CS 6015 – Software Engineering
Adam Miles
Due Date: April 24th 2018

Overview:

A brief description of the implementation of 3 unique popcount methods, and a closer look at the optimized assembly instructions.

Compiler optimizations -O3.

I used:

```
gcc -S -O3 -march=native popcount.c
```

The results are the following, on my machine (N = 10000).

136.300000 ns / op : popcount_1_data
354.300000 ns / op : popcount_1_control
30.800000 ns / op : popcount_4_data
201.500000 ns / op : popcount_4_control
17.000000 ns / op : popcount_8_data
14.700000 ns / op : popcount_16_data
126.100000 ns / op : popcount_kernighan
7.900000 ns / op : popcount64a
7.100000 ns / op : popcount64b
4.500000 ns / op : popcount64c
3.300000 ns / op : popcount64_fast

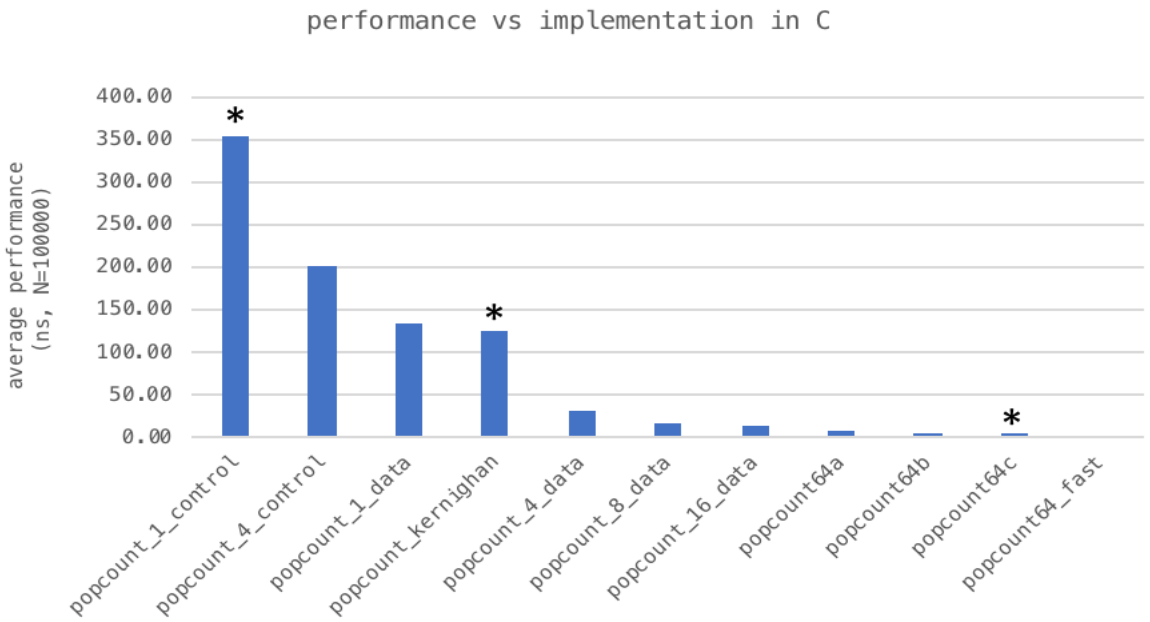

Figure 1 - ranked performance (N = 10). * denotes method selected for study.

After analyzing each of the 3 selected methods, they were implemented and tested in 4 different languages. I attempted to put together a comparison of the performance of each.

The results are more revealing of how each language performs from the perspective of a novice programmer, who does not have a grasp of the languages to utilize them optimally. But the comparison is no less interesting from a performance perspective, in that context.

Languages:

GO was selected, since it seems to be used quite heavily on the backend at software shops like dropbox.

RUST was selected, since out of all the languages I researched it appeared to be a very capable systems-level language.

Python was selected out of morbid curiosity and ease of implementation, since it is a pretty capable general-purpose language but not particularly well suited to this project. I did not expect it to perform very well.

Javascript was selected since it is ubiquitous. It is even showing up in embedded systems thanks to the proliferation of IOT devices. It seems to be displacing Lua as a go to scripting language for embedded applications, in some specific cases.

System:

All methods tests on:

Hardware Overview:

```
  Model Name:            MacBook Pro
  Model Identifier:      MacBookPro14,1
  Processor Name:        Intel Core i5
  Processor Speed:       2.3 GHz
  Number of Processors:  1
  Total Number of Cores: 2
  L2 Cache (per Core):   256 KB
  L3 Cache:              4 MB
  Memory:                8 GB
```

PART 1

Method 1:

```
int popcount_1_control(uint64_t x) {
  int c = 0;
  for (int i = 0; i < 64; i++) {
    if (x & 1)
      c++;
    x >>= 1;
  }
  return c;
}
```

I chose to examine this method since it is easy to reason about and provides a clear, legible, naive baseline (sans extra trickiness). I opted for the control instead of the data implementation as it was the slowest. It moves through a uint64_t one bit at a time, bitwise-anding the lowest but with one and incrementing a counter if true (1&1).

A snippet of the optimized assembly instructions shows that the compiler did some serious loop unrolling.

```
LCPI1_0:
      .quad 6                         ## 0x6
      .quad 5                         ## 0x5
      .quad 4                         ## 0x4
      .quad 3                         ## 0x3
LCPI1_2:
      .quad 10                        ## 0xa
      .quad 9                         ## 0x9
      .quad 8                         ## 0x8
      .quad 7                         ## 0x7
LCPI1_3:
      .quad 14                        ## 0xe
      .quad 13                        ## 0xd
      .quad 12                        ## 0xc
      .quad 11                        ## 0xb
.
.
.
LCPI1_15:
      .quad 54                        ## 0x36
      .quad 53                        ## 0x35
      .quad 52                        ## 0x34
      .quad 51                        ## 0x33
      .section    __TEXT,__literal4,4byte_literals
      .p2align    2
LCPI1_1:
```

```
      .long 1                         ## 0x1
      .section    __TEXT,__text,regular,pure_instructions
      .globl _popcount_1_control
      .p2align    4, 0x90
_popcount_1_control:                    ## @popcount_1_control
      .cfi_startproc
## BB#0:
      pushq %rbp
```

I think this is because it was given an explicit bound in the for loop (i < 64). There is no loop in the optimized code. Furthermore, the compiler is using quads to better break up the arithmetic. Its basically a huge set of 4 byte literal instructions. For some reason LCPI1_1: was listed out of sequence. It was the one instruction set that used a long instead of quads to perform its share of the computation.

Method 2:

```
int popcount_kernighan(uint64_t n) {
  int count = 0;
  while (n) {
    count += n & 0x1u;
    n >>= 1;
  }
  return count;
}
```

I selected the kernighan popcount method for this second routine since it is similarly brute force but is more efficient (in theory and bourne out by testing to be 50% more efficient than the naive solution). The idea behind this looks like an attempt to save work, and only iterate through the number while it has bits left that are non-zero. For the normal distribution we tested over, this improvement makes sense.

The optimized assembly instructions were not unrolled, because the conditional while() likely prevents it from plotting them out ahead of time. The inner loop operations are pretty straightforward, and break down almost explicitly based on the code. It performs the right hand side & operation, adds the result to the count register, then performs the right-shift before jumping back to the top of the loop. testq %rdi, %rdi is the condition check (comparing to 0, equivalent to cmp %rdx, 0).

```
_popcount_kernighan:                    ## @popcount_kernighan
      .cfi_startproc
## BB#0:
      pushq %rbp
Lcfi20:
      .cfi_def_cfa_offset 16
Lcfi21:
```

```
      .cfi_offset %rbp, -16
      movq  %rsp, %rbp
Lcfi22:
      .cfi_def_cfa_register %rbp
      xorl  %eax, %eax
      testq %rdi, %rdi
      je    LBB6_2
      .p2align    4, 0x90
LBB6_1:                                 ## =>This Inner Loop Header: Depth=1
      movl  %edi, %ecx
      andl  $1, %ecx
      movl  %eax, %eax
      addq  %rcx, %rax
      shrq  %rdi
      jne   LBB6_1
LBB6_2:
                                        ## kill: %EAX<def> %EAX<kill> %RAX<kill>
      popq  %rbp
      retq
      .cfi_endproc
                                        ## -- End function
      .globl_popcount64a            ## -- Begin function
```

Method 3:

```
static const uint64_t m1  = 0x5555555555555555;
static const uint64_t m2  = 0x3333333333333333;
static const uint64_t m4  = 0x0f0f0f0f0f0f0f0f;
static const uint64_t h01 = 0x0101010101010101;


int popcount64c(uint64_t x) {
    x -= (x >> 1) & m1;
    x = (x & m2) + ((x >> 2) & m2);
    x = (x + (x >> 4)) & m4;
    return (x * h01) >> 56;
}
```

This method took a little longer to figure out. Still not certain I completely get it, despite having stepped through it in an IDE a number of times. It uses a series of uint64_t constants, to help mask and calculate the number of set bits, and moves the collected results down . The result seems pretty eloquent, and something the compiler could really sink its teeth into since its a series of explicit computations regardless of the input. The leaq sneaks in, and serves as an addition it looks like.

I focused on popcount64c, since it has the fewest number of arithmetic operations (according to the wiki, of any known implementation for systems with fast mul). I

thought it would be interesting to check out how this fast routine would port across the various languages.

```
_popcount64c:                           ## @popcount64c
      .cfi_startproc
## BB#0:
      pushq %rbp
Lcfi29:
      .cfi_def_cfa_offset 16
Lcfi30:
      .cfi_offset %rbp, -16
      movq  %rsp, %rbp
Lcfi31:
      .cfi_def_cfa_register %rbp
      movq  %rdi, %rax
      shrq  %rax
      movabsq     $6148914691236517205, %rcx ## imm = 0x5555555555555555
      andq  %rax, %rcx
      subq  %rcx, %rdi
      movabsq     $3689348814741910323, %rax ## imm = 0x3333333333333333
      movq  %rdi, %rcx
      andq  %rax, %rcx
      shrq  $2, %rdi
      andq  %rax, %rdi
      addq  %rcx, %rdi
      movq  %rdi, %rax
      shrq  $4, %rax
      leaq  (%rax,%rdi), %rax
      movabsq     $1085102592571150095, %rcx ## imm = 0xF0F0F0F0F0F0F0F
      andq  %rax, %rcx
      movabsq     $72340172838076673, %rax ## imm = 0x101010101010101
      imulq %rcx, %rax
      shrq  $56, %rax
                                        ## kill: %EAX<def> %EAX<kill> %RAX<kill>
      popq  %rbp
      retq
      .cfi_endproc
                                        ## -- End function
      .globl _popcount64_fast        ## -- Begin function popcount64_fast
      .p2align    4, 0x90
```

Looking through the assembly, we can see that it pretty much directly translates the const uint64_t to immediate constants (as seen in the movabsq). The LHS and substraction are handled next (andq, subq). From there it proceeds directly through the rest of the arithmetic operations and returns the collected total of set bits (in the top 8 bits I believe). This set of only 12 explicit arithmetic ops (only one multiply) per run very clearly describes why this method is miles ahead of the others in performance. The solution to the problem was distilled into a form that makes best use of what the CPU is good at.

PART 2

For each of the languages selected, I implemented the tests as closely to the C version as possible. I did not subtract out the loop overhead, it is included in the total run time. I did however run the tests explicitly, rather than iterate through a list of implementations.

GO

One of the things I learned about go is that there are no asserts! Apparently this is something of a hot button issue. This was their rationale:

" Go doesn't provide assertions. They are undeniably convenient, but our experience has been that programmers use them as a crutch to avoid thinking about proper error handling and reporting. Proper error handling means that servers continue operation after non-fatal errors instead of crashing. Proper error reporting means that errors are direct and to the point, saving the programmer from interpreting a large crash trace. Precise errors are particularly important when the programmer seeing the errors is not familiar with the code."

Overall the language was easy to pick up and run with.

Within my popcount/ directory in the CS 2015 repository, run the following command to see the test results for N=10000.

```
>> go run popcount.go

>> popcount_1_control   215 ns / op
>> popcount_kernighan   40 ns / op
>> popcount64c          3 ns / op
```

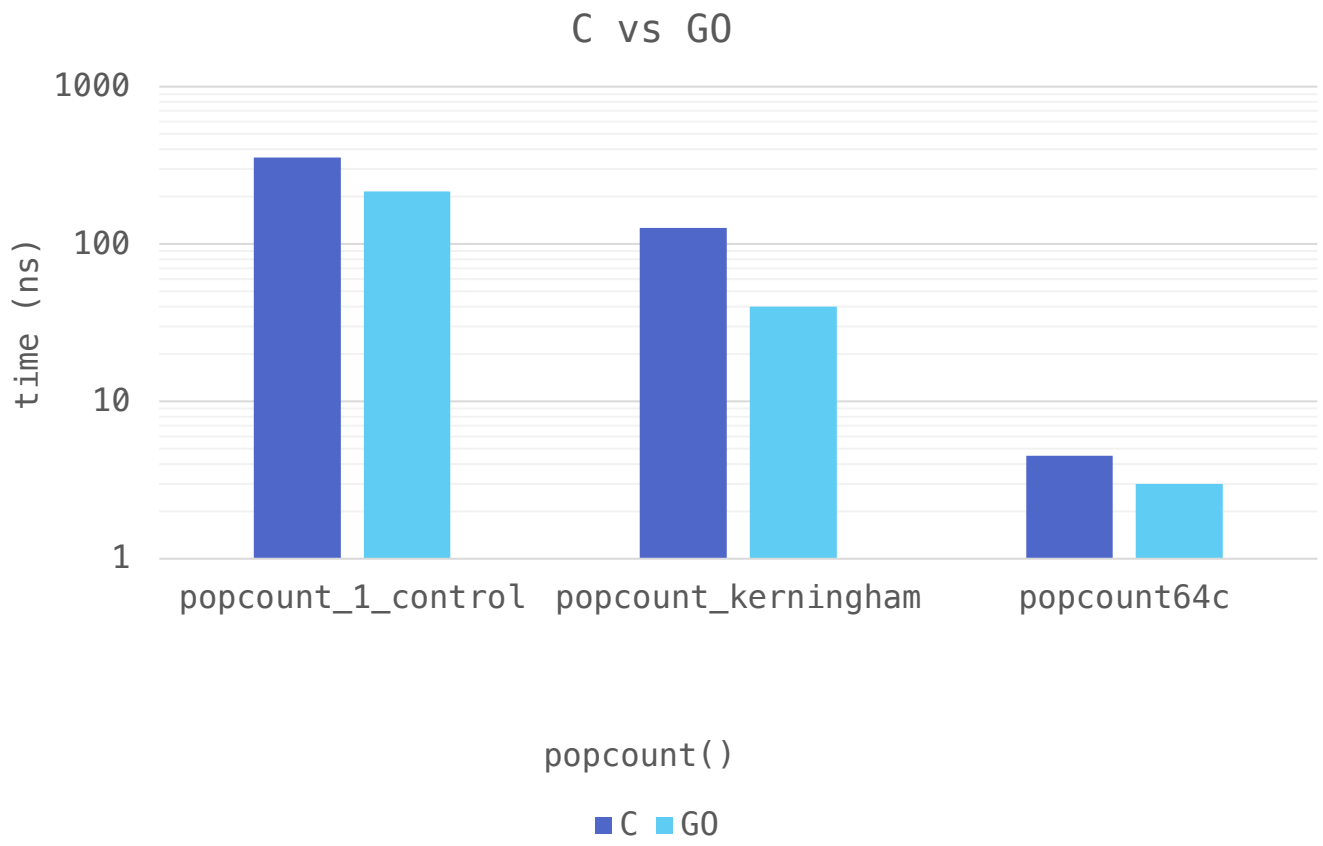

Figure 2 - C vs Go results for average popcount() over N=100000. Y-axis is log scale.

Interestingly, the results were a little better than C, likely because of the extra function call. This really shows that Go can be a highly performant choice for systems level projects like this.

PYTHON

Python was initially a bit of a nightmare for this purpose. I mistakenly thought it would be the easiest to get up and running with. Vanilla Python lacks a uint64 data type. A preliminary survey said that Numpy was relied on to provide a more complete set of data types to work with. It also provided the necessary bitwise operators for the uint64 types, which were incompatible with the generic python operators.

There are a number of bug reports still open. Many of them involve manually handling uint64s, especially with bitwise operations. Google really led me astray on this one, once people seemed to be convinced that numpy was the way to go to handle this type.

After talking with Matt in class, he showed me a way to use random to get a collection of random bits that was amenable to the standard bit shifting operations. This improved the performance considerably, but it is still miles behind the C implementations. I assume the overhead is due to the dynamic typing and all the extra work python is doing behind the scenes (GC etc). For an application like this, I would have appreciated stronger typing.

One nice feature of the timeit() method is that GC gets turned off while it is running. But the timeit() function was not amenable to the iterable way we needed to test each popcount and aggregate the sum. I could not come up with a design that would match up with the other languages, in terms of fairness and recording the sum.

To test my implementation, run the popcount.py in the project folder.

```
>> python3 popcount.py

>> popcount_1_control    7773.463800549507ns / op
>> popcount_kernighan    7308.593799825758ns / op
>> popcount64c           690.4084999405313ns / op
```

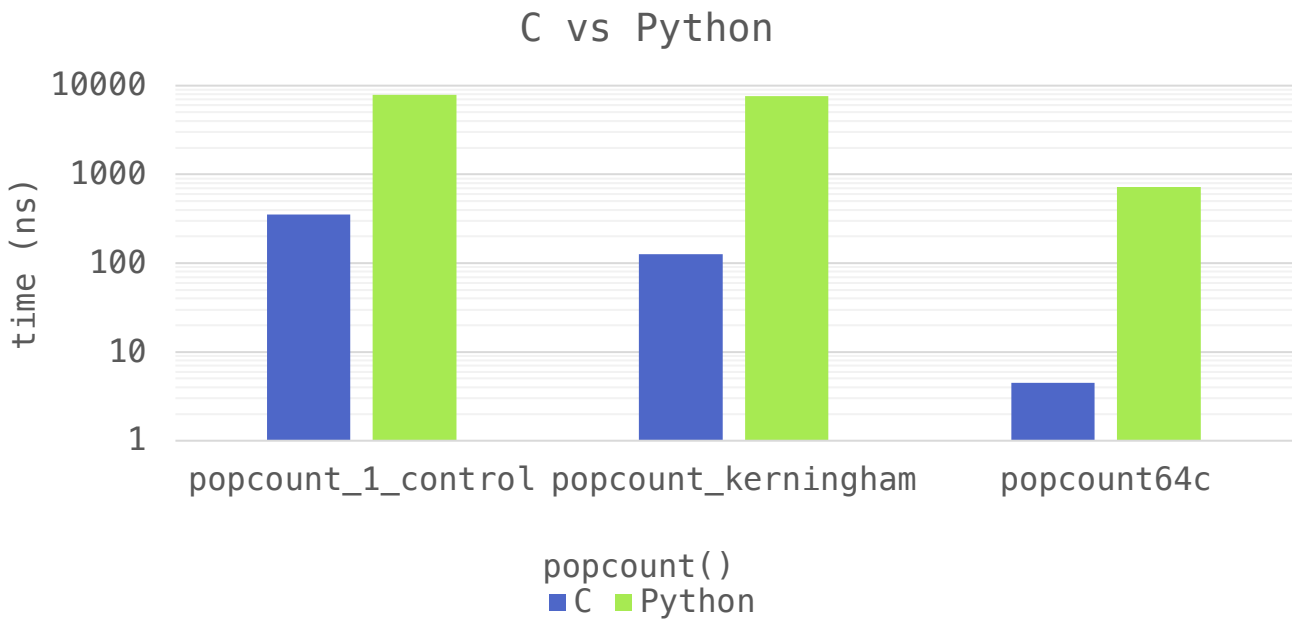

Figure 3 - Python vs C popcount methods. Y-axis is log scale.

RUST

At face value rust seemed like a great choice. It is certainly advertised as a next-gen systems language. I found it a little more difficult than Go to get up and running. In reading about it, it looks like Mozilla was trying to do something fairly ambitious: improve the performance of their browser without having to worry about some of the memory management pitfalls inherent to C++.

I was intrigued by this reddit post, under "Why Rust?" by user llogiq.

> "If you want to learn systems programming, having rustc as your coach will keep your head free to worry about other things than

> memory safety. As one famous programmer once said: Pascal is like wearing a straightjacket, C is like playing with knives, and C++ is juggling flaming chainsaws. In that metaphor, Rust is like doing parkour while suspended on strings & wearing protective gear. Yes, it will sometimes look a little ridiculous, but you'll be able to do all sorts of cool moves without hurting yourself."

Personally, I found that vanilla rust is replete with thoughtful tools that seem to address many of the pain points encountered in day to day work with C and C++. It has a nice set of testing tools built in, and a well stocked ecosystem of helpful crates. Cargo, Rust's packet manager, has features that were clearly born of some deep seeded frustration with Cmake. It seems just as powerful at the outset, and

The same approach was taken as in go-lang above: there were few a idiosyncrasies but it was really nice to have support for explicit types of the size required built directly into the language standard. It worked very hard to make sure there were no undefined behaviors (implicit conversions between type operations, explicit mutable/immutable declarations etc), which felt a little rigid at first but certainly kept the code in line.

It took me a day to get up and running, and in the end I needed serious help to get the testing working (used Bart Massey's posted test techniques as a resource after I was at least happy with the implementation of each function). I would have liked to spend more time on it, but this shortcut at least got me some results quickly and taught me a lot about Rust. The last commit that shows fully my own work is commit a95cf5d31e1bfc9e8fe9859158453d4e672fe9d1 (marked as such in commit comment).

Install cargo, and use:

```
>> cargo run
```

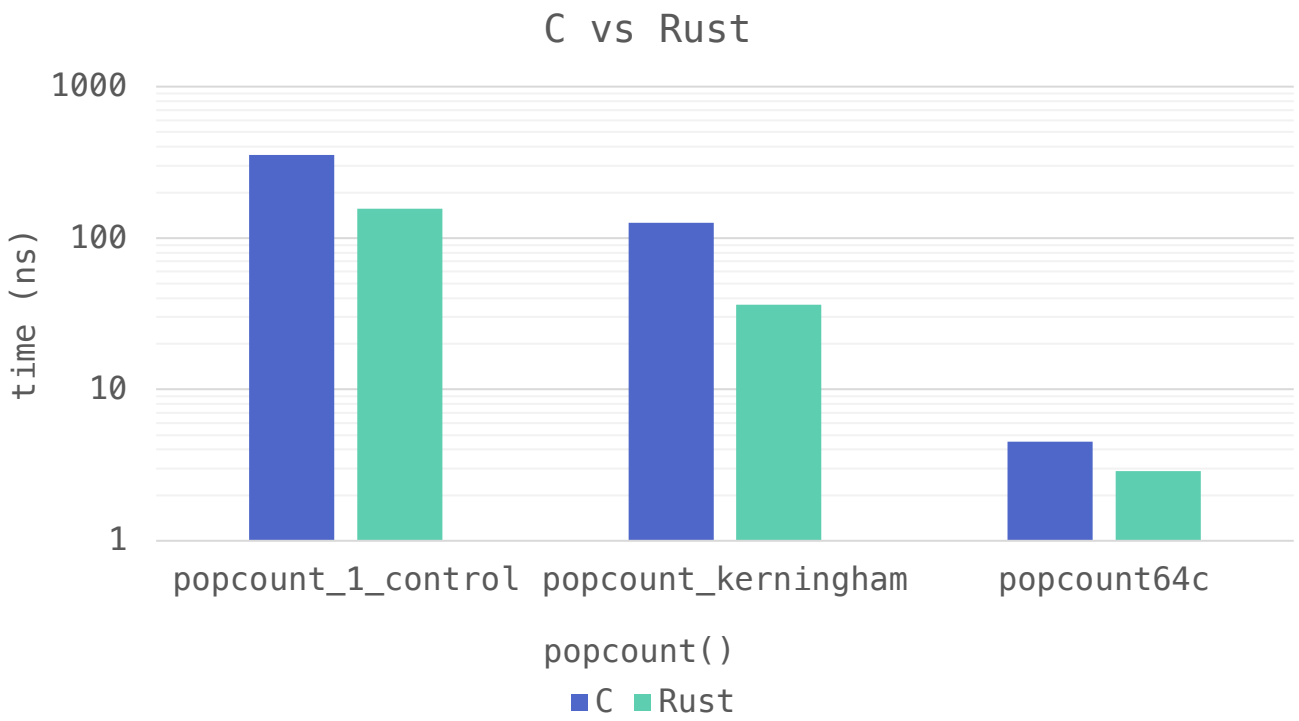

Figure 4 - Rust vs C comparison (C with extra overhead).

The results are on par with Go, and are probably very similar to C if the C code were modified slightly to remove the extra function call.

JAVASCRIPT (ES6)

Javascript was fun to work with, but to make things easier I gave in and stuck with the Uint32 type, since no Uint64's are available without a significant amount of effort. I could have at the very least worked with two Uint32's in tandem. So, the timings will not really make sense in comparison to the other systems but can serve as place holders for now.

One of the major differences, is that the Uint32 in javascript is an array object, built on the standard array. The interface is nice, but it does add some overhead. Microtiming in javascript is not as straightforward as I thought it would be. There are vulnerabilities from timing attacks, so JS relies on the browser to provide things like precision time instances. For node, this presents a problem since there is no browser object to work with. Thankfully the node folks built a substantial library for testing outside the browser environment. The method I employed helps keep the optimizer from skipping over the predictable timing liability of a microtest.

To run, Node must be installed, as well as the timing api (perf_hooks, can be installed with npm the node packet manager).

```
>> node popcount.js
>> popcount_1_control    297.07 ns / op
>> popcount_kernighan    88.35 ns / op
>> popcount64c           97.84 ns / op
```

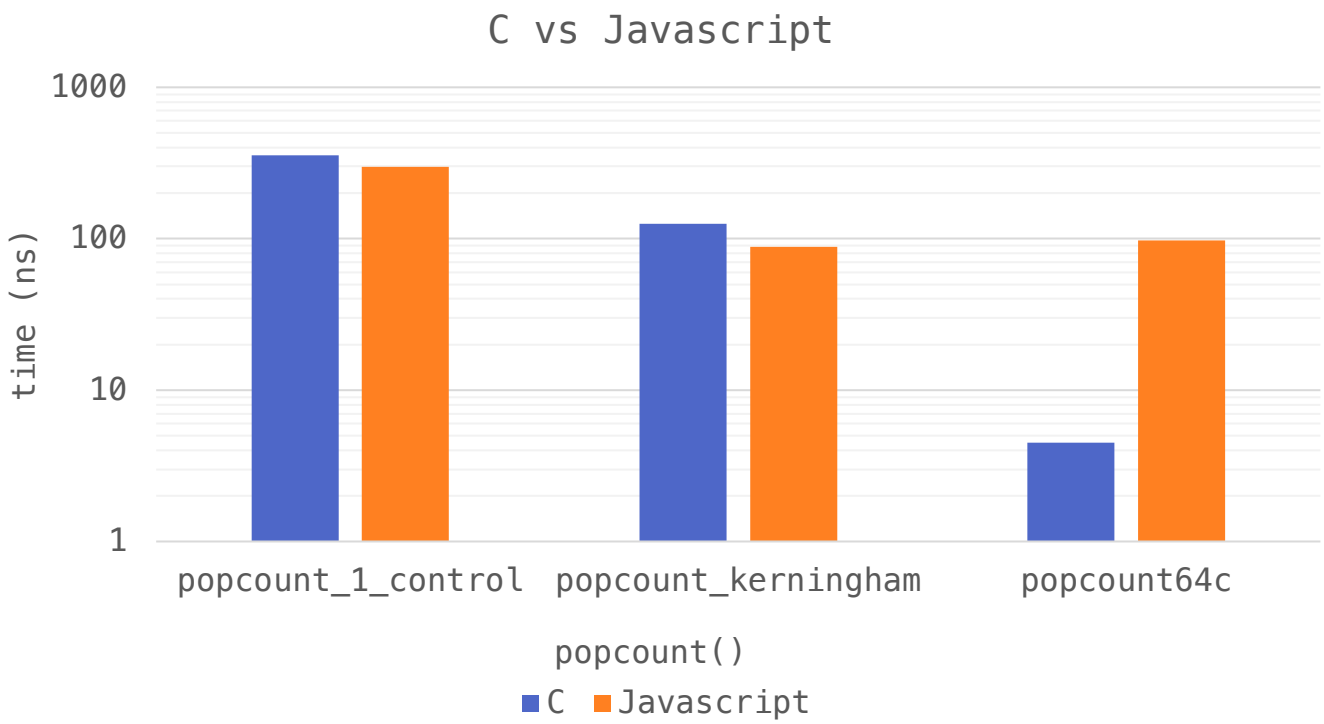

Figure 5 - Comparison of javascript Uint32 popcount methods to the C implementations.

Javascript has similar performance to the other C-like languages for the naive, array based implementations, but seems to fall behind for the fastest implementations that take advantage of the CPU friendly, efficient arithmetic operations on strongly typed ints. This might bely the array based representation of the Uint32 type in java.

SUMMARY

While I was not able to implement a particularly knowing, efficient version of each of the selected popcount methods in each language, this was the first time I had worked with Go, Rust, or Javascript (in this way). The assembly review also exposed some weaknesses in my ability to reason through what the compiler is doing, which was a great exercise. I was pleased to have some basic performance intuitions validated through testing. If there were more time, the test libraries built for each language or build environment would have been fun to explore.

Finally, it certainly reinforced the importance of understanding the breadth of systems involved when approaching a software task: thinking about everything from the capabilities and suitability of the high level languages through to the compiler and even the CPU architecture were important considerations with each implementation.

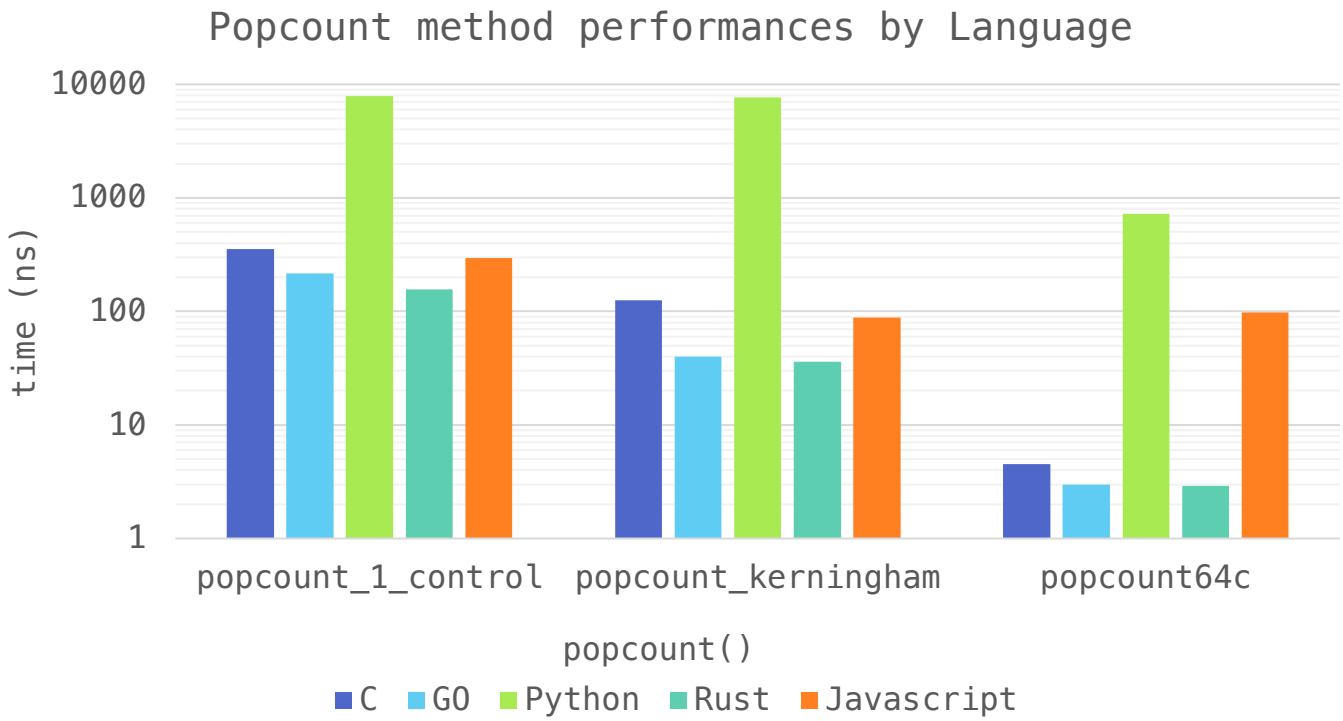

Figure 6 - Comparative performance of popcount implementations, across 5 languages.

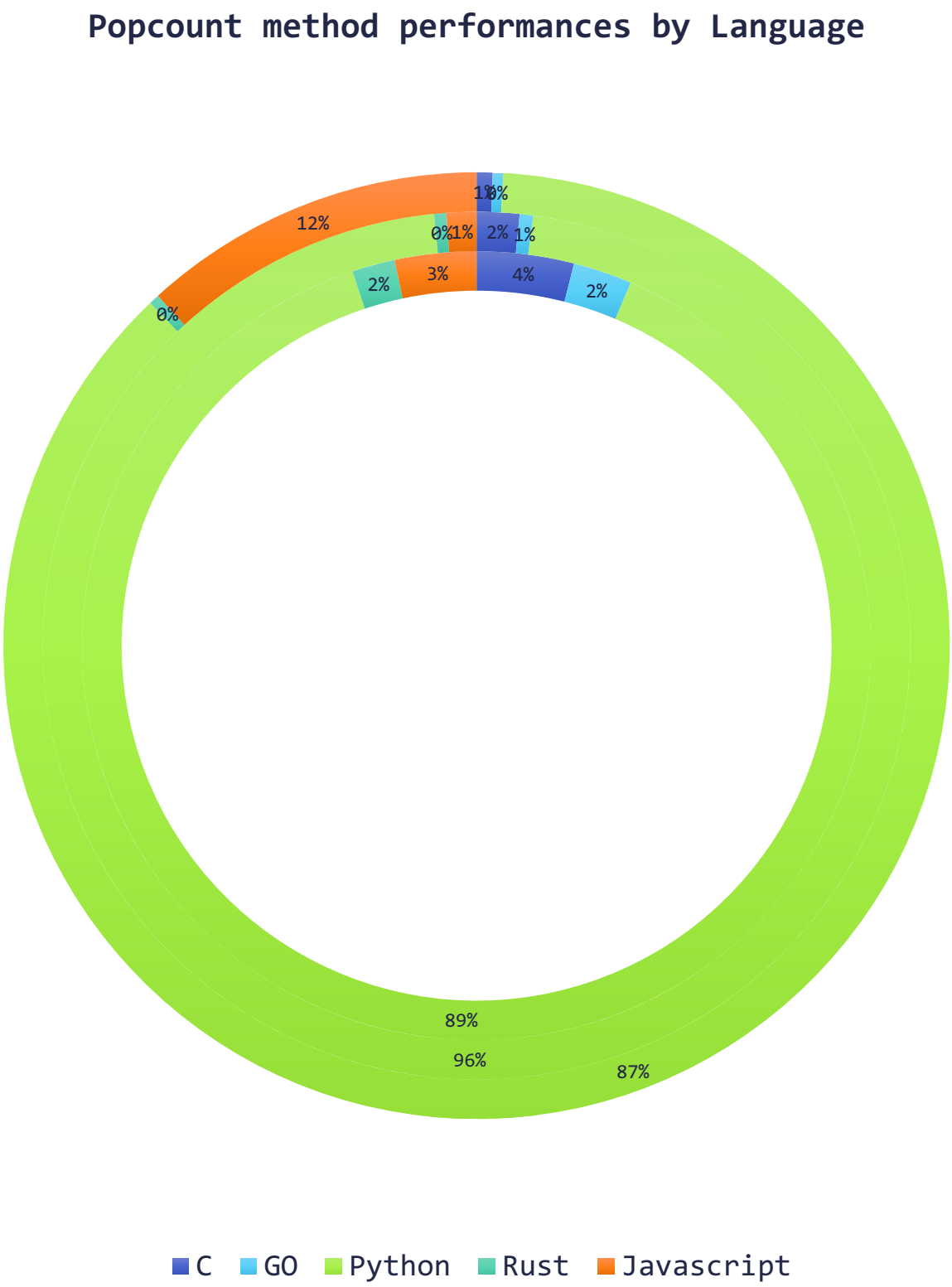

Figure 7 - Alternate comparative performance of popcount implementations, across 5 languages.