

C++ Summit 2020

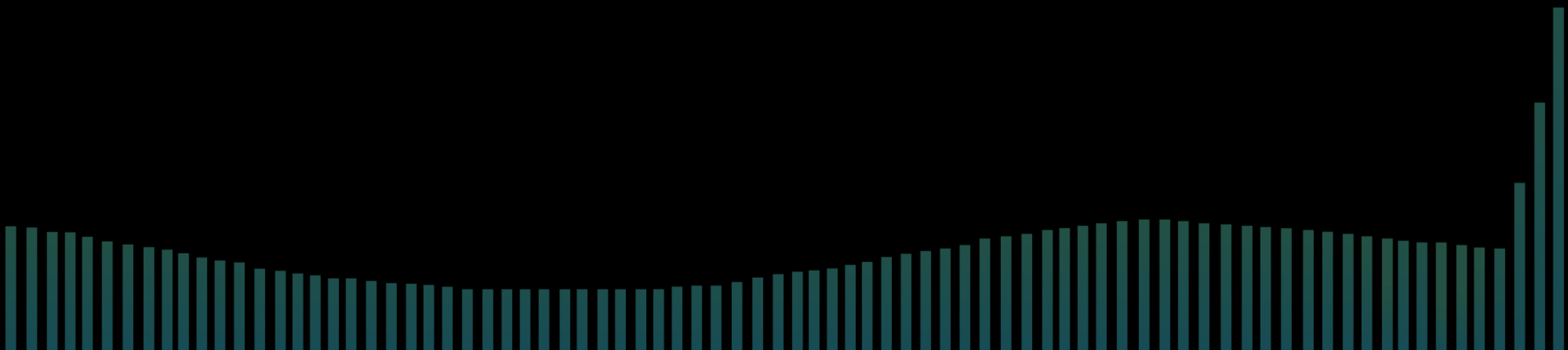
吴咏炜
博览首席咨询师

C++ 性能调优 纵横谈

自我介绍

- 学编程超过 35 年
- 近 30 年 C++ 老兵
- 热爱 C++ 和开源技术
- 多次在 C++ 大会上推广 C++ 新特性
- 对精炼、跨平台的代码有特别偏好
- 目前从事 C++ 方面的咨询工作

Premature optimization is the root of all evil.



We should forget about small efficiencies, say about 97 percent of the time: premature optimization is the root of all evil.

—Donald Knuth, “Structured programming with go to statements”, *ACM Computing Surveys*, **6**, 4 (December 1974), p. 268. CiteSeerX: 10.1.1.103.6084

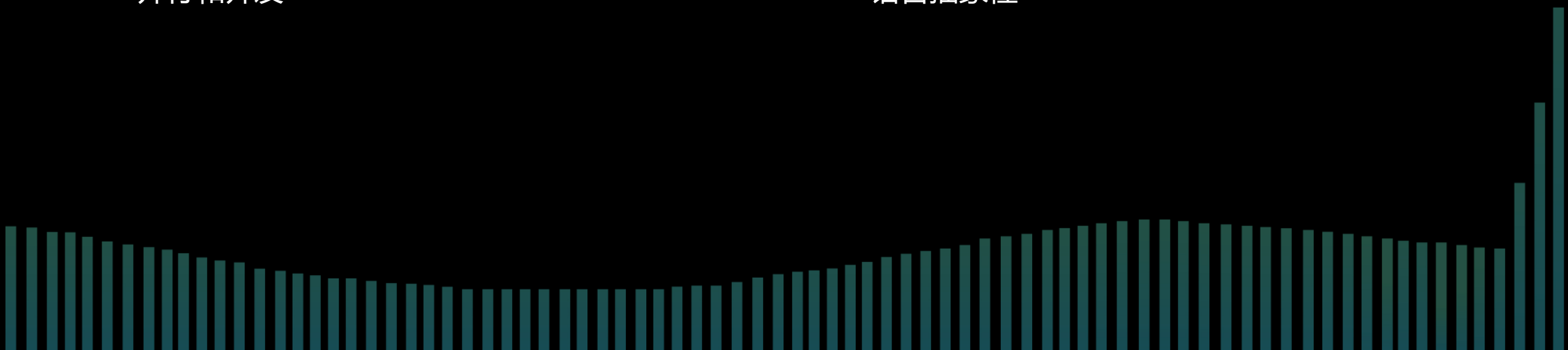
影响性能的架构因素

硬件

- 存储层次体系
- 处理器的乱序执行和流水线
- 并行和并发

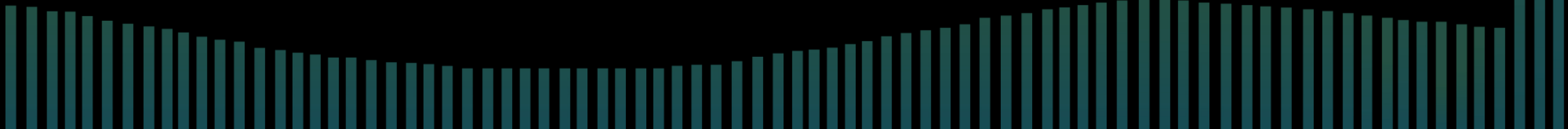
软件

- 系统调用开销
- 编译器优化
- 语言抽象性



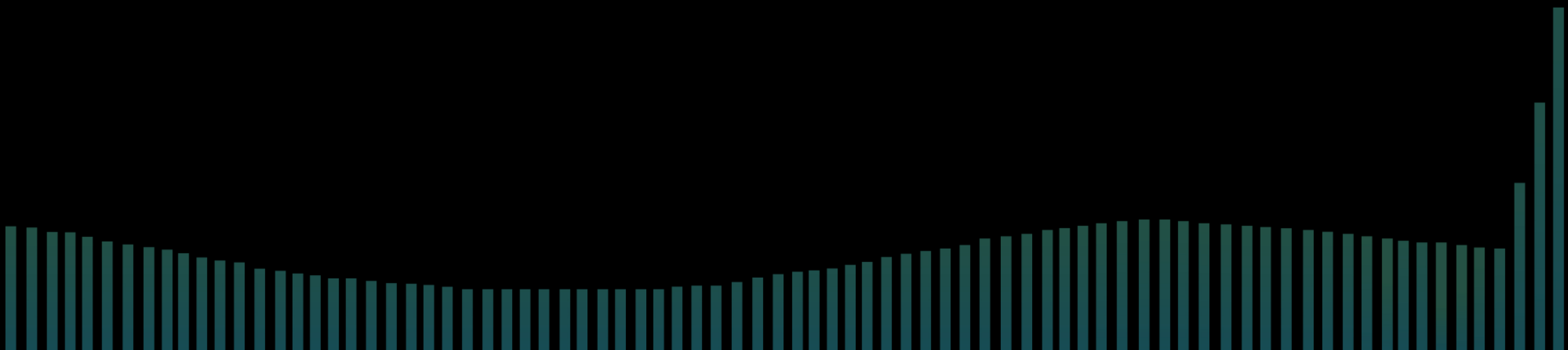
系统调用开销

- read
- write
- open
- close
- mmap
- gettimeofday
- ...



编译器优化

后面细说……



语言抽象性

C

```
Obj obj;
```

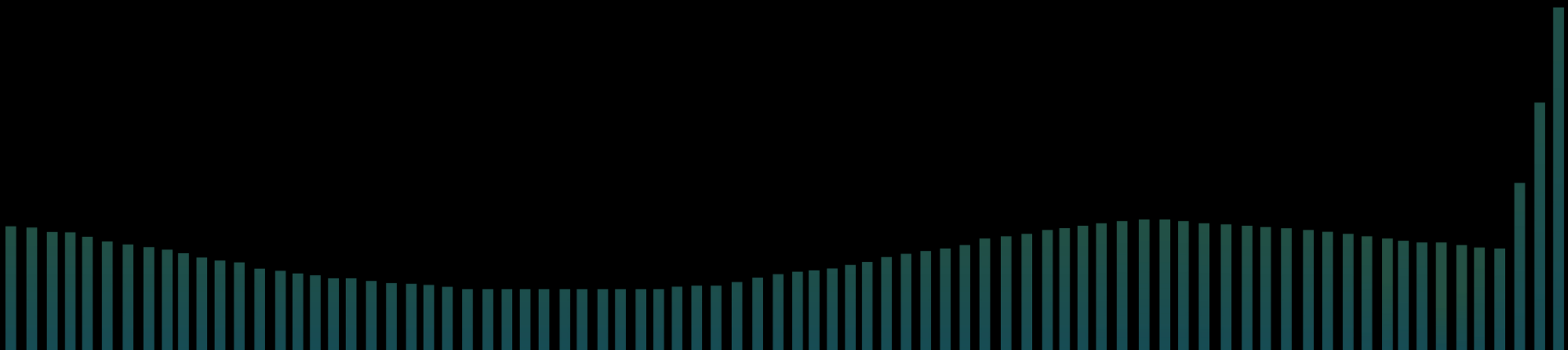
- 在栈上分配了 `sizeof(Obj)` 字节, $O(1)$ 开销

C++

```
Obj obj;
```

- 在栈上分配了 `sizeof(Obj)` 字节, $O(1)$ 开销
- 调用 `obj` **构造函数**, $O(?)$
- 到达下面的 `}` 时调用**析构函数**, $O(?)$

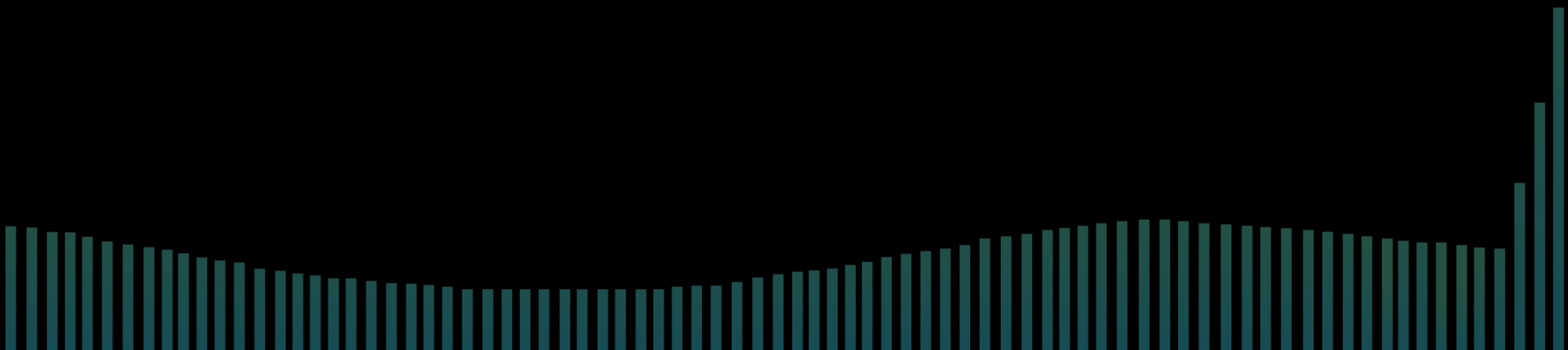
为什么要用 C++ ?



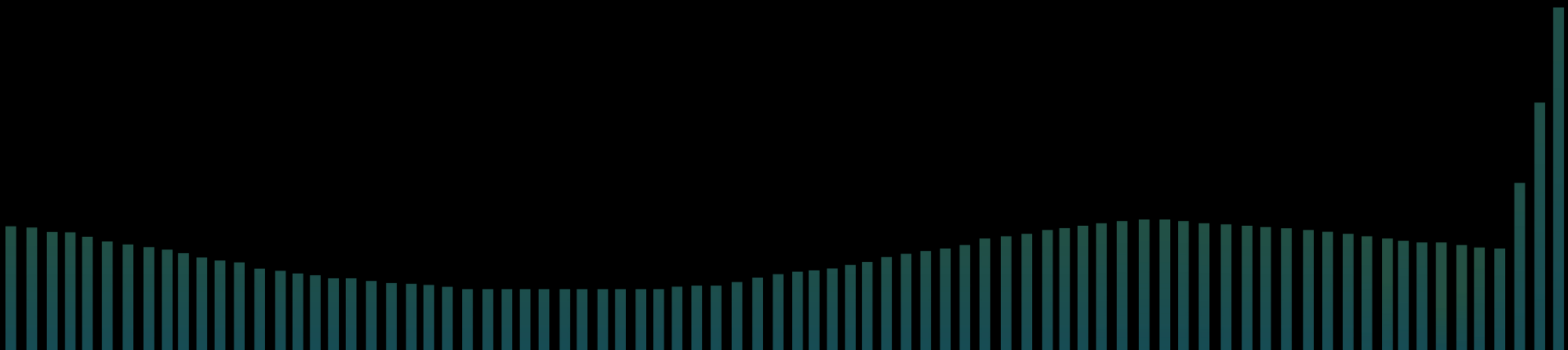
优化领域的阿姆达尔定律

$$S = \frac{1}{1 - P + \frac{P}{S_P}}$$

性能需要测试！



测不准问题.....



你知道结果吗？

```
char buffer[80];
auto t1 = clock();
for (auto i = 0; i < LOOPS; ++i) {
    memset(buffer, 0, sizeof buffer);
}
auto t2 = clock();
printf("%g\n",
       (t2 - t1) * 1.0 / CLOCKS_PER_SEC);
```

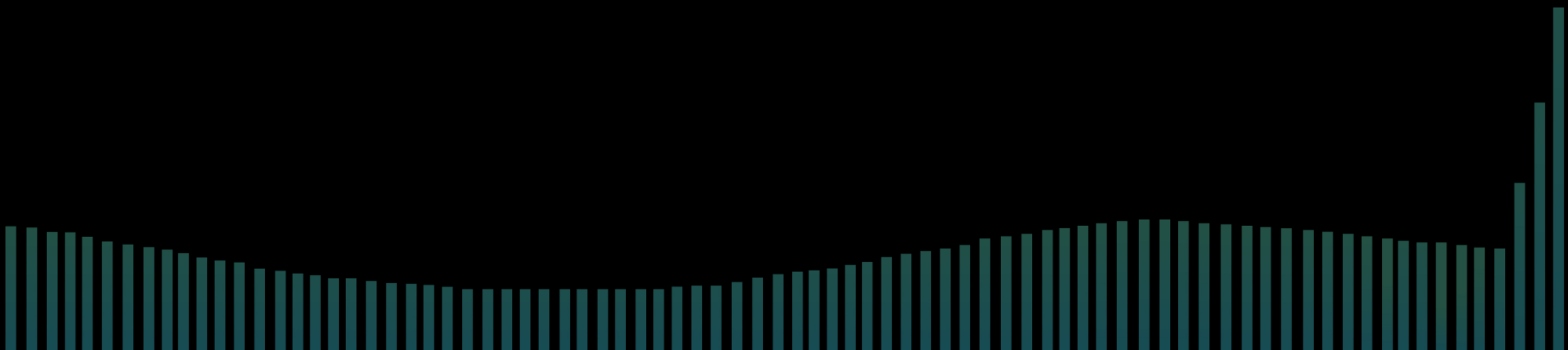
```
char buffer[80];
auto t1 = clock();
for (auto i = 0; i < LOOPS; ++i) {
    for (size_t j = 0; j < sizeof buffer;
         ++j) {
        buffer[j] = 0;
    }
}
auto t2 = clock();
printf("%g\n",
       (t2 - t1) * 1.0 / CLOCKS_PER_SEC);
```

GCC 8 的测试结果

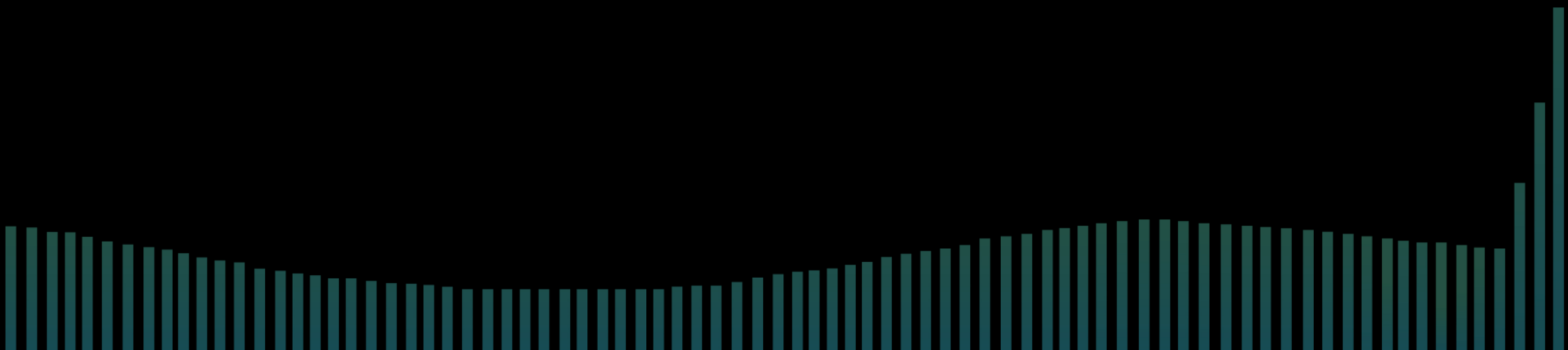
编译选项	memset:手工循环 (时间)
-O0	1:55
-O1	1:5
-O2	100000:1



原因：对 buffer 的写入被优化没了！



volatile ?



GCC 8 的测试结果 (volatile)

编译选项	memset:手工循环 (时间)
-O0	1:25
-O1	1:5
-O2	1:5

volatile 本身会妨碍优化……

```
volatile char buffer[80];
...
for (size_t j = 0; j < sizeof buffer; ++j) {
    buffer[j] = 0;
}
```

; GCC 10 下可能产生的汇编 (x86-64)

```
    xor     eax, eax
.L2:
    mov     BYTE PTR buffer[rax], 0
    add     rax, 1
    cmp     rax, 80
    jne     .L2
    ret
```

```
char buffer[80];
...
for (size_t j = 0; j < sizeof buffer; ++j) {
    buffer[j] = 0;
}
```

; GCC 10 下可能产生的汇编 (x86-64)

```
    pxor     xmm0, xmm0
    movaps   XMMWORD PTR buffer[rip], xmm0
    movaps   XMMWORD PTR buffer[rip+16], xmm0
    movaps   XMMWORD PTR buffer[rip+32], xmm0
    movaps   XMMWORD PTR buffer[rip+48], xmm0
    movaps   XMMWORD PTR buffer[rip+64], xmm0
    ret
```

<https://godbolt.org/z/vnbEaT>

编译器的优化魔法

在没有同步原语的情况下，编译器可以（通常为了性能）在（当前线程）结果不变的情况下自由地调整执行顺序

- 同步原语包括互斥锁操作、内存屏障、原子操作等

- 例子：`x = a; y = 2;` 可以变为 `y = 2; x = a;`

<https://godbolt.org/z/rbq1q9>

- 例子：`x = y + 1; y = x + 2;` 可以变为 `t = y; y += 3; x = t + 1;`



<https://godbolt.org/z/GEqxEs>

- 局部变量可能被完全消除
- 全局变量只保证在下一个同步点到来之前写回到内存里
- `volatile` 声明会禁止编译器进行相关的优化

C++ ▼

x86-64 gcc 4.4.7 ▼



-02

```
1  int x;
2  int y;
3  int a;
4
5  int main()
6  {
7      x = a;
8      y = 2;
9  }
10
```

A ⚙️ 🔍 📄 + 🔗

```

1  main:
2      mov     eax, DWORD PTR a[rip]
3      mov     DWORD PTR y[rip], 2
4      mov     DWORD PTR x[rip], eax
5      xor     eax, eax
6      ret
7
8  x:
9      .zero   4
10
11 y:
12     .zero   4
13
14 a:
15     .zero   4

```

A ▾

+

▾

v

+

+

+

+

+

+

C++ ▾

```

1  int x;
2  int y;
3
4  int main()
5  {
6      x = y + 1;
7      y = x + 2;
8  }
9

```

x86-64 gcc 4.4.7 ▾

✓

-O2

A ▾

⚙

▾

+

+

+

+

```

1  main:
2      mov     eax, DWORD PTR y[rip]
3      lea     edx, [rax+1]
4      add     eax, 3
5      mov     DWORD PTR y[rip], eax
6      xor     eax, eax
7      mov     DWORD PTR x[rip], edx
8      ret
9
10 x:
11     .zero   4
12
13 y:
14     .zero   4

```

防优化技巧

- 使用全局变量
- 使用锁来当作简单的内存屏障
- 可使用 `__attribute__((noinline))` 来防止意外内联
- 引用变量来防止其被优化掉

dont_optimize_away

```
// dont_optimize_away.h
```

```
void fake_reference(char* ptr);
```

```
template <typename T>
```

```
inline void dont_optimize_away(T&& datum)
```

```
{
    fake_reference(
        reinterpret_cast<char*>(&datum));
}
```

```
// dont_optimize_away.cpp
```

```
#include "dont_optimize_away.h"
```

```
#include <stdio.h>    // putchar
```

```
#include <unistd.h>   // getpid
```

```
static auto pid = getpid();
```

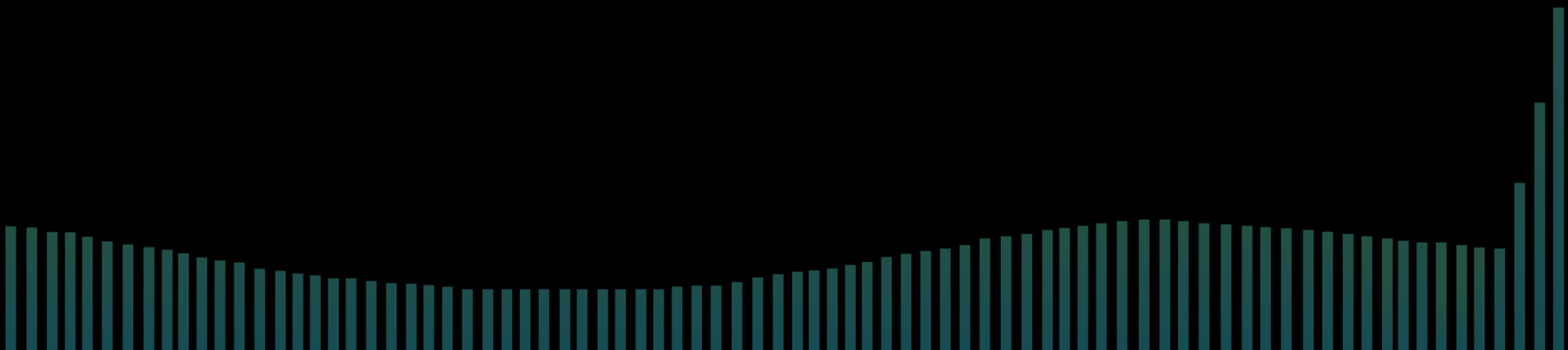
```
void fake_reference(char* ptr)
```

```
{
    if (pid == 0) {
        putchar(*ptr);
    }
}
```


GCC 10 的测试结果 (dont_optimize_away)

编译选项	memset:手工循环 (时间)
-O0	1:16
-O1	1:4
-O2	1:1

锁和额外函数调用的开销问题……



Linux 的时钟函数：某平台某环境的某次测试结果

函数	精度（微秒）	耗时（时钟周期）
clock	1	~1800
gettimeofday	1	~69
clock_gettime	0.0265(38)	~67
std::chrono::system_clock	0.0274(38)	~68
std::chrono::steady_clock	0.0272(28)	~68
std::chrono::high_resolution_clock	0.0275(20)	~69
rdtsc	0.00965(48)	~24

Windows 的时钟函数：某平台某环境的某次测试结果

函数	精度（微秒）	耗时（时钟周期）
clock	1（毫秒）	~160
GetTickCount	15.63(49)（毫秒）	~10
GetPerformanceCounter	0.1	~61
GetSystemTimeAsFileTime	15.63(49)（毫秒）	~20
GetSystemTimePreciseAsFileTime	0.1	~100
std::chrono::system_clock	0.1	~100
std::chrono::steady_clock	0.1	~160
std::chrono::high_resolution_clock	0.1	~160
rdtsc	0.00973(93)	~25

RDTSC 指令 (Read Time Stamp Counter)

- x86 和 x64 系统上的的首选计时方式
- MSVC 和 GCC 都提供内置函数 `__rdtsc` , 开发者不再需要自行用内联汇编实现
- 较新的 CPU 一般提供 `constant_tsc` 功能 , CPU 频率变化不会影响 TSC 计时
- 较新的 CPU 一般提供 `nonstop_tsc` 功能 , 休眠不影响 TSC 计时
- 多核 CPU 一般能同步 TSC , 但多 CPU 插槽的系统该指令可能仍有不一致问题
- TSC 频率和 CPU 参考主频不一定一致
 - 自行测量或使用 `dmesg | grep 'tsc.*MHz'`

实现一个小性能分析器 (profiler)

- 在构造时记录时间
- 在析构时再记录时间，并记录这次函数调用的内部消耗时间到全局变量里
- 程序退出时，打印记录的信息
- 单个的数据记录和整体的初始化、打印都依赖于 RAII

小性能分析器的使用

- 使用枚举 `profiled_functions` 记录所有需要分析的函数的编号
- 使用全局变量 `g_name_map` 记录函数编号和名称的映射关系
- 在函数内用宏 `PROFILE_CHECK` 记录函数的开销
- 使用循环重复执行待测代码，操作全局变量
- 在非时间记录部分，调用锁操作，确保消除过度优化或乱序执行

示例：测量时钟的开销

```

enum profiled_functions { PF_MEASURE_CLOCK_OVERHEAD_SINGLE };
name_mapper g_name_map[] = {
    {PF_MEASURE_CLOCK_OVERHEAD_SINGLE, "measure_clock_overhead_single"},
    {-1, NULL}};
void measure_clock_overhead_single()
{
    PROFILE_CHECK(PF_MEASURE_CLOCK_OVERHEAD_SINGLE);
    now = std::chrono::high_resolution_clock::now();
}
...
for (int i = 0; i < LOOPS; ++i) {
    std::lock_guard guard{mutex};
    measure_clock_overhead_single();
}
  
```


示例：测量时钟的开销

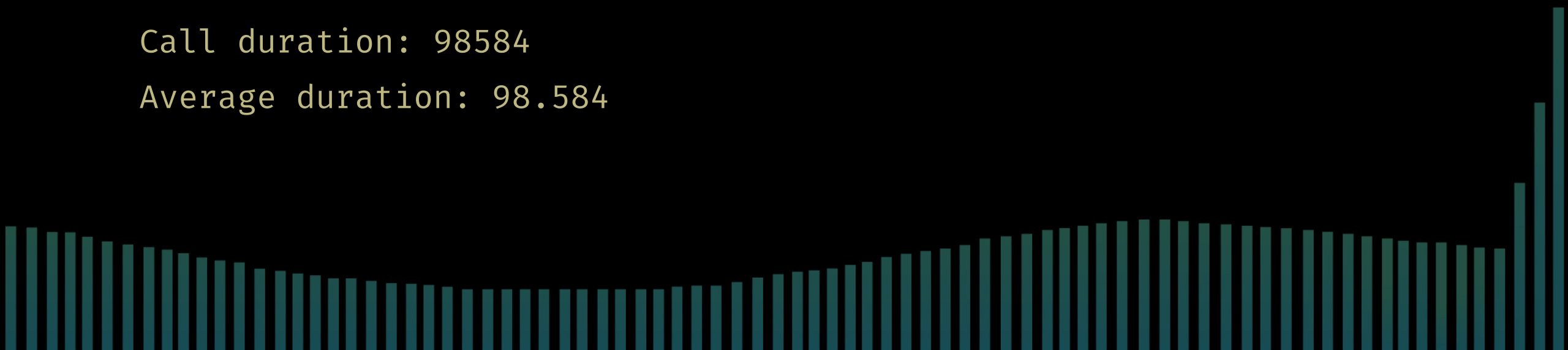
high_resolution_clock takes 94 cycles on average

0 measure_clock_overhead_single:

Call count: 1000

Call duration: 98584

Average duration: 98.584



示例：测量函数调用和虚函数调用的额外开销

```

class space_checker {
public:
    bool isspace(char ch)
    {
        return std::isspace(ch);
    }
};

class space_checker_noinline {
public:
    bool isspace(char ch);
};

__attribute__((noinline)) bool
space_checker_noinline::isspace(char ch)
{
    return std::isspace(ch);
}
  
```

```

class space_checker_intf {
public:
    virtual ~space_checker_intf() = default;
    virtual bool isspace(char ch) = 0;
};

class space_checker_virt : public space_checker_intf {
public:
    bool isspace(char ch) override;
};

bool space_checker_virtual::isspace(char ch)
{
    return std::isspace(ch);
}
  
```

示例：测量函数调用和虚函数调用的额外开销

0 count_space:

Call count: 10000

Call duration: 1724452

Average duration: 172.445

1 count_space_noinline:

Call count: 10000

Call duration: 2902176

Average duration: 290.218

2 count_space_virtual:

Call count: 10000

Call duration: 4300125

Average duration: 430.012

每次函数调用的开销：

$$\frac{290 - 172}{47} \approx 2.5$$

每次虚函数调用的开销：

$$\frac{430 - 172}{47} \approx 5.5$$

两种性能测试方式

插桩测试

- 开销随测试范围而变
- 插桩本身可能影响测试结果
- 测试结果可以较为精确、稳定
- 适合对单个函数进行性能调优
- 需要修改源代码或构建过程

采样测试

- 总体开销可控
- 一般不影响程序“热点”
- 基于统计，误差较大
- 适合用来寻找程序的热点
- 可以完全在程序外部进行测试

gprof 简介

基本使用

```
g++ -O1 -g -pg ...
```

```
./可执行程序名
```

```
gprof 可执行程序名 gmon.out > gprof.out
```

```
vim -c 'set nowrap' gprof.out
```

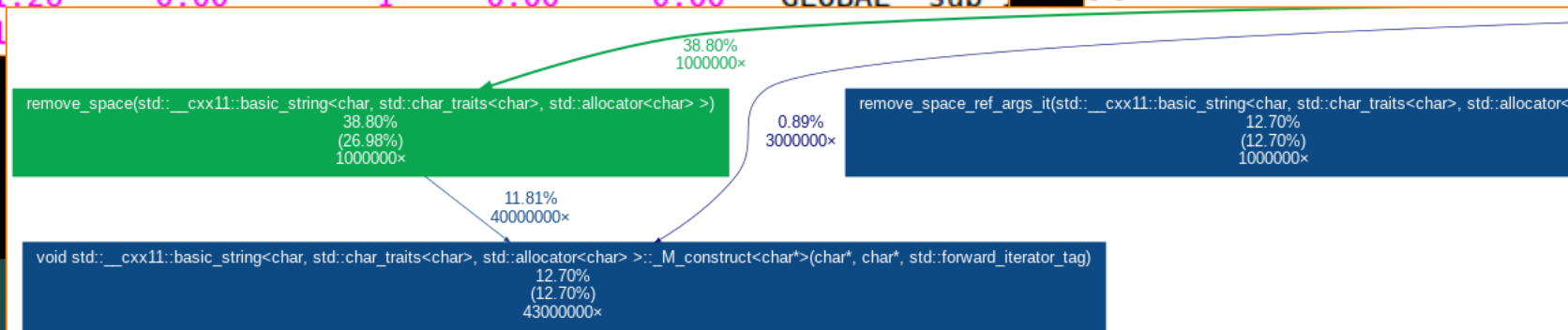
gprof 的输出

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
27.03	0.34	0.34	1000000	340.64	489.76	remove_space(s
12.72	0.50	0.16	43000000	3.73	3.73	void std::__cx
12.72	0.66	0.16	1000000	160.30	160.30	remove_space_
11.93	0.81	0.15	1000000	150.28	150.28	remove_space_
9.54	0.93	0.12	1000000	120.23	120.23	remove_space_
8.75	1.04	0.11	1000000	110.21	110.21	remove_space_
7.95	1.14	0.10	1000000	100.19	100.19	remove_space_
5.57	1.21	0.07	1000000	70.13	70.13	remove_space_
1.59	1.23	0.02	1000000	20.04	20.04	remove_space_
1.59	1.25	0.02				main
0.80	1.26	0.01				nvwa::fast_mu
0.00	1.26	0.00	1	0.00	0.00	GLOBAL sub
0.00	1					

index	% time	self	children	called	name
[1]	99.2	0.02	1.23		<spontaneous>
		0.34	0.15	1000000/1000000	main [1]
		0.16	0.00	1000000/1000000	remove_space
		0.15	0.00	1000000/1000000	remove_space
		0.12	0.00	1000000/1000000	remove_space
		0.11	0.00	1000000/1000000	remove_space
		0.10	0.00	1000000/1000000	remove_space
		0.07	0.00	1000000/1000000	remove_space
		0.02	0.00	1000000/1000000	remove_space
		0.01	0.00	3000000/43000000	void std::__
[2]	38.8	0.34	0.15	1000000	main [1]
		0.15	0.00	40000000/43000000	remove_space(std
		0.01	0.00	3000000/43000000	void std::__
[3]	12.7	0.16	0.00	43000000	main [1]
		0.15	0.00	40000000/43000000	remove_spac
		0.01	0.00	3000000/43000000	void std::__cx
[4]	12.7	0.16	0.00	1000000/1000000	main [1]
		0.16	0.00	1000000	remove_space_ref
[5]	11.9	0.15	0.00	1000000/1000000	main [1]
		0.15	0.00	1000000	remove_space_ref
[6]	9.5	0.12	0.00	1000000/1000000	main [1]
		0.12	0.00	1000000	remove_space_all
[7]	8.7	0.11	0.00	1000000/1000000	main [1]
		0.11	0.00	1000000	remove_space_res



gperftools 简介

安装 (Ubuntu)

```
sudo apt install google-perftools
```

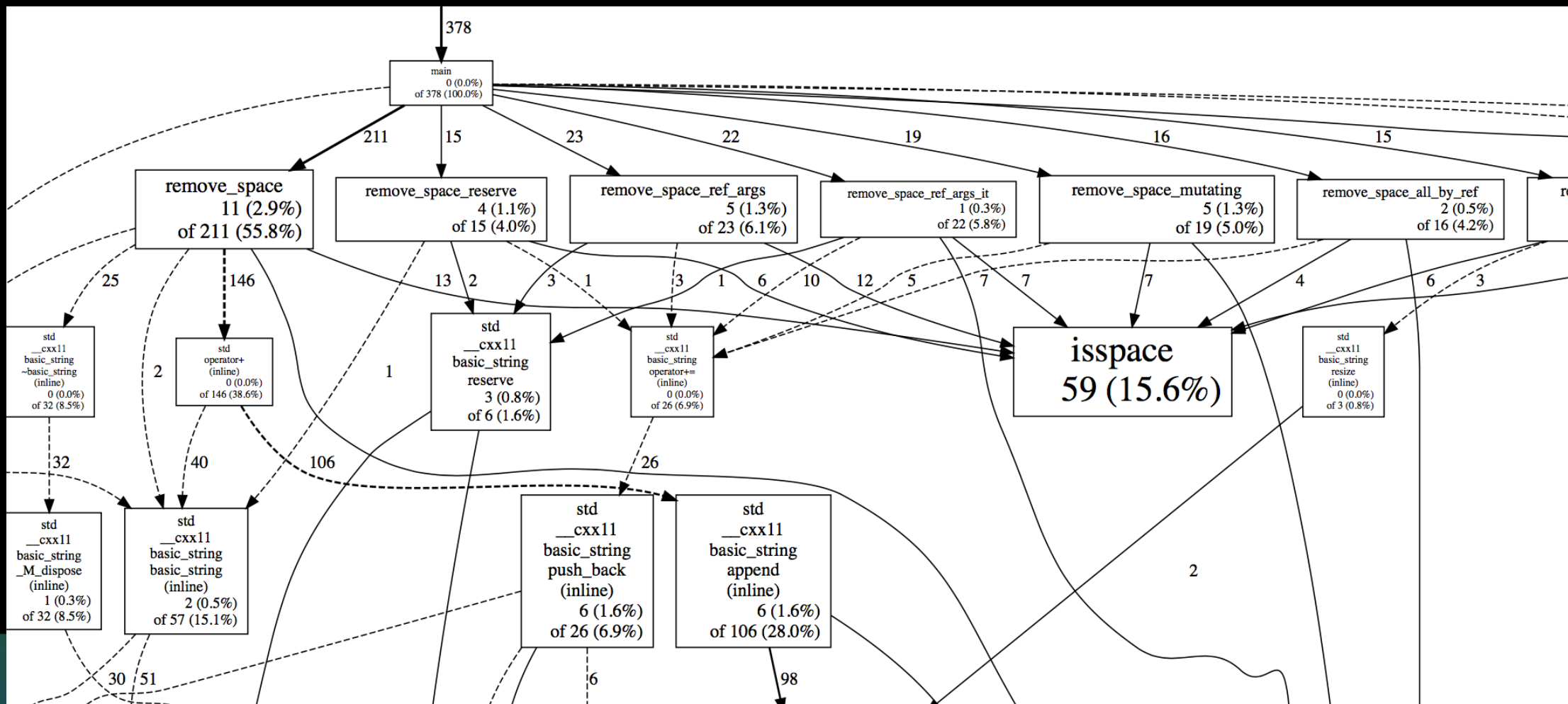
基本使用

```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libprofiler.so.0 \
```

```
  CPUPROFILE=test.prof ./可执行程序名
```

```
google-pprof --svg 可执行程序名 test.prof > test.svg
```

gperftools 的 SVG 输出



编译器选项 – 汇编代码生成

GCC

- `-S`
- `-masm=intel` (可选)
- 使用 `c++filt` 转换函数名
 - Vim 里 `:%!c++filt`

MSVC

- `/Fa`

浏览器里的汇编代码生成

Compiler Explorer: <https://godbolt.org/>

- 多种编译器 (GCC , Clang , MSVC , 等等)
- 多种平台 (x86-64 , ARM , 等等)
- 可自行设定编译 (优化) 选项
- 友好的过滤设置 , 一眼看到关注的内容
- 可在线运行 , 在浏览器里看到文本输出
- 允许使用常用 C++ 库 (Boost , Catch2 , fmt , GSL , range-v3 , spdlog , 等等)
-

A ▾

+

▾

v

+

+

+

+

+

+

C++ ▾

```

1  int x;
2  int y;
3
4  int main()
5  {
6      x = y + 1;
7      y = x + 2;
8  }
9

```

x86-64 gcc 4.4.7 ▾

✓

-O2

A ▾

⚙

▾

+

+

+

+

```

1  main:
2      mov     eax, DWORD PTR y[rip]
3      lea     edx, [rax+1]
4      add     eax, 3
5      mov     DWORD PTR y[rip], eax
6      xor     eax, eax
7      mov     DWORD PTR x[rip], edx
8      ret
9  x:
10     .zero   4
11  y:
12     .zero   4

```

未过滤的 GCC 汇编输出超过 30 行

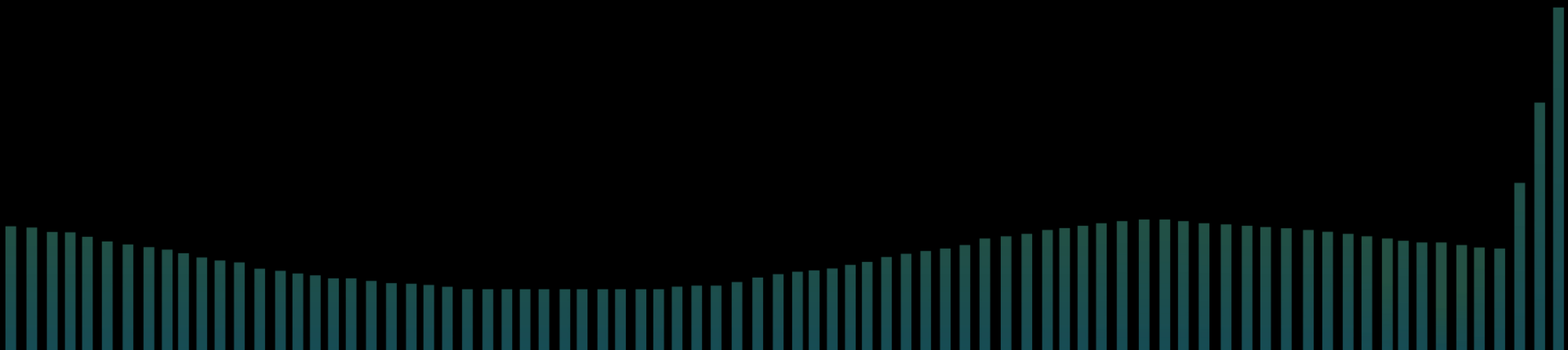
Compiler Explorer 示例

- C 的 Hello world : <https://godbolt.org/z/vns4n8>
- C++ 的 Hello world : <https://godbolt.org/z/bP9KTa>
- 简单循环和 strlen : <https://godbolt.org/z/5sb8bj>
- Ranges 和编译期优化 : <https://godbolt.org/z/foj6bq>

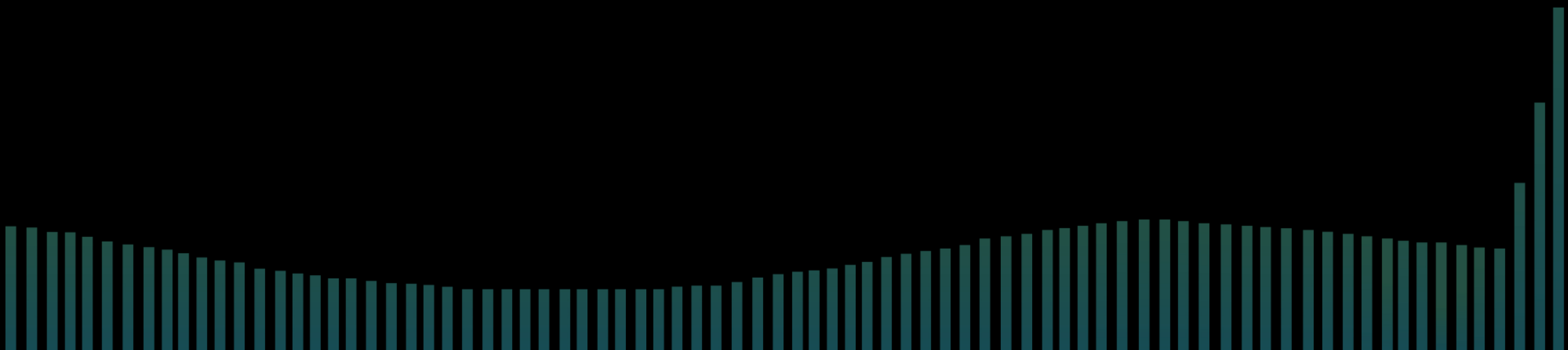
编译器选项 – 优化级别

- -O0 (默认) : 不开启优化, 方便功能调试
- -Og : 方便调试的优化选项 (比 -O1 更保守)
- -O1 : 保守的优化选项
 - 当前 GCC 上打开了 45 个优化选项
- -Os : 产生小代码的优化选项 (比 -O2 更保守, 并往产生较小代码的方向优化)
- -O2 : 常用的发布优化选项 (对错误编码容忍度低)
 - 当前 GCC 上比 -O1 额外打开 48 个优化选项
 - 包括自动内联函数和严格别名规则
- -O3 : 较为激进的优化选项 (对错误编码容忍度最低)
 - 当前 GCC 上比 -O2 额外打开 16 个优化选项
- -Ofast : 打开可导致不符合 IEEE 浮点数等标准的性能优化选项

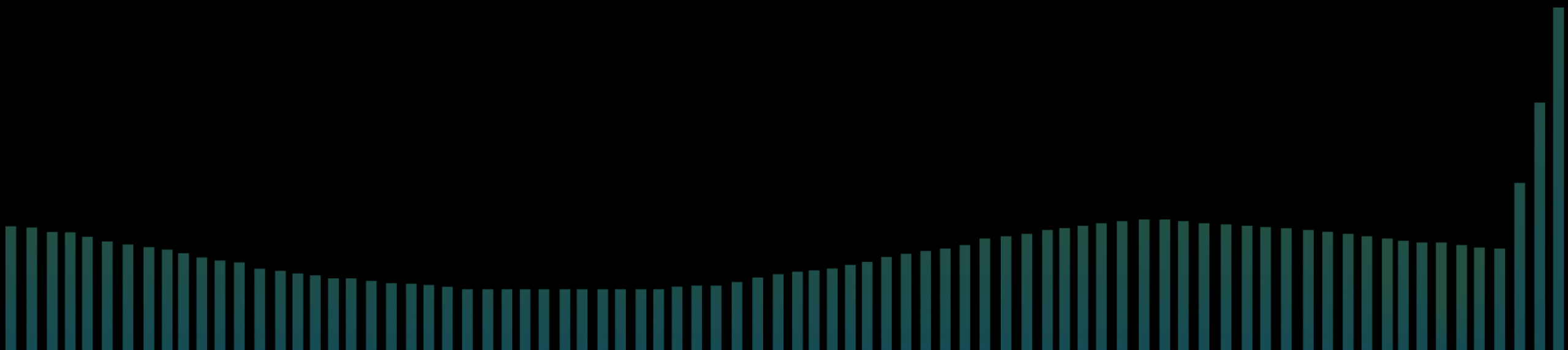
`std::sort` 比 `qsort` 更快！



快多少？



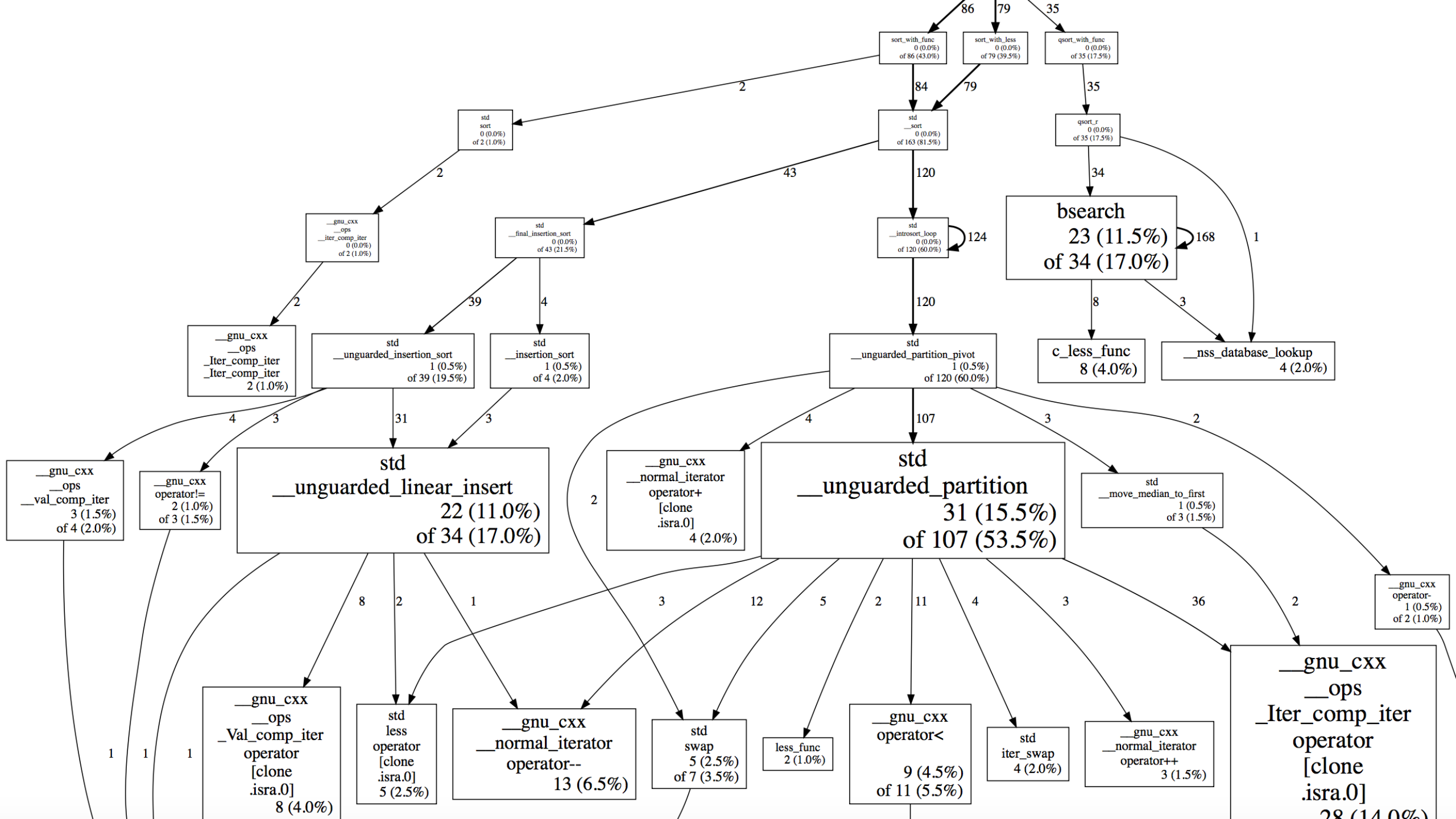
It depends. . . .



优化和内联的影响

	-O0	-O2 -fno-inline	-O2	-O2 对 -O0 的提升
sort_with_less	340981(1717)	197830(675)	24414(333)	14x
sort_with_func	334601(1843)	209241(609)	46384(220)	7.2x
qsort_with_func	133801(1814)	87014(790)	85323(535)	1.6x

单单内联即可产生一个数量级的性能差异！



循环优化（热点优化）

- 循环会放大代码中的低效率
- 把不必要的反复执行的代码提到循环外面

```
for (size_t i = 0; i < strlen(s); ++i) {  
    ...  
}
```

优化 I – 长度不变的情况

```
int sp_count = 0;
for (size_t i = 0,
      len = strlen(s);
      i < len; ++i) {
    if (isspace(s[i])) {
        ++sp_count;
    }
}
```

```
int sp_count = 0;
// 如果 s 是字符数组或 string
for (char ch : s) {
    if (isspace(ch)) {
        ++sp_count;
    }
}
```

优化 II – 长度可变的情况

```
for (size_t i = 0, len = strlen(s); i < len; ++i) {  
    if (isspace(s[i])) {  
        memmove(&s[i], &s[i + 1], len - i);  
        len--;  
    }  
}
```

如果使用 std::string 的话

```
for (auto it = s.begin(); it != s.end(); ++it) {  
    if (isspace(*it)) {  
        s.erase(it);  
    }  
}
```

注意隐藏开销

- 构造
- 析构
- 值参数的拷贝构造
-

```
std::string remove_space(std::string s)
{
    ...
}
```

对象参数的一般原则

- 如果不修改对象的话，使用 `const Obj&`
- 如果需要修改对象的话，使用 `Obj&`
- 如果需要在对象的新拷贝上进行操作的话，使用 `Obj`
 - 可以利用对象移动的可能性

```
// 如果返回新对象的话  
string remove_space(const string& s);
```

```
// 如果修改既有对象的话  
void remove_space(std::string& s);
```

```
// 从传入的对象生成结果  
string greet(string name)  
{  
    name += ", nice to meet your!";  
    return name;  
}
```


string 接口使用建议

- 不推荐使用 `const string&`（除非调用方确保有现成的 `string` 对象）
- 如果不需要修改字符串内容，使用 `string_view` 或 `const char*`
- 如果仅在函数内部修改字符串的内容，使用 `string`
- 如果需要修改调用者的字符串内容，使用 `string&`

消除不必要的反复构造和析构

```
for (...) {  
    std::string s;  
    s += "Hi, ";  
    ...  
    s += "Best regards,";  
    process(s);  
}
```



```
std::string s;  
for (...) {  
    s.clear();  
    s += "Hi, ";  
    ...  
    s += "Best regards,";  
    process(s);  
}
```

大幅减少了对内存管理器的调用次数

多线程优化

- 多线程加锁和竞争是性能杀手
- 能使用 atomic 就不要使用 mutex
- 如果读比写多很多，使用读写锁（shared_mutex）而不是独占锁（mutex）
- 使用线程本地（thread_local）变量

volatile vs atomic

```
std::atomic<int> a;
```

```
void increment_atomic(
    std::atomic<int>& n)
{
    for (int i = 0; i < LOOPS; ++i) {
        ++n;
    }
}
```

```
volatile int v;
```

```
void increment_volatile(
    volatile int& n)
{
    for (int i = 0; i < LOOPS; ++i) {
        ++n;
    }
}
```

*** Incrementing on the same atomic int

Result is 200000

*** Incrementing on the same volatile int

Result is 101893

简单增一、原子增一和加锁增一

	单线程（时钟周期数）	双线程（时钟周期数）
increment_volatile	~7	~7
increment_atomic	~19	~116
increment_with_lock	~72	~18000

thread_local 成员变量

```
class Obj {  
    ...  
public:  
    static Obj& get_instance();  
private:  
    static thread_local Obj inst_;  
};
```

```
thread_local Obj Obj::inst_;  
  
Obj& Obj::get_instance()  
{  
    return inst_;  
}
```

thread_local 变量

```
class Obj {  
    ...  
public:  
    static Obj& get_instance();  
};
```

```
Obj& Obj::get_instance()  
{  
    thread_local Obj inst;  
    return inst;  
}
```

算数表达式优化

$$x \cdot a \cdot b \cdot c = x \cdot (a \cdot b \cdot c)?$$

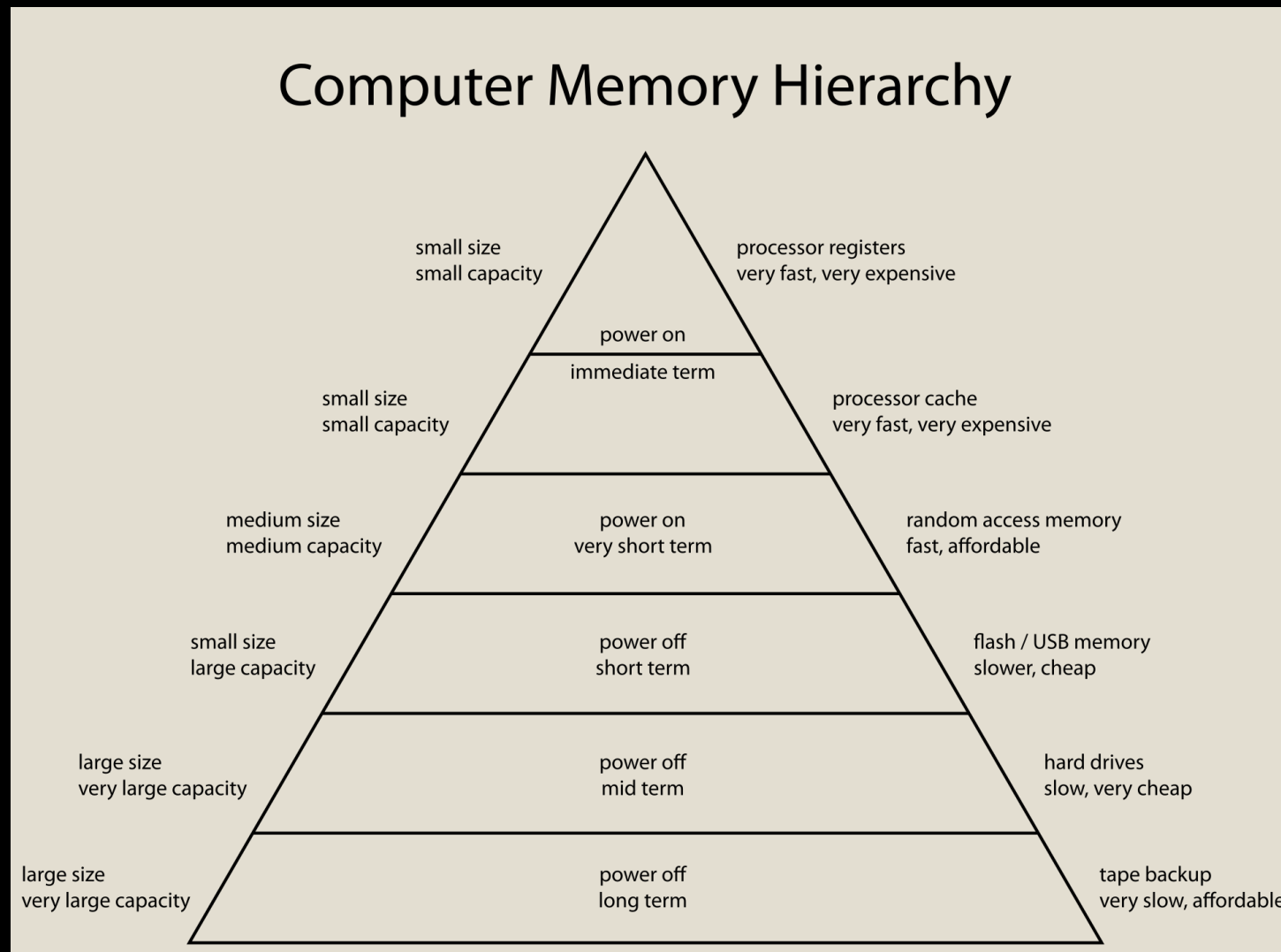
<https://godbolt.org/z/3Y15Es>

浮点数表达式优化

- 把常量归并到一起
 - $a \cdot x \cdot b \Rightarrow (a \cdot b) \cdot x$
- 手工简化表达式
 - $a \cdot x^2 + b \cdot x + c \Rightarrow (a \cdot x + b) \cdot x + c$
- 除法优化
 - $x / a \Rightarrow x \cdot (1 / a)$
- 注意在有硬件浮点运算单元的机器上可能 double 比 float 要快
- 关键代码上使用性能测试进行检查

缓存和性能

- 内存分级和访问性能
- 相邻数据访问较快
- 占用内存少 == 快
- 缓存竞争问题



矩阵转置中的缓存竞争

矩阵大小	内存占用 (KB)	每元素耗时 (时钟周期)	优化后耗时 (时钟周期)
63x63	31	1.2	—
64x64	32	1.2	—
65x65	33	1.2	—
127x127	126	1.3	—
128x128	128	4.8	1.4
129x129	130	1.3	—
255x255	508	1.7	—
256x256	512	11.6	1.9
257x257	516	1.8	—
511x511	2040	2.5	—
512x512	2048	16.3	3.6
513x513	2056	3.9	—

不需要的优化

- 把 $x * 8$ 写成 $x \ll 3$; 或把 $x * 5$ 写成 $(x \ll 2) + x$
 - 变乘为移位和加法是 GCC 在 **-O0** 时就会做的 !
- 提取公共表达式
 - 编译器会自动做
- 略去本地变量的初始化
 - 编译器经常会自动做 ; 安全编码检查要求初始化

x * 8 的非优化编译结果

```
int calculate(int x)
{
    return x * 8;
}
```

```
calculate(int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    sal     eax, 3
    pop     rbp
    ret
```

x * 5 的优化编译结果

```
int calculate(int x)
{
    return x * 5;
}
```

```
calculate(int):
    lea    eax, [rdi+rdi*4]
    ret
```

公共表达式提取示例

```
int calculate(int x, int y, int z)
{
    int a = (x * x + y * y) + 3;
    int b = (x * x + y * y) / z;
    return a + b;
}
```

```
calculate(int, int, int):
    imul    edi, edi
    mov     r8d, edx
    imul    esi, esi
    add     esi, edi
    mov     eax, esi
    cdq
    idiv    r8d
    lea     eax, [rsi+3+rax]
    ret
```

消除无用的初始化 I

```

int get1();
int get2();

int test()
{
    int retcode = 0;
    retcode = get1();
    if (retcode != 0) return retcode;
    retcode = get2();
    if (retcode != 0) return retcode;
    return retcode;
}
  
```

```

test():
    sub    rsp, 8
    call   get1()
    test   eax, eax
    je     .L5
    add    rsp, 8
    ret

.L5:
    add    rsp, 8
    jmp    get2()
  
```


消除无用的初始化 II

```

int g = 10;
void get(int& x)
{
    x = g + 1;
}
int test()
{
    int value = 0;
    get(value);
    return value + 32;
}
  
```

```

get(int&):
    mov     eax, DWORD PTR g[rip]
    add     eax, 1
    mov     DWORD PTR [rdi], eax
    ret

test():
    mov     eax, DWORD PTR g[rip]
    add     eax, 33
    ret

g:
    .long   10
  
```

无法消除初始化的情况

```
void get(int& x);
```

```
int test()
{
    int value = 0;
    get(value);
    return value + 32;
}
```

```
test():
```

```
sub    rsp, 24
lea    rdi, [rsp+12]
mov    DWORD PTR [rsp+12], 0
call   get(int&)
mov    eax, DWORD PTR [rsp+12]
add    rsp, 24
add    eax, 32
ret
```

C++ 在语言上不区分出参和入参的结果

其他性能调优手段

- C++：控制流优化，输入输出优化，内存优化，算法优化，.....
- 工具：[Intel VTune Profiler](#)，[perf](#)，.....
- 编译器：[Intel C++ Compiler](#)，[Clang](#)，.....
- 并行和并发编程：多线程，并行执行策略，[OpenMP](#)，.....
- 异构计算：[CUDA](#)，[OpenCL](#)，[SYCL](#)，[Data Parallel C++](#) / [oneAPI](#)，.....
- 开源第三方库

参考资料

- Agner Fog, "Software optimization resources".
<https://www.agner.org/optimize/>.
- Kurt Guntheroth, *Optimized C++*. O'Reilly, 2016.

需要鉴别资料中的过时信息和错误

代码

https://github.com/adah1972/cpp_summit_2020

