SM4 算法原理：

1.常量定义：

SM4_FK，SM4_CK

S 盒（8 位输入，8 位输出的非线性置换）

2.加密流程：

明文分组（128 位）

密钥扩展（生成 32 个轮密钥）

32 轮迭代变换

最终变换（反序输出）

T-table 优化：

核心思路：预计算 S 盒与线性变换的组合结果，减少实时计算量

1.预计算 T_enc[i] = L(S(i) << 24)（加密用）

2.预计算 T_key[i] = L’(S(i) << 24)（密钥扩展用）

3.代码实现：

```
def _init_t_tables(self):
    self.T_enc = [0] * 256  # 加密用 T 表
    self.T_key = [0] * 256  # 密钥扩展用 T 表
    for i in range(256):
        s = SM4_SBOX[i]
        self.T_enc[i] = self._l_transform(s << 24)
        self.T_key[i] = self._l_prime_transform(s << 24)
```

```python
def _round_function(self, x0: int, x1: int, x2: int, x3: int, rk: int) -> int:
    tmp = x1 ^ x2 ^ x3 ^ rk
    # 使用预计算 T 表加速
    t = self.T_enc[tmp >> 24]
    t ^= (self.T_enc[(tmp >> 16) & 0xFF] >> 8)
    t ^= (self.T_enc[(tmp >> 8) & 0xFF] >> 16)
    t ^= (self.T_enc[tmp & 0xFF] >> 24)
    return x0 ^ t
```

AES-NI 优化：

AES-NI 是 Intel 处理器的加密加速指令集，虽专为 AES 设计，但可通过封装调用提升 SM4 性能。代码中通过检测硬件支持，优先使用优化路径：

```python
def _check_hardware_support(self):
    self.aesni_supported = False
    try:
        from cryptography.hazmat.primitives.ciphers import Cipher
        from cryptography.hazmat.backends import default_backend
```

```
        backend = default_backend()
        if hasattr(backend, 'has_aesni_support') and backend.has_aesni_support():
            self.aesni_supported = True
            print("AES-NI 硬件加速支持已启用")
    except ImportError:
        pass  # 降级为 T-table 实现
```

GFNI 优化：

GFNI 是新一代加密指令集，支持有限域运算和置换操作，可直接加速 S 盒和线性变换：

```
def _check_gfni_support(self):
    self.gfni_supported = False
    if os.name == 'posix':
        with open('/proc/cpuinfo', 'r') as f:
            cpuinfo = f.read()
            if 'gfni' in cpuinfo and 'vpbroadcastd' in cpuinfo:
                self.gfni_supported = True
                 # 检测到 GFNI 支持
```

GCM 优化：

1.核心组件：

计数器模式（CTR）：用于加密和解密

生成计数器块：CB = IV || 0x00000001

加密：C_i = P_i ^ E(K, CB_i)，CB_{i+1} = CB_i + 1

2.GHASH函数：用于消息认证

基于GF(2^128)乘法的哈希函数

输入：关联数据、密文、长度信息

输出：认证标签

实现代码：

```
def encrypt(self, key: bytes, iv: bytes, plaintext: bytes, associated_data: bytes = b'') -> Tuple[bytes, bytes]:
    # 生成哈希密钥H = SM4(K, 0^128)
    h = self.sm4.encrypt_block(b'\x00' * 16, key)
    # 计数器模式加密
    cb = iv + b'\x00\x00\x00\x01'
    # 初始计数器块
    ciphertext = []
    for block in [plaintext[i:i+16] for i in range(0, len(plaintext), 16)]:
        ctr = self.sm4.encrypt_block(current_cb, key)
         # 加密计数器
        ciphertext_block = bytes([b ^ c for b, c in zip(block,
```

```
ctr[:len(block)])])
        ciphertext.append(ciphertext_block)
        current_cb = _inc_iv(current_cb)
         # 计数器递增
    # 计算认证标签
    ghash_input = associated_data + b''.join(ciphertext)
    ghash_input += struct.pack(">QQ", len(associated_data)*8,
len(ciphertext)*8)
    s = self._ghash(h, ghash_input)
    tag = bytes([b ^ c for b, c in zip(s,
self.sm4.encrypt_block(cb, key))])
    return b''.join(ciphertext), tag
```

其中GF(2^128)乘法实现：

```
def _gf128_mul(a: int, b: int) -> int:
    p = 0x87
    # 不可约多项式：x^128 + x^7 + x^2 + x + 1
    result = 0
    for i in range(128):
        if b & 1:
            result ^= a
        a <<= 1
        if a & (1 << 128):
```

```
        a ^= p << 127
        # 模不可约多项式
    b >>= 1
return result & ((1 << 128) - 1)
```

5.性能测试：

```
# 标准测试向量
key = bytes.fromhex("0123456789abcdeffedcba9876543210")
plaintext =
bytes.fromhex("0123456789abcdeffedcba9876543210")
expected_ciphertext =
bytes.fromhex("681edf34d206965e86b3e94f536e4246")
# 验证加密正确性
ciphertext = sm4_base.encrypt_block(plaintext, key)
assert ciphertext == expected_ciphertext, 基础实现加密错误
```