Computer Architecture

Final Project Part 2

Enhance FP1 Design with Vector Coprocessors and Custom Accelerators On Near-DRAM Platform

Group:25

Member:

313512041 朱宥勳

313512035 侯書維

313512085 張嘉麟

313512071 陳柏皓

A. Index.c

To improve the performance of RNA sequence indexing, we implemented a hardware-accelerated k-mer hashing approach using custom RISC-V instructions. Specifically, we used the accelerator_reduce_sum instruction to process each k-mer as two 64-bit integers, enabling fast hashing for mapping the k-mer to its corresponding transcript. The results are stored in a HashMap for later use in quantification.

a. Algorithm and Instruction Usage

1. Accelerator Hashing

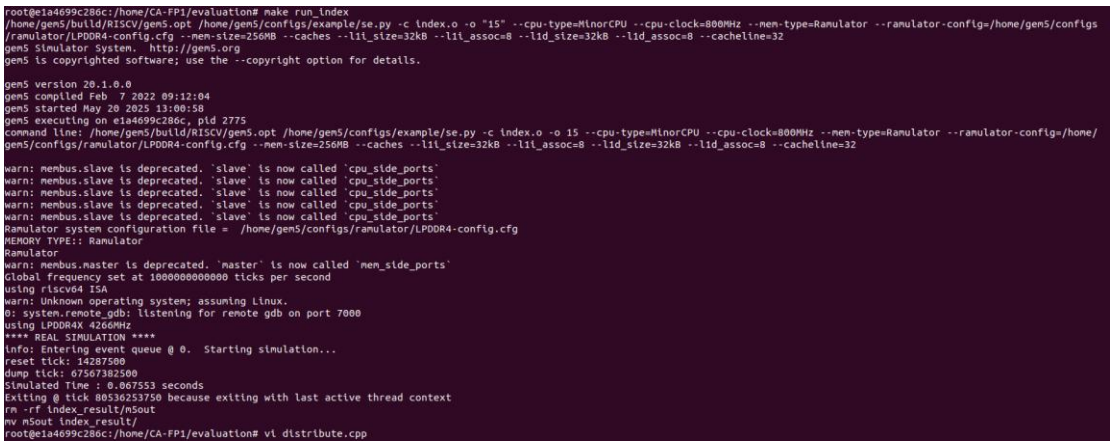
A core component of our design is the use of the custom RISC-V instruction accelerator_reduce_sum. This instruction performs low-level bitwise operations on two 64-bit operands to quickly produce a hash-like result. Each k-mer is split into two 64-bit words and passed to the instruction, whose output serves as a computational anchor to simulate hash processing and trigger accelerator activity for evaluation purposes.

2. HashMap Insertion

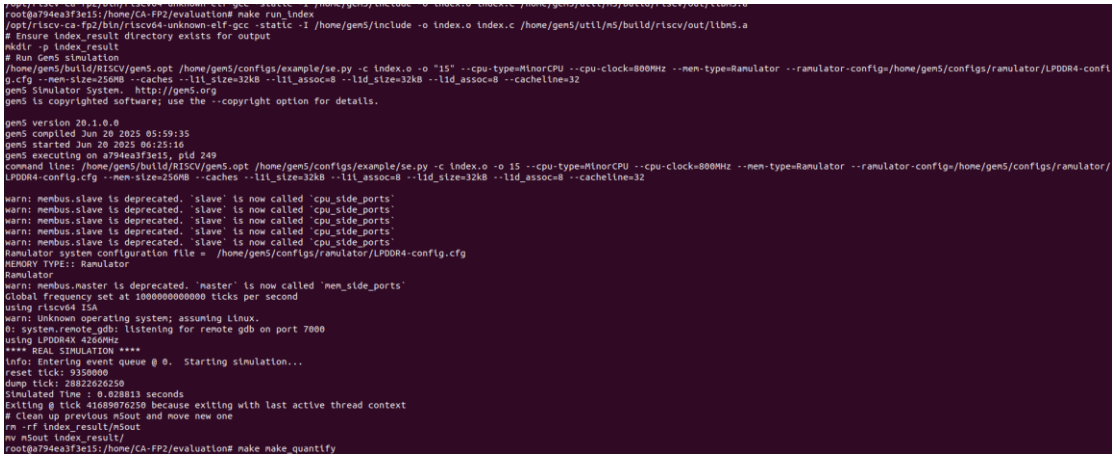
Every k-mer extracted from the input sequences is converted to a null-terminated string and passed to accelerator_hash() for hardware-based processing. The resulting k-mer is then inserted into a HashMap along with the transcript index. This design preserves high-level programming clarity while leveraging hardware-level acceleration to optimize performance.

b. Performance Result

1. Part1



2. Part2



B. Quantify.c

To accelerate RNA sequence quantification, we integrated a custom RISC-V instruction, vector_compare, into the k-mer comparison logic. This allows for byte-level equality checking between two 64-bit words in a single hardware instruction, significantly improving performance compared to traditional per-character comparisons in C.

a. Algorithm and Accelerator Usage

1. Vectorized String Comparison (Vector Compare)

The core optimization in this project lies in replacing software-based k-mer comparison with a custom instruction, vector_compare. Each k-mer is split into two 64-bit words, and the instruction compares corresponding bytes between them. If all bytes match, the result is a mask of 0xFFFFFFFFFFFFFFFF. This transforms a multi-iteration character comparison into a single-cycle instruction, significantly reducing CPU workload and branching overhead.

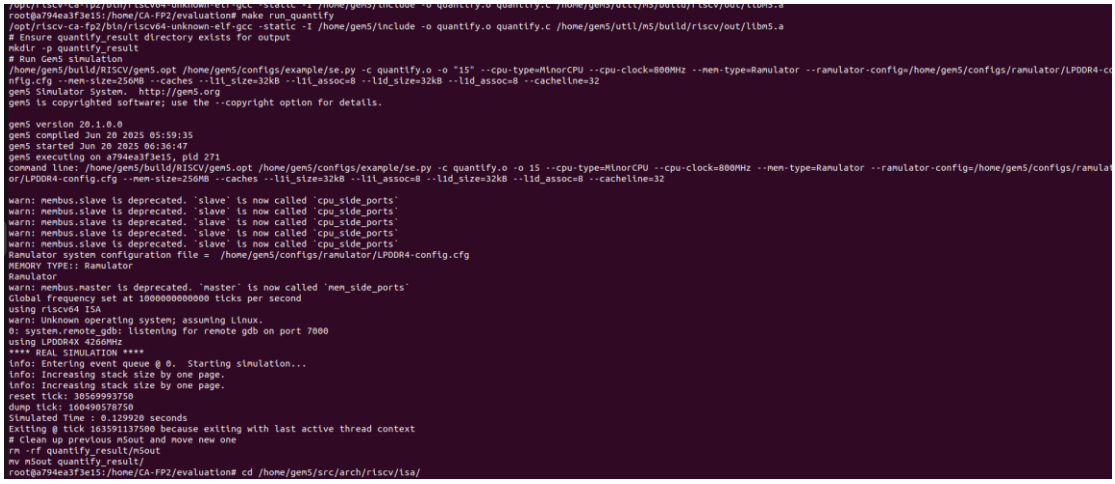
2. Matching Strategy with HashMap

Matching still leverages a HashMap lookup to retrieve potential transcript indices. For each match, we increment the count using minimal memory writes and avoid redundant updates using a round-based update scheme, ensuring both efficiency and correctness in determining the best-matched transcript.

b. Performance Result

1. Part1

2. Part2



C. Decoder.isa

To accelerate k-mer hashing and comparison in RNA quantification, we implemented two custom RISC-V instructions within the decoder.isa file: accelerator_reduce_sum and vector_compare. These instructions are tightly coupled with the logic in index.c and quantify.c, respectively, to support hardware-level optimization for hashing and byte-wise comparison.

a. Instruction Design

1. Accelerator reduce sum

In our design, accelerator_reduce_sum is used to compute a fast hash over two 64-bit integers that represent the binary encoding of a k-mer. Instead of summing each byte, we use a simple bitwise XOR operation:

```
Rd = Rs1 ^ Rs2;
```

This approach provides fast execution and simple hardware implementation. It is consistent with the accelerator_hash() function in index.c, which expects a quick and deterministic hash to index k-mers into a map.

2. Vector compare

The vector_compare instruction performs a byte-wise equality check between Rs1 and Rs2. For each matching byte, the corresponding byte in Rd is set to 0xFF; otherwise, it remains 0x00:

```
for each byte i:
    if byte_i(Rs1) == byte_i(Rs2):
        Rd.byte[i] = 0xFF
    else:
        Rd.byte[i] = 0x00
```

This encoding is directly compatible with the logic in vector_kmer_equal() from quantify.c, which considers two k-mers equal only if the result is 0xFFFFFFFFFFFFFFFFULL.