C++ now 2025

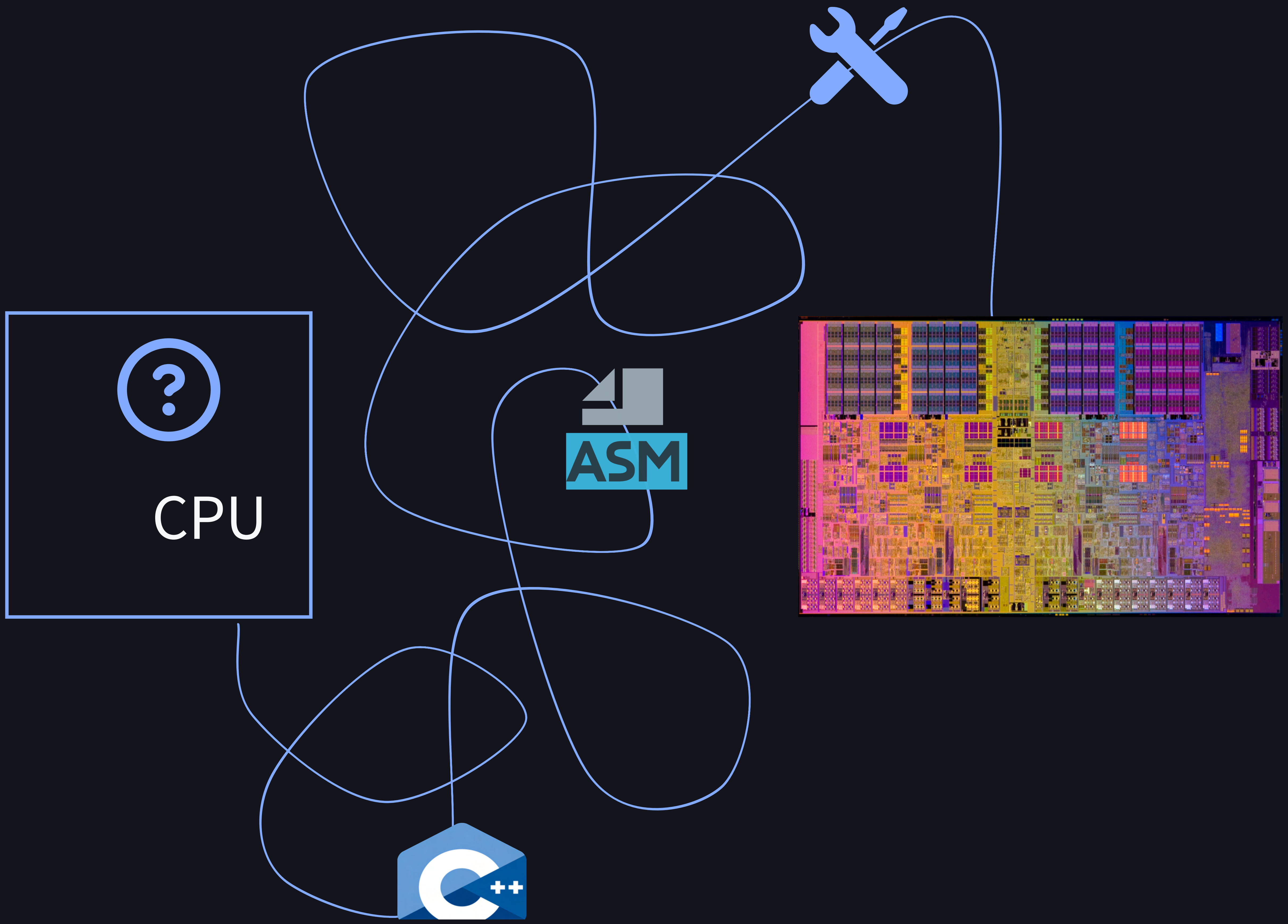
C++ as a Microscope Into Hardware

Linus Boehm

↔ $

ENGINEERS GATE

+ x86-64


CPU
ASM

```cpp
int fun() {
  return 0;
}
```

```bash
g++ -g -c -O1 return_zero.cpp # -g:debug info, -c:no main, -O1:optimize
less return_zero.o
```

```
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@<80>^E^@^@^@^@^@^@^@^@^@^@@^@^@^@
^E^@^@^@^@O^@^@^@^@^@^@^@^@^@^@^F^@^@^@^@^@^@^@^A<9C>^C^D^Eint^@^@^A^Q^A%^N^S^K^C^NESC^N^Q^A^R^G^P^W^@^@^B.
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@F^@^@^@^B^@&^@^@^@^A^A<FB>^N^M^@^A^A^A^A^@^@^@^A^@^@^A^@return_zero.cpp^@^@^@
-march=x86-64 -g -O1^@_Z3funv^@/home/lboehm/repos/talks/2025/cppnow/code^@^@GCC: (GNU) 8.5.0 20210514 (Red Ha
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@<F1><FF>^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^A^@^@^
^@^C^@^D^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^F^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^G^@^@^
^@^C^@^M^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^N^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^L^@^@^
^@^@
^@^@^@^F^@^@^@^@^@^@^@^@^@^@^L^@^@^@^@^@^@^@
```

```cpp
int fun() {
  return 0;
}
```

```bash
g++ -g -c -O1 return_zero.cpp # -g:debug info, -c:no main, -O1:optimize
xxd -b return_zero.o          # -b:binary dump
```

```
00000000: 01111111 01000101 01001100 01000110 00000010 00000001  .ELF..
00000006: 00000001 00000000 00000000 00000000 00000000 00000000  ......
0000000c: 00000000 00000000 00000000 00000000 00000001 00000000  ......
00000012: 00111110 00000000 00000001 00000000 00000000 00000000  >.....
00000018: 00000000 00000000 00000000 00000000 00000000 00000000  ......
0000001e: 00000000 00000000 00000000 00000000 00000000 00000000  ......
00000024: 00000000 00000000 00000000 00000000 10000000 00000101  ......
0000002a: 00000000 00000000 00000000 00000000 00000000 00000000  ......
00000030: 00000000 00000000 00000000 00000000 01000000 00000000  ....@.
```

fetch → decode → execute

```cpp
int fun() {
  return 0;
}
```

```bash
g++ -g -c -O1 return_zero.cpp # -g:debug info, -c:no main, -O1:optimize
objdump -d -C -Mintel return_zero.o # -d:disassemble,-Mintel:intel syntax
```

```
return_zero.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <fun()>:
   0:   b8 00 00 00 00          mov    eax,0x0
   5:   c3                      ret
```

fetch → decode → execute

Registers / Instructions

```cpp
int many_args(int, int, int, int,
              int, int, int num7) {
  return num7;
}
```

```
//////////// -O1
  <many_args(int, int, int, int, int, int, int)>:
   8b 44 24 08          mov     eax,DWORD PTR [rsp+0x8]
   c3                   ret
```

- Variable instruction length (Combined Intel 64 Manuals)
- `eax` / `rax` : return register (32/64 bit)
- Volatile/caller owned
- `rdi` , `rsi` , … : function arguments
- Calling convention: System V AMD64 ABI
- Memory operands `mov    eax,DWORD PTR [rsp+0x8]`

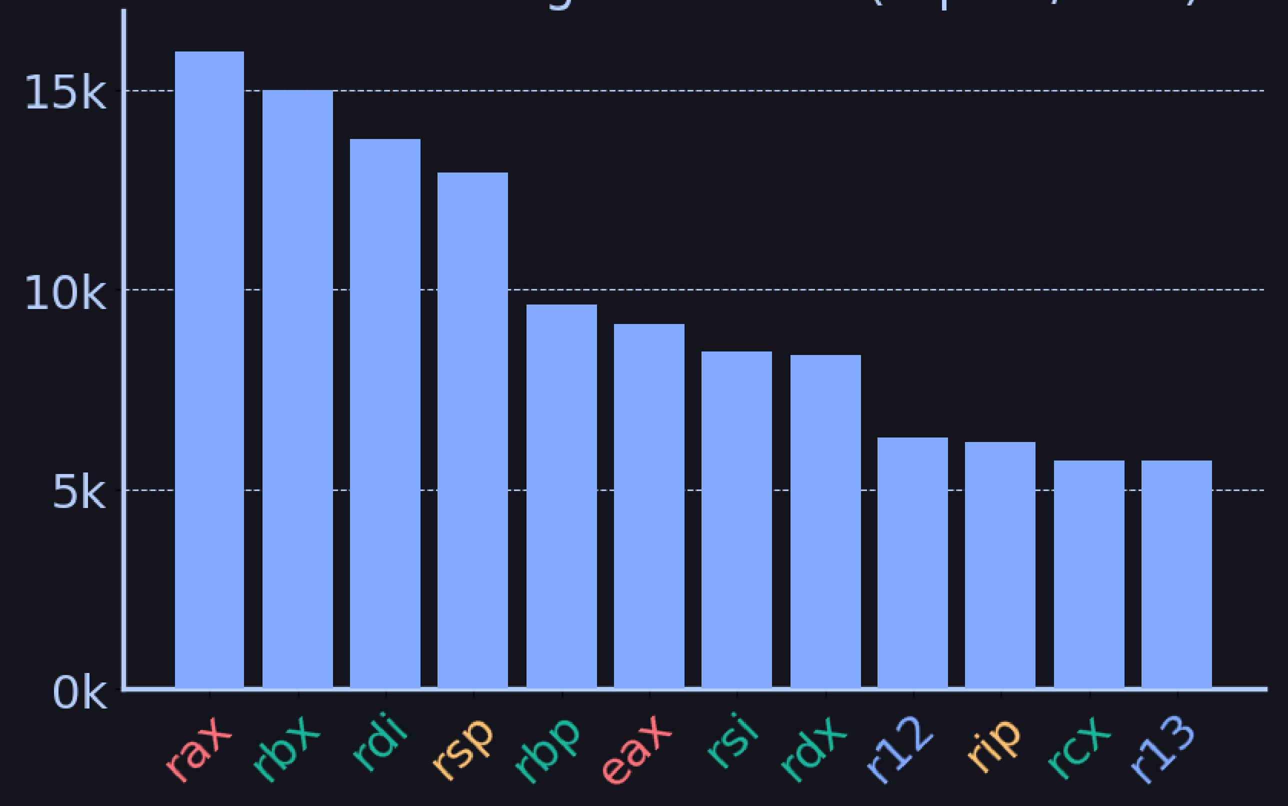
- Questions
  - How many instructions/registers are there?
  - What are common ones?

`/usr/bin/gcc` registers

```
# disassemble gcc
objdump -d /usr/bin/gcc
```

total registers: 186k

Cumulative register count (top 12/62%)



rax — 64 bit

eax — 32 bit

ax — 16 bit

ah al — 8 bit

rax — Return

rbp — Base pointer

rip — Instruction pointer

rsp — Stack pointer

rflags — Flags register (Carry, sign, ...)

Function calling convention

rdi rsi rdx rcx r8 r9

Scratch/volatile/callee owned

rax rcx rdi rdx rsi r8 r9 r10 r11

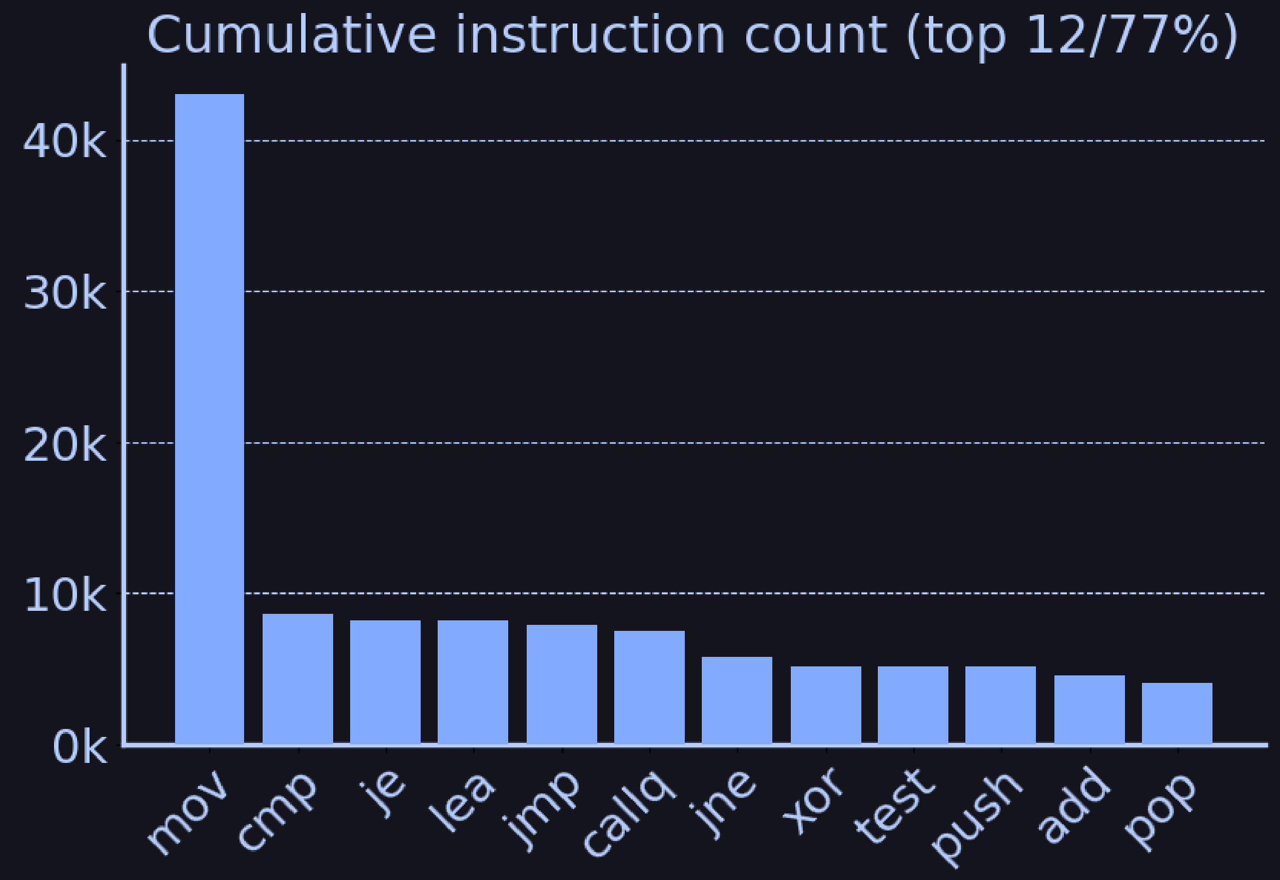
Caller owned (restored by callee)

rbx rbp r12 r13 r14 r15

`/usr/bin/gcc` instructions

```
# disassemble gcc
objdump -d /usr/bin/gcc
```

```
total instructions:    146k
distinct instructions: 161
```



- `mov` : copy data (e.g. into registers for processing)
- `cmp` : compare numbers (if statements, loops)
- `je` : jump if previous comp values were equal (loops)
- `lea` : fancy way of doing arithmetic (adding)
- `callq` : function call
- `jmpq` : unconditional jump
- `jne` : jump not equal
- `xor` : xor (e.g. to null out registers)
- `test` : ANDs operands and updates flags
- `push` : pushes value on stack, stack management
- `add` : adds two values
- `pop` : pops value from stack, stack management

> the x86 instruction set is […] overcomplicated, and **redundantly redundant**.
>
> […] it remains Turing-complete when reduced to just **one instruction**.
>
> — Stephen Dolan, University of Cambridge, 2013 — [`mov` is Turing-complete]

movfuscator

find_primes.c:

```c
#include <stdio.h>

#define size 8190
#define sizepl 8191
char flags[sizepl];

int main() {
  int i, prime, k, count, iter;
  for (iter = 1; iter <= 100; iter++) {
    count = 0;
    for (i = 0; i <= size; i++)
      flags[i] = 1;
    for (i = 0; i <= size; i++) {
      if (flags[i]) {
        prime = i + i + 3;
        k = i + prime;
        while (k <= size) {
          flags[k] = 0;
          k += prime;
        }
        count = count + 1;
      }
    }
  }
  printf("found %d primes", count);
}
```

https://github.com/xoreaxeaxeax/movfuscator

https://en.wikipedia.org/wiki/Byte_Sieve

movfuscator

```
movcc ./find_primes.c -o find_primes_mov
objdump -Mintel -d ./find_primes_mov
```

```
mov     eax,DWORD PTR [ecx*4+0x83f8160]
mov     edx,DWORD PTR ds:0x85f8194
mov     DWORD PTR [eax],edx
mov     DWORD PTR ds:0x83f8164,0x804f048
mov     eax,DWORD PTR [ecx*4+0x83f8160]
mov     edx,DWORD PTR ds:0x85f8198
mov     DWORD PTR [eax],edx
mov     DWORD PTR ds:0x83f8164,0x804f050
mov     eax,DWORD PTR [ecx*4+0x83f8160]
mov     edx,DWORD PTR ds:0x85f819c
mov     DWORD PTR [eax],edx
mov     edx,DWORD PTR ds:0x85f81a0
mov     DWORD PTR [eax+0x4],edx
mov     DWORD PTR ds:0x83f8164,0x804f058
mov     eax,DWORD PTR [ecx*4+0x83f8160]
mov     edx,DWORD PTR ds:0x85f81a4
```

```
gcc ./find_primes.c -o find_primes_gcc
objdump -Mintel -d ./find_primes_gcc
```

```
.LC0:
        .string "found %d primes"
main:
        sub     rsp, 8
        mov     r9d, 100
        mov     edi, OFFSET FLAT:flags+8191
        jmp     .L2
.L5:
        add     r8d, 1
.L4:
        add     rsi, 1
        add     rcx, 3
        add     rdx, 2
        cmp     rcx, 24576
        je      .L13
.L7:
```

movfuscator

```
movcc ./find_primes.c -o find_primes_mov
objdump -Mintel -d ./find_primes_mov
```

```
000  mov    eax,DWORD PTR [ecx*4+0x83f8160]
001  mov    edx,DWORD PTR ds:0x85f8194
002  mov    DWORD PTR [eax],edx
003  mov    DWORD PTR ds:0x83f8164,0x804f048
004  mov    eax,DWORD PTR [ecx*4+0x83f8160]
005  mov    edx,DWORD PTR ds:0x85f8198
006  mov    DWORD PTR [eax],edx
007  mov    DWORD PTR ds:0x83f8164,0x804f050
008  mov    eax,DWORD PTR [ecx*4+0x83f8160]
009  mov    edx,DWORD PTR ds:0x85f819c
010  mov    DWORD PTR [eax],edx
011  mov    edx,DWORD PTR ds:0x85f81a0
012  mov    DWORD PTR [eax+0x4],edx
013  mov    DWORD PTR ds:0x83f8164,0x804f058
014  mov    eax,DWORD PTR [ecx*4+0x83f8160]
015  mov    edx,DWORD PTR ds:0x85f81a4
```

fetch → decode → execute

Toolbox Digression #1

- `objdump`: disassemble binary into assembly
- `perf` (`perf list` for available counters)

```
perf stat -e duration_time,cpu-migrations:u,instructions,cycles:u ls
```

```
 Performance counter stats for 'ls':

           1490843 ns   duration_time:u
                 0      cpu-migrations:u
           1031098      instructions:u            #    1.06  insn per cycle
            971675      cycles:u

       0.001490843 seconds time elapsed

       0.000000000 seconds user
       0.001473000 seconds sys
```

movfuscator — `perf`

```
$ perf stat -e instructions,duration_time ./find_primes_mov

 Performance counter stats for './find_primes_mov':

     8,896,386,311      instructions:u
     6,819,887,233 ns   duration_time:u
```

F.A.Q.

- Q: Why did you make this? A: I thought it would be funny.

- `find_primes.c`: `movcc` version uses **800x** instructions; **2,500x** slower than `gcc` version.
- The mov-only DOOM renders approximately one frame every **7 hours**.

Recap — Binary

- ELF file spec
- Instruction encoding
- Calling convention

- Tools: `objdump`, `perf`

- Instructions
- Registers

- `mov       eax, DWORD PTR [rsp+8]` (relative to stack pointer)
- `mov       eax, DWORD PTR [rsi]` (any memory address)

Memory Layout

```cpp
#include <algorithm>
#include <string>
#include <vector>

int global = 10;
void foo() {}

int main(int argc, char** argv) {
  int stack1;
  int stack2;
  int* heap1 = new int();
  int* heap2 = new int();

  std::vector<std::pair<std::string, void*>> items;
  items.emplace_back("globals", reinterpret_cast<void*>(&global));
  items.emplace_back("functions", reinterpret_cast<void*>(&foo));

  items.emplace_back("stack1", reinterpret_cast<void*>(&stack1));
  items.emplace_back("stack2", reinterpret_cast<void*>(&stack2));
  items.emplace_back("heap1", reinterpret_cast<void*>(heap1));
  items.emplace_back("heap2", reinterpret_cast<void*>(heap2));

  items.emplace_back("command line args", reinterpret_cast<void*>(argv));
  items.emplace_back("environment variables", getenv("HOME"));

  std::ranges::sort(items, std::greater<>{},
                    [](const auto& p) { return p.second; });
  for (const auto& [name, ptr] : items) {
    printf("%-25s %p\n", name.c_str(), ptr);
  }
}
```

```
environment variables    0x7ffc56785fe8
command line args        0x7ffc567844b8
stack1                   0x7ffc5678434c
stack2                   0x7ffc56784348
heap2                    0x18f5e2d0
heap1                    0x18f5e2b0
globals                  0x406058
functions                0x4011a6
```

Questions

- What if two programs use the same memory address?
- Where do heap and stack meet?

Memory — addresses

```cpp
#include <stdio.h>
#include <atomic>
#include <chrono>
#include <thread>

std::atomic<int> num = 40;

int main() {
  ++num;
  std::this_thread::sleep_for(
    std::chrono::seconds(1));
  printf("addr: %p, val: %d\n",
         &num, num);
}
```

```
./globals_loc & ./globals_loc
```

```
addr: 0x602034, val: 41
addr: 0x602034, val: 41
```

- Memory addresses aren't real
- OS maintains table for address translation
- TLB (translation lookaside buffer) caches recent translations

Memory — Stack

```cpp
#include <iostream>

static constexpr auto ONE_MiB = (1U << 20U);
struct OneMiB { char data[ONE_MiB]; };

void measureStackSize(OneMiB* first_addr) {
  OneMiB local_var;
  const auto diff = first_addr - &local_var;
  std::cout << "stack size: " << diff
    << " MiB" << std::endl;
  measureStackSize(first_addr);  // recurse
}

int main() {
  OneMiB first;
  measureStackSize(&first);
  return 0;
}
```

```
stack size: 1 MiB
stack size: 2 MiB
stack size: 3 MiB
stack size: 4 MiB
stack size: 5 MiB
stack size: 6 MiB
Segmentation fault (core dumped)
```

Memory — Heap

```cpp
#include <cassert>
#include <iostream>

static constexpr auto ONE_GiB = (1U << 30U);
struct OneGiB { char data[ONE_GiB]; };

int main() {
  OneGiB *curr_addr;

  for (int cnt = 0;; ++cnt) {
    try {
      curr_addr = new OneGiB;
    } catch (const std::bad_alloc &e) {
      std::cout << "malloc failed after: "
                << cnt << " GiB" << std::endl;
      break;
    }
  }
}
```

```
malloc failed after: 131070 GiB
```

`htop`:

CPU	VIRT	CPU%	MEM%	TIME+	Command
3	127T	0.0	0.4	0:00.98	./heap_limit

Question

- What happens when we use the memory?

Memory — Mapping time + perf stat

```cpp
static constexpr auto ONE_GiB = (1U << 30U);
struct OneGiB { char data[ONE_GiB]; };

int main() {
  OneGiB* curr_addr;

  for (int cnt = 0; cnt < 30; ++cnt) {
    curr_addr = new OneGiB;
  }
}
```

```
g++ -O3 heap_no_write.cpp -o heap_no_write
sudo perf stat -e duration_time ./heap_no_write
```

```
0.001664263 seconds time elapsed
```

```
sudo perf stat -e page-faults ./heap_no_write
```

```
126          page-faults
```

```cpp
static constexpr auto ONE_GiB = (1U << 30U);
struct OneGiB { char data[ONE_GiB]; };

int main() {
  OneGiB* curr_addr;

  for (int cnt = 0; cnt < 30; ++cnt) {
    curr_addr = new OneGiB{};  // value init
  }
}
```

```
g++ -O3 heap_write.cpp -o heap_write
sudo perf stat -e duration_time ./heap_write
```

```
19.227826577 seconds time elapsed
```

```
sudo perf stat -e page-faults ./heap_write
```

```
7.864.447          page-faults
```

Toolbox Digression #2

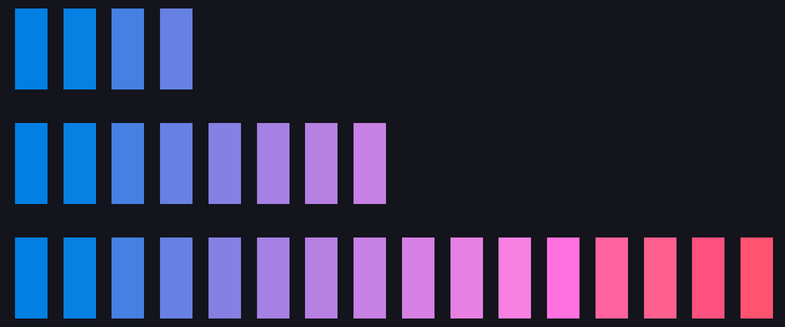
- `tsc` — time stamp counter

```cpp
struct ProfileScope {
  ProfileScope() {
    tsc_start_ = __rdtsc();
  }
  ~ProfileScope() {
    const uint64_t elapsed = __rdtsc() - tsc_start_;
    std::cout << elapssed << std::endl;
  }
  uint64_t tsc_start_;
};
```

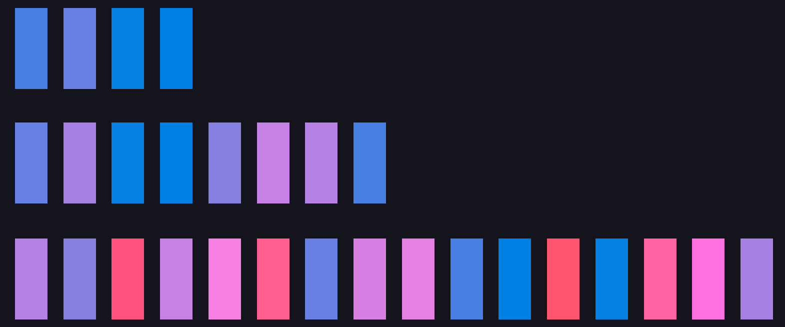
[Memory — Demo]

Memory — measurment patterns

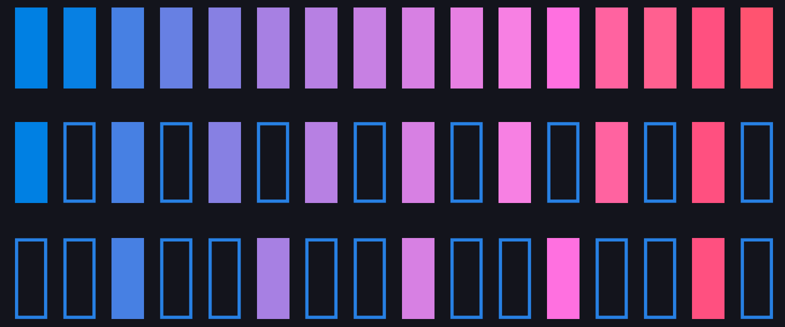
Sequential, `f(range)`:



Random, `f(range)`:



Stride, `f(stride)`:



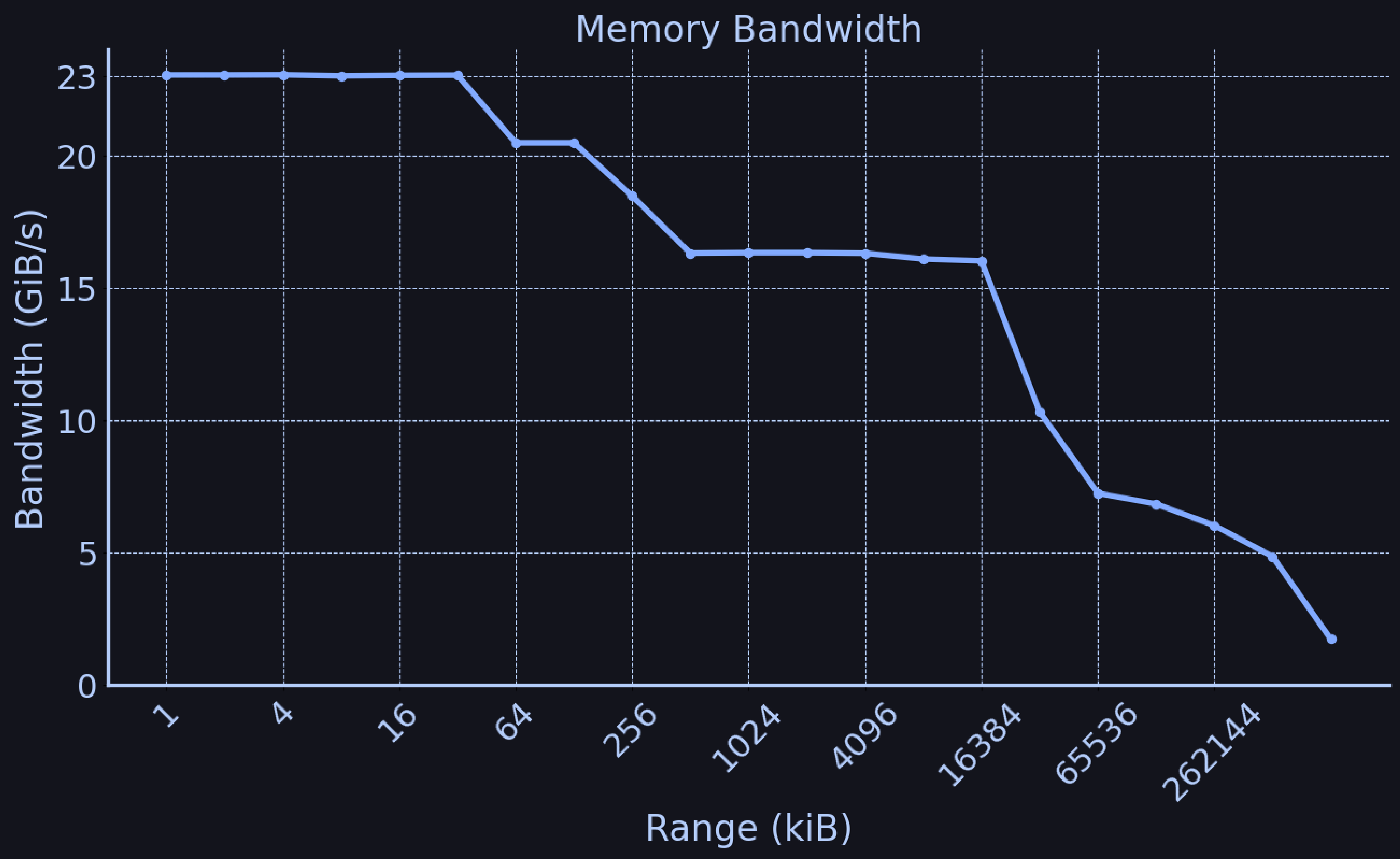
Toolbox Digression #3

- Google Benchmark
  - Benchmarking library
  - Statistical analysis

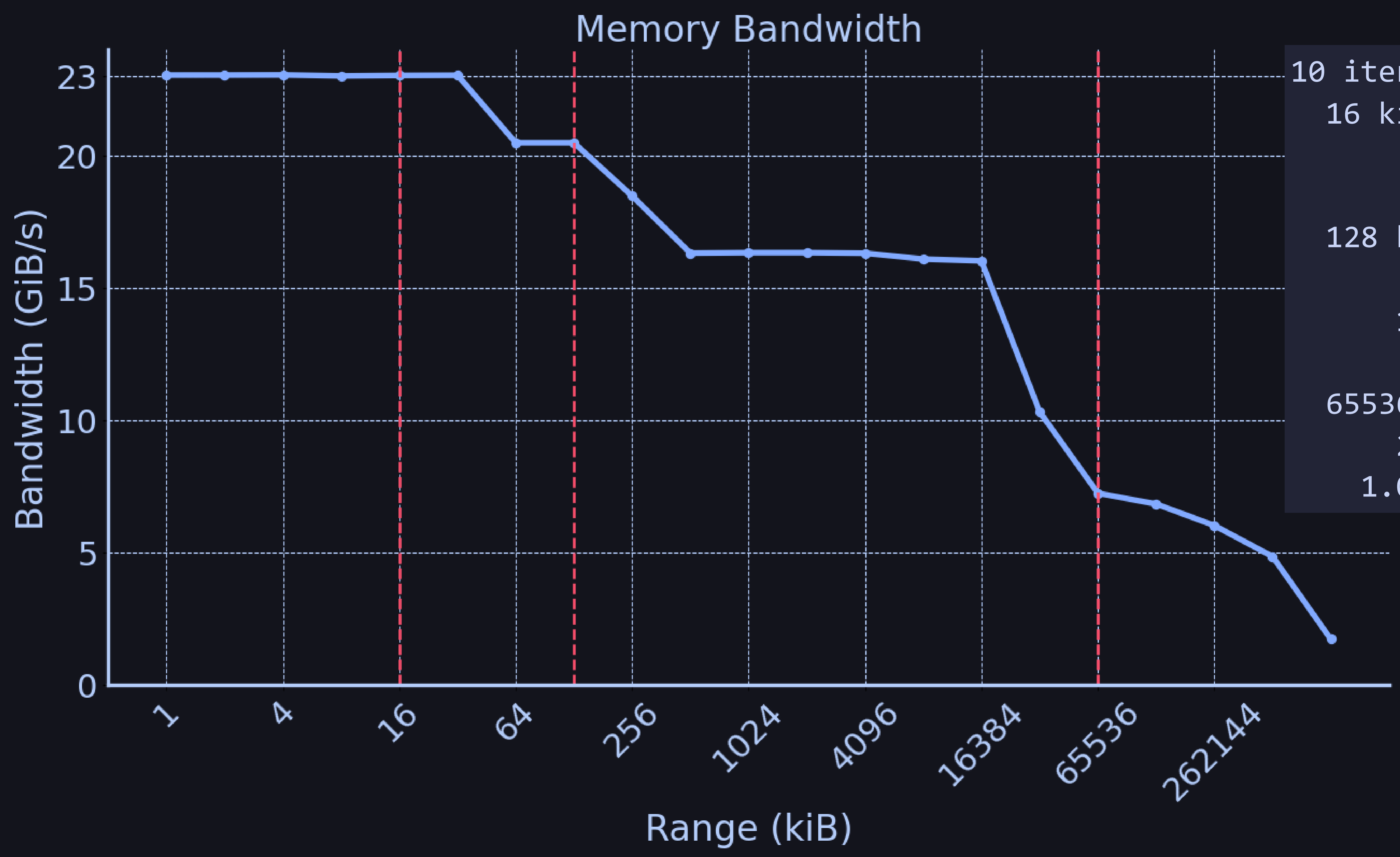
```cpp
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
  for (auto _ : state) {
    // test code
  }
}
BENCHMARK(BM_SomeFunction);  // register test
BENCHMARK_MAIN();
```

Memory — sequential write [DEMO]

Memory — sequential write

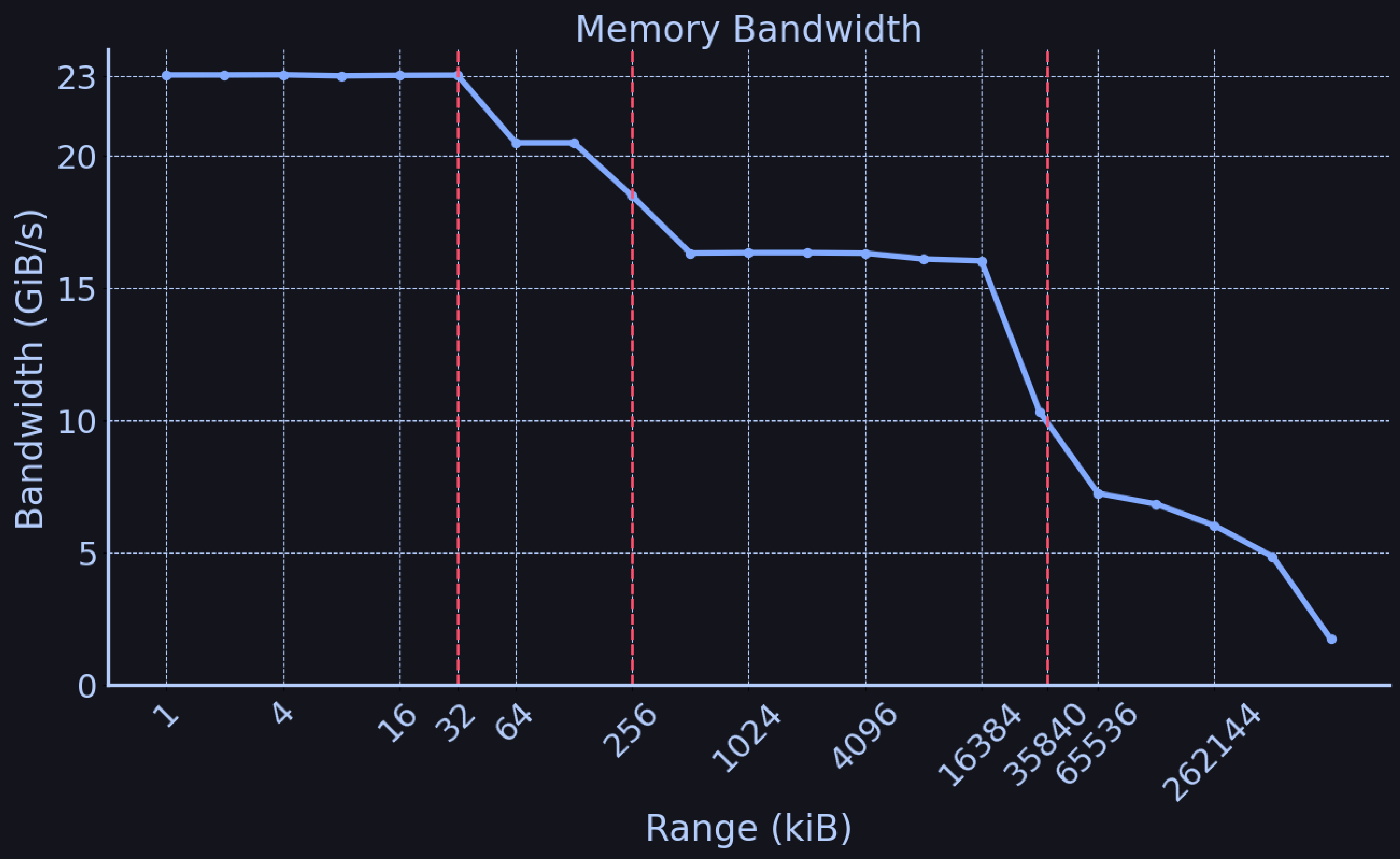


Memory — sequential write

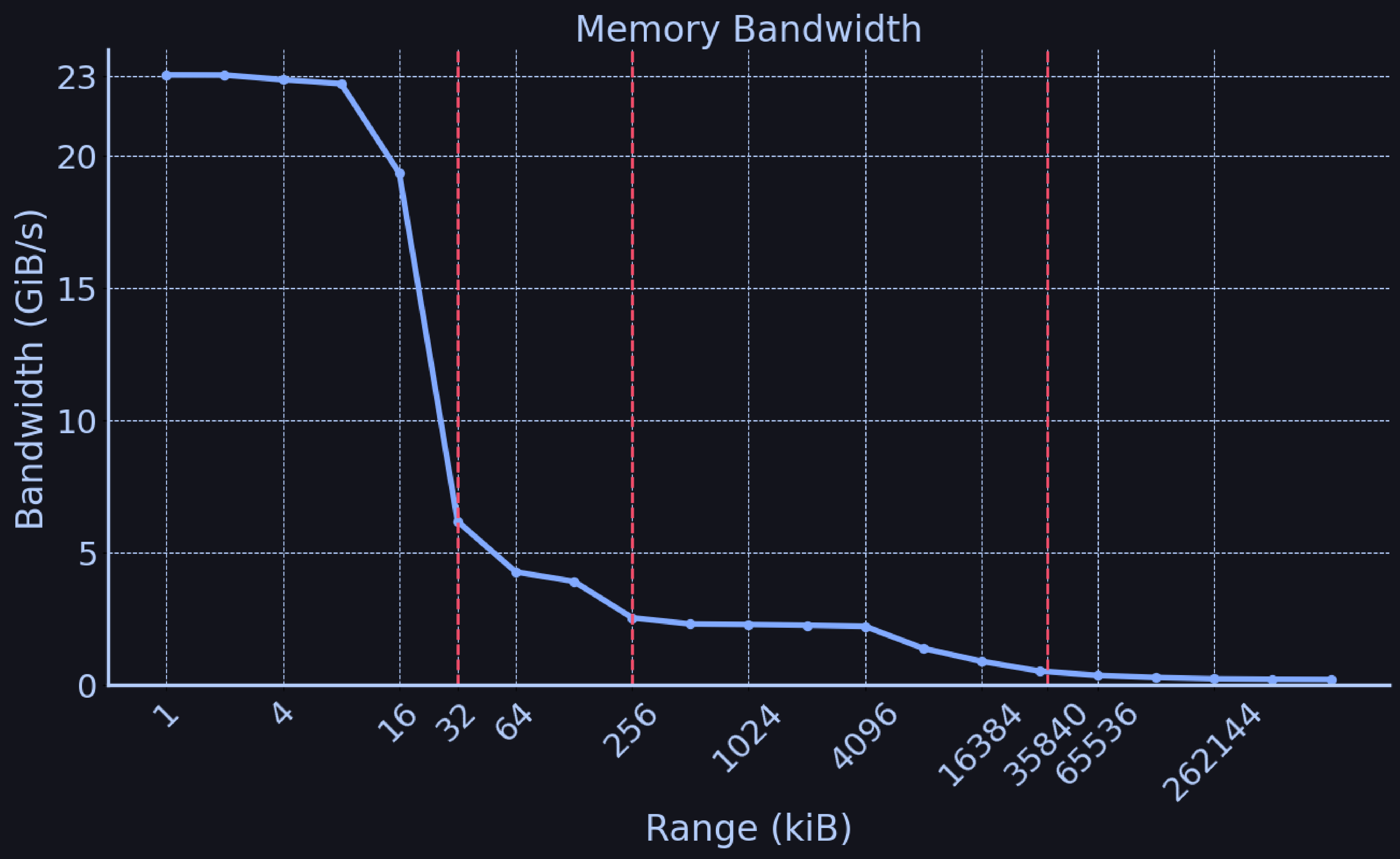



```
10 iterations:
  16 kiB
              2.852 cache-misses:u
             53.946 L1-dcache-load-misses:u
  128 kiB:
              2.341 cache-misses:u
        167.904.734 L1-dcache-load-misses:u

  65536 kiB:
        294.949.282 cache-misses:u
      1.049.649.135 L1-dcache-load-misses:u
```

Memory — sequential write

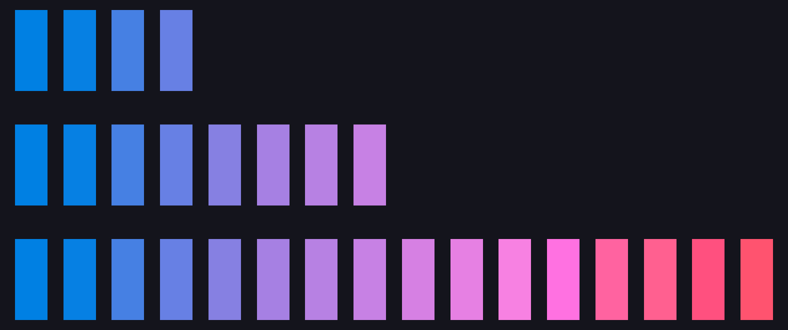


Memory — random write

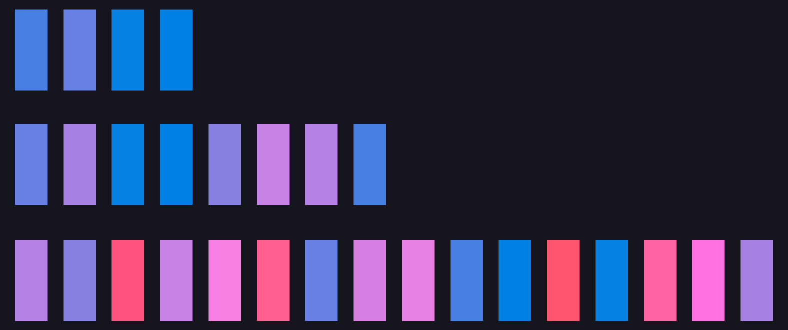


Cache — strided copy

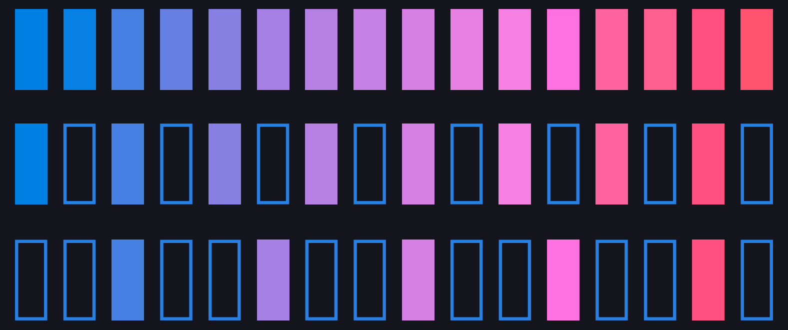
Sequential, `f(range)`:

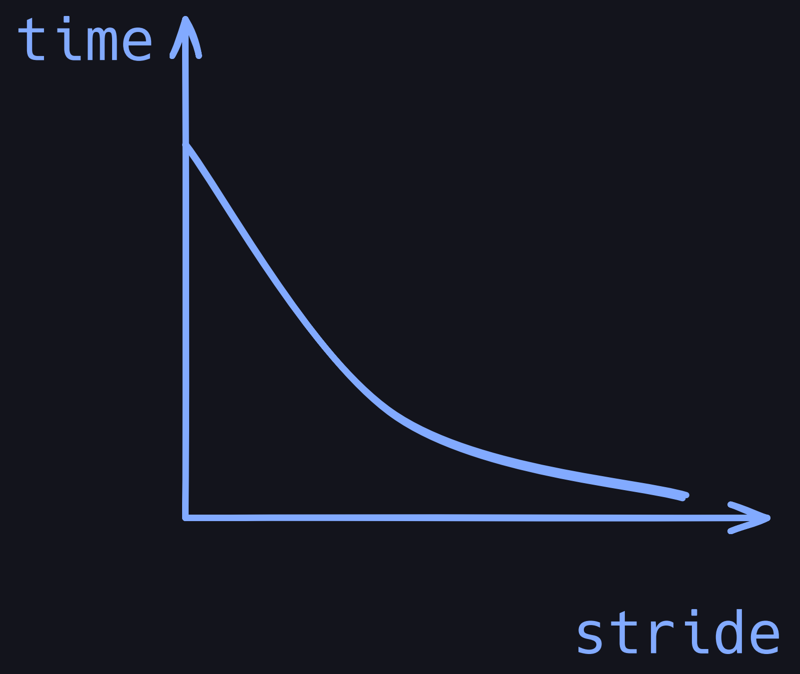


Random, `f(range)`:

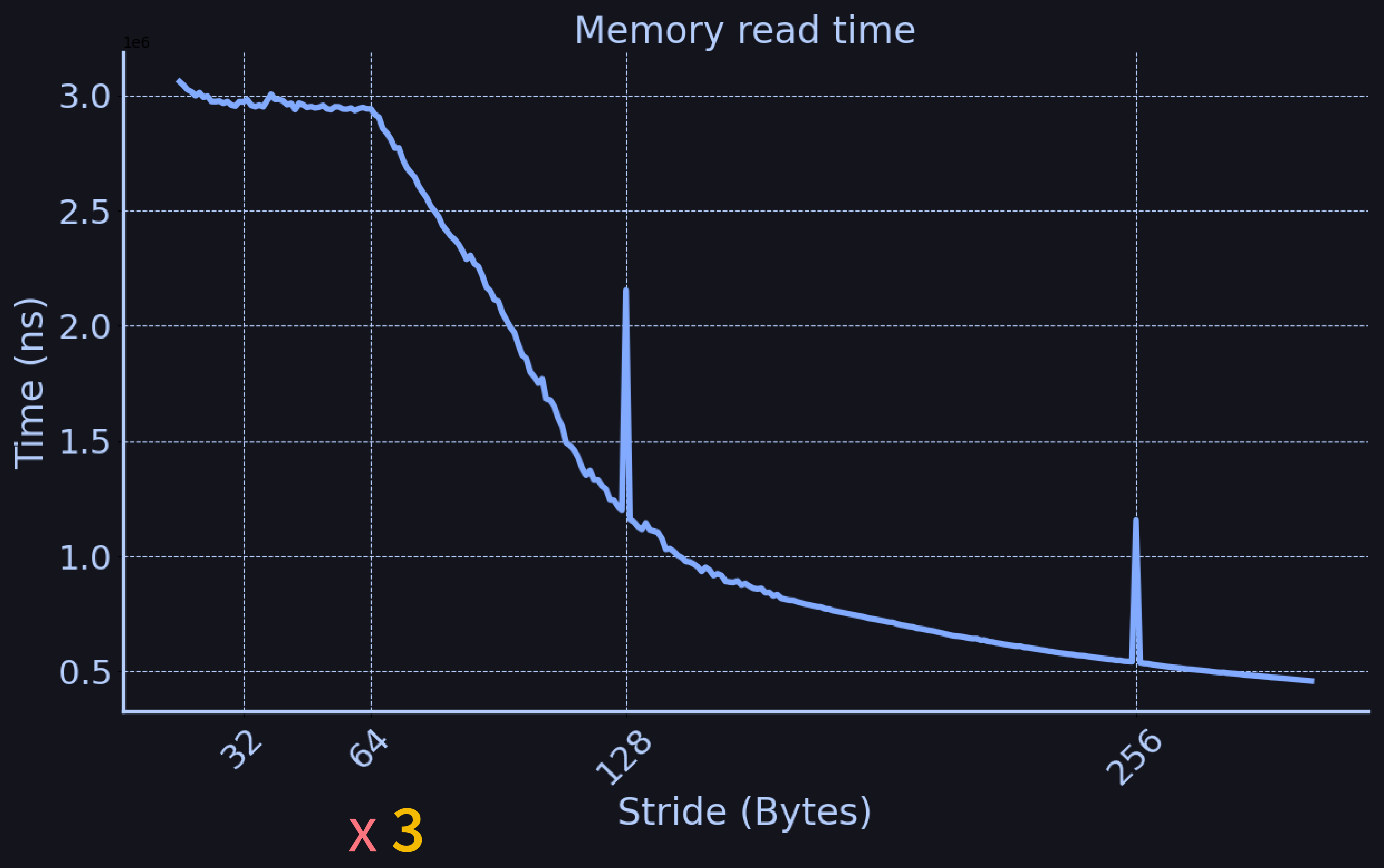


Stride, `f(stride)`:





Memory — strided copy



Stride: 1 byte

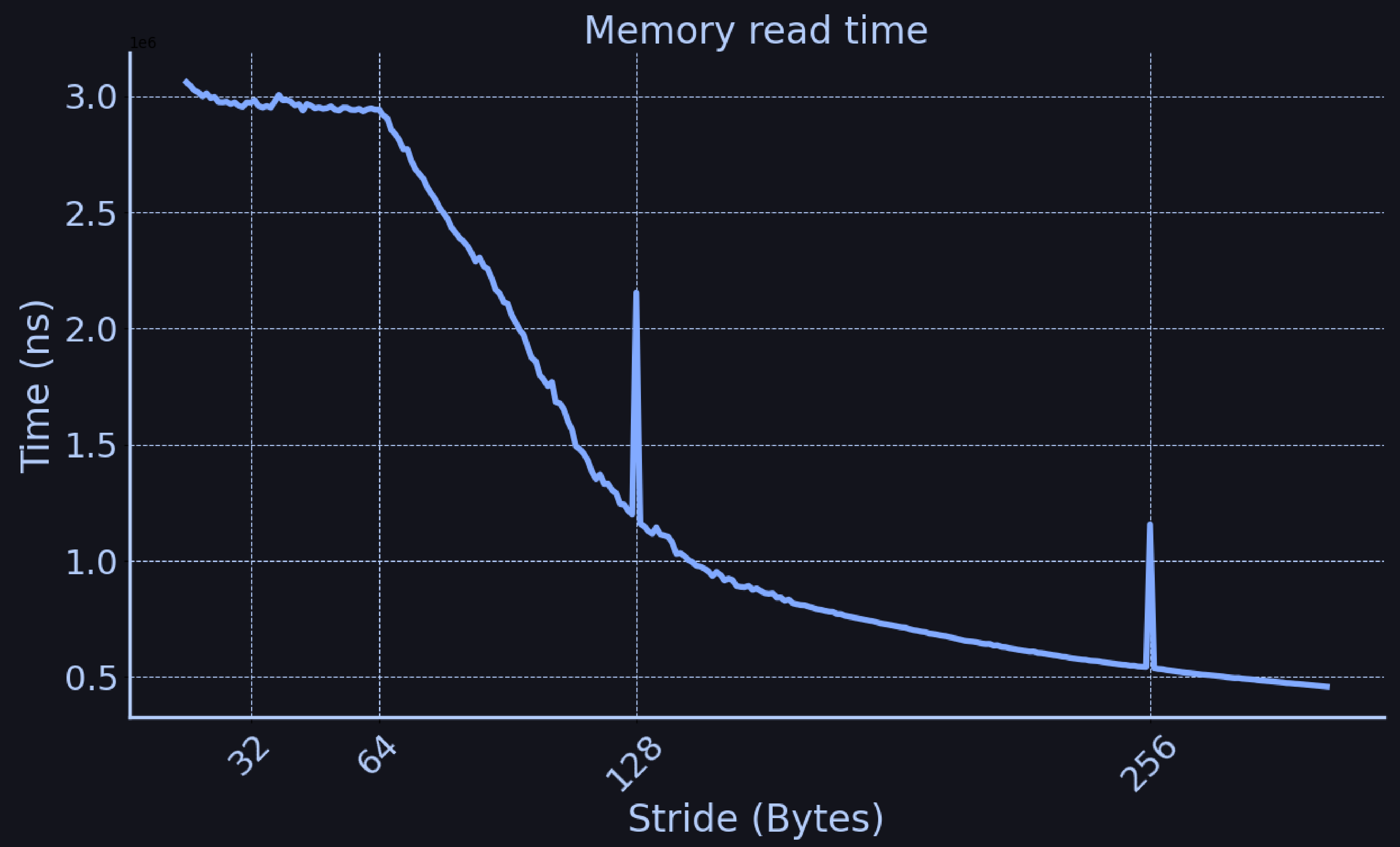
Stride: 2 bytes

Stride: 3 bytes

Stride: 4 bytes

Stride: 5 bytes

Memory — strided copy



L1	set 1	set 2	…	set N
way 1	___	___	…	___
…	…	…	…	…
way M	___	___	…	___

cache_sz = cache_line_sz · #ways · #sets

#ways = associativity

set index bits

Address: `xxxxxxxx xxxxxxxx 00111010 01101010 00001110 01000110 01110100 01xxxxxx`

16 bits: ignored

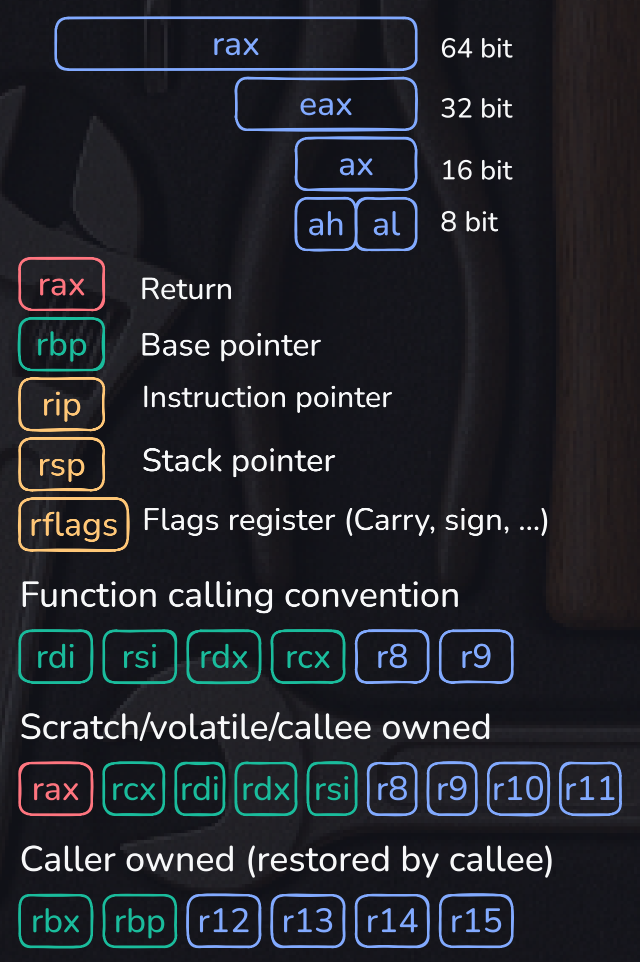
6 bits: cache line

Recap — Memory

- Memory layout & heap/stack
- Virtual memory (paging, TLB)
- Data Caching
  - Cache levels (L1, L2, L3)
  - Cache lines
  - Prefetching
  - Cache associativity
- Tools: `htop`, `tsc`, `google benchmark`

Toolbox Digression #4

- `nasm` (assembler)
  - Call assembly from C++
  - Exact control over executed code




```cmake
enable_language(ASM_NASM)
set(CMAKE_ASM_NASM_FLAGS "-f elf64")
add_executable(addExample)
target_sources(addExample PRIVATE
               add_example.cpp add_example.asm)
```

```nasm
global add
section .text

add:
    mov rax, rdi   ; copy first argument into rax
    add rax, rsi   ; add second argument
    ret
```

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

extern "C" int64_t add(int64_t a, int64_t b);

int main() {
  int64_t result = add(10, 20);
  std::cout << "Result: " << result << std::endl;
  return 0;
}
```

CPU — frontend / backend

add

```
ASM_add:
    %rep 100000
        add rax, 9
    %endrep
    ret
```

add loop

```
ASM_add_loop:
.loop:
    add rax, 9
    dec rdi
    jnz .loop
    ret
```

nop

```
ASM_nop:
    %rep 100000
        nop
    %endrep
    ret
```

nop loop

```
ASM_nop_loop:
.loop:
    nop
    dec rdi
    jnz .loop
    ret
```

9 byte nop

```
ASM_nop_long:
    %rep 100000
        nop_9_byte
    %endrep
    ret
```

9 byte nop loop

```
ASM_nop_long_loop:
.loop:
    nop_9_byte
    dec rdi
    jnz .loop
    ret
```

	add	add_loop	nop	nop_loop	nop_9B	nop_9B_loop
time (μs)	32	32	8	32	51	32
instructions (k)	101	301	101	301	101	301
uops_issued.any (k)	101	201	101	201	101	201
uops_executed.core (k)	101	201	1	101	1	101
icache.misses	55	8	1	7	10.897	7

fetch → decode → execute

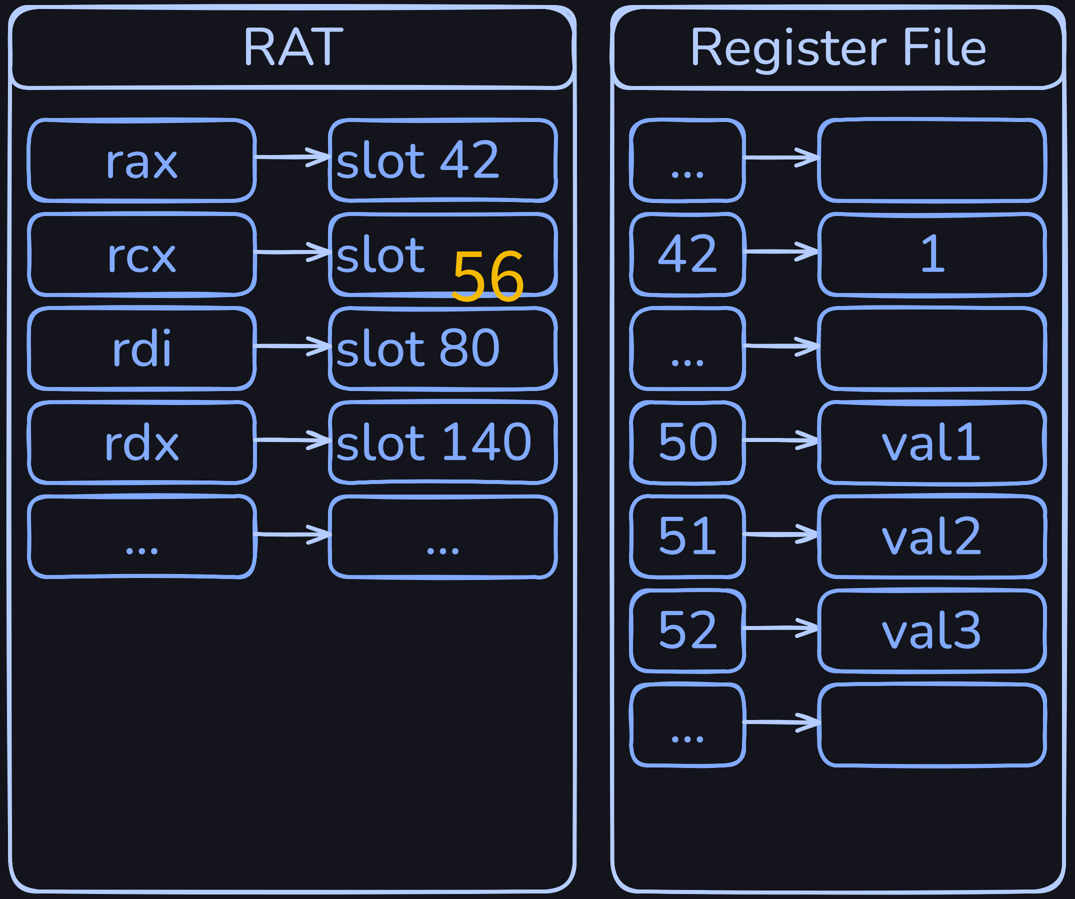
[Branch Prediction — Demo]

[Registers — Demo]

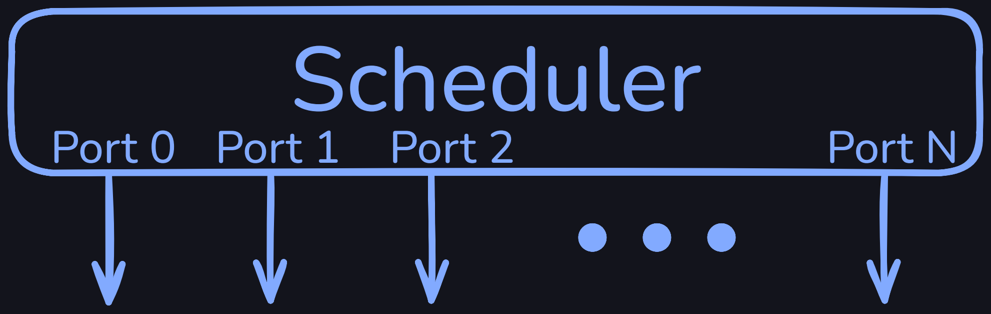
Registers

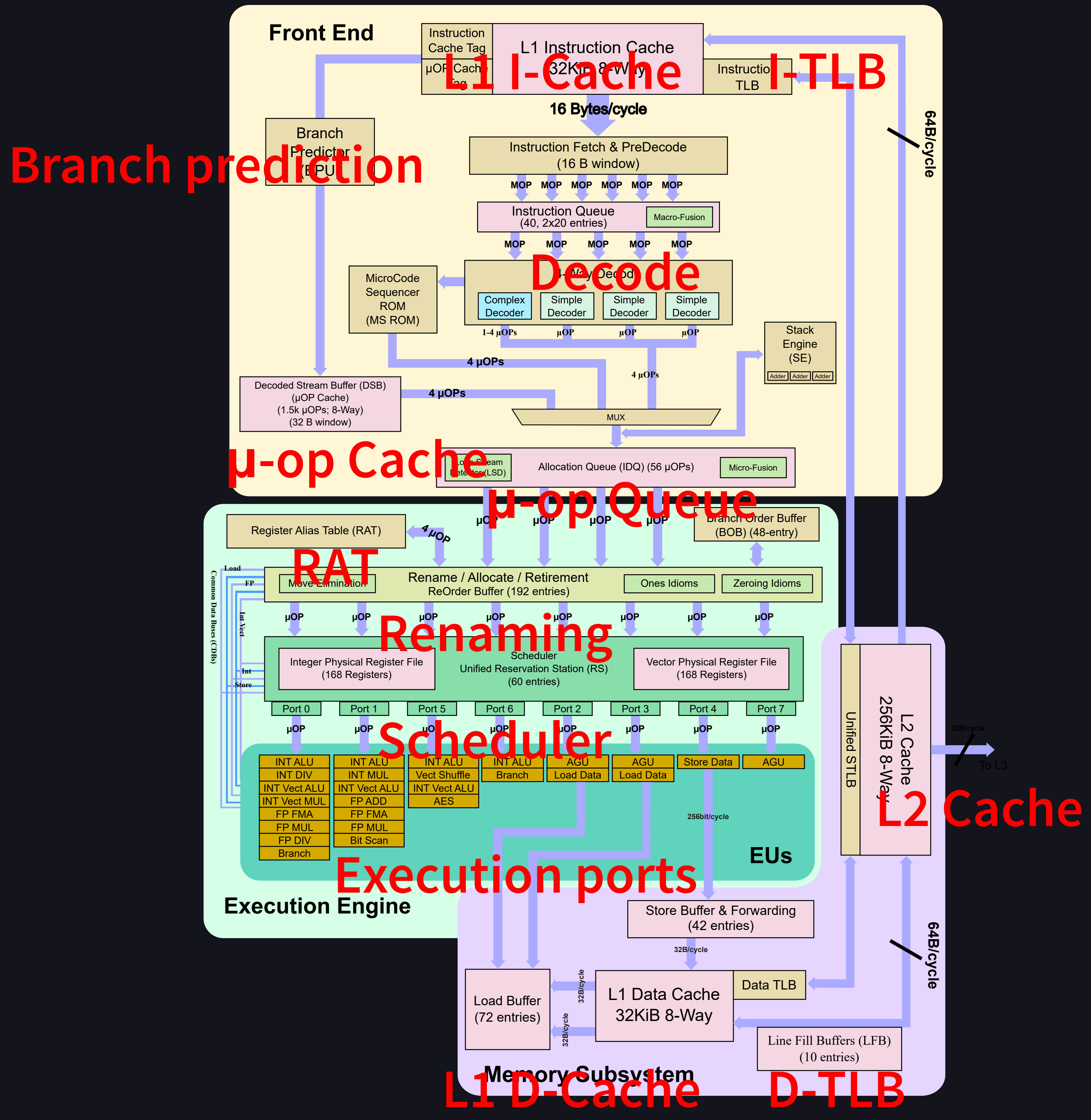
```
ASM_add:
    %rep 4000
      add rcx, 1     ; rcx += 1
    %endrep
    ret
```

```
ASM_mov_add:
    mov rax, 1
    %rep 4000
      mov rcx, rax ; rcx = rax
      add rcx, 1   ; rcx += 1
    %endrep
    ret
```



instruction	dest	op1	op2
`...`	–	–	–
`mov rcx, rax`	s51	s42	–
`add rcx, 1`	s52	s51	1
~~`mov rcx, rax`~~	~~s53~~	~~s42~~	~~–~~
`add rcx, 1`	s54	s53	1
~~`mov rcx, rax`~~	~~s55~~	~~s42~~	~~–~~
`add rcx, 1`	s56	s55	1
`...`	–	–	–



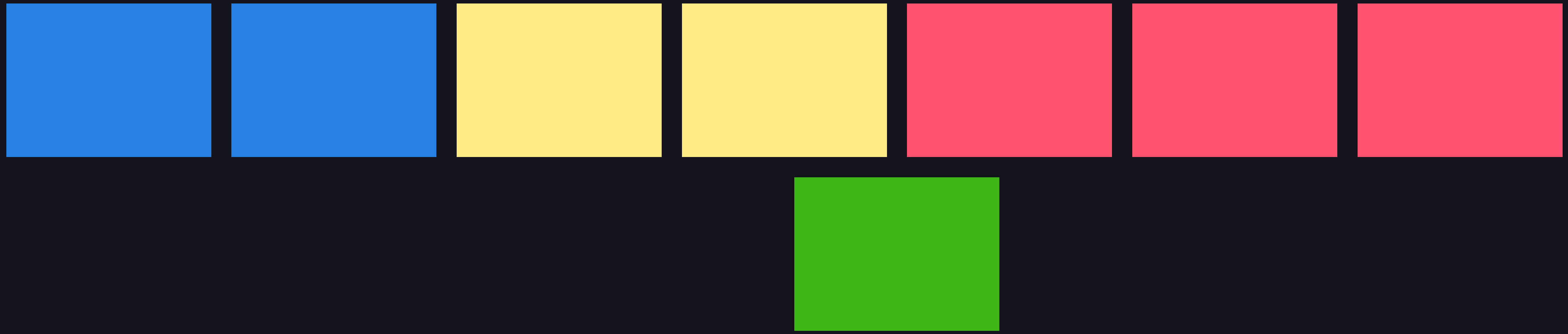

Front End
L1 I-Cache
I-TLB
Branch prediction
Decode
µ-op Cache
µ-op Queue
RAT
Renaming
Scheduler
Execution ports
L2 Cache
L1 D-Cache
D-TLB
L1 Instruction Cache
Instruction Cache Tag
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
Instruction Queue (40, 2x20 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
Complex Decoder
Simple Decoder
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (32 B window)
4 µOPs
MUX
Allocation Queue (IDQ) (56 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
Branch Order Buffer (BOB) (48-entry)
Rename / Allocate / Retirement ReOrder Buffer (192 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (168 Registers)
Scheduler Unified Reservation Station (RS) (60 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
FP MUL
FP DIV
Branch
INT MUL
FP ADD
Bit Scan
Vect Shuffle
AES
AGU
Load Data
Store Data
EUs
Execution Engine
Unified STLB
256KiB 8-Way
Store Buffer & Forwarding (42 entries)
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

Sorted vector

- maintain sorted vector<T>, support insert/remove/update element

```cpp
sorted_values_.push_back(new_val);
std::sort(sorted_values_.begin(), sorted_values_.end());
```

- Test: 1'000'000 elements, 1'000 inserts: **46s**



Sorted vector

- maintain sorted `vector<T>`, support insert/remove/update element

```cpp
sorted_values_.push_back(new_val);
std::sort(sorted_values_.begin(), sorted_values_.end());
```

- Test: `1'000'000` elements, `1'000` inserts: **46s**

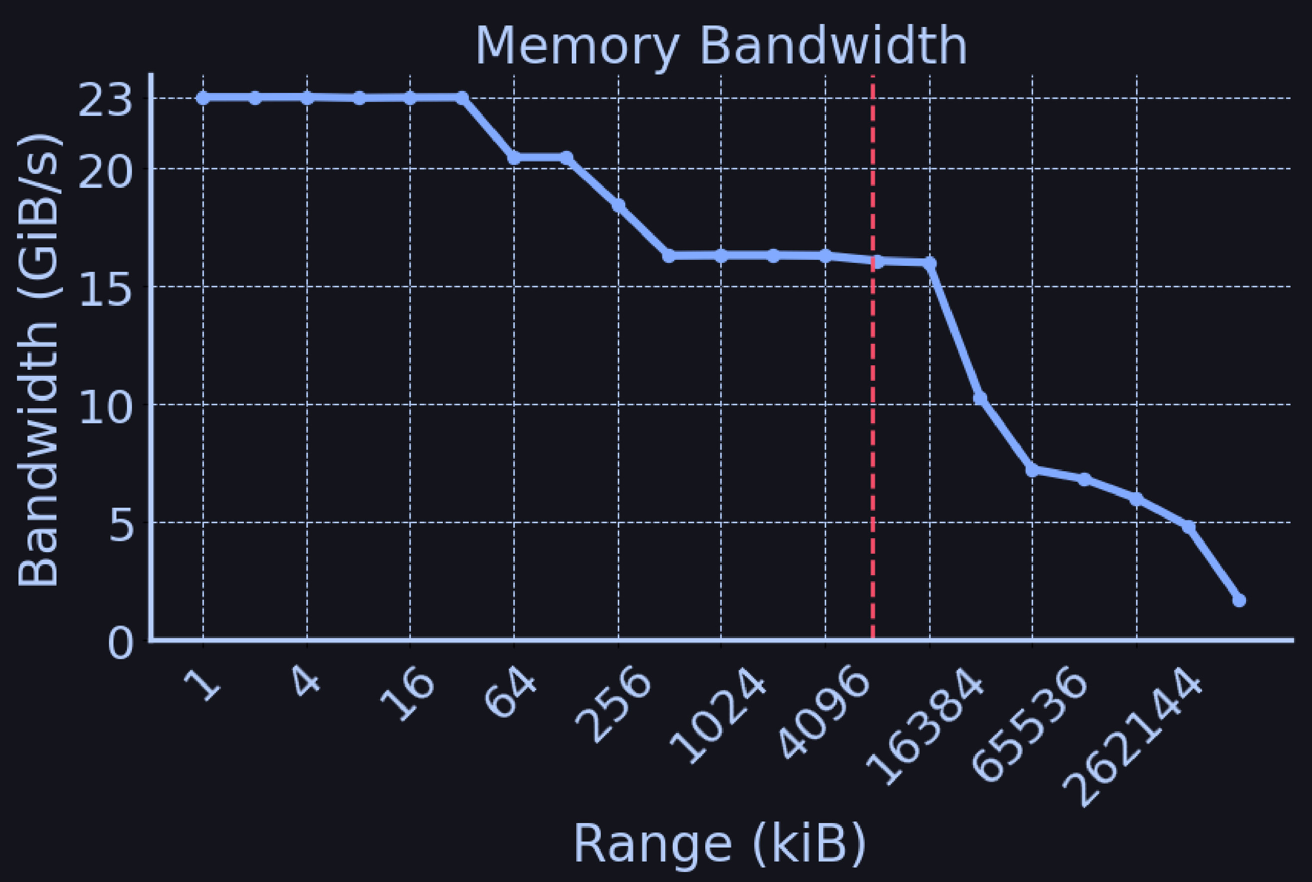
```cpp
auto it = std::lower_bound(sorted_values_.begin(), sorted_values_.end(), new_val);
sorted_values_.insert(it, new_val);
```

Sorted vector

- maintain sorted `vector<T>`, support insert/remove/update element

```cpp
sorted_values_.push_back(new_val);
std::sort(sorted_values_.begin(), sorted_values_.end());
```

- Test: `1'000'000` elements, `1'000` inserts: **46s**




```python
nr_moves = 1000000 / 2 * 1000

bw_gbyte_s = 16
bw_int64_s = (bw_gbyte_s << 30) / 8
expected_ms = int(nr_moves / bw_int64_s * 1e3)

print(f"expected duration: {expected_ms}ms")
```

```
expected duration: 232ms
```

Sorted vector

- maintain sorted `vector<T>`, support insert/remove/update element
- test with: `1'000'000` elements, `1'000` events

- **Insert** (`std::lower_bound + insert`): **204ms** (calculated: **232ms**)
- **Remove** (`std::lower_bound + erase`): **208ms**
- **Update** (`remove + insert`): **427ms**

Sorted vector

- maintain sorted `vector<T>`, support insert/remove/update element
- test with: `1'000'000` elements, `1'000` events

- **Insert** (`std::lower_bound + insert`): **204ms** (calculated: **232ms**)
- **Remove** (`std::lower_bound + erase`): **208ms**
- **Update** (`remove + insert`): **427ms**

```cpp
auto curr_loc = std::lower_bound(sorted_values_.begin(), sorted_values_.end(), existing);
auto new_loc = std::lower_bound(sorted_values_.begin(), sorted_values_.end(), new_val);
*curr_loc = new_val;  // update value

if ((curr_loc == new_loc) || ((curr_loc + 1) == new_loc)) { continue; }  // same location

auto start = std::min(curr_loc, new_loc);
auto end = std::max(curr_loc + 1, new_loc);
auto mid = (new_loc < curr_loc) ? curr_loc : curr_loc + 1;
std::rotate(start, mid, end);
```

Sorted vector

- maintain sorted `vector<T>`, support insert/remove/update element
- test with: `1'000'000` elements, `1'000` events

- **Insert** (`std::lower_bound + insert`): **204ms** (calculated: **232ms**)
- **Remove** (`std::lower_bound + erase`): **208ms**
- **Update** (`remove + insert`): **427ms**
- **Update** (`2x std::lower_bound + std::rotate`): **141ms**

Memory Usage Python

- sum all quantities

```python
things = get_things(5)
quantities = [thing.qty_ for thing in things]
print(sum(quantities))
```

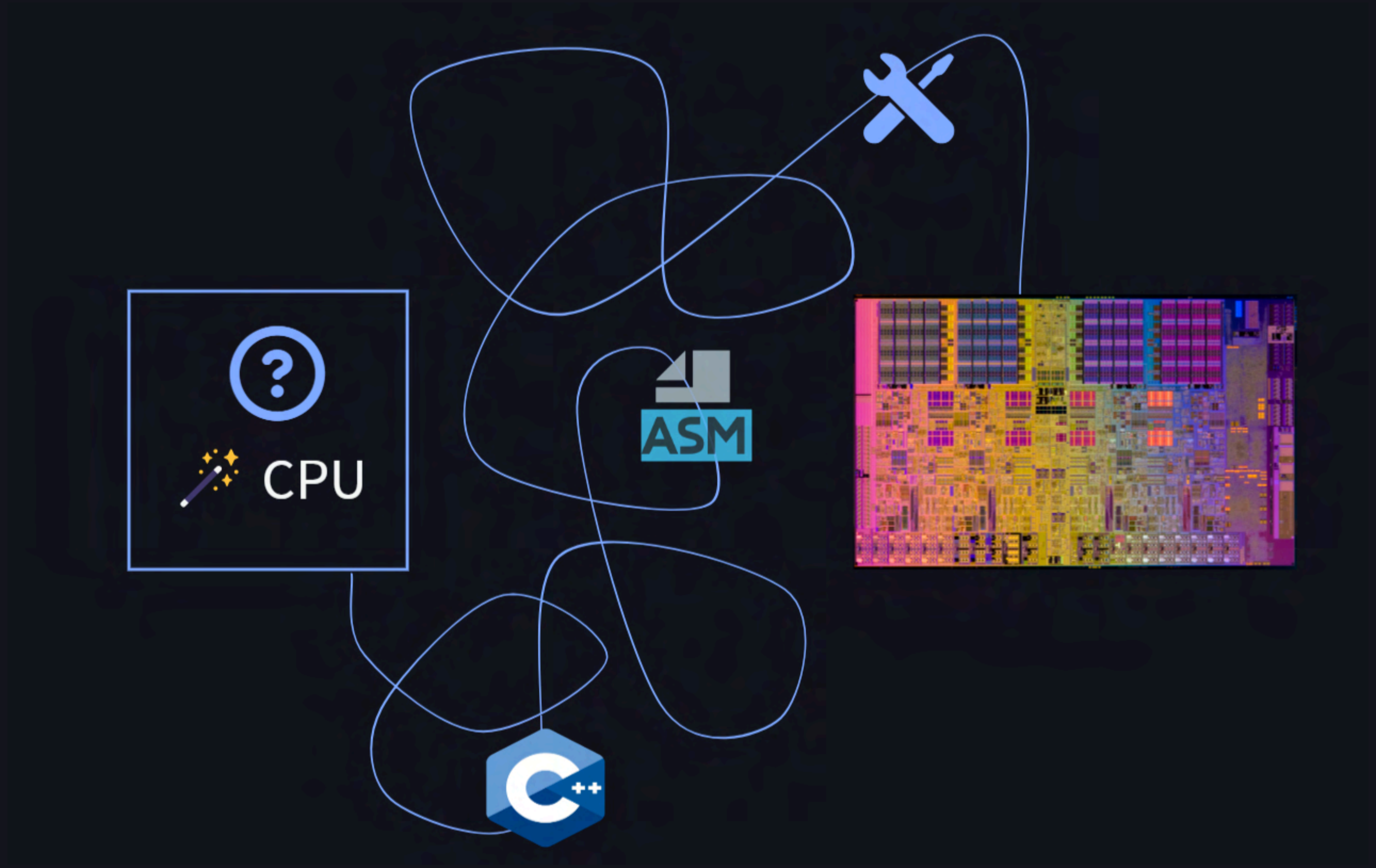
→ mem usage: 9MB

```python
things = get_things(10000000)
quantities = [thing.qty_ for thing in things]
print(sum(quantities))
```

→ mem usage: 404MB

```python
things = get_things(10000000)
quantities = (thing.qty_ for thing in things)
print(sum(quantities))
```

→ mem usage: 9MB



github.com/linusboehm/perf_stuff/

: `objdump` , `perf` , `htop` , `rdtsc` , `google benchmark` , `nasm` , `llvm-mca` ,
`compiler-explorer` , `beman exemplar`