2023 암호분석경진대회

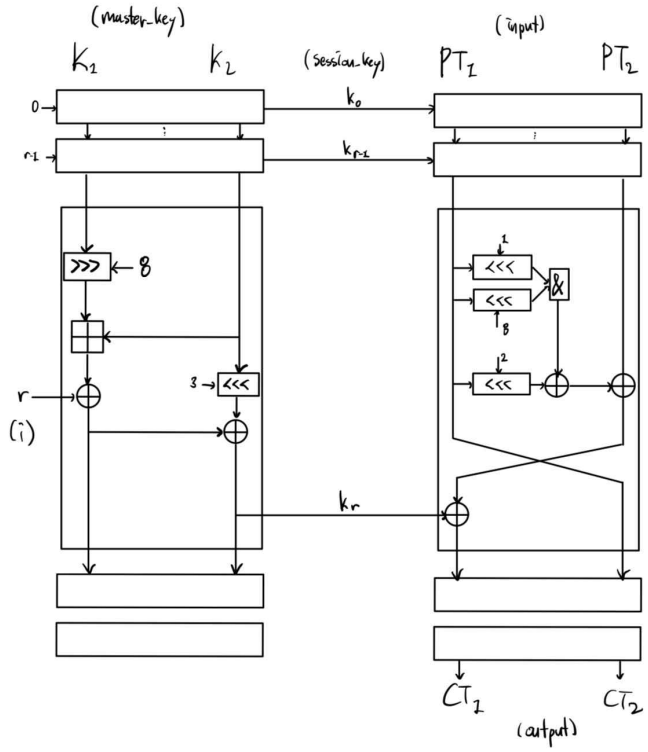
4번 문제

0. 파일 설명

** 문제풀이 코드는 "KUICS_4번 > contest.c" 에서도 확인할 수 있습니다.*
** 각 단계 최적화용 벤치마크 코드는 "KUICS_4번 > optim"에 있습니다.*
** KUICS_4번 > optim > test.sh"코드로 벤치마크를 실행합니다.*

1. 문제 풀이

해당 문제는 SPECK의 Key Scheduler와 SIMON의 Round Function을 결합한 블록 암호화 알고리즘을 AVX2 명령어를 사용해 가속화하는 것이다. 본 팀은 AVX2 명령어를 사용해 연산 속도를 가속화하는 방법과 메모리 접근의 효율성을 향상시키는 방법 두 개로 나누어 최적화를 진행하였다.



[그림 1] 문제에서 제시한 암호화 알고리즘의 도식. C 버전의 변수명을 소괄호로 표시.

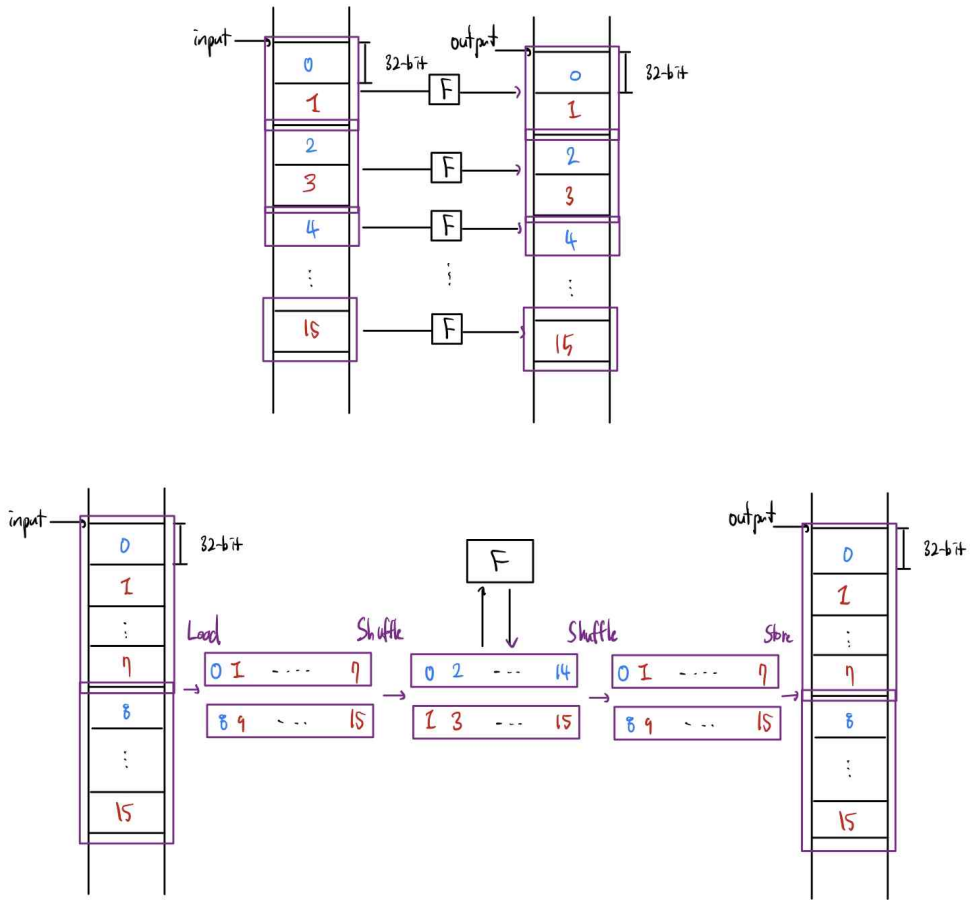
1.1 AVX2를 활용한 연산 가속
1.1.1 8개 블록 동시처리

가. C 버전 알고리즘은 함수 호출 1회에 64-bit 평문 블록 1개를 암호화하며, 64-bit 블록을 32-bit 블록 두 개로 나누어 연산을 진행한다.
나. C 버전 알고리즘에서 모든 연산은 32-bit 단위로 이루어진다.
다. AVX2 명령어 셋의 처리 단위는 256-bit이며, 32-bit 단위로 연산을 처리할 수 있다.

256 = 32 x 8 이기 때문에, AVX2 명령어를 활용하면 8개 block을 동시 처리 가능하다는 것을 알 수 있다.

8개 block이 정상적으로 동시 처리가 이루어지기 위해서는 같은 block이 AVX2 레지스터에서 동일한 32-bit 칸에 위치해야 한다. 이를 그림으로 나타내면 [그림 2] 와 같다. 파란색과 빨간색으로 나타낸 32-bit 블록들이 순서

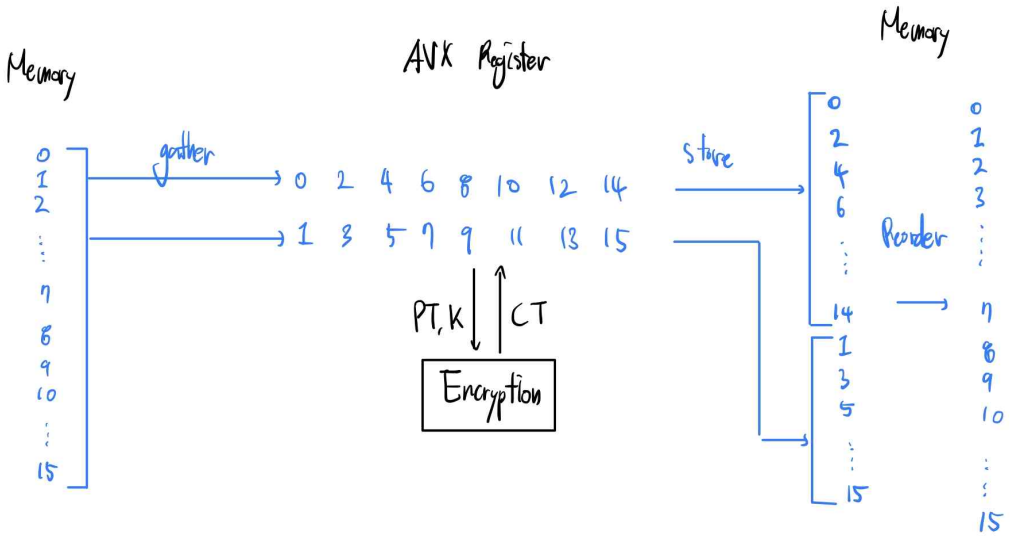
대로 각 64-bit 블록에 해당한다. [그림 2 (상)]에서는 각 블록이 별도로 처리되지만, [그림 2 (하)]에서는 동시에 8개 블록이 처리됨을 확인할 수 있다.



[그림 2] C 버전 (상) 과 8개 블록 동시처리 최적화 (하)

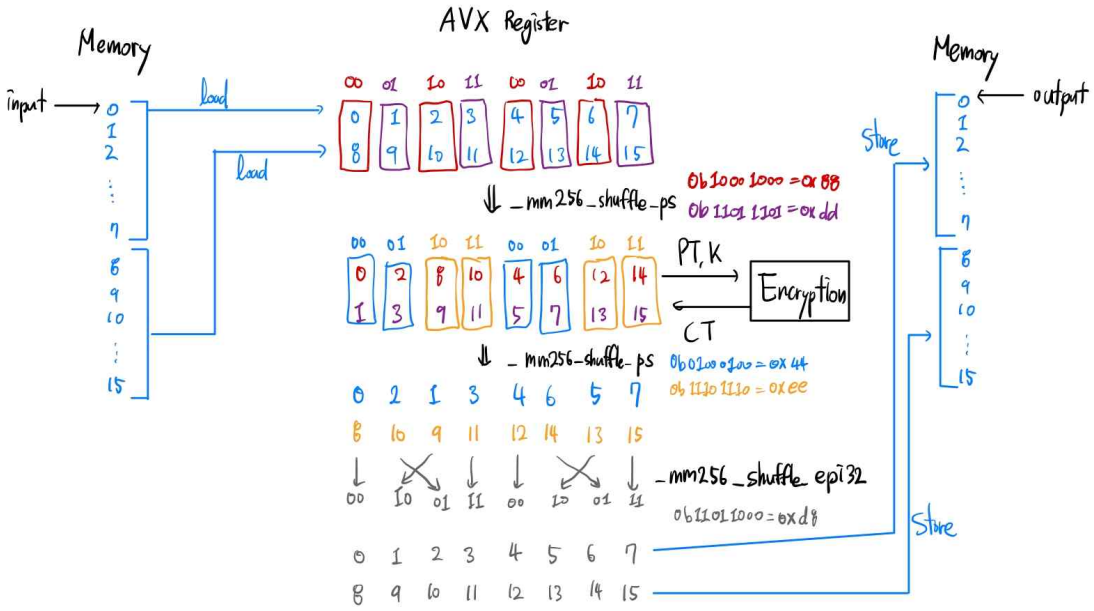
같은 블록을 동일한 위치에 블록을 위치시키는 방법으로 두 가지 방법을 실험하였다.

첫 번째는 _mm256_i32gather_epi32 명령어를 사용한 방법이다. AVX2의 gather 명령어를 사용하여 메모리에서 32-bit 간격마다 위치한 32-bit 데이터를 load할 수 있다. 그러나 gather에 대응하는 store명령인 _mm256_i32scatter_epi32 명령어는 AVX2에서 지원하지 않기 때문에 (AVX512에서 지원) 256-bit store를 한 뒤 재배치하는 작업이 필요하다. 이를 도식화하면 [그림 3] 과 같다.



[그림 3] _mm256_i32gather_epi32 명령을 사용한 8개 블록 동시처리

두 번째는 256-bit 단위로 8개 블록 데이터를 한 번에 읽어온 뒤 AVX2 레지스터 내부에서 재배열하는 방법이다. 암호화 연산 전후로 블록들이 메모리 순서대로 위치하지 않는다는 점이 첫 번째 방법과의 차이점이다. 이를 도식화하면 [그림 4]와 같다.



[그림 4] _mm256_shuffle_ps, _mm256_shuffle_epi32 명령을 사용한 8개 블록 동시처리

실험 결과 두 번째 방법이 더 효율적이었다. 따라서 최종 코드에서는 두 번째 최적화 방법을 사용하였다.

1,1,2 8-bit rotation 연산을 8-bit 재배열 연산으로 대체

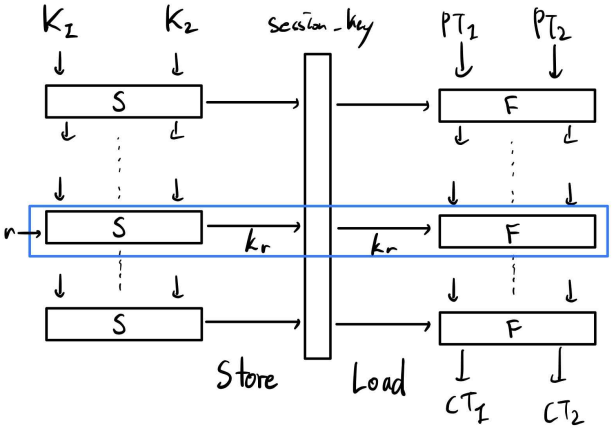
Rotation 연산 중 8-bit 단위로 이루어지는 연산에 대한 최적화 방법이다. 일반적으로 ROR, ROL 연산은 2번의 shift와 1번의 or 연산을 사용한다. 그러나 8-bit rotation의 경우 8-bit block을 재배치하는 연산 1회로 처리할 수 있다.

본 팀의 AVX2 버전 알고리즘에서는 이를 _mm256_shuffle_epi8를 사용하여 구현하였으며, r8, l8 변수가 각각 ROR, ROL을 수행할 때 필요한 인자이다.

1.2 메모리 접근 효율성 향상

1.2.1 Key Schedule과 Round Function을 같은 loop으로 병합

C 버전 알고리즘에서는 Key Scheduler과 Round Function이 각각의 함수로 분리되어 있다. 이 사이에서 Round Key를 session_key라는 변수로 전달하는데, 각 Round Key는 (1) 다른 Round에서 사용되지 않으며 (2) Scheduler와 Round Function 사이에서 접근되지 않는다. Round Key는 생성, load, store, 사용의 cycle 동안 값의 접근, 변조되지 않기 때문에, 중간의 load, store 과정을 생략하고 생성된 직후 Round Function에서 사용하도록 최적화를 진행할 수 있다.



[그림 5] 각 Round의 키 생성과 암호화는 독립적

[그림 5] 에서 S가 Key Scheduler고 F가 Round Function이다. session_key는 Round Key를 전달하는 역할만 수행하기 때문에 Key를 생성하고 바로 사용한다면 session_key를 사용하지 않아도 된다.

본 팀의 코드에서는 new_key_gen과 new_block_cipher 두 함수를 AVX2_cipher 함수로 병합하고, 각 함수에 존재하는 for loop을 하나로 묶어 Key Scheduler와 Round Function을 동일한 loop에 구현했다.

1.2.2 For loop 2회를 1회로 병합

C 버전 코드의 Round Function을 간소화하면 아래와 같다.

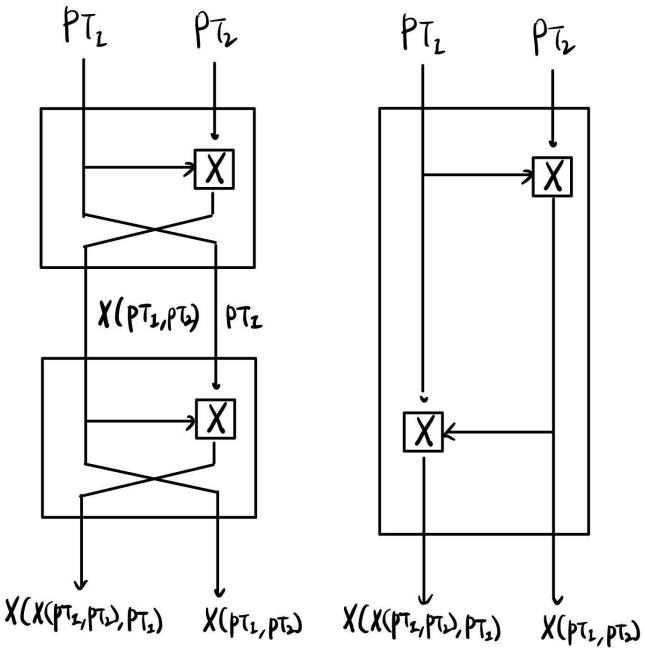
```
for (i = 0; i < NUM_ROUND; i++)
{
    pt2 = X(pt1, pt2, session_key[i]);

    tmp1 = pt1;
    pt1 = pt2;
    pt2 = tmp1;
}
```

이를 도식화하면 [그림 6 (좌)]와 같다. 한 Box가 loop 1회와 같으며, 교차된 선은 변수의 swap을 나타낸다. 이를 [그림 6 (우)]와 같이 개선하면 swap 2회를 없앨 수 있다.

해당 최적화를 간소화된 코드로 나타내면 아래와 같다.

```
for (i = 0; i < NUM_ROUND; i += 2)
{
    pt2 = X(pt1, pt2, session_key[i]);
    pt1 = X(pt2, pt1, session_key[i + 1]);
}
```

본 팀의 코드에서는 위 최적화를 적용하여 AVX_cipher 함수의 loop을 구현하였다.



[그림 6] 두 암호화 Round의 병합을 통한 Swap 제거. 최적화 전 (좌), 최적화 후 (우)

2. 문제풀이 코드

```
========= 파일 1: answer.c =========
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <xmmintrin.h>
#include <emmintrin.h>
#include <immintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <time.h>

// round of block cipher
#define NUM_ROUND 80

// size of plaintext and key size
#define BLOCK_SIZE 512
#define P_K_SIZE 2
#define SESSION_KEY_SIZE NUM_ROUND

// basic operation
#define ROR(x, r) ((x >> r) | (x << (32 - r)))
#define ROL(x, r) ((x << r) | (x >> (32 - r)))

// example: AVX2 functions; freely remove this code and write what you want in here!
```

```
#define INLINE inline __attribute__((always_inline))

#define LOAD(x) _mm256_loadu_si256((__m256i*)x)
#define STORE(x, y) _mm256_storeu_si256((__m256i*)x, y)
#define XOR(x, y) _mm256_xor_si256(x, y)
#define OR(x, y) _mm256_or_si256(x, y)
#define AND(x, y) _mm256_and_si256(x, y)
#define SHUFFLE8(x, y) _mm256_shuffle_epi8(x, y)
#define ADD(x, y) _mm256_add_epi32(x, y)
#define SHIFT_L(x, r) _mm256_slli_epi32(x, r)
#define SHIFT_R(x, r) _mm256_srli_epi32(x, r)

#define SCALAR(x) _mm256_set1_epi32(x)
#define SHUFFLE_2(x, y, i) ((__m256i)_mm256_shuffle_ps(x, y, i))
#define SHUFFLE32(x, i) _mm256_shuffle_epi32(x, i)
#define _ROL(x, r) OR(SHIFT_L(x, r), SHIFT_R(x, 32 - r))

int64_t cpucycles(void) {
    unsigned int hi, lo;
    __asm__ __volatile__("rdtsc\n\t"
                         : "=a"(lo), "=d"(hi));
    return ((int64_t)lo) | (((int64_t)hi) << 32);
}

// 64-bit data
// 64-bit key
// 32-bit x 22 rounds session key
void new_key_gen(uint32_t* master_key, uint32_t* session_key) {
    uint32_t i = 0;
    uint32_t k1, k2, tmp;

    k1 = master_key[0];
    k2 = master_key[1];

    for (i = 0; i < NUM_ROUND; i++) {
        k1 = ROR(k1, 8);
        k1 = k1 + k2;
        k1 = k1 ^ i;
        k2 = ROL(k2, 3);
        k2 = k1 ^ k2;
        session_key[i] = k2;
    }
}

void new_block_cipher(uint32_t* input, uint32_t* session_key, uint32_t* output) {
    uint32_t i = 0;
    uint32_t pt1, pt2, tmp1, tmp2;
```

```
    pt1 = input[0];
    pt2 = input[1];

    for (i = 0; i < NUM_ROUND; i++) {
        tmp1 = ROL(pt1, 1);
        tmp2 = ROL(pt1, 8);
        tmp2 = tmp1 & tmp2;
        tmp1 = ROL(pt1, 2);
        tmp2 = tmp1 ^ tmp2;
        pt2 = pt2 ^ tmp2;
        pt2 = pt2 ^ session_key[i];

        tmp1 = pt1;
        pt1 = pt2;
        pt2 = tmp1;
    }

    output[0] = pt1;
    output[1] = pt2;
}

////////////////////////////////////////////////////////////////////////////////////////
void AVX2_cipher(uint32_t* master_key, uint32_t* input, uint32_t* output) {
    uint32_t i = 0;
    __m256i k1, k2, pt1, pt2, t1, t2;
    __m256i r8, l8;
    r8 = _mm256_set_epi32(
        0x0c0f0e0d, 0x080b0a09, 0x04070605, 0x00030201,
        0x0c0f0e0d, 0x080b0a09, 0x04070605, 0x00030201);
    l8 = _mm256_set_epi32(
        0x0e0d0c0f, 0x0a09080b, 0x06050407, 0x02010003,
        0x0e0d0c0f, 0x0a09080b, 0x06050407, 0x02010003);

    // Load Key
    t1 = LOAD( master_key    );
    t2 = LOAD(&master_key[8]);

    // Shuffle s.t. operands are aligned to be processed together
    // k1: first element of each block
    // k2: second element of each block
    k1 = SHUFFLE_2(t1, t2, 0x88);
    k2 = SHUFFLE_2(t1, t2, 0xdd);

    // Load Plaintext
    t1 = LOAD( input );
    t2 = LOAD(&input[8]);
```

```
    // Shuffle s.t. operands are aligned to be processed together
    // k1: first element of each block
    // k2: second element of each block
    pt1 = SHUFFLE_2(t1, t2, 0x88);
    pt2 = SHUFFLE_2(t1, t2, 0xdd);

    for (; i < NUM_ROUND; i += 2) {
        // Key Scheduler
        k1 = XOR(ADD(SHUFFLE8(k1, r8), k2), SCALAR(i));
        k2 = XOR(k1, _ROL(k2, 3));

        // Block Cipher
        pt2 = XOR(k2, XOR(pt2, XOR(_ROL(pt1, 2), AND(_ROL(pt1, 1), SHUFFLE8(pt1, l8)))));

        // Key Scheduler
        k1 = XOR(ADD(SHUFFLE8(k1, r8), k2), SCALAR(i + 1));
        k2 = XOR(k1, _ROL(k2, 3));

        // Block Cipher
        pt1 = XOR(k2, XOR(pt1, XOR(_ROL(pt2, 2), AND(_ROL(pt2, 1), SHUFFLE8(pt2, l8)))));
    }

    // Shuffle s.t. elements have same position as one right after Load
    // Store Ciphertext
    STORE( output    , SHUFFLE32(SHUFFLE_2(pt1, pt2, 0x44), 0xd8));
    STORE(&output[8], SHUFFLE32(SHUFFLE_2(pt1, pt2, 0xee), 0xd8));
}
////////////////////////////////////////////////////////////////////////////////////////////

int main() {
    long long int kcycles, ecycles, dcycles;
    long long int cycles1, cycles2;
    int32_t i, j;

    // C implementation
    uint32_t input_C[BLOCK_SIZE][P_K_SIZE] = { 0, };
    uint32_t key_C[BLOCK_SIZE][P_K_SIZE] = { 0, };
    uint32_t session_key_C[BLOCK_SIZE][SESSION_KEY_SIZE] = { 0, };
    uint32_t output_C[BLOCK_SIZE][P_K_SIZE] = { 0, };

    // AVX implementation
    uint32_t input_AVX[BLOCK_SIZE][P_K_SIZE] = { 0, };
    uint32_t key_AVX[BLOCK_SIZE][P_K_SIZE] = { 0, };
    uint32_t session_key_AVX[BLOCK_SIZE][SESSION_KEY_SIZE] = { 0, };
    uint32_t output_AVX[BLOCK_SIZE][P_K_SIZE] = { 0, };
```

```
    // random generation for plaintext and key.
    srand(0);

    for (i = 0; i < BLOCK_SIZE; i++) {
        for (j = 0; j < P_K_SIZE; j++) {
            input_AVX[i][j] = input_C[i][j] = rand();
            key_AVX[i][j] = key_C[i][j] = rand();
        }
    }

    // execution of C implementation
    kcycles = 0;
    cycles1 = cpucycles();
    for (i = 0; i < BLOCK_SIZE; i++) {
        new_key_gen(key_C[i], session_key_C[i]);
        new_block_cipher(input_C[i], session_key_C[i], output_C[i]);
    }
    cycles2 = cpucycles();
    kcycles = cycles2 - cycles1;
    printf("C implementation runs in ..... %8lld cycles", kcycles / BLOCK_SIZE);
    printf("\n");

    // KAT and Benchmark test of AVX implementation
    kcycles = 0;
    cycles1 = cpucycles();
    ////////////////////////////////////////////////////////////////////////////////////////////
    for (i = 0; i < BLOCK_SIZE; i += 8) {
        AVX2_cipher(key_AVX[i], input_AVX[i], output_AVX[i]);
    }
    ////////////////////////////////////////////////////////////////////////////////////////////
    cycles2 = cpucycles();
    kcycles = cycles2 - cycles1;
    printf("AVX implementation runs in ..... %8lld cycles", kcycles / BLOCK_SIZE);
    printf("\n");

    for (i = 0; i < BLOCK_SIZE; i++) {
        for (j = 0; j < P_K_SIZE; j++) {
            if (output_C[i][j] != output_AVX[i][j]) {
                printf("Test failed!!!\n");
                return 0;
            }
        }
    }
}
==================================
```

3. 실행 결과

3.1 실험 환경

Virtualbox 가상환경에서 실험을 진행하였으며, OS와 GCC 버전은 아래와 같다.

OS	Ubuntu 22.04.3
GCC	gcc 11.4.0

[표 1] 실험 환경

```
4 > vagrant ssh
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-79-generic x86_64)
```

```
vagrant@ubuntu-jammy:/vagrant/optim$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

[그림 7] 실험 환경 스크린샷

3.2 실험 결과

3.2.1 최적화 과정

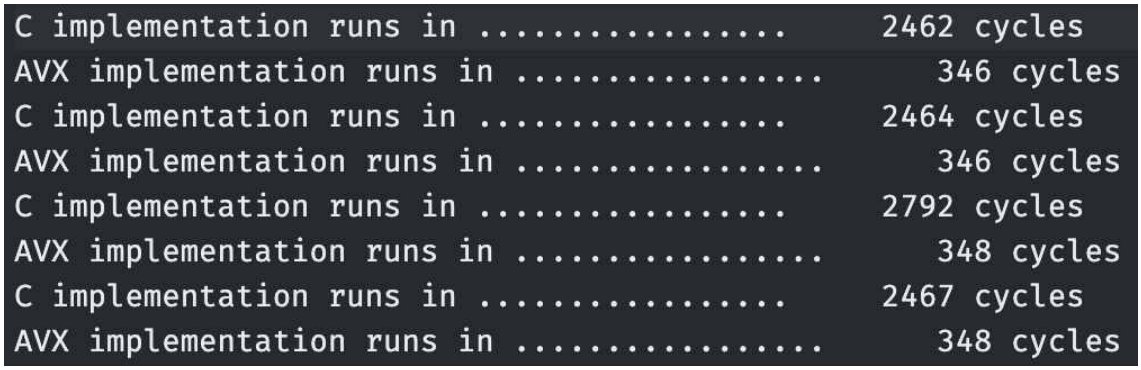
동일한 코드를 10,000회 반복 수행한 결과의 평균을 계산하였다. 각 단계의 최적화를 누적 적용한 코드를 실험하였으며, 4_merge_two_loop.c 가 최종 제출본과 동일한 최적화를 적용한 버전이다. 최종 버전에서 약 7.0배 개선된 결과를 보여준다.

```
vagrant@ubuntu-jammy:/vagrant/optim$ ./test.sh
Processing ./0_no_optim.c...
2489.8896

Processing ./1_session_key_load_store.c...
1618.5018

Processing ./2_8_block_parallel_gather.c...
432.5616

Processing ./2_8_block_parallel_shuffle.c...
422.1499

Processing ./3_byte_reorder.c...
384.1248

Processing ./4_merge_2_loop.c...
355.7709
```

[그림 7] 최적화 실행속도 비교

3.2.2 최종 버전

[그림 8]은 최종 버전이 정상 작동함을 보여주는 실행결과이다.



[그림 8] 최종 버전 실행결과

4. 참고

4.1 추가 정의한 매크로

#define SCALAR(x) _mm256_set1_epi32(x)
모든 32-bit 데이터가 같은 값으로 초기화된 AVX2 변수를 만든다.

#define SHUFFLE_2(x, y, i) ((__m256i)_mm256_shuffle_ps(x, y, i))
AVX2 변수 2개에서 i 패턴으로 값을 가져온다. i의 4비트씩 x, y의 128-bit에서 데이터를 가져오는데 사용되며, 2비트씩 128-bit 블록의 32-bit 단위 index로 작동한다.

#define SHUFFLE32(x, i) _mm256_shuffle_epi32(x, i)
하나의 AVX2 변수에서 i 패턴으로 32-bit 데이터를 섞는다. i의 각 2비트가 128-bit 블록의 32-bit 단위 index로 사용된다.

#define _ROL(x, r) OR(SHIFT_L(x, r), SHIFT_R(x, 32 - r))
ROL 매크로를 AVX2 버전으로 구현하였다.