Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

Кафедра физики открытых систем
наименование кафедры

Реализация алгоритмов блочного шифрования на языке Python с использованием архитектуры CUDA

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

Студента 4 курса 431 группы

направления 09.03.02 «Информационные системы и технологии»
код и наименование направления

факультета нелинейных процессов
наименование факультета

Лепаева Антона Александровича
фамилия, имя, отчество

Научный руководитель
доцент, д.ф.-м.н., доцент _____ С.А. Куркин
должность, уч. степень, уч. звание дата, подпись инициалы, фамилия

Зав. кафедрой:
д.ф.-м.н., профессор _____ А.А. Короновский
должность, уч. степень, уч. звание дата, подпись инициалы, фамилия

Саратов 2018 г.

Содержание

Введение

Шифрование данных на данный момент является весьма актуальной проблемой. Через интернет передаются огромные объемы данных - начиная от простых сообщений в IRC чатах и заканчивая денежными транзакциями и приватными данными пользователей. Так как эти данные в сети Интернет могут быть перехвачены злоумышленником, необходимо защитить их. Для этого были разработаны и продолжают разрабатываются различные методы и алгоритмы, помогающие скрыть истинное содержимое данных. В данной работе будет рассматриваться блочное шифрование, в частности алгоритм RC5.

Будет также произведено сравнение времени выполнения последовательной и параллельной программы на разных объёмах данных. Рассмотрим параллелизацию кода при помощи фреймворка CUDA и его дальнейшую оптимизацию.

1. Алгоритм RC5

В криптографии RC5 представляет собой блок-шифр с симметричным ключом, известный своей простотой. Разработанный Рональдом Ривестом в 1994 году, RC означает «Rivest Cipher». Кандидат RC6 Advanced Encryption Standard (AES) был основан на RC5.

В отличие от многих схем, RC5 имеет переменный размер блока (32, 64 или 128 бит), размер ключа (от 0 до 2040 бит) и количество раундов (от 0 до 255). Первоначальным предлагаемым выбором параметров был размер блока 64 бит, 128-битный ключ и 12 раундов.

Ключевой особенностью RC5 является использование зависящих от данных ротаций. RC5 также состоит из нескольких сложений по модулю и операции исключающего ИЛИ (XOR). Общая структура алгоритма - это сеть, подобная сети Фейстеля. Процедуры шифрования и дешифрования могут быть реализованы всего в нескольких строках кода.

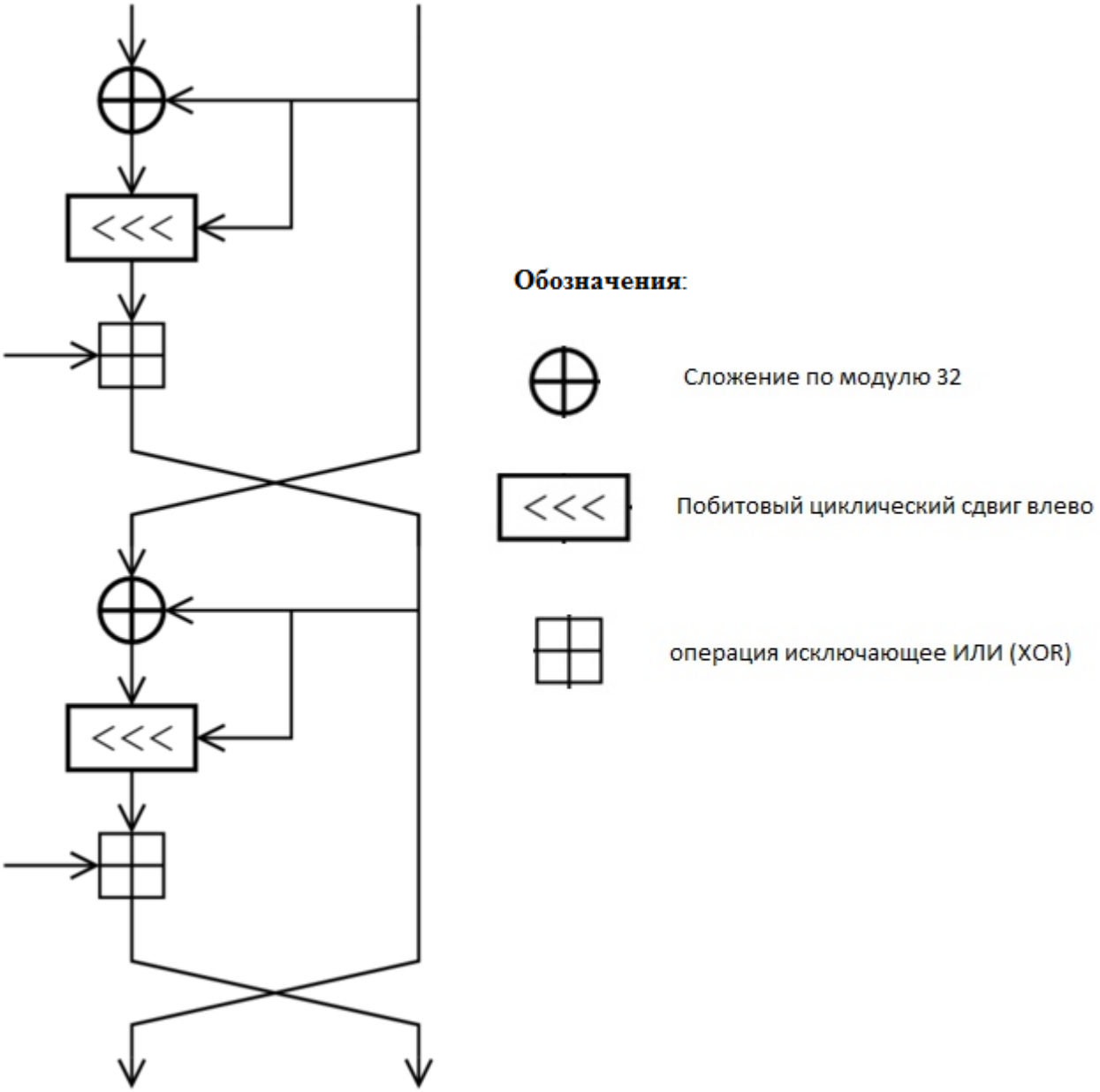

Рисунок 1 - Блок-схема одного раунда в RC5

Алгоритм состоит из трех функций: расширение ключа, шифровка, дешифровка. Проще всего представить их в виде псевдокода, взятого из оригинальной статьи Ривеста (комментарии переведены) [2]:

```
# Функция расширения ключа:
# Разбиваем ключ K на слова
# u = w / 8
c = ceiling( max(b, 1) / u )
# L изначально список длиной c заполненый нулями длины w бит
```

```
for i = b-1 down to 0 do:
   L[i/u] = (L[i/u] << 8) + K[i]
# Инициализируем независимый от ключа псевдослучайный массив S
# S изначально лист неопределенных слов (длины w) длины  t=2(r+1)
S[0] = P
for i = 1 to t-1 do:
   S[i] = S[i-1] + Q

# Основной цикл выдачи ключа
i = j = 0
A = B = 0
do 3 * max(t, c) times:
   A = S[i] = (S[i] + A + B) <<< 3
   B = L[j] = (L[j] + A + B) <<< (A + B)
   i = (i + 1) % t
   j = (j + 1) % c
```

Здесь (для w = 32):

$P = Odd((e - 2) * 2^w) = 0xB7E15163$

$Q = Odd((\phi - 2)*2^w) = 0x9E3779B9$

- «магические» константы. Функция Odd(x) возвращает ближайшее к *x* нечетное число.

```
# Функция шифровки,
# «<<<» - циклический побитовый сдвиг влево
# «^» - побитовое исключающее ИЛИ
A = A + S[0]
B = B + S[1]
for i = 1 to r do:
   A = ((A ^ B) <<< B) + S[2 * i]
   B = ((B ^ A) <<< A) + S[2 * i + 1]
```

```
return A, B
```

Здесь r – количество раундов. Рекомендуется: 12 – 20.

```
# Функция дешифровки. Прямолинейное обращение функции шифровки
for i = r down to 1 do:
   B = ((B - S[2 * i + 1]) >>> A) ^ A
   A = ((A - S[2 * i]) >>> B) ^ B
B = B - S[1]
A = A - S[0]
return A, B
```

2. Режимы блочного шифрования

Блок-шифр сам по себе позволяет скрытую запись только одного блока известной длины. Для сообщения переменной длины информация должна быть разделена на отдельные блоки шифрования. В простейшем случае, называемом режимом электронной кодовой книги (ECB), сообщение разбивается на отдельные блоки размера блока шифрования (с возможным расширением последнего блока при помощи амортизирующих битов), и каждый блок зашифровывается и дешифруется по отдельности (к нему применяется схема, изображенная на рис. 2).

Однако такая "наивная" методология независимой шифровки блоков в целом небезопасна – одинаковые блоки открытого текста могут генерировать одинаковые блоки зашифрованного текста (для аналогичного ключа), поэтому паттерны внутри сообщения открытого текста становятся очевидными в выводе зашифрованного текста.

Для преодоления этого ограничения разработано множество режимов блочного шифрования [3]. Режим представляет из себя некий метод связывания блоков между собой, а также с некоторым случайным блоком, называемым вектором инициализации. Вектор инициализации обязательно должен быть случайным или псевдослучайным, чтобы не позволить злоумышленнику вывести соотношения между сегментами шифротекста, применяя схему шифрования повторно с одним и тем же ключом.

Далее рассмотрим 4 самых основных режима блочного шифрования: ECB (режим электронной кодовой книги), CBC (режим сцепления блоков шифротекста), PCBC (режим распространяющегося сцепления блоков шифра) и CFB (режим обратной связи по шифротексту).

2.1. Режим электронной кодовой книги (ECB)

Самый простой режим. В нем исходный текст разбивается на блоки, которые затем шифруются независимо друг от друга (рис. 2).

Недостатком этого метода является то, что идентичные блоки исходного текста зашифрованы в одинаковые блоки зашифрованного текста; таким образом, он не скрывает закономерности данных. В некотором смысле он не обеспечивает конфиденциальность сообщений, и его не рекомендуется использовать в криптографических протоколах вообще.

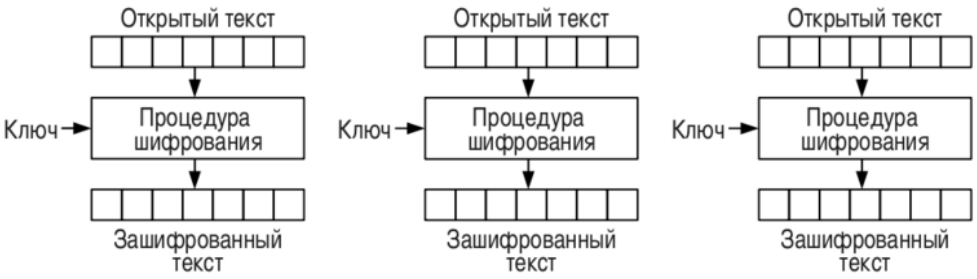

Рисунок 2 - Блок-схема режима ECB

3. Библиотека PyCUDA

PyCUDA позволяет получить доступ к API параллельных вычислений Nidia CUDA с Python, что предоставляет ряд преимуществ, таких как:

- Очистка объекта привязана к времени жизни объекта. Эта идиома, часто называемая RAII в C ++, значительно упрощает запись кода без утечек и сбоев. Также, PyCUDA отслеживает за зависимостями.
- Абстракции, такие как pycuda.driver.SourceModule и pycuda.gpuarray.GPUArray, делают программирование CUDA более удобным.
- PyCUDA предоставляет всю полноту API драйверов CUDA.
- Все ошибки CUDA автоматически преобразуются в исключения Python.
- Основной уровень PyCUDA написан на C ++, что обеспечивает высокую производительность.

В данной работе мы будем использовать библиотеку PyCUDA для того чтобы задействовать вычислительные мощности видеокарты.

4. Практическая часть

В рамках данной работы была реализована программа на языке Python (версия 3.6) с применением фреймворка NumPy для осуществления математических операций с числами фиксированной длины в битах. Программа осуществляет шифрование/дешифрование файлов произвольного размера, произвольного содержания (двоичные) при помощи алгоритма RC5. Из режимов блочного шифрования реализованы: ECB, CBC, PCBC, CFB. Программа легко может быть расширена для применения практически любого алгоритма шифрования: нужно просто передать соответствующие объекты функций в основную функцию *process* (рис. 3). Полный исходный код программы находится в приложении А.

```
def process(block_fn, opmode_fn, expand_fn, key, filename_in,
filename_out, block_struct="2I", strip_bytes=0,
            IV=None):
    """ block_fn: функция, [де]шифрующая отдельно взятый блок,
например rc5_encrypt/rc5_decrypt
        opmode_fn: функция, реализующая алгоритм блочного шифрования
        expand_fn: функция для расширения ключа. Если None, то ключ
не расширяется
        key: ключ
        filename_in: имя входного файла
        filename_out: имя выходного файла
        block_struct = "2I": Структура одного блока, подаваемого на
вход алгоритму.
            2 беззнаковых 4-байтовых целых числа по умолчанию
        strip_bytes = 0: сколько байт отбросить при записи выходного
файла. 0 по умолчанию
        IV: вектор инициализации
    """
```

Рисунок 3 - Сигнатура и описание параметров функции *process*

Далее приведен вывод программы (режим CFB). Для обозначения непечатных ASCII символов в Python применяется нотация \x{код символа}, код символа – шестнадцатиричное число 0 - 255:

1. Текстовый файл

```
=== ДО шифрования:
Привет, мир! Hello, world!
=== После шифрования:
6  байт добавлено
b'k\xfb.m\nS\x8c\x0e\xd9\x19[\xfaK\xe1\xd7@_z>4|\x1e\xf7o\x8e.h3\xd9K!\x88'
=== После дешифрования:
Привет, мир! Hello, world!
```

2. Изображение JPEG (выводятся первые 64 байта файла). Заметно, что сигнатура файла (“JFIF”) восстановлена после дешифровки.

```
=== ДО шифрования:
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x02\x00\x00d\x00d\x00\x00\xff\xec\x00\x11Duck
y\x00\x01\x00\x04\x00\x00\x00P\x00\x00\xff\xee\x00\x0eAdobe\x00d\xc0\x00\x00\x00\x01
\xff\xdb\x00\x84\x00\x02\x02\x02\x02'... еще 19260 байт
=== После шифрования:
4  байт добавлено
b'\xdd\xd9\xa7\xac\x80s\n\xbb\xfbX\xc4\x9e`c\xbf<7\x93\xdf\xb9T\x00\x9a\x8f\xaa@\x0e
3\xd0\xe1f\xb0E\xb7\x1c\xfd\xea\xa1\xe4\xb2T<\xb6\x81\xd5d\xd9+\xe0\xe1L\x95a\x9b3\x
e9F\xa1\xd6\xaa\xb2\xfa\x04\x02'... еще 19264 байт
=== После дешифрования:
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x02\x00\x00d\x00d\x00\x00\xff\xec\x00\x11Duck
y\x00\x01\x00\x04\x00\x00\x00P\x00\x00\xff\xee\x00\x0eAdobe\x00d\xc0\x00\x00\x00\x01
\xff\xdb\x00\x84\x00\x02\x02\x02\x02'... еще 19260 байт
```

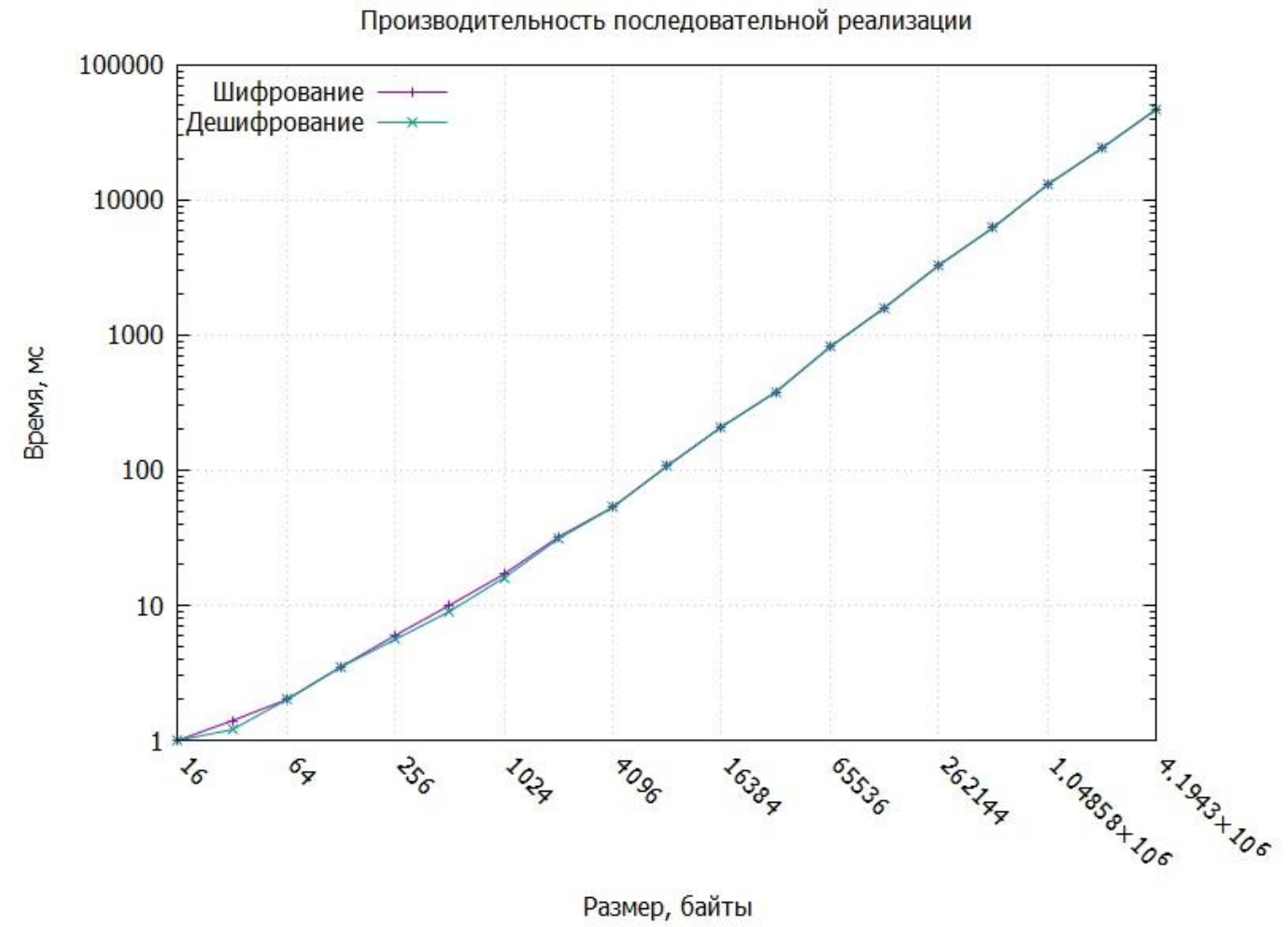

Рисунок 4 – Производительность последовательной реализации.

На рисунке 4 видно, что процедуры шифрования и дешифрования почти не различаются по временным затратам, поэтому далее будем рассматривать только процедуру шифрования.

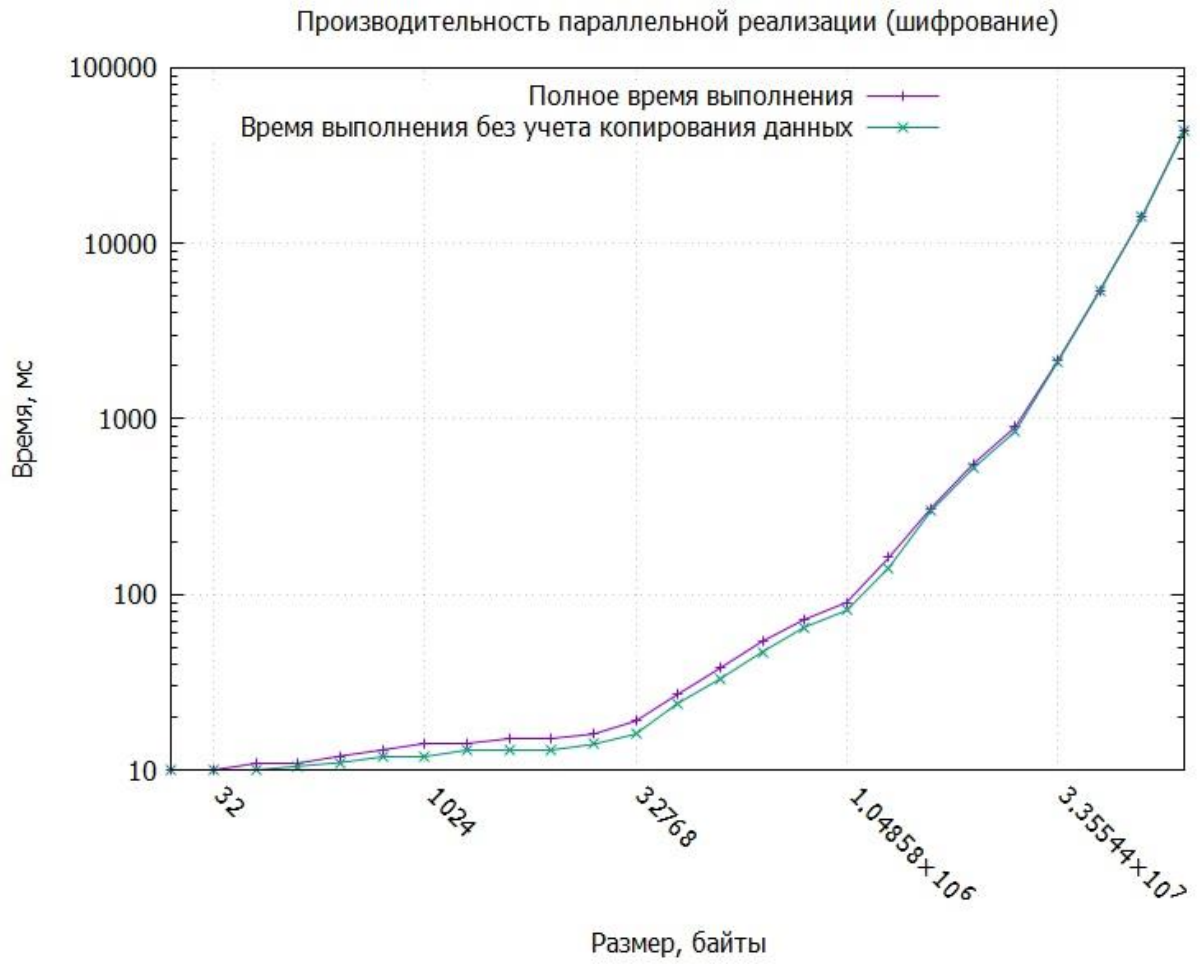

Рисунок 5 - Зависимость времени выполнения кода распараллеленного с применением технологии CUDA от объёма шифруемых данных

На рис. 5 изображено время выполнения параллельного кода. Мы измеряем время выполнения двумя способами. Первый способ - измеряем полное время выполнение программы. Второй способ - измеряем время выполнения параллельного ядра без учёта операций копирования данных, инициализации CUDA и т.п. График построен в двойном логарифмическом масштабе. По оси X - объём информации в байтах, по оси Y - время в миллисекундах. Заметно, что на маленьких объёмах информации (до 16 КБ) рост времени выполнения незначителен. Это связано с тем, что количество блоков алгоритма RC5 достаточно мало (меньше 2048 блоков) сопоставимо с количеством стриминговых процессоров видеокарты, которые используются технологией CUDA для вычислений. По мере возрастания

объёма шифруемой информации возрастает нагрузка на стриминговые процессоры, и следовательно увеличивается время выполнения программы, что мы и наблюдаем на второй части графика. Также заметно, что на таких небольших объёмах данных время передачи данных на устройство (в данном случае видеокарту) незначительно.

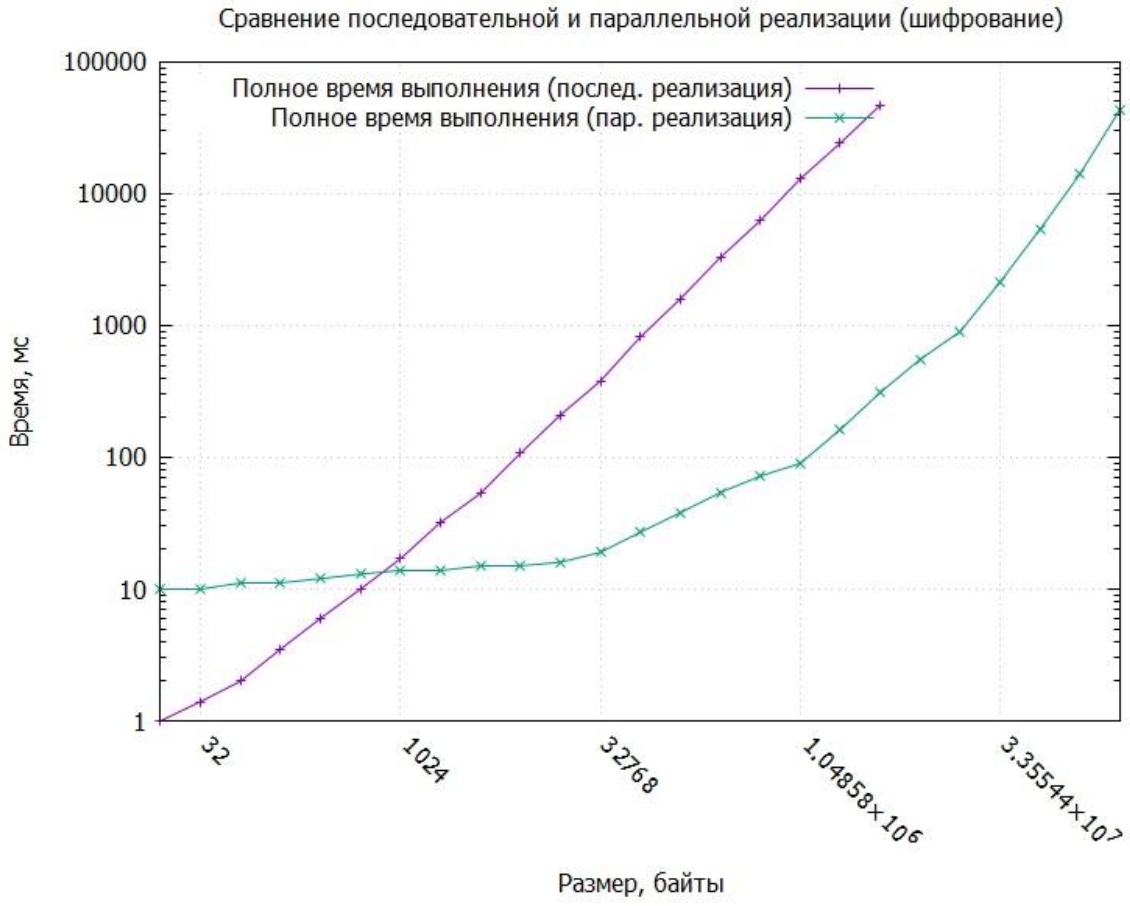

Рисунок 6 - Сравнение времени выполнения последовательного и параллельного кода для процедуры шифрования

На рисунке 6 сопоставлены времена выполнения последовательного и параллельного кода. Заметно, что параллельный код начинает выигрывать у последовательного кода только после достижения некоторого объёма данных. Это связано с тем, что параллельная реализация затрачивает некоторое количество времени на инициализацию и передачу данных на видеокарту.

5. Оптимизация параллельной реализации

Теперь попытаемся оптимизировать реализованную нами программу. Применим последовательно 3 различные оптимизации и посмотрим, какое влияние они окажут на производительность.

1. Оптимизация 1 – Pinned-память.

При распределении памяти ЦП, которая будет использоваться для передачи данных на графический процессор, можно выбрать два типа памяти: прикрепленную (pinned) и обычную память. Pinned-память – это память, выделенная с помощью функции cudaMallocHost, которая обеспечивает улучшенные скорости передачи. Обычная память – это память, выделенная с использованием функций drv.In/drv.Out (PyCuda). Pinned-память немного дороже для распределения и освобождения, но обеспечивает более высокую пропускную способность передачи для больших передач памяти.

Главная особенность pinned-памяти состоит в том, что она не может быть разбита на страницы, в результате чего её передача на GPU значительно ускоряется, так как драйверу нужно проделать меньше работы. При выделении хост-памяти обычным способом (drv.In/drv.Out в PyCuda) драйвер вынужден создать дополнительное pinned-хранилище (рис. 7) и скопировать данные в него. Мы избегаем этого, выделяя требуемый объем pinned-памяти вручную.

Данная оптимизация проводится целиком и полностью на стороне хоста, и не затрагивает CUDA код. В коде программы, представленном в приложении А строки, затронутые данной оптимизацией, оканчиваются комментарием с пометкой «[O-1]».

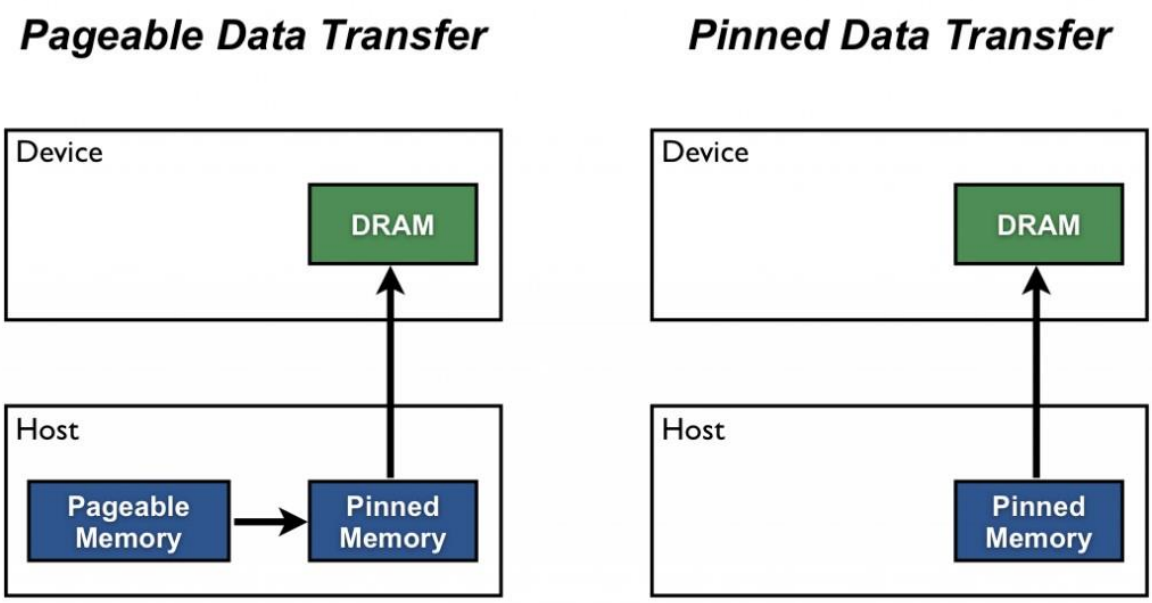

Рисунок 7 - Сравнение передачи данных с использованием pinned-памяти и без

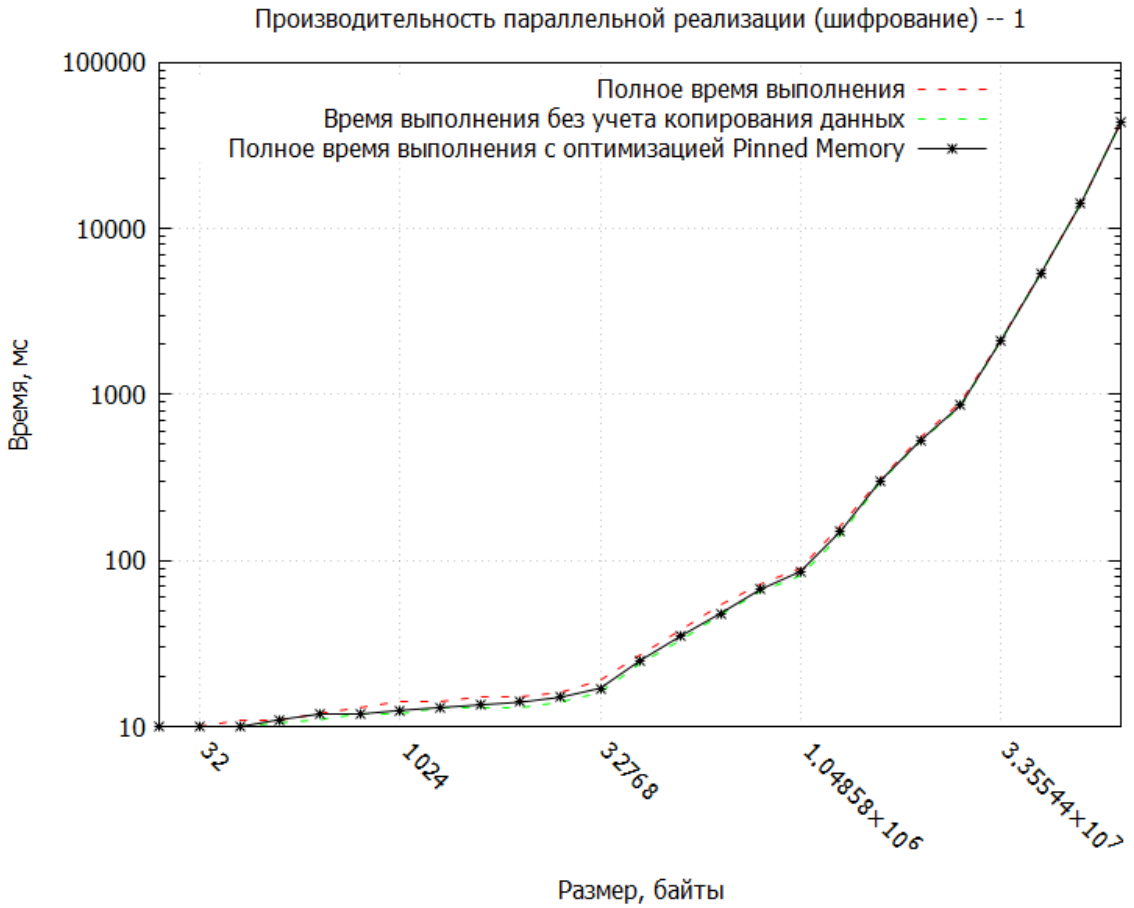

Рисунок 8 - Сравнение времени выполнения неоптимизированного кода и кода, использующего pinned-память

2. Оптимизация 2 – векторизация.

Данная оптимизация весьма незамысловата. Под шифруемые данные в нашей программе выделен массив 32-битных целочисленных значений, но обрабатываем мы их группами по 2 (блоками). Так как на этапе выполнения CUDA-кода мы гарантируем, что размер данных кратен 8, то можем заменить пары значений uint32_t в CUDA-коде CUDA-векторами uint2. Это позволит в полной мере использовать векторные LOAD/STORE инструкции CUDA-процессоров (SIMD-инструкции), оптимизирует доступ к памяти (путем использования тех же самых векторных инструкций).

Эта оптимизация применяется к CUDA-коду, и почти не затрагивает хост-код. В приложении А строки, затронутые оптимизацией векторизации, отмечены комментарием «[О-2]».

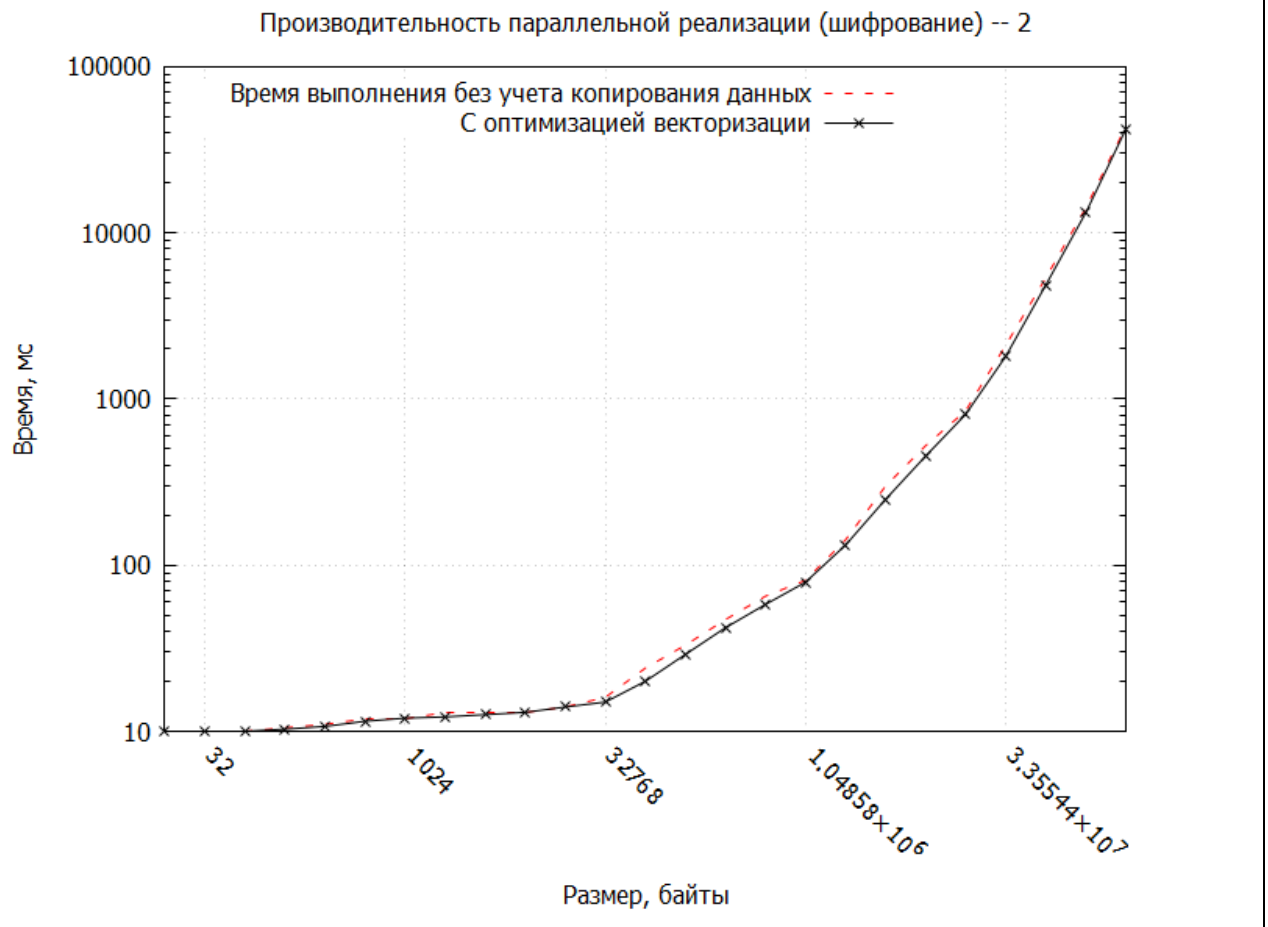

Рисунок 9 - Сравнение времени выполнения неоптимизированной функции-ядра и её векторизованной версии

3. Оптимизация 3 – развертка циклов

Эта оптимизация заключается в простом применении директивы развертки циклов. Синтаксис данной директивы:

```
#pragma unroll N
for (…) {
   тело цикла
}
```

Это служит подсказкой для CUDA, что данный цикл необходимо развернуть в последовательность повторяющихся инструкций. Параметр N указывает, сколько итераций включить в один блок развернутого цикла. В нашем случае количество итераций известно и равно 20, поэтому мы укажем N = 20. Полученный код будет более оптимален, так как счетчик цикла будет удален компилятором (что позволит сэкономить несколько инструкций инкремента и освободит один CUDA-регистр); более того, компилятор сможет использовать более агрессивные оптимизации относительно тела цикла (например, самостоятельно векторизовать какие-либо инструкции).

Развертка циклов – это оптимизация CUDA-кода, и она не затрагивает хост-код. В приложении А строки, затронутые этой оптимизацией, отмечены комментарием «[O-3]».

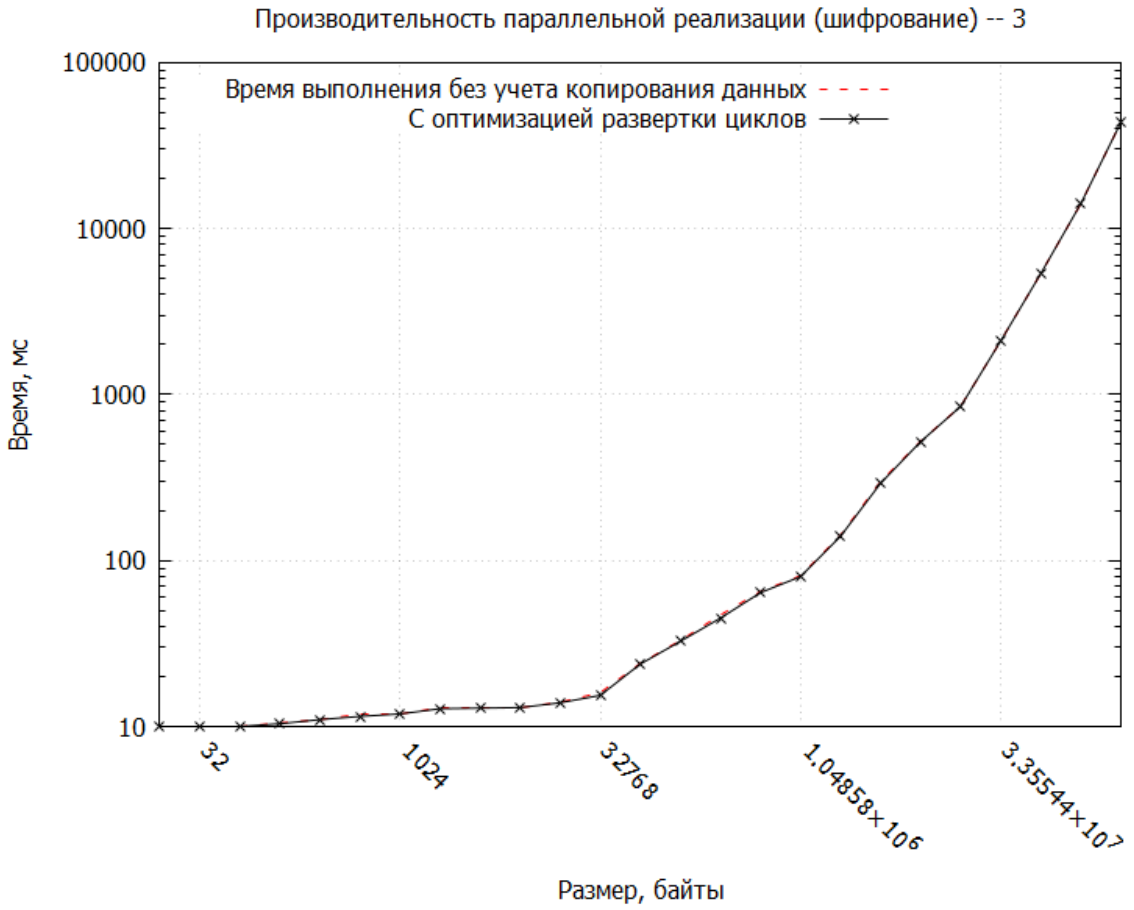

Рисунок 10 - Сравнение времени выполнения неоптимизированной функции-ядра с версией, использующей развертку циклов

Отметим, что время выполнения улучшилось очень незначительно – улучшение составило <1% от времени выполнения неоптимизированной программы. Это связано с тем, что CUDA по умолчанию уже пытается применить развертку циклов к циклам с известным количеством итераций. Однако по умолчанию компилятор CUDA пытается найти баланс между размером исполняемого кода (если развернуть цикл из N итераций полностью, то физический объем кода в байтах может значительно возрасти, если N будет достаточно велико) и его скоростью (чем более развернут цикл – тем быстрее выполнение). Указывая N = 20 в директиве развертки, мы фактически полностью избавляемся от цикла, что дает незначительный прирост по сравнению с поведением компилятора по умолчанию.

4. Объединение оптимизаций

Теперь объединим все оптимизации в рамках одной программы. График сравнения неоптимизированного кода и кода со всеми тремя оптимизациями приведен на рисунке 11.

В результате применения всех вышеперечисленных оптимизаций время выполнения улучшилось в среднем на 10.8% от неоптимизированной версии.

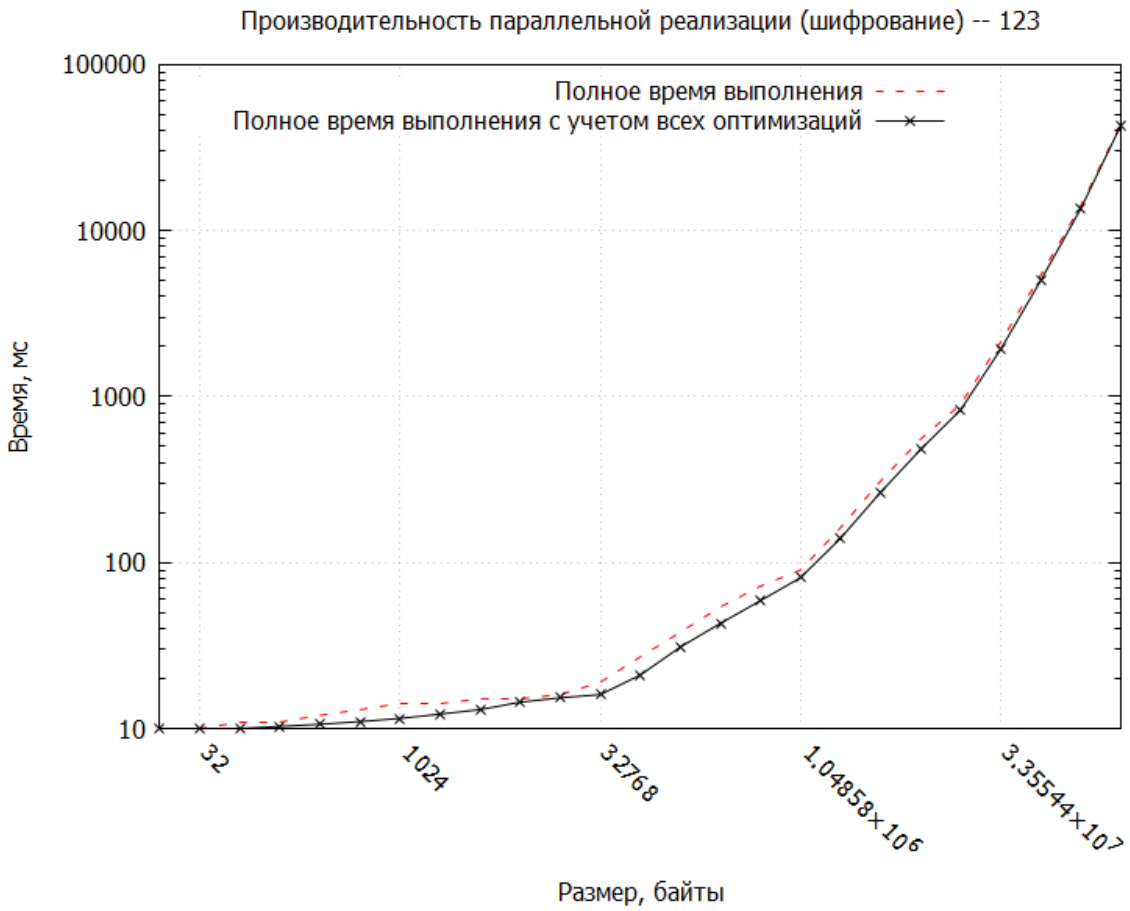

Рисунок 11 - Сравнение времени выполнения неоптимизированной версии программы с оптимизированной

Заключение

Был рассмотрены методы блочного шифрования, в частности алгоритм RC5, который был реализован последовательным (для режимов ECB, CBC, PCBC, CFB) и параллельным (для режима ECB) способом. Реализована программа на языке Python, осуществляющая шифрование файлов при помощи алгоритма RC5.

Было произведено сравнение времени выполнения последовательной и параллельной программы на разных объёмах данных. Было показано, что реализация данного алгоритма при помощи технологии CUDA позволяет получить значительный прирост в производительности.

Была проведена оптимизация параллельной реализации программы. В результате применения всех рассмотренных нами оптимизаций (Pinned-память, векторизация, развертка циклов) время выполнения улучшилось в среднем на 10.8% от неоптимизированной версии..

Список литературы

1. Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Edition. Bruce Schneier. ISBN: 978-0-471-11709-4.
2. Rivest, R. L. (1994). "The RC5 Encryption Algorithm"/ Proceedings of the Second International Workshop on Fast Software Encryption (FSE) 1994e. pp. 86–96.
3. NIST Computer Security Division's (CSD) Security Technology Group (STG). "Block cipher modes". Cryptographic Toolkit. NIST.
4. B. Moeller (May 20, 2004), Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures