Lecture 8. SIMD and Vectorization

Introductions to Data Systems and Data Design

Ce Zhang

What is SIMD?

Flynn's Taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD (Uniprocessor)	MISD
Multiple Data	**SIMD** (Vector)	MIMD (Multicore)

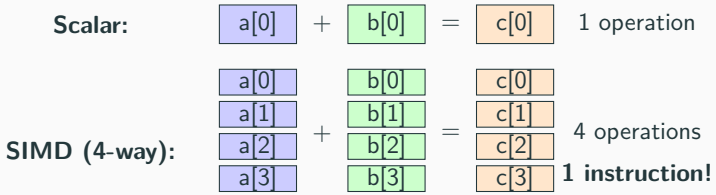
Flynn's Taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD (Uniprocessor)	MISD
Multiple Data	**SIMD** (Vector)	MIMD (Multicore)

SIMD = Single Instruction, Multiple Data

One instruction operates on multiple data elements simultaneously.

SIMD: The Big Picture



Why SIMD Matters

Theoretical speedup: 4-8x for vectorizable code

Extension	Register Width	Doubles	Floats
SSE/SSE2	128 bits	2-way	4-way
AVX/AVX2	256 bits	4-way	8-way
AVX-512	512 bits	8-way	16-way

Why SIMD Matters

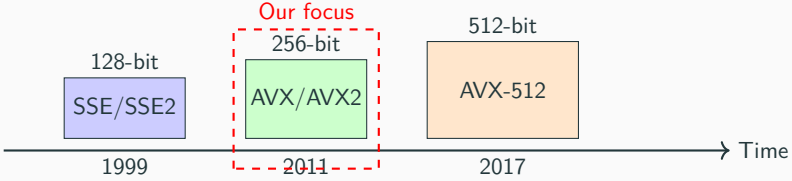
Theoretical speedup: 4-8x for vectorizable code

Extension	Register Width	Doubles	Floats
SSE/SSE2	128 bits	2-way	4-way
AVX/AVX2	256 bits	4-way	8-way
AVX-512	512 bits	8-way	16-way

SIMD + ILP together:

- ILP: Multiple *different* instructions in parallel
- SIMD: One instruction on *multiple data*
- Combine both for maximum throughput!

Evolution of x86 SIMD

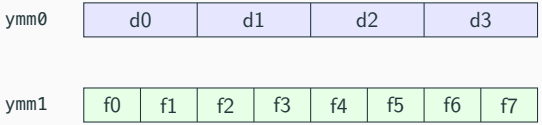


AVX2 is widely available (Sandy Bridge 2011+). We'll focus on it.

AVX Registers and Data Types

AVX Vector Registers

16 registers: `ymm0` - `ymm15`, each 256 bits wide



Same register, different interpretations!

Data Types in C/C++

```
#include <immintrin.h>  // AVX intrinsics header

__m256d  d;  // 4 doubles (256 bits)
__m256   f;  // 8 floats (256 bits)
__m256i  i;  // 32 bytes / 16 shorts / 8 ints / 4 longs
```

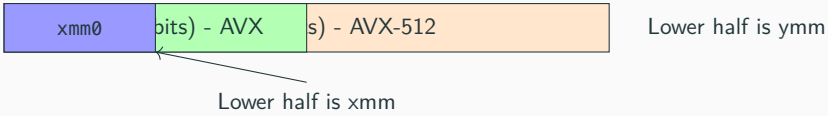
Data Types in C/C++

```
#include <immintrin.h>  // AVX intrinsics header

__m256d  d;  // 4 doubles (256 bits)
__m256   f;  // 8 floats (256 bits)
__m256i  i;  // 32 bytes / 16 shorts / 8 ints / 4 longs
```

Naming convention:

- `__m256` = 256-bit register
- `d` suffix = double precision
- `i` suffix = integer
- No suffix = single precision (float)

Register Hierarchy



`xmm0` is the lower 128 bits of `ymm0`, which is the lower 256 bits of `zmm0`.

AVX Intrinsics

What Are Intrinsics?

Intrinsics = C functions that map directly to assembly instructions

```
// C code with intrinsics
__m256d a = _mm256_load_pd(ptr);    // Load 4 doubles
__m256d b = _mm256_load_pd(ptr2);
__m256d c = _mm256_add_pd(a, b);    // Add 4 doubles
_mm256_store_pd(result, c);          // Store 4 doubles
```

```
; Generated assembly
vmovapd ymm0, [rdi]      ; Load aligned
vmovapd ymm1, [rsi]
vaddpd  ymm0, ymm0, ymm1 ; Packed double add
vmovapd [rdx], ymm0      ; Store aligned
```

Like writing assembly, but in C!

Intrinsic Naming Convention

`_mm256_<operation>_<suffix>`

Part	Meaning
`_mm256`	256-bit AVX operation
`_mm`	128-bit SSE operation
`_mm512`	512-bit AVX-512 operation

Suffix	Data Type
`pd`	Packed double (4 doubles)
`ps`	Packed single (8 floats)
`epi32`	Packed 32-bit integers
`si256`	256-bit integer

Load and Store

```
// Aligned load/store (pointer must be 32-byte aligned!)
__m256d a = _mm256_load_pd(ptr);     // Load 4 doubles
_mm256_store_pd(ptr, a);              // Store 4 doubles

// Unaligned load/store (any pointer)
__m256d b = _mm256_loadu_pd(ptr);    // Slower on some CPUs
_mm256_storeu_pd(ptr, b);
```

Load and Store

```
// Aligned load/store (pointer must be 32-byte aligned!)
__m256d a = _mm256_load_pd(ptr);     // Load 4 doubles
_mm256_store_pd(ptr, a);              // Store 4 doubles

// Unaligned load/store (any pointer)
__m256d b = _mm256_loadu_pd(ptr);    // Slower on some CPUs
_mm256_storeu_pd(ptr, b);
```

Alignment matters!

- Aligned: pointer address divisible by 32
- Unaligned load on aligned boundary: seg fault!
- Modern CPUs: unaligned is nearly as fast (but prefer aligned)

Setting Constants

```
// Set all elements to the same value
__m256d ones = _mm256_set1_pd(1.0);
// Result: [1.0, 1.0, 1.0, 1.0]

// Set each element individually (note: reverse order!)
__m256d v = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
// Result: [1.0, 2.0, 3.0, 4.0]  (LSB first!)

// Set to zero
__m256d zero = _mm256_setzero_pd();
// Result: [0.0, 0.0, 0.0, 0.0]
```

Setting Constants

```
// Set all elements to the same value
__m256d ones = _mm256_set1_pd(1.0);
// Result: [1.0, 1.0, 1.0, 1.0]

// Set each element individually (note: reverse order!)
__m256d v = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
// Result: [1.0, 2.0, 3.0, 4.0]  (LSB first!)

// Set to zero
__m256d zero = _mm256_setzero_pd();
// Result: [0.0, 0.0, 0.0, 0.0]
```

Warning: `_mm256_set_pd` takes arguments in **reverse order**!

Arithmetic Operations

```
__m256d c;

c = _mm256_add_pd(a, b);   // c[i] = a[i] + b[i]
c = _mm256_sub_pd(a, b);   // c[i] = a[i] - b[i]
c = _mm256_mul_pd(a, b);   // c[i] = a[i] * b[i]
c = _mm256_div_pd(a, b);   // c[i] = a[i] / b[i]

c = _mm256_sqrt_pd(a);     // c[i] = sqrt(a[i])
c = _mm256_max_pd(a, b);   // c[i] = max(a[i], b[i])
c = _mm256_min_pd(a, b);   // c[i] = min(a[i], b[i])

// Fused multiply-add (AVX2): c = a*b + c
c = _mm256_fmadd_pd(a, b, c);  // One instruction!
```

Performance: Latency and Throughput

Operation	Latency	Throughput
`add_pd`	4 cycles	2/cycle
`mul_pd`	4 cycles	2/cycle
`div_pd`	13-14 cycles	1/8 cycle
`sqrt_pd`	18 cycles	1/12 cycle
`fmadd_pd`	4 cycles	2/cycle
`load_pd`	7 cycles	2/cycle

Performance: Latency and Throughput

Operation	Latency	Throughput
`add_pd`	4 cycles	2/cycle
`mul_pd`	4 cycles	2/cycle
`div_pd`	13-14 cycles	1/8 cycle
`sqrt_pd`	18 cycles	1/12 cycle
`fmadd_pd`	4 cycles	2/cycle
`load_pd`	7 cycles	2/cycle

FMA is crucial: `a*b + c` in one instruction with same latency as add alone!

Example: Vector Addition

Scalar Version

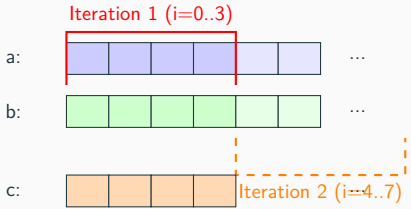
```
// add_scalar.cpp
void add_scalar(double* a, double* b, double* c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}

$ g++ -O1 add_scalar.cpp -o add_scalar && ./add_scalar
Time: 85 ms (100M elements)
```

SIMD Version

```
// add_simd.cpp
#include <immintrin.h>

void add_simd(double* a, double* b, double* c, int n) {
    for (int i = 0; i < n; i += 4) {
        __m256d va = _mm256_loadu_pd(a + i);
        __m256d vb = _mm256_loadu_pd(b + i);
        __m256d vc = _mm256_add_pd(va, vb);
        _mm256_storeu_pd(c + i, vc);
    }
}
```

```
$ g++ -O1 -mavx2 add_simd.cpp -o add_simd && ./add_simd
Time: 42 ms (100M elements)
```

Visualizing the SIMD Loop



Each loop iteration processes **4 elements** with one SIMD instruction.

Example: Sum Reduction (Revisited)

The Challenge

From ILP lecture: reductions have dependency chains.

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];  // Each add depends on previous!
}
```

SIMD alone doesn't help — we still have a single accumulator.

The Challenge

From ILP lecture: reductions have dependency chains.

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];  // Each add depends on previous!
}
```

SIMD alone doesn't help — we still have a single accumulator.

Solution: SIMD + multiple accumulators!

SIMD Sum: Step 1 - Vector Accumulator

```
__m256d vsum = _mm256_setzero_pd();  // [0, 0, 0, 0]

for (int i = 0; i < n; i += 4) {
    __m256d v = _mm256_loadu_pd(a + i);
    vsum = _mm256_add_pd(vsum, v);   // 4 parallel sums!
}

// vsum = [sum0, sum1, sum2, sum3]
// Final: sum = sum0 + sum1 + sum2 + sum3
```

SIMD Sum: Step 1 - Vector Accumulator

```
__m256d vsum = _mm256_setzero_pd();  // [0, 0, 0, 0]

for (int i = 0; i < n; i += 4) {
    __m256d v = _mm256_loadu_pd(a + i);
    vsum = _mm256_add_pd(vsum, v);   // 4 parallel sums!
}

// vsum = [sum0, sum1, sum2, sum3]
// Final: sum = sum0 + sum1 + sum2 + sum3
```

Now we have **4 independent accumulator chains** — 4x ILP!

SIMD Sum: Step 2 - Horizontal Reduction

```
// Reduce [sum0, sum1, sum2, sum3] to a single value
// Method 1: Extract and add scalars
double result[4];
_mm256_storeu_pd(result, vsum);
double sum = result[0] + result[1] + result[2] + result[3];
```

SIMD Sum: Step 2 - Horizontal Reduction

```
// Reduce [sum0, sum1, sum2, sum3] to a single value
// Method 1: Extract and add scalars
double result[4];
_mm256_storeu_pd(result, vsum);
double sum = result[0] + result[1] + result[2] + result[3];

// Method 2: Use horizontal add (faster)
__m128d low  = _mm256_castpd256_pd128(vsum);      // [sum0, sum1]
__m128d high = _mm256_extractf128_pd(vsum, 1);    // [sum2, sum3]
__m128d sum128 = _mm_add_pd(low, high);           // [sum0+sum2, sum1+sum3]
sum128 = _mm_hadd_pd(sum128, sum128);             // [total, total]
double sum = _mm_cvtsd_f64(sum128);               // Extract scalar
```

Complete SIMD Sum

```
double sum_simd(double* a, int n) {
    __m256d vsum = _mm256_setzero_pd();

    for (int i = 0; i < n; i += 4) {
        __m256d v = _mm256_loadu_pd(a + i);
        vsum = _mm256_add_pd(vsum, v);
    }

    // Horizontal reduction
    __m128d low = _mm256_castpd256_pd128(vsum);
    __m128d high = _mm256_extractf128_pd(vsum, 1);
    __m128d sum128 = _mm_add_pd(low, high);
    sum128 = _mm_hadd_pd(sum128, sum128);
    return _mm_cvtsd_f64(sum128);
```

SIMD Sum: Even Better with Unrolling

```
double sum_simd_unrolled(double* a, int n) {
    __m256d vsum1 = _mm256_setzero_pd();
    __m256d vsum2 = _mm256_setzero_pd();  // Two vector accumulators!

    for (int i = 0; i < n; i += 8) {
        __m256d v1 = _mm256_loadu_pd(a + i);
        __m256d v2 = _mm256_loadu_pd(a + i + 4);
        vsum1 = _mm256_add_pd(vsum1, v1);
        vsum2 = _mm256_add_pd(vsum2, v2);
    }

    __m256d vsum = _mm256_add_pd(vsum1, vsum2);
    // ... horizontal reduction ...
}
```

Sum Performance Comparison

Version	Time	Speedup
Scalar naive	152 ms	1x
Scalar 4x unrolled	42 ms	3.6x
SIMD (1 accumulator)	42 ms	3.6x
SIMD + 2x unrolled	25 ms	6x

Sum Performance Comparison

Version	Time	Speedup
Scalar naive	152 ms	1x
Scalar 4x unrolled	42 ms	3.6x
SIMD (1 accumulator)	42 ms	3.6x
SIMD + 2x unrolled	25 ms	6x

Key insight: SIMD with one accumulator = scalar with 4 accumulators

For maximum performance: **SIMD + unrolling** together!

Example: Dot Product

Scalar Dot Product

```
double dot_scalar(double* a, double* b, int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

SIMD Dot Product with FMA

```
double dot_simd(double* a, double* b, int n) {
    __m256d vsum = _mm256_setzero_pd();

    for (int i = 0; i < n; i += 4) {
        __m256d va = _mm256_loadu_pd(a + i);
        __m256d vb = _mm256_loadu_pd(b + i);
        // FMA: vsum = va * vb + vsum (one instruction!)
        vsum = _mm256_fmadd_pd(va, vb, vsum);
    }

    // Horizontal reduction (same as sum)
    __m128d low = _mm256_castpd256_pd128(vsum);
    __m128d high = _mm256_extractf128_pd(vsum, 1);
    __m128d sum128 = _mm_add_pd(low, high);
```

Dot Product Performance

```
$ g++ -O1 -mavx2 -mfma dot.cpp -o dot && ./dot
Scalar: 168 ms
SIMD:    35 ms
```

4.8x speedup!

- FMA does multiply+add in one instruction
- 4 parallel multiply-adds per iteration
- Still need unrolling for maximum throughput

Handling Conditionals

The Problem with Branches

```
// Scalar version with branch
for (int i = 0; i < n; i++) {
    if (a[i] > 0.5)
        b[i] = a[i] + 1.0;
    else
        b[i] = a[i] - 1.0;
}
```

SIMD processes 4 elements at once — what if some need +1 and others need −1?

The Problem with Branches

```
// Scalar version with branch
for (int i = 0; i < n; i++) {
    if (a[i] > 0.5)
        b[i] = a[i] + 1.0;
    else
        b[i] = a[i] - 1.0;
}
```

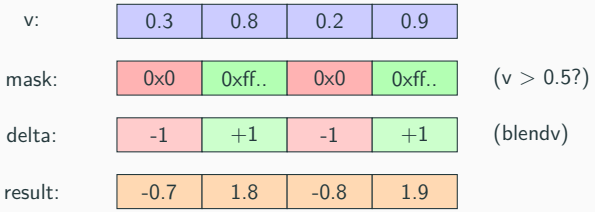
SIMD processes 4 elements at once — what if some need +1 and others need −1?

Solution: Compute **both** paths, use **mask** to select results.

SIMD Comparison and Masking

```
__m256d threshold = _mm256_set1_pd(0.5);
__m256d ones = _mm256_set1_pd(1.0);
__m256d mones = _mm256_set1_pd(-1.0);

for (int i = 0; i < n; i += 4) {
    __m256d v = _mm256_loadu_pd(a + i);

    // Compare: creates mask (all 1s or all 0s per element)
    __m256d mask = _mm256_cmp_pd(v, threshold, _CMP_GT_OQ);

    // Blend: select from ones or mones based on mask
    __m256d delta = _mm256_blendv_pd(mones, ones, mask);

    __m256d result = _mm256_add_pd(v, delta);
```

Visualizing Masking



Compiler Vectorization

Can the Compiler Do This Automatically?

```
void add(double* a, double* b, double* c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

```
$ g++ -O3 -march=native add.cpp -S
$ grep vaddpd add.s
    vaddpd  (%rsi,%rax), %ymm0, %ymm0   # Yes! Vectorized!
```

Can the Compiler Do This Automatically?

```
void add(double* a, double* b, double* c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

```
$ g++ -O3 -march=native add.cpp -S
$ grep vaddpd add.s
    vaddpd  (%rsi,%rax), %ymm0, %ymm0   # Yes! Vectorized!
```

Yes, often! With `-O3 -march=native`, compilers auto-vectorize simple loops.

When Auto-Vectorization Fails

1. **Aliasing** — pointers might overlap

```
void add(double* a, double* b, double* c, int n) {
    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];  // What if c overlaps a or b?
}
```

Fix: Use `restrict` keyword or `#pragma`

```
void add(double* restrict a, double* restrict b,
         double* restrict c, int n) { ... }
```

When Auto-Vectorization Fails (cont'd)

2. **Complex control flow**

```
while (a[i] > 0) {  // Not a countable loop!
    ...
}
```

3. **Function calls in loop**

4. **Floating-point associativity** (for reductions)

```
sum += a[i];  // Can't reorder without -ffast-math
```

Fix: Use `-ffast-math` or `-fassociative-math`

Helping the Compiler

```
// Tell compiler there's no aliasing
#pragma GCC ivdep
for (int i = 0; i < n; i++)
    c[i] = a[i] + b[i];

// Force vectorization (GCC)
#pragma GCC optimize("tree-vectorize")

// Check if compiler vectorized
$ g++ -O3 -fopt-info-vec-optimized file.cpp
file.cpp:5: optimized: loop vectorized using 32 byte vectors
```

When to Use Intrinsics vs Compiler

Use compiler vectorization when:

- Simple loops (add, multiply arrays)
- Performance is “good enough”
- Portability matters

Use intrinsics when:

- Complex algorithms (reductions, shuffles)
- Need guaranteed vectorization
- Maximum performance required
- Compiler fails to vectorize

Alignment

Why Alignment Matters

32-byte alignment = address divisible by 32

```
// Unaligned (any address)
double* a = (double*)malloc(n * sizeof(double));

// Aligned to 32 bytes
double* a = (double*)aligned_alloc(32, n * sizeof(double));
```

Why Alignment Matters

32-byte alignment = address divisible by 32

```
// Unaligned (any address)
double* a = (double*)malloc(n * sizeof(double));

// Aligned to 32 bytes
double* a = (double*)aligned_alloc(32, n * sizeof(double));
```

Performance impact:

- Modern CPUs: unaligned nearly as fast
- But: `_mm256_load_pd` on unaligned = **crash!**
- Aligned loads enable more optimizations

Allocating Aligned Memory

```
// C11 standard
double* a = aligned_alloc(32, n * sizeof(double));

// POSIX
double* a;
posix_memalign((void**)&a, 32, n * sizeof(double));

// C++ 17
double* a = static_cast<double*>(
    std::aligned_alloc(32, n * sizeof(double)));

// On stack
alignas(32) double a[1024];
```

Safe Loading Pattern

```
void process(double* a, int n) {
    int i = 0;

    // Handle unaligned prefix
    while (((uintptr_t)(a + i) & 31) != 0 && i < n) {
        // Scalar processing
        process_one(a[i]);
        i++;
    }

    // Main loop: aligned
    for (; i + 4 <= n; i += 4) {
        __m256d v = _mm256_load_pd(a + i);  // Safe!
        // ...
```

Practical Guidelines

SIMD Best Practices

1. **Use aligned memory** when possible (32-byte for AVX)
2. **Combine SIMD with unrolling** for maximum ILP
3. **Minimize shuffles** — they're expensive
4. **Process data in chunks of vector width** (4 doubles for AVX)
5. **Handle remainders** — what if `n % 4 != 0`?
6. **Benchmark!** — SIMD doesn't always help (memory-bound code)

When SIMD Helps Most

Good candidates:

- Element-wise operations (add, multiply arrays)
- Reductions (sum, max, dot product)
- Stencil computations
- Image processing
- Small matrix operations

Poor candidates:

- Already memory-bound code
- Highly irregular access patterns
- Lots of branches/conditionals
- Gather/scatter intensive

Complete Example: Array Operations

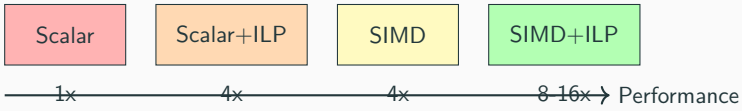
```
#include <immintrin.h>
#include <cstdlib>

void saxpy_simd(float* y, float a, float* x, int n) {
    __m256 va = _mm256_set1_ps(a);  // Broadcast a

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        vy = _mm256_fmadd_ps(va, vx, vy);  // y = a*x + y
        _mm256_storeu_ps(y + i, vy);
    }
```

Summary

Key Takeaways

1. **SIMD** = process 4-8 elements with one instruction
   - AVX: 4 doubles or 8 floats per instruction
2. **Intrinsics** give you direct control over SIMD
   - `_mm256_add_pd`, `_mm256_load_pd`, etc.
3. **SIMD + ILP** together for maximum performance
   - Use multiple vector accumulators
4. **Compiler can auto-vectorize** simple cases
   - But intrinsics give you more control
5. **Alignment** matters — use 32-byte aligned memory

Performance Hierarchy



Try It Yourself

```
# Compile with AVX support
$ g++ -O3 -mavx2 -mfma program.cpp -o program

# Check what instructions are generated
$ g++ -O3 -mavx2 -S program.cpp
$ grep -E "vadd|vmul|vfma" program.s

# See if compiler auto-vectorized
$ g++ -O3 -mavx2 -fopt-info-vec program.cpp

# Use perf to measure
$ perf stat ./program
```

Reference: Intel Intrinsics Guide