AAHLS Final Report

The Hardware Acceleration of Monocular Visual Odometry

Team Members: 張惇宥 朱怡蓁 許家維 Advisor: Prof. 賴瑾

■ Github: Chang-Tun-Yu/Mono-Visual-Odometry-by-HLS (github.com)

■ Introduction

Odometry is the device that estimates the change in position over time by using the data from the motion sensor. Instead of depending on the motion sensor, visual odometry (VO) determines the position and orientation by analyzing sequential camera images. According to different camera settings, VO can be categorized as Monocular VO using a single camera and Stereo VO using dual cameras. This technique has been widely used in a variety of robotic applications, and whether the odometry can provide reliable data in real-time is an important factor that may affect the success of the overall system.

In this project, we focus on Monocular VO, and our target is to use high-level synthesis to port the visual odometry algorithm to the FPGA platform, hoping to effectively shorten the computing time and enhance the frame rate of the system.

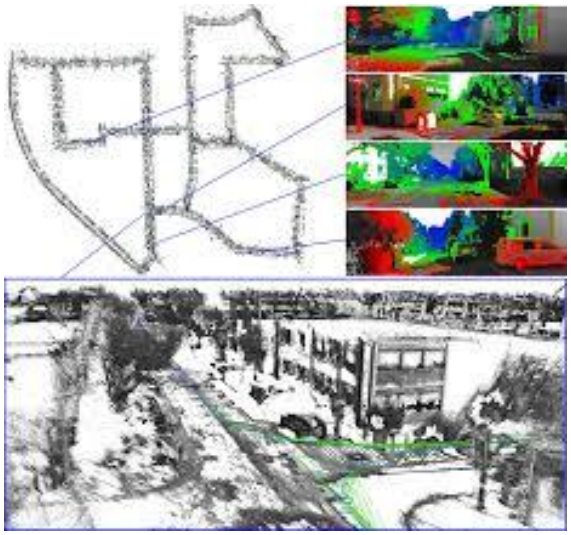

Fig. 1. An Example of Visual Odometry.

■ Monocular Visual Odometry Algorithm

1. **Input:** Input image sequence.
2. **Feature Detection:** Filter input image with 5 x 5 Blob, Corner, and Sobel masks and employ non-maximum-suppression on the filtered images to obtain feature candidates, the illustration is shown in the following figure.

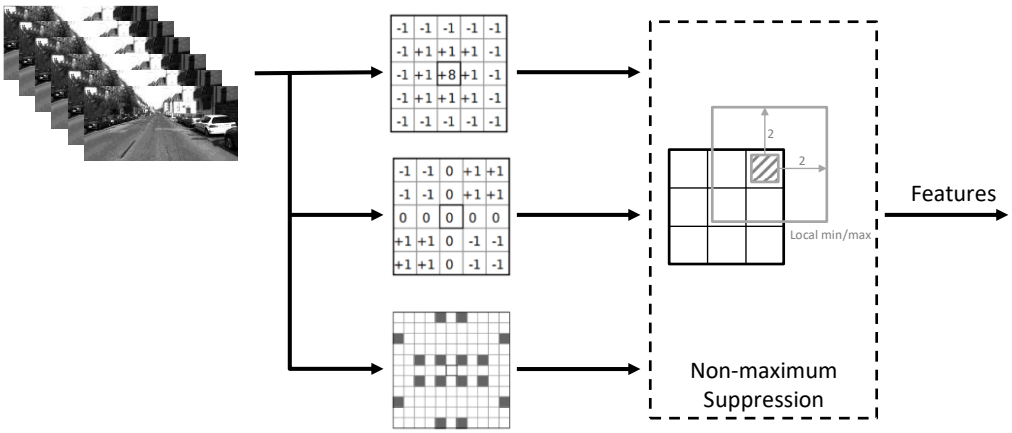

Fig. 2. The process of finding feature candidates.

3. **Feature Matching:** Adopt local search and circular match methods to eliminate sporadic outliers, thereby reducing unnecessary computations. Fig. 3 illustrates the circular match method diagrammatically.

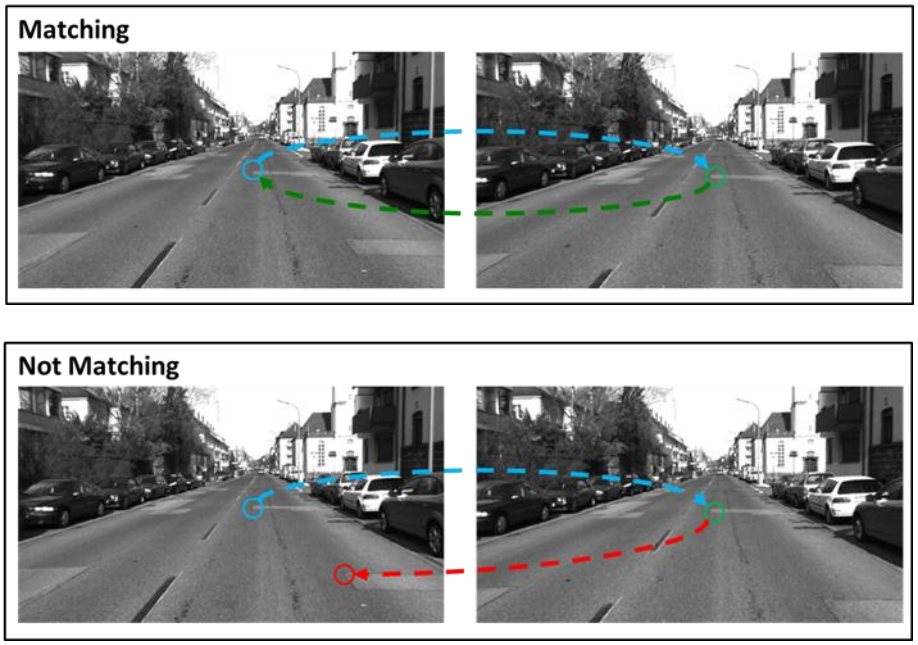

Fig. 3. The illustration of the circular match method.

4. **Bucket Strategy**: Use the bucket strategy, which is illustrated in Fig. 4, to reduce the number of features and make the features distributed in the image domain more evenly.

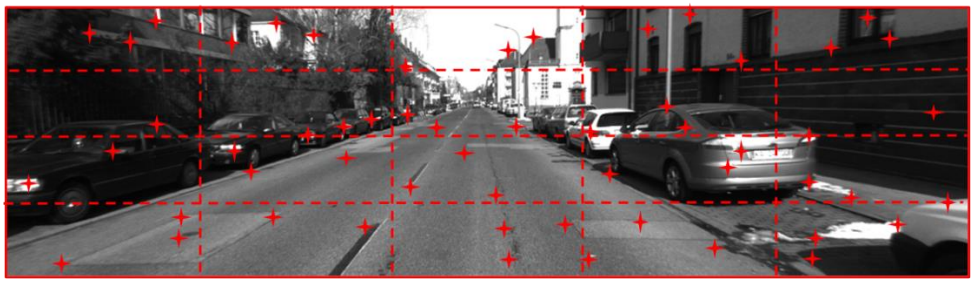

Fig. 4. The illustration of the bucketing strategy:
The entire image space is split into several buckets of the same size, and the maximum number of features of each bucket is limited.

5. **Compute the essential matrix**: Use the 8-point algorithm with RANSAC to compute the essential matrix

6. **Output:** Estimate ***R*** and ***t*** using the essential matrix obtained in the previous step.

■ Source Code

Our project is based on LIBVISO2 (Library for Visual Odometry 2) [1], a cross-platform C++ library with MATLAB wrappers.

■ Dataset

Our project use Karlsruhe Dataset: Stereo Video Sequences + rough GPS Poses [2] to implement visual odometry.

■ Profiling

In order to better elaborate the overall architecture of this project, we used “gprof” to explore the function hierarchy and estimate the runtime of each function, as shown in Fig. 5.

With Fig. 5, it is not hard to observe that the main function calls a total of four sub-functions during runtime, namely myMatching, myComputeFeature, myRemoveOuliers and updateMotion. Among them, myMatching takes the most computing time, accounting for 95% of the total runtime, while updateMotion is the least, only taking 0.8%. Considering that updateMotion is obviously not a bottleneck in computing time and is the last stage of the execution, we decided to keep it running

on the CPU, and port the other three onto the FPGA platform.

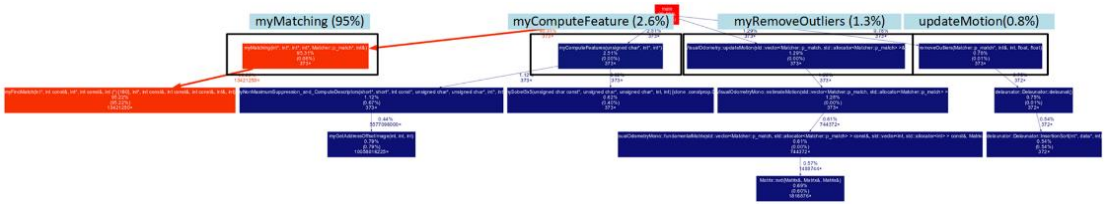

Fig. 5. Function hierarchy and runtime estimation.

■ 1st Kernel: Compute Feature

Kernel Design

Data is arranged into bucket in compute feature kernel rather than matching kernel to lower the burden of the latter kernel. In this way, we can strike a great balance between different kernels. Originally, we plan to use *#pragma DATAFLOW* to design compute feature kernel because it contains data rearrangement instead of lots of computation, as shown in Fig. 6. However, it is not the bottleneck and we encounter some memory issue. Thus, we just modify the C++ code to be synthesizable as shown in Fig. 7.

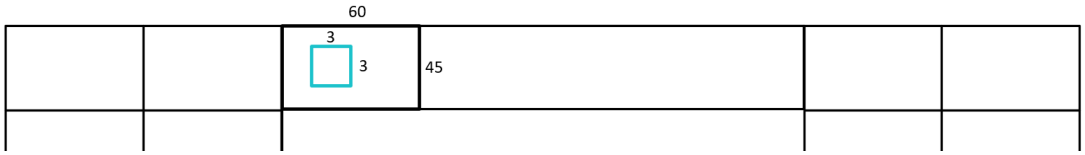

Fig. 6. Arranging data in compute feature.

```
void  myCheckerboard5x5 ( const uint8_t in[IMG_SIZE], int16_t out[IMG_SIZE], int w, int h ) {
    const int16_t weight[5][5] = {
        {-1, -1, 0, 1, 1},
        {-1, -1, 0, 1, 1},
        {0, 0, 0, 0, 0},
        {1, 1, 0, -1, -1},
        {1, 1, 0, -1, -1}
    };
    for (int y = 2; y < h-2; y++) {
        for (int x = 2; x < w-2; x++) {
            int16_t tmp = 0;
            for (int ky = -2; ky <= 2; ky++) {
                for (int kx = -2; kx <=2 ; kx++) {
                    tmp += in[myGetAddressOffsetImage(x+kx, y+ky, w)] * weight[ky+2][kx+2];
                }
            }
            out[myGetAddressOffsetImage(x, y, w)] = tmp;
        }
    }
}
```

Fig. 7. Implemented C++ code.

Future Work

The memory access is not efficient in current design. There is no burst access, and there is also no parallelism, either. If the matching kernel becomes faster, compute feature kernel will become bottleneck. Thus, we plan to use window buffer to balance both the performance and memory usage. As illustrated in Fig. 8, if we implement with both window and line buffer, data can be further fully reused.

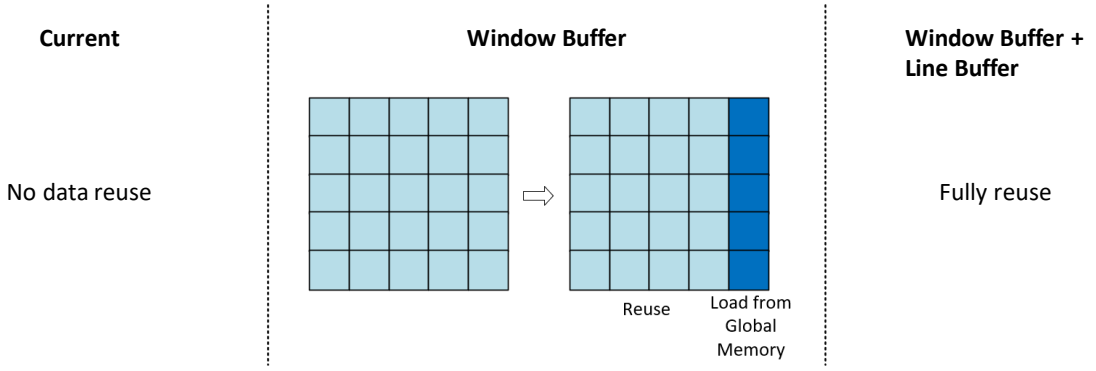

Fig. 8. Data reuse in compute feature.

■ 2nd Kernel: Matching

Kernel Design

1. Data-level parallelism

We decide to unroll the loop to accelerate the process of finding matching. At first, we just use HLS command: #pragma unroll as shown in Fig. 9, while it didn't work. We thought that it's because *d* is a data array in structure *origin*, and the tool cannot directly recognize it. We have no choice but to implement a brute-force adder tree. Finally, it reaches pipeline with II = 1.

Improved

```
find_col: for (int col_idx = m_buffer_num[u_bin_buffer][bin_class][v_bin_min]; col_idx < m_buffer_num[u_bin_buffer][bin_class][v_bin_max]; col_idx++) {
  //#pragma HLS UNROLL factor=parallel
  Feature_Point target = m_buffer[u_bin_buffer][bin_class][col_idx];
  if (target.u>=u_min && target.u<=u_max && target.v>=v_min && target.v<=v_max) {
    psum = 0;
    calc: for (int i = 0; i < 32; i++) {
      #pragma HLS UNROLL  factor=32
      ap_uint<8> a = origin.d.range((i+1)*8-1, 8*i);
      ap_uint<8> b = target.d.range((i+1)*8-1, 8*i);
      tmp[i] = ABS(a, b);
    }
    // adder tree
    psum_03 = tmp[0] + tmp[1] + tmp[2] + tmp[3];
    psum_47 = tmp[4] + tmp[5] + tmp[6] + tmp[7];
    psum_811 = tmp[8] + tmp[9] + tmp[10] + tmp[11];
    psum_1215 = tmp[12] + tmp[13] + tmp[14] + tmp[15];
    psum_1619 = tmp[16] + tmp[17] + tmp[18] + tmp[19];
    psum_2023 = tmp[20] + tmp[21] + tmp[22] + tmp[23];
    psum_2427 = tmp[24] + tmp[25] + tmp[26] + tmp[27];
    psum_2831 = tmp[28] + tmp[29] + tmp[30] + tmp[31];

    psum_015 = psum_03 + psum_47 + psum_811 + psum_1215;
    psum_1631 = psum_1619 + psum_2023 + psum_2427 + psum_2831;

    psum = psum_015 + psum_1631;
```

Original

```
calc: for (int i = 0; i < 32; i++) {
  #pragma HLS UNROLL  factor=32
  psum += (int16_t) ABS(origin.d[i], target.d[i]);
}
```

Fig. 9. Data-level parallelism in matching.

2. Data buffer and fetch strategies

Assume we don't buffer any feature point, and the size of each feature point is 48 bytes with 16 bytes coordinate and class and 32 bytes of characteristics. There are 20000 feature points in each frame, and 1500 points are searched in average on previous frame for each point. We can calculate **DRAM traffic of each frame** as follows:

$$20000 \times 1500 \times 48 \times 2 = 2.8GB$$

There is an int32 port to access data from DRAM. We can calculate **time to access data** as follows:

$$\frac{1.4G}{4 \times 300M} = 2.34\ s$$

Assume that there is a buffer, so each data is only accessed once. The **DRAM traffic of each frame** is:

$$20000 \times 48 \times 2 = 1.92MB$$

And the **time** it takes **to access data** is lowered to:

$$\frac{1.92M}{4 \times 300M} = 1.6\ ms$$

In this way, we cut down on the time to access data by **1462.5x**. Therefore, we design a buffer that can contains 7 columns of bins as shown in Fig. 10. Besides, the bins in each column are aggregated to flatten a loop in finding match.

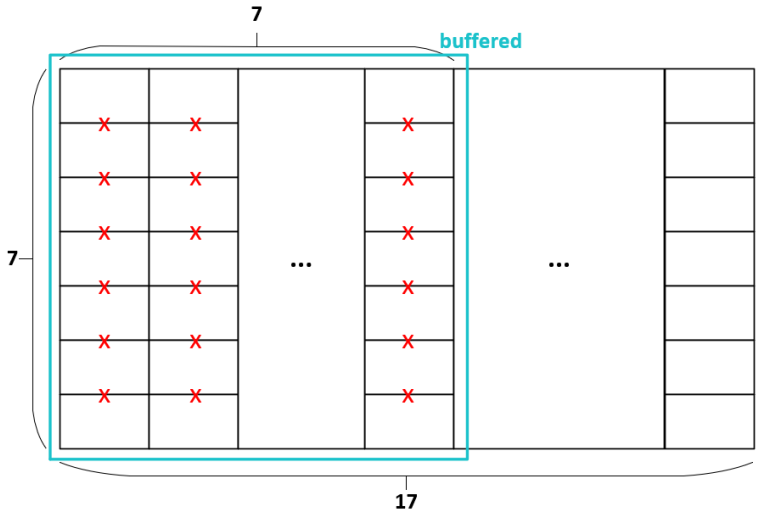

Fig. 10. Data buffer in matching kernel.

We try to analyze this problem by roofline model. Fig. 11 shows the different configuration of performance and bandwidth. We assume the peak performance of matching kernel, which is depended on the FPGA resource, is 300 GOps/sec. The peak off-chip global memory total bandwidth is 256 Gbyte/sec. However, the peak performance and the peak bandwidth decrease to 9.6 GOps/sec and 1.2 Gbyte/sec, respectfully. This is because the resource and bandwidth are not fully-utilized.

Fig. 11 also shows the two different kinds of kernel design. The first one is there is no any on-chip buffer, so there is no data reuse in this kernel. The **operational intensity** of this kernel is:

$$\frac{20000 \times 1500 \times 96 \times 2}{20000 \times 1500 \times 48 \times 2} = 2$$

The second one it there are many data buffer as shown in Fig. 10. The data can be fully reuse. The **operational intensity** of this kernel is:

$$\frac{20000 \times 1500 \times 96 \times 2}{20000 \times 48 \times 2} = 3000$$

By the analysis of roofline model, we can conclude the buffering strategy we use now is too aggressive (over-design). In current design (both resource and bandwidth are under-utilized), operational intensity is enough if it is larger than 6 Ops/byte. Even we modify our design to fully-utilized resource and bandwidth in the future. The operational intensity is still enough because it is large than 1.17 Ops/byte.

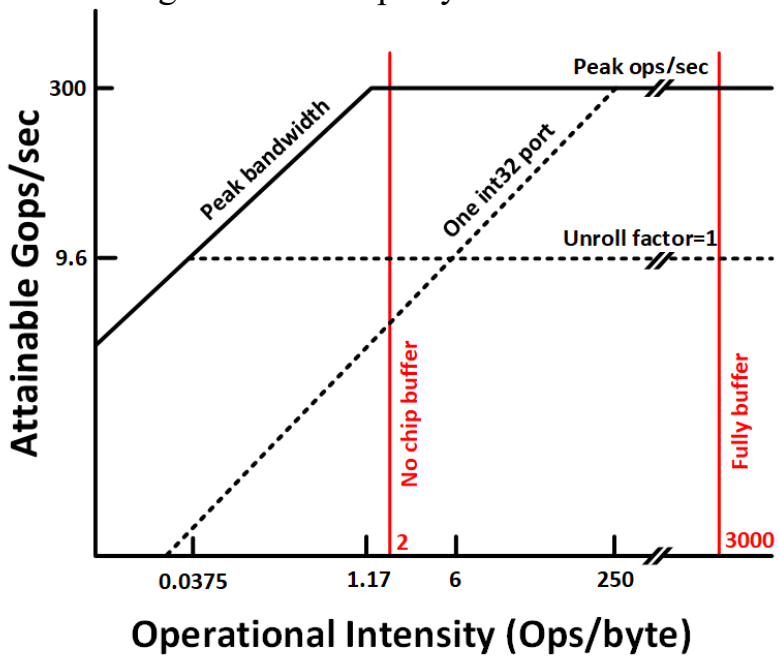

Fig. 11. Roofline analysis of matching kernel.

Fig. 12 shows the fetching strategies for data buffer. Here we use *#pragma PIPELINE II=1* explicitly in for loop to enable burst transfer.

original

```
fetch_perv: for (int k=0; k < n1p[_bin]; k++) {
  Feature_Point fp;
  fp.u = m1p[_bin*MAX2_BIN_OFFSET+12*k+0];
  fp.v = m1p[_bin*MAX2_BIN_OFFSET+12*k+1];
  fp.val = m1p[_bin*MAX2_BIN_OFFSET+12*k+2];
  fp.type = m1p[_bin*MAX2_BIN_OFFSET+12*k+3];
  // memcpy(fp.d, m1p+_bin*MAX2_BIN_OFFSET+12*k+4, 32);
  fp.d.range(31, 0) = m1p[_bin*MAX2_BIN_OFFSET+12*k+4];
  fp.d.range(63, 32) = m1p[_bin*MAX2_BIN_OFFSET+12*k+5];
  fp.d.range(95, 64) = m1p[_bin*MAX2_BIN_OFFSET+12*k+6];
  fp.d.range(127, 96) = m1p[_bin*MAX2_BIN_OFFSET+12*k+7];
  fp.d.range(159, 128) = m1p[_bin*MAX2_BIN_OFFSET+12*k+8];
  fp.d.range(191, 160) = m1p[_bin*MAX2_BIN_OFFSET+12*k+9];
  fp.d.range(223, 192) = m1p[_bin*MAX2_BIN_OFFSET+12*k+10];
  fp.d.range(255, 224) = m1p[_bin*MAX2_BIN_OFFSET+12*k+11];
  mp_buffer[c][mp_buffer_num[c][v_bin]+k] = fp;
}
```

improved

```
fetch_perv: for (int k=0; k < n1p[_bin]; k++) {
  Feature_Point fp;
  for (int ii=0; ii<12; ii++) {
    #pragma HLS PIPELINE II=1
    int32_t mp_data = m1p[_bin*MAX2_BIN_OFFSET+12*k+ii];
    if (ii==0) {
      fp.u = mp_data;
    }
    if (ii==1) {
      fp.v = mp_data;
    }
    if (ii==2) {
      fp.val = mp_data;
    }
    if (ii==3) {
      fp.type = mp_data;
    }
    if (ii>=4) {
      fp.d.range(32*(ii-3)-1, 32*(ii-4)) = mp_data;
    }
  }
  mp_buffer[c][mp_buffer_num[c][v_bin]+k] = fp;
}
```

Fig. 12. Fetching strategies in matching kernel.

3. Reduce BRAM usage

We use lots of data buffer to reduce time to access data from DRAM repeatedly, and it will use up BRAM resource. To solve this issue, we use *#pragma HLS bind_storage* to map some data buffers to Ultra Ram as illustrated in Fig. 13. Finally, we lower the BRAM usage to 433 as shown in Fig. 14.

```
Feature_Point mc_buffer[7][4][COL_BIN_FEATURE_MAX];
int32_t       mc_buffer_num[7][4][V_BIN_NUM+1];
Feature_Point mp_buffer[7][4][COL_BIN_FEATURE_MAX];
int32_t       mp_buffer_num[7][4][V_BIN_NUM+1];
int32_t       col_matching_cnt[U_BIN_NUM][4];
Matching_cand mc_matching[U_BIN_NUM][4][COL_BIN_FEATURE_MAX];
Matching_cand mp_matching[U_BIN_NUM][4][COL_BIN_FEATURE_MAX];
static int _p_matched_num;
#pragma HLS bind_storage variable=mc_buffer type=RAM_1P impl=uram
#pragma HLS bind_storage variable=mp_buffer type=RAM_1P impl=uram
#pragma HLS bind_storage variable=mc_matching type=RAM_1P impl=uram
#pragma HLS bind_storage variable=mp_matching type=RAM_1P impl=uram
```

Fig. 13. The use of Ultra Ram.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF
myMatching										no	433	15	12392
myMatching_Pipeline_VITIS_LOOP_241_1_VITIS_LOOP_242_2					145410	7.270E5		145410		no	0	0	41
fetch_col_bin_1								-1		no	0	10	3208
VITIS_LOOP_28_1					4	20.000	1		4	no			
fetch_v_bin_fetch_class					?	?	?		28	no			
fetch_perv					?	?	57			no			
fetch_col_bin_1_Pipeline_VITIS_LOOP_40_2					15	75.000		15		no	0	0	446
VITIS_LOOP_40_2					13	65.000	3	1	12	yes			
fetch_cur					?	?	57			no			
iter									17	no			
iter_class									4	no			
cur_iter_col										no			
find_match										no	0	2	2481
find_u_bin										no			
find_match_Pipeline_find_col										no	0	0	1436
find_col							5	1		yes			

Fig. 14. Synthesis report for matching.

Future Work

1. Data-level parallelism in finding match

First of all, we can unroll the find match loop to increase the number of parallelisms. At the same time, set *#pragma HLS ARRAY_RESHAPE* on m_buffer to get parallelism.

```
find_col: for (int col_idx = m_buffer_num[u_bin_buffer][bin_class][v_bin_min]; col_idx < m_buffer_num[u_bin_buffer][bin_class][v_bin_max]; col_idx++)
  #pragma HLS UNROLL factor=parallel
  Feature_Point target = m_buffer[u_bin_buffer][bin_class][col_idx];
  if (target.u>=u_min && target.u<=u_max && target.v>=v_min && target.v<=v_max) {
    psum = 0;
    calc: for (int i = 0; i < 32; i++) {
      #pragma HLS UNROLL  factor=32
      ap_uint<8> a = origin.d.range((i+1)*8-1, 8*i);
      ap_uint<8> b = target.d.range((i+1)*8-1, 8*i);
      tmp[i] = ABS(a, b);
    }
    // adder tree
```

Fig. 15. Future work for data-level parallelism.

2. Data buffer and fetching strategies

In the previous discussion, we assume there are only one int32 port. If we use multiple port and leverage automatic port width resizing, the HBM bandwidth can be fully utilized. For example, the total bandwidth of 8 ports of int32 (auto resize to 512 bits) is:

$$8 \times (\frac{512}{8}) \times 300MB/s = 153GB/s$$

It almost reaches total HBM bandwidth ($201GB/s$). In this way, we can make great use of HBM.

```
  read_a:
    for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
      result[x] = a[i * N + x];
    }
```

```
[connectivity]
sp=krnl.mp_0:HBM[0:2]
sp=krnl.mp_1:HBM[3:5]
sp=krnl.mp_2:HBM[6:8]
...
```

Connect 3 pseudo channels to each port

Fig. 16. HBM example [3].

■ 3rd Kernel: Remove Outliers

Kernel Design

Here we implement Delaunay Triangulation to remove outliers. Delaunay Triangulation is an algorithm with point coordinates as inputs and a series of triangles as outputs. We adopt the sweep circle algorithm [4] with *O(NlogN)* complexity. First of all, it finds the first triangle with the centroid *O* inside. Then it iterates all the points in the order of distance to *O*. In the process, the projection of next point hits the frontiers and a new

triangle is formulated as shown in Fig. 17 (a). After that, walking to the left side starts Fig. 17 (b), walking on the left follower site Fig. 17 (c), end of walking Fig. 17 (d).

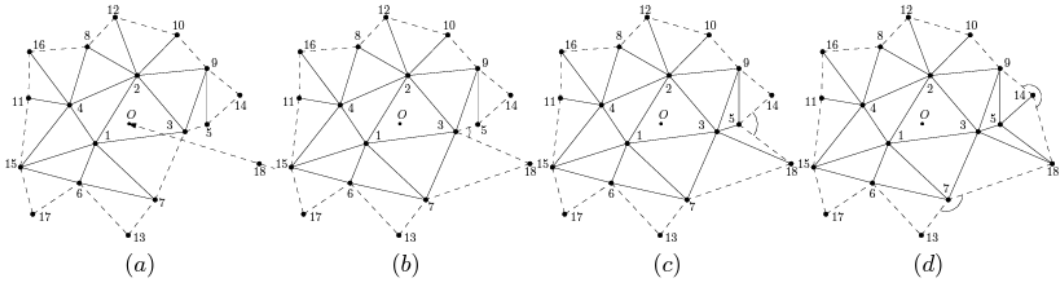

Fig. 17. Delaunay Triangulation.

After finishing Delaunay Triangulation, we can use triangles with previous and current points to calculate support value. In each triangle, we can calculate the movement vector as shown in red arrows in Fig. 18. Any two movement vectors for each triangle is subtracted and then compared with the threshold. If the value is small enough, which means points are reliable, its support value is added by one. Finally, the outliers are eliminated with low support value.

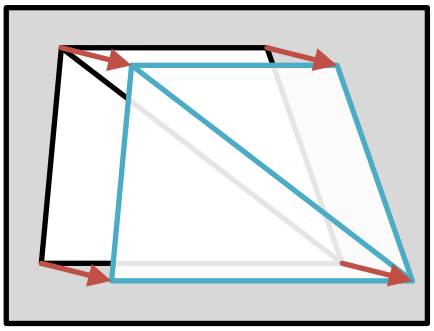

Fig. 18. Method of removing outliers.

Because this kernel is not bottleneck, only storage optimization is performed here. During Delaunay Triangulation, lots of memory are used to buffer edges, triangles, and neighbors. We implement C++ union here to reuse data array. For example, one float is 4 bytes and can be transformed into two int16. Besides, we change all the doubles into floats without performance loss. Finally, the BRAM usage is lowered by **12%**.

```
typedef float data_type;
union data{
    struct {
        int16_t prev;
        int16_t next;
    } hull;
    data_type f;

};
```

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1626	-
FIFO	-	-	-	-	-
Instance	8	406	145432	134256	0
Memory	560	-	96	163	0
Multiplexer	-	-	-	2653	-
Register	-	-	588	-	-
Total	568	406	146116	138698	0
Available	2688	5952	1743360	871680	640
Available SLR	1344	2976	871680	435840	320
Utilization (%)	21	6	8	15	0
Utilization SLR (%)	42	13	16	31	0

Fig. 19. Data reuse and kernel resource utilization.

Future Work

In the Delaunay Triangulation, we found that it takes long time for sorting all the points in the distance to centroids. We can implement odd even transposition sort, which is a parallel sorting method as shown in Fig. 20 to accelerate sorting process.

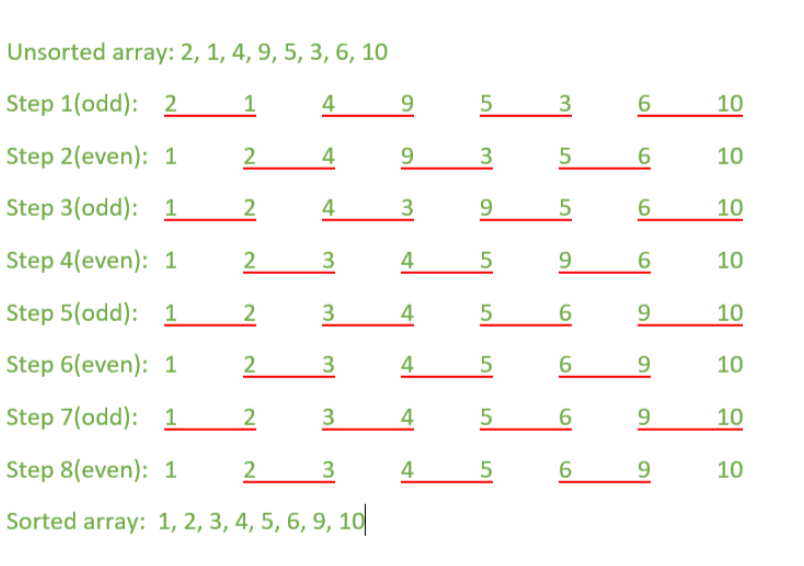

Fig. 20. Odd even transposition sort.

Delaunay Triangulation finds one triangle then find next. The only way we can perform parallelism is divide and conquer. By dividing all the points into different zones, we can do Delaunay Triangulation simultaneously as shown in Fig. 21.

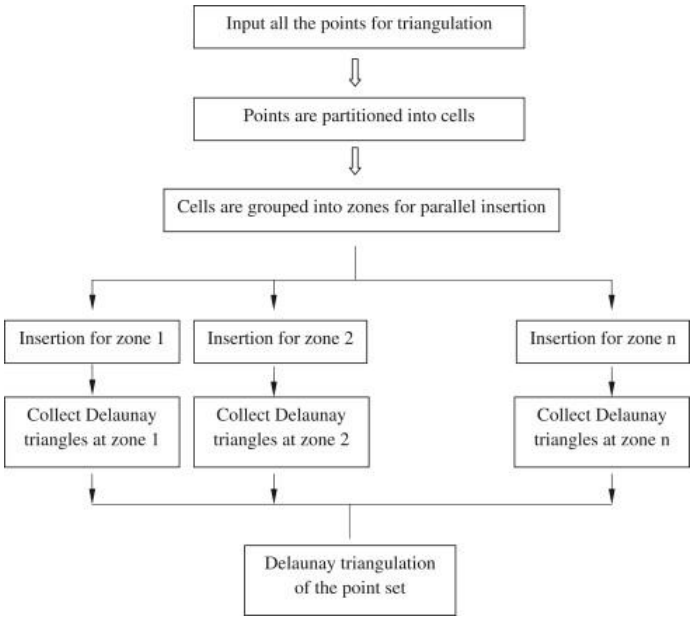

Fig. 21. Parallel Delaunay Triangulation [5].

■ System Architecture

We make use of OpenCL events in our design. As illustrated in Fig. 22, there are double buffers between two continuous kernels. At first, data for an image is moved from host to global memory. The first kernel performs computing features and outputs results to current buffer. Both current and previous features are used in matching kernel. Once the matching is finished, move the current features to previous one. After that, remove outliers and transfer the result to host. Because the process of updating motions is fast, this step is performed on CPU after getting the p_match data.

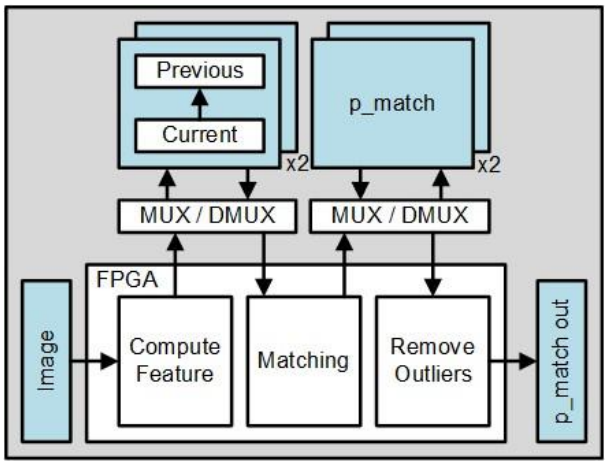

Fig. 22. System Architecture.

We take Xilinx tutorial as reference to implement double buffer. With double buffers, three pipeline kernels can be fully utilized. The pseudo code for OpenCL double buffer is shown in Fig. 23. If the buffer is already in used, it needs to wait until event finishes, when we perform motion updating at the same time. List of events for this work is shown in Fig. 24.

```
Cl::Buffer DB[2]
for i = 0...numImg-1 do
   flag ← i%2
   if i >=2 then
      wait until finish migrating i-2th result to host
   endif
   DB[flag] ← create buffer by using host pointer
   set kernel arguments
   add events into queue with dependencies
endfor
```

Fig. 23. Pseudo code for OpenCL double buffer [6].

```
//1. move img to global memory
std::vector<cl::Event> eventList{events_compute[i-2]};
q.enqueueMigrateMemObjects({db_Img[flag]}, 0, &eventList, &events_write_Img[i]);

//2. copy previous feature
std::vector<cl::Event> eventList2{events_compute[i - 1]};
q.enqueueCopyBuffer(db_maxc[!flag], db_maxp[flag], 0, 0, sizeof(int32_t) * MAX_FEATURE_ARRAY_SIZE, &eventList2, &events_write_move[i]);
std::vector<cl::Event> eventList3{events_compute[i - 1]};
q.enqueueCopyBuffer(db_maxc_num[!flag], db_maxp_num[flag], 0, 0, sizeof(int32_t) * BIN_NUM, &eventList3, &events_write_move2[i]);

//3. compute feature
std::vector<cl::Event> eventList4 {events_write_Img[i], events_write_move[i], events_write_move2[i], events_init_DB[i]};
q.enqueueTask(myComputeFeatures, &eventList4, &events_compute[i]);

//4. matching
std::vector<cl::Event> eventList5{events_compute[i], events_match[i-1]};
q.enqueueTask(myMatching, &eventList5, &events_match[i]);

//5. removeOutliers
std::vector<cl::Event> eventList6{events_match[i], events_read_res[i-1]};
q.enqueueTask(removeOutliers, &eventList6, &events_remove[i]);

//6. copy output to host memory
std::vector<cl::Event> eventList7{events_remove[i], events_read_res[i - 1]};
q.enqueueMigrateMemObjects({db_p_matched_res[flag], db_p_matched_res_cnt[flag]}, CL_MIGRATE_MEM_OBJECT_HOST, &eventList7, &events_read_res[i]);
```

Fig. 24. List of events.

■ Performance Evaluation

The timeline trace is shown in Fig. 25. We can find that three kernels are well-balanced with almost same execution time. In Fig. 26, the final camera positions are printed to check the correctness. Besides, kernel execution time is measured. The processing rate is calculated as follows:

$$\frac{1}{384m} = 2.6 \text{ frame/second}$$

Compared to intel i7 CPU with 0.59 frame/second, this work accelerates by **4.4x**.

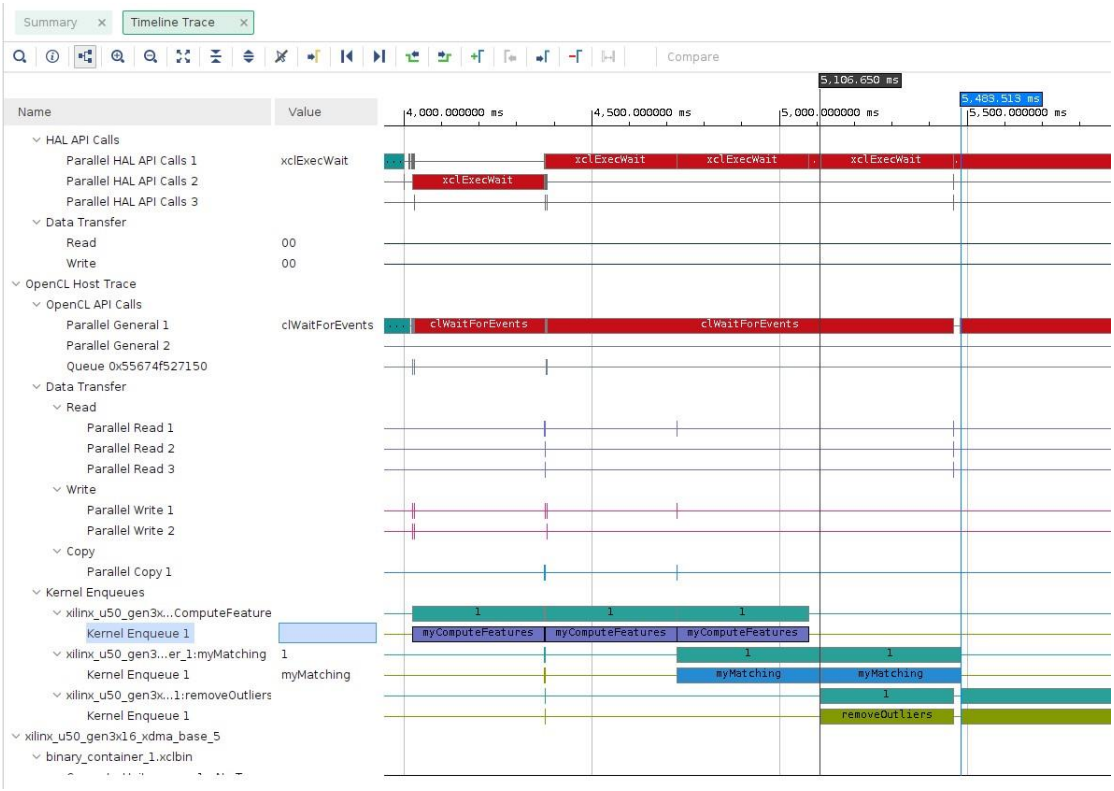

Fig. 25. Timeline trace.

```
finish get_matches 1: 207
207 86.4734
   0.9999989    0.0012017    0.0009044    0.0419179
  -0.0012003    0.9999981   -0.0015428   -0.1387353
  -0.0009063    0.0015417    0.9999984    1.4398840
   0.0000000    0.0000000    0.0000000    1.0000000

Processing: Frame: 2
[matching ]: Completed kernel
[removeOutliers ]: Completed kernel
[cpy output ]: Completed buffer migrate
finish get_matches 2: 170
170 79.4118
   0.9999894    0.0030650    0.0034318    0.0735500
  -0.0030620    0.9999949   -0.0008790   -0.3118980
  -0.0034345    0.0008685    0.9999937    3.0916182
   0.0000000    0.0000000    0.0000000    1.0000000

INFO: Finish kernel execution
INFO: Finish E2E execution
------------------------------------------------------
INFO: init DB from host to device: 0 us
------------------------------------------------------
INFO: Write Img host to device: 105.799 us
------------------------------------------------------
INFO: cpy previous features: 1284.92 us
------------------------------------------------------
INFO: cpy previous features: 336.093 us
------------------------------------------------------
INFO: Data transfer single result from device to host: 409.763 us
------------------------------------------------------
INFO: Kernel compute1 execution: 350183 us
------------------------------------------------------
INFO: Kernel match1 execution: 384050 us
------------------------------------------------------
INFO: Kernel remove1 execution: 352718 us
------------------------------------------------------
```

Fig. 26. Kernel time analysis.

■ Conclusion

Visual odometry is suitable for HLS optimization and experiments with lots of computation. However, this work is such a big project. Although there is opensource C++ code, we, three people, still make lots of efforts to first understand the algorithm, clean the code (many class object even with inheritance), and final perform HLS space exploration. During these three weeks, we learn how to implement HLS, shorten the kernel development process, and use OpenCL APIs.

■ Reference

[1] Repo: https://www.cvlibs.net/software/libviso/

[2] Dataset: https://www.cvlibs.net/datasets/karlsruhe_sequences/

[3] Xilinx Documentation: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Automatic-Port-Width-Resizing

[4] Ahmad Biniaz et al., "A faster circle-sweep Delaunay triangulation algorithm," Advances in Engineering Software, Pages 1-13, Volume 43, Issue 1, 2012.

[5] S.H.Lo, "Parallel Delaunay triangulation—Application to two dimensions," Finite Elements in Analysis and Design, Volume 62, Pages 37-48, 2012.

[6] Xilinx Tutorial: https://github.com/Xilinx/Vitis_Accel_Examples/blob/master/host/overlap/src/host.cpp