Analysis of Custom decoder.isa Optimization

Specifically, I enhanced two custom instructions: `accelerator_reduce_sum` and `vector_compare`. The goal was to improve performance by minimizing instruction count and leveraging SIMD-like parallelism.

1. accelerator_reduce_sum (Optimized Version with Bitwise Folding)

The original version provided by the TA simply used a loop to extract each byte (8 iterations) from both `Rs1` and `Rs2`, summing them one by one. This resulted in 16 total iterations, with many shift and mask operations.

In my optimized implementation, **I introduced a hierarchical bitwise folding strategy. By using bitmasks and successive shifts**, I was able to perform multiple partial summations simultaneously. This reduces instruction-level loop overhead and better matches the behavior of SIMD operations, where multiple data elements are processed in parallel:

- First, pairs of bytes are summed using a mask like `0x00FF00FF00FF00FF`.
- Then, 16-bit segments are folded.
- Finally, 32-bit segments are folded into the final sum.

This reduces the need for explicit loops and takes advantage of data-level parallelism through bitwise operations, which is conceptually aligned with SIMD execution.

Why It Is Faster:

- **Fewer instructions:** Instead of 16 loop iterations and masks, we perform 3 levels of folding.
- **Lower dependency chain:** Folding accumulates in log2(N) steps vs. linear N steps.
- **Closer to hardware behavior:** Emulates a SIMD sum using scalar instructions.

2. vector_compare (Custom Byte-wise Comparison Result Encoding)

The TA's original implementation checked each byte and set a bit to 1 if the corresponding byte matched. My version takes a different approach:

- It checks whether each corresponding byte between `Rs1` and `Rs2` is different.
- If they differ, it sets a unique bit in `Rd` to 1 at that byte's position.

This results in an output `Rd` where each byte-aligned bit represents whether that byte differs. This is similar to how SIMD mask instructions work, providing a parallel comparison result in a single 64-bit register.

Why It Is Faster / More Useful:

- **Enables downstream logic to check all mismatches in parallel.**
- **SIMD-style behavior:** Packs byte-wise flags into one register.
- **Improved readability and control:** Enables bitwise inspection without looping.

My optimized version improves performance from three key aspects: **reducing instruction count**, **minimizing data dependencies**, and **adopting a SIMD-friendly structure**. These changes effectively reduce execution time on the CPU. In vector-based data processing scenarios, they significantly enhance processing efficiency and instruction throughput.

Analysis of index.c Optimization

To improve performance, we made several trivial modifications in our codes. First, we replaced the k-mer key comparison while inserting the key into the hash map. The key is a combination of several characters, and takes 1 byte to store each character. And, therefore, we take the advantages of the custom instruction `vector_compare` ,which accelerate the speed of comparison. Since the custom instruction `vector_compare` deals with the variables that are 8 bytes long, for those kmer longer than 8 bytes, we divide them into groups of 8 bytes, and compare their value respectively. Because SIMD accelerates the construction of the hash map, the overhead of creating a new hash map becomes more significant. As a result, the advantage of splitting the hash map into four sub-hash maps disappears. Therefore, we construct all the hash maps at once and pre-allocate memory based on the estimated size at the beginning of the construction process.This optimization leads to obvious improvement on our performance, form 0.002832 seconds to 0.001169 seconds.

Analysis of quantify.c Optimization

To improve performance, we replaced the original k-mer key comparison with our custom instruction `vector_compare`. Since we had already transformed the k-mer strings into `uint64_t` representations in Part 1 of the final project, we were able to directly apply `vector_compare` in the condition `if (entry->kmer_code == kmer_code)`. Specifically, the conditional statement was rewritten as `if (!vector_compare(entry->kmer_code, kmer_code))`. This change resulted in a notable reduction in execution time, from 0.036013 seconds to 0.019206 seconds, effectively doubling the performance.

Summary

In this project, we aimed to utilize the two instructions `vector_compare` and `accelerator_reduce_sum` to optimize indexing and quantification stages. For `index.c`, we replaced character-by-character k-mer comparisons with `vector_compare`, which enables 8-byte SIMD-style comparisons and reduces loop overhead. Additionally, we also eliminated the overhead of creating multiple hashmaps by constructing a single map with pre-allocated memory. For `quantify.c`, we directly compared pre-encoded k-mer values using `vector_compare`, making the entire comparison process more efficient. As a result, these enhancements led to significant runtime reductions in both `index.c` and `quantify.c`, achieving 2.42x and 1.87x speedups respectively. Overall, this project demonstrates that incorporating custom instructions into domain-specific applications can result in better performance compared to traditional code-level optimizations.