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)

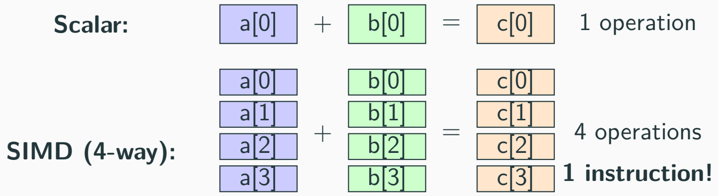
Flynn's Taxonomy

	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

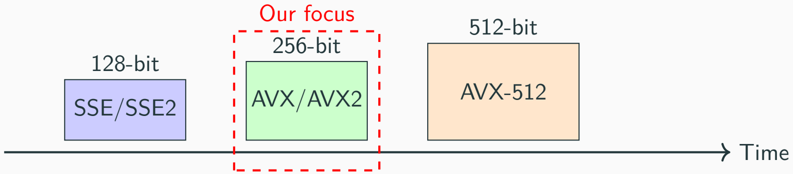
Theoretical speedup: 4-8x for vectorizable code

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

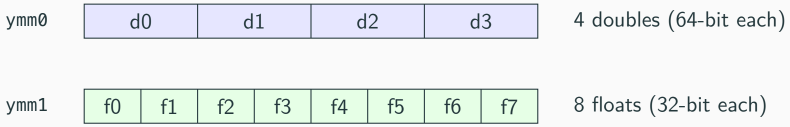


AVX2 is widely available (Sandy Bridge 2011+). We'll focus on it.

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
```

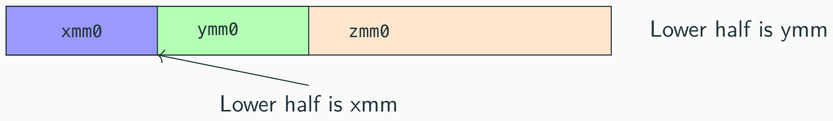
Data Types in C/C++

```
#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

```c
// 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: 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

```c
// 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

```
// sum_v1.cpp
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

```cpp
// sum_v2.cpp
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

```
$ cd examples && make sum
```

Version	Time
SIMD (1 accumulator)	104 ms
SIMD + 2x unrolled	52 ms

Sum Performance Comparison

```
$ cd examples && make sum
```

Version	Time
SIMD (1 accumulator)	104 ms
SIMD + 2x unrolled	52 ms

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

Example: Dot Product

Scalar Dot Product

```cpp
// dot_v1.cpp
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

```cpp
// dot_v2.cpp
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);
    sum128 = _mm_hadd_pd(sum128, sum128);
    return _mm_cvtsd_f64(sum128);
}
```

Dot Product Performance

```
$ cd examples && make dot
```

- 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

```c
// 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

```c
// 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

```
$ cd examples && make cond
```

```
// conditional.cpp
__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);
    _mm256_storeu_pd(b + i, result);
}
```

Visualizing Masking

v:	0.3	0.8	0.2	0.9	
mask:	0x0	0xff..	0x0	0xff..	($v > 0.5$?)
delta:	-1	+1	-1	+1	(blendv)
result:	-0.7	1.8	-0.8	1.9	

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 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

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

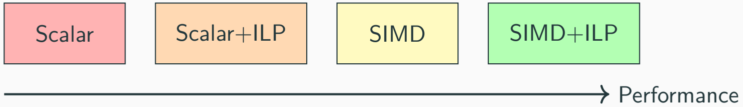
Good candidates:

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

Poor candidates:

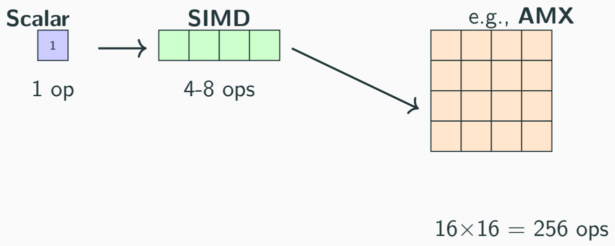
- Already memory-bound code
- Highly irregular access patterns
- Lots of branches/conditionals
- Gather/scatter intensive

Performance Hierarchy



Beyond SIMD: Scalar $\rightarrow$ Vector $\rightarrow$ Matrix

The Progression: Scalar → Vector → Matrix



Apple AMX: Matrix Coprocessor

AMX = Apple Matrix eXtensions (M1/M2/M3 chips, undocumated but it is there)

- Dedicated matrix coprocessor (not just wider SIMD)
- Computes **16×16 outer products** in one instruction
- Used by Accelerate framework, PyTorch, etc.

Apple AMX: Matrix Coprocessor

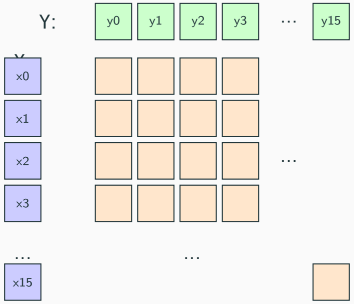
AMX = Apple Matrix eXtensions (M1/M2/M3 chips, undocumated but it is there)

- Dedicated matrix coprocessor (not just wider SIMD)
- Computes **16×16 outer products** in one instruction
- Used by Accelerate framework, PyTorch, etc.

Register layout:

Register	Size	Contents (fp32)
X[0-7]	64 bytes each	16 floats per register
Y[0-7]	64 bytes each	16 floats per register
Z	4 KB total	16×16 accumulator matrix

AMX Operation: Outer Product



```
Z[i][j] += X[i] * Y[j]
```

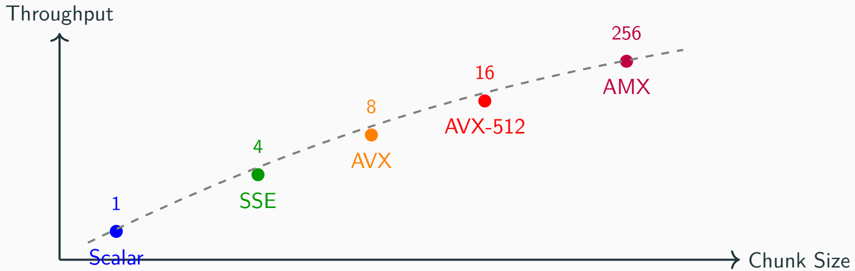
$16 \times 16 = 256$ FMAs

One instruction!

AMX in C++: A Glimpse

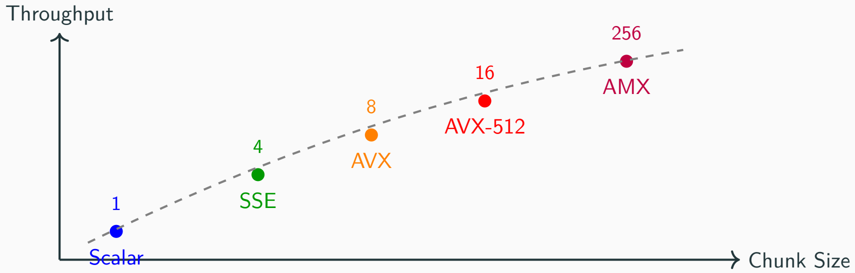
```
#include "amx.h"  // Apple's undocumented AMX "intrinsics"

// Compute 16x16 outer product: Z += X * Y^T
void amx_outer_product(float* X, float* Y, float* Z) {
    AMX_SET();  // Enable AMX coprocessor

    // Load 16 floats into X register 0
    AMX_LDX(ldx_operand(X, 0));

    // Load 16 floats into Y register 0
    AMX_LDY(ldy_operand(Y, 0));

    // FMA: Z[i][j] += X[i] * Y[j] for all i,j in [0,16)
    AMX_FMA32(fma32_operand(0, 0, 0));

    // Store result (64 bytes per row, 16 rows)
    for (int row = 0; row < 16; row++)
        AMX_STZ(stz_operand(Z + row * 16, row));

    AMX_CLR();  // Disable AMX
}
```

The Big Picture: Processing in Chunks



Trend: Increasing parallelism through larger “chunks” of computation

The Big Picture: Processing in Chunks



Trend: Increasing parallelism through larger “chunks” of computation

On GPUs: This becomes **Tensor Cores** (NVIDIA)

```
// PTX instruction: 16×8×16 matrix multiply = 4096 FLOPs!
mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16
    {d0,d1,d2,d3}, {a0,a1,a2,a3}, {b0,b1}, {c0,c1,c2,c3};
```

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

Try It Yourself

```
# Run all examples
$ cd examples && make run

# Build with -O3 for comparison
$ cd examples && make fast && make run

# Use perf to measure
$ cd examples && make perf
```

Reference: Intel Intrinsics Guide