Computer Architecture Final Project

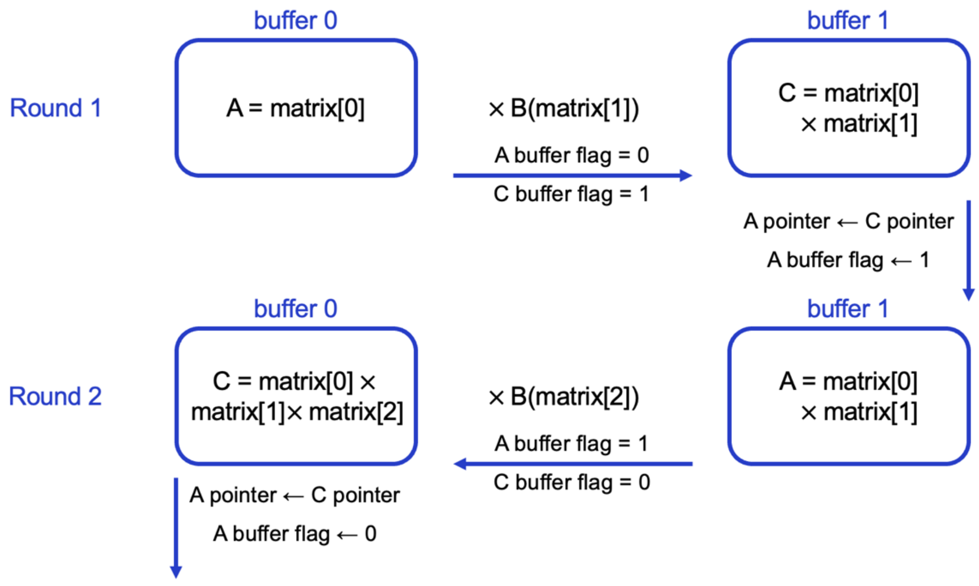
1. Design Specification

   This program performs matrix chain multiplication in RISC-V assembly. The goal is to compute a final result matrix by multiplying a sequence of input matrices. Each matrix is stored in a row-major one-dimensional array of integers. In this design, registers are carefully managed with only t0–t6, a0–a7, and s0–s11 are used.

2. Techniques and Implementation Strategies

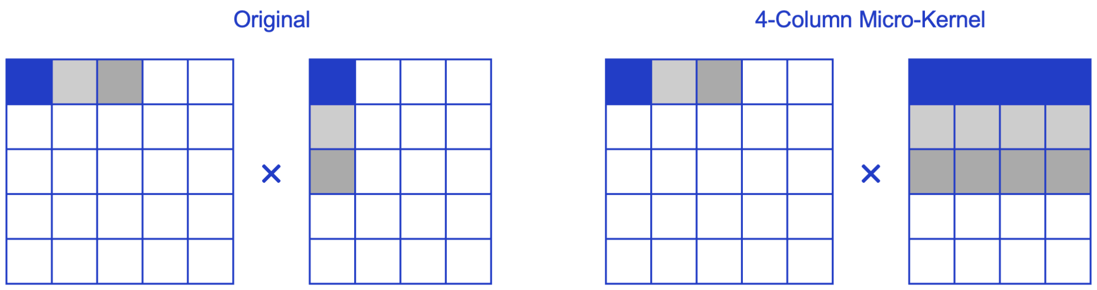
   2.1. Double buffer management

   To manage intermediate results without repeatedly allocating and freeing memory, preallocated buffers are used. More specifically, a double-buffering strategy is implemented to avoid overwriting data during matrix multiplications. Two memory buffers alternate roles in each iteration: one buffer holds the input matrix (A), and the other stores the output result (C).



2.2. 4-Column Micro-Kernel

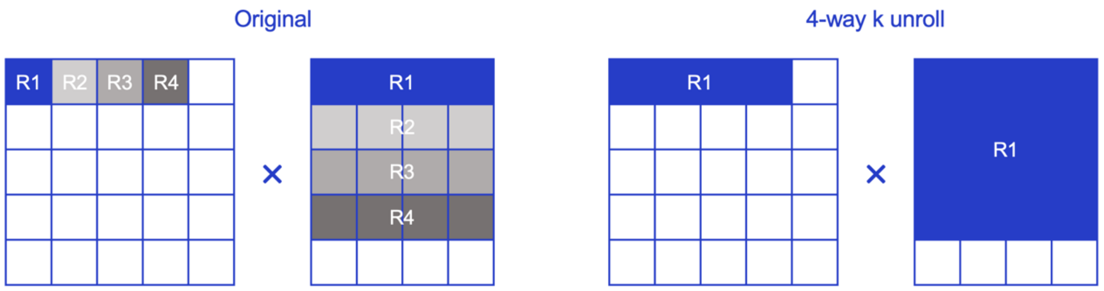
To increase throughput and reuse loaded data, the multiplication kernel is designed to compute four columns of the output matrix at a time.



In this approach, A[i][k] only needs to be loaded once and can be reused to compute four columns. Thus, the total number of iterations is reduced to one-fourth.

2.3. 4-Way Loop Unrolling on k

Instead of iterating one-by-one through the shared dimension k, each loop iteration handles four consecutive elements from a row of matrix A, along with their corresponding columns from matrix B.



This unrolling reduces the total number of loop iterations by a factor of four and minimizes branch checks. The similar processing pattern also makes better use of the cache, since the data being accessed tends to stay in nearby memory locations.

3. Overall Computation Flow

The 4-column micro-kernel is a strategy applied to the column (j) dimension, while the 4-way unrolling technique is applied to the accumulation (k) dimension. These two techniques are used together to optimize performance. Combining all techniques shown above, the overall flow of our matrix multiplication is presented as:

1) Inputs: matrices, rows, cols, count (from a0–a3)
2) Two buffers (buf0, buf1) are allocated for alternating use
3) matrices[0] is copied into buf0 as the initial matrix(A)
4) For each matrix from matrices[1] to matrices[count-1]:
   Multiply the current result (stored in either buf0 or buf1) with the next matrix.
   Each multiplication computes 4 columns at a time using a 4-way unrolled inner loop.
   The output(C) is written into the alternate buffer.
   A and C pointer are swapped.
5) After all multiplications, the final result is stored in the active buffer and returned via a0

4. Reflections

This project made us realize how challenging it is to design a program in assembly. Memory access for matrices, row and column arrays plays a critical role, yet it is not intuitive. Managing pointers and memory addresses requires careful attention, and although it is similar with C, assembly makes it much harder to understand at first glance. Debugging is also far more difficult compared to C or other high-level languages, since we cannot rely on print statements or automatic error detection. After this, we will surely appreciate any language that allows direct output. On the other hand, working at this low-level system gave us a deeper understanding of RISC-V and how a processor operates step by step. This insight allowed us to refine our algorithm with greater awareness of how data moves and is processed. Additionally, by tuning cache size, we observed that the L1 cache has the most significant impact on execution time.