CS270 Course Project Report: Quick Alignment Algorithm for Burst Photography

Rundong Li
36273800
SIST, ShanghaiTech University

***Abstract*—Photography quality of cell phones has received lot of attention. Limited by small radius lens and tiny COMS sensors, digital images captured by mobile phones are easily suffer from noise on low illuminated areas, and unable to capture dynamic range on strongly illuminated areas. One possible solution is taking a burst of frames with different exposure settings, then merging into single high quality image. This rises the challenge of effectively aligning different frames before or after a series of other sophisticated operations. We try to address this challenge in this course project.**

Note: to meet the strict constrains of real-world applications, the proposed algorithm is quite straight forward but implementing it requires lots of engineering efforts. Thus we organize this report more like a technical manual than academic paper, focusing on the engineering decisions and optimization techniques we taken, as well as the algorithm itself.

I. Scope

This technical manual describes the design, implementation and optimization of L1/L2 mixed alignment algorithm based on [**?**]. This manual is organized as following sections:

1) Proposed algorithm and application scenario;
2) Implementation and optimization;
3) Results and discussions;

This implementation is open-sourced at https://github.com/CrazyRundong/burst-align under GPL-v3 License.

II. Proposed Algorithm and Application Scenario

One possible technique to address dark noise and poor dynamic range introduced by optical and semiconductor limitations are generating HDR pictures by burst photography [**?**], [**?**]. To be brief, burst photography means taking several frames with different exposure settings within certain time window, pick one sharpest frame as *reference frame*, then merge corresponding regions among remaining *alternative frames* to perform denosing and dynamic range adjustment (Figure 1). One key challenge is to locate and match these corresponding regions. This locate-and-match procedure is called *alignment*.

Precision-targeted alignment problem is well-studied, solutions include optical flow [**?**], dense corresponding [**?**], and over-segmentation then directly reason about geometry and occlusion [**?**]. In order to acquire high alignment precision, these solutions are usually slow and computation expensive, e.g. the top-5 entries on KITTI optical flow benchmark [**?**] require between 1.7 to 107 minutes to perform complete alignments per M-pixes.

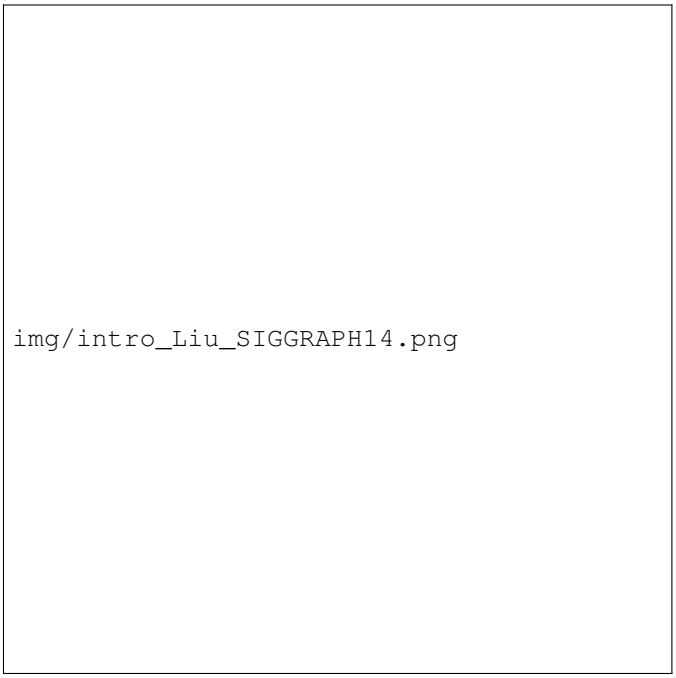

Fig. 1: A burst of frames can be align and merged into single high dynamic range photo. [**?**]

However, majority of application scenarios of burst photography are relevant to mobile devices, e.g. cell phones and drones with small radius lens and tiny COMS sensors. Computation budgets of these devices are quite constrained, alignment algorithms thus should be efficient and fast. We propose a tile-based, coarse to fine alignment algorithm, named *L1/L2 mixed alignment* based on fast L2 alignment proposed in [**?**] and fast cross-correlation proposed in [**?**].

The proposed L1/L2 mixed alignment algorithm is quite straight forward: we firstly split reference frame into *regions*, then search corresponding regions on alternative frames. Once a certain region in an alternative frame is find to match the reference region, the output alignment of this pair of regions is set to their spatial mismatches.

Detailed design decisions of proposed algorithm include:

a) Corresponding regions are rectangle tiles: Instead of performing pixel-wise alignment, our minimum alignment units are rectangle regions called *tiles*. This decision is based on: objects are not likely to have obvious deformation within

short time window, rectangle tiles thus can be served as good matching template;

b) Restricting search regions and introducing image pyramids to reduce computation: Search regions should be large enough to capture moving object among frames, however computation costs is squared w.r.t. search region size. Thus we firstly scale input frames into different spatial sizes, computing alignment on coarsest frames, then search matching tiles on larger frames starting from interpolated previous alignments. Search regions can be restricted thanks to this interpolate and initializing strategy;

c) Tile matchness on coarse levels are measured by L2 distance...: As illustrated in §4.1 of [**?**], computing L2 distances among one fixed tile (T) and a slide tile (I) within a search region can be implemented as one un-normalized box-filter plus a FFT-based matrix multiplication:

$$\begin{aligned} D_2(u,v) &= \sum_{y=0}^{n-1}\sum_{x=0}^{n-1} \|T(x,y) - I(x+u+u_0, y+v+v_0)\|_2^2 \\ &= \|T\|_2^2 + \mathrm{box}(I \circ I, n) - 2\mathcal{F}^{-1}(\mathcal{F}(I) \circ \mathcal{F}(T)) \end{aligned}$$

where D_2 is the L2 distance between reference tile at (u, v) and alternative tile at $(u+u_0, v+v_0)$, T is the reference frame, I is current alternative frame, n the size of tile, and (u_0, v_0) the interpolated alignment (the "start point") from coarser pyramid level.

d) ...and use L1 distance on largest pyramid level: Although L2 distance computation can be optimized in algorithm level, on largest search size the brute force L1 distance computation is still faster than optimized L2 implementation. e.g. on `x86_64` platform, absolute value computing can be accreted by `_mm_abs_pi8` intrinsic [**?**].

After all, the proposed L1/L2 mixed alignment can be formulated as Algorithm 1. Next we describe the details in our implementation and optimization.

Data: reference frame I_{ref}, alternative frames $I_{1\dots N}$, image pyramid level L, downsample ratio $r_{1\dots L}$ at each pyramid level, tile size $t_{1\dots L}$ at each pyramid level, search region $s_{1\dots L}$ at each pyramid level;
Result: alignment map $D_{1\dots N}$ on each alternative frames w.r.t. I_{ref};

1 $P_{ref} = \emptyset$;
2 **for** $r \leftarrow r_{1\dots L}$ **do**
3 $\quad P_{ref} = P_{ref} \cup \{\texttt{avg\_pool}(I_{ref}, r)\}$;
4 **end**
5 **for** $n \leftarrow 1 \dots N$ **do**
6 $\quad P_{alt} = \emptyset$;
7 $\quad$ **for** $r \leftarrow r_{1\dots L}$ **do**
8 $\quad\quad P_{alt} = P_{alt} \cup \{\texttt{avg\_pool}(I_n, r)\}$;
9 $\quad$ **end**
10 $\quad$ **for** $i \leftarrow L \dots 1$ **do**
11 $\quad\quad n_H = P_{ref}[i].height / \frac{t_i}{2} - 1$;
12 $\quad\quad n_W = P_{ref}[i].width / \frac{t_i}{2} - 1$;
13 $\quad\quad$ **if** $i == L$ **then**
14 $\quad\quad\quad d_i = \texttt{zeros}(n_H, n_W, 2)$;
15 $\quad\quad$ **else**
16 $\quad\quad\quad d_i = \texttt{interpolate}(d_{i+1}, n_H, n_W)$;
17 $\quad\quad$ **end**
18 $\quad\quad$ split $P_{ref}[i]$ and $P_{alt}[i]$ into tiles T_{ref} and T_{alt};
19 $\quad\quad$ **for** $y \leftarrow 1 \dots n_H$ **do**
20 $\quad\quad\quad$ **for** $x \leftarrow 1 \dots n_W$ **do**
21 $\quad\quad\quad\quad$ **for** $dx, dy \in s_i$ **do**
22 $\quad\quad\quad\quad\quad$ **if** *i == 1* **then**
23 $\quad\quad\quad\quad\quad\quad$ use L1 distance $dist_1$;
24 $\quad\quad\quad\quad\quad$ **else**
25 $\quad\quad\quad\quad\quad\quad$ use L2 distance $dist_2$;
26 $\quad\quad\quad\quad\quad$ **end**
27 $\quad\quad\quad\quad\quad a_y, a_x = \arg_{dx,dy} \min dist(T_{ref}[y,x], T_{alt}[y+dy+d_i[y,x,0], x+dx+d_i[y,x,1]])$;
28 $\quad\quad\quad\quad\quad d_i[y,x,0] = a_y$;
29 $\quad\quad\quad\quad\quad d_i[y,x,1] = a_x$;
30 $\quad\quad\quad\quad$ **end**
31 $\quad\quad\quad$ **end**
32 $\quad\quad$ **end**
33 $\quad\quad$ **if** $i == 1$ **then**
34 $\quad\quad\quad D_n = d_i$;
35 $\quad\quad$ **end**
36 $\quad$ **end**
37 **end**

Algorithm 1: Tile based L1/L2 mixed alignment.

III. Analysis, Implementation and Optimization

For demonstrating purpose, we firstly build a pure Python implementation `demo.py`. As will be illustrated in §IV, this algorithm can yield reasonable results but the naive loop-based Implementation requires ~ 127 seconds for each burst, which is infeasible for real-world deployment.

To speed up this implementation, we firstly analysis Algorithm 1, try to identify potential optimization opportunities. Next we implement these optimizations by generated C-language kernels and evaluate the performance gain. Finally we explain other engineering details and efforts we had taken to ensure the correctness and precision of our implementation.

A. Analyze the Algorithm

The proposed L1/L2 mixed alignment is basically a brute-force search with inherent parallelism, we analysis this algorithm along the formulation in Algorithm 1 to identify optimization opportunities.

a) For alternativa frames: This out-most loop (line 5 to 37) is both data and control independent among each loop steps. However parallelize in this granularity could be unfriendly to memory: note that memory location of each alternative images $I_{1\dots N}$ could be different, memory accessing continuity thus could not be guaranteed if we spawn a pool of

threads and let each thread align one of the alternative images.

b) For pyramid levels: To restrict the search region size, each search step is initialized by interpolated previous search steps (line 10 to 36, the initializing logic is from line 13 to 17). Because of this data-dependency, parallelism can not be applied along each pyramid levels. However, there still exist optimization opportunities on initializing step: note the interpolate operation in line 16 and pooling operation in line 8. Since these operations are nearly element-wised and is performed on continuous I_n and d_{i+1}, we could parallel along the rows of these operations before optimizing the next-step alignments.

c) For search tiles: Tile-based searching is formulated as double-loop (line 19 to line 32). The number of tiles spans from 6 (the top-most pyramid level) to $\sim$ 188K (the bottom pyramid level) for 13M-pixes input, the search process and output location of each tiles are not conflict to others. Thus we can easily get speedup by launching multiple threads, each thread for aligning one tile. Note that a large number of threads will be launched on lower pyramid levels, if all these threads executed in parallel, memory bandwidth could immediately become performance bottleneck. However the deploy target, ARM or x86_64 CPU, is usually capable of executing only $4 \sim 16$ threads in parallel, so we shall not take too much concern about the memory bandwidth problem.

d) For search locations: Each tile will slide though a rectangle search region (line 19, 20) and accumulate pixel wise differences. Since we have assigned all threads to each tiles as described in previous paragraph, no further parallelism could be utilized. However, since most of modern CPUs do support single-instruction-multiple-data (SIMD) execution pattern, we can further vectorize the subtraction and accumulation procedure on each search location, then utilize SIMD intrinsic to get more speedups.

SIMD basically means that, target CPU gets multiple arithmetic logic units (ALUs) available at each pipeline step, so multiple data could be loaded, computed and dumped at single instruction time. These instructions are wrapped as hardware-specific, language-level *intrinsics* so we could not be bothered writing assemble to get SIMD speedup. e.g. when computing the L1 distances between two 32-element 16-bit (`unsigned short` for C99) row-vector $I_{0...31}$ and $T_{0...31}$, instead of performing 32 loop steps

```
T tmp = 0;
for (int i = 0; i < 31; ++i) {
    tmp += fabs(I[i] - T[i]);
}
```

we can use `_mm512_sub_epi16` intrinsic to do these 32 subtractions, then use `_mm512_add_epi16` to accumulate all 32 elements, both in *single instruction time*:

```
__mm512i i_vec = _mm512_loadu_epi16(I);
__mm512i t_vec = _mm512_loadu_epi16(T);
__mm512i dst;
dst = _mm512_sub_epi16(i_vec, t_vec);
dst = _mm512_abs_epi16(dst);
// take reduction sum
dst = _mm512_add_epi16(dst,
      _mm512_srli_epi16(dst, 16));
```

As illustrated above, the computation heavy logic could be optimized from > 32 instruction time to 3 instruction time.

B. Optimizations

In this section we describe the implementation of aforementioned optimizations.

a) Rewrite performance-critical code by C: As illustrated in Algorithm 1, vast majority of alignment precess is based on loop, which is difficult to vectoring. Besides the analysis we investigated in §III-A, we run Python performance profiler `cProfile` to locate the performance-critical code, then rewrite them by C-language kernels. To be specific, we rewrote the average pooling kernels, the bilinear interpolate kernels and per-layer alignment kernels (line $18 \sim 32$ in Algorithm 1) in `Cython` format then generate efficient C-code by [**?**].

b) Utilize parallelism: As we discussed in §III-A, we decide to parallel at three granularity: the element-wise down-sampling and interpolation, and the dependency-free tile-wise searching. To be specific, for down-sampling and interpolation, since the internal matrices, e.g. input images and alignment maps, are all C-styled row-major memory chunk, we thus apply OpenMP [**?**] on the row-level, so each thread could be able to access continuous memory. OpenMP is wrapped into `prange` (paralleled range) in Cython:

```
for i in prange(h_out, nogil=True):
    for j in range(w_out):
        # compute scaling and indexing...
        for c in range(channels):
            # per-channel weighted sum...
```

Note that we have taken out the GIL (Global Interpreter Lock) [**?**] in the beginning of OpenMP invoking by `nogil=True`. This is necessary for speedup Cython generated C-code, otherwise the GIL will prevent multiple CPU cores from executing spawned threads in parallel.

c) Utilize vectorization: As analysis in III-A0d, when iterating dx though T_{ref}, T_{alt}, both cache and memory is continuous, thus we use Intel x86_64 intrinsics to optimize the search loop (line 21 to 30). Since our experiment platform is equipped AVX2-enabled Intel i7-8750H CPU [**?**], we used the intrinsic `_mm512_sub_epi16`, `_mm512_abs_epi16` and `_mm512_add_epi16` available on AVX2 instruction set to compute the L1 distance. These three intrinsics are used to compute vector subsection, vector absolute value and vector accumulation, respectively. The computation pattern is identical to our previous discussion.

All above optimizations can be find in our Cython implementation `align_cpu.pyx`. As will be illustrated in §IV, these optimizations introduce orders of magnitudes of accelerating to our implementation.

C. Other Implementation Details

There are several other implementation details in our optimization. We modified the bilinear interpolate algorithm to get better interpolates on image corners. We also write unit tests for all C-kernels to ensure the correctness.

a) Corners in bilinear interpolation: In standard bilinear interpolation, when computing the output value $y(u, v)$ from input x and scale factor s, source indices are usually taken by directly multiply output indices by scale factor, e.g.:

$$u_0 = \lfloor u \cdot s \rfloor$$
$$u_1 = \lceil u \cdot s \rceil$$

However, when interpolating four corner pixels, output values are sampled from only the relevant source input, thus context information will be miss. We thus add a little "bias" to the source selection:

$$u'_0 = \lfloor (u + 0.5) \cdot s - 0.5 \rfloor$$
$$u'_1 = \lceil (u + 0.5) \cdot s - 0.5 \rceil$$

Since 2d bilinear interpolations are all sampled from adjacent pixels, this little "bias" can help improve the quality of interpolation corners.

b) Unit tests for C-kernels: Since we must ensure the correctness of our implementation before and after each optimization, we wrote unit tests for all C-kernels. To be specific, we use `pytest` framework to run coverage tests, test input are random tensors generated by RNG with deterministic random seed, test ground truth are generated by PyTorch `nn.AvgPool2d`, `nn.BilinearUpsample2d` functions, alignment ground truth is set to the naive Python alignment output. Detailed implementations could be find at `test_align_cpu.py`.

IV. Results and discussions

In this section we illustrate our experimental results on HDR+ dataset [**?**] and accelerating introduced from our optimizations.

In Figure 2 we illustrate the results from proposed algorithm, applied on a burst of seagull images taken from HDR+ dataset. In Figure 2a, alignment map is computed by a 3-level image pyramid, while in Figure 2b a 4-level image pyramid is used. In both settings, each pyramid level is downsampled by ratio of 4 from previous level, tile size on each pyramid level is set to 16, search regions is within a ± 4 pixes rectangle for each tile.

Proposed algorithm successfully matched corresponding regions on both settings (see the response areas on alignment maps, which are wings of seagulls). Note that when deeper pyramid is used (Figure 2b), the alignment algorithm will generate less false-positive responses (see the left-bottom regions of alignment maps). However deeper pyramid requires more computation budget, so this accuracy-speed tradeoff should be taken into consideration when applying this algorithm in real-world applications.

(a) alignment with 3-level image pyramid

(b) alignment with 4-level image pyramid

Fig. 2: Alignment on burst with moving objects. In both sub-figures, top-left is the reference frame, bottom-left is the alternative frame, right-bottom is the alignment map.

Another important criterion of proposed algorithm is speed. We benchmark our implementations on a laptop with 2.2GHz Core i7 processor. As illustrated in Table I, after utilizing the optimizations we proposed in §III, time consumption of aligning two 13M-pixes raw images reduces from 127.47

Implementation	Input	Pyramid level	Time
naive Python	13M-pix $\times 2$	3	127.47s
multi-thread C	13M-pix $\times 2$	3	**0.32**s
multi-thread C	13M-pix $\times 2$	4	0.35s
multi-thread C	13M-pix $\times 10$	3	3.40s

TABLE I: Running time of our implementations.

seconds to 0.32 seconds, which gives us $\mathbf{398.34}\times$ speedup. Also note that our implementations runs at 2.94 FPS, which should be fast enough for real-world applications. Further speedup can be archived by more aggressive optimizations, e.g. using OpenCL or CUDA to further utilize parallelism.