Math-Net.Ru

Общероссийский математический портал

D. B. Fomin, Реализация блочного шифра типа XSL с MDS-матрицей линейного преобразования на NVIDIA CUDA, *Матем. вопр. криптогр.*, 2015, том 6, выпуск 2, 99–108

DOI: <https://doi.org/10.4213/mvk148>

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: 54.80.97.219

26 апреля 2023 г., 12:57:19

УДК: 519.671

Implementation of an XSL block cipher with MDS-matrix linear transformation on NVIDIA CUDA

D. B. Fomin

Technical committee for standardization (TC 26), Moscow

Получено 16.IX.2014

In this article we consider NVIDIA GPU implementation aspects of an XSL block cipher over the finite field with MDS-matrix linear transformation. We compare obtained results with some other block ciphers.

Реализация блочного шифра типа XSL с MDS-матрицей линейного преобразования на NVIDIA CUDA

Д. Б. Фомин

Технический комитет по стандартизации (ТК 26), Москва

Аннотация. Рассматриваются вопросы, связанные с реализацией на NVIDIA GPU блочного шифра типа XSL над конечным полем с MDS-матрицей линейного преобразования. Проводится сравнение полученных результатов с другими блочными шифрами.

Ключевые слова: CUDA, GPU, AES, GOST, KASUMI, block cipher, fast implementation, CUDA, GPU, AES, GOST, KASUMI, блочный шифр, быстрая реализация

Citation: *Mathematical Aspects of Cryptography*, 2015, vol. 6, no. 2, pp. 99–108 (Russian).

© 2015 D. B. Fomin

Introduction

Modern GPUs are very efficient massively parallel computing devices with big internal memory. The performance of modern graphics cards significantly exceed the CPU performance for algorithms that have high data level parallelism. Therefore, effective implementation of different algorithms on GPUs is an interesting problem.

CUDA is a software and hardware architecture being develop by NVIDIA company. It allows to produce non-graphical computing using GPUs. First NVIDIA GPU which supports general-purpose computing on graphics processing units was presented in 2006, however there were a lot of encryption algorithms implementation with other technology (for example AES on OpenGL [1] in 2006). This researches were very important and interesting in view of allowed unloading of the CPU. Below we shall omit the name of manufacturer company.

In this paper we consider an XSL block cipher implementation over the finite field $GF(2^8)$ with MDS-matrix linear transform and block-size 128 bit. We present different algorithmic optimization of this algorithm. In the second part of the article we provide a comparison of implementation effectiveness for the following algorithms: AES-128, AES-256, GOST 28147-89, KASUMI.

1. CUDA

Modern GPUs have a lot of streaming multiprocessors with 32 CUDA cores. Each CUDA core works at low clock frequencies $\sim 1.5\text{GHz}$ and has 128KB register file (256KB in Kepler architecture GPUs), three different cash type: texture, global (L1), constant.

CUDA architecture use its own model of parallelism, with name: SIMT (single instructions multiple thread). Virtually all threads work in parallel and have the same rights of memory access. Threads are grouped into blocks. The global synchronization between threads in different blocks is generally impossible, and for one block of threads it is performed through a special memory, that is called a shared memory. All threads in a block are divided into groups of size 32, that are called warps. All threads in a warp at the same time perform the same instruction.

Each GPU has the following types of memory: global, shared, registers, constant, texture and local.

Global memory has a large volume (up to 6GB) with very low access speed, and often it is used for data transfer to device. Texture memory is global memory with specific attributes, address to texture memory produced with using special texture cache.

Shared memory is much faster than local and global memory. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory and it is always used as user cache. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules (banks) that may be accessed simultaneously. Therefore, any memory load or store of n addresses that spans b distinct memory banks may be served simultaneously, yielding an effective bandwidth that is b times as high as the bandwidth of a single bank. On devices of compute capability 2.x and 3.x, each multiprocessor has 64KB of on-chip memory that may be partitioned between L1 cache and shared memory. For devices of compute capability 2.x, there are two settings, 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache, on device of compute capability 3.x each multiprocessor also has settings with 32KB shared memory / 32KB L1 cache. By default the 48KB shared memory setting is used. This may be configured during runtime API.

Constant memory has a capacity of 64KB, and cached in a special cache of 8 KB. Constant memory has short latency and high bandwidth when all threads access the same location.

Registers is fast memory, has only thread scope. Local memory does not physically exist and put in global memory by the compiler if using a lot of registers. Therefore, the program should use a limited number of variables.

2. Algorithm implementation parameters

There are different implementation parameters: number of blocks, number of threads in block, encryption data size, compiler flags needed for chosen implementation parameters, cash configuration flags, data allocation, different structures of tables and etc.

All implementations of algorithms considered were written on the CUDA extension of the C language. Constant memory in CUDA program has a very important property: the lifetime of a constant memory variable is equal to the same of the application [2], therefore all used implementation parameters are described by the preprocessor directives. We don't use parallel load in shared memory: on the one hand, in order to be able to compile the program in automatic mode with different values of implementation parameters (for instance, we may vary the numbers of blocks and threads), on the other hand, the time of data loading in shared memory is less than the time of data encryption in block, because the size of encrypting data is equal to or more than 128 MB.

We consider random plaintext encryption in ECB mode, because it may be performed in parallel. Encryption time is a kernel working time (without moving data between GPU memory and RAM).

Implementation parameters such as number of blocks and threads were chosen by "CUDA GPU Occupancy Calculator" in compliance with GPU architecture and other implementation parameters.

By using the Python programming language we wrote a supplementary program that chooses implementation parameters (by means of changing preprocessor directives), run earlier created makefile with required flags (in order to compile and run our application) and arranged data output.

3. Description of the block cipher and its implementation

We denote by XSL block cipher an AES-type cipher with following transformations: X — adding the key, S — nonlinear bijective transform (permutation), L — linear transformation. In this paper we will assume that algorithm receives on input 16-element vectors over $GF(2^8)$.

We say that an invertible linear transform L is MDS [3] transformation of $V_{16}(2^8) \setminus \{0\}$, if

$$\forall a \in V_{16}(2^8) \setminus \{0\} : |\omega(aL) + \omega(a)| \geq 16,$$

where $\omega(a)$ is a number of nonzero elements in the vector a .

In this article we study the algorithm (such as [4]) with encryption function:

$$F_K(x) : V_{16}(2^8) \mapsto V_{16}(2^8),$$

$$F_K(x) = X(K_{10})LSX(K_9) \dots LSX(K_2)LSX(K_1)(x),$$

where $K \in V_{32}(2^8)$ is a master key, and $K_1, \dots, K_{10} \in V_{16}(2^8)$ are round keys, derived from a master key K .

This algorithm may be implemented with lookup tables (t -box) to merge s -box and linear transform. Since L is MDS transform, we need 16 lookup tables. Each table accepts one byte as input and outputs a 128-bit column vector. Let a be a round input, then round output is represented as

$$X(K_i)LS(a) = T_0(a_0) \oplus T_1(a_1) \oplus \dots \oplus T_{15}(a_{15}) \oplus K_i,$$

where $a \in V_{16}(2^8)$, $a = (a_0 \| a_1 \| \dots \| a_{15})$.

Since the size of the tables is too large to be allocated in shared memory, in this implementation we use constant and texture memory. Table 2 presents different parameters for the algorithm implementation.

The most efficient is the implementation that uses texture memory for t -box allocation and registers for ciphered text, because it uses faster memory without thread synchronization. Best performance on GT 610 is 82 MB/sec, on GT 640 — 513 MB/sec and on GTX Titan — 2 273 MB/sec.

Table 1. Hardware and software specification

OS	OpenSUSE 12.3		
CUDA compiler	nvcc ver. 5.5		
Graphics accelerator	NVIDIA GT 610	NVIDIA GT 640	NVIDIA GTX Titan
CUDA-cores	48	384	2688
GPU memory	512 MB	1 GB	6 GB
Processor clock (MHz)	810(1620 Boost)	797	876
CPU	Intel Xeon CPU E5335	Intel Xeon CPU E5335	Intel Core i7-4770K

Table 2. Implementation parameters

T-box allocation	Texture memory, constant memory				
Size of data in thread	8 bit/thread	16 bit/thread	32 bit/thread	64 bit/thread	128 bit/thread
Data allocation	Shared memory				Registers
Round keys allocation	Shared memory				

3.1. Implementation of the algorithm with linear transformation given by LRS over the finite field

Consider linear transform L over the field $GF(2^8)$: $L(a) = f^{16}(a)$, $a \in V_{16}(2^8)$ (such as in [4]), where $f(x)$ is a companion matrix of the characteristic polynomial of LRFS over the field $GF(2^8)$. It is possible to represent L with 16 lookup tables: $Lh_0, \dots, Lh_{15}$ with total size 32KB, if we define $\hat{L}(a) = f^8(a)$:

$$\hat{L}(a) = \hat{L}(a_0 \| a_1 \| \dots \| a_{15}) = (Lh_0(a_0) \oplus Lh_1(a_1) \oplus \dots \oplus Lh_{15}(a_{15})) \| a_0 \| \dots \| a_7,$$

where $Lh_i \in V_8(2^8)$, $i \in \overline{0, 15}$. Each table (t -box) accepts one byte as input and outputs with a 64-bit column vector.

It is also possible to represent s-box and linear transform with 24 tables:

$$Lh_0, \dots, Lh_7, Th_0, Th_1, \dots, Th_{15},$$

with total size 48 KB, where $Th_i \in V_8(2^8)$, $i \in \overline{0, 15}$. Then if $v = (v_0 \| \dots \| v_7) = Th_0(a_0) \oplus Th_1(a_1) \oplus \dots \oplus Th_{15}(a_{15})$, then

$$LS(a) = (Lh_0(v_0) \oplus Lh_1(v_1) \oplus \dots \oplus Lh_7(v_7) \oplus Th_8(a_8) \oplus Th_9(a_9) \oplus \dots \oplus Th_{15}(a_{15})) \| v.$$

Table 3 presents different implementation parameters.

Table 3. Implementation parameters

T-box allocation	Shared memory, constant memory, texture memory
T-box type	16 t-box and one s-box, 24 t-boxes
Size of data in thread	128 bit/thread
Data allocation	Registers
Round key allocation	Shared memory

The best performance is achieved with 24 t -boxes. Results on GT 610 is 202 MB/sec, on GT 640 — 887 MB/sec and on GTX Titan — 5 518 MB/sec.

To compare the performance of discussed algorithm with that of other popular algorithms we had implemented also: GOST 28147-89, KASUMI, AES-128, AES-256. For AES-128 and AES-256 algorithms there are a lot of implementations on GPU (see [8–10]), for GOST 28147-89 and KASUMI algorithms some recommendation for implementation are available (GOST: [11], KASUMI: [12]). It is very difficult to compare different implementations presented in other papers without the source code. Therefore for the comparison of the algorithms mentioned we had to implement them.

4. GOST 28147-89 implementation

A block cipher algorithm GOST 28147-89 [5] has been chosen for comparison due to its simple hardware implementation. GOST is a block cipher with 256-bit key and 64-bit input and output. For our purposes we consider various ways of defining permutation and linear transformation (s -box or t -box) and apply various algorithmic optimization.

Due to the fact that GOST has 4-bit s -boxes, one s -box may be represented with one 64-bit register variable, and s -box transform may be implemented with this register variable and shift with mask.

Table 4. GOST implementation parameters

<i>T</i> -box (<i>s</i> -box) allocation	Constant memory, shared memory, registers
<i>T</i> -box (<i>s</i> -box) type	4-bit <i>s</i> -box 8 tables (8 different <i>s</i> -boxes)/1 table (8 identical <i>s</i> -boxes), 8-bit <i>S</i> -box 4 tables (8 different <i>s</i> -boxes)/1 table (8 identical <i>s</i> -boxes), eight 4/8-bit <i>t</i> -box (<i>s</i> -box with linear transform)
Data allocation	Registers
Round keys allocation	Constant memory, shared memory, registers
Data type	two 32-bit words, 64-bit word

But the shift operation on the considered GPU takes the same number of cycles as the access to the shared memory; so the implementation with eight 4-bit *s*-boxes grouped into four 8-bit *s*-boxes will be more effective.

The best performance was achieved when allocating *s*-boxes in the shared memory, using registers for round keys and 32-bit data type. The result on GT 610 is 479 MB/sec, on GT 640 — 1 512 MB/sec, and on GTX Titan — 9 372 MB/sec (tab. 5). It should be noted that among all implementations, that don't use shared memory for *s*-box (*t*-box) the best performance was obtained for implementation with register *s*-boxes: 2 589 MB/sec on GTX Titan, 283 MB/sec on GT 640 and 108 MB/sec on GT 610.

Table 5. *T*-box (*s*-box) type and maximum performance of GOST

<i>T</i> -box (<i>S</i> -box) type	Performance (MB/sec)		
	GTX Titan	GT 640	GT 610
One 8-bit <i>s</i> -box	9 372	1 512	479
Four 8-bit <i>t</i> -boxes	8 346	1 342	477
Four 8-bit <i>s</i> -boxes	7 787	1 290	443
Eight 4-bit <i>t</i> -box	4 462	726	273

5. KASUMI implementation

KASUMI block cipher algorithm has been chosen for comparison due to its simple hardware implementation on processors with limited resources (such as ARM). KASUMI is a block cipher with 128-bit key and 64-bit input and output. The peculiarity of this algorithm is that the round functions in the main Feistel network are irreversible Feistel-like network transformations. Each round function uses a round key which consists of eight 16-bit subkeys derived from the original 128-bit key using a fixed key schedule such that additional memory for key deploy may not be used.

Table 6. KASUMI implementation parameters

S-box allocation	Constant memory, shared memory, registers		
Data allocation	Registers		
Round key allocation	Registers	Shared memory	Constant memory
Evaluate round keys	No	Yes/No	Yes/No

The best performance was achieved for allocating *s*-boxes in shared memory and using more shared memory to compute round keys. The results on GT 610 is 422 MB/sec, on GT 640 1 689 MB/sec, and on GTX Titan — 13 901 MB/sec are given in Table 6.

6. AES implementation

AES block cipher [7] was chosen due to its structural similarity to the considered algorithm and a large number of publications on its effective GPU implementation. AES is a block cipher with 128 bits input/output and has three different key lengths: 128, 192 and 256 bits.

We take the parameters for our implementation following [8–10], where implementation on CUDA was considered.

Table 7. AES implementation parameters

<i>T</i> -box allocation	Shared memory
Data allocation	Registers
Round keys allocation	Shared memory
Size of data in thread	128 bit/thread

The implementation results of AES-128 and AES-256 are presented in Table 9.

Table 8. Maximum performance of AES algorithms

Algorithm	Performance (MB/sec)		
	GTX Titan	GT 640	GT 610
AES-128	12 786	2 013	681
AES-256	10 851	1 719	573

Table 9. Maximum performance of different algorithms

Algorithm	Performance (MB/sec)		
	GTX Titan	GT 640	GT 610
XSL block cipher	2 273	513	82
XSL block cipher with an LRS-type linear transformation	5 518	887	202
GOST 28147-89	9 372	1 512	479
KASUMI	13 901	1 689	442
AES-128	12 786	2 013	681
AES-256	10 851	1 719	573

Conclusions

We've studied different implementations of several widely used block ciphers and have compared them with the implementation of a new block cipher. The fastest algorithm is AES, because its t -box is small enough to be allocated in the shared memory of GPU. The throughput of implementations of both block cipher GOST 28147-89 and KASUMI on GT 610 as well as on GT 640 are comparable. But on GTX Titan the implementation of KASUMI is the fastest one, because in this case the quantity of registers per thread is optimal for GPU occupancy.

The main property of the discussed algorithm affecting the efficiency of its implementation on GPUs is a comparatively large size of t -boxes that needs to be stored in memory. However, given the prospects of development of NVIDIA graphics devices in the nearest future, this property will be not critical and much

higher speed of encryption may be achieved on new NVIDIA GPUs [13] even without changing approaches to the implementation of the algorithm.

References

- [1] Yamanouchi T., *GPU Gems 3. Chapter 36. AES Encryption and Decryption on the GPU*, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch36.html.
- [2] *CUDA Toolkit documentation*, <http://docs.nvidia.com/>.
- [3] Glukhov M.M., “On mixing linear transforms for block ciphers”, *Математические вопросы криптографии*, **2:2** (2011), 5–40 (in Russian).
- [4] Shishkin V.A., “Design principles of a prospective block cipher with 128 bit block length” (Presentation at RusCrypto’2013) (in Russian <http://www.ruscrypto.ru/resource/summary/rc2013>).
- [5] *GOST 28147-89. Information Processing Systems. Cryptographic Protection. Cryptographic Transformation Algorithm* (in Russian).
- [6] Matsui M., Tokita T., “MISTY, KASUMI and Camellia cipher algorithm development”, ISSN 1345-3041, *Mitsubishi Electric Advance (Mitsubishi Electric corp.)*, **100**, 2–8.
- [7] National Institute of Standards and Technology (NIST), “FIPS-197 : Advanced Encryption Standard (AES)”, 2001.
- [8] Iwai K., Nishikawa N., Kurokawa T., “Acceleration of AES encryption on CUDA GPU”, *Int. J. Networking and Computing*, **2:1** (2012), 131–145.
- [9] Kipper M., Slavkin J., Denisenko D., “Implementing AES on GPU”, University of Toronto, 2009, 10 pp.
- [10] Manavski S.A., “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography”, IEEE Int. Conf. Signal Process. Communic. (ICSPC 2007) (24–27 November 2007, Dubai, United Arab Emirates), 65–68.
- [11] Lebedev P.A., “Comparison of old and new cryptographic hash function National Standards of Russian Federation on CPUs and NVIDIA GPUs”, *Математические вопросы криптографии*, **4:2** (2011), 73–80 (in Russian)
- [12] Cermak B. M., *Programovatelne graficke procesory a jejich aplikace v kryptografii*, Diplomova prace. Masarykova univerzita, 2011 (in Czech)
- [13] “5 things you should know about the new Maxwell GPU architecture”, <https://devblogs.nvidia.com/parallelforall/5-things-you-should-know-about-new-maxwell-gpu-architecture>.