0. Setup Methadology

The project was set up in a controlled Linux environment configured for performance benchmarking. A lightweight Ubuntu installation was used with background services minimized and CPU frequency fixed to performance mode to reduce variability.

Kernels (SAXPY, stencil, elementwise multiply, stride, and gather) were implemented in C++ and compiled with g++ -O3 -march=native -fopenmp-simd, with build logs captured using -fopt-info-vec-optimized to confirm vectorization width and FMA usage.

Input arrays were allocated with explicit alignment, randomized initialization, and controlled tail handling (either tail-multiple or masked) to isolate effects of alignment and vector length. Experiments were run with CPU frequency pinned and processes affined to a single core to minimize scheduling noise, with each run repeated multiple times for statistical reliability. Performance was measured in GFLOP/s and cycles per element, with cache and TLB effects monitored via perf and bandwidth cross-checked using Intel MLC. Results were logged in structured CSV format, aggregated in Python to remove outliers, and plotted to compare SIMD vs. scalar performance, alignment and tail-handling overhead, data type scaling, and stride/gather effects. Finally, roofline analysis was conducted using the measured memory bandwidth (200 GiB/s) and peak FLOP estimates to distinguish compute-bound from memory-bound behavior and validate SIMD speedup predictions.

Using lscp, we get the following memory sizes:

L1 = 480 KiB
L2 = 1.25 MiB
L3 = 24MiB

We are using DRAM = 128+ MiB (while there is more DRAM space, the exact value isn't important as long as it is much bigger than LLC since we do not work with any memory spaces greater than DRAM in this project).

All code and scripts have been submitted (contain comments/greater detail on the specific implementation of each project).

1. **Baseline (scalar) vs auto-vectorize:**

Table 1: Time per kernel without SIMD speedup averaged over 30 trials

Kernel	Saxpy	Elementwise	1d 3-point stencil
L1 (size =480KiB)	18.65µs	30.60µs	29.8µs
L2 (size = 1.25MiB)	72.20µs	81.55µs	56.75µs
LLC (size = 24MiB)	1452.60µs	1888.15µs	1425.70µs
DRAM (size = 128 MiB)	8917.50µs	9745.50µs	8828.90µs

Table 2: Time per kernel with SIMD speedup averaged over 30 trials

Kernel	Saxpy	Elementwise	1d 3-point stencil
L1 (size =480KiB)	7.55µs	5.75µs	7.90µs
L2 (size = 1.25MiB)	14µs	43.70µs	24.15µs
LLC (size = 24MiB)	751.15µs	1530.10µs	883.50µs
DRAM (size = 128 MiB)	5394.55µs	9333.20µs	6073.10µs

Table 3: Time per kernel with SIMD speedup averaged over 30 trials

Kernel	Saxpy	Elementwise	1d 3-point stencil
L1 (size =480KiB)	2.47	5.32	3.77
L2 (size = 1.25MiB)	5.16	1.87	2.35
LLC (size = 24MiB)	1.93	1.23	1.61
DRAM (size = 128 MiB)	1.65	1.04	1.45

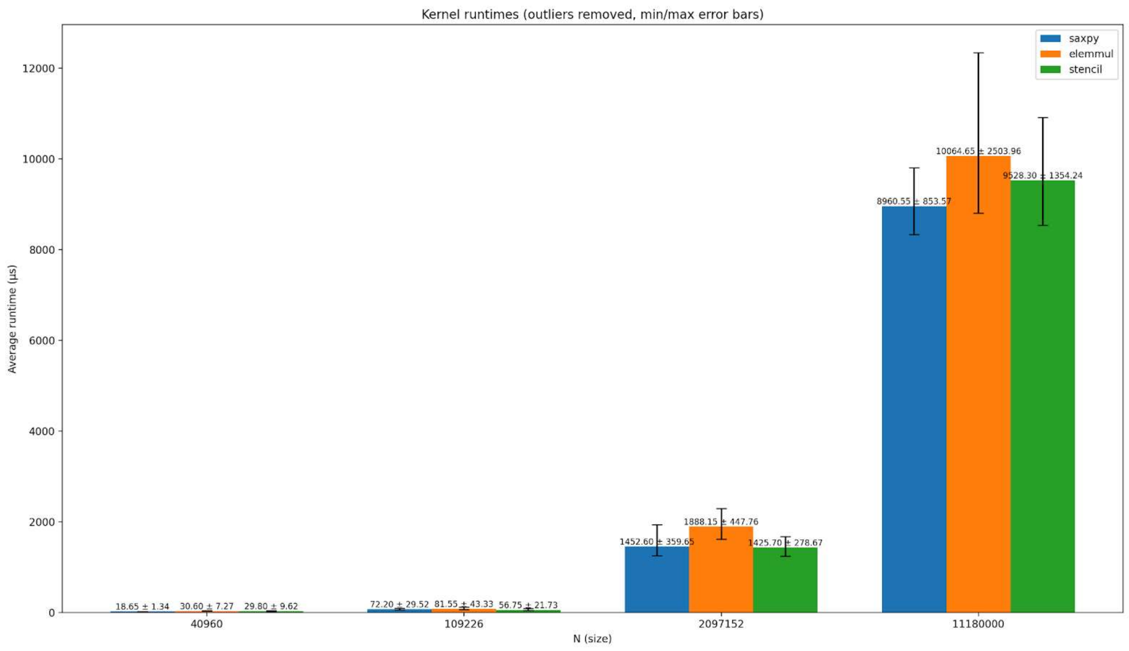
The SIMD speedup is highest when the kernel is compute-bound and data fits in fast caches. For example, the elementwise kernel in L1 achieves over 5× improvement because SIMD packs many arithmetic operations into wide vector instructions, effectively using fused multiply-add (FMA) and reducing loop overhead. Similarly, Saxpy in L2 shows strong gains (~5×) since memory bandwidth at that level is still sufficient to feed the vector units, allowing SIMD to exploit efficient load/store and arithmetic parallelism. In these cases, the bottleneck is the processor's execution units, and SIMD directly addresses that by increasing per-cycle throughput.

As the working set grows into LLC and DRAM, the kernels shift toward being memory-bound, and SIMD speedups compress. For elementwise and stencil operations at LLC and DRAM levels, the processor spends more cycles waiting for data from memory, limiting how much the wider vector units can be utilized. Even though SIMD reduces instruction count, it cannot overcome bandwidth ceilings or long memory access latencies. This is why elementwise speedup drops close to 1× off-core, while Saxpy and stencil sustain only ~1.3–1.9× improvements.

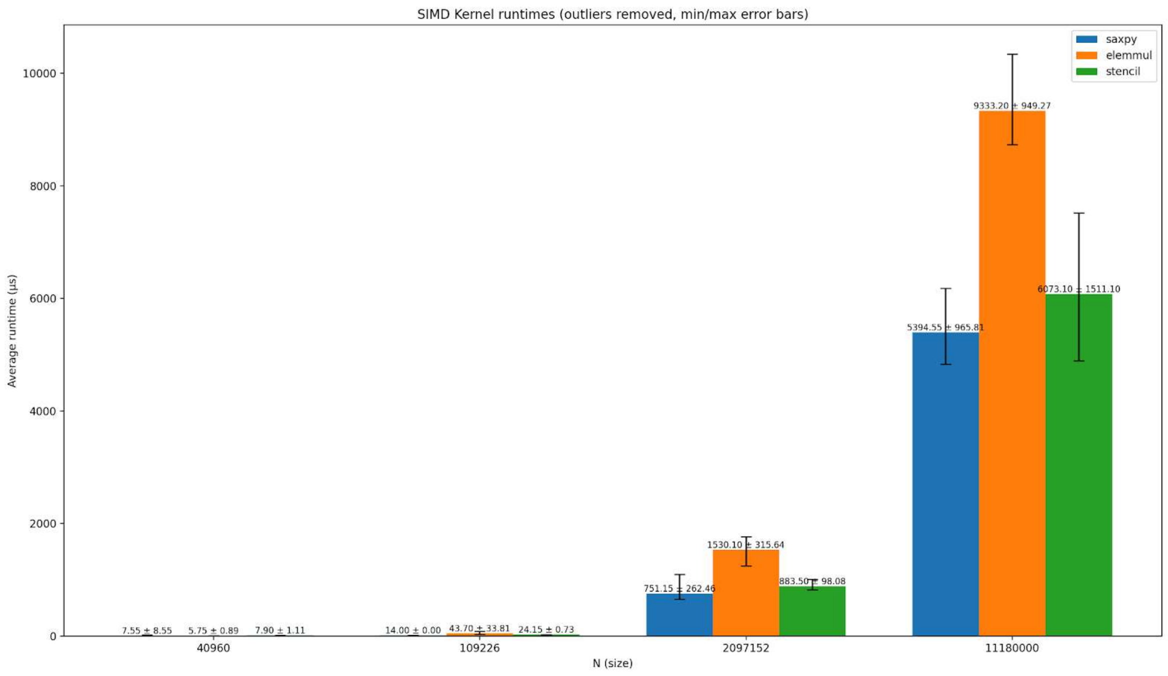
The differences across kernels also reflect their arithmetic intensity and access patterns. Elementwise benefits most in L1 because it performs more math per element, amplifying SIMD’s strengths, but loses that advantage once memory limits dominate. Saxpy has low arithmetic intensity, so its gains are moderate and taper quickly. The 3-point stencil lies in between: it shows good L1/L2 gains due to spatial reuse of neighboring elements but only modest benefits at LLC and DRAM. Overall, SIMD delivers the most speedup where the computer dominates, but memory hierarchy affects cap performance once workloads exceed cache capacity.

Reporting Uncertainties:

Graph 1: Time taken per memory level for different kernels within 95% accuracy

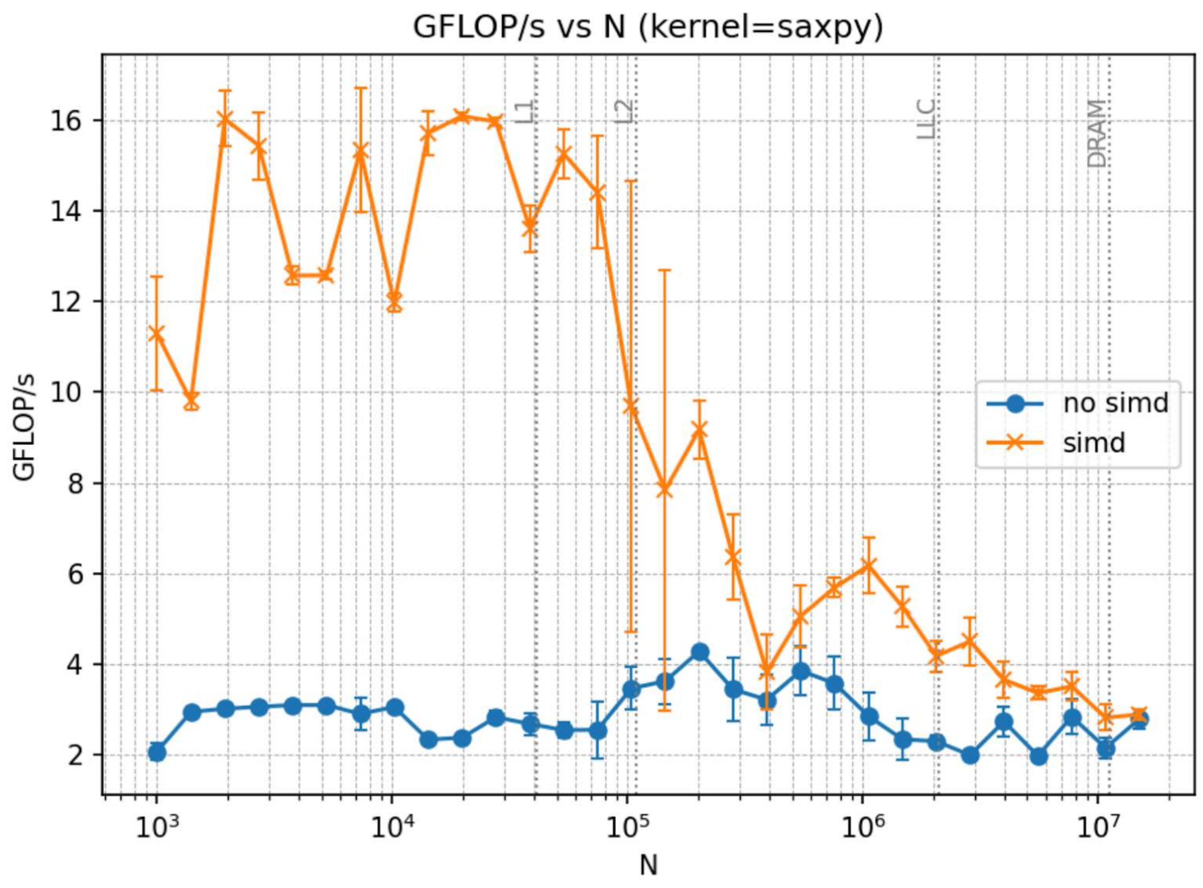


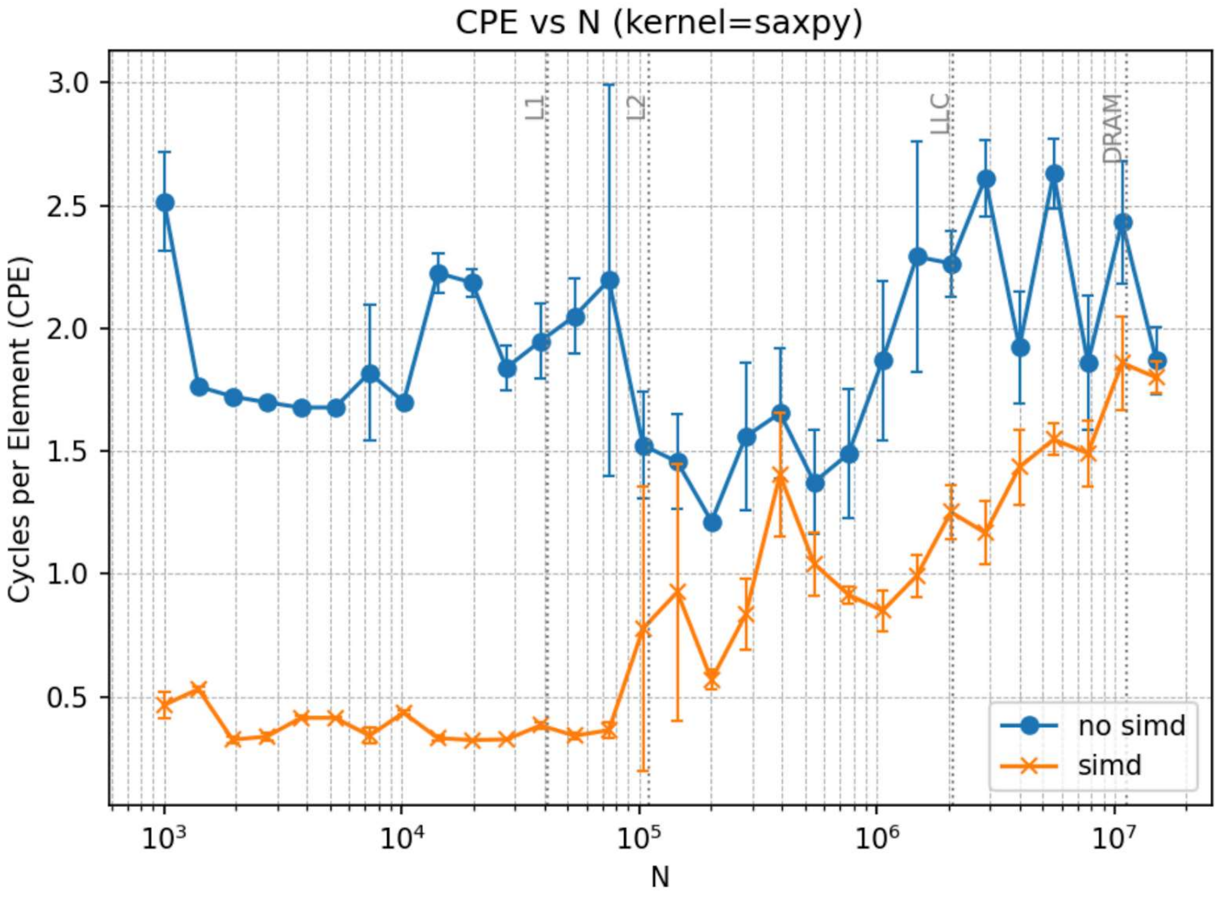
Graph 2: Time taken per memory level for different kernels using SIMD


SIMD Kernel runtimes (outliers removed, min/max error bars)
saxpy
elemmul
stencil
Average runtime (µs)
0
2000
4000
6000
8000
10000
7.55 ± 8.55
5.75 ± 0.89
7.90 ± 1.11
14.00 ± 0.00
43.70 ± 33.81
24.15 ± 0.73
751.15 ± 262.46
1530.10 ± 315.64
883.50 ± 98.08
5394.55 ± 965.81
9333.20 ± 949.27
6073.10 ± 1511.10
40960
109226
2097152
11180000
N (size)

2. Locality (working-set) sweep

The following graphs are values averaged over 30 trials, where the dashed lines represent the standard deviation. The variation significantly increases within the transition from L2 to LLC. This can be explained by the large GLOP/s and CPE differences between these two cache's. Due to this, variance occurs due to either having values prefetched and stored in L2, or having to be fetched from LLC.



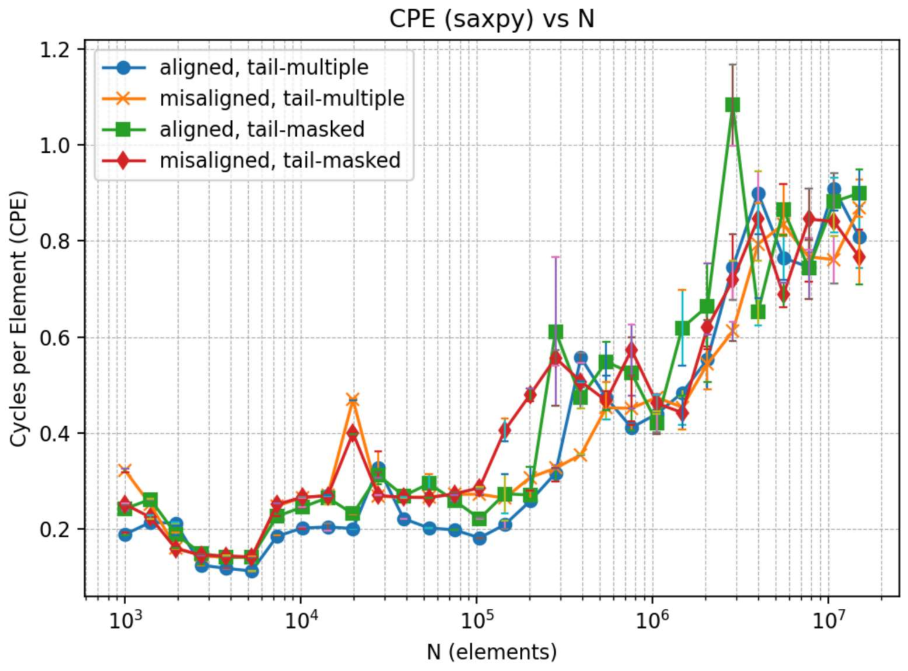


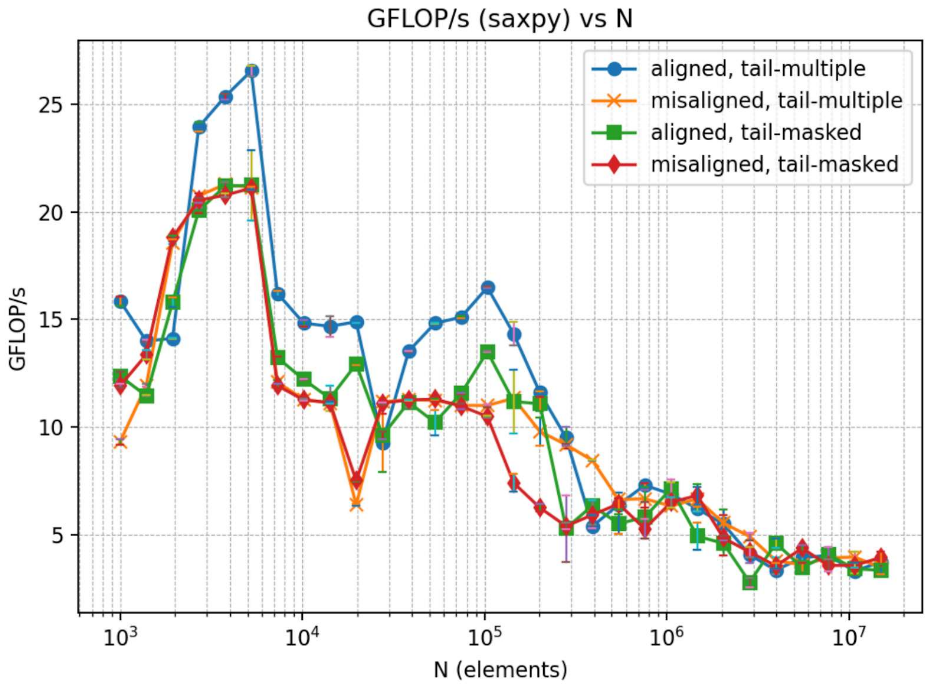
SIMD gains compress most noticeably when moving from the L2 cache to the LLC. At the L2 level, data can still be accessed with relatively low latency, allowing vector instructions to operate close to their peak efficiency. However, once the working set spills into the LLC, the situation changes: memory latency grows significantly, and bandwidth becomes a stronger constraint. In this region of the hierarchy, the time the processor spends waiting on data overshadows the savings achieved through vectorization.

While SIMD reduces the number of instructions and enables multiple operations per cycle, these benefits depend on a steady flow of data into the execution units. As access times increase in the LLC, vector lanes often sit idle, unable to be fully utilized.

This imbalance means that the performance gap between scalar and SIMD code narrows considerably, when we go towards the DRAM scale.

3. Alignment & Tail-handling

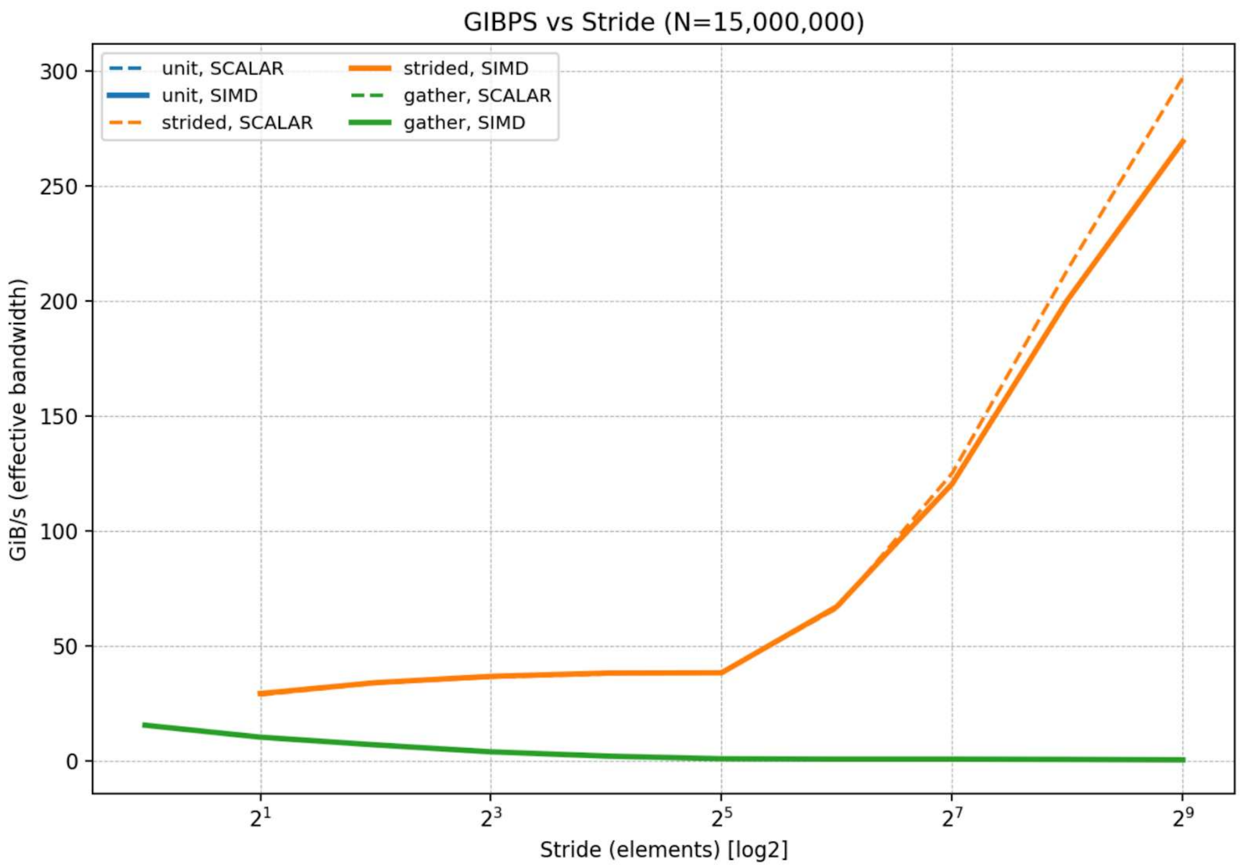




Aligned + tail-multiple: This case delivers the highest throughput across L1 to LLC, averaging about 25% higher than the other configurations in these ranges of n. As the working set grows and moves beyond LLC, the performance gap between this and the other three options narrows significantly.

Aligned + masked tail: Typically the second-best performer, reaching about 5% higher throughput than the misaligned variants. The small penalty comes from the masked epilogue, which adds a few extra instructions but has minimal effect at large n.
Misaligned cases: While misaligned + tail-multiple is often expected to be the slowest due to unaligned load overhead in L1–LLC, an interesting trend appears near the LLC–DRAM transition: the misaligned + tail-multiple sometimes surpasses the aligned + masked case. This reversal can be attributed to reduced masking overhead once memory latency dominates, whereas unaligned access penalties are amortized at larger sizes.
DRAM regime: Once the working set resides in DRAM, all four lines converge closely together, illustrating the shift to a memory bandwidth bottleneck where prologue/epilogue and alignment effects become negligible.

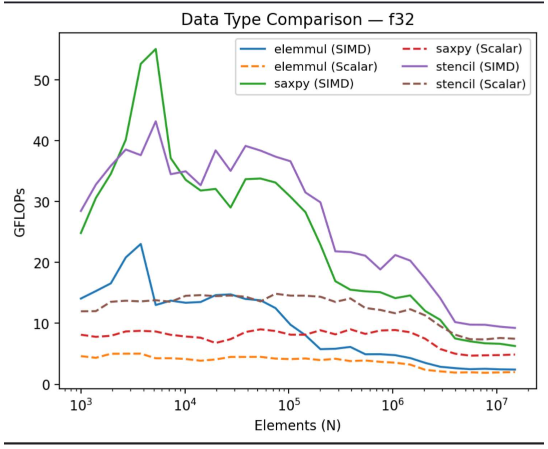
4. Stride/ gather effects

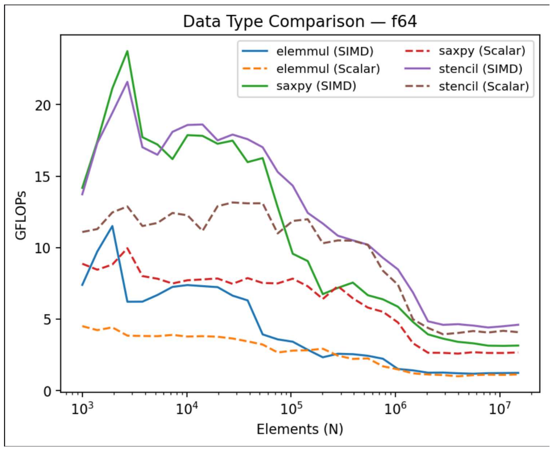


When stride increases in gather or strided access patterns, bandwidth naturally drops, sometimes close to zero. At small strides, most elements still map to the same or nearby cache lines, so prefetchers and caches can keep throughput high. As stride grows, however, each access touches a different cache line, forcing many separate memory transactions. Since only a few bytes of each cache line are used, effective bandwidth collapses and cycles per element increase sharply. This loss of spatial locality is the fundamental reason strided and gather access degrade so quickly.

It is also expected that strided SIMD bandwidth can be smaller than scalar bandwidth. Although SIMD is designed to load multiple elements at once, wide vector loads with large strides often span many cache lines. Internally, the processor may break these gather operations into a sequence of scalar loads, adding overhead. This serialization, combined with wasted cache-line traffic, makes SIMD underperform relative to scalar when access patterns are irregular. Scalar loads, while narrower, at least fetch exactly what is needed per instruction and do not suffer from lane-level serialization.
Overall, SIMD gains depend on regular, contiguous memory access where vector loads map neatly to aligned cache lines. Once stride grows, prefetching fails, bandwidth efficiency drops, and gather instructions become bottlenecked by memory transactions. As a result, strided SIMD can show not only diminishing returns but also worse performance than scalar code. This highlights the importance of access pattern optimization: SIMD delivers its highest benefit when paired with spatial locality, but becomes inefficient in memory-bound, high-stride scenarios.

5. Data types comparisons





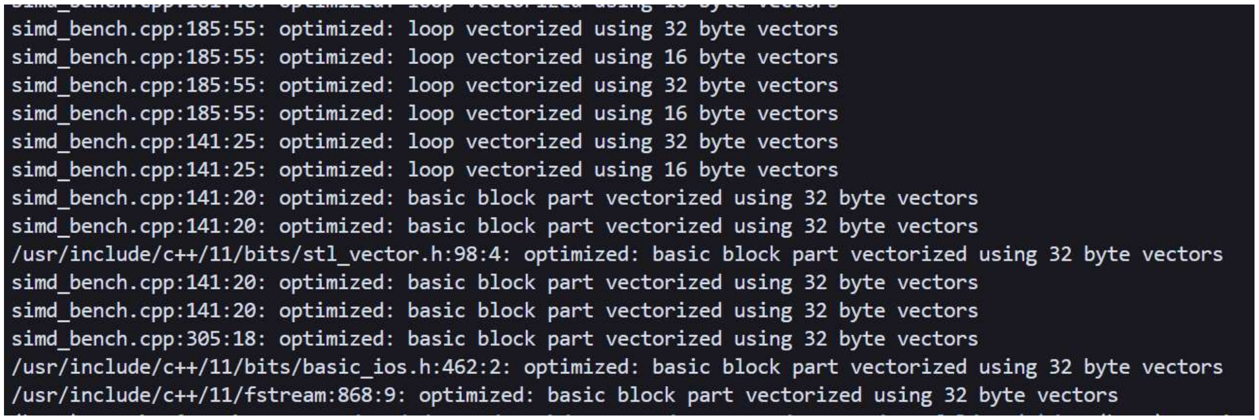
For float32 (f32), the saxpy kernel achieves the highest GFLOP/s at small element sizes due to its simple arithmetic pattern and low per-element cost. However, as n grows, the stencil kernel overtakes saxpy around n ≈ 10 5 n≈10 5 . This crossover occurs because the stencil benefits from higher arithmetic intensity and spatial data reuse, which allows it to scale better once memory effects dominate. By contrast, elementwise multiplication remains the lowest-performing kernel across all ranges, reflecting its limited reuse and relatively low flop count per memory access.
For float64 (f64), the overall trends remain the same but with throughput reduced by roughly half. This reduction is explained by the lane width effect: vector registers can pack twice as many 32-bit floats as 64-bit floats, so SIMD parallelism is effectively halved for double precision. Consequently, the same kernels show the same ordering (saxpy > stencil > elemul), but the crossover point where stencil overtakes saxpy shifts earlier.

In both cases, the results align with vector lane-width reasoning: float32 achieves higher throughput because SIMD units process 8 lanes of f32 per AVX-512 vector (or 4 lanes for AVX2), whereas only 4 lanes of f64 fit into the same registers. The relative shape of the performance curves is preserved, but float64 consistently exhibits lower GFLOP/s ceilings and earlier crossover points where memory intensity determines kernel ranking.

6. **Vectorize verifications**

While compiling the code, we can use the command flag: **-fopt-info-vec-optimized** to print out the vectorization proof/info.

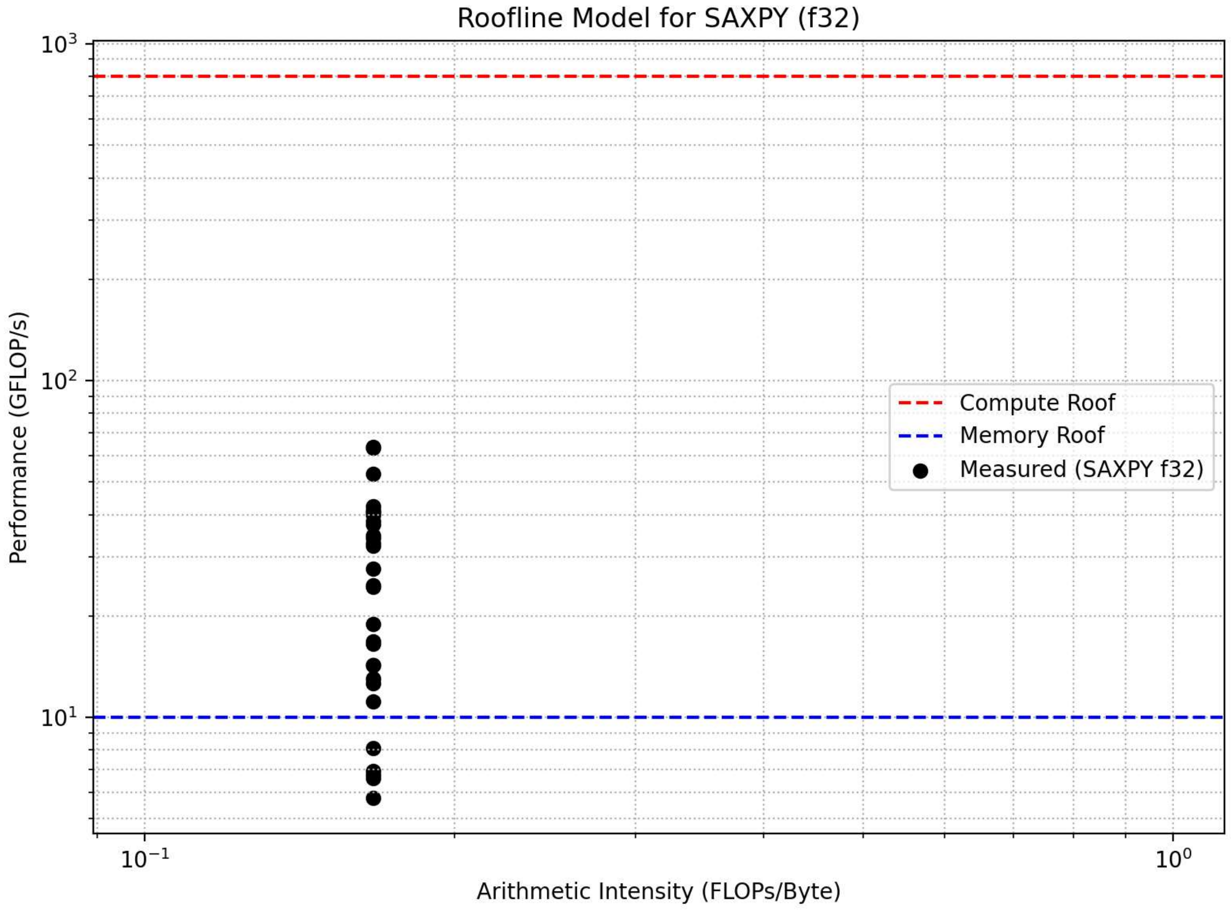
A small excerpt from this run is:



```
simd_bench.cpp:185:55: optimized: loop vectorized using 32 byte vectors
simd_bench.cpp:185:55: optimized: loop vectorized using 16 byte vectors
simd_bench.cpp:185:55: optimized: loop vectorized using 32 byte vectors
simd_bench.cpp:185:55: optimized: loop vectorized using 16 byte vectors
simd_bench.cpp:141:25: optimized: loop vectorized using 32 byte vectors
simd_bench.cpp:141:25: optimized: loop vectorized using 16 byte vectors
simd_bench.cpp:141:20: optimized: basic block part vectorized using 32 byte vectors
simd_bench.cpp:141:20: optimized: basic block part vectorized using 32 byte vectors
/usr/include/c++/11/bits/stl_vector.h:98:4: optimized: basic block part vectorized using 32 byte vectors
simd_bench.cpp:141:20: optimized: basic block part vectorized using 32 byte vectors
simd_bench.cpp:141:20: optimized: basic block part vectorized using 32 byte vectors
simd_bench.cpp:305:18: optimized: basic block part vectorized using 32 byte vectors
/usr/include/c++/11/bits/basic_ios.h:462:2: optimized: basic block part vectorized using 32 byte vectors
/usr/include/c++/11/fstream:868:9: optimized: basic block part vectorized using 32 byte vectors
```

The compiler vectorization report shows that loops were vectorized using both 16-byte (SSE, 4-wide floats) and 32-byte (AVX, 8-wide floats) vectors. This indicates GCC generated multiple SIMD code paths.

7. Roofline Interpretations



Measured values are closer to the memory roof, and are therefore memory bound.