SENG 440: *Computing Convergence Method*

By: Matthew Trent (V00982038) · Gabriel Maryshev (V00993574)

15 slides excluding 5 benchmark graphs

CCM for the Raspberry Pi 4 B 💻

- It's a "shift and add"-type algorithm for calculating transcendental functions.
- Uses *fixed-point*, opposed to *floating-point arithmetic* for efficiency and precision.
- Our implementation is tailored specifically for our *Raspberry Pi 4 B*, *64-bit ARM Cortex-A72 processor*, *ARMv8-A architecture*, *gcc-optimized*.

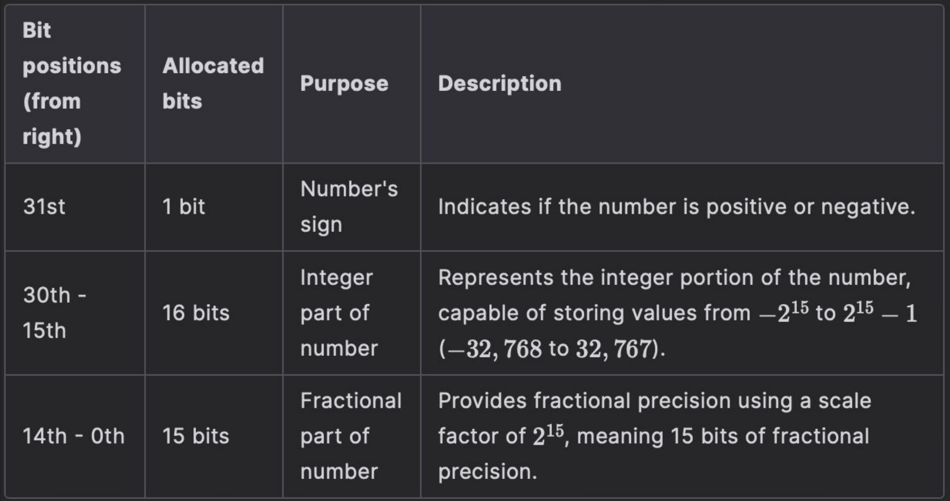
My Raspberry Pi 😁

Design requirements 📝

1. We need to select one of four transcendental functions to perform CCM on: *log2(M)*, *e^M*, *M^(1/2)*, or *M^(1/3)*.
2. We need to take in inputs wider than the ideal theoretical range.
3. CCM needs to be implemented using fixed-point arithmetic.
4. Determine the bottleneck(s) in the CCM algorithm.
5. Optimize the algorithm.
6. Determine the speed-up of the new implementation.

Design choices 👨‍🔬

- We chose the *base-2 logarithm*.
- Argument range:
    - Practical: *0 < M <= 2^16*.
    - Theoretical: *0 < M <= 0.5*.
- Barr Group and MISRA C's Embedded C Coding Standards.
- Chose scale factor of *2^15*.



Bit positions (from right)	**Allocated bits**	**Purpose**	**Description**
31st	1 bit	Number's sign	Indicates if the number is positive or negative.
30th - 15th	16 bits	Integer part of number	Represents the integer portion of the number, capable of storing values from -2^{15} to $2^{15} - 1$ ($-32,768$ to $32,767$).
14th - 0th	15 bits	Fractional part of number	Provides fractional precision using a scale factor of 2^{15}, meaning 15 bits of fractional precision.

32 of possible 64-bit delegation

Our run script for the Pi 🏃

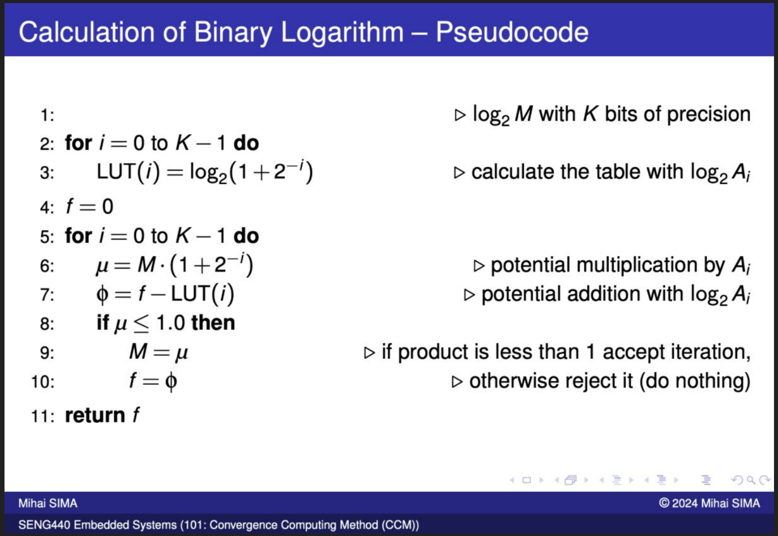
Purpose: Simple code transfer, compilation, and execution on Raspberry Pi.

- Used like: *./run.sh /path/to/file.c -some -flags -here*.
- Transfers files and compiles with GCC.
- Generates assembly and performance reports.
- Executes binaries, returns output: *~/asm* and *~/stats*.

Default flags:

- -mcpu=cortex-a72, -O3, -fno-stack-protector, -fomit-frame-pointer, -lm.

Core implementation 🛠️

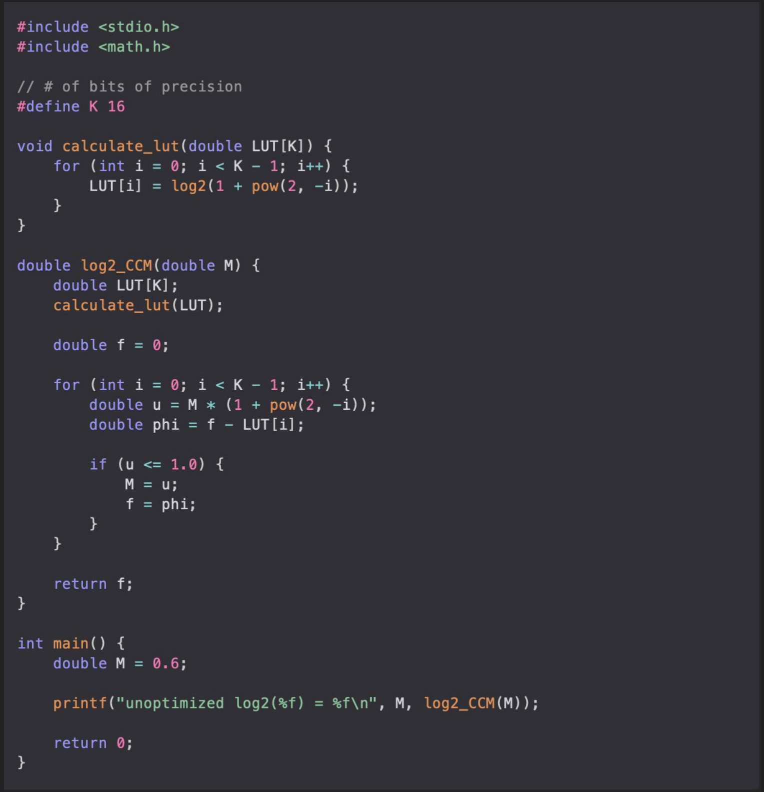


Calculation of Binary Logarithm – Pseudocode

```
1:                                              ▷ $\log_2 M$ with $K$ bits of precision
2: for $i = 0$ to $K - 1$ do
3:     $\mathrm{LUT}(i) = \log_2(1 + 2^{-i})$     ▷ calculate the table with $\log_2 A_i$
4: $f = 0$
5: for $i = 0$ to $K - 1$ do
6:     $\mu = M \cdot (1 + 2^{-i})$               ▷ potential multiplication by $A_i$
7:     $\phi = f - \mathrm{LUT}(i)$               ▷ potential addition with $\log_2 A_i$
8:     if $\mu \leq 1.0$ then
9:         $M = \mu$                              ▷ if product is less than 1 accept iteration,
10:        $f = \phi$                             ▷ otherwise reject it (do nothing)
11: return $f$
```

Mihai SIMA © 2024 Mihai SIMA

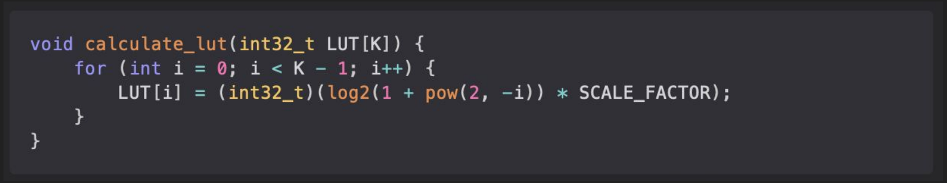
SENG440 Embedded Systems (101: Convergence Computing Method (CCM))



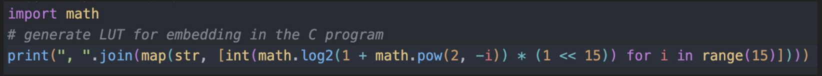
```c
#include <stdio.h>
#include <math.h>

// # of bits of precision
#define K 16

void calculate_lut(double LUT[K]) {
    for (int i = 0; i < K - 1; i++) {
        LUT[i] = log2(1 + pow(2, -i));
    }
}

double log2_CCM(double M) {
    double LUT[K];
    calculate_lut(LUT);

    double f = 0;

    for (int i = 0; i < K - 1; i++) {
        double u = M * (1 + pow(2, -i));
        double phi = f - LUT[i];

        if (u <= 1.0) {
            M = u;
            f = phi;
        }
    }

    return f;
}

int main() {
    double M = 0.6;

    printf("unoptimized log2(%f) = %f\n", M, log2_CCM(M));

    return 0;
}
```

Optimizations: part 1 ⚡

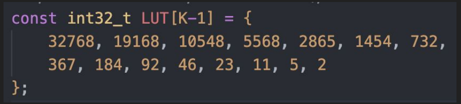
Dynamic to defined LUT:



```c
void calculate_lut(int32_t LUT[K]) {
    for (int i = 0; i < K - 1; i++) {
        LUT[i] = (int32_t)(log2(1 + pow(2, -i)) * SCALE_FACTOR);
    }
}
```

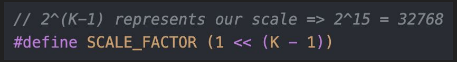


```python
import math
# generate LUT for embedding in the C program
print(", ".join(map(str, [int(math.log2(1 + math.pow(2, -i)) * (1 << 15)) for i in range(15)])))
```

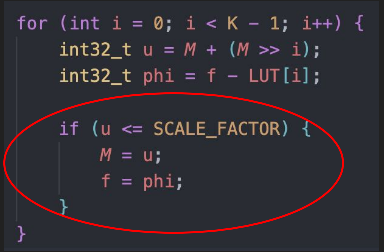


```c
const int32_t LUT[K-1] = {
    32768, 19168, 10548, 5568, 2865, 1454, 732,
    367, 184, 92, 46, 23, 11, 5, 2
};
```

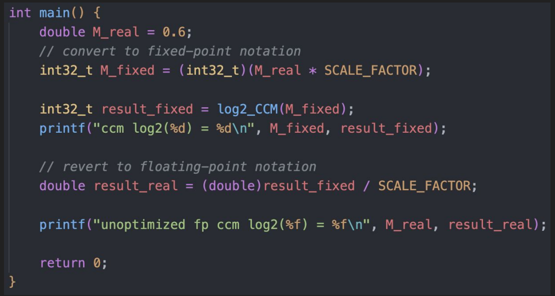
Fixed-point arithmetic:



```c
// 2^(K-1) represents our scale => 2^15 = 32768
#define SCALE_FACTOR (1 << (K - 1))
```



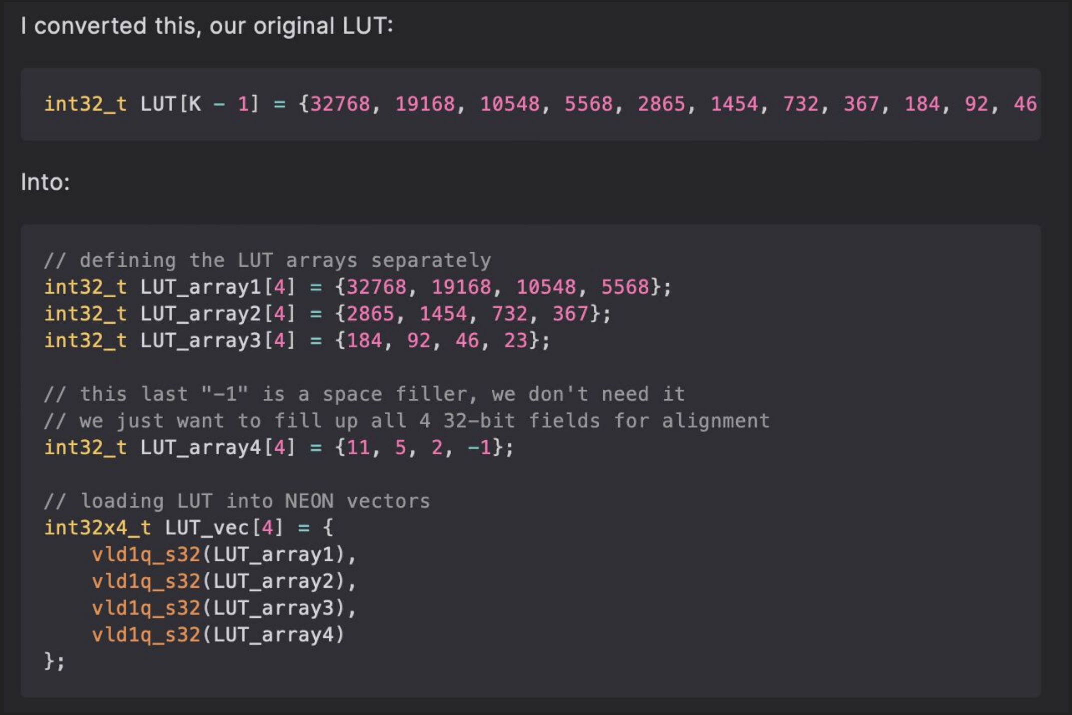
```c
for (int i = 0; i < K - 1; i++) {
    int32_t u = M + (M >> i);
    int32_t phi = f - LUT[i];

    if (u <= SCALE_FACTOR) {
        M = u;
        f = phi;
    }
}
```



```c
int main() {
    double M_real = 0.6;
    // convert to fixed-point notation
    int32_t M_fixed = (int32_t)(M_real * SCALE_FACTOR);

    int32_t result_fixed = log2_CCM(M_fixed);
    printf("ccm log2(%d) = %d\n", M_fixed, result_fixed);

    // revert to floating-point notation
    double result_real = (double)result_fixed / SCALE_FACTOR;

    printf("unoptimized fp ccm log2(%f) = %f\n", M_real, result_real);

    return 0;
}
```

Optimizations: part 2 ⚡

SIMD (NEON):



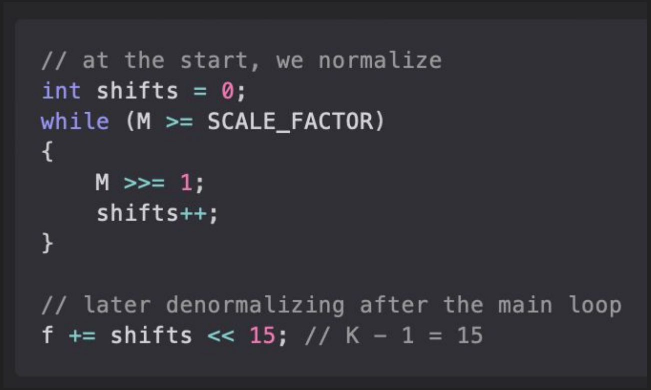
I converted this, our original LUT:

```c
int32_t LUT[K - 1] = {32768, 19168, 10548, 5568, 2865, 1454, 732, 367, 184, 92, 46
```

Into:

```c
// defining the LUT arrays separately
int32_t LUT_array1[4] = {32768, 19168, 10548, 5568};
int32_t LUT_array2[4] = {2865, 1454, 732, 367};
int32_t LUT_array3[4] = {184, 92, 46, 23};

// this last "-1" is a space filler, we don't need it
// we just want to fill up all 4 32-bit fields for alignment
int32_t LUT_array4[4] = {11, 5, 2, -1};

// loading LUT into NEON vectors
int32x4_t LUT_vec[4] = {
    vld1q_s32(LUT_array1),
    vld1q_s32(LUT_array2),
    vld1q_s32(LUT_array3),
    vld1q_s32(LUT_array4)
};
```

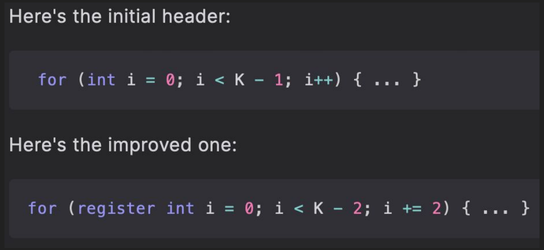
Input normalization:



```c
// at the start, we normalize
int shifts = 0;
while (M >= SCALE_FACTOR)
{
    M >>= 1;
    shifts++;
}

// later denormalizing after the main loop
f += shifts << 15; // K - 1 = 15
```

Optimizations: part 3 ⚡

Loop header:



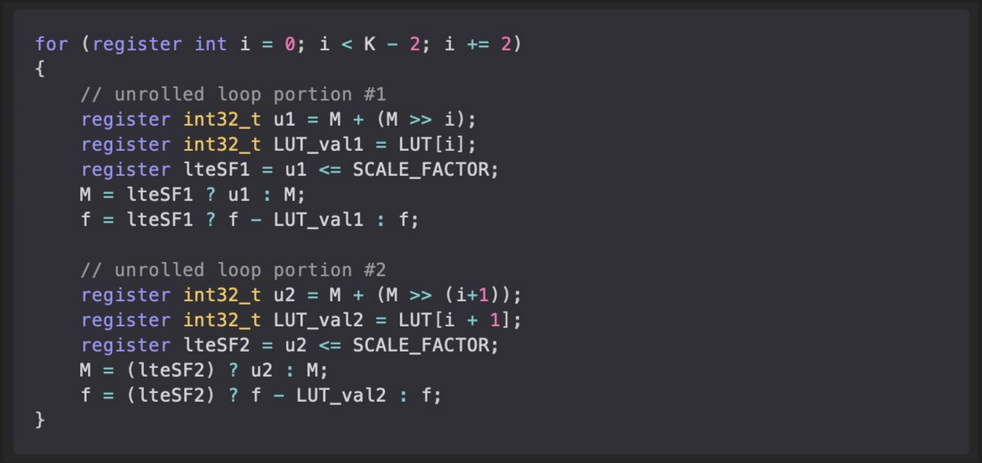
Here's the initial header:

```
for (int i = 0; i < K - 1; i++) { ... }
```

Here's the improved one:

```
for (register int i = 0; i < K - 2; i += 2) { ... }
```

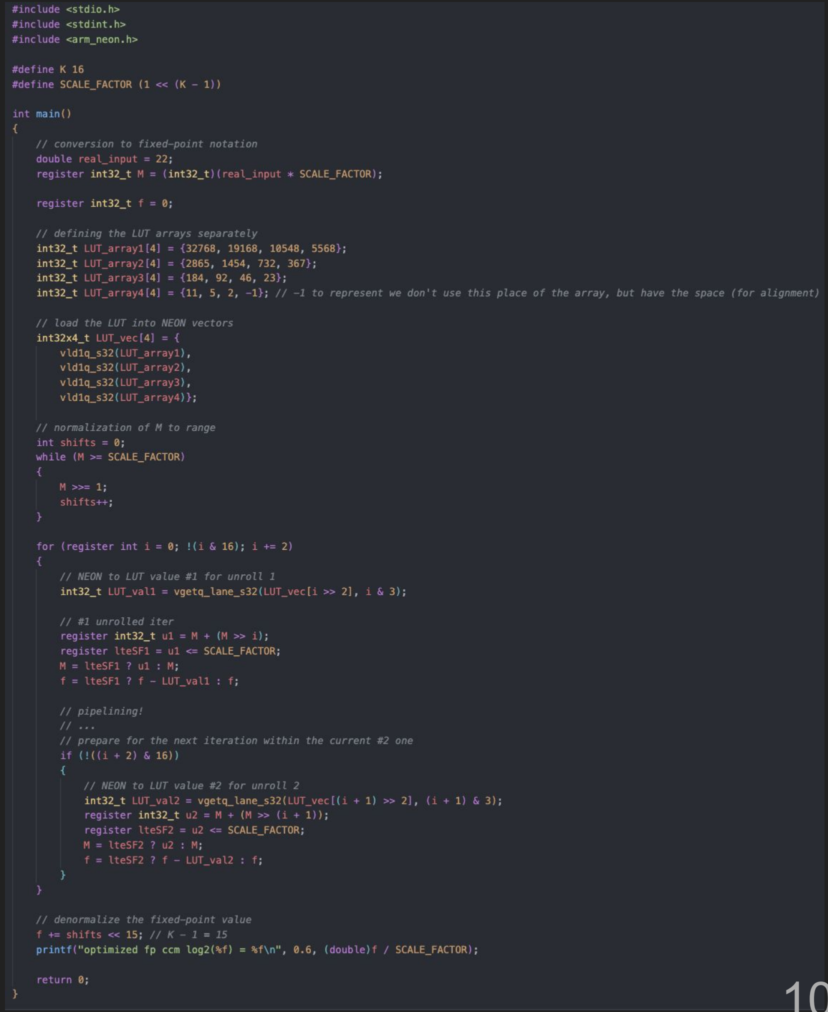
Register usage, ternary operators, bitwise ops, & loop unrolling:



```
for (register int i = 0; i < K - 2; i += 2)
{
    // unrolled loop portion #1
    register int32_t u1 = M + (M >> i);
    register int32_t LUT_val1 = LUT[i];
    register lteSF1 = u1 <= SCALE_FACTOR;
    M = lteSF1 ? u1 : M;
    f = lteSF1 ? f - LUT_val1 : f;

    // unrolled loop portion #2
    register int32_t u2 = M + (M >> (i+1));
    register int32_t LUT_val2 = LUT[i + 1];
    register lteSF2 = u2 <= SCALE_FACTOR;
    M = (lteSF2) ? u2 : M;
    f = (lteSF2) ? f - LUT_val2 : f;
}
```

Optimizations: part 4 ⚡

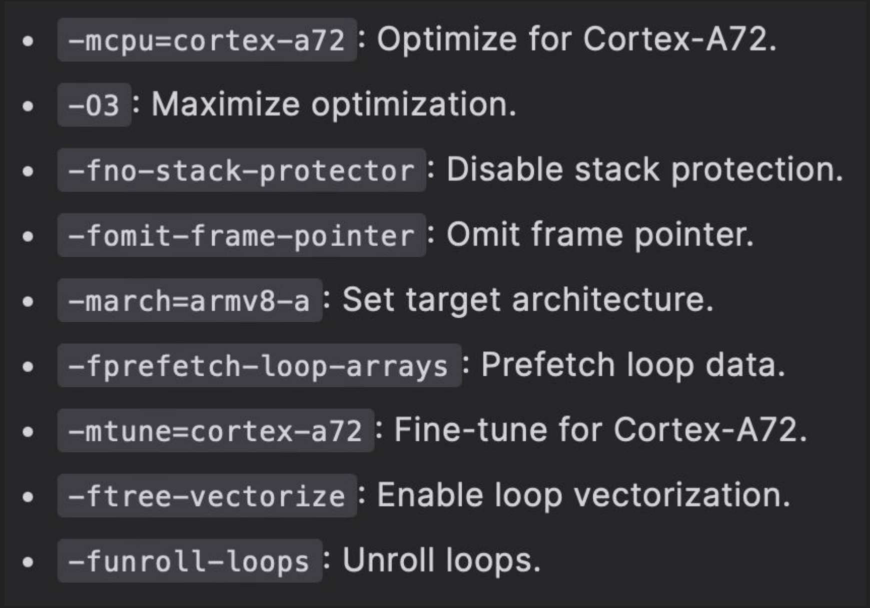
- Several new ones*, now our *final version*.
- Overall:
  - *Operator strength reduction.
  - *Reducing function call overheads.
  - *Register keywords.
  - *Locality of variable definitions.
  - *Bitwise operations and comparisons.
  - *Software pipelining.
  - Predicate operations.
  - Loop unrolling.
  - SIMD (NEON).
  - Fixed point arithmetic.
  - Simple asymptotic optimization (analysis).



```c
#include <stdio.h>
#include <stdint.h>
#include <arm_neon.h>

#define K 16
#define SCALE_FACTOR (1 << (K - 1))

int main()
{
    // conversion to fixed-point notation
    double real_input = 22;
    register int32_t M = (int32_t)(real_input * SCALE_FACTOR);

    register int32_t f = 0;

    // defining the LUT arrays separately
    int32_t LUT_array1[4] = {32768, 19168, 10548, 5568};
    int32_t LUT_array2[4] = {2865, 1454, 732, 367};
    int32_t LUT_array3[4] = {184, 92, 46, 23};
    int32_t LUT_array4[4] = {11, 5, 2, -1}; // -1 to represent we don't use this place of the array, but have the space (for alignment)

    // load the LUT into NEON vectors
    int32x4_t LUT_vec[4] = {
        vld1q_s32(LUT_array1),
        vld1q_s32(LUT_array2),
        vld1q_s32(LUT_array3),
        vld1q_s32(LUT_array4)};

    // normalization of M to range
    int shifts = 0;
    while (M >= SCALE_FACTOR)
    {
        M >>= 1;
        shifts++;
    }

    for (register int i = 0; !(i & 16); i += 2)
    {
        // NEON to LUT value #1 for unroll 1
        int32_t LUT_val1 = vgetq_lane_s32(LUT_vec[i >> 2], i & 3);

        // #1 unrolled iter
        register int32_t u1 = M + (M >> i);
        register lteSF1 = u1 <= SCALE_FACTOR;
        M = lteSF1 ? u1 : M;
        f = lteSF1 ? f - LUT_val1 : f;

        // pipelining!
        // ...
        // prepare for the next iteration within the current #2 one
        if (!((i + 2) & 16))
        {
            // NEON to LUT value #2 for unroll 2
            int32_t LUT_val2 = vgetq_lane_s32(LUT_vec[(i + 1) >> 2], (i + 1) & 3);
            register int32_t u2 = M + (M >> (i + 1));
            register lteSF2 = u2 <= SCALE_FACTOR;
            M = lteSF2 ? u2 : M;
            f = lteSF2 ? f - LUT_val2 : f;
        }
    }

    // denormalize the fixed-point value
    f += shifts << 15; // K - 1 = 15
    printf("optimized fp ccm log2(%f) = %f\n", 0.6, (double)f / SCALE_FACTOR);

    return 0;
}
```

Flags ⛳

- None for versions 1-4.
- Optimized as much as we could for our *final version*, 5:
- These overrode the ones we defined earlier as defaults in our run script.



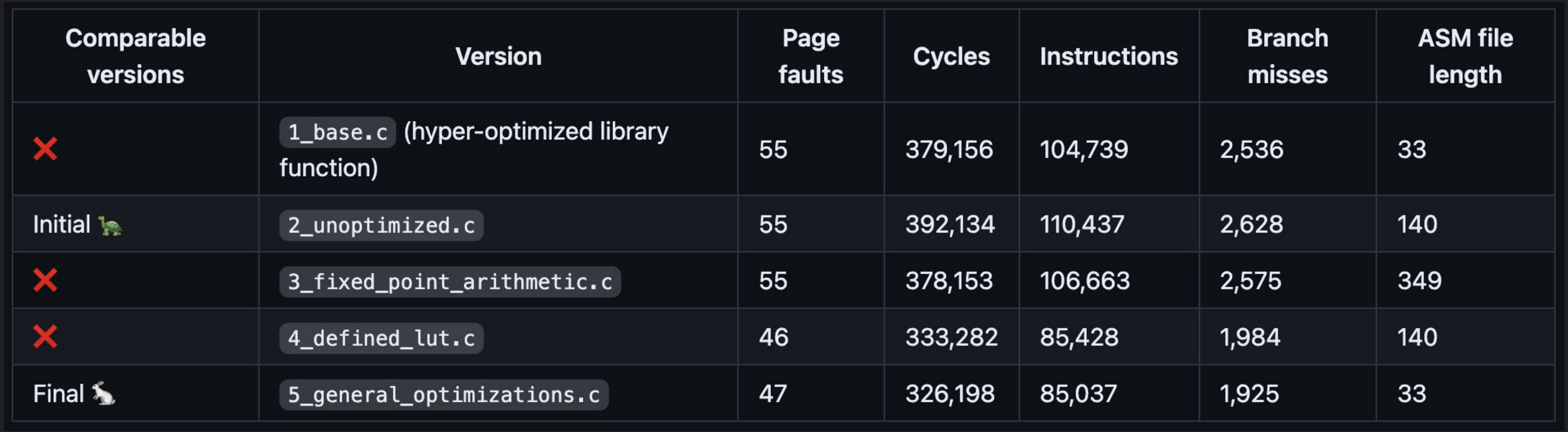
- `-mcpu=cortex-a72` : Optimize for Cortex-A72.
- `-03` : Maximize optimization.
- `-fno-stack-protector` : Disable stack protection.
- `-fomit-frame-pointer` : Omit frame pointer.
- `-march=armv8-a` : Set target architecture.
- `-fprefetch-loop-arrays` : Prefetch loop data.
- `-mtune=cortex-a72` : Fine-tune for Cortex-A72.
- `-ftree-vectorize` : Enable loop vectorization.
- `-funroll-loops` : Unroll loops.

Assembly output into ~/asm 👾

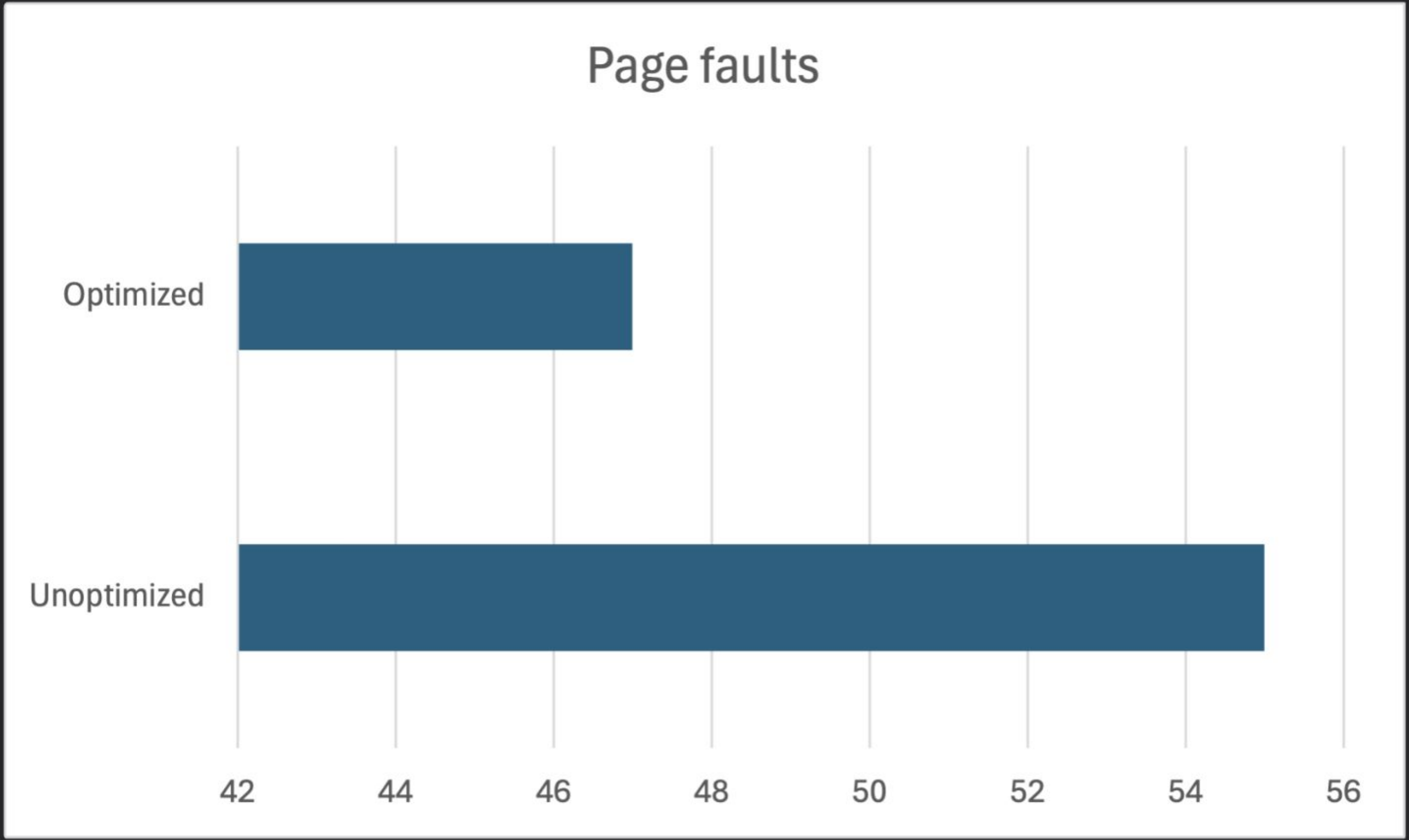
- Optimized version:
    - Shows fewer branch misses.
    - Less load/store cycles.
    - Smaller Instruction count.
    - Efficiently handles conditional operations without branching.
        - Closest equivalent in ARMv8-A is: *csel*, *csinc*, *csinv*, *cset*, etc.
    - Our code seemed *optimized-enough without diving much into assembly details*.

Benchmarking 💨

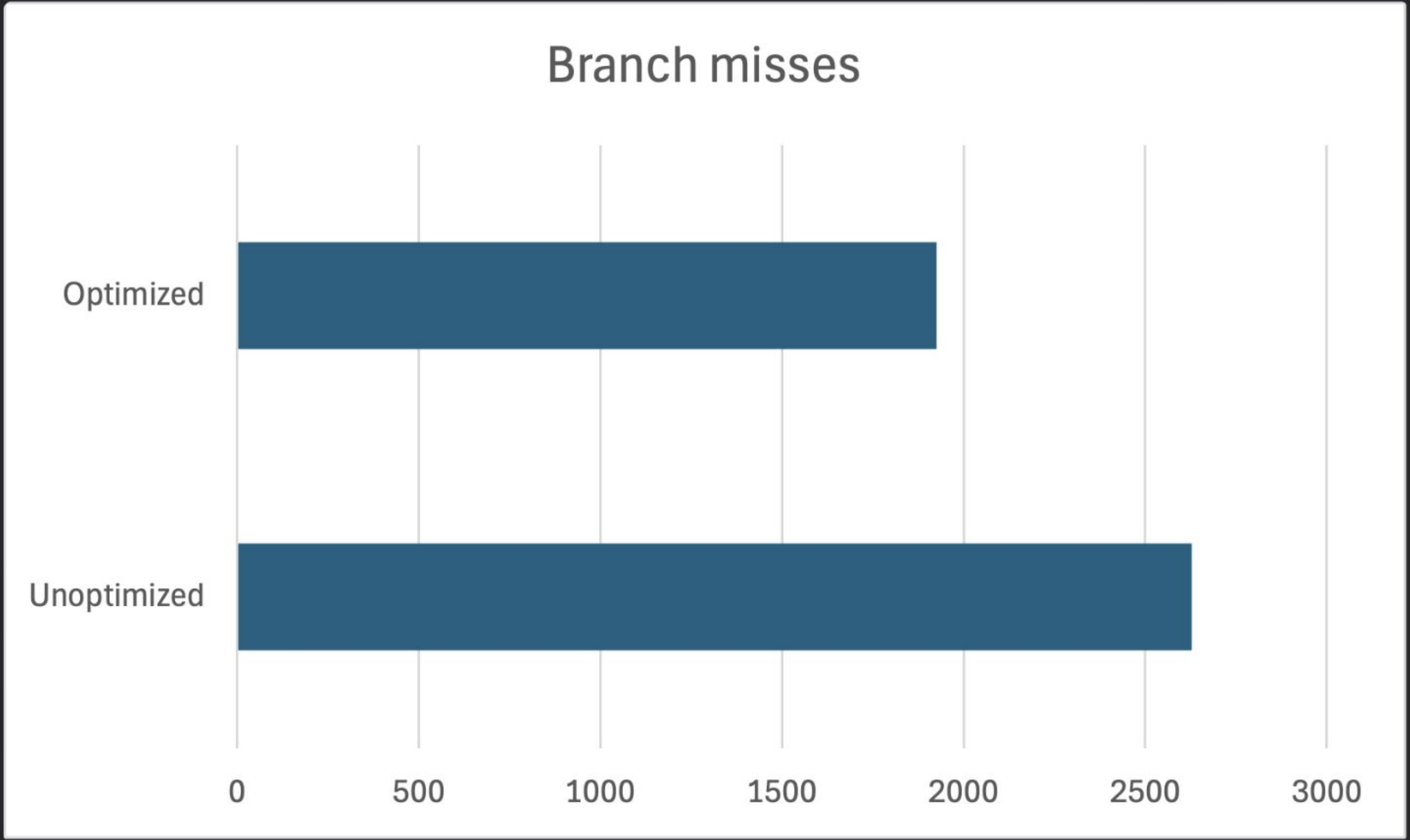
Used Linux's Perf to benchmark all 5 versions:



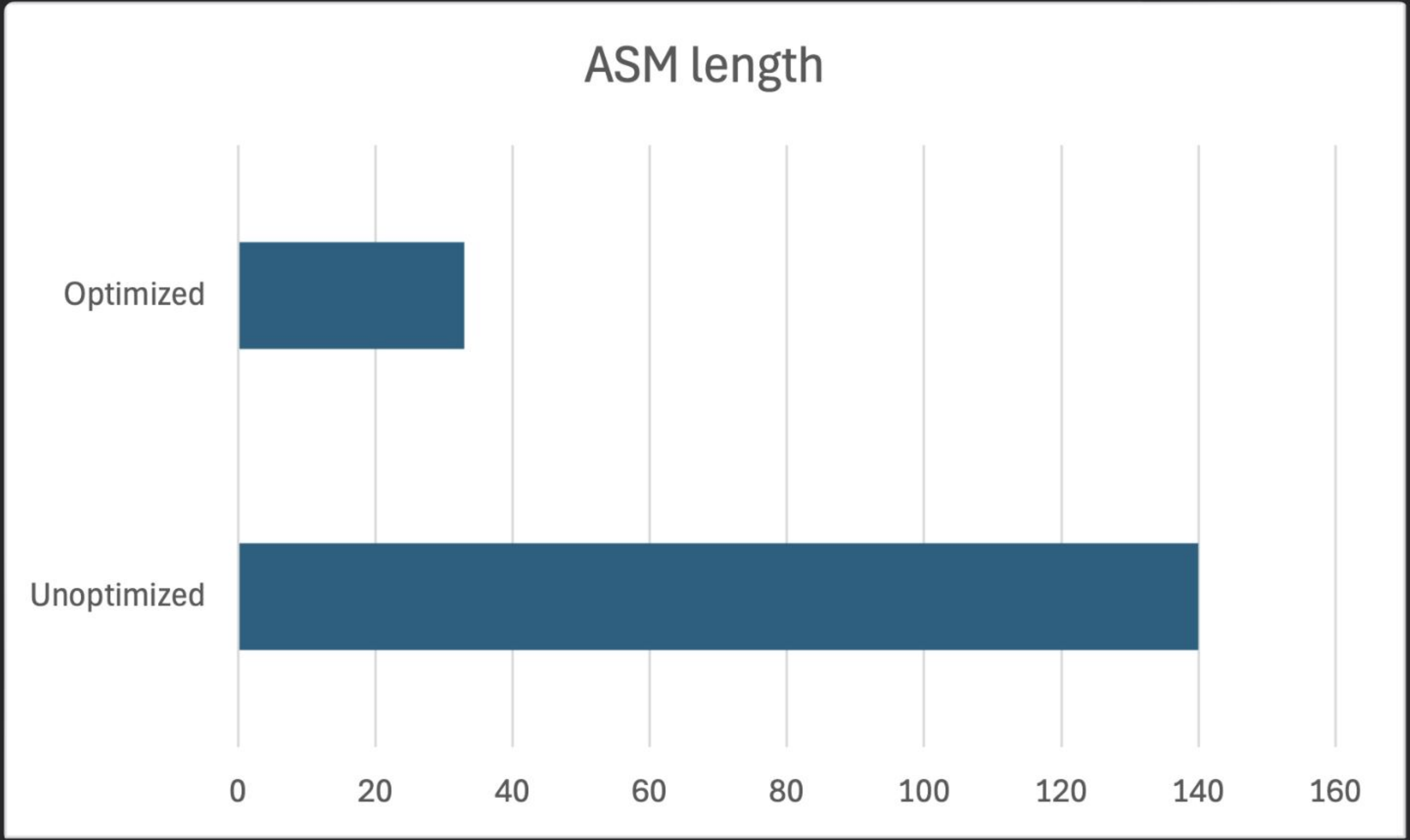
Comparable versions	Version	Page faults	Cycles	Instructions	Branch misses	ASM file length
❌	`1_base.c` (hyper-optimized library function)	55	379,156	104,739	2,536	33
Initial 🐢	`2_unoptimized.c`	55	392,134	110,437	2,628	140
❌	`3_fixed_point_arithmetic.c`	55	378,153	106,663	2,575	349
❌	`4_defined_lut.c`	46	333,282	85,428	1,984	140
Final 🐇	`5_general_optimizations.c`	47	326,198	85,037	1,925	33



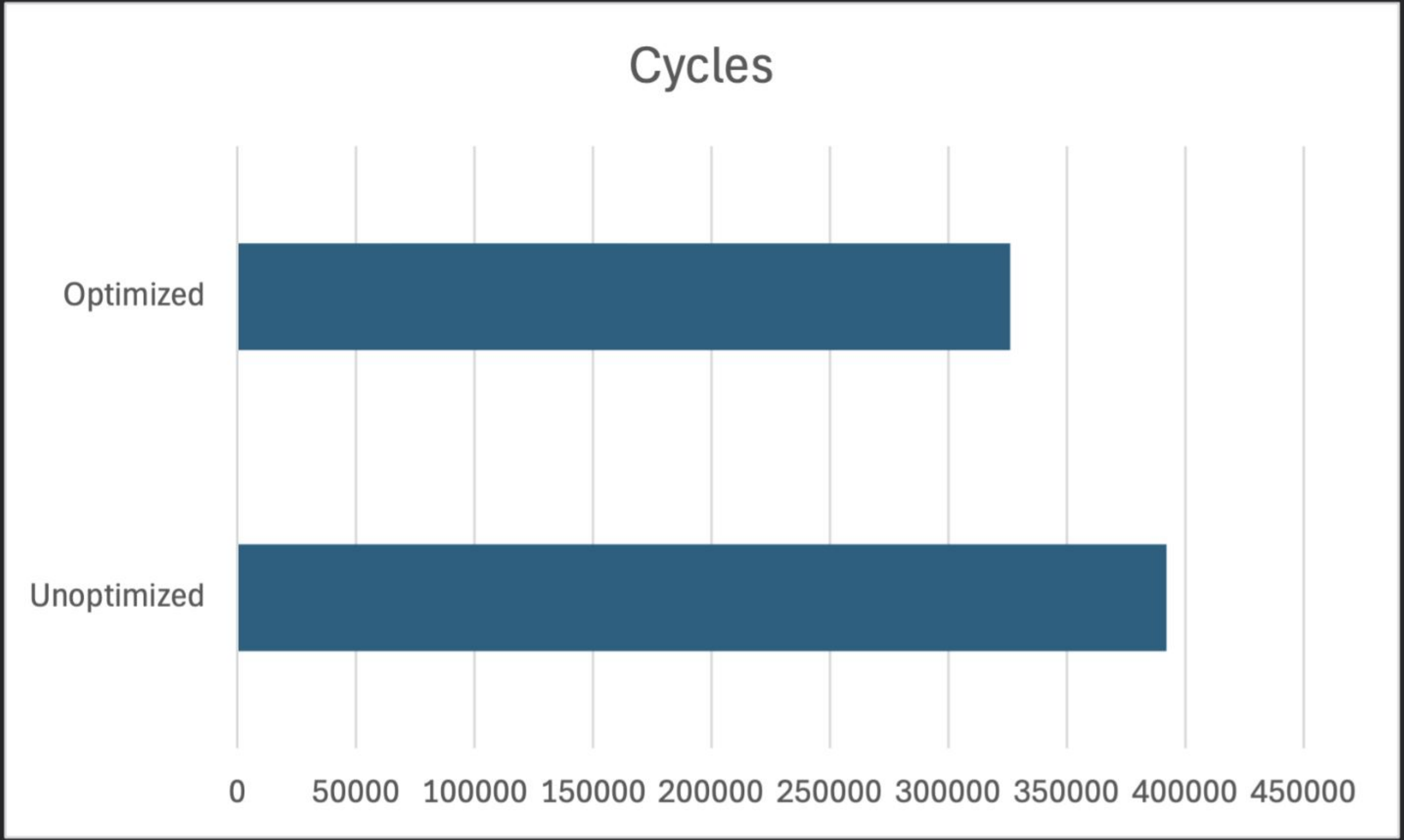
16% less page faults.



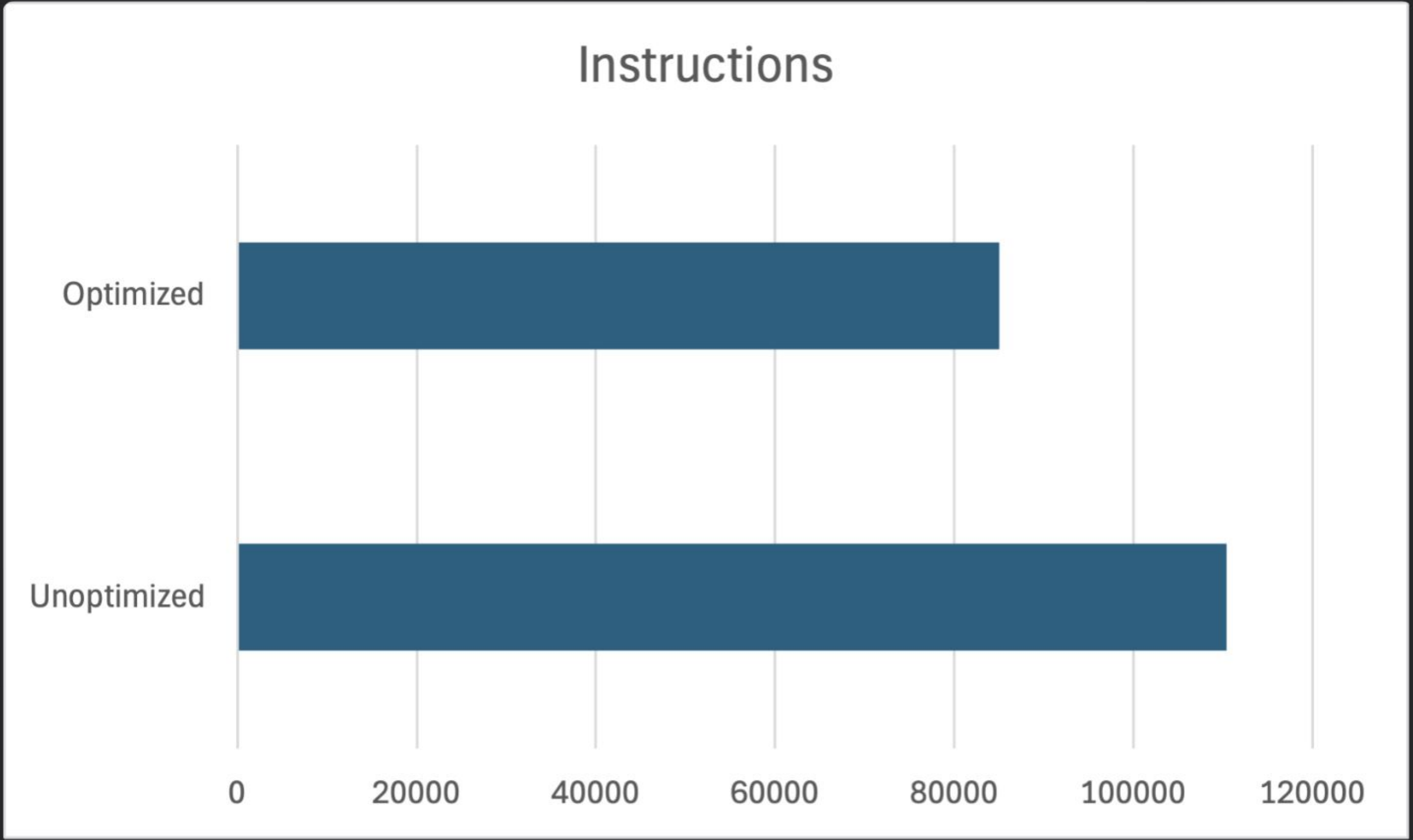
31% fewer branch misses.



124% fewer lines of ASM.



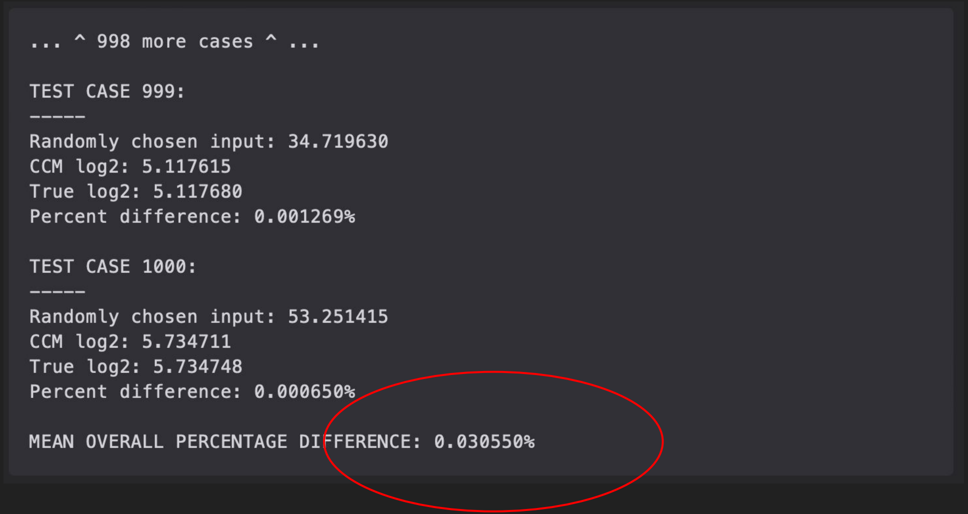
18% fewer cycles.



26% fewer instructions.

Proof of correctness ✅

- Ran *1,000* times for certainty.
- Random *0 < M <= 100* values.
- Could test up to *2^16*.
- *0.030550%* mean deviation from "*true log_2*".
- Expected difference given memory and speed trade-off.



```
... ^ 998 more cases ^ ...

TEST CASE 999:
-----
Randomly chosen input: 34.719630
CCM log2: 5.117615
True log2: 5.117680
Percent difference: 0.001269%

TEST CASE 1000:
-----
Randomly chosen input: 53.251415
CCM log2: 5.734711
True log2: 5.734748
Percent difference: 0.000650%

MEAN OVERALL PERCENTAGE DIFFERENCE: 0.030550%
```

Conclusions and profiling 🔮

- Would have used Valgrind, but:
  - Non-long running.
  - Weren’t concerned about memory leaks.
- Would have used Cachegrind, but:
  - That focused on cache optimization while (most of) ours worked on algorithmic speed-up.
- Achieved *43% improvement* on-average across *5 key algorithm metrics*.