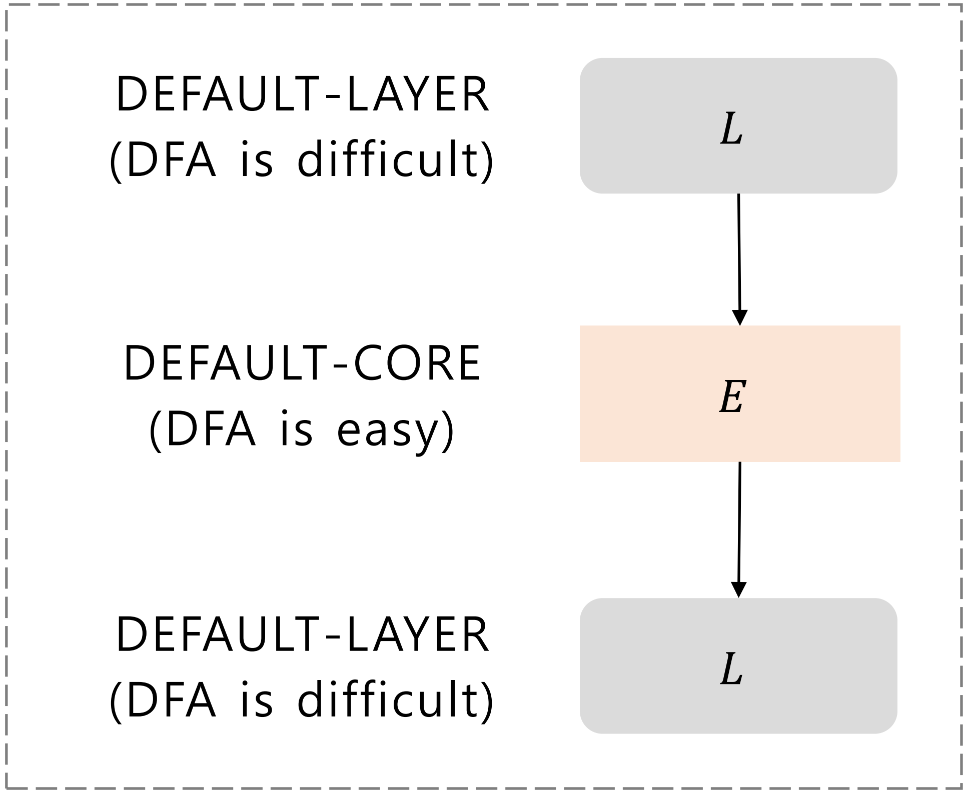
DEFAULT GPU 구현

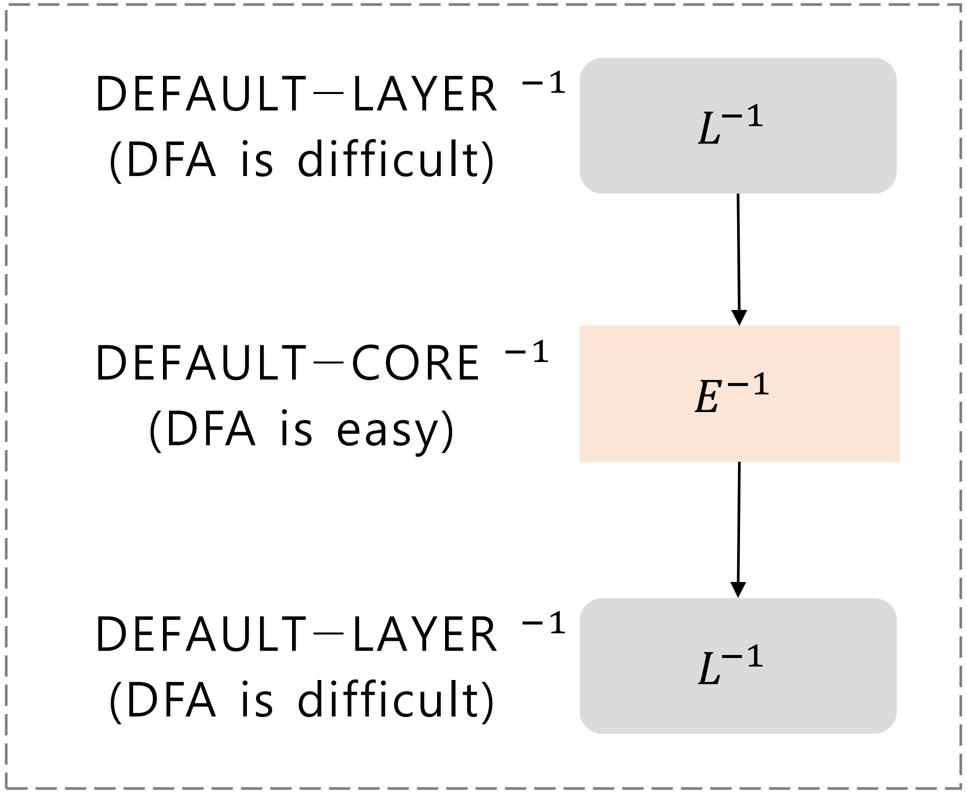
송민호

유튜브: https://youtu.be/X6ziNt7HIVY

DEFAULT 전체 구조

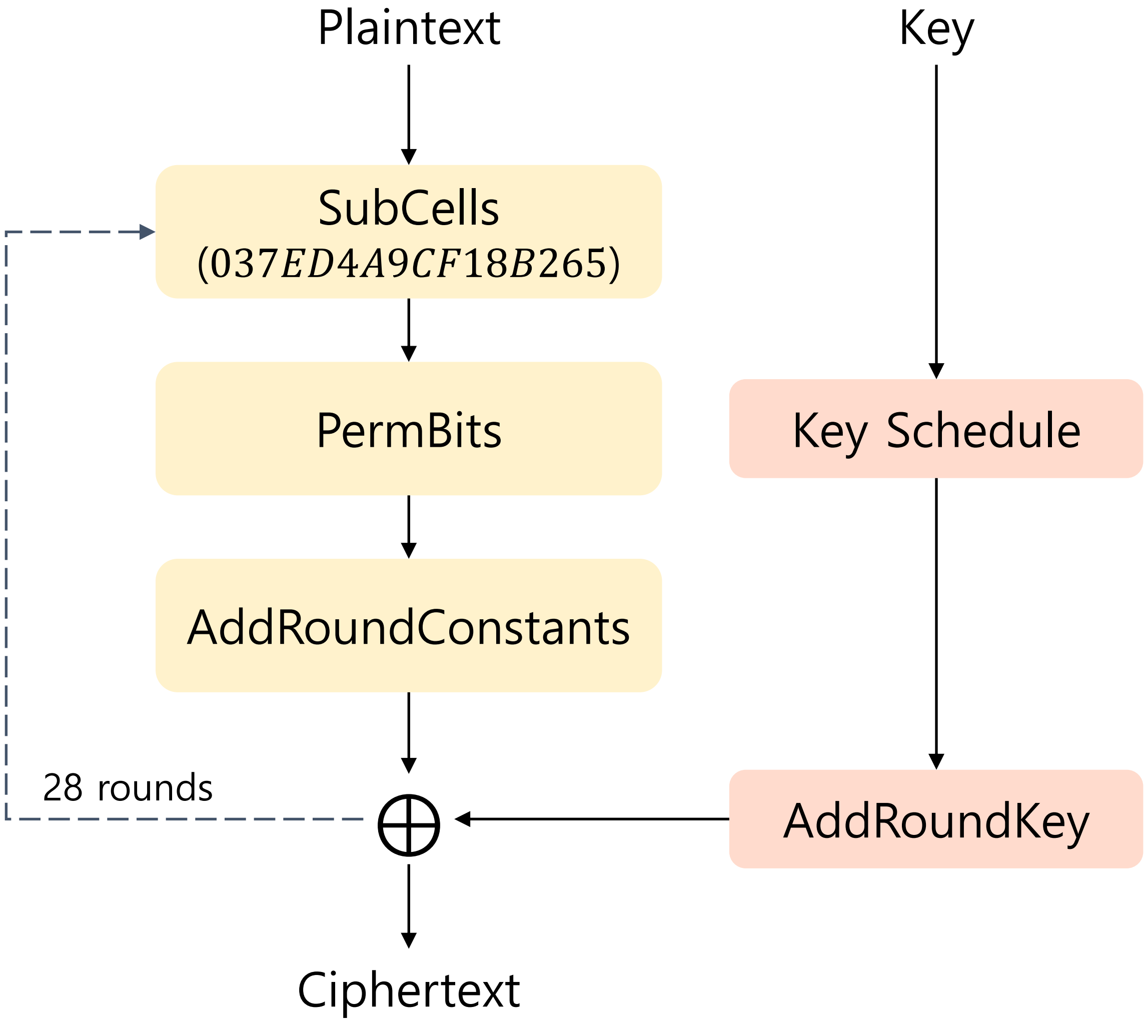


Encryption

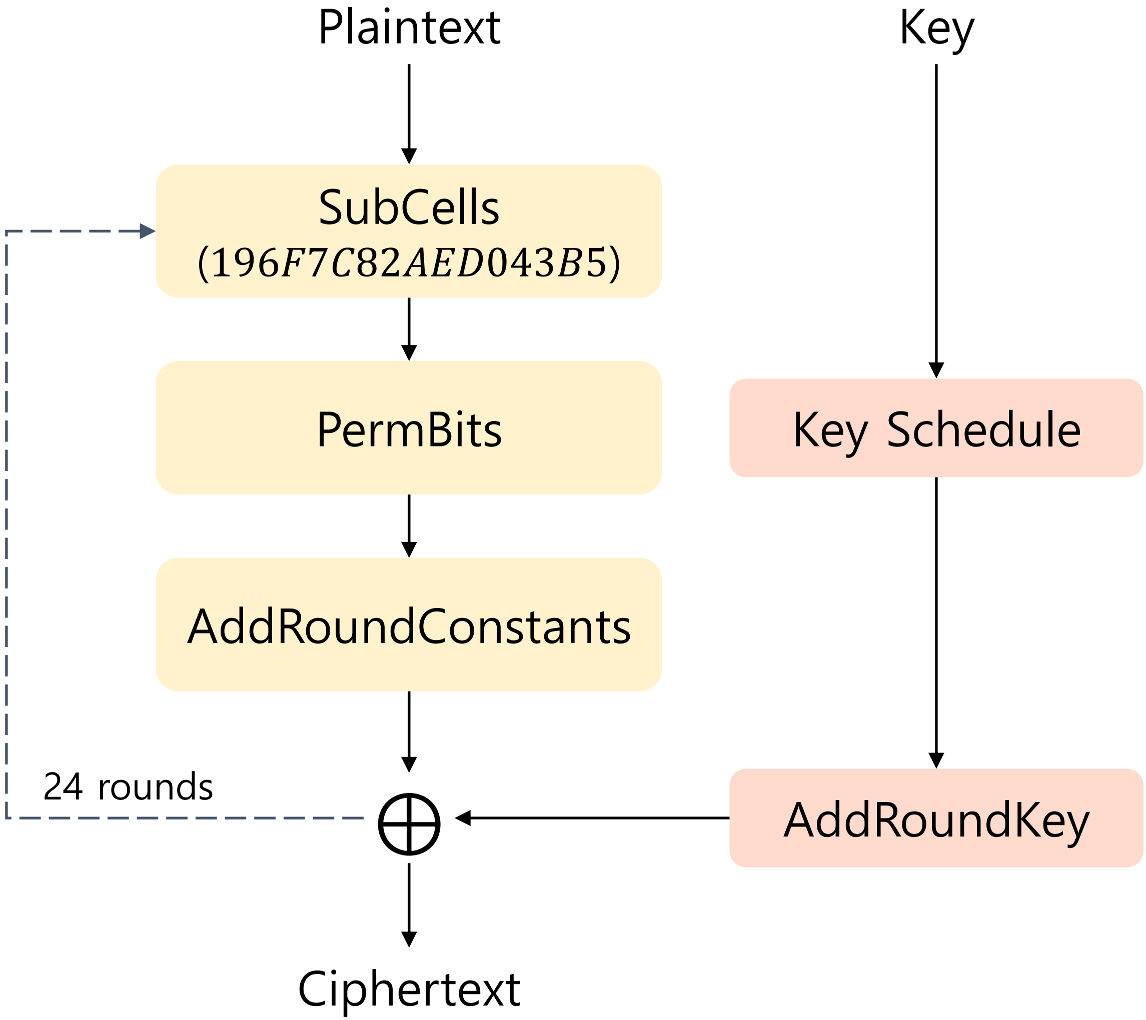


Decryption

DEFAULT 전체 구조 - LAYER, CORE



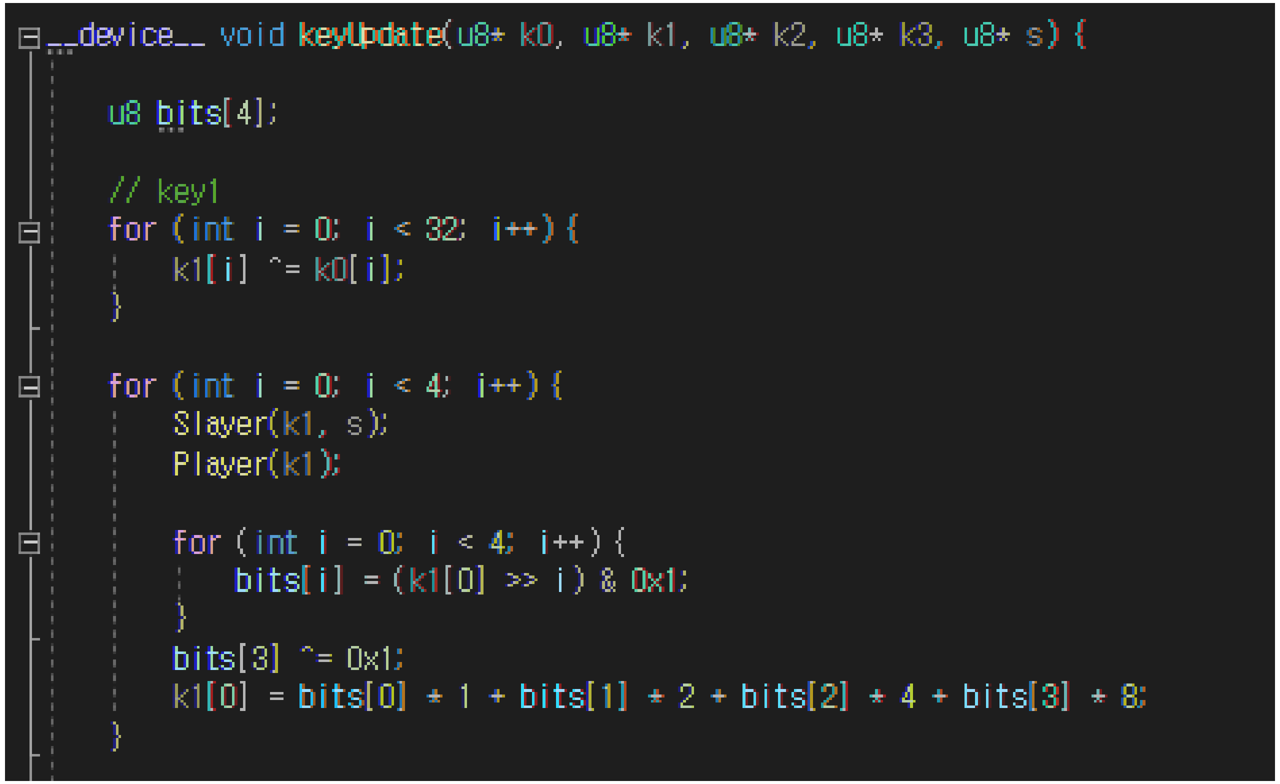
DEFAULT-LAYER



DEFAULT-CORE

Key

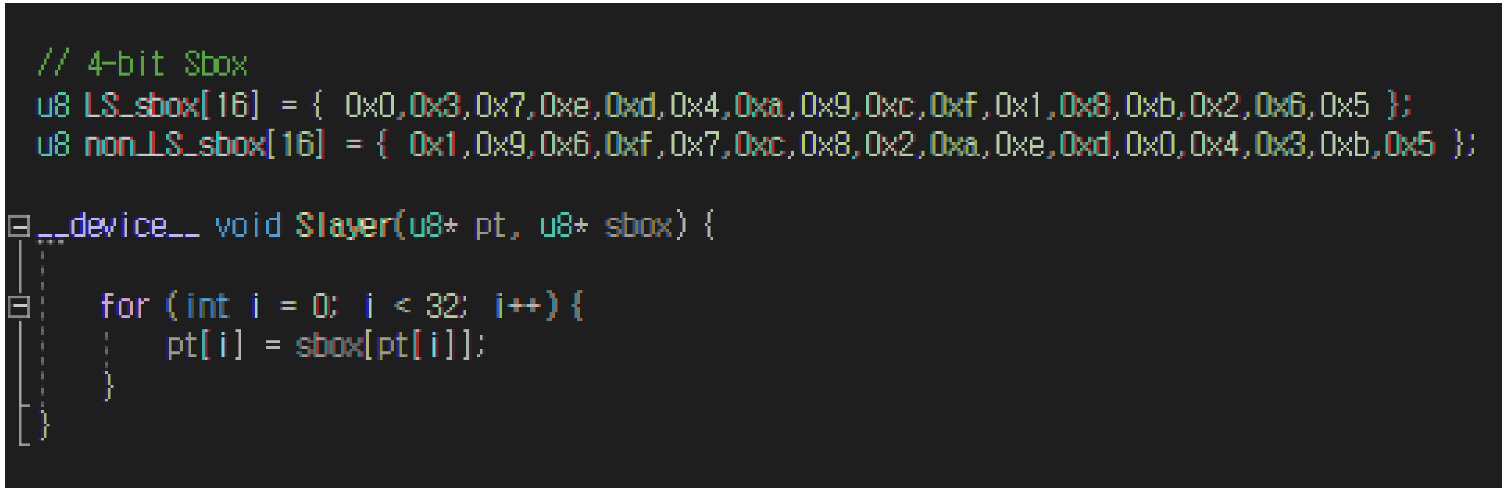
- KeySchedule
  - 128-bit master key K를 사용하여 4개의 128-bit subkey 생성(K_0, K_1, K_2, K_3)



```
__device__ void keyUpdate(u8* k0, u8* k1, u8* k2, u8* k3, u8* s) {

    u8 bits[4];

    // key1
    for (int i = 0; i < 32; i++) {
        k1[i] ^= k0[i];
    }

    for (int i = 0; i < 4; i++) {
        Slayer(k1, s);
        Player(k1);

        for (int i = 0; i < 4; i++) {
            bits[i] = (k1[0] >> i) & 0x1;
        }
        bits[3] ^= 0x1;
        k1[0] = bits[0] * 1 + bits[1] * 2 + bits[2] * 4 + bits[3] * 8;
    }
```

SubCells

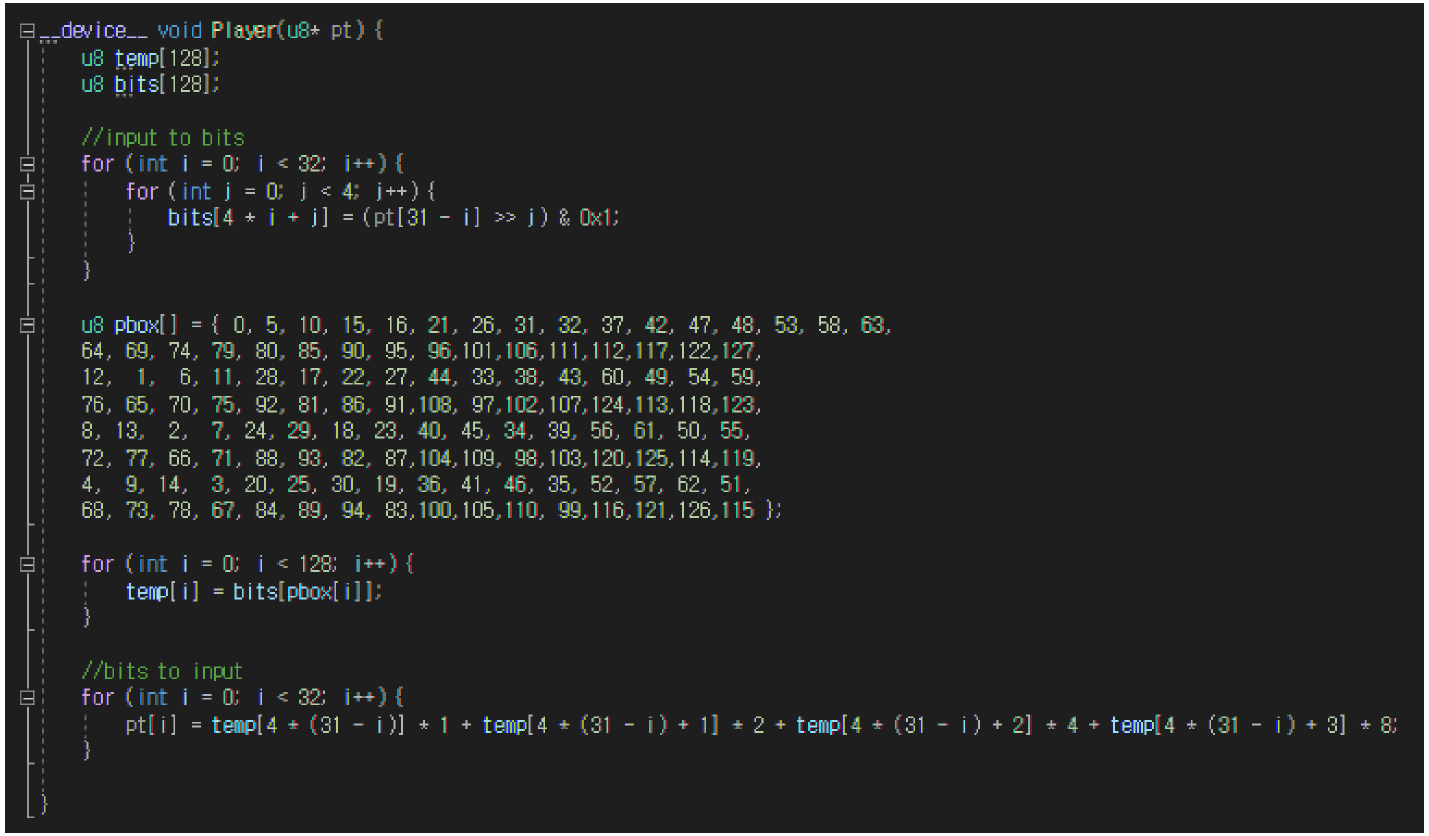
- DEFAULT-LAYER
  - 4-bit LS Sbox 사용 (S = 037ED4A9CF18B265)
- DEFAULT-CORE
  - 4-bit non_LS Sbox 사용 (S = 196F7C82AED043B5)



```
// 4-bit Sbox
u8 LS_sbox[16] = { 0x0,0x3,0x7,0xe,0xd,0x4,0xa,0x9,0xc,0xf,0x1,0x8,0xb,0x2,0x6,0x5 };
u8 non_LS_sbox[16] = { 0x1,0x9,0x6,0xf,0x7,0xc,0x8,0x2,0xa,0xe,0xd,0x0,0x4,0x3,0xb,0x5 };

__device__ void Slayer(u8* pt, u8* sbox) {

    for (int i = 0; i < 32; i++) {
        pt[i] = sbox[pt[i]];
    }
}
```

Permutation

- 연산을 위해 nibble_to_bits 과정을 겪음



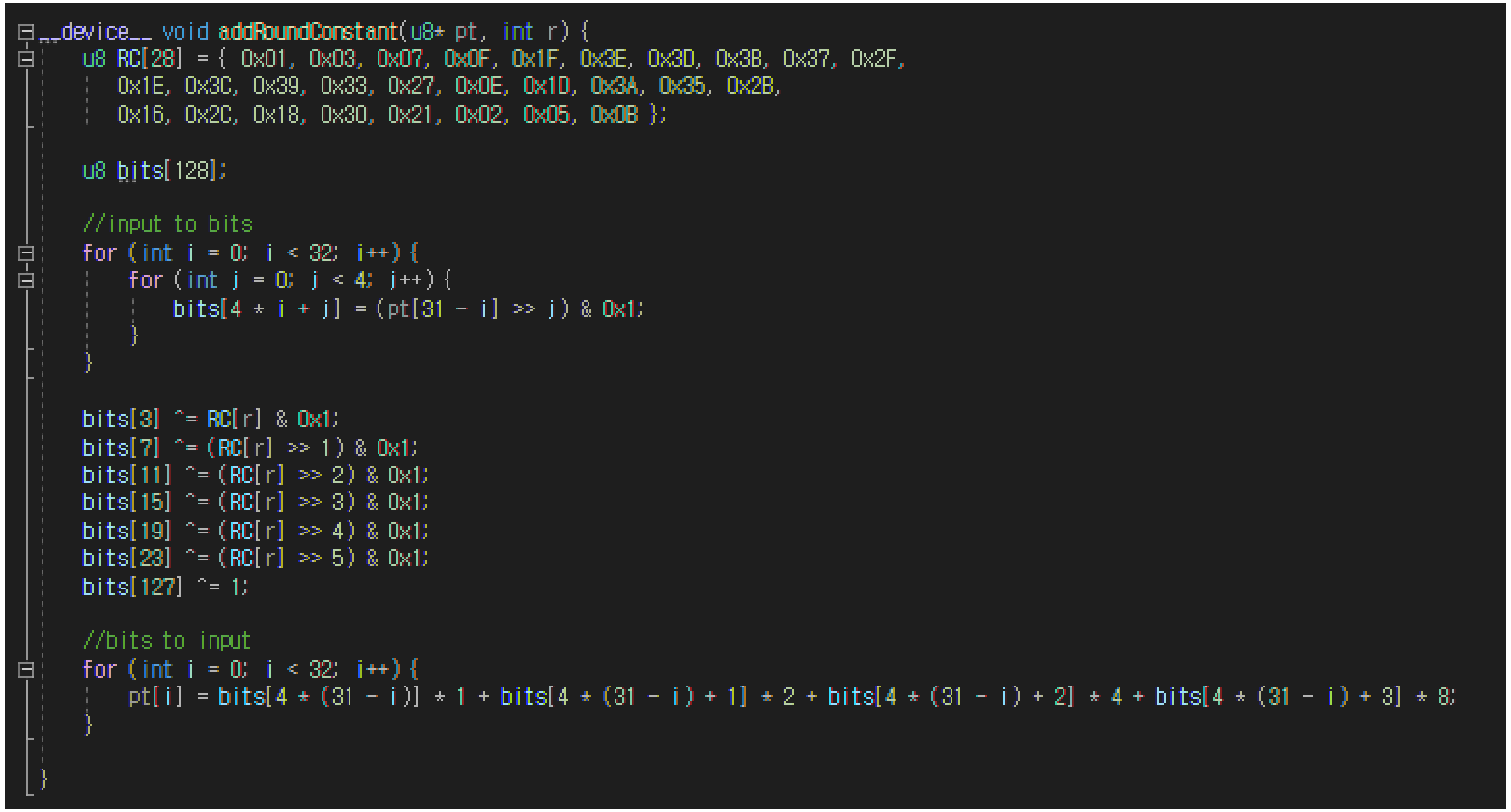
```
__device__ void Player(u8* pt) {
    u8 temp[128];
    u8 bits[128];

    //input to bits
    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 4; j++) {
            bits[4 * i + j] = (pt[31 - i] >> j) & 0x1;
        }
    }

    u8 pbox[] = { 0, 5, 10, 15, 16, 21, 26, 31, 32, 37, 42, 47, 48, 53, 58, 63,
    64, 69, 74, 79, 80, 85, 90, 95, 96,101,106,111,112,117,122,127,
    12,  1,  6, 11, 28, 17, 22, 27, 44, 33, 38, 43, 60, 49, 54, 59,
    76, 65, 70, 75, 92, 81, 86, 91,108, 97,102,107,124,113,118,123,
    8, 13,  2,  7, 24, 29, 18, 23, 40, 45, 34, 39, 56, 61, 50, 55,
    72, 77, 66, 71, 88, 93, 82, 87,104,109, 98,103,120,125,114,119,
    4,  9, 14,  3, 20, 25, 30, 19, 36, 41, 46, 35, 52, 57, 62, 51,
    68, 73, 78, 67, 84, 89, 94, 83,100,105,110, 99,116,121,126,115 };

    for (int i = 0; i < 128; i++) {
        temp[i] = bits[pbox[i]];
    }

    //bits to input
    for (int i = 0; i < 32; i++) {
        pt[i] = temp[4 * (31 - i)] * 1 + temp[4 * (31 - i) + 1] * 2 + temp[4 * (31 - i) + 2] * 4 + temp[4 * (31 - i) + 3] * 8;
    }

}
```

AddRoundConstant

- 단일 비트 “1”과 Round Constant $C = c_5c_4c_3c_2c_1c_0$ 와의 XOR 연산

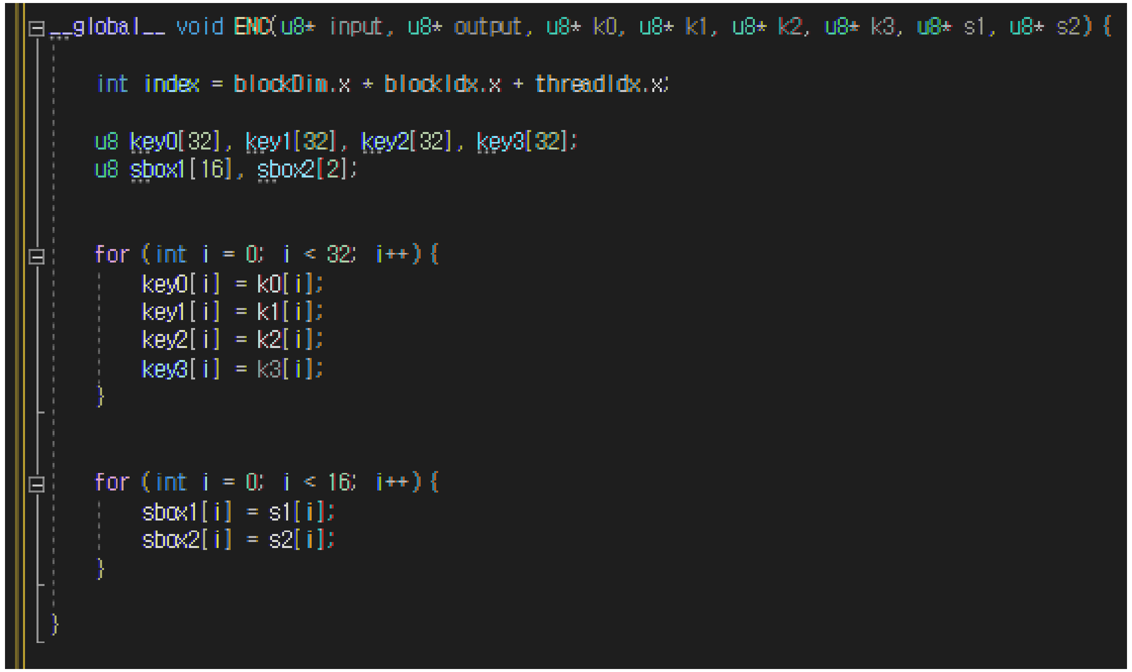
$$w_{127} = w_{127} \oplus 1, \qquad w_{23} = w_{23} \oplus c_5, \qquad w_{19} = w_{19} \oplus c_4$$

$$w_{15} = w_{15} \oplus c_3, \qquad w_{11} = w_{11} \oplus c_2, \qquad w_7 = w_7 \oplus c_1, \qquad w_3 = w_3 \oplus c_0$$

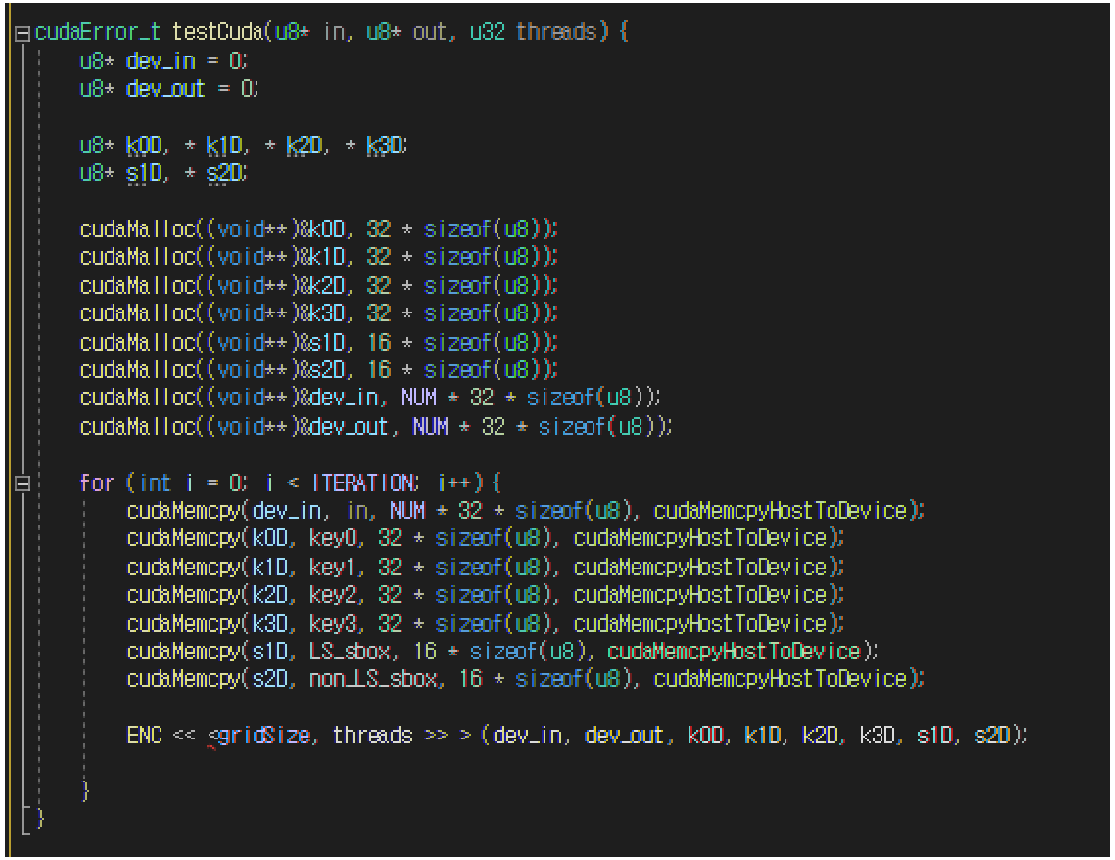


```
__device__ void addRoundConstant(u8* pt, int r) {
    u8 RC[28] = { 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F,
        0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B,
        0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B };

    u8 bits[128];

    //input to bits
    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 4; j++) {
            bits[4 * i + j] = (pt[31 - i] >> j) & 0x1;
        }
    }

    bits[3] ^= RC[r] & 0x1;
    bits[7] ^= (RC[r] >> 1) & 0x1;
    bits[11] ^= (RC[r] >> 2) & 0x1;
    bits[15] ^= (RC[r] >> 3) & 0x1;
    bits[19] ^= (RC[r] >> 4) & 0x1;
    bits[23] ^= (RC[r] >> 5) & 0x1;
    bits[127] ^= 1;

    //bits to input
    for (int i = 0; i < 32; i++) {
        pt[i] = bits[4 * (31 - i)] * 1 + bits[4 * (31 - i) + 1] * 2 + bits[4 * (31 - i) + 2] * 4 + bits[4 * (31 - i) + 3] * 8;
    }

}
```

Global memory 구현

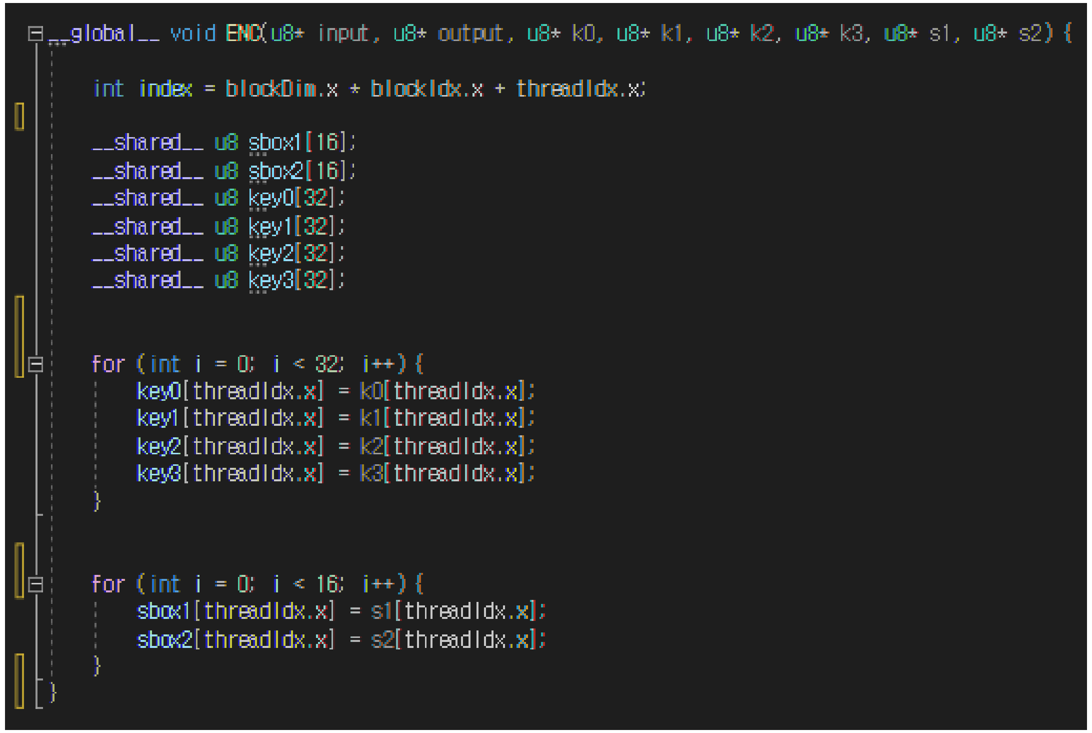


```
__global__ void ENC(u8* input, u8* output, u8* k0, u8* k1, u8* k2, u8* k3, u8* s1, u8* s2) {

    int index = blockDim.x * blockIdx.x + threadIdx.x;

    u8 key0[32], key1[32], key2[32], key3[32];
    u8 sbox1[16], sbox2[2];


    for (int i = 0; i < 32; i++) {
        key0[i] = k0[i];
        key1[i] = k1[i];
        key2[i] = k2[i];
        key3[i] = k3[i];
    }


    for (int i = 0; i < 16; i++) {
        sbox1[i] = s1[i];
        sbox2[i] = s2[i];
    }

}
```

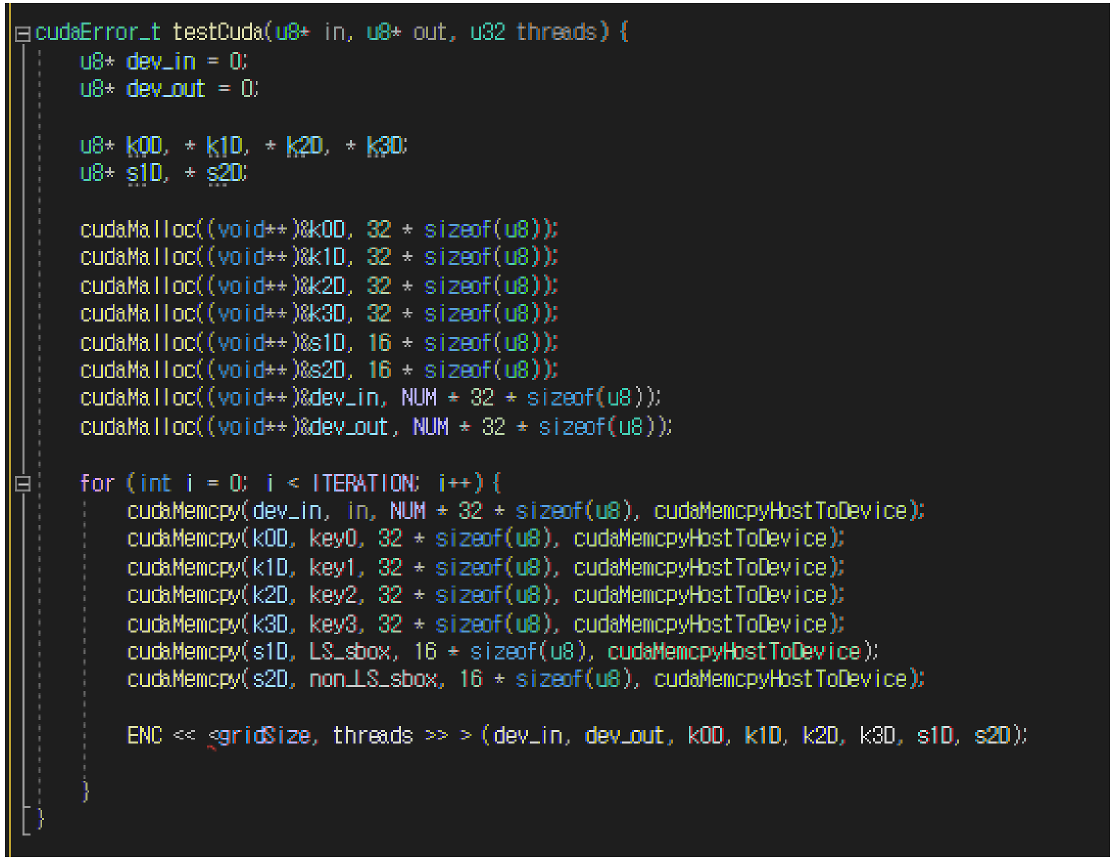


```
cudaError_t testCuda(u8* in, u8* out, u32 threads) {
    u8* dev_in = 0;
    u8* dev_out = 0;

    u8* k0D, * k1D, * k2D, * k3D;
    u8* s1D, * s2D;

    cudaMalloc((void**)&k0D, 32 * sizeof(u8));
    cudaMalloc((void**)&k1D, 32 * sizeof(u8));
    cudaMalloc((void**)&k2D, 32 * sizeof(u8));
    cudaMalloc((void**)&k3D, 32 * sizeof(u8));
    cudaMalloc((void**)&s1D, 16 * sizeof(u8));
    cudaMalloc((void**)&s2D, 16 * sizeof(u8));
    cudaMalloc((void**)&dev_in, NUM * 32 * sizeof(u8));
    cudaMalloc((void**)&dev_out, NUM * 32 * sizeof(u8));

    for (int i = 0; i < ITERATION; i++) {
        cudaMemcpy(dev_in, in, NUM * 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k0D, key0, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k1D, key1, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k2D, key2, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k3D, key3, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(s1D, LS_sbox, 16 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(s2D, non_LS_sbox, 16 * sizeof(u8), cudaMemcpyHostToDevice);

        ENC << <gridSize, threads >> > (dev_in, dev_out, k0D, k1D, k2D, k3D, s1D, s2D);

    }
}
```

Shared memory 구현



```
__global__ void ENC(u8* input, u8* output, u8* k0, u8* k1, u8* k2, u8* k3, u8* s1, u8* s2) {

    int index = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ u8 sbox1[16];
    __shared__ u8 sbox2[16];
    __shared__ u8 key0[32];
    __shared__ u8 key1[32];
    __shared__ u8 key2[32];
    __shared__ u8 key3[32];


    for (int i = 0; i < 32; i++) {
        key0[threadIdx.x] = k0[threadIdx.x];
        key1[threadIdx.x] = k1[threadIdx.x];
        key2[threadIdx.x] = k2[threadIdx.x];
        key3[threadIdx.x] = k3[threadIdx.x];
    }


    for (int i = 0; i < 16; i++) {
        sbox1[threadIdx.x] = s1[threadIdx.x];
        sbox2[threadIdx.x] = s2[threadIdx.x];
    }
}
```



```
cudaError_t testCuda(u8* in, u8* out, u32 threads) {
    u8* dev_in = 0;
    u8* dev_out = 0;

    u8* k0D, * k1D, * k2D, * k3D;
    u8* s1D, * s2D;

    cudaMalloc((void**)&k0D, 32 * sizeof(u8));
    cudaMalloc((void**)&k1D, 32 * sizeof(u8));
    cudaMalloc((void**)&k2D, 32 * sizeof(u8));
    cudaMalloc((void**)&k3D, 32 * sizeof(u8));
    cudaMalloc((void**)&s1D, 16 * sizeof(u8));
    cudaMalloc((void**)&s2D, 16 * sizeof(u8));
    cudaMalloc((void**)&dev_in, NUM * 32 * sizeof(u8));
    cudaMalloc((void**)&dev_out, NUM * 32 * sizeof(u8));

    for (int i = 0; i < ITERATION; i++) {
        cudaMemcpy(dev_in, in, NUM * 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k0D, key0, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k1D, key1, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k2D, key2, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(k3D, key3, 32 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(s1D, LS_sbox, 16 * sizeof(u8), cudaMemcpyHostToDevice);
        cudaMemcpy(s2D, non_LS_sbox, 16 * sizeof(u8), cudaMemcpyHostToDevice);

        ENC << <gridSize, threads >> > (dev_in, dev_out, k0D, k1D, k2D, k3D, s1D, s2D);

    }
}
```

구현 결과

Iteration: 10

Type	Thread	Block	Throughput
Global memory	128	32,768	7.6364
Shared memory(Sbox)			7.0352
Shared memory(RoundKey)			6.8944
Shared memory(Both)			6.8660

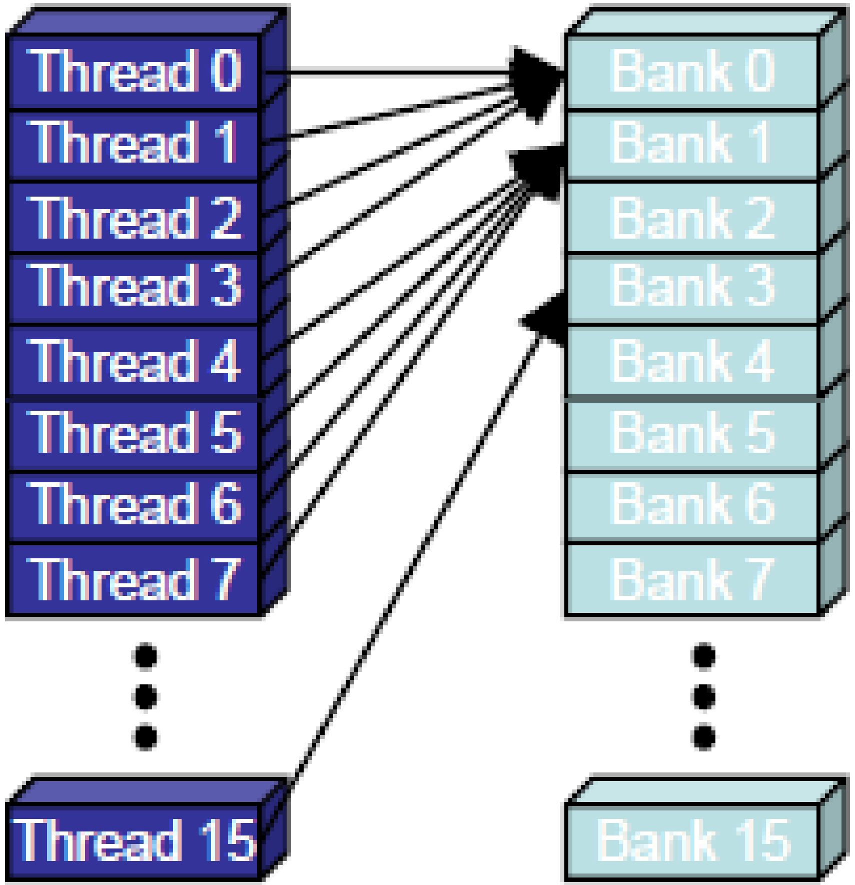
메모리 접근 속도: Shared > Global

생각과 다르게 Shared memory를 사용한 구현의 결과가 더 좋지 않음

Bank conflict 예상

Bank conflict

- Bank conflict
  - 서로 다른 thread가 같은 bank에 접근했을 때 발생하는 문제
  - 각각의 thread는 해당 bank에 접근하기 위해 순차적으로 변하게 됨
  - 병렬 연산 불가



Q & A