CSCI 2021: Program Performance
Micro-Optimizations

Chris Kauffman

Last Updated:
Wed Apr 14 10:05:39 AM CDT 2021
Logistics

Reading Bryant/O’Hallaron

- Ch 6: Memory System
- Ch 5: Optimization

Goals

- Permanent Storage
- Optimization Overview
- Micro-optimizations

P4 Reminders

- Search Benchmark: report times that are > 1e-02
- Writeup: answers are 3-4 sentences, supported with tables of times

Upcoming Events

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 4/12</td>
<td>Storage</td>
</tr>
<tr>
<td>Wed 4/14</td>
<td>Micro Opts Lab: Preprocessor</td>
</tr>
<tr>
<td>Fri 4/16</td>
<td>Micro Opts</td>
</tr>
<tr>
<td>Mon 4/19</td>
<td>Review P4 Due</td>
</tr>
<tr>
<td>Wed 4/21</td>
<td>Exam 3</td>
</tr>
<tr>
<td>Fri 4/23</td>
<td>Virtual Memory</td>
</tr>
</tbody>
</table>
Caution: Should I Optimize?

- Optimizing program execution time saves CPU time, costs Human time
  - CPU Time: cheap
  - Human Time: expensive
- Determine if there is a NEED to optimize
- **Benchmark** your code - if it is fast enough, move on
- When optimizing, use data/tools to direct Human Effort (benchmarks/profiler)
- **Never sacrifice correctness** for speed

First make it **work**, then make it **right**, then make it **fast**.

- Kent Beck
What to Optimize First

In order of impact

1. Algorithms and Data Structure Selection
2. Elimination of unneeded work/hidden costs
3. Memory Utilization
4. Micro-optimizations

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

– Donald Knuth
Prema Turopt is tasked by her boss to optimize performance of function get_min().

The current version of the function code looks like the code to the right.

Prema immediately jumps to the code for bubble_sort() and alters the code to enable better processor pipelining.

This leads to a 2.5% improvement in speed.

```
int get_min(storage_t *st){
    int *arr = malloc(sizeof(int)*get_size(st));
    for(int i=0; i<get_size(st); i++){
        arr[i] = get_element(st,i);
    }
    bubble_sort(arr, get_size(st));
    int ans = arr[0];
    free(arr);
    return ans;
}
```

Suggest several alternatives that Prema should have explored
**Answers: Optimize This**

1. Don’t use bubblesort: $O(N^2)$. Use an $O(N \log N)$ sort like Quicksort, Heapsort, Mergesort.

2. Why sort at all? Determine the minimum element with the “get” loop.

3. What is the cost of get_element() and get_size()? Is there a more efficient iterator or array-extraction mechanism?

4. What data structure is used in storage_t? If it is already sorted such as a binary search tree or binary heap, there may be a more efficient way to determine the minimum element.

5. If get_min() is called frequently, cache the min by adding a field to storage_t and modifying other code around it; frequently used strategy such as in Java’s String class for hashCode() to get $O(1)$ lookup.

```c
int get_min(storage_t *st){
    int *arr = malloc(sizeof(int)*get_size(st));
    for(int i=0; i<get_size(st); i++){
        arr[i] = get_element(st,i);
    }
    bubble_sort(arr, get_size(st));
    int ans = arr[0];
    free(arr);
    return ans;
}
```
Exercise: Eliminating Unnecessary Work

void lower1(char *s) {
    for (long i=0; i < strlen(s); i++){
        if (s[i] >= 'A' && s[i] <= 'Z'){
            s[i] -= ('A' - 'a');
        }
    }
}

void lower2(char *s) {
    long len = strlen(s);
    for (long i=0; i < len; i++){
        if (s[i] >= 'A' && s[i] <= 'Z'){
            s[i] -= ('A' - 'a');
        }
    }
}

▶ Bryant/O’Hallaron Figure 5.7
▶ Two versions of a lower-casing function
▶ Lowercase by subtracting off constant for uppercase characters: alters ASCII code
▶ Examine them to determine differences
▶ Project speed differences and why one will be faster
Answers: Eliminating Unnecessary Work

- `strlen()` is $O(N)$: searches for \0 character in for() loop
- Don’t loop with it if possible

```c
void lower1(char *s) {
    for (long i=0; i < strlen(s); i++){
        if (s[i] >= 'A' && s[i] <= 'Z'){
            s[i] -= ('A' - 'a');
        }
    }
}
```

```c
void lower2(char *s) {
    long len = strlen(s);
    for (long i=0; i < len; i++){
        if (s[i] >= 'A' && s[i] <= 'Z'){
            s[i] -= ('A' - 'a');
        }
    }
}
```

```c
long strlen(char *s) {
    long len = 0;
    while(s[len] != '\0'){
        len++;
    }
    return len;
}
```
Exercise: Do Memory References Matter?

```c
void sum_range1(int start,
    int stop,
    int *ans)
{
    *ans = 0;
    for(int i=start; i<stop; i++){
        *ans += i;
    }
}

void sum_range2(int start,
    int stop,
    int *ans)
{
    int sum = 0;
    for(int i=start; i<stop; i++){
        sum += i;
    }
    *ans = sum;
}
```

- What is the primary difference between the two routines above?
- What effect if any will this have on runtime?
Answers: Do Memory References Matter?

- `sum_range1()` makes repeated memory references
- `sum_range2()` uses a local variable with only a couple memory references

```c
void sum_range1(int start, int stop, int *ans) {
    *ans = 0;
    for(int i=start; i<stop; i++) {
        *ans += i; // main mem ref
    }
}
```

```c
void sum_range2(int start, int stop, int *ans) {
    int sum = 0; // likely register
    for(int i=start; i<stop; i++) {
        sum += i; // add to register
    }
    *ans = sum; // one main-mem ref
}
```

Primary difference is repeated access to Main Memory VS Register, this should indicate `sum_range2()` performs better BUT…
Memory References Matter, Compiler May Change Them

```
lila> gcc -Og sum_range.c  # Limit opt
lila> ./a.out 0 1000000000
sum_range1: 1.9126e+00 secs
sum_range2: 2.6942e-01 secs

- Minimal optimizations
- Memory reference definitely matters

lila> gcc -O1 sum_range.c  # Opt plz!
lila> ./a.out 0 1000000000
sum_range1: 2.8972e-01 secs
sum_range2: 2.7569e-01 secs

- Observe code differences between -Og and -O1
- Why is performance improved so much?
```

### Compiled with -Og: limited opt
```
sum_range1:
  movl  $0, (%rdx)  # init MEMORY
  jmp   .LOOTOP
.BODY:
  addl  %edi, (%rdx)  # MEMORY add
  addl  $1, %edi     # in loop
.LOOPTOP:
  cmpl  %esi, %edi
  jl    .BODY
  ret
```

### Compiled with -O1: some opt
```
sum_range1:
  cmpl  %esi, %edi
  jge   .END
  movl  $0, %eax  # init REGISTER
.LOOP:
  addl  %edi, %eax  # REGISTER add
  addl  $1, %edi   # in loop
  cmpl  %edi, %esi
  jne   .LOOP
  movl  %eax, (%rdx)  # MEMORY write
.END:
  ret
```
Dash-O! Compiler Optimizes for You

- gcc can perform many **micro-optimizations**, almost NEVER macro optimizations
- Series of `-Ox` options authorize use of various micro-opts
- We will use `-Og` at times to disable many optimizations
  - `-Og`: Optimize debugging: “…offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.”
- Individual optimizations can be enabled and disabled

- `-0` or `-01`: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With `-0`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- `-02`: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-0`, this option increases both compilation time and the performance of the generated code.
- `-03`: Optimize yet more. `-03` turns on all optimizations specified by `-02` and also…
- `-0fast`: Disregard strict standards compliance. (!)
gcc -O or gcc -O1 turns on the following optimization flags:

- fauto-inc-dec -fbranch-count-reg -fcombine-stack-adjustments
--fcompare-elim fcprop-registers -fdce -fdefer-pop -fdelayed-branch
--fdse -ffforward-propagate fguess-branch-probability -fif-conversion2
--fif-conversion finline-functions-called-once -fipa-pure-const
--fipa-profile -fipa-reference fmerge-constants -fmove-loop-invariants
--freorder-blocks -fshrink-wrap fshrink-wrap-separate
--fsplit-wide-types -fssa-backprop -fssa-phiopt -ftree-bit-ccp
-ftree-cmp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-dce
-ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre
--ftree-phiprop -ftree-sink ftree-slsr -ftree-sra -ftree-pta
--ftree-ter -funit-at-a-time

- Some combination of these enables \texttt{sum\_range2()} to fly as fast as \texttt{sum\_range1()}

- We will look at some “by-hand” versions of these optimizations but let the compiler optimize for you whenever possible
Exercise: Allocation and Hidden Costs

Consider the following **Java** code

```java
public class StringUtils{
    public static String repString(String str, int reps)
    {
        String result = "";
        for(int i=0; i<reps; i++){
            result = result + str;
        }
        return result;
    }
}
```

- Give a Big-O estimate for the runtime
- Give a Big-O estimate for the memory overhead
Strings are **immutable** in Java (Python, many others)

Each iteration must
- **allocate** new memory for a new string sized `result.length + str.length`
- Copy `result` to the first part
- Copy `str` to the second part

Leads to $O(N^2)$ complexity

Much worse memory usage: as much as $O(N^2)$ wasted memory for garbage collector to clean up

```java
public class StringUtils{
    public static String repString(String str, int reps) {
        String result = "";
        for(int i=0; i<reps; i++){
            result = result + str;
        }
        return result;
    }

    // Efficient version
    public static String repString2(String str, int reps) {
        StringBuilder result = new StringBuilder();
        for(int i=0; i<reps; i++){
            result.append(str);
        }
        return result.toString();
    }
}
```
Exercise: Loop Unrolling

- Have seen copying loop iterations manually may lead to speed gains

**Why?** Which of the following unrolled versions of `sum_rangeX()` seems fastest?

**Why the second loop** in `sum_rangeB()` and `sum_rangeC()`?

```c
void sum_rangeA(long stop, long *ans){
    long sum=0, i;
    for(i=0; i<stop; i++){
        sum += i+0;
    }
    *ans = sum;
}

void sum_rangeB(long stop, long *ans){
    long sum = 0, i;
    for(i=0; i<stop-3; i+=3){
        sum += (i+0);
        sum += (i+1);
        sum += (i+2);
    }
    for(; i<stop; i++){
        sum += i;
    }
    *ans = sum;
}

void sum_rangeC(long stop, long *ans){
    long sum0=0, sum1=0, sum2=0, i;
    for(i=0; i<stop-3; i+=3){
        sum0 += (i+0);
        sum1 += (i+1);
        sum2 += (i+2);
    }
    for(; i<stop; i++){
        sum0 += i;
    }
    *ans = sum0 + sum1 + sum2;
}
```
Exercise: Loop Unrolling

Expectations

<table>
<thead>
<tr>
<th>Version</th>
<th>Notes</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum_rangeA()</td>
<td>Not unrolled</td>
<td>Baseline</td>
</tr>
<tr>
<td>sum_rangeB()</td>
<td>Unroll x3, same destinations for sum</td>
<td>Less good</td>
</tr>
<tr>
<td>sum_rangeC()</td>
<td>Unroll x3, different destinations sum add</td>
<td>Expected Best</td>
</tr>
</tbody>
</table>

Actual Performance

apollo> gcc -Og unroll.c
apollo> ./a.out 1000000000
sum_rangeA: 1.0698e+00 secs
sum_rangeB: 6.2750e-01 secs
sum_rangeC: 6.2746e-01 secs

phaedrus> ./a.out 1000000000
sum_rangeA: 2.8913e-01 secs
sum_rangeB: 5.3285e-01 secs
sum_rangeC: 2.6774e-01 secs

Unrolling is Unpredictable

- Performance Gains vary from one compiler+processor to another
- All unrolling requires cleanup loops like those in the B/C versions: add on remaining elements
GCC Options to Unroll

- gcc has options to unroll loops during optimization
- Unrolling has unpredictable performance implications so unrolling is **not enabled** for -O1, -O2, -O3
- Can manually enable it with compiler options like -funroll-loops to check for performance bumps

```
apollo> gcc -Og unroll.c
## limited compiler opts
apollo> ./a.out 1000000000
sum_rangeA: 1.0698e+00 secs
sum_rangeB: 6.2750e-01 secs
sum_rangeC: 6.2746e-01 secs

apollo> gcc -Og -funroll-loops unroll.c
## loops unrolled by compiler
apollo> ./a.out 1000000000
sum_rangeA: 7.0386e-01 secs
sum_rangeB: 6.2802e-01 secs
sum_rangeC: 6.2797e-01 secs

apollo> gcc -Og -funroll-loops -fvariable-expansion-in-unroller unroll.c
apollo> ./a.out 1000000000  ## loops unrolled + multiple intermediates used
sum_rangeA: 5.2711e-01 secs
sum_rangeB: 6.2759e-01 secs
sum_rangeC: 6.2750e-01 secs
```

```
apollo> gcc -O3 unroll.c
## Many opts, no unrolling
apollo> ./a.out 1000000000
sum_rangeA: 9.4124e-01 secs
sum_rangeB: 4.1833e-01 secs
sum_rangeC: 4.1832e-01 secs
```

18
Do Conditionals Matter?

Consider two examples of adding even numbers in a range

```
1 // CONDITION version
2 long sum_evensA(long start, long stop){
3     long sum=0;
4     for(int i=start; i<stop; i++){
5         if((i & 0x01) == 0){
6             sum += i;
7         }
8     }
9     return sum;
10 }
11 // STRAIGHT-LINE version
12 long sum_evensB(long start, long stop){
13     long sum=0;
14     for(int i=start; i<stop; i++){
15         int odd = i & 0x01;
16         int even_mask = odd - 1;
17         // 0x00000000 for odd
18         // 0xFFFFFFFF for even
19         sum += even_mask & i;
20     }
21     return sum;
22 }
```

Timings for these two are shown below at two levels of optimization.

```
lila> gcc -Og condloop.c
lila> a.out 0 400000000
sum_evensA: 1.1969e+00 secs
sum_evensB: 2.8953e-01 secs
# 4x speedup

lila> gcc -O3 condloop.c
lila> a.out 0 400000000
sum_evensA: 2.3662e-01 secs
sum_evensB: 9.6242e-02 secs
# 2x speedup
```

Message is simple: eliminate conditionals whenever possible to improve performance.
Exercise: Row Sums with Function v Macro

How is a macro different from a function call?

Which of the below codes will run faster and why?

```c
1 int mget(matrix_t mat,
2    int i, int j)
3 {
4   return
5    mat.data[i*mat.cols + j];
6 }
7 int vset(vector_t vec,
8    int i, int x)
9 {
10   return vec.data[i] = x;
11 }
12 void row_sumsA(matrix_t mat,
13    vector_t sums)
14 {
15   for(int i=0; i<mat.rows; i++){
16     int sum = 0;
17     for(int j=0; j<mat.cols; j++){
18       sum += mget(mat,i,j);
19     }
20     vset(sums, i, sum);
21   }
22 }
```

```c
#define MGET(mat,i,j) ((mat).data[((i)*((mat).cols)) + (j)])

#define VSET(vec,i,x) ((vec).data[(i)] = (x))

void row_sumsB(matrix_t mat,
vector_t sums)
{
   for(int i=0; i<mat.rows; i++){
      int sum = 0;
      for(int j=0; j<mat.cols; j++){
         sum += MGET(mat,i,j);
      }
      VSET(sums, i, sum);
   }
}
```
Answers: Row Sums with Function v Macro

- `row_sumsA()` uses standard function calls to retrieve elements
- `row_sumsB()` uses macros to do the element retrieval
- A macro is a textural expansion done by the preprocessor: insert the literal text associated with the macro
- See macro results with `gcc -E func_v_macro.c` which stops after preprocessor step (early)
- Function calls cost some operations but not many
- Function calls prevent optimization across boundaries
- Cannot pipeline effectively when jumping around, using registers for arguments, restoring registers, etc
- Macros can alleviate this but they are a pain to write and notoriously buggy
- Better to let the compiler do this for us
Inlining Functions/Procedures

- **Function Inlining** inserts the body of a function where it would have been called.
- Turned on fully partially at -O2 and fully at -O3.
- Enables other optimizations blocked by function boundaries.
- Can only be done if source code (C file) for function is available.
- Like loop unrolling, function inlining has trade-offs:
  - Enables pipelining.
  - More predictable control.
  - More register pressure.
  - Increased code size.

```
val> FILES="func_v_macro.c matvec_util.c"
val> gcc -Og $FILES
val> ./a.out 16000 8000
row_sums_FUNC: 2.8037e-01 secs
row_sums_MACRO: 9.2829e-02 secs

val> gcc -Og -finline-small-functions $FILES
val> ./a.out 16000 8000
row_sums_FUNC: 1.3620e-01 secs
row_sums_MACRO: 1.2969e-01 secs

val> gcc -O3 $FILES
val> ./a.out 16000 8000
row_sums_FUNC: 3.1132e-02 secs
row_sums_MACRO: 3.6975e-02 secs

- Inlining typically most effective for small functions (getters/setters).
```
Profilers: gprof and Friends

▶ **Profiler:** a tool that monitors code execution to enable performance optimizations

▶ **gprof** is stock on Linux systems, interfaces with gcc

▶ **Compile with profiling options:** gcc -pg

▶ **Run code to produce data file**

▶ **Examine with gprof**

▶ **Note:** gcc version 6 and 7 contain a bug requiring use of -no-pie option, not a problem on apollo

```
# Compile
# -pg : instrument code for profiling
# -no-pie : bug fix for new-ish gcc's
> gcc -pg -no-pie -g -Og -o unroll unroll.c

> ls
unroll unroll.c

> ./unroll 1000000000
sum_rangeA: 2.9401e-01 secs
sum_rangeB: 5.3164e-01 secs
sum_rangeC: 2.6574e-01 secs

# gmon.out now created with timing info
> ls
  gmon.out unroll unroll.c

> file gmon.out
gmon.out: GNU prof performance data

> gprof -b unroll
... output on next slide ...
```
gprof output for unroll

> gprof -b unroll
Flat profile:
Each sample counts as 0.01 seconds.

% cumulative  self   self  total
     time  seconds seconds  calls  ms/call  ms/call  name
50.38  0.54   0.54   1  544.06  544.06  sum_rangeB
26.12  0.83   0.28   1  282.11  282.11  sum_rangeA
24.26  1.09   0.26   1  261.95  261.95  sum_rangeC

Call graph

index  %   self   children  called name
[1]  100.0 0.00  1.09    main [1]
 0.54  0.00  1/1       sum_rangeB [2]
 0.28  0.00  1/1       sum_rangeA [3]
 0.26  0.00  1/1       sum_rangeC [4]

[2]  50.0 0.54  0.00    1       sum_rangeB [2]

[3]  25.9 0.28  0.00    1       sum_rangeA [3]

[4]  24.1 0.26  0.00    1       sum_rangeC [4]
gprof Example: Dictionary Application

```plaintext
> ./dictionary < craft-67.txt
... Total time = 0.829561 seconds
> gprof -b dictionary

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>cumulative seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.07</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>1</td>
<td>180.25</td>
<td>180.25</td>
<td>sort_words</td>
</tr>
<tr>
<td>19.47</td>
<td>0.25</td>
<td>0.07</td>
<td>0.07</td>
<td>463016</td>
<td>0.00</td>
<td>0.00</td>
<td>find_ele_rec</td>
</tr>
<tr>
<td>13.91</td>
<td>0.30</td>
<td>0.05</td>
<td>0.05</td>
<td>2862749</td>
<td>0.00</td>
<td>0.00</td>
<td>Strlen</td>
</tr>
<tr>
<td>8.34</td>
<td>0.33</td>
<td>0.03</td>
<td>0.03</td>
<td>463016</td>
<td>0.00</td>
<td>0.00</td>
<td>lower1</td>
</tr>
<tr>
<td>2.78</td>
<td>0.34</td>
<td>0.01</td>
<td>0.01</td>
<td>463017</td>
<td>0.00</td>
<td>0.00</td>
<td>get_token</td>
</tr>
<tr>
<td>2.78</td>
<td>0.35</td>
<td>0.01</td>
<td>0.01</td>
<td>463016</td>
<td>0.00</td>
<td>0.00</td>
<td>h_mod</td>
</tr>
<tr>
<td>2.78</td>
<td>0.36</td>
<td>0.01</td>
<td>0.01</td>
<td>20451</td>
<td>0.00</td>
<td>0.00</td>
<td>save_string</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>463017</td>
<td>0.00</td>
<td>0.00</td>
<td>get_word</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>463016</td>
<td>0.00</td>
<td>0.00</td>
<td>insert_string</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>20451</td>
<td>0.00</td>
<td>0.00</td>
<td>new_ele</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>7</td>
<td>0.00</td>
<td>0.00</td>
<td>add_int_option</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>add_string_option</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>init_token</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>new_table</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>parse_options</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>show_options</td>
</tr>
<tr>
<td>0.00</td>
<td>0.36</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>360.50</td>
<td>word_freq</td>
</tr>
</tbody>
</table>
```
gprof Example Cont’d: Dictionary Application

```bash
> ./dictionary < craft-67.txt  ## After upgrading sort_words() to qsort()  
... Total time = 0.624172 seconds
> gprof -b dictionary
```

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>60.08</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>15.02</td>
<td>0.15</td>
<td>0.03</td>
</tr>
<tr>
<td>10.01</td>
<td>0.17</td>
<td>0.02</td>
</tr>
<tr>
<td>5.01</td>
<td>0.18</td>
<td>0.01</td>
</tr>
<tr>
<td>5.01</td>
<td>0.19</td>
<td>0.01</td>
</tr>
<tr>
<td>5.01</td>
<td>0.20</td>
<td>0.01</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Exercise: Quick Review

1. What’s the first thing to consider when optimization seems necessary?
2. What kinds of optimizations would have the biggest impact on performance?
3. What is the smartest way to “implement” micro-optimizations, to get their benefit with minimal effort?
Answers: Quick review

1. What’s the first thing to consider when optimization seems necessary?
   A: Is optimization really necessary? Or is there something else that would be more worth the effort (e.g. fixing bugs, adding features, improving documentation, etc.)

2. What kinds of optimizations would have the biggest impact on performance?
   A: From most to least important
   - Algorithms and Data Structure Selection
   - Elimination of unneeded work/hidden costs
   - Memory Utilization
   - Micro-optimizations (today’s lecture)

3. What is the smartest way to “implement” micro-optimizations, to get their benefit with minimal effort?
   A: Use the compiler to mechanically perform code transforms to achieve micro-optimizations. Using -O2 will produce faster-running code because the compiler is transforming generated assembly instructions from C sources.