Write fast code like a native

CppNorth '24

Saksham Sharma

Why me?

- Director, Quant Research Tech at Tower Research Capital
  - High frequency trading firm based out of NYC
- Develop low latency trading systems
  - C++
- Develop high throughput research systems
  - C++ and Python
- Member of WG21 - ISO C++ Standardization Committee
  - Not an expert

Why me?

- Talk a lot on Python and C++ (often in the same breath)
- Program analysis research and functional programming in a past life
- Love performance, software abstractions, and clean APIs

Why fast?

- More work done
- Better user experience
- Helps compete with alternatives
    - Faster DBs are adopted more
    - Faster websites get more users
    - Faster trading strategies do better in the market

Why C++?

Why anything else?

- Zero cost abstractions
  - APIs and language constructs are designed to facilitate performance
- Very close to bare metal
  - Write inline assembly if needed
- A huge number of compile time hints to compiler
  - Templates etc provide compile-time polymorphism
  - Helps generate faster code

How do we make our code faster?

Not so fast :)

The process - and this talk

- Numbers! Numbers! Numbers!
  - “Fast” and “slow” mean nothing
  - We’ll measure in cycles, microseconds, nanoseconds
  - Develop an intuition

The process - and this talk

- Form a mental model of how long things “should” take
  - Understand latencies of various computing operations
  - Have a baseline of latencies from numerous sources
- Prioritize the important parts of your code
  - “Hot path” / “Fast path”
  - Hide ugliness there behind nice abstractions
- Experience!
  - Look at case studies, like today’s talk :)

Story time

HFT - High frequency trading

- AKA: Low Latency C++
- Really really fast
- Really really computationally intensive
- Really really correct

Burj Khalifa

Height:
828 meters
2,722 feet

Speed of light:
~ 1 foot per ns

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

When a Microsecond Is an Eternity: High Performance Trading Systems in C++

CppCon 2017: Carl Cook "When a Microsecond Is an Eternity: High Performance Trading Systems in C++"

A very good minimum time (wire to wire) for a software-based trading system is around 2.5us

That's less than the time it takes light to travel from the top of the spire to the ground

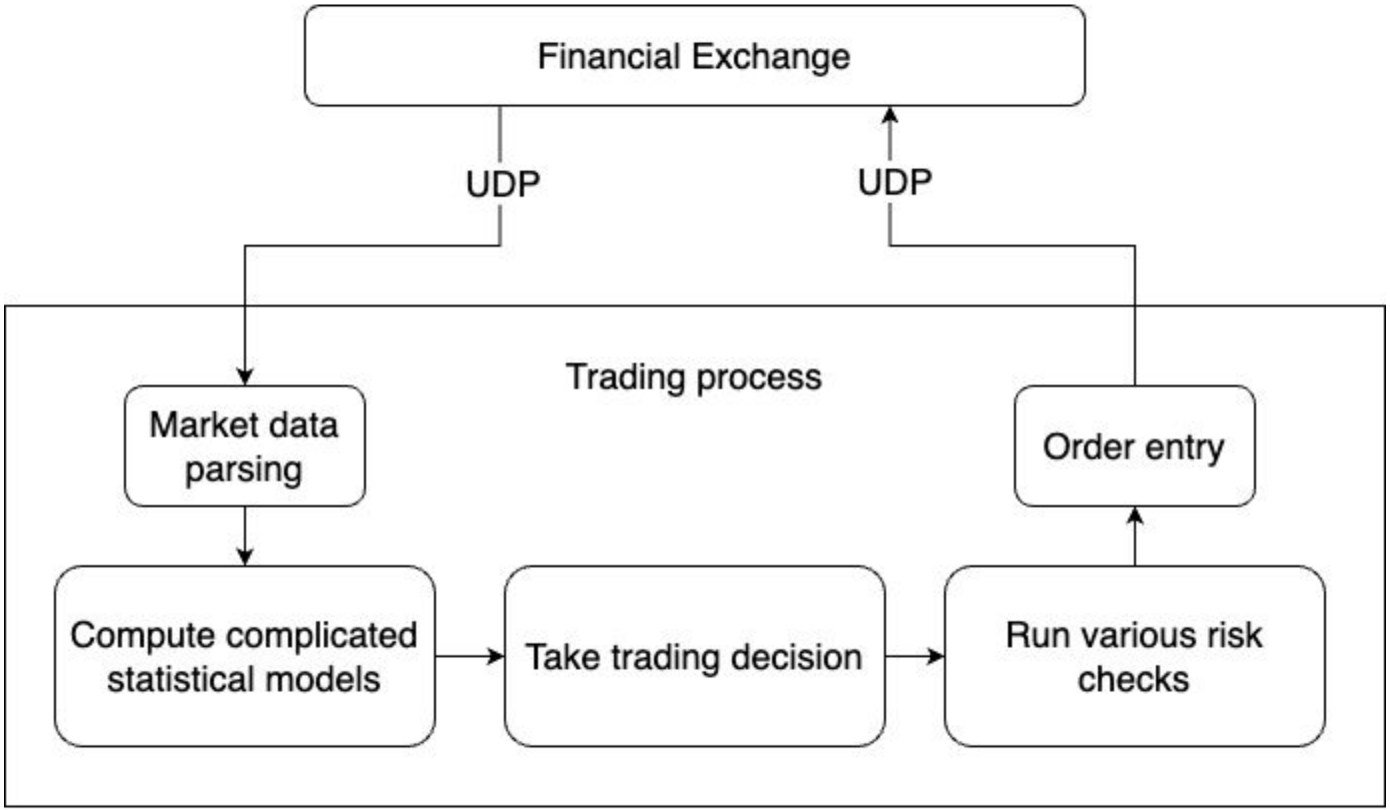
Photo by Donald Tong / CC BY-SA 3.0

cppcon | 2017

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

When a Microsecond Is an Eternity: High Performance Trading Systems in C++

Flow chart of HFT work



HFT - High frequency trading

- Very important to have predictable runtime of your code
    - 2.5us runtime, tails shouldn’t be 25us
    - Can’t afford cache misses
    - Can’t afford mallocs
    - Can’t afford syscalls
- “send a trading order” hot-path is very rare
    - Cache misses aplenty!
- Computational logic is very complex and ever-evolving
    - Hard to write fast code without understanding how much time it takes

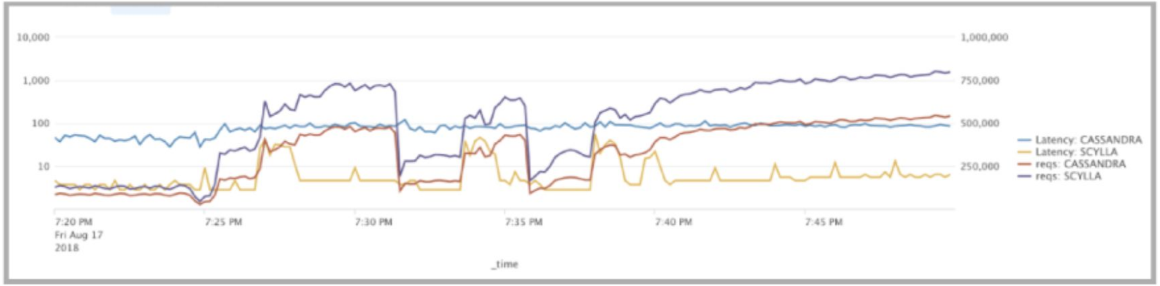
Now coming back to the real software development world

How Does Cassandra Compare to ScyllaDB?

Comcast, a major user of ScyllaDB, benchmarked ScyllaDB against Cassandra prior to deploying in production. They found that under their synthetic load tests they were able to achieve 8ms latency with ScyllaDB compared to 100ms with Cassandra, a 92% reduction in typical latency and a 95% reduction in their p99 latency.

Testing Cassandra and ScyllaDB side-by-side, Comcast measured significant differences in the long-tail of performance, as shown by the metrics below:

Cassandra		**Median**	**90%**	**99%**	**99.90%**
	DeviceRecording-Read	3.959	22.338	84.318	218.15
	DeviceRecording-Write	1.248	4.094	31.914	117.344
Scylla					
	DeviceRecording-Read	0.523	0.982	3.839	9.786
	DeviceRecording-Write	0.609	0.913	2.556	7.152



HTTPS://WWW.SCYLLADB.COM/GLOSSARY/LOW-LATENCY-DATABASE/

TODO: Add more examples

So, when did we go from microseconds to milliseconds?

Latencies!

Latencies!

Arguably my favorite topic...

Arguably the most boring topic...

Case study 1 - Matrix transpose

```
void transpose(float* src, float* dst,
               int N, int M) {
  for (int n = 0; n < N * M; n++) {
      int i = n / N;
      int j = n % N;
      dst[n] = src[M * j + i];
  }
}
```

Any estimates how long this code takes? For:

- N = 1000
- M = 1000

Case study 2 - Sum of numbers

```
int looper(int n) {
    int i = 0;
    int total = 0;
    while (i < n) {
        total += i;
        i++;
    }
    return total;
}
```

Any estimates how long this code takes? For n ==

- 10
- 1000
- 1000,0000

Case study 3 - IO

```
// NO CODE
```

How long does it take to:

- Write 1 GB to a disk
- Send 1 GB over network
- Read 1 GB from network

HOW DO WE EVEN ANSWER THAT?

Latencies

- Know thy CPU

Know thy CPU

- Modern CPUs are really fast
  - 3-5 billion “clock ticks” per second per core
  - What is a tick? What can you do in one tick?
- Modern CPUs understand “assembly”
  - Instructions stored “in memory” that the CPU evaluates “in order”
- Modern CPUs are not “serial”
  - It can evaluate a lot of things in parallel
  - As long as they’re not interdependent (sometimes they are)

Know thy CPU - Assembly

Here's some assembly code for the x86 architecture.

0s and 1s in the memory / program that you compile.

```
test    edi, edi
jle     .LBB1_1
lea     eax, [rdi - 1]
lea     ecx, [rdi - 2]
imul    rcx, rax
shr     rcx
lea     eax, [rdi + rcx]
dec     eax
ret
```

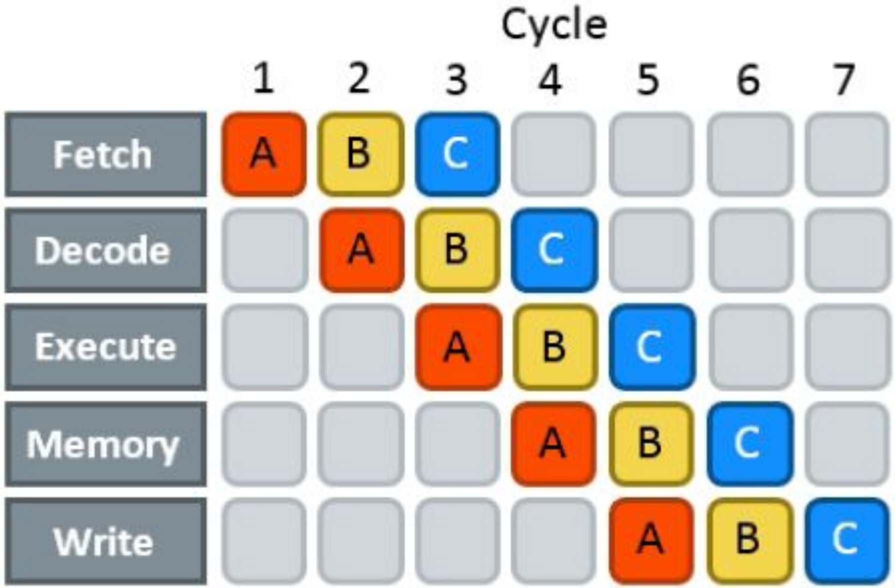
Know thy CPU - Instruction pipeline

Each instruction goes through a:

- Fetch   (load assembly code from “memory”)
- Decode (interpret)
- Execute (read inputs, compute arithmetic)
- Write   (results back)

Know thy CPU - Instruction pipeline

But not serially...



https://en.algorithmica.org/hpc/pipelining/

Know thy CPU - Cache

- All your data is in memory or on disk.
- Too slow! (we’ll see later)
- CPU loads recently used / possibly required data in cache
  - Much faster to access storage, directly on the CPU chip

Latencies

- Know thy CPU
- Profile!
  - Using high precision profiling tools

Profiling latencies - clock

```cpp
using namespace std::chrono;

auto start = high_resolution_clock::now();
...
...
auto end = high_resolution_clock::now();

duration_cast<nanoseconds>(
  end - start
).count()
```

The usual, profiling using time!

- Real
- Sensitive to jitter
  - Can be a good thing

Profiling latencies - Rdtsc

```
// Cycle counts
uint64_t get_tsc() {
    uint32_t l;
    uint64_t h;
    __asm__("rdtsc" : "=a"(l), "=d"(h));
    return l | (h << 32);
}

auto start = get_tsc();
transpose(v.data(), output, N, M);
auto end = get_tsc();
```

Count the number of CPU cycles that passed during this period.

- As noisy as clock

Profiling latencies - Rdtscp

```cpp
// Cycle counts
uint64_t get_tsc() {
    uint32_t l;
    uint64_t h;
    __asm__("rdtscp" : "=a"(l), "=d"(h));
    return l | (h << 32);
}

auto start = get_tsc();
transpose(v.data(), output, N, M);
auto end = get_tsc();
```

Same as *rdtsc* but...

- Ensures it is not reordered around other instructions

Profiling latencies - Rdpmc

```c
// Intel PMU
unsigned long a, d, c;
c = (1UL << 30);
__asm__ volatile(
    "rdpmc" : "=a"(a),
    "=d"(d) : "c"(c)
);
return (a | (d << 32));
```

Count the number of assembly instructions that elapsed.

- Much more deterministic
- Much harder to set up!
  - Single CPU
  - Requires “opening”

Profiling latencies - Rdpmc

There’s others

- Cycle count (same as before, kind of)
- Branch misses
- Total branches
- L2 cache misses
- L3 cache misses
- L3 cache hits

Latencies

- Know thy CPU
- Profile!
  - Using high precision profiling tools
- Read the assembly
  - https://godbolt.org

COMPILER EXPLORER

Add... More Templates

Sponsors intel Google think-cell

Share Policies Other

C++ source #1

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```cpp
void transpose(float* src, float* dst,
               int N, int M) {
  for (int n = 0; n < N * M; n++) {
    int i = n / N;
    int j = n % N;
    dst[n] = src[M * j + i];
  }
}
```

x86-64 gcc 14.1 (Editor #1)

x86-64 gcc 14.1 -O3

A Output... Filter... Libraries Overrides + Add new... Add tool...

```asm
transpose(float*, float*, int, int):
        mov     r8, rdi
        mov     edi, edx
        mov     r9, rsi
        mov     r11d, edx
        imul    edi, ecx
        test    edi, edi
        jle     .L1
        movsx   r10, edi
        xor     esi, esi
        cmp     ecx, 1
        jne     .L4
.L6:
        mov     eax, esi
        cdq
        idiv    edi
        add     edx, eax
        movsx   rdx, edx
        movss   xmm0, DWORD PTR [r8+rdx*4]
        movss   DWORD PTR [r9+rsi*4], xmm0
        add     rsi, 1
        cmp     r10, rsi
        jne     .L6
.L1:
        ret
.L4:
        mov     eax, esi
        cdq
        idiv    r11d
        imul    edx, ecx
        add     edx, eax
        movsx   rdx, edx
        movss   xmm0, DWORD PTR [r8+rdx*4]
        movss   DWORD PTR [r9+rsi*4], xmm0
        add     rsi, 1
        cmp     rsi, r10
        jne     .L4
        ret
```

Maybe there's an easier way?

Latencies

Develop an intuition for how long things take!

Arithmetic latency

- A CPU can do basic arithmetic on numerical types in <1ns
  - +, -, boolean operations, bit operations, ...
- Multiplication and division in ~1ns or so
  - ~1 billion operations in one second!
- Assuming CPU has the numbers loaded up in cache

Memory and cache latency

- Reading and writing to memory is a bit slow
  - ~60-100ns
- The CPU has a (limited) cache to speed things up
- Recently used data can be found there
  - Very recently used (L1) => ~4 cycles
  - Recently used (L2)   => ~10 cycles
  - A bit older (L3)   => ~40 cycles

Memory and cache latency

- Reading and writing to memory is a bit slow
  - ~60-100ns
- The CPU has a (limited) cache to speed things up
- Recently used data can be found there
  - Very recently used (L1) => ~100 KB
  - Recently used (L2) => ~2 MB
  - A bit older (L3) => ~64 MB

Syscall and Memory allocation latency

- Allocating memory (malloc) can be ~100ns-5us
  - Task of finding a new space in the heap
  - Arbitrary STL data structures can hit a malloc
- Any kernel operation (syscall) can be 1-1000s of us
  - Changing process parameters etc
  - Fetching the pid of your process
  - Reads / writes to a file or network

Latencies

Best source for this:

“Agner Fog instruction tables”

https://www.agner.org/optimize/instruction_tables.pdf

Latency:	This is the delay that the instruction generates in a dependency chain. The numbers are minimum values. Cache misses, misalignment, and exceptions may increase the clock counts considerably. Where hyperthreading is enabled, the use of the same execution units in the other thread leads to inferior performance. Subnormal numbers, NAN's and infinity do not increase the latency. The time unit used is core clock cycles, not the reference clock cycles given by the time stamp counter.
Reciprocal throughput:	The average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread.

Integer instructions

Instruction	Operands	μops fused domain	μops unfused domain	μops each port	Latency	Reciprocal through put	Comments
Move instructions							
MOV	r,i	1	1	p0156		0.25	
MOV	r,r	1	1	p0156	1	0.25	
MOV	r8,r8h r8h,r8	1	1	p0156	2	0.25	
MOV	r8/16,m	1	2	p23 p0156		0.5	
MOV	r32/64,m	1	1	p23	3	0.5	all addressing modes
MOV	m,r	1	2	p49 p78	2	1	
MOV	m,i	1	2	p49 p78		1	
MOVNTI	m,r	2	2	p23 p4	~400	~30	
MOVSX MOVZX MOVSXD	r,r	1	1	p0156	1	0.25	
MOVSX MOVZX	r16,m8	1	2	p23 p0156	3	0.5	

Putting it all together

Matrix transpose

```
void transpose(float* src, float* dst,
               int N, int M) {
   for (int n = 0; n < N * M; n++) {
       int i = n / N;
       int j = n % N;
       dst[n] = src[M * j + i];
   }
}
```

Any estimates how long this code takes? For:

- N = 10
- M = 10

Matrix transpose

```
void transpose(float* src, float* dst,
               int N, int M) {
   for (int n = 0; n < N * M; n++) {
       int i = n / N;
       int j = n % N;
       dst[n] = src[M * j + i];
   }
}
```

- We use godbolt
- We read assembly
- We use our intuition

If X = N * M,

- X multiplications, X modulus
- X reads and writes

Matrix transpose

```cpp
#include <cstdint>
#include <vector>
#include <iostream>
#include <chrono>

// Intel PMU
uint64_t get_tsc() {
    uint32_t low;
    uint64_t high;
    __asm__("rdtsc" : "=a"(low), "=d"(high));
    return low | (high << 32);
}

__attribute__((noinline)) void transpose(float* src, float* dst,
                int N, int M) {
    for (int n = 0; n < N * M; n++) {
        int i = n / N;
        int j = n % N;
        dst[n] = src[M * j + i];
    }
}

int main() {
    int N, M;
    std::vector<float> v;
    std::cin >> N >> M;
    std::cout << "N is " << N << " and M is " << M << std::endl;
    for (int i = 0; i < N * M; i++) {
        float x;
        std::cin >> x;
        v.push_back(x);
    }
    int attempt = 10;
    while (attempt--) {
        auto start_t = std::chrono::high_resolution_clock::now();
        auto start = get_tsc();
        std::vector<float> output;
        output.resize(N * M);
        transpose(v.data(), output.data(), N, M);
        auto end = get_tsc();
        auto end_t = std::chrono::high_resolution_clock::now();
        std::cout << "Attempt: " << attempt << "\t"<< (end - start) << " cycles\t"
```

x86-64 gcc 14.1 -O3

```asm
get_tsc():
        rdtsc
        sal     rdx, 32
        mov     eax, eax
        or      rax, rdx
        ret
transpose(float*, float*, int, int):
        mov     r8, rdi
        mov     edi, edx
        mov     r9, rsi
        mov     r11d, edx
        imul    edi, ecx
        test    edi, edi
        jle     .L4
        movsx   r10, edi
        xor     esi, esi
```

Output (0/0) x86-64 gcc 14.1 - 2300ms (401759B) ~26375 lines filtered Compiler License

Executor x86-64 gcc 14.1 (C++, Editor #1)

x86-64 gcc 14.1 -O3

Execution arguments...

3 3 7 8 1 2 0 2 8 7 2

```
Program stdout
N is 3 and M is 3
Attempt: 9      1828 cycles     810 ns
Attempt: 8      186 cycles      109 ns
Attempt: 7      186 cycles      100 ns
Attempt: 6      133 cycles      90 ns
Attempt: 5      79 cycles       60 ns
Attempt: 4      80 cycles       60 ns
Attempt: 3      79 cycles       60 ns
Attempt: 2      80 cycles       60 ns
Attempt: 1      185 cycles      121 ns
Attempt: 0      79 cycles       70 ns
```

x86-64 gcc 14.1 - 858ms

https://godbolt.org/z/dbhfG57aW

Matrix transpose

- We use godbolt!
- We read assembly!
- We use our intuition

```
Attempt: 1   728 cycles   283 ns
Attempt: 2   804 cycles   283 ns
Attempt: 3   716 cycles   249 ns
Attempt: 4   776 cycles   267 ns
Attempt: 5   720 cycles   250 ns
Attempt: 6   772 cycles   266 ns
Attempt: 7   774 cycles   267 ns
Attempt: 8   714 cycles   248 ns
Attempt: 9   712 cycles   248 ns
Attempt: 10  722 cycles   250 ns
```

Matrix transpose

Latency goes down as we repeat the same operation again and again!

Thanks to...

  instruction and data cache

```
Attempt: 1   728 cycles   283 ns
Attempt: 2   804 cycles   283 ns
Attempt: 3   716 cycles   249 ns
Attempt: 4   776 cycles   267 ns
Attempt: 5   720 cycles   250 ns
Attempt: 6   772 cycles   266 ns
Attempt: 7   774 cycles   267 ns
Attempt: 8   714 cycles   248 ns
Attempt: 9   712 cycles   248 ns
Attempt: 10  722 cycles   250 ns
```

What about a larger N and M? (1000)

Matrix transpose

Reaching the limits of cache here :)

~1e4 times bigger data

~6000 times slower

Not bad!

```
Attempt: 1   6079624 cycles   1900 us
Attempt: 2   5304328 cycles   1657 us
Attempt: 3   4585658 cycles   1433 us
Attempt: 4   4556150 cycles   1424 us
Attempt: 5   4551800 cycles   1422 us
Attempt: 6   4551650 cycles   1422 us
Attempt: 7   4552914 cycles   1422 us
Attempt: 8   4550336 cycles   1422 us
Attempt: 9   4552926 cycles   1422 us
Attempt: 10  4550762 cycles   1422 us
```

Let's isolate the cache effects

Matrix transpose

- N == M == 1000
- Random input
- Input vector is kept same each time
- Output vector is fixed
- Both instruction and data cache get hot

```
v.clear();
for (int i = 0; i < N * M; i++) {
    v.push_back(static_cast<float>(dist(gen)));
}
for (int attempt = 1; attempt <= 10; attempt++) {
   output.clear();
   auto start_t = high_resolution_clock::now();
   auto start = get_tsc();
   transpose(v.data(), output.data(), N, M);
   auto end = get_tsc();
   auto end_t = high_resolution_clock::now();
   ....
}
```

Matrix transpose

- N == M == 1000
- Random input
- Input vector is kept same each time
- Output vector is fixed
- Both instruction and data cache get hot

```
Attempt: 1   6538272 cycles   2043 us
Attempt: 2   6140224 cycles   1919 us
Attempt: 3   5308954 cycles   1659 us
Attempt: 4   5267920 cycles   1646 us
Attempt: 5   5265190 cycles   1645 us
Attempt: 6   5367700 cycles   1677 us
Attempt: 7   5453816 cycles   1704 us
Attempt: 8   5337600 cycles   1668 us
Attempt: 9   5402570 cycles   1688 us
Attempt: 10  5355182 cycles   1673 us
```

Matrix transpose

- N == M == 1000
- Random input
- Input vector is **clobbered each time**
- Output vector is fixed
- **Only instruction cache stays hot**

```
void clobber_cache(size_t size_in_mb) {
    size_t size = size_in_mb * 1024 * 1024;
    std::vector<char> memory_block(size);
    for (size_t i = 0; i < size; i += 64) {
        memory_block[i] = rand() % 256;
    }
}

for (int attempt = 1; attempt <= 10; attempt++) {
    output.clear();
    clobber_cache(1000);  // size in MBs
    auto start_t = high_resolution_clock::now();
    auto start = get_tsc();
```

Matrix transpose

- `N == M == 1000`
- Random input
- Input vector is **clobbered each time**
- Output vector is fixed
- **Only instruction cache stays hot**

Attempt: 1	4208508 cycles	1686 us
Attempt: 2	4170348 cycles	1670 us
Attempt: 3	4238142 cycles	1698 us
Attempt: 4	4121882 cycles	1651 us
Attempt: 5	4195206 cycles	1681 us
Attempt: 6	4157590 cycles	1665 us
Attempt: 7	4203496 cycles	1684 us
Attempt: 8	4128082 cycles	1654 us
Attempt: 9	4189100 cycles	1678 us
Attempt: 10	4146992 cycles	1661 us

WHAT ABOUT RDPMC / INSTRUCTION COUNT?

Instruction count

- Highly deterministic
- Instruction count for transpose
  - N == M == 10 => 1109
  - N == M == 1000 => 11000009
- 1e4 times more

```
auto start_pmc = get_pmc_i();
for (int n = 0; n < N * M; n++) {
    int i = n / N; int j = n % N;
    dst[n] = src[M * j + i];
}
auto end_pmc = get_pmc_i();
```

Sum of numbers

Sum of numbers

```
int looper(int n) {
    int i = 0;
    int total = 0;
    while (i < n) {
        total += i;
        i++;
    }
    return total;
}
```

Any estimates how long this code takes? For n ==

- 10
- 1000
- 1000,0000

https://godbolt.org/z/155zzfheE

Sum of numbers

```
N = 1000

Attempt: 1   153 ns
Attempt: 2   45 ns
Attempt: 3   41 ns
Attempt: 4   36 ns
Attempt: 5   35 ns
Attempt: 6   32 ns
Attempt: 7   30 ns
Attempt: 8   33 ns
Attempt: 9   35 ns
Attempt: 10  36 ns
```

```
N = 1000000

Attempt: 1   166 ns
Attempt: 2   50 ns
Attempt: 3   33 ns
Attempt: 4   34 ns
Attempt: 5   31 ns
Attempt: 6   30 ns
Attempt: 7   34 ns
Attempt: 8   32 ns
Attempt: 9   32 ns
Attempt: 10  33 ns
```

What do we look for?

https://godbolt.org/z/155zzfheE

Sum of numbers

- Cycle count => Unchanged
- Latency => Unchanged!
- Instruction count => 15 !! (rdpmc)
- Assembly!

Sum of numbers

```
looper(int):                            # @looper(int)
        test    edi, edi
        jle     .LBB1_1
        lea     eax, [rdi - 1]
        lea     ecx, [rdi - 2]
        imul    rcx, rax
        shr     rcx
        lea     eax, [rdi + rcx]
        dec     eax
        ret
.LBB1_1:
        xor     eax, eax
        ret
main:                                   # @main
```

Looks rather short?

```
(N - 1) * (N - 2) / 2
```

https://godbolt.org/z/155zzfheE

Of course, this is "cheating"

Sum of numbers - -O0, no optimizations

```
N = 1000

Attempt: 1   7778 cycles  2 us
Attempt: 2   7022 cycles  2 us
Attempt: 3   6954 cycles  2 us
Attempt: 4   6690 cycles  2 us
Attempt: 5   6760 cycles  2 us
Attempt: 6   7088 cycles  2 us
Attempt: 7   7132 cycles  2 us
Attempt: 8   7018 cycles  2 us
Attempt: 9   7022 cycles  2 us
Attempt: 10  6762 cycles  2 us

PMC instrs : 10029
```

```
N = 1000000

Attempt: 1   7416834 cycles    2317 us
Attempt: 2   7318012 cycles    2286 us
Attempt: 3   7341468 cycles    2294 us
Attempt: 4   7276734 cycles    2274 us
Attempt: 5   7251226 cycles    2266 us
Attempt: 6   7353562 cycles    2298 us
Attempt: 7   7323466 cycles    2288 us
Attempt: 8   7329502 cycles    2290 us
Attempt: 9   7333450 cycles    2291 us
Attempt: 10  7327512 cycles    2289 us

PMC instrs : 10000034
```

https://godbolt.org/z/155zzfheE

Takeaways from performance profiling

- Microbenchmarks help guide our intuition
    - As opposed to perf and such tools
    - We learn actual numbers - helping make better performance tradeoffs
- Most CPU operations aren't *too* expensive
- Most IO operations *are* expensive

IO latency

- IO (File / network) operations can be 1ms to even start
  - Old school HDDs => 5ms
  - Writes to network => 5-50us
  - SSDs / NVMes => 50us
- Reading random data from memory takes ~100ns
- We don’t notice if the CPU operations take 100x longer

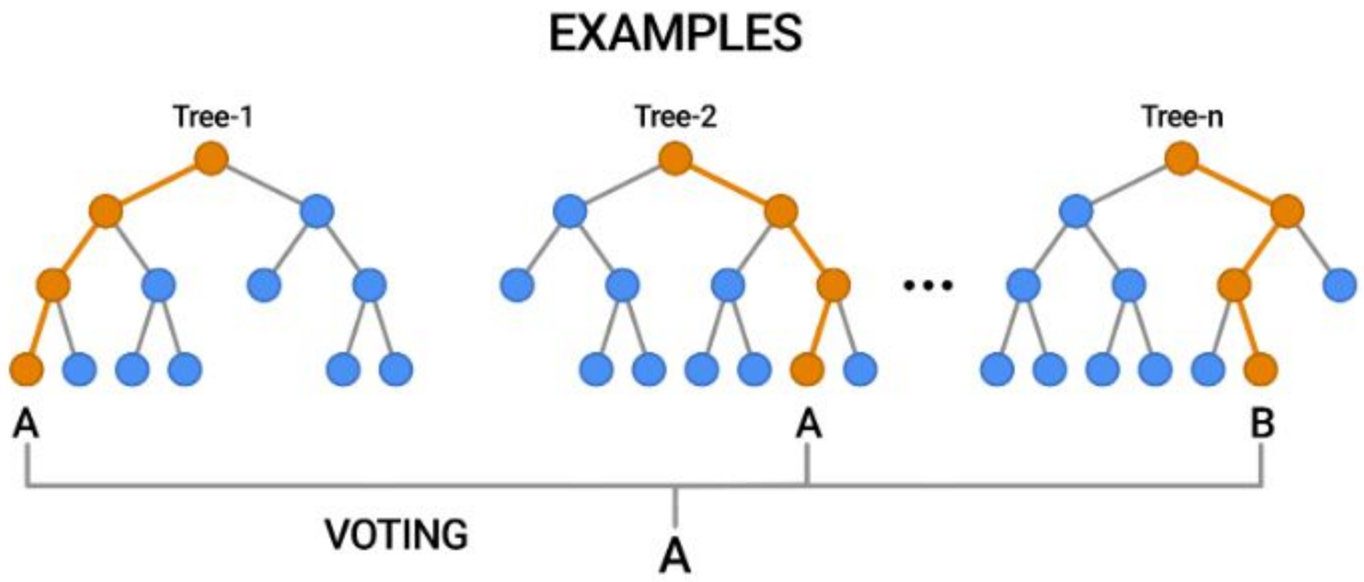
IO

How long does it take to:

- Write 1 GB to a disk
    - Throughputs of ~5 GBps => ~0.2 s
- Send 1 GB over network
    - Throughputs of ~10 Gbps => ~1s
- Read 1 GB from network
    - Throughputs of ~10 Gbps => ~1s

CASE STUDY: XGBOOST

- A highly optimized library written in C++
- Commonly used in ML world for performance
    - Often through Python bindings
    - Input data is usually numpy arrays, thin wrapper around C data



https://blog.tensorflow.org/2021/05/introducing-tensorflow-decision-forests.html

Let's profile!

Methodology

- Train some forests of trees
  - Depth 3, Test data 100 / 1000 rows or evals
- Evaluate them on randomly generated data
- Remove constant effects of going through data
  - Large data
  - Use 1 tree to get the baseline overhead
  - Try 10 - 100 - 1000 trees to understand performance
- Sadly, using Python API
  - Wrapper with C++ profiling utilities we saw before

Xgboost - 100 evals

Num Trees		Instructions per tree?	Tree evals	Total
1		?	1 * 100	100
10		?	10 * 100	1000
100		?	100 * 100	10,000
1000		?	1000 * 100	100,000

```cpp
bst_float PredValue(const SparsePage::Inst &inst,
                    const std::vector<std::unique_ptr<RegTree>> &trees,
                    const std::vector<int> &tree_info, std::int32_t bst_group,
                    RegTree::FVec *p_feats, std::uint32_t tree_begin, std::uint32_t tree_end) {
  bst_float psum = 0.0f;
  p_feats->Fill(inst);
  for (size_t i = tree_begin; i < tree_end; ++i) {
    if (tree_info[i] == bst_group) {
      auto const &tree = *trees[i];
      bool has_categorical = tree.HasCategoricalSplit();
      auto cats = tree.GetCategoriesMatrix();
      bst_node_t nidx = -1;
      if (has_categorical) {
        nidx = GetLeafIndex<true, true>(tree, *p_feats, cats);
      } else {
        nidx = GetLeafIndex<true, false>(tree, *p_feats, cats);
      }
      psum += (*trees[i])[nidx].LeafValue();
    }
  }
  p_feats->Drop();
  return psum;
}
```

Xgboost - 100 evals

Num Trees	Latency	Latency minus fixed (us)	Instructions	Instructions minus fixed
1 (fixed)	1.66ms	0us	9,495,362	0
10	1.67ms	10us	9,557,047	61,685
100	1.74ms	80us	10,060,099	564,737
1000	2.33ms	670us	15,193,188	5,697,826

That’s roughly 20 instructions per tree evaluation

Xgboost - 10,000 evals

Num Trees	Latency	Latency minus fixed (ms)	Instructions	Instructions minus fixed
1 (fixed)	6.27ms	0us	53,291,227	0
10	6.73ms	0.46ms	58,368,335	5,077,108
100	12.94ms	6.21ms	109,338,800	56,047,573
1000	71.25ms	64.98ms	619,118,169	565,826,942

That’s roughly 56 instructions per tree evaluation

Xgboost - 10,000 evals

Num Trees	Latency	Latency minus fixed (ms)	Instructions	Instructions minus fixed
1 (fixed)	6.27ms	0us	53,291,227	0
10	6.73ms	0.46ms	58,368,335	5,077,108
100	12.94ms	6.21ms	109,338,800	56,047,573
1000	71.25ms	64.98ms	619,118,169	565,826,942

Or 0.1ns per instruction, the magic of pipelining!

Code design and latency

Code design and latency

- We know various latencies now
- How does this map back to real code?
- Real code has:
  - Functions
  - Classes
  - Structs
  - Conditions
  - Math

Does my code design impact latency?

- If else branches
- Normal function calls
- Function pointer calls
- Virtual functions
- Templated functions

Does my code design impact latency?

- If else branches
- Normal function calls
- Function pointer calls
- Virtual functions
- Templated functions

If else branches

- Not a major concern unless you're doing HFT
- Wastes some instructions prefetched into the cache
- Branch predictor helps in repeat cases but best to avoid
- Hard to assign a number, but main issue is cache

If else branches

- Output a grid delimited by “*” on all ends
- What’s wrong with this program?

Branch in the hot path

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    if (i == 0 || j == 0 ||
          i == N - 1 || j == M - 1) {
      dest[i][j] = '*';
    } else {
      dest[i][j] = ' ';
    }
  }
}
```

If else branches

```
for (int i = 0; i < N; i++) {
   if (i == 0 || i == N - 1) {
      for (int j = 0; j < M; j++)
         dest[i][j] = '*';
   } else {
      dest[i][0] = '*'; dest[i][M - 1] = '*';
      for (int j = 1; j < M - 1; j++)
         dest[i][j] = ' ';
   }
}
```

If else branches

```
for (int j = 0; j < M; j++) {
    dest[0][j] = '*';
    dest[N - 1][j] = '*';
}
for (int i = 1; i < N - 1; i++) {
    dest[i][0] = '*';
    dest[i][M - 1] = '*';
    for (int j = 1; j < M - 1; j++) {
        dest[i][j] = ' ';
    }
}
```

Huge perf difference - Old Code

```
Attempt: 1   1988379 cycles   795 us
Attempt: 2   2004067 cycles   802 us
Attempt: 3   2004488 cycles   801 us
Attempt: 4   2004300 cycles   801 us
Attempt: 5   2002897 cycles   801 us
Attempt: 6   1999758 cycles   800 us
Attempt: 7   1986691 cycles   794 us
Attempt: 8   2000182 cycles   800 us
Attempt: 9   2003578 cycles   801 us
Attempt: 10  2002415 cycles   801 us

Instrs 8002545, branches 504504, branch misses 1001
```

Huge perf difference - new Code

```
Attempt: 1   1012412 cycles   405 us
Attempt: 2   1012312 cycles   405 us
Attempt: 3   1032924 cycles   413 us
Attempt: 4   1017076 cycles   407 us
Attempt: 5   1029133 cycles   411 us
Attempt: 6   1017403 cycles   407 us
Attempt: 7   1027091 cycles   411 us
Attempt: 8   1017133 cycles   407 us
Attempt: 9   1013870 cycles   405 us
Attempt: 10  1032361 cycles   413 us

Instrs 2767017, branches 254998, branch misses 1008
```

If else branches

- Simple lossless transformation
- Way fewer branches
- Slight hit to instruction cache though
- Might be a hit to code readability
- Absolute cost:
  - 1 cycle or so?
  - Worse if we hit a cache miss

Does my code design impact latency?

- If else branches
- Normal function calls
- Function pointer calls
- Virtual functions
- Templated functions

Normal function calls

- Go to a different place suddenly
- Instruction cache misses
- Address is known at compile time
  - Compiled into the instr

```
__attribute__((noinline)) void myfunc;

void task() {
  myfunc();
  // other stuff
}
```

Normal function calls

- Absolute cost:
    - Instruction pipeline flush
    - Cache miss risk

```
__attribute__((noinline)) void myfunc;

void task() {
    myfunc();
    // other stuff
}
```

Does my code design impact latency?

- If else branches
- Normal function calls
- Function pointer calls
- Virtual functions
- Templated functions

Function pointer calls

- Go to a different place suddenly
- Instruction cache misses

And.....

```
using FuncT = void (*)();
FuncT* myfunc;

void task() {
    myfunc();
    // other stuff
}
```

Function pointer calls

- Have to go to pointer location to read the next location
- More load on data cache

```
using FuncT = void (*)();
FuncT* myfunc;

void task() {
    myfunc();
    // other stuff
}
```

Does my code design impact latency?

- If else branches
- Normal function calls
- Function pointer calls
- Virtual functions
- Templated functions

Virtual function calls

- Go to a different place suddenly
- Instruction cache misses

And.....

```cpp
class Interface {
  public:
   virtual void myfunc() = 0;
};

void task(Interface* interface) {
   interface->myfunc();
   // other stuff
}
```

Aside: VTable

```
struct InterfaceVTable {
   void (*myfunc)(void*);
};
```

```
class MyInterface {
   InterfaceVTable* vtable;
  public:
   void myfunc() {
      vtable->myfunc(this);
   }
};
```

Virtual function calls

- Go to a different place suddenly
- Instruction cache misses
- One extra data cache lookup (potential miss)

```cpp
class Interface {
  public:
   virtual void myfunc() = 0;
};

void task(Interface* interface) {
   interface->myfunc();
   // other stuff
}
```

Does my code design impact latency?

- If else branches
- Normal function calls
- Function pointer calls
- Virtual functions
- Templated functions

Templated rewrite of virtual code

```cpp
class Interface {
  public:
   virtual void myfunc() = 0;
};

void task(Interface* obj) {
   obj->myfunc();
   // other stuff
}
```

```cpp
template <typename T>
concept HasMyFunc = requires(T obj) {
   { obj.myfunc() } -> same_as<void>;
};

template <HasMyFunc T>
void task(T* obj) {
   obj->myfunc();
   // other stuff
}
```

Templated functions and classes

Templates are just one way to “optimize”:

- ✅ Avoid indirections
- ✅ Avoid unnecessary branching
- 🚫 Reuse instruction cache as much as possible

No good way to do all three :)

Templated functions and classes

- Good way to remove most runtime indirection overhead
  - Branches, virtual jumps, ...
- Bad way if the same object isn’t reused each time
  - Instruction cache pollution for each unique type
- Bad for compile times and binary sizes
- Cost?
  - Just instruction cache hits

Principles

Principles to live (write) by

- Focus your energy on the hot path
- Prematurely-plan, not optimize
- Design APIs that encourage performance
- Measure! Profile!
    - Until it becomes obvious

Principles to live (write) by

- Focus your energy on the hot path
- Prematurely-plan, not optimize
- Design APIs that encourage performance
- Measure! Profile!
  - Until it becomes obvious

The hot path

- Only relevant if your program is not throughput heavy
- Find the important part of your code
  - When the trading algorithm wants to send an order
  - When the user requests some information
- Find the less-important part of your code
  - Re-sharding some internal memory data structure
  - Book-keeping after the important work has been done

The hot path

- Optimize the hot path
  - Don’t blow out your cache on the slow path
- What breaks?
  - Profile-guided-optimizations (PGO) don’t get this
  - The CPU’s cache doesn’t get it
- What fixes this?
  - Careful analysis informed by latency numbers
  - Profile! Profile! Profile!

Principles to live (write) by

- Focus your energy on the hot path
- Prematurely-plan, not optimize
- Design APIs that encourage performance
- Measure! Profile!
  - Until it becomes obvious

Premature optimization

Premature optimization is the root of all evil

-- Donald Knuth

Premature optimization

- Two way doors
    - Decisions with few external ramifications
    - Impact touches internals of some well contained module
- One way doors
    - Decisions which are hard to undo
    - Impact touches a broad API surface

Examples

API Case study 1

```
struct MyNetworkStruct {
    bool is_ready;
    double price;
    int size;
};
```

- Writing a multi-process communication system
- Sending this struct over the network
- What’s wrong here?

API Case study 1

```
struct MyNetworkStruct {
    bool is_ready;
    double price;
    int size;
};
```

- Struct byte padding
- Once released, we cannot undo without upgrading every software

API Case study 1

```
struct MyNetworkStruct {
   double price;
   int size;
   bool is_ready;
} __attribute__((packed));

static_assert(
  sizeof(MyNetworkStruct) == 13
);
```

One way door!

Optimize prematurely

API Case study 2

```
int get_or_set_default(
     std::map<int, int>& m,
     int key,
     int default_value
) {
   if (m.find(key) == m.end()) {
       m[key] = default_value;
   }
   return m[key];
}
```

Do we know what's inefficient here?

API Case study 2

```
int get_or_set_default(
     std::map<int, int>& m,
     int key,
     int default_value
) {
   if (m.find(key) == m.end()) {
       m[key] = default_value;
   }
   return m[key];
}
```

- Wasteful std::map lookup
- Could be optimized under the hood later
- Although perhaps after this talk....

API Case study 2

```cpp
int get_or_set_default(
     std::map<int, int>& m,
     int key,
     int default_value
) {
   auto [it, inserted] =
      m.emplace(key, default_value);
   return it->second;
}
```

Better :)

And while we're at it... Godbolt

API Case study 2 - The old code

```
get_or_set_default(std::map<int, int, std::less<int>, std:::
        push    rbp
        push    rbx
        push    rax
        mov     ebp, edx
        mov     rbx, rdi
        mov     dword ptr [rsp + 4], esi
        mov     rax, qword ptr [rdi + 16]
        test    rax, rax
        je      .LBB0_5
        lea     rdx, [rbx + 8]
        mov     rcx, rdx
.LBB0_2:                                # %while.body.i.i.i
        xor     edi, edi
        cmp     dword ptr [rax + 32], esi
        setl    dil
        cmovge  rcx, rax
        mov     rax, qword ptr [rax + 8*rdi + 16]
```

```
        test    rax, rax
        jne     .LBB0_2
        cmp     rcx, rdx
        je      .LBB0_5
        cmp     dword ptr [rcx + 32], esi
        jle     .LBB0_6
.LBB0_5:                                # %if.then
        lea     rsi, [rsp + 4]
        mov     rdi, rbx
        call    std::map<int, int, std::less<int>,
        mov     dword ptr [rax], ebp
.LBB0_6:                                # %if.end
        lea     rsi, [rsp + 4]
        mov     rdi, rbx
        call    std::map<int, int, std::less<int>,
        mov     eax, dword ptr [rax]
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
```

API Case study 2 - The new code

```
get_or_set_default2(std::map<int, int, st
        push    rax
        mov     dword ptr [rsp + 4], esi
        mov     dword ptr [rsp], edx
        lea     rsi, [rsp + 4]
        mov     rdx, rsp
        call    std::pair<std::_Rb_tree_i
        mov     eax, dword ptr [rax + 36]
        pop     rcx
        ret
```

Principles to live (write) by

- Focus your energy on the hot path
- Prematurely-plan, not optimize
- Design APIs that encourage performance
- Measure! Profile!
    - Until it becomes obvious

Design APIs that encourage performance

- Think like the user and plan
- Virtual function interface vs templated

```cpp
void task(Interface* obj) {
    obj->myfunc();
    // other stuff
}
```

```cpp
template <HasMyFunc T>
void task(T* obj) {
    obj->myfunc();
    // other stuff
}
```

Design APIs that encourage performance

- Think like the user and plan
- Virtual function interface vs templated
- Help user avoid wasteful expensive operations

```cpp
if (m.find(key) == m.end()) {
    m[key] = default_value;
}
return m[key];
```

```cpp
auto [it, inserted] =
   m.emplace(key, default_value);
return it->second;
```

Principles to live (write) by

- Focus your energy on the hot path
- Prematurely-plan, not optimize
- Design APIs that encourage performance
- Measure! Profile!
  - Until it becomes obvious

Measure! Profile!

- The more we measure, the better we get at it
- Look up “Agner Fog instruction tables”
  - https://www.agner.org/optimize/instruction_tables.pdf

Measure! Profile!

- The more we measure, the better we get at it
- Look up "Agner Fog instruction tables"
  - https://www.agner.org/optimize/instruction_tables.pdf

IMO, the real trick behind writing fast code like a native

Summary

Summary

- Know how long common CPU operations take
  - Memory, cache, arithmetic, jumps, branches
- Understand what code your compiler might generate
  - Virtual functions, structs, ...
- Microbenchmark to improve your intuition
- Prematurely optimize the one-way trapdoor decisions
- Go fast...

Thank you!