Ссылка на статью:
// Ученые записки УлГУ. Сер. Математика и информационные технологии. 2023, № 2, с. 19-27.

Поступила: 20.10.2023
Окончательный вариант: 20.10.2023

УДК 519.7

Высокоскоростная программная реализация алгоритма хэширования «Стрибог»

Гафуров И.Р.

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

В статье описывается высокоскоростная программная реализация алгоритма хэширования «Стрибог» из ГОСТ Р 34.11-2012 с использованием процессорных инструкций (MMX, SSE2, SSE4.1), LUT-таблиц и технологии CUDA. Применение данных способов оптимизации позволило увеличить скорость получения хэша файла по сравнению с классической реализацией (без оптимизаций).

***Ключевые слова**: ГОСТ Р 34.11-2012, «Стрибог», SIMD, CUDA, MMX, SSE, LUT-таблицы.*

Введение

В последние годы уровень защиты информации стал одним из главных приоритетов. Алгоритмы хэширования играют важную роль в обеспечении целостности и безопасности данных. Одним из таких считается алгоритм «Стрибог» из ГОСТ Р 34.11-2012, который приобрел большую популярность благодаря своей высокой степени стойкости.

Однако, при использовании такого алгоритма важной задачей является достижение максимально возможной скорости его выполнения. Для этого необходимо разработать эффективную программную реализацию, которая позволит обрабатывать большие объемы данных в достаточно короткое время.

Основной акцент работы будет сделан на подходе применения методов оптимизации и анализе результатов исследования. Результаты данной работы могут помочь понять производительность реализаций алгоритма «Стрибог» и принять рациональные решения при выборе метода оптимизации данного алгоритма.

1. Программная реализация алгоритма «Стрибог» с использованием предвычисленных таблиц

Из [1] нам известно, что преобразуемый блок разделён на порции по 8 байт, где каждая порция представляется 64-битным двоичным числом. Затем берётся каждая порция и каждому её биту соотносится строка из матрицы линейного преобразования A, который задан в [1]. Если бит равен нулю, соответствующая строка из матрицы A исключается, а если бит равен единице, соответствующая строка из матрицы A остается. Затем оставшиеся строки из матрицы A ксорятся, и полученное число записывается в виде очередной 8-байтовой порции в результирующий вектор.

Зададим множество, включающее в себя значения матрицы подстановки из пункта 5.2 в [1]:

$$S^* = \{252, 238, 221, 17, 207, 110, 49, 22, \dots, 182\}.$$

Значения этого множества представляют собой всевозможные варианты байт 8-байтовой порции преобразуемого блока.

Введём матрицу для линейного преобразования множества двоичных векторов, который содержит в себе константы из пункта 5.4 в [1] следующим образом:

$$A = \begin{pmatrix} 8e20faa72ba0b470 \\ 47107ddd9b505a38 \\ ad08b0e0c3282d1c \\ \vdots \\ 641c314b2b8ee083 \end{pmatrix}_{64\times 1}$$

Пусть функция Bit(x) возвращает позиции единиц в байте очередной порции преобразуемого блока. Используя это и выше заданные математические объекты, составим формулу, задающую множество всевозможных вариантов LS-преобразования для каждого из байта некоторой порции:

$$\left\{ \bigoplus_{k\in Bit(j)} A_{63-8*i-k} \middle| i = \overline{0,7}, j \in S^* \right.$$

В этой формуле i – номер байта порции, j – возможное значения байта порции, k – позиции единиц в j.

Отсюда получаем следующую предвычисленную таблицу для рассматриваемого преобразования, где строка $i \in \overline{0,7}$ содержит все варианты результата преобразования байта порции с соответствующим номером:

$$\begin{pmatrix} \bigoplus_{k\in Bit(252)} A_{63-k} & \bigoplus_{k\in Bit(238)} A_{63-k} & \dots & \bigoplus_{k\in Bit(238)} A_{63-k} \\ \bigoplus_{k\in Bit(252)} A_{63-8*1-k} & \bigoplus_{k\in Bit(238)} A_{63-8*1-k} & \dots & \bigoplus_{k\in Bit(238)} A_{63-8*1-k} \\ \dots & \dots & \dots & \dots \\ \bigoplus_{k\in Bit(252)} A_{63-8*7-k} & \bigoplus_{k\in Bit(238)} A_{63-8*7-k} & \dots & \bigoplus_{k\in Bit(238)} A_{63-8*7-k} \end{pmatrix}_{8\times 256}$$

Подставляя значения, получим:

$$\begin{pmatrix} d01f715b5c7ef8e6 & 16fa240980778325 & \dots & 27945a985d5bd4d6 \\ de553f8c05a811c8 & 1906b59631b4f565 & \dots & b00fa37af42f0376 \\ \dots & \dots & \dots & \dots \\ e63f55ce97c331d0 & 25b506b0015bba16 & \dots & d6a30f258c153427 \end{pmatrix}_{8\times 256}$$

Таблицу выше именуем tableLPS.

У нас получилось восемь строк по 2^8 8-байтовых значений. В них содержатся все вероятные результаты LS-преобразования.

Проведём LPS-преобразование над 64-байтовым блоком $A = (A_7 \parallel A_6 \parallel \cdots \parallel A_0)$, где $A_{\overline{0,7}}$ – её 8-байтовые составляющие, т.е. $A_0 = (a_0 \parallel a_1 \parallel \cdots \parallel a_7)$, $A_1 = (a_8 \parallel a_9 \parallel \cdots \parallel a_{15})$, …, $A_7 = (a_{55} \parallel a_{56} \parallel \cdots \parallel a_{63})$.

Преобразование осуществляется следующим образом:

$$\mathrm{LPS}\begin{pmatrix} A_7 \\ A_6 \\ \vdots \\ A_0 \end{pmatrix}_{8\times1} = \begin{pmatrix} \mathrm{tableLPS}_{0,a_7} \oplus \mathrm{tableLPS}_{1,a_{15}} \oplus \ldots \oplus \mathrm{tableLPS}_{7,a_{63}} \\ \mathrm{tableLPS}_{0,a_6} \oplus \mathrm{tableLPS}_{1,a_{14}} \oplus \ldots \oplus \mathrm{tableLPS}_{7,a_{62}} \\ \vdots \\ \mathrm{tableLPS}_{0,a_0} \oplus \mathrm{tableLPS}_{1,a_8} \oplus \ldots \oplus \mathrm{tableLPS}_{7,a_{56}} \end{pmatrix}_{8\times1}$$

Реализовать функцию, которая составит предвычисленную таблицу для LS-преобразования можно, например, следующим образом:

```
for (j = 0; j < 8; j++)
{
    for (i = 0; i < 256; i++)
    {
        uint8_t t = Pi[i];
        uint64_t a = 0;
        for (k = 0; k < 8; k++)
        {
            if (t & 1)
                a ^= A[63 - 8 * j - k];
            t >>= 1;
        }
        tableLPS[j][i]=a;
    }
}
```

Полученную LUT-таблицу будем использовать при программной реализации со всеми последующими методами оптимизации.

2. Программная реализация алгоритма «Стрибог» с использованием векторных инструкций MMX

В данном случае, программная реализация будет включать в себя следующие SIMD инструкции MMX [6]:

1) pxor – выполняет побитовое исключающее ИЛИ двух 64-битных целочисленных векторов;
2) punpcklbw (punpckhbw) – извлекает младшие (старшие) 32 бита из двух 64-битных целочисленных векторов и собирает их в 64-битный целочисленный вектор по схеме, представленной на рис. 1 и 2.

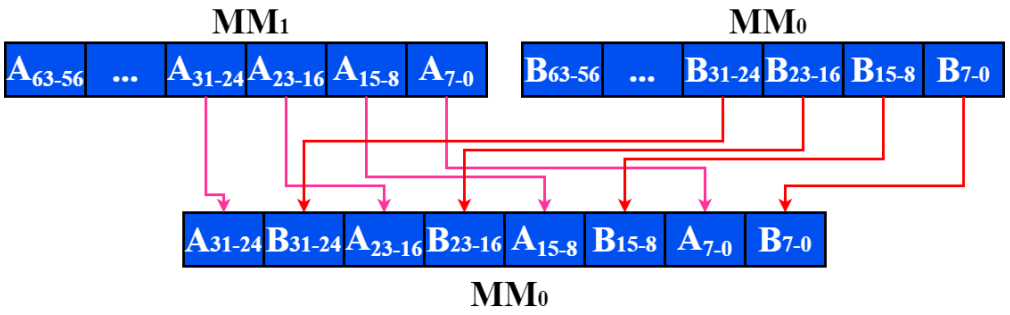

Рис. 1. Выполнение инструкции «punpcklbw mm0, mm1»

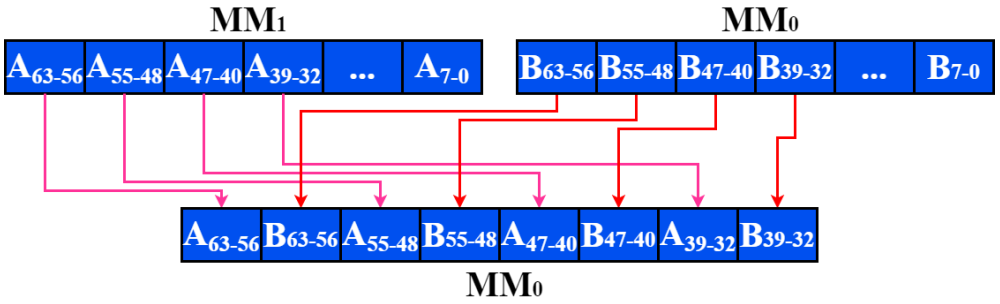

Рис. 2. Выполнение инструкции «punpckhbw mm0, mm1»

3) punpckldq (punpckhdq) – извлекает младшие (старшие) 32 бита из двух 64-битных целочисленных векторов и собирает их в 64-битный целочисленный вектор по схеме, представленной на рис. 3.

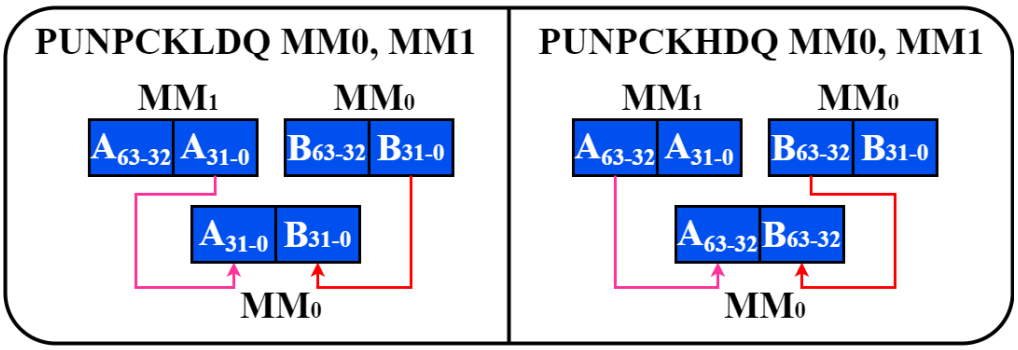

Рис. 3. Выполнение инструкций «punpckldq mm0, mm1» и «punpckhdq mm0, mm1»

4) movq – копирует 64 бита из исходного операнда (второй операнд) в целевой операнд (первый операнд).

Покажем применение этих инструкций в реализации функции сжатия.

Пусть хэшируемой блок равен m=01323130393837363534333231303938373635343332313039383736353433323130393837363534333231303938373635343332313039383736353433323130393837363534333231303938373635343332313039383736353433323130.

В первую очередь проводим LPS-преобразование результата ксора $h=0_0\ldots0_{511}0_{512}$ и $N=0_0\ldots0_{511}0_{512}$. Применим предвычисленную таблицу tableLPS, которую нашли в предыдущем пункте. Для простоты описания, воспользуемся набором инструкций из заголовочного файла

mmintrin.h.

```
i=0;
while(i < 8){
    __m64 buf;
    buf = _mm_cvtsi64_m64(A[0][*(p++)]);
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[1][*(p++)]));
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[2][*(p++)]));
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[3][*(p++)]));
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[4][*(p++)]));
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[5][*(p++)]));
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[6][*(p++)]));
    buf = _mm_xor_si64(buf, _mm_cvtsi64_m64(tableLPS[7][*(p++)]));
    data[i++] = _mm_cvtm64_si64(buf);
}
```

В коде выше, p – транспонированный результат $h \oplus N$, причём этот результат представляется в виде вектор-столбца из 8 строк с 64-битными элементами, _mm_cvtsi64_m64 – функция преобразования из int64 в тип mmx, _mm_cvtm64_si64 - функция преобразования из типа mmx в int64, _mm_xor_si64 – исключающее ИЛИ двух 64-битных целочисленных векторов, data[i] – вектор из 8-ми 64-битных чисел (результат LPS-преобразования).

Для написания функции транспонирования идеально подошла комбинация инструкций из рис. 1, рис. 2 и рис. 3.

В результате LPS-преобразования мы получаем K_1 = b383fc2eced4a574b383fc2eced4a57 4b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383f c2eced4a574.

Продолжая действовать по алгоритму, получим, что результат хэширования тот же, что и в примере из [1].

3. Программная реализация алгоритма «Стрибог», используя векторные инструкции SSE2 и SSE4.1

Теперь рассмотрим инструкции SSE2 и SSE4.1, обозначив их как в наборе инструкций из заголовочных файлов emmintrin.h и smmintrin.h соответственно.

Инструкции SSE2 [6]:

1) _mm_cvtsi64_si128 (movq) – возвращает вектор, младшим элементом которого является 64-битное число, при этом обнуляет старшие биты
2) _mm_load_si128, _mm_store_si128 (movdqa) – загружает 128-битное целочисленное число из оперативной памяти в xmm регистр (и наоборот)
3) _mm_xor_si128 (pxor) – выполняет побитовое исключающее ИЛИ двух 128-битных целочисленных векторов;

Инструкции SSE4.1 [6]:

1) _mm_extract_epi8 (pextrb) – извлекает 8-битное число из элемента упакованного целочисленного массива, выбранного по индексу.
2) _mm_insert_epi64 (pextrq) – извлекает 64-битное число из элемента упакованного целочисленного массива, выбранного по индексу.

Покажем применение этих инструкций в реализации функции сжатия g. Пусть m, h и N будут такими же, как и в предыдущем пункте.

В данном случае обойдёмся без транспонирования результата $h \oplus N$. Для размещения преобразовываемого блока используем регистры xmm10, xmm11, xmm12, xmm13. В качестве буфера используем xmm0.

Проведём LPS-преобразование над 64-байтовым блоком $A = (A_3 \parallel A_2 \parallel A_1 \parallel A_0)$, где $A_{\overline{0,3}}$ – её 16-байтовые составляющие, т.е. $A_0 = (a_0 \parallel a_1 \parallel a_2 \parallel a_3)$, $A_1 = (a_4 \parallel a_5 \parallel a_6 \parallel a_7)$, $A_2 = (a_8 \parallel a_9 \parallel a_{10} \parallel a_{11})$, $A_3 = (a_{12} \parallel a_{13} \parallel a_{14} \parallel a_{15})$.

Чтобы формула не выглядела громоздкой, переименуем здесь tableLPS на F. В скобках напротив $a_{\overline{0,15}}$ будем указывать номер байта.

Преобразование осуществляется следующим образом:

$$\mathrm{LPS}\begin{pmatrix} A_3 \\ A_2 \\ A_1 \\ A_0 \end{pmatrix}_{4\times 1} = \begin{pmatrix} F_{0,a_0(6)} \oplus F_{1,a_0(14)} \oplus F_{2,a_1(6)} \oplus \ldots \oplus F_{7,a_3(14)} \parallel F_{0,a_0(7)} \oplus F_{1,a_0(15)} \oplus F_{2,a_1(7)} \oplus \ldots \oplus F_{7,a_3(15)} \\ F_{0,a_0(4)} \oplus F_{1,a_0(12)} \oplus F_{2,a_1(4)} \oplus \ldots \oplus F_{7,a_3(12)} \parallel F_{0,a_0(5)} \oplus F_{1,a_0(13)} \oplus F_{2,a_1(5)} \oplus \ldots \oplus F_{7,a_3(13)} \\ F_{0,a_0(2)} \oplus F_{1,a_0(10)} \oplus F_{2,a_1(2)} \oplus \ldots \oplus F_{7,a_3(10)} \parallel F_{0,a_0(3)} \oplus F_{1,a_0(11)} \oplus F_{2,a_1(3)} \oplus \ldots \oplus F_{7,a_3(11)} \\ F_{0,a_0(0)} \oplus F_{1,a_0(8)} \oplus F_{2,a_1(0)} \oplus \ldots \oplus F_{7,a_3(8)} \parallel F_{0,a_0(1)} \oplus F_{1,a_0(9)} \oplus F_{2,a_1(1)} \oplus \ldots \oplus F_{7,a_3(9)} \end{pmatrix}_{4\times 1}$$

Программную реализацию можно представить следующим образом (пример для преобразования младших 128-ми бит):

```
uint64_t buf0, buf1;
buf0=tableLPS[0][_mm_extract_epi8(xmm10, 0)];
buf0^=tableLPS[1][_mm_extract_epi8(xmm10, 8)];
buf0^=tableLPS[2][_mm_extract_epi8(xmm11, 0)];
buf0^=tableLPS[3][_mm_extract_epi8(xmm11, 8)];
buf0^=tableLPS[4][_mm_extract_epi8(xmm12, 0)];
buf0^=tableLPS[5][_mm_extract_epi8(xmm12, 8)];
buf0^=tableLPS[6][_mm_extract_epi8(xmm13, 0)];
buf0^=tableLPS[7][_mm_extract_epi8(xmm13, 8)];
buf1=tableLPS[0][_mm_extract_epi8(xmm10, 1)];
buf1^=tableLPS[1][_mm_extract_epi8(xmm10, 9)];
buf1^=tableLPS[2][_mm_extract_epi8(xmm11, 1)];
buf1^=tableLPS[3][_mm_extract_epi8(xmm11, 9)];
buf1^=tableLPS[4][_mm_extract_epi8(xmm12, 1)];
buf1^=tableLPS[5][_mm_extract_epi8(xmm12, 9)];
buf1^=tableLPS[6][_mm_extract_epi8(xmm13, 1)];
buf1^=tableLPS[7][_mm_extract_epi8(xmm13, 9)];
```

```
xmm0= _mm_insert_epi64(_mm_cvtsi64_si128(buf0), buf1, 1);
```

В результате LPS-преобразования мы получаем K_1 = b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574b383fc2eced4a574.

4. Программная реализация алгоритма «Стрибог», используя технологию CUDA

В алгоритме хэширования «Стрибог» есть некоторые операции, которые быстро выполняются с помощью технологии CUDA [4, 5]. Однако функция сжатия имеет особенность – для вычисления каждого последующего хэш-кода необходимо знать предыдущий. Это усложняет реализацию алгоритма с использованием технологии CUDA. Однако, мы можем получить хэш-сумму файла используя один поток видеокарты, или нескольких файлов, но выделяя на каждый файл по потоку.

Схема распараллеливания и пример подход к реализации функции хэширования представлены в [2].

5. Сравнение полученных результатов

Возможности целевой платформы значимы для реализации и выполнения программы. В связи с этим, перед переходом к сравнению полученных результатов опишем конфигурацию ЭВМ и среду разработки, которые применялись при реализации.

Замеры проводились на персональном компьютере и ноутбуке со следующими конфигурациями:

1) Персональный компьютер (стенд №1):
   - **CPU:** Intel Core i3 9100f, 3.6GHz
   - **GPU:** NVIDIA GeForce GT 1030, 2 GB DDR4
   - **RAM:** Kingston 2x8 GB (KF3200C16D4/8GX), DDR4-2400
   - **SSD:** Kingston A400, 240 GB, SATA 3
2) Ноутбук Acer Aspire 5750G (стенд №2):
   - **CPU:** Intel Core i3 2330m, 2.2GHz
   - **RAM:** Samsung 2x4 GB (M471B5273DH0-CH9/4GX), DDR3-1333
   - **SSD:** Kingston A400, 120 GB, SATA 3

Программная реализация была произведена с применением языка C++ в среде разработки Visual Studio 2022 Professional. Операционная система: Windows 10 версии 22H2.

Сравнение скорости получения хэш-суммы программной реализацией «Стрибог» с применением различных методов оптимизации приведено в таблицах 1 и 2.

Таблица 1. Скорости получения хэш-суммы файлов разными программными реализациями алгоритма «Стрибог» (данные находятся в ОЗУ)

	Предвы-численная таблица	Векторные инструкции MMX	Совмещение векторных инструкций SSE2 и	Технология CUDA

				SSE4.1	
Скорость хэширования	Стенд №1	79.3 МБайт/с	86.4 МБайт/с	120.4 МБайт/с	203.3 МБайт/с
	Стенд №2	45.2 МБайт/с	43.8 МБайт/с	58.2 МБайт/с	–

Таблица 2. Скорости получения хэш-суммы файлов разными программными реализациями алгоритма «Стрибог» (данные находятся на SSD)

		Предвы-численная таблица	Векторные инструкции MMX	Совмещение векторных инструкций SSE2 и SSE4.1	Технология CUDA
Скорость хэширования	Стенд №1	55.3 МБайт/с	60.2 МБайт/с	87.6 Мбайт/с	147.6 МБайт/с
	Стенд №2	31.2 МБайт/с	30.3 МБайт/с	41.3 Мбайт/с	–

При сравнении реализаций алгоритма хэширования «Стрибог» на графическом и центральном процессоре наблюдается меньший прирост производительности по сравнению с аналогичной реализацией алгоритма шифрования «Магма» из ГОСТ Р 34.12-2015 [3]. Это объясняется спецификой стандарта «Стрибог», так как для реализации функции сжатия необходимо предыдущее значение, что не даёт параллельно вычислять значения нескольких блоков.

Из таблиц 1 и 2 видно, что благодаря использованию большого числа параллельных потоков, графические процессоры с технологией CUDA обрабатывают файлы быстрее, чем центральный процессор.

Использование процессорных инструкций также значительно сократила время хэширования, что также хорошо показано в [2] при применении инструкций к алгоритмам шифрования. Результаты те же, значительно сократилось количество инструкций и циклов, необходимых для обработки большого объема данных. При этом уменьшилось количество операций загрузки и выгрузки данных в процессор, что также способствовало ускорению получения хэш-суммы.

Заключение

В работе был реализован алгоритм хэширования из ГОСТ Р 34.11-2012 с применением процессорных инструкций, LUT-таблицы и технологии CUDA.

Применение SIMD инструкций процессора позволило продемонстрировать большую эффективность при работе с массивами данных. Непосредственная работа с низкоуровневыми командами позволило сократить отведённое процессорное время на выполнение каждого

рассматриваемого криптографического алгоритма.

Применение технологии CUDA значительно увеличило производительность и позволило эффективно работать с данными в разы увеличив скорость получения хэша. Однако было обнаружено, что реализации алгоритма хэширования «Стрибог» имеют сниженный потенциал использования параллелизма. Это объясняется особенностями ГОСТ Р 34.11-2012, который требует предыдущего значения для реализации функции сжатия, не позволяя параллельно рассчитывать значения нескольких блоков.

Результаты работы позволяют лучше понять производительность реализаций алгоритма хэширования «Стрибог» и являются полезными в области разработки программного обеспечения и повышении производительности.

Список литературы

1. ГОСТ Р 34.12-2015. *Информационная технология. Криптографическая защита информации. Функция хэширования*. М.: Стандартинформ, 2012
2. Гафуров И.Р. *Высокоскоростная программная реализация алгоритмов шифрования из ГОСТ Р 34.12-2015* // Ученые записки УлГУ. Сер. Математика и информационные технологии. 2022, № 2, с. 38-48.
3. Гафуров И.Р. *Методы оптимизации программной реализации блочного шифра «Магма»* // Ученые записки УлГУ. Сер. Математика и информационные технологии. УлГУ. Электрон. журн. 2022, № 1, с. 8-16.
4. Тумаков Д.Н, Чикрин Д.Е, Егорчев А.А. *Технология программирования CUDA: учебное пособие*. Казань: Казанский государственный университет, 2017. 112 с.
5. Варыгина М.П. *Основы программирования в CUDA: учебное пособие.* Красноярский государственный педагогический университет им. В.П. Астафьева. Красноярск, 2012. 138 с.
6. Intel®64 and IA-32 Architectures Software Developer’s Manual. Intel Corporation, 2016.

High-speed software implementation of the «Stribog» hashing algorithm

Gafurov, I.R.

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

In this work, a high-speed software implementation of the hashing algorithm "Stribog" from GOST R 34.11-2012 is carried out using processor instructions (MMX, SSE2, SSE4.1), LUT tables and CUDA technology. The use of these optimization methods allowed to increase the speed of obtaining the hash of the file compared to the classical implementation (without optimizations).

Keywords: *GOST R 34.11-2012, "Stribog", SIMD, CUDA, MMX, SSE, LUT tables*