병렬처리의 또다른 방법 SIMD

SIMD 입문

박다원

병렬처리

병렬처리의 또 다른 방법 SIMD

Single Instruction Multiple Data

- 한번의 명령어에 여러가지 데이터를 처리

- xmm, ymm등과 같은 고용량 레지스터에 동일한 크기의 여러가지 자료형을 적재후 연산

- 어떻게 이게 되는가는… 매직

사용 사례

이미 곳곳에 사용되고 있다.

- 행렬연산 같은경우 한 행과 한열 이렇게 4가지의 데이터를 병렬적으로 연산하는 일이 잦은데 이때 이러한 연산을 빠르게 처리 가능

- 이미 여러분들의 프로젝트 심연 어딘가에는 적용이되어 열심히 굴러가고 있을것

간단하게 짚고 가는 CPU 상식

레지스터와 어셈블리

레지스터

CPU의 저장공간

- CPU내에는 캐시메모리이외에 실제 연산시에 사용될 데이터를 저장하는 공간인 레지스터가 존재

- 보통 이 레지스터의 크기에 따라 32비트 아키텍쳐 64비트 아키텍쳐를 논함

- 굉장히 비싼 메모리이기 때문에 32비트 64비트등 굉장히 작은 크기

- 하지만 xmm, ymm같은 특수 레지스터는 128비트, 256비트등으로 조금 더 큼

어셈블리

기계어와 1:1대응되는 명령어

- 굉장히 단순하다.
- 레지스터를 직접적으로 제어하여 연산한다.

```
_asm
{
    movdqa xmm0, arrayA
    movdqa xmm1, arrayB
    paddd xmm0, xmm1
    movdqa arrayResult, xmm0
}
```

SIMD

SIMD의 사용법

SIMD사용 법

CPU에는 SIMD용 명령어가 존재

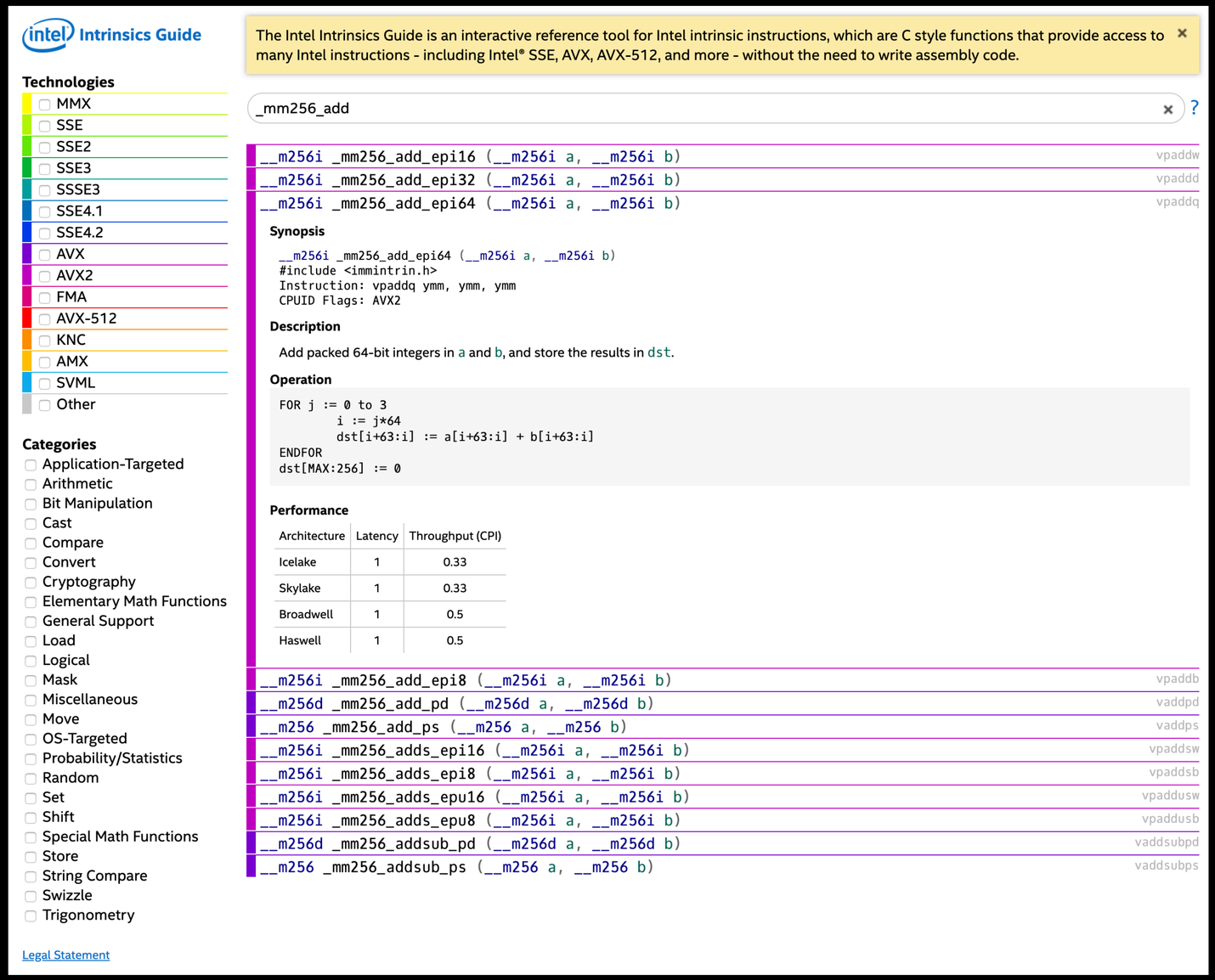
- SIMD는 CPU단에서 명령어들이 준비되어 있음
- 해당 명령어를 통해 사용가능

Intrinsics함수

명령어를 함수 형태로

- 레지스터를 자료형으로
- 명령어를 함수로
- 아까 어셈블리보다는 확실히 훨씬 쉽다.

https://software.intel.com/sites/landingpage/IntrinsicsGuide/



SIMD 활용

SIMD 활용
백준문제 풀어보기

이동

성공 분류

Platinum II

고속 푸리에 변환 수학

난이도 제공: solved.ac — 난이도 투표하러 가기

시간 제한	메모리 제한	제출	정답	맞은 사람	정답 비율
1 초	512 MB	1841	647	347	46.892%

문제

N개의 수가 있는 X와 Y가 있다. 이때 X나 Y를 순환 이동시킬 수 있다. 순환 이동이란 마지막 원소를 제거하고 그 수를 맨 앞으로 다시 삽입하는 것을 말한다. 예를 들어, {1, 2, 3}을 순환 이동시키면 {3, 1, 2}가 될 것이고, {3, 1, 2}는 {2, 3, 1}이 된다. 순환 이동은 0번 또는 그 이상 할 수 있다. 이 모든 순환 이동을 한 후에 점수를 구하면 된다. 점수 S는 다음과 같이 구한다.

S = X[0]×Y[0] + X[1]×Y[1] + ... + X[N-1]×Y[N-1]

이때 S를 최대로 하면 된다.

입력

첫째 줄에 N이 주어진다. 둘째 줄에는 X에 들어있는 N개의 수가 주어진다. 셋째 줄에는 Y에 있는 수가 모두 주어진다. N은 60,000보다 작거나 같은 자연수이고, X와 Y에 들어있는 모든 수는 100보다 작은 자연수 또는 0이다.

출력

첫째 줄에 S의 최댓값을 출력한다.

예제 입력 1 복사

```
4
1 2 3 4
6 7 8 5
```

예제 출력 1 복사

```
70
```

1067번 이동

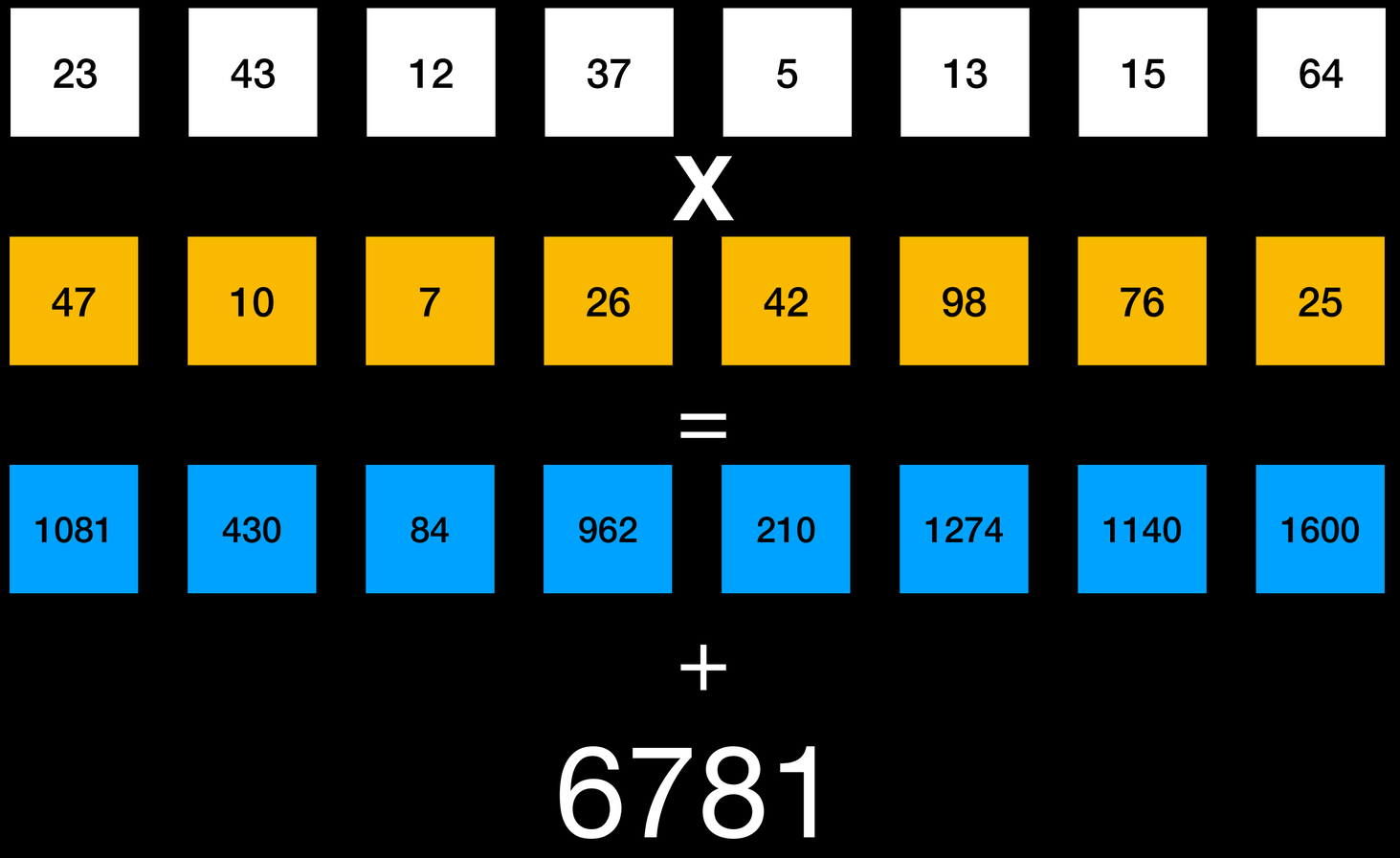
플레2 문제를 날로 먹는 방법

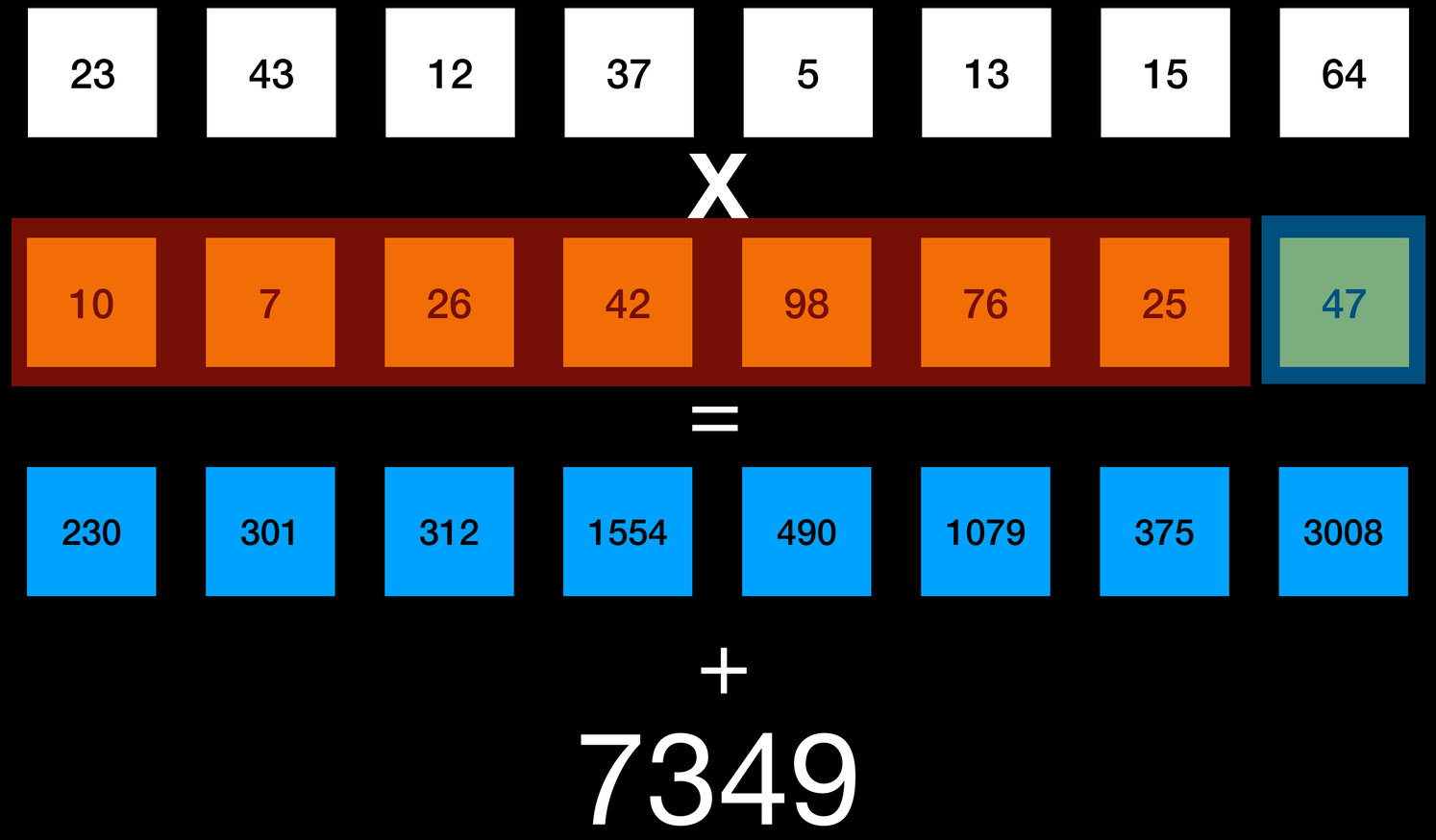
- 원래는 FFT문제

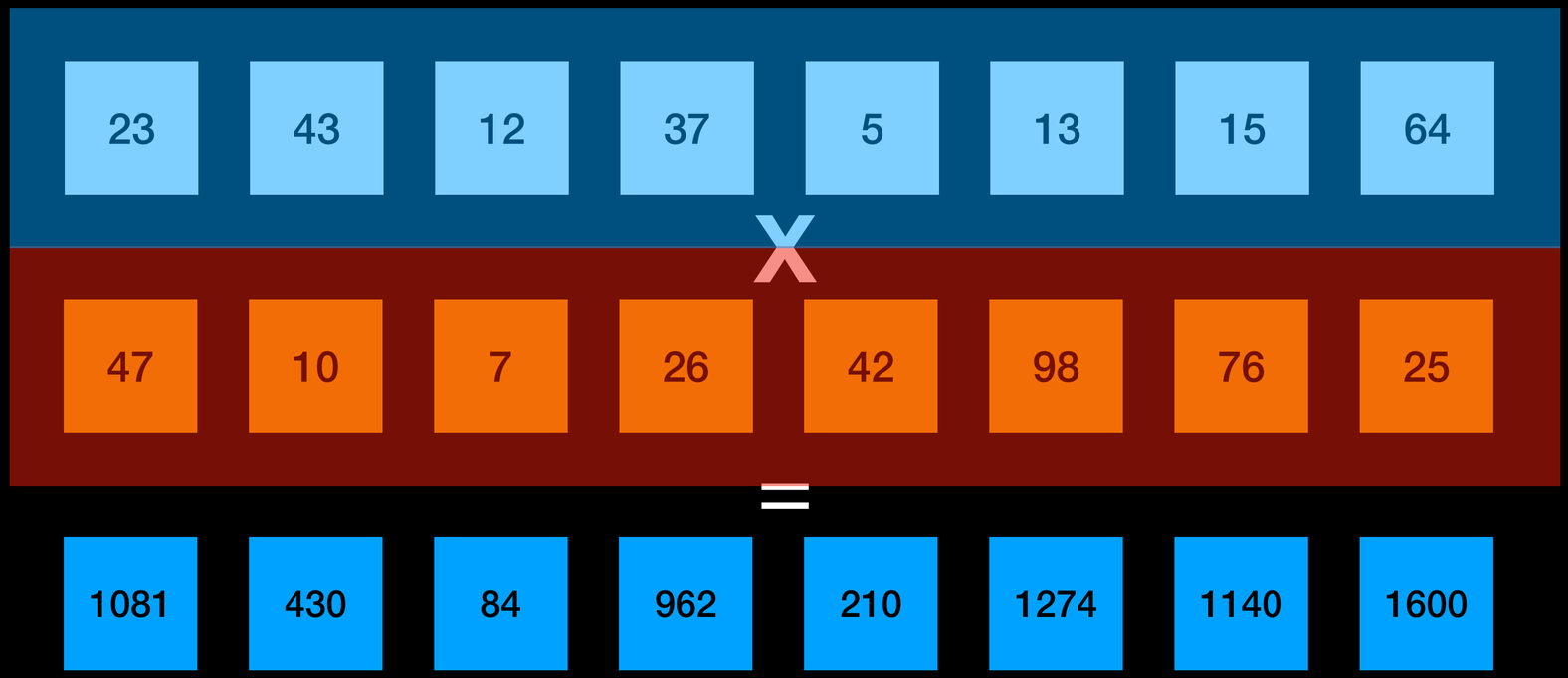
- 곱셈과 덧셈을 대략 60000*60000회 즉 36억회정도

- 단순한 연산들이니 초당 수억회이상 연산을 기대해볼수 있지만 그래도 36억회는 너무 많다.

- 하지만 한번에 여러개의 데이터를 한번에 연산한다면…?


23 43 12 37 5 13 15 64
x
47 10 7 26 42 98 76 25
=
1081 430 84 962 210 1274 1140 1600
+
6781


23 43 12 37 5 13 15 64
x
10 7 26 42 98 76 25 47
=
230 301 312 1554 490 1079 375 3008
+
7349


23 43 12 37 5 13 15 64
x
47 10 7 26 42 98 76 25
=
1081 430 84 962 210 1274 1140 1600
+
6781

몇개까지 한번에 할수 있을까?

- 현재 사용할 AVX2에서 SIMD용 레지스터 ymm의 크기는 256비트
- 8비트자료형 32개 16비트자료형 16개 32비트 자료형 8개 64비트 자료형 4개
- 각숫자는 -100~100사이의 수로 8비트로 표현가능
- 그러나, 합의 최대 크기는 100 * 100 * 60000 = 6억
- 합을 저장하려면 32비트의 자료형이 요구된다.

두가지 연산을 SIMD로

곰셈을 병렬화 해보자

- Alignas CPP11 표준 메모리 정렬
- __m256i 256비트를 저장할수 있는 자료형
- _mm256_setzero_si256 0으로 초기화
- _mm256_load_si256 해당주소로 부터 256비트를 읽어와 저장
- _mm256_mullo_epi16 32비트중 하위 16비트 끼리 곱하여 32비트 자료형 형태로 저장
- _mm256_add_epi32 32비트끼리 덧셈
- _mm256_store_si256 배열에 저장

```
void simd32bit8data()
{
    const int simdMax = n / SIMD_COUNT;
    const int normalMax = n % SIMD_COUNT;
    alignas(32) int arr[SIMD_COUNT];

    int ans = 0;
    for ( int i = 0; i < n; ++i )
    {
        __m256i tmpSimd = _mm256_setzero_si256();
        int tmp = 0;
        for ( int j = 0; j < simdMax; ++j )
        {
            int base = j * SIMD_COUNT;
            int baseY = base + i;
            __m256i xSimd = _mm256_load_si256((const __m256i*)(x + base));
            __m256i ySimd = _mm256_load_si256((const __m256i*)(y + baseY));
            __m256i result = _mm256_mullo_epi16( xSimd, ySimd );
            tmpSimd = _mm256_add_epi32(tmpSimd, result);
        }

        _mm256_store_si256((__m256i*)arr, tmpSimd);
        tmp += arr[0] + arr[1] + arr[2] + arr[3] + arr[4] + arr[5] + arr[6] + arr[7];

        for ( int j = 0; j < normalMax; ++j )
        {
            const int index = simdMax * SIMD_COUNT + j;
            tmp += x[ index ] * y[ index + i ];
        }
        ans = max(ans, tmp);
    }
```

24344478	jf297	1067	시간 초과			C++17	1914 B	19일 전

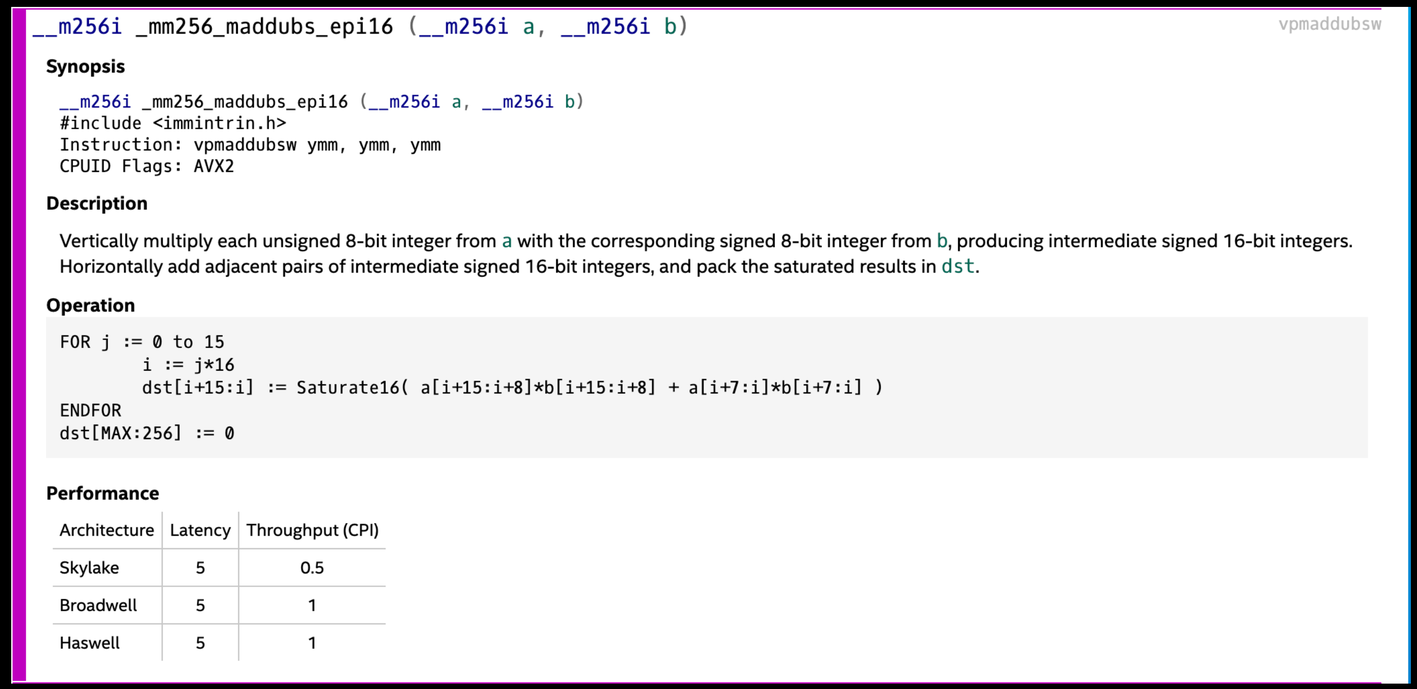
시간초과

8배로는 부족하다...

마법의 함수 등장

8비트 자료형으로 더 많은 데이터를 처리

- SIMD에는 굉장히 특이한 함수가 많다.
- _mm256_maddubs_epi16은 그중 하나로 수직적으로는 곱하고 수평적으로는 더해주는 함수



`__m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)` vpmaddubsw

Synopsis

```
__m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vpmaddubsw ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

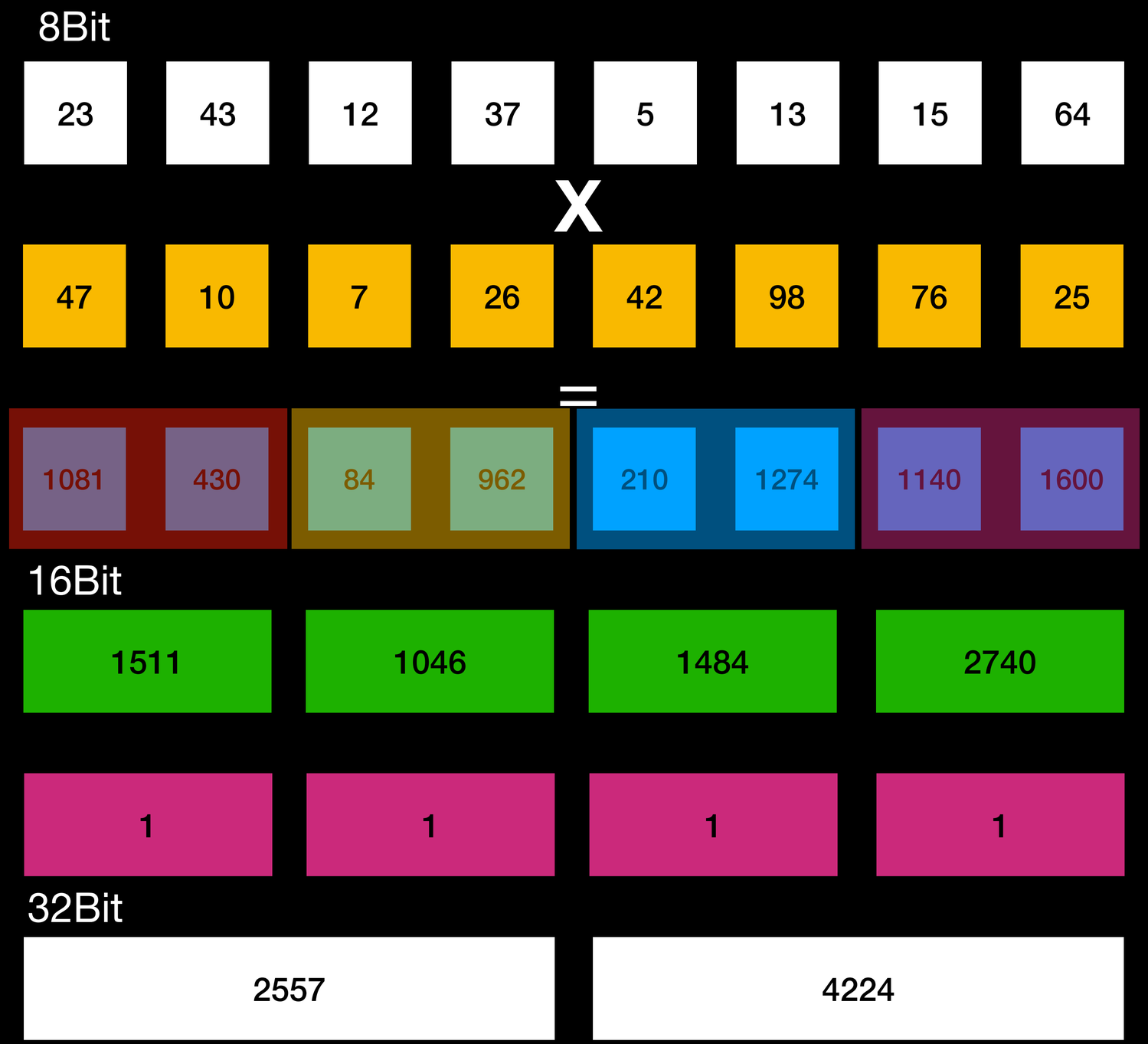
Vertically multiply each unsigned 8-bit integer from a with the corresponding signed 8-bit integer from b, producing intermediate signed 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers, and pack the saturated results in dst.

Operation

```
FOR j := 0 to 15
        i := j*16
        dst[i+15:i] := Saturate16( a[i+15:i+8]*b[i+15:i+8] + a[i+7:i]*b[i+7:i] )
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	5	0.5
Broadwell	5	1
Haswell	5	1


8Bit
23
43
12
37
5
13
15
64
X
47
10
7
26
42
98
76
25
=
1081
430
84
962
210
1274
1140
1600
16Bit
1511
1046
1484
2740
1
1
1
1
32Bit
2557
4224

이거다!

8비트의 데이터로 곱셈덧셈이 가능해졌다.

구현방법

약간의 트릭을 이용

- 실제 데이터 두개를 8비트로 곱하고
- 여기에 1로 채워진 데이터를 연산하여 더하기

```
void simd8bit32data()
{
    const int simdMax = n / SIMD_COUNT;
    const int normalMax = n % SIMD_COUNT;
    alignas(32) int arr[SIMD_COUNT];

    int ans = 0;
    for ( int i = 0; i < n; ++i )
    {
        __m256i tmpSimd = _mm256_setzero_si256();
        int tmp = 0;
        for ( int j = 0; j < simdMax; ++j )
        {
            int base = j * SIMD_COUNT;
            int baseY = base + i;
            __m256i xSimd = _mm256_load_si256((const __m256i*)(x + base));
            __m256i ySimd = _mm256_load_si256((const __m256i*)(y + baseY));
            __m256i result16 = _mm256_maddubs_epi16( xSimd, ySimd );
            __m256i result32 = _mm256_madd_epi16( result16, _mm256_set1_epi16(1) );
            tmpSimd = _mm256_add_epi32(tmpSimd, result32);
        }

        _mm256_store_si256((__m256i*)arr, tmpSimd);
        tmp += arr[0] + arr[1] + arr[2] + arr[3] + arr[4] + arr[5] + arr[6] + arr[7];

        for ( int j = 0; j < normalMax; ++j )
        {
            const int index = simdMax * SIMD_COUNT + j;
            tmp += x[ index ] * y[ index + i ];
        }
        ans = max(ans, tmp);
    }

    cout << ans << endl;
}
```

24344526	jf297	1067	맞았습니다!!	2956 KB	428 ms	C++17	1025 B	19일 전

야호!