Bhav Wadhwa| Student Number: 301609355| Project Report

Performance Evaluation of C, Assembly, and SIMD Implementations of Insertion Sort on Modern CPUs

Introduction

Modern CPUs consist of various sophisticated microarchitectural features. From out of order execution, multi level caches, branch prediction to all the way to SIMD vector units. These features significantly influence the efficiency of the code runs. Even in the cases where multiple implementations of the same algorithm, differences in coding style, compiler optimization, branching structure, and memory access patterns can lead to dramatically different performance characteristics.

This project is an exploration of how these practices can affect the runtime in practice by evaluating several implementations of the same algorithm. Specifically, this project analyzes:

- Compiler Optimization Levels
- Branch predictability
- Cache Behaviour
- SIMD vs Scalar Performance
- Hand Written Assembly vs Compiler Generated Assembly

By implementing the same algorithm in multiple ways, I was able to observe exactly how CPU features influence performance.

Algorithm Chosen

I selected insertion sort as the algorithm for this study. Insertion sort is a simple sorting algorithm that allows us to build the final sorted list one element at a time by inserting each item into its correct position within the already-sorted portion of the list.

CPU & Hardware (SFU asb9838-A05)

- Operating System: Ubuntu Linux (64-bit environment)
- Model: Intel Core i9-11900 (11th Gen “Rocket Lake”, 2021)
- Architecture: x86-64
- Cores/Threads: 8 physical cores, 16 threads (Hyper-Threading)

Tools and Libraries: Perf, Val grind, clock_gettime, lxcpu, version2-master

Cache Hierarchy

- L1 Data Cache: 48 KiB per core → 384 KiB total
- L1 Instruction Cache: 32 KiB per core → 256 KiB total
- L2 Cache: 512 KiB per core → 4 MiB total
- L3 Cache: 16 MiB shared across all 8 cores

Code implementations

This project evaluates 5 implementations of the same algorithm that are written at different abstraction levels. All versions take an array on 64-bit integers of input size 20000. Following are the implementations:

1) Personal C Implementation: The first version is a straightforward, “textbook style” insertion sort that is written in C. It performs extra memory operations and non-optimized branching.
2) Optimized C Implementations: A compiler friendly version that stores the key once and shift elements, minimizing branches and memory writes.
3) Scalar Assembly Implementation: A hand-written x86-64 assembly version using scalar branches and explicit memory movement.
4) SIMD C implementation (Vector Class Library): A high-level SIMD version using AVX2 vectors to accelerate block shifts while keeping comparisons scalar.

5) SIMD Assembly Implementation (AVX2): A manually optimized assembly implementation that uses a 256-bit block shifts for faster movement.

Methodology

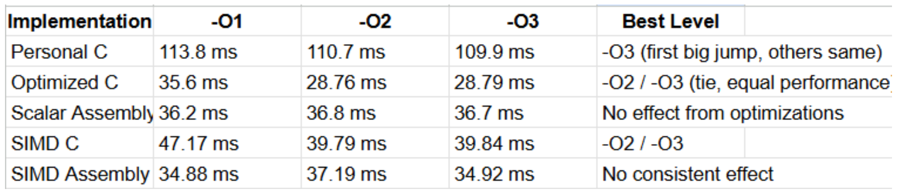
All experiments were performed on a Linux machine, using consistent compilation settings across all implementations. This section describes both **how** the different insertion-sort implementations were evaluated and what the results revealed about compiler optimizations, branch prediction, cache behavior, and SIMD performance. All the code was run with -std=c++17 -march=haswell flag and fixed input size of 20000, Val grind was run to ensure no memory leaks.

Experiment Type	Purpose	Optimization Level	Array Type	Which Implementation Used?
Optimization level test	Compare timings of all 5 implementations	O1, O2, O3	Random	All Implementations
SIMD Effectiveness test	Test viability of SIMD	O3	Random	SIMD C, SIMD Assembly, Scalar C
Branch Prediction	Understand impact of branches	O3	Random, Sorted, Reverse Sorted	Optimized C and Scalar Assembly
Cache Behaviour	Determine memory bottlenecks	O3	Random, Sorted, Reverse sorted	Optimized C and Scalar Assembly

Results

1) Runtime Comparison on Random Array (Using clock_gettime)

When all six implementations were tested on the same randomly generated array (20,000 elements), the following observations were observed.



Implementation	-O1	-O2	-O3	Best Level
Personal C	113.8 ms	110.7 ms	109.9 ms	-O3 (first big jump, others same)
Optimized C	35.6 ms	28.76 ms	28.79 ms	-O2 / -O3 (tie, equal performance
Scalar Assembly	36.2 ms	36.8 ms	36.7 ms	No effect from optimizations
SIMD C	47.17 ms	39.79 ms	39.84 ms	-O2 / -O3
SIMD Assembly	34.88 ms	37.19 ms	34.92 ms	No consistent effect

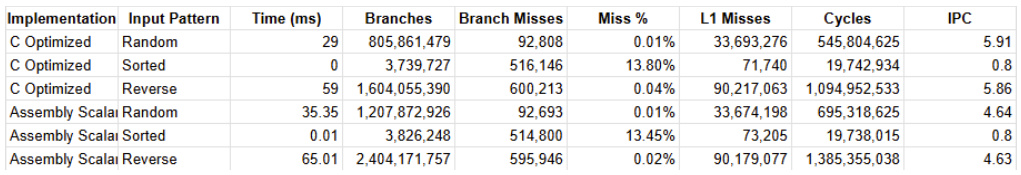
Key Observations:

- Personal C improves only at -O1, with -O2/-O3 nearly identical.
- Optimized C gains major speedup at -O2, with -O3 unchanged.
- SIMD C improves at -O2, but remains slower overall and Scalar and SIMD assembly show no change, since compiler optimizations cannot modify hand-written assembly.

Analysis: When moving to -O1 and -O2, the compiler:

- allocated key variables to registers and removed redundant loads and stores.
- optimized the shifting loop and rearranged instructions to hide memory latency
- This allows optimized C to reach 29ms, outperforming even hand-written assembly. Because insertion sort is not SIMD-friendly, the extra transformations at -O3 give no meaningful benefit.

2) Branch Prediction and Cache Behaviour Results (Using Perf and lxcpu)



Implementation	Input Pattern	Time (ms)	Branches	Branch Misses	Miss %	L1 Misses	Cycles	IPC
C Optimized	Random	29	805,861,479	92,808	0.01%	33,693,276	545,804,625	5.91
C Optimized	Sorted	0	3,739,727	516,146	13.80%	71,740	19,742,934	0.8
C Optimized	Reverse	59	1,604,055,390	600,213	0.04%	90,217,063	1,094,952,533	5.86
Assembly Scalar	Random	35.35	1,207,872,926	92,693	0.01%	33,674,198	695,318,625	4.64
Assembly Scalar	Sorted	0.01	3,826,248	514,800	13.45%	73,205	19,738,015	0.8
Assembly Scalar	Reverse	65.01	2,404,171,757	595,946	0.02%	90,179,077	1,385,355,038	4.63

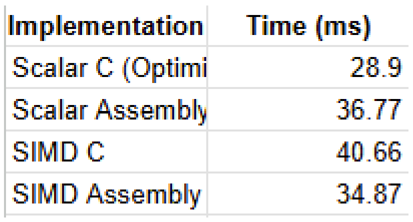
Key Observations:

- **O**ptimized C consistently outperforms hand-written assembly on random and reverse arrays, because the compiler achieves higher IPC and smarter scheduling, even though assembly executes fewer instructions.
- Branch mispredictions are negligible for random and reverse inputs (0.01–0.04%), showing the CPU predicts the data-dependent branch extremely accurately. Sorted inputs show higher miss % only because the execution is extremely short and warmup dominates.
- L1 cache misses dominate runtime, ranging from 33M (random) to 90M (reverse). These memory stalls, not branch behavior explains us the major timing differences.
- Runtime directly reflects algorithmic work: sorted arrays have almost no inner-loop shifting, random arrays require moderate work, and reverse arrays require the maximum number of comparisons and shifts.

Analysis:

- Cache behavior is the primary bottleneck: tens of millions of L1 misses dictate performance, while branch mispredictions contribute almost nothing.
- Optimized C achieves higher instruction-level parallelism than hand-written assembly, giving it significantly higher IPC and better ability to hide memory latency.
- Total runtime scales with the amount of work the algorithm performs—sorted inputs nearly skip the inner loop, random inputs do moderate work, and reverse inputs perform the maximum number of comparisons and shifts.

3) SIMD Results (Using clock_gettime)



Implementation	Time (ms)
Scalar C (Optimi	28.9
Scalar Assembly	36.77
SIMD C	40.66
SIMD Assembly	34.87

Key Observations:

- Despite SIMD's theoretical 4× data-parallel capability, the optimized scalar C implementation is the fastest (28.9ms), outperforming SIMD C (40.6ms) and SIMD assembly (34.8ms).
- Due to overhead (vector loads/stores, boundary checks, fallback scalar paths) that outweighs the advantages of processing four elements at once, both SIMD implementations operate 20–40% slower.
- The serial, data-dependent structure of insertion sort restricts vectorization; SIMD can only speed up the shifting stage, which is not the main expense.

Analysis:

- Insertion sort rarely shifts large enough contiguous blocks to amortize SIMD's setup cost; most shifts are only 1-3 elements, which is why SIMD performs poorly.
- When it comes to utilizing SIMD, it can only help speed up a very small part of working on an algorithm because finding the right place to insert data is the most intensive part of an algorithm and this is inherently a sequential process. Therefore, it will never be able to take advantage of SIMD capabilities.
- Also, there is an overhead with SIMD that scalar code does not have such as branching, switching modes, YMM register management, etc. which will make SIMD slower overall compared to scalar code.
- Because performance is dominated by memory latency rather than computation, SIMD's wider ALUs provide no benefit allowing optimized scalar C to remain superior.

Conclusion

Overall, this project shows that algorithmic behaviour and memory access patterns dominate performance on modern CPUs. Although SIMD is often assumed to be faster, insertion sort's serial, data-dependent structure prevents meaningful vectorization, making compiler-optimized scalar C the fastest implementation. Hand-written assembly also underperforms because modern compilers generate superior instruction scheduling and register allocation. Ultimately, selecting an algorithm that aligns with hardware characteristics matters far more than low-level manual tuning.