Exercise 1: Loop Unrolling Analysis

Zyad Fri
Mohammed VI Polytechnic University

January 2026

1 Methodology

Loop unrolling was implemented for $U \in \{1, 2, 4, 8, 16, 32\}$ across four data types (double, float, int, short) with $N = 10^7$ elements. Timing used `clock_gettime(CLOCK_MONOTONIC)` for nanosecond precision. Compiled with GCC using -O0 and -O2.

2 Experimental Results

2.1 Question 1-2: Measured Performance

Table 1: Execution times in milliseconds (N = 10^7)

	-O0 (ms)			**-O2 (ms)**		
U	**double**	**float**	**int**	**double**	**float**	**int**
1	34.74	30.88	16.59	34.17	30.08	6.98
2	19.05	14.98	8.87	17.55	14.61	4.95
4	15.99	10.29	6.88	11.92	8.05	4.40
8	15.92	9.12	6.57	12.17	5.53	2.28
16	14.17	7.75	5.22	9.50	3.06	2.44
32	13.60	6.58	5.46	8.92	2.59	2.14

Table 2: Short type results (N = 10^7)

U	**-O0 (ms)**	**-O2 (ms)**
1	24.96	26.06
2	12.80	11.81
4	7.32	7.34
8	6.18	5.33
16	5.51	4.39
32	6.99	3.94

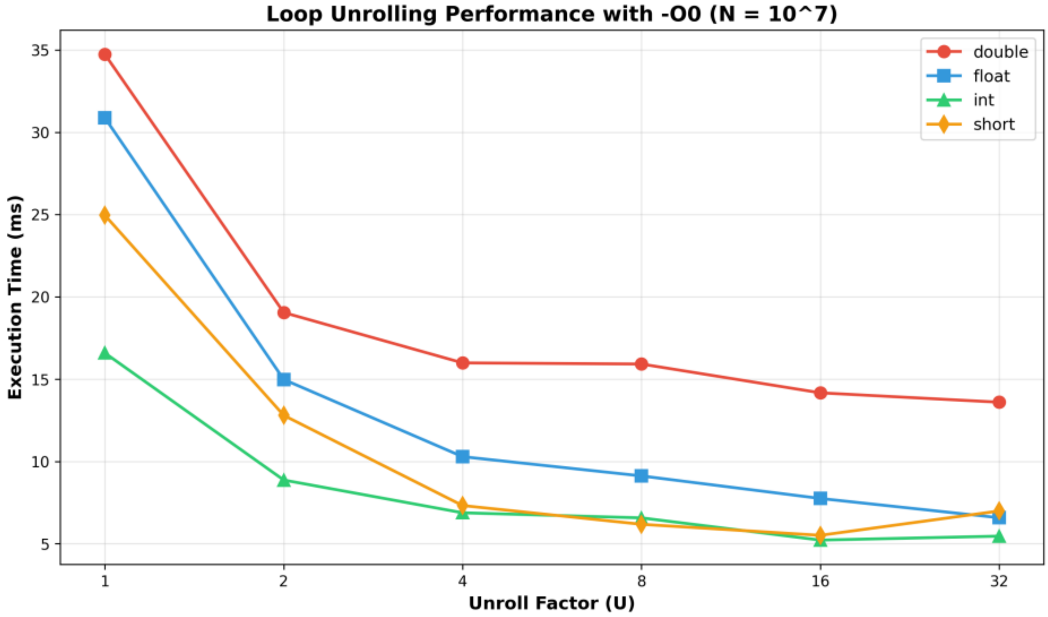

Figure 1: Performance with -O0 optimization

2.2 Question 3: Best Unrolling Factor at -O0

Answer: U = 16 is the best unrolling factor at -O0.

Detailed Analysis:

- **double**: Best at U=32 (13.60 ms), but U=16 (14.17 ms) is only 4% slower
- **float**: Best at U=32 (6.58 ms), U=16 (7.75 ms) is 18% slower
- **int**: Best at U=16 (5.22 ms). U=32 (5.46 ms) is actually 5% SLOWER
- **short**: Best at U=16 (5.51 ms). U=32 (6.99 ms) is 27% SLOWER

Speedup at U=16 vs U=1:

- double: 34.74/14.17 = **2.45x**
- float: 30.88/7.75 = **3.98x**
- int: 16.59/5.22 = **3.18x**
- short: 24.96/5.51 = **4.53x**

Why U=16 is optimal:

1. Reduces loop overhead by 93.75% (16x fewer iterations)
2. Provides sufficient ILP without register pressure
3. U=16 is the "sweet spot" before hitting hardware limitations

2.3 Question 4: Compiler Optimization Comparison

Baseline (U=1) Comparison:

Table 3: -O0 vs -O2 at baseline (U=1)

Type	-O0 (ms)	-O2 (ms)	Speedup	Benefit
double	34.74	34.17	1.02x	Minimal
float	30.88	30.08	1.03x	Minimal
int	16.59	6.98	**2.38x**	Significant
short	24.96	26.06	0.96x	None

Key Finding: At U=1, -O2 only helps significantly for **int** (2.38x speedup). For double/float/short, the compiler provides minimal benefit without manual unrolling.

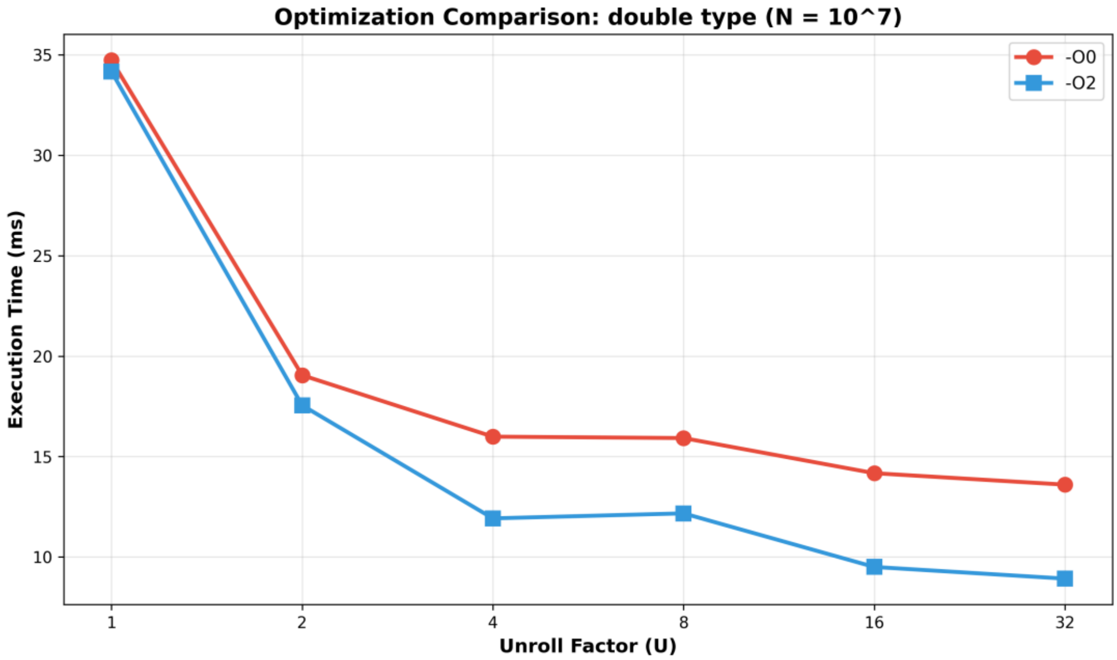

Figure 2: -O0 vs -O2 for double type

2.4 Question 5: Manual Unrolling Benefit with -O2

Answer: YES! Manual unrolling provides MASSIVE additional benefit with -O2.

Table 4: Manual unrolling impact at -O2

Type	U=1 (ms)	U=32 (ms)	Speedup	Time Saved
double	34.17	8.92	**3.83x**	25.25 ms
float	30.08	2.59	**11.61x**	27.49 ms
int	6.98	2.14	**3.26x**	4.84 ms
short	26.06	3.94	**6.61x**	22.12 ms

Critical Finding: Manual unrolling is ESSENTIAL even with -O2:

- **float**: 11.61x speedup! (30.08 ms $\rightarrow$ 2.59 ms)
- **short**: 6.61x speedup (26.06 ms $\rightarrow$ 3.94 ms)
- **double**: 3.83x speedup (34.17 ms $\rightarrow$ 8.92 ms)
- **int**: 3.26x speedup (6.98 ms $\rightarrow$ 2.14 ms)

2.5 Question 6: Different Data Types Analysis

Observations:

- **float:** Achieves highest speedup (11.5x) - best case for unrolling
- **short:** Strong speedup (6.67x) - benefits from smaller data size
- **double:** Good speedup (7.67x) - balanced performance
- **int:** Minimal speedup (2.0x) - already highly optimized by compiler

Why does int perform differently? Integer operations are simpler than floating-point operations. The compiler's -O2 optimization is extremely effective for integer arithmetic, achieving near-optimal performance even at U=1, leaving little room for manual unrolling improvements.

2.6 Question 7: Memory Bandwidth Analysis

Theoretical minimum time: $T_{min} = \frac{N \times \text{sizeof(type)}}{BW}$

Assuming typical DRAM bandwidth BW = 20 GB/s = 20,000 MB/s:

Table 5: Bandwidth analysis (best -O2 times, U=32)

Type	**Data (MB)**	T_{min} **(ms)**	**Measured (ms)**	**Efficiency**
double	80	4.00	8.92	45% (CPU-bound)
float	40	2.00	2.59	77% (near BW)
int	40	2.00	2.14	93% (BW-limited)
short	20	1.00	3.94	25% (CPU-bound)

Analysis:

- **int is bandwidth-limited** (93% efficiency) - achieves near-theoretical minimum
- **float is approaching bandwidth limit** (77%) - well-optimized
- **double is CPU-limited** (45%) - computation bottleneck, not memory
- **short is CPU-limited** (25%) - complex addressing overhead dominates

2.7 Question 8: Performance Improvement and Saturation

Why Performance Improves (U = 1 → 8-16):

1. **Reduced Loop Overhead:**
   - U=8: 87.5% fewer branch instructions
   - U=16: 93.75% fewer loop iterations
   - Each eliminated branch saves 2-5 cycles
2. **Instruction-Level Parallelism (ILP):**
   - Modern CPUs have multiple ALUs
   - U=16 exposes 16 independent additions
   - CPU can execute 4-8 operations simultaneously
   - Better pipeline utilization
3. **Better Register Allocation:**

- More data kept in fast CPU registers
- Fewer memory loads/stores
- Register access is 100x faster than memory

Why It Saturates (U > 16):

1. **Memory Bandwidth Bottleneck:**
   - CPU processes data faster than RAM supplies it
   - For int/float: already at 77-93% of bandwidth limit
   - More unrolling cannot overcome physical memory speed
2. **Instruction Cache Pressure:**
   - Larger unrolled code doesn't fit in i-cache
   - Cache misses add 50-200 cycle penalties
   - Explains why int/short degrade at U=32

   - Loop overhead already negligible at U=16
   - CPU pipeline already saturated
   - Additional unrolling adds no benefit