Ссылка на статью:
// Ученые записки УлГУ. Сер. Математика и информационные технологии. УлГУ. Электрон. журн. 2022, № 1, с. 8-16.

Поступила: 25.04.2022
Окончательный вариант: 03.05.2022

УДК 519.7

Методы оптимизации программной реализации блочного шифра «Магма»

Гафуров И.Р.

gafurov.ils@yandex.ru
УлГУ, Ульяновск, Россия

В работе проводится исследование методов оптимизации программной реализации алгоритма «Магма» из ГОСТ Р 34.12-2015. Применение исследованных методов позволило увеличить скорость шифрования по сравнению с уже известными реализациями на примере OpenSSL.

***Ключевые слова**: алгоритм «Магма», технология CUDA, язык Assembler*

Введение

Из отечественных алгоритмов шифрования самым ходовым в России является «Магма» из ГОСТ Р 34.12-2015. Он применяется в различных криптопровайдерах, например, CryptoProCSP, VipNetCSP, LissiCSP. Помимо этого, используется в составе протокола установления защищённого канала связи в сети Интернет TLS 1.2 (1.3). Используется и при разработке программного обеспечения и оборудования для защиты ведомственных каналов связи. Однако в современном мире существует проблема роста хранимой и обрабатываемой информации, но большее внимание привлекает увеличение её доли, которая должна быть защищена. Отсюда следует, что криптографические приложения из года в год должны обрабатывать всё больший объём информации за тоже время.

Задача определения способов оптимизации того или иного криптографического приложения более трудоемкая, чем её разработка, ведь в каждой прикладной области существуют характерные требования к уровню защиты информации. Поэтому перед лицами, занимающимися оптимизацией приложения, встаёт вопрос выбора между необходимым уровнем защиты и эффективностью.

У людей, занимающихся оптимизацией программных реализаций криптографических алгоритмов, существует проблема отсутствия единого подхода для увеличения скорости

шифрования/расшифрования, т.к. каждый алгоритм нуждается в индивидуальном рассмотрении. Проведём исследование некоторых методов оптимизации отечественного алгоритма «Магма», описание которого приводится в [1].

1. Алгоритмическая оптимизация

В случае применения данного метода оптимизации используются разного рода математические, а также логические методы для улучшения алгоритма. Успешность такой оптимизации зависит от способности программиста к алгоритмической оптимизации программы. Данная способность проявляется в понимании программистом предметной области: владением им основных идей применяемых алгоритмов и особенностей предметной области программы. В первую очередь это замена алгоритмов на более быстродействующие [3]. Например, замена пузырьковой сортировки массива на быструю сортировку, дискретное преобразования Фурье заменяется на быстрое преобразование Фурье и т.д. В некоторых случаях возможна оптимизация программы за счет снижение точности, однако не во всех предметных областях это возможно, как, например, в нашем случае.

В [1], где описывается алгоритм «Магма», говорится, что после того, как правая (левая) половина шифруемого блока складывается по модулю 32 с текущим итерационным ключом, каждая 4-битная часть поочерёдно с использованием таблицы перестановки заменяется на другое 4-битное число. Реализуя данный алгоритм «в лоб», мы часто обращаемся в таблицу замены, что не оптимально. В связи с этим предлагается провести тонкую работу с оперативной памятью, а именно составить предвычисленные вектора, которые позволят значительно сократить как время на обращение в таблицу, так и время на получение 4-ёх бит из шифруемого блока.

Изначально мы имеем таблицу перестановки размером 8x16, состоящая из 128 4-ёх битных значений. Для её использования в нашей усовершенствованной функции запишем её в виде вектора размерностью 128:

```
static const uint8_t Pi[128] = {
        1, 7, 14, 13, 0, 5, 8, 3, 4, 15, 10, 6, 9, 12, 11, 2,
        8, 14, 2, 5, 6, 9, 1, 12, 15, 4, 11, 0, 13, 10, 3, 7,
        5, 13, 15, 6, 9, 2, 12, 10, 11, 7, 8, 1, 4, 3, 14, 0,
        7, 15, 5, 10, 8, 1, 6, 13, 0, 9, 3, 14, 11, 4, 2, 12,
        12, 8, 2, 1, 13, 4, 15, 6, 7, 0, 10, 5, 3, 14, 9, 11,
        11, 3, 5, 8, 2, 15, 10, 13, 14, 1, 7, 4, 12, 9, 6, 0,
        6, 8, 2, 3, 9, 10, 5, 12, 1, 14, 4, 7, 11, 13, 0, 15,
        12, 4, 6, 2, 10, 5, 11, 9, 14, 8, 13, 7, 0, 3, 15, 1
};
```

Объявим 4 вектора размерностью 256, элементами которых будут 64-х битные значения:

```
uint64_t Pi1[256], Pi2[256], Pi3[256], Pi4[256];
```

Первые 8 старших бит элемента вектора Pi1 будут являться блоком замены для первых 8 старших бит правой (или левой, в зависимости от итерации) части блока. Остальные биты содержат нули и никак далее использоваться не будут.

Вторые 8 старших бит элемента вектора Pi2 также, как и в случае с вектором Pi1, будут являться блоком замены, только на этот раз для вторых 8 старших бит части блока. Вектора Pi3 и Pi4 также хранят блоки замены, но для третьего и четвёртого набора 8 старших бит соответственно.

Таким образом, мы получили предвычисленные векторы. Визуальное представление заполнения векторов Pi1, Pi2, Pi3, Pi4 можно увидеть на рис. 1.

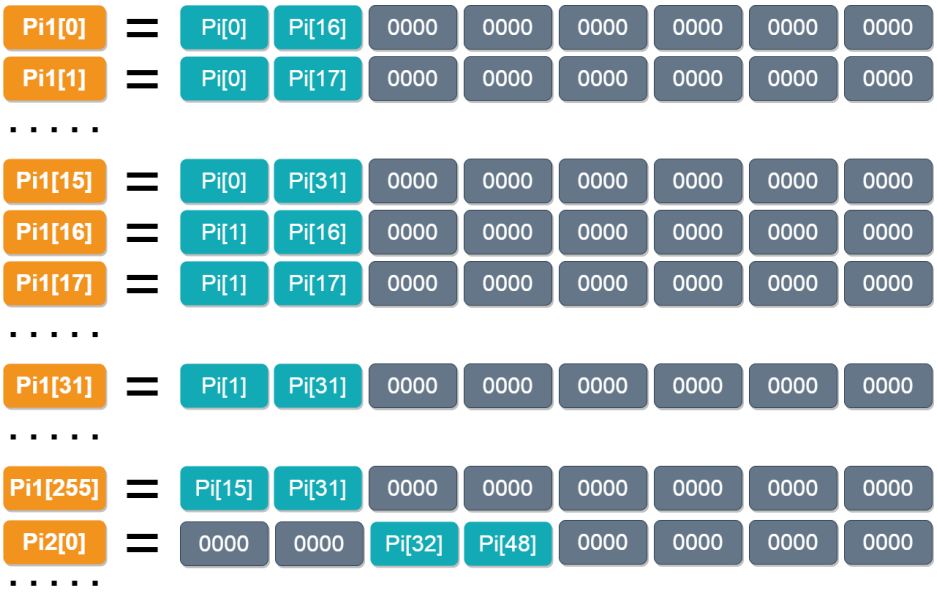

Рис. 1. Схема алгоритмической оптимизации

Теперь мы можем осуществлять перестановку, например, следующим образом:

```
vector = R_part + key[0]; // сложение правой части блока
с итерационным ключом
vector = Pi1[vector >> 24 & 255] | Pi2[vector >> 16 &
255] | Pi3[vector >> 8 & 255] | Pi4[vector & 255]; //
перестановка и формирование новой части блока
```

Следующие методы оптимизации будут применяться совместно с данной алгоритмической оптимизацией.

2. Ассемблерные вставки

Ассемблерная вставка – это возможность использовать в основном коде программы, написанном на языке высокого уровня, вставки из низкоуровневого кода [4]. Когда мы запускаем процесс компиляции, компилятор переводит код с высокого уровня на низкий. После перевода возникает большой объём ненужного кода. Ассемблерные вставки позволяют избежать избыточного кода, и при этом уменьшить число операций.

Быстродействие ассемблерных вставок также заключается в прямом доступе к системе. Код на ассемблере позволяет выполнить прямое обращение к ядру операционной системы, что упрощает программу и делает ее более быстродействующей.

Данные вставки осуществляют обращение к специфическим функциям процессоров различных типов, которые не включены в стандартных библиотеках языков высокого уровня.

Заменим содержание функции шифрования на ассемблерную вставку. Для проверки данного метода оптимизации рассмотрим вставку, которая будет являться переводом операций на языке Си с некоторыми исключениями. Рассматриваемое далее решение можно назвать решением «в лоб», т.к. ассемблер позволяет работать с памятью намного эффективней, чем реализация, приведённая ниже.

Пусть мы имеем массив key состоящий из 8 элементов, каждый из которых является 1/8 частью 256-битного ключа шифрования; R_part и L_part – правая и левая половина шифруемого блока соответственно; G_part и vect – промежуточные 32-ые части блоков.

При написании вставки будут применяться регистры общего назначения eax, ebx, ecx, edx, а также команды rol – циклический сдвиг влево, mov – копирование данных из операнда-источника в операнд-получатель, add – сложение, shr – логический сдвиг вправо, and – побитовое логическое И, or – побитовое логическое ИЛИ, xor – логическое сложение по модулю два.

Одна итерация алгоритма шифрования в виде ассемблерной вставки представлена ниже:

```
_asm
{
mov edx, key;
mov eax, R_part;
add eax, edx + 0;
mov vect, eax;
shr eax, 18h;
and eax, 0ffh;
mov ecx, vect;
shr ecx, 10h;
and ecx, 0ffh;
mov edx, Pi1[eax * 8];
or edx, Pi2[ecx * 8];
mov eax, vect;
shr eax, 8;
and eax, 0ffh;
or edx, Pi3[eax * 8];
mov ecx, vect;
and ecx, 0ffh;
or edx, Pi4[ecx * 8];
rol edx, 11;
mov eax, L_part;
xor eax, edx;
mov L_part, eax;
…}
```

3. Применение технологии CUDA

Технология CUDA (Compute Unified Device Architecture) - вычислительная программно-аппаратная архитектура, разработанная корпорацией NVIDIA, которая может использовать графические карты серии NVIDIA, поддерживающие технологию GPGPU (General-purpose computing on graphics processing units), т.е. технологию произвольных вычислений на видеокартах, для ускорения некоторых сложных вычислений распараллеливая их [5].

Архитектура CUDA построена вокруг масштабируемого массива многопоточных потоковых мультипроцессоров [6]. Когда программа CUDA на CPU хоста вызывает сетку ядра, блоки сетки перечисляются и распределяются мультипроцессорам с доступной емкостью выполнения, причём каждый из блоков всецело выполняется на одном из мультипроцессоров. Когда блоки потока завершаются, новые блоки запускаются на освобожденных мультипроцессорах. Отсюда следует, что, имея маленькое количество потоковых мультипроцессоров можно отправить на выполнение сетку с огромным числом блоков.

Общая схема распараллеливания представлена на рис. 2. Для использования технологии CUDA применим следующие параметры: count – количество 64-битных блоков, thread-количество потоков. В случае если количество потоков окажется больше количества блоков, то вызов функции на GPU осуществится с count потоками, иначе массив текса делится по thread блоков и для каждого из новых массивов выполняется функция на GPU.

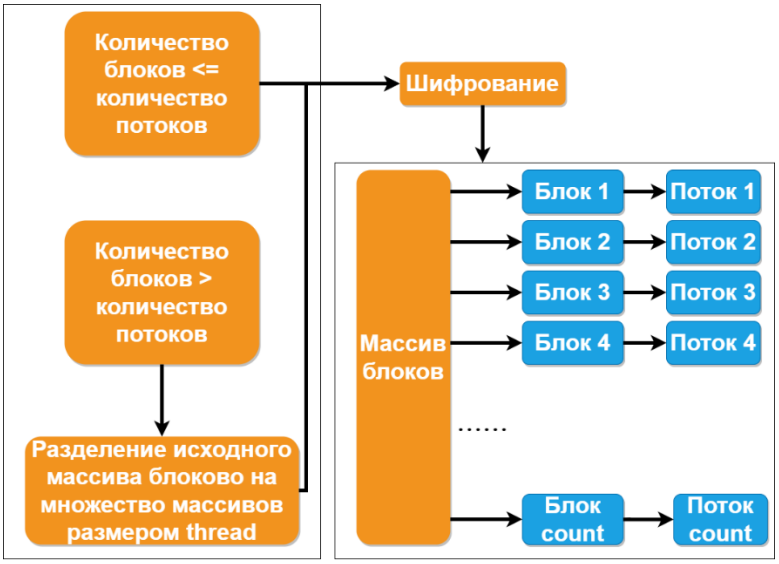

Рис. 2. Общая схема распараллеливания

Рассмотрим участок кода, когда количество блоков меньше количества потоков:

```
uint64_t blocks[count]; // массив блоков шифруемого сообще-
ния
uint64_t *dev_blocks; // копия массива blocks для шифрования
на девайсе
uint64_t size = count * sizeof(uint64_t); // размер массива
dev_blocks
```

```
cudaMalloc( (void**)&dev_blocks, size); // выделяем память
на девайсе
cudaMemcpy(dev_blocks, blocks, size, cudaMemcpyHostToDe-
vice); // копируем данные на девайс
Encrypt <<< 1, count>>>(dev_blocks); // запускаем функцию
Encrypt на девайсе, передавая ей параметры 1 – количество
используемых ядер, count – количество потоков, dev_blocks –
массив блоков
```

По рассмотренному участку кода видно, что вызвать функцию на GPU не составляет труда. Рассмотрим организацию функции шифрования:

```
__global__ void Encrypt(uint64_t *dev_block, uint32_t
*key)
{
   uint64_t i = blockIdx.x * blockDim.x + threadIdx.x;
   uint32_t R_part = block[i] & 0x00000000ffffffff;
   uint32_t L_part = dev_block[i] >> 32;
   …
}
```

На участке кода выше threadIdx.x – индекс текущей нити в блоке, blockDim.x – размер блока потока, blockIdx.x – индекс блока потока. Зная их, мы получаем глобальный индекс нити – i. Это необходимо для параллельного шифрования отдельных 64-битных блоков. Дальше выполняется обычное шифрование, реализация которого совпадает с реализацией на CPU.

4. Сравнение полученных результатов

Перед тем как перейти к сравнению результатов оптимизации и подведению по ним итога необходимо описать конфигурацию ЭВМ и среду разработки, которые применялись при реализации и замере времени шифрования данных программной реализации алгоритма «Магма» с разными методами оптимизации.

Замеры проводились на машине со следующей конфигурацией:

- **CPU:** Intel Core 2 Quad Q8400 с 4 МБ кэш-памяти, частотой 2.66 ГГц
- **RAM:** SAMSUNG 2x4 ГБ (M378B52773DH0-CKO), работающие на частоте 1066 МГц
- **GPU:** NVIDEA GeForce GT 440 с памятью объёмом 1 ГБ и графическим процессором GF108, частота процессора 823 МГц, частота памяти 800 МГц
- **HDD:** ST500LM012 HN-M500MBB

Программная реализация была произведена с применением языков Си и Assembler в среде разработки Visual Studio 2013 и использованием набора инструкций CUDA версии 8.0. Операционная система, установленная на ЭВМ: Windows 8.1 x64 версии 6.3.

Измерение времени шифрования проводилось использованием стандартной библиотеки языки Си time.h.

На рис. 3 и 4 представлены графики зависимости времени работы программной реализации алгоритма «Магма» с разными методами оптимизации от объёма данных, находящиеся на HDD и на RAM соответственно.

Серым цветом выделен график, показывающий зависимость времени шифрования с применением библиотеки библиотекой OpenSSL от объёма данных, синим цветом – с применением алгоритмической оптимизации, жёлтый – с применением ассемблерных вставок, оранжевый – с применением технологии CUDA.

При шифровании данных, находящихся на HDD был применён режим шифрования ECB из [2].

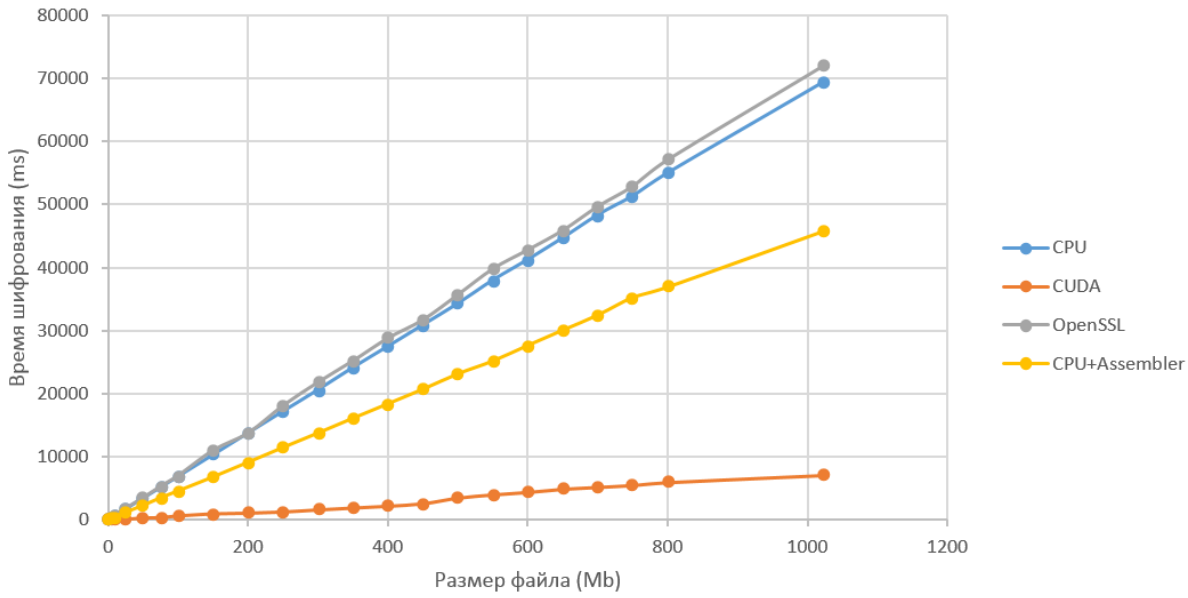

Рис. 3. График зависимости времени шифрования от объёма шифруемой информации, находящейся на HDD

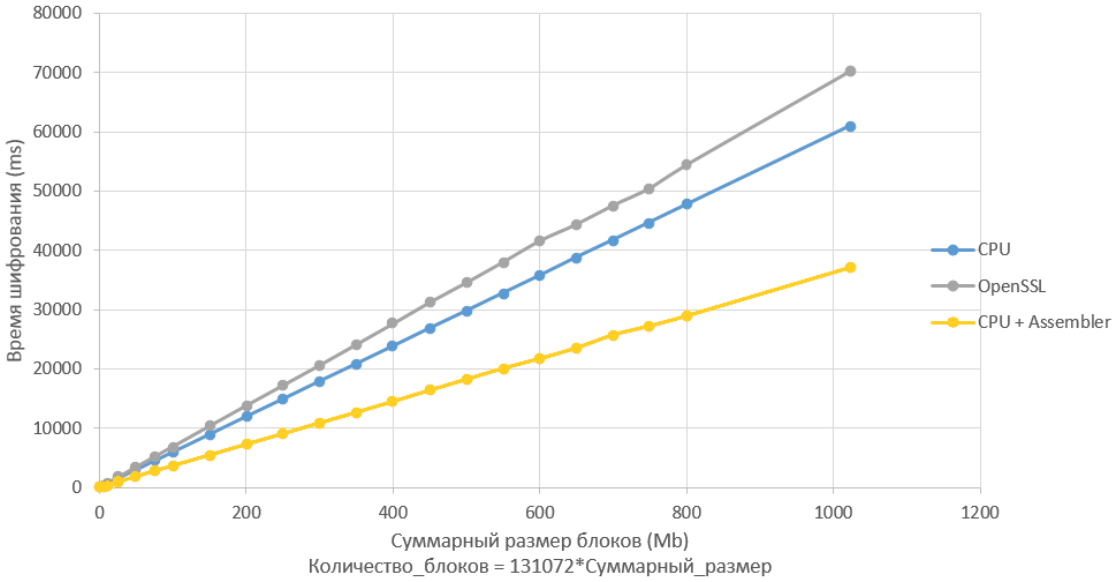

Рис. 4. График зависимости времени шифрования от объёма шифруемой информации, находящейся на RAM

По графикам можно увидеть, что за счет использования большого количества параллельно работающих потоков, GPU, использующие технологию CUDA, обрабатывают файлы быстрее, чем это делает CPU.

Ассемблерная вставка также значительно сократила время шифрования. Это произошло, как было сказано ранее, из-за предоставления возможности прямого обращения к ядру операционной системы.

Описанные выше методы использовались с применением алгоритмической оптимизацией. Без ассемблерной вставки и технологии CUDA алгоритмическая оптимизация также является неплохим решением для оптимизации программной реализации. Программная реализация алгоритма «Магма» со всеми рассмотренными методами оптимизации превзошла по скорости шифрования библиотеку OpenSSL.

Заключение

В работе были исследованы три метода для оптимизации программной реализации алгоритма «Магма» из ГОСТ Р 34.12-2015. Результаты проделанной работы показали правильность выбранного подхода к решению вопроса оптимизации программы. Стоит заметить, что существуют шифры, скорость шифрования и расшифрования файлов которых близка к скорости копирования файлов. Более того, такие шифры обладают свойством совершенности, введённым К. Шенноном, и применяются в особо важных случаях. Более подробно о них можно найти, например, в [7].

Исследованные методы оптимизации позволят реализовать криптографическую библиотеку, которая сможет составить конкуренцию ходовым, например, OpenSSL, Crypto++ и другим. При дальнейшем решении вопроса можно рассмотреть такие методы, как: использование OpenCL, применение процессорных инструкций (SSE 2,3,4; MMX; AVX и т.д.), эффективная работа с памятью и другие существующие методы.

Список литературы

1. ГОСТ Р 34.12-2015. *Информационная технология. Криптографическая защита информации. Блочные шифры.* М.: Стандартинформ, 2016
2. ГОСТ Р 34.13-2015. *Информационная технология. Криптографическая защита информации. Режимы работы блочных шифров.* М.: Стандартинформ, 2016
3. Симонова О. Н. Особенности оценки качества и оптимизации алгоритмов симметричного шифрования // *Молодой ученый.* 2016, № 9 (113), с. 79–81.
4. Дубков В. П. *Программирование на Ассемблере: учеб. -метод. пособие. в 2 ч., ч. 1.* Минск: БГУ, 2010. 48 с.
5. Тумаков Д.Н, Чикрин Д.Е, Егорчев А.А. *Технология программирования CUDA: учебное пособие*. Казань: Казанский государственный университет, 2017. 112 с.

6. Варыгина М.П. *Основы программирования в CUDA: учебное пособие*. Красноярский государственный педагогический университет им. В.П. Астафьева. Красноярск, 2012. 138 с.
7. Рацеев С.М. *Математические методы защиты информации: учебное пособие для вузов.* СПб.: Лань, 2022. 544 с.

Methods of optimization of the software implementation of the block cipher "Magma"

Gafurov, I.R.

gafurov.ils@yandex.ru
Ulyanovsk State University, Ulyanovsk, Russia

In this paper, a study of methods for optimizing the software implementation of the Magma algorithm from GOST R 34.12-2015 is carried out. The use of the studied methods allowed to increase the encryption speed compared to already known implementations using the example of OpenSSL.

Keywords: *Magma algorithm, CUDA technology, Assembler language.*