Math-Net.Ru

Общероссийский математический портал

Р. А. Lebedev, Сравнение старого и нового стандартов РФ на криптографическую хэш-функцию на ЦП и графических процессорах NVIDIA, *Матем. вопр. криптогр.*, 2013, том 4, выпуск 2, 73–80

DOI: <https://doi.org/10.4213/mvk84>

Использование Общероссийского математического портала Math-Net.Ru подразумевает, что вы прочитали и согласны с пользовательским соглашением

<http://www.mathnet.ru/rus/agreement>

Параметры загрузки:

IP: 54.172.240.79

26 апреля 2023 г., 07:36:19

УДК: 519.719.2

Comparison of old and new cryptographic hash function standards of the Russian Federation on CPUs and NVIDIA GPUs

P. A. Lebedev

Moscow Institute of Electronics and Mathematics NRU HSE, Moscow

Получено 18.IX.2012

We present optimization guidelines and implementations of cryptographic hash functions GOST R 34.11-94 and GOST R 34.11-2012. Results for x86_64 CPUs and NVIDIA CUDA-capable GPUs are provided for our and several other well-known implementations. It is shown that the new standard may be twice as fast as the old one on modern CPUs, but it may be slower on embedded devices and GPUs. The results given for our implementation are the fastest among all the tested implementations on both platforms.

Key words: hash function, Stribog, GOST R 34.11-94, GOST R 34.11-2012, CUDA

Сравнение старого и нового стандартов РФ на криптографическую хэш-функцию на ЦП и графических процессорах NVIDIA

П. А. Лебедев

Московский институт электроники и математики НИУ ВШЭ, Москва

Аннотация. В работе представлены способы оптимизации и соответствующие реализации семейства криптографических хэш-функций ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012. Приводятся результаты для процессоров архитектуры x86_64 и видеокарт NVIDIA с поддержкой технологии CUDA для авторской и других известных реализаций. Показано, что новая хэш-функция на современных ЦП может работать вдвое быстрее, чем старая, но медленнее на графических картах и системах с ограниченными ресурсами. Результаты авторских реализаций на обеих платформах показывают наибольшую эффективность среди всех протестированных.

Ключевые слова: хэш-функция, Стрибог, ГОСТ Р 34.11-94, ГОСТ Р 34.11-2012, CUDA

Citation: *Mathematical Aspects of Cryptography*, 2013, vol. 4, no. 2, pp. 73–80 (Russian).

© 2013 P. A. Lebedev

1. Introduction

Stribog is a family of cryptographic hash functions that has replaced GOST R 34.11-94 [1] as the national standard of the Russian Federation under the name GOST R 34.11-2012 [2]. We'll be using both the development and the standard name in this work interchangeably. The description given in the standard is unfortunately not suitable for direct implementation that strives to be as fast as possible. In this work we describe necessary algorithmic and platform-specific optimizations to achieve maximum performance.

A study of implementation details for these hash functions using GPGPU computing is important to test how well-suited is the choice of primitive operations and data units for modern massively parallel architectures. NVIDIA CUDA [3] was chosen as an implementation platform for this work for its ease of use, maturity and popularity. We have also developed a portable implementation for the CPU and an optimized assembly variant which utilizes SIMD instructions of the modern x86_64 platform.

To compare the performance of the new hash function and of the old standard, we have developed implementations of GOST R 34.11-94 hash function for all mentioned platforms and variants. We present the results obtained and compare them to other known available implementations.

2. Stribog hash function family

The base structure follows that of its predecessor: input message is split into blocks of fixed size. If the message is not a multiple of block size, it is padded. Initial internal hash state is updated by sequentially processing input blocks with the compression function. Number of message bits processed and a checksum of all blocks are calculated in parallel. After that a block containing total message length (MD-strengthening) and the final checksum block are processed by compression function to finalize the hash value. The block and internal state (not including block checksum and message length) size for Stribog hash function is 512 bits. For the purpose of representing message length and block checksum, blocks are treated as little-endian numbers modulo 2^{512} .

The family consists of two hash functions with hash value size 256 and 512 bits respectively that differ by the initial state and what part of final state is declared as a result. This makes their performance identical. All the results in this work apply to both of them.

The main difference of the proposed hash function is in its compression function. Its base primitive operation denoted LPS is a composition of three transforms:

1. S – nonlinear bijection. Treats 512 operand bits as an array of 64 bytes and replaces each of them according to the predefined substitution table.
2. P – byte reordering. Rearranges operand bytes in the predefined order.
3. L – linear transformation. Operand is treated as 8 64-bit vectors and each of them is replaced by the result of multiplication over $GF(2)$ with a predefined 64×64 matrix.

Only LPS and bitwise XOR of 512-bit blocks is used in the compression function. Together with addition modulo 2^{512} these are the only operations used in Stribog hash function.

The following general optimizations were done by Kazymyrov and Shevchuk in [5]:

1. Use a lookup table to unroll multiplication in a linear transformation per byte. The size of this lookup table is 16 KB.
2. Byte reordering transformation described in the standard is essentially an 8×8 byte matrix transpose. It is performed by reading input for the L transform with a stride 8.
3. Substitution transformation may be performed “for free” by reordering the precomputed multiplication table in accordance with the substitution table. This optimization becomes possible by choosing exactly one byte as unroll factor for the multiplication stage.

Listing 1: GOST R 34.11-2012 compression function

```
const uint8_t pi[256] = { /* pi constant */ };
const uint64_t A[64] = { /* A constant */ };

uint64_t mul_table[8][256];

void precalc_mul_table()
{
    for(int i=0; i<8; ++i)
        for(int j=0; j<256; ++j){
            uint64_t t = 0;
            uint8_t p = pi[j];
            for(int k=0; k<8; ++k)
                if(p < (1<<k))
                    t ^= A[(i<<3) | k];
        }
}
```

```

        mul_table[i][j] = t;
    }
}

void lps(uint64_t* out, const uint8_t* in)
{
    for(int i=0; i<8; ++i){
        uint64_t t = mul_table[0][in[i]];
        for(int k=1; k<8; ++k)
            t ^= mul_table[k][in[i+k*8]];
        out[i] = t;
    }
}

```

These optimizations are sufficient for a portable implementation. The corresponding code in C is shown in listing 1. We have developed an optimized implementation for the x86_64 architecture that makes use of the expanded CPU register file. By utilizing most general purpose and SSE registers it is possible to store all the hash state except block checksum and all the temporary values. Block checksum is updated by addition modulo 2^{512} that may be efficiently done with add-with-carry instruction on general purpose registers. Storing it there is, however, impractical as it would occupy half of them for an operation that takes place only once per block. We store 512-values of temporary hash state in SSE registers. The LPS transform extracts bytes from SSE to general-purpose registers, performs table lookup, accumulates temporary results in 8 general-purpose 64-bit registers and then reassembles the result back to SSE registers. These operations require SSE 4.1 for maximum efficiency, but may be easily emulated with SSE 2 (which is available on all x86_64 CPUs) with a slight reduction in speed.

Since each output of the compression function depends on its output for the previous block, it is impossible to distribute its calculation for blocks of the same message. This feature is not specific to Stribog and is shared by many hash functions, including GOST R 34.11-94. This precludes the trivial way to distribute hash computation on such a highly-parallel platform as NVIDIA CUDA.

The hash state for Stribog hash function including all temporaries is too large to fit into the register limit of the GPU. Nevertheless, we have tested such an implementation (“simple”) that causes compiler to use local memory. We have also developed a “hybrid” design: a small group of threads within the same warp processes blocks of the data stream sequentially. Data of 512-bit state variables is distributed between threads in units of built-in integer types where possible and

the rest of data is stored in shared memory structures allocated per thread work group. This design avoids synchronization between threads since all threads that process a single data stream are part of the same warp and execute in lock-step.

Even though the native data type for Stribog is 64-bit wide, using 32-bit variables in the hybrid approach yields about 30 % higher performance, because this is the native data unit for the GPU. The simple implementation with one thread per data stream is about as fast as the hybrid 64-bit one.

We have tested different ways to store constant data for the algorithm and found that using texture lookups for multiplication table and constant memory for the C round key constant gives best results. Addition modulo 2^{512} is implemented with carry-lookahead adder using intra-warp parallel scan. Putting a reasonable limit on data stream length, addition in calculation of total message bits processed can be simplified to a single operation on native integers. Bitwise XOR operations for 512-bit values are trivially distributed between threads. These operations take very little time in comparison to the core LPS transform. 32-bit (16 threads per message) and 64-bit (8 threads per message) implementations were developed. 32-bit implementation needs several extra operations, but is faster because no current NVIDIA GPU has native 64-bit integer instructions.

Overall, Stribog hash function is relatively straightforward to implement on common hardware platforms and allows simple modification for parallel implementation.

3. GOST R 34.11-94 hash function

GOST R 34.11-94 [1] hash function design differs in the size of block and hash state, which is 256 bit. The compression function uses GOST 28147-89 symmetrical encryption and different permutations based on linear feedback shift registers. While at the level of most trivial operations it is also limited to XORs and table lookups, it inconsistently uses 1, 2 or 4-byte elements for substitutions and permutations.

Most open source implementations appear to be based on the work of Saarinen [8] that contains the following algorithmic optimizations:

1. 8 4-bit s-boxes are combined into 4 8-bit s-boxes. Their values are stored expanded to 32 bits and pre-rotated by 11 bits to the left which allows to skip this step at runtime. The size of s-box lookup table is 4 KB.
2. Nested compositions of ψ transform are computed from a product matrix and performed on 32-bit values by selecting needed 16-bit parts with masks and shifts.

Our optimized assembly implementation of GOST R 34.11-94 uses the same design as Stribog implementation discussed above. Reduced size of state allows us to fit the block checksum into general-purpose registers. We perform various permutations and transformations with combinations of shifts, masks and shuffles. To avoid loading 128-byte constants from memory, we simulate contiguous masks with combinations of shifts. This implementation depends on extended byte shuffle instruction from SSE3 and can also benefit from SSE4.1.

In contrast with Stribog, the structure of GOST R 34.11-94 hash function doesn't allow for easy distribution of operations between multiple threads. The hybrid approach we implemented uses 4 threads per data stream. This allows us to perform four encryptions in parallel after all keys have been generated. For other operations each thread holds 64 bits of the 256-bit values. To perform permutations, each thread stores its part of value into shared memory and then reads and combines different words from the whole value to form its 64-bit part. We use table-based indexing for the nested ψ transforms, other transforms are easy enough to perform directly. Unfortunately, this approach introduces even more operations into already complex algorithm and does not improve performance in comparison with "simple" design with a single thread per data stream.

GOST R 34.11-94 is more cumbersome to implement than Stribog and requires different platform-dependent approaches. It may be not possible to gain performance with a parallel implementation on many platforms.

4. Results

All implementations were run on Intel Core i7-920 CPU @ 2.67 GHz and NVIDIA GTX 580 GPU. Results are shown for the hybrid 32-bit GOST R 34.11-2012 (Stribog) and simple GOST R 34.11-94 implementations. We used the following implementations for comparison with ours:

- openssl-ccgost — ccgost engine from OpenSSL toolkit [6], version 1.0.1c;
- rhash — RHash utility [7], version 1.2.9;
- kazymyrov — Implementation of hash function Stribog by Oleksandr Kazymyrov [5], revision 400a39cc126d04f660cbcdcddeb8060177a563ea3.

The results are shown in table 1:

OpenSSL implementation of GOST R 34.11-94 is slow because it contains neither algorithmic nor code optimizations. Implementation in RHash is a proper one and contains several assembly optimizations, but most of the code is written in portable C. Our portable implementation of GOST is mostly identical to the one in RHash and optimized version for x86_64 architecture is 30 % faster.

Table 1. Performance results

	GOST R 34.11-94		Stribog-512	
Implementation	MB/s	Cycles/byte	MB/s	Cycles/byte
CPU, 1 core/thread				
Openssl-ccgost	18	143	—	—
Rhash	49	52	—	—
Kazymyrov	—	—	38	67
Lebedev	64	40	94	27
GPU, 8192 data streams				
Lebedev	1697	—	608	—

Implementation of Stribog by Kazymyrov suffers from using big-endian representation, performs several extra data copies and contains several other minor inefficiencies. Our portable implementations for Stribog and GOST R 34.11-94 are comparable in speed, and optimized Stribog is twice as fast.

GPU results are shown for processing 8192 data streams of 16 KB each. These values were determined experimentally, raising them further gives diminishing returns. No clock cycle data is given for the GPU, because the complex nature of thread scheduling for thousands of data streams makes this metric meaningless, at least in comparison with CPUs. Using GPU to process few data streams in parallel is meaningless, because GPU performance for the single data thread is orders of magnitude slower than that of the CPU. This means that GOST R 34.11-2012 implementation has an advantage in that it needs less data streams to fully utilize the GPU since it uses more threads per data stream.

Author of this work possesses no knowledge of other GPU-based implementations for GOST R 34.11-2012 hash function at the moment of writing. As for the GOST R 34.11-94 hash function, the only other known implementation is [4]. We were able to test a demo version of this software on a machine with NVIDIA GTX 550 GPU with the result of $1.38 \cdot 10^6$ passwords/second. Assuming that each password fits in a single data block, processing one password should be roughly equivalent to processing three blocks of data, which gives combined speed of 126 MB/s. Our implementation on the same GPU performs at 269 MB/s. Unfortunately, we're not able to conduct a proper comparison due to the different purpose and closed-source nature of this software.

Despite having smaller source code size, fully unrolled machine code for Stribog hash function is larger than GOST R 34.11-94 on both platforms due to higher number of iterations.

On the CPU GOST R 34.11-2012 is faster because it uses only XOR and data move operations. GOST R 34.11-94, on the other hand, uses a lot of heavy

shuffle operations as its operations don't translate well into CPU instruction set, specifically, SSE. Lookup tables for both algorithms fit into L1 cache. On the GPU the old standard has higher performance because it uses only 4 threads per data stream versus 16 for the new standard. The higher size of substitution tables used by the new standard may also contribute to the performance difference.

We conclude that the implementations we have developed constitute the current state of the art for cryptographic hash functions standardized in the Russian Federation. The new GOST R 34.11-2012 standard possesses the following properties in comparison to the old one:

- It is easier to implement and optimize both algorithmically and on specific hardware.
- Optimized versions are 47 % faster on modern x86_64 CPUs as GOST R 34.11-2012 closely corresponds to the processor instruction set. This speeds up common operations.
- Increased size of state and lookup tables makes it slower on massively parallel hardware, such as GPUs, mainly used for attacks. This also makes implementation on other platforms limited in resources, such as embedded systems, more difficult.

References

1. Government standard of the Russian Federation. Information Technology. Cryptographic data security. Hash function. GOST R 34.11-94.
2. National standard of the Russian Federation. Information Technology. Cryptographic data security. Hash function. GOST R 34.11-2012.
3. *NVIDIA Corporation* NVIDIA CUDA C Programming Guide Version 4.2 4/16/2012. — http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.
4. *InsidePro Software* Extreme GPU Bruteforcer.— <http://www.insidepro.com/eng/egb.shtml>.
5. *Kazymyrov, O., Shevchuk, O.* Implementation of hash function Stribog. — <https://github.com/okazymyrov/stribog>.
6. OpenSSL Project. — <http://openssl.org>.
7. RHash Program. — <http://rhash.anz.ru>.
8. *Saarinen, M.-J. O.* Implementation of GOST R 34.11-94 (with bugfix by David G. Hook). — <http://www.autochthonous.org/crypto/gosthash.tar.gz>.