# A Technical Breakdown of FlowFormer

Taskeen Jafri

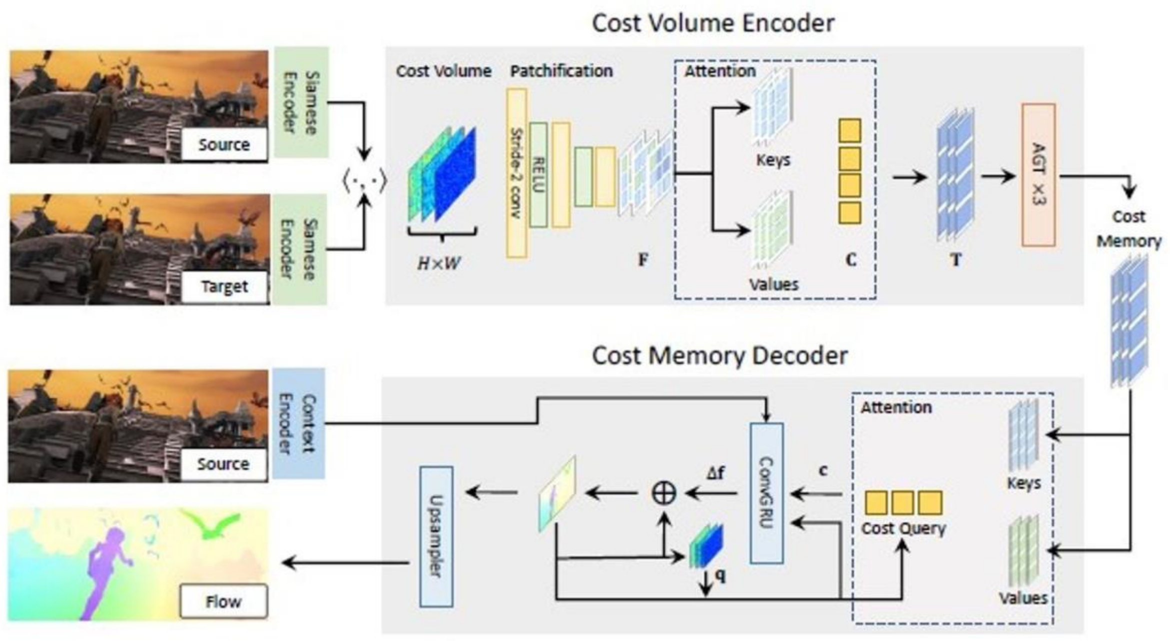
**Abstract.** Flow transFormer, commonly known as FlowFormer [1], is a model for calculation of optical flow between 2 images. FlowFormer combines convolutional layers, positional encoding, attention mechanism and uses a transformer based architecture, giving high-quality optical flows and achieving state-of-the-art results. It tokenizes the 4D cost volume built from image pairs, encodes the cost tokens using Alternate-Group Transformation layers (AGT), then decodes using dynamic positional cost queries.

## 1. Introduction

Optical Flow is the concept in computer vision which calculates the pattern of apparent motion between 2 or more images.
In this context, cost means the measure of dissimilarity between 2 frames.

## 2. Method

The FlowFormer follows an encoding-decoding method for the prediction of Optical Flow. This is the flow of the encoding-decoding process:



(image copied from the paper)

### 2.1 Cost Memory Encoder

The above displayed picture shows that 2 images are given as input to the FlowFormer, the source and target images. These images are of the size ($H_1$, $W_1$). A backbone vision network is used to extract the $H×W×Df$ feature map, from the input $H_1×W_1×3RGB$ Image. The values of H and W are set as ($H$, $W$) = ($H_I$/8, $W_I$/8) in the FlowFormer. Then, a 4D cost volume is constructed of the size ($H$×$W$×$H$×$W$), by computing the dot-product similarities between all pixel pairs between source and target feature maps. The 4D cost volume can be seen as a series of 2D cost maps.

Typically, previous CNN models used to simply calculate the correspondence points between the target and source image. This, however, tolls a heavy computational cost. To find a way around this problem, the FlowFormer encodes the whole 4D cost volume into a *cost memory,* and decreases dimensionality. This helps reduce the computational load, while also ensuring the quality of the final product.

Directly encoding the 4D cost volume is too heavy, since the model would have to calculate a lot of tokens. So, it is carried out in 3 steps:

1. **Cost Map Patchification:**

The model performs patchification of the Cost Map (2D), of size $H$×$W$. We denote the pixel x's cost map by $Mx$. It passes these cost maps through a series of strided (Stride=2) Convolutional layers and ReLU layers to give the output as ($H$/8×$W$/8×$Dp$). The value of $Dp$ was specified as 64 in the paper.

Here is the breakdown:
conv1: H×W → (H/2 × W/2 × $Dp$/4), which is images of size of H/2 × W/2 with $Dp$/4 channels.
ReLU-1: no change in dimensions/size of the feature map.
conv2: (H/2 × W/2 × $Dp$/4) → (H/4, W/4, $Dp$/2)
ReLU-2: no change in dimension/size of the feature map.
conv3: (H/4 × W/4 × $Dp$/2) → (H/8 × W/8 × $Dp$)
ReLU-3: no change in dimension/size of the feature map.

2. **Cost Patch token embedding:**

Although the patchification results in a sequence of patch feature vectors for each source pixel, the number of patch features are still too high, hindering the efficiency of the model. Hence, the patch features $Fx$ are summarized via 'K' amount of latent codewords. The latent codewords are denoted by C ∈ $R^{K×D}$. The latent codewords query each source-pixel's cost-patch features to further summarize each cost map into 'K' latent codewords of 'D' dimensions via the dot-product mechanism. The latent codewords are randomly initialized but updated via back-propagation, increasing accuracy of the model.

After obtaining the latent codewords, we require the latent representation. This is done via the following steps:
$Kx$ = Conv1×11×1(Concat($F_x$, PE))
$Vx$ = Conv1×11×1(Concat($F_x$, PE))
$Tx$ = Attention($C$, $Kx$, $Vx$)

Attention mechanism: The model basically computes the similarity between the values C (latent codewords) and $Kx$ (Keys), and generates a set of attention weights. These weights determine the importance and contribution of each value in *Vx* towards the final product, which is *Tx* in this case.

*Kx* and *Vx* are the keys and value tensors of the feature vectors respectively. The Feature vector, *Fx*, is concatenated with the positional embedding of length *Dp*. This is a very smart way to integrate the positional features of the pixels, which will be very useful in step 3. Thus, the cost map of the source pixel x can summarized into K latent representations, $T_x \in R^{K\times D}$ by conducting multi-head dot-product attention with queries (C), keys (*Kx*), values (*Vx*).

Since (in most cases), K×D ≪ H×W, this provides a much more compact representation of 4D cost volume, $T \in R^{H\times W\times K\times D}$.

3. **Cost memory encoding:**

Although the 4D cost volume, $T \in R^{H\times W\times K\times D}$**,** is much more compact than the original cost volume, the amount of computational power required to convert it into tokens is still a lot, since computational cost quadratically increases with number of tokens.

So, the FlowFormer presents yet another trick up it's sleeve: The Alternate-Group Transformer layer. It divides the tokens (*Kx*,*Vx*,*C*) into subgroups of H × W groups that contains K tokens (*Tx*) and K groups that contains H×W tokens (*Ti*), and encodes the tokens inside the groups via self-attention and spatially separable self-attention (ss self-attention).

The amount of tokenization remains the same, but the amount of relations the model has to compute decreases drastically. Here is a very simple example for understanding purposes: Suppose we have the sentence "The quick brown fox jumps over the lazy dog." We need to tokenize it. What the model would do, is take the word "The" and calculate its relations with all the other words: [The, quick], [The, brown], etc, making a total of 8 computations only for the word "The". Moving on, it will do the same for "quick", "brown", and so on until all of them are calculated.

However, by grouping the sentence into: 'The quick', 'brown fox', 'jumps over the lazy dog', the model only calculates the relations between [The, quick], [brown, fox], [jumps, over], [jumps, the], [jumps, lazy] and [jumps, dog]. Thus, it decreases the number of computations from 72 to 6.
The data lost is minimal between the tokens within different groups. This is because, if you recall, the model processed the feature vectors along with their positional embeddings

$$K_x = \mathrm{Conv}_{1x1}(\mathrm{Concat}(F_x, \mathrm{PE}))$$
$$V_x = \mathrm{Conv}_{1x1}(\mathrm{Concat}(F_x, \mathrm{PE}))$$
$$T_x = \mathrm{Attention}(C, K_x, V_x)$$

The AGT Layer processes the tokens in *T* in 2 different ways to ensure there is no loss between the tokens across 2 different groups. It divides the tokens in *T* into H×W groups containing K tokens (called *Tx*) and K groups containing H×W tokens (called *Ti*).

Approach 1: The tokens in group *Tx* are processed using self-attention, which calculates weights among the tokens within the group, capturing intra-group relations between tokens. The self-attention calculates the weights of each token, and it is given as an output in the format of tensor. This tensor is passed through FFN (Feed-Forward Network), which consists of fully connected layer and activation functions, which apply linear transformation and non-linear activations to produce the final output.

Hence, *Tx* looks like this:
$Tx = \mathrm{FFN}(\text{Self-attention}(Tx(1), Tx(2), \ldots , Tx(K))$
where $Tx(i)$ denotes the i-th latent representation for coding the source pixel x's cost map. The updated *Tx*'s are then re-organized back to obtain the updated 4D cost volume *T*.

Approach 2: The tokens in group *Ti* are processed using SS-self-attention, which calculates relations between tokens across different groups. Similar to *Tx*, the tensor after passing through SS-self-attention is then passed through the FFN.
$Ti = \mathrm{FFN}(\text{SS-self-attention}(Ti(1), Ti(2), \ldots , Ti(K))$
where $Ti(j)$ denotes the j-th group. The updated *Ti*'s are then re-organized back to obtain the updated 4D cost volume *T*.

In FlowFormer, there are 3 AGT Layers, allowing effective information exchange and tokenization.

Following the above 3 steps allows the model to effectively convert the H×W×H×W 4D cost volume into H×W×K tokens of length D. The H×W×K tokens are known as *cost memory*.

## 2.2 Cost Memory Decoder

The decoding of the Cost Memory takes place in 2 steps: Cost Memory aggregation and Recurrent flow prediction.

1. **Cost Memory Aggregation:**

For predicting the flows of H×W source pixels, the FlowFormer model generates a sequence of H×W cost queries (*Qx*) , i.e., each cost query is responsible for a single pixel via co-attention on the cost memory.

To generate *Qx*, first the location of the pixel in the source image is estimated to the target image using the current/given flow: p = x + f(x). Here, x is the source pixel, f(x) is the flow, and p is the prediction of the source pixel x in the target image. Then, with the pixel p as the center, we crop out a 9×9 window from the cost map *Mx*: $qx = \mathrm{Crop}_{9\times9}(Mx,p)$.

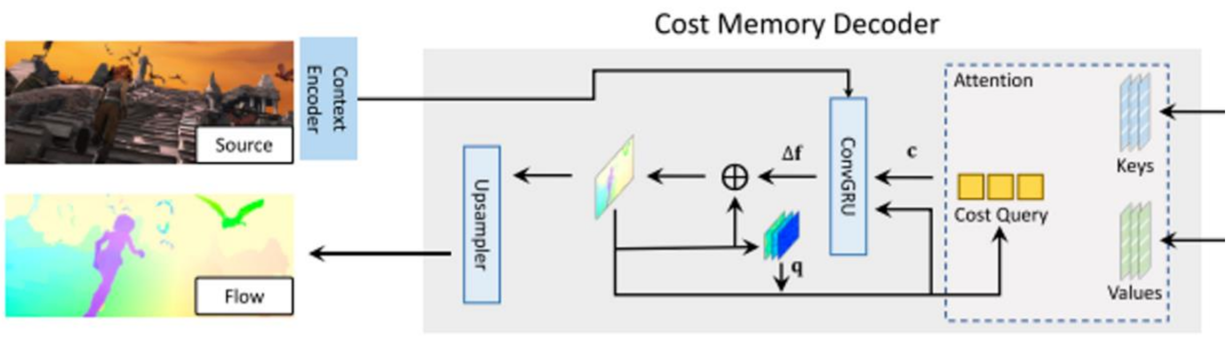
Along with Cost query *Qx*, we also calculate the keys (*Kx*) and values (*Vx*) for the cost map of source pixel x:

$Qx = FFN(FFN(qx)+PE(p))$
$Kx = FFN(Tx)$
$Vx = FFN(Tx)$
$c_x = Attention(Qx,Kx,Vx)$
The cost query, *Qx*, is dynamically updated in terms of the fed position, it is called 'dynamic positional cost query'.

An important point to note is, calculating the keys and values once again is unnecessary. The values calculated while encoding the Cost Memory can be re-used during decoding, reducing the computational load.

2. **Recurrent Flow Prediction:**

The calculation of updated Optical Flow (Δf(x)) is done by 'ConvGRU'. It requires 4 input values: $c_x$, $q_x$, $t_x$, *f(x)*. Of 4, *cx* (aggregated cost features), *qx* (query features) and *f*(*x*) (current optical flow estimation) have been calculated. The term *tx* represents the contextual information from the source image. This is calculated by passing the source image through a few convolutional layers, along with activation functions such as ReLU and pooling layers.
Here's a visual representation of how this takes place:



(image copied from the paper)

The predicted flow is calculated by the following equation:
$\Delta f(x) = \text{ConvGRU}(\text{Concat}(cx, qx), tx, f(x))$.

ConvGRU module is an important part of flow calculation. Essentially, the ConvGRU module acts like a memory cell that keeps track of past information and uses it to make predictions about future states.

The flows generated at each iteration are upsampled to the size of the source image via a convex upsampler. The process of calculation of optical flow is supervised by ground-truth flows at all recurrent iterations with increasing weights, meaning as the iterations pass, the weights increase. The model initially does not know much about the problem, so the weights to the values for calculation of loss is low. As the iterations increase, the model starts learning about the problem, hence the weights are increased.

The model is evaluated by Sintel, KITTI datasets.

The model is pre-trained on the datasets of FlyingChairs dataset for 120k iterations with batch_size = 8, and FlyingThings dataset for 120k iterations with batch_size = 6. The authors also use a one-cycle learning rate scheduler. The highest learning rate is set as $2.5 \times 10^{-4}$ on FlyingChairs and $1.25 \times 10^{-4}$ on the other training sets.

## References:

Zhaoyang Huang, Xiaoyu Shi, Chao Zhang, Qiang Wang, Ka Chun Cheung, Hongwei Qin, Jifeng Dai, and Hongsheng Li.