Introduction to SIMD

Fundamental High Performance Computing Cluster Practice

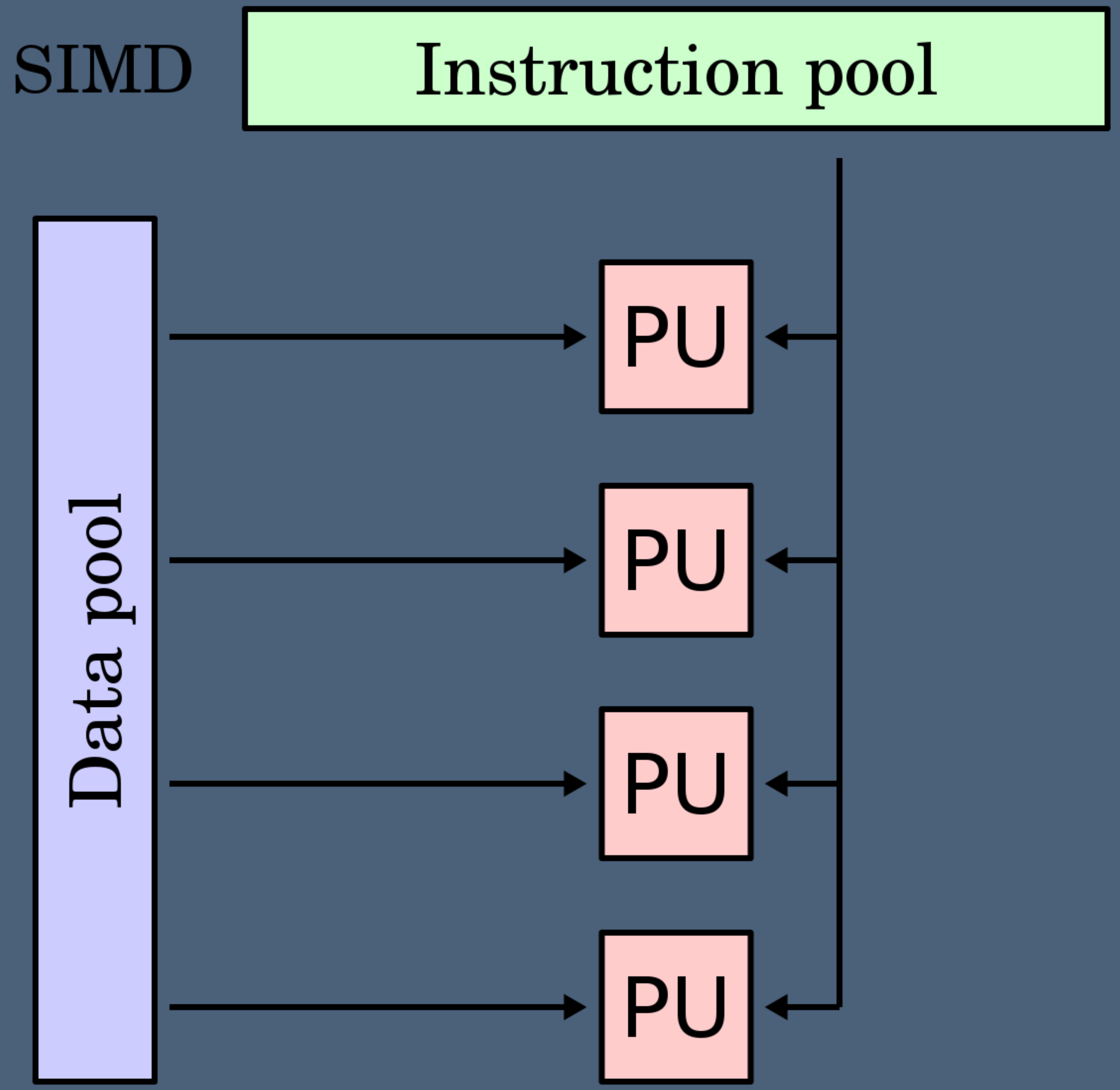
nevikw39

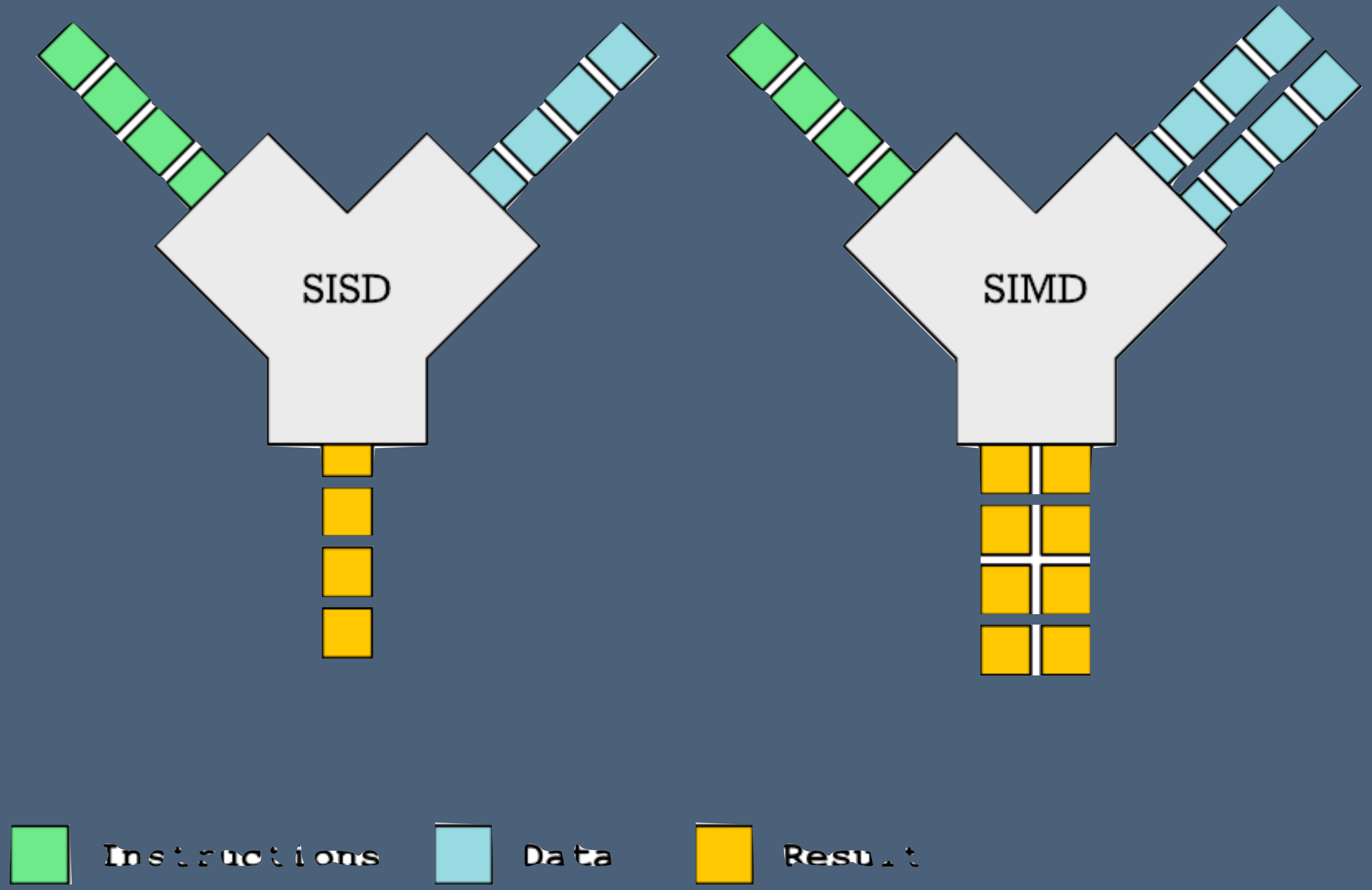
Background Ideas

- In most compute-intensive scenario, we often need apply the same operations to a series of data
- Nowadays, machine learning algorithms usually transform inputs to vectors
  - word2vec
  - wav2vec 2.0
  - Or even Life2Vec

SIMD

Single Instruction Multiple Data





Flynn's Taxonomy

Michael, 1966

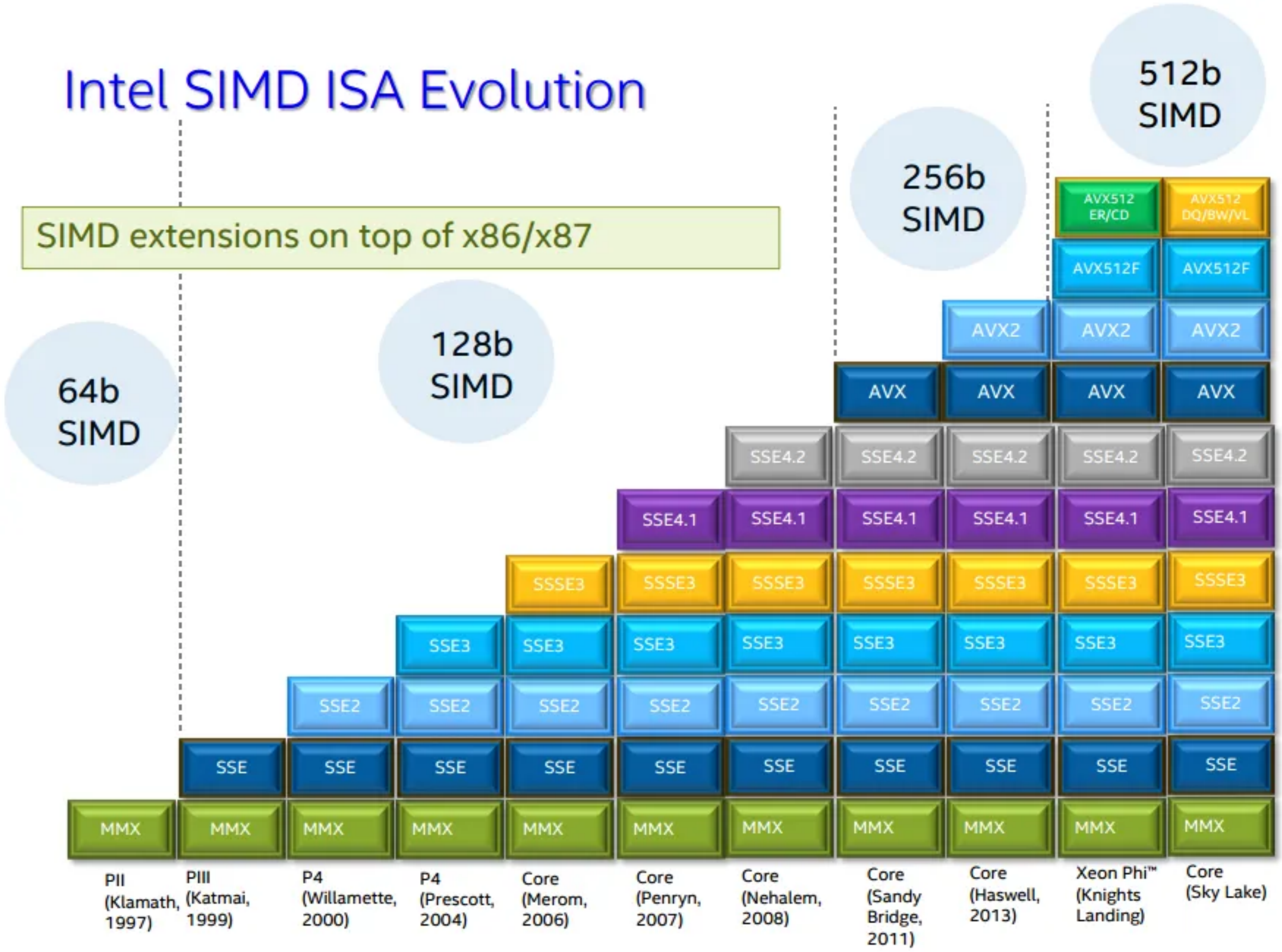
	Single Data	**Multiple Data**
Single Instruction	SISD e.g., scalar CPU	SIMD e.g., vector processor, SIMD instructions, GPU
Multiple Instruction	MISD e.g., systolic array (Google's TPU)	MIMD e.g., multi-core CPU

Characteristics of Different SIMD Solutions

- Vector processor
  - Variable-length data
  - Ad-hoc application
  - Cray
- SIMD Instruction
  - Fix-length data
  - Generic application
- GPU
  - Multiple threads (SIMT, Single Instruction Multiple Thread)

Intel Intrinsics History

Intel SIMD ISA Evolution



MMX

Multi-Media Extension, 1996

- Pentium (32-bit)
- 64-bit width
- Integer-type operations only
- Reuse mantissa part of extended-precision floating-point in x87 FPU as registers

SSE & SSE 2, 3, 4.1, 4.2

Streaming SIMD Extension, 1999

- Pentium 3 (32-bit)
- 128-bit width
- Support single-precision floating-point (integer and double since SSE 2, 2001)
- Introduce 8 new dedicated XMM registers (16 registers in 64-bit processor)
- In fact, nowadays floating-point math adopts SSE

AVX & AVX 2

Advanced Vector Extension, 2011 & 2013

- Sandy Bridge (64-bit)
- 256-bit width
- Support integer, single- and double-precision floating-point
- Introduce 16 YMM registers, lower halves of which are treated as the original XMM ones

AVX 512

Advanced Vector Extension, 2016

- Skylake (64-bit)
- 512-bit width
- Support integer, single- and double-precision floating-point
- Introduce 32 ZMM registers, lower halves of which are treated as the original YMM ones

Criticisms & Issues

“Die a Painful Death?”

- Performance & clock rate, power consumption, die space
- Complex variants
- Not all instructions for 128-bit / 256-bit have corresponding ones for 256-bit / 512-bit

SIMD Programming

Fine-grained, Low-level

Functions that provide access to other instructions without writing assembly code.

What are Intrinsics?

Example

Population Count

- POPCNT, the CPUID flag / instruction came at the same time with SSE 4.2
- With GCC, one could use its **`__builtin_popcount()`** and **`__builtin_popcountl()`**
- The corresponding Intel Intrinsics are **`_popcnt32()`** and **`_popcnt64()`**

Check What CPU Flags are Supported

- Linux
  - `lscpu`
  - `cat /proc/cpuinfo`
- macOS
  - `sysctl -a machdep.cpu.features`

Compiler Flags

- GCC
  - `-m<target>`
- ICC
  - `-x<code>`

Header Files

```
#include <mmintrin.h>  // MMX
#include <xmmintrin.h> // SSE
#include <emmintrin.h> // SSE2
#include <pmmintrin.h> // SSE3
#include <tmmintrin.h> // SSSE3
#include <smmintrin.h> // SSE4.1
#include <nmmintrin.h> // SSE4.2
#include <immintrin.h> // AVX, AVX2, FMA
```

Data Types

- 128-bit
    - `__m128`
    - `__m128d`
    - `__m128i`
- 256-bit
    - `__m256`
    - `__m256d`
    - `__m256i`
- 512-bit
    - `__m512`
    - `__m512d`
    - `__m512i`

Intrinsics Pattern

`_mm<width>_<operation>_<type>()`

- Width
  - 512, 256, or empty for 128
- Type
  - **`ps`** for packed single, **`pd`** for packed double; **`epi8`**, **`epi32`** and **`epi64`** for packed integer types operations, respectively
  - **`si512`**, **`si256`** and **`si128`** for operations on the whole integer register

Operation

Load / Store

- `load`, `store`
  - Load / store data between memory and register. Note that memory must be aligned (XMM requires 16-byte, YMM 32-byte, ZMM 64-byte)
- `loadu`, `storeu`
  - Load / store data between memory and register. Memory could be un-aligned.

Operation

Set

- **`setzero`**
  - Reset all bits in the register
- **`set1`**
  - Broadcast a value to all elements in the register
- **`set`**
  - Set values from lower to higher
- **`setr`**
  - Set values in reverse order

Operation

Extract

- **`extract`**
  - Get the element in the register

Operation

Arithmetic - Addition & Subtraction

- **`add`**, **`sub`**
  - Leave overflow as-is
- **`adds`**, **`subs`**
  - Integer addition and subtraction with *“saturation”*. Overflow results would be bound to maximum or minimum

Operation

Arithmetic - Multiplication & Division

- `mul`
  - Note that for integers, only even-indexed elements would be extended and multiplied
- `mullo`, `mulhi`
  - For integer multiplication, the results would be the lower / higher halves
- `div`
  - Only for floating-point numbers

Operation

Arithmetic - Bitwise

- **`and`**, **`or`**, **`xor`**
- **`slli`**, **`sll`**, **`sllv`**
  - Shift left logically
- **`srli`**, **`srl`**, **`srlv`**
  - Shift right logically
- **`srai`**, **`sra`**, **`srav`**
  - Shift right arithmetically

Operation

Arithmetic - Comparison

- **`cmpeq`**, **`cmpge`**, **`cmpgt`**, **`cmple`**, **`cmplt`**
  - If the condition is true, all bits of the element in result would be set

Operation

Arithmetic - Math

- `min`, `max`, `avg`, `abs`
  - Average only available for unsigned integers (ceiling up)
- `sqrt`, `rcp`, `rsqrt`
  - Reciprocal ones are approximate

Operation

Shuffle & Permute

- **`shuffle`**
  - Select from 2 registers
- **`permute`**
  - Select from a register
- **`blend`**, **`blendv`**
  - Blend 2 registers

Intel Intrinsics Guide

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

- Latency
  - The delay generated by the dependency-chain
- Throughput
  - Note that it is in CPI (cycle per instruction), which is reciprocal to common ways, IPC (instruction per cycle)

Case Study

https://github.com/nevikw39/HPC-Vec

Compute the Angle between 2 Vectors

- We have that $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos \theta$ and $|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$
- Thus we only need implement the dot product operation

Scalar Version

```
float dot(const float *const lhs, const float *const rhs)
{
    float y = 0;
    for (int i = 0; i < N; i++)
        y += i[lhs] * i[rhs];
    return y;
}
```

Vectorization with SSE

Naïve Approach

```
float dot_naive(const float *const lhs, const float *const rhs)
{
    assert(sizeof(float) == 4);
    float y = 0;
    for (int i = 0; i < N; i += 4)
    {
        __m128 li = _mm_load_ps(lhs + i), ri = _mm_load_ps(rhs + i);
        __m128 tmp = _mm_mul_ps(li, ri);
        _Alignas(16) static float arr[4];
        _mm_store_ps(arr, tmp);
        y += *arr + 1[arr] + 2[arr] + 3[arr];
    }
    return y;
}
```

Vectorization with SSE

Better Approach

```
float dot(const float *const lhs, const float *const rhs)
{
    assert(sizeof(float) == 4);
    __m128 y = _mm_setzero_ps();
    for (int i = 0; i < N; i += 4)
    {
        __m128 li = _mm_load_ps(lhs + i), ri = _mm_load_ps(rhs + i);
        __m128 tmp = _mm_mul_ps(li, ri);
        y = _mm_add_ps(y, tmp);
    }
    return y[0] + y[1] + y[2] + y[3];
}
```

Vectorization with AVX

```
float dot(const float *const lhs, const float *const rhs)
{
    assert(sizeof(float) == 4);
    __m256 y = _mm256_setzero_ps();
    for (int i = 0; i < N; i += 8)
    {
        __m256 li = _mm256_load_ps(lhs + i), ri = _mm256_load_ps(rhs +
i);
        __m256 tmp = _mm256_mul_ps(li, ri);
        y = _mm256_add_ps(y, tmp);
    }
    return y[0] + y[1] + y[2] + y[3] + y[4] + y[5] + y[6] + y[7];
}
```

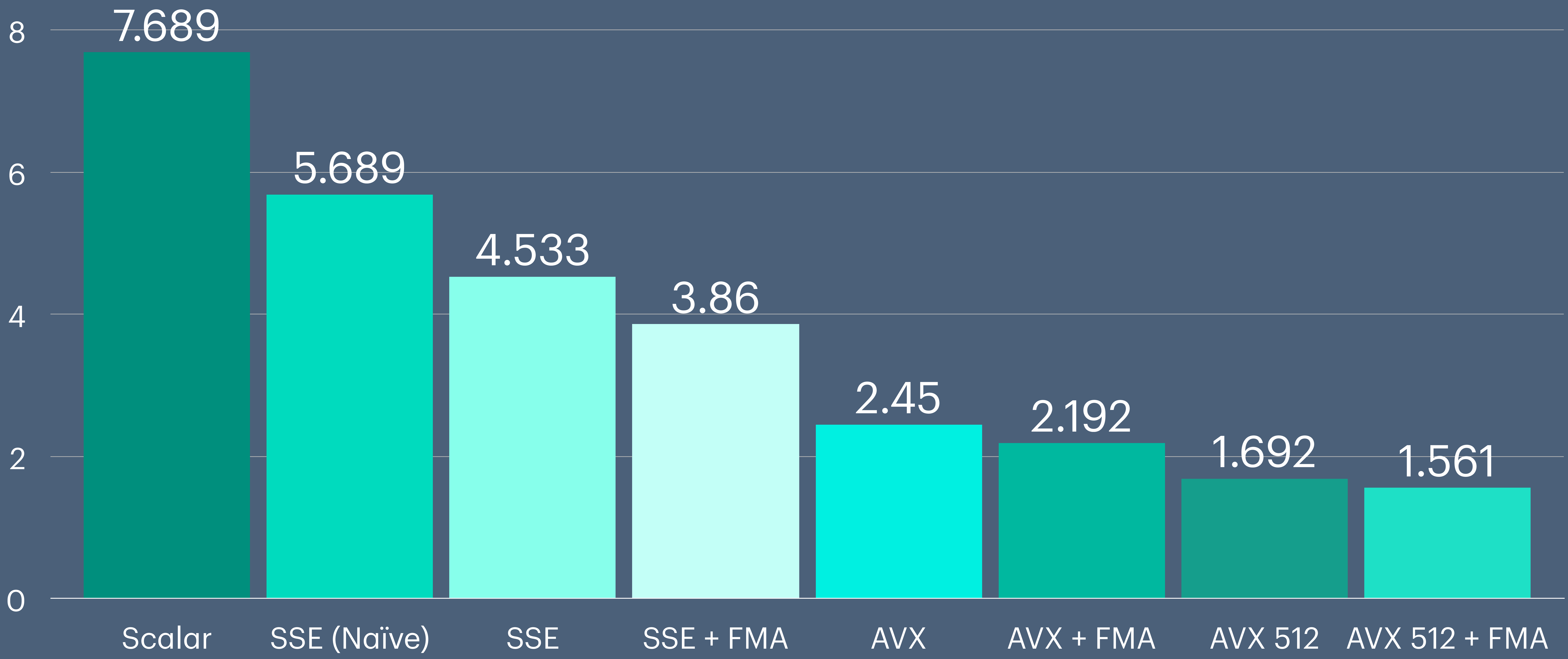
Vectorization with AVX 512

```
float dot(const float *const lhs, const float *const rhs)
{
    assert(sizeof(float) == 4);
    __m512 y = _mm512_setzero_ps();
    for (int i = 0; i < N; i += 16)
    {
        __m512 li = _mm512_load_ps(lhs + i), ri = _mm512_load_ps(rhs + i);
        __m512 tmp = _mm512_mul_ps(li, ri);
        y = _mm512_add_ps(y, tmp);
    }
    return y[0] + y[1] + y[2] + y[3] + y[4] + y[5] + y[6] + y[7] + y[8] +
y[9] + y[10] + y[12] + y[12] + y[13] + y[14] + y[15];
}
```

FMA

Fused Multiply-Add

- Perform rounding only once
  - Improve precision
- Leverage the pipeline
  - Improve performance
- Require AVX (even for 128-bit types)


8
6
4
2
0
7.689
5.689
4.533
3.86
2.45
2.192
1.692
1.561
Scalar
SSE (Naïve)
SSE
SSE + FMA
AVX
AVX + FMA
AVX 512
AVX 512 + FMA

Compute the Hamming Distance between 2 Strings

Count the number of indices that the values are not equal

- With vector instructions, we could compare several pairs of characters efficiently
- In light of the previous example, we should avoid reducing result too frequently
- For 8-bit integers, the result of equality comparison would be 0xFF (or -1) if equal or 0 otherwise
- If we accumulate the results, an 8-bit integer could maintain up to 256 times

Scalar Version

```
int hamming_dist()
{
    int y = N;
    for (int i = 0; i < N; i++)
        y -= i[str_a] == i[str_b];
    return y;
}
```

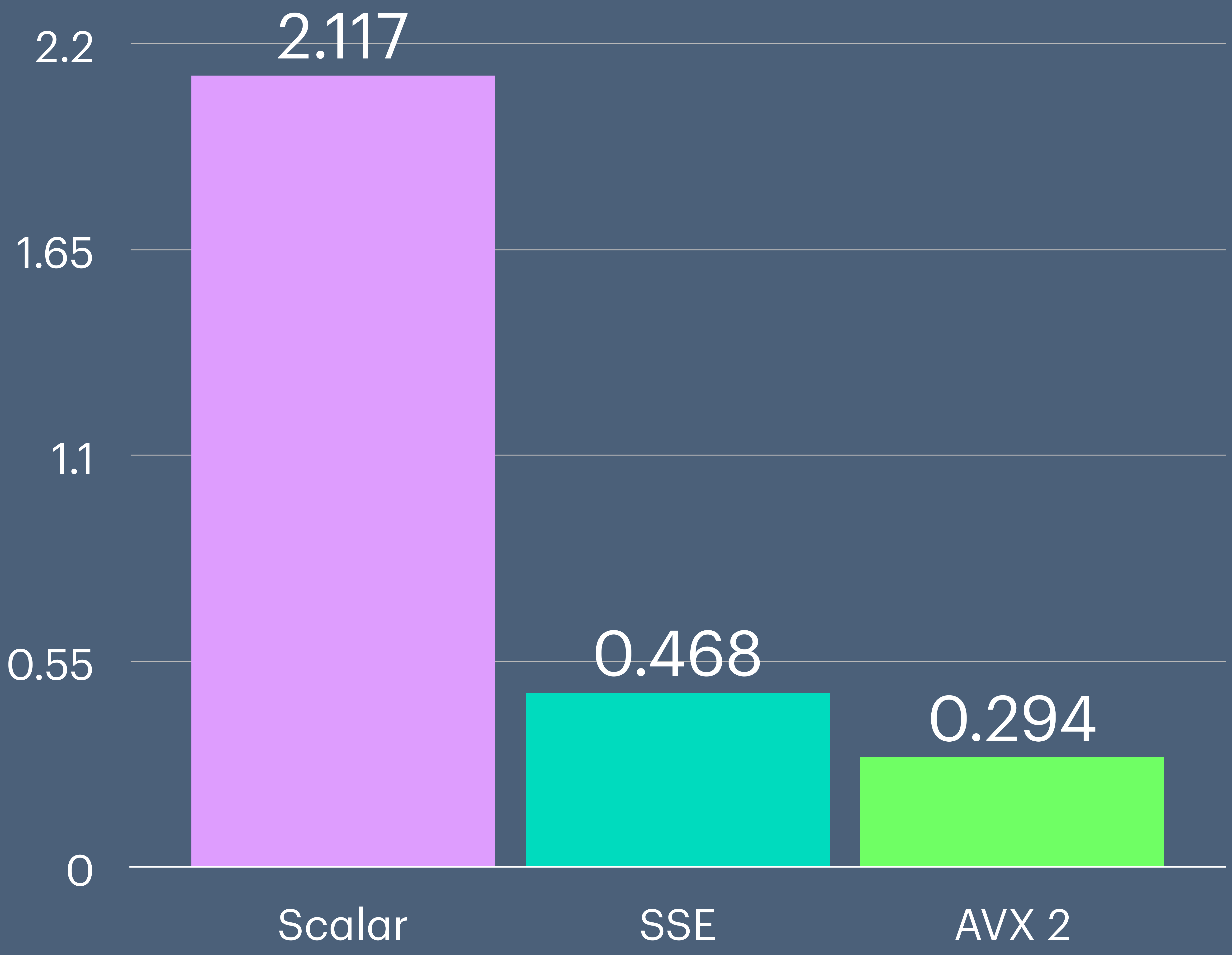
Vectorization with SSE

```c
int hamming_dist()
{
    int y = N;
    __m128i x = _mm_setzero_si128();
    _Alignas(16) signed char arr[16];
    for (int i = 0; i < N; i += 16)
    {
        __m128i a = _mm_load_si128(str_a
+ i), b = _mm_load_si128(str_b + i);
        __m128i tmp = _mm_cmpeq_epi8(a,
b);
        x = _mm_add_epi8(x, tmp);
        if (!(i % (16 * 256)))
        {
            _mm_store_si128(arr, x);
            for (int j = 0; j < 16; j++)
                y -= (-j[arr] % 256 +
256) % 256;
            x = _mm_setzero_si128();
        }
    }
    if (!(N % (16 * 256)))
    {
        _mm_storeu_si128(arr, x);
        for (int j = 0; j < 16; j++)
            y -= (-j[arr] % 256 + 256) %
256;
    }
    return y;
}
```

Vectorization with AVX 2

_mm256_cmpeq_epi8() requires AVX 2

```
int hamming_dist()
{
    int y = N;
    __m256i x = _mm256_setzero_si256();
    _Alignas(32) signed char arr[32];
    for (int i = 0; i < N; i += 32)
    {
        __m256i a =
_mm256_load_si256(str_a + i), b =
_mm256_load_si256(str_b + i);
        __m256i tmp =
_mm256_cmpeq_epi8(a, b);
        x = _mm256_add_epi8(x, tmp);
        if (!(i % (32 * 256)))
        {
            _mm256_store_si256(arr, x);
            for (int j = 0; j < 32; j++)
                y -= (-j[arr] % 256 +
256) % 256;
            x = _mm256_setzero_si256();
        }
    }
    if (!(N % (32 * 256)))
    {
        _mm256_storeu_si256(arr, x);
        for (int j = 0; j < 32; j++)
            y -= (-j[arr] % 256 + 256) %
256;
    }
    return y;
}
```


2.2
1.65
1.1
0.55
0
2.117
0.468
0.294
Scalar
SSE
AVX 2

Find the Maximum Prefix Sum

- This example is far more complicated than previous ones as there are data dependencies
- Maximum could still be found by vectorized instructions
- For prefix sum, we could update by $2 \log_2 n$ vectorized instructions
- Note that the 256-bit shift and shuffle instructions are constrained by the 128-bit lane boundary

Scalar Version

```
int max_prefix()
{
    int y = 0;
    for (int i = 0, ps = 0; i < N; i++)
    {
        ps += i[arr];
        if (y < ps)
            y = ps;
    }
    return y;
}
```

Vectorization with SSE 4.1

_mm_max_epi32() requires SSE 4.1

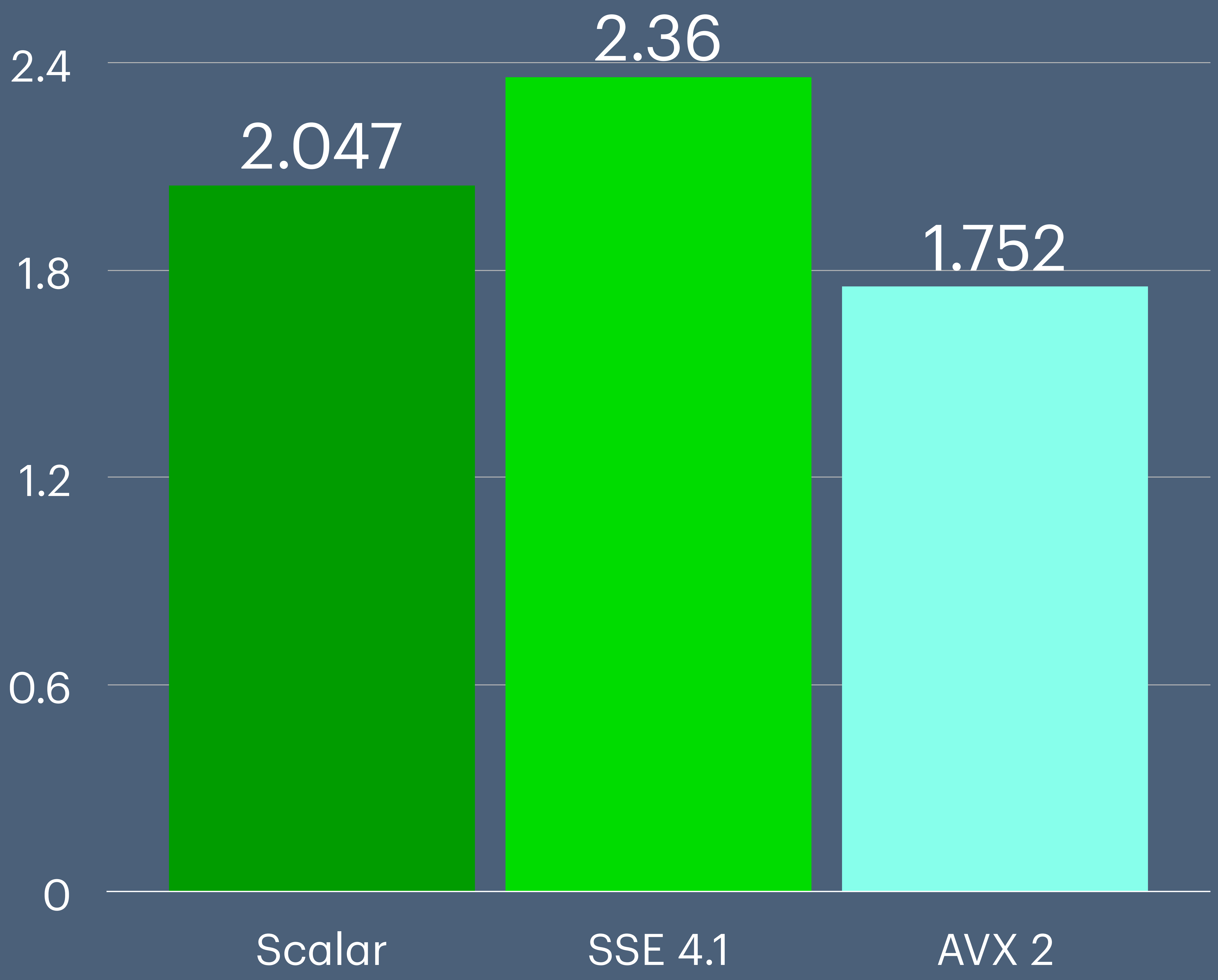
```
int max_prefix()
{
    assert(sizeof(int) == 4);
    __m128i mx =
_mm_setzero_si128(), sum =
_mm_setzero_si128();
    for (int i = 0; i < N; i += 4)
    {
        __m128i tmp =
_mm_load_si128(arr + i);
        tmp = _mm_add_epi32(tmp,
_mm_slli_si128(tmp, 4));
        tmp = _mm_add_epi32(tmp,
_mm_slli_si128(tmp, 8));
        sum = _mm_add_epi32(sum,
tmp);

        mx = _mm_max_epi32(mx, sum);
        sum = _mm_shuffle_epi32(sum,
0b11111111);
    }
    mx = _mm_max_epi32(mx,
_mm_slli_si128(mx, 4));
    mx = _mm_max_epi32(mx,
_mm_slli_si128(mx, 8));
    return _mm_extract_epi32(mx, 3);
}
```

Vectorization with AVX 2

_mm256_max_epi32() requires AVX 2

```
int max_prefix()
{
    assert(sizeof(int) == 4);
    __m256i mx = _mm256_setzero_si256(), sum =
_mm256_setzero_si256();
    for (int i = 0; i < N; i += 8)
    {
        __m256i tmp = _mm256_load_si256(arr + i);
        tmp = _mm256_add_epi32(tmp,
_mm256_alignr_epi8(tmp,
_mm256_permute2x128_si256(tmp, tmp,
_MM_SHUFFLE(0, 0, 2, 0)), 16 - 4));
        tmp = _mm256_add_epi32(tmp,
_mm256_alignr_epi8(tmp,
_mm256_permute2x128_si256(tmp, tmp,
_MM_SHUFFLE(0, 0, 2, 0)), 16 - 8));
        tmp = _mm256_add_epi32(tmp,
_mm256_permute2x128_si256(tmp, tmp,
_MM_SHUFFLE(0, 0, 2, 0)));
        sum = _mm256_add_epi32(sum, tmp);
        mx = _mm256_max_epi32(mx, sum);
        sum =
_mm256_set1_epi32(_mm256_extract_epi32(sum, 7));
    }
    mx = _mm256_max_epi32(mx,
_mm256_alignr_epi8(mx,
_mm256_permute2x128_si256(mx, mx, _MM_SHUFFLE(0,
0, 2, 0)), 16 - 4));
    mx = _mm256_max_epi32(mx,
_mm256_alignr_epi8(mx,
_mm256_permute2x128_si256(mx, mx, _MM_SHUFFLE(0,
0, 2, 0)), 16 - 8));
    mx = _mm256_max_epi32(mx,
_mm256_permute2x128_si256(mx, mx, _MM_SHUFFLE(0,
0, 2, 0)));
    return _mm256_extract_epi32(mx, 7);
}
```


2.4
1.8
1.2
0.6
0
2.047
2.36
1.752
Scalar
SSE 4.1
AVX 2

Auto-Vectorization

GCC

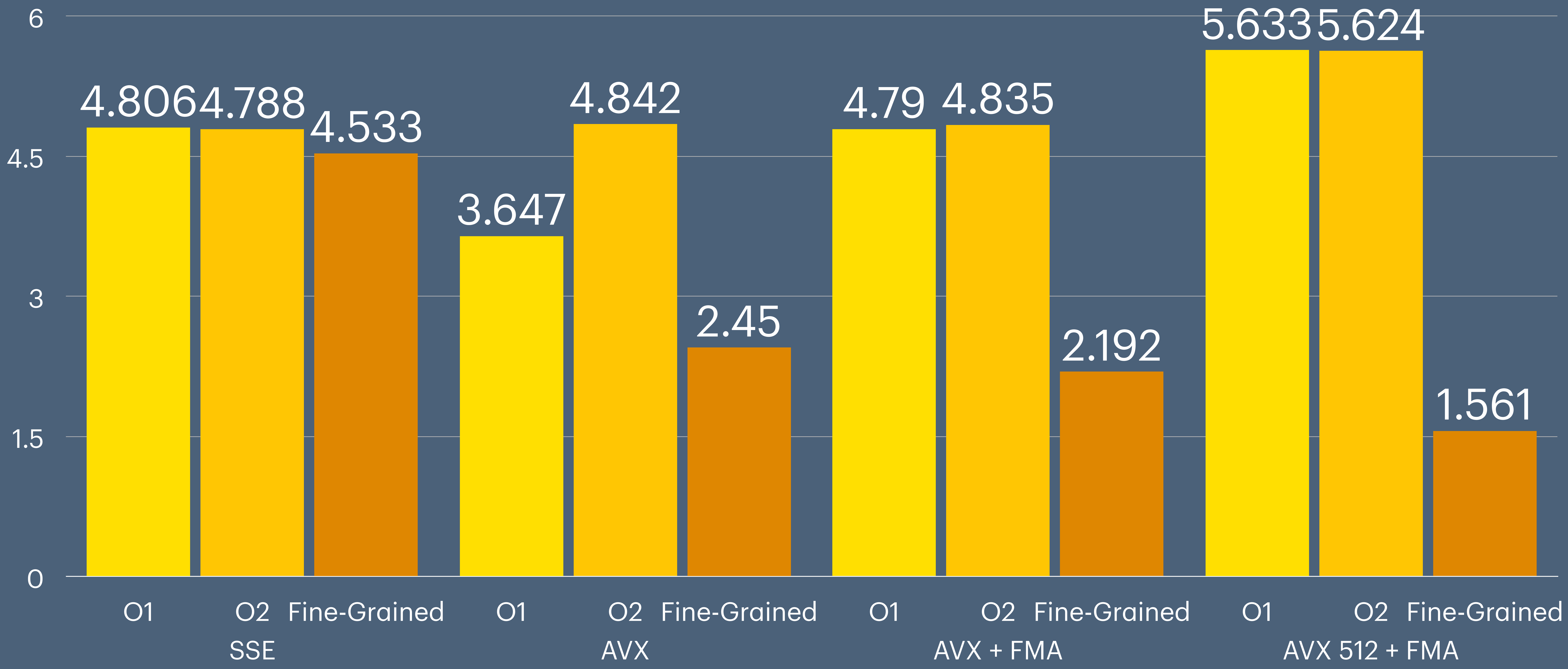
- `-ftree-optimize`
  - This is included since -O3 and above
- `#pragma GCC ivdep`
  - Tell the compiler that there is no dependency

OpenMP SIMD

Appears in OpenMP 4.0

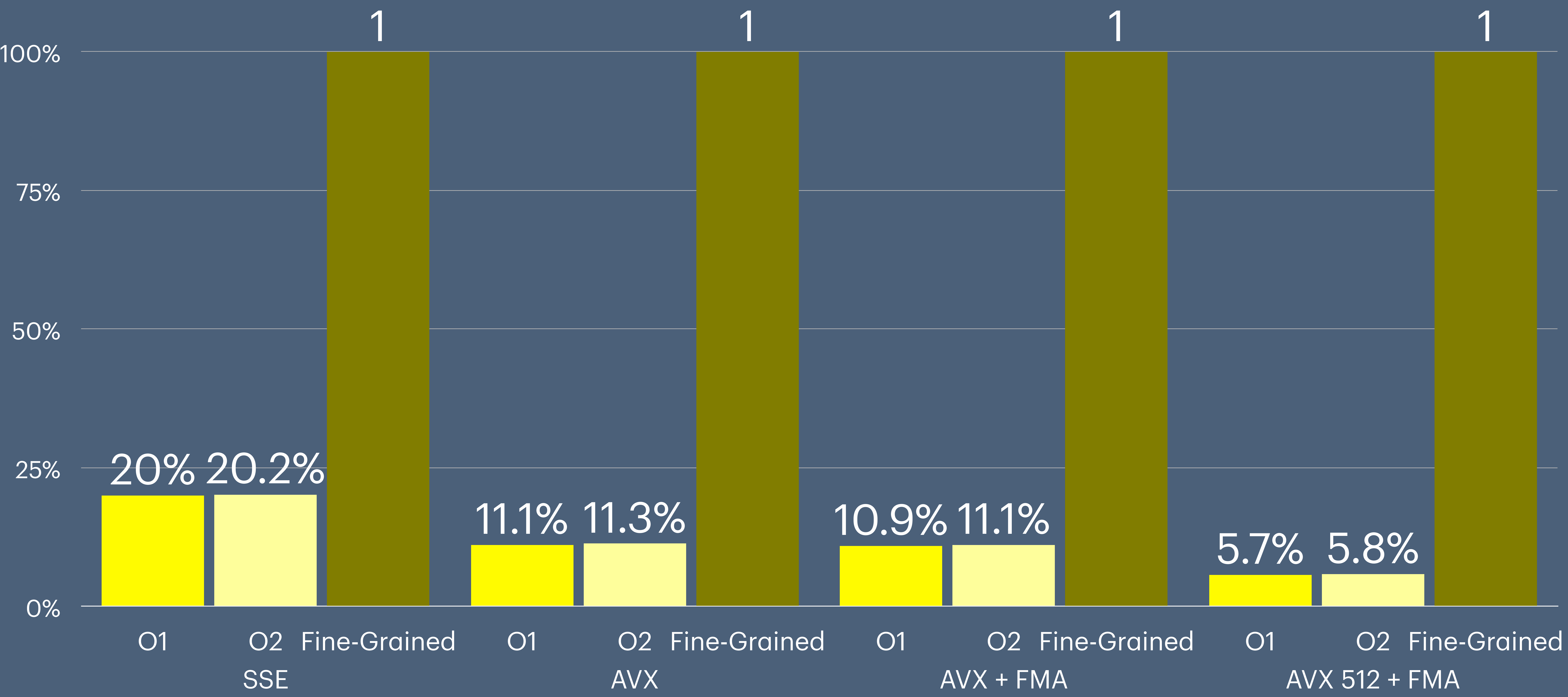
- `#pragma omp simd`
- Compiler optimization should also be set

Wall Time



6
4.5
3
1.5
0
4.806
4.788
4.533
3.647
4.842
2.45
4.79
4.835
2.192
5.633
5.624
1.561
O1
O2
Fine-Grained
SSE
AVX
AVX + FMA
AVX 512 + FMA

Packed Operation Percentage


O1
O2
Fine-Grained
SSE
O1
O2
Fine-Grained
AVX
O1
O2
Fine-Grained
AVX + FMA
O1
O2
Fine-Grained
AVX 512 + FMA