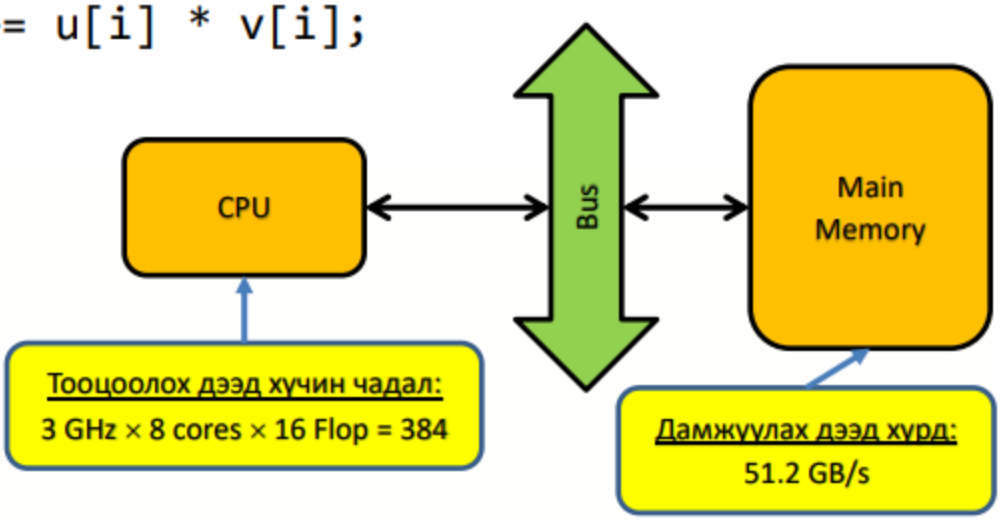
Лаб 3 Тайлан (B170900033)

Вектор регистр ашигласан алгоритмууд

Зорилго: Лекцийн хичээл дээр үзсэн ойлголтуудаа батагана, Параллел алгоритмын оновчлол хийж сурах, дадлагажих

Example: von neumann architecture dot product тооцоолол

```
// Dot Product
double dotp = 0.0;
for (int i = 0; i<n; i++)
    dotp += u[i] * v[i];
```



- Тооцоолох хугацаа: $t_{comp} = \frac{2\ GFlop}{384\ GFlop/s} = 5.2\ ms$
  - Үйлдлийн тоо: $2 \cdot n = 2^{31}$ Flops = 2 GFlops
- Өгөгдөл зөөх хугацаа: $t_{mem} = \frac{16\ GB}{51.2\ GB/s} = 312.5\ ms$
  - Зөөх өгөгдлийн хэмжээ: $2 \cdot 2^{30} \cdot 8$ B = 16 GB
- Ажиллах хугацаа : $t_{exec} \geq max(5.2ms, 312.5ms) = 312.5ms$
  - Боломжит хүчин чадал: $\frac{2\ GFlop}{312.5\ ms} = 6.4\ GFlop/s$ (<2% of peak)

1. AVX2 ашигласан Transpose-and-Multiply

```
%%writefile lab3_1AVX.cpp
#include <random>// prng
#include <iostream>
#include <cstdint>
```

```
#include <cstdlib>
#include <inttypes.h>
#include <vector>
#include <immintrin.h>  // AVX intrinsics

#include "hpc_helpers.hpp"

void plain_tmm(float * A, float * Bt, float * C, uint64_t M,uint64_t L, uint64_t N

    #pragma omp parallel for collapse(2)
    for (uint64_t i = 0; i < M; i++)
        for (uint64_t j = 0; j < N; j++) {
            float accum = float(0);
            for (uint64_t k = 0; k < L; k++)
                accum += A[i*L+k]*Bt[j*L+k];
            C[i*N+j] = accum;
       }
}

void plain(float * A, float * B, float * C, uint64_t M,uint64_t L, uint64_t N) {

    #pragma omp parallel for collapse(2)
    for (uint64_t i = 0; i < M; i++)
        for (uint64_t j = 0; j < N; j++) {
            float accum = float(0);
            for (uint64_t k = 0; k < L; k++)
                accum += A[i*L+k]*B[j*L+k];
            C[i*N+j] = accum;
       }
}

inline float hsum_sse3(__m128 v) {
    __m128 shuf = _mm_movehdup_ps(v);        // broadcast elements 3,1 to 2,0
    __m128 maxs = _mm_add_ps(v, shuf);
    shuf        = _mm_movehl_ps(shuf, maxs); // high half -> low half
    maxs        = _mm_add_ss(maxs, shuf);
    return        _mm_cvtss_f32(maxs);
}

inline float hsum_avx(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);   // low 128
    __m128 hi = _mm256_extractf128_ps(v, 1); // high 128
           lo = _mm_add_ps(lo, hi);          // max the low 128
    return hsum_sse3(lo);                    // and inline the sse3 version
}

void avx2_tmm(float * A,float * Bt,float * C,uint64_t M,uint64_t L,uint64_t N) {

    #pragma omp parallel for collapse(2)
    for (uint64_t i = 0; i < M; i++)
        for (uint64_t j = 0; j < N; 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(Bt+j*L+k);
```

```
                X = _mm256_add_ps(X, _mm256_mul_ps(AV, BV));
            }
            C[i*N+j] = hsum_avx(X);
        }
    }

    void avx2(float * A,float * B,float * C,uint64_t M,uint64_t L,uint64_t N) {



        #pragma omp parallel for collapse(2)
        for (uint64_t i = 0; i < M; i++)
            for (uint64_t j = 0; j < N; 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_add_ps(X, _mm256_mul_ps(AV, BV));
                }
                C[i*N+j] = hsum_avx(X);
            }
    }

    int main() {

        const uint64_t M = 1UL <<  10;
        const uint64_t L = 1UL <<  11;
        const uint64_t N = 1UL <<  12;

        printf("%" PRId64 "\n", M);

        TIMERSTART(alloc_memory)
        auto A = static_cast<float*>(_mm_malloc(M*L*sizeof(float), 32));
        auto B = static_cast<float*>(_mm_malloc(L*N*sizeof(float), 32));
        auto C = static_cast<float*>(_mm_malloc(M*N*sizeof(float), 32));
        auto Bt = static_cast<float*>(_mm_malloc(N*L*sizeof(float), 32));
        TIMERSTOP(alloc_memory)

        // Init matrix
        TIMERSTART(init)
        for (uint64_t i = 0; i < M*L; i++)
            A[i] = rand() % 100;   // 0 - 99 numbers
            //A[i] = 3.0;

        for (uint64_t i = 0; i < N*L; i++)
            B[i] = rand() % 100;   // 0 - 99 numbers
            //B[i] = 3.0;
        TIMERSTOP(init)

        TIMERSTART(transpose)
        #pragma omp parallel for collapse(2)
        for (uint64_t i = 0; i < L; i++)
            for (uint64_t j = 0; j < N; j++)
                Bt[j*L+i] = B[i*N+j];
        TIMERSTOP(transpose)
```

```
    TIMERSTART(plain)
    plain(A, B, C, M, L, N); // C = A * B
    TIMERSTOP(plain)

    TIMERSTART(plain_tmm)
    plain_tmm(A, Bt, C, M, L, N); // C = A * B Transpose
    TIMERSTOP(plain_tmm)


    TIMERSTART(avx2)
    avx2_tmm(A, B, C, M, L, N); // C = A * B AVX
    TIMERSTOP(avx2)

    TIMERSTART(avx2_tmm)
    avx2_tmm(A, Bt, C, M, L, N); // C = A * Bt AVX
    TIMERSTOP(avx2_tmm)

    // dahin huwaasrilah
    TIMERSTART(free_memory)
    _mm_free(A);
    _mm_free(B);
    _mm_free(C);
    _mm_free(Bt);
    TIMERSTOP(free_memory)
}
```

```
 Overwriting lab3_1AVX.cpp
```

```
%%script bash
g++ -mavx2 -std=c++17 lab3_1AVX.cpp -o out
./out
```

```
 1024
 # elapsed time (alloc_memory): 0.000104157s
 # elapsed time (init): 0.137947s
 # elapsed time (transpose): 0.27989s
 # elapsed time (plain): 30.4553s
 # elapsed time (plain_tmm): 30.2639s
 # elapsed time (avx2): 8.82198s
 # elapsed time (avx2_tmm): 10.1595s
 # elapsed time (free_memory): 0.00596352s
```

Дүгнэлт : Лекц 5-дээр үзсэн AVX2 ийн Transpose-and-Multiply алгоритм ийг хэрэгжүүлж, хэр их хугацаа өнгөрсөнг хэвлэж гаргав.

plain : энгийн 2 матриц үржүүлэх

plain_tmm : матриц эргүүлж үржүүлсэн

avx2 : avx2 регистр ажиглаж үржүүлсэн

avx2_tmm : avx2 transpose хиж үржүүлсэн

Доорх зурганд өөрийн компьютер дээр ажилуулж харьцуулж үзэв 512*512 харьцаатай матриц үржүүлэх дээр хугацааны ялгаа харагдахгүй байгаа ч матриц ийн уртаа ихсгэх тусам AVX2 регистр ашигласан нь илүү хурдаг харагдаж байна.

CPU **i3-2330M 2core 4threads**

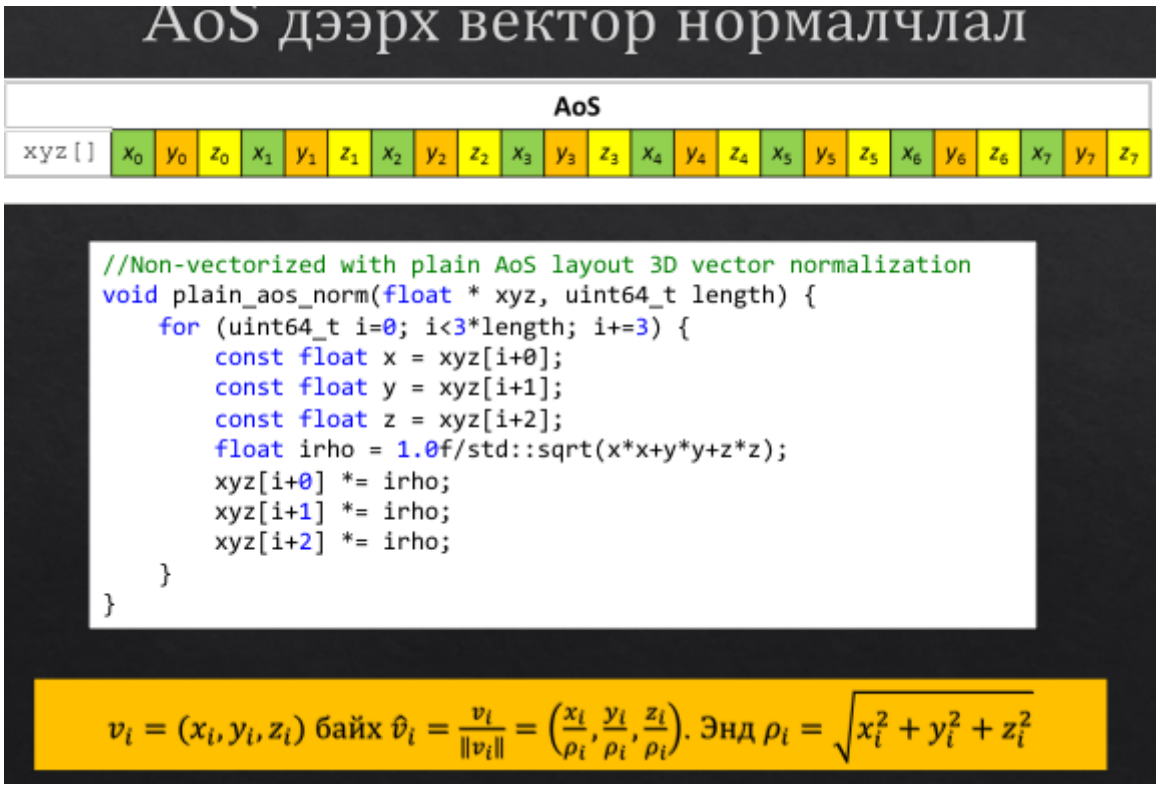
```
512
# elapsed time (alloc_memory): 5.266e-05s
# elapsed time (init): 0.0137612s
# elapsed time (transpose): 0.00452589s
# elapsed time (plain): 0.672817s
# elapsed time (plain_tmm): 0.644178s
# elapsed time (avx2): 0.295497s
# elapsed time (avx2_tmm): 0.268335s
# elapsed time (free_memory): 0.000386406s
```

```
1024
# elapsed time (alloc_memory): 6.7851e-05s
# elapsed time (init): 0.0422046s
# elapsed time (transpose): 0.0247373s
# elapsed time (plain): 5.48877s
# elapsed time (plain_tmm): 5.43769s
# elapsed time (avx2): 2.27716s
# elapsed time (avx2_tmm): 2.28212s
# elapsed time (free_memory): 0.00129521s
```

```
2048
# elapsed time (alloc_memory): 4.9538e-05s
# elapsed time (init): 0.187601s
# elapsed time (transpose): 0.0988899s
# elapsed time (plain): 43.3761s
# elapsed time (plain_tmm): 42.9776s
# elapsed time (avx2): 17.5426s
# elapsed time (avx2_tmm): 17.5776s
# elapsed time (free_memory): 0.00492091s
```

```
4096
# elapsed time (alloc_memory): 5.4949e-05s
# elapsed time (init): 0.635694s
# elapsed time (transpose): 0.404278s
# elapsed time (plain): 444.993s
# elapsed time (plain_tmm): 476.417s
# elapsed time (avx2): 187.772s
# elapsed time (avx2_tmm): 156.09s
# elapsed time (free_memory): 0.0179526s
```

2. AoS дээрх Vectorized нормалчлал



Simple Code

```
#include <cstdint>      // uint32_t
#include <iostream>     // std::cout
#include <random>       // prng
#include <inttypes.h>

// timers distributed with this book
#include "../../include/hpc_helpers.hpp"

void aos_init(float * xyz, uint64_t length) {

    std::mt19937 engine(42);
    std::uniform_real_distribution<float> density(-1, 1);

    for (uint64_t i = 0; i < 3*length; i++)
        xyz[i] = density(engine);
}

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;
    }
}

void aos_check(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 rho = x*x+y*y+z*z;

        if ((rho-1)*(rho-1) > 1E-6)
            std::cout << "error too big at position "
                      << i << std::endl;
    }
}

int main () {

    const uint64_t num_vectors = 1UL << 18;

    printf("%" PRId64 "\n", num_vectors);
```

```
    TIMERSTART(alloc_memory)
    auto xyz = new float[3*num_vectors];
    TIMERSTOP(alloc_memory)

    TIMERSTART(init)
    aos_init(xyz, num_vectors);
    TIMERSTOP(init)

    TIMERSTART(plain_aos_normalize)
    plain_aos_norm(xyz, num_vectors);
    TIMERSTOP(plain_aos_normalize)

    TIMERSTART(check)
    aos_check(xyz, num_vectors);
    TIMERSTOP(check)

    TIMERSTART(free_memory)
    delete [] xyz;
    TIMERSTOP(free_memory)
}
```

AVX Code

```
#include <random>       // prng
#include <cstdint>      // uint32_t
#include <iostream>     // std::cout
#include <immintrin.h>  // AVX intrinsics
#include <inttypes.h>

// timers distributed with this book
#include "../../include/hpc_helpers.hpp"

void aos_init(float * xyz, uint64_t length) {

    std::mt19937 engine(42);
    std::uniform_real_distribution<float> density(-1, 1);

    for (uint64_t i = 0; i < 3*length; i++)
        xyz[i] = density(engine);
}

void avx_aos_norm(float * xyz, uint64_t length) {

    for (uint64_t i = 0; i < 3*length; i += 3*8) {

        ////////////////////////////////////////////////////////////////
        // AOS2SOA: XYZXYZXY ZXYZXYZX YZXYZXYZ --> XXXXXXX YYYYYYY ZZZZZZZZ
        ////////////////////////////////////////////////////////////////

        // registers: NOTE: M is an SSE pointer (length 4)
        __m128 *M = (__m128*) (xyz+i);
        __m256 M03;
        __m256 M14;
        __m256 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 am shuffeling...
        __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));

        ////////////////////////////////////////////////////////////////////
        // SOA computation
        ////////////////////////////////////////////////////////////////////

        // R <- X*X+Y*Y+Z*Z
        __m256 R = _mm256_add_ps(_mm256_mul_ps(X, X),
                   _mm256_add_ps(_mm256_mul_ps(Y, Y),
                                 _mm256_mul_ps(Z, Z)));
        // R <- 1/sqrt(R)
               R = _mm256_rsqrt_ps(R);

        // normalize vectors
        X = _mm256_mul_ps(X, R);
        Y = _mm256_mul_ps(Y, R);
        Z = _mm256_mul_ps(Z, R);

        ////////////////////////////////////////////////////////////////////
        // SOA2AOS: XXXXXXX YYYYYYY ZZZZZZZZ -> XYZXYZXY ZXYZXYZX YZXYZXYZ
        ////////////////////////////////////////////////////////////////////

        // everyday I am shuffeling...
        __m256 RXY = _mm256_shuffle_ps(X,Y, _MM_SHUFFLE(2,0,2,0));
        __m256 RYZ = _mm256_shuffle_ps(Y,Z, _MM_SHUFFLE(3,1,3,1));
        __m256 RZX = _mm256_shuffle_ps(Z,X, _MM_SHUFFLE(3,1,2,0));
        __m256 R03 = _mm256_shuffle_ps(RXY, RZX, _MM_SHUFFLE(2,0,2,0));
        __m256 R14 = _mm256_shuffle_ps(RYZ, RXY, _MM_SHUFFLE(3,1,2,0));
        __m256 R25 = _mm256_shuffle_ps(RZX, RYZ, _MM_SHUFFLE(3,1,3,1));

        // store in AOS (6*4=24)
        M[0] = _mm256_castps256_ps128(R03);
        M[1] = _mm256_castps256_ps128(R14);
        M[2] = _mm256_castps256_ps128(R25);
        M[3] = _mm256_extractf128_ps(R03, 1);
        M[4] = _mm256_extractf128_ps(R14, 1);
        M[5] = _mm256_extractf128_ps(R25, 1);
    }
}
```

```
void aos_check(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 rho = x*x+y*y+z*z;

        if ((rho-1)*(rho-1) > 1E-6)
            std::cout << "error too big at position "
                      << i << std::endl;
    }
}
int main () {

    const uint64_t num_vectors = 1UL << 28;
    const uint64_t num_bytes = 3*num_vectors*sizeof(float);

    printf("%" PRId64 "\n", num_vectors);

    TIMERSTART(alloc_memory)
    auto xyz = static_cast<float*>(_mm_malloc(num_bytes , 32));
    TIMERSTOP(alloc_memory)

    TIMERSTART(init)
    aos_init(xyz, num_vectors);
    TIMERSTOP(init)

    //AoS(Array of structure ): утгуудыг солбилцуулан хадгалсан нэг массив.

    TIMERSTART(avx_aos_normalize)
    avx_aos_norm(xyz, num_vectors);
    TIMERSTOP(avx_aos_normalize)

    TIMERSTART(check)
    aos_check(xyz, num_vectors);
    TIMERSTOP(check)

    TIMERSTART(free_memory)
    _mm_free(xyz);
    TIMERSTOP(free_memory)
}
```

Дүгнэлт : Лекц 6-дээр үзсэн AoS дээрх Vectorized нормалчлал гэсэн параллел алгоритмыг хэрэгжүүлэв.

AOS форматад 3D векторыг 256 бит регистр SoA формат руу шилжүүлнэ. SoA форматыг ашиглан Vectorized SIMD тооцооллоно. SoA ээс үр дүнг AoS формат руу шилжүүлнэ. Векторыг холих үйлдэл ашиглана.

Доорх зурганд AVX2 register - ээр хийсэн хийгэггүйг харьцуулж үзэв. **262144 * 3** ийн урртай үүсгэнэ.

CPU i3-2330M 2core 4threads

```
262144
# elapsed time (alloc_memory): 2.1589e-05s
# elapsed time (init): 0.0665853s
# elapsed time (plain_aos_normalize): 0.00545705s
# elapsed time (check): 0.00255743s
# elapsed time (free_memory): 0.00027949s
```

```
262144
# elapsed time (alloc_memory): 2.1754e-05s
# elapsed time (init): 0.0722883s
# elapsed time (avx_aos_normalize): 0.00339975s
# elapsed time (check): 0.00452701s
# elapsed time (free_memory): 0.00042766s
```

```
268435456
# elapsed time (alloc_memory): 2.5498e-05s
# elapsed time (init): 66.9237s
# elapsed time (plain_aos_normalize): 5.76117s
# elapsed time (check): 2.71748s
# elapsed time (free_memory): 0.243316s
```

```
268435456
# elapsed time (alloc_memory): 2.2938e-05s
# elapsed time (init): 69.9485s
# elapsed time (avx_aos_normalize): 3.27273s
# elapsed time (check): 3.11333s
# elapsed time (free_memory): 0.225991s
```

✓ 1m 21s completed at 1:59 AM