Adaptive Spot-Guided Transformer for Consistent Local Feature Matching  
——Supplementary Material——

Jiahuan Yu1,* Jiahao Chang1, Jianfeng He1, Tianzhu Zhang1,2† Jiyang Yu3, Feng Wu1
1University of Science and Technology of China, 2Deep Space Exploration Lab, 3China Academy of Space Technology

In this supplementary material, we first introduce the general sparse attention operator in Section 1. In Section 2, we provide some details about our experiment. In Section 3, we show additional visualizations about the spot-guided attention and adaptive scaling modules.

1. General Sparse Attention Operator

Due to irregular key/value token number for each query in Spot Attention, the naive implementation by PyTorch [7] is not efficient for memory and computation, which uses a mask to set unwanted values in the attention map to 0. More generally, the same problem also exists when the numbers of key corresponding to queries are not the same. Inspired by PointNet [8] and Stratified Transformer [4], we implement a general sparse attention operator using CUDA that is efficient in terms of memory and computation. We attempt to only compute the necessary attention between much less query/key tokens.

We can divide a vanilla attention operator into 3 steps. Inputs are grouped as query Q, key K and value V. First, the attention map A is computed by dot production as $A = QK^T$. Then, a softmax operator is performed on the attention map: $A_s = \text{softmax}(A/\sqrt{K})$. Finally, the updated query O can be obtained by $O = A_sV$. We optimize these three steps separately.

In the step 1, because only a few results in A are useful for sparse attention, we do not need to compute the full A. Instead, we compute the dot productions between $L_m$ pairs of query and key. $M_q$ and $M_k$ record the indexes of query and key tokens whose dot productions are needed. The length of $M_q$ and $M_k$ are both $L_m$. Here, we denote the sparse attention map as $\text{attn}$, which is calculated by

$$\text{attn}[i] = Q[M_q[i]]K[M_k[i]]^T, \quad i = 0, 1, \ldots, L_m - 1. \quad (1)$$

In the step 2, we group the elements in $\text{attn}$ with the same query index and apply softmax on each group. The result is denoted as $\text{attn}_s$.

In the step 3, we compute the updated query

$$O[q] = \sum_{M_k[i]=q} \text{attn}_s[i] \cdot V[M_k[i]]. \quad (2)$$

All of three steps are implemented in CUDA.

Compared with the naive implementation using PyTorch [7], our highly optimized implementation reduces the memory and time complexity from $O(N_q \cdot N_k \cdot N_h \cdot N_d^2)$ to $O(L_m \cdot N_h \cdot N_d^2)$, where $N_q$, $N_k$ and $N_h$ are separately the numbers of query tokens, key tokens and attention heads, and $N_d$ is the dimension of each head. Considering $L_m \ll N_q \cdot N_k$, our implementation is much more efficient than the naive implementation.

In particular, we also calculate the matching matrix in spot-guided attention in this way and set the probability of unrelated pixels to 0, which can greatly reduce the memory and computation cost.

2. Experimental Details

2.1. Training Details

To reduce the GPU memory, we randomly sample 50% of ground truth matches to supervise the matching matrix at the coarse stage. And we sample 20% of the maximum number of coarse-level possible matches at the fine stage.
We train ASTR on MegaDepth \cite{5} for 15 epochs. The initial learning rate is $1 \times 10^{-3}$, with a linear learning rate warm-up for 15000 iterations. The learning rate curve is shown in Figure 2.

### 2.2. Differences between Baseline and LoFTR

There are two main differences between our baseline and LoFTR \cite{9}.

1. **Normalized Positional Encoding.** LoFTR \cite{9} adopts the absolute sinusoidal positional encoding by following \cite{1}:

   \[
   \text{PE}_i(x, y) = \begin{cases} 
   \sin(w_k \cdot x), & i = 4k \\
   \cos(w_k \cdot x), & i = 4k + 1 \\
   \sin(w_k \cdot y), & i = 4k + 2 \\
   \cos(w_k \cdot y), & i = 4k + 3
   \end{cases},
   \]

   where $w_k = \frac{1}{10000w^2}$, $d$ denotes the number of feature channels and $i$ is the index for feature channels. Considering the gap in image resolution between training and testing, we utilize the normalized positional encoding as \cite{2}, which is proven to mitigate the impact of image resolution changes in \cite{2}. The normalized positional encoding $\text{NPE}_i(\cdot, \cdot)$ can be expressed as

   \[
   \text{NPE}_i(x, y) = \text{PE}_i(x \cdot \frac{W_{\text{train}}}{W_{\text{test}}}, y \cdot \frac{H_{\text{train}}}{H_{\text{test}}}),
   \]

   where $W_{\text{train/test}}$ and $H_{\text{train/test}}$ are width and height of training/testing images.

2. **Convolution in Attention.** Chen et al. \cite{2} find that replacing the self attention with convolution can improve the performance. Hence, we deprecate self attention and MLP, and utilize a $3 \times 3$ convolution in our ASTR.

### 2.3. CNN Backbone

Here we leverage a deepened version of Feature Pyramid Network (FPN) \cite{6}, which achieves a minimum resolution of 1/32. The initial dimension for the stem is still 128 as LoFTR \cite{9}, and the number of feature channels for subsequent stages is [128, 196, 256, 256, 256].

### 3. Visualization Results

In Figure 3, we pick up two similar adjacent pixels as queries and visualize the corresponding attention maps of vanilla and our spot-guided attention for comparison. The vanilla attention mechanism is vulnerable to repetitive textures, while our spot-guided attention can focus on the correct areas in these repeated texture regions. Because large scale variation occurs frequently on outdoor datasets, we
mainly visualize the grids from the adaptive scaling module and corresponding depth maps on MegaDepth [5]. As shown in Figure 4, our adaptive scaling module can adjust the size of grids according to depth information.

References


