ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)
ФАКУЛЬТЕТ АЭРОКОСМИЧЕСКИХ ТЕХНОЛОГИЙ

Лабораторная 6

Числа с плавающей точкой (Asm)

Рогозин Владимир
Группа Б03-106

Пункт 1: Инициализация нецелых чисел и операции с ними

Посмотрим как инициализируются типа *float* и *double*. В программе ниже создаются две локальные переменные – один *float* и один *double*. Видим, что при инициализации значения записываются в стек через специальный регистр %xmm0 и привычную команду *mov*, но со специальным суффиксом. Как видно из второй картинки, в ассемблерном листинге числа с плавающей точкой, так как являются всё тем же набором единиц и нулей, представляются в виде целых чисел, записанных в десятичной системе, двоичная запись которых соответствует виду, в котором хранится нецелое число в памяти (знак + порядок + мантисса). Отсюда становится ясно, почему для *float*'ов и *double*'ов нужны специальные команды – чтобы машина понимала как именно производить операции с битами. Так как размер чисел двойной точности 64 бита, *double* представляется двумя целыми числами (по 32 бита каждое).

```
movsd   .LC0(%rip), %xmm0
movsd   %xmm0, -8(%rbp)
movss   .LC1(%rip), %xmm0
movss   %xmm0, -12(%rbp)
```

(a) Инициализация *float* и *double*

```
        .align 8
.LC0:
        .long   2576980378
        .long   1074895257
        .align 4
.LC1:
        .long   1074245140
```

(b) В представлении ассемблера

Теперь арифметические операции с *float*'ами и *double*'ами. Ниже представлены две программы и их ассемблерные листинги. Можем видеть, что команды те же что и для целых чисел, но с другими суффиксами. К тому же сами числа хранятся в специальных 128-битных регистрах %xmm0, %xmm1 и т.д. Всего таких регистров 8 штук.

```
#include <stdio.h>

int main() {
        float a = 5.5;
        float b = 2.25;
        float c = a + b;
        float d = a - b;
        float e = a * b;
        float f = a / b;
        return 0;
}
```

(a) Арифм. операции *float*

```
#include <stdio.h>

int main() {
        double a = 5.5;
        double b = 2.25;
        double c = a + b;
        double d = a - b;
        double e = a * b;
        double f = a / b;
        return 0;
}
```

(b) Арифм. операции *double*

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movss   .LC0(%rip), %xmm0
        movss   %xmm0, -24(%rbp)
        movss   .LC1(%rip), %xmm0
        movss   %xmm0, -20(%rbp)
        movss   -24(%rbp), %xmm0
        addss   -20(%rbp), %xmm0
        movss   %xmm0, -16(%rbp)
        movss   -24(%rbp), %xmm0
        subss   -20(%rbp), %xmm0
        movss   %xmm0, -12(%rbp)
        movss   -24(%rbp), %xmm0
        mulss   -20(%rbp), %xmm0
        movss   %xmm0, -8(%rbp)
        movss   -24(%rbp), %xmm0
        divss   -20(%rbp), %xmm0
        movss   %xmm0, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(a) Листинг *float*

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movsd   .LC0(%rip), %xmm0
        movsd   %xmm0, -48(%rbp)
        movsd   .LC1(%rip), %xmm0
        movsd   %xmm0, -40(%rbp)
        movsd   -48(%rbp), %xmm0
        addsd   -40(%rbp), %xmm0
        movsd   %xmm0, -32(%rbp)
        movsd   -48(%rbp), %xmm0
        subsd   -40(%rbp), %xmm0
        movsd   %xmm0, -24(%rbp)
        movsd   -48(%rbp), %xmm0
        mulsd   -40(%rbp), %xmm0
        movsd   %xmm0, -16(%rbp)
        movsd   -48(%rbp), %xmm0
        divsd   -40(%rbp), %xmm0
        movsd   %xmm0, -8(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Листинг *double*

Пункт 2: Ускоренное среднее арифметическое элементов массива

Сначала на C++ напишем простейшую программку для вычисления среднего арифметического статического массива из *double*'ов. В цикле происходит суммирование элементов массива, после этого получившаяся сумма делится на количество элементов. Сразу посмотрим на оптимизацию O3, а точнее её листинг.

```
.L10:
    addsd   (%rsi), %xmm0
    addq    $16, %rsi
    addsd   -8(%rsi), %xmm0
    cmpq    %rdi, %rsi
    jne .L10
```

(a) Суммирование Asm

```
for (int i = 0; i < N; ++i) {
    aver += massive[i];
}
```

(b) Суммирование C++

В регистре %rsi изначально лежал адрес нулевого элемента массива. Видим, что, благодаря оптимизации, за одну итерацию происходит два сложения (компилятор сам устроил конвейер операций), но происходят они по очереди, а не одной командой. Попробуем, с помощью ассемблерной вставки, используя распараллеливание сложения при помощи SSE ещё ускорить программу. Будем складывать два числа за одну операцию, по окончании цикла получим две суммы, сложим их и получим искомую сумму. Ассемблерная вставка представлена ниже на картинке.

После этого сравним время работы программ для массивов 16,..., 16 000 000 элементов, будем находить среднее арифметическое 20 раз, каждый раз генерировать новый массив. Результаты представлены на картинках ниже.

```
asm
(
    "leaq   massive(%rip), %r11\n" // Адрес нулевого элемента изначально кладём в %r11
    "movq   $15998, %rbx\n"
    "leaq   (%r11, %rbx, 8), %r12\n" // Адрес предпоследнего элемента (для выхода из цикла)
    "movq   $2, %rbx\n"
    "movupd massive(%rip), %xmm0\n" // Перед циклом кладём первые два double'а в %xmm0,
                                    //в %xmm0 будут копиться две суммы
    "Cycle:\n"
    "addpd  (%r11, %rbx, 8), %xmm0\n" // Складываем по два элемента за одну операцию
    "leaq   (%r11, %rbx, 8), %r11\n" // Теперь %r11 указывает на элемент через один
    "cmp    %r11, %r12\n"
    "jne    Cycle\n"
    "movhpd %xmm0, aver(%rip)\n" // Отделяем первую сумму
    "movsd  aver(%rip), %xmm1\n"
    "addsd  %xmm0, %xmm1\n" // Складываем обе суммы
    "movsd  %xmm1, aver(%rip)\n"
);
```

Рис. 5: Ассемблерная вставка для ускорения работы кода

```
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 0 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 0 ns
```

(a) $N = 16$, оптимизация O3

```
Time consumed: 200 ns
Time consumed: 0 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 0 ns
Time consumed: 100 ns
Time consumed: 100 ns
```

(b) $N = 16$, ассемблерная вставка

```
Time consumed: 400 ns
Time consumed: 100 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 400 ns
Time consumed: 100 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 100 ns
Time consumed: 300 ns
Time consumed: 100 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 200 ns
Time consumed: 100 ns
```

(a) $N = 160$, оптимизация O3

```
Time consumed: 200 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 200 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
Time consumed: 100 ns
```

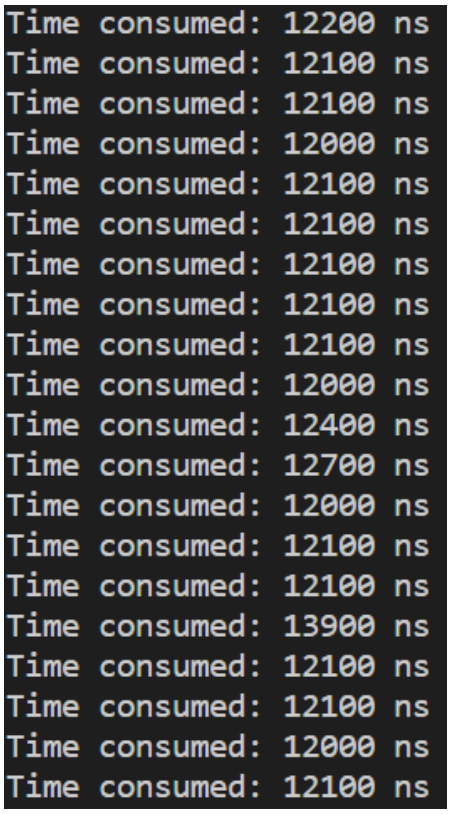
(b) $N = 160$, ассемблерная вставка

```
Time consumed: 1300 ns
Time consumed: 1200 ns
Time consumed: 1200 ns
Time consumed: 1200 ns
Time consumed: 1200 ns
Time consumed: 1200 ns
Time consumed: 1200 ns
Time consumed: 1300 ns
Time consumed: 1300 ns
Time consumed: 1300 ns
Time consumed: 1200 ns
Time consumed: 2800 ns
Time consumed: 1200 ns
Time consumed: 1300 ns
Time consumed: 1300 ns
Time consumed: 1200 ns
Time consumed: 1200 ns
Time consumed: 1300 ns
Time consumed: 1200 ns
Time consumed: 1300 ns
```

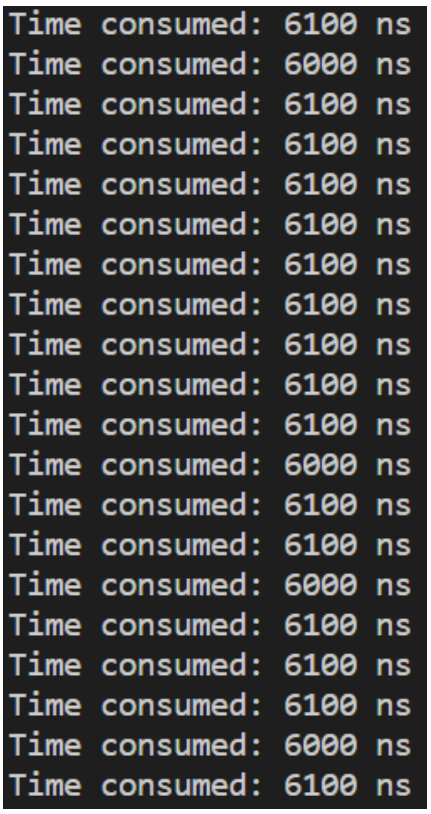
(a) $N = 1600$, оптимизация O3

```
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 600 ns
Time consumed: 600 ns
Time consumed: 600 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 600 ns
Time consumed: 700 ns
Time consumed: 600 ns
Time consumed: 900 ns
Time consumed: 700 ns
Time consumed: 700 ns
Time consumed: 600 ns
Time consumed: 700 ns
```

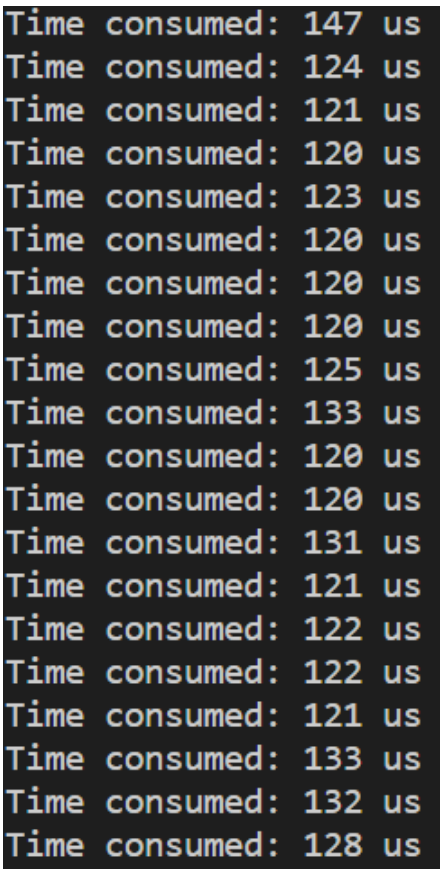
(b) $N = 1600$, ассемблерная вставка



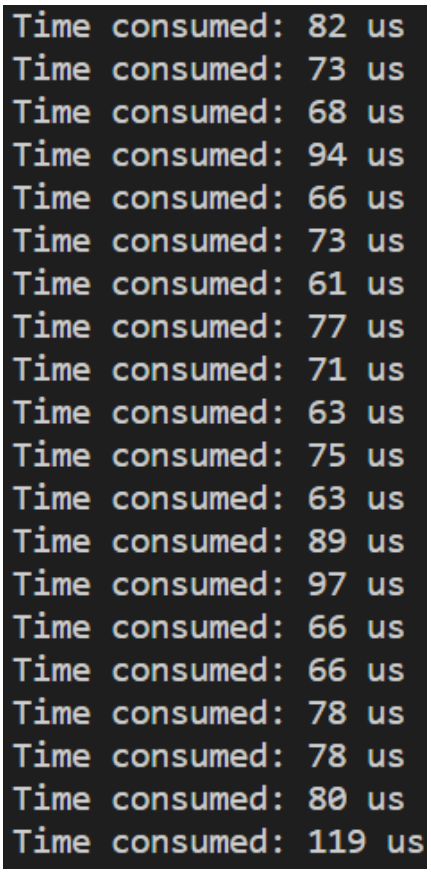
(a) $N = 16000$, оптимизация O3



(b) $N = 16000$, ассемблерная вставка



(a) $N = 160000$, оптимизация O3



(b) $N = 160000$, ассемблерная вставка

```
Time consumed: 1350 us
Time consumed: 1329 us
Time consumed: 1426 us
Time consumed: 1323 us
Time consumed: 1302 us
Time consumed: 1436 us
Time consumed: 1479 us
Time consumed: 1327 us
Time consumed: 1405 us
Time consumed: 1288 us
Time consumed: 1310 us
Time consumed: 1424 us
Time consumed: 1424 us
Time consumed: 1286 us
Time consumed: 1444 us
Time consumed: 1297 us
Time consumed: 1335 us
Time consumed: 1307 us
Time consumed: 1445 us
Time consumed: 1438 us
```

(a) $N = 1600000$, оптимизация O3

```
Time consumed: 1121 us
Time consumed: 907 us
Time consumed: 1000 us
Time consumed: 1315 us
Time consumed: 1099 us
Time consumed: 932 us
Time consumed: 1098 us
Time consumed: 964 us
Time consumed: 1248 us
Time consumed: 974 us
Time consumed: 1060 us
Time consumed: 1047 us
Time consumed: 1029 us
Time consumed: 947 us
Time consumed: 945 us
Time consumed: 957 us
Time consumed: 1074 us
Time consumed: 935 us
Time consumed: 944 us
Time consumed: 1052 us
```

(b) $N = 1600000$, ассемблерная вставка

```
Time consumed: 13358 us
Time consumed: 12804 us
Time consumed: 12728 us
Time consumed: 12695 us
Time consumed: 13112 us
Time consumed: 12773 us
Time consumed: 12735 us
Time consumed: 13651 us
Time consumed: 12698 us
Time consumed: 13360 us
Time consumed: 12712 us
Time consumed: 12702 us
Time consumed: 12772 us
Time consumed: 12706 us
Time consumed: 12651 us
Time consumed: 12850 us
Time consumed: 12719 us
Time consumed: 12659 us
Time consumed: 12721 us
Time consumed: 12727 us
```

(a) $N = 16000000$, оптимизация O3

```
Time consumed: 8181 us
Time consumed: 8112 us
Time consumed: 7935 us
Time consumed: 8250 us
Time consumed: 8127 us
Time consumed: 11204 us
Time consumed: 10133 us
Time consumed: 8802 us
Time consumed: 9401 us
Time consumed: 10055 us
Time consumed: 8678 us
Time consumed: 8072 us
Time consumed: 8207 us
Time consumed: 8141 us
Time consumed: 8157 us
Time consumed: 8329 us
Time consumed: 8081 us
Time consumed: 8154 us
Time consumed: 8100 us
Time consumed: 7994 us
```

(b) $N = 16000000$, ассемблерная вставка

Занесём результаты в таблицу.

Таблица 1: Результаты измерений часть 1

	$N = 16$		$N = 160$		$N = 1600$		$N = 16 \cdot 10^3$		$N = 16 \cdot 10^4$	
№	О3, нс	SSE, нс	О3, нс	SSE, нс	О3, нс	SSE, нс	О3, нс	SSE, нс	О3, мкс	SSE, мкс
1	100	200	400	200	1300	700	12200	6100	147	82
2	100	0	100	100	1200	700	12100	6000	124	73
3	0	0	200	100	1200	700	12100	6100	121	68
4	0	100	200	100	1200	700	12000	6100	120	94
5	0	100	200	100	1200	700	12100	6100	123	66
6	100	100	100	200	1200	700	12100	6100	120	73
7	0	100	100	100	1200	600	12100	6100	120	61
8	100	100	400	100	1300	600	12100	6100	120	77
9	100	100	100	100	1300	600	12100	6100	125	71

	О3, нс	SSE, нс	О3, нс	SSE, нс	О3, нс	SSE, нс	О3, нс	SSE, нс	О3, мкс	SSE, мкс
10	0	100	200	100	1300	700	12000	6100	133	63
11	100	100	200	100	1200	700	12400	6100	120	75
12	0	100	200	100	2800	700	12700	6000	120	63
13	100	100	200	100	1200	600	12000	6100	131	89
14	100	100	100	100	1300	700	12100	6100	121	97
15	100	100	300	100	1300	600	12100	6000	122	66
16	0	100	100	100	1200	900	13900	6100	122	66
17	100	100	200	100	1200	700	12100	6100	121	78
18	0	0	200	100	1300	700	12100	6100	133	78
19	100	100	200	100	1200	600	12000	6000	132	80
20	0	100	100	100	1300	700	12100	6100	128	119

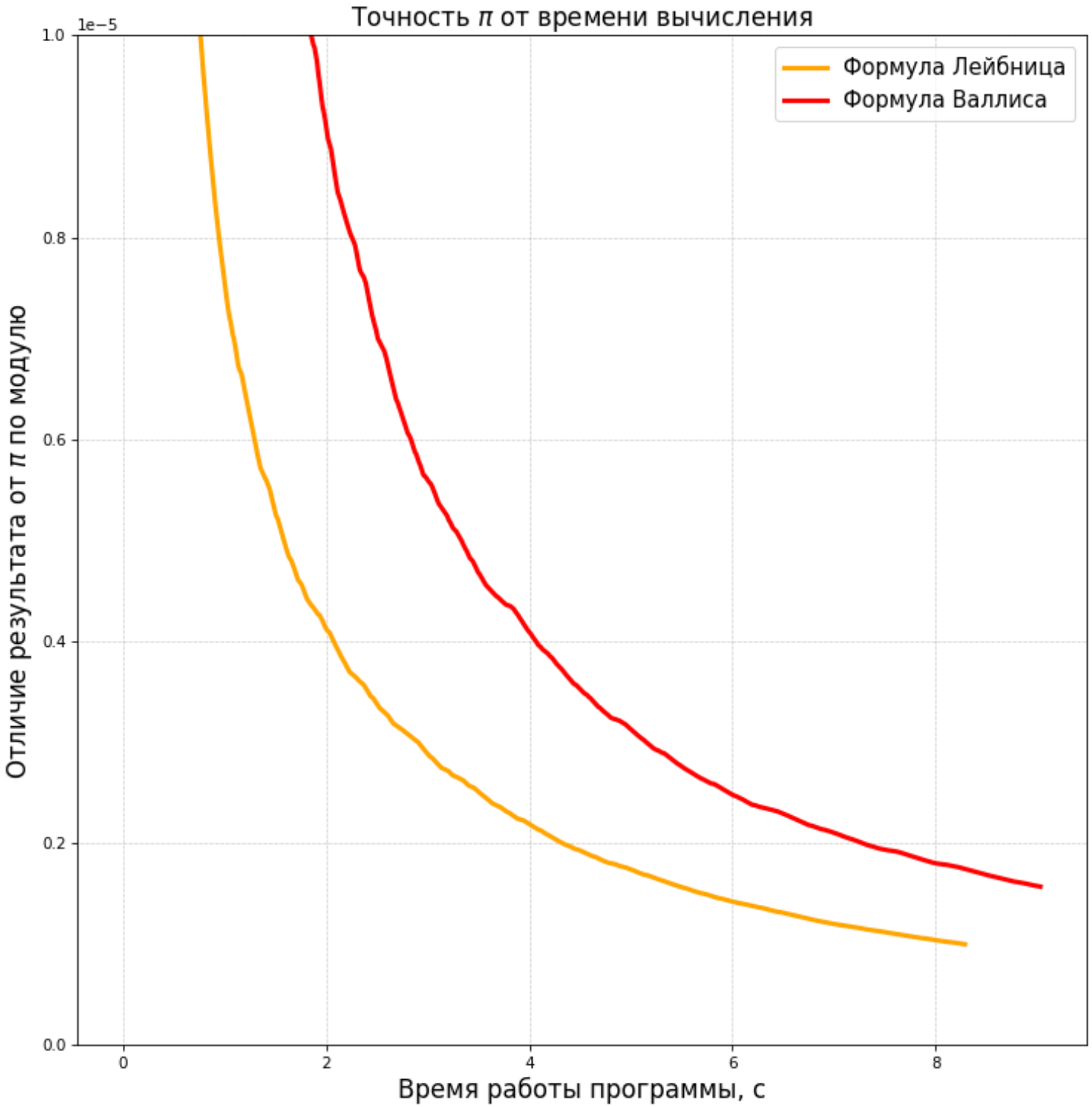
Таблица 2: Результаты измерений часть 2

	$N = 16 \cdot 10^5$		$N = 16 \cdot 10^6$			$N = 16 \cdot 10^5$		$N = 16 \cdot 10^6$	
№	О3, мкс	SSE, мкс	О3, мкс	SSE, мкс	№	О3, мкс	SSE, мкс	О3, мкс	SSE, мкс
1	1350	1121	13358	8181	11	1310	1060	12712	8678
2	1329	907	12804	8112	12	1424	1047	12702	8072
3	1426	1000	12728	7935	13	1424	1029	12772	8207
4	1323	1315	12695	8250	14	1286	947	12706	8141
5	1302	1099	13112	8127	15	1444	945	12651	8157
6	1436	932	12773	11204	16	1297	957	12850	8329
7	1479	1098	12735	10133	17	1335	1074	12719	8081
8	1327	964	13651	8802	18	1307	935	12659	8154
9	1405	1248	12698	9401	19	1445	944	12721	8100
10	1288	974	13360	10055	20	1438	1052	12727	7994

При достаточно больших размерах массива ($N > 1000$) программа с ассемблерной вставкой работает заметно быстрее, при малых N обе работают примерно одинаково.

Пункт 3: Среднее время работы вычисления знака числа π

В этом пункте посмотрим на точность вычисления числа π от времени работы программы. Формулы 3 и 4 из предыдущей лабораторной почти сразу же дают точность порядка $\sim 10^{-16}$ (с 25-ой и 30-ой итерации соответственно), формула 5 даёт неверный результат с определённого момента, поэтому будем строить графики только для первых двух (Формула Лейбница и Формула Валлиса).



Из графика видно, что порядка 8-ми секунд требуется первой формуле для получения точности порядка $\sim 10^{-6}$ и 9-ти секунд требуется второй.

Пункт 4: Ускорение вычисления среднего арифметического с помощью конвейера операций на C++

Теперь попробуем ускорить процесс нахождения среднего арифметического, немного иначе суммируя элементы множества. Так как инструкции для основного процессора и сопроцессора могут выполняться одновременно, то есть смысл попытаться сделать так, чтобы за одну итерацию математический сопроцессор выполнял столько же действий, сколько и основной. При таком раскладе потребуется меньше итераций, и при этом время, затрачиваемое на одной итерации, не увеличится, потому что основному процессору все равно потребуется сделать столько же операций за одну итерацию. Будем применять такую оптимизацию к задаче о нахождении среднего арифметического массива элементов.

```
Time consumed: 49710 us
Time consumed: 46748 us
Time consumed: 45852 us
Time consumed: 45507 us
Time consumed: 45991 us
Time consumed: 46288 us
Time consumed: 46746 us
Time consumed: 50287 us
Time consumed: 46220 us
Time consumed: 45461 us
Time consumed: 46486 us
Time consumed: 47019 us
Time consumed: 45573 us
Time consumed: 46470 us
Time consumed: 45190 us
Time consumed: 45320 us
Time consumed: 46145 us
Time consumed: 45376 us
Time consumed: 46517 us
Time consumed: 45265 us
```

(a) Одна сумма

```
Time consumed: 23153 us
Time consumed: 23285 us
Time consumed: 24119 us
Time consumed: 23551 us
Time consumed: 23291 us
Time consumed: 23513 us
Time consumed: 23279 us
Time consumed: 23274 us
Time consumed: 23088 us
Time consumed: 23334 us
Time consumed: 23253 us
Time consumed: 23151 us
Time consumed: 23634 us
Time consumed: 24208 us
Time consumed: 23425 us
Time consumed: 24727 us
Time consumed: 23101 us
Time consumed: 23287 us
Time consumed: 23377 us
Time consumed: 23373 us
```

(b) Две суммы

```
Time consumed: 16836 us
Time consumed: 15899 us
Time consumed: 15976 us
Time consumed: 19434 us
Time consumed: 15772 us
Time consumed: 15985 us
Time consumed: 16712 us
Time consumed: 16081 us
Time consumed: 16169 us
Time consumed: 17391 us
Time consumed: 17516 us
Time consumed: 16565 us
Time consumed: 16159 us
Time consumed: 16123 us
Time consumed: 15751 us
Time consumed: 15621 us
Time consumed: 15682 us
Time consumed: 15745 us
Time consumed: 15829 us
Time consumed: 15678 us
```

(a) Три суммы

```
Time consumed: 14750 us
Time consumed: 16454 us
Time consumed: 17572 us
Time consumed: 16901 us
Time consumed: 15349 us
Time consumed: 15347 us
Time consumed: 16037 us
Time consumed: 14641 us
Time consumed: 14649 us
Time consumed: 14777 us
Time consumed: 14366 us
Time consumed: 14771 us
Time consumed: 14415 us
Time consumed: 14394 us
Time consumed: 14363 us
Time consumed: 14398 us
Time consumed: 14405 us
Time consumed: 14398 us
Time consumed: 14443 us
Time consumed: 14449 us
```

(b) Четыре суммы

```
Time consumed: 14787 us
Time consumed: 18109 us
Time consumed: 14600 us
Time consumed: 15200 us
Time consumed: 14320 us
Time consumed: 14548 us
Time consumed: 14546 us
Time consumed: 15226 us
Time consumed: 14635 us
Time consumed: 14080 us
Time consumed: 14540 us
Time consumed: 14526 us
Time consumed: 14328 us
Time consumed: 14464 us
Time consumed: 14093 us
Time consumed: 14518 us
Time consumed: 18588 us
Time consumed: 18556 us
Time consumed: 15514 us
Time consumed: 15200 us
```

(a) Пять сумм

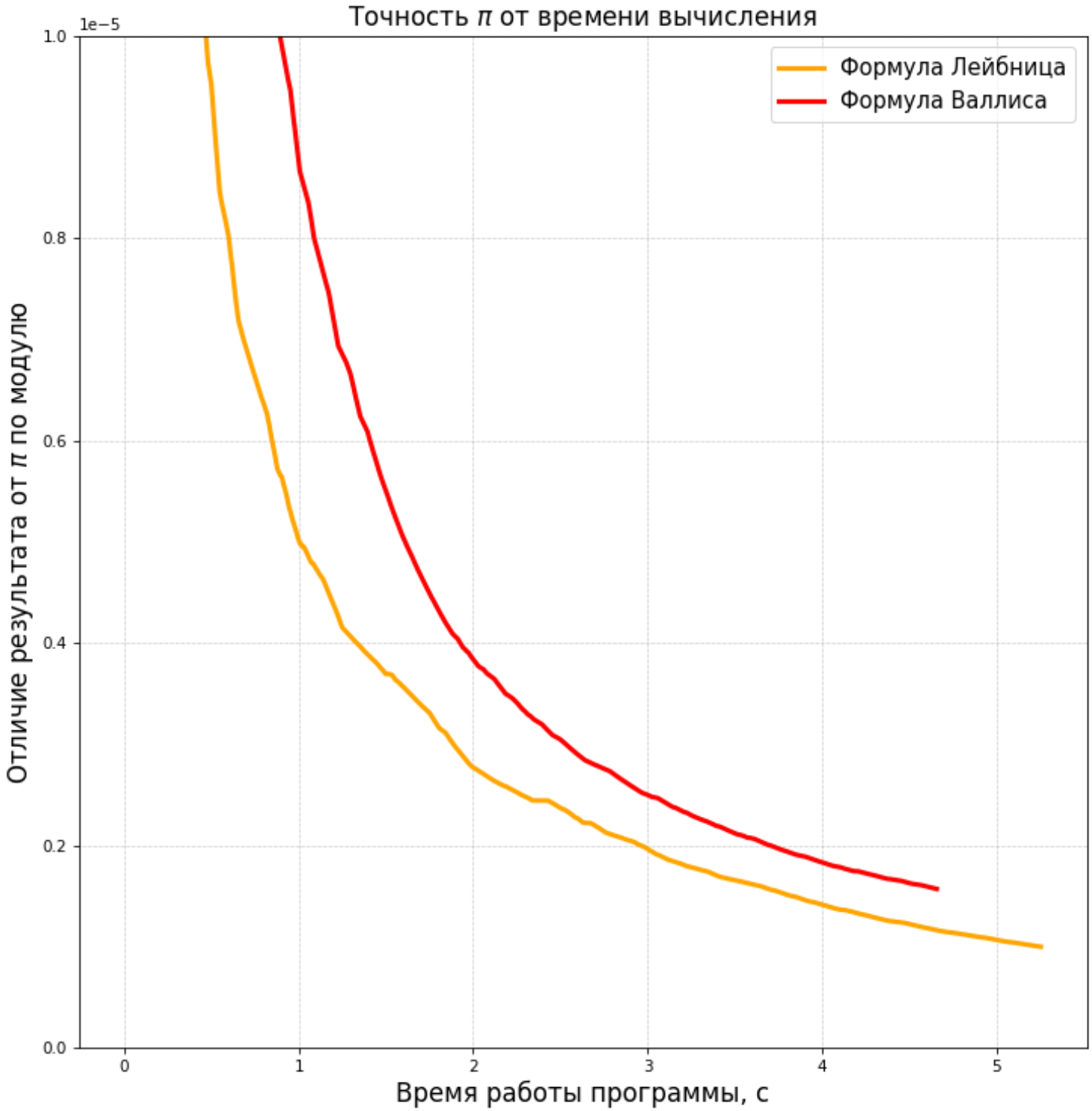
```
Time consumed: 15819 us
Time consumed: 14684 us
Time consumed: 14236 us
Time consumed: 14136 us
Time consumed: 14324 us
Time consumed: 13935 us
Time consumed: 13896 us
Time consumed: 19616 us
Time consumed: 14380 us
Time consumed: 14075 us
Time consumed: 14733 us
Time consumed: 14165 us
Time consumed: 14330 us
Time consumed: 14278 us
Time consumed: 15600 us
Time consumed: 13943 us
Time consumed: 14361 us
Time consumed: 14225 us
Time consumed: 15230 us
Time consumed: 14273 us
```

(b) Шесть сумм

До определённого момента создавать суммы выгодно, затем время работы практически не меняется.

Пункт 5:

Теперь попробуем ускорить вычисление числа π из пункта 3 с помощью конвейера операций и отключения денормализованных чисел. Сделаем две суммы и два произведения в случае первой и второй формул соответственно. Ниже представлен получившийся график.



Как видно, время работы заметно уменьшилось с 8-ми до примерно 5-ти секунд для заданной точности $\sim 10^{-6}$, и это только при двух суммах/произведениях, вероятно, вычисление можно ещё ускорить, подобрав оптимальное количество операций сложения/умножения за одну итерацию. Одно только отключение денормализованных чисел ускоряет вычисление на $\sim 0{,}3$ секунды, весь остальной эффект достигается с помощью конвейера операций.