CS 259 Lab 3

Implementation Summary -

DOT8 Instruction Footprint:

func7	rs2	rs1	func3	rd	opcode
1	<address>	<address>	0	<address>	0x0B

New Instruction Definition -

> vx_intrinsics.h

- Defined the dot8 instruction encoding using custom instruction space
- Created a new vx_instrinsic function vx_dot8 that invokes the new instruction with three operands - the input registers rs1, rs2, and the destination register rd.

SimX Implementation -

> decode.cpp

- A new case was added to recognise the case when funct7 is 1 and funct3 is 0 for custom instruction opcode, and set the right registers to operate on.

> execute.cpp

- New logic was added to compute the dot product of the two operands as per the spec.
- The four int8_t register values from each of the operands is unpacked and then the int32_t dot product is computed.

```
+            int8_t A1 = rsdata[t][0].u & 0xFF;
+            int8_t A2 = (rsdata[t][0].u >> 8) & 0xFF;
+            int8_t A3 = (rsdata[t][0].u >> 16) & 0xFF;
+            int8_t A4 = (rsdata[t][0].u >> 24) & 0xFF;
+
+            int8_t B1 = rsdata[t][1].u & 0xFF;
+            int8_t B2 = (rsdata[t][1].u >> 8) & 0xFF;
+            int8_t B3 = (rsdata[t][1].u >> 16) & 0xFF;
+            int8_t B4 = (rsdata[t][1].u >> 24) & 0xFF;
+
+            int32_t dot_product = (A1*B1) + (A2*B2) + (A3*B3) + (A4*B4);
```

> func_unit.cpp

- A new AluType DOT8 is added, which emulates the latency of the operation since the computation logic is implemented in the execute itself.

RTLSim Implementation -

> VX_decode.sv

- A new case was added to recognise the case when funct7 is 1 and funct3 is 0 for custom instruction opcode, and set the right registers to operate on and the flags.
    - alu_type = ALU_TYPE_DOT8 (new ALU defined)
    - use_PC = false (PC is not being used in any computation)
    - use_imm = false (there is no immediate operand)
    - use_rd = 1 (destination register rd being used)
    - use_rs1 = 1 (source register rs1 being used)
    - use_rs2 = 1 (source register rs2 being used)

> VX_alu_unit.sv

- The new alu unit for dot8 computation is instantiated in the alu_unit for it to be used when the new dot8 instruction is encountered.

> VX_alu_dot8.sv

- The skeleton code provided for the alu dot8 is used with the computation logic filled in.
- The 4 individual 8-bit values are extracted from the bit masks.

```
+    wire signed [7:0] A1 = a[7:0];
+    wire signed [7:0] A2 = a[15:8];
+    wire signed [7:0] A3 = a[23:16];
+    wire signed [7:0] A4 = a[31:24];
+
+    wire signed [7:0] B1 = b[7:0];
+    wire signed [7:0] B2 = b[15:8];
+    wire signed [7:0] B3 = b[23:16];
+    wire signed [7:0] B4 = b[31:24];
+
+    assign c = (A1 * B1) + (A2 * B2) + (A3 * B3) + (A4 * B4);
```

CPU Implementation -

> main.cpp

- Similar to simX implementation, the code for matmul_cpu unpacks the operands into four 8-bit integers and then computes the dot product.

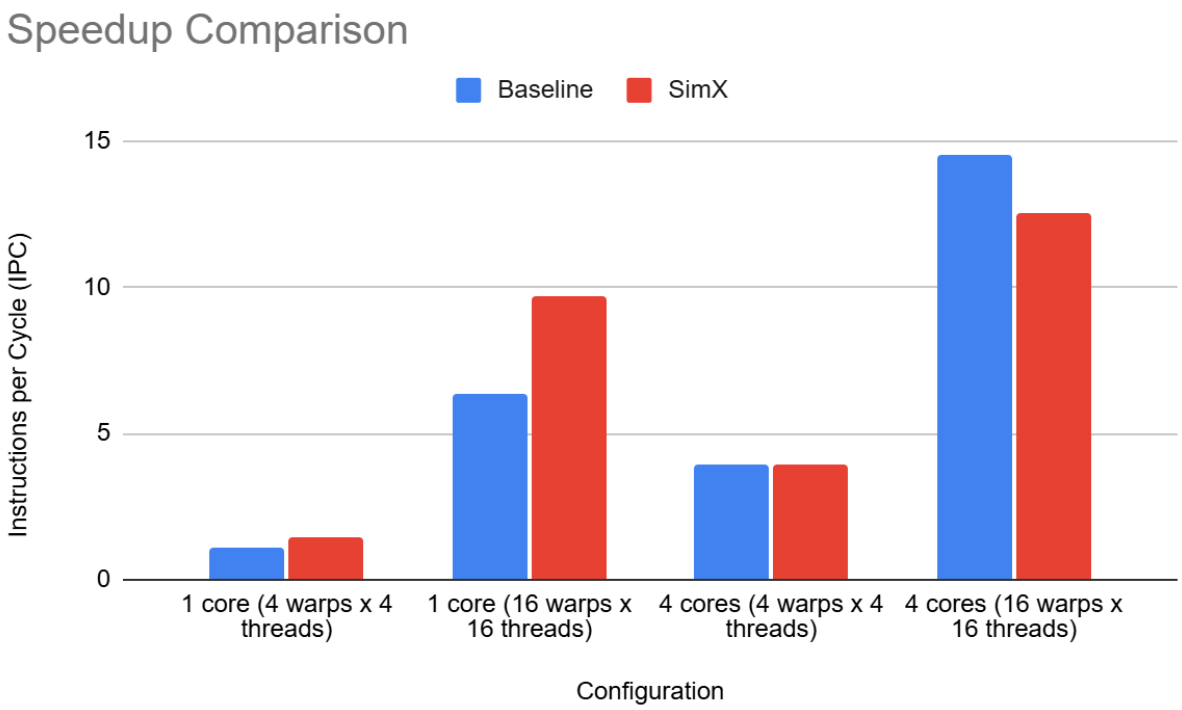
> kernel.cpp

- Since the GPU hardware works on packed values, the kernel packs values into 32-bit operands before spawning threads to work on those packed values.

Instructions per Cycle -

RTLSim is not working properly, even the demo code. (Campuswire #93). Will share the updated results when that is resolved.

Cores	Warps x Threads	Baseline	SimX
1	4 x 4	1.087411	1.438041
1	16 x 16	6.377837	9.724751
4	4 x 4	3.971515	3.970534
4	16 x 16	14.496276	12.526804



- For a single core, the GPU implementation is efficiently leveraging the hardware dot-product accelerator, leading to fewer instructions executed by the core (and fewer cycles) compared to the baseline, which relies entirely on general-purpose ALU operations.

- In multi-core configurations, contention for shared resources like memory bandwidth becomes a limiting factor. The GPU implementation, while more computationally efficient, still needs to load data into registers before performing the accelerated operations. Memory bandwidth becomes the bottleneck, negating the performance gains from the accelerator.