On-chip Unsupervised Spike Sorting Accelerator

Zhewei Jiang (zj2139)
Jamis Johnson (jmj2180)

1. MOTIVATION

Current state of prosthetic brain-computer-interface (BCI) operate via decoding spike rate (per neuron) of target brain region. Typical implants used for upper limb movement have ~70 electrodes. The implanted electrodes are extracellular and records activities in noisy environment, with each channel receiving spikes from 1~6 neurons. Spike sorting is a necessary stage between spike detection and prosthetic movement decoding. Existing researches currently employ on-chip communication between detected spike and non-invasive portion of the BCI where spike sorting and decoding occur.
The power cost of communication dominates the front-end of the BCI, can be greatly reduced if spike sorting is performed on-chip from roughly (32~64) (8~12bit) samples/spike to 3~4 bits/spike. Throughput upper-bound of the communication channel can be significantly increased. We propose a new algorithmic flow to allow low cost hardware implementation of unsupervised on-chip spike sorting.

2. ALGORITHM

Online unsupervised clustering algorithms require preliminary knowledge of the feature, whether it's parameter based (e.g. number of clusters for K-means, density cutoff for DBSCAN) or feature property (e.g. noise variance for sequential leader, "temperature" for paramagnetic clustering). The flow of the algorithm begins with computing the feature distribution. Here we use the spike peak and hyperpolarization voltages as feature.

a. Pre-training Feature Distribution Estimation

Memory requirement of the 2D features distribution is dependent on desired distribution resolution which allows us to use lower resolution to mimic smoothing operation in Kernel Density Estimation technique. The memory requirement is then only the square root of the full 2D feature space. In this case, the features do not overlap when mapped to the same axis, otherwise (2n instead of n^2 of the memory is required).

The typical kernel is Gaussian as shown below, in which the variance value is the smoothing parameter of the kernel, not necessarily associated with the features' modal variance.

$$kernel(x) = \frac{1}{\left(\sigma\sqrt{2\pi}\right)^{d}} * e^{\left(-\frac{x^2}{2\sigma^2}\right)}$$

The trivial case of KDE is histogram, by selecting bin width, the distribution estimation can be computed with the following kernel, which allows single update per spike as well as lower memory requirement.

$$Kernel(x,w) = \begin{cases} 1, & w * \left\lfloor \frac{x}{w} \right\rfloor < x < w * \left\lceil \frac{x}{w} \right\rceil \\ 0, & overwise \end{cases}$$

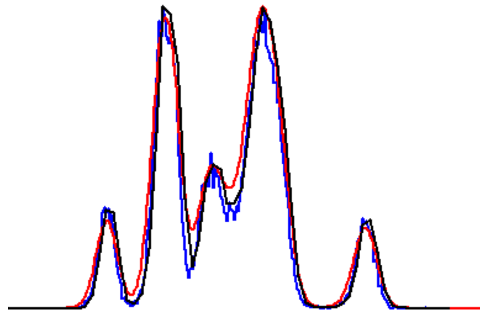

Figure 1: (a) blue line is the actual distribution of the features; (b) red line is the KDE result using Gaussian kernel at a variance of 2^{-6} of feature range; (c) black line is the trivial case of KDE result (histogram), with the same smoothing parameter as the Gaussian case.

b. Laplacian operation

The algorithm then performs convolution on the estimated distribution with simple Laplace operator. This step can be integrated with the previous step, using Kernel[-1,2,-1] from before. The training criterion is based on the second derivative of the distribution trends, such that even overlapping modes can be differentiated as long as the mode peak is not completely masked.

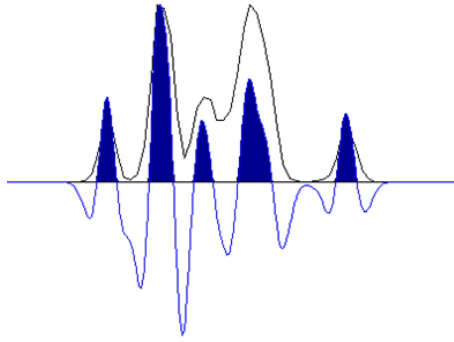

Figure 2: Distribution after Laplacian, with shaded region representing informative sample region, and the lowest point between modes indicating feature boundary

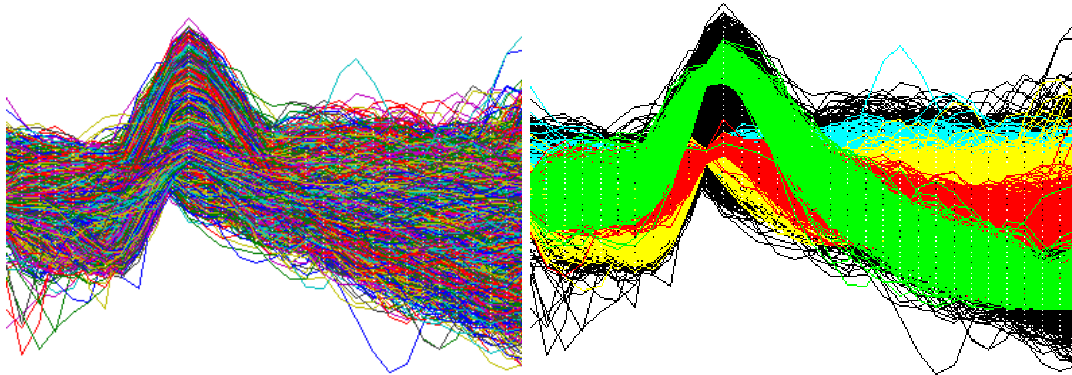

Figure 3: (a) left figure: input spikes; (b) right figure: colored spikes are in each informative region, and black spikes are not used for training

c. Training: Cluster Update Policy

Each spike during training stage is checked against the informative sample vector mask, only when both peak and hyperpolarization voltage falls within informative region can it be used for training. Eligible spikes is then represented by 2 indexes indicating the particular mode it belongs in.
Due to the fact that the two features occupy 1D distribution curve, possible false positives can occur when an outlier spike has the peak in the informative region of one cluster, while the depolarization voltage in the informative region of another cluster.
The policy of updating clusters is: for each informative spike, its feature index pair is compared against all existing clusters, if it exists, nothing is done, else, it is added as a new cluster. The following bimodal state machine is used to track

each cluster, clusters with no match in a set amount of time is downgraded and eventually vacated.

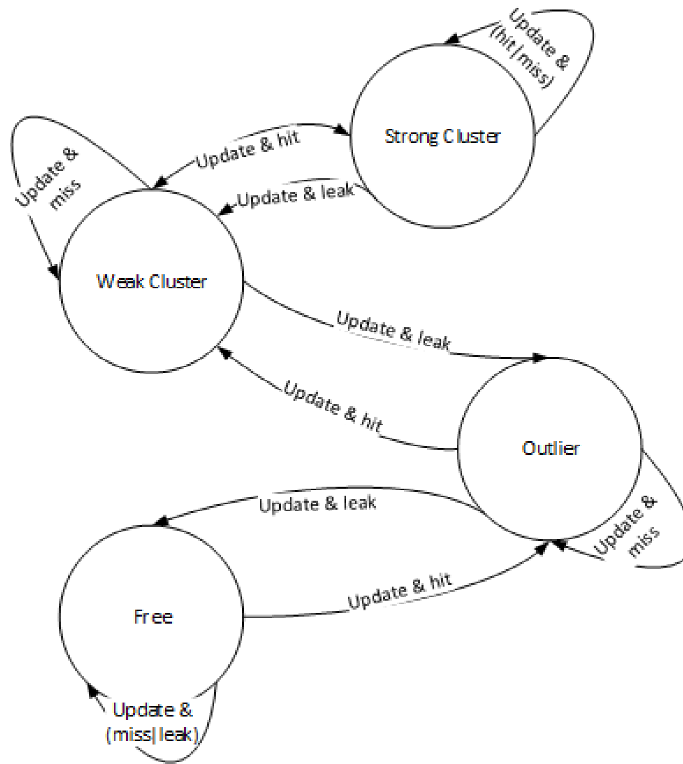

Figure 4: bimodal state machine for cluster usefulness

d. Sorting

Sorting is similar to training in that each spike's feature indexes is compared against all learnt clusters. It is designated to the matching or the closest cluster (adjacent grid in feature space). No update is needed.

3. DESIGN

The accelerator interfaces with bus such that the address (during master write) encodes commands, and the data contain the feature value (16 bits total, 12 of which are used), it is used as address for memory address and used as values for comparison for grid location.
Master read can access sorting output and other for-test values.
The software portion of the design acts as testbench and scheduling.

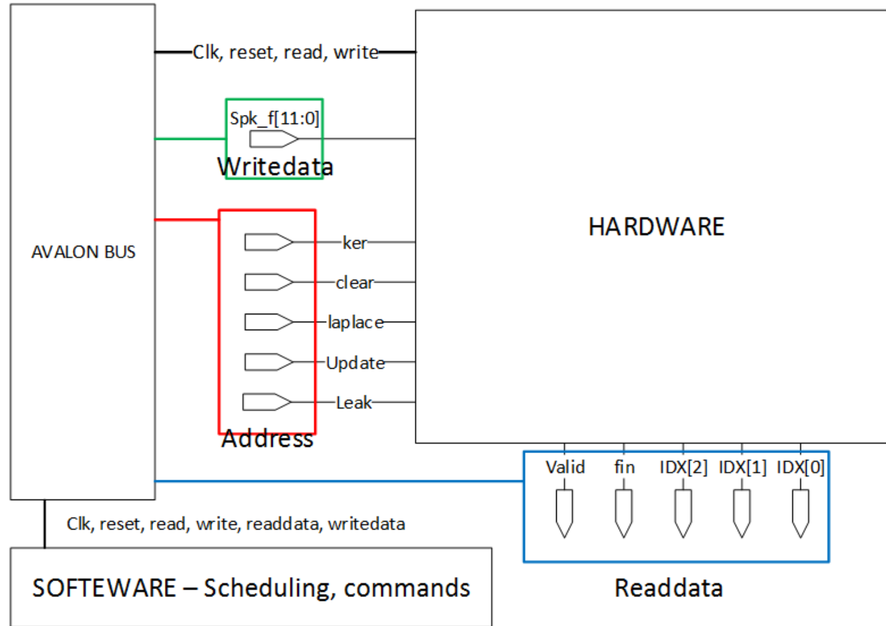

Figure 5: system level design

4. HARDWARE

Top-level

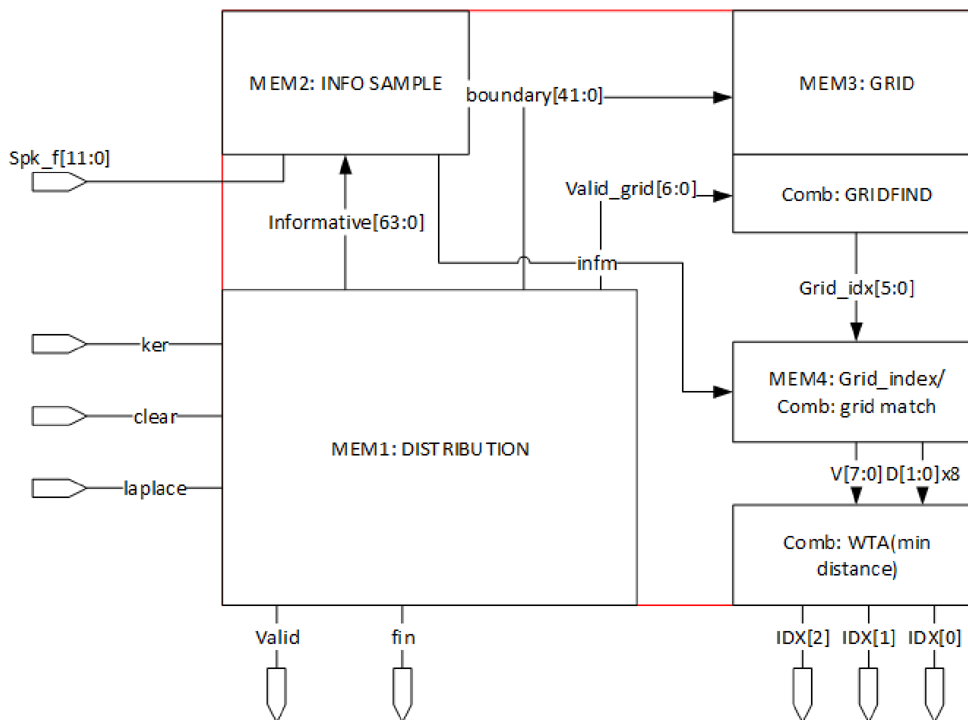

Figure 6: top-level hardware block diagram

The hardware design is derived from a research project targeting subthreshold operating ASIC with power gating support, several design decisions in this design are legacy from other constraints, such as FSM edge trigger scheme for set-up and hold.

The memory modules in the design contains

- 64x16b RAM for storing feature distribution
- 64b shift registers for storing informative feature vector mask
- 7x6b shift registers for storing boundary values
- 8x6b CAM for storing cluster indexes

Main combinational logic computes the following

- Grid location computation (x2): comparators (x7) and adder
- Index match & adjacency test (x8): custom logic, hardware cost close to adder (x2)
- Closest cluster computation (x7 tree structure 4 2 1): comparator and muxes

Sequential logic

- Memory module FSM for updating histogram, overflow handling
- Laplace operation FSM for computing informative sample and boundary locations
- Usefulness update, typical bimodal state machine

a. Distribution Memory

i. Top level [DISTR]

Main memory module shown below in figure 7 is for storing distribution estimation. Memory access has two modes: access through feature (used as address, writedata is readdata + 1) and traversal access (CADDR in figure, controlled by FSM) when handling overflow (right-shift all stored values) and laplacian.

Informative sample memory is implemented as shift registers, read port are 64 to 1 muxes, asynchronous access.
The Laplacian block in figure 8 computes (F(i)<<1) > (F(i-1) + F(i + 1) + offset) to determine informative sample region, based on distribution curvature.

Figure 7: Module DISTR top-level diagram, with informative sample vector mask and laplacian block (no FSM shown)

ii. Laplacian Module

Figure 8: Laplacian block from figure 7

b. Module GC (Grid_Cluster)

i. Top level [GC]

The top level GC module is shown below. FIS and SEGS are memory modules keeping informative sample vector mask and boundary information, which are parts of DISTR, shown here for completeness.

GRIDFIND is the module that finds the features' location in feature space.
ATTR computes the proximity to each grid considered a cluster.
WTA finds the cluster with minimum distance to feature indexes.
CAM_CLUSTER contains the cluster information (indexes of modes) and usefulness state machines tracking the validity of each cluster.

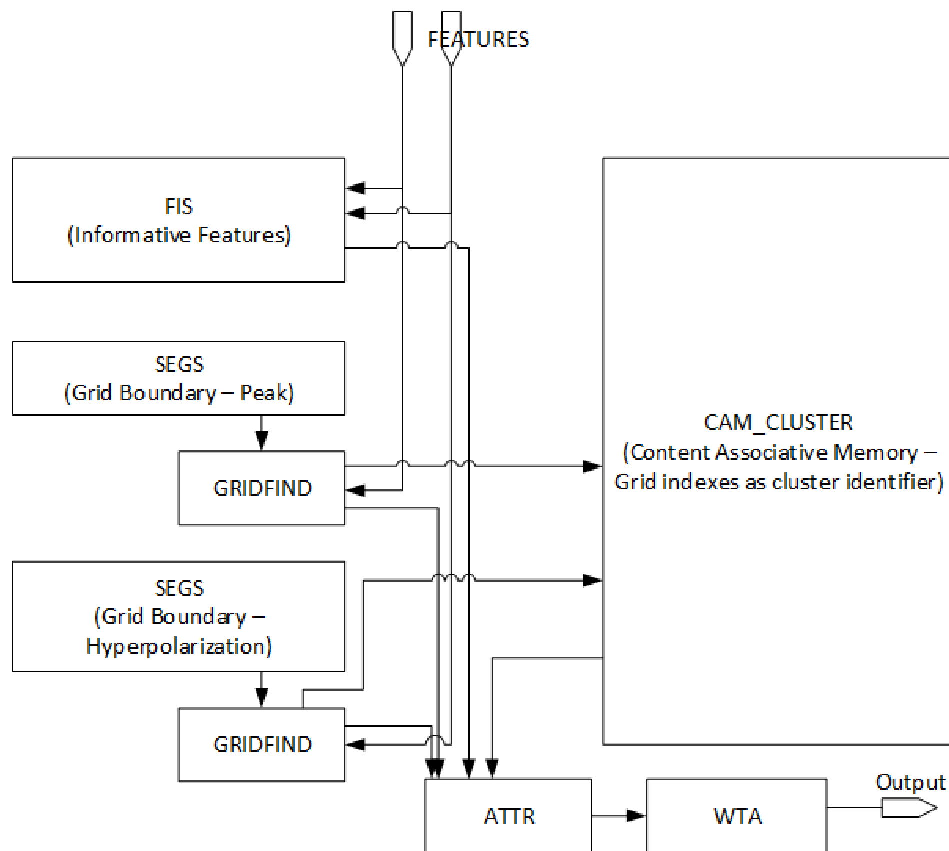

Figure 9: Top level block diagram for GC which include modules for computing features' indexes in feature space

The GC module requires no registers to function, a legacy attribute from the ASIC version of the design. Registers are inserted between modules to reduce setup time risk.

ii. Boundary Registers

The boundary registers are controlled by laplace operation FSM inside of DISTR module, the FSM receives informative sample during traversal of distribution memory as input to proper update boundary informative. Valid boundary bits accompany the actual boundary value for accurate GRIDFIND operation.

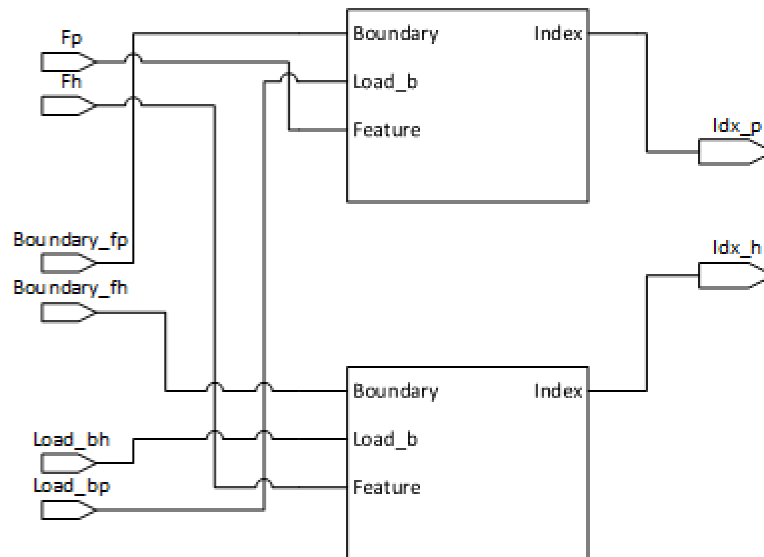

Figure 10: Top level boundary modules

Figure 10 shows that each feature has its own boundary computation block each though both share the same distribution: Boundary_fp and Boundary_fh are the same wires in the implementation.

Figure 11 shows the boundary computation block (showing 4 entries, 8 are implemented in design). The memory element is shift registers inside of DISTR module, the computation is CAM style, the feature is broadcast to all entries for comparison, the same of the comparator outputs shows the index of boundaries it is immediately after.

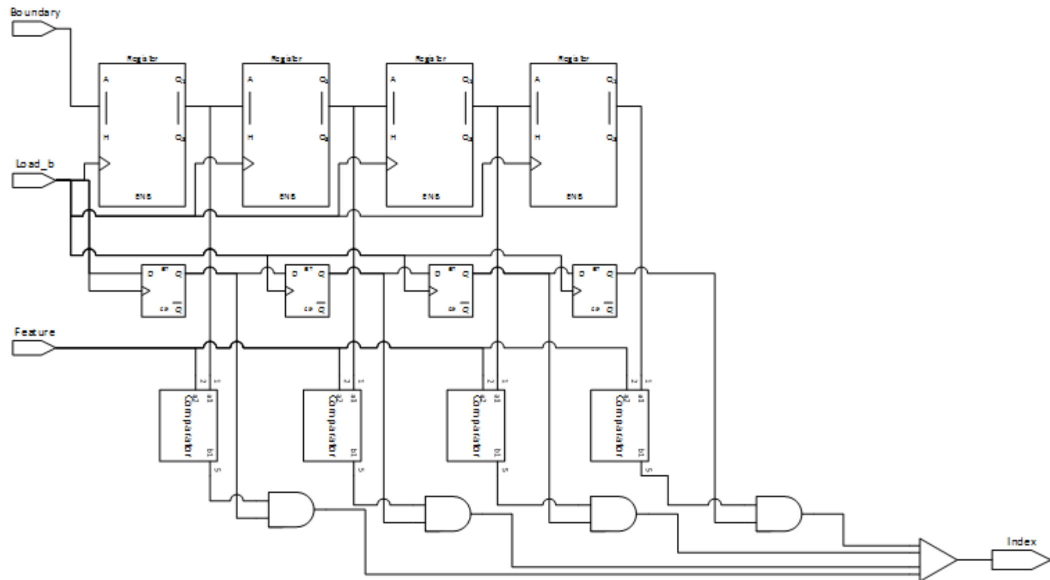

Figure 11: shift register based boundary memory with comparators and summation for index computation

iii. Cluster CAM Entry

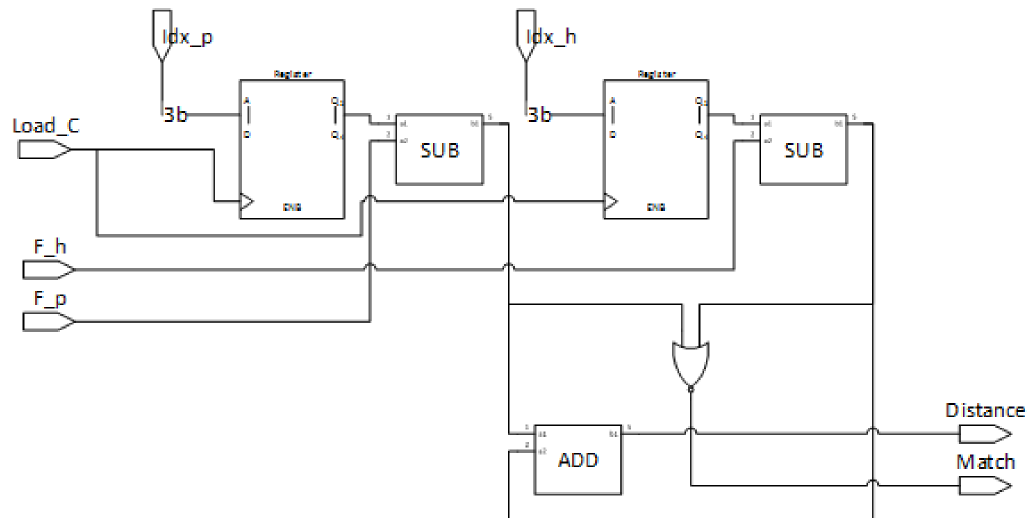

Figure 12: single entry in the cluster CAM

The single entry is compared against the feature index and potentially be overwritten by the indexes based on usefulness tracker and vacancy tracker.

The ADD and SUB block in figure 12 are modified in the implementation to output valid proximity, as in any feature index non-adjacent to the particular cluster grid is ignored, reducing hardware cost of the CAM.

c. Module WTA (Winner-Take-All)

i. WTA Tree

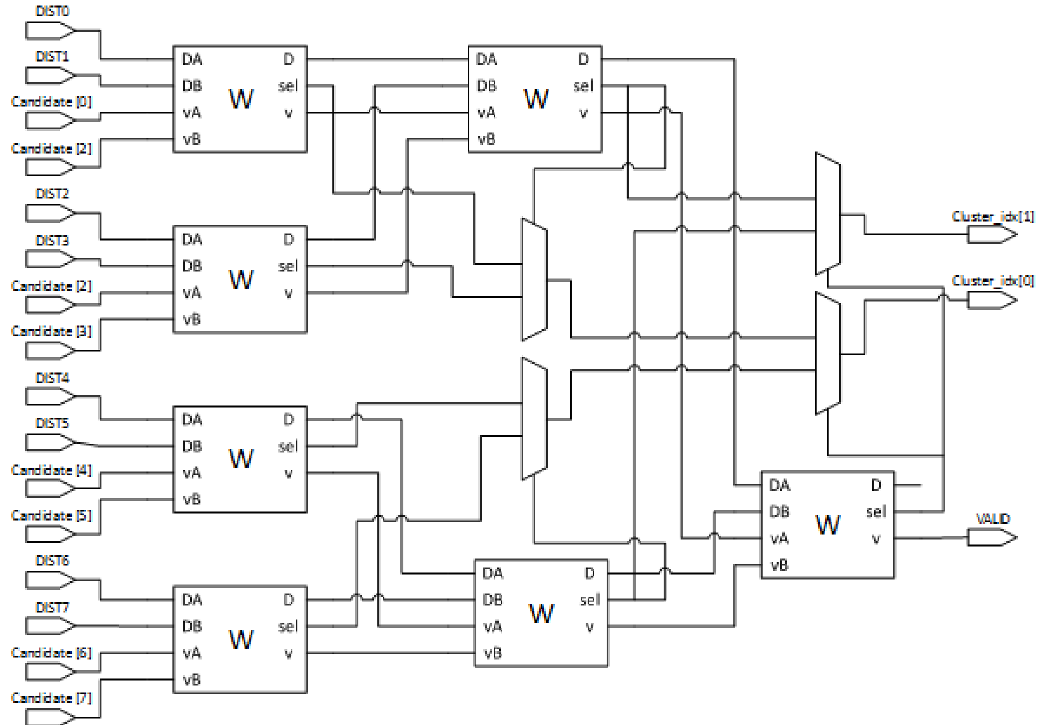

Figure 13: Comparator tree for least grid granularity distance

ii. W module

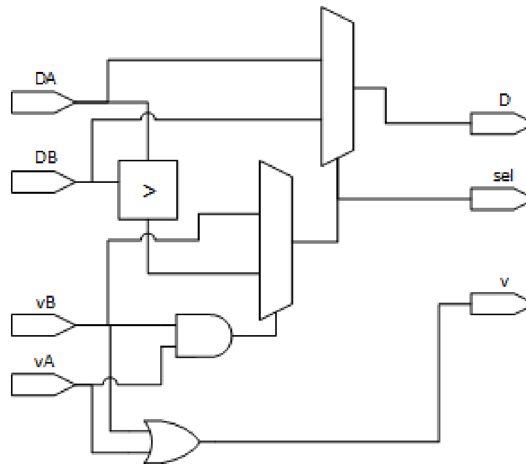

Figure 14: single comparator with validity vector mask

This is the final stage of the accelerator, the output is not registered explicitly, but is kept linked at the top level to an registered output. The

output is stable one cycle before valid bit, and retain the value until new spike is received.

5. TIMING DESIGN

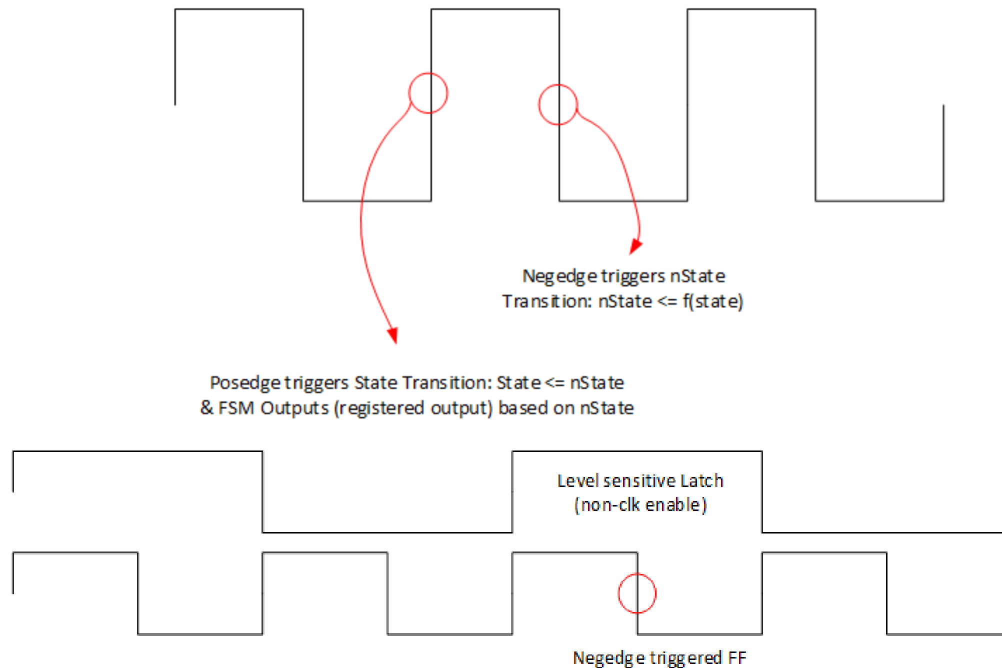

Figure 15: (a) top figure shows the FSM timing scheme; (b) bottom figure shows the legacy timing scheme for ASIC usefulness tracker and the corresponding negedge trigger implementation on FPGA

The timing design is as follows:

Command
Ker:
4 cycle update -- 2 read/write pairs in normal case
in case of memory overflow, FSM traverse memory with 64 read/write pairs, right shifting all data (this gives recent spikes more weight, which is desirable even though the distribution estimation isn't precise)
At the end of the operation, FIN is flagged and only deasserted after master_read.

Laplace:
Same as the overflow case of Ker, but instead of 2 read/write pair, each entry undergoes a single read, the buffer registers facilitate Laplace operation and control inputs to secondary Laplace FSM.

At the end of the operation, the same FIN flag is asserted and require a read from master to enter training stage.

Update/Leak:
These two commands incur no control flow, as all computation prior to actual cluster update are combinational (with inserted flip-flops). The commands are given 3 cycle delay to match the valid values of feature index. No flag is used, but valid bit is always observed at output.

6. DEBUG

a. ModelSim verification

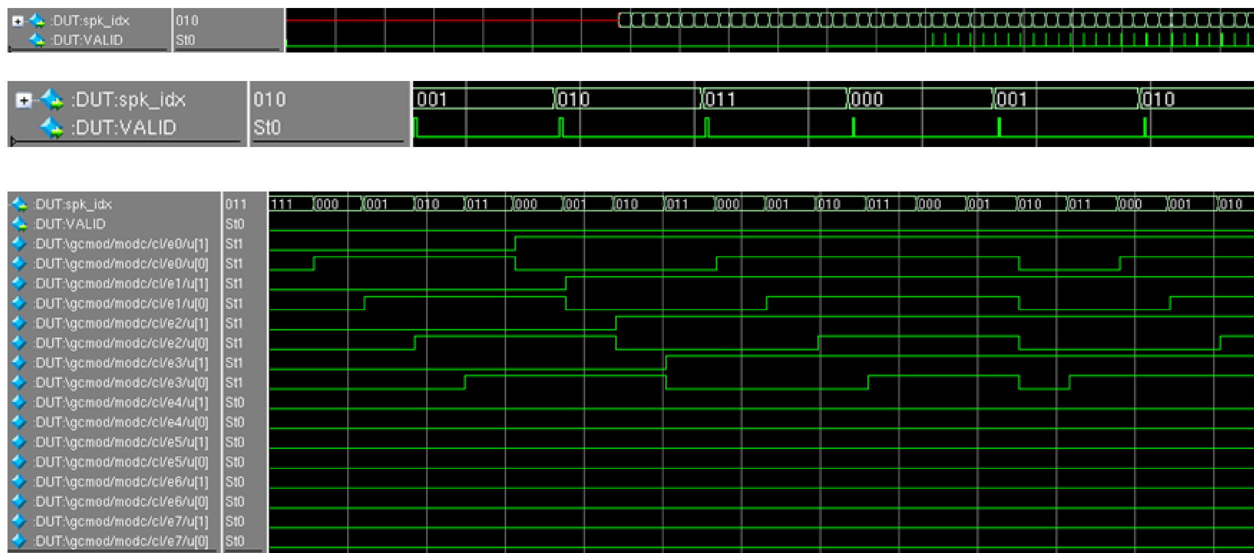

Figure 16: ModelSim verification (a) top graph: valid bit and index bits output through all stages, indeterminate index during KDE and Laplacian, correct index through training stage with mute valid bit, and final sorting stage with correct index and valid flag; (b) middle graph: zoomed in view of sorting stage of looping inputs; (c) usefulness state machine during training, increments when cluster match and decrement at leakage.

b. System Console verification

The design-for-test is at the top-level in which intermediate results from the modules can be placed in the read register where the software can access. Currently, the occupancy of the CAM and count of boundaries can be muxed into outputs based on read command.

Due to time constraint, debugging is not yet finished. We currently experience filled boundary and empty CAM. While this points to error before GC module, DISTR has correct FSM behavior with flagging FIN bit.

DISTR FSM resets to idle only after master_read in the cases of “ker” or “lapalce” commands, with FIN flagged before it’s read.
At the moment, Ker and Laplace function correctly in top level DISTR FSM, but with possible error in FSM outputs or secondary Laplace FSM leading to update/leak never updating CAM, maybe due to timing mismatch between update and vacancy tracking, which are last minute changes before I realize that the reason we have zero outputs with FIN flag are due to during the presentation is due to user space program sending odd address commands to 2B channels.

7. EXPERIENCE

The design experience for the FPGA implementation was quite different from what I’m used to in term of RTL for ASIC, I had adopted a different coding style for the FPGA implementation as SystemVerilog has slightly different register coding approach to verilog, timing constraint also causes several architectural modifications in term of segmenting critical paths.
The testability of the design didn’t enter my consideration until quite late into the integration stage. I had relied on modelsim for verification, with no user space c testbench, leading to insufficient time for debugging.
Aside of unfinished debugging, the project can be improved with algorithmic enhancements. As the chip utilization is very low for this design (a target for the ASIC version of the design), 2D distribution and associated boundaries location algorithms such as gradient descent can be employed.

8. ACKNOWLEDGEMENT

Thanks to Prof. Edwards for the course and guidelines for project scope.
Thanks to both TAs for answering questions and debugging board and server.
Thanks to Prof. Mingoo Seok for the spike sorting algorithm iteration and feedbacks.

Appendix

SystemVerilog

```
module SPIKESORT(input logic   clk,
                input logic              reset,
                input logic [15:0]  features,
                input logic              write,
                input logic              chipselect,
                input logic [5:0]      op,//address
                input logic              read,

                output logic [15:0]    sorterOut );

        // features
         logic [11:0]              spk_f;
        logic [2:0]                spk_idx;
        logic [41:0]               Bph;
        logic [6:0]                vph;
        logic [4:0]                commands;
        logic                      sel, clear, ker, laplace, fin, infm, update, leak, v_output;
        logic                      lp, upd, lp0, lp1, lp2, upd0, upd1, upd2;
        logic [7:0]                ocp;

        assign lp = commands[1];
        assign upd = commands[2];
        assign leak = commands[3];
        assign ker = commands[4];

        assign sorterOut[2:0] = spk_idx;
        assign sorterOut[4] = fin;
        assign sorterOut[3] = v_output;
        assign sorterOut[15:8] = (sel)?ocp:{vph,1'b0};

        DISTR memdistr(.init(reset), .addr(spk_f), .*);
        GC gcmod(.*);

  always_ff @(posedge clk) begin
    if (reset) begin
                        spk_f <= 12'b000000000000;
    end else if (chipselect && write) begin
                        spk_f <= {features[13:8], features[5:0]};
                        commands <= op[4:0];
          end
          sel <= op[5];
        end
```

```
    always_ff @(posedge clk) begin
      lp0 <= lp;
      lp1 <= lp0;
      lp2 <= lp1;
      laplace <= lp2;

      upd0 <= upd;
      upd1 <= upd0;
      upd2 <= upd1;
      update <= upd2;
    end
endmodule
```

```
module DISTR(input logic clk, init, clear, ker, laplace, read,
                                   input logic [11:0] addr,
                                   output logic fin,
                                   output logic infm,
                                   output logic [41:0] Bph,
                                   output logic [6:0] vph);

logic [15:0]        wData, ovrData, rdData;
logic [5:0]         addr_m;
logic [5:0]         nxroll, roll;
logic               ovfl;

// mem control state machine
logic [3:0]         nstate, state;
logic                rd, wr, sqAddr, incr, z, p;

// laplace op state machine
logic [2:0]         lpstate, nlpstate;
logic [15:0]        buff [2:0];
logic [15:0]        min, nxmin;
logic [5:0]         bound, nxbound;
logic               push_is, push_seg, en_min;
logic               in_is;

parameter IDLE = 4'b0000, CL = 4'b0001, RSH = 4'b0010, WSH = 4'b0011, CSH = 4'b0100, RP = 4'b0101,
WP = 4'b0110, RLP = 4'b0111, FIN = 4'b1001, CSHF = 4'b1010, FIN2 = 4'b1011, FLP = 4'b1100, RH =
4'b1101, WH = 4'b1110;
parameter STANDBY = 3'b000, FT = 3'b001, FP = 3'b010, TR = 3'b011, CRS = 3'b100, PK = 3'b101;


assign ovrData = (z)?16'b0000000000000000:(rdData >> 1);
assign wData = (sqAddr)?ovrData:(rdData+1);
assign addr_m = (sqAddr)?roll:((p)?addr[11:6]:addr[5:0]);
assign ovfl = &rdData[15:1];
// laplace
assign in_is = ((buff[1]<<1) > (buff[2] + buff[0] + 16'b0000000000001000));

MEMD memkde(.clk(clk), .rd(rd), .wr(wr&(~clk)), .addr(addr_m), .wData(wData), .rdData(rdData));
FIS featureInfSmp(.clk(clk), .addr(addr), .push(push_is), .in(in_is), .infm(infm));
SEGS segments(.clk(clk), .push(push_seg), .in(bound), .boundary(Bph), .vph(vph));

always_ff @(posedge clk)
begin
  case (state)
    IDLE:
    begin
      rd = 1'b0;
      wr = 1'b0;
```

```
  incr = 1'b0;
  sqAddr = 1'b0;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b0;
end
CL:
begin
  rd = 1'b0;
  wr = 1'b1;
  incr = 1'b1;
  sqAddr = 1'b1;
  z = 1'b1;
  fin = 1'b0;
                    push_is = 1'b0;
  p = 1'b0;
end
RSH:
begin
  rd = 1'b1;
  wr = 1'b0;
  incr = 1'b0;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b0;
end
WSH:
begin
  rd = 1'b0;
  wr = 1'b1;
  incr = 1'b0;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b0;
end
CSH:
          begin
  rd = 1'b0;
  wr = 1'b0;
  incr = 1'b1;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b0;
```

```
  push_is = 1'b0;
  p = 1'b0;
end
CSHF:
begin
  rd = 1'b0;
  wr = 1'b0;
  incr = 1'b1;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b0;
end
RP:
begin
  rd = 1'b1;
  wr = 1'b0;
  incr = 1'b0;
  sqAddr = 1'b0;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b1;
end
WP:
begin
  rd = 1'b0;
  wr = 1'b1;
  incr = 1'b0;
  sqAddr = 1'b0;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b1;
end
RH:
begin
  rd = 1'b1;
  wr = 1'b0;
  incr = 1'b0;
  sqAddr = 1'b0;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b0;
end
WH:
```

```
begin
  rd = 1'b0;
  wr = 1'b1;
  incr = 1'b0;
  sqAddr = 1'b0;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b0;
  p = 1'b0;
end
RLP:
begin
  rd = 1'b1;
  wr = 1'b0;
  incr = 1'b1;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b0;
  push_is = 1'b1;
  p = 1'b0;
end
FIN:
begin
  rd = 1'b0;
  wr = 1'b0;
  incr = 1'b0;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b1;
  push_is = 1'b0;
  p = 1'b0;
end
FIN2:
begin
  rd = 1'b0;
  wr = 1'b0;
  incr = 1'b0;
  sqAddr = 1'b1;
  z = 1'b0;
  fin = 1'b1;
  push_is = 1'b0;
  p = 1'b0;
end
FLP:
begin
  rd = 1'b0;
  wr = 1'b0;
  incr = 1'b0;
```

```
      sqAddr = 1'b1;
      z = 1'b0;
      fin = 1'b1;
      push_is = 1'b1;
      p = 1'b0;
    end
    default:
    begin
      rd = 1'b0;
      wr = 1'b0;
      incr = 1'b0;
      sqAddr = 1'b0;
      z = 1'b0;
      fin = 1'b0;
      push_is = 1'b0;
      p = 1'b0;
    end
  endcase
end
always @(posedge clk) begin
  if (init) begin
    roll <= 6'b000000;
    bound <= 6'b000000;
    min <= 16'b1111111111111111;
    state = IDLE;
    lpstate = STANDBY;
    buff[2] <= 16'b0000000000000000;
    buff[1] <= 16'b0000000000000000;
    buff[0] <= 16'b0000000000000000;
  end else begin
    roll <= nxroll;
    bound <= nxbound;
    min <= nxmin;
    state = nstate;
    lpstate = nlpstate;
    if (push_is) begin
      buff[2] <= rdData;
      buff[1] <= buff[2];
      buff[0] <= buff[1];
    end
  end
end

always_comb
begin
    nxroll <= (fin)?(6'b000000):((incr)?(roll + 1):roll);
    nxbound <= (buff[2]<min)?(addr_m - 6'b000011):bound;
    nxmin <= (en_min)?((buff[2]<min)?buff[2]:min):(16'b1111111111111111);
```

```
case (state)
  IDLE:
    if (ker)
      nstate = RP;
    else if (laplace)
      nstate = RLP;
    else
      nstate = IDLE;
  RSH:
    nstate = WSH;
  WSH:
    if (&roll)
      nstate = CSHF;
    else
      nstate = CSH;
  CSH:
    nstate = RSH;
  CSHF:
    nstate = FIN;
  RP:
    nstate = WP;
  WP:
    if (ovfl)
      nstate = RSH;
    else
      nstate = RH;
  RH:
    nstate = WH;
  WH:
    if (ovfl)
      nstate = RSH;
    else
      nstate = FIN;
  RLP:
    if (&roll)
      nstate = FLP;
    else
      nstate = RLP;
  FIN:
  if (read)
      nstate = IDLE;
  else
      nstate = FIN;
  FIN2:
  if (read)
    nstate = IDLE;
  else
```

```
    nstate = FIN2;
  FLP:
  if (read)
    nstate = IDLE;
  else
    nstate = FLP;
  default:
    nstate = IDLE;
endcase

case (lpstate)
  STANDBY:
    if (push_is)
      nlpstate = FT;
    else
      nlpstate = STANDBY;
  FT:
    if (in_is)
      nlpstate = FP;
    else
      nlpstate = FT;
  FP:
    if (in_is)
      nlpstate = FP;
    else
      nlpstate = TR;
  TR:
    if (in_is)
      nlpstate = CRS;
    else
      nlpstate = TR;
  CRS:
    if (&roll)
      nlpstate = STANDBY;
    else if (in_is)
      nlpstate = PK;
    else
     nlpstate = TR;
  PK:
    if (&roll)
      nlpstate = STANDBY;
    else if (in_is)
      nlpstate = PK;
    else
      nlpstate = TR;
  default:
    nlpstate = STANDBY;
endcase
```

```
end
// laplace states
always_ff @(posedge clk)
begin
  case (lpstate)
    STANDBY:
    begin
      push_seg = 1'b0;
      en_min = 1'b0;
    end
    FT:
    begin
      push_seg = 1'b0;
      en_min = 1'b0;
    end
    FP:
    begin
      push_seg = 1'b0;
      en_min = 1'b0;
    end
    TR:
    begin
      push_seg = 1'b0;
      en_min = 1'b1;
    end
    CRS:
    begin
      push_seg = 1'b1;
      en_min = 1'b0;
    end
    PK:
    begin
      push_seg = 1'b0;
      en_min = 1'b0;
    end
    default:
    begin
      push_seg = 1'b0;
      en_min = 1'b0;
    end
  endcase
end

endmodule


// memery for distribution
module MEMD(input logic clk, rd, wr,
```

```
                          input logic [5:0] addr,
                          input logic [15:0] wData,
                          output logic [15:0] rdData);

logic [15:0]      kde [63:0];

always_ff @(negedge clk) begin
    if (rd) begin
        rdData <= kde[addr];
    end
    if (wr) begin
        kde[addr] <= wData;
    end
end

endmodule

// shift registers for informative features
module FIS(input logic clk, push, in,
                          input logic [11:0] addr,
                          output logic infm);

logic [63:0]      informative_sample;

assign infm = informative_sample[addr[11:6]]&informative_sample[addr[5:0]];

always_ff @(negedge clk) begin
    if (push) begin
        informative_sample[62] <= in;
        informative_sample[61:0] <= informative_sample[62:1];
    end
end

endmodule

// shift registers for boundary information
module SEGS(input logic clk, push,
                          input logic [5:0] in,
                          output logic [41:0] boundary,
                          output logic [6:0] vph);

logic [5:0]   regfile [6:0];

assign boundary = {regfile[6],regfile[5],regfile[4],regfile[3],regfile[2],regfile[1],regfile[0]};

always_ff @(negedge clk) begin
    if (push) begin
        regfile[6] <= in;
```

```
            regfile[5] <= regfile[6];
            regfile[4] <= regfile[5];
            regfile[3] <= regfile[4];
            regfile[2] <= regfile[3];
            regfile[1] <= regfile[2];
            regfile[0] <= regfile[1];
            vph <= {1'b1, vph[6:1]};
        end
end

endmodule
```

```
module GC(input logic reset,
                                        input logic [11:0] spk_f,
                                        input logic [41:0] Bph,
                                        input logic [6:0] vph,
                                        //input logic start,
                                        input logic clk, update, leak, infm,
                                        output logic v_output,
                                        output logic [2:0] spk_idx,
                                        output logic [7:0] ocp);

          logic [5:0]          gridi;
          logic [5:0]          grid_idx;

          GRIDFIND gridmod(.*);
          CLUSTERS modc(.grid_idx(gridi), .*);

          always_ff @(posedge clk)
          begin
                    gridi <= grid_idx;
          end

endmodule

module CLUSTERS(input logic reset,
                                                  input logic clk,
                                                  input logic update, leak, infm,
                                                  input logic [5:0] grid_idx,
                                                  output logic v_output,
                                                  output logic [2:0] spk_idx,
                                                  output logic [7:0] ocp);

          logic [1:0]          d0,d1,d2,d3,d4,d5,d6,d7;
          logic [1:0]          wd0,wd1,wd2,wd3,wd4,wd5,wd6,wd7;
          logic [7:0]          v, vwta;
          logic [7:0]          hits;

          always_ff @(posedge clk)
          begin
                    vwta <= v;
          end

          CAM_CLUSTER cl(.*);
          WTA wtamod(.v(vwta), .d0(wd0),.d1(wd1),.d2(wd2),.d3(wd3),.d4(wd4),.d5(wd5),.d6(wd6),.d7(wd7),
.spk_idx(spk_idx), .v_output(v_output));

endmodule
```

```
module GRIDFIND(input logic [11:0] spk_f,
                input logic [41:0] Bph,
                input logic [6:0] vph,
                output logic [5:0] grid_idx);

    REGION peakregion(.boundary(Bph), .v(vph), .feature(spk_f[11:6]), .idx(grid_idx[5:3]));
    REGION hypregion(.boundary(Bph), .v(vph), .feature(spk_f[5:0]), .idx(grid_idx[2:0]));

endmodule

module REGION(input logic [41:0] boundary,
              input logic [6:0] v,
              input logic [5:0] feature,
              output logic [2:0] idx);

    logic [6:0]          gr, vgr;

    assign gr[6] = feature >= boundary[41:36];
    assign gr[5] = feature >= boundary[35:30];
    assign gr[4] = feature >= boundary[29:24];
    assign gr[3] = feature >= boundary[23:18];
    assign gr[2] = feature >= boundary[17:12];
    assign gr[1] = feature >= boundary[11:6];
    assign gr[0] = feature >= boundary[5:0];

    assign vgr = gr&v;

    assign idx = vgr[6]+vgr[5]+vgr[4]+vgr[3]+vgr[2]+vgr[1]+vgr[0];

endmodule
```

```
module CAM_CLUSTER(input logic clk, reset,
                                        //input logic start,
                                        input logic update,
                                        input logic leak,
                                        input logic [5:0] grid_idx,
                                        output logic [1:0] d0,
                                        output logic [1:0] d1,
                                        output logic [1:0] d2,
                                        output logic [1:0] d3,
                                        output logic [1:0] d4,
                                        output logic [1:0] d5,
                                        output logic [1:0] d6,
                                        output logic [1:0] d7,
                                        output logic [7:0] v,
                                        output logic [7:0] ocp);

            logic [7:0] hits;
            logic [7:0] nC;
            logic [7:0] nC_c;
            logic match;

    assign match = |(hits&ocp);

    VAC vac_find(.nC(nC_c), .grow(~match), .*);
    CAMENTRY e0(.newC(nC[0]), .grow(~match), .hit(hits[0]), .adjacent(d0), .v_in(v[0]), .v_clu(ocp[0]), .*);
    CAMENTRY e1(.newC(nC[1]), .grow(~match), .hit(hits[1]), .adjacent(d1), .v_in(v[1]), .v_clu(ocp[1]), .*);
    CAMENTRY e2(.newC(nC[2]), .grow(~match), .hit(hits[2]), .adjacent(d2), .v_in(v[2]), .v_clu(ocp[2]), .*);
    CAMENTRY e3(.newC(nC[3]), .grow(~match), .hit(hits[3]), .adjacent(d3), .v_in(v[3]), .v_clu(ocp[3]), .*);
    CAMENTRY e4(.newC(nC[4]), .grow(~match), .hit(hits[4]), .adjacent(d4), .v_in(v[4]), .v_clu(ocp[4]), .*);
    CAMENTRY e5(.newC(nC[5]), .grow(~match), .hit(hits[5]), .adjacent(d5), .v_in(v[5]), .v_clu(ocp[5]), .*);
    CAMENTRY e6(.newC(nC[6]), .grow(~match), .hit(hits[6]), .adjacent(d6), .v_in(v[6]), .v_clu(ocp[6]), .*);
    CAMENTRY e7(.newC(nC[7]), .grow(~match), .hit(hits[7]), .adjacent(d7), .v_in(v[7]), .v_clu(ocp[7]), .*);

 always_ff @(negedge clk)
 begin
  nC <= nC_c;
 end

endmodule

module VAC(input logic [7:0] ocp,
                            input logic update, grow,
                            output logic [7:0] nC);

    assign nC[0] = update & grow&(~ocp[0]);
    assign nC[1] = update & grow&(~ocp[1])&(~nC[0]);
    assign nC[2] = update & grow&(~ocp[2])&(~(nC[1]|nC[0]));
    assign nC[3] = update & grow&(~ocp[3])&(~(nC[2]|nC[1]|nC[0]));
```

```
    assign nC[4] = update & grow&(~ocp[4])&(~(nC[3]|nC[2]|nC[1]|nC[0]));
    assign nC[5] = update & grow&(~ocp[5])&(~(nC[4]|nC[3]|nC[2]|nC[1]|nC[0]));
    assign nC[6] = update & grow&(~ocp[6])&(~(nC[5]|nC[4]|nC[3]|nC[2]|nC[1]|nC[0]));
    assign nC[7] = update & grow&(~ocp[7])&(~(nC[6]|nC[5]|nC[4]|nC[3]|nC[2]|nC[1]|nC[0]));

endmodule


module CAMENTRY(input logic clk, reset,

                                input logic newC,
                                input logic grow,
                                input logic update,
                                input logic leak,
                                input logic [5:0] grid_idx,
                                output logic hit,
                                output logic [1:0] adjacent,
                                output logic v_in,
                                output logic v_clu);

        logic [1:0]     u;
        logic [8:0]     entry;
        logic [1:0]     uout;

        logic load;
        logic v;
         logic hit_raw;

        BIMODAL bm0(.load(load), .in(u), .up(hit), .down(leak), .out(uout));
        GRIDADJ ga0(.grid_idx(grid_idx), .c_idxes(entry), .hit(hit), .adjacent(adjacent), .v(v));

        assign v_clu = |u;
        assign v_in = v&v_clu;
        assign load = reset|update;

        always_ff @(negedge clk)
        begin
                if (load) begin
                        u <= (newC)?2'b01:uout;
                end
                if (newC) begin
                        entry <= grid_idx;
                end
        end

endmodule

module BIMODAL(input logic load,
```

```
                                        input logic [1:0] in,
                                        input logic up,
                                        input logic down,
                                        output logic [1:0] out);

        logic [1:0]     u, d, x, y;
        // up
        assign u[1] = |in;
        assign u[0] = ~(in[0]&(~in[1]));
        // down
        assign d[1] = &in;
        assign d[0] = in[1]&(~in[0]);

        assign x = (up&u[1])?u:in;
        assign y = (down)?d:x;
        assign out = (load)?y:in;

endmodule

module GRIDADJ(input logic [5:0] grid_idx,
                                        input logic [5:0] c_idxes,
                                        output logic [1:0] adjacent,
                                        output logic hit,
                                        output logic v);

    logic [2:0]             match_b, adj_b, c_b;

    ATTR_F  attr_fp(.A(grid_idx[5:3]), .B(c_idxes[5:3]), .match(match_b[1]), .adj(adj_b[1]), .c(c_b[1]));
    ATTR_F  attr_fh(.A(grid_idx[2:0]), .B(c_idxes[2:0]), .match(match_b[0]), .adj(adj_b[0]), .c(c_b[0]));

    assign v = &c_b;
    assign hit = &match_b;
    assign adjacent = adj_b[2]+adj_b[1]+adj_b[0];

endmodule

module ATTR_F(input logic[2:0] A,
                                        input logic[2:0] B,
                                        output logic match,
                                        output logic adj,
                                        output logic c);


    logic [2:0]         X, E;
    logic                   m, n;

    assign X = A^B;
    assign E = ~X;
```

```
    assign m = A[2]^A[1];
    assign n = A[1]^A[0];
    assign match = ~|X;
    assign adj = X[0]&((E[2]&E[1])|(E[2]&X[1]&n)|(X[2]&X[1]&m&(~n)));
    assign c = match | adj;

endmodule
```

```
module WTA(input logic [7:0]       v,
                    input logic [1:0]  d0, d1, d2, d3, d4, d5, d6, d7,
                    output logic [2:0]             spk_idx,
                    output logic                                v_output);

          logic [3:0] v4;
          logic [1:0] v2;

          logic i4_0, i4_1, i4_2, i4_3;
          logic it1, it2;
          logic i2_0, i2_1;

          logic [1:0] d4_0, d4_1, d4_2, d4_3;
          logic [1:0] d2_0, d2_1;
          logic [1:0] d_fin;

          WINNER          w4_0(.vA(v[0]), .vB(v[1]), .distA(d0), .distB(d1), .dist_l(d4_0), .v(v4[0]), .sel(i4_0));
          WINNER          w4_1(.vA(v[2]), .vB(v[3]), .distA(d2), .distB(d3), .dist_l(d4_1), .v(v4[1]), .sel(i4_1));
          WINNER          w4_2(.vA(v[4]), .vB(v[5]), .distA(d4), .distB(d5), .dist_l(d4_2), .v(v4[2]), .sel(i4_2));
          WINNER          w4_3(.vA(v[6]), .vB(v[7]), .distA(d6), .distB(d7), .dist_l(d4_3), .v(v4[3]), .sel(i4_3));

          WINNER          w2_0(.vA(v4[0]), .vB(v4[1]), .distA(d4_0), .distB(d4_1), .dist_l(d2_0), .v(v2[0]),
.sel(i2_0));
          WINNER          w2_1(.vA(v4[2]), .vB(v4[3]), .distA(d4_2), .distB(d4_3), .dist_l(d2_1), .v(v2[1]),
.sel(i2_1));

          WINNER          wFin(.vA(v2[0]), .vB(v2[1]), .distA(d2_0), .distB(d2_1), .dist_l(d_fin), .v(v_output),
.sel(spk_idx[2]));

          assign it1 = (i2_0)?i4_0:i4_1;
          assign it2 = (i2_1)?i4_2:i4_3;

          assign spk_idx[1] = (spk_idx[2])?i2_0:i2_1;
          assign spk_idx[0] = (spk_idx[1])?it1:it2;

endmodule

module WINNER(input logic                  vA,
               input logic                 vB,
               input logic [1:0]       distA,
               input logic [1:0]       distB,

               output logic [1:0]      dist_l,
                              output logic                          v,
                              output logic                          sel);

          logic                lesser;
```

```
    assign lesser = distA < distB;
    assign sel = (vA&vB)?vB:lesser;
    assign v = vA | vB;
    assign dist_l = (sel)?distA:distB;

endmodule
```

User Space C File

```
/*
 * Userspace program that communicates with the led_vga device driver
 * primarily through ioctls
 *
 */

#include <stdio.h>
#include "vga_led.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define LINES 14822
#define LINES 10
#define CMD_KERNEL_DENSITY_ESTIMATION 1
#define CMD_LAPLACE 2
#define CMD_TRAIN_NORMAL 4
#define CMD_TRAIN_LEAKAGE 12
#define CMD_SORT 0

int vga_led_fd;

/* Read and print the maxmin values */
unsigned short read_data() {
	vga_led_arg_t vla;
	if (ioctl(vga_led_fd, VGA_LED_READ, &vla)) {
	perror("ioctl(VGA_LED_READ) failed\n");
	return;
	}
	return vla.data;
}

/* Write max and min values to vga */
void write_maxmin(const unsigned short maxmin, unsigned char command)
{
	vga_led_arg_t vla;
	vla.data = maxmin;
	vla.addr = command;
	if (ioctl(vga_led_fd, VGA_LED_WRITE, &vla)) {
	perror("ioctl(VGA_LED_WRITE) failed");
	return;
	}
```

```
	//printf("fpga comm success\n");
}

/* parse file f and load vals into minimum and maximum arrays */
void load_csv(char *f, unsigned short maxminarr[]) {
	// open file
	char buf[1024];
	FILE *file = fopen(f,"r");
	if (!file) {
	fprintf(stderr, "Can't open file.\n");
	exit(EXIT_FAILURE);
	}

	// loop through file line by line
	int i = 0;
	while (fgets(buf, sizeof buf, file) != NULL) {
	int max;
	int min;

	// scan integers into max and min
	if (sscanf(buf, "%d,%d", &max, &min) != 2) {
	// check for failure
	fprintf(stderr, "sscanf failed.");
	exit(EXIT_FAILURE);
	}

	//printf("%i and %i\n", max,min);

	// represent max and min in binary by concatenating
	unsigned short maxmin = (unsigned short) max;
	maxmin = maxmin << 8;
	maxmin = maxmin + min;

	maxminarr[i] = maxmin;
	i++;
	}

	fclose(file);
}

void print_as_binary(unsigned short x)
{
	static char b[9];
	b[0] = '\0';

	while (x) {
	if (x & 1)
	strcat(b, "1");
```

```
        else
        strcat(b, "0");

        x >>= 1;
        }
        printf("%s\n",b);
}

const char * byte_to_binary(unsigned short x)
{
        static char b[9];
        b[0] = '\0';

        int z;
        for (z = 128; z > 0; z >>= 1)
        {
        strcat(b, ((x & z) == z) ? "1" : "0");
        }

        printf("%s\n",b);
}

int main()
{
        vga_led_arg_t vla;
        int i;
        static const char device[] = "/dev/vga_led";

        //printf("VGA LED Userspace program started\n");

        if ( (vga_led_fd = open(device, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", device);
        return -1;
        }

        // extract minimum and maximum values from csv
        unsigned short maxminarr[LINES];
        // TODO change back to features.csv
        // load_csv("./features.csv", maxminarr);
        load_csv("./features.csv", maxminarr);

        unsigned short rdata = 0;

        // kernel density estimation
        printf("===================\n");
        printf("execute kernel density estimation\n");
        for (i = 0; i < LINES; i++) {
        //printf("\n");
```

```
// write max and min to fpga
write_maxmin(maxminarr[i], CMD_KERNEL_DENSITY_ESTIMATION);

// read data from fpga until non-zero
//printf("processing ");
while (rdata == 0) {
//printf(".");
rdata = read_data();
}
//printf("\n");
//printf("result received = %d\n", rdata);
}

// start laplace on fpga
printf("====================\n");
printf("execute laplace\n");
write_maxmin(42, CMD_LAPLACE);
rdata = 0;
while (rdata < 8) {
rdata = read_data();
}

// training
printf("====================\n");
printf("execute training\n");
for (i = 0; i < LINES; i++) {
if (i%128 == 0) {
// update with leakage
write_maxmin(maxminarr[i], CMD_TRAIN_LEAKAGE);
} else {
// update normally
write_maxmin(maxminarr[i], CMD_TRAIN_NORMAL);
}
// continue training when sample is done processing
rdata = 0;
while (rdata < 8) {
rdata = read_data();
}
print_as_binary(rdata);
//printf("trained return val: %d\n", (rdata));
}

// sorting
printf("====================\n");
printf("execute sorting\n");
for (i = 0; i < LINES; i++) {
// send sort signal
```

```
        write_maxmin(maxminarr[i], CMD_SORT);
        // continue training when sample is done processing
        //rdata = 0;
        rdata = read_data();
        /*
        while (rdata < 8) {
        rdata = read_data();
        }
        */
        //printf("sorted value: %d\n", (rdata - 8));
        }

        printf("Spike Sorting userspace program terminating\n");
        return 0;
}
```

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_led.h"

#define DRIVER_NAME "vga_led"

/*
 * Information about our device
 */
struct vga_led_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    unsigned short data;
} dev;

/*
 * Write concatenated max and min value
 * Assumes max and min are concatenated properly
 */
static void write_data(int addr, u8 data)
{
    iowrite16(data, dev.virtbase + addr);
    dev.data = data;
}

//static void read_data()

/*
 * Handle ioctl() calls from userspace:
 * Read or write the data on single digits.
 * Note extensive error checking of arguments
 */
static long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    vga_led_arg_t vla;

            //if (copy_from_user(&vla, (vga_led_arg_t *) arg, sizeof(vga_led_arg_t)))
```

```
        //          return -EACCES;
        //write_data(cmd, vla.data);

        //return 0;

    switch (cmd) {
    case VGA_LED_WRITE:
         if (copy_from_user(&vla, (vga_led_arg_t *) arg, sizeof(vga_led_arg_t)))
                 return -EACCES;
         write_data(vla.addr, vla.data);
         break;

    case VGA_LED_READ:
        /* TODO
         if (copy_from_user(&vla, (vga_led_arg_t *) arg,
        sizeof(vga_led_arg_t)))
        return -EACCES;
         //if (vla.digit > 8)
        //          return -EINVAL;
         vla.data = dev.data;
        */
         if (copy_from_user(&vla, (vga_led_arg_t *) arg, sizeof(vga_led_arg_t)))
        return -EACCES;
        vla.data = (unsigned short) ioread16(dev.virtbase);
         if (copy_to_user((vga_led_arg_t *) arg, &vla, sizeof(vga_led_arg_t)))
        return -EACCES;
        break;

    default:
         return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_led_fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = vga_led_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_led_misc_device = {
    .minor          = MISC_DYNAMIC_MINOR,
    .name           = DRIVER_NAME,
    .fops           = &vga_led_fops,
};
```

```
/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_probe(struct platform_device *pdev)
{
    //static unsigned char welcome_message[VGA_LED_DIGITS] = {
            //          0x3E, 0x7D, 0x77, 0x08, 0x38, 0x79, 0x5E, 0x00};
    int i, ret;

    /* Register ourselves as a misc device: creates /dev/vga_led */
    ret = misc_register(&vga_led_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
            ret = -ENOENT;
            goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                                   DRIVER_NAME) == NULL) {
            ret = -EBUSY;
            goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
            ret = -ENOMEM;
            goto out_release_mem_region;
    }

    /* Display a welcome message */
    //for (i = 0; i < VGA_LED_DIGITS; i++)
    //    write_digit(i, welcome_message[i]);

    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&vga_led_misc_device);
    return ret;
}

/* Clean-up code: release resources */
```

```
static int vga_led_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&vga_led_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_led_of_match[] = {
    { .compatible = "altr,vga_led" },
    {},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_led_driver = {
    .driver    = {
            .name    = DRIVER_NAME,
            .owner    = THIS_MODULE,
            .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .remove    = __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_led_exit(void)
{
    platform_driver_unregister(&vga_led_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

typedef struct {
  unsigned char addr;  // typically: command for fpga
  unsigned short data; // typically: concatenated max-min (ie. max*2^8 + min)
} vga_led_arg_t;

#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t *)
#define VGA_LED_READ  _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t *)

#endif
```