EE 180

Winter 2022

Lab 2

Lab 2 Report

Adrian Saldana & Noor Fakih

Board's host name: ee180-23z.stanford.edu

Assignment/Problem Description:

Lab 2 required us to optimize a C++ implementation of Sobel filter for an 800MHz ARM Cotex-A9 processor on the Zedboard development board. The goal of the assignment was to experience how having an understanding of computer architecture can result in highly efficient code and the different methods to do so such as vectorizing and strength reduction. In the first part of the lab, we optimize the single-threaded performance of the Sobel filter. In the second part, we build upon our single-thread optimization to create a multithreaded version that uses thread-level parallelism.

Discussion:

Solution/Program Description

The program takes the input video and applies a Sobel filter to each frame of the video. Each frame gets passed to grayScale(), which returns an approximation of the gradient of the image density function and is commonly used to perform edge detection on images. The grayscale frame is then passed to sobelCalc which then performs a 2-D convolution, transforming each pixel into a weighted average of the 3x3 grid of pixels around it.

The multi thread version has two threads working in parallel. Each thread will process half of each frame, and come together to produce a full frame.

Known Bugs and/or Errors

Since we were given a skeleton starter code for the lab, there was already a procedure for ensuring proper image capture and the mechanism for viewing the filtered output. The starter code already had an implementation for a single thread sobel filter, it just was not optimized. Our implementation of the assignment does not have bugs.

EE 180

Winter 2022

Lab 2

Optimization and Results

Starting fps with no optimization - 3

Part 1: Single Thread Case

Original

```
void grayScale(Mat& img, Mat& img_gray_out)
{
  double color;

  // Convert to grayscale
  for (int i=0; i<img.rows; i++) {
    for (int j=0; j<img.cols; j++) {
      color = .114*img.data[STEP0*i + STEP1*j] +
              .587*img.data[STEP0*i + STEP1*j + 1] +
              .299*img.data[STEP0*i + STEP1*j + 2];
      img_gray_out.data[IMG_WIDTH*i + j] = color;
    }
  }
}
```

Inefficient operations:

1. multiplying with a float
2. Using floats
3. Repeatedly multiplying and recalculating the same value
4. Double for-loop and only processing one pixel at a time

Optimized

```
void grayScale(Mat& img, Mat& img_gray_out)
{
  //Image Size 640x480
  uint8x16x3_t intlv_rgb;
  uint8_t *grey_ptr = img_gray_out.data;
  unsigned char * ptr = img.data;
  float i_end = (img_gray_out.rows*img_gray_out.cols) >> 4;

  for (int i=0; i < i_end; i++) {
    intlv_rgb = vld3q_u8(ptr); // De-interleaving
    // Loading values
     uint8x16_t r = intlv_rgb.val[0];
     uint8x16_t g = intlv_rgb.val[1];
     uint8x16_t b = intlv_rgb.val[2];
    // Applying weight
    r  = vshrq_n_u8(r,3); //2^-3 = 0.125
    g = vshrq_n_u8(g,1); //2^-1 = 0.5
    b = vshrq_n_u8(b,2); //2^-2 = 0.25
    // Sum of modified RGB
    uint8x16_t sum = vaddq_u8(r,g);
    sum = vaddq_u8(sum,b);
    // Storing Greyscale
    vst1q_u8(grey_ptr, sum);
    ptr = ptr + 48;
    grey_ptr = grey_ptr + 16;
  }
}
```

We restructured the double for-loop into a single for-loop for easier pointer manipulation. We then used neon vectors and intrinsics so we can process 16 pixels at a time. The neon intrinsic vld3q de-interleaves the 8 bit RGB values in the source img into three vectors, each with 16 elements. We apply the weight by bit-shifting so we can minimize the number of bits needed to represent the rgb values and maximize the number of pixels we can process at a time. A bit shift to the right is equivalent to multiplying by ½ which is approximately 0.587 and is significantly faster operation than float multiplication.

Original

```
void sobelCalc(Mat& img_gray, Mat& img_sobel_out)
{
  Mat img_outx = img_gray.clone();
  Mat img_outy = img_gray.clone();

  // Apply Sobel filter to black & white image
  unsigned short sobel;

  // Calculate the x convolution
  for (int i=1; i<img_gray.rows; i++) {
    for (int j=1; j<img_gray.cols; j++) {
      sobel = abs(img_gray.data[IMG_WIDTH*(i-1) + (j-1)] -
          img_gray.data[IMG_WIDTH*(i+1) + (j-1)] +
          2*img_gray.data[IMG_WIDTH*(i-1) + (j)] -
          2*img_gray.data[IMG_WIDTH*(i+1) + (j)] +
          img_gray.data[IMG_WIDTH*(i-1) + (j+1)] -
          img_gray.data[IMG_WIDTH*(i+1) + (j+1)]);

      sobel = (sobel > 255) ? 255 : sobel;
      img_outx.data[IMG_WIDTH*(i) + (j)] = sobel;
    }
  }

  // Calc the y convolution
  for (int i=1; i<img_gray.rows; i++) {
    for (int j=1; j<img_gray.cols; j++) {
     sobel = abs(img_gray.data[IMG_WIDTH*(i-1) + (j-1)] -
           img_gray.data[IMG_WIDTH*(i-1) + (j+1)] +
           2*img_gray.data[IMG_WIDTH*(i) + (j-1)] -
           2*img_gray.data[IMG_WIDTH*(i) + (j+1)] +
           img_gray.data[IMG_WIDTH*(i+1) + (j-1)] -
           img_gray.data[IMG_WIDTH*(i+1) + (j+1)]);

     sobel = (sobel > 255) ? 255 : sobel;

     img_outy.data[IMG_WIDTH*(i) + j] = sobel;
    }
  }

  // Combine the two convolutions into the output image
  for (int i=1; i<img_gray.rows; i++) {
    for (int j=1; j<img_gray.cols; j++) {
      sobel = img_outx.data[IMG_WIDTH*(i) + j] +
    img_outy.data[IMG_WIDTH*(i) + j];
      sobel = (sobel > 255) ? 255 : sobel;
      img_sobel_out.data[IMG_WIDTH*(i) + j] = sobel;
    }
  }
}
```

Optimized

```
void sobelCalc(Mat& img_gray, Mat& img_sobel_out)
{
  unsigned short sobel;

  // Calculate both convolutions
  for (int i=1; i<img_gray.rows-1; i++) {
    for (int j=1; j<img_gray.cols; j++) {
      sobel = abs((
      img_gray.data[IMG_WIDTH*(i-1) + (j-1)] +
      img_gray.data[IMG_WIDTH*(i-1) + (j)] -

      img_gray.data[IMG_WIDTH*(i+1) + (j)] -
      img_gray.data[IMG_WIDTH*(i+1) + (j+1)] +

      img_gray.data[IMG_WIDTH*(i) + (j-1)] -
      img_gray.data[IMG_WIDTH*(i) + (j+1)]) << 1 );

      sobel = (sobel > 255) ? 255 : sobel;
      img_sobel_out.data[IMG_WIDTH*(i) + (j)] = sobel;
    }
  }
}
```

Inefficient operations:

1. 3 double for-loops when you only need one
2. Repetitive code
3. Separately calculating x and y convolution
4. Making two Mats
5. Repeatedly accessing + writing to img_gray

Instead of calculating the x and y convolution separately, writing the data in separate structs, then adding them together and writing that to output, we did that all at once. In our single double for-loop, we calculate the sum of the x and y convolution of the image, check once that no pixels are over 255 (max value stored in 8 bits), and write it to output. There is significantly less multiplication, repetitive code and loading/writing data in our version as well.

Part 2: Multi Thread Case

Original

```
while (1) {
  // Allocate memory to hold grayscale and
  sobel images
  img_gray = Mat(IMG_HEIGHT, IMG_WIDTH,
  CV_8UC1);
  img_sobel = Mat(IMG_HEIGHT, IMG_WIDTH,
  CV_8UC1);

  pc_start(&perf_counters);
  src = cvQueryFrame(video_cap);
  pc_stop(&perf_counters);

  cap_time = perf_counters.cycles.count;
  sobel_l1cm = perf_counters.l1_misses.count;
  sobel_ic = perf_counters.ic.count;

  // LAB 2, PART 2: Start parallel section
  pc_start(&perf_counters);
  grayScale(src, img_gray);
  pc_stop(&perf_counters);

  gray_time = perf_counters.cycles.count;
  sobel_l1cm += perf_counters.l1_misses.count;
  sobel_ic += perf_counters.ic.count;

  pc_start(&perf_counters);
  sobelCalc(img_gray, img_sobel);
  pc_stop(&perf_counters);

  sobel_time = perf_counters.cycles.count;
  sobel_l1cm += perf_counters.l1_misses.count;
  sobel_ic += perf_counters.ic.count;
  // LAB 2, PART 2: End parallel section
```

Optimized ->

```
while (1) {
if(thread0_id == myID){
  // Allocate memory to hold grayscale and sobel images
  img_gray = Mat(IMG_HEIGHT, IMG_WIDTH, CV_8UC1);
  img_sobel = Mat(IMG_HEIGHT, IMG_WIDTH, CV_8UC1);

  pc_start(&perf_counters);
  src = cvQueryFrame(video_cap);
  pc_stop(&perf_counters);

  cap_time = perf_counters.cycles.count;
  sobel_l1cm = perf_counters.l1_misses.count;
  sobel_ic = perf_counters.ic.count;
}
pthread_barrier_wait(&while_barrier);

  // set img up for different threads
  if (thread0_id == myID) {
  adj_src = src(Range(0, IMG_HEIGHT/2), Range::all());
   // top half
  adj_gray = img_gray(Range(0, IMG_HEIGHT/2), Range::
  all());
  } else {
  adj_src = src(Range(IMG_HEIGHT/2, IMG_HEIGHT), Range
  ::all()); // bottom half
  adj_gray = img_gray(Range(IMG_HEIGHT/2, IMG_HEIGHT),
   Range::all());
  }
  pthread_barrier_wait(&pregray_barrier);

  // LAB 2, PART 2: Start parallel section
  /* grayScale call */

if(thread0_id == myID){
  pc_start(&perf_counters);
}
  grayScale(adj_src, adj_gray);
  pthread_barrier_wait(&gray_barrier); //
  gray_barrier = barrier for grayScale

if(thread0_id == myID){
  pc_stop(&perf_counters);

  // update counters with grayScale performance
  gray_time = perf_counters.cycles.count;
  sobel_l1cm += perf_counters.l1_misses.count;
  sobel_ic += perf_counters.ic.count;
}
```

We were provided with a version of sobel_mt.cpp that launched two threads, killed one of them instantly and processed the image with just one thread. In our implementation, we use both threads to process each frame of the video - thread0 processes the top half of the image and thread1 processes the bottom half. We also ensure that only one thread (thread0) does things like interact with the performance counters, captures the source image, and writes out to the excel sheet. We also added barriers periodically to ensure the threads move in parallel.

Results:

Single thread optimized results:

```
$ ./sobel -n 50
$ cat st_perf.csv
Percent of time per function
Capture, 28.9443%
Grayscale, 11.2574%
Sobel, 45.5905%
Display, 14.2079%

Summary
Frames per second, 53.6481
Cycles per frame, 1.65098e+07
Energy per frames (mJ), 26.096
Total frames, 50

Hardware Stats (Cap + Gray + Sobel + Display)
Instructions per cycle, 0.917514
L1 misses per frame, 122093
L1 misses per instruction, 0.00813071
Instruction count per frame, 1.50162e+07
```

Multi thread optimized results:

```
$ ./sobel -m -n 50
$ cat mt_perf.csv
Percent of time per function
Capture, 39.8283%
Grayscale, 8.24243%
Sobel, 31.9872%
Display, 19.9421%

Summary
Frames per second, 75.5468
Cycles per frame, 1.1865e+07
Energy per frames (mJ), 18.5315
Total frames, 50

Hardware Stats (Cap + Gray + Sobel + Display)
Instructions per cycle, 0.892471
L1 misses per frame, 94669.4
L1 misses per instruction, 0.00905673
Instruction count per frame, 1.04529e+07
$
```

Lessons Learned/Epilogue:

1. General methods for optimization
    a. Loop unrolling
    b. Strength reduction
    c. Condense repetitive code
2. Multithreading
3. Vectorization