Lucas Kanade Optical Flow – from C to OpenCL on CV SoC

Dmitry Denisenko

July 8, 2014

Optical Flow Inputs and Outputs

frame i

frame i+1

Optical flow frame using i+1 and i

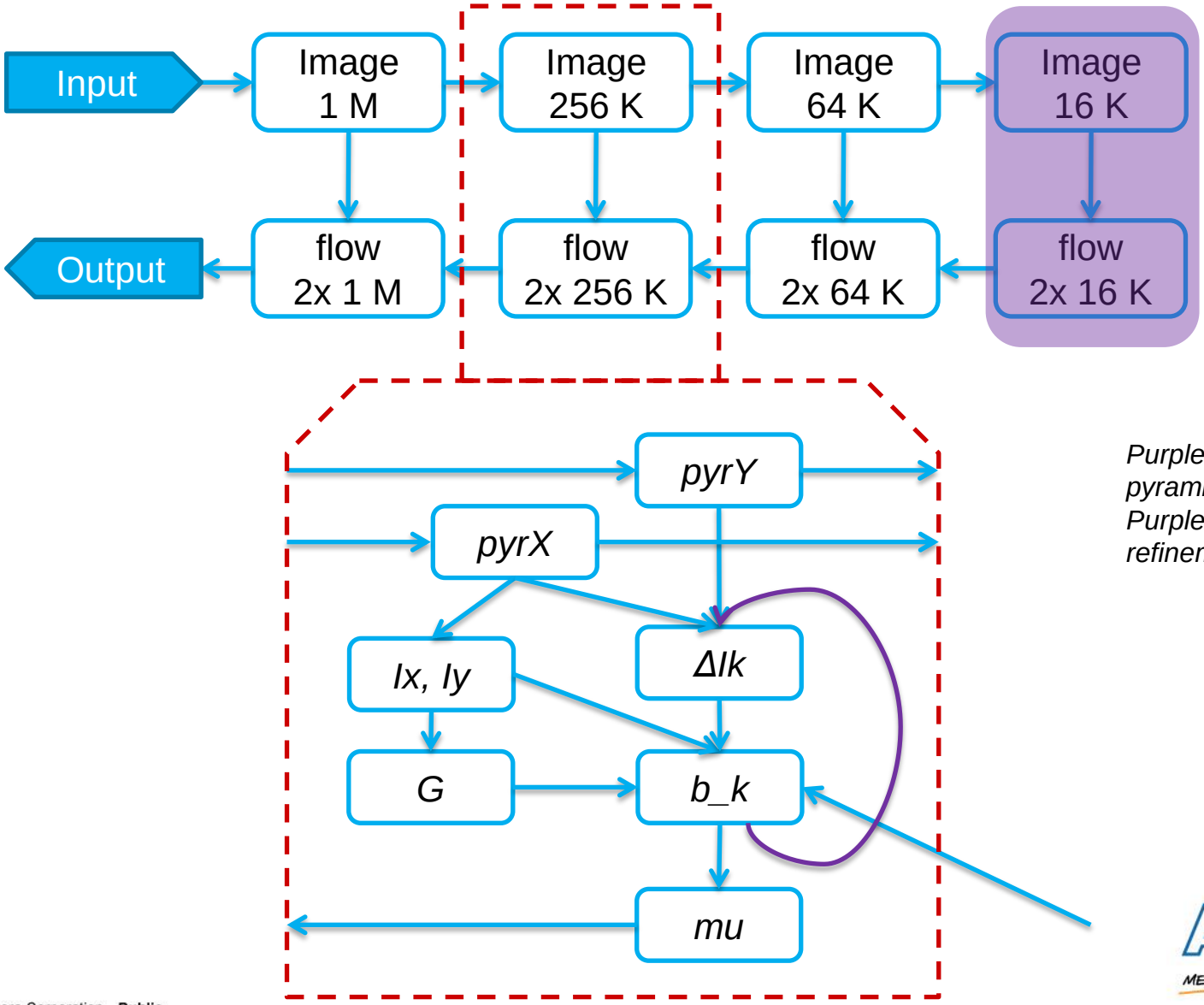
Van in the middle is moving to the right, black sedan and van on the left are moving to the left.
Image is unfiltered, so cloud, tree, and road textures show up as local motions.

direction legend

Overview

- **C Model**
- **Straight conversion from C to OpenCL**
  - Clear no-fit
- **Shift registers + 2D arithmetic loop**
- **Shift registers + 1D arithmetic loop**
- **Shift registers and no loop**

Data Flow in LK (pyramidal, iterative)



Purple rectangle is one pyramidal level.
Purple arrow is iterative refinement.

C Model

- **Handy for:**
  - Reference output
  - Understand the algorithm
  - Figure out smallest data sizes for all computation steps
    - float vs int vs short vs char
    - Find smallest data type that avoid overflow and gives correct answer.
- **Writing code in C-style:**
  - mallocs / frees for temporary buffers.
  - Assume full frame is available.
  - Use many nested loops to process data.
  - Many array dereferences to get pixel values.
    - Very little performance gain in being careful with memory accesses for small (~ 1MB) images since CPU caches are huge.

C Model

```
// Image access macros without and with bounds checking.
#define PO(image,x,y,W) ((image)[(y)*(W)+(x)])
#define P(image,x,y,W,H) ((image)[( (y)>=H ? H-1 : ((y)<0 ? 0:(y)) )*(W)+( (x)>=W ? W-1 : ((x)<0 ? 0 : (x)) )])

void optical_flow_for_images (uchar *im1, uchar *im2, int W_in, int H_in,
                              float *g1, float *g2, uchar *pyr1, uchar *pyr2, uchar *out) {

  // Compute pyramid levels for both images and store them in pyr1/pyr2
  build pyramid structure (im1, im2, W_in, H_in, pyr1, pyr2, pyr_start1, pyr_start2);

  for (int ipyr = NUM_PYRAMID_LEVELS-1; ipyr >= 0; ipyr--) {
    // calculate image derivates once per pyramid level
    int *Ix = (int*)malloc(W*H*sizeof(int));
    int *Iy = (int*)malloc(W*H*sizeof(int));
    for (int j = 0; j < H; j++) {
      for (int i = 0; i < W; i++) {
        // Image derivatives in x and y directions. Equations #19, 20
        PO(Ix,i,j,W) = (P(im1, i+1, j, W, H) - P(im1, i-1, j, W, H)) >> 1;
        PO(Iy,i,j,W) = (P(im1, i, j+1, W, H) - P(im1, i, j-1, W, H)) >> 1;
    } }

    // Default value for output and flow guess is 0
    memset (cur_g, 0, 2*W*H*sizeof(float));
    memset (out, 255, 3*W*H);
    for (int j = WINDOW_SIZE; j < H - WINDOW_SIZE; j ++) {
      for (int i = WINDOW_SIZE; i < W - WINDOW_SIZE; i ++) {
        float G_inv[4] = {0.0f, 0.0f, 0.0f, 0.0f};
        if (!get gradient inv (Ix, Iy, i, j, W, G_inv)) {
           // Non-invertable matrix.
           continue; }
        compute flow for pixel (i, j, im1, im2, cur_g, prev_g, G_inv, out);
} } } }
```

C Model

```
// Calculate inverse of spatial gradient matrix. Eq #23.
int get_gradient_inv (int *Ix, int *Iy, int i, int j,
                      int W, float *G_inv) {
  int G[4] = {0, 0, 0, 0};
  for (int wj = -WINDOW_SIZE; wj <= WINDOW_SIZE; wj++) {
    for (int wi = -WINDOW_SIZE; wi <= WINDOW_SIZE; wi++)
{
      int cIx = PO(Ix,i+wi,j+wj,W);
      int cIy = PO(Iy,i+wi,j+wj,W);
      G[0] += cIx * cIx;
      G[1] += cIx * cIy;
      G[2] += cIx * cIy;
      G[3] += cIy * cIy;
    }
  }
  // Will contain inverse of G
  return get_matrix_inv (G, G_inv);
}
```

```
// Downsample image by one stage of the pyramid.
//    Output size is (W+1)/2 x (H+1)/2
void get_next_pyramidal_image (uchar *im_in, uchar
*im_out, int W, int H) {

  int outW = (W + 1)/2;
  for (int j = 0; j < (H + 1)/2; j++) {
    for (int i = 0; i < (W + 1)/2; i++) {
      // Formula #2 in the paper.
      PO(im_out, i, j, outW) =
          (P(im_in, 2*i, 2*j, W,H) >> 2) +
         ((P(im_in, 2*i-1, 2*j, W,H) +
           P(im_in, 2*i+1, 2*j, W,H) +
           P(im_in, 2*i, 2*j-1, W,H) +
           P(im_in, 2*i, 2*j+1, W,H)) >> 3) +
         ((P(im_in, 2*i-1, 2*j-1, W,H) +
           P(im_in, 2*i+1, 2*j+1, W,H) +
           P(im_in, 2*i+1, 2*j-1, W,H) +
           P(im_in, 2*i-1, 2*j+1, W,H)) >> 4);
} } }
```

ALTERA
MEASURABLE ADVANTAGE™

C Model

```
// Given G inverse for given pixel (i,j), do iterative flow computation
void compute_flow_for_pixel (int i, int j, uchar *im1, uchar *im2, float *cur_g, float *prev_g, float G_inv[4], uchar *out) {
   // Guess for optical flow for each iteration
   float mu[2] = {0.0f, 0.0f};
   for (int iter = 0; iter < ITER_PER_PYRAMID_LEVEL; iter++) {
      // Image mismatch vector. Formula #29
      float b_k[2] = {0, 0};
      for (int wj = -WINDOW_SIZE; wj <= WINDOW_SIZE; wj++) {
         for (int wi = -WINDOW_SIZE; wi <= WINDOW_SIZE; wi++) {
            float gi = 2 * P(prev_g, 2*((i+wi)/2),   (j+wj)/2, W, H/2);
            float gj = 2 * P(prev_g, 2*((i+wi)/2)+1, (j+wj)/2, W, H/2);
            // Image difference. Formula #30
            uchar im2_val = get_interpolated_value (im2, i+wi+gi+mu[0], j+wj+gj+mu[1], W, H);
            int deltaIk = P(im1, i+wi, j+wj, W, H) - im2_val;
            b_k[0] += deltaIk * PO(Ix,i+wi,j+wj,W);
            b_k[1] += deltaIk * PO(Iy,i+wi,j+wj,W);
         }
      }
      // Guess for next iteration + current flow. Formulas #28 & #31
      mu[0] = mu[0] + G_inv[0] * b_k[0] + G_inv[1] * b_k[1];
      mu[1] = mu[1] + G_inv[2] * b_k[0] + G_inv[3] * b_k[1];
   }
   float fx = mu[0] + 2 * P(prev_g, 2*(i/2),   j/2, W, H/2);
   float fy = mu[1] + 2 * P(prev_g, 2*(i/2)+1, j/2, W, H/2);

   P(cur_g, 2*i,   j, 2*W, H) = fx;
   P(cur_g, 2*i+1, j, 2*W, H) = fy;

   uchar *ptr = &(P(out, 3*i, j, 3*W, H));
   computeColor (fx * FLOW_SCALING_FACTOR, fy * FLOW_SCALING_FACTOR, ptr);
}
```

ALTERA
MEASURABLE ADVANTAGE™

Straight OpenCL from C Model

- **Pull top-level function (optical_flow_for_images) into a kernel file.**
- **Don't need ND-range kernel. Let compiler extract parallelism from loops.**
  - So NO change to loops.
- **Can't do these things in OpenCL kernels:**
  - malloc, memset so rip out temp buffers
  - Re-compute Ix/Iy when needed, don't bother storing.
- **After go no fits, keep removing features until fits on CV:**
  - No pyramidal, non-iterative.
  - Non-iterative ➔ no need to store G/G_inv. Compute it at the same time as the image mismatch vector.
- **60 to 70% utilization, ~ 4 seconds / frame.**

ALTERA
MEASURABLE ADVANTAGE™

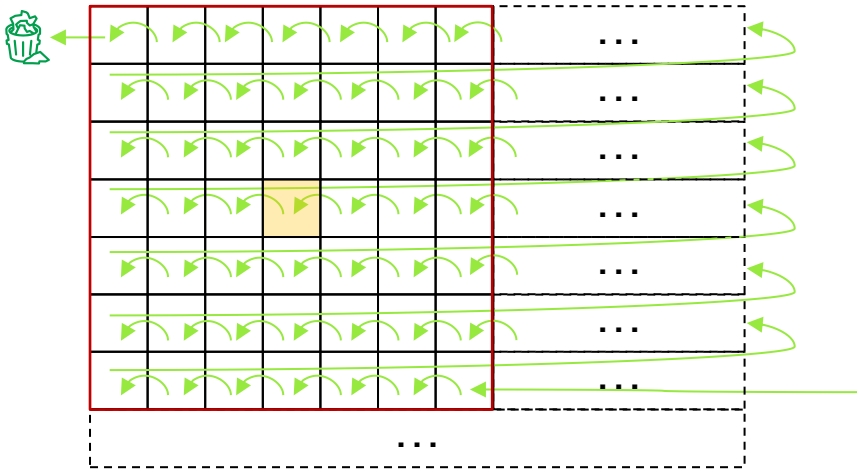
Straight OpenCL from C Model

```
#define Ix(im1,i,j,W,H) ((P(im1, (i)+1, (j), W, H) - P(im1, (i)-1, (j), W, H)) >> 1)
#define Iy(im1,i,j,W,H) ((P(im1, (i), (j)+1, W, H) - P(im1, (i), (j)-1, W, H)) >> 1)
kernel void optical_flow_for_images (constant uchar * restrict im1, global uchar *restrict im2, int W_in, int H_in,
                                     global uchar * restrict out, constant int * restrict colorwheel, int ncols) {
   while (j < H - WINDOW_SIZE) {
      float G_inv[4]; float mu[2]; int G[4]; float b_k[2];
      int wj = -WINDOW_SIZE, wi = -WINDOW_SIZE;
      while (wj <= WINDOW_SIZE) {
         // Image difference. Formula #30
         uchar im2_val = P(im2, (int)(i+wi), (int)(j+wj), W, H);
         int deltaIk = P(im1, i+wi, j+wj, W, H) - im2_val;
         int cIx = Ix(im1, i+wi,j+wj,W,H);
         int cIy = Iy(im1, i+wi,j+wj,W,H);
         G[0] += cIx * cIx;  G[1] += cIx * cIy;
         G[2] += cIx * cIy;  G[3] += cIy * cIy;
         b_k[0] += deltaIk * cIx;   b_k[1] += deltaIk * cIy;
         wj = (wi>WINDOW_SIZE) ? (wj+1) : wj;
         wi = (wi>WINDOW_SIZE) ? -WINDOW_SIZE : (wi+1);
      }
      grad_invertible = get_matrix_inv (G, G_inv);
      // Guess for next iteration. Formulas #28 & #31
      mu[0] = mu[0] + G_inv[0] * b_k[0] + G_inv[1] * b_k[1];
      mu[1] = mu[1] + G_inv[2] * b_k[0] + G_inv[3] * b_k[1];
      float fx = mu[0] + 0; //2 * P(prev_g, 2*(i/2),   j/2, W, H/2);
      float fy = mu[1] + 0; //2 * P(prev_g, 2*(i/2)+1, j/2, W, H/2);
      global uchar *ptr = &(P(out, 3*i, j, 3*W, H));
      if (grad_invertible) {
         computeColor (colorwheel, ncols, fx, fy, ptr);  }
      j = (i+1)>=(W-WINDOW_SIZE) ? (j+1) : j;
      i = (i+1)>=(W-WINDOW_SIZE) ? WINDOW_SIZE : (i+1);
} }
```

ALTERA
MEASURABLE ADVANTAGE™

The way to go – Shift Register!

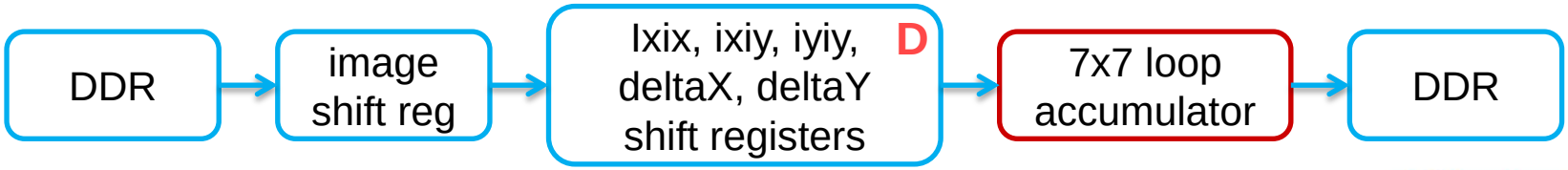
- **Shift register design patter is a great way to implement image processing functions.**
  - All tap points must be constants!
- **Only take as much data as needed, as soon as it's available.**
  - Working on WS x WS image portion requires (WS-1) x WIDTH + WS register.
  - Since need derivatives for each point in the window, need additional 1-pixel border around the window.
    - For WINDOW_SIZE=7 and WIDTH=1280, shift register size becomes 8x1280+9=10,249.
  - Will need multiple shift registers, some will be smaller if don't need the whole window.



All values within red square are tapped. Computed value is for the orange square.

Shift registers with 2D arithmetic loop

- **The core of the algorithm, compute_flow_for_pixel(), computes parts of the gradient matrix and image difference in 7x7 neighbourhood of the pixel.**
- **Shift register code must use constant indices to tap the shift register ➔ must unroll all loops that access the shift register.**
- **For every pixel, need to compute:**
  - Ix, Iy, Ix*Ix, Ix*Iy, Iy*Iy, ΔI*Ix, ΔI*Iy
  - Have 7 shift registers:
    - Image1, image2, last five values above.
- **Total computation load per pixel = 245 adds + 5 mults:**
  - Shift register updating: 5 mult, 0 adds
  - Flow computation: 49 x 5 = 245 adds



Shift registers marked with D are deep.

Shift registers with 2D arithmetic loop: Code

```
kernel void optical_flow_for_images (global uchar * restrict im1, global uchar * restrict im2, int W_in, int H_in, global uchar4 * restrict out) {

  #define RES_REG_SIZE ( (WINDOW_SIZE+1) * WIDTH + (WINDOW_SIZE+2) )
  uchar im1_reg[RES_REG_SIZE], im2_reg[RES_REG_SIZE];
  int ixix_reg[RES_REG_SIZE],  ixiy_reg[RES_REG_SIZE], iyiy_reg[RES_REG_SIZE], deltaK_ix_reg[RES_REG_SIZE], deltaK_iy_reg[RES_REG_SIZE];

  #pragma unroll
  for (uint i = 0; i < RES_REG_SIZE; i++) { <every_reg[i] = 0> }

  int i = 0, j = 0;
  while (j < HEIGHT) {
    #pragma unroll
    for (uint i_shift = 1; i_shift < RES_REG_SIZE; i_shift++) { <every_reg[i_shift-1] = every_reg[i_shift];>  }

    im1_reg[RES_REG_SIZE-1] = PO(im1, i, j, WIDTH);          im2_reg[RES_REG_SIZE-1] = PO(im2, i, j, WIDTH);
    int deltaK = im1_reg[RES_REG_SIZE-1] - im2_reg[RES_REG_SIZE-1];
    int cIx = (im1_reg[RES_REG_SIZE] - im1_reg[RES_REG_SIZE-2]) >> 1;
    int cIy = (im1_reg[RES_REG_SIZE-1 + WIDTH] - im1_reg[RES_REG_SIZE-1 - WIDTH]) >> 1;
    ixix_reg[RES_REG_SIZE-1] = cIx * cIx;   ixiy_reg[RES_REG_SIZE-1] = cIx * cIy;   iyiy_reg[RES_REG_SIZE-1] = cIy * cIy;
    deltaK_ix_reg[RES_REG_SIZE-1] = deltaK * cIx;    deltaK_iy_reg[RES_REG_SIZE-1] = deltaK * cIy;

    // Avoiding borders
    if (j >= WINDOW_SIZE && j < (HEIGHT - WINDOW_SIZE) && i >= WINDOW_SIZE && i < (WIDTH - WINDOW_SIZE)) {
      #pragma unroll
      for (int wj = 1; wj <= WINDOW_SIZE; wj++) {
        #pragma unroll
        for (int wi = 1; wi <= WINDOW_SIZE; wi++) {
          G[0] += PO(ixix_reg, wi, wj, WIDTH);   G[1] += PO(ixiy_reg, wi, wj, WIDTH);    G[3] += PO(iyiy_reg, wi, wj, WIDTH);
          b_k[0] += PO(deltaK_ix_reg, wi, wj, WIDTH);    b_k[1] += PO(deltaK_iy_reg, wi, wj, WIDTH);
      }  }
      G[2] = G[1];
      get_matrix_inv (G, G_inv);
      float fx = G_inv[0] * b_k[0] + G_inv[1] * b_k[1];   float fy = G_inv[2] * b_k[0] + G_inv[3] * b_k[1];
    }
    <convert (fx,fy) to color and save result to out(i,j)>
    j = (i+1)>=(WIDTH) ? (j+1) : j;   i = (i+1)>=(WIDTH) ? 0 : (i+1);
}  }
```

Storing ixix, ixiy, iyiy instead of simply ix and iy saves multiplications: do them once per pixel here instead of 49 times in loop below.

New values for im1_reg & im2_reg are taken from DDR. Other reg's are updates based on tapped values from im1_reg & im2_reg.

Wait for shift registers to have valid data.

Fully unroll 7x7 loops to have constant taps into shift reg's.

"Funny" nested loop structure.

ALTERA
MEASURABLE ADVANTAGE™

Shift registers with 2D arithmetic loop: results

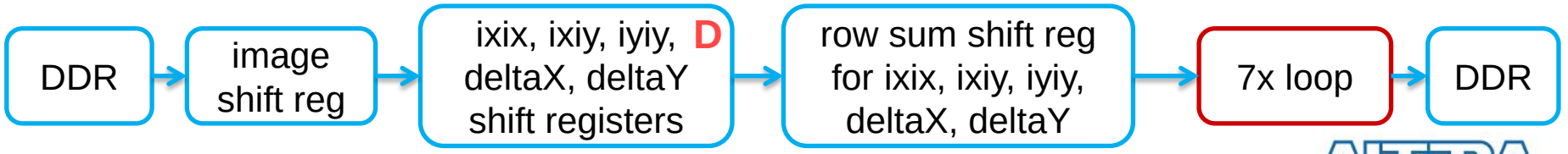
- **Final result is ~350% device utilization**
  - Due to highly unrolled loops.
- **Compiler sizes shift registers correctly.**
  - E.g. im1_reg is declared to have 8xWIDTH+9 elements. However, it's tapped only at 3 points: <last>, <last>-WIDTH, <last>-2xWIDTH. So it only needs to be around 2xWIDTH big.

ALTERA
MEASURABLE ADVANTAGE™

Shift registers with 1D arithmetic loop

- **2D arithmetic loop re-does computations 49 times for every pixel.**
- **If keep *running row sums* over 7 entries for every pixel, only need to sum these row sums over the 7 rows.**
  - Updating row sum: subtract pixel value 7 positions to the left, add new pixel value.
- **Total computation load per pixel = 45 adds + 10 mults:**
  - Shift register updating: 10 mult, 10 adds
  - Flow computation: 7 x 5 = 35 adds
- **Results:**
  - 55% utilization, 429 / 553 M10Ks
  - 85MHz fmax
  - Runs at 1 pixel / cycle, which is ~85 FPS.

This optimization relies on the same operation applied to all pixels!

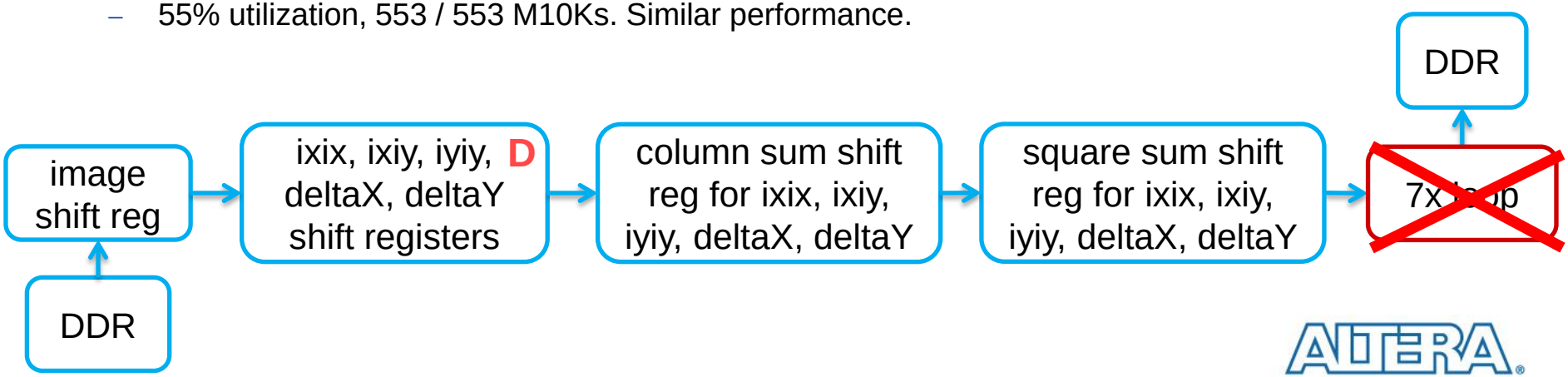


Shift registers with no arithmetic loop

- **Using 1D arithmetic loop still has redundant computation:**
  - Just think about the final computation for a pixel *below* the current one.
- **Keep running column sums AND running square sums.**
  - running column sum is updated by subtracting top-most element and adding new one.
  - Running square sum is updated by subtracting left-most running column and adding new column on the right.
- **Total computation load per pixel = 20 adds + 10 mults:**
  - Shift register updating: 10 mult, 20 adds
  - Flow computation: 0 mults, 0 adds

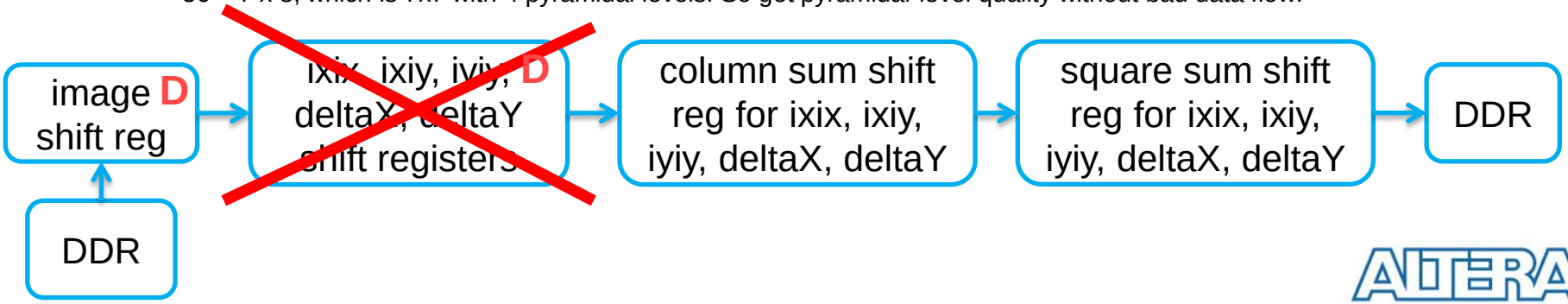
Arithmetic load is independent of window size!

- **Results:**
  - 55% utilization, 553 / 553 M10Ks. Similar performance.



Shift registers with no arithmetic loop, version2

- **The area for arithmetic has been reduced but the RAM count is large.**
  - Each of five deep registers is 32 bytes x ~ 10K → requires 16K/256 = 64 M10Ks.
- **Get rid of ixix, ixiy, iyiy, deltaK_ix, deltaK_iy shift registers**
  - Re-compute from image shift registers on the fly.
  - Images are uchars instead of ints for ixix etc, 2 deep registers instead of 5 → 1/10 x data size reduction.
    - However, image shift register has to be one row deeper → 9/8 x data size increase. Total impact is 11% of the original size.
- **Result: 44% utilization, 263 / 553 M10Ks (down from 553 M10Ks)**
- **56x56 window fits with this implementation:**
  - 80 MHz fmax, **76 FPS**
  - 44% utilization, 500 / 553 M10Ks, 61 / 112 DSPs.
  - 56 = 7 x 8, which is 7x7 with 4 pyramidal levels. So get pyramidal-level quality without bad data flow!



Window size 7 vs 56

Window size = 7

Window size = 56

Still gotta fix the borders. I know...