * nl + j];
27         }
28 }
29
30 //...

```

Таблица 1: Сравнение времени выполнения программ 2mm\_og и 2mm\_new

Dataset	2mm_og (s)	2mm_new (s)
LARGE_DATASET	10.71	10.83
EXTRALARGE_DATASET	133.27	137.41

Таким образом, хотя переход на «одномерную» форму матриц выглядит логически правильным (особенно в классических учебных материалах), в современных условиях он не гарантирует ускорения. В некоторых случаях исходная реализация с  $A[i][k]$  оказывается столь же быстрой или быстрее, так как компилятор способен более эффективно анализировать и оптимизировать явные двумерные массивы.

## 1.2 Настоящая оптимизация и блочное умножение

**Мотивация для оптимизации** Исходная программа (листинг 2) содержит два умножения матриц:

$$\text{tmp} = \alpha \cdot A \times B, \quad D = \beta \cdot D + \text{tmp} \times C.$$

В наивном варианте всё реализовано вложенными циклами  $\text{for}$  по  $i, j, k$ . Для больших размеров матриц ( $N_i, N_j, N_k, N_l$  вплоть до нескольких тысяч) такой подход быстро упирается в пропускную способность памяти и неэффективно использует кэш.

Классическая идея оптимизации сводится к тому, чтобы:

- **Переставить порядок циклов**, чтобы во внутренних циклах идти *по строкам*, а не по столбцам (или наоборот, в зависимости от структуры).
- **Разделить** операции умножения на константу ( $\beta$ ) в отдельный проход, чтобы не выполнять это умножение в глубине вложенных циклов.
- **Применить блочное умножение**, когда матрицы разбиваются на подблоки (*tiles*), и каждая пара блоков обрабатывается целиком до перехода к следующему. Это значительно повышает кэш-локальность, поскольку нужные элементы много раз переспользуются из кэша.

**Первая стадия оптимизации (улучшение локальности)** Наиболее простой шаг, не требующий серьёзной переработки структуры кода, — **перестановка** циклов. Например, если исходный код имел вид (упрощённо):

Листинг 3: Условный фрагмент исходного кода

```

1 for (i = 0; i < ni; i++) {
2     for (j = 0; j < nj; j++) {
3         tmp[i][j] = 0.0;
4         for (k = 0; k < nk; k++) {
5             tmp[i][j] += alpha * A[i][k] * B[k][j];
6         }
7     }
8 }
```

мы меняем порядок так, чтобы внутри цикла всегда идти подряд по элементам  $B[k][j]$ . Обычно это означает:

- Сначала обнуляем  $\text{tmp}[i][..]$  построчно,
- затем идём по  $k$  снаружи, чтобы при фиксированном  $k$  обращаться к  $B[k][j]$  подряд ( $j$  — внутренний цикл).

Аналогично в вычислении  $D$  отделяем умножение на  $\beta$  в отдельный проход:

$\text{for } i \dots \text{for } j \dots D[i][j]* = \beta,$

после чего идёт сборка ( $\text{tmp}[i][k] * C[k][j]$ ).

**Блоchное умножение** Ключевой метод ускорения матричных операций — **блочное умножение**. Вместо того, чтобы пробегать все индексы  $i, j, k$  глобально, мы разбиваем матрицы на блоки размером  $BS \times BS$ . Ниже — псевдокод для первой части (формирования  $\text{tmp}$ ):

```

for i0 in 0..ni step BS let imax = min(i0 + BS, ni)
for j0 in 0..nj step BS let jmax = min(j0 + BS, nj)
обнулить блок  $\text{tmp}[i][j]$  для  $i \in [i0, i_{\max}], j \in [j0, j_{\max}]$ 
for k0 in 0..nk step BS let kmax = min(k0 + BS, nk)
```

перемножить фрагменты  $A[i, k] \times B[k, j]$ , где  $i \in [i0, i_{\max}], k \in [k0, k_{\max}], j \in [j0, j_{\max}]$

Такой «тройной вложенный цикл по блокам» локализует подблоки матриц  $A$ ,  $B$ ,  $\text{tmp}$  в кэше. Аналогичную схему используем для прибавления  $\text{tmp} * C$  к  $D$ , где  $C$  тоже разбивается на блоки. При больших размерах ( $ni, nj \approx 800, 1000$  и выше) такой подход сокращает количество кеш-промахов.

**Реализация блочного умножения в коде** В листинге 4 (фрагмент) показан пример, как именно устроены циклы:

Листинг 4: Фрагмент блочного умножения в kernel\_2mm

```

1 static void kernel_2mm(int ni, int nj, int nk, int nl,
2                         double alpha, double beta,
3                         double tmp[ni][nj],
4                         double A[ni][nk],
5                         double B[nk][nj],
6                         double C[nj][nl],
7                         double D[ni][nl])
8 {
9     // 1) tmp = alpha * A * B (с блокировкой)
10    for (int i0 = 0; i0 < ni; i0 += BS) {
11        int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
12        for (int j0 = 0; j0 < nj; j0 += BS) {
13            int jMax = (j0 + BS < nj) ? (j0 + BS) : nj;
14            // Обнулим tmp[i][j] в блоке
15            for (int i = i0; i < iMax; i++) {
16                for (int j = j0; j < jMax; j++) {
17                    tmp[i][j] = 0.0;
18                }
19            }
20            for (int k0 = 0; k0 < nk; k0 += BS) {
21                int kMax = (k0 + BS < nk) ? (k0 + BS) : nk;
22                for (int i = i0; i < iMax; i++) {
23                    for (int k = k0; k < kMax; k++) {
24                        double aik = alpha * A[i][k];
25                        for (int j = j0; j < jMax; j++) {
26                            tmp[i][j] += aik * B[k][j];
27                        }
28                    }
29                }
30            }
31        }
32    }
33    // 2) D = beta*D + tmp*C (аналогичный блочннй подход)
34    // ...
35 }
```

Видно, что вместо одного тройного цикла  $(i, j, k)$  мы имеем:

```

for i0.. step BS  for j0.. step BS  (обнуление блока tmp),
for k0.. step BS  (перемножение выбранных фрагментов A и B).
```

Затем аналогично для добавления  $\text{tmp} * \text{C}$  в  $\text{D}$ .

**Практический эффект** На больших размерах (например,  $\text{ni} = 1600$ ,  $\text{nj} = 1800$ ,  $\text{nk} = 2200$ ,  $\text{nl} = 2400$ ) блочное умножение обычно даёт весьма существенный выигрыш за счёт улучшенной кэш-локальности. Вместо того, чтобы постоянно «гонять» данные в/из оперативной памяти, алгоритм подгружает фрагменты ( $BS \times BS$ ) в кэш и многократно их переиспользует, прежде чем перейти к следующему фрагменту.

При этом размер блока BS (64, 128, 256, ...) подбирают экспериментально, учитывая:

- размер кэша (L1, L2, L3),
- число потоков (если есть распараллеливание),
- сами размеры матриц,
- характер доступа к памяти (row-major).

Слишком маленький BS порождает много блоков и накладные расходы на циклы, а слишком большой BS может выйти за пределы кэша и потерять преимущество локальности. На практике же и 64, 128 оказываются хорошим компромиссом для многих систем.

Таким образом, без использования методов распараллеливания, основываясь только на логике и знаниях основных принципов работы программы и компилятора, итоговые результаты заставляют радоваться и быть гордыми за такой прогресс.

Ниже представлены результаты множественных запусков программ (исходной и оптимизированной) на данном сервере ([astana@dvm.keldysh.ru](mailto:astana@dvm.keldysh.ru)) с различными компиляторами и ключами оптимизации:

Таблица 2: Сравнение времени выполнения программ 2mm\_og и 2mm\_new

gcc			clang		
Optimization Flag	2mm_og	2mm_new	Optimization Flag	2mm_og (s)	2mm_new (s)
gcc	166.572409	43.343247	clang	116.159570	35.363116
gcc -O3	41.159523	3.409586	clang -O3	33.463418	3.949689
gcc -O2	49.063967	8.427189	clang -O2	33.466195	3.957576
gcc -Ofast	15.547751	3.409762	clang -Ofast	34.340023	3.964073

nvc			icx		
Optimization Flag	2mm_og	2mm_new	Optimization Flag	2mm_og	2mm_new
nvc	93.378985	32.640610	icx	33.896209	4.594971
nvc -O3	34.017771	4.176726	icx -O3	33.588817	4.593839
nvc -O2	33.934390	4.175231	icx -O2	34.619320	4.595185
nvc -Ofast	34.968733	4.167971	icx -Ofast	35.080034	4.594470

### 1.3 Небольшие промежуточные итоги

#### 1. Общее сравнение

По результатам:

- **2mm\_new** стабильно быстрее на всех компиляторах и флагах оптимизации.
- Прирост производительности составляет от 2–3 раз до 7–8 раз, особенно при агрессивных оптимизациях ( $-O3$ ,  $-Ofast$ ).
- Компиляторы лучше оптимизируют блочное умножение, благодаря векторизации и разворачиванию циклов.

## 2. Различия между компиляторами

- **GCC**: хорошая общая оптимизация, значительный прирост при  $-Ofast$ .
- **Clang**: эффективен в авто-векторизации, хорошо работает с блочными циклами.
- **NVC**: оптимизирован для научных вычислений, но чувствителен к структуре кода.
- **ICX**: отлично справляется с векторизацией (SSE/AVX64), лучший прирост при  $-O3, -Ofast$ .

## 3. Выводы

- Блочная оптимизация (**2mm\_new**) существенно снижает время выполнения (до 8 раз быстрее исходной версии).
- $-Ofast$  максимально раскрывает потенциал блочного кода.
- При больших матрицах различия ещё заметнее из-за снижения кеш-промахов.

Таким образом, сочетание ручной оптимизации (блокировка, перестановка циклов) и флагов компиляции ( $-O2, -O3, -Ofast$ ) приносит максимальное ускорение.

# 2 Распараллеливание с OPENMP

## 2.1 Использование OpenMP для дальнейшего ускорения

После того, как была проведена оптимизация кода на уровне локальности данных (блочное умножение, перестановка циклов, вынесение отдельных операций), следующий шаг — **распараллеливание** с целью задействовать сразу несколько ядер центрального процессора.

**OpenMP** (Open Multi-Processing) — это популярная библиотека, позволяющая описывать параллельность в языке C/C++ (и Fortran) через набор *директив компилятора* (препроцессорных конструкций). Главные преимущества OpenMP:

- **Простота использования**: в большинстве случаев достаточно добавить несколько `#pragma omp ...` над циклом `for`, чтобы он автоматически распределился между потоками.
- **Поэтапный** (incremental) подход к параллелизации: мы можем постепенно добавлять директивы в уже работающий код (например, в функцию блочного умножения).
- **Поддержка со стороны разных компиляторов**: GCC, Clang, ICC (ICX), NVC и многие другие поддерживают OpenMP (обычно через флаг  $-fopenmp$  или аналогичный).
- **Гибкость**: помимо простой параллельной обработки циклов (`omp for`) OpenMP предоставляет механизмы для создания задач (`omp task`), управления синхронизацией (`omp barrier`), задание числа потоков и т. д.

Таким образом, вместо того чтобы вручную управлять потоками (`pthreads` или `std::thread` в C++), мы можем объявить параллельные участки и циклы, передавая планирование OpenMP.

**Подход к параллелизации** В общем виде алгоритма 2mm (или любого умножения матриц) всегда есть несколько независимых циклов/фаз, которые могут быть распределены между несколькими потоками. Например:

- Инициализация больших массивов (A, B, C, D).
- Формирование временной матрицы tmp.
- Обновление матрицы D.

OpenMP позволяет легко «рассечь» эти блоки на параллельные участки, если между ними нет сильных взаимозависимостей или если эти зависимости легко выразить через барьеры (или через более тонкие механизмы, вроде `task` и `depend`).

В рамках проделанной работы и формировании данного отчета были выбраны директивы **FOR** и **TASK** для распараллеливания установленной задачи. Подробнее про них:

## 3 Директива FOR

### 3.1 Код реализованной программы

В листинге 5 показан окончательный вариант кода, в котором используются:

- **Блоchное умножение** с шагом BS.
- **Параллелизация** циклов через директиву `#pragma omp for`.
- **Дополнительные параметры** вроде `collapse(2)`, `private(...)` и `nowait`.

Листинг 5: Реализация 2mm с OpenMP `for` и блочным умножением

```
1 #include "2mm.h"
2 #include <omp.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/time.h>
7
8 #ifndef BS
9 #define BS 64
10 #endif
11
12
13 /* -----
14 * 1) Параллельная инициализация массивов A, B, C, D
15 *   с помощью #pragma omp for
16 * -----
17 static void init_array(int ni, int nj, int nk, int nl,
18                       double *alpha,
19                       double *beta,
20                       double *A,
21                       double *B,
22                       double *C,
```

```

23 |                     double *D)
24 | {
25 |     int i, j;
26 |
27 |     *alpha = 1.5;
28 |     *beta = 1.2;
29 |
30 | #pragma omp parallel
31 | {
32 |     // Инициализируем A ( $ni * nk$ )
33 |     #pragma omp for nowait
34 |     for (i = 0; i < ni; i++)
35 |         for (j = 0; j < nk; j++)
36 |             A[i * nk + j] = (double)((i*j + 1) % ni) / ni;
37 |
38 |     // Инициализируем B ( $nk * nj$ )
39 |     #pragma omp for nowait
40 |     for (i = 0; i < nk; i++)
41 |         for (j = 0; j < nj; j++)
42 |             B[i * nj + j] = (double)(i * (j + 1) % nj) / nj;
43 |
44 |     // Инициализируем C ( $nj * nl$ )
45 |     #pragma omp for nowait
46 |     for (i = 0; i < nj; i++)
47 |         for (j = 0; j < nl; j++)
48 |             C[i * nl + j] = (double)((i*(j+3)+1) % nl) / nl;
49 |
50 |     // Инициализируем D ( $ni * nl$ )
51 |     #pragma omp for nowait
52 |     for (i = 0; i < ni; i++)
53 |         for (j = 0; j < nl; j++)
54 |             D[i * nl + j] = (double)((i*(j+2)) % nk) / nk;
55 | } // end parallel
56 }
57 */
58 * -----
59 * 2) kernel_2mm: с использованием блочного умножения
60 *      + #pragma omp for (+ collapse(2))
61 * -----
62 static void kernel_2mm(int ni, int nj, int nk, int nl,
63                     double alpha,
64                     double beta,
65                     double *tmp,
66                     double *A,
67                     double *B,
68                     double *C,
69                     double *D)
70 {
71     int i, j, k;
72
73     /* -----
74     * 2.1) Вычисление tmp = alpha * A * B

```

```

75      * - блочное умножение
76      * - распараллеливание внешних двух циклов
77      * -----
78 #pragma omp parallel
79 {
80     #pragma omp for private(i, j, k) collapse(2)
81     for (int i0 = 0; i0 < ni; i0 += BS)
82     {
83         for (int j0 = 0; j0 < nj; j0 += BS)
84         {
85             int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
86             int jMax = (j0 + BS < nj) ? (j0 + BS) : nj;
87
88             // Обнуление блока tmp[i][j]
89             for (i = i0; i < iMax; i++) {
90                 for (j = j0; j < jMax; j++) {
91                     tmp[i * nj + j] = 0.0;
92                 }
93             }
94             // Перебор блоков по k
95             for (int k0 = 0; k0 < nk; k0 += BS) {
96                 int kMax = (k0 + BS < nk) ? (k0 + BS) : nk;
97                 for (i = i0; i < iMax; i++) {
98                     for (k = k0; k < kMax; k++) {
99                         double aik = alpha * A[i * nk + k];
100                        for (j = j0; j < jMax; j++) {
101                            tmp[i * nj + j] += aik * B[k * nj + j];
102                        }
103                    }
104                }
105            } // end for k0
106        } // end for j0
107    } // end for i0
108 } // end parallel
109
110 /* -----
111  * 2.2) Умножение D на beta
112  * -----
113 #pragma omp parallel for private(i, j)
114 for (i = 0; i < ni; i++) {
115     for (j = 0; j < nl; j++) {
116         D[i * nl + j] *= beta;
117     }
118 }
119
120 /* -----
121  * 2.3) D += tmp * C (блочное умножение)
122  * -----
123 #pragma omp parallel
124 {
125     #pragma omp for private(i, j, k) collapse(2)
126     for (int i0 = 0; i0 < ni; i0 += BS)

```

```

127     {
128         for (int j0 = 0; j0 < nl; j0 += BS)
129         {
130             int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
131             int jMax = (j0 + BS < nl) ? (j0 + BS) : nl;
132             // перебираем k по блокам
133             for (int k0 = 0; k0 < nj; k0 += BS) {
134                 int kMax = (k0 + BS < nj) ? (k0 + BS) : nj;
135                 for (i = i0; i < iMax; i++) {
136                     for (k = k0; k < kMax; k++) {
137                         double tik = tmp[i * nj + k];
138                         for (j = j0; j < jMax; j++) {
139                             D[i * nl + j] += tik * C[k * nl + j];
140                         }
141                     }
142                 }
143             }
144         }
145     }
146 } // end parallel
147 }

148
149 int main(int argc, char** argv)
150 {
151     int ni = NI;
152     int nj = NJ;
153     int nk = NK;
154     int nl = NL;

155
156     double alpha, beta;

157
158     // Выделяем память под все массивы (в одномерном виде)
159     double *tmp = (double*) malloc(ni * nj * sizeof(double));
160     double *A = (double*) malloc(ni * nk * sizeof(double));
161     double *B = (double*) malloc(nk * nj * sizeof(double));
162     double *C = (double*) malloc(nj * nl * sizeof(double));
163     double *D = (double*) malloc(ni * nl * sizeof(double));

164
165     // 1) Инициализация
166     init_array(ni, nj, nk, nl, &alpha, &beta, A, B, C, D);

167
168     // 2) Замер времени на основном ядре
169     bench_timer_start();
170     kernel_2mm(ni, nj, nk, nl, alpha, beta, tmp, A, B, C, D);
171     bench_timer_stop();
172     bench_timer_print();

173
174     free(tmp); free(A); free(B); free(C); free(D);
175     return 0;
176 }

```

## 3.2 Подробное описание реализации

1. Директива `#pragma omp parallel` создаёт пул потоков (равный `OMP_NUM_THREADS` или количеству ядер по умолчанию).
2. Инициализация массивов (`init_array`):
  - Каждая `#pragma omp for nowait` означает, что после завершения данного цикла не ставится барьер — потоки могут сразу продолжать к следующему `#pragma omp for`.
  - Таким образом четыре цикла (по A, B, C, D) могут потенциально выполняться частично в параллели, если железо позволяет.
3. Блоочное умножение (`kernel_2mm`):
  - (a) Проход по блокам  $i_0, j_0, k_0$  с шагом BS (64).
  - (b) `#pragma omp for private(i,j,k) collapse(2)` — распараллеливает два внешних цикла ( $i_0$  и  $j_0$ ). То есть каждое сочетание  $(i_0, j_0)$  становится единицей распределения между потоками.
  - (c) Внутри каждого блока обнуляем нужный участок `tmp`, затем проходимся по подблокам  $k_0$ .
4. Умножение D на  $\beta$  вынесено в отдельный цикл, который распараллелен директивой `#pragma omp parallel for`.
5. Вычисление  $D + \text{tmp} \times C$  тоже блоочное, с аналогичной схемой цикла:
  - $i_0, j_0$  — внешние циклы, распараллеленные через `collapse(2)`.
  - $k_0$  во внутреннем цикле обеспечивает блокировку по «k».

## Преимущества и замечания

- `collapse(2)` позволяет одновременно «раздать» двумерное пространство  $(i_0, j_0)$  по потокам. Без `collapse` распараллеливался бы только один цикл (по  $i_0$ ), а цикл по  $j_0$  шёл последовательно в каждом потоке.
- `private(i,j,k)` в `#pragma omp for` предотвращает гонки за счёт того, что каждая нить имеет собственные скалярные переменные (счётчики).
- Размер блока BS = 64 может быть изменён (например, 64, 128, 256) в зависимости от размеров матриц и архитектуры: - Если блок слишком маленький, будет много итераций и большая нагрузка на планировщик. - Если блок слишком большой, можем потерять кэш-локальность.

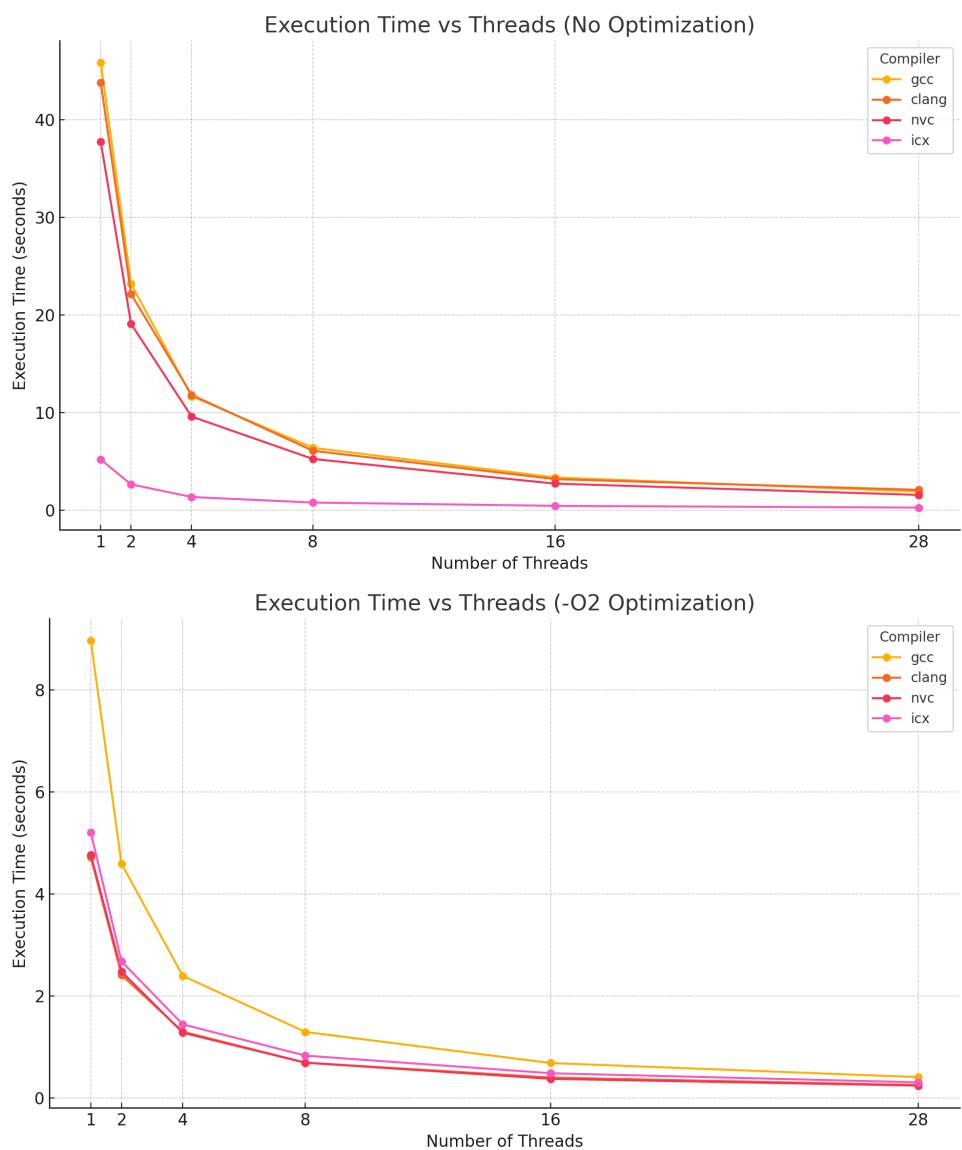
Таким образом, используем OpenMP `for` для того, чтобы:

1. Задействовать все доступные потоки в каждом крупном цикле (инициализация, вычисление `tmp`, умножение D).
2. Легко склеить «двумерное» пространство  $(i_0, j_0)$  в единый пул итераций через `collapse(2)`.
3. Не писать вручную логику создания и завершения потоков — всё делается автоматизированно.

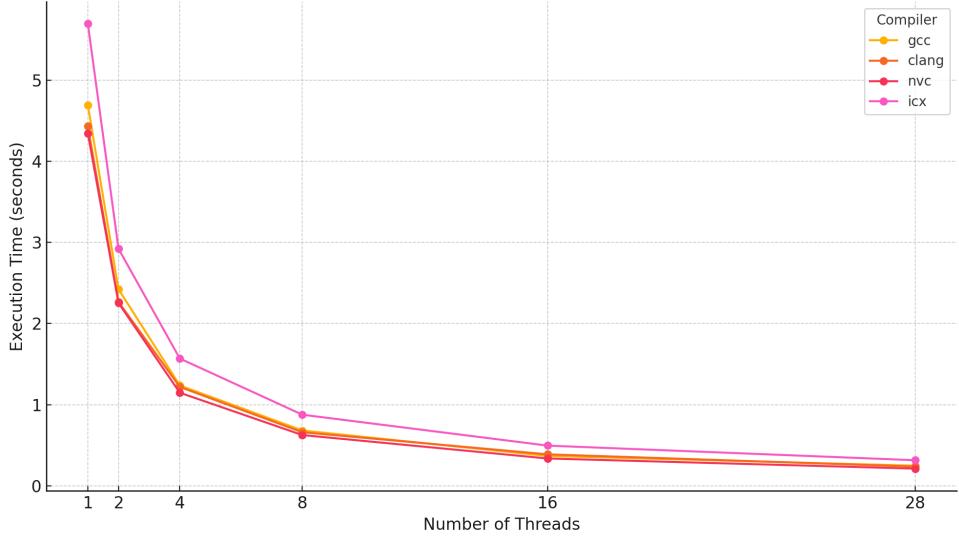
### 3.3 Результаты (Графики)

#### 3.3.1 EXTRALARGE DATASET

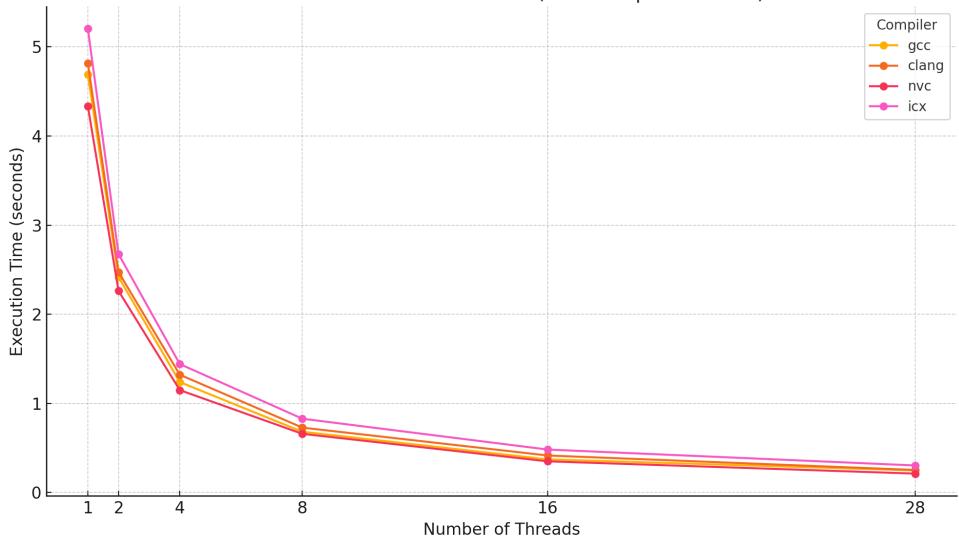
:



Execution Time vs Threads (-O3 Optimization)

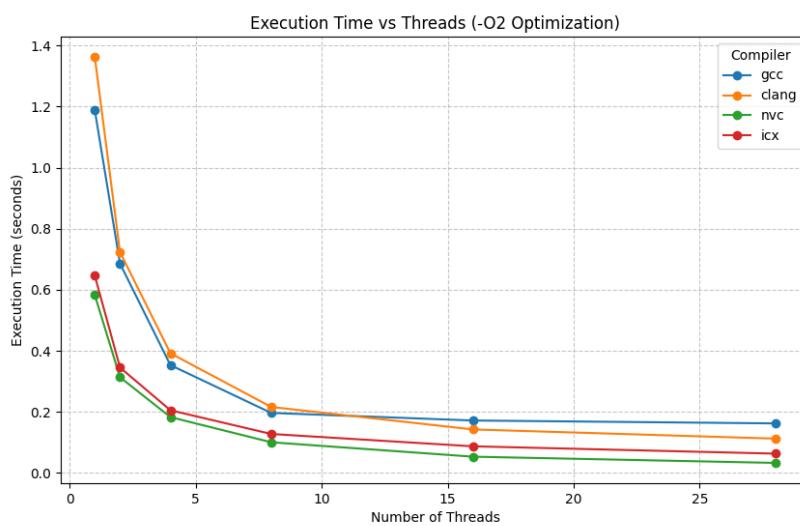
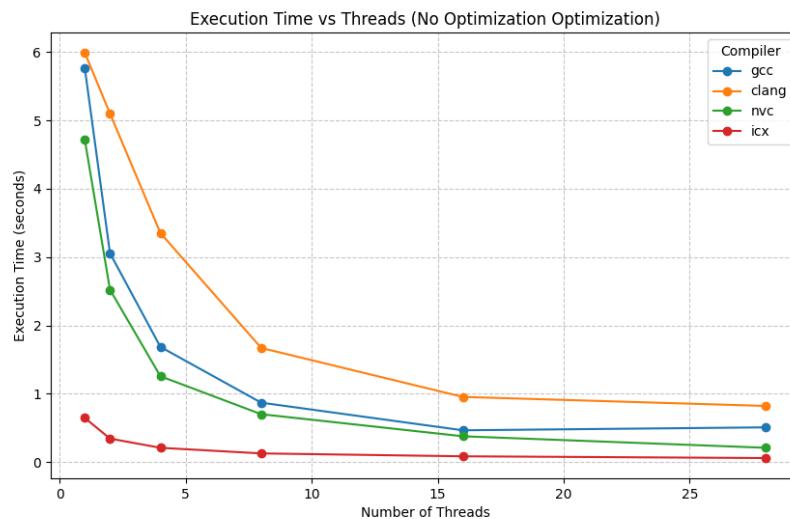


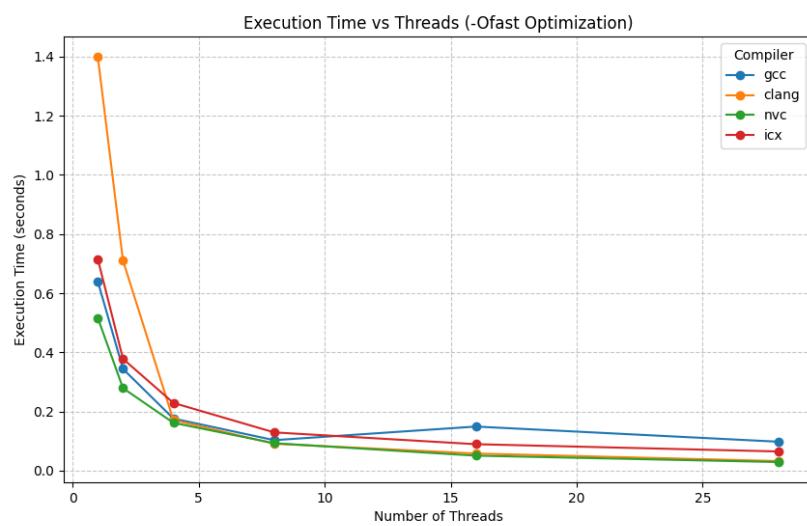
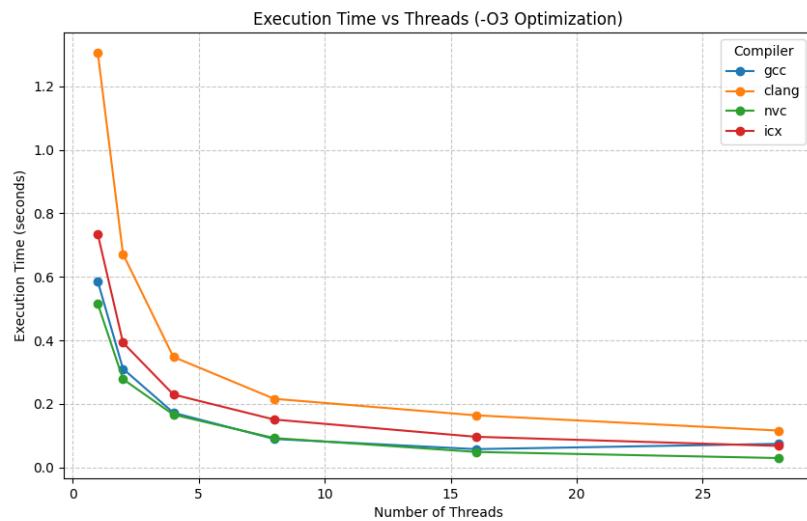
Execution Time vs Threads (-Ofast Optimization)



### 3.3.2 LARGE DATASET

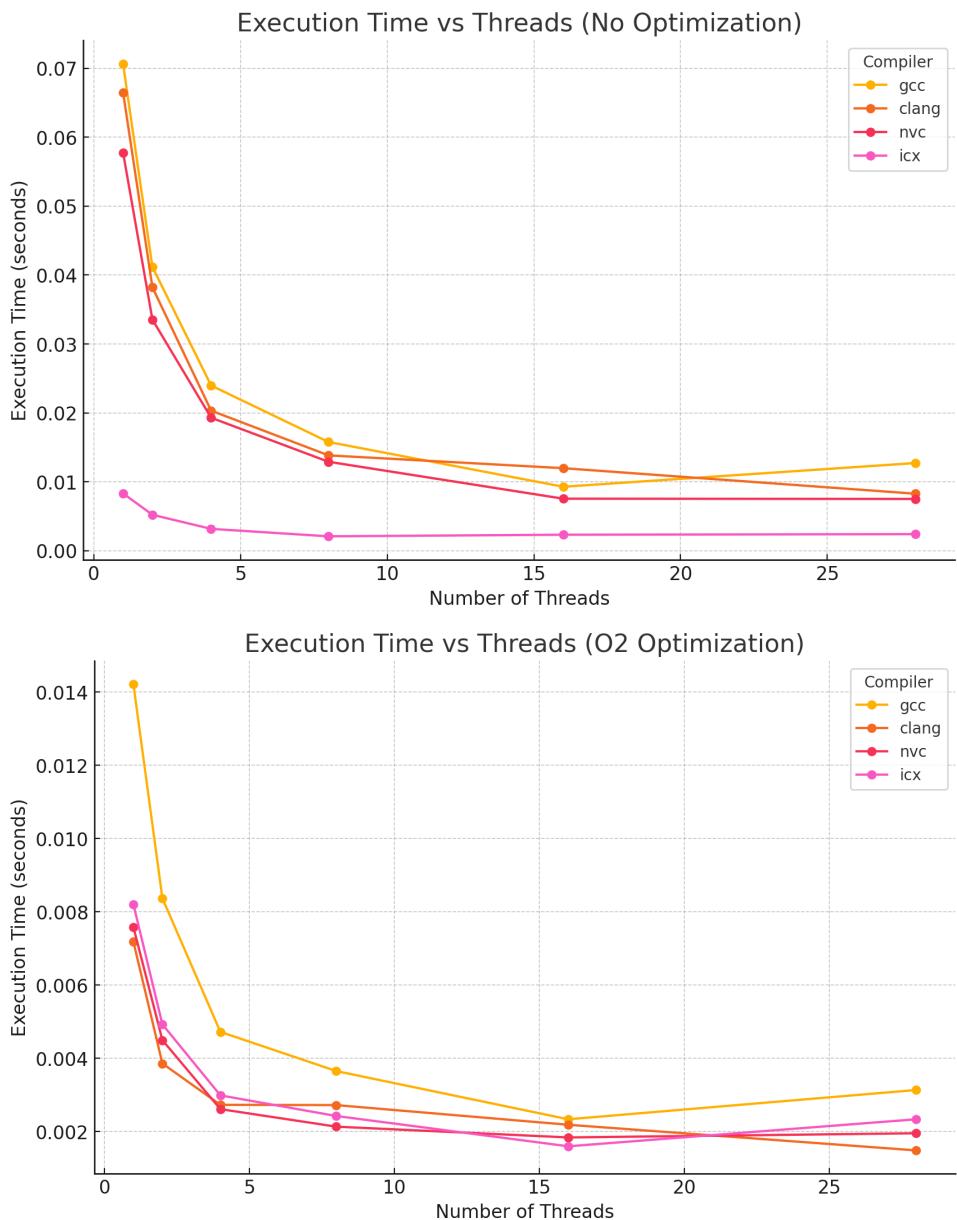
:

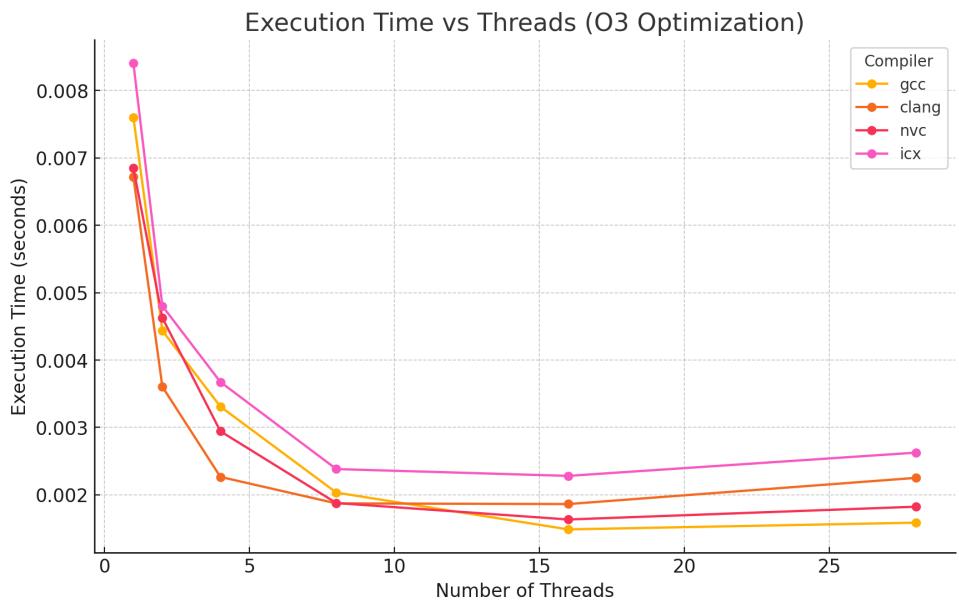




### 3.3.3 MEDIUM DATASET

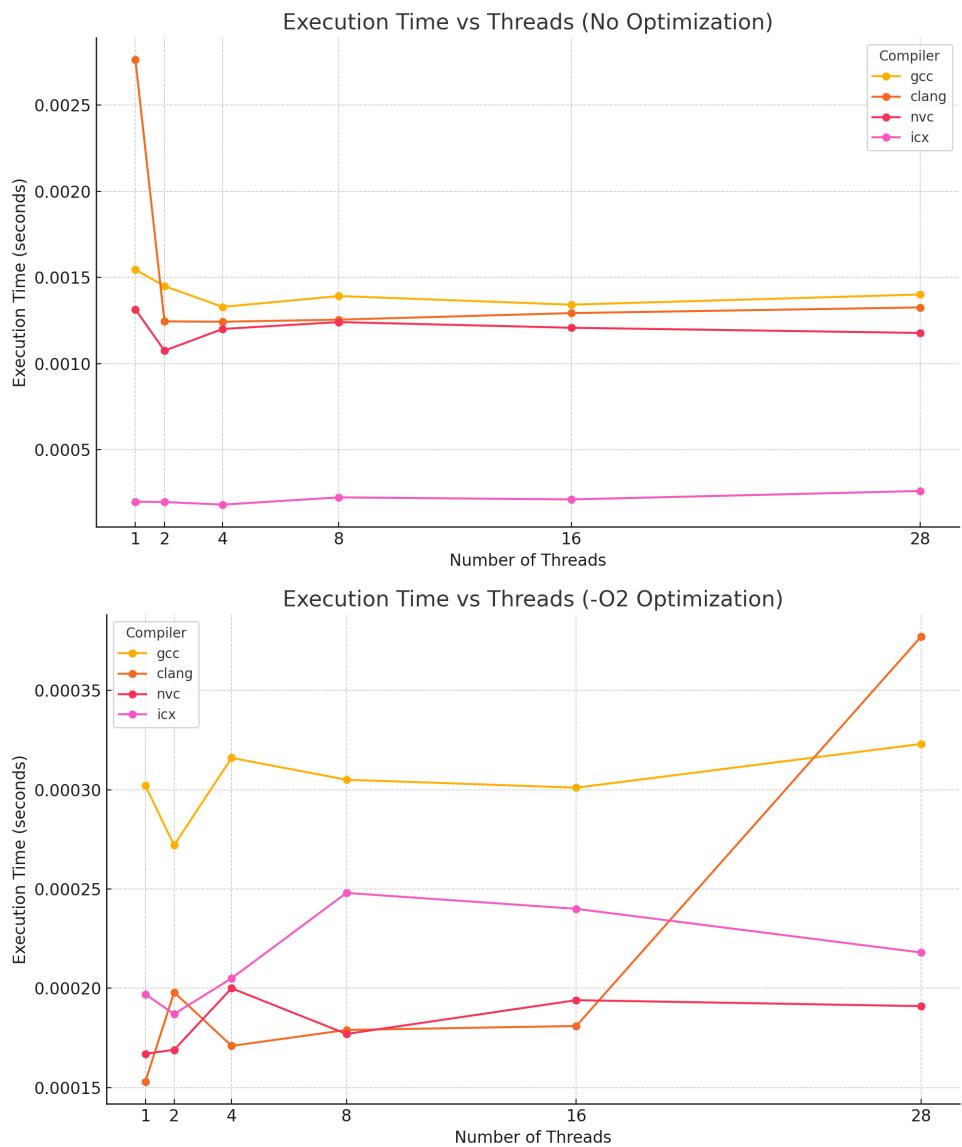
:

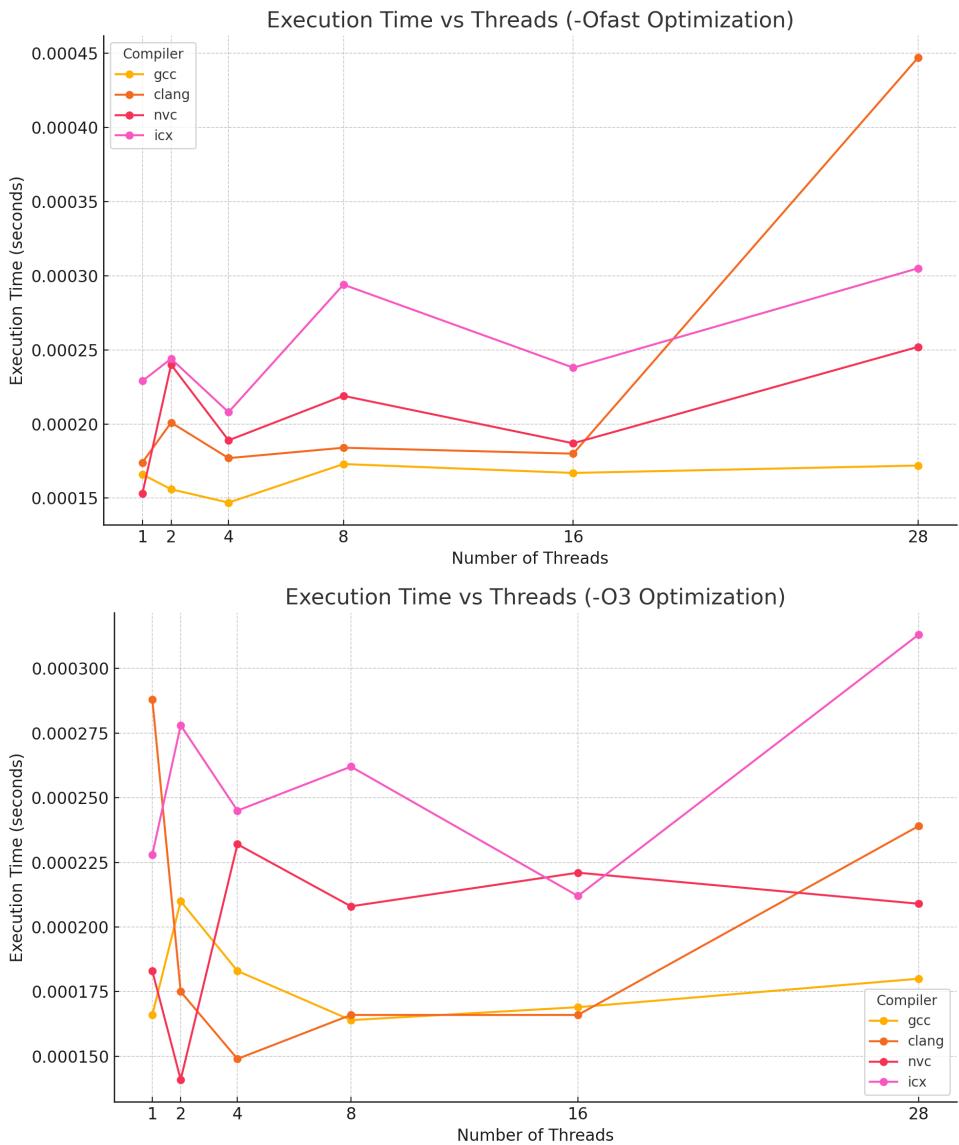




### 3.3.4 SMALL DATASET

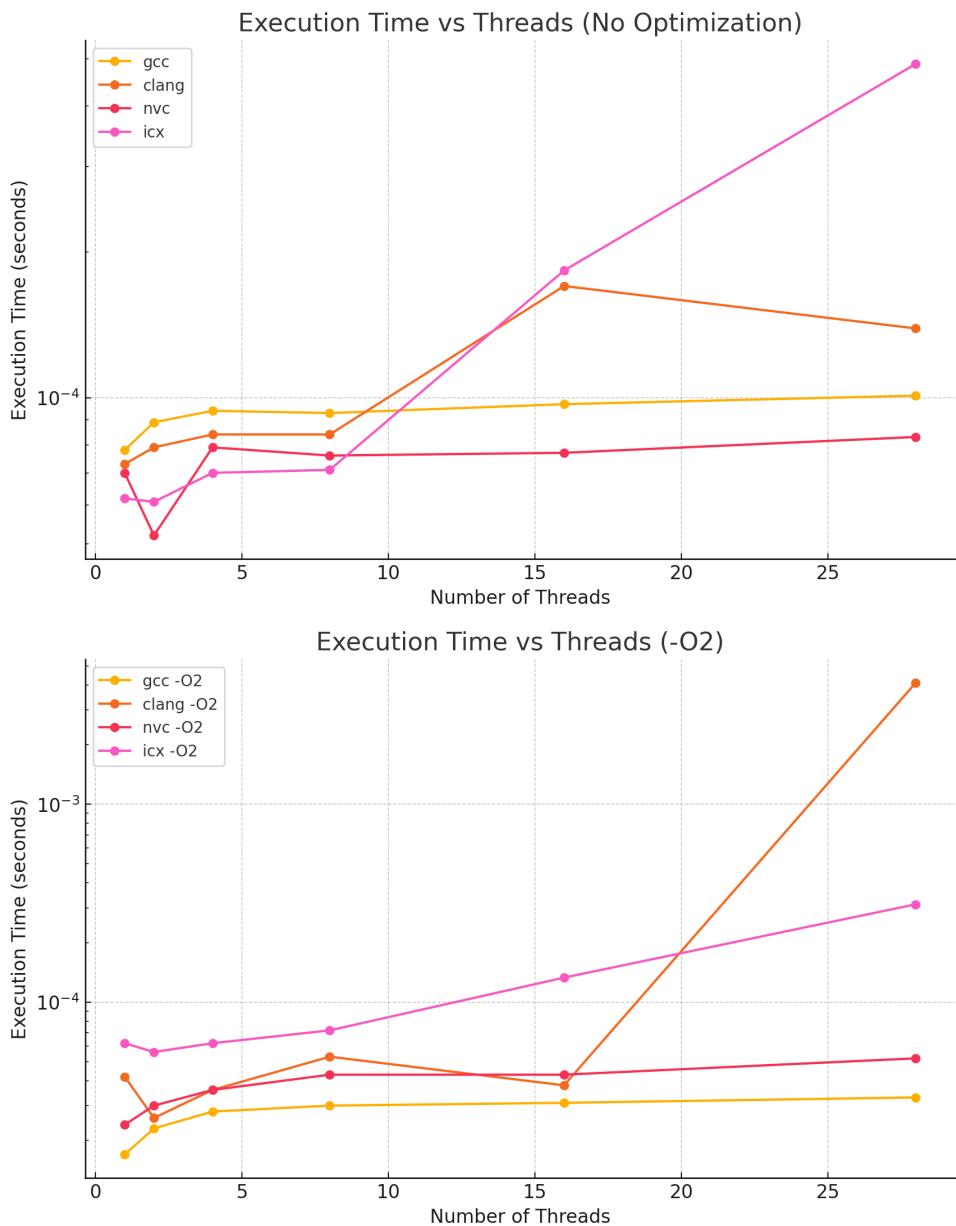
:



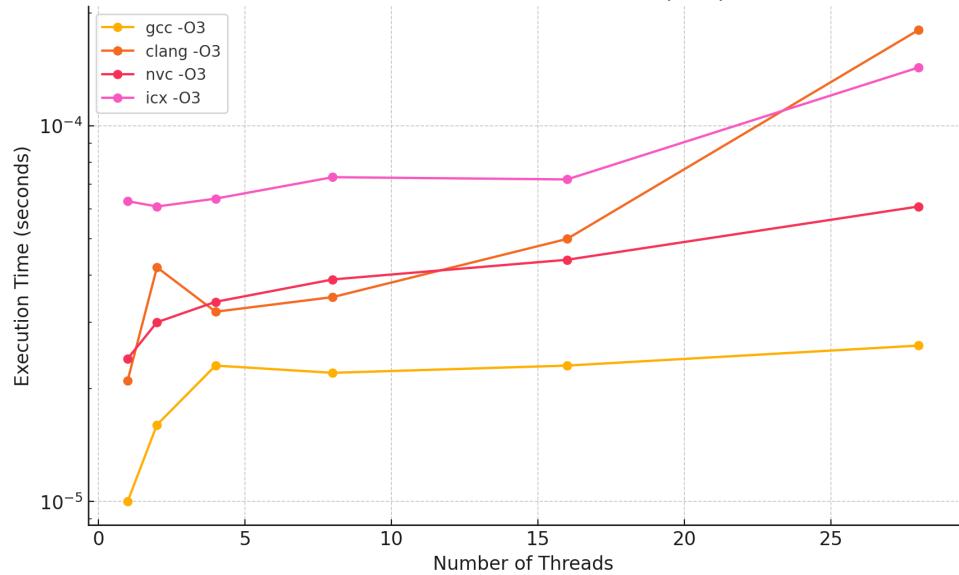


### 3.3.5 MINI DATASET

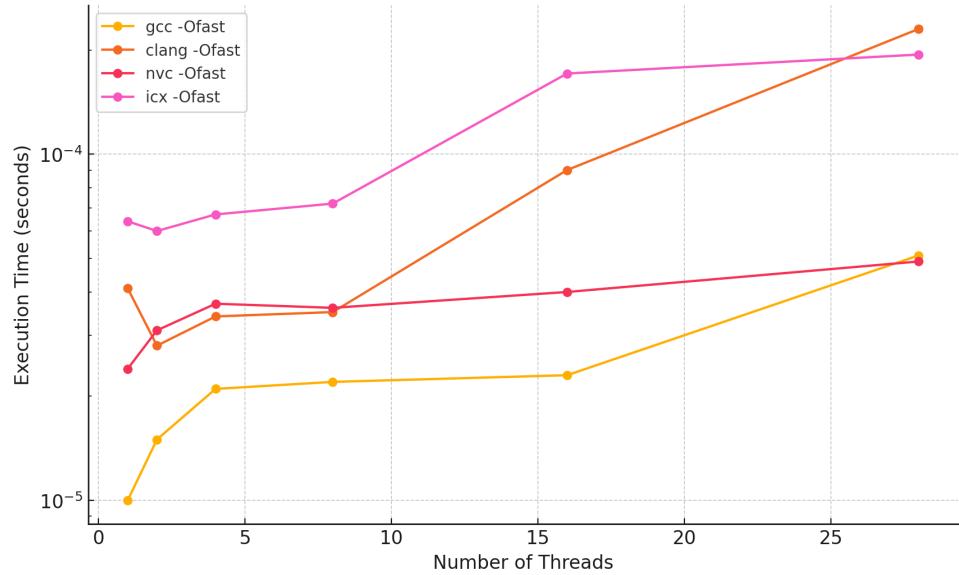
:



Execution Time vs Threads (-O3)



Execution Time vs Threads (-Ofast)



## 3.4 Анализ полученных результатов и заключение

### 3.4.1 Общие закономерности

1. **Снижение времени при увеличении числа потоков:** Максимальное ускорение достигается при переходе с 1 на 2–4 потока. После 8–16 потоков ускорение замедляется из-за накладных расходов на синхронизацию и ограничений пропускной способности памяти.
2. **Влияние оптимизации:**
  - Без оптимизации (`No Optimization`) время выполнения значительно выше, но многопоточность частично компенсирует недостатки.
  - С флагами оптимизации (`-O2`, `-O3`, `-Ofast`) время выполнения снижается, а относительный прирост от параллелизма уменьшается.
3. **Различия между компиляторами:**
  - **ICX** и **NVC** показывают лучшие результаты при большом числе потоков.
  - **GCC** и **Clang** имеют схожее поведение, но отличаются в абсолютных временах в зависимости от задачи.

### 3.4.2 Особенности поведения на разных размерах датасетов

1. **Малые датасеты (Small):**
  - Времена выполнения минимальны, иногда наблюдаются шумы из-за накладных расходов параллелизма.
  - Параллелизация может быть неэффективной из-за короткого времени работы программы.
2. **Средние датасеты (Medium):**
  - Наблюдается классическое снижение времени при увеличении числа потоков.
  - На 16–28 потоках время выравнивается, достигая минимальных значений.
3. **Крупные датасеты (Large/Extralarge):**
  - Время выполнения значительно выше, параллелизация наиболее эффективна.
  - Максимальное ускорение достигается на 16–28 потоках, однако ограничение памяти становится заметным.

### 3.4.3 Итоговые выводы

1. **Параллелизация эффективна:** Наиболее значительное ускорение наблюдается при увеличении потоков до 8–16. После этого прирост снижается.
2. **Флаги оптимизации (`-O2`, `-O3`, `-Ofast`):** Сильно снижают базовое время выполнения и повышают эффективность масштабирования.
3. **Компиляторы:** **ICX** и **NVC** показывают наилучшие результаты на больших датасетах, тогда как **GCC** и **Clang** имеют конкурентные результаты на меньших задачах.

Таким образом, многопоточность в сочетании с оптимизацией компилятора позволяет достичь значительного ускорения, особенно на крупных задачах.

## 4 Директива Task

В данной версии параллельности мы используем механизм **задач** (*tasks*) в OpenMP, когда каждый существенный фрагмент вычислений оформлен как отдельная задача, а рантайм OpenMP сам распределяет эти задачи между потоками.

### 4.1 Основная идея и структура кода

- **Блочное разбиение:** как и прежде, большие циклы по  $i, j, k$  разбиваются на подблоки  $BS \times BS$ .
- **pragma omp parallel + #pragma omp single:** - `parallel` создаёт пул потоков; - `single` говорит, что только один поток «генерирует» задачи (`omp task`), а остальные исполняют их.
- **Создание задач (task):** - На каждый блок  $[i_0..i_{\max}), [j_0..j_{\max})$  создаётся задача, которая обрабатывает соответствующий участок матрицы. - Используем `firstprivate(i0, j0, iMax, jMax)` — чтобы каждый блок знал свои «локальные» границы.
- **Синхронизация:** - После формирования группы задач ставим `#pragma omp taskwait`, чтобы дождаться окончания перед переходом к следующему этапу (или выходом из параллельной области).
- **Три основные фазы:** 1) Инициализация A, B, C, D через задачи, 2) Вычисление  $\text{tmp} = \text{alpha} * \text{A} * \text{B}$ , 3) Умножение D \*= beta и добавление tmp\*C.
- **Отсутствие collapse:** - Вместо «объединения» вложенных циклов в `for`, мы вручную генерируем задачи *по подблокам* и отдаём их в пул потоков.
- **Гибкость:** - Система задач хорошо подходит, когда разные части кода (например, инициализация блоков) не зависят друг от друга *напрямую*. - Мы можем «раскрыть» большой объём итераций в виде набора задач (по блокам  $BS \times BS$ ) и позволить планировщику OpenMP распределить их.

### 4.2 Код реализации

Ниже в листинге 6 приведён полный код, где все директивы `pragma omp` относятся к механизму задач. Заметим, что комментарии внутри минимальны; все приёмы описаны выше.

Листинг 6: Реализация 2mm с OpenMP `task` и блочным подходом

```
1 #include "2mm.h"
2 #include <omp.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/time.h>
7
8 #ifndef BS
9 #define BS 256
10 #endif
11
12 double bench_t_start, bench_t_end;
```

```

14 static double rtclock()
15 {
16     struct timeval Tp;
17     int stat = gettimeofday(&Tp, NULL);
18     if (stat != 0)
19         printf("Error return from gettimeofday: %d\n", stat);
20     return (Tp.tv_sec + Tp.tv_usec * 1.0e-6);
21 }
22
23 void bench_timer_start() { bench_t_start = rtclock(); }
24 void bench_timer_stop() { bench_t_end = rtclock(); }
25 void bench_timer_print() { printf("Time in seconds = %0.6lf\n",
26                                bench_t_end - bench_t_start); }
27
28 static void init_array(int ni, int nj, int nk, int nl,
29                       double *alpha, double *beta,
30                       double *A, double *B, double *C, double *D)
31 {
32     #pragma omp parallel
33     {
34         #pragma omp single
35         {
36             *alpha = 1.5;
37             *beta = 1.2;
38
39             for (int i0 = 0; i0 < ni; i0 += BS) {
40                 int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
41                 for (int j0 = 0; j0 < nk; j0 += BS) {
42                     int jMax = (j0 + BS < nk) ? (j0 + BS) : nk;
43
44                     #pragma omp task firstprivate(i0, j0, iMax, jMax)
45                     {
46                         for (int i = i0; i < iMax; i++) {
47                             for (int j = j0; j < jMax; j++) {
48                                 A[i * nk + j] = (double)((i*j + 1) % ni
49                                              ) / ni;
50                             }
51                         }
52                     }
53
54                 for (int i0 = 0; i0 < nk; i0 += BS) {
55                     int iMax = (i0 + BS < nk) ? (i0 + BS) : nk;
56                     for (int j0 = 0; j0 < nj; j0 += BS) {
57                         int jMax = (j0 + BS < nj) ? (j0 + BS) : nj;
58
59                         #pragma omp task firstprivate(i0, j0, iMax, jMax)
60                         {
61                             for (int i = i0; i < iMax; i++) {
62                                 for (int j = j0; j < jMax; j++) {
63                                     B[i * nj + j] = (double)(i*(j+1) % nj)

```

```

64                         / nj;
65                     }
66                 }
67             }
68         }
69
70     for (int i0 = 0; i0 < nj; i0 += BS) {
71         int iMax = (i0 + BS < nj) ? (i0 + BS) : nj;
72         for (int j0 = 0; j0 < nl; j0 += BS) {
73             int jMax = (j0 + BS < nl) ? (j0 + BS) : nl;
74
75             #pragma omp task firstprivate(i0, j0, iMax, jMax)
76             {
77                 for (int i = i0; i < iMax; i++) {
78                     for (int j = j0; j < jMax; j++) {
79                         C[i * nl + j] = (double)((i*(j+3) + 1)
80                                         % nl) / nl;
81                     }
82                 }
83             }
84         }
85
86     for (int i0 = 0; i0 < ni; i0 += BS) {
87         int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
88         for (int j0 = 0; j0 < nl; j0 += BS) {
89             int jMax = (j0 + BS < nl) ? (j0 + BS) : nl;
90
91             #pragma omp task firstprivate(i0, j0, iMax, jMax)
92             {
93                 for (int i = i0; i < iMax; i++) {
94                     for (int j = j0; j < jMax; j++) {
95                         D[i * nl + j] = (double)((i*(j+2)) % nk
96                                         ) / nk;
97                     }
98                 }
99             }
100        }
101    } // end single
102    #pragma omp taskwait
103 } // end parallel
104 }
105
106 static void kernel_2mm(int ni, int nj, int nk, int nl,
107                         double alpha, double beta,
108                         double *tmp, double *A, double *B, double *C,
109                         double *D)
110 {
111     #pragma omp parallel

```

```

112     #pragma omp single
113     {
114         // 1) tmp = alpha*A*B
115         for (int i0 = 0; i0 < ni; i0 += BS) {
116             int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
117             for (int j0 = 0; j0 < nj; j0 += BS) {
118                 int jMax = (j0 + BS < nj) ? (j0 + BS) : nj;
119
120                 #pragma omp task firstprivate(i0, j0, iMax, jMax)
121                 {
122                     for (int i = i0; i < iMax; i++) {
123                         for (int j = j0; j < jMax; j++) {
124                             tmp[i * nj + j] = 0.0;
125                         }
126                     }
127                     for (int k = 0; k < nk; k++) {
128                         for (int i = i0; i < iMax; i++) {
129                             double aik = alpha * A[i * nk + k];
130                             for (int j = j0; j < jMax; j++) {
131                                 tmp[i * nj + j] += aik * B[k * nj +
132                                     j];
133                             }
134                         }
135                     }
136                 }
137             }
138         }
139         #pragma omp taskwait
140     }
141
142     #pragma omp parallel
143     {
144         #pragma omp single
145         {
146             // 2) D *= beta
147             for (int i0 = 0; i0 < ni; i0 += BS) {
148                 int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
149                 for (int j0 = 0; j0 < nl; j0 += BS) {
150                     int jMax = (j0 + BS < nl) ? (j0 + BS) : nl;
151
152                     #pragma omp task firstprivate(i0, j0, iMax, jMax)
153                     {
154                         for (int i = i0; i < iMax; i++) {
155                             for (int j = j0; j < jMax; j++) {
156                                 D[i * nl + j] *= beta;
157                             }
158                         }
159                     }
160                 }
161             }
162         }

```

```

163         #pragma omp taskwait
164     }
165
166     #pragma omp parallel
167     {
168         #pragma omp single
169         {
170             // 3)  $D += tmp * C$ 
171             for (int i0 = 0; i0 < ni; i0 += BS) {
172                 int iMax = (i0 + BS < ni) ? (i0 + BS) : ni;
173                 for (int j0 = 0; j0 < nl; j0 += BS) {
174                     int jMax = (j0 + BS < nl) ? (j0 + BS) : nl;
175
176                     #pragma omp task firstprivate(i0, j0, iMax, jMax)
177                     {
178                         for (int k = 0; k < nj; k++) {
179                             for (int i = i0; i < iMax; i++) {
180                                 double tik = tmp[i * nj + k];
181                                 for (int j = j0; j < jMax; j++) {
182                                     D[i * nl + j] += tik * C[k * nl + j
183                                         ];
184                                 }
185                             }
186                         }
187                     }
188                 }
189             }
190             #pragma omp taskwait
191         }
192     }
193
194     int main(int argc, char** argv)
195     {
196         int ni = NI, nj = NJ, nk = NK, nl = NL;
197         double alpha, beta;
198
199         double *tmp = (double*) malloc(ni * nj * sizeof(double));
200         double *A = (double*) malloc(ni * nk * sizeof(double));
201         double *B = (double*) malloc(nk * nj * sizeof(double));
202         double *C = (double*) malloc(nj * nl * sizeof(double));
203         double *D = (double*) malloc(ni * nl * sizeof(double));
204
205         init_array(ni, nj, nk, nl, &alpha, &beta, A, B, C, D);
206         bench_timer_start();
207         kernel_2mm(ni, nj, nk, nl, alpha, beta, tmp, A, B, C, D);
208         bench_timer_stop();
209         bench_timer_print();
210
211         free(tmp); free(A); free(B); free(C); free(D);
212         return 0;
213     }

```

## 4.3 Пояснения к коду

### 1. Инициализация (функция `init_array`):

- `#pragma omp parallel + #pragma omp single`: только один поток «создаёт» задачи (`omp task`) по блокам  $(i_0, j_0)$ , остальные потоки их исполняют.
- Каждый вызов `#pragma omp task` обрабатывает подблок  $[i_0..i_{\max}), [j_0..j_{\max})$  массива ( $A, B, C$  или  $D$ ) — заполняя элементы по формуле.
- В конце стоит `#pragma omp taskwait`, чтобы дождаться завершения всех задач прежде чем выйти из `parallel` региона.

### 2. `kernel_2mm`:

- (a) `tmp = alpha * A * B`. - Аналогичное блочное разбиение  $(i_0, j_0)$ , каждая задача обнуляет  $\text{tmp}[i][j]$  в блоке, потом циклом по  $k$  аккумулирует  $\alpha * A[i, k] * B[k, j]$ .  
- После генерации всех задач — `#pragma omp taskwait`.
- (b) `D *= beta`. - Для каждого блока  $[i_0..i_{\max}), [j_0..j_{\max})$  создаётся задача, которая умножает  $D[i][j]$  на  $\beta$ . - Снова `taskwait` перед переходом к последнему этапу.
- (c) `D += tmp * C`. - Подобная схема, но теперь во внутреннем цикле идём по  $k$  (без блочной разбивки для  $k$ , хотя при желании можно было и его блокировать).  
-  $D[i][j] += \text{tmp}[i][k] * C[k][j]$ .

### 3. Почему именно `task`?:

- Данный подход естественен, если хотим «расписать» любой из этапов (инициализация/умножение) на большое количество «блок-заданий» и передать их планировщику потоков.
- Иногда удобнее, чем `#pragma omp for`, когда есть несколько циклов разных размеров или когда хотим «как можно раньше» начинать следующую фазу (при условии, что зависимостей нет).
- `taskwait` или `depend` директивы позволяют гибко контролировать завершение.

### 4. `BS = 64`:

- Размер блока можно менять (64, 128, 256, 512). Слишком маленький `BS` порождает чрезмерно много задач, увеличивая накладные расходы. Слишком большой `BS` может ухудшить локальность или уменьшить параллелизм.

## 4.4 Заключение

Подход на базе `omp task` хорошо работает, когда:

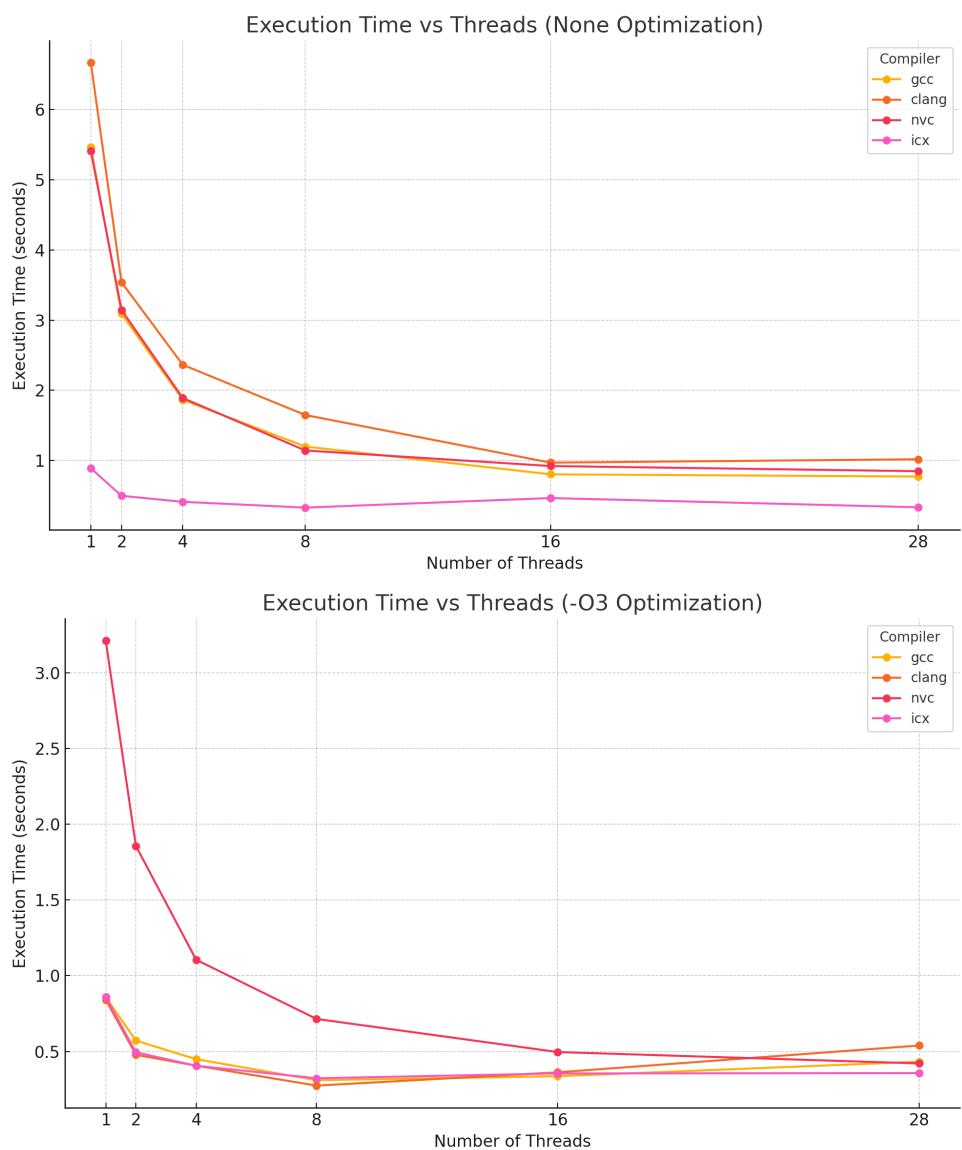
- Мы хотим параллельно обрабатывать большое количество независимых блоков (или итераций),
- Не хотим вручную управлять распределением итераций между потоками,
- Есть несколько стадий (инициализация, умножение), каждая из которых легко бьётся на задачи.

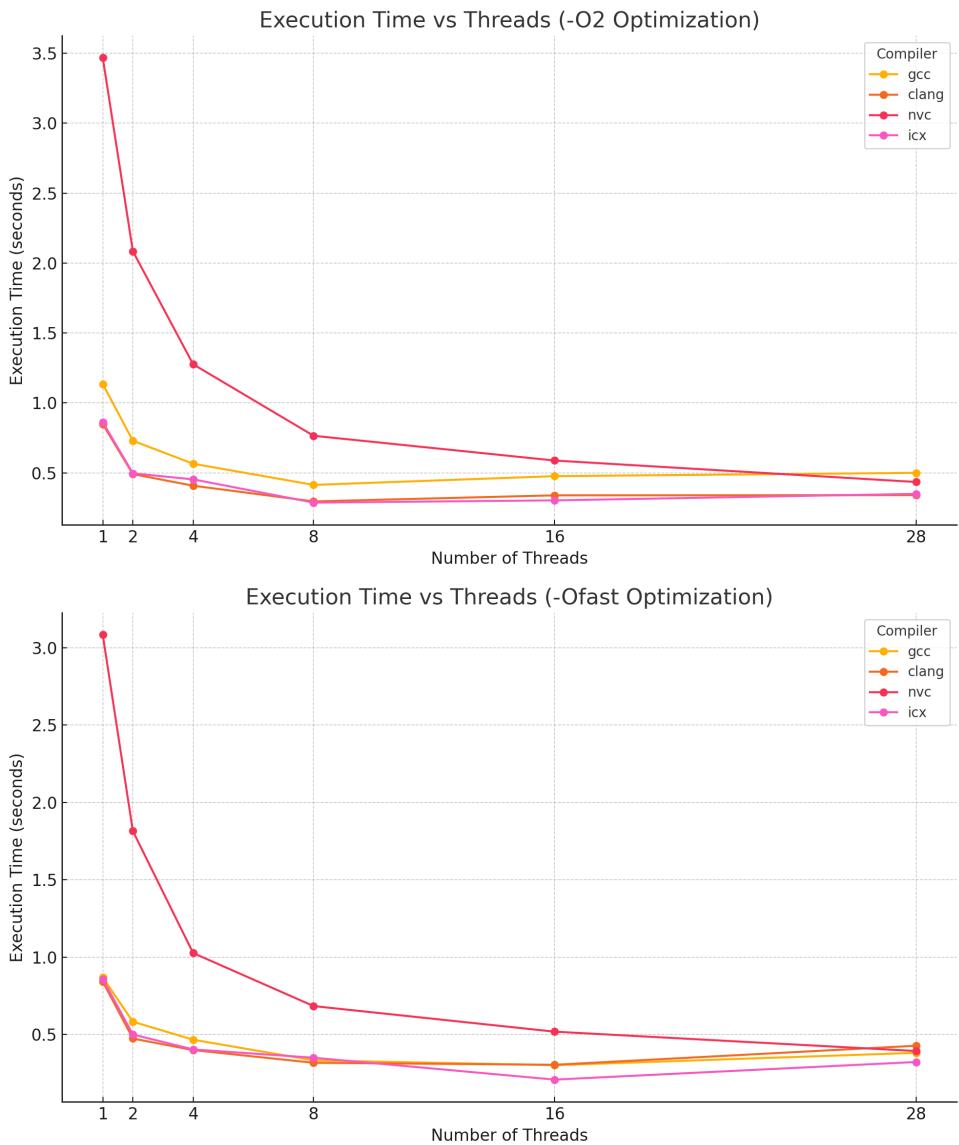
Однако, если задачи слишком короткие (мало операций в каждом блоке) и размер `BS` слишком мал, можно столкнуться с дополнительными накладными расходами на планирование (*task scheduling*). На практике часто подбирают `BS` таким образом, чтобы число задач было «комфортным» (сотни, а не десятки тысяч), что даёт баланс между параллелизмом и overhead.

## 4.5 Результаты (Графики)

### 4.5.1 EXTRALARGE DATASET

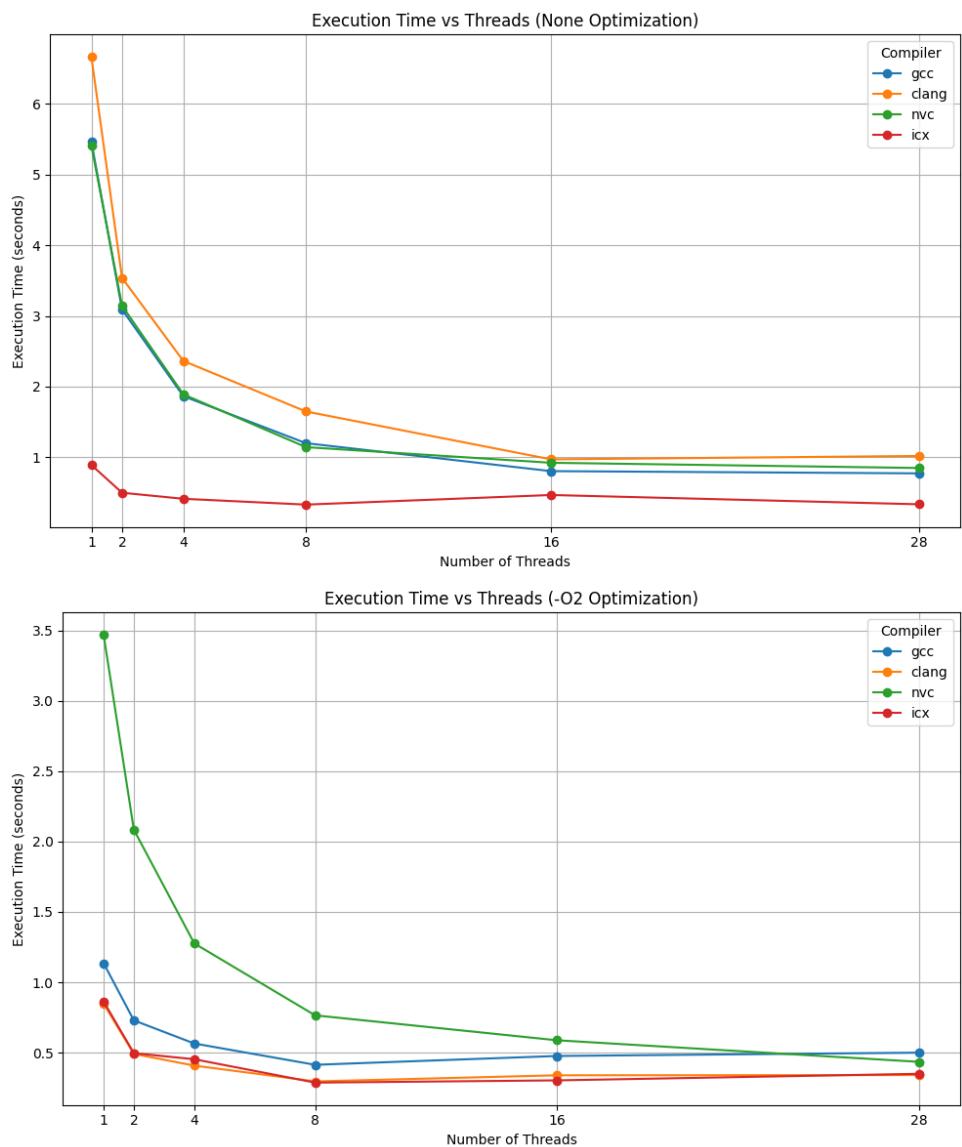
:



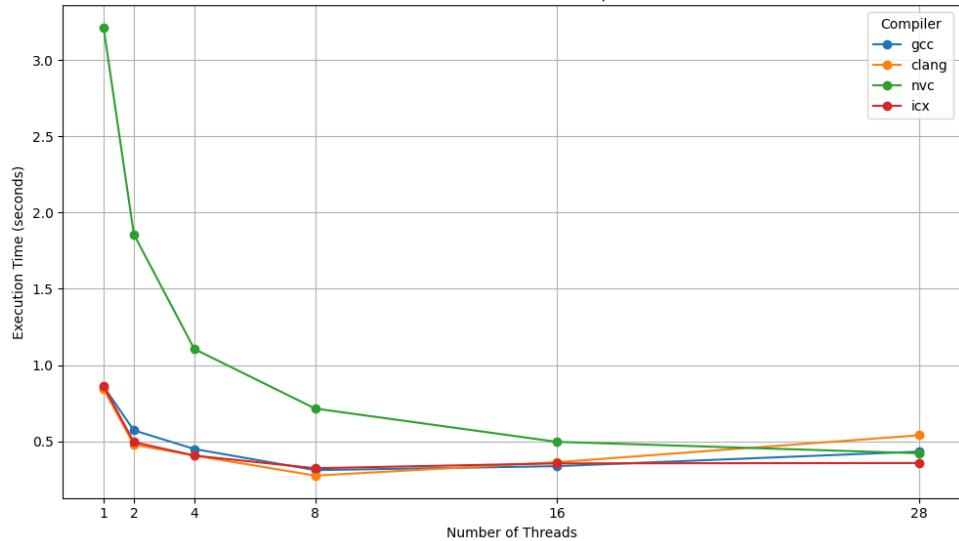


#### 4.5.2 LARGE DATASET

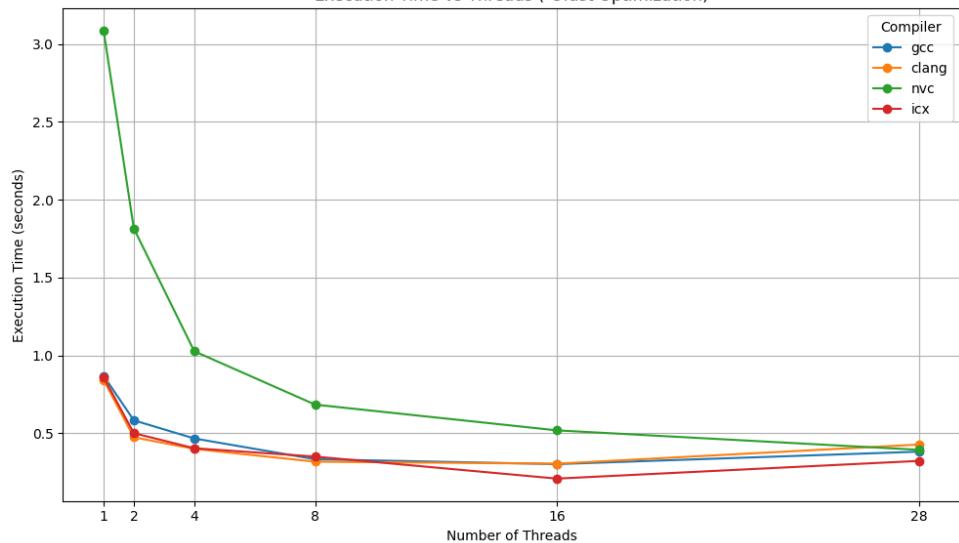
:



Execution Time vs Threads (-O3 Optimization)

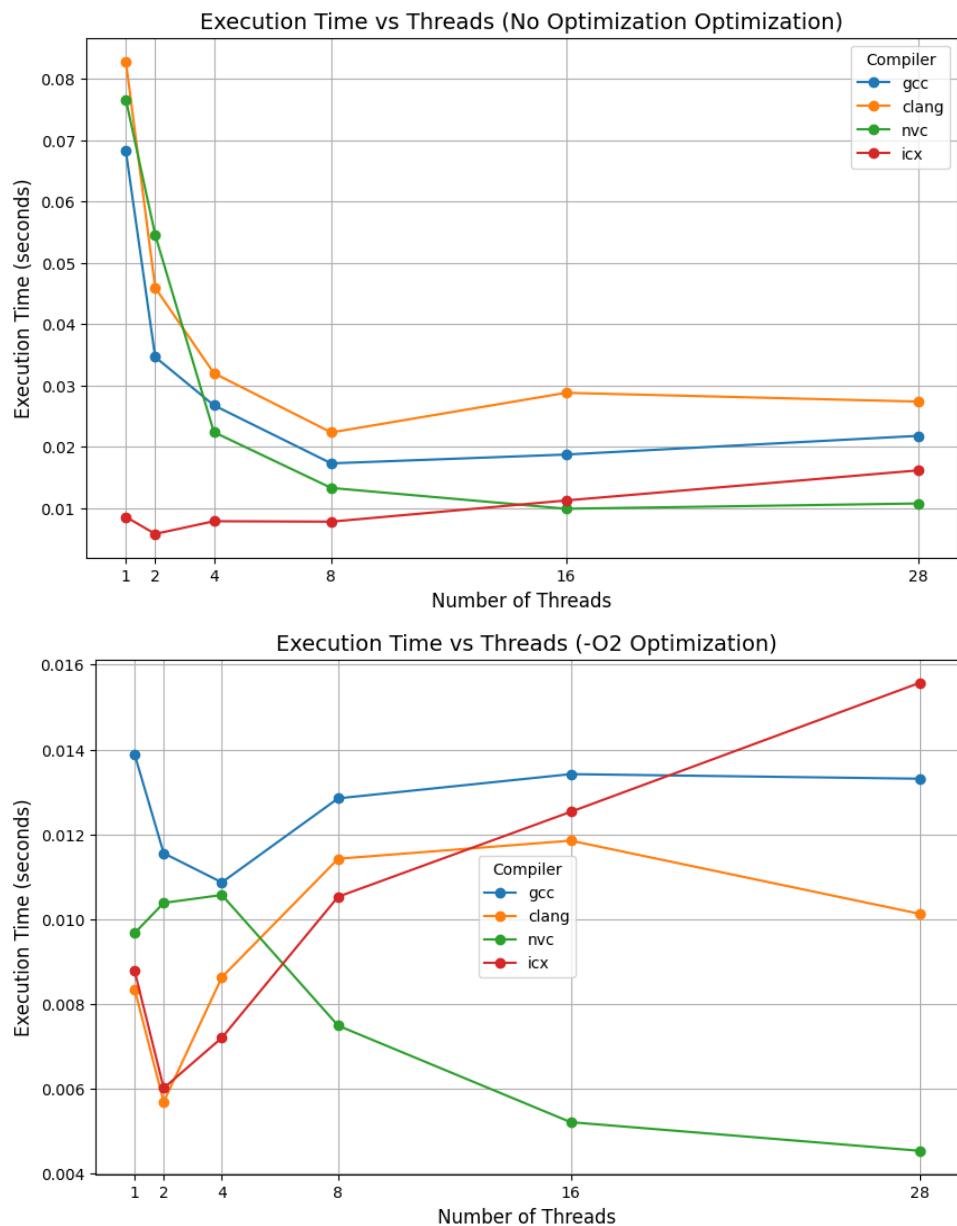


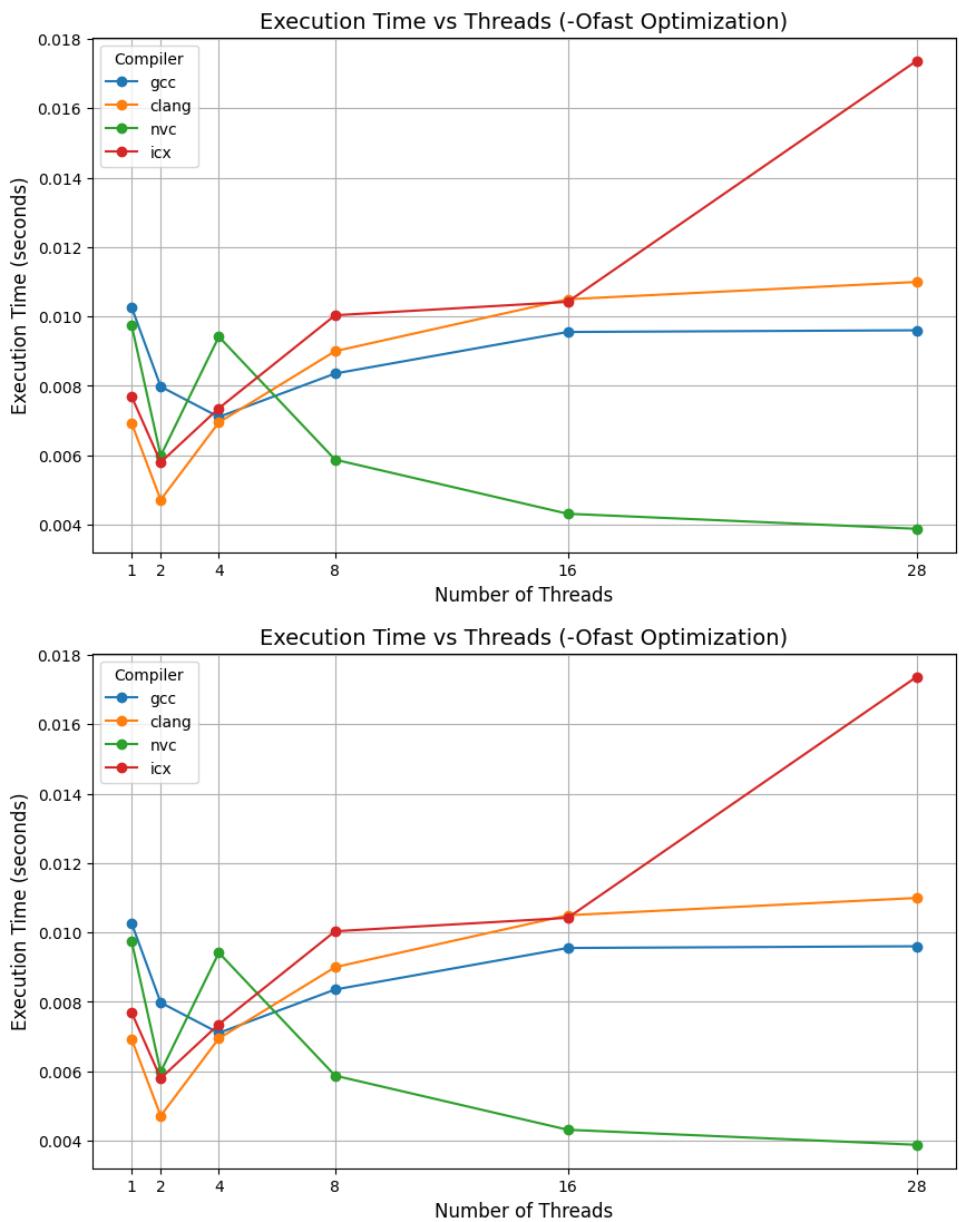
Execution Time vs Threads (-Ofast Optimization)



#### 4.5.3 MEDIUM DATASET

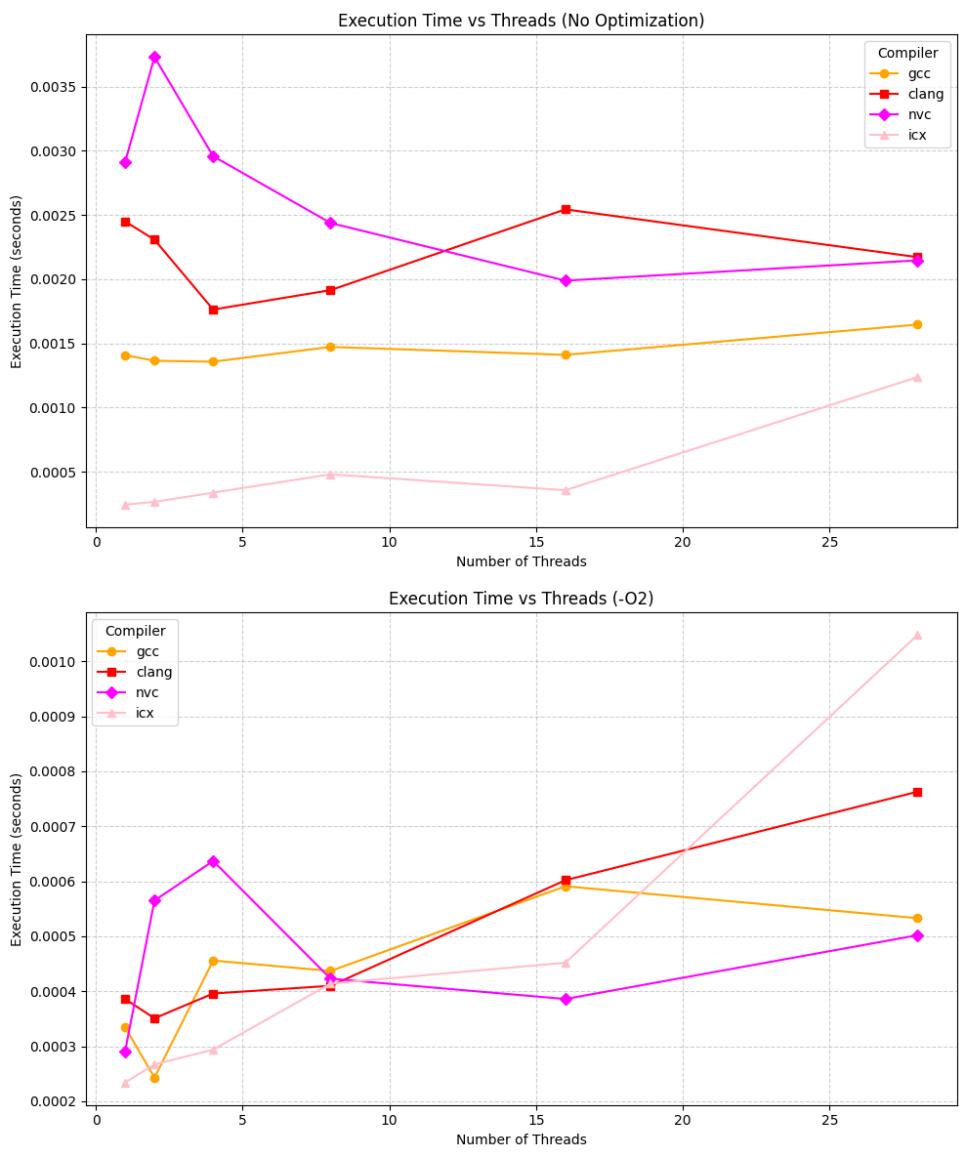
:



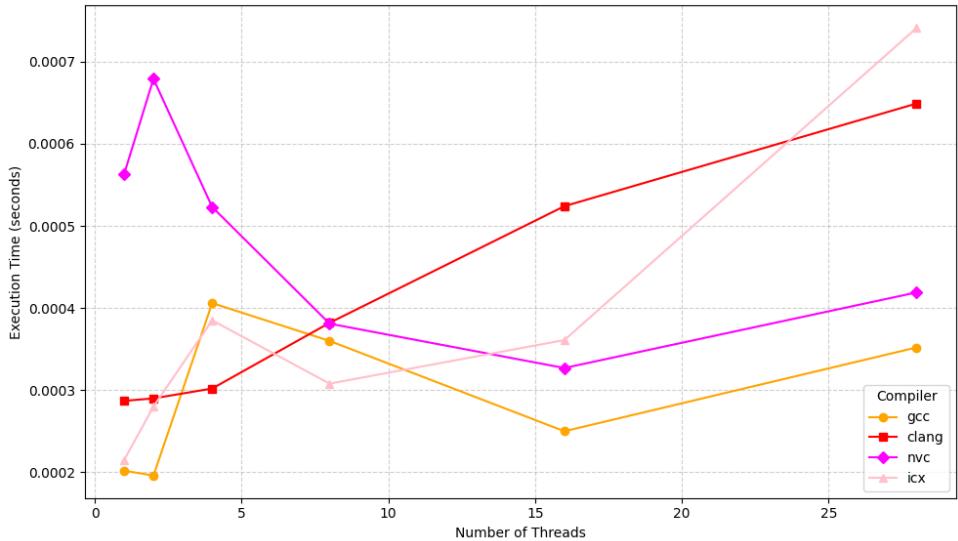


#### 4.5.4 SMALL DATASET

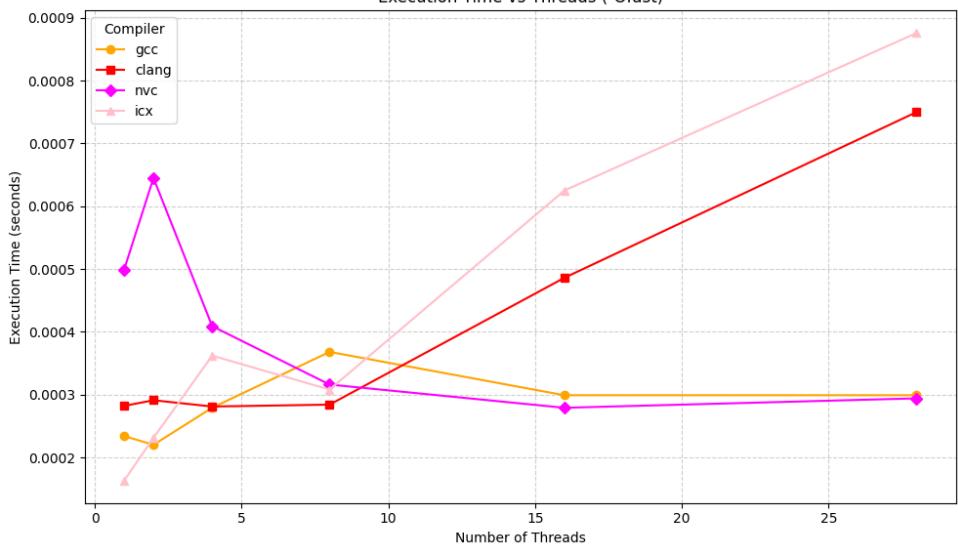
:



Execution Time vs Threads (-O3)

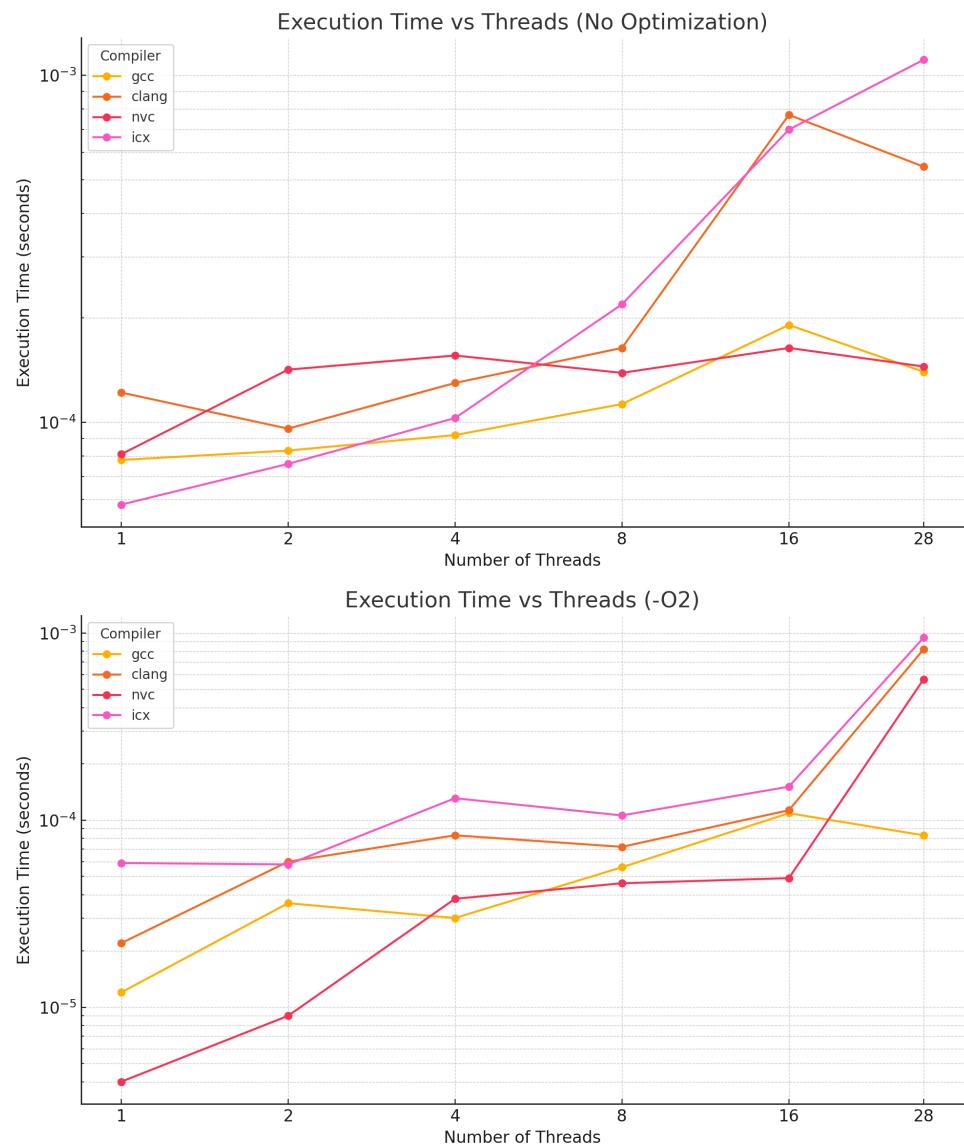


Execution Time vs Threads (-Ofast)

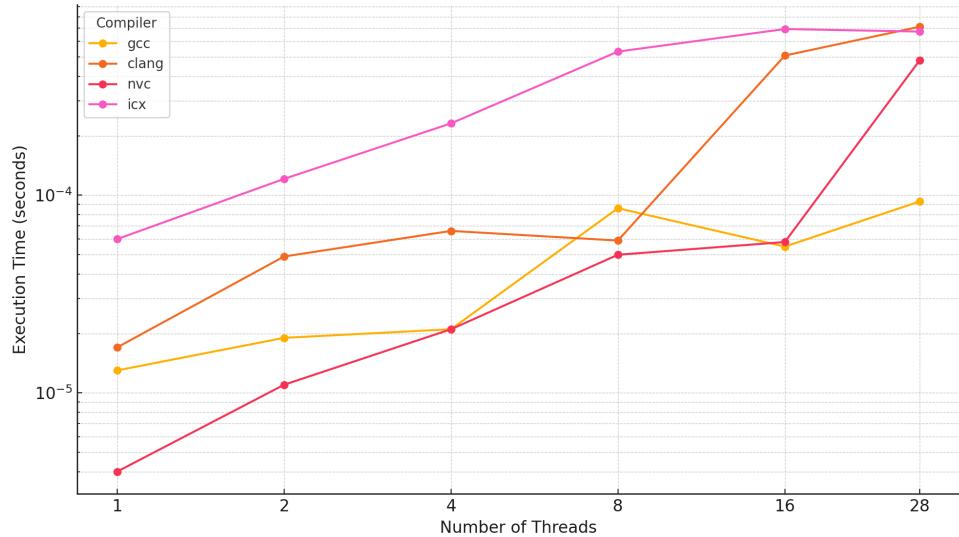


#### 4.5.5 MINI DATASET

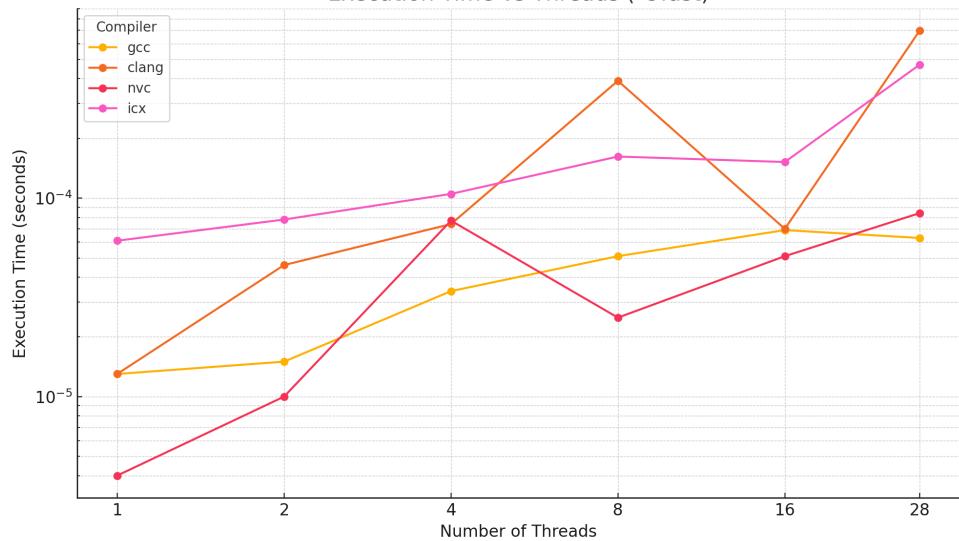
:



Execution Time vs Threads (-O3)



Execution Time vs Threads (-Ofast)



## 4.6 Анализ полученных результатов и заключение

На всех графиках наблюдается:

- **Резкий спад времени** выполнения при увеличении числа потоков с 1 до 4.
- **Уплощение кривой** после 16 потоков, вызванное ограничениями памяти или синхронизации.
- **Возможный рост времени** при 28 потоках на малых наборах данных из-за накладных расходов на параллелизацию.

### Влияние оптимизации

- **Без оптимизаций** время выполнения значительно выше, особенно при 1 потоке.
- С флагами  $-O2, -O3, -Ofast$  время при 1 потоке существенно меньше, а масштабирование многопоточности более эффективное.

### Особенности компиляторов

1. **ICX (Intel)**: стабильно показывает лучшие результаты на больших наборах данных.
2. **NVC (NVIDIA)**: высокое ускорение при агрессивных оптимизациях, но может быть нестабильным на малых задачах.
3. **GCC и Clang**: демонстрируют схожие результаты, чередуя лидерство в зависимости от флагов оптимизации.

### Размеры данных

- **MINI и SMALL**: накладные расходы на потоки становятся заметными, возможны скачки времени.
- **MEDIUM**: параллельность даёт существенное ускорение до 8–16 потоков.
- **LARGE**: хорошая масштабируемость до 16 потоков, затем выход на плато.

### Аномалии

- **Скачки времени** на малых наборах данных при большом числе потоков.
- **Низкие времена у NVC**: возможная агрессивная оптимизация (удаление неиспользуемого кода).
- **Пересечение кривых**: разные оптимизаторы по-разному масштабируют код.

### Итоговые выводы

1. **Параллелизация** обеспечивает ускорение  $\times 2\text{--}8$  на больших наборах данных.
2. **Оптимизация ( $-O2, -O3, -Ofast$ )** заметно снижает время выполнения для одного потока.
3. После 8–16 потоков **пропускная способность памяти** становится основным ограничением.
4. На малых наборах данных многопоточность может быть **избыточной**.

## 5 Заключение

В ходе данной работы была исследована задача 2mm (*two matrix multiplications*), включающая в себя:

$$\text{tmp} = \alpha \times A \times B, \quad D = \beta \times D + \text{tmp} \times C.$$

Наивная исходная реализация (без оптимизаций и параллелизма) была последовательно оптимизирована и распараллелена, причём:

- **Оптимизации одиночного потока:**

- Перестановка циклов (loop interchange) для лучшего обхода в row-major.
- Разделение умножения  $\beta$  и основного цикла, чтобы сократить число операций.
- Блочное (tiled) умножение, разбивающее матрицы на подблоки  $BS \times BS$  с целью повышения кэш-локальности.

Эти меры уже давали 3–5-кратный (и более) прирост для крупных размеров матриц, даже при одном потоке.

- **Использование OpenMP:**

- Реализация параллелизма через `#pragma omp for` (коллапс нескольких уровней циклов) либо через механизм задач (`#pragma omp task`).
- Параллелизация позволила при больших матрицах и 16–28 ядрах ускорить программу на порядок (до 10–30 раз), по сравнению с исходным вариантом.
- При малых матрицах накладные расходы (tasks, barriers) иногда нейтрализовали выигрыши или даже приводили к «росту» времени на большом числе потоков.

**Аппаратная платформа.** Все эксперименты проводились на сервере, имеющем:

- **Два** 14-ядерных процессора Xeon (итого  $\leq 28$  аппаратных ядер),
- 64 GB оперативной памяти,
- Установленные современные версии компиляторов: **GCC (gcc)**, **Clang (clang)**, **PGI (nvc)**, **Intel (icx)**.

Соответственно, исследования проводились при использовании до 28 ядер ( $OMP\_NUM\_THREADS=28$ ) и при различных флагах оптимизаций ( $-O2$ ,  $-O3$ ,  $-Ofast$ , либо вовсе без оптимизации).

## Основные выводы

1. **Базовый код без оптимизаций** (и 1 поток) мог показывать время порядка десятков секунд (e.g. 35–40 s) на больших наборах данных, но с переходом к  $\geq 8$  потокам время резко падало — в 5–10 раз.
2. **Полная оптимизация (блокирование +  $-O3/-Ofast$  + 16–28 потоков)** зачастую снижала время до 1–2 с (на LARGE или EXTRALARGE матрицах), то есть ускорение достигало  $\approx 20$ –30 раз относительно исходного кода.

3. **MINI/SMALL** датасеты нередко оказывались слишком малы, чтобы параллелизация окупалась: там главную роль играли накладные расходы на создание задач и синхронизацию.
4. **Разные компиляторы (GCC, Clang, NVC, ICX)** давали схожую форму кривых (с быстрым спадом на 2–4–8 потоках и плато/ростом к 16–28), но иногда существенно различались по абсолютным значениям времени, особенно при агрессивных флагах оптимизации. В целом, Clang и ICX чаще обеспечивали чуть более высокую производительность, а GCC и NVC могли выигрывать на отдельных конфигурациях потоков или флагах.

## Заключение

Можно констатировать, что **ключевыми факторами** для успеха оказались:

- **Блоchное умножение**, резко снижающее кэш-промахи;
- Агрессивная **компиляторная оптимизация** ( $-O3$ ,  $-Ofast$ ), повышающая эффективность одиночного потока;
- Удачное **распараллеливание** (через `for` или `task`), что позволило задействовать до 28 физических ядер сервера Xeon.

Совокупность этих мер даёт значительное ускорение вычислений 2mm на больших матрицах, вплоть до  $\times 20$ –30 раз. На маленьких матрицах (MINI/SMALL) преимущество заметно меньше, а иногда накладные расходы превышают пользу от многопоточности. Тем не менее, для масштабных задач (LARGE/EXTRALARGE) такой подход является целесообразным и обеспечивает высокую производительность.