Bottleneck Analysis of Matrix-Vector (Baseline)

1. Overview of the Loop

The loop in compute_tst performs:

- Load a float: vmovss (%rdx), %xmm0
- Load a float and multiply: vmulss (%rax), %xmm0, %xmm0
- Add the previous value to the result: vaddss %xmm0, (%rdx)
- Move the result into another register: vmovss %xmm0, (%rcx)

This structure represents a single iteration of a matrix-vector multiplication

2. Instruction Bottleneck Highlights

Based on the given mca report:

Instruction	Latency	RThroughput	Notes
vmovss (%rdx), %xmm0	5 cycles	0.50	Semi high latency load
vmulss (%rax), %xmm0, %xmm0	9 cycles	0.50	High latency multiply
vaddss %xmm0, (%rdx)	9 cycle	0.50	High latency addition
vmovss %xmm0, (%rcx)	1 cycle	0.50	Low latency store

Critical Observations:

- Additions and multiplications have a high cycle cost
- Floating-point operations are the primary cost.
- Memory access is moderately expensive but not the worst bottleneck.

3. Resource Bottleneck

From mca and osaca report:

- Floating-point pipelines show high pressure, particularly ports 5 and 6.
- Load/store units are moderately utilized.

Port pressure summary from osaca:

Port 0: 0.66 cycles

Port 1: 0.65 cycles

Ports 2/3: 1.00 cycles each (busy every cycle)

Port 5: 0.69 cycles

Ports 4/7/8/9: 0.50 cycles

each Port 11: 1.00 cycles

Critical Observations:

- There’s a loop-carried dependency for the vaddss instruction which limits the loop
- Floating-point operations are not too expensive.
- The store instructions are not a bottleneck

4. Loop-Carried Dependencies

From osaca:

vaddss (%rcx), %xmm0, %xmm0, 3 cycles | [87, 88]

- This loop is limited by this addition instruction

5. Summary

In total, the floating-point operations are not the most expensive instruction and neither are the load or stores. We see that there is a loop-carried dependency in the addition instruction since we are adding two specific indexes that must be loaded.

6. Final Bottleneck Diagnosis

Floating-Point Execution and Serial Dependency are the primary bottlenecks.

Each addition operation depends directly on the previous result, which creates a loop-carried dependency that limits the loop. Memory accesses could also be optimized in each iteration. The main bottleneck in this loop is the presence of floating-point additions and having to load multiple indexes from memory.

Unrolling the loop could provide the most benefit to reduce the dependencies in each loop, as well as fetching the index values before each iteration.