Санкт-Петербургский государственный университет

Математико-Механический факультет

Кафедра Системного Программирования

Группа 20Б.11-мм

Казанцев Антон Алексеевич

Реализация быстрых операций длинной арифметики для x64 Отчёт по учебной практике

Отчёт по учебной практике

Научный руководитель:
ст. преп. Баклановский М.В.

Санкт-Петербург 2022

Оглавление

Введение 3
Цель работы 3
Задачи 3
Обзор функций библиотеки GNU MP 4
Выбор реализуемых функций 4
Технологии процессора intel x64, позволяющие увеличить скорость выполнения кода 5
Алгоритм тестирования 5
Описание графиков 5
Сложение 6
Тест GNU MP 6
Тест наивной реализации на языке Си 7
Тест наивной реализации на Assembler 8
Изучение поведения графика, кэш процессора 9
Улучшение и тест реализации на Assembler 11
Умножение 13
Тест GNU MP (512 байт) 14
Тест алгоритма “в столбик” на Assembler (512 байт) 15
Тест GNU MP (256 байт) 16
Тест алгоритма “в столбик” на Assembler (265 байт) 17
Умножение алгоритмом Карацубы 18
Вывод 19
Список литературы 19

Введение

Алгоритмы длинной арифметики являются неотъемлемой частью современных задач программирования. Скорость вычислений длинной арифметики может значительно влиять на эффективность работы ПО.

С ростом производительности вычислителей растёт скорость выполнения таких вычислений. Можно ли добиться большей скорости выполнения операций с длинными числами, чем у распространённых библиотек длинной арифметики, за счёт использования особенностей процессора intel x64?

Цель работы: написание программы, реализующей быстрое выполнение операций длинной арифметики для intel x64.

Задачи:

- изучить существующие реализации длинной арифметики
- выбрать набор операций длинной арифметики для реализации
- изучить особенности устройства процессора intel x64 и возможные способы ускорить вычисления
- составить качественную тестовую базу
- научиться сравнивать скорость вычислений двух и более различных реализаций длинной арифметики
- написать свою реализацию, используя инкрементный метод разработки для качественного анализа роста производительности по отношению к применяемым технологиям процессора

Обзор функций библиотеки GNU MP

GNU MP – наиболее распространённая бесплатная библиотека, позволяющая работать с длинной арифметикой на языках C/C++.

Данная библиотека предоставляет следующий функционал:

- типы данных:
  - целые числа
  - рациональные числа
  - числа с плавающей запятой
- функции для целых чисел:
  - сложение
  - вычитание
  - сравнение
  - умножение (алгоритм Карацубы)
  - совмещённые операции сложения и умножения
  - взятие модуля
  - деление с получением целого или нецелого результата
  - возведение в степень
  - логические операции

Выбор реализуемых функций

Для реализации был выбран функционал:

- типы данных:
  - целые числа
- функции для целых чисел:
  - сложение
  - вычитание
  - сравнение
  - умножение (алгоритм “в столбик”)
  - умножение (алгоритм Карацубы)
  - деление с получением целого результата
  - возведение в степень
- важные оговорки
  - реализуемые функции не выделяют память самостоятельно
  - функции соответствуют call convention для C++

Технологии процессора intel x64, позволяющие увеличить скорость выполнения кода

В процессоре x64 есть возможность организовать конвейерные вычисления и использовать 32-байтовые регистры для оптимизации вычислений длинной арифметики.

В итоговой версии работы будет представлены две реализации длинной арифметики:

- с использованием 64-битных регистров в качестве разрядов длинного числа
- с использованием 32-байтовых регистров в качестве разрядов длинного числа

Алгоритм тестирования

При тестировании алгоритма, написанного для intel x64, важно помнить про ряд особенностей данного процессора:

- конвейеризация
- отсутствие гарантии исполнения команд в том порядке, в каком они написаны

Таким образом, замер времени работы алгоритма должен осуществляться следующим образом:

1. Остановка конвейера
2. Замер времени старта
3. Вызов функции с тестируемым алгоритмом
4. Остановка конвейера
5. Замер времени финиша

Такой алгоритм замера времени гарантирует, что полученное значение корректно отражает скорость работы вызываемой функции.

Описание графиков

Все графики с результатами тестов, встреченные в работе содержат в себе информацию:

- горизонтальная ось – время выполнения теста в тактах процессора
- вертикальная ось – количество тестов, исполнившихся за данное время

Каждому значению на горизонтальной оси соответствует только одно значение на вертикальной оси.

Как правило, графики шире по горизонтальной оси, чем показано в работе. Обрезание графиков обусловлено аномально долгим исполнением некоторых тестов, вследствие обработки операционной системой аппаратных прерываний и прочих событий. Количество тестов, исполнившихся за такое время незначительно и игнорируется в данной работе.

Сложение

Сложение длинных чисел является одной из важнейших операций, поскольку активно используется в операциях умножения.

Тест GNU MP

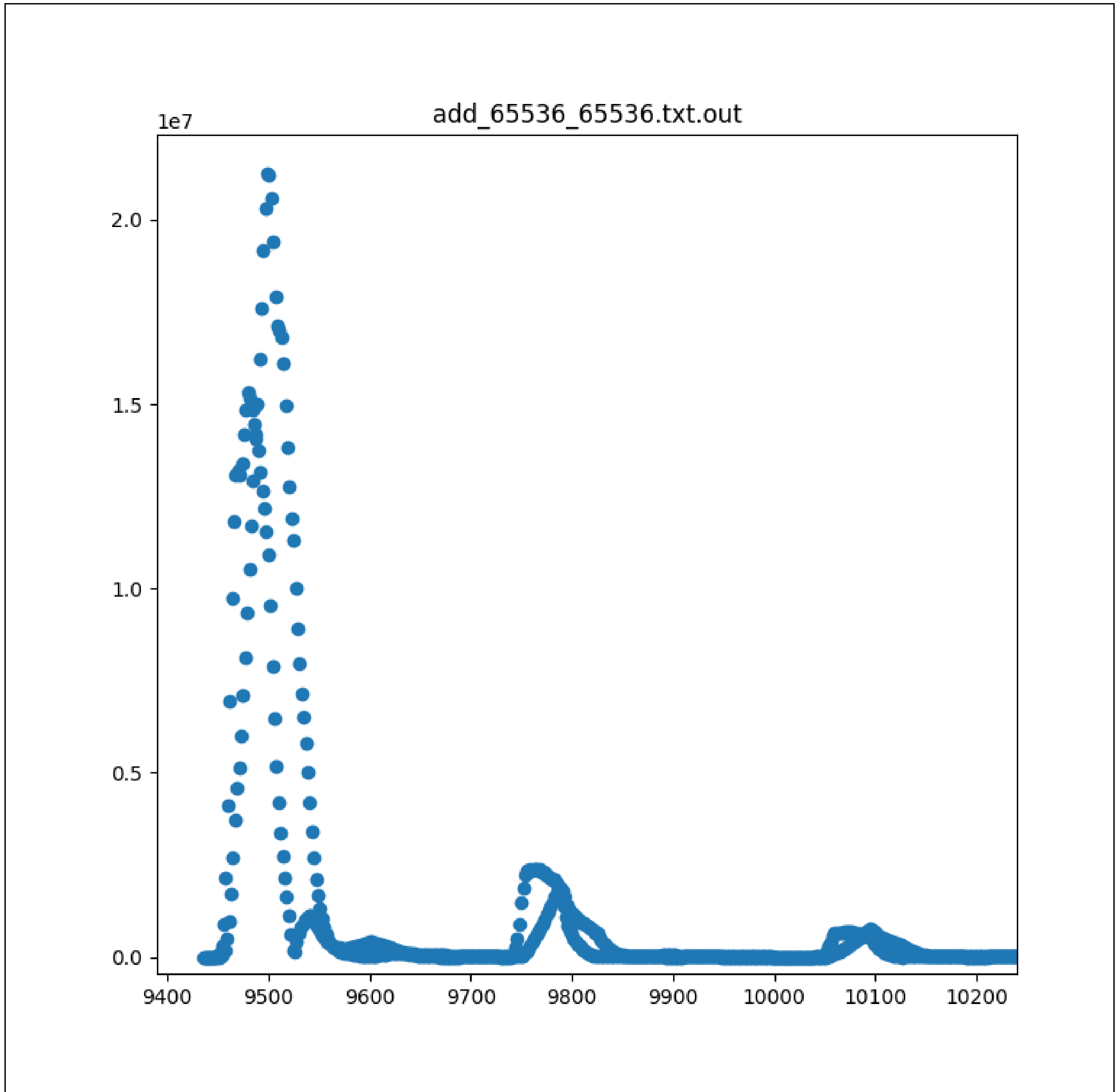

График скорости работы GNU MP при сложении двух чисел размером по 65536 шестнадцатеричных разрядов (32 кб)

Как видно из графика, большинство тестов сложения двух чисел размером 32 кб функцией сложения из библиотеки GNU MP завершились приблизительно за 9500 тактов процессора.

Тест наивной реализации на языке Си

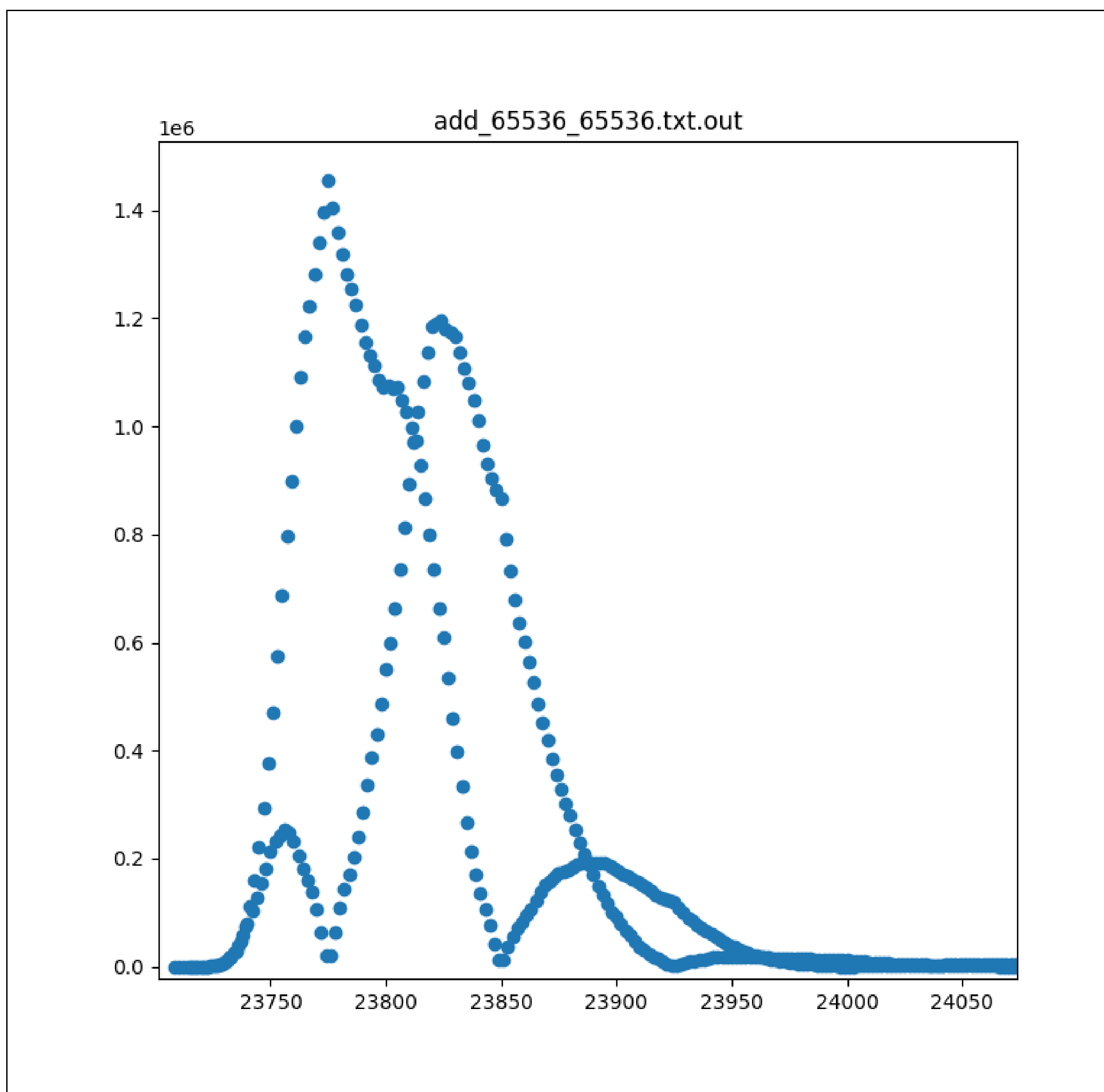

График скорости работы наивной реализации на Си при сложении двух чисел размером по 65536 шестнадцатеричных разрядов (32 кб)

После проведения данного теста, попытки выполнения поставленной задачи с использованием только лишь возможностей языка Си, были прекращены.

Данное решение связано с тем, что в синтаксисе языка Си нет достаточного функционала для реализации алгоритмов длинной арифметики с размером одного разряда длинного числа более чем 8 байт. Также при реализации такой задачи на Си возникает проблема недостаточного контроля за результатом компиляции. Для получения алгоритма, который бы удовлетворил поставленным требованиям необходим контроль за каждой исполняемой во время сложения командой.

Тест наивной реализации на Assembler

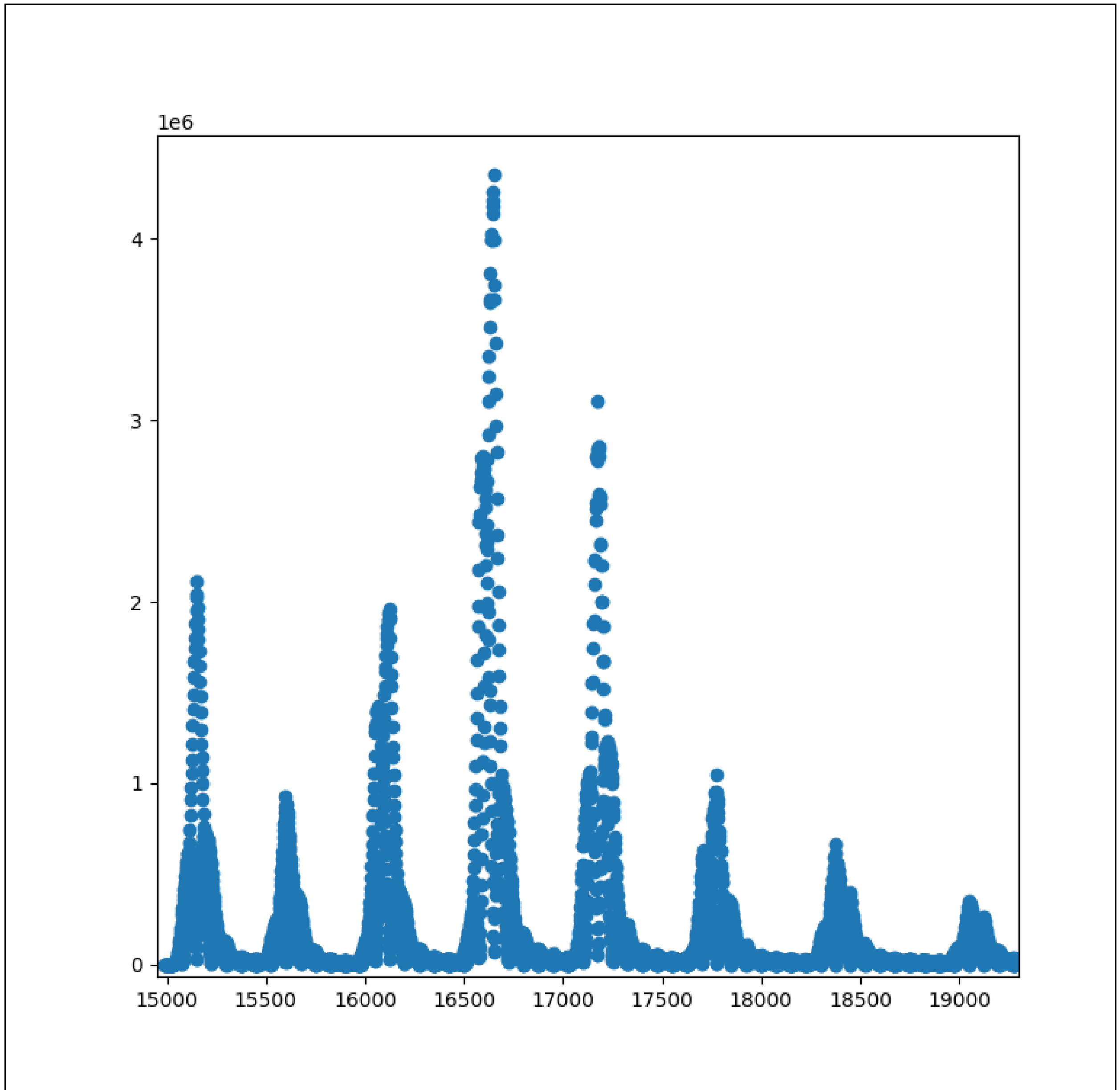

График скорости работы наивной реализации на Assembler при сложении двух чисел размером по 65536 шестнадцатеричных разрядов (32 кб)

Первая наивная реализация алгоритма сложения длинной арифметики, написанная на ассемблере, уже смогла обогнать во многом похожий алгоритм на Си. Однако на графике отчётливо видно 8 пиков, чего не встречалось ранее при тестах. С чем связано данное поведение алгоритма?

Изучение поведения графика, кэш процессора

Наиболее вероятная причина появления восьми горбов на графике – специфическая работа кэша процессора. При доступе процессора в память сначала производится проверка, хранит ли кэш запрашиваемые из памяти данные. Случай, в котором линия с запрашиваемыми данными находится в кэше называется попаданием в кэш, обратный же случай называется кэш-промахом. Во время работы могут случиться как попадания в кэш, так и промахи. За счёт различного количества попаданий и промахов возникает несколько групп тестов, различных по времени работы.

Известный способ увеличения количества попаданий в кэш при работе с массивами – выравнивание адресов начала массивов по 64 байта.

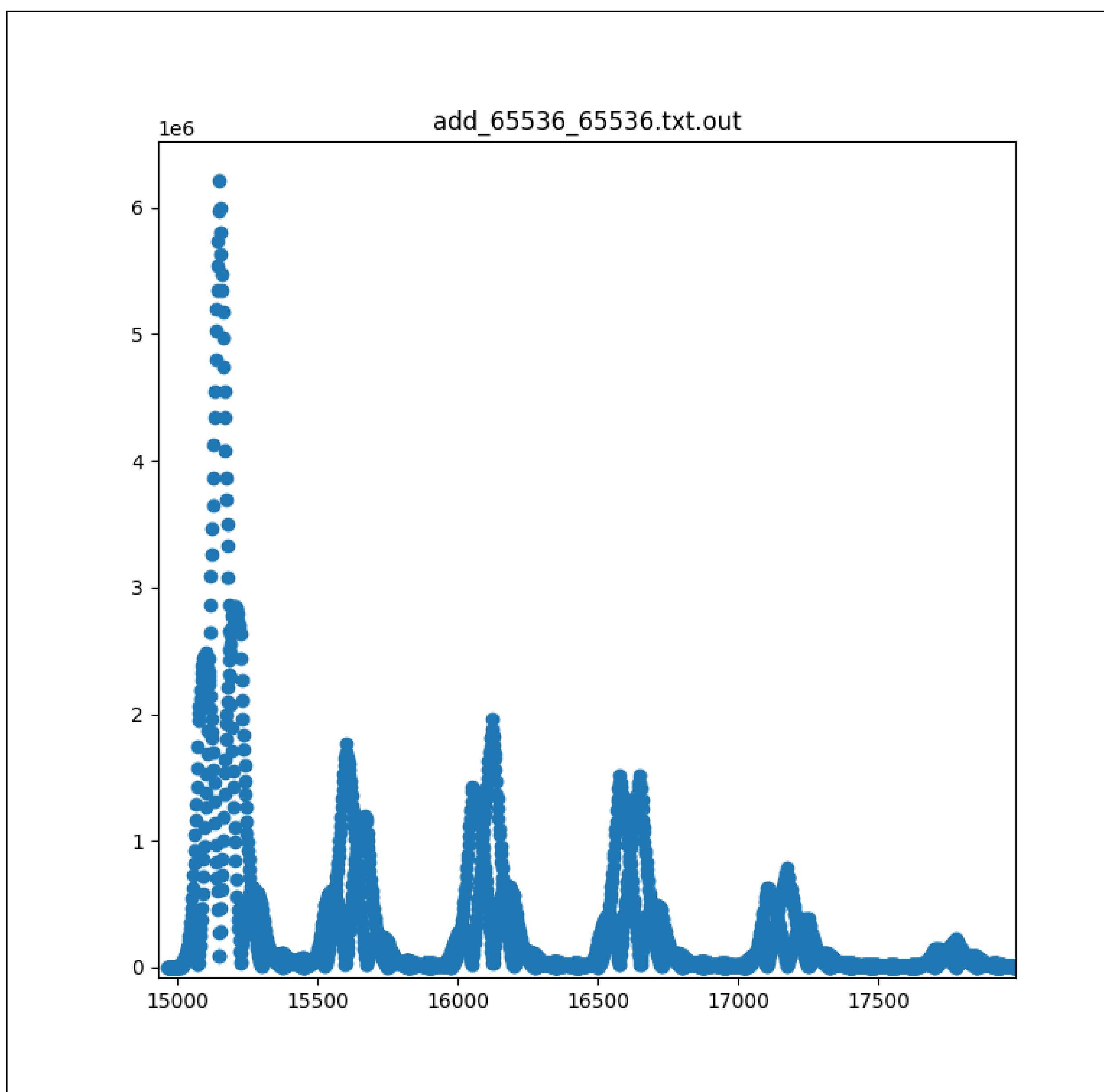

График скорости работы наивной реализации на Assembler при сложении двух чисел размером по 65536 шестнадцатеричных разрядов (32 кб), массивы длинных чисел выровнены в памяти по 64 байта

После выравнивания количество горбов уменьшилось до шести. Однако стоит попробовать ещё один метод уменьшения количество промахов.

В процессорах intel x64 используется N-way set associative кэш. (N-канальный)

Для увеличения количества попаданий нужно расположить массивы на расстоянии, равном половине канала. Таким образом получится уменьшить количество наложений друг на друга данных внутри одного канала, а следовательно – увеличить вероятность попадания в кэш.

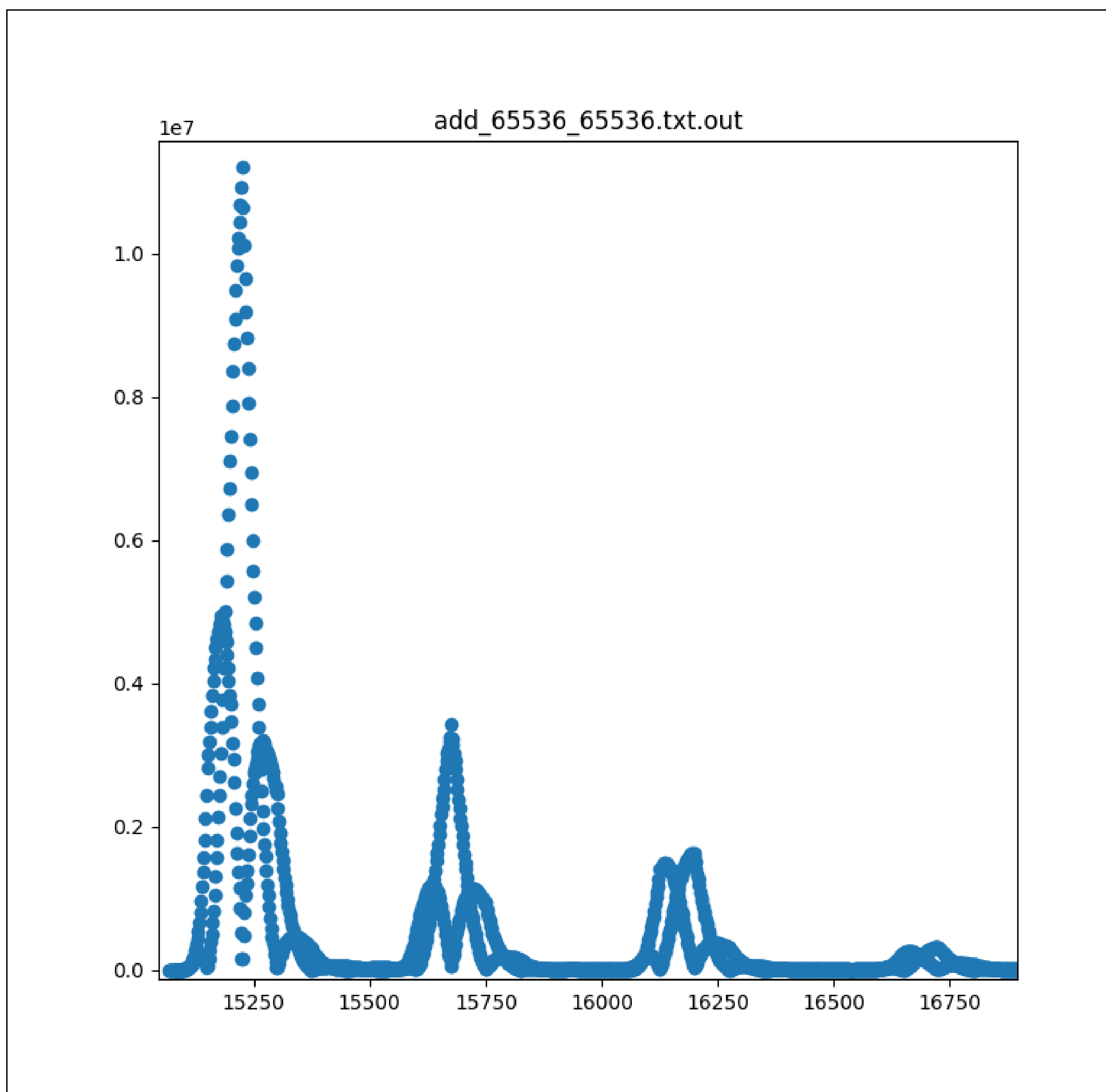

График скорости работы наивной реализации на Assembler при сложении двух чисел размером по 65536 шестнадцатеричных разрядов (32 кб), массивы длинных чисел выровнены в памяти по 64 байта, между массивами половина канала кэша

Таким образом, за счёт манипуляций с расположением массивов длинных чисел в памяти удалось решить проблему восьми горбов.

Улучшение и тест реализации на Assembler

Для того чтобы улучшить алгоритм, разберём наиболее уязвимую к потере скорости его часть – основной цикл сложения.

- rax – целочисленный регистр размером 64 бит
- cf – carry flag, хранит 1, если предыдущая операция над числами была завершена с переполнением
  (единственная такая операция в данном коде - adc)
- rsi – хранит адрес массива с первым слагаемым, которое гарантированно меньше по количеству разрядов, чем второе
- rdi – хранит адрес массива со вторым слагаемым
- [] – разыменование адреса
- rcx – целочисленный регистр размером 64 бит, хранит длину наименьшего по количеству разрядов слагаемого (rsi)

sum_loop:	Указатель начала цикла
lodsq	rax = [rsi]; rsi = rsi + 1;
adc rax, qword ptr [rdi]	rax = rax + [rdi] + cf
stosq	[rdi] = rax; rdi = rdi + 1;
loop sum_loop	rcx = rcx – 1; if rcx != 0 jump sum_loop;

В данном коде использована специальная команда для реализации цикла “loop”, которая иногда может работать медленнее, чем её аналог, составленный из более простых операторов.

sum_loop:	Указатель начала цикла
lodsq	rax = [rsi]; rsi = rsi + 1;
adc rax, qword ptr [rdi]	rax = rax + [rdi] + cf
stosq	[rdi] = rax; rdi = rdi + 1;
dec rcx	rcx = rcx - 1
jnz sum_loop	if rcx != 0 jump sum_loop;

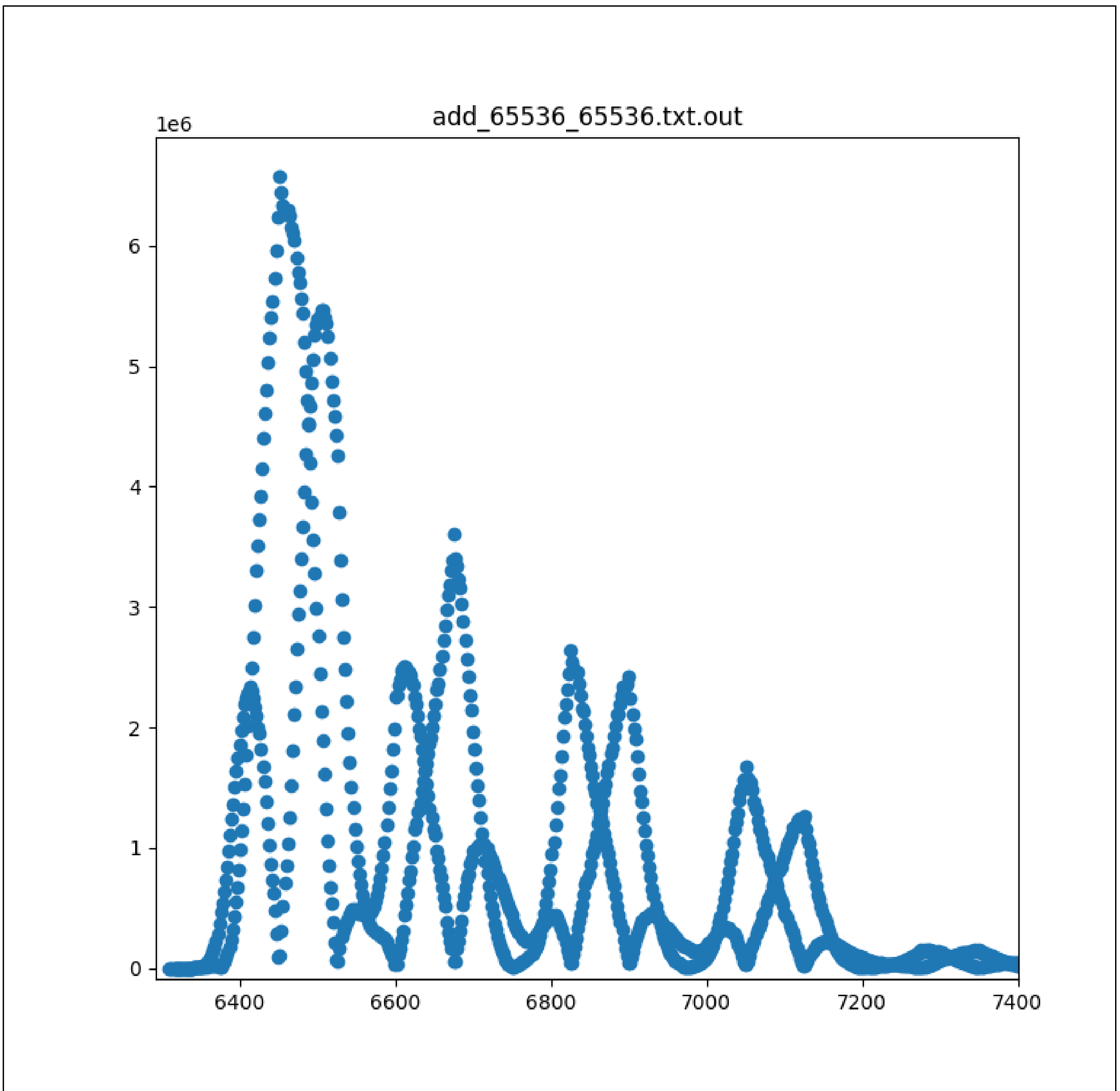

График скорости работы наивной реализации на Assembler при сложении двух чисел размером по 65536 шестнадцатеричных разрядов (32 кб), замена “loop” на аналог из более простых операторов

Как можно увидеть по графику, данная модификация быстрее версии с использованием “loop” приблизительно в 2.5 раза, а также приблизительно в 1.5 раза обгоняет GNU MP.

Эта реализация алгоритма является финальной на момент написания данной работы.

Умножение

Операция умножения является наиболее востребованной операцией со стороны пользователя библиотеки. Именно на ускорение данной операции должно уделяться особое внимание.

Основной алгоритм умножения, который будет использоваться в итоговой реализации – алгоритм умножения Карацубы. Однако, как известно данный алгоритм уступает по скорости алгоритму умножения “в столбик” на небольших числах.

Таким образом, основной алгоритм – умножение Карацубы, алгоритм для более маленьких чисел, а также алгоритм, применяемый в рекурсивном умножении Карацубы при переходе на умножение достаточно небольших чисел – умножение “в столбик”.

Тест GNU MP (512 байт)

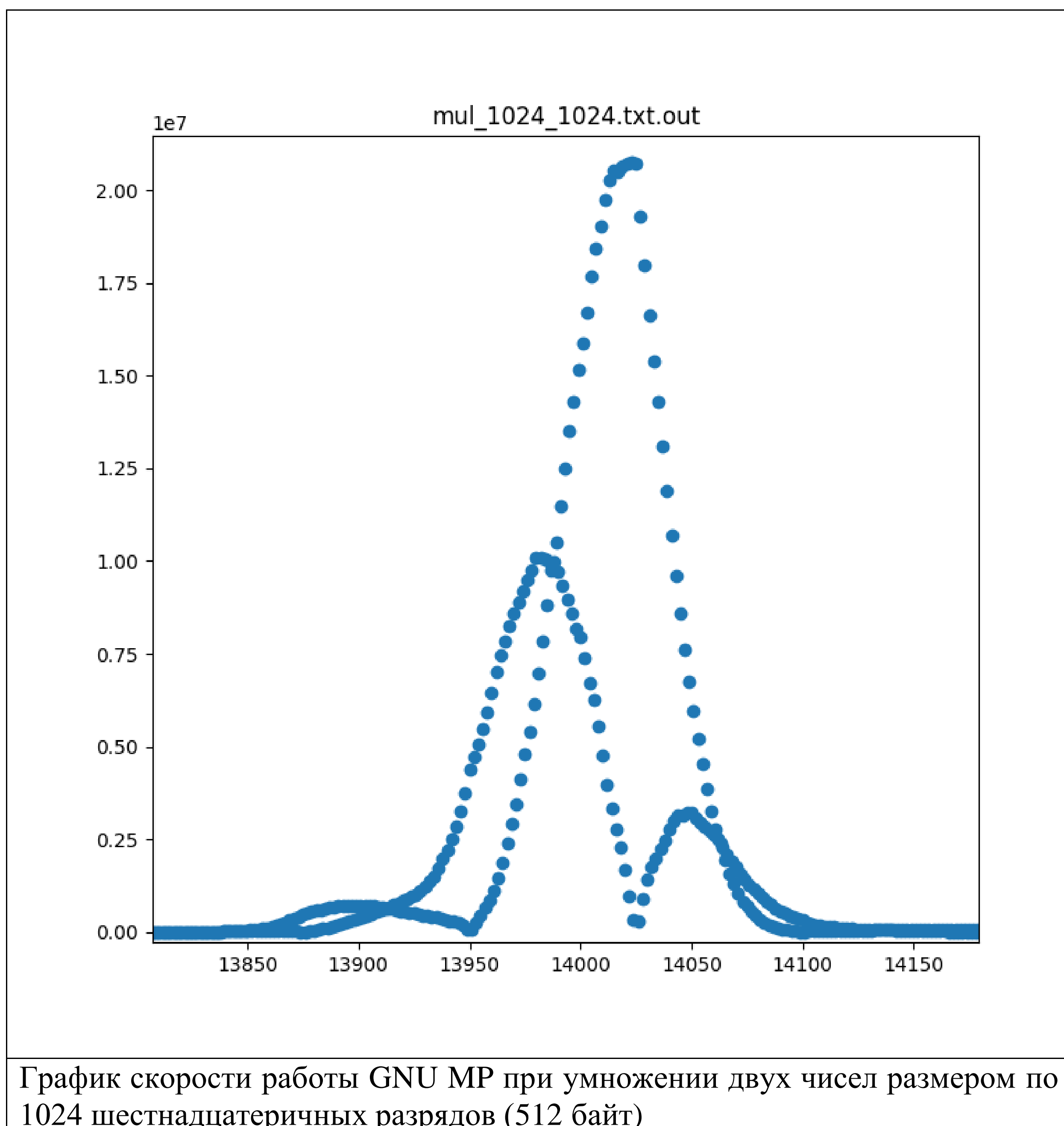

График скорости работы GNU MP при умножении двух чисел размером по 1024 шестнадцатеричных разрядов (512 байт)

Тест алгоритма “в столбик” на Assembler (512 байт)

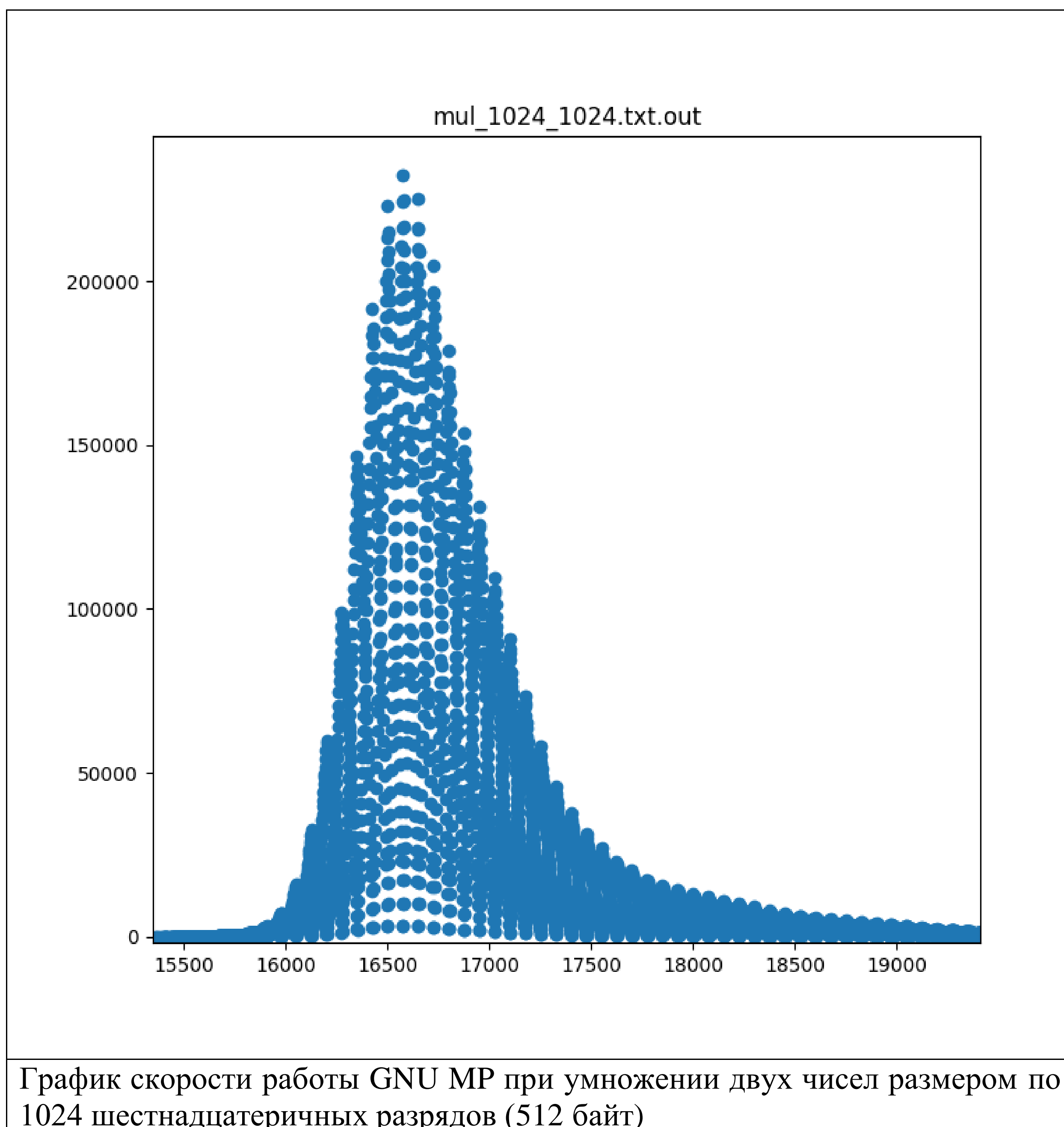

График скорости работы GNU MP при умножении двух чисел размером по 1024 шестнадцатеричных разрядов (512 байт)

Как видно из графиков, написанный алгоритм умножения “в столбик”, уступает по скорости GNU MP. Однако стоит отметить то, что GNU MP реализует алгоритм Карацубы, а отставание по скорости незначительно.

Так как написанный алгоритм использует целые числа размером в 8 байт в качестве разрядов длинного числа, можно вычислить что в данном тесте умножались 2 числа по 64 разряда. Это довольно большие числа для алгоритма сложностью $O(n^2)$.

Попробуем протестировать представленные алгоритмы на числах в 2 раза меньшего размера.

Тест GNU MP (256 байт)

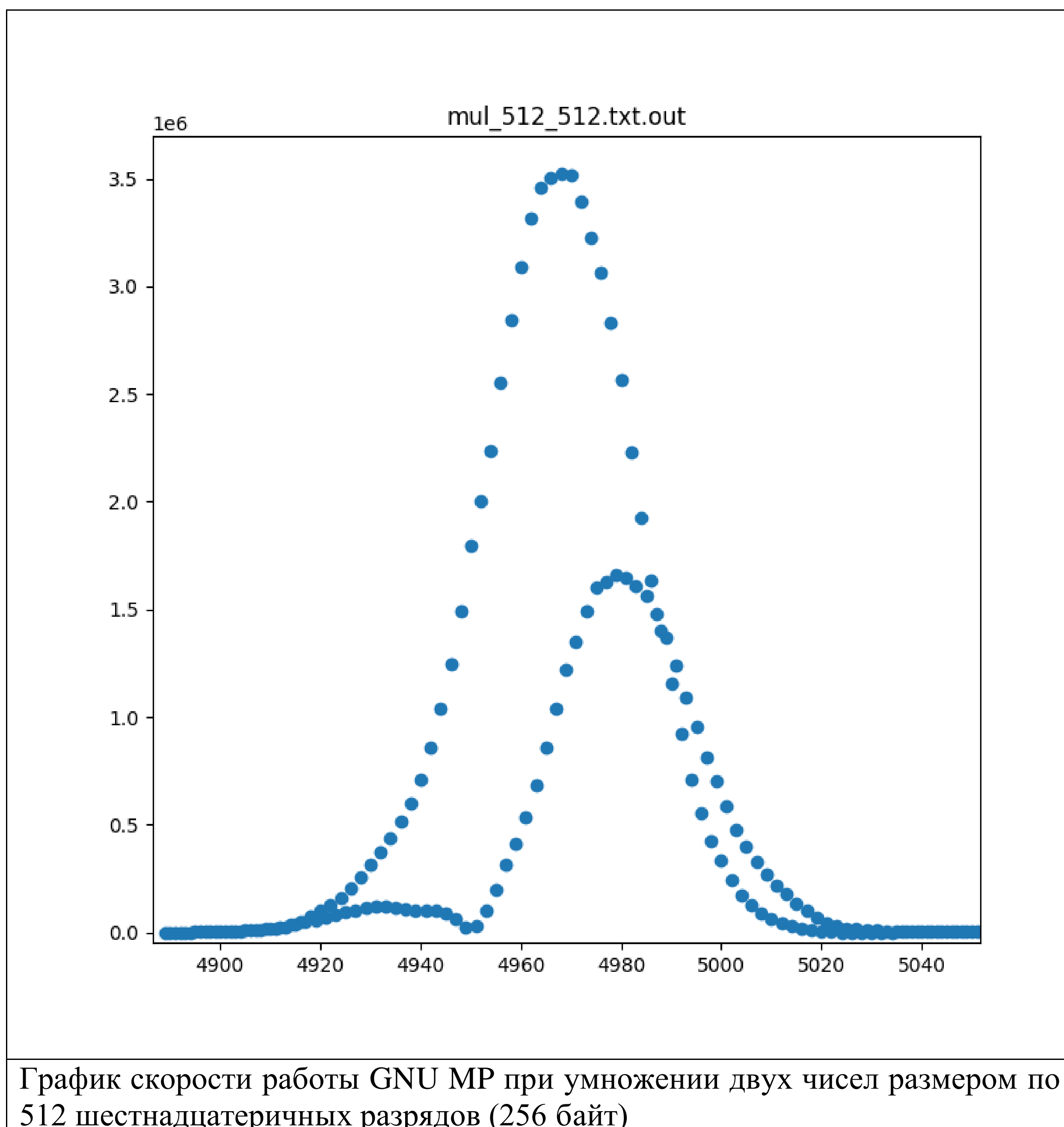

График скорости работы GNU MP при умножении двух чисел размером по 512 шестнадцатеричных разрядов (256 байт)

Тест алгоритма “в столбик” на Assembler (265 байт)

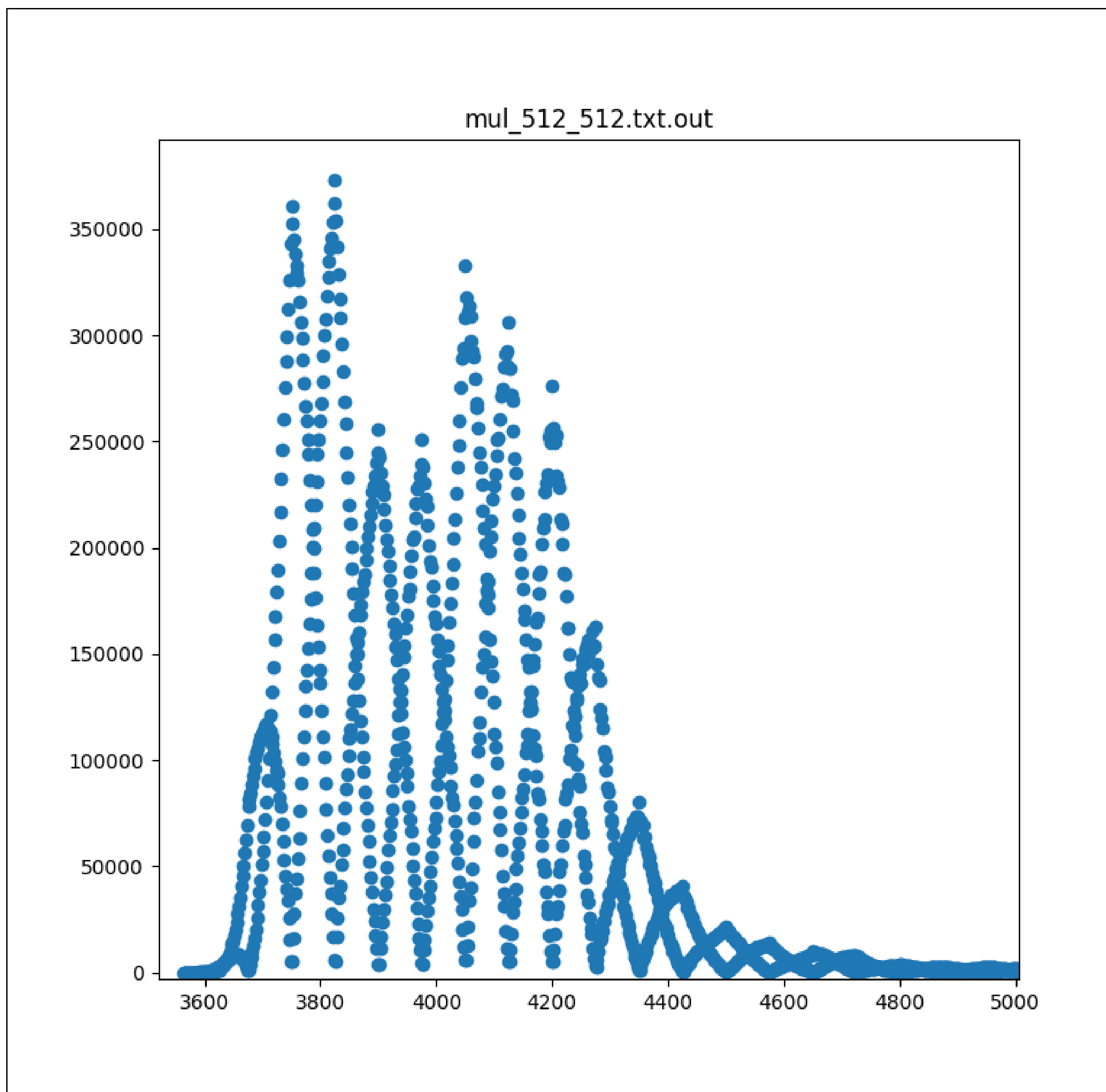

График скорости работы GNU MP при умножении двух чисел размером по 512 шестнадцатеричных разрядов (256 байт)

Как можно увидеть на графиках, написанная в ходе работы реализация алгоритма “в столбик” при умножении чисел, размером в 256 байт обгоняет GNU MP.

В дальнейшем при написании алгоритма Карацубы будут проведены тесты, для того что бы определить, где происходит обгон по скорости алгоритмом Карацубы алгоритма умножения “в столбик”. Для чисел соответствующего размера будут применены наиболее подходящие алгоритмы длинного умножения.

Умножение алгоритмом Карацубы

К сожалению, к моменту написания данной работы, алгоритм не получил своей реализации. Однако были проведены некоторые тесты, которые могут быть интересны в дальнейшем при его написании.

Как известно, алгоритм Карацубы рекурсивен и предполагает деление исходной задачи на несколько аналогичных подзадач. Для наибольшей скорости работы интересно знать, как будет осуществляться такое деление.

Для поверхностной оценки алгоритма деления на подзадачи был написан скрипт в стиле brute-force на python, который считает количество элементарных умножений для того или иного деления исходных длинных чисел. Явной зависимости вывести не удалось, но примерно 92% всех делений при умножении чисел от 1 до 3000 разрядов были связаны со степенью двойки. Алгоритм предпочитал выбирать либо 2^n старших разрядов большего из перемножаемых чисел, либо 2^n младших. Остальные 8% делений не были связаны со степенью двойки и встречались при умножении как чисел маленького размера, так и большого.

В дальнейшем при реализации умножения алгоритмом Карацубы должен быть так же реализован алгоритм выбора оптимального деления исходной задачи на подзадачи.

Вывод

В ходе работы была изучена скорость необходимых алгоритмов существующей и популярной реализации длинной арифметики.

Был выбран, частично реализован и протестирован на корректность и скорость функционал будущей библиотеки для осуществления быстрых операций длинной арифметики.

В работе представлено подробное сравнение скоростей алгоритмов.

Определены пути развития работы.

Список литературы

[1] GNU MP The GNU Multiple Precision Arithmetic Library Edition 6.0.0 25 March 2014: https://gmplib.org/gmp-man-6.0.0a.pdf

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual