SIMD Computing

Performance you have already payed for

I am building Unum since 2015

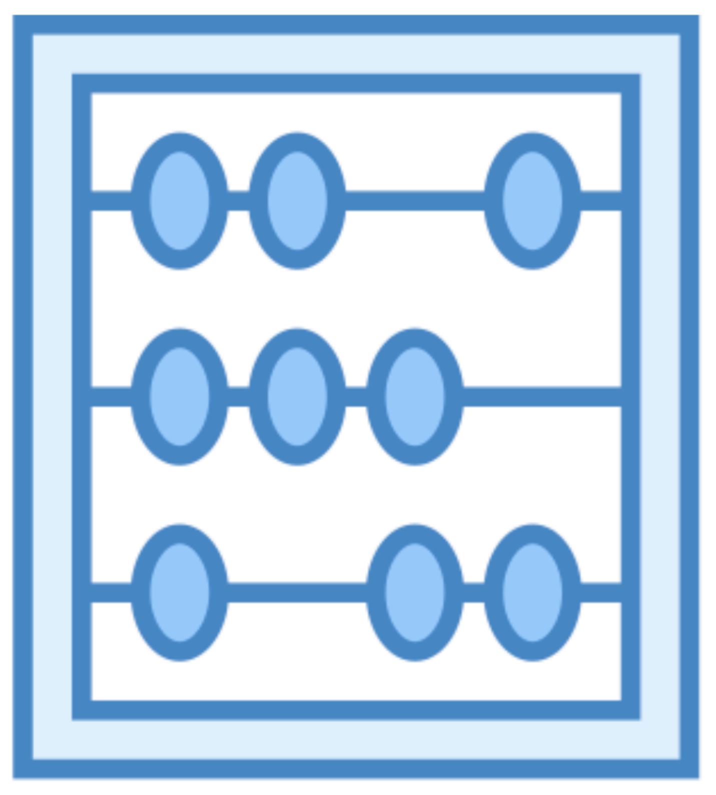
A neuro-symbolic computing framework

Hey, I’m Ashot!

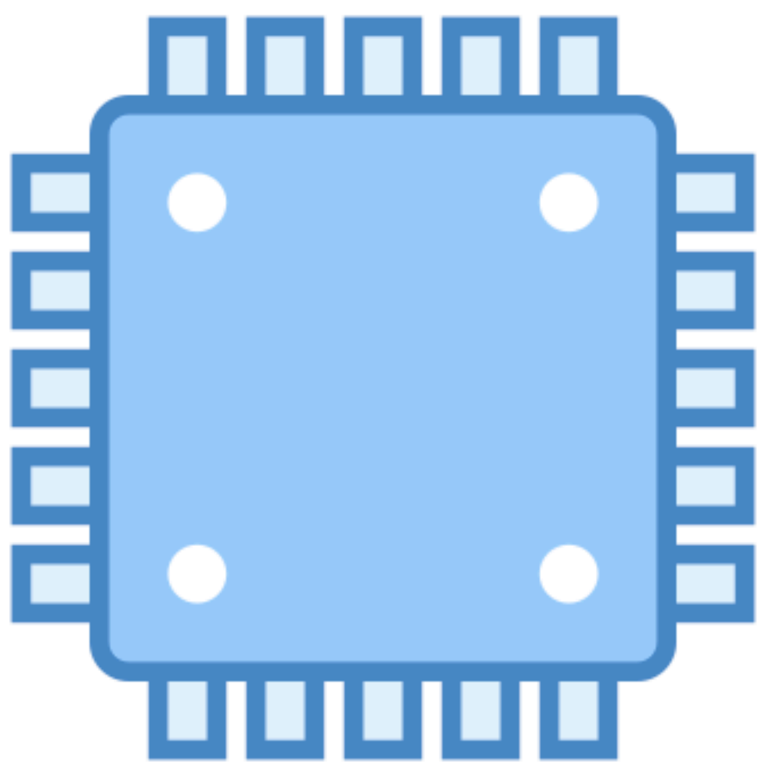
👶 Wrote my first line of code in the elementary school.
👦 Received my first freelance Web-Dev order in the middle school.
🧔 Launched my first profitable IT business in the high school.
🔭 Dropped my Astrophysics degree. Twice.
🏢 Already spent 5 years building Unum without external funding.
🌍 Visited over 50 countries across 4 continents, lived in 11 of them.
👅 Fluent in Russian, Armenian & English. Intermediate in a few other languages.
💻 These days I code in C++ 20, Python, Swift & LISPs.

Why am I here?

I love all kinds of computing and want to share!



Analog



Electrical

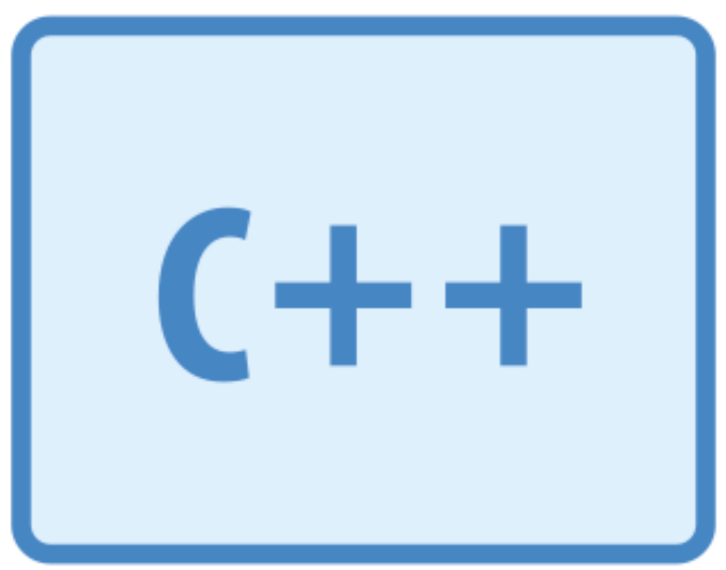
Optical

Quantum

Why are you here?

Let’s compare substring search performance on x86 silicon

Python
10 MB/s



C/C++
2 GB/s

SIMD
12 GB/s

github.com →
AshVardanian →
CppBenchSubstrSearch

The contents

Won’t be covered

- What’s parallelism?
- What’s SIMD?
- Branchless coding
- Boyer-Moore-Horspool
- Knuth-Morris-Pratt
- Rabin-Karp
- Aho-Corasic
- Commentz-Walter

Will be covered

- Brute Force Substring Search
- AVX2
- Speculative execution
- ARM NEON vs. x86 AVX-2
- The evil in AVX-512
- L0, L1, L2 Licenses
- Tools & Benchmarks
- Recommendations & Stories

Definitions

According to Intel

- **Mnemonic** is a register-invariant operation name.
  - Compare Equality.
  - Add.
  - Multiply.
  - And.

- **Intrinsic** is a built-into-compiler function that is replaced with instruction(s).
  - __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b);

- **Instruction** is register-specific.
  - vpcmpeqdy 0x3 (%rcx), %ymm0, %ymm4

SIMD Computing

Substring Search

```
std::string_view("apple") == std::string_view("banana")
```

```cpp
inline bool are_equal(std::string_view a, std::string_view b) noexcept {
    if (a.size() != b.size())
        return false;
    for (size_t i = 0; i < a.size(); i++)
        if (a[i] != b[i])
            return false;
    return true;
}
```

```cpp
inline bool are_equal(std::string_view a, std::string_view b) noexcept {
    if (a.size() != b.size())
        return false;
    size_t i = 0;
    for (; i < a.size() && a[i] == b[i]; i++)
        ;
    return i == a.size();
}
```

```
inline bool are_equal(char const *a, char const *b, char const *const a_end) noexcept {
    for (; a != a_end && *a == *b; a++, b++)
        ;
    return a_end == a;
}
```

cppreference.com   Create account   Search

Page   Discussion   View   Edit   History

C++ / Strings library / std::basic_string

std::basic_string<CharT,Traits,Allocator>::find

`size_type find( const basic_string& str, size_type pos = 0 ) const;`	(1)	(until C++11)
`size_type find( const basic_string& str, size_type pos = 0 ) const noexcept;`		(since C++11) (until C++20)
`constexpr size_type find( const basic_string& str, size_type pos = 0 ) const noexcept;`		(since C++20)
`size_type find( const CharT* s, size_type pos, size_type count ) const;`	(2)	(until C++20)
`constexpr size_type find( const CharT* s, size_type pos, size_type count ) const;`		(since C++20)
`size_type find( const CharT* s, size_type pos = 0 ) const;`	(3)	(until C++20)
`constexpr size_type find( const CharT* s, size_type pos = 0 ) const;`		(since C++20)
`size_type find( CharT ch, size_type pos = 0 ) const;`	(4)	(until C++11)
`size_type find( CharT ch, size_type pos = 0 ) const noexcept;`		(since C++11) (until C++20)
`constexpr size_type find( CharT ch, size_type pos = 0 ) const noexcept;`		(since C++20)
`template < class T > size_type find( const T& t, size_type pos = 0 ) const noexcept(/* see below */);`	(5)	(since C++17) (until C++20)
`template < class T > constexpr size_type find( const T& t, size_type pos = 0 ) const noexcept(/* see below */);`		(since C++20)

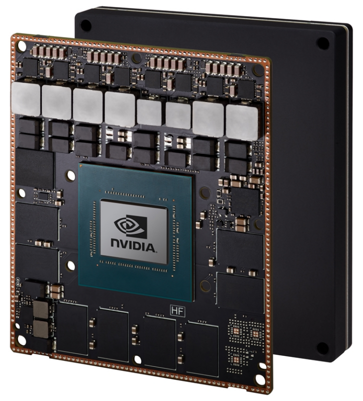
```cpp
struct stl_t {

    size_t next_offset(span_t haystack, span_t needle) noexcept {
        using str_view_t = std::basic_string_view<uint8_t>;
        str_view_t h_stl {haystack.data, haystack.len};
        str_view_t n_stl {needle.data, needle.len};
        size_t off = h_stl.find(n_stl);
        return off == str_view_t::npos ? not_found_k : off;
    }
};

/**
 *  \return Total number of matches.
 */
template <typename engine_at, typename callback_at>
size_t find_all(span_t haystack, span_t needle, engine_at &&engine, callback_at &&callback) {

    size_t last_match = 0;
    size_t next_offset = 0;
    size_t count_matches = 0;
    for (; (last_match = engine.next_offset(haystack.after_n(next_offset), needle)) != not_found_k;
         count_matches++, next_offset = last_match + 1)
        callback(last_match);

    return count_matches;
}
```

```cpp
struct naive_t {

    size_t next_offset(span_t haystack, span_t needle) noexcept {

        if (haystack.len < needle.len)
            return not_found_k;

        for (size_t off = 0; off <= haystack.len - needle.len; off++) {
            if (are_equal(haystack.data + off, needle.data, needle.len))
                return off;
        }

        return not_found_k;
    }
};
```

What have we accomplished so far?

Meet our lab rats!

Benchmark	IoT	Laptop	Server
python	4 MB/s	14 MB/s	11 MB/s
stl_t	560 MB/s	1,2 GB/s	1,3 GB/s
naive_t	520 MB/s	1 GB/s	900 MB/s



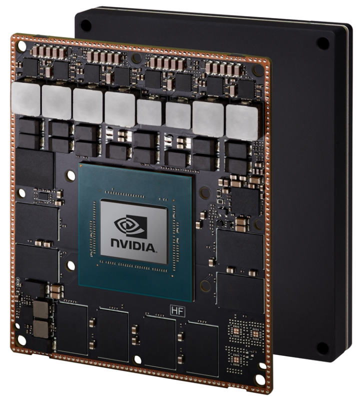
```cpp
struct prefixed_t {

    size_t next_offset(span_t haystack, span_t needle) noexcept {

        if (needle.len < 5)
            return naive_t {}.next_offset(haystack, needle);

        // Precomputed constants.
        uint8_t const *h_ptr = haystack.data;
        uint8_t const *const h_end = haystack.data + haystack.len - needle.len;
        size_t const n_suffix_len = needle.len - 4;
        uint32_t const n_prefix = *reinterpret_cast<uint32_t const *>(needle.data);
        uint8_t const *n_suffix_ptr = needle.data + 4;

        for (; h_ptr <= h_end; h_ptr++) {
            if (n_prefix == *reinterpret_cast<uint32_t const *>(h_ptr))
                if (are_equal(h_ptr + 4, n_suffix_ptr, n_suffix_len))
                    return h_ptr - haystack.data;
        }

        return not_found_k;
    }
};
```

3 levels of IFs instead of 2 ?!

What have we accomplished

With prefix matching

Benchmark	IoT	Laptop	Server
`python`	4 MB/s	14 MB/s	11 MB/s
`stl_t`	560 MB/s	1,2 GB/s	1,3 GB/s
`naive_t`	520 MB/s	1 GB/s	900 MB/s
`prefixed_t`	2 GB/s	3,3 GB/s	3,5 GB/s



SIMD Computing

Speculative Out-of-Order Execution

```cpp
struct prefixed_t {

    size_t next_offset(span_t haystack, span_t needle) noexcept {

        if (needle.len < 5)
            return naive_t {}.next_offset(haystack, needle);

        // Precomputed constants.
        uint8_t const *h_ptr = haystack.data;
        uint8_t const *const h_end = haystack.data + haystack.len - needle.len;
        size_t const n_suffix_len = needle.len - 4;
        uint32_t const n_prefix = *reinterpret_cast<uint32_t const *>(needle.data);
        uint8_t const *n_suffix_ptr = needle.data + 4;

        for (; h_ptr <= h_end; h_ptr++) {
            if (n_prefix == *reinterpret_cast<uint32_t const *>(h_ptr))
                if (are_equal(h_ptr + 4, n_suffix_ptr, n_suffix_len))
                    return h_ptr - haystack.data;
        }

        return not_found_k;
    }
};
```

3 levels of IFs instead of 2 ?!

Yes, with 4x less comparisons

```c
uint8_t const *const h_end = haystack.data + haystack.len - needle.len;
__m256i const n_prefix = _mm256_set1_epi32(*(uint32_t const *)(needle.data));

uint8_t const *h_ptr = haystack.data;
for (; (h_ptr + 32) <= h_end; h_ptr += 32) {

    __m256i h0 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr)), n_prefix);
    __m256i h1 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 1)), n_prefix);
    __m256i h2 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 2)), n_prefix);
    __m256i h3 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 3)), n_prefix);
    __m256i h_any = _mm256_or_si256(_mm256_or_si256(h0, h1), _mm256_or_si256(h2, h3));
    int mask = _mm256_movemask_epi8(h_any);

    if (mask) {
        for (size_t i = 0; i < 32; i++) {
            if (are_equal(h_ptr + i, needle.data, needle.len))
                return i + (h_ptr - haystack.data);
        }
    }
}
```

256 bits fits 8x 32-bit integers!

Let’s compare 4x8 prefixes per loop cycle!

```
uint32_t needles[8] = {0,0,0,0,0,0,0,0};
```

```
_mm256_set1_epi32(*(uint32_t const *)(needle.data));
```

```
uint32_t matches[8] = {a[0] == b[0], a[1] == b[1], … };
```

```
                                      _mm256_loadu_si256((__m256i const *)(h_ptr))
__m256i h1 = _mm256_cmpeq_epi32(                                                  , n_prefix);


__m256i h_any = _mm256_or_si256(_mm256_or_si256(h0, h1), _mm256_or_si256(h2, h3));
int mask = _mm256_movemask_epi8(h_any);
```

```
(matches0 | matches1) | (matches2 | matches3)
```

```
__m256i → int → bool
```

What have we accomplished so far?

Our first SIMD approach using AVX2 is 3x faster!

Benchmark	IoT	Laptop	Server
`python`	4 MB/s	14 MB/s	11 MB/s
`stl_t`	560 MB/s	1,2 GB/s	1,3 GB/s
`naive_t`	520 MB/s	1 GB/s	900 MB/s
`prefixed_t`	2 GB/s	3,3 GB/s	3,5 GB/s
`prefixed_avx2_t`		8,5 GB/s	10,5 GB/s

Let’s speculate a little!

```
uint8_t const *const h_end = haystack.data + haystack.len - needle.len;
__m256i const n_prefix = _mm256_set1_epi32(*(uint32_t const *)(needle.data));

uint8_t const *h_ptr = haystack.data;
for (; (h_ptr + 32) <= h_end; h_ptr += 32) {

    __m256i h0_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr));
    int masks0 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h0_prefixes, n_prefix));
    __m256i h1_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr + 1));
    int masks1 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h1_prefixes, n_prefix));
    __m256i h2_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr + 2));
    int masks2 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h2_prefixes, n_prefix));
    __m256i h3_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr + 3));
    int masks3 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h3_prefixes, n_prefix));
```

```
__m256i h0 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr)), n_prefix);
__m256i h1 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 1)), n_prefix);
__m256i h2 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 2)), n_prefix);
__m256i h3 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 3)), n_prefix);
__m256i h_any = _mm256_or_si256(_mm256_or_si256(h0, h1), _mm256_or_si256(h2, h3));
int mask = _mm256_movemask_epi8(h_any);
```

Same Mnemonics + More Instructions = Higher Performance?!

```
__m256i h0_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr));
int masks0 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h0_prefixes, n_prefix));
__m256i h1_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr + 1));
int masks1 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h1_prefixes, n_prefix));
__m256i h2_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr + 2));
int masks2 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h2_prefixes, n_prefix));
__m256i h3_prefixes = _mm256_loadu_si256((__m256i const *)(h_ptr + 3));
int masks3 = _mm256_movemask_epi8(_mm256_cmpeq_epi32(h3_prefixes, n_prefix));
int mask = masks0 | masks1 | masks2 | masks3;
```

What have we accomplished so far?

Speculation-restricting SIMD is ~30% slower than flexible.

Benchmark	IoT	Laptop	Server
`prefixed_t`	2 GB/s	3,3 GB/s	3,5 GB/s
`prefixed_avx2_t`		8,5 GB/s	10,5 GB/s
`speculative_avx2_t`		12 GB/s	9,7 GB/s
`speculative_avx512_t`			10 GB/s
`speculative_neon_t`	4,3 GB/s		

How much can the CPU speculate?

intel.com doesn't say

Intel® Core™ i9-9880H Processor

16M Cache, up to 4.80 GHz

- Specifications
  - **Essentials**
  - CPU Specifications
  - Supplemental Information
  - Memory Specifications
  - Processor Graphics
  - Expansion Options
  - Package Specifications
  - Advanced Technologies
  - Security & Reliability
- Ordering and Compliance
- Product Images
- Compatible Products
- Drivers and Software
- Technical Documentation

Essentials Export specifications

Product Collection	9th Generation Intel® Core™ i9 Processors
Code Name	Products formerly Coffee Lake
Vertical Segment	Mobile
Processor Number ?	i9-9880H
Status	Launched
Launch Date ?	Q2'19
Lithography ?	14 nm
Recommended Customer Price ?	$556.00

CPU Specifications

# of Cores ?	8
# of Threads ?	16
Processor Base Frequency ?	2.30 GHz
Max Turbo Frequency ?	4.80 GHz
Cache ?	16 MB Intel® Smart Cache

How much can the CPU speculate?

wikichip.org knows the L1 cache size

[Edit/Modify Cache Info]

Cache Organization

L1$	512 KiB	**L1I$**	256 KiB	8x32 KiB	8-way set associative	
		L1D$	256 KiB	8x32 KiB	8-way set associative	write-back
L2$	2 MiB			8x256 KiB	4-way set associative	write-back
L3$	16 MiB			8x2 MiB	16-way set associative	write-back

Re-order buffer
From Wikipedia, the free encyclopedia

Instruction window
From Wikipedia, the free encyclopedia

Speculative execution
From Wikipedia, the free encyclopedia

Out-of-order execution
From Wikipedia, the free encyclopedia

In theory we have up to 32KiB instructions cache per core. The practical OoOE queue depth should probably be closer to 16 load/store entries.

```
__m256i h0 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr)), n_prefix);
__m256i h1 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 1)), n_prefix);
__m256i h2 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 2)), n_prefix);
__m256i h3 = _mm256_cmpeq_epi32(_mm256_loadu_si256((__m256i const *)(h_ptr + 3)), n_prefix);
__m256i h_any = _mm256_or_si256(_mm256_or_si256(h0, h1), _mm256_or_si256(h2, h3));
int mask = _mm256_movemask_epi8(h_any);
```

ARM & x86: How different are they?

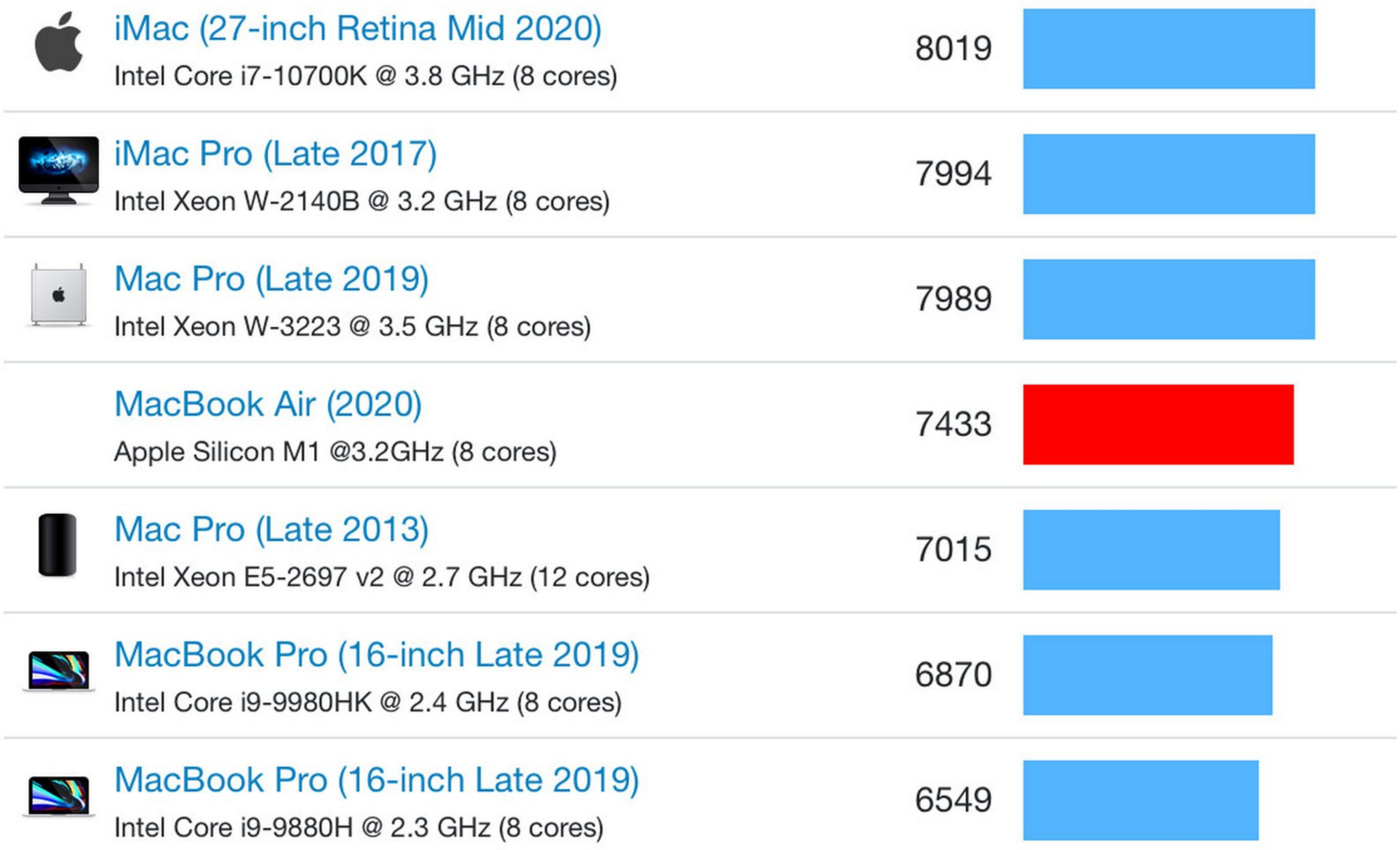
```
uint32x4_t masks0 = vceqq_u32(vld1q_u32((uint32_t const *)(h_ptr)), n_prefix);
uint32x4_t masks1 = vceqq_u32(vld1q_u32((uint32_t const *)(h_ptr + 1)), n_prefix);
uint32x4_t masks2 = vceqq_u32(vld1q_u32((uint32_t const *)(h_ptr + 2)), n_prefix);
uint32x4_t masks3 = vceqq_u32(vld1q_u32((uint32_t const *)(h_ptr + 3)), n_prefix);

uint32x4_t masks = vorrq_u32(vorrq_u32(masks0, masks1), vorrq_u32(masks2, masks3));
uint64x2_t masks64x2 = vreinterpretq_u64_u32(masks);
bool has_match = vgetq_lane_u64(masks64x2, 0) | vgetq_lane_u64(masks64x2, 1);
```

ARM is efficient!

	IoT	IoT	IoT	Laptop	Server
ISA	ARM	ARM	ARM	X86	X86
Process	12 nm TSMC	12 nm TSMC	12 nm TSMC	14 nm Intel	14 nm Intel
Cores	2	4	8	8 (16t)	22 (44t)
TDP/Core	2 W	1 W	0,5 W	5,6 W	6,3 W
CPU Frequency	2,1 GHz	1,7 GHz	1,2 GHz	2,3 GHz	2,1 GHz
Performance/Core	3,3 GB/s	2,5 GB/s	2,1 GB/s	12 GB/s	10,5 GB/s
Bytes/Joule	1,6 GB/J	2,5 GB/J	4,2 GB/J	2,1 GB/J	1,6 GB/J

ARM-ageddon

Is coming to consumer desktops



- Intel i9-9980HK
  - 45 W
  - 6870 points
  - 152 points/W
- Apple Silicon M1
  - 10 W
  - 7433 points
  - 753 points/W

5 nm ARM is 5x more efficient than 14 nm Intel.

Can the the compiler replace us?

Auto-vectorization quality varies between LLVM & GCC

Benchmark	IoT	Laptop	Server
`prefixed_t`	2 GB/s	3,3 GB/s	3,5 GB/s
`prefixed_avx2_t`		8,5 GB/s	10,5 GB/s
`speculative_avx2_t`		12 GB/s	9,7 GB/s
`speculative_avx512_t`			10 GB/s
`speculative_neon_t`	4,3 GB/s		
`prefixed_autovec_t`		~ 1,5 GB/s	~ 4 GB/s

SIMD Computing

Tooling & Benchmarks

/home/vtune/intel/advixe/projects/CppBench - Intel Advisor Beta
File View Help
Start Survey Analysis
Welcome
e000
Vectorization Workflow
Threading Workflow
OFF Batch mode
Run Roofline
Collect
With Callstacks
For All Memory Levels
1. Survey Target
Collect
Mark Loops for Deeper Analysis
Select checkboxes in the Survey & Roofline tab to mark loops for other Advisor analyses.
-- There are no marked loops --
1.1 Find Trip Counts and FLOP
Collect
Trip Counts
FLOP
-- Analyze all loops --
2.1 Check Memory Access Patter...
Collect
-- No loops selected --
2.2 Check Dependencies
Collect
-- No loops selected --
Re-finalize Sur...
Elapsed time: 248,48s
Vectorized
Not Vectorized
FILTER: All Modules
All Sources
Loops And Functions
All Threads
Customize View
Summary
Survey & Roofline
Refinement Reports
INTEL ADVISOR BETA
ROOFLINE
Function Call Sites and Loops
Performance Issues
CPU Time
Total Time
Self Time
Type
Why No Vectorization?
Vectorized Loops
Vecto...
Gain ...
VL (V...
Compute Performance
Self GFLOPS
Self AI
[loop in search<av::speculative_avx2_t>] 40,235s 40,235s Vectorized (Bo... AVX2 8; 32 0 0
[loop in search<av::naive_t>] 40,193s 40,149s Scalar 0 0
[loop in search<av::hybrid_avx2_t>] 40,197s 40,109s Scalar 0 0
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
Loop in search<av::speculative_avx2_t>
40,235s
Vectorized (Body)
Total time
AVX; AVX2
Instruction Set
40,235s
Self time
Static Instruction Mix Summary
Dynamic Instruction Mix Summary
Memory 52% (38398187549, 13)
Compute 8% (5912700712, 2)
Other 40% (29555097813, 10.01)
CPU Total Time
1,36259e-08s
Per Iteration
0,11430s
Per Instance
Average Trip Counts: 8388607
GFLOPS: 0
GINTOPS: 2,57
Roofline
Int32 Vector Add Peak
L2 Bandwidth
L3 Bandwidth
DRAM Bandwidth
100
10
1
10
3.82 GINTOPS (↑1.5x)
2.57 GINTOPS
0.27 INTOP/Byte
INTOP/Byte (Arithmetic Intensity)
This loop is mostly memory bound but may also be compute bound
The performance of the loop is bounded by the DRAM bandwidth.
You can switch to the Recommendations tab to see optimization recommendations in the Roofline Conclusions section.
Traits
Inserts
Intel Advisor
[bin64]
/home/vtune/intel/adv...

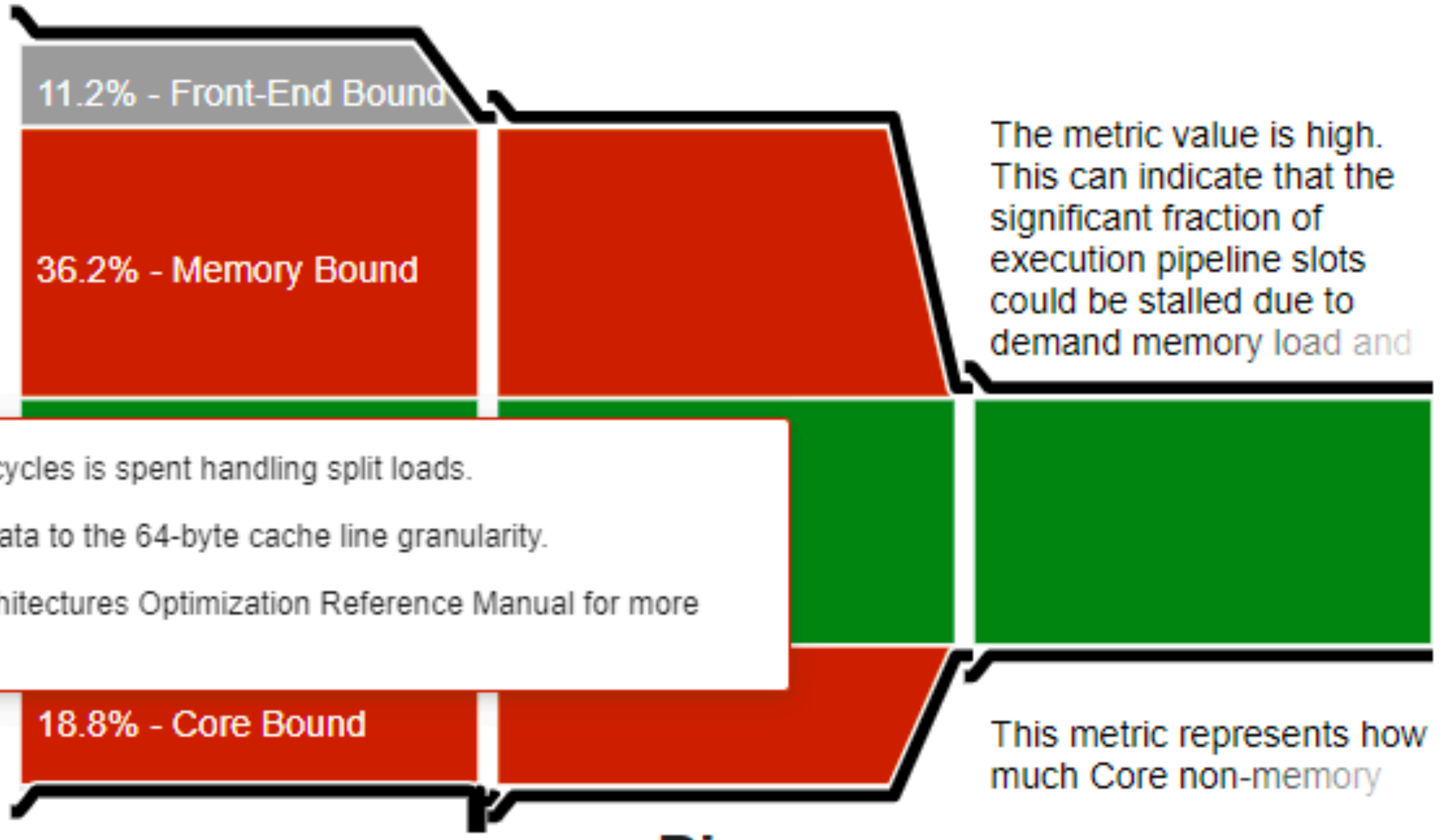
Elapsed Time: 53.168s

Clockticks:	824,638,500,000	
Instructions Retired:	753,207,000,000	
CPI Rate:	1.095	
MUX Reliability:	0.991	
Retiring:	33.1%	of Pipeline Slots
Front-End Bound:	11.2%	of Pipeline Slots
Bad Speculation:	0.7%	of Pipeline Slots
Back-End Bound:	55.0%	of Pipeline Slots
Memory Bound:		
L1 Bound:		
DTLB Overhead:		
Loads Blocked by Store Forwarding:		
Lock Latency:		
Split Loads:	37.3%	of Clockticks
4K Aliasing:	0.0%	of Clockticks
FB Full:	0.5%	of Clockticks
L2 Bound:	12.0%	of Clockticks
L3 Bound:	13.0%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	0.0%	of Clockticks
SQ Full:	0.8%	of Clockticks
DRAM Bound:	4.2%	of Clockticks
Store Bound:	0.1%	of Clockticks
Core Bound:	18.8%	of Pipeline Slots
Divider:	2.4%	of Clockticks
Port Utilization:	21.2%	of Clockticks
Cycles of 0 Ports Utilized:	26.8%	of Clockticks
Cycles of 1 Port Utilized:	10.3%	of Clockticks
Cycles of 2 Ports Utilized:	10.0%	of Clockticks
Cycles of 3+ Ports Utilized:	17.0%	of Clockticks
Vector Capacity Usage (FPU):	6.2%	
Average CPU Frequency:	2.7 GHz	
Total Thread Count:	327	
Paused Time:	0s	

Issue: A significant portion of cycles is spent handling split loads.

Tips: Consider aligning your data to the 64-byte cache line granularity.

See the Intel 64 and IA-32 Architectures Optimization Reference Manual for more details.



µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Intel VTune

Retiring:	35.1%	of Pipeline Slots
Front-End Bound:	4.3%	of Pipeline Slots
Bad Speculation:	0.2%	of Pipeline Slots
Back-End Bound:		
Memory Bound:		
L1 Bound:	[illegible]	[illegible]
L2 Bound:	42.5% ⚑	of Clockticks
L3 Bound:	0.4%	of Clockticks
DRAM Bound:	2.2%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	12.8% ⚑	of Pipeline Slots

This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance.

We collected 19e9 L1 cache hits and 30e9 L1 cache misses, when requesting nearby addresses!

Some mysteries are just not meant to be solved!

```cpp
    bm::RegisterBenchmark("stl", search<stl_t>)->MinTime(default_secs_k);
    bm::RegisterBenchmark("naive", search<naive_t>)->MinTime(default_secs_k);
    bm::RegisterBenchmark("prefixed", search<prefixed_t>)->MinTime(default_secs_k);
#ifdef __AVX2__
    bm::RegisterBenchmark("simultaneous_avx2", search<speculative_avx2_t>)
        ->MeasureProcessCPUTime()
        ->MinTime(default_secs_k)
        ->UseRealTime()
        ->Threads(1)
        ->Threads(2)
        ->Threads(count_threads_k / 2)
        ->Threads(count_threads_k);
#endif
```

Google Benchmark!

```
-------------------------------------------------------------------------------------------------
Benchmark                                                  Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------------------------
stl/min_time:5.000                                 417237303 ns    415840765 ns           17 bytes/s=1.29105G/s bytes/s/core=1.29105G/s
naive/min_time:5.000                               463408929 ns    462852933 ns           15 bytes/s=1.15992G/s bytes/s/core=1.15992G/s
prefixed/min_time:5.000                            148634454 ns    148470213 ns           47 bytes/s=3.61602G/s bytes/s/core=3.61602G/s
prefixed_avx2/min_time:5.000                        53778471 ns     53723628 ns          129 bytes/s=9.9932G/s bytes/s/core=9.9932G/s
hybrid_avx2/min_time:5.000                          53606280 ns     53586867 ns          128 bytes/s=10.0187G/s bytes/s/core=10.0187G/s
speculative_avx2/min_time:5.000                     43984034 ns     43962859 ns          156 bytes/s=12.2119G/s bytes/s/core=12.2119G/s
simultaneous_avx2/min_time:5.000/real_time/threads:1  44952777 ns   44931867 ns          158 bytes/s=11.943G/s bytes/s/core=11.943G/s
simultaneous_avx2/min_time:5.000/real_time/threads:2  21819874 ns   43626261 ns          318 bytes/s=24.6047G/s bytes/s/core=12.3023G/s
simultaneous_avx2/min_time:5.000/real_time/threads:8   6588667 ns   52572785 ns          800 bytes/s=81.484G/s bytes/s/core=10.1855G/s
simultaneous_avx2/min_time:5.000/real_time/threads:16  9680513 ns  149853379 ns          672 bytes/s=55.4589G/s bytes/s/core=3.46618G/s
naive/[a-z]/min_time:5.000                         441841304 ns    441581643 ns           14 bytes/s=1.21579G/s bytes/s/core=1.21579G/s
naive/[A-Za-z]/min_time:5.000                      344485733 ns    344315800 ns           20 bytes/s=1.55924G/s bytes/s/core=1.55924G/s
```

The Tools

And the pitfalls

- Intel Advisor.
    - Static analysis.
- Godbolt.
    - Less assembly doesn’t mean faster.
    - Code execution order is loosely defined on OoOE CPUs.
- Intel VTune.
    - Runtime profiling.
    - When measuring code vectorization - only includes floats.
    - Subjectively, as complicated as CMake.
- **Google Benchmark**.
    - CPU **frequency scaling** should be turned off on desktop.

SIMD Computing

Pitfalls & Recommendations

Diminishing returns with AVX-512

High Complexity

Year	1997	1999	2001	2004	2006	2006	2008	2011	2013	2015
Extension	MMX	SSE	SSE2	SSE3	SSSE3	SSE4.1/2	AVX	FMA	AVX2	AVX-512
Mnemonics	+46	+62	+70	+10	+16	+54	+89	+20	+135	+347

Minimal Support (Intel-only)

<table>
<tr><th>AVX-512 Subset</th><th>F</th><th>CD</th><th>ER</th><th>PF</th><th>4FMAPS</th><th>4VNNIW</th><th>VPOPCNTDQ</th><th>VL</th><th>DQ</th><th>BW</th><th>IFMA</th><th>VBMI</th><th>VNNI</th><th>BF16</th><th>VBMI2</th><th>BITALG</th><th>VPCLMULQDQ</th><th>GFNI</th><th>VAES</th><th>VP2INTERSECT</th></tr>
<tr><td>Knights Landing (Xeon Phi x200) processors (2016)</td><td colspan="2" rowspan="8">Yes</td><td colspan="2" rowspan="2">Yes</td><td colspan="16">No</td></tr>
<tr><td>Knights Mill (Xeon Phi x205) processors (2017)</td><td colspan="2">Yes</td><td>Yes</td><td colspan="13">No</td></tr>
<tr><td>Skylake-SP, Skylake-X processors (2017)</td><td colspan="2" rowspan="6">No</td><td colspan="2" rowspan="6">No</td><td rowspan="4">No</td><td colspan="3" rowspan="6">Yes</td><td colspan="10">No</td></tr>
<tr><td>Cannon Lake processors (2018)</td><td colspan="2">Yes</td><td colspan="8">No</td></tr>
<tr><td>Cascade Lake processors (2019)</td><td colspan="2" rowspan="2">No</td><td rowspan="4">Yes</td><td colspan="7">No</td></tr>
<tr><td>Cooper Lake processors (2020)</td><td>Yes</td><td colspan="6">No</td></tr>
<tr><td>Ice Lake processors (2019)</td><td rowspan="2">Yes</td><td colspan="2" rowspan="2">Yes</td><td rowspan="2">No</td><td colspan="5" rowspan="2">Yes</td><td>No</td></tr>
<tr><td>Tiger Lake processors (2020)</td><td>Yes</td></tr>
</table>

It gets worse

L0, L1, L2 Frequencies on Intel Xeon Gold 5120

Mode	Base	Turbo Frequency/Active Cores													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Normal	2,200 MHz	3,200 MHz	3,200 MHz	3,000 MHz	3,000 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,600 MHz	2,600 MHz
AVX2	1,800 MHz	3,100 MHz	3,100 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,200 MHz	2,200 MHz
AVX512	1,200 MHz	2,900 MHz	2,900 MHz	2,500 MHz	2,500 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz

Frequency scaling

Brands by Intel

- 2005, Enhanced Intel SpeedStep Technology (EIST)
- 2006, Dynamic Acceleration Technology (DAT)
- 2008, Turbo Boost Technology (TBT)
- 2010, Turbo Boost Technology 2.0 (TBT 2.0)
- 2015, Speed Shift Technology (SST)
- 2016, Turbo Boost Max Technology 3.0 (TBMT)
- 2018, Thermal Velocity Boost (TVB)
- 2019, Speed Select Technology (SST)

Frequency scaling licenses

In case of Intel

Mode	Instruction	Base Frequency	All Turbo Frequency	1-Core Turbo Frequency	Potential Base Loss	Potential Turbo Loss
L0	`_mm_add_epi64`, `_mm256_add_epi64`, `_mm256_addnot_si256`	2,2 GHz	2,6 GHz	3,2 GHz	0%	0%
L1	`_mm256_mul_epu32`, `_mm256_add_ps`, `_mm512_mul_epi64`	1,8 GHz	2,2 GHz	3,1 GHz	18%	31%
L2	`_mm256_mullo_epi64`, `_mm512_add_ps`, `_mm512_mullox_epi64`	1,2 GHz	1,6 GHz	2,9 GHz	45%	38%

Mixing Light & Heavy instructions

Causes Soft & Hard Transitions and CPU Halting

- Heavy operations:
    - Load/Store
    - All float operations
    - Integer multiplication ≥ 256b
    - Shuffle/Blend ≥ 256b

- Use of any AVX-512 causes hard L1 transition
- Other transitions are soft and need sufficient demand
- Mean transition time is ~8 micro seconds or ~25K CPU cycles!

Final Recipe

What to use aside from vanilla C++?

- Integer operations:
  - 256b AVX2 instructions on x86
  - 128b NEON instructions on ARM

- Floating-point operations:
  - Most FP programs are data-parallel and auto-vectorizable
  - For other critical operations use 128b instructions

- Avoid integer divisions & modulus at ~~any cost~~ of 50 cycles on Cannon Lake!
- ~~Obviously~~ don't use AVX-512 unless you absolutely need to
  - Video coders process 8x8 pixels x8-bits per color = 512 bits at a time

Final Recipe

When to use? To optimize hot data path

- When you have less data than GPU threads
- More data than GPU memory
- When latency is more important than throughput, examples:
  - Deep Learning inference

- Solution has linear worst case complexity
- Solution isn’t data parallel, examples:
  - Compression
  - Encoding
  - Parsing

What’s the future like

For Assembly developers

- CISC becomes CISC-ier:
  - X86 receives Advanced Matrix eXtensions
  - ARM receives SVE • NEON = SVE2 with up to 2048-bit registers
  - L4 cache prefetching intrinsics for Sapphire Rapids?

- RISC becomes ~~RISC-ier~~ more popular!
  - A RISC core already lives in every CISC core
  - Simpler ISAs generally have higher frequencies to compensate throughput
  - Custom chips for every workload in hyperscalers: AWS, GCP, Azure
  - Flexible compilers, domain-specific plugins: MLIR, TVM

Expected Results

What we have achieved @ unum.xyz

- ML-oriented Geometry:
  - Vanilla Cpp ~16 ns
  - SIMD ~3 ns
- Set Lookups:
  - std::unordered_set ~ 150 ns
  - unum::set ~ 15 ns
- Random Numbers Generators:
  - rand() ~ 900 MB/s
  - unum::rand() ~ 12 GB/s
- RegEx Matching:
  - std::regex ~ 300 MB/s
  - unum::regex ~ 14 GB/s
- DB Scans:
  - PostgreSQL ~ 50 MB/s
  - UnumDB ~ 3 GB/s
- Text Indexing:
  - ElasticSearch ~ 10 MB/s
  - UnumDB ~ 450 MB/s