2023 암호분석경진대회

4번 문제

외부로 유출되면 안되는 민감한 정보의 경우 안전한 암호화 알고리즘을 통해 해당 데이터를 암호화하고 이를 저장하거나 전송하게 된다. 빅데이터시대가 도래함에 따라 암호화를 필요로하는 정보의 양이 급격히 늘어나고 있다. 따라서 빅데이터 처리에 소모되는 시간을 단축시키기 위한 고속 암호화 기술에 대한 연구가 활발히 진행되고 있다. Intel CPU 상에서 암호를 고속구현하는 방법 중 하나는 SIMD 명령어 셋을 이용하여 암호화 연산을 병렬 처리 (data-parallelism) 하는 것이다. 아래에는 Intel 프로세서 상에서 C언어로 동작하도록 구현된 암호화 알고리즘이 명시되어 있다. 해당 암호화 알고리즘을 Intel 프로세서 상의 AVX2 명령어 셋을 활용하여 고속 병렬 구현하시오.

```
#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)

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 new_key_gen_AVX2(uint32_t* master_key, uint32_t* session_key){
        //example: AVX2 codes; freely remove this code and write what you want in here!
        __m256i x0, x1, tmp;

        x0 = LOAD(master_key);
        x1 = LOAD(&master_key[8]);
        tmp = ADD(x0,x1);
        STORE(&session_key[0],tmp);
}

//check input_length --> multiple of 8 * 64-bit
void new_block_cipher_AVX2(uint32_t* input, uint32_t* session_key, uint32_t* output){
        //example: AVX2 codes; freely remove this code and write what you want in here!
        __m256i x0, x1, tmp;

        x0 = LOAD(input);
        x1 = LOAD(&input[8]);
        tmp = ADD(x0,x1);
        STORE(&output[0],tmp);
}

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();


////////////////////////////////////////////////////////////////////////////////////////
	//These functions (new_key_gen, new_block_cipher) should be replaced to "new_key_gen_AVX2" and "new_block_cipher_AVX2".
	for(i=0;i<BLOCK_SIZE;i++){
		new_key_gen(key_AVX[i], session_key_AVX[i]);			// this is for testing!
		new_block_cipher(input_AVX[i], session_key_AVX[i], output_AVX[i]);// this is for testing!
	}
////////////////////////////////////////////////////////////////////////////////////////
	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;
			}
		}
	}

	cycles2 = cpucycles();
	kcycles = cycles2-cycles1;
	printf("AVX		implementation runs in ..... %8lld cycles", kcycles/BLOCK_SIZE);
	printf("\n");
```

}

주의사항

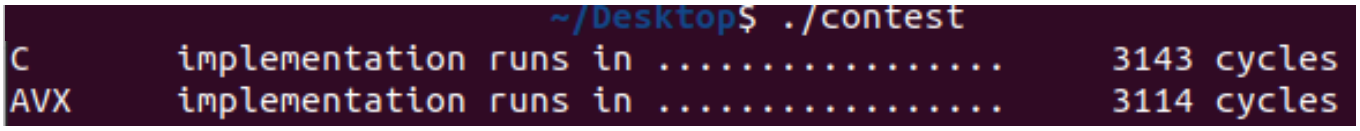
1) 구현 타겟 플랫폼은 Intel 프로세서이며 AVX2 intrinsics 명령어셋을 활용하여 구현하도록 한다. 즉 AVX2 어셈블리는 허용하지 않는다. 상세한 정보는 웹사이트를 참고하도록 한다.
(https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html)

2) 위의 코드에서 빨간색으로 표현된 부분만 수정이 가능하다. 암호화함수와 키스캐쥴링 함수를 합치는 것도 허용한다. for문을 삭제하거나 내부 파라미터를 수정하는 것도 허용한다.

3) 프로그램 컴파일과 실행에는 다음 명령어들을 사용한다. Linux 상에서 gcc를 활용하여 코딩해야 하며 Windows 및 Mac에서의 코딩은 허용하지 않는다. 결과물은 contest.c 하나만을 받으며 파일 쪼개기와 makefile 그리고 파일 형식 변환은 허용하지 않는다.

```
>> gcc -mavx2 -o contest contest.c
>> ./contest
```

현재 프로그램은 AVX2 구현이 안되어 있기에 실행 결과는 아래와 같다. 즉 실제 결과에서는 AVX2가 훨씬 더 빠른 성능이 도출되어야 한다.



4) 레퍼런스 코드 내의 변환 함수의 경우 암호학적으로 완전한 함수가 아닌 경량화된 함수이기에 충돌쌍이 발생할 수도 있다. 이 경우 충돌쌍 또한 정확한 값을 찾은 것으로 간주한다.

5) 본 레퍼런스 코드에 대한 테스팅 환경은 아래와 같다. vmware에서 테스트를 수행해도 되며 Linux 환경을 실제 설치하여 수행해도 상관없다. 단 결과물은 Linux 환경에서 동작하는 C 코드만을 허용한다.
- Ubuntu 22.04 LTS (https://ubuntu.com/download/desktop)
- vmplayer (https://www.vmware.com/go/getplayer-win)

6) 결과물은 다음 2종을 포함한다.
- C 코드 (테스트 벡터 확인 과정, 벤치마크 과정)
- 문서 (구현 기법 상세, 테스트 벡터 확인 결과, 벤치마크 결과)

7) 평가방법은 다음과 같다.
- 테스트 벡터 통과 (30점, 절대 평가)
- 문서화 (30점, 절대평가)
- 벤치마크 결과 (40점, 상대평가)