# Assignment 8: Loop Optimizations

Cayden Lund (u1182408)

4 October 2024

Repository: https://github.com/caydenlund/brainforge

## Simple Loops

As defined for assignment 8, *simple loops* are innermost loops that contain no I/O, have 0 net pointer movement, and change the cell at `mem_ptr[0]` by exactly 1 or −1. The simple nature of these loops makes them easy to flatten, and, once flattened, can in some cases allow cascading flattenings of outermore loops.

The most basic kind of simple loop is one that was an innermost loop before any changes are made to the program. These are made of a list of two kinds of instructions: `Move(stride)` and `Add(offset)`. `Move(stride)` is a shorthand for a series of `<` or `>` instructions; for example, `<<<` would be represented by `Move(-3)`, and `>><>` would be represented by `Move(2)`. Similarly, `Add(delta)` is a shorthand for a series of `-` or `+` instructions. These are flattened in a straightforward manner: for any given `Add(delta)` instruction, we get the sum of the preceding `Move(stride)` instructions as a position, and we replace the `Add(delta)` instruction with a new kind of instruction, `AddDynamic(target, delta)`, which performs the addition `mem_ptr[target] += sign * delta * mem_ptr[0]`, where `sign` is 1 if `mem_ptr[0]` is changed by exactly −1, and −1 if `mem_ptr[0]` is changed by exactly 1. At the end of the flattened loop, we also add another new instruction, `Zero`, which sets `mem_ptr[0] = 0`. Here are some examples of these basic innermost simple loops:

- `[-]` becomes `Loop([Add(-1)])`, which becomes `[Zero]`.
- `[->>+<<]` becomes `Loop([Add(-1), Move(2), Add(1), Move(-2)])`, which becomes `[AddDynamic(2, 1), Zero]`.
- `[->+>++<<<]` becomes `Loop([Add(-1), Move(1), Add(1), Move(1), Add(2), Move(-3)])`, which becomes `[AddDynamic(1, 1), AddDynamic(2, 2), Zero]`.

Here's the unoptimized (AMD64) assembly for the example `[->>++<<]`:

```
    cmpb $0, (%r12)
    je .loop_post_0
.loop_pre_0:
    addb $-1, (%r12)
    addq $2, %r12
    addb $2, (%r12)
    addq $-2, %r12
    cmpb $0, (%r12)
    jne .loop_pre_0
.loop_post_0:
```

And here's the corresponding optimized assembly for the same example:

```
movzbl (%r12), %r13d
imul $2, %r13d
addb %r13b, 2(%r12)
movb $0, (%r12)
```

For a loop that contains a simple loop, after the contained simple loop has been flattened, the outermore loop can sometimes be flattened, too. These are a little more complex because we have to consider the new `AddDynamic(target, delta)` and `Zero` instructions. In order to enforce the basic loop constraints, we need to make sure that the `Zero` instructions don't change `mem_ptr[0]`—that is, the preceding `Move(stride)` instructions must not sum to zero. We also track all of the cells that have been `Zero`-ed. If we come across an `Add(delta)` instruction that modifies a cell that has been `Zero`-ed, then we can't flatten the loop with the multiplication rules of earlier. Here are some examples of loops that can be flattened after their nested loops are flattened:

- `[->[-]<]` becomes `Loop([Add(-1), Move(1), Loop([Add(-1)]), Move(-1)])`, which becomes `Loop([Add(-1), Move(1), Zero, Move(-1)])`, which in turn becomes `[Move(1), Zero, Move(-1), Zero]`.
- `[->[-]>>+<<<]` becomes `Loop([Add(-1), Move(1), Loop([Add(-1)]), Move(2), Add(1), Move(-3)])`, which becomes `Loop([Add(-1), Move(1), Zero, Move(2), Add(1), Move(-3)])`, which in turn becomes `[Move(1), Zero, Move(-1), AddDynamic(3, 1), Zero]`.

Theoretically, we could also flatten loops that contain `AddDynamic(target, delta)` instructions with a corresponding `Zero` instruction. If we come across an `AddDynamic(target, delta)` instruction, then we need to make sure that the `target` is not `mem_ptr[0]`, and that the current position is not at `mem_ptr[0]`—because the value in cell `mem_ptr[0]` changes, we're not adding a constant each iteration, so we can't flatten with multiplication. We also need to make sure that the value in the cell at the source location isn't modified within the loop other than the corresponding `Zero` instruction. If these constraints aren't violated, then we can flatten this loop by propagating forward the `Zero` instructions as-is, and by propagating forward the `AddDynamic(target, delta)` instruction as-is. Tracking all of these constraints is tricky, and so I haven't been able to successfully implement it yet. Here is an example of a loop that could be optimized, but my optimization doesn't cover:

- `[->[->+<]<]`, which becomes `Loop([Add(-1), Move(1), Loop([Add(-1), Move(1), Add(1), Move(-1)]), Move(-1)])`, which in turn becomes `Loop([Add(-1), Move(1), AddDynamic(1, 1), Zero, Move(-1)])`, which could be optimized to `[Move(1), AddDynamic(1, 1), Zero, Move(-1), Zero]`.

## Memory Scans

A *memory scan* is a loop made of only < and > instructions, where the net change to the memory pointer inside the loop is a power of 2. These loops search for cells of memory with a stored value of zero. Modern processors have a vector arithmetic unit that's quite powerful and able to operate on regions of memory quickly. Because I'm generating AMD64 assembly code, I use AVX2 instructions to optimize these memory scans.

First, at the beginning of the body of the code, I load static masks for a stride of 1, 2, and 4 into vector registers `%ymm1`, `%ymm2`, and `%ymm4`, respectively. These masks are 32 bytes in length, and each byte has the value of 0 if it's on a "legal" index for the given stride and `0xFF` otherwise. In other words, `%ymm1` is a 32-byte-long set of zeroes because every index is legal; `%ymm2` is a 32-byte-long set of alternating bytes

between 0 (even indices) and 255 (odd indices); and `%ymm4` is a 32-byte-long set in the pattern `0`, `0xFF`, `0xFF`, `0xFF`, repeated seven more times.

Then, whenever a memory scan is performed, `%ymm0` is initialized to 32 bytes of zeroes. If the stride is positive, then 32 bytes starting at `mem_ptr[0]` are loaded into register `%ymm3`; otherwise (if negative), 32 bytes starting at `mem_ptr[-31]` are loaded into register `%ymm3`. After that, register `%ymm3` is bitwise-ored with the relevant mask for the stride. Each byte of the result is then compared with the zero vector, `%ymm0`, and the result is stored as bits in register `%eax` (i.e., 1 for match, 0 for non-match). Finally, we branch on whether any match was found: if not, we add 32 to the memory pointer and loop; otherwise, we use the `bsf` instruction (or `bsr` for reverse) to get the least-significant set bit, which is added to the memory pointer so that the memory pointer now points at the first legal cell with a value of 0.

Here's an example of the unoptimized memory scan loop, for input `[>>>>]`:

```
    cmpb $0, (%r12)
    je .loop_post_0
.loop_pre_0:
    addq $4, %r12
    cmpb $0, (%r12)
    jne .loop_pre_0
.loop_post_0:
```

And here's its optimized counterpart:

```
;# Beginning of the program---this is only loaded once:
    vmovdqu mask_1(%rip), %ymm1
    vmovdqu mask_2(%rip), %ymm2
    vmovdqu mask_4(%rip), %ymm4

;# ...

.scan_start_0:
    vmovdqu (%r12), %ymm3
    vpxor %ymm0, %ymm0, %ymm0
    vpor %ymm3, %ymm4, %ymm3
    vpcmpeqb %ymm3, %ymm0, %ymm3
    vpmovmskb %ymm3, %eax
    bsf %eax, %eax
    jnz .scan_finish_0
    addq $32, %r12
    jmp .scan_start_0
.scan_finish_0:
```

## Benchmarks

For each of the following benchmarks, I ran the compiled binary 100 times. These tests were all done on the same machine, a laptop with a Ryzen 5 5500U CPU. The median runtime is shown.

| Program | Unoptimized Runtime | Runtime with Simple Loop Opt. | Runtime with Mem. Scan Opt. | Runtime with Simple Loop + Mem. Scan Opt. |
|---|---|---|---|---|
| `bench.b` | 0.237s | 0.003s | 0.237s | 0.003s |
| `bottles.b` | 0.002s | 0.002s | 0.002s | 0.002s |
| `deadcodetest.b` | 0.002s | 0.001s | 0.002s | 0.002s |
| `hanoi.b` | 3.799s | 0.044s | 3.798s | 0.044s |
| `hello.b` | 0.002s | 0.002s | 0.002s | 0.002s |
| `long.b` | 3.168s | 0.202s | 3.170s | 0.203s |
| `loopremove.b` | 0.002s | 0.002s | 0.002s | 0.002s |
| `mandel.b` | 0.803s | 0.759s | 0.805s | 0.757s |
| `serptri.b` | 0.002s | 0.002s | 0.002s | 0.002s |
| `twinkle.b` | 0.002s | 0.002s | 0.002s | 0.002s |