CPU Algorithm Design

Exercise 3 **Students:** Vishal Mangukiya, Konstantin Benz

3.1 Adapting reduce and transform

The input containers in `reduce_LoopUnrolling_view.hpp` and `transform_LoopUnrolling_view.hpp` were adapted to use range-based views as requested in the assignment.
In the reduction routines, the memory-backed containers were replaced with `std::views::repeat(1.0f, N)`, using `decltype(std::views::repeat(...))` to define a compatible member variable. This makes the code compute-bound and avoids unnecessary memory usage.
For the transform routines, the input values were changed to `std::ranges::views::iota(0, N)`. Additionally, the output container `W` was replaced by a fixed-size `std::vector<Real>(256)` with modulo-indexed access to enable reuse of the output buffer and simulate non-memory-bound processing.

Challenges encountered:

- Initially, the range variables `V` and `W` were only declared inside each function. However, since multiple functions need access to them, we had to promote them to class-level member variables.
- Using `std::views::repeat` or `std::views::iota` as class members required careful type declarations. Simple type aliases like `std::ranges::repeat_view` or `std::ranges::iota_view` caused type mismatch errors when assigning views with bounds.
- The correct approach was to use `decltype(std::views::repeat(...))` and `decltype(std::views::iota(...))` for the member declarations, as this ensured compatibility with the generated view types and compiler support on the cluster (GCC 14).
- Some loops were not vectorized according to compiler warnings. Since manual unrolling was explicitly requested in the assignment, we did not attempt further refactoring in these cases.

All modified functions were compiled and tested successfully via the targets `reduceVbenchmarkUnroll` and `transformVbenchmarkUnroll`.

3.2 Adapting benchTransformUnrollLoopPeelingDirective

The function `benchTransformUnrollLoopPeelingDirective` was implemented using `std::views::iota` for input generation and a fixed-size `std::vector` with modulo indexing for the output buffer. The loop is unrolled using the custom `UNROLLFACTOR` macro and processes SIMD batches via `xsimd::batch`. We ensured compatibility by using `static_assert(unroll_factor % simd_width == 0)` and unaligned load/store operations for safe access to the input and output.

We used `unrollScript.sh` to benchmark the function for various unroll factors. The results are stored in the `changingUnrollFactor/res/` directory on the cluster and in the `benchmark` folder in the submitted archive and will be analyzed in section 3.5.

3.3 Adapting benchReduceUnrollTreeDirective

The function `benchReduceUnrollTreeDirective` was implemented using a recursive tree-reduction strategy with a configurable tree degree. The reduction starts with a `std::array<Real, unroll_factor>` which is filled from the `std::views::repeat` input.

A loop successively reduces this array in-place using groups of `tree_deg` elements per node. This process continues as long as the array size is divisible by `tree_deg`. If a remainder is left (i.e. fewer than `tree_deg` values remain), a final sequential sum is performed.

This design ensures flexibility for various unroll and tree degrees. We used `static_assert(unroll_factor % tree_deg == 0)` to catch invalid configurations at compile time. The remainder of the input (`N % unroll_factor`) is processed separately via a scalar OpenMP loop.

The implementation successfully compiles and runs within the benchmark target `reduceVbenchmarkUnroll`.

3.4 Adapting benchReduceUnrollSimdXHorizontal and benchReduceUnrollSimdXVertical

Horizontal

The function `benchReduceUnrollSimdXHorizontal` performs a flat reduction of all elements within an `unroll_factor`-sized block. Each SIMD batch of `simd_width` elements is summed and accumulated into a single SIMD register, which is then reduced to scalar using `xsimd::reduce_add`.

Vertical

The function `benchReduceUnrollSimdXVertical` uses a tree-like structure of SIMD lanes. It maintains `unroll_factor / simd_width` SIMD accumulators, one for each logical reduction path. Input vectors are dispatched into different SIMD accumulators in parallel. The final scalar result is obtained by reducing all lanes individually.

3.5 Benchmarking