Performance Optimization

"Theory & Practice"

CPL 课程助教 宁锐
2025 年 11 月 28 日
rning@smail.nju.edu.cn

CONTENTS

性能问题分析与调优

1. 内存访问

内存访问：空间局部性

- **空间局部性**：存储空间上彼此比较接近的数据元素被集中使用的现象
  i. 按递增顺序遍历数组
  ii. 按行访问矩阵
- 当空间局部性较低时，缓存行中的许多字（word）实际上不会被使用。
- 优化原则：最大化缓存利用率
  i. 最大化空间局部性
  ii. 倾向于使用小的数据类型
  iii. 倾向于使用栈上数据

```c
// File: spatial.c

// Bad Practice
for (int k = 0; k < N; k++) {
    // row * column
    sum += A[i][k] * B[k][j];
}

// Good Practice
for (int k = 0; k < N; k++) {
    // row * row
    sum += A[i][k] * B[j][k];
}

// 加速比(-O3): 8.46x
```

内存访问：预取 (Prefetch)

- `__builtin_prefetch` 通过在数据被访问之前提前将其加载到缓存中，来降低缓存未命中带来的延迟。
- 它不仅可以用来提高空间局部性，也可以用来提高时间局部性。
- 另外，`-fprefetch-loop-arrays`也可以用来指导编译器发射预取指令。
- 优化原则:
  i. 预取即将要读取的多个(>1)值
  ii. 预取的数据大小小于缓存行大小

```
// File: prefetch.c

// Bad Practice
for (int i = 0; i < N; i++) {
    ret += arr[i];
}

// Good Practice
for (int i = 0; i < N; i += 4) {
    for (int j = 0; j < 4; ++j) {
        __builtin_prefetch(&arr[i + j]);
    }
    for (int j = 0; j < 4; ++j) {
        ret += arr[i + j];
    }
}
// 加速比(-O3): 1.80x
```

2. 循环优化

循环优化：循环展开 (Loop Unrolling)

- 循环展开是一种循环转换优化技术，通过减少循环迭代次数来优化代码。
- 这种优化可以生成更高效的程序，但是会增加可执行文件的大小。
- 优势：
  i. 减少分支指令
  ii. 允许使用 SIMD 指令提高效率
- 缺陷：
  i. 增加二进制文件大小
  ii. 占用更多的指令缓存

```c
// File: unroll.c

// Bad Practice
for (int i = 0; i < N; i++)
    arr[i] = x;

// Good Practice
for (int i = 0; i < N; i += 4) {
    // Suppose N is multiple of 4
    arr[i] = x;
    arr[i + 1] = x;
    arr[i + 2] = x;
    arr[i + 3] = x;
}

// 加速比(-O3): 2.38x
```

循环优化：循环展开 (Loop Unrolling)

除了手动展开循环的方法，也可以通过伪指令让编译器自动展开循环:

- `#pragma unroll` (clang)
- `#pragma GCC unroll` (gcc)

```
// Good practice
#pragma unroll 4
for (int i = 0; i < N; ++i) {
    arr[i] = x;
}

// Equivalent
for (int i = 0; i < N; i += 4) {
    // Suppose N is multiple of 4
    arr[i] = x;
    arr[i + 1] = x;
    arr[i + 2] = x;
    arr[i + 3] = x;
}
```

3. 指令并行

指令并行 (Instruction Level Parallelism)

- 现代超标量处理器可以在一个指令周期内完成多条指令。
- 这意味着不需要多线程的帮助，即使是单核心执行的程序，也有可能在处理器上被并行执行。
- 优化原则：
  i. 并行执行的指令间不允许存在依赖关系
  ii. 正确的打破朴素实现中的依赖关系

```
// File: ilp.c

// Bad Practice
for (int i = 0; i < N; ++i) {
    ret += arr[i];
}
return ret;

// Good Practice
for (int i = 0; i < N / 4; ++i) {
    a += arr[i][0];
    b += arr[i + N / 4 * 1];
    c += arr[i + N / 4 * 2];
    d += arr[i + N / 4 * 3];
}
return a + b + c + d;
// 加速比(-O3): 1.50x
```

指令并行 VS. 循环展开

- 循环展开:
  i. 有利于编译器生成更加高效的指令
  ii. 减少 CPU 执行分支指令的条数
- 指令并行:
  i. 适用于有依赖的情况，如`ilp.c`中的求和
  ii. 无法减少 CPU 实际执行的指令条数

```c
// Loop Unrolling
for (int i = 0; i < N; i += 4) {
    arr[i] = x;
    arr[i + 1] = x;
    arr[i + 2] = x;
    arr[i + 3] = x;
}

// Instruction Level Parallelism
for (int i = 0; i < N / 4; ++i) {
    a += arr[i];
    b += arr[i + N / 4 * 1];
    c += arr[i + N / 4 * 2];
    d += arr[i + N / 4 * 3];
}
```

4. 算术类型

算术类型

- **整数 vs. 浮点**: 一般来说，整数类型的运算始终快于浮点。
- **32 位整数 vs. 64 位整数**: 64 位整数的运算总是更慢。
- **32 位整数 vs. 更小的整数类型**: 一般来说，32 位整数总是更快。
- 优化原则:
  i. 尽可能避免使用 64 位整数类型
  ii. 尽可能使用 32 位整数类型而不是更小的类型

```c
// File: type.c

// Bad Practice
int8_t ret;
for (int i = 0; i < N; i++) {
    ret += *(int8_t *)&arr[i];
}

// Good Practice
int32_t ret;
for (int i = 0; i < N; i++) {
    ret += *(int32_t *)&arr[i];
}

// 加速比(-O3): 1.68x
```

5. 内建函数

内建函数：常见内建函数 (Compiler Intrinsics)

- 大多数现代编译器提供了比特操作的内建函数：
  - `__builtin_popcount(x)`：计算当前数值对应的二进制中 1 的个数
  - `__builtin_clz(x)`：计算当前数值对应的二进制中前导 0 的个数
  - `__builtin_ctz(x)`：计算当前数值对应的二进制中尾部 0 的个数
  - `__builtin_ffs(x)`：计算当前数值对应的二进制中低位 1 的下标

内建函数：用法

- 计算整数的`log2`

```
inline unsigned log2(unsigned x) {
    return 31 - __builtin_clz(x);
}
```

内建函数：用法

- 检查一个数是否是 2 的幂

```
inline unsigned is_power2(unsigned x) {
    return __builtin_popcount(x) == 1;
}
```

内建函数：用法

- 把一个数最低位的 1 置为 0

```
inline unsigned bit_clear(unsigned x) {
    int pos = __builtin_ffs(x); // range [0, 31]
    x &= ~(1u << pos);
    return x;
}
```

6. 编译器优化

编译器优化：常量折叠 (Constant Folding)

- 如果操作数为编译期常量，则编译期作运算。

```
// Before Optimization
int x = (2 + 3) * y;
bool foo = bar & false;

// After Optimization
int x = 5 * y;
bool foo = false;
```

编译器优化：代数化简 (Algebraic Simplification)

- 核心: 构造代数上的等价形式
- Division by Invariant Integers Using Multiplication @ PLDI'1994

```
// Reassociation
int foo = (a + 1) + 2;
int bar = a + (1 + 2);

// Commutativity
int foo = 2 + a + 4;
int bar = 2 + 4 + a;

// Strength Reduction
uint32_t foo = a / 10;
uint32_t bar = (uint64_t)a * 0xCCCCCCCD >> 35;
```

编译器优化：常量传播 (Constant Propagation)

- 如果一个变量的值是编译期常量，则把这个变量全部替换成常量。

```
// Before Optimization
int x = 5;
int y = x * 2;
int z = a[y];

// After Optimization
int x = 5;
int y = 10;
int z = a[10];
```

编译器优化：死代码消除 (Dead Code Elimination)

```
// Before Optimization
int x = y * y; // x is dead!
... // x never used
x = z * z;

// After Optimization
... // x never used
int x = z * z;
```

编译器优化：公共子表达式消除 (Common Subexpression Elimination)

```
// Before Optimization
a[i * 4] = 1;
a[i * 4 + 1] = 1;
a[i * 4 + 2] = 1;

// After Optimization
int t = i * 4;
a[t] = 1;
a[t + 1] = 1;
a[t + 2] = 1;
```

编译器优化：循环不变量外提 (Loop-Invariant Code Motion)

```
// Before Optimization
while (b) {
    z = y / x;
    ... // y, x not updated
}

// After Optimization
z = y / x;
while (b) {
    ... // y, x not updated
}
```

7. 性能分析

性能分析

- 工具: cargo-flamegraph
  - MacOS: `brew install cargo-flamegraph`
  - 其他平台: 参考 `https://github.com/flamegraph-rs/flamegraph`

欢迎交流 ~

幻灯片: https://ryani.org/assets/talks/perf.pdf

代码仓库: https://github.com/njurui/CPL-2025-Perf