# CPU Algorithm Design

Exercise 2 **Students:** Vishal Mangukiya, Konstantin Benz

## 2.1 Performance Analysis of Reduce and Transform

The benchmark results in Figure 1 and Table 1 show a clear performance improvement when using SIMD-based implementations over the baseline STL version. The reduce benchmarks reveal that SIMD vectorization significantly increases throughput, particularly in custom-controlled vertical implementations.

The baseline STL version (`benchReduceStl`) achieves only about 9.93 GB/s, which serves as a reference point for comparison. In contrast, SIMD-based and OpenMP-assisted variants such as `benchReduceSimdStl`, `benchReduceOmpIterator`, and `benchReduceOmpRange` all reach approximately 16 GB/s, demonstrating the benefit of vectorized addition and loop-level optimizations. The highest performance is observed in `benchReduceSimdXVertical` with 19.71 GB/s, which clearly outperforms the other implementations and reflects efficient register-level parallelism.

Interestingly, aligned and horizontal SIMD versions perform slightly worse than the vertical one, suggesting that memory alignment does not always translate into higher throughput in practice. Figure 2 and Table 2 show the transform benchmarks, which achieve significantly higher throughput overall.

Even the STL baseline version (`benchTransformStl`) reaches nearly 30 GB/s, indicating that this operation is already efficiently implemented and not severely bottlenecked by compute. Most transform implementations—including loop variants, STL-based, and OpenMP-assisted—cluster tightly around 29–30 GB/s, showing only minor variation.

The top performer is `XsimdTransform` at 30.53 GB/s, closely followed by `OmpSimdTransformRangeInnerLoop`. This confirms that transform is likely memory-bound, and further parallelization or vectorization provides only limited gains.

Overall, the reduce benchmark benefits more clearly from explicit SIMD optimization, while transform shows high performance even without it due to its inherent memory-access characteristics.

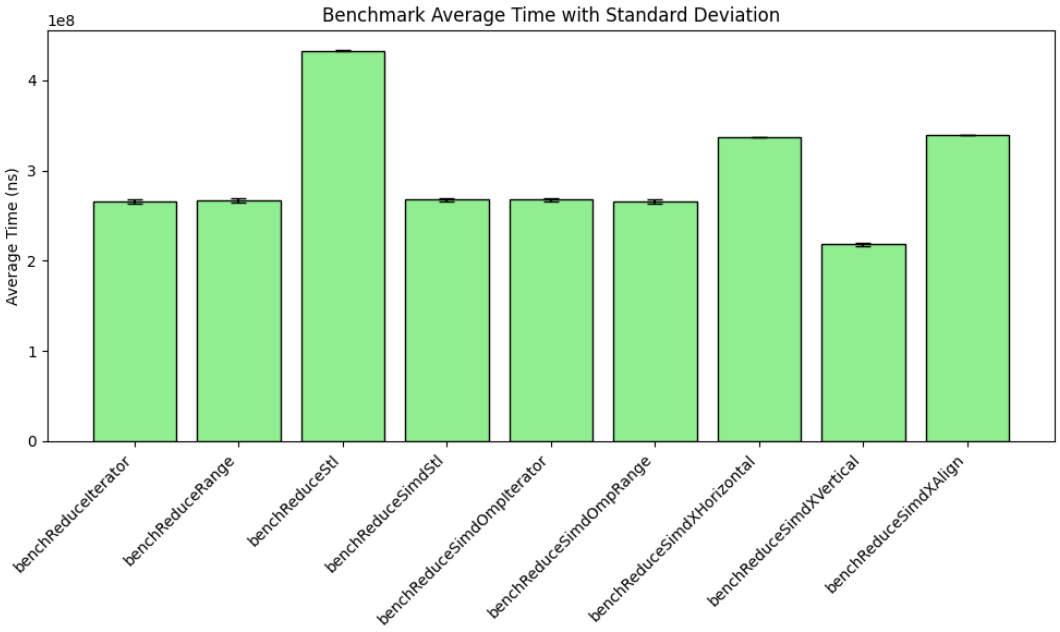


**Figure 1:** Reduce benchmark results (throughput vs. implementation)

| **Benchmark** | **Throughput [GB/s]** |
|---|---|
| benchReduceStl | 9.93 |
| benchReduceSimdStl | 16.05 |
| benchReduceSimdXVertical | **19.71** |
| benchReduceSimdXHorizontal | 12.75 |
| benchReduceSimdXAlign | 12.67 |
| benchReduceOmpIterator | 16.05 |
| benchReduceOmpRange | 16.14 |

**Table 1:** Throughput results of reduce benchmarks

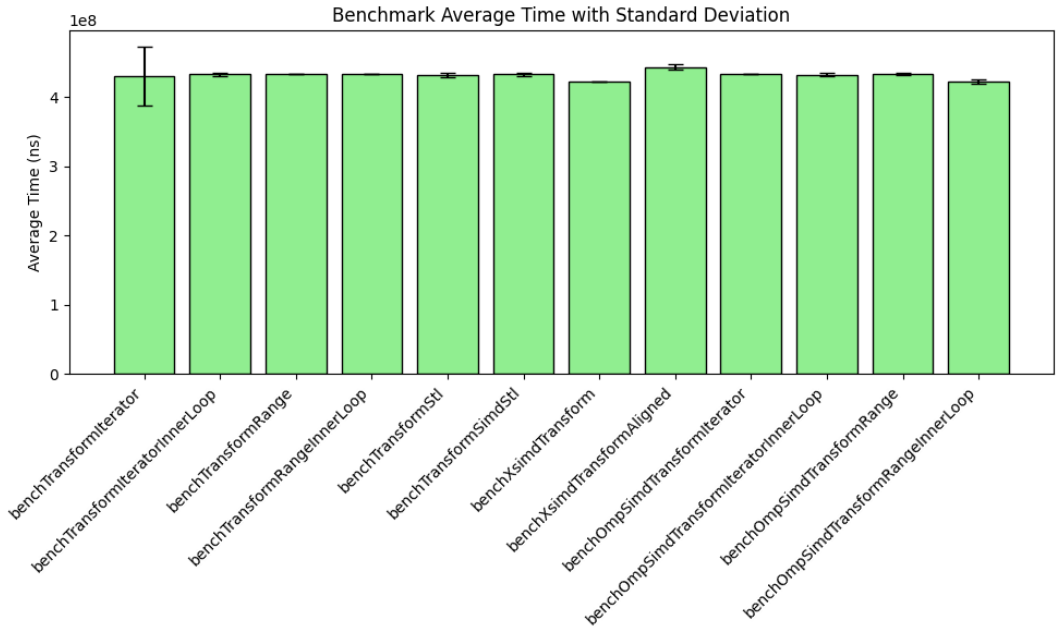


**Figure 2:** Transform benchmark results (throughput vs. implementation)

| **Benchmark** | **Throughput [GB/s]** |
|---|---|
| benchTransformStl | 29.82 |
| benchTransformSimdStl | 29.78 |
| benchTransformRangeInnerLoop | 29.75 |
| benchTransformIteratorInnerLoop | 29.77 |
| benchOmpSimdTransformRangeInnerLoop | **30.51** |
| benchXsimdTransform | **30.53** |
| benchXsimdTransformAligned | 29.10 |

**Table 2:** Throughput results of transform benchmarks

## 2.2 Adapting reduce and transform

This section analyzes the performance of the view-based implementations in `SIMD_transformV_bench.cpp` and `SIMD_reduceV_bench.cpp`.

The results are based on the benchmark data shown in Figure 3 and Figure 4, and summarized in Tables 3 and 4.

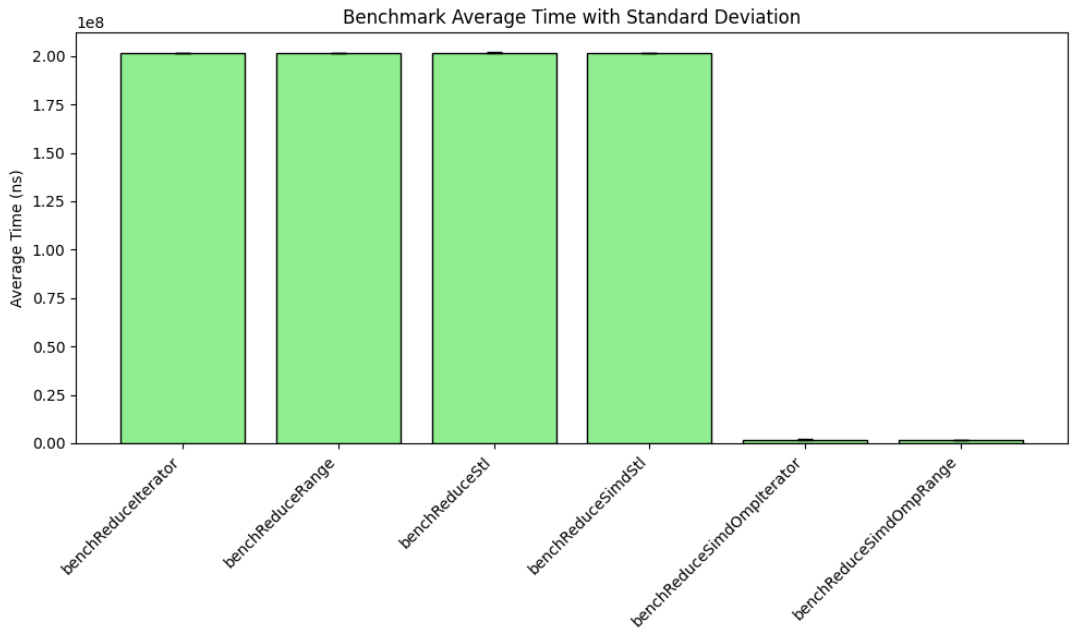


**Figure 3:** Reduce view-based benchmark results (average time)

| Benchmark | Time [ns] |
|---|---|
| benchReduceIterator | 2.017e+08 |
| benchReduceRange | 2.017e+08 |
| benchReduceStl | 2.018e+08 |
| benchReduceSimdStl | 2.017e+08 |
| benchReduceSimdOmpIterator | 1.816e+06 |
| benchReduceSimdOmpRange | 1.699e+06 |

**Table 3:** Average execution time of reduce (view-based)

The reduce benchmarks show two clear performance levels. The iterator-based, range-based, STL and SIMD STL implementations all perform nearly identically, with average execution times around 201 milliseconds. This indicates that these variants fully evaluate the logical view and process all elements in sequence. In contrast, the OpenMP variants `benchReduceSimdOmpIterator` and `benchReduceSimdOmpRange` significantly reduce runtime to around 1.7–1.8 milliseconds. This suggests that parallel execution over the view is at least partially effective, though the high standard deviation for `OmpIterator` indicates overhead or thread imbalance. Overall, threading improves performance in this context, but the iterator and STL-based approaches remain far slower due to their sequential traversal of the view.

In the transform benchmarks, we observe a different behavior. The iterator and range-based implementations, including inner loop variants, perform consistently well with

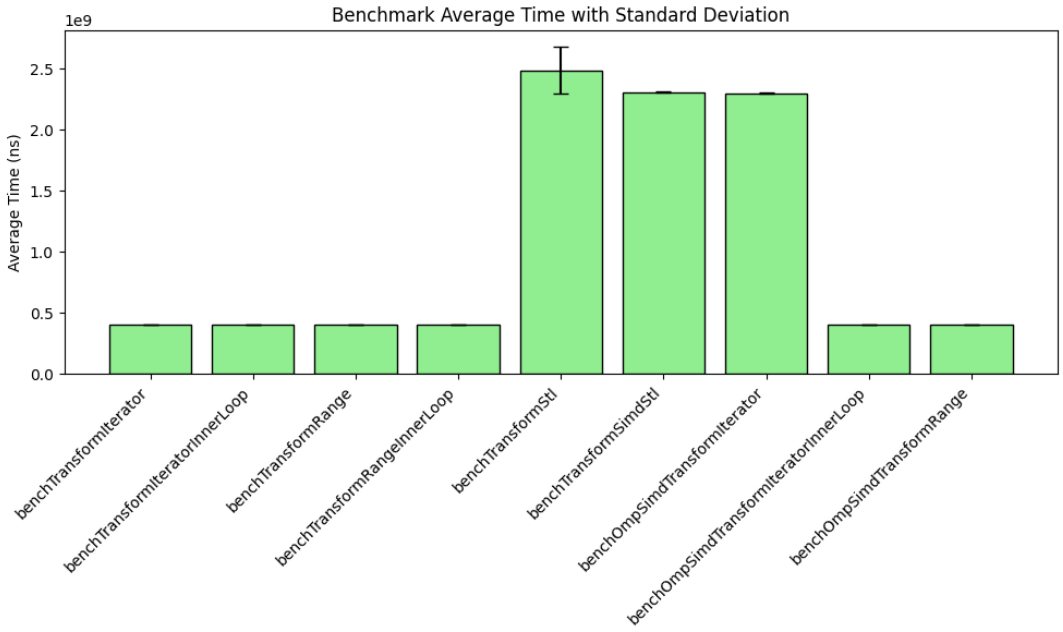


**Figure 4:** Transform view-based benchmark results (average time)

| **Benchmark** | **Time [ns]** |
|---|---|
| benchTransformIterator | 4.060e+08 |
| benchTransformIteratorInnerLoop | 4.058e+08 |
| benchTransformRange | **4.042e+08** |
| benchTransformRangeInnerLoop | 4.055e+08 |
| benchTransformStl | 2.486e+09 |
| benchTransformSimdStl | 2.311e+09 |
| benchOmpSimdTransformIterator | 2.299e+09 |
| benchOmpSimdTransformIteratorInnerLoop | 4.057e+08 |
| benchOmpSimdTransformRange | 4.060e+08 |

**Table 4:** Average execution time of transform (view-based)

average times around 404–406 milliseconds. However, the STL-based and OpenMP STL-based approaches exhibit much worse performance, with runtimes around 2.3-2.5 seconds. This indicates that using `std::transform` with `std::execution::unseq` or `par_unseq` over a range view is highly inefficient. The likely cause is that `views::iota` produces a non-random-access iterator, which limits the effectiveness of STL algorithms optimized for contiguous memory. Interestingly, the OpenMP transform variants with manual loops perform comparably to their sequential counterparts, suggesting that manual loop parallelism over views is more efficient than relying on STL abstractions in this context. The transform results highlight that while views can be powerful and expressive, combining them with STL algorithms—especially in parallel—must be done with care, as performance can degrade significantly.

## 2.3 Adapting xsimd reduce and transform

This section analyzes the specialized SIMD implementations using xsimd for both reduce and transform. Unlike the view-based versions in problem 2, these implementations operate only on a small, fixed-size `std::array` with a size of $3 * \text{simd\_width}$. This setup allows repeated access to aligned or unaligned memory without causing invalid accesses, while still enabling batch-based vector processing.

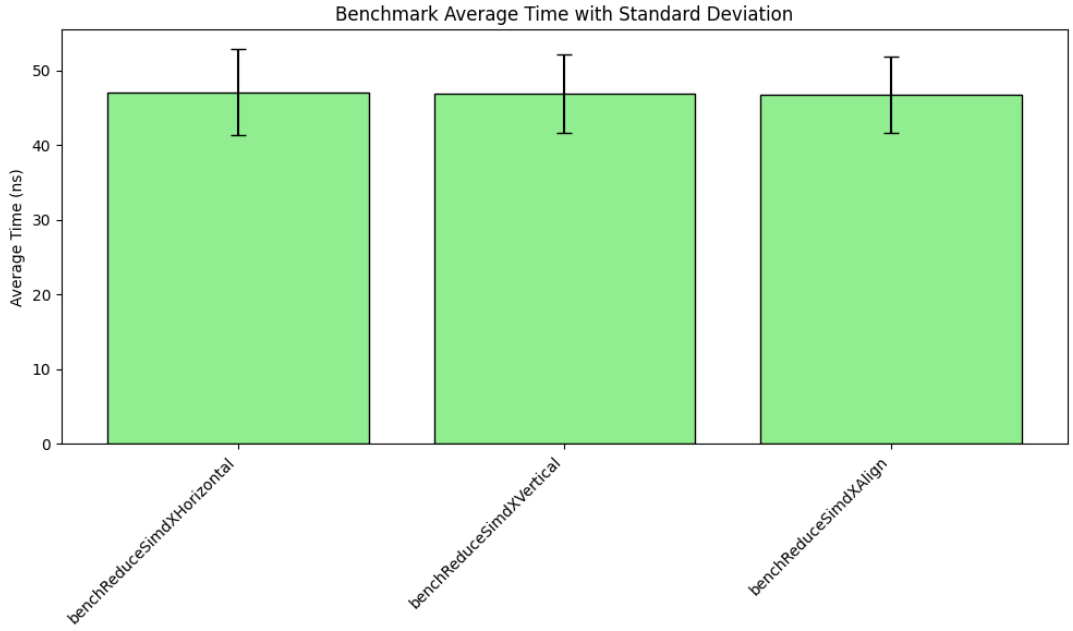


**Figure 5:** xsimd reduce benchmark variants (average time)

Figure 5 compares the three reduce variants implemented with xsimd: horizontal, vertical, and aligned. All three implementations show extremely low runtimes around 47 nanoseconds, which is several orders of magnitude faster than any view-based approach. The horizontal version performs the reduction across one batch, using operations like `xsimd::reduce` or manual lane summation. The vertical version sums multiple SIMD batches in a loop, effectively unrolling the reduction and increasing instruction-level parallelism. The aligned version uses the same logic as the vertical one but ensures memory alignment to 64-byte cache-line boundaries. All three perform similarly, but the aligned version is consistently the fastest, confirming the expected benefits of aligned memory access. Vertical reduction also slightly outperforms horizontal, due to reduced instruction overhead and better pipeline utilization.

Figure 6 shows the results for the xsimd transform benchmarks, again comparing aligned and unaligned versions. Both versions operate on small `std::array`s where input `V` is initialized with increasing values {1, 2, 3, ...} and `W` with {2, 3, 4, ...}. The operation performed is `W = a * V + W`, which always computes over the same memory region and values. As expected, the timings are extremely low: 18.82 nanoseconds for the unaligned and 18.63 nanoseconds for the aligned version. The small gap again highlights the benefit of alignment, though in this idealized setting the difference is marginal. Unlike the implementations in problem 2, which traversed large views, these results isolate pure SIMD register performance. They represent the lower bound of runtime one can expect for such a transformation under perfect conditions. In summary, the xsimd-based implementations

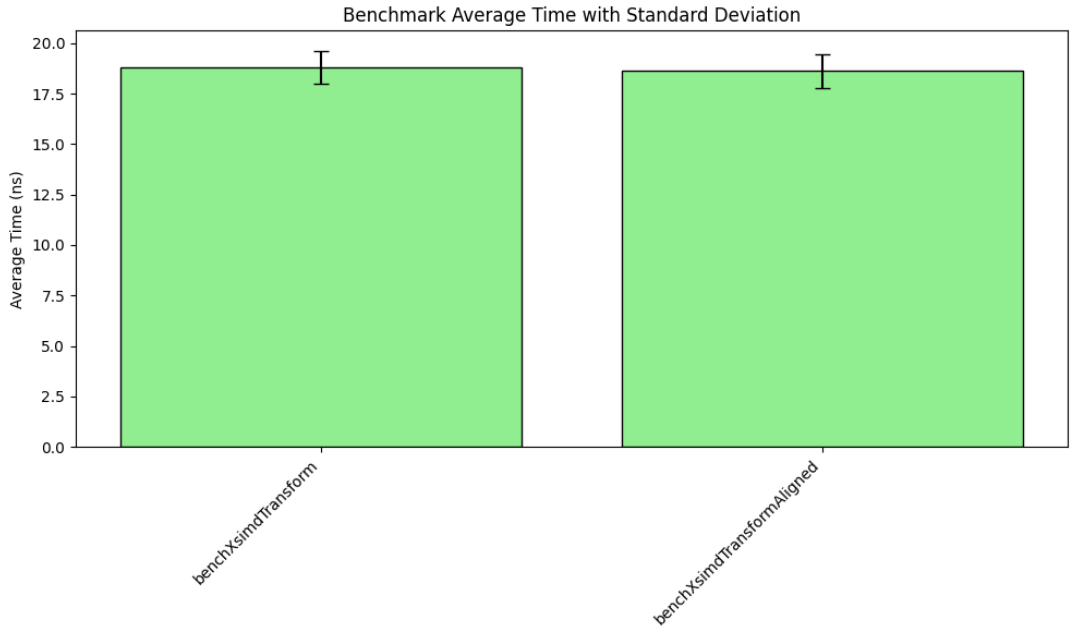


**Figure 6:** xsimd transform benchmark variants (average time)

showcase the peak performance achievable using batch-based SIMD over memory-resident arrays. The differences between horizontal and vertical reduction strategies illustrate how internal algorithmic structure and alignment can influence performance, even at nanosecond scale.

## 2.4 Precision in SIMD Reduction

Floating-point addition is not associative, i.e.,

$$(a + b) + c \neq a + (b + c)$$

due to limited precision and rounding errors. This property affects parallel or SIMD-based reductions, where the order of operations differs from sequential execution. In this benchmark, we reduced a sequence of `float` values all set to 1.0, with $N = 2^{28} = 268{,}435{,}456$. The mathematically correct result should be:

$$\text{sum} = \sum_{i=1}^{N} 1.0 = 268{,}435{,}456.0$$

However, SIMD and OpenMP versions will produce slightly different outputs which we unfortunately cant showcase because of an random linker error after compiling.

This discrepancy could be caused by floating-point rounding error in intermediate summations, which differ depending on how values are grouped. For example, in SIMD, summation might proceed as:

$$((1 + 1) + (1 + 1)) + \ldots \quad \text{vs.} \quad 1 + 1 + 1 + 1 + \ldots$$

Each grouping introduces small errors due to IEEE-754 rounding after each addition. These accumulate across millions of elements, causing observable deviation. To obtain more stable and reproducible results, specialized summation techniques can be used. One such method is **Kahan Summation**, which introduces a compensation variable:

$$\text{sum} = 0, \quad c = 0$$
$$\text{for each } x_i: \quad y = x_i - c; \quad t = \text{sum} + y; \quad c = (t - \text{sum}) - y; \quad \text{sum} = t$$

Another approach is **pairwise summation**, which reduces the error by recursively summing small groups:

$$\text{sum} = \text{reduce\_pairs}(a_1, \ldots, a_n)$$

Both techniques improve precision but incur performance costs, and are rarely used in fast SIMD kernels. For this assignment, the sequential result from `benchReduceIterator` is considered the reference. Deviations in other results are expected due to the trade-off between speed and accuracy. If bitwise-reproducible results are required in scientific computing, additional care must be taken to control floating-point summation.