CS 259: GPU Microarchitecture LAB 1

Group Member: Kejun Wu, Yujie Cao

DOT8 Instruction

func7	rs2	rs1	func3	rd	opcode
3	5-bits addr	5-bits addr	0	5-bits addr	0x0B

The purpose is to add a custom instruction `VX_DOT8` to 4-core Vortex RISC-V GPGPU for accelerating integer matrix multiplication operations. This instruction computes the dot product of two 4-element int8 vectors and returns an int32 result.

$$\text{Dot Product} = (A1 * B1 + A2 * B2 + A3 * B3 + A4 * B4)$$

The instruction format is as follows:

$$\text{VX\_DOT8 rd, rs1, rs2}$$

Where each source registers rs1 and rs2 hold four int8 elements.

$$rs1 := \{A1, A2, A3, A4\}$$

$$rs2 := \{B1, B2, B3, B4\}$$

$$rd := destination\ int32\ result$$

The "vx_intrinsics.h" is modified to define the DOT8 instruction that invokes the new instruction with three operands: rs1, rs2, rd.

DOT8 Simx Implementation

`Decode.cpp`: A new case was added to recognize the case when the funct7=3 and funct3=0 for the new custom instruction DOT8.

`Execute.cpp`: New logic was added to implement dot products of rs1 and rs2. The four int8_t register values from each operand are unpacked and then the int32_t dot product is computed.

`Funct_unit.cpp`: A new ALU type is added. We assume a 2-cycle latency for its execution.

DOT8 RTL Implementation

`VX_gpu_pkg.sv`: The unused ALU instruction slot is assigned to DOT8.

`VX_config.vh`: DOT8 instruction execution latency is defined as 2 clock cycles.

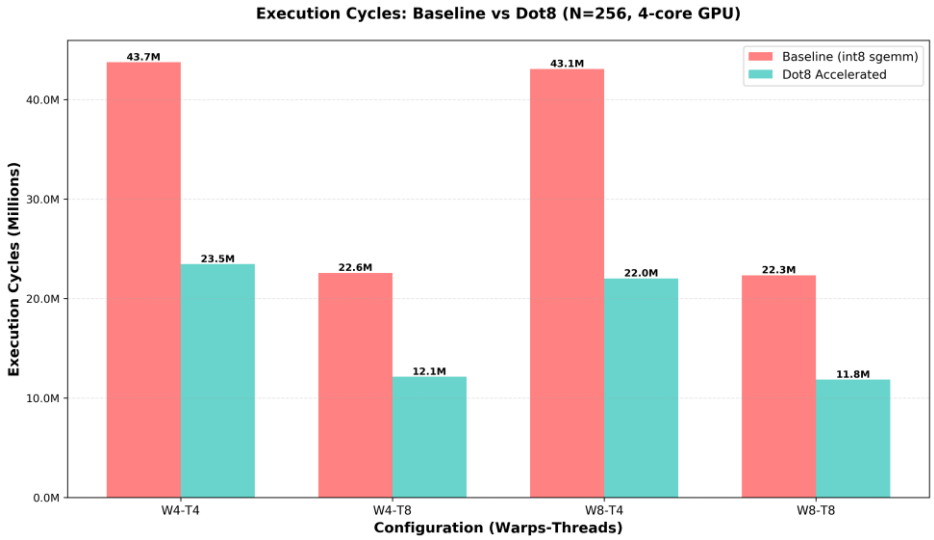
`VX_trace_pkg.sv`: Add DOT8 instruction debug trace output.

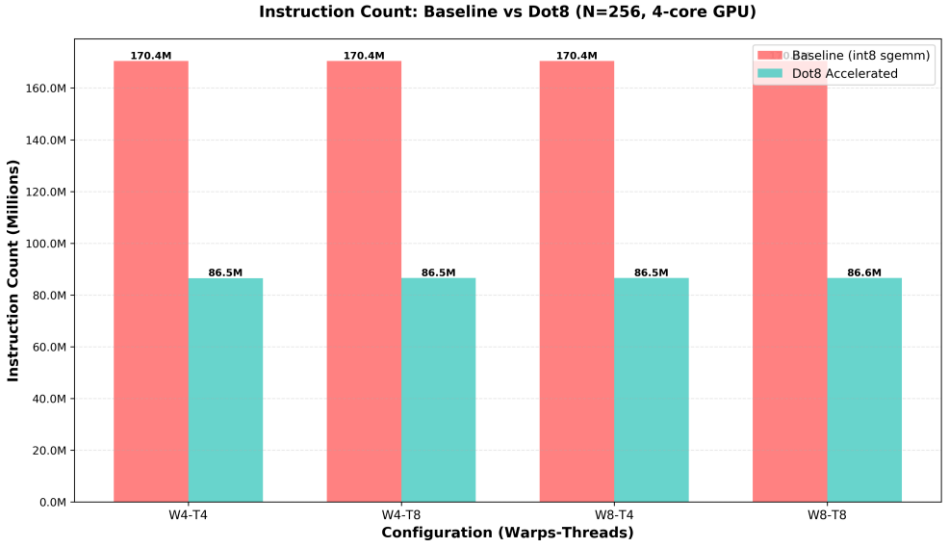
`VX_decode.sv`: This file is used to decode the new instruction. I select and specify the ALU functional unit and register usage for executing this new instruction.

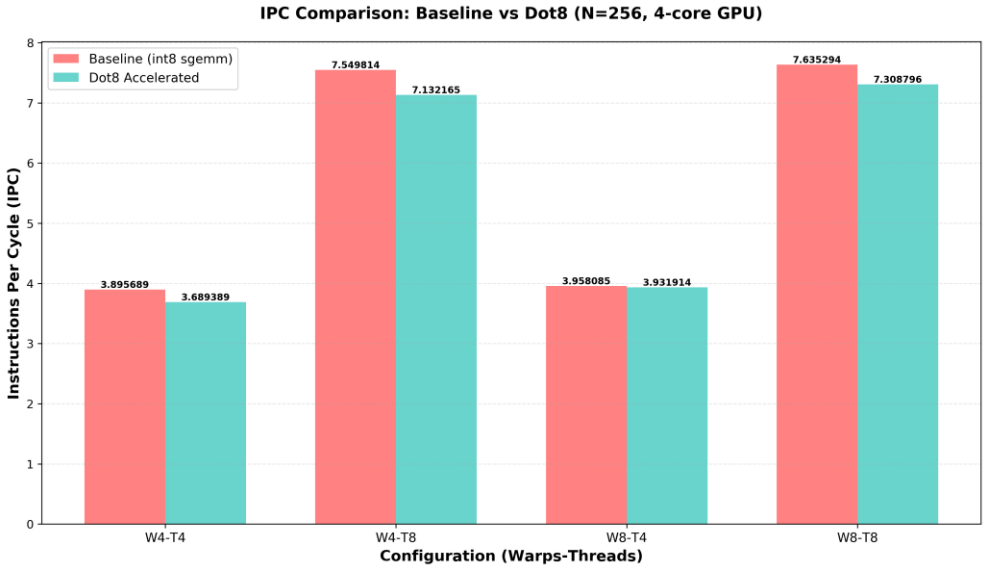
`VX_alu_dot8.sv`: This new module was created to implement DOT8. This module was used for the DOT8 hardware execution unit.

`VX_alu_unit.sv`: The new VX_alu_dot8 instance was added as a 3rd sub-unit after VX_alu_muldiv.

Results plots and analysis of SimX







Workload

- Matrix Size: 256×256
- Input Type: int8_t (8-bit signed integers)
- Output Type: int32_t (32-bit signed integers)
- Test Configurations:
  - 4 warps × 4 threads = 16 threads
  - 4 warps × 8 threads = 32 threads
  - 8 warps × 4 threads = 32 threads
  - 8 warps × 8 threads = 64 threads

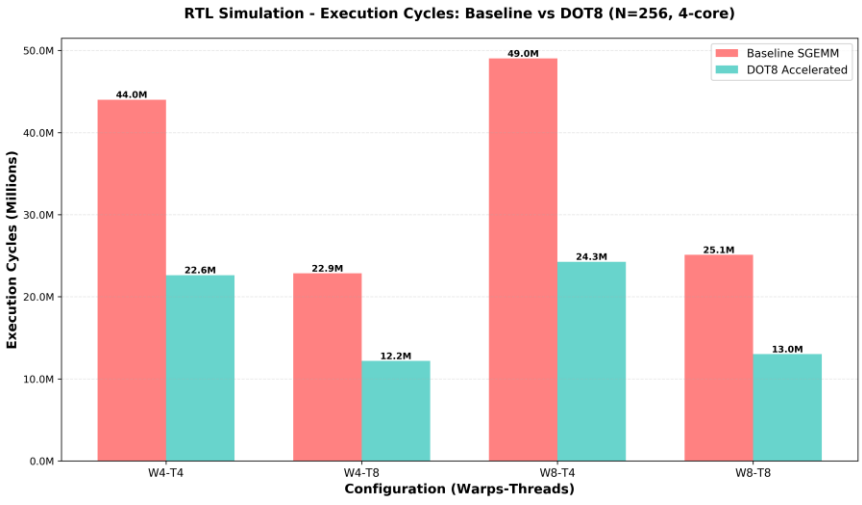
Detailed Analysis

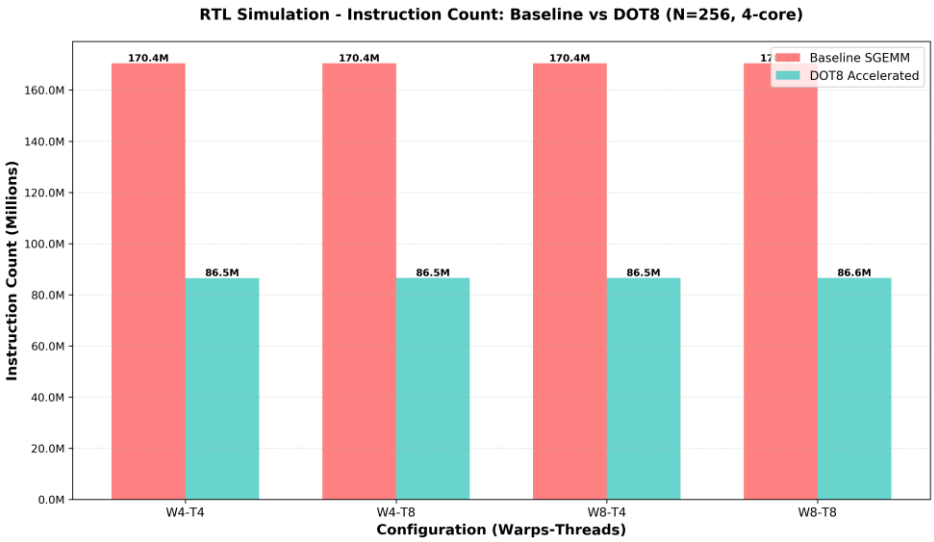
For instruction counts, across all four configurations, Dot8 consistently cuts total cycles by about 1.86–1.96× versus the INT8 sgemm baseline. The other visible trend is that T8 halves the cycles vs T4 (for a fixed W) for both kernels, showing better latency hiding and issue utilization with more threads per warp. In contrast, increasing W from 4→8 barely changes cycles at fixed T, which means four warps already provide enough independent work

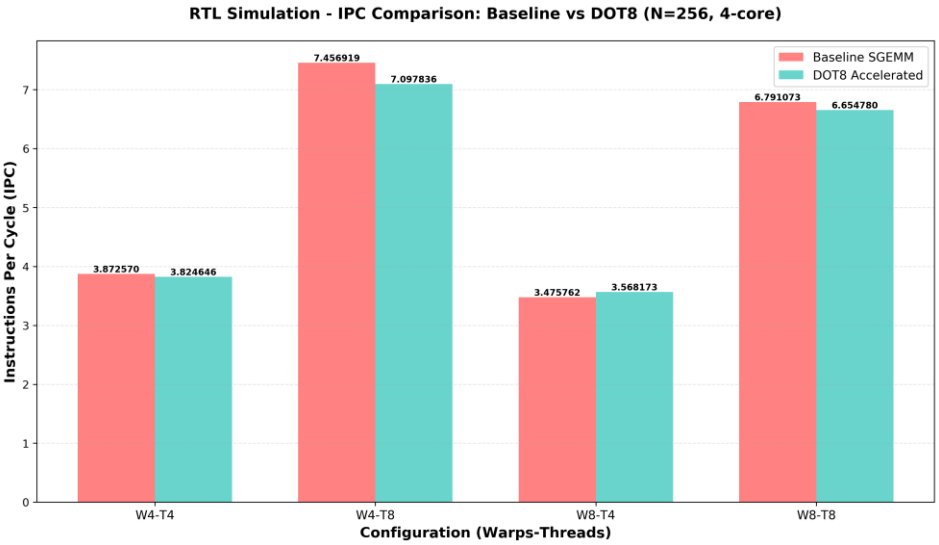
Execution cycles are almost halved by Dot8. That is reasonable since Dot8 performs a small dot-product on four INT8 pairs and accumulates into INT32 in a single op, the arithmetic instruction count drops. The near constancy of instruction counts across (W,T) for a given kernel is also expected. Changing parallelism alters how work is scheduled, not how many instructions each thread executes for a fixed problem size (N=256).

IPC rises strongly from T4 to T8, confirming that more threads per warp keep the machine busier. IPC changes little when we increase the number of warps, meaning that the four warps already provide sufficient work. Notably, Dot8's IPC is slightly lower than baseline. Dot8 drastically reduces arithmetic instructions, but the memory/loop/control fraction doesn't shrink, and at T8 the cores are already near their issue/bandwidth ceiling. Since ~2× fewer instructions and ~1.9× cycle-level speedup, the IPC is slightly lower. Generally speaking, W4-T8 combination achieves the best performance and maintains the best tradeoff.

Results plots and analysis of RTL







Workload

- Matrix Size: 256×256
- Input Type: int8_t (8-bit signed integers)
- Output Type: int32_t (32-bit signed integers)
- Test Configurations:
  - 4 warps × 4 threads = 16 threads
  - 4 warps × 8 threads = 32 threads
  - 8 warps × 4 threads = 32 threads
  - 8 warps × 8 threads = 64 threads

Detailed Analysis

Assignment 6 is the RTL counterpart to Assignment 5. We run the same GEMM-like workload through the full RTL pipeline, not just the cycle-level SimX model. As for the instruction counts, instructions are approximately halved by DOT8 and largely invariant to (W,T), as expected because (W,T) change scheduling rather than algorithmic work per thread. Total work is fixed by the problem size N=256, while DOT8 packs more computation per instruction.

In terms of execution cycles, DOT8 consistently reduces runtime by ~1.9–2.0× across all (W,T) settings. When we increase increasing threads per warp from 4 to 8, the cycles are nearly nearly halved. Besides, when we increase warps from 4 to 8 at fixed T, the cycles are slightly increased. Two forces are at play. First, DOT8 shrinks the dynamic instruction stream, so fewer cycles are needed. Second, as you we thread per warp (T) or warps (W), the baseline already achieves higher utilization via SIMT lane parallelism and latency hiding. The relative win of DOT8 diminishes because the machine is closer to saturation.

As for IPC, DOT8 is mostly lower than sgemm baseline, when both of them use int8_t input and int32_t output. When we increase T, there is an obvious improvement, showing wider per-warp activity improves issue utilization. Meanwhile, increasing W did not provide a good improvement, which is consistent with extra warps contending for LSU/issue slots/MSHRs. Slight DOT8 IPC dips are reasonable. After compressing the compute stream, the unchanged parts (loads/stores, pointer updates, loops) dominate, and specialized FU throughput/latency or scheduling may cap IPC.

Summary

In summary, Dot8 yields the observed ~1.9–2.0× cycle speedups, with nearly 2× instruction reduction. More threads per warp improve SIMD efficiency, while extra warps mostly add contention. The best performance for N=256 matrix multiplication occurs at W4-T8 configuration.