Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э. Баумана

(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии» (ИУ7)

ОТЧЁТ ПО УЧЕБНОЙ ПРАКТИКЕ

Тип практики: Проектно-технологическая

Название предприятия: НУК ИУ МГТУ им. Н. Э. Баумана

Студент:
Постнов Степан Андреевич, группа ИУ7-21Б _____
(подпись, дата)

Руководитель от предприятия:
Старший преподаватель ИУ-7
Ломовской Игорь Владимирович (подпись, дата)

Руководитель от кафедры:
Ассистент кафедры ИУ-7
Кострицкий Александр Сергеевич (подпись, дата)

Оценка: _____

2022

Оглавление

1. Автоматизация функционального тестирования

Цель работы:

В рамках данного задания было необходимо реализовать набор скриптов для автоматизации запуска функциональных тестов лабораторных работ по курсу «Программирование на Си».

Проделанная работа:

Идея автоматизации заключается в использовании перенаправления ввода/вывода. Данные для тестирования каждой лабораторной работы подготавливаются заранее. Позитивный (*pos_case.sh*) и негативный (*neg_case.sh*) сценарии тестирования реализуются раздельно. За «обход» тестовых данных и вызов позитивного и негативного сценариев отвечает скрипт (*func_tests.sh*). Сравнение полученного результата работы программы и эталонного выполняется с помощью с компаратора (*comparator.sh*). Было реализовано два универсальных компаратора: 1) Компаратор для сравнения последовательностей действительных чисел, располагающихся в двух текстовых файлах, с игнорированием остального содержимого; 2) Компаратор для сравнения содержимого, располагающегося после первого вхождения подстроки «*Result:␣*», двух текстовых файлов. Также дополнительно были реализованы скрипты релизной и отладочной сборок исполняемого файла (*build_release.sh*) и (*build_debug.sh*) соответственно и скрипт удаления «мусорных» файлов из рабочей области (*clean.sh*).

Вывод:

Реализованные скрипты помогли мне в написании лабораторных работ, уменьшив время поиска ошибок и их устранения.

2. Изучение этапов компиляции и строения исполняемого файла

Цель работы:

Целью данной работы является изучение процесса получения исполняемого файла и организации объектных и исполняемых файлов.

Проделанная работа:

Первая часть задания

Для сравнения «ручной» сборки и сборки компилятором *gcc* использовался последовательный вызов следующих утилит: 1) *cpp* (обработка препроцессором и получение единицы трансляции); 2) c99 (трансляция на язык «ассемблера»); 3) as (ассемблирование в объектный файл); 4) ld (компоновка).

Чтобы сравнить этапы получения исполняемого файла компиляторов *clang* и *gcc*, сборка производилась с ключами -v и -save-temps, которые указывают компиляторам сохранять все созданные в процессе компиляции файлы и выводить подробную справку о вызываемых командах.

Вторая часть задания

Изучить и сравнить секции объектных (исполняемых) файлов, дизассемблировать объектные файлы помогла утилита *objdump*, предназначенная для отображения различной информации об объектных (исполняемых) файлах. Для определения, различаются ли файлы или нет, использовалась утилита *diff*.

Выводы:

Первая часть задания

По результатам выполнения первой части задания можно сделать вывод, что видимых отличий между этапами сборки вручную и установленным компилятором *gcc* нет. Также было установлено, что компилятор *clang* выполняет на один этап получения исполняемого файла больше, чем *gcc*, однако глобально выполняет одну и ту же задачу с ним и делает это схожим образом.

Вторая часть задания

По результатам выполнения второй части задания можно сделать вывод, что в объектные и исполняемые файлы с отладочной информацией добавлены секции .debug, содержащие отладочную информацию. При этом остальное содержимое не различается. Таким образом, размер *debug* файлов больше, чем размер обычных.

Были исследованы отличия дизассемблированного объектного файла от полученной программы на языке ассемблера, на основе которых была составлена следующая сравнительная таблица, по данным которой был сделан вывод, что различия в дизассемблерном и ассемблерном кодах имеются, однако исходный код программы сохраняется:

Дизассемблерный код	**Ассемблерный код**
Содержит ассемблерный код и сопоставленный ему машинный	Содержит только ассемблерный код
Кол-во секций равно 7	Кол-во секций равно 3

Было исследовано, в какие секции попадают функции, глобальные переменные и локальные переменные:

Объекты программы	**Секции**
Функции	.text
Глобальные переменные (проинициализированные)	.data
Глобальные переменные (непроинициализированные)	До линковки переменная попадет в секцию *COM*, после чего будет направлена в секцию .data, если она инициализирована в другом файле. Иначе в секцию .bss
Локальные переменные	-

3. Отладка

Цель работы:

Приобретение умения самостоятельно производить трассировку приложения и делать отладку.

Проделанная работа:

Главным инструментом для выполнения данного задания послужил отладчик *gdb*, обладающий множеством полезных команд для нахождения ошибок и устранения ошибок. Например, вызвав команды *x* и *sizeof*, можно узнать адрес и размер соответственно любой переменной программы, а благодаря команде *rwatch* можно установить точку наблюдения в любое место. Таким образом, используя *gdb*, можно с легкостью изучить представление в памяти многомерных статических массивов целых чисел, строк, структур с выравниванием и без.

Вывод:

Знакомство с отладчиком *gdb* позволило сэкономить время на отладке программ во время написания лабораторных работ по курсу «Программирование на СИ», а также помогло разобраться с тонкостями языка СИ.

4. Замерные эксперименты

Цель работы:

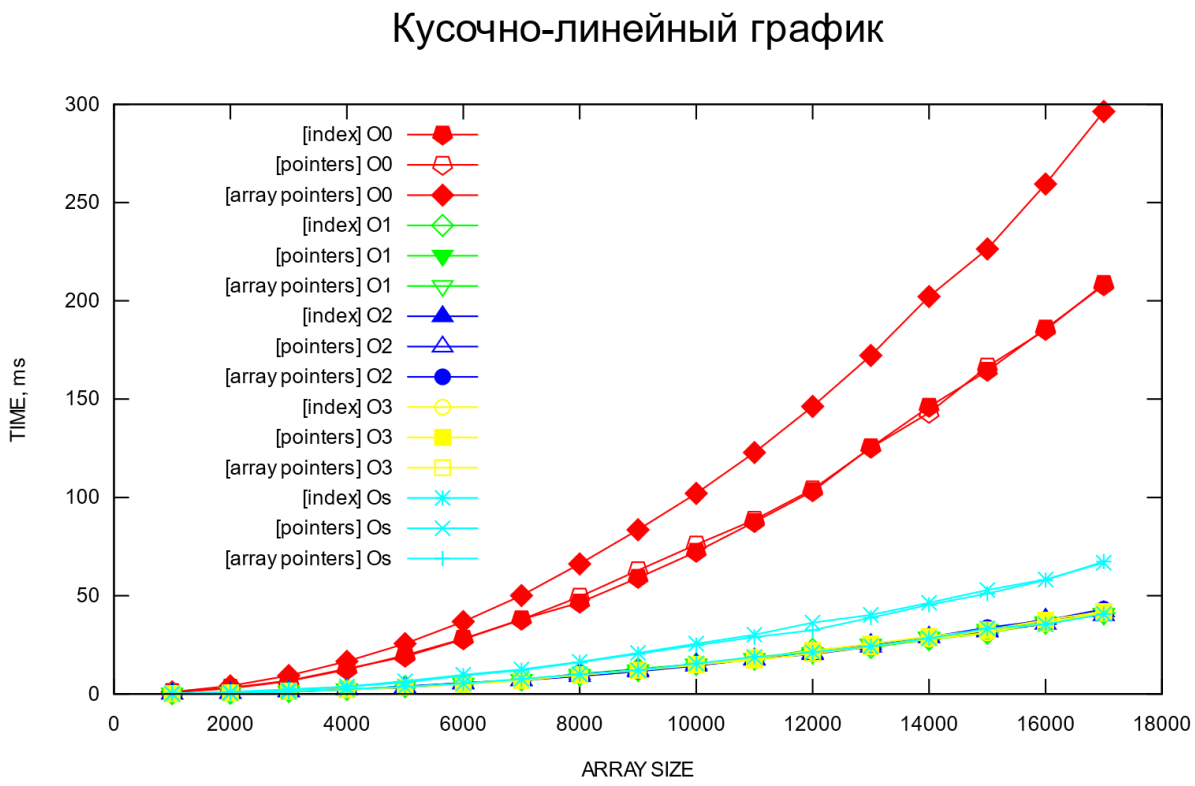
Целью данной работы являлось измерение времени работы программ и определение зависимости полученных результатов от выбранной реализации.

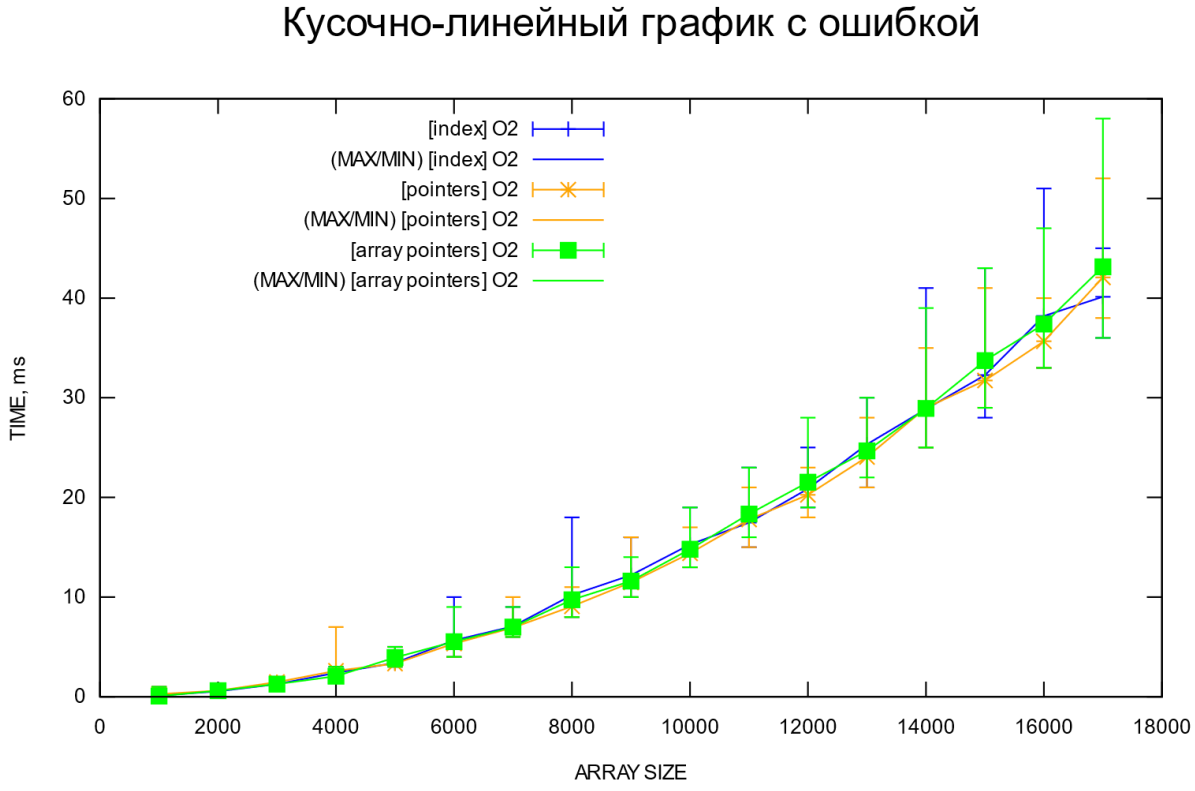
Проделанная работа:

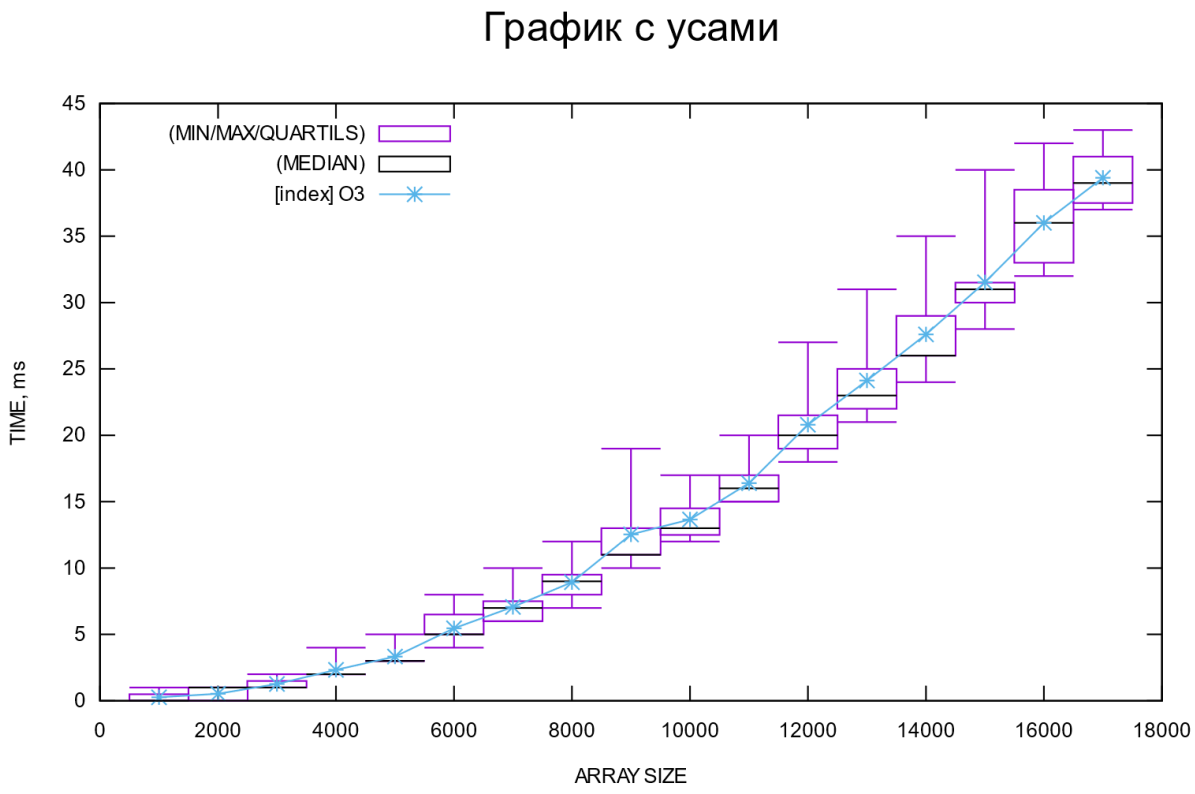
В ходе выполнения данного задания, были реализованы определенные программы и скрипты, позволяющие получить весь набор необходимых исполняемых файлов (*build_apps.sh*), добавить некоторые данные в датасет экспериментов (*update_data.sh*), подготовить данные из набора, провести первичный анализ: посчитать среднее арифметическое, медианное, найти максимум и минимум, вычислить нижний и верхний квартили (*make_preproc.py*), получить графики зависимостей (*make_postproc.py*), получить данные эксперимента (*go.sh* - скрипт вызывает по очереди четыре описанных выше).

Выводы:

Первый замерный эксперимент

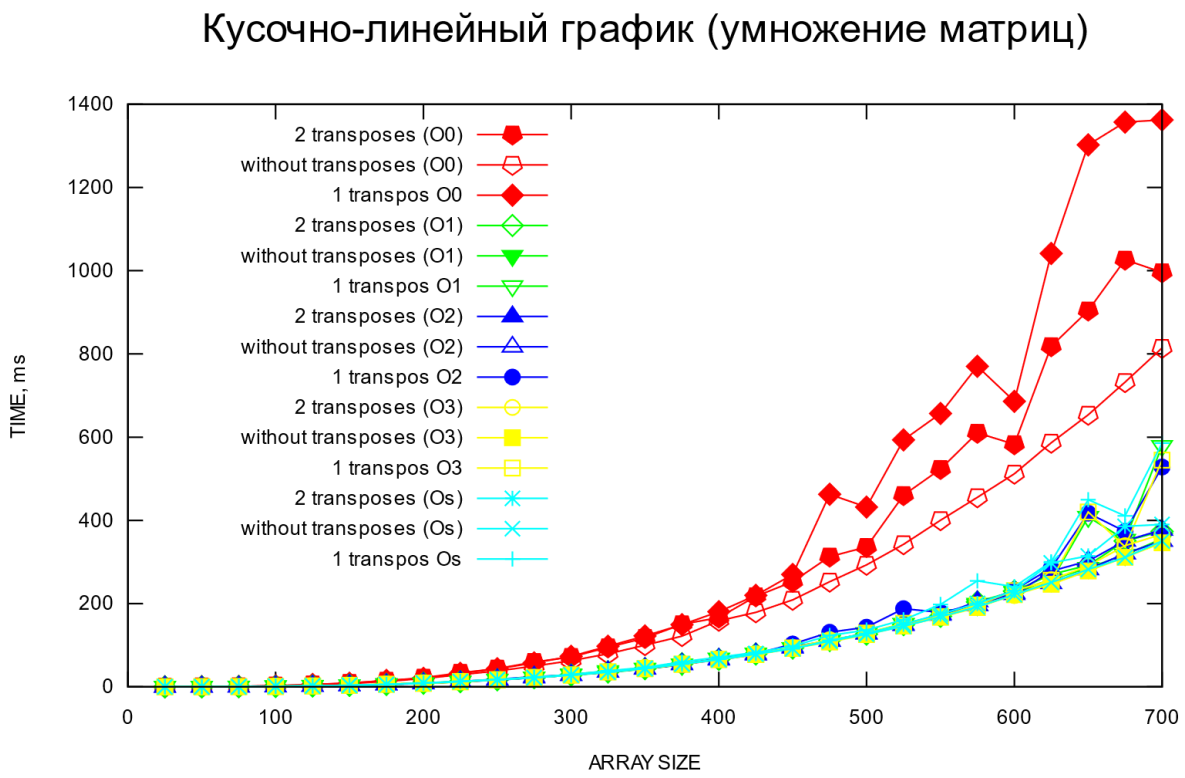




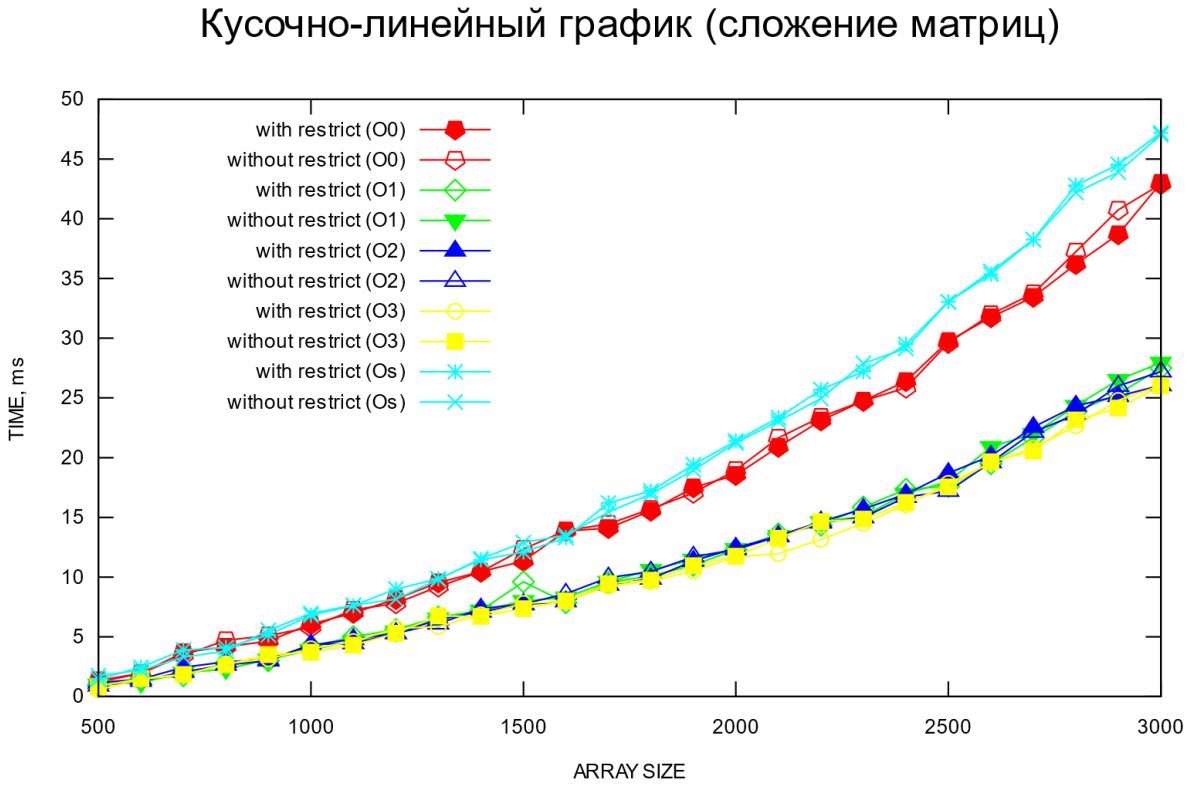


Способ обработки массива через указатели является быстрее, чем обработка при помощи операции индексации (a[i]) и формальной замены индексации (*(a + i)), так как программа не хранит в памяти дополнительную переменную «i», определяющую смещение при движении по массиву. Однако результаты эксперимента при уровне оптимизации O0 показывают обратное, что означает неудачно выбранную реализацию программы.

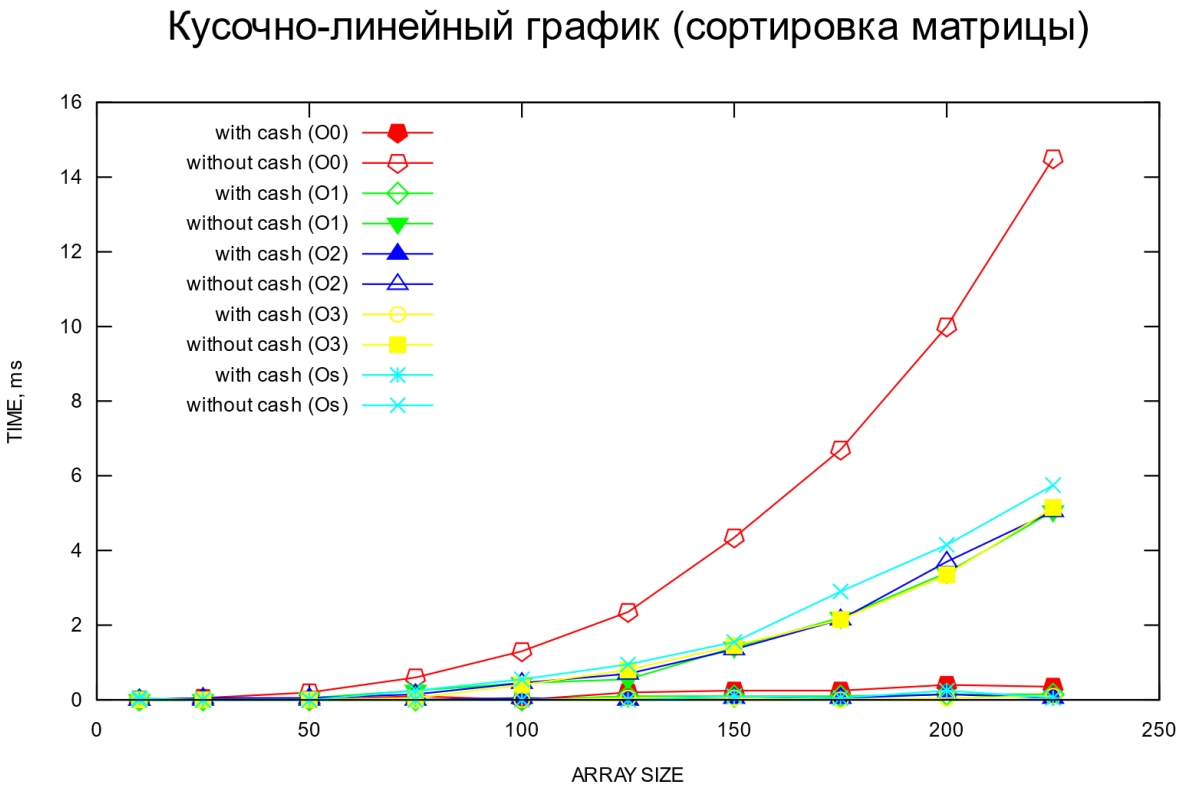
Второй замерный эксперимент



Исходя из графика, можно сделать вывод, что при транспонировании только первой матрицы, программа работает дольше всего, так как при перемножении циклы идут по столбцам обеих двумерных массивов; при транспонировании двух матриц время работы больше, чем без транспонирования, так как перемножение происходит прежним образом, но с дополнительными вызовами функции транспонирования. Также можно заметить, что при некоторых размерностях (например, при 600) происходит выброс, обусловленный определенным шагом процессора, при котором возникает конфликт кэша данных.



Исходя из графика, можно сделать вывод, что для данной реализации программ использование уровня оптимизации «Os» нецелесообразно, так как время работы заметно увеличивается из-за слишком большого сокращения кода, что уменьшает скорость («по закону сохранения»). Также из документации становится понятно, что «restrict» позволяет компилятору (в некоторых случаях) ускорить код , сделав дополнительные оптимизации, однако в данном случае ощутимой разницы во времени работы не наблюдается.



Исходя из графика, можно сделать вывод, что сортировка с кэшированием работает гораздо быстрее сортировки без кэширования, так как компьютер не тратит каждый раз время на подсчет суммы в строке, заходя в цикл и увеличивая сложность алгоритма.