Detailed Functions and Optimizations

---

1. AVX-512 Intrinsics

The AVX-512 (Advanced Vector Extensions) provides a set of SIMD instructions that can process 512 bits of data in parallel. Here's a closer look at the key AVX-512 intrinsics, how to utilize them effectively, and their implications on performance:

1.1. Common AVX-512 Intrinsics

Here's a summary of some common AVX-512 intrinsics you can use in your compiler:

- Load/Store Intrinsics:
  - `_mm512_load_ps`: Load 16 single-precision floating-point values from memory.
  - `_mm512_store_ps`: Store 16 single-precision floating-point values to memory.
- Arithmetic Intrinsics:
  - `_mm512_add_ps`: Add two 512-bit vectors of floats.
  - `_mm512_sub_ps`: Subtract two 512-bit vectors of floats.
  - `_mm512_mul_ps`: Multiply two 512-bit vectors of floats.
- Logical Intrinsics:
  - `_mm512_and_ps`: Perform bitwise AND on two 512-bit vectors.
  - `_mm512_or_ps`: Perform bitwise OR on two 512-bit vectors.
- Comparison Intrinsics:
  - `_mm512_cmp_ps`: Compare two vectors of floats for equality or other relations.

1.2. Example: AVX-512 Vector Multiplication in C++

Here's a C++ implementation that demonstrates multiplying two vectors using AVX-512:

```
include <immintrin.h> #include <iostream> void multiply_arrays(float* a, float* b, float* result, int n) { for (int i = 0; i <= n - 16; i += 16) { __m512 vecA = _mm512_load_ps(&a[i]); // Load vector A __m512 vecB = _mm512_load_ps(&b[i]); //
```

```
Load vector B __m512 vecResult = _mm512_mul_ps(vecA, vecB); // Multiply
_mm512_store_ps(&result[i], vecResult); // Store result } } int main() { const int
n = 32; float a[n], b[n], result[n]; for (int i = 0; i < n; ++i) { a[i] = i *
1.0f; b[i] = (i + 1) * 1.0f; } multiply_arrays(a, b, result, n); // Output result
for (int i = 0; i < n; ++i) { std::cout << result[i] << " "; // Should print the
product of a and b } return 0; }
```

2. Loop Unrolling

Loop unrolling is a powerful optimization technique that can improve performance by minimizing loop overhead and maximizing the use of available vectorization opportunities.

2.1. Example: Loop Unrolling with AVX-512

Here's how you can implement loop unrolling to improve performance in both C++ and assembly.

C++ Implementation with Loop Unrolling:

```
void add_arrays_unrolled(float* a, float* b, float* result, int n) { for (int i =
0; i <= n - 32; i += 32) { // Process 32 elements unrolled into two AVX-512
operations __m512 vecA1 = _mm512_load_ps(&a[i]); __m512 vecB1 =
_mm512_load_ps(&b[i]); __m512 vecResult1 = _mm512_add_ps(vecA1, vecB1);
_mm512_store_ps(&result[i], vecResult1); __m512 vecA2 = _mm512_load_ps(&a[i +
16]); __m512 vecB2 = _mm512_load_ps(&b[i + 16]); __m512 vecResult2 =
_mm512_add_ps(vecA2, vecB2); _mm512_store_ps(&result[i + 16], vecResult2); } }
```

Assembly Implementation with Loop Unrolling:

```
section .data a: times 32 dd 1.0 b: times 32 dd 2.0 result: times 32 dd 0.0 section .text global _start _start: ; Unroll loop to process 32 floats vmovaps zmm1, [a] ; Load first 16 elements vmovaps zmm2, [b] vaddps zmm1, zmm1, zmm2 ; Add first 16 vmovaps [result], zmm1 ; Store first result vmovaps zmm1, [a + 64] ; Load next 16 elements vmovaps zmm2, [b + 64] vaddps zmm1, zmm1, zmm2 ; Add next 16 vmovaps [result + 64], zmm1 ; Store second result ; Exit program mov rax, 60 xor rdi, rdi syscall
```

3. Multithreading with OpenMP

Leveraging multithreading can significantly enhance performance, especially when combined with SIMD operations. By using OpenMP, you can parallelize your computations across multiple cores of the Intel Core i9-14900K.

3.1. Multithreading Example with OpenMP

Here's how you can implement multithreading with OpenMP and AVX-512:

```
#include <immintrin.h> #include <omp.h> void add_arrays_multithreaded(float* a, float* b, float* result, int n) { #pragma omp parallel for // Parallelize the loop for (int i = 0; i <= n - 16; i += 16) { __m512 vecA = _mm512_load_ps(&a[i]); // Load 16 floats __m512 vecB = _mm512_load_ps(&b[i]); // Load 16 floats __m512 vecResult = _mm512_add_ps(vecA, vecB); // Add them _mm512_store_ps(&result[i], vecResult); // Store the result } }
```

This implementation will allow OpenMP to handle the distribution of iterations across available threads, maximizing performance by utilizing the CPU's multiple cores.

4. Combining Optimizations

When building your compiler, it's crucial to recognize opportunities for combining these optimizations:

1. Identifying Vectorizable Code: Your compiler should analyze loops and identify regions that can benefit from SIMD instructions.
2. Applying Loop Unrolling: When detecting loops with a small iteration count, apply loop unrolling to reduce overhead.
3. Integrating Multithreading: Use threading libraries like OpenMP to parallelize independent computations across multiple cores.
4. Instruction Fusion: Where possible, arrange instructions such that two or more instructions can be fused into one micro-op by the processor, reducing the number of executed instructions.

5. Native Assembly Optimization Techniques

To further optimize performance, you can consider the following native assembly techniques tailored for Intel Core i9-14900K:

- Prefetching: Use prefetch instructions like `prefetchnta` to load data into cache before it is needed, reducing cache misses.
- Branch Prediction: Arrange your code to be more predictable for the branch predictor in modern CPUs, minimizing pipeline stalls.
- Register Allocation: Optimize register usage to minimize memory access, ensuring frequently used variables remain in CPU registers.

6. Profiling and Benchmarking

Once you've implemented these optimizations, it’s essential to profile your application to identify bottlenecks. Use tools like Intel VTune Profiler or gprof to measure performance and find areas for further improvement.

Conclusion

Building a compiler that efficiently utilizes the capabilities of the Intel Core i9-14900K requires a deep understanding of its architecture, including AVX-512, effective loop unrolling, and multithreading strategies. By integrating these techniques into your

compiler, you can generate optimized code that fully leverages the power of modern CPUs.