CS259 - LAB3

The objective of this lab is to introduce you to the basics of extending GPU microarchitecture to accelerate a kernel in hardware
.
We will be using the Vortex OpenGPU (https://github.com/vortexgpgpu/vortex) for this experiment.

You will be using the latest code from branch**: develop**

Part1 - ISA Extension (1 points)

You will add a new RISC-V custom instruction for computing the integer dot product; **VX_DOT8.**

VX_DOT8 calculates the dot product of two small vectors of int8 integers.
Each source register (rs1 and rs2) holds four int8 elements. The operation proceeds as follows:

Extract four int8 values from each source register:
rs1 holds A1, A2, A3, A4
rs2 holds B1, B2, B3, B4
Perform the dot product operation:
Dot Product = (A1*B1 + A2*B2 + A3*B3 + A4*B4)
The result, which could potentially be a 32-bit integer due to overflow, is stored in the destination register (rd).

Use the R-Type RISC-V instruction format.

| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 bits | 5 bit| 5 bit| 3 bits |5 bit| 7 bits |

- opcode: Specific opcode reserved for custom instructions.
- rs1, rs2: Source registers containing vectors of four `int8` values each.
- rd: Destination register.
- funct3 and funct7: Used for further specifying the operation (like saturation and accumulation options).

Use custom extension opcode=0x0B with func7=1 and func3=0;
Your new instruction will be implemented as a new operation for the ALU unit.

You will need to modify vx_instrinsics.h to define your new VX_DOTP8 instruction
Read the following doc to understand **insn** speudo instruction format
https://sourceware.org/binutils/docs/as/RISC_002dV_002dFormats.html

```
// DOT8
inline int vx_dot8(int a, int b) {
    size_t ret;
    asm volatile (".insn r ?, ?, ?, ?, ?, ?" : "=r"(?) : "i"(?), "r"(?),
"r"(?));
    return ret;
}
```

We recommend checking out how VX_SPLIT and VX_PRED instructions are decoded in SimX as reference.

Part2 - Sample matrix multiplication (2 points)

Implement a simple matrix multiplication GPU code that uses your new H/W extension.
Here is a basic CPU implementation example:

```
void matrixMultiply(int8_t A[][N], int8_t B[][N], int32_t C[][N], int N) {
   int i, j, k;
   int packedA, packedB;
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         C[i][j] = 0;
         for (k = 0; k < N; k += 4) {
            // Pack 4 int8_t elements from A and B into 32-bit integers
            packedA = *((int*)(A[i] + k));
            packedB = *(int*)(B[k] + j)
                        | (*(int*)(B[k+1] + j) << 8)
                        | (*(int*)(B[k+2] + j) << 16)
                        |  (*(int*)(B[k+3] + j) << 24);
            // Accumulate the dot product result into the C matrix
            C[i][j] += vx_dot8(packedA, packedB);
         }
      }
   }
}
```

- Clone sgemmx test under tests/regression/sgemmx into a new folder tests/regressions/dot8.
- Set PROJECT name to dot8 in tests/regressions/dot8/Makefile
- Update matmul_cpu in main.cpp to operate on int8_t matrices.
- Update kernel_body in tests/regressions/dot8/kernel.cpp to use vx_dot8

Part3 - SimX implementation (3 points)

Modify the cycle level simulator to implement the custom ISA extension.
We recommend checking out how VX_SPLIT and VX_PRED instructions are decoded in SimX as reference.

Update op_string() in decode.cpp to print out the new instruction.
Update Emulator::decode() in decode.cpp to decode the new instruction format.

```
switch (func7) {
case 1:
  switch (func3) {
  case 0:  // DOT8
    instr->setDestReg(rd, RegType::Integer);
    instr->addSrcReg(rs1, RegType::Integer);
    instr->addSrcReg(rs2, RegType::Integer);
    Break;
```

Update AluType enum in types.h to add DOT8 type
Update Emulator::execute() in execute.cpp to implement the actual VX_DOT8 emulation.
You will execute the new instruction on the ALU functional unit.

```
switch (func7) {
case 1:
  switch (func3) {
  case 0: { // DOT8
      trace->fu_type = FUType::ALU;
      trace->alu_type = AluType::DOT8;
      trace->used_iregs.set(rsrc0);
      trace->used_iregs.set(rsrc1);
      for (uint32_t t = thread_start; t < num_threads; ++t) {
        if (!warp.tmask.test(t))
          continue;
        // TODO:
      }
      rd_write = true;
    } break;
  } break;
}
```

Update AluUnit::tick() in func_unit.cpp to implement the timing of VX_DOT8.
You will assume 2 cycles latency for the dot-product execution.

```
case AluType::DOT8:
 // TODO:
 break;
```

Part4 - RTL implementation (4 points)

Modify the RTL code to implement the custom ISA extension.
You will need to modify VX_decode.sv, VX_alu_unit.sv primarily.
You will add a new module called VX_alu_dot8 and add it VX_alu_unit.sv similar to how we added aVX_alu_muldiv.
You will assume 2 cycles latency for the operation.
We recommend checking out how MULT instructions are decoded in RTL as reference.

Modify the RTL code to implement the custom ISA extension. We recommend checking out how MULT instructions are decoded and executed in RTL as reference.

Update hw/rtl/VX_define.vh to define INST_ALU_DOT8 as 4'b0001
Update hw/rtl/VX_config.vh to define LATENCY_DOT8 as 2
Update dpi_trace() in hw/rtl/VX_gpu_pkg.sv to print the new instruction
Update hw/rtl/core/VX_decode.sv to decode the new instruction.
Select the ALU functional unit for executing this new instruction.

```
7'h01: begin
   case (func3)
      3'h0: begin // DOT8
         ex_type = // TODO: destination functional unit
         op_type = // TODO: instruction type
         use_rd =  // TODO: writing back to rd
         // TODO: set using rd
         // TODO: set using rs1
         // TODO: set using rs2
      end
      default:;
   endcase
End
```

Create a new VX_alu_dot8.sv module that implements DOT8

```
`include "VX_define.vh"

module VX_alu_dot8 #(
   parameter `STRING INSTANCE_ID = "",
   parameter NUM_LANES = 1
) (
   input wire        clk,
   input wire        reset,

   // Inputs
   VX_execute_if.slave execute_if,

   // Outputs
   VX_commit_if.master commit_if
);
   localparam PID_BITS = `CLOG2(`NUM_THREADS / NUM_LANES);
   localparam PID_WIDTH = `UP(PID_BITS);
   localparam TAG_WIDTH = `UUID_WIDTH + `NW_WIDTH + NUM_LANES + `XLEN +
`NR_BITS + 1 + PID_WIDTH + 1 + 1;
   localparam LATENCY_DOT8 = `LATENCY_DOT8;
   localparam PE_RATIO = 2;
   localparam NUM_PES = `UP(NUM_LANES / PE_RATIO);

   `UNUSED_VAR (execute_if.data.op_type)
   `UNUSED_VAR (execute_if.data.op_mod)
   `UNUSED_VAR (execute_if.data.use_PC)
   `UNUSED_VAR (execute_if.data.use_imm)
   `UNUSED_VAR (execute_if.data.tid)
   `UNUSED_VAR (execute_if.data.rs3_data)

   wire [NUM_LANES-1:0][2*`XLEN-1:0] data_in;

   for (genvar i = 0; i < NUM_LANES; ++i) begin
      assign data_in[i][0 +: `XLEN] = execute_if.data.rs1_data[i];
      assign data_in[i][`XLEN +: `XLEN] = execute_if.data.rs2_data[i];
   end

   wire pe_enable;
   wire [NUM_PES-1:0][2*`XLEN-1:0] pe_data_in;
   wire [NUM_PES-1:0][`XLEN-1:0] pe_data_out;

   // PEs time-multiplexing
```

```
VX_pe_serializer #(
   .NUM_LANES  (NUM_LANES),
   .NUM_PES    (NUM_PES),
   .LATENCY    (LATENCY_DOT8),
   .DATA_IN_WIDTH (2*`XLEN),
   .DATA_OUT_WIDTH (`XLEN),
   .TAG_WIDTH  (TAG_WIDTH),
   .PE_REG     (1)
) pe_serializer (
   .clk        (clk),
   .reset      (reset),
   .valid_in   (execute_if.valid),
   .data_in    (data_in),
   .tag_in     ({
      execute_if.data.uuid,
      execute_if.data.wid,
      execute_if.data.tmask,
      execute_if.data.PC,
      execute_if.data.rd,
      execute_if.data.wb,
      execute_if.data.pid,
      execute_if.data.sop,
      execute_if.data.eop
   }),
   .ready_in   (execute_if.ready),
   .pe_enable  (pe_enable),
   .pe_data_in (pe_data_in),
   .pe_data_out(pe_data_out),
   .valid_out  (commit_if.valid),
   .data_out   (commit_if.data.data),
   .tag_out    ({
      commit_if.data.uuid,
      commit_if.data.wid,
      commit_if.data.tmask,
      commit_if.data.PC,
      commit_if.data.rd,
      commit_if.data.wb,
      commit_if.data.pid,
      commit_if.data.sop,
      commit_if.data.eop
   }),
   .ready_out  (commit_if.ready)
);
```

```
    // PEs instancing
    for (genvar i = 0; i < NUM_PES; ++i) begin
        wire [31:0] a = pe_data_in[i][0 +: 32];
        wire [31:0] b = pe_data_in[i][32 +: 32];
        // TODO:
        wire [31:0] result;
        `BUFFER_EX(result, c, pe_enable, LATENCY_DOT8);
        assign pe_data_out[i] = result;
    end

Endmodule
```

Update hw/rtl/core/VX_alu_unit.sv to add your new VX_alu_dot8 instance as a 3rd sub-unit after VX_alu_muldiv.

Report:

You will compare your new accelerated sgemm program with the existing sgemm kernel under the regression codebase. You will use N=128 and (warps=4, threads=4) and (Warps=16, threads=16) for 1 and 4 cores on the SimX and RTLSIM driver. Plot the IPC,

Bonus (2 points):
You will get a bonus if you write an optimized kernel that leverages the GPU local memory to tile the computation on 16x16 elements (256 bytes tiles).

What to submit?
1- zip file containing report.pdf and changes.diff.
The report should be about 2 pages showing the generated plots.