SM4 字节切片高性能实现

龚子睿[1,2]，郭 华[1,2*]，陈 晨[1]，张宇轩[1]，关振宇[1]
（1. 北京航空航天大学网络空间安全学院，北京 100191；
2. 复杂关键软件环境全国重点实验室，北京 100191）

摘要：SM4 是中国自主研发的对称密码算法，目前广泛应用于国家政府部门，但其性能问题制约着算法进一步的推广和应用。在现有 S 盒研究基础上考虑了线性变换 L 的结构特点，将计算 S 盒过程中的仿射变换融合至线性变换中，进而提出了新的 SM4 函数结构。相比于原始的函数结构，提出的新结构在字节切片的适配性上更优，并基于该新结构提出了一种 SM4 字节切片优化方法，可降低线性部分的开销、提升指令吞吐率。使用 GFNI 指令集和 AES-NI 指令集分别实现本文提出的 SM4 字节切片优化方法，在消耗的指令条数和指令吞吐率方面均优于采用相同指令集的优化方法。实验结果表明，所提出的优化方法采用 GFNI 指令集的实现速率最高可达到 35 947 Mbps，优于公开文献的最好结果 30 026 Mbps。在不支持 GFNI 的处理器上，优化方法可使用 AES-NI 指令集实现，可以达到 5 410 Mbps，因此具备一定的通用性。

关键词：SM4 算法；软件优化实现；字节切片；单指令多数据技术；GFNI 指令集
中图分类号：TP309.7 **文献标识码**：A

High Performance Implementation of SM4 Byte Slicing

GONG Zirui[1,2], GUO Hua[1,2*], CHEN Chen[1], ZHANG Yuxuan[1], GUAN Zhenyu[1]
(1. School of Cyber Science and Technology, Beihang University, Beijing 100191, China;
2. State Key Laboratory of Complex & Critical Software Environment (CCSE), Beijing 100191, China)

Abstract: The SM4 is a symmetric cryptographic algorithm independently developed in China, which is widely applied in national governments, but its performance constrains that the algorithm is further popularized and applied. At present, the optimization of the SM4 efficiency is mainly focused on the S-box with the highest computational cost. Scholars propose theoretical optimization methods such as lookup tables and bit slicing, and cooperating with single instruction multiple data technology to improve the high-performance SM4 encryption. On the basis of the S-box and linear transformation L, the affine transformation of calculating S-box process is combined to linear transformation, and a novel SM4 function structure is proposed. Compared with the original function structure, the proposed structure is superior to compatibility for byte slicing. An optimization method for the SM4 byte slicing based on novel structure is proposed, which reduces the overhead of linear part and increase the instruction throughput. Galois field new instruction (GFNI) and AES new instruction (AES-NI) sets are adopted to respectively implement the SM4 byte slicing optimization method proposed in this paper, the proposed optimization method is better than other optimization methods for the same instruction set in the instruction number and instruction throughput. The experimental results show that the encryption speed of the optimization method achieves up to 35 947 Mbps on the GFNI instruction set, which is better than the maximum result of 30 026 Mbps in public literature. the optimization method adopts the AES-NI instruction set to achieve a speed of 5 410 Mbps on unsupported GFNI processors, so it has certain universality.

Keywords: SM4; software optimization implementation; byte slicing; single instruction multiple data; GFNI set

基金项目：国家重点研发计划（2021YFB2700200）；大学生创新创业训练计划（X202210006242）；北京市自然科学基金（4242022）；国家自然科学基金（62172025，U2241213）。

引用格式：龚子睿，郭 华，陈 晨，等. SM4 字节切片高性能实现 [J]. 网络空间安全科学学报，2023，1（3）：86－96.

0　引言

SM4 分组密码算法[1]是我国自主研发的商用密码算法，应用于无线网络通信 WAPI 和 TLS 传输协议。SM4 算法在 2012 年成为密码行业标准，在 2016 年上升为国家标准，并在 2021 年成为 ISO/IEC 国际标准。目前 SM4 算法广泛应用于政府部门，为信息系统提供可靠的数据安全保障，但算法效率上仍与国际通用的 AES 算法存在一定差距，直接影响了 SM4 算法的推广和普及。

查表法是密码算法常见的优化方法，例如 AES 算法可以通过合并字节代替与列混淆操作，进而构造 4 kB 的查找表来降低列混淆的开销[2]。使用大规模的查找表存在遭受缓存计时攻击的风险[3]，学者们更倾向于选择常数时间的优化方法。一些学者将原先 8 bitS 盒拆分成多个 4 bit 查找表，通过常数时间的并行查表指令加速[4-6]。

比特切片是另一种常见的优化方法，且可抵抗缓存计时攻击[7]。比特切片的思想最初由 Biham[8]提出，它将数据重新编排成以比特为单位的形式，使用软件模拟硬件逻辑门运算。Adomnicai 等人[9-10]改进比特切片，基于固定切片技术优化 GIFT 和 AES 算法。比特切片的效率被算法 S 盒的复杂性制约，学者们通过选择函数[11-12]、复合域技术[13-15]或设计特殊的搜索算法[16]来化简 S 盒的逻辑表达式。

还有一种常数时间的优化方法需要借助处理器提供的特殊指令，通过 S 盒的代数表达式进行计算，能够绕过分析复杂的逻辑函数。基于 PERMUTE 指令，Hamburg[17]加速了 AES 算法 S 盒中的有限域求逆；基于 AES-NI（AES New Instructions）指令集[18]，除了能够直接用于加速 AES 算法[19]，还能够加速与 AES 算法 S 盒一致或相似的算法[20-23]；基于 GFNI（Galois Field New Instructions）指令集[24]，能够加速基于有限域的算法 S 盒[6]或者基于有限域运算的密码算法[25-26]。

单指令多数据技术（SIMD，single instruction multiple data）和单指令多线程技术（SIMT，single instruction multiple thread）是算法软件优化过程中的常用技术，借助并行的思想加速算法实现。SIMD 技术常借助 SIMD 指令集，例如单指令多数据流扩展指令集（SSE，streaming SIMD extensions）和高级向量指令集（AVX，advanced vector extensions）；SIMT 技术常见的使用方式是借助 GPU 的并行能力进行加速[27-30]。

针对 SM4 性能提升这一问题，国内外学者进行了广泛的研究，关注点聚焦在 S 盒的计算上。在 CPU 软件实现方向，一部分学者通过查表的方式计算 S 盒；另一部分学者通过表达式计算 S 盒，有操作逻辑门函数的逻辑表达式和基于有限域理论的代数表达式 2 种。针对查表的方法，郎欢等人[31]将 SM4 算法的 S 盒与线性变换合并，构造查找表以节约线性变换的开销，使算法加解密速率达到 2 437 Mbps。Kwon 等人[5]利用向量化查表指令实现 8 bit 并行查表，在 Apple A13 Bionic 平台上达到 8.62 cycles/byte。Guo[6]结合 L1 缓存优化 8 bit 并行查表，将速率进一步提升至 6.74 cycles/byte。关于 S 盒的逻辑表达式，Zhang 等人[32]利用比特切片优化 SM4 算法，在虚拟货币应用环境下的速率提升了 0.8～1.2 倍。Miao 等人[33]改进数据编排方式，将速率提升至 15.26 Gbps。张笑从等人[12]采用选择函数优化 S 盒逻辑门表达式，获得 497 个逻辑门的 S 盒表达式。陈晨等人[14]基于复合域技术将 S 盒化简至 175 个逻辑门。王磊等人[15]基于塔域技术将 S 盒比特切片逻辑门个数降低至 115 个。Xu 等人[16]通过搜索算法和三操作数指令 vpternlogd 优化 S 盒逻辑函数，将速率提升至 30 026 Mbps。对于 S 盒的代数表达式，Saarinen[21]利用 SM4 与 AES 的 S 盒结构相似性，借助 AES-NI 指令集快速实现 SM4 算法 S 盒。Guo[6]在 Saarinen 的基础上采用 GFNI 指令集替代 AES-NI 指令集，速率达到 1.51 cycles/byte，比采用 AES-NI 指令集的版本获得了 127%的提升。

在 SM4 算法 CPU 软件优化方向，学者们针对开销最大的 S 盒部分进行了广泛的研究，将优化重心放在了 S 盒的计算上，提出了许多成熟的优化方法。Guo[6]和 Saarinen[21]采用 GFNI/AES-NI 指令集的优化方法虽然在速率上不及 Xu 等人[16]的比特切片，但它们在处理线性部分的方式以及指令吞吐率上存在着一定的提升空间。当使用 GFNI/AES-NI 指令集优化 SM4 算法时，线性操作除了 S 盒后紧接着的线性变换 L，还有调用 GFNI/AES-NI 指令集前后执行的矩阵乘法（仿射变换）。Guo 和 Saarinen 将这几个线性操作分别处理，未能考虑到线性操作的可合并性。在指令的吞吐率层面，GFNI/AES-NI 指令集的指令延迟较大，虽然 CPU 底层采用流水线的技术来提升指令的吞吐率，但充分利用流水线的条件是需要有多组数据同时处理。上

述方法处理的数据分组数小，指令之间输入输出相关性大，无法充分利用流水线。

本文在现有 S 盒的研究基础上考虑线性变换的优化空间，将 S 盒中的仿射变换与后续的线性变换合并，给出了新的 SM4 轮函数结构，基于此结构提出了 SM4 字节切片优化方法。本文的优化方法改进了目前基于 GFNI/AES-NI 指令集优化方法中线性部分的处理方式，解决了算法线性变换不适合字节切片的问题，并改善了指令吞吐率方面存在的效率问题。实验结果表明，本文提出的优化方法优于采用相同指令集的同类方法，采用 GFNI 指令集配合 AVX512 指令集的实现在 i5-11400 处理器上达到 35 947 Mbps，采用 AES-NI 指令集配合 SSE 指令集的实现在 i5-8265U 处理器上达到 5 410 Mbps。

本文的结构如下：第 1 节介绍 SM4 算法结构，并简单描述了使用 GFNI 指令集和 AES-NI 指令集计算 SM4 算法 S 盒的原理；第 2 节介绍新的 SM4 轮函数表示法以及 SM4 字节切片优化方法的设计；第 3 节介绍基于本文 SM4 优化方法的实现；第 4 节是测试结果；第 5 节总结全文。

1 预备知识

1.1 SM4 算法

SM4 算法的分组和密钥长度均是 128 bit，加密和解密函数的区别仅在于轮密钥输入顺序。设 SM4 明文输入为 $(X_0, X_1, X_2, X_3) \in (Z_2^{32})^4$，32 轮的轮密钥 $rk_i \in Z_2^{32}$，$i \in [0, 31]$，密文输出 $(Y_0, Y_1, Y_2, Y_3) = (X_{35}, X_{34}, X_{33}, X_{32}) \in (Z_2^{32})^4$。则 SM4 算法轮函数可如公式（1）所示：

$$X_{i+4} = X_i \oplus L[\tau(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i)] \tag{1}$$

非线性变换 τ 为 $Z_2^{32} \to Z_2^{32}$ 可逆变换，由 4 个并行的 $Z_2^8 \to Z_2^8$ 的 S 盒变换 S_s 构成。将输入 $X \in Z_2^{32}$ 记为 $(X[0], X[1], X[2], X[3]) \in (Z_2^8)^4$，则 τ 可定义为：

$$\tau(X) = S_s(X[0]) \| S_s(X[1]) \| S_s(X[2]) \| S_s(X[3]) \tag{2}$$

线性变换 L 为 $Z_2^{32} \to Z_2^{32}$ 线性函数，由循环移位和异或运算构成，记 $rotl_i$ 为 32 bit 循环左移 i 位，则线性变换 L 可定义为：

$$L(X) = X \oplus rotl_2(X) \oplus rotl_{10}(X) \oplus rotl_{18}(X) \oplus rotl_{24}(X) \tag{3}$$

在文献［34］中，Liu 等人破解了 SM4 算法 S 盒的代数结构，由一个有限域 $GF(2)^8$ 求逆 I_s 以及前后的 2 个 8 bit 仿射变换 $\boldsymbol{M}_s \cdot x \oplus \boldsymbol{C}_s$ 构成，所使用的不可约多项式为 $p_s = x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$，具体结构如公式（4）所示：

$$S_s(x) = \boldsymbol{M}_s \cdot I_s(\boldsymbol{M}_s \cdot x \oplus \boldsymbol{C}_s) \oplus \boldsymbol{C}_s \tag{4}$$

公式（3）所示的线性变换 L 可以写成 32×32 的矩阵形式，该矩阵是一个分块矩阵，结构形式如公式（5）的 $\boldsymbol{M}_L$。分块矩阵 $\boldsymbol{M}_L$ 的分块元素是 8×8 的矩阵 $\boldsymbol{B}_1$、$\boldsymbol{B}_2$ 和 $\boldsymbol{B}_3$，且 $\boldsymbol{B}_3 = \boldsymbol{B}_1 + \boldsymbol{B}_2$，具体取值如公式（6）所示：

$$\boldsymbol{M}_L = \begin{bmatrix} \boldsymbol{B}_1 & \boldsymbol{B}_2 & \boldsymbol{B}_2 & \boldsymbol{B}_3 \\ \boldsymbol{B}_3 & \boldsymbol{B}_1 & \boldsymbol{B}_2 & \boldsymbol{B}_2 \\ \boldsymbol{B}_2 & \boldsymbol{B}_3 & \boldsymbol{B}_1 & \boldsymbol{B}_2 \\ \boldsymbol{B}_2 & \boldsymbol{B}_2 & \boldsymbol{B}_3 & \boldsymbol{B}_1 \end{bmatrix} \tag{5}$$

$$\boldsymbol{B}_1 = \begin{bmatrix} 10100000 \\ 01010000 \\ 00101000 \\ 00010100 \\ 00001010 \\ 00000101 \\ 00000010 \\ 00000001 \end{bmatrix}, \boldsymbol{B}_2 = \begin{bmatrix} 00100000 \\ 00010000 \\ 00001000 \\ 00000100 \\ 00000010 \\ 00000001 \\ 10000000 \\ 01000000 \end{bmatrix},$$

$$\boldsymbol{B}_3 = \begin{bmatrix} 10000000 \\ 01000000 \\ 00100000 \\ 00010000 \\ 00001000 \\ 00000100 \\ 10000010 \\ 01000001 \end{bmatrix} \tag{6}$$

1.2 使用 GFNI 指令集加速 SM4 算法 S 盒

GFNI 指令集于 Intel 第三代 Xeon 可扩展处理器中被引入，旨在加速密码算法和安全应用程序。GFNI 指令集共提供 3 条指令来实现 AES 算法对应的有限域运算，它们在 Ice Lake Xeon 平台均具有 3 周期的延迟和 1 周期的 CPI（Clock Per Instruction），具体的指令细节如表 1 所示。

表 1 GFNI 指令集

Table 1 Galois Field New Instructions

汇编指令	C 语言接口	指令介绍
vgf2p8mulb	_mm512_gf2p8mul_epi8	有限域乘法
vgf2p8affineinvqb	_mm512_gf2p8affineinv_epi64_epi8	求逆仿射变换
vgf2p8affineqb	_mm512_gf2p8affine_epi64_epi8	仿射变换

计算 S 盒需要用到的指令为 vgf2p8affineinvqb 和 vgf2p8affineqb，为表示方便，将它们分别简写成 AffineInv 和 Affine。记求逆函数为 I_a，对应的 AES 算法有限域不可约多项式 $p_a = x^8+x^4+x^3+x+1$，指令对应的操作如公式（7）所示：

$$\begin{aligned}&\mathrm{AffineInv}(x, \boldsymbol{A}, C) = \boldsymbol{A}\cdot I_a(x)\oplus \boldsymbol{C}\\&\mathrm{Affine}(x, \boldsymbol{A}, C) = \boldsymbol{A}\cdot x\oplus \boldsymbol{C}\end{aligned} \tag{7}$$

Guo[6] 给出了利用 GFNI 指令集加速 SM4 算法 S 盒的方法，原理是将 S 盒中的求逆函数 I_s 通过同构映射函数 π 转化成 I_a。记函数 $\pi: x_s \to x_a$ 是将 SM4 算法的有限域映射到 AES 算法的有限域的函数，进而 SM4 算法的有限域的求逆 $I_s(x) = \pi^{-1}\cdot I_a(\pi(x))$，结合第 1.1 节的公式（4）中给出的 S 盒代数结构可得到最终的 S 盒表达式：

$$\begin{aligned}S_s(x) = (\boldsymbol{M}_s\cdot\pi^{-1})\cdot I_a[(\pi\cdot\boldsymbol{M}_s)\cdot x\oplus(\pi\cdot\boldsymbol{C}_s)]\oplus\\ C_s = \boldsymbol{M}_{G2}\cdot I_a(\boldsymbol{M}_{G1}\cdot x\oplus\boldsymbol{C}_{G1})\oplus\boldsymbol{C}_{G2}\end{aligned} \tag{8}$$

计算 S 盒需要 1 条 vgf2p8affineqb 指令和 1 条 vgf2p8affineinvqb 指令，前者计算内层的仿射变换，后者计算求逆和外层的仿射变换。

1.3　使用 AES-NI 指令集加速 SM4 算法 S 盒

AES-NI 指令集提供 AES 算法的硬件实现，总共提供 7 条指令完成 AES 加解密以及密钥拓展操作，具体的汇编指令和 C 语言接口如表 2 所示。

表 2　AES-NI 指令集

Table 2　AES New Instructions

汇编指令	C 语言接口	指令介绍
aes[enc/dec]	_mm_aes[enc/dec]_si128	AES 轮函数加/解密
aes[enc/dec]last	_mm_aes[enc/dec]last_si128	AES 最后一轮加/解密
aesimc	_mm_aesimc_si128	AES 逆列混淆
aeskeygenassist	_mm_aeskeygenassist_si128	AES 密钥拓展

使用 AES-NI 指令集加速 SM4 算法 S 盒需要用到的指令是 aesenclast，该指令在 Intel Skylake 架构下具有 4 周期的延迟和 1 周期的 CPI，借助该指令可以快速计算 AES 算法 S 盒，进而计算有限域求逆。因为 AES 最后一轮加密的步骤是行移位、字节代替和轮密钥加，若在调用该指令前预先执行逆行移位，并将输入的轮密钥置零，则等价于调用 AES 算法的 S 盒。又因为 AES 算法的 S 盒结构为 $S_a(x) = \boldsymbol{M}_a\cdot I_a(x)\oplus\boldsymbol{C}_a$，与本文 1.2 节的推导过程类似，可进一步推出计算 SM4 算法的 S 盒表达式：

$$\begin{aligned}S_s(x) = (\boldsymbol{M}_s\cdot\pi^{-1}\cdot\boldsymbol{M}_a)\cdot S_a[(\pi\cdot\boldsymbol{M}_s)\cdot x\oplus\\ (\pi\cdot\boldsymbol{C}_s)]\oplus(\boldsymbol{M}_s\cdot\pi^{-1}\cdot\boldsymbol{M}_a^{-1}\cdot\boldsymbol{C}_a\oplus\boldsymbol{C}_s) =\\ \boldsymbol{M}_{A2}\cdot S_a(\boldsymbol{M}_{A1}\cdot x\oplus\boldsymbol{C}_{A1})\oplus\boldsymbol{C}_{A2}\end{aligned} \tag{9}$$

针对计算过程中的 8 bit 仿射变换，将 8 bit 数拆分成高 4 bit 和低 4 bit，分别使用重排指令查表完成运算，最后通过异或将结果合并。

2　新函数结构与字节切片优化方法设计

目前针对 SM4 算法的软件优化均将 S 盒和线性变换 L 分开考虑，在计算 SM4 算法 S 盒时，无论采用公式（8）还是公式（9），都是“仿射-f-仿射”（f 为非线性函数）的形式。若将仿射变换和后续的线性变换 L 合并，可构造 SM4 算法新函数结构，降低线性操作的开销。新函数结构的线性变换与 AES 算法的列混淆相似，是一个分块矩阵乘法。与由 32 bit 整体循环移位构成的线性变换 L 不同，分块矩阵乘法的表示形式将操作拆分成字节内的仿射变换与字节间的异或扩散，操作的单位是字节。由于 S 盒的操作粒度同样也是字节，故构造的新函数结构以字节作为操作的基本单位，在字节切片上具有优良的适配性，基于该新结构设计的字节切片优化方法能够获得较大的效率提升。

2.1　构造新的 SM4 函数结构

为了表示方便，将 SM4 算法的非线性变换 τ 改写成如公式（10）的形式：

$$\tau(X) = \boldsymbol{M}_2\cdot f(\boldsymbol{M}_1\cdot X\oplus\boldsymbol{C}_1)\oplus\boldsymbol{C}_2 \tag{10}$$

公式（10）中的 $\boldsymbol{M}_1$、$\boldsymbol{M}_2$ 是 32×32 矩阵，$\boldsymbol{C}_1$、$\boldsymbol{C}_2$ 是 32×1 列向量，f 是一个可以高效计算非线性函数。根据所采用优化算法的不同，这些参数可以有多种不同的取值。在本文 1.2 节的 GFNI 指令集优化算法中，f 是公式（11）所示的有限域求逆，$\boldsymbol{M}_1$、$\boldsymbol{M}_2$、$\boldsymbol{C}_1$、$\boldsymbol{C}_2$ 的取值如公式（12）所示：

$$f(X) = I_a(X[0])\|I_a(X[1])\|I_a(X[2])\|I_a(X[3]) \tag{11}$$

$$\boldsymbol{M}_{1|2} = \begin{bmatrix}M_{G1|2}&0&0&0\\0&M_{G1|2}&0&0\\0&0&M_{G1|2}&0\\0&0&0&M_{G1|2}\end{bmatrix},\quad \boldsymbol{C}_{1|2} = \begin{bmatrix}C_{G1|2}\\C_{G1|2}\\C_{G1|2}\\C_{G1|2}\end{bmatrix} \tag{12}$$

结合公式（5）给出的线性变换的矩阵形式，SM4 算法迭代的轮函数可改写为：

$$X_{i+4} = X_i \oplus \boldsymbol{M}_L \cdot (\boldsymbol{M}_2 \cdot f(\boldsymbol{M}_1 \cdot (X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i) \oplus \boldsymbol{C}_1) \oplus \boldsymbol{C}_2) \quad (13)$$

公式（13）与公式（1）的区别主要是将公式（2）的非线性变换 τ 展开，并将公式（3）的线性变换 L 改写成矩阵 $\boldsymbol{M}_L$ 的乘法形式。将所有线性操作均改写成矩阵乘法可以便于后续分析与合并。在式（13）中，矩阵 $\boldsymbol{M}_2$ 和矩阵 $\boldsymbol{M}_L$ 显然能够合并，但由于矩阵 $\boldsymbol{M}_1$ 和矩阵 $\boldsymbol{M}_2$ 间隔一个非线性函数 f，故矩阵 $\boldsymbol{M}_1$ 无法直接跨越 f 和后续的矩阵 $\boldsymbol{M}_2$ 合并。但因为不同轮的 f 之间全部是由异或和矩阵乘法构成的操作，故它可以上移和上一轮的矩阵 $\boldsymbol{M}_2$ 合并。

根据矩阵乘法的线性性质，可将矩阵 $\boldsymbol{M}_1$ 的乘法运算与上一步的异或运算互换位置，即：

$$\boldsymbol{M}_1 \cdot (X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i) \oplus \boldsymbol{C}_1 = (\boldsymbol{M}_1 \cdot X_{i+1}) \oplus (\boldsymbol{M}_1 \cdot X_{i+2}) \oplus (\boldsymbol{M}_1 \cdot X_{i+3}) \oplus (\boldsymbol{M}_1 \cdot rk_i \oplus \boldsymbol{C}_1) = X'_{i+1} \oplus X'_{i+2} \oplus X'_{i+3} \oplus rk'_i \quad (14)$$

令 $X'_i = \boldsymbol{M}_1 \cdot X_i$，$rk'_i = \boldsymbol{M}_1 \cdot rk_i \oplus \boldsymbol{C}_1$，式（14）可视为先计算出乘法的结果 X'_i，之后再进行异或运算。因为 X_i 具备一定的迭代关系，故 X'_i 同样具备类似的性质。因此，可改写 SM4 轮函数的输入输出，在 32 轮循环使用 X'_i 替代原先的 X_i，使用 rk'_i 替代原先的 rk_i。为了使算法轮函数结束时生成 X'_{i+4}，需要对算法第 i 轮的输出结果 X_{i+4} 乘上矩阵 $\boldsymbol{M}_1$。记算法线性变换 L（矩阵 $\boldsymbol{M}_L$ 乘法）输出为 T，则有：

$$X'_{i+4} = \boldsymbol{M}_1 \cdot X_{i+4} = \boldsymbol{M}_1 \cdot (X_i \oplus T) = \boldsymbol{M}_1 \cdot X_i \oplus \boldsymbol{M}_1 \cdot T = X'_i \oplus \boldsymbol{M}_1 \cdot T \quad (15)$$

记 $t = X'_{i+1} \oplus X'_{i+2} \oplus X'_{i+3} \oplus rk'_i$，结合公式（1）中的 SM4 算法轮函数表示，公式（5）中 SM4 算法线性变换的矩阵形式、公式（10）中改写的非线性变换形式和公式（15）中的推导结果，可得到如公式（16）所示的推导结果。此时，$\boldsymbol{M}_2$、$\boldsymbol{M}_L$、$\boldsymbol{M}_1$ 这 3 个矩阵的乘法操作依次执行，可将它们合并为一个矩阵 $\boldsymbol{M}'_L$；同理可以将 $\boldsymbol{M}_1 \cdot \boldsymbol{M}_L \cdot \boldsymbol{C}_2$ 合并为 $\boldsymbol{C}'$。

$$X'_{i+4} = X'_i \oplus \boldsymbol{M}_1 \cdot \boldsymbol{M}_L \cdot (\boldsymbol{M}_2 \cdot f(t) \oplus \boldsymbol{C}_2) = X'_i \oplus \boldsymbol{M}_1 \cdot \boldsymbol{M}_L \cdot \boldsymbol{M}_2 \cdot f(t) \oplus \boldsymbol{M}_1 \cdot \boldsymbol{M}_L \cdot \boldsymbol{C}_2 = X'_i \oplus \boldsymbol{M}'_L \cdot f(t) \oplus \boldsymbol{C}' \quad (16)$$

最终可得到如公式（17）所示的 SM4 算法新的轮函数形式：

$$X'_{i+4} = X'_i \oplus \boldsymbol{M}'_L \cdot f(X'_{i+1} \oplus X'_{i+2} \oplus X'_{i+3} \oplus rk'_i) \oplus \boldsymbol{C}' \quad (17)$$

SM4 算法输入（X_0，X_1，X_2，X_3），需要对输入乘以矩阵 $\boldsymbol{M}_1$ 转化成（X'_0，X'_1，X'_2，X'_3）；对于第 i 轮的轮密钥 rk_i，需要预处理成 $rk'_i = \boldsymbol{M}_1 \cdot rk_i \oplus \boldsymbol{C}_1$；在迭代计算过程，按照公式（17）所示的轮函数进行迭代；在最终结果输出前，需要将迭代计算出的 X'_i 转化回 X_i，这可通过乘以矩阵 $\boldsymbol{M}_1$ 的逆矩阵 $\boldsymbol{M}_1^{-1}$ 实现。因此，可将 SM4 加密函数改写成算法 1 的形式。

算法 1：SM4 新函数结构

输入：plaintext，rk_i

输出：ciphertext

FOR $i=0$ to 31 do

$rk'_i \leftarrow \boldsymbol{M}_1 \cdot rk_i \oplus \boldsymbol{C}_1$

END FOR

X_0，X_1，X_2，$X_3 \leftarrow$ plaintext

X'_0，X'_1，X'_2，$X'_3 \leftarrow \boldsymbol{M}_1 \cdot (X_0, X_1, X_2, X_3)$

FOR $i=0$ to 31 do

$t_1 \leftarrow X'_{i+1} \oplus X'_{i+2} \oplus X'_{i+3} \oplus rk'_i$

$t_2 \leftarrow f(t_1)$

$t_3 \leftarrow \boldsymbol{M}'_L \cdot t_2 \oplus \boldsymbol{C}'$

$X_{i+4} \leftarrow X'_i \oplus t_3$

END FOR

X_{32}，X_{33}，X_{34}，$X_{35} \leftarrow M_1^{-1} \cdot (X'_{32}, X'_{33}, X'_{34}, X'_{35})$

ciphertext $\leftarrow X_{35}, X_{34}, X_{33}, X_{32}$

由于矩阵 $\boldsymbol{M}_2$、$\boldsymbol{M}_L$、$\boldsymbol{M}_1$、$\boldsymbol{C}_2$ 均为分块矩阵的形式，并且矩阵 $\boldsymbol{M}_1$、$\boldsymbol{M}_2$ 均是对角阵，故此时矩阵 $\boldsymbol{M}'_L$ 和矩阵 $\boldsymbol{C}'$ 的形式如公式（18）所示，且矩阵 $\boldsymbol{B}'_1$、$\boldsymbol{B}'_2$、$\boldsymbol{B}'_3$ 同样满足 $\boldsymbol{B}'_3 = \boldsymbol{B}'_1 + \boldsymbol{B}'_2$，这与 AES 算法的列混淆具有极高的相似度。

$$\boldsymbol{M}'_L = \begin{bmatrix} \boldsymbol{B}'_1 & \boldsymbol{B}'_2 & \boldsymbol{B}'_2 & \boldsymbol{B}'_3 \\ \boldsymbol{B}'_3 & \boldsymbol{B}'_1 & \boldsymbol{B}'_2 & \boldsymbol{B}'_2 \\ \boldsymbol{B}'_2 & \boldsymbol{B}'_3 & \boldsymbol{B}'_1 & \boldsymbol{B}'_2 \\ \boldsymbol{B}'_2 & \boldsymbol{B}'_2 & \boldsymbol{B}'_3 & \boldsymbol{B}'_1 \end{bmatrix}, \boldsymbol{C}' = \begin{bmatrix} \boldsymbol{c}' \\ \boldsymbol{c}' \\ \boldsymbol{c}' \\ \boldsymbol{c}' \end{bmatrix} \quad (18)$$

矩阵 $\boldsymbol{M}'_L$ 对 32 bit 字 X 的乘法运算可以通过公式（19）的表达式计算，其中符号 $B'_i X$ 定义为 $\boldsymbol{B}'_i \cdot X[0] \| \boldsymbol{B}'_i \cdot X[1] \| \boldsymbol{B}'_i \cdot X[2] \| \boldsymbol{B}'_i \cdot X[3]$。

$$\boldsymbol{M}'_L \cdot X = \boldsymbol{B}'_1 X \oplus rotl_8(\boldsymbol{B}'_2 X) \oplus rotl_{16}(\boldsymbol{B}'_2 X) \oplus rotl_{24}(\boldsymbol{B}'_1 X \oplus \boldsymbol{B}'_2 X) \quad (19)$$

和原始结构相比，本文提出的 SM4 算法新结

构具有如下几点优势：

1）新结构中公式（19）的循环移位是 8 的整数倍，在不支持循环移位的处理器上可以通过 1 条字节重排指令快速实现，而线性变换 L 的循环移位则需要左、右移位和异或共 3 条指令；

2）新结构将 SM4 算法 S 盒中串行执行的 2 次仿射变换转变成互不依赖的矩阵 $\boldsymbol{B}'_1$、$\boldsymbol{B}'_2$ 的乘法运算，软件实现时能更好利用处理器的流水线；

3）新结构将线性变换 L 转变成字节粒度的矩阵运算，解决了线性变换不适合字节切片的问题。

2.2　SM4 字节切片方法分析设计

在字节切片下，算法输入的明文数据有特殊的排布方式，如图 1 所示，和原始排布相比，相当于将每个数据分组的第 i 字节提取至同一个寄存器中。SM4 算法字节切片使用 16 个寄存器存储数据，第 i 个寄存器存放分组的第 i 字节。在寄存器长度大于 8 bit 的平台上可以并行处理多分组数据，若寄存器长度为 w 字节，则并行分组数为 w。对于轮密钥，由于采用了字节切片的操作，轮密钥也需要相应地进行切片，将 32 bit 的轮密钥拆分成 4 个 8 bit 的数据，并进行复制扩展以配合明文数据的结构。

字节切片能够消除字节间数据移动产生的开销，当循环移位数为 8 的整数倍时，每个字节被分散至不同的寄存器之中，可以通过调换寄存器的顺序隐式完成循环移位。在 SM4 算法的线性变换 L 中，循环移动的位数分别是 2、10、18、24。由于移位数不完全是 8 的整数倍，不适合字节切片。

和原始结构相比，本文 2.1 节中提出的新函数结构更能发挥出字节切片的优势，新函数结构将非

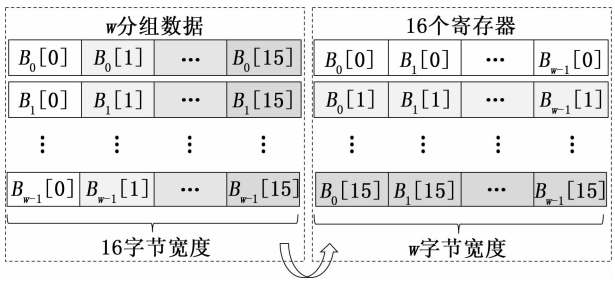

图 1　字节切片

Fig. 1　Byte Slicing

线性变换 τ 中的 2 个矩阵乘法与线性变换 L 融合后得到矩阵 $\boldsymbol{M}'_L$。虽然有 $\boldsymbol{B}'_1$、$\boldsymbol{B}'_2$、$\boldsymbol{B}'_3$ 这 3 个矩阵乘法，但根据 $\boldsymbol{B}'_3=\boldsymbol{B}'_1+\boldsymbol{B}'_2$ 的关系式还可将乘法次数降低至 2 次，故在矩阵乘法次数上与修改前无异。基于新结构进行的字节切片优化可进一步降低线性部分开销，获得更大的效率提升。

字节切片优化后，SM4 新结构中矩阵 $\boldsymbol{M}'_L$ 的乘法运算可通过图 2 的方式计算，只包含 8×8 矩阵乘法和异或操作。如图 2 所示，模块输入 4 个 8 bit 字 $T[0]$ $T[1]$ $T[2]$ $T[3]$，输出 4 个 8 *bit* 字 $out[0]$ $out[1]$ $out[2]$ $out[3]$。线段箭头表示数据的流向，线的走势意为数据的走向。线与线之间的空心圆点是连线交点，表示数据均源自或去往同一操作模块。输入 $T[i]$ 先计算矩阵乘法得到 $\boldsymbol{B}'_1 \cdot T[i]$ 和 $\boldsymbol{B}'_2 \cdot T[i]$，随后将结果异或得到 $\boldsymbol{B}'_3 \cdot T[i]$，最后经过异或运算得到输出 $out[i]$。矩阵 $\boldsymbol{M}'_L$ 的乘法运算总共需要 16 次字节异或和 8 次 8×8 的矩阵乘法。

因为存在高效的计算 8 bit 仿射变换的算法，故计算 $\boldsymbol{M}'_L \cdot X \oplus \boldsymbol{C}'$ 中的矩阵乘法和异或运算可以同时进行，过程中的子运算为 $\boldsymbol{B}'_i \cdot X[j] \oplus \boldsymbol{c}'$。

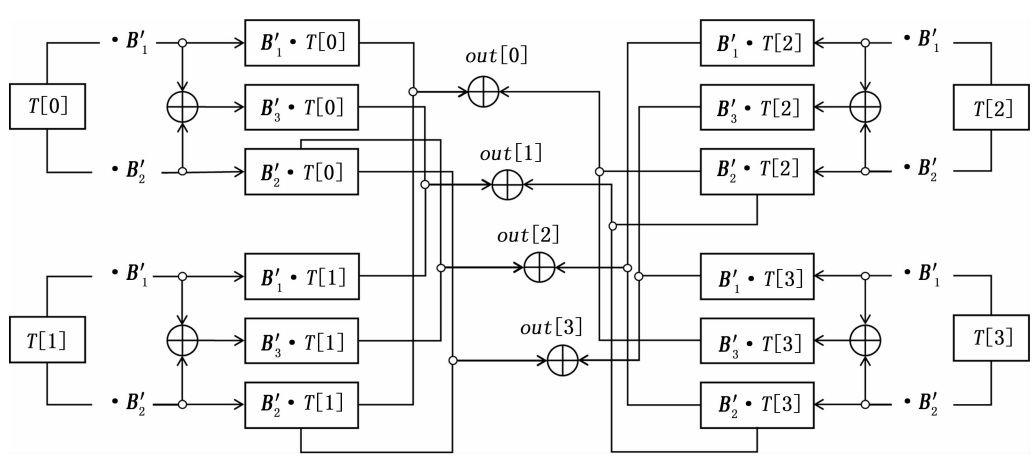

图 2　线性变换模块

Fig. 2　Linear Transformation Module

字节切片形式的 SM4 算法轮函数可写成如算法 2 的形式。

算法 2：字节切片形式的 SM4 算法轮函数

输入：$(X'_i, X'_{i+1}, X'_{i+2}, X'_{i+3}), rk'_i$

输出：X'_{i+4}

$t \leftarrow X'_{i+1} \oplus X'_{i+2} \oplus X'_{i+3} \oplus rk'_i$

$t \leftarrow f(t)$

FOR $i=1$ to 2 do

FOR $j=0$ to 3 do

$t_{i,j} \leftarrow \boldsymbol{B}'_i \cdot t[j] \oplus \boldsymbol{c}'$

END FOR

END FOR

$X'_{i+4}[0] \leftarrow X'_i[0] \oplus t_{1,0} \oplus t_{2,1} \oplus t_{2,2} \oplus (t_{1,3} \oplus t_{2,3})$

$X'_{i+4}[1] \leftarrow X'_i[1] \oplus t_{1,1} \oplus t_{2,2} \oplus t_{2,3} \oplus (t_{1,0} \oplus t_{2,0})$

$X'_{i+4}[2] \leftarrow X'_i[2] \oplus t_{1,2} \oplus t_{2,3} \oplus t_{2,0} \oplus (t_{1,1} \oplus t_{2,1})$

$X'_{i+4}[3] \leftarrow X'_i[3] \oplus t_{1,3} \oplus t_{2,0} \oplus t_{2,1} \oplus (t_{1,2} \oplus t_{2,2})$

3 基于新结构的字节切片方法设计实现

3.1 字节切片数据编排

因为输入的 16 字节分组数据在内存中的排布顺序形如（$B[0]\ B[1] \ldots B[15]$），并不是符合字节切片的数据形式，所以需要进行数据编排操作，将数据 $B[j]$ 聚拢以便后续并行处理。Intel 处理器平台提供了 vpunpck［h/l］dq、vpunpck［h/l］qdq、vpshufb 这几条操作，数据的 SIMD 指令、开销与逻辑指令基本相同，实现字节切片的数据编排共需要 80 条这些指令。在输出前，需要对数据进行反编排操作，将字节切片形式转化为普通形式。因为数据编排从另一方面可看作矩阵转置算法，故编排与反编排算法完全一致，输出前再一次调用编排算法即可完成反编排操作。

字节切片的流程分为 2 次 32 bit 切片与 1 次数据重排，图 3 展示了数据编排过程中寄存器 R_0 数据的变换过程。图中以 128 bit 长度的寄存器为例，通过数据编排将 16 个分组的 $B[0]$ 聚集。$B_i[0] \ldots B_i[15]$ 对应分组 i 的 16 个 $Byte$，针对 256、512 位宽的寄存器横向扩展即可。第一次 32 比特切片后，寄存器中存储的是 4 个 32 比特字，每个形如 $B_i[0]\ B_i[1]\ B_i[2]\ B_i[3]$ 的 32 比特字对应的 $SM4$ 算法的 32 比特字 X_0。接着通过数据重排将 4 个 $B_i[0]$ 聚拢，$B_i[1]$、$B_i[2]$、$B_i[3]$ 同理。最后再次进行 32 比特切片，得到 16 个分组的 $B[0]$。

32 比特切片的数据编排可视为从每个 16Byte 的数据分组中取出对应 32 比特字 X_0、X_1、X_2、X_3，可将该操作拆分成 $ExtractX_0$、$ExtractX_1$、$ExtractX_2$、$ExtractX_3$ 这 4 个函数。操作对应的 4 个函数如下所示，输入 4 个寄存器数据（$r_0\ r_1\ r_2\ r_3$），取出对应的 X_i。由于存在可以复用的中间数据，故一次 32 比特切片操作总计消耗 8 条指令。为表示方便，公式中将 vpunpck［h/l］dq 指令和 vpunpck［h/l］qdq 指令简写成 pck［h/l］和 pck［h/l］q。

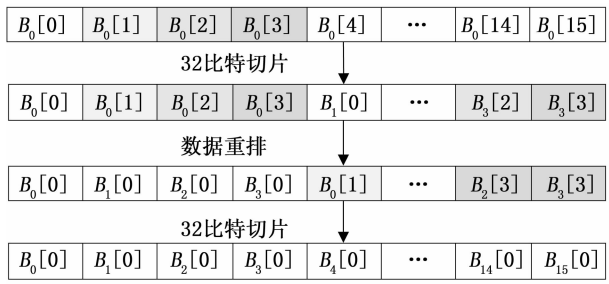

图 3 字节切片数据编排

Fig. 3 Data Transformation of Byte Slicing

$ExtractX_0(r_0\ r_1\ r_2\ r_3) = \text{pckhq}(\text{pckh}(r_3, r_2), \text{pckh}(r_1, r_0))$

$ExtractX_1(r_0\ r_1\ r_2\ r_3) = \text{pcklq}(\text{pckh}(r_3, r_2), \text{pckh}(r_1, r_0))$

$ExtractX_2(r_0\ r_1\ r_2\ r_3) = \text{pckhq}(\text{pckl}(r_3, r_2), \text{pckl}(r_1, r_0))$

$ExtractX_3(r_0\ r_1\ r_2\ r_3) = \text{pcklq}(\text{pckl}(r_3, r_2), \text{pckl}(r_1, r_0))$

数据重排操作是将寄存器中数据以字节为单位进行重新排列，对每个寄存器分别执行一次重排操作，通过 vpshufb 指令配合一个索引表 MapIndex 可快速完成。

综上，字节切片数据编排算法如下所示。输入的 $d[i]$ 对应第 i 个寄存器中的数据，16 个 $d[i]$ 表示 16 个寄存器数据。输出的 $B[i]$ 对应全部数据分组的第 i 字节，16 个 $B[i]$ 代表字节切片后的 16 个字节。

算法 3：字节切片数据编排

输入：$d[16]$

输出：$B[16]$

$t[16] \leftarrow [0, \ldots 0]$

FOR $i=0$ to 3 do

$a_0, a_1, a_2, a_3 = 0+4i, 1+4i, 2+4i, 3+4i$

$t[0+i] \leftarrow ExtractX_0(d[a_0], d[a_1], d[a_2], d[a_3])$

$t[4+i] \leftarrow ExtractX_1(d[a_0], d[a_1], d[a_2], d[a_3])$

$t[8+i] \leftarrow ExtractX_2(d[a_0], d[a_1], d[a_2], d[a_3])$

$t[12+i] \leftarrow ExtractX_3(d[a_0], d[a_1], d[a_2], d[a_3])$

END FOR
FOR i=0 to 15 do
$t[i]$←vpshufb(MapIndex, $t[i]$)
END FOR
FOR i=0 to 3 do
$a_0, a_1, a_2, a_3 = 0+4i, 1+4i, 2+4i, 3+4i$
$B[0+i] \leftarrow ExtractX_0(t[a_0], t[a_1], t[a_2], t[a_3])$
$B[1+i] \leftarrow ExtractX_1(t[a_0], t[a_1], t[a_2], t[a_3])$
$B[2+i] \leftarrow ExtractX_2(t[a_0], t[a_1], t[a_2], t[a_3])$
$B[3+i] \leftarrow ExtractX_3(t[a_0], t[a_1], t[a_2], t[a_3])$
END FOR

3.2　使用 GFNI 指令集加速 SM4 轮函数

得益于 GFNI 指令集能够快速实现求逆－仿射（vgf2p8affineinvqb）操作，公式（17）中的 I_a 与后续的仿射变换 $\boldsymbol{B}'_i \cdot x \oplus \boldsymbol{c}'$ 可通过调用 1 次求逆－仿射指令完成，同时 $\boldsymbol{B}'_3 \cdot I_a(x)$ 可利用关系式 $\boldsymbol{B}'_3 \cdot I_a(x) = \boldsymbol{B}'_1 \cdot I_a(x) \oplus \boldsymbol{B}'_2 \cdot I_a(x)$ 调用 1 条异或指令实现。虽然 $\boldsymbol{B}'_3 \cdot I_a(x)$ 同样可以利用求逆－仿射指令完成，但其延迟是异或操作的 3 倍，开销更大，故最终选用异或实现。特别地，仿射变换中与常数的异或操作可以融入求逆－仿射指令中，无需额外调用异或指令。

表 3 从指令开销上对比了 Guo[6] 提出的方法和本文基于 GFNI 指令集的字节切片优化方法。在 Guo 提出的 GFNI 指令集优化方法中，对数据采取 32 比特字切片，在 GFNI 指令集配合 AVX512 指令集的实现中，并行的分组数为 16，按照公式（1）迭代一轮需要 1 次仿射、1 次求逆－仿射、8 次异或操作和 4 次循环移位操作，平均而言每分组需要 0.062 5 次仿射、0.062 5 次求逆－仿射、0.5 次异或和 0.25 次循环移位操作。对于本文的字节切片优化方法，在 GFNI 指令集配合 AVX512 指令集的实现中，并行分组数为 64，按照公式（17）迭代一轮需要 8 次求逆－仿射和 32 次异或操作，平均每分组需要 0.125 次求逆－仿射和 0.5 次异或操作。

指令 vgf2p8affineqb 和 vgf2p8affineinvqb 在 Ice Lake Xeon 平台上开销相同，故可把二者视为等价。由表 3 的对比可知，针对平均每分组使用的指令数，本文的方法无需进行循环移位操作，比 Guo 提出的方法具备更少的指令条数。此外，在 Guo 的方法中 vgf2p8affineqb 和 vgf2p8affineinvqb 指令需要依次调用，前者的结果为后者的输入，需要 6 个时钟周期的延迟。本文提出的方法使用 vgf2p8affineinvqb 指令时每条指令的输入与输出独立，故可使得指令平均消耗周期可以逼近吞吐率，获得更好的效率，在指令效率的分析上要优于 Guo 的方法。

表 3　GFNI 指令集优化方法平均指令开销对比

Table 3　Average Instruction Overhead Comparison Based on GFNI

优化方法	异或	仿射	求逆－仿射	循环移位
文献[6]	0.5	0.062 5	0.062 5	0.25
本文	0.5	0	0.125	0

3.3　使用 AES-NI 指令集加速 SM4 轮函数

在不支持 GFNI 指令集的处理器上可使用 AES-NI 指令集替代，配合更为常见的 SSE 指令集。8 比特仿射变换可以通过将 8 比特数拆分成高 4 bit 和低 4 bit，利用重排（vpshufb）指令分别查表完成变换，最后通过异或将结果合并。操作步骤分为下方的 3 步，通过预先计算好的查找表 T_h 和 T_l 计算输入 x 的仿射变换 y，总共需要 2 次与运算、1 次移位运算、2 次重排和 1 次异或。

1）$t_1 = x$ & 0x0F
2）$t_2 = (x >> 4)$ & 0x0F
3）y = vpshufb（T_h, t_1）⊕vpshufb（T_l, t_2）

因为 SSE 并没有循环移位的指令，需要使用左右移位和异或替代，故本文优化方法比 Saarinen[21] 的方法在线性变换上拥有更低的开销。Saarinen 中对线性变换进行了优化，将部分循环移位改用重排指令简化计算，采用 4 分组并行共消耗 1 次左移、1 次右移、5 次异或和 3 次重排，平均每分组消耗 0.25 次左移、0.25 次右移、1.25 次异或和 0.75 次重排；本文字节切片优化方法 16 分组并行共需要 16 次异或，平均每分组只消耗 1 次异或，开销更低。在平均每分组的矩阵乘法次数上，本文算法与同类算法没有差别，都是 2 次。但通过重排指令实现 8 比特乘法时需要提取出高/低 4 bit，本文因为是针对同样的输入进行 2 次乘法，故只需要提取 1 次，Saarinen 的方法 2 次乘法的输入不同，需要提取 2 次，开销更大。同时，本文所提出的方法输入的数据分组数更大，故 AES-NI 指令集的指令吞吐率要更高。

4　测试结果

测试所使用的处理器为 Intel i5-11400 和 Intel i5-8265U，具体细节如表 4 所示。因为本文 SM4 字节切片算法采用 C 语言编写，寄存器分配和指令排布将由编译器自行优化，不同编译器的优化结

果不同，故最终的测试结果会有浮动。此外，本文在测试优化算法时均开启 Intel 处理器的睿频加速（Turbo Boost）技术。

表 4 测试环境

Table 4 Environments for Efficiency Test

处理器型号	架构	基本频率/GHz	最大睿频/GHz
Intel i5-11400	Rocket Lake	2.60	4.40
Intel i5-8265U	Whiskey Lake	1.60	3.90

Guo[6]并未开源代码，为控制变量，本文复现了其基于 GFNI 指令集和 AVX512 指令集的优化方法。同时为了更好地对比，本文除字节切片外还额外实现了 2 种优化方法：一种是采用本文的新轮函数结构但并未采用字节切片的优化方法，记为方法 1，用于展示新结构的优越性；另一种是与 Guo 采用同样的方法，但将输入的数据分组扩大 4 倍以提升指令吞吐率，记为方法 2，用于展示字节切片对线性部分的优化。

表 5 基于 GFNI 指令集的 SM4 算法速率测试

Table 5 Rate Test of SM4 Based on GFNI

优化方法	并行分组数	指令集	速率/Mbps	速率/cpb	平台
文献[6]	16	GFNI+AVX512	15 720	1.26	i5-11400
方法 1	16	GFNI+AVX512	18 126	1.09	i5-11400
方法 2	64	GFNI+AVX512	32 133	0.615	i5-11400
字节切片（本文）	64	GFNI+AVX512	35 947	0.548	i5-11400

根据表 5 的测试结果，基于本文新结构的方法（方法 1）速率要优于文献［6］的方法。方法 2 是为增加指令吞吐率而设计的方法，在扩大输入分组数后效率增长了约 1 倍，故 GFNI 指令集的指令开销是制约 Guo 的方法的重要因素。本文字节切片的速率在方法 2 的基础上减少了其中循环移位的开销，获得约 10%的性能提升。

表 6 是使用 AES-NI 指令集配合 SSE 指令集的测试结果，为了展示本文新结构的优越性，本文额外实现了方法 3，方法 3 采用了本文提出的新结构但并未采用字节切片。根据表 6，新结构的计算开销比原始结构更低，方法 3 的加密速率达到了 2 530 Mbps，优于 Saarinen 的 2 147 Mbps。与 Saarinen 的方法相比，本文字节切片方法的实现速率达到了 5 410 Mbps，是其 2.5 倍。

表 6 基于 AES-NI 指令集的 SM4 算法速率测试

Table 6 Rate Test of SM4 Based on AES-NI

优化方法	并行分组数	指令集	速率/Mbps	速率/cpb	平台
文献[21]	4	AES-NI+SSE	2 147	6.78	i5-8265U
方法 3	4	AES-NI+SSE	2 530	5.30	i5-8265U
字节切片（本文）	16	AES-NI+SSE	5 410	2.50	i5-8265U

表 7 将本文提出的 SM4 字节切片方法与目前文献中的优化方法进行对比，根据表 7 中的测试结果，本文基于 AES-NI 指令集配合 SSE 指令集的字节切片优化方法速率（5 410 Mbps）要优于查表法[31]（2 437 Mbps）、比特切片配合选择函数[12]（2 580 Mbps）和比特切片配合复合域[13]（5 076 Mbps）中采用 AVX2 指令集的方法，在不支持 GFNI 指令集的环境下也能达到比较好的效果。在同一测试环境下，本文基于 GFNI 指令集的字节切片速率达到 35 947 Mbps，是公开文献最优结果[16]的 1.34 倍，是同类方法[6]的 2.28 倍。

表 7 SM4 软件优化方法横向对比

Table 7 Horizontal Comparison of SM4 Software Optimization Methods

优化方法	指令集	速率/Mbps	平台
AES-NI[21]	AES-NI+SSE	2 147	i5-8265U
查表法[31]	AVX2	2 437	i7-6700
比特切片+选择函数[12]	AVX2	2 580	i7-7700HQ
比特切片+复合域[13]	AVX2	5 076	Ryzen 7 4800H
字节切片(本文)	AES-NI+SSE	5 410	i5-8265U
比特切片+塔域[15]	AVX512	6 673	i7-11800H
比特切片[33]	AVX2	15 626	i7-8700
GFNI[6]	GFNI+AVX512	15 720	i5-11400
比特切片[16]	AVX512	26 810	i5-11400
字节切片(本文)	GFNI+AVX512	35 947	i5-11400

5 总结

本文将非线性变换 τ 中的 2 个矩阵乘法与线性变换 L 融合，给出了新的 SM4 函数结构，并采用字节切片技术进一步加速实现。本文的优化方法改进了 SM4 算法目前基于 GFNI/AES-NI 指令集的优

化方法中线性部分的处理方式，解决了 SM4 算法线性变换不适合字节切片的问题，并改善了指令吞吐率方面存在的效率问题。在采用 GFNI 指令集配合 AVX512 指令集的实现方法中，与 Guo[6] 提出的采用同样指令集的优化方法进行对比，在理论分析上具备优越性，且测试结果表明有较大的效率提升。SM4 算法采用字节切片的效率达到 35 947 Mbps，是 Xu 等人[16] 提出方法的 1.34 倍，是 Guo[6] 提出方法的 2.28 倍。由于 AVX512 和 GFNI 均是 Intel 较新的指令集，只有近几年新推出的 CPU 架构才能支持，但在不支持 GFNI 指令集的 CPU 上可采用 AES-NI 指令集进行实现，采用 AES-NI 指令集配合 SSE 指令集的实现速率可达到 5 410 Mbps。且本文的优化方法具备一定程度的通用性，在其它平台还可以采用 Permute 指令实现。后续可以通过汇编语言进一步优化实现，采用汇编能更细致地操作寄存器和指令，根据具体的处理器环境布置指令可以达到更好的优化效果。

参考文献：

[1] 国家密码管理局. 信息安全技术 SM4 分组密码算法：GB/T 32907 - 2016 [S]. 北京：中国标准出版社，2016.

[2] DAEMEN J，RIJMEN V. The design of Rijndael [M]. Berlin：Springer，2020：53 - 63.

[3] BONNEAU J，MIRONOV I. Cache-collision timing attacks against AES [N] // Lecture Notes in Computer Science (LNSC). Cryptographic Hardware and Embedded Systems-CHES 2006：8th International Workshop，Yokohama，Japan，2006：201 - 215.

[4] FUJII H，RODRIGUES F C，LóPEZ J. Fast AES implementation using ARMv8 ASIMD without cryptography extension [C] // Lecture Notes in Computer Science (LNSC). Information Security and Cryptology-ICISC 2019：22nd International Conference，Seoul，South Korea，Cham：Springer International Publishing，2020：84 - 101.

[5] KWON H，KIM H，EUM S，et al. Optimized implementation of SM4 on AVR micro-controllers，RISC-V Processors，and ARM Processors [J]. IEEE Access，2022，10：80225 - 80233.

[6] GUO W J. Efficient constant-time implementation of SM4 with Intel GFNI instruction set extension and arm NEON coprocessor [J/OL]. Cryptology ePrint Archive. 2022. https:// eprint. iacr. org/2022/1154.

[7] KÄSPER E，SCHWABE P. Faster and timing-attack resistant AES-GCM [C] //Lecture notes in computer science (LNSC). cryptographic hardware and embedded systems-CHES 2009：11th International Workshop，Lausanne，Switzerland. Berlin：Springer，2009：1 - 17.

[8] BIHAM E. A fast new DES implementation in software [C] // Lecture Notes in Computer Science (LNSC). Fast Software Encryption-FSE 97：4th International Workshop，Haifa，Israel，Berlin：Springer，1997：260 - 272.

[9] ADOMNICAI A，NAJM Z，PEYRIN T. Fixslicing：a new GIFT representation：fast constant-time implementations of GIFT and GIFT-COFB on ARM Cortex-M [J]. IACR Transactions on Cryptographic Hardware and Embedded Systems，2020 (3)：402 - 427.

[10] ADOMNICAI A，PEYRIN T. Fixslicing AES-like ciphers：new bitsliced AES speed records on ARM-Cortex M and RISC-V [J]. IACR Transactions on Cryptographic Hardware and Embedded Systems，2021 (1)：402 - 425.

[11] KWAN M. Reducing the gate count of bitslice DES [J/OL]. IACR Cryptology ePrint Archive，2000. https://eprint. iacr. org/2000/251. pdf.

[12] 张笑从，郭　华，张习勇，等. SM4 算法快速软件实现 [J]. 密码学报，2020，7 (6)：799 - 811.

[13] CANRIGHT D. A very compact S-box for AES [C] //Lecture Notes in Computer Science (LNSC). Cryptographic Hardware and Embedded Systems-CHES 2005：7th International Workshop，Edinburgh，UK. Berlin：Springer，2005：441 - 455.

[14] 陈　晨，郭　华，王　闯，等. 一种基于复合域的国密 SM4 算法快速软件实现方法 [J]. 密码学报，2023，10 (2)：289 - 305.

[15] 王　磊，龚　征，刘　哲，等. 基于塔域的 SM4 算法快速软件实现 [J]. 密码学报，2022，9 (6)：1081 - 1098.

[16] XU R Q，XIANG Z J，LIN D，et al. High-throughput block cipher implementations with SIMD [J]. Journal of Information Security and Applications，2022，70：103333.

[17] HAMBURG M. Accelerating AES with vector permute instructions [C] //Lecture Notes in Computer Science (LNSC). Cryptographic Hardware and Embedded Systems-CHES 2009：11th International Workshop，Lausanne. Berlin：Springer，2009，5747：18 - 32.

[18] ROTT J K. Intel advanced encryption standard instructions (AES-NI) [EB/OL]. (2012 - 02 - 02) [2024 - 01 - 26]. https://www. intel. com/content/www/us/en/developer/articles/technical/advanced-encryption-

standard-instructions-aes-ni. html.

[19] DRUCKER N, GUERON S, KRASNOV V. Making AES great again: the forthcoming vectorized AES instruction [C] // Advances in Intelligent Systems and Computing. 16th International Conference on Information Technology-New Generations-ITNG 2019, Las Vegas, Nevada, Cham: Springer, 2019, 800: 37 - 41.

[20] AOKI K, MATUSIEWICZ K, ROLAND G, et al. Byte slicing Grøstl: improved intel AES-NI and vector-permute implementations of the SHA-3 finalist Grøstl [C] // Communications in Computer and Information Science (CCIS). International Joint Conference on E-Business and Telecommunications-ICETE 2011: 8th International Workshop, Berlin: Springer, 2012, 314: 281 - 295.

[21] SAARINEN M J O. SM4ni [EB/OL]. (2019 - 02 - 28) [2023 - 08 - 18]. https://github. com/ mjosaarinen/sm4ni.

[22] DRUCKER N, GUERON S. Fast constant time implementations of ZUC-256 on x86 CPUs [C] // Proceedings of the 16th IEEE Annual Consumer Communications and Networking Conference-CCNC, Las Vegas, New York: IEEE, 2019: 1 - 7.

[23] KIVILINNA J. Block ciphers: fast implementations on x86-64 architecture [D]. University of Oulu, 2013.

[24] TOWNER D, KINSELLA R. Galois field new instructions (GFNI) technology guide [EB/OL]. (2021 - 12 - 09) [2023 - 08 - 18]. https://networkbuilders. intel. com/solutionslibrary/galois-field-new-instructions-gfni-technology-guide.

[25] DRUCKER N, GUERON S, KRASNOV V. The comeback of Reed Solomon codes [C] //2018 IEEE 25th Symposium on Computer Arithmetic-ARITH 2018, Amherst, MA, New York: IEEE, 2018: 125 - 129.

[26] DRUCKER N, GUERON S. Speed up over the rainbow [C] // Advances in Intelligent Systems and Computing. 18th International Conference on Information Technology-New Generations-ITNG 2021, Virtual Mode, Cham: Springer, 2021, 1346: 131 - 136.

[27] 解文博. 基于 GPU 和切片的分组密码算法高速实现方法研究 [D]. 桂林: 桂林电子科技大学, 2021.

[28] KHAN A H, AL-MOUHAMED M A, ALMOUSA A, et al. AES-128 ECB encryption on GPUs and effects of input plaintext patterns on performance [C] //15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing-SNPD, Las Vegas, 2014. New York: IEEE, 2014: 1 - 6.

[29] ABDELRAHMAN A AFOUAD M M, DAHSHAN H, et al. High performance CUDA AES implementation: a quantitative performance analysis approach [C] //2017 Computing Conference, New York: IEEE, 2017: 1077 - 1085.

[30] HAJIHASSANI O, MONFARED S K, KHASTEH S H, et al. Fast AES implementation: a high-throughput bitsliced approach [J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30 (10): 2211 - 2222.

[31] 郎 欢, 张 蕾, 吴文玲. SM4 的快速软件实现技术 [J]. 中国科学院大学学报, 2018, 35 (2): 180 - 187.

[32] ZHANG J, MA M, WANG P. Fast implementation for SM4 cipher algorithm based on bit-slice technology [C] // Lecture Notes in Computer Science (LNCS). 3rd Smart Computing and Communication-SMARTCOMP 2018, Tokyo. Berlin: Springer, 2018: 104 - 113.

[33] MIAO X, GUO C, WANG M, et al. How fast can SM4 be in software? [C] //Lecture Notes in Computer Science (LNCS). 18th International Conference on Information Security and Cryptology-Inscrypt 2022, Beijing, China, Cham: Springer, 2023: 3 - 22.

[34] LIU F, JI W, HU L, et al. Analysis of the SMS4 block cipher [C] //Lecture Notes in Computer Science (LNCS). 12th Australasian Conference on Information Security and Privacy-ACISP 2007. Berlin: Springer, 2007: 158 - 170.