Орчин үеийн архитектурууд

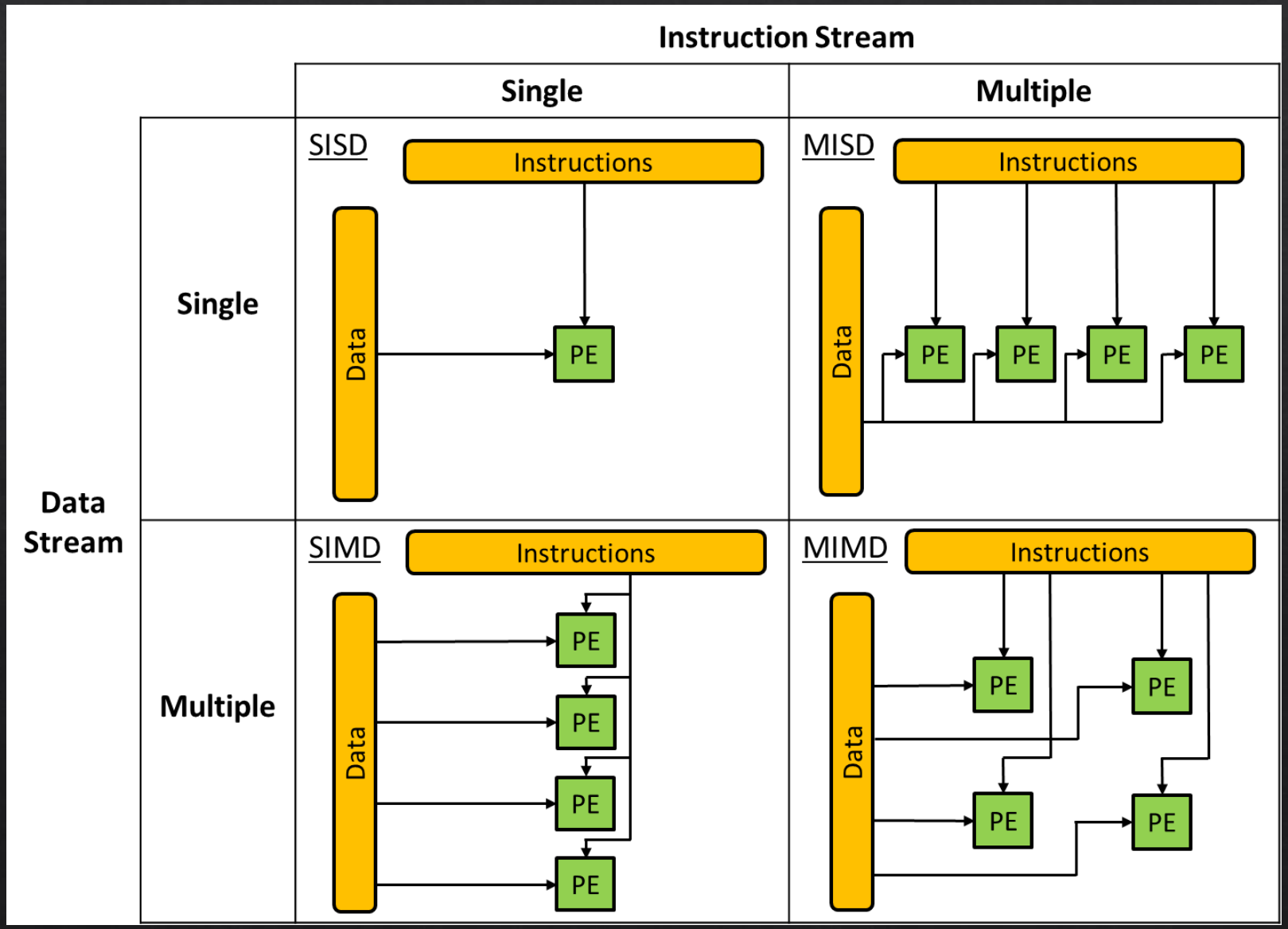
F.CS306 ПАРАЛЛЕЛ ПРОГРАММЧЛАЛ – Лекц 6

Г.ГАНБАТ
ganbatg@must.edu.mn

Хичээлийн агуулга

- Flynn’s taxonomy
- SIMD параллелизмын үндэс
- түгээмэл CPU дээрх алгоритмын C / C ++ векторчлолын тухай

Flynn's Taxonomy (1966)

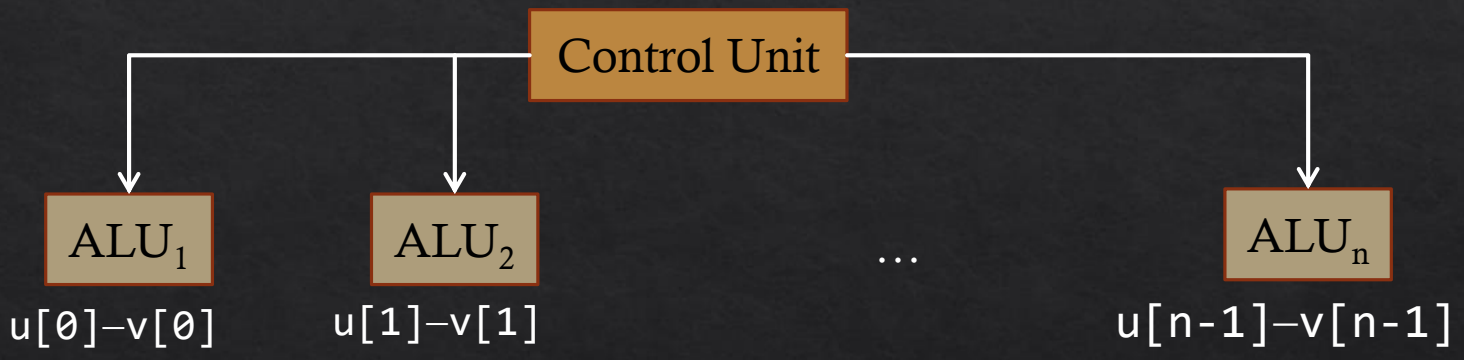


Параллелизмын түвшин

- **Multiple cores.**
  - Олон цөмтэй MIMD параллелизм
  - thread-үүдийг асинхрон, бие даасан байдлаар ажиллуулдаг
- **Vector units.**
  - Цөмүүд нь SIMD суурилсэн вектор нэгжтэй
  - Өгөгдлийн түвшинд параллелчилдаг
- **Instruction-level parallelism**.
  - Pipelining: instruction fetch, instruction decode and register fetch, execute, memory access, and register write-back.
  - Superscalar: олон (бие даасан) зааварчилгааг зэрэгцээгээр ажиллуулдаг.

SIMD (Single Instruction, Multiple Data)

```
//Mapping element-wise subtraction onto SIMD
for (i = 0; i<n; i++) w[i] = u[i]–v[i];
```

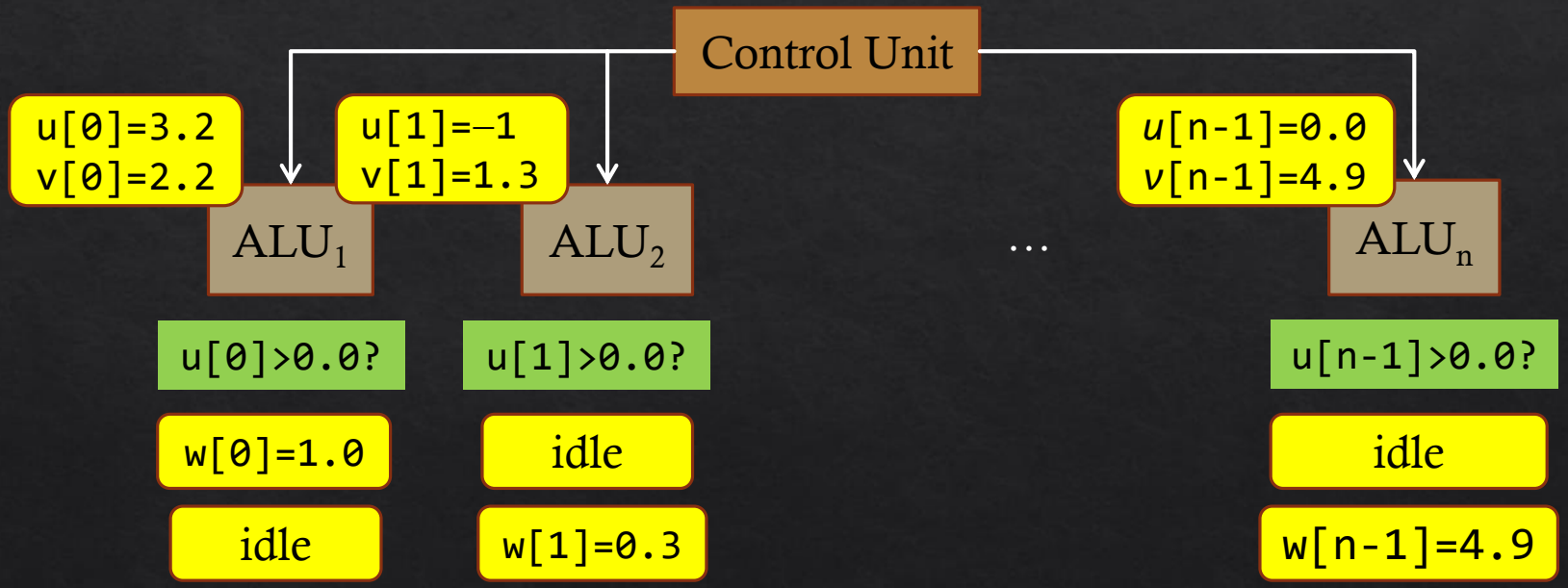


- Хэрэв бидэнд олон өгөгдөл, олон ALU байхгүй бол яах вэ?
  - Ажлыг хувааж, давталтаар гүйцэтгэнэ.

SIMD

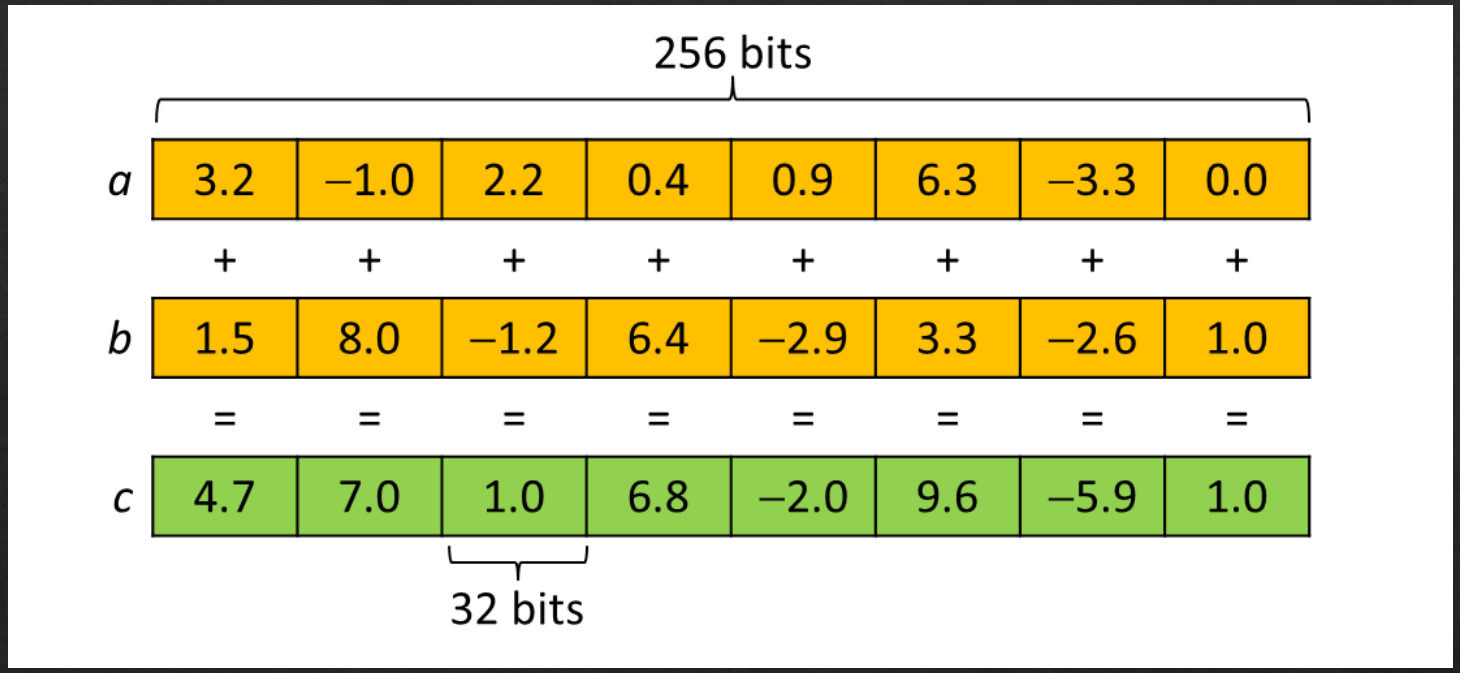
```
//Mapping a Conditional Statement onto SIMD
for (i = 0; i<n; i++)
    if (u[i] > 0)
        w[i] = u[i]-v[i];
    else
        w[i] = u[i]+v[i];
```

Бүх ALU ижил зааврыг ажилуулах (зэрэгцээгээр) шаардлагатай эсвэл idle горимд байна. ***Жишээний хувьд үр ашиг 50%.***



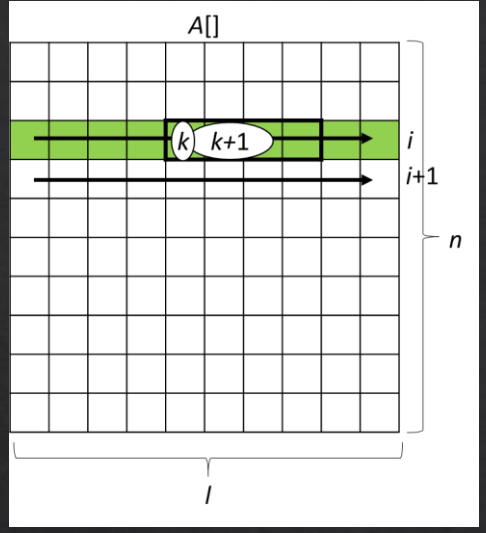
- Орчин үеийн CPU цөмүүд ерөнхийдөө хэд хэдэн өгөгдлийн элементүүдтэй зэрэгцээ ажилладаг вектор нэгжүүдтэй байдаг.
- CUDA-enabled GPU-ийн thread-үүд SIMD маягаар ажилладаг

AVX2 регистрүүдтэй SIMD

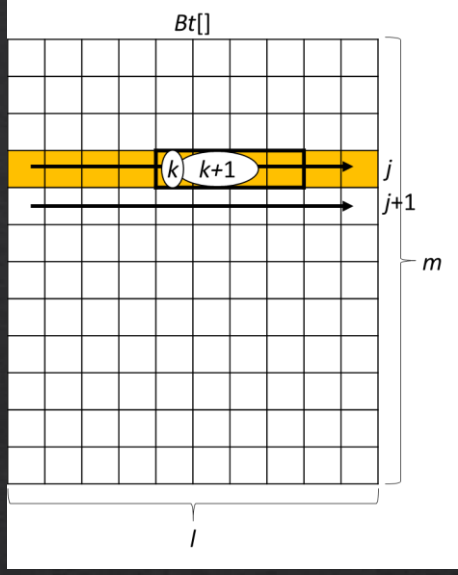


- 8 single-precision (32-bit) floating-point тоо хадгалсан 2ш Advanced Vector Extensions (AVX) регистрийн параллел нэмэх үйлдэл. Тоо бүрийг *vector* гэнэ.

```
// C/C++ дээрх AVX2-програмчлал
__m256 a, b, c;           // AVX регисртүүдийг зарлах
...                       // a, b-д утга онооно
c = _mm256_add_ps(a, b); // c[0:8] = a[0:8] + b[0:8]
```

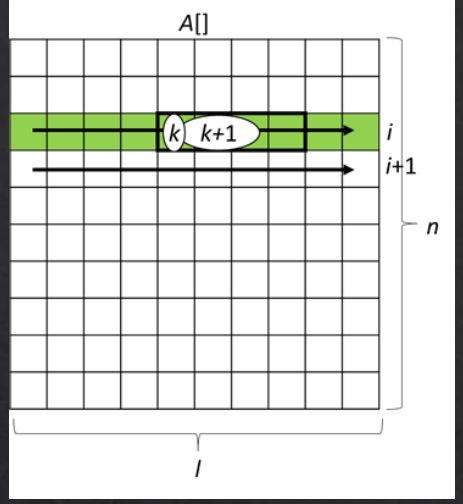


AVX2 програмчлал: эргүүлсэн MatMult

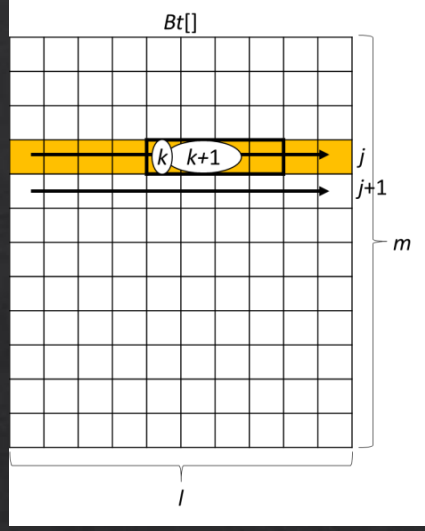


AV	A[i*L]	A[i*L+1]	A[i*L+2]	A[i*L+3]	A[i*L+4]	A[i*L+5]	A[i*L+6]	A[i*L+7]
	*	*	*	*	*	*	*	*
BV	B[j*L]	B[j*L+1]	B[j*L+2]	B[j*L+3]	B[j*L+4]	B[j*L+5]	B[j*L+6]	B[j*L+7]
	+	+	+	+	+	+	+	+
X	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]
	=	=	=	=	=	=	=	=
X	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]

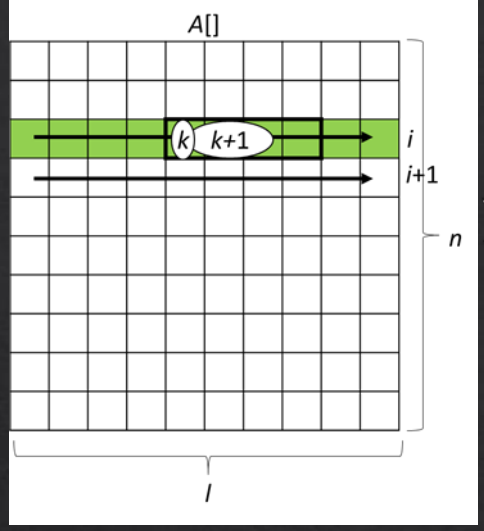
`_mm256_fmadd_ps(AV,BV,X)` эргүүлсэн матрицийн **векторчилсон** үржвэрийн функц



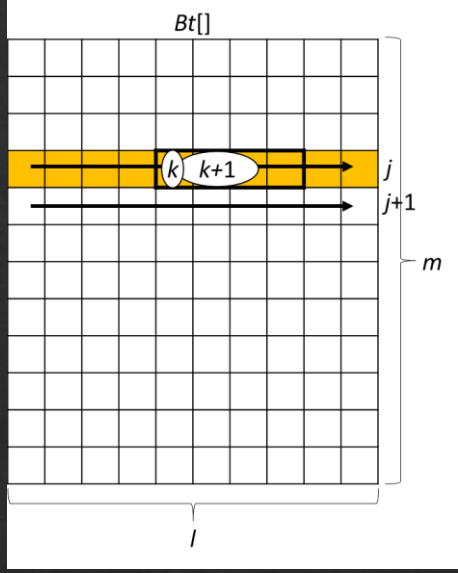
AVX2 програмчлал: эргүүлсэн MatMult




```
// AVX2-той Transpose-and-Multiply
void avx2_tmm(float * A, float * B,float * C,
              uint64_t M, uint64_t L,uint64_t N) {
    for (uint64_t i=0; i<N; i++)
        for (uint64_t j=0; j<M; j++) {
            __m256 X = _mm256_setzero_ps();
            for (uint64_t k=0; k<L; k+=8) {
                const __m256 AV = _mm256_load_ps(A+i*L+k);
                const __m256 BV = _mm256_load_ps(B+j*L+k);
                X = _mm256_fmadd_ps(AV, BV, X);
            }
            C[i*N+j] = hsum_avx(X);
        }
}
```



AVX2 програмчлал: эргүүлсэн MatMult болон энгийн MatMult хоёрын ялгаа



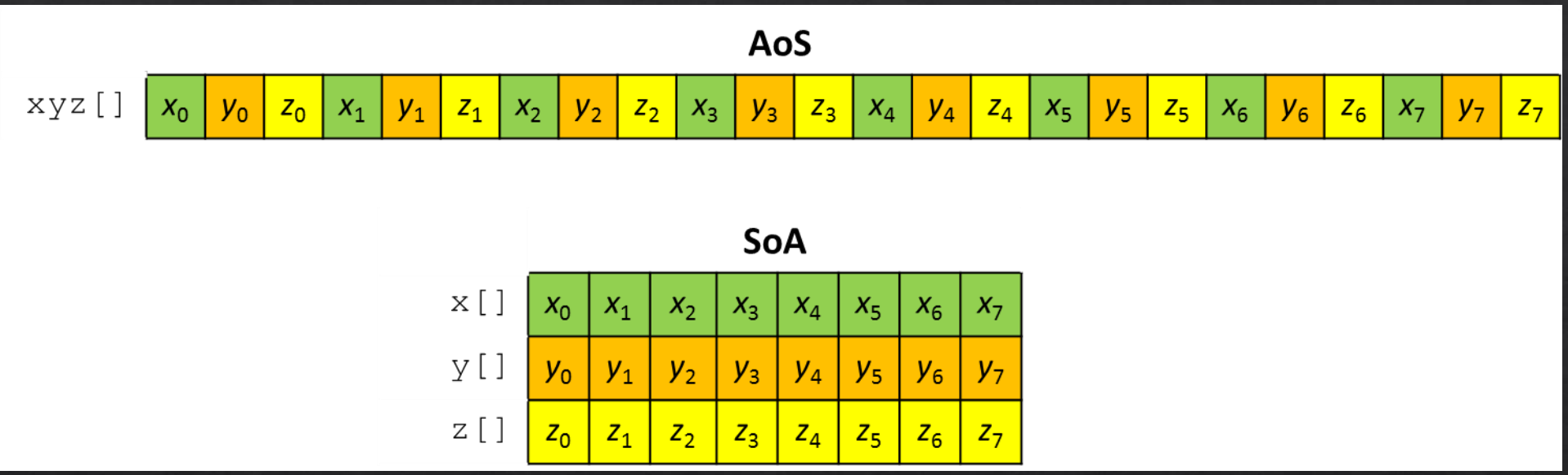
i7-6800K: m = 1K, l = 2K, n = 4K	
`#elapsed time (plain_tmm):`	`12.29s`
`#elapsed time (avx2_tmm):`	`2.13s`

Speedup: 5.8

6-core i7-6800K: m = 1K, l = 2K, n = 4K OpenMP-ын 12 thread	
`#elapsed time (avx2_tmm_multi):`	`0.32s`

Speedup: 6.7×5.8

AoS ба SoA



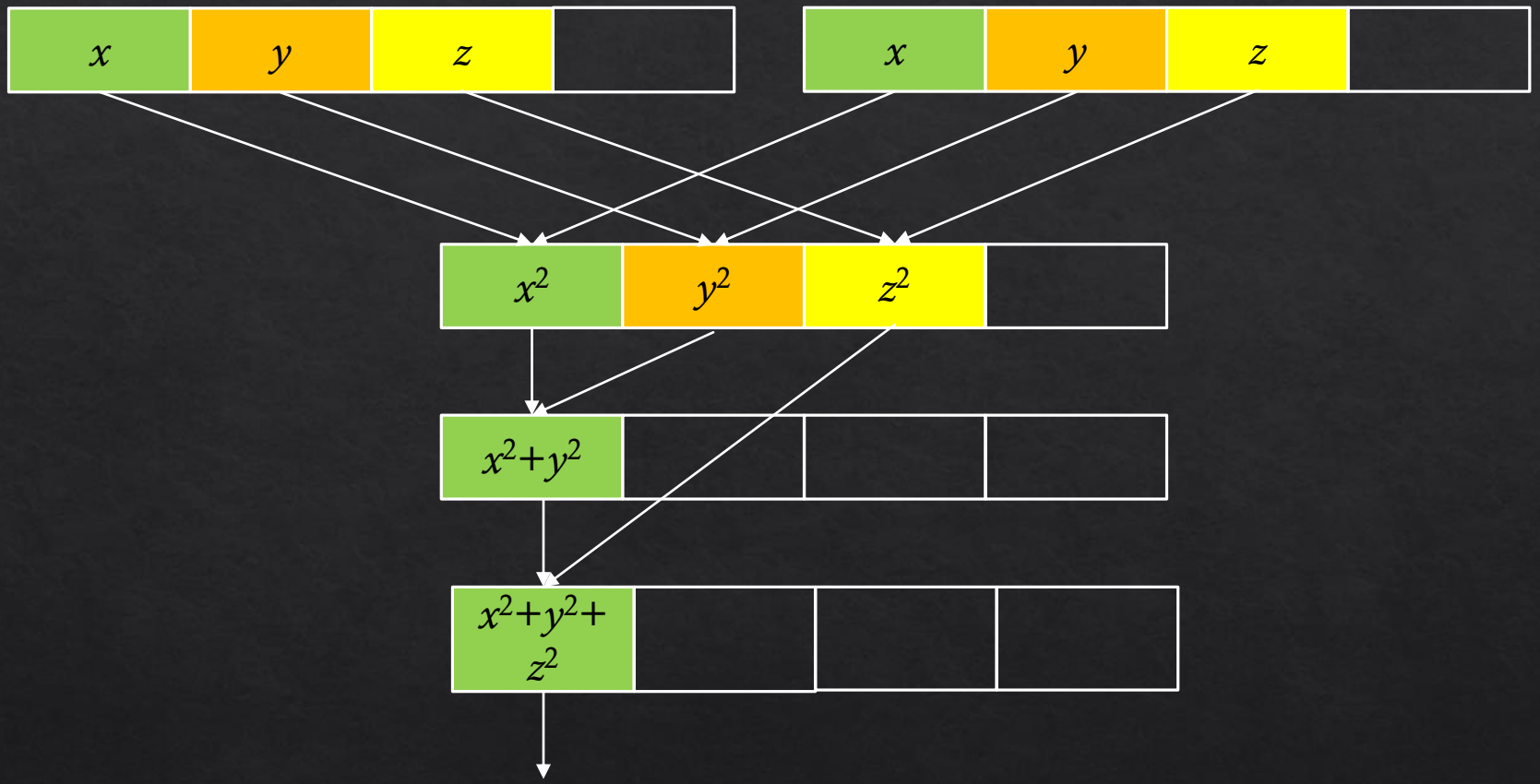
- Векторыг нормалчлах зорилгоор AoS ба SoA ын SIMD-friendliness байдлыг харьцуулахад *n* ширхэг бодит тоон 3D векторын цуглуулга хэрэглэнэ.
- **AoS (Array of Structures):** утгуудыг солбицуулан хадгалсан нэг массив
- **SoA(Structure of Arrays):** хэмжээс бүрт нэг массивтэй. Массив бүр зөвхөн холбогдох хэмжээсийн утгуудыг хадгалдаг

AoS дээрх вектор нормалчлал


```
//Non-vectorized with plain AoS layout 3D vector normalization
void plain_aos_norm(float * xyz, uint64_t length) {
    for (uint64_t i=0; i<3*length; i+=3) {
        const float x = xyz[i+0];
        const float y = xyz[i+1];
        const float z = xyz[i+2];
        float irho = 1.0f/std::sqrt(x*x+y*y+z*z);
        xyz[i+0] *= irho;
        xyz[i+1] *= irho;
        xyz[i+2] *= irho;
    }
}
```

$v_i = (x_i, y_i, z_i)$ байх $\hat{v}_i = \frac{v_i}{\|v_i\|} = \left(\frac{x_i}{\rho_i}, \frac{y_i}{\rho_i}, \frac{z_i}{\rho_i}\right)$. Энд $\rho_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$

AoS дээрх вектор нормалчлал



- 128-бит регистр бүрэн ашиглагдахгүй
- Квадратуудыг нэгтгэхэд хөршүүдийн хоорондох үйлдлүүдийг шаарддаг ба ингэснээр урвуу квадрат язгуурыг тооцоолоход зөвхөн нэг утга үлдэнэ
- Том вектор регистрүүд олон байх нь ашиггүй

SoA дээрх вектор нормалчлал

SoA

x[]	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
y[]	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
z[]	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7

$$v_i = (x_i, y_i, z_i) \text{ байх } \hat{v}_i = \left(\frac{x_i}{\rho_i}, \frac{y_i}{\rho_i}, \frac{z_i}{\rho_i}\right),$$

$$\text{Энд } \rho_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$$

```
//AVX-Vectorized SoA layout 3D vector normalization
void avx_soa_norm(float *x, float *y, float *z, uint64_t length) {
    for (uint64_t i=0; i<length; i+=8) {
        __m256 X = _mm256_load_ps(x+i);   // aligned loads
        __m256 Y = _mm256_load_ps(y+i);
        __m256 Z = _mm256_load_ps(z+i);

       __m256 R = _mm256_fmadd_ps(X,X,   // R <- X*X+Y*Y+Z*Z
                                  _mm256_fmadd_ps(Y,Y,_mm256_mul_ps(Z,Z)));

        R = _mm256_rsqrt_ps(R); // R <- 1/sqrt(R)

        _mm256_store_ps(x+i, _mm256_mul_ps(X, R)); // aligned stores
        _mm256_store_ps(y+i, _mm256_mul_ps(Y, R));
        _mm256_store_ps(z+i, _mm256_mul_ps(Z, R));
    }
}
```

Жишээ:
AoS дээрх Vectorized нормалчлал

1. AoS форматад гурван 3D векторыг гурван 256-bit регистр ашиглан SoA формат руу шилжүүлнэ.
2. SoA форматыг ашиглан Vectorized SIMD тооцооллыг хийнэ.
3. SoA-ээс үр дүнг AoS формат руу шилжүүлнэ.

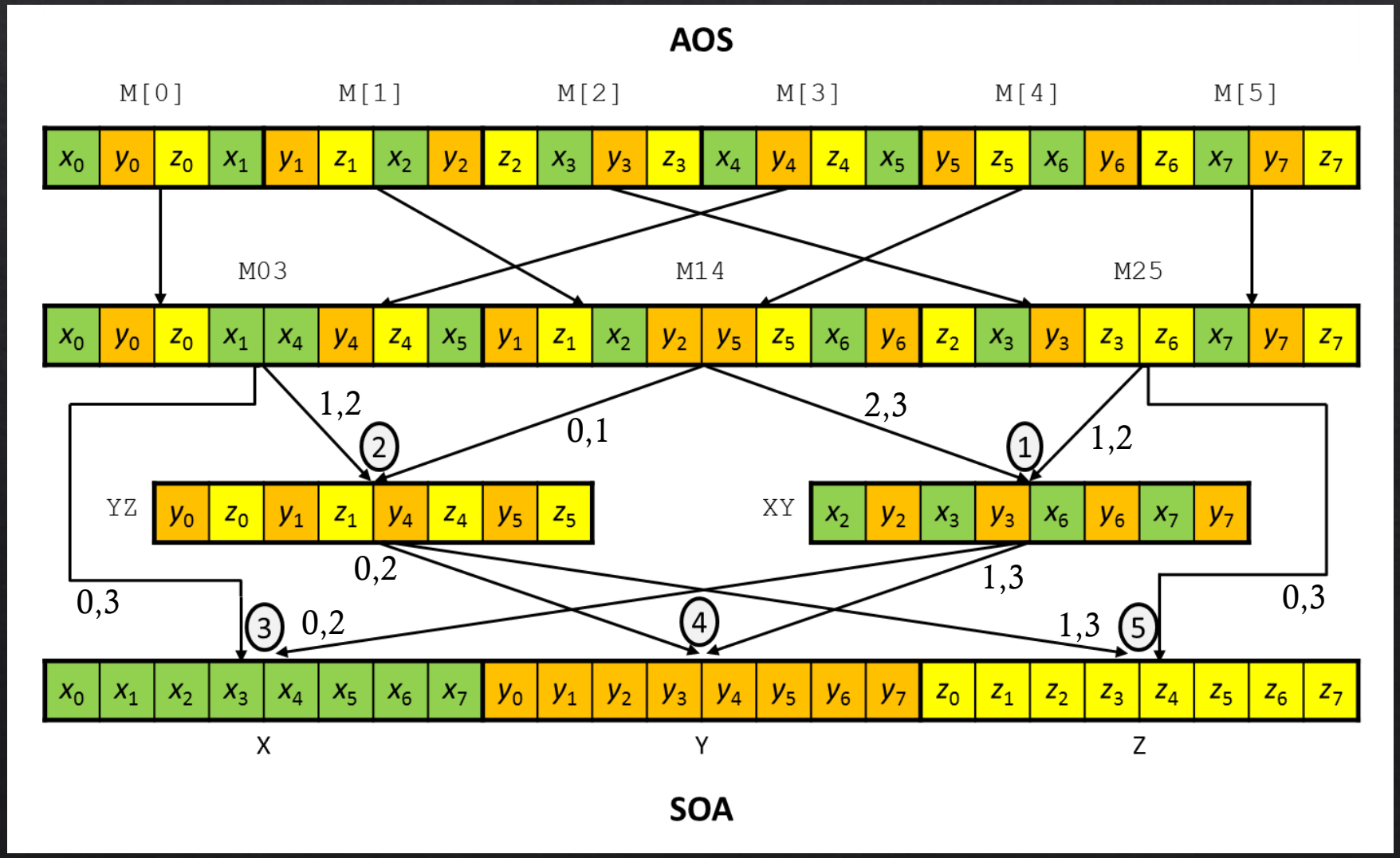
i7-6800K: $n = 2^{28}$	
`#elapsed time (plain_aos_normailze):`	`0.72s`
`#elapsed time (avx_aos_normalize):`	`0.33s`

Speedup: 2.2

AOS2SOA

```
// AOS2SOA: XYZXYZXYZXYZXYZXYZXYZXYZ --> XXXXXXX YYYYYYY ZZZZZZZ
for (uint64_t i=0; i<length; i+=8) {
    // registers: NOTE: M is an SSE pointer (length 4)
    __m128 *M=(__m128*) (xyz+i);
    __m256 M03, M14, M25;
    // load lower halves
    M03 = _mm256_castps128_ps256(M[0]);
    M14 = _mm256_castps128_ps256(M[1]);
    M25 = _mm256_castps128_ps256(M[2]);
    // load upper halves
    M03 = _mm256_insertf128_ps(M03 ,M[3],1);
    M14 = _mm256_insertf128_ps(M14 ,M[4],1);
    M25 = _mm256_insertf128_ps(M25 ,M[5],1);
    // everyday I'm shuffling...
    __m256 XY = _mm256_shuffle_ps(M14, M25, _MM_SHUFFLE( 2,1,3,2));
    __m256 YZ = _mm256_shuffle_ps(M03, M14, _MM_SHUFFLE( 1,0,2,1));
    __m256 X = _mm256_shuffle_ps(M03, XY , _MM_SHUFFLE( 2,0,3,0));
    __m256 Y = _mm256_shuffle_ps(YZ , XY , _MM_SHUFFLE( 3,1,2,0));
    __m256 Z = _mm256_shuffle_ps(YZ , M25, _MM_SHUFFLE( 3,0,3,1));
}
```

Vectorized Shuffling хэрэглэн AoS -ээс SoA үүсгэх



Баяралалаа.