Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

# Optical Flow Estimation with Separable Cost Volume

Simon Tobias Bihlmaier

| | |
|---|---|
| **Course of Study:** | Informatik |
| **Examiner:** | Prof. Dr.-Ing. Andrés Bruhn |
| **Supervisor:** | Azin Jahedi, M.Sc. |
| **Commenced:** | June 1, 2022 |
| **Completed:** | December 1, 2022 |

## Abstract

Optical Flow Estimation is an important task in computer vision that involves finding correspondences between subsequent frames. Recently many approaches have focused on learning to estimate optical flow using neural networks. Constructing and processing correlation volumes using convolutional neural networks is applied in many works and yields good results. Separable Flow by Zhang *et al.* is an extension for correlation volume based methods such as Recurrent All-Pairs Field Transforms for Optical Flow by Teed and Deng. It separates the four dimensional correlation volume into two correlation volumes with only one instead of two displacement dimensions. At the time of its release, state of the art estimation quality results were reported for Separable Flow on the Sintel and KITTI datasets. By investigating the implementation provided by the authors, significant changes of the model structure and training schedule compared to the paper can be discovered.

The goal of this thesis is to verify the published claims about the training regime, model structure, estimation quality and number of parameters. This is accomplished by reverting identifiable changes in multiple ablation steps. Evaluating the ablation step closest to the published description shows that the claimed estimation quality can not be reproduced. The provided model implementation combined with the published training schedule performs the most similar to the results of the paper. Additionally, the claim that the four dimensional correlation volume does not need to be stored in order to compute the three dimensional correlation volumes is investigated. This claim is verified by providing an alternative parallel implementation for Graphics Processing Units that fulfills the storage constraint. At the cost of longer computation times, the memory consumption of Separable Flow can be reduced during training and inference. In an effort to improve the estimation quality, Global Motion Aggregation by Jiang *et al.* is added to Separable Flow. On the ablation training schedule, the combined model achieves better results than Global Motion Aggregation in isolation.

## Kurzfassung

Die Bestimmung des optischen Flusses ist eine wichtige Aufgabe im Maschinensehen, die darin besteht Korrespondenzen zwischen aufeinanderfolgenden Bildern zu finden. In letzter Zeit haben sich viele Ansätze mit dem Lernen der optischen Fluss Bestimmung mithilfe neuronaler Netze beschäftigt. Die Konstruktion und Verarbeitung von Korrelationsvolumina mit Konvolutionalen Neuronalen Netzwerken wird in vielen Arbeiten verwendet und führt zu guten Ergebnissen. Separable Flow von Zhang *et al.* ist eine Erweiterung für Methoden wie Recurrent All-Pairs Field Transforms for Optical Flow von Teed und Deng, die Korrelationsvolumina nutzen. Es trennt das vierdimensionale Korrelationsvolumen in zwei Korrelationsvolumina mit nur einer statt zwei Verschiebungsdimensionen auf. Zum Zeitpunkt der Veröffentlichung erzielte Separable Flow die besten Ergebnisse auf den Sintel und KITTI Datensätzen. Bei der Untersuchung der von den Autoren bereitgestellten Implementierung konnten signifikante Änderungen in der Modellstruktur und im Trainingsplan gegenüber der Veröffentlichung festgestellt werden.

Das Ziel dieser Masterarbeit ist es, die Behauptungen der Veröffentlichung über das Trainingsregime, die Modellstruktur, die Qualität der Schätzung und die Anzahl der Parameter zu überprüfen. Dies wird erreicht, indem identifizierbare Änderungen in mehreren Ablationsschritten rückgängig gemacht werden. Die Auswertung des Ablationsschritts, welcher der veröffentlichten Beschreibung am nächsten kommt, zeigt, dass die behauptete Schätzungsqualität nicht reproduziert werden kann. Die bereitgestellte Modellimplementierung in Kombination mit dem veröffentlichten Trainingsplan kommt den Ergebnissen der Veröffentlichung am nächsten. Außerdem wird die Behauptung, dass das vierdimensionale Korrelationsvolumen nicht gespeichert werden muss, um die dreidimensionalen Korrelationsvolumina zu berechnen, untersucht. Diese Behauptung wird überprüft, indem eine alternative parallele Implementierung für Grafikprozessoren beschrieben wird, welche die Speicherbeschränkung erfüllt. Auf Kosten längerer Berechnungszeiten kann der Speicherverbrauch von Separable Flow während des Trainings und der Inferenz reduziert werden. Um die Qualität der Schätzung zu verbessern, wird Global Motion Aggregation von Jiang *et al.* zu Separable Flow hinzugefügt. Auf dem Ablationstrainingsplan erzielt das kombinierte Modell bessere Ergebnisse als Global Motion Aggregation allein.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Abbreviations

**BS** batch size. 39

**CNN** Convolutional Neural Network. 21

**Conv-GRU** Convolutional Gated Recurrent Unit. 25

**EPE** end point error. 41

**GMA** Global Motion Aggregation. 19, 63

**GPU** Graphics Processing Unit. 5, 11, 16

**GRU** Gated Recurrent Unit. 24, 25

**LR** learning rate. 39

**LSTM** Long Short-Term Memory. 24

**RAFT** Recurrent All-Pairs Field Transform for Optical Flow. 16, 27, 48, 49, 50, 53

**RNN** Recurrent Neural Network. 23, 24, 25

**SGA** Semi-Global Guided Aggregation. 18, 19, 47

**SGM** Semi-Global Matching. 18

**WD** weight decay. 39

# 1 Introduction

Correspondence problems appear in many areas of computer vision. A correspondence problem is the task of matching two sets of entities such that the mapping fulfills predefined criteria. Optical flow estimation is a prominent example for such a problem in Computer Vision. It is defined as finding a displacement field between corresponding pixels of two images.

Dynamic scenes possess several characteristics that make optical flow estimation challenging. Arbitrary motion of objects can lead to occlusions by rotation, moving behind other objects or moving out of frame. Varying illumination as well as shading can lead perceived flow that is not reflected in true object motion relative to the camera.

Despite being a challenging problem, it is worthwhile to pursue solutions since it can be used as a component of many low level vision tasks. Examples include object motion and depth estimation as well as segmentation of moving objects [ODo05]. This enables higher level tasks such as robot navigation [CGN14], video editing [TBKP12], scene reconstruction with dynamic objects [ZZL+20] and hand gesture recognition in human computer interaction [CT98].

Due to the challenges mentioned, formulating objective functions that enforce the flow field to be close to the real object motion is non-trivial. The simplest matching objective is the gray value constancy assumption, where pixels are matched based on how similar their gray values are. Due to the large search space and limited domain of gray values, matches between pixels have a high probability to be ambiguous. This means that pixels may have multiple matches with the same gray value.

Block matching combats this problem by considering the neighborhood of each pixel for gray value constancy. By depending on differences between the gray values of both pixels neighborhoods, ambiguities in the matching score of candidates are avoided.

Variational methods such as the method of Horn and Schunck [HS81], formulate the optical flow task as a global optimization problem. An objective function over all pixels needs to be formulated. This function may contain a term that penalizes deviations from the gray value constancy assumption in the flow field called data term. An additional term can be introduced to enforce smoothness of the flow field and resolve ambiguities. Oftentimes, improvements to the optimization objective necessitate more complicated solution strategies which may lead to long computation times.

Deep learning based methods avoid the formulation of an objective function. Instead, supervised deep learning methods receive training data that includes the ground truth flow field. Therefore, the objective function is implicitly specified through examples. Innovations in such methods stem from modifications to the model structure such that the model better suits the optical flow estimation task. Improvements can also be achieved by providing more and better quality training data [BWSB12; DFI+15a; GLU12; MG15; MIH+16] or modifying the parameters of the training procedure.

The first deep learning techniques for optical flow estimation estimated the flow field in one pass through a fully convolutional network. “Flownet: Learning optical flow with convolutional networks” by Dosovitskiy *et al.* [DFI+15b] introduced the first convolutional neural network architecture for optical flow. The images were processed by a large number of convolutional layers connected in a sequential manner with forward skip-connections. Following this seminal work, many other works have proposed improvements over conventional two dimensional (2D) fully convolutional networks, for example by adding 3D convolutional layers [TBF+16] or coarse-to-fine processing [RB17].

Recently, the iterative optical flow prediction method of “RAFT: Recurrent All Pairs Field Transforms for Optical Flow” [TD20] has gained a tremendous amount of popularity because of its improved the estimation quality over previous methods and comparatively short training times. Many other successful methods in the following years such as [JCL+21; JMRB22; ZWPT21] have been based on it. Especially “Separable Flow: Learning Motion Cost Volumes for Optical Flow Estimation” by Zhang *et al.* [ZWPT21] stands out because of its estimation quality improvement over RAFT, which led to becoming the state of the art optical flow estimation method on the Sintel [BWSB12] and KITTI [GLU12; MG15] datasets at the time of submission. With the release of the supplementary material and the implementation of the model, it was discovered that both contained significant differences to their description in the paper. Particularly striking differences in the implementation are the use of the 4D correlation volume during motion refinement and leaving out learned channels of the 3D correlation volume. Those changes take away from the main innovation of the method which lies in the use of learned 3D correlation volumes. Because of these differences, the question arises whether it is possible to reproduce the results in the paper by reverting the changes made in the provided implementation. In this thesis, the question will be answered by reversing changes of the implementation and evaluating the model. Furthermore, the authors claimed that it is possible to compute the 3D correlation volumes without storing the 4D correlation volume as an intermediary result. However, they do not provide any evidence to back up their claim. This claim will be investigated by providing a parallel computation strategy for Graphics Processing Units, implementing it and evaluating the results. Finally, to improve the estimation quality of Separable Flow, Global Motion Aggregation [JCL+21] will be added as a submodule to aggregate the motion features. The approach for integrating the submodule into Separable Flow will be described and the results of this expanded model will be discussed.

# 2 Related Work

Several techniques have been introduced to deep-learning based optical flow estimation methods in recent years. They include the use of cost volumes in end-to-end trainable neural networks, the use of coarse-to-fine approaches, iterative refinement and lookup of features as well as cost volume aggregation and techniques to improve occlusion handling. In the following sections, these techniques that laid the foundation for RAFT and in consequence Separable Flow will be discussed.

**Cost Volumes for Deep-Learning Methods** Cost volumes have been used in many previous works on correspondence problems. Deep Learning methods using cost volumes can be distinguished by their ability to be trained end-to-end.

The method of DC Flow introduced by Xu *et al.* [XRK17] constructs the 4D cost volume using a learned feature embedding for each pixel. It is not end-to-end trainable, necessitating a loss function for learning the embedding that is detached from the main optical flow prediction task. The cost volume is computed in a manner comparable to RAFT, using the pixel-pairwise scalar product of the features. However the features are normalized and the result subtracted from one to convert the correlation value into a cost value that is related to the euclidean distance between the feature vectors.

In contrast to the previous method, end-to-end methods can pass the gradient of the flow estimation loss through the cost volume. Consequently, the image features can be trained directly using a single objective such as minimizing the end point error.

Dosovitskiy *et al.* [DFI+15b] compared the purely convolutional architecture FlowNetS with an architecture that uses a correlation operation on learned image features called FlowNetC. The correlation operation is range-limited, thus correlation values are only computed within the specified displacement range. They found that FlowNetC performed worse than FlowNetS on large displacements. This led to the hypothesis that limiting the maximum displacement of the correlation operation may render the model incapable to predict large motions.

Similarly to FlowNet, PWC-Net [SYLK18] is a coarse-to-fine approach that computes the cost volume up to a maximum displacement at each resolution level from the warped image features. They argue that the maximum displacement can be set to a small value because the relative covered radius doubles with each level.

Yang and Ramanan [YR19] enable correlation volumes to express pixel similarities according to multiple different pixel embeddings. They propose the use of 4D correlation volumes with an additional similarity channels dimension. In contrast to the multi channel 3D correlation volumes of Separable Flow, they do not aggregate the correlation to merge all channels into one. Instead, a hypothesis selection network is used to compute a weighting between flow hypothesis emerging from each of the similarity channels.

Deformable Cost Volumes is a method introduced by Lu *et al.* [LVW+20]. They strive to address the problem of coarse-to-fine approaches of not being able to estimate the motion of small, fast moving objects well. Furthermore, the method avoids artifacts created by warping the second frame. To accomplish this, the unmodified second frame features in a neighborhood around the flow estimate are used to construct the cost volume for a limited displacement range. This process can also be seen as a sampling of values from a full displacement range correlation volume. For each location in the feature map of frame one, the locations in a neighborhood around the correspondence in frame two are sampled from the full range correlation volume. Therefore, the method differs significantly from the previous approaches that calculate the cost volume from the warped second frame feature map. The method RAFT presented in the next paragraph builds on the idea of incorporating the flow estimate with the correlation volume by sampling from the correlation volume instead of warping the second frame.

RAFT [TD20] expands on the idea of moving operations from the images to the cost volume. In addition to sampling from the cost volume instead of warping the image features, they also move the multi-scale aspect from the image features to the cost volume. Not the image features, but the full 4D cost volume is pooled along the displacement dimensions to create a cost volume pyramid.

Previous methods rely on operations such as the dot product as a similarity metric between the features. However, the image features comparison metric can also be learned. Wang *et al.* [WZD+20] propose "Displacement-invariant matching cost learning for accurate optical flow estimation" (DICL) which learns a matching cost network that receives the concatenated image features of one pixel in both frames as input and outputs the similarity. The matching cost network is applied separately for each displacement hypothesis in the search window.

As an extension of 4D correlation volume based on image feature similarities, cost volumes may be improved by applying a learned compression. Zhang *et al.* [ZWPT21] introduce Separable Flow, which learns two 3D cost volumes as a compression of the non-learned 4D cost volume of a host method such as RAFT. The 3D cost volumes have multiple channels that are either based on the statistics or a learned aggregation of one of the displacement dimensions.

**Cost Volume Aggregation** Cost volume aggregation is another important aspect of the optical flow estimation method Separable Flow. In the related domain of stereo matching, cost volume aggregation has already been used in some methods to promote the similarity of neighboring displacements [Hir08; HRB+12; ZPYT19; ZQY+20]. Works on stereo matching have inspired to cross over such methods for stereo cost volumes to optical flow cost volumes.

Xu *et al.* [XRK17] adapt Semi-Global Matching (SGM) [Hir08] from 3D stereo to 4D optical flow cost volumes. By processing the cost volume using their version of SGM before inferring the flow, they were able to improve the accuracy of their method.

The cost volume aggregation in Separable Flow is also related to SGM. It is based on the previous works GANet and DSMNet [ZPYT19; ZQY+20] on stereo matching. "GA-Net: Guided Aggregation Net for End-to-end Stereo Matching" introduced an end-to-end learnable stereo cost volume aggregation method called Semi-Global Guided Aggregation (SGA) that remodels SGM such that it is end-to-end trainable. Because Separable Flow creates two 3D cost volumes, each of

them has only a single displacement dimension like in stereo matching. Thus, Semi-Global Guided Aggregation (SGA) layers from GANet can be applied to the separated cost volumes without any major modifications.

**Occlusion Handling** Depending on the type of approach that is chosen, different strategies are applied to handle occlusions. In variational methods, occlusions are often seen as outliers because they lead to violations in the gray value constancy assumption. Thus, optimization objectives are formulated to limit the influence of large deviations from the gray value constancy assumption on the energy function [BBPW04; ZPB07]. This means that the occlusion problem is solved alongside many other effects causing violations such as noise and illumination changes.

Many deep learning methods also do not explicitly reason about occlusions. Instead, the model structure is chosen such that information is well propagated between pixels, including occluded ones. This facilitates learning good predictions for occluded pixels. In the case of Global Motion Aggregation (GMA) introduced by Jiang *et al.* [JCL+21], the flow information called motion features is propagated between all pixels. Information is aggregated in a weighted sum, where the weights result from the self similarity of the first image features projected by a learned matrix.

Separable Flow compresses the 4D correlation volume into two 3D correlation volumes and applies multiple SGA layers to the 3D correlation volumes. Zhang *et al.* conclude that their method Separable Flow is able to overcome the ambiguities caused by occlusions because it can exploit "[. . .] non-local contextual information and prior knowledge [. . .]" [ZWPT21, p.10814].

## Thesis Organization

The further contents of this thesis are organized as follows.

In the next chapter, key concepts used in this thesis are explained. This includes structures used in neural networks, the basics of optical flow, and detailed explanations of the three main optical flow estimation methods covered in this thesis.

Chapter 4 investigates the differences between the paper and provided implementation of Separable Flow. The identified changes are reverted and the resulting models evaluated in multiple steps.

After that, Chapter 5 explores whether it is possible to omit the storage of the 4D correlation volume to separate it into two 3D correlation volumes. An implementation accomplishing this is described and the memory savings as well as the computation time are measured.

To further improve the estimation quality, Chapter 6 describes how Global Motion Aggregation can be integrated with Separable Flow. The final accuracy is evaluated for different model versions from Chapter 4.

Finally, Chapter 7 concludes the thesis by giving a summary of the results and making suggestions for further work.

# 3 Foundation

This chapter compiles the foundations that are relevant to this thesis. Sections 3.1–3.3 describe well established neural network structures that are used in deep optical flow estimation. Following this, Section 3.4 and 3.5 contain basic definitions for optical flow. Then the three deep optical flow estimation methods RAFT, Separable Flow and Global Motion Aggregation are detailed. Finally, Section 3.9 gives a short summary on general purpose computation with GPUs.

## 3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) can consist of different combinations of layers. In this section, the most common layer types in convolutional neural networks are described.

**Convolutional Layers** The most important layer in convolutional neural networks is the convolutional layer, since it usually holds trainable parameters. This layer can be parameterized by the number of input channels $C_{\text{in}}$ and output channels $C_{\text{out}}$, kernel size, padding size and stride.

The most common convolutional layer for processing images is two dimensional. In 2D convolutional layers, the aforementioned parameters such as kernel size and padding are also two dimensional. For an input tensor of shape $(C_{\text{in}}, H, W)$, the convolution layer produces an output tensor of shape $(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$. Furthermore, the kernel weights are of shape $(C_{\text{out}}, C_{\text{in}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$ and the bias weights tensor has the size $C_{\text{out}}$.

$$H_{\text{out}} = \left\lfloor \frac{H + 2 \cdot \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor \tag{3.1}$$

$$W_{\text{out}} = \left\lfloor \frac{W + 2 \cdot \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor \tag{3.2}$$

Equations (3.1) and (3.2) relate the size of the input of the convolutional layer to the size of the output. Depending on the parameters of the layer, the spatial size of the output $(H_{\text{out}}, W_{\text{out}})$ may increase or decrease. For example, if the stride is 1 and the kernel size is equal to $2 \cdot \text{padding} + 1$ then the output size will be the same as the input size.

$$(\hat{x}, \hat{y}) = (x \cdot \text{stride}[0], y \cdot \text{stride}[1]) \tag{3.3}$$

Each spatial index $(x, y)$ of the output can be mapped to an index range from $(\hat{x}, \hat{y})$ to $(\hat{x}+H_k, \hat{y}+W_k)$ in the padded input tensor "in_pad", as displayed in Equation (3.3). This is the spatial range that is aggregated in every channel to arrive at the output for the corresponding index. To shorten the equation, the alias $(H_k, W_k)$ was chosen for the kernel size $(\text{kernel\_size}[0], \text{kernel\_size}[1])$.

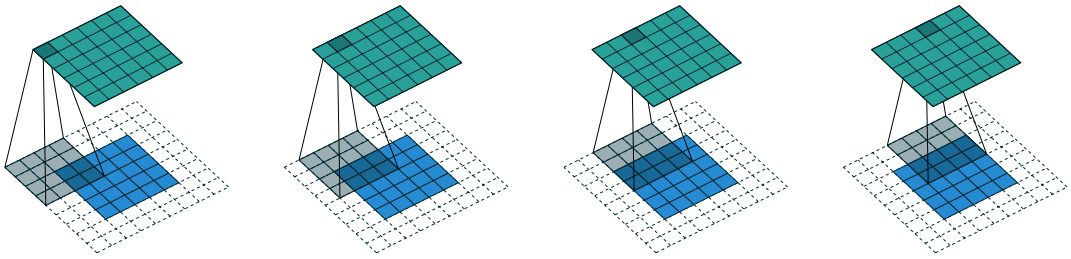

**Figure 3.1:** Visualization of applying a convolutional layer with kernel size $(4, 4)$, padding size $(2, 2)$ and stride $(1, 1)$. Input tensor: blue, Padding: white, Output tensor: cyan. Image source: [DV16].

$$\text{out}(c_{\text{out}}, x, y) = \text{bias}(c_{\text{out}}) + \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{i=0}^{H_k-1} \sum_{j=0}^{W_k-1} \text{weight}(c_{\text{out}}, c_{\text{in}}, i, j) \cdot \text{in\_pad}(c_{\text{in}}, \hat{x} + i, \hat{y} + j) \quad (3.4)$$

Equation (3.4) shows how the convolution is computed for every channel $c_{\text{out}}$ and spatial index $(x, y)$ of the output "out". For each channel in the output image, the spatial aggregation over all channels in the input is calculated. In the spatial range specified by the kernel size, the input tensor is aggregated in a sum weighted by the kernel. The bias of the corresponding output channel is added to the result.

Figure 3.1 illustrates the spatial aggregation of four different output indices. In the lower part of the image, the input is shown in blue with white padding and dark grey aggregation range that is weighted by the kernel. Above the input, the output tensor is displayed in cyan, with the resulting output index marked in dark grey. After each step in the sequence, the output index is moved by one according to the stride of $(1, 1)$.

**Pooling Layers** Convolutional neural networks often make use of pooling layers to reduce the spatial size of the output. Since the operation is very similar to a convolutional layer, the same parameters of kernel size, padding size and stride apply. Each index of the output summarizes a patch of the input, by applying functions without learnable parameters such as the maximum or the average.

$$\text{out}(c, x, y) = \max_{\substack{i \in \{0,\dots,H_k-1\} \\ j \in \{0,\dots,W_k-1\}}} \text{in\_pad}(c, \hat{x} + i, \hat{y} + j) \quad (3.5)$$

Two dimensional maximum pooling is performed as specified in Equation (3.5). Equations (3.1)–(3.3) apply to pooling as well. In contrast to convolutional layers, no mixing between channels takes place. Therefore, each channel of the output corresponds to exactly one channel of the output.

$$\text{out}(c, x, y) = \frac{1}{H_k \cdot W_k} \sum_{\substack{i \in \{0,\dots,H_k-1\} \\ j \in \{0,\dots,W_k-1\}}} \text{in\_pad}(c, \hat{x} + i, \hat{y} + j) \quad (3.6)$$

Average pooling, as specified in Equation (3.6) calculates the average of the input patch $(\hat{x}, \hat{y})$ to $(\hat{x} + H_k, \hat{y} + W_k)$ corresponding to each output index $(x, y)$. Unlike maximum pooling, in every pass each value in the input patch contributes to the result. Therefore, non-zero gradients will be passed to more input indices instead of only the index of the maximum for each output.

**Normalization Layers** Another highly popular ingredient of convolutional networks are normalization layers. Over time different normalization strategies such as Batch Normalization [IS15], Layer Normalization [BKH16], Instance Normalization [UVL16] and Group Normalization [WH18] have been suggested. Normalization is usually performed between two learnable layers of the network. During training the output distribution of the layers changes. In direct consequence the input to the next layer does as well. Therefore, in the next training iterations, following layers will have to compensate for this shift in the distribution by adjusting their parameters. Training may be sped up by stabilizing the distribution of the output and avoiding such adjustments.

Ioffe and Szegedy [IS15] refer to this as internal covariate shift in their work on Batch Normalization. They keep statistics over the mini-batch used during one training step and normalize the distribution accordingly.

$$\mu(c) = \frac{1}{B \cdot H \cdot W} \sum_{b=0}^{B-1} \sum_{x=0}^{H-1} \sum_{y=0}^{W-1} \text{in}(b, c, x, y) \tag{3.7}$$

$$\sigma^2(c) = \frac{1}{B \cdot H \cdot W} \sum_{b=0}^{B-1} \sum_{x=0}^{H-1} \sum_{y=0}^{W-1} (\text{in}(b, c, x, y) - \mu(c))^2 \tag{3.8}$$

$$\text{out}(b, c, x, y) = \gamma(c) \left( \frac{\text{in}(b, c, x, y) - \mu(c)}{\sqrt{\sigma^2(c)}} \right) + \beta(c) \tag{3.9}$$

Equations (3.7)–(3.9) show Batch Normalization for an input with channel dimension $C$ and spatial dimensions $(H, W)$. The mean and variance of each channel is computed over the batch dimension $B$ and spatial dimensions. Finally, the input "in" is shifted and scaled to center the output distribution around zero with a standard deviation of one. A learned affine transform with parameters $\gamma$ and $\beta$ is applied to ensure that the normalization can be reverted, such that the output distribution of the layer is not restricted. At inference time, expectation $\mu$ and variance $\sigma$ are set to the dataset statistics aggregated over the course of training instead of the batch statistics. Batch normalization is usually performed between the output of the convolutional layer and the activation function. As a result, the effect of the convolutional layer bias is neutralized by shifting the channel mean to zero and can be omitted. The shift parameter $\beta$ of the batch norm functionally replaces the bias.

## 3.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are one of the most common types of models for sequence modeling tasks such as text and speech processing. Such networks apply the same function with trainable weights to a hidden state and input in each time-step to update their hidden state, the memory of the network, for the next time-step. Hence, in the backward pass the gradient has to be propagated backward though time meaning through each application of the recurrent update function. Because of this, even in a single backward pass, the update function weights can accumulate large gradients since many time-steps contribute to the those same weights. This can lead to exploding or vanishing gradients, which can prevent the weights from converging or lead to infeasibly slow training [Hoc91].

**Long Short-Term Memory** To combat the exploding and vanishing gradient problem, Hochreiter and Schmidhuber [HS97] introduced a recurrent neural network architecture called Long Short-Term Memory (LSTM). Through the use of gating, it is less susceptible and can therefore be used with long sequences.

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + W_{ic}c_{t-1} + b_i) \quad \text{input gate} \tag{3.10}$$
$$f_t = \sigma(W_{fx}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \quad \text{forget gate} \tag{3.11}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c) \quad \text{cell state} \tag{3.12}$$
$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + W_{oc}c_t + b_o) \quad \text{output gate} \tag{3.13}$$
$$h_t = o_t \circ \tanh(c_t) \quad \text{hidden state} \tag{3.14}$$
$$y_t = W_{yh}h_t + b_y \quad \text{output} \tag{3.15}$$

Equations (3.10)–(3.15) show the function computed by a LSTM cell [SSB14]. This function receives a sequence of inputs $x_t$ and produces the outputs $y_t$ for time-steps $t = (1, \ldots, T)$. In this case, $\sigma$ is the logistic sigmoid function, $\circ$ is the hadamard product, weight matrices are denoted by $W$ and bias vectors by $b$. Both the cell state and the hidden state are passed on to the next time-step. Intuitively, the forget gate $f_t$ determines how much each component of the previous cell state $c_{t-1}$ contributes to the next cell state $c_t$, while the input gate $i_t$ controls the contribution of the input $x_t$ to $c_t$. Furthermore, the output gate $o_t$ scales the activated cell state controlling the hidden state $h_t$ which is then projected to form the cell output $y_t$.

**Gated Recurrent Unit** Cho *et al.* [CMBB14] presented a new variant of recurrent cell called Gated Recurrent Unit (GRU) for neural machine translation. GRU and LSTM units have been compared by Chung *et al.* [CGCB14], who conducted experiments with both units on their estimation quality for different sequence modeling tasks. They showed that both LSTM units and GRUs performed better than traditional RNNs. Depending on the task, either a LSTM or GRU model was more successful. Therefore, they could not conclude whether one of the models was strictly superior. However, GRUs have fewer gates and therefore fewer trainable parameters.

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z) \quad \text{update gate} \tag{3.16}$$
$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r) \quad \text{reset gate} \tag{3.17}$$
$$\tilde{h}_t = \tanh(W_{\tilde{h}x}x_t + W_{\tilde{h}r}(r_t \circ h_{t-1}) + b_{\tilde{h}}) \quad \text{candidate activation} \tag{3.18}$$
$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t \quad \text{hidden state} \tag{3.19}$$

The computation realized by each GRU is displayed in equations (3.16)–(3.19). As previously, inputs are denoted by $x_t$ and outputs $y_t$ may be computed as projections of the hidden state $h_t$. Furthermore, weight matrices and bias vectors are denoted by $W$ and $b$. Since there is no cell state in GRUs, the only connection to the previous time-step $t - 1$ is through the hidden state $h_{t-1}$. The reset gate $r_t$ is able to scale the contribution of the previous hidden state $h_{t-1}$ to the candidate activation $\tilde{h}_t$. Finally, the hidden state $h_t$ is an elementwise interpolation between the candidate activation $\tilde{h}_t$ and the previous hidden state $h_{t-1}$. This mixture is controlled by the update gate, which is therefore able to determine which parts of the hidden state are retained or updated. LSTM units have a similar update for their cell state, with separate forget and input gate to control the mixture of the previous and updated state, as compared to the single update gate in GRU units.

**Convolutional Gated Recurrent Unit** To preserve local information in convolutional recurrent architectures for video processing, Ballas *et al.* [BYPC16] introduced Convolutional Gated Recurrent Units (Conv-GRUs). Previous works processed the images using convolutional networks to extract features before passing them to the recurrent inputs $x_t$ of a fully connected RNN. In contrast, this work replaced the fully connected layers which are learnable affine projections inside the recurrent unit with convolutional layers.

$$z_t = \sigma(\text{Conv}([h_{t-1}, x_t], W_z)) \qquad \text{update gate} \tag{3.20}$$

$$r_t = \sigma(\text{Conv}([h_{t-1}, x_t], W_r)) \qquad \text{reset gate} \tag{3.21}$$

$$\tilde{h}_t = \tanh(\text{Conv}([r_t \circ h_{t-1}, x_t], W_h)) \qquad \text{candidate activation} \tag{3.22}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t \qquad \text{hidden state} \tag{3.23}$$

Equations (3.20)–(3.23) show the calculation performed in Conv-GRUs [BYPC16]. All fully connected layers in the GRU are replaced with convolutional layers. Weights of convolutional kernels including the bias are denoted by $W$ and $[\cdot, \cdot]$ denotes the concatenation of two tensors along the channels dimension. The convolutional layers denoted by Conv() have the same number of output channels as the gates and double the number of input channels to fit $[h_{t-1}, x_t]$.

## 3.3 Attention Mechanism

In their influential work on machine translation "Attention is all you need", Vaswani *et al.* [VSP+17] introduce Transformer networks. Convolutional and recurrent layers for processing variable-sized inputs are replaced by a new type of attention mechanism called Scaled Dot-Product Attention in their architecture. The main advantage over recurrent and convolutional networks is that information is aggregated globally and non-sequentially in a single layer. In contrast, convolutional networks require several layers and recurrent networks require multiple iterations to spread information across the full spatial size of the input.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad Q \in \mathbb{R}^{M \times d_k}, K \in \mathbb{R}^{N \times d_k}, V \in \mathbb{R}^{N \times d_v} \tag{3.24}$$

The attention is computed as displayed in Equation (3.24). The query, key and value matrices $Q$, $K$ and $V$ are stacked from multiple vectors. For example, the query matrix $Q = (q_0, \ldots, q_M)^T$ packs query vectors $q_i$ as its rows. Usually, the query, key and value vectors are learned projections of the feature vectors served as input to the attention mechanism. All pairs of query vectors $q_i$ and key vectors $k_j$ are compared via the dot product in the matrix operation $QK^T$ to form a matrix of similarity scores. Vaswani *et al.* [VSP+17] suspect that the performance of dot-product attention is diminished by large similarity scores that may be caused by the long sum of the dot-product for large vectors. Therefore, each similarity score is divided by the square root of the size of the query and key vectors $d_k$. After that, the softmax operation converts the similarity scores into weights. The operation is applied to all similarities $q_i^T k_0/\sqrt{d_k}, \ldots, q_i^T k_N/\sqrt{d_k}$ for one query vector $q_i$ such that the weights sum to one. Finally, all value vectors are weighted depending on how well their key matched with the current query vector. Therefore, the sum $(q_i^T k_0/\alpha)v_0 + \cdots + (q_i^T k_N/\alpha)v_N$ realized by the last matrix multiplication with value matrix $V$ results in a mixture of all value vectors for each query vector $q_i$.

## 3.4 Optical Flow

Optical Flow Estimation is a correspondence problem where the two sets of entities to be matched are the pixels of two consecutive images in a sequence.

$$\Omega = \{0, 1, \ldots, H-1\} \times \{0, 1, \ldots, W-1\} \qquad \text{domain} \tag{3.25}$$

$$\hat{\Omega} = \{0, 1, \ldots, 255\}^3 \qquad \text{co-domain} \tag{3.26}$$

$$I : \Omega \to \hat{\Omega} \qquad \text{digital image} \tag{3.27}$$

Discrete color images $I_1$ and $I_2$ are defined as described in equations (3.25)–(3.27). The domain $\Omega$ is a rectangular region with $HW$ samples arranged in a grid with a spacing of one. Furthermore, the co-domain $\hat{\Omega}$ is quantised and vector valued with color channels for red, green and blue. Each pixel from the domain is mapped by the images to an element from the co-domain.

Now, the goal of optical flow estimation is to find a displacement field $f(i, j) = (u, v)$ that matches each pixel $(i, j)$ of image $I_1$ to its corresponding pixel index $(i + u, j + v)$ in image $I_2$. The flow estimate $(u, v)$ may also be non-integer to match pixels $(i, j)$ with sub-pixel precision to a position between samples. In this thesis, the corresponding pixel index will usually be referred to by $(u, v)$ directly for simplicity and brevity.

## 3.5 Error Measures for Optical Flow

Error measures express the performance of a model on a specific image pair in terms of a numerical value. They can be used to evaluate and compare methods or as a loss function during training.

Oftentimes, error measures rely on norms, which also appear in other contexts in this thesis.

$$\|x\|_p = \left( \sum_{i \in \text{indices}(x)} |x(i)|^p \right)^{1/p} \qquad p\text{-norm} \tag{3.28}$$

$$\|x\|_1 = \sum_{i \in \text{indices}(x)} |x(i)| \qquad \text{absolute value norm} \tag{3.29}$$

$$\|x\|_2 = \sqrt{\sum_{i \in \text{indices}(x)} x^2(i)} \qquad \text{euclidean norm} \tag{3.30}$$

$$\|x\|_\infty = \max_{i \in \text{indices}(x)} |x(i)| \qquad \text{maximum norm} \tag{3.31}$$

The $p$-norm (3.28) is a generalized norm, where the parameter $p$ can be set to real values that are equal or larger than one. Examples for $p$-norms are the absolute value norm (3.29), the euclidean norm (3.30), and maximum norm (3.31). Although the maximum does not appear in this section, it is used to limit the extent of the local grid for the lookup operation in Section 3.6.1.

$$\text{AEE}(f_{\text{gt}}, f_e) = \frac{1}{HW} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \|f_{\text{gt}}(i, j) - f_e(i, j)\|_2 \tag{3.32}$$

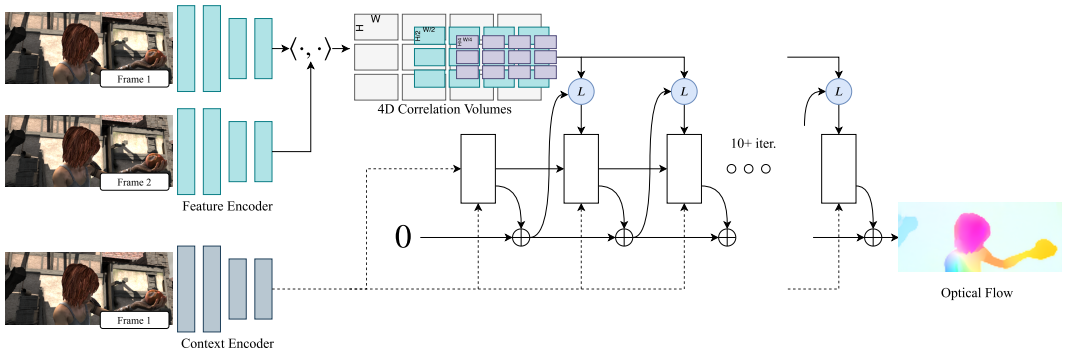


**Figure 3.2:** Overview of RAFT. Image Source: [TD20].

The Average Endpoint Error (AEE) is one of the most common quality measures for comparing optical flow estimation methods. It corresponds to the average over the euclidean distance between the ground truth and estimated flow vector at each of the pixels of image $I_1$.

$$\begin{aligned} \mathrm{TAEE}(f_{\mathrm{gt}}, f_e) &= \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} ||f_{\mathrm{gt}}(i,j) - f_e(i,j)||_1 \\ &= \|f_{\mathrm{gt}} - f_e\|_1 \end{aligned} \tag{3.33}$$

RAFT uses the $l_1$-distance between the ground truth and estimated flow field in its loss function. It can be seen as the total absolute end point error, since it uses the absolute value norm instead of the euclidean norm between two flow vectors. Furthermore, it is not normalized by the number of pixels and is therefore not suitable for comparison of methods as it is dependent on the image size.

## 3.6 RAFT: Recurrent All-Pairs Field Transforms for Optical Flow

This section gives an overview of RAFT [TD20]. RAFT is an end-to-end trainable, deep optical flow estimation method implemented in PyTorch [PGM+19]. At the date of submission, it achieved state-of-the-art estimation quality on the Sintel [BWSB12] and KITTI [GLU12; MG15] optical flow benchmark datasets. As host method of Separable Flow with the best reported results, it is and important foundation for this thesis.

Figure 3.2 shows the structure of RAFT, which can be divided into two phases. Phase one is a computationally expensive preparation phase, where all operations are only performed once to provide high quality features for the second phase. In phase two iterative updates are performed to improve the flow estimate.

**Initialization Phase** The first phase starts with the input of two frames from adjacent time-steps at full resolution $(H_{\mathrm{full}}, W_{\mathrm{full}})$. A convolutional feature encoder following the ResNet [HZRS16] structure is employed to compute image features $F_1, F_2$ at an eighth of the image resolution for both frames. Thus, the resolution of the image features is $(\lfloor 1/8 H_{\mathrm{full}} \rfloor, \lfloor 1/8 W_{\mathrm{full}} \rfloor) = (H, W)$.

Instead of the indices of the input image pixels, the spatial indices $(i, j)$ of the image features will be referred to as pixels. The channels dimension size is increased from the 3 color channels to 256 feature channels. Another identical encoder is responsible for extracting context features $F_C$ and the initial hidden state $H_0$ from frame one.

$$C(i, j, u, v) = F_1(i, j) \cdot F_2(u, v) \tag{3.34}$$

Afterward, the correlation volume is computed by dot product between the image features $F_1$ and $F_2$ of all pairs of pixels $(i, j)$ and $(u, v)$, as specified in Equation (3.34). The last two dimensions $(u, v)$ of the correlation volume correspond to the index of the pixel in $F_2$ that the pixel $(i, j)$ from $F_1$ is displaced to. By applying average pooling with kernel size $(2, 2)$ and stride $(2, 2)$ over the displacement dimensions, the displacement dimensions are halved in size. This is repeated three times and each repetition adds one volume to the correlation volume pyramid $\{C^0, C^1, C^2, C^3\}$ with $l_{\max} = 4$ levels. Because of that, the dimension size of the volumes is $(H, W, \lfloor H/2^l \rfloor, \lfloor W/2^l \rfloor)$. The displacement dimension sizes are halved from each pyramid level $l - 1$ to the next level $l$.

**Iterative Refinement Phase** Prior to the iterative update phase, the flow is initialized to zero. Alternatively, the warm-start initialization strategy initializes the flow with a forward-projection of the estimate from the frame pair of the previous time-step. Each iteration of the second phase starts by performing the lookup operation described in Section 3.6.1. The lookup operation combines the current flow estimate with the correlation volume to extract values from the volume reflecting the current search progress at each index of $F_1$. Since the lookup operation is performed on each pyramid level, the resulting multi-scale correlation features are of shape $(H, W, l_{\max} \cdot (2r + 1)^2)$, where $r$ is the lookup radius. After that, the motion feature encoder applies two convolutional layers to both the flow and lookup features and combines the result using another convolutional layer.

$$z_t = \sigma(\text{Conv}_{3\text{x}3}([h_{t-1}, x_t], W_z)) \tag{3.35}$$

$$r_t = \sigma(\text{Conv}_{3\text{x}3}([h_{t-1}, x_t], W_r)) \tag{3.36}$$

$$\tilde{h}_t = \tanh(\text{Conv}_{3\text{x}3}([r_t \circ h_{t-1}, x_t], W_h)) \tag{3.37}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t \tag{3.38}$$

Equations (3.35)–(3.38) show the operations of the update module, which is a 2D convolutional GRU (see Section 3.2) with kernel size $(3, 3)$. In every time-step, the recurrent update module receives the concatenation of the context features $F_C$ and current motion features $F_{\text{motion}}$ as well as the previous hidden state $H_{t-1}$. The output is the new hidden state $H_t$ for the current time-step.

Two additional convolutional layers are applied to $H_t$ to arrive at the flow update $\nabla f_{t-1}$ which is at $1/8$ of the image resolution and the same resolution as the image features. Next the flow can be updated as follows: $f_t = f_{t-1} + \nabla f_{t-1}$. At inference time only the final, up-sampled flow after the last refinement step is returned. In contrast, during training the flow estimate at every refinement step is up-sampled to image resolution and returned.

**Loss Function** The loss function for RAFT has to map the series of flow estimates $(f_1, ..., f_N)$ emitted by the model to a scalar loss value. This is accomplished by accumulating the deviations from the ground truth flow of all flow estimates in the series.

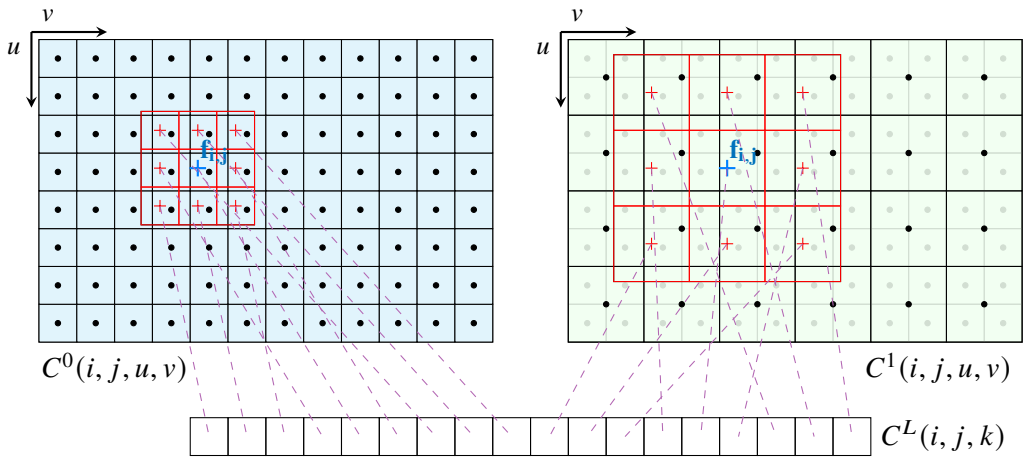


**Figure 3.3:** Illustration of the RAFT lookup operation.

$$\mathscr{L} = \sum_{i=1}^{N} \gamma^{N-i} \|\mathbf{f}_i - \mathbf{f}_{\text{gt}}\|_1 \tag{3.39}$$

For each estimate in the sequence, the $l_1$ distance to the ground truth flow is assigned as its sub-loss. All sub-losses are part of a weighted sum, where the weights $\gamma^{N-i}$ increase exponentially from one member to the next. The estimate after the first refinement step $f_1$ is assigned the lowest weight of $\gamma^{N-1}$ whereas the final estimate $f_N$ is weighted with 1. During the training of RAFT the weight decay is set to $\gamma = 0.8$.

### 3.6.1 The Correlation Volume Lookup Operation

The correlation volume lookup is an operation that can be seen as a replacement for warping the second frame prior to computing cost correlation volume with a limited displacement range. It collects values for each index $(i, j)$ in $C(i, j, u, v)$ on a local grid around the current flow estimate $f_t(i, j) = (\tilde{u}, \tilde{v})$ within a radius $r$.

Figure 3.3 illustrates this lookup operation. The blue grid corresponds to correlation volume $C^0(i, j, u, v)$ at pyramid level $l = 0$ whereas the green grid corresponds to level $l = 1$. Each grid represents one slice of the correlation volume for some index $(i, j)$. Consequently, each cell corresponds to an integer valued displacement index $(u, v)$. Since $C^1(i, j, u, v)$ is one step below the largest pyramid level, it is the result of average pooling along the displacements dimensions of level zero which is represented by doubling the cell size and the faded subdivision. In this case, $C^0$ would be of shape $(H, W, H, W)$, while $C^1$ would have the shape $(H, W, \lfloor H/2 \rfloor, \lfloor W/2 \rfloor)$. Local grids are shown as red cells with crossed out centers. The local grid surrounds the flow estimate $f_{i,j}$, with increments of one between grid cell centers. As the cell size increases with each level, so does the lookup radius relative to the cells of the largest pyramid level ($l = 0$). Since $f_{i,j}$ and all grid centers will usually be non-integer, bilinear interpolation is used to sample values from the correlation volumes. Samples from all levels are concatenated, where each level contributes $(2r + 1)^2$ samples per index. The resulting tensor contains $l_{\text{max}} \cdot (2r + 1)^2$ per index $(i, j)$ and is therefore of shape $(H, W, l_{\text{max}} \cdot (2r + 1)^2)$.

$$(C^L)^l(i,j,\mathrm{dx}) = \mathrm{sample\_bilinear}(C^l(i,j), \frac{1}{2^l} f(i,j) + \mathrm{dx}) \tag{3.40}$$

$$\mathrm{dx} \in \mathcal{N} = \{\mathrm{dx} | \mathrm{dx} \in \mathbb{Z}^2, \|\mathrm{dx}\|_\infty \leq r\} \tag{3.41}$$

Equations (3.40) and (3.41) describe the sampling process and grid size. For the each index $(i, j)$ and grid node offset dx, a sample is taken from $C^l(i, j)$ at a sampling point. Each sampling point is calculated from the flow estimate $f(i, j) = (\tilde{u}, \tilde{v})$ which is scaled by $1/2^l$ to fit the scale of the pyramid level $l$. The flow estimate $f$ refers to the sub-pixel correspondence location $(\tilde{u}, \tilde{v})$ in $F_2$. Then the grid offset dx is added to move the sampling point to the corresponding local cell center. Equation (3.41) states that each grid offset is an integer valued two dimensional vector where the absolute value of each of the entries is less than or equal to the lookup radius $r$.

### 3.6.2 Alternative Lookup: On-Demand 4D Correlation Volume Computation

The lookup operation can be performed directly on the image features $F_1$ and $F_2$ instead of the 4D correlation volume $C$. This is motivated by the fact that the multi-scale correlation features at each iteration step for each index $(i, j)$ only need the indices around the current flow estimate. Therefore, by computing the values of $C$ on demand as needed by the lookup operation, the time complexity can be reduced from

$$O\big((HW)^2 F + MHW \cdot l_{\max} \cdot (2r+1)^2\big) \qquad \text{to} \qquad O\big(MHWF \cdot l_{\max} \cdot (2r+1)^2\big)\,. \tag{3.42}$$

The first time complexity in Equation (3.42) has two parts. The number of pixels squared $(HW)^2$ scalar products of the image features of length $F$ need to be calculated because the full correlation volume is computed during the initialization phase. During the refinement phase, for each of the $M$ iterations, lookup operations on each flow estimate $(HW)$ and level $(l_{\max})$ with correlation radius $r$ are performed.

For the alternative lookup, no all-pairs features correlation needs to be computed during the initialization phase. However, for the lookup in each refinement iteration, the correlation volume indices need to be computed on demand. In consequence, the complexity of the features scalar product contributes the additional factor $F$.

$$C^l(i,j,m,n) = \mathrm{avgPool2D}(C(i,j), \mathrm{kernel} = (2^l, 2^l), \mathrm{stride} = (2^l, 2^l))(m,n) \tag{3.43}$$

$$= \frac{1}{2^{2l}} \sum_{p=0}^{2^l-1} \sum_{q=0}^{2^l-1} C(i,j,2^l m + p, 2^l n + q) \tag{3.44}$$

$$= \frac{1}{2^{2l}} \sum_{p=0}^{2^l-1} \sum_{q=0}^{2^l-1} F_1(i,j) \cdot F_2(2^l m + p, 2^l n + q) \tag{3.45}$$

$$= F_1(i,j) \cdot \frac{1}{2^{2l}} \Big( \sum_{p=0}^{2^l-1} \sum_{q=0}^{2^l-1} F_2(2^l m + p, 2^l n + q) \Big) \tag{3.46}$$

$$= F_1(i,j) \cdot \mathrm{avgPool2D}(F_2, \mathrm{kernel} = (2^l, 2^l), \mathrm{stride} = (2^l, 2^l))(m,n) \tag{3.47}$$

Equations (3.43)–(3.47) were adapted with heavy modifications from [TD20] and form the basis for Chapter 5. They show how values of the indices from each pyramid of $C^l(i, j, m, n)$ are calculated. The first line (3.43) corresponds to computing the correlation volume pyramid, where the full

volume is stored and pooled. At each level the step size of the pooled volume relative to the image features $F_2$ increases by a factor of $2^l$ which is reflected in the indices of the second line (3.44) of the equation. The average pooling is represented by sums over the size of the pooling window and dividing by the number of indices covered. In line three (3.45), the correlation volume $C$ is expanded with the on-demand computation of the same correlation volume index. Since the indices of $F_1(i, j)$ do not change, it is a constant factor that can be moved out of the sum in line four (3.46). As the average pooling formulation remains unchanged, the normalizing factor and sums can be replaced by the corresponding pooling function in line five (3.47).

As a result, the alternative lookup implementation requires to build a feature pyramid of $F_2$ during the initialization phase. Compared to the regular implementation of computing and pooling $C$, this is inexpensive. The alternative implementation is particularly attractive for large images, because the factor $(HW)^2$ of the regular implementation grows quadratically compared to $(HW)$ of the alternative implementation which is linear in the number of pixels. Still, computing the full correlation volume $C$ from $F_1$ and $F_2$ for large images of size $(1088, 1920)$ only took 17% of total inference time. Therefore, even for large images, the regular computation of $C$ is not a bottleneck [TD20].

## 3.7 Separable Flow: Learning Cost Volumes for Optical Flow Estimation

Separable Flow [ZWPT21] is an end-to-end trainable cost correlation volume extension. The method is designed to be applicable to any method that uses cost correlation volumes. Therefore, in some cases, the paper describes the general structure of the method to keep it universally applicable. In other cases it is more specific to RAFT since they evaluated their strategy primarily in combination with RAFT. For example, motion regression as flow initialization strategy for the RAFT refinement module is discussed, however the paper does not explain how the correlation volume lookup is performed for 3D correlation volumes. Like RAFT, Separable Flow is implemented using PyTorch [PGM+19].

This section describes the structure of Separable Flow when used with RAFT, while staying as close as possible to the paper.

To illustrate the initialization phase of Separable Flow, Figure 3.4 displays the data dependencies between the most important intermediary results. Cost volume separation, three dimensional cost volume aggregation and motion regression are the main contributions of the method. They give rise to the addition of aggregation weights $W_{\text{SGA}}^{C_u}$ and $W_{\text{SGA}}^{C_v}$ as image features in the second block from the top as well as the complete 3D correlation volume block shown in orange. The following sections will describe these additions.

**Cost Volume Separation** The self-adaptive cost separation of the 4D cost volume into two 3D cost volumes is the main innovation that is responsible for the name of the method. In RAFT, the 4D cost correlation volume contains scores for each pixel $(i, j)$ in frame one to match with any pixel $(u, v)$ in frame two. However, Separable Flow scores the coordinates $u$ and $v$ of the matching

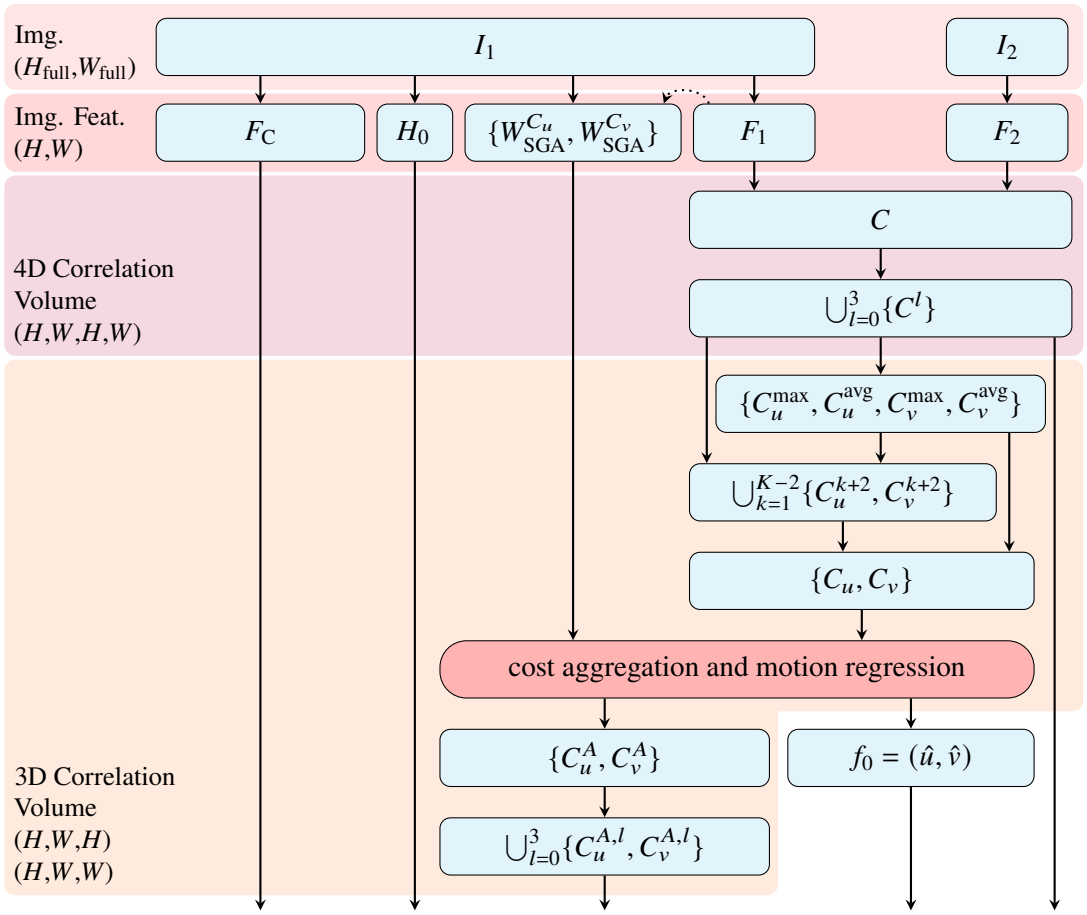


**Figure 3.4:** Data dependencies between tensors during the initialization phase of Separable Flow with RAFT as base method.

pixel in frame two independently for pixel $(i, j)$ in frame one. Therefore, the first volume $C_u$ only contains scores for $(i, j, u)$ and the second volume $C_v$ for $(i, j, v)$.

$$\text{separate} : C \in \mathbb{R}^{H\times W\times|U|\times|V|} \rightarrow C_u \in \mathbb{R}^{H\times W\times|U|\times K}, C_v \in \mathbb{R}^{H\times W\times|V|\times K} \tag{3.48}$$

Equation (3.48) shows the domains of the 4D and 3D cost volumes as inputs and outputs of the separate function. In this case, $(H, W)$ is the size of the image features, $(|U|, |V|)$ is the displacement dimension size and $K$ the number of 3D correlation feature channels. The displacement dimensions extent $(|U|, |V|)$ may differ from the spatial size $(H, W)$ of the image features $F_1$ and $F_2$ for example due to down-sampling in the correlation volume pyramid.

$$C_u^1(i,j,u) = \frac{1}{|V|}\sum_{v\in V} C(i,j,u,v) \qquad C_v^1(i,j,v) = \frac{1}{|U|}\sum_{u\in U} C(i,j,u,v) \tag{3.49}$$

$$C_u^2(i,j,u) = \max_{v\in V}(C(i,j,u,v)) \qquad C_v^2(i,j,v) = \max_{u\in U}(C(i,j,u,v)) \tag{3.50}$$

The first two channels of the 3D correlation volumes correspond to the average and maximum statistics over one of the displacement dimensions of $C$, as shown in equations (3.49) and (3.50). If the correlation volume has the subscript $u$, then $C_u$ maintains the corresponding dimension and

the complementary dimension that is referred to by index $v$ is summarized by its maximum or average.

$$A_u = \phi_u(C_v^{1:2}) \in \mathbb{R}^{H\times W\times|V|\times K-2} \qquad A_v = \phi_v(C_u^{1:2}) \in \mathbb{R}^{H\times W\times|U|\times K-2} \tag{3.51}$$

So far, there is no learnable component to the 3D correlation volume. This is changed by Equation (3.51), which introduces learned attention tensors $A_u$ and $A_v$. Each of them is computed by applying one of the three dimensional convolution layers $\phi_u$ and $\phi_v$. Unfortunately, parameters such as kernel size of the convolutional layer are not specified. Because each of the remaining $K-2$ correlation volume channels requires its own attention weights, the number of convolution output channels is set to $K-2$. Similarly, the size of the input channels is determined by the input tensor $C_u^{1,2}$ with a channel dimension of 2. The convolution layers process the summary of the complementary displacement dimension $C_v^{1,2}$ for $C_u^{3:K}$ instead of the full 4D correlation volume $C$ for efficiency. For kernels sizes that are larger than $(1, 1, 1)$, the attention weights may integrate correlation values from neighboring pixels or the displacement along the their complementary displacement dimension.

$$\begin{aligned} C_u^k(i,j,u) &= \sigma(A_u^{k-2}(i,j,:)) \cdot C(i,j,u,:) \\ C_v^k(i,j,v) &= \sigma(A_v^{k-2}(i,j,:)) \cdot C(i,j,:,v) \end{aligned} \tag{3.52}$$

Finally, $C_u^{3:K}$ results from a sum weighted by the attention $\sigma(A_u^k(i,j,:))$ over the $(i,j,u)$ slice of the 4D correlation volume $C(i,j,u,:)$ for the channels $k \in \{3,\dots,K\}$. By applying the softmax operation $\sigma(\bullet)$ to the weights, they are normalized such that all weights are positive and the sum along the displacement dimension is one. The attention-based framework allows for arbitrary sizes of $U$ and $V$ since the statistics and convolution stencil only depend on the channels dimension to stay fixed, which is independent of $|U|$ and $|V|$. This, for example, allows using the same convolutional layers and thus the same separation module on each level of the correlation volume pyramid of RAFT with varying displacement dimension sizes.

As shown in Figure 3.4, the 4D correlation volume phase results in a 4D correlation volume pyramid. To make use of the advantages of the multi-scale correlation volumes, the correlation volumes $C^l$ at every level $l$ may be separated. Because the Separable Flow paper does not consider the 4D correlation volume pyramid, the following approach to make use of all $C^l$ is not described in the paper. However it is part of the implementation. After separating all $C^l$ into $C_u^l$ and $C_v^l$, the displacement dimensions are up-sampled and concatenated as follows.

$$C_u^l \in \mathbb{R}^{H\times W\times\lfloor H/2^l\rfloor\times K} \xrightarrow{\text{up-sampling}} \tilde{C}_u^l \in \mathbb{R}^{H\times W\times H\times K} \tag{3.53}$$

$$C_u^{l=0}, \tilde{C}_u^{l=1}, \dots, \tilde{C}^{l=l_{\max}} \in \mathbb{R}^{H\times W\times H\times K} \xrightarrow{\text{concatenation}} C^u \in \mathbb{R}^{H\times W\times H\times(l_{\max}\cdot K)} \tag{3.54}$$

By up-sampling, as can be seen in Equation (3.53), all multiscale 3D correlation volumes are up-sampled to have the same spatial size of $(H, W, H, K)$. Because the spatial dimensions match, concatenation across the 3D correlation channels dimension can be applied to merge all levels $l$ into a single tensor. This tensor corresponds to one of the 3D correlation volumes $C_u$ and $C_v$ which contain the $K = 4$ features for each level of the correlation volume pyramid with $l_{\max}$ levels. The resulting shapes are $(H, W, H, (l_{\max} \cdot K))$ and $(H, W, W, (l_{\max} \cdot K))$. The single-tensor representation is especially convenient for the next step because all levels can be aggregated on the same resolution, eliminating the need for pyramidal aggregation.

**Correlation Volume Aggregation** To aggregate non-local information from the 3D correlation volumes, an end-to-end trainable aggregation module inspired by semi-global matching is applied called semi-global guided aggregation. This aggregation method was introduced by Zhang *et al.* [ZPYT19] and is intended for correlation cost volume aggregation in stereo matching networks. However, the similarity between the stereo correlation volumes and the separated multi-channel 3D correlation volumes for optical flow allows the aggregation to be applied in both cases. The 3D correlation volumes have one displacement dimension and multiple channels, like the ones in GA-Net [ZPYT19].

In Separable Flow, the cost aggregation network uses four global aggregation layers and eight 3D convolutional layers in an encoder-decoder configuration. Separate cost aggregation modules are applied to the correlation volumes $C_u$ and $C_v$. Resulting aggregated correlation volumes are denoted by $C_u^A$ and $C_v^A$ and are of shape $(H, W, |U|)$ and $(H, W, |V|)$. Accordingly the aggregation module contracts the 3D correlation volume channels into a single channel.

Figure 3.4 displays the aggregation of the cost volumes as a red box for RAFT. The cost volume aggregation module has two inputs: Aggregation Weights $\{W_{\text{SGA}}^{C_u}, W_{\text{SGA}}^{C_v}\}$ and 3D correlation volumes $\{C_u, C_v\}$. The origin of the weights for the aggregation volumes is not mentioned in the paper. However, in the implementation the weights created by passing image $I_1$ and image features $F_1$ through a residual 2D convolutional network. Guidance of shape $(H, W, 20)$ with 5 filter weights for each of the 4 filter directions is emitted.

**Motion Regression** For initializing the flow $f_0$ and to provide an additional term to the loss connected directly to the 3D correlation volume, motion regression is employed.

$$\mathbf{f}_0 = \{\hat{u}, \hat{v}\} \qquad \hat{u} = U \cdot \sigma(C_u^A(i, j, :)) \qquad \hat{v} = V \cdot \sigma(C_v^A(i, j, :)) \tag{3.55}$$

The initial flow $f_0$ is assigned the concatenated displacement estimates for both channels. Each channel is the result of a sum over all integer disparity candidates $U$ and $V$ weighted by the softmax of the aggregated 3D correlation volumes $C_u^A$ and $C_v^A$. Thus, the disparity value associated with the largest correlation score contributes the most to the final estimated displacement.

**Lookup Operation for 3D Correlation Volumes** In the same way as RAFT, the lookup operation has to be performed during the motion refinement phase. However, Separable Flow differs from RAFT as it replaces the 4D correlation volume by the 3D correlation volumes. Therefore the lookup operation has to be adapted to work with 3D correlation volumes. Again, the lookup for 3D correlation volumes is not specified in the paper, since it is specific to using RAFT as base method.

$$(C_u^L)^l(i, j, \text{dx}) = \text{sample\_linear}(C_u^l(i, j), \frac{1}{2^l} f(i, j, 0) + \text{dx}) \tag{3.56}$$

$$(C_v^L)^l(i, j, \text{dx}) = \text{sample\_linear}(C_v^l(i, j), \frac{1}{2^l} f(i, j, 1) + \text{dx}) \tag{3.57}$$

$$\text{dx} \in \mathcal{N} = \{\text{dx} | \text{dx} \in \mathbb{Z}^1, \|\text{dx}\|_\infty \leq r\} \tag{3.58}$$

The implementation provides code for the 1D lookup operation. They make only minimal changes to adapt the code for the 2D lookup to work with the 3D correlation volumes $C_u$ and $C_v$. Equations (3.56) and (3.57) show how the lookup is performed on ever level $l$ of the correlation

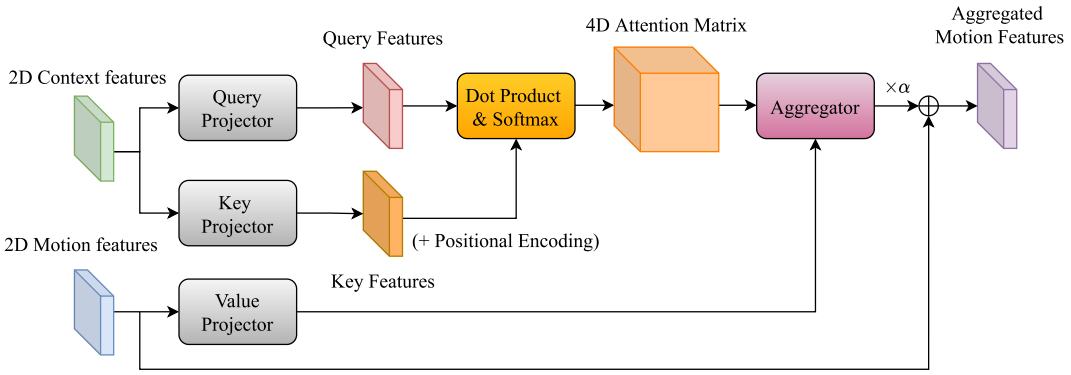


**Figure 3.5:** Overview of Global Motion Aggregation (GMA). Image Source: [JCL+21].

volume pyramid. The only difference is that all tensors have one dimension less. Firstly, the 3D correlation volume $C_u$ has one dimension less than $C$. Furthermore, the sampling point as sum between the flow estimate and local grid offset is 1D instead of 2D. For $C_u$, only the first component of the flow $f(i, j, 0)$ is picked and the corresponding offset dx is scalar as well. As a result of 1D lookup, the size of the lookup features is not quadratic in the lookup radius $r$ but it is linear. In consequence the resulting shapes of the lookup over all levels $C_u^L$ and $C_v^L$ are of shape $(H, W, l_{\max} \cdot (2r + 1))$ where $l_{\max}$ is the number of 3D correlation volume pyramid levels.

**Loss Function** Similar to RAFT, a loss function with multiple terms and exponentially increasing weights is used.

$$\mathscr{L} = \sum_{i=0}^{N} \gamma^{N-i} \|\mathbf{f}_i - \mathbf{f}_{\text{gt}}\|_1 \tag{3.59}$$

Equation (3.59) shows the loss function of Separable Flow. An important difference to the loss function of RAFT in Equation (3.39) is the inclusion of the initial flow $f_0$ from motion regression. The loss term for $f_0$ is integrated by starting the sum at index $i = 0$ instead of $i = 1$. Therefore, the intial flow has the smallest weight of all terms with $\gamma^N$ where gamma is chosen as $\gamma = 0.8$.

## 3.8 Learning to Estimate Hidden Motions with Global Motion Aggregation

Global Motion Aggregation [JCL+21] provides an aggregation module for the motion features of RAFT. The goal is to propagate motion information to occluded pixels from similar pixels according to the context features of image $I_1$. To identify such pixels, a similiarity measure needs to be defined.

$$\theta(x_i) = W_{\text{qry}} x_i \qquad \phi(x_i) = W_{\text{key}} x_i \qquad \sigma(x_i) = W_{\text{val}} y_i \tag{3.60}$$

This is achieved by first projecting the context features using individual learned projection matrices $W$ as can be seen in Equation (3.60).

A relative positional encoding might be added to the key features, however this has been shown to worsen the results in most cases. Afterwards, the resulting query and key features are compared using the pixel-pairwise dot product, as shown by the yellow box in Figure 3.5.

$$f(a_i, b_j) = \frac{\exp(a_i^T b_j / \sqrt{D})}{\sum_{j=1}^{N} \exp(a_i^T b_j / \sqrt{D})} \tag{3.61}$$

Equation (3.61) shows how the scaled dot product attention is computed for each pair of query features $a_i$ and key features $b_j$, where $D$ is the feature size. The equation is a reformulation of Equation (3.24), where the softmax is applied to the scaled dot-product. In this case, the softmax is written out explicitly as a fraction of the exponentiated products. Furthermore, the query and key features are not stacked into matrices and the value features are not directly aggregated. By storing the results for each pair of features, each contributing two spatial dimensions, the four dimensional attention matrix is created.

$$\hat{y}_i = y_i + \alpha \sum_{j=0}^{N} f(\theta(x_i), \phi(x_j))\sigma(y_j) \tag{3.62}$$

Finally, the projected motion features are aggregated for each pixel $i$ with context features $x_i$ and motion features $y_i$. Pixels with matching context feature projections have larger attention matrix entries. Hence, their motion feature vector projections contribute more to the sum and are therefore more impactful. As a last step all motion feature aggregates are scaled by the same learned factor and added to the original motion features $y_i$. The resulting aggregated motion features are denoted by $\hat{y}_i$.

Instead of $[y_i, x_i]$ as in RAFT, the convolutional GRU update module now receives $[y_i, \hat{y}_i, x_i]$. This means that the update module receives both the aggregated and original motion features. Jiang *et al.* [JCL+21] argue that their model may be able to learn selecting between the local flow estimate and the globally aggregated estimate, depending on whether the local information is sufficient.

## 3.9 General Purpose GPU Computing

This section explains the basics of GPU computing that are relevant to this thesis. It is based on NVIDIA GPUs as well as the CUDA general-purpose parallel computing platform [NVI22]. GPUs have an architecture that is generally tailored towards the single instruction, multiple data paradigm and parallel processing. This means that the same operation can be performed efficiently with many different values as arguments in parallel. Consequently, they are especially suitable for tasks which can benefit from thousands of parallel threads.

**Streaming Multiprocessor** The smallest unit of computation on NVIDIA GPUs is the Streaming Multiprocessor (SM), which runs a group 32 threads called warp in lockstep. This means that all threads in the warp execute the same instruction at the same time. Branching execution paths caused for example by if-statements can lead to each of the paths being processed in sequence. Therefore, divergence inside the warp should be avoided to keep all threads running in parallel.

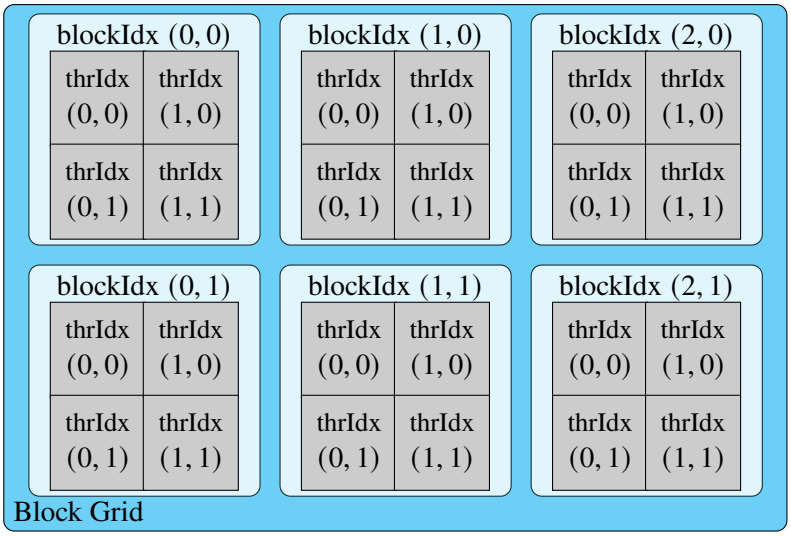


**Figure 3.6:** Example for a thread block grid layout with a block dimension of $(2, 2)$ and a block grid of size $(3, 2)$.

**Thread Block Grids** To organize and schedule the distribution of threads onto SMs, the concept of thread blocks and grids is introduced. Thread blocks are groups of threads which run on the same SM. They are organized in a one, two or three dimensional grid. Because of this, each thread has the builtin 1D, 2D or 3D vectors "threadIdx", "blockIdx" and "blockDim". These vectors are assigned such that each thread can identify the index of its block within the grid and its own index within the block. Usually all threads run the same same program with the same arguments. Hence, these variables are the main source of information which can be used during programming to ensure that all threads operate on different data or even behave differently. Figure 3.6 shows an example where the threads per block and number of blocks are configured as follows:

$$\text{blockDim} = (2, 2) \qquad \text{numBlocks} = (3, 2) \tag{3.63}$$

In this example, the task may be to perform some operation for each pixel of an image of size $(6, 4)$. Then, each thread would calculate its pixel index as

$$\begin{aligned} \text{pixelIdx.x} &= \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x} \\ \text{pixelIdx.y} &= \text{blockIdx.y} \cdot \text{blockDim.y} + \text{threadIdx.y} \end{aligned} \tag{3.64}$$

By using the pixel index, each thread is able to identify and therefore read and write values of the image array relative to its pixel index. An additional blocks dimension may be added with the size of the number of images $B$ in the batch to process multiple images in the same way. In this case the block dimension would be $(B, 2, 2)$.

**CUDA Kernels** The program that runs in parallel on the GPU is called kernel, which is implemented as a templated C++ function. Its template parameters determine the number of blocks and the shape of each thread block as discussed in the previous example. Hence they have to be known at compile time. As soon as the kernel is started, its blocks are scheduled to run on the SMs of the GPU. Blocks of the same kernel may be scheduled in an arbitrary order and run at different times on the same SM or on different SMs in parallel.

**Memory** GPUs usually have a large global memory and multiple levels of caches with increasing size. Here, the focus will lie on the global and shared memory. The global memory has a size in the order of Gigabytes and is accessible by all threads on the same GPU. Its access latency is much higher than the one of the shared memory. In this thesis, the kernel will usually receive references to the input and output tensors in global memory. Each thread can then read from the input tensors and write the result to its assigned index in the output tensor. In contrast to the global memory, the shared memory is small as it is usually configured to be 48KB by default. Shared memory is much faster than global memory, as it shares its physical hardware with the L1 cache. Also, it can only be accessed by threads of in the same thread block. Because of that, it is used in this thesis to share values loaded from global memory as well as intermediary results between the threads.

**Synchronization** Threads can be synchronized on different levels. Since the number of threads in a block may exceed the number of threads in a warp, threads in the same block may not always run in lockstep. To avoid data races between threads of the same block, synchronization on the block level can be applied. Block level synchronization is an operation that prevents threads from advancing past the synchronization point until all threads have arrived at the this point. For example, synchronization can be applied in case all threads first write the shared memory at their index before all threads may read arbitrary indices. By performing synchronization after the writing phase, threads are forced to wait until all indices of the shared memory have been updated. Consequently, the following read operations are guaranteed to read updated values.

# 4 Investigation of Differences between Paper and Implementation

This chapter investigates differences between the paper and its implementation. The implementation that accompanies the paper differs in several aspects from its description. One of them is the training schedule and the settings that are used during training. Furthermore, several modifications to the model structure could be identified in the implementation. These are described in the second section of this chapter.

## 4.1 Differences in the Training: Parameters and Schedule

The training parameters and order of datasets have significant impact on the training outcome and final estimation quality. When investigating the training schedule and parameters, significant changes were discovered. Therefore, this section is dedicated to describing the differences between the parameters of the paper and the implementation.

All changes are marked in Table 4.1. The dataset schedule shown in the first three columns remains unchanged. During the first and second stage, training is performed solely on the Flying Chairs [DFI+15a] (C) and Flying Things [MIH+16] (T) datasets. The third stage is called the Sintel [BWSB12] finetuning stage, where samples from the Sintel (S), KITTI [MG15] (K), HD1K [KNH+16] (H), Chairs and Things datasets are mixed. All datasets are combined in a weighted mixture which may be used to balance the different number of samples in each dataset. Finally, stage four draws samples from the KITTI dataset and is called KITTI finetuning.

| **Schedule** | | | **Training Parameters** | | | | |
|---|---|---|---|---|---|---|---|
| **Stage** | **Weights** | **Data** | **LR** | **BS** | **WD** | **Crop Size** | **Iterations** |
| Chairs | - | C | 4e-4 | 12 | 1e-4 | $[368, 496] \rightarrow [\mathbf{320}, \mathbf{448}]$ | 100K $\rightarrow$ **50K** |
| Things | Chairs | T | 1.25e-4 | 6 $\rightarrow$ **8** | 1e-4 | $[400, 720] \rightarrow [\mathbf{448}, \mathbf{768}]$ | 100K |
| Sintel | Things | C+T+S+K+H | 1.25e-4 | 6 $\rightarrow$ **8** | 1e-5 | $[368, 768] \rightarrow [\mathbf{384}, \mathbf{832}]$ | 100K |
| KITTI | Sintel | K | 1e-4 | 6 $\rightarrow$ **8** | 1e-5 | $[288, 960] \rightarrow [\mathbf{320}, \mathbf{1024}]$ | 50K |

**Table 4.1:** Differences in the training schedule and parameters between the paper and the **implementation**. The table contains the training Stage, from which stage to load the Weights, the Datasets to use for training, learning rate (LR), batch size (BS), weight decay (WD), the image crop size and number of refinement iterations.

The number of training iterations is is shown in the last column. One training iteration corresponds to one parameter optimization step for a batch of samples. In the paper, 100K iterations are specified for the first dataset in the schedule, however the implementation uses 50K iterations. Moreover, the batch size increased from 6 to 8 frame pairs. This change was most likely made to ensure that each of the 4 GPUs receives uniformly sized batch-splits of size 2.

Another change is in the image patch size, which specifies the height and width of the patches that are extracted from the dataset during data augmentation. The model structure dictates that the input frames spatial dimension sizes are divisible by 64. This means that it is impossible to use the patch sizes specified in the paper because some of their dimensions do not fulfill the criterion. Most likely, the divisibility criterion is the reason for the authors using different patch sizes which fulfill the criterion while still being close to the ones specified in the paper. The part of the model that creates the divisibility constraint is the convolutional encoder-decoder network in the 3D correlation volume aggregation module. Like RAFT, the image features have spatial dimension $(1/8 H_{\text{full}}, 1/8 W_{\text{full}})$. At the lowest level, the encoder-decoder network down-samples this dimension to $(1/64 H_{\text{full}}, 1/64 W_{\text{full}})$. If $H$ and $W$ are not divisible by 64, then the remainder of the division is lost, resulting in a smaller spatial dimension after up-sampling. Consequently, the spatial dimensions may not match with previous tensor sizes prior to down-sampling. In this case, subsequent operations like elementwise addition that rely on identical spatial dimension of the tensor operands are impossible. Therefore, the model can only be applied to patches which fulfill the divisibility criterion.

The loss function of the implementation also differs from the paper. According to the paper, the motion regression step returns only the final motion regressed flow. The following equation with $N = 12$ for 12 motion refinement flows $\mathbf{f}_1, \ldots, \mathbf{f}_N$ and the additional final motion regression flow $\tilde{\mathbf{f}}_0$ was published.

$$w(i) = \gamma^{N-i} \quad \text{for} \quad i \in \{0, 1, \ldots, N\} \tag{4.1}$$

$$\mathscr{L} = \gamma^N \|\tilde{\mathbf{f}}_0 - \mathbf{f}_{\text{gt}}\|_{\mathbf{1}} + \sum_{i=1}^{N} w(i) \|\mathbf{f}_i - \mathbf{f}_{\text{gt}}\|_{\mathbf{1}} \tag{4.2}$$

$$\text{weights} = [\gamma^N, \ldots, \gamma^1, 1] \tag{4.3}$$

The motion regression step in the implementation returns two additional flow estimates $\tilde{\mathbf{f}}_{(-1)}, \tilde{\mathbf{f}}_{(-2)}$ computed by motion regression from intermediary steps of the aggregation of the correlation volume. To incorporate the additional flow estimates, the following loss equation is provided.

$$w(i) = 0.5 - \gamma^N + \gamma^{N-i} \quad \text{for} \quad i \in \{0, 1, \ldots, N\} \tag{4.4}$$

$$\mathscr{L} = 0.1\|\tilde{\mathbf{f}}_{(-2)} - \mathbf{f}_{\text{gt}}\|_{\mathbf{1}} + 0.3\|\tilde{\mathbf{f}}_{(-1)} - \mathbf{f}_{\text{gt}}\|_{\mathbf{1}} + 0.5\|\tilde{\mathbf{f}}_0 - \mathbf{f}_{\text{gt}}\|_{\mathbf{1}} + \sum_{i=1}^{N} w(i) \|\mathbf{f}_i - \mathbf{f}_{\text{gt}}\|_{\mathbf{1}} \tag{4.5}$$

$$\text{weights} = [0.1, 0.3, 0.5, 0.5 - \gamma^N + \gamma^{N-1}, \ldots, 0.5 - \gamma^N + \gamma^1, 1.5 - \gamma^N] \tag{4.6}$$

The weights array lists the weights for all 13 or 15 flow estimates. In both cases, gamma is assigned to be 0.8 or 0.85 depending on the training stage. Therefore, the implemented loss function weights the final motion regressed initial flow estimate much stronger with a weight of 0.5 instead of $0.8^{12} \approx 0.07$ for $\gamma = 0.8$ and $N = 12$.

In the evaluation parts of each section, two training schedules of different lengths appear. Namely, the full schedule (C+T+S+K+H) consisting of three or four training stages, including the Sintel optionally also the KITTI finetuning stages. Second, the pre-training schedule (C+T) which contains only the initial two stages.

| Model | | | | Sintel | | KITTI | |
|---|---|---|---|---|---|---|---|
| **Method** | **#Par** | **Chairs It.** | **Data** | **clean** | **final** | **epe** | **F1** |
| RAFT | 5.257M | 100K | C+T | 1.43 | 2.71 | 5.04 | 17.4 |
| SepFlow | 6.0M | 100K | C+T | **1.30** | **2.59** | **4.60** | **15.9** |
| Downloaded | 8.346M | n/a | C+T | 1.30 | 2.60 | 4.61 | 15.9 |
| Provided | 8.346M | 50K | C+T | 1.55 | 2.89 | 5.69 | 19.6 |
| Provided | 8.346M | 100K | C+T | 1.31 | 2.68 | 5.08 | 17.0 |

**Table 4.2:** Changing Training Schedule: 50K and 100K chairs training iterations

To train the model significant amounts resources are required, especially regarding the GPU memory and computation time. It can be trained using multiple GPUs. Training one model for one ablation step takes up to five days on two NVIDIA RTX A6000 GPUs, while it takes 2.5 days on four GPUs of this type. Thus, performing ablations requires substantial amounts of computation time even on expensive hardware. In consequence, the number of ablation steps that can be trained is limited. Due to the limited training time available ablations are only performed on the pre-training schedule, as is also the case in the publications of RAFT and Separable Flow. For the same reason, no experiments were conducted for reverting the loss function to that of the paper. All experiments were conducted with the loss function that came with the implementation, which is the second loss function specified by equations (4.4)–(4.6). The estimation quality of the ablation steps is evaluated using the Sintel and KITTI training dataset. Because their ground truth flow is publically available, an unlimited number evaluations can be performed.

Table 4.2 shows the differences of estimation quality when training the original repository model (*Provided*) with 50K and 100K training iterations. Above these results, the first two rows contain the results published for RAFT and Separable Flow (*SepFlow*). In the third column, the results of evaluating the trained model provided by the Separable Flow authors (*Downloaded*) are shown. For this model, the number of training iterations is unknown. The *Downloaded* model has almost identical results to those reported in the paper. However, its parameter count is much higher with an increase of 2.346M trainable parameters from a reported 6M to 8.346M. Using 50K training iterations on the chairs dataset as specified by the implementation, the estimates of the *Provided* model are of much lower quality on both Sintel and KITTI than the *Downloaded* model. By moving from 50K to 100K chairs iterations, the *Provided* results are much closer to those of the *Downloaded* and *SepFlow* model. This shows that the change in training iterations is an important first step towards reproducing the results of the paper. However, especially on KITTI, the results of *Provided* are still worse. The KITTI end point error (EPE) differs by 0.47 from *SepFlow* which is a higher error than the *RAFT* model has.

## 4.2 Differences in the Model Structure

The largest differences between the paper and the implementation lie in the model structure. Three main differences of the model structure can be identified. These three changes will be described in the next three subsections. Furthermore, the approach to revert the changes will be detailed.

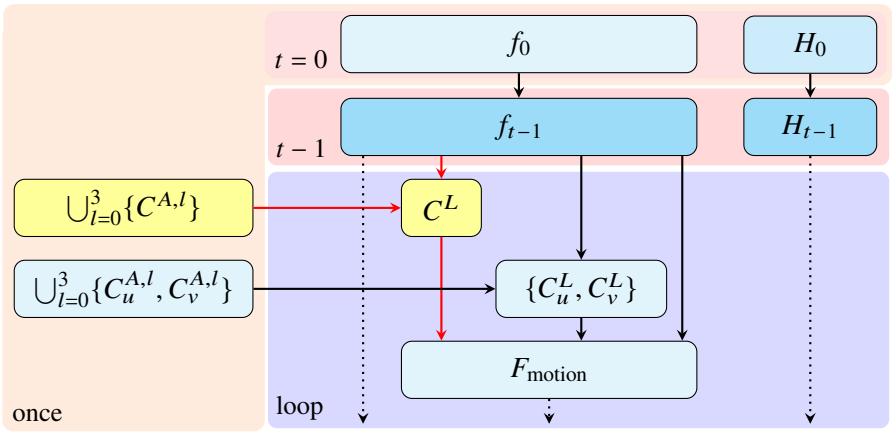


**Figure 4.1:** Data dependencies of tensors during the refinement phase of Separable Flow. Only the first part with the correlation features lookup is shown. Highlighted in yellow: 4D correlation volume pyramid and multi-scale correlation features $C^L$ from the 4D lookup operation.

### 4.2.1 Motion Features use 4D Correlation Volume

The first major structural change lies in the computation of the motion features. Figure 4.1 shows the first part of the Separable Flow refinement phase. All data dependencies related to the change are highlighted with yellow boxes for tensors and red arrows for dependencies.

To calculate the motion features, each refinement iteration receives the current flow estimate and the correlation volume as input. According to the paper, the 4D correlation volume $C(i, j, u, v)$ is replaced by the concatenated, aggregated 3D correlation volumes $[C_u^A(i, j, u), C_v^A(i, j, v)]$. However, in the implementation, the aggregated 4D correlation volume pyramid $\bigcup_{l=0}^{3}\{C^{A,l}\}$ is passed alongside the 3D correlation volume pyramid $\bigcup_{l=0}^{3}\{C_u^{A,l}C_v^{A,l}\}$ to the refinement iteration module. In each refinement step, the 2D lookup operation is performed on all levels $l$ of $C^{A,l}$. Similarly, for $C_u^{A,l}$ and $C_v^{A,l}$ the lookup is applied with a one dimensional local grid of the displacement value indices instead of a two dimensional one. The multi-scale correlation features $C^L$, $C_u^L$, $C_v^L$ are a concatenation of the results of the lookup on all levels along the channels dimension, as described in Section 3.7. Finally, motion features are computed using a convolutional neural network that receives the flow estimate and the correlation features of all three correlation volumes. To arrive at the motion features, separate convolutional layers process each of the four inputs. After that an additional convolutional layer is applied to the concatenated, preprocessed inputs, resulting in the final motion features.

To remove the 4D correlation volume $C^A$ from the motion features and revert the change to the model, the following steps are taken. First the lookup of the 4D correlation volume during the refinement iterations is disabled because the resulting lookup features $C^L$ are no longer required. Furthermore, the motion feature encoder is modified to account for the removal of $C^L$. This modification entails taking away the convolution layers responsible for processing $C^L$. In addition, the final channels dimension of the convolutional layer of the motion encoder can be decreased. This decrease originates from the fact that the features of $C^L$ no longer contribute to the concatenation along the channels dimension. In consequence, reverting the change leads to a reduction in the

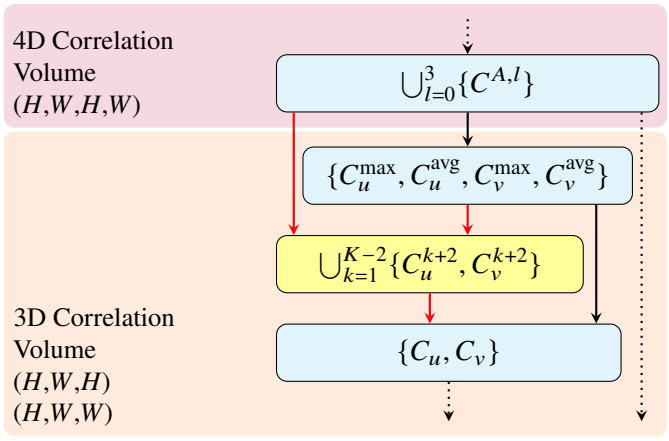


**Figure 4.2:** Excerpt of the data dependencies of tensors during the initialization phase of Separable Flow. Only the cost volume separation part is of the initialization phase is shown. Highlighted in yellow: Tensor containing the attention-based channels of the 3D correlation volume.

parameter count of the model since the size of the multi-channel learnable convolution kernels decreases. The exact impact of removing $C^A$ from the motion features with regard to the parameter count and estimation quality will be discussed in Section 4.2.5.

### 4.2.2 Attention-Based Channels for the 3D Correlation Volume

The next change is located in the channels of the 3D correlation volumes. As described in Chapter 3, those volumes have $K$ channels. For the initial two channels, the maximum and average statistics of the 4D correlation volume are computed. Statistical channels are mentioned in the paper and are also present in the provided implementation. However, the paper also describes the 4D correlation volume to have $K-2$ further channels. They are computed in an attention-weighted sum over one of the displacement dimensions of the 4D correlation volume. The learned attention and channels are missing from the implementation.

Figure 4.2 illustrates this change in the context of an excerpt from the initialization phase of Separable Flow. The yellow box shows the missing attention-based channels $\bigcup_{k=1}^{K-2}\{C_u^{k+2}, C_v^{k+2}\}$. These channels depend on the 4D correlation volume pyramid $\bigcup_{l=0}^{3}\{C^{A,l}\}$ that is compressed over all levels $l$. Besides the 4D correlation volume pyramid, the attention-based channels also depend on the maximum and average statistics $\{C_u^{\max}, C_u^{\text{avg}}, C_v^{\max}, C_v^{\text{avg}}\}$ of the 4D correlation volume for computing the attention weights. The weights' computation is performed by a convolutional layer, which is the only learnable component of the cost volume separation.

Considering the fact that the learnable part of cost volume separation is missing and the title "Separable Flow: Learning Motion Cost Volumes for Optical Flow Estimation", it is especially striking that the implementation skips and important part of learning the 3D correlation volumes. Only the 3D correlation volume aggregation remains as the second learnable part of the 3D correlation volume discussed in the paper. Section 4.2.3 discusses another structural change related to learning the correlation volume which only exists in the implementation.

```python
def separate(corr_pyramid, attention_nets, num_levels=4):

        corr_u_lvls = []
        corr_v_lvls = []

        for i in range(num_levels):
            corr = corr_pyramid[i]

            # ... compute C^max_u, C^avg_u, C^max_v, C^avg_v
            # corr_u = concatenate(C^max_u, C^avg_u)
            # corr_v = concatenate(C^max_v, C^avg_v)

            attention1, attention2 = attention_nets

            a_u = attention1(corr_v)
            a_u = a_u.unsqueeze(dim=2)
            a_u = a_u.softmax(dim=3)

            a_v = attention2(corr_u)
            a_v = a_v.unsqueeze(dim=3)
            a_v = a_v.softmax(dim=2)

            adaptive_corr_u = torch.einsum(
                    'bcuvij,bcuvij->bcuij',a_u, shaped_corr)
            adaptive_corr_v = torch.einsum(
                    'bcuvij,bcuvij->bcvij',a_v, shaped_corr)

            corr_u = torch.cat((corr_u, adaptive_corr_u), dim=1)
            corr_v = torch.cat((corr_v, adaptive_corr_v), dim=1)

            corr_u = F.interpolate(corr_u, [h2, h1, w1],
                    mode='trilinear', align_corners=True)
            corr_v = F.interpolate(corr_v, [w2, h1, w1],
                    mode='trilinear', align_corners=True)

            corr_u_lvls.append(corr_u)
            corr_v_lvls.append(corr_v)

        corr_u_final = torch.cat(corr_u_lvls, dim=1)
        corr_v_final = torch.cat(corr_v_lvls, dim=1)

        return corr_u_final, corr_v_final
```

**Listing 4.1:** Implementation of Attention-Based Channels

Reverting the change requires implementing the missing attention-based channels. A simplified version of the 3D correlation volume computation is shown in listing 4.1. Some of the variables have been renamed and lines are moved or summarized as compared to the implementation. This is done to highlight the semantics of the code and make it more concise. To separate the displacement dimensions of the 4D correlation volume pyramid, the channels of the 3D correlation volume are computed on each level. Two lists `corr_u_lvls` and `corr_v_lvls` are created in lines 3 and 4 to store the up-sampled 3D correlation volume of every pyramid level. A loop iterates over all pyramid levels `i` (line 6). On each level, the current level of the 4D correlation volume is loaded (line 7). Lines 9 to 11 represent the computation of the maximum and average channels, which are concatenated and stored in variables `corr_u` and `corr_v`. This part was summarized because it was already present in the provided implementation. Separate attention networks `attention1` and `attention2` are passed to the function `separate` from the main Separable Flow module, since they need to persist as long as the main module exists. They consist of a single 3D convolution layer with a kernel size of `(3,3,3)` and appropriate padding `(1,1,1)` to preserve the spatial dimension size during the operation. These networks are used to compute the 3D attention weights `a_u` and `a_v` of shapes `(batch, K-2, V, H, W)` and `(batch, K-2, U, H, W)` from the maximum and average channels. Because each of the $K-2$ attention-based channel has its own attention weights channel, the convolution operation receives 2 maximum and average channels and returns the attention weights tensor with $K-2$ channels. To create a uniform shape for the attention weights tensors, the missing $U$ or $V$ displacement dimension is replaced by a dimension of size one. The resulting shapes are `(batch, K-2, 1, V, H, W)` for `a_u` and `(batch, K-2, U, 1, H, W)` for `a_v`. Finally, the **`softmax`** operation is applied to the remaining displacement dimension that has a size large than one, such that the values of all indices along this dimension sum to one. The attention weighted sum over the batch and spatial dimension of the 4D correlation volume is implemented by the PyTorch operation **`einsum`**. It enables the use of the Einstein summation convention, that makes it possible to specify tensor operations in a clear and concise fashion. On the left side of the equation, the dimensions of both inputs are named with one letter each and separated by a comma. After the arrow, the desired dimensions of the output are specified in terms of the dimensions of the input. The v dimension is not present on the output side in line 24 thus for each index of the output $(b, c, u, i, j)$ the sum over the v dimension of the inputs product at the current $(b, c, u, v, i, j)$ index is taken.

$$\underline{\text{output}}(b, c, u, i, j) = \sum_{v=0}^{V-1} \underline{\text{input1}}(b, c, u, v, i, j) \cdot \underline{\text{input2}}(b, c, u, v, i, j) \tag{4.7}$$

Equation (4.7) describes the operation mathematically.

Lines 31-34 show how the spatial dimension on each pyramid level is up-sampled, such that the spatial dimensions of each level are identical to the highest resolution level $(\mathbf{H}, H, W)$ and $(\mathbf{W}, H, W)$. To accomplish this, trilinear interpolation is performed between values of each tensor with lower spatial dimension size in the displacement dimensions. By up-sampling all levels to the same resolution, they can be concatenated along the channels dimension, resulting in the final 3D correlation volumes `corr_u_final` and `corr_v_final` (lines 39-40).

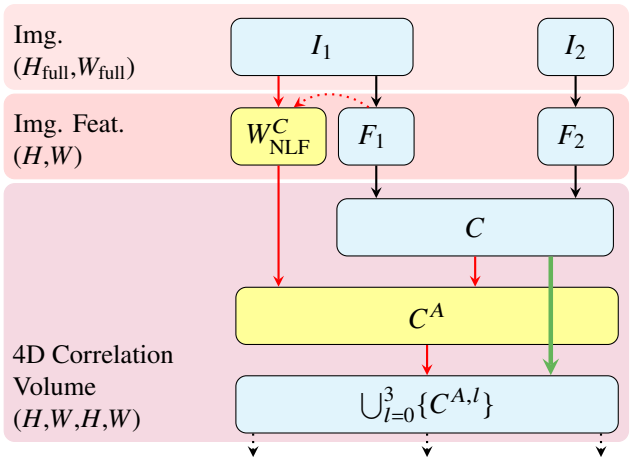


**Figure 4.3:** Excerpt of the data dependencies of tensors during the initialization phase of Separable Flow. Only the 4D correlation volume aggregation part of the initialization phase is shown. Highlighted in yellow: Filter weights tensor $W^C_{\text{NLF}}$ and aggregated 4D correlation volume $C^A$.

### 4.2.3 Aggregation of the 4D Correlation Volume

Before the 3D correlation volumes can be computed by compressing the information of the 4D correlation volume, first the 4D correlation volume pyramid has to be calculated. The final major structural change can be found in the construction of the 4D correlation volume. In the paper, Section 3.2.2 is dedicated to learning correlation volume aggregation. This chapter only mentions aggregation of the 3D correlation volumes but not the 4D volume. In contrast, the provided model uses non-local filters (NLF) to aggregate the 4D correlation volume before down-sampling to create the pyramid.

Figure 4.3 shows the 4D cost volume aggregation in the context of the 4D correlation volume construction during the Separable Flow initialization phase. The NLF weights $W^C_{\text{NLF}}$ and the aggregated cost volume $C^A$ are highlighted in yellow. To compute the NLF weights, the same guidance subnetwork is used as for the 3D correlation volume aggregation weights, receiving the image features and the image itself. Then the filter operation is applied to the 4D correlation volume $C$ with the filter weights

The use of non-local filters for 3D stereo correlation volume aggregation has been investigated and therein the correlation volume for stereo matching has height, width, displacement and feature dimensions [ZPYT19]. Therefore, they can be utilized to aggregate the 3D correlation volumes of Separable Flow. However, the optical flow 4D correlation volume has height, width as well as horizontal and vertical displacement dimensions. It was not possible to retrieve the source paper or any kind of documentation of the specific NLF for 4D correlation volumes that was implemented in the provided Python and CUDA C++ code. From the code, it is apparent that the filter is applied in a sequential manner in four directions (left, right, up, down). Each directional filter application updates the 4D correlation volume and the updated volume is then passed to the next directional filter application.

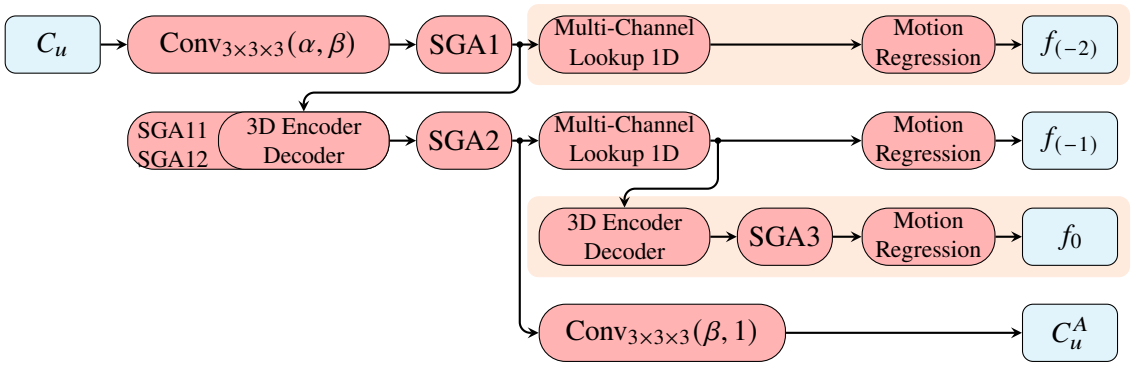


**Figure 4.4:** Illustration of the provided motion aggregation and regression network. Orange Background: Suggested operations to be removed.

Since the NLF does not change the shape of the 4D correlation volume, all following operations are not affected by omitting the NLF operation. Consequently, the change can be reverted by skipping the NLF operation between the 4D correlation volume creation and average pooling for the correlation volume pyramid. After removing the NLF operation, the 4D correlation volume creation from the image features is then directly followed by building the 4D correlation volume pyramid. In addition, the guidance subnetwork is modified to only compute the 3D correlation volume aggregation weights $\{W_{\text{SGA}}^{C_u}, W_{\text{SGA}}^{C_v}\}$. Layers of the subnetwork that are only relevant for 4D correlation volume aggregation $W_{\text{NLF}}^{C}$ are removed.

### 4.2.4 3D Motion Aggregation and Motion Regression Network

This section describes the 3D motion aggregation network as it is present in the provided implementation. The publication of Separable Flow contains only little information besides the number of eight 3D convolutional and four Semi-Global Guided Aggregation (SGA) layers. It is not possible to deduce the intended published structure from the publication. In the implementation more convolutional layers and one additional SGA layer are used. However, too little information is available to find concrete changes of the implementation. For example, it is unclear which convolutional layers should be removed to arrive at the published layer count.

Figure 4.4 shows the simplified structure of the provided motion aggregation and regression network. The 3D correlation volumes $C_u$ or $C_v$ resulting from the channels separation are passed to separate networks. They have $\alpha$ channels, which are in the number of pyramid levels $l_{\text{max}}$ times the number of separation channels $K$. First, a 3D convolution produces a tensor with $\beta$ channels from $C_u$ which is then processed by the first SGA layer (SGA1). After that, the first auxiliary flow estimate $f_{(-2)}$ is produced by applying a zero-centered lookup operation over all channels with a fixed displacement range to result in a tensor with displacements $\{-d_{\text{max}}, \ldots, -1, 0, 1, \ldots, d_{\text{max}}\}$ for each channel and first frame feature index. The motion regression module then reduces the channels dimension to one by using a single convolutional layer and applies motion regression as described in Section 3.7. A 3D convolutional encoder decoder network with additional SGA11 and SGA12 layers is used to process the result from SGA1. This layer is responsible for the divisibility constraint mentioned in Chapter 4.1. Next, another SGA layer (SGA2) is applied to the previous result. Again, multi channel lookup and motion regression allow for another auxiliary flow estimate $f_{(-1)}$. The result of

| Model | | | | Sintel | | KITTI | |
|---|---|---|---|---|---|---|---|
| **Method** | **#Par** | **Chairs It.** | **Data** | **clean** | **final** | **epe** | **F1** |
| RAFT | 5.257M | 100K | C+T | 1.43 | 2.71 | 5.04 | 17.4 |
| SepFlow | 6.0M | 100K | C+T | **1.30** | **2.59** | **4.60** | **15.9** |
| Downloaded | 8.346M | n/a | C+T | 1.30 | 2.60 | 4.61 | 15.9 |
| Provided | 8.346M | 50K | C+T | 1.55 | 2.89 | 5.69 | 19.6 |
| Provided | 8.346M | 100K | C+T | 1.31 | 2.68 | 5.08 | 17.0 |
| NoCorr | 7.602M | 50K | C+T | 1.93 | 3.56 | 11.31 | 34.7 |
| NoCorr | 7.602M | 100K | C+T | 1.93 | 3.51 | 11.12 | 35.4 |
| ↑ & K=4 | 7.606M | 50K | C+T | 2.00 | 3.53 | 11.84 | 38.3 |
| ↑ & K=4 | 7.606M | 100K | C+T | 1.78 | 3.25 | 10.29 | 31.0 |
| ↑ & ↑ & NoAgg | 7.557M | 50K | C+T | 1.85 | 3.34 | 10.42 | 32.3 |
| ↑ & ↑ & NoAgg | 7.557M | 100K | C+T | 1.81 | 3.33 | 10.50 | 32.0 |

**Table 4.3:** Results for model structure ablations. Bold: Best overall results, Underlined: Best results of models trained in this thesis.

the lookup is processed further by an additional 3D convolutional encoder decoder network as well as the final SGA3 layer. To produce the final initial flow estimate $f_0$, motion regression is applied to the output of SGA3. Finally, the aggregated 3D correlation volume $C_u^A$ is computed by applying a convolution that reduces the channels dimension of the output of SGA2 from $\beta$ to one.

In conclusion, the seemingly trivial 3D cost aggregation network mentioned in the publication is a complex motion aggregation and regression network. Due to the sparse description, it is not possible to find concrete changes of the network structure. However, as previously discussed for the loss function, the published loss does not utilize auxiliary flow estimates. Therefore, a viable modification strategy may be to use $f_{(-1)}$ as initial flow estimate and remove all operations corresponding to only $f_{(-2)}$ and $f_0$. This would reduce the parameter count, especially by removing the second encoder decoder network. Furthermore, by removing SGA3, the published count of four SGA layers would be reached. However, because of the uncertainty of the modification strategy and limited training time for ablations, such modifications were not explored with experiments.

### 4.2.5 Evaluation of Structual Changes

Now that all major structural differences have been discussed and reverted, this section discusses the results with respect to the estimation quality and parameter count. Table 4.3 shows all the results. Once more, the published results for RAFT, Separable Flow and trained model provided by the Separable Flow authors *Downloaded* are shown in the topmost three rows. The pre-training schedule with the first two training stages of Flying Chairs and Flying Things was used for the ablations. For these methods, the best overall results are reported for Separable Flow on the Sintel and KITTI training datasets.

| Model | | | Sintel(test) | |
|---|---|---|---|---|
| **Method** | **Chairs It.** | **Data** | **clean** | **final** |
| RAFT (2-view) | 100K | C+T+S+K+H | 1.94 | 3.18 |
| SepFlow (2-view) | 100K | C+T+S+K+H | 1.50 | 2.67 |
| NoAgg (2-view) | 100K | C+T+S+K+H | 2.31 | 3.87 |

**Table 4.4:** Results for finetuning the final ablation step model on Sintel.

Below, the fourth and fifth row *Provided* show the results of the provided model for both 50K and 100K chairs iterations, as discussed in Section 4.1. These results are already close to those of Separable Flow and *Downloaded*.

Results for structural ablation steps described in this section can be seen in the last six columns. They are referred to as *NoCorr*, *K=4* and *NoAgg* and will be discussed in the next paragraphs.

The first ablation step called *NoCorr* is a model that omits use of the 4D correlation volume in the motion features. It corresponds to the change discussed in Section 4.2.1. As discussed in this section, the *Provided* is modified such that the change is reverted. A large drop off in estimation quality, for both 50K and 100K chairs iterations compared to the *Provided* model can be observed on both Sintel and KITTI. Consequently, the inclusion of 4D correlation volume motion features seems to have a large positive impact. In addition to the loss in quality, the model has 0.744M fewer parameters than *Provided*. The cause of this is the removal of convolutions in the motion features network, which are responsible for processing the 4D correlation features.

In the next structural ablation step *K=4*, two attention-based channels are added to the *NoCorr* model from the previous ablation step. These channels are implemented as described in Section 4.2.2. By adding two attention-based channels to the 3D correlation volume, some of the lost quality of the previous ablation step can recuperated. This is evident from the improved results for 100K training iterations compared to the previous ablation step. Especially in relation to the small increase of merely 0.004M parameters, adding these extra channels is effective.

In the final ablation step *NoAgg*, the aggregation of the 4D correlation volume is omitted. Once more, the model from the previous ablation step *K=4* is modified according to Section 4.2.3 to revert the change. The aggregation seems to have a slight positive impact on the estimation quality, as removing them leads to a small drop in performance. Surprisingly, the performance for 50K training iterations is improved compared to the previous step. Furthermore, in this final ablation step, the estimation quality gap between the model with 50K Flying Chairs iterations compared to 100K iterations is quite small, similarly as in the first structural ablation step *NoCorr*. The larger gap in estimation quality between the model with 50K and 100K chairs iterations for the *Provided* model may be caused by its larger parameter count compared to *NoCorr* and *NoAgg*. More training iterations may be required to make full use of the 4D correlation motion features and additional parameters. Overall, the change saves around 0.05M parameters, owing to the exclusion of the filter weights network from the model.

Table 4.4 shows the results for models finetuned on the third training stage, which is the Sintel stage and evaluated on the Sintel test dataset. In the first two columns, the published results for RAFT and Separable Flow are listed. Below, the results of the refined final ablation step *NoAgg*

are displayed. To evaluate the model, the results were submitted to the Sintel leaderboard website. It can be observed that the estimation quality differences between the listed models are similar to those of the models on the ablation training schedule. Therefore, the model from the final ablation step *NoAgg* is still unable to reproduce the reported results. The estimation quality of all models is lower when finetuning for Sintel and evaluating on Sintel test compared to the ablation training and evaluation schedule.

## 4.3 Discussion

By finding and reverting the major structural changes of the model, it was possible to fit the model that is described in the paper more closely. However, the results show that the model with the structure specified in the original implementation performs best and is closest to the reported estimation quality. Especially removing the 4D correlation volume as input to the motion features decreased the performance significantly. Only adding the attention-based channels improved the results. Intuitively, this seems reasonable because it is the only change that added parameters and operations to the network, where all other changes do the opposite.

Considering the results of the structural ablations, it seems like achieving the results reported in the paper is not possible with the structure from the paper, but only the one provided by the implementation. This is backed by the provided model trained by the authors (*Downloaded*), which has the same structure and parameter count of the *Provided* model. The results of the *Downloaded* model were evaluated as part of this thesis and are very close to the results reported in the paper. Hence, the key to the achieving the same estimation quality as the *Downloaded* model with the provided model implementation has to be located in changes of operations without trainable parameters or in the training procedure of the model.

While the change from 50K to 100K training iterations on the Flying Chairs dataset did improve the performance on the ablation schedule, the results were still slightly worse than the ones of the *Downloaded* model. Therefore, the authors of Separable Flow may have still used a different set of training parameters than the implementation. However, they are not recoverable from the published descriptions or the implementation.

Although the claims of the authors with respect to their combination of model structure, number of trainable parameters and estimation quality could not be reproduced, their conclusion still holds for the ablation results of this thesis: “The main benefit of our method is [...] its improved accuracy (compared to parameter count, speed and accuracy of RAFT)” [ZWPT21, p. 10814]. The *Provided* implementation with 100K training iterations achieves results that are superior to RAFT, although the model structure is different and parameter count 1.4 times higher than stated in the Separable Flow publication.

By changing the structure of the 3D motion aggregation and regression network as suggested in Section 4.2.4, it might be possible to fit the description more closely. However, since the suggested change is only based on evidence from the sparse published description, it may yet introduce new differences to the model evaluated in the Separable Flow publication. For this reason as well as the limited number of ablation steps caused by the long and computationally expensive training, no experiments were conducted on this change.

# 5 Memory Saving Strategy for Separable Flow

One interesting claim of the paper is that the 4D correlation volume $C$ does not need to be stored as an intermediary result for the computation of the 3D correlation volumes $C_u$ and $C_v$ [ZWPT21]. If $C$ is only used for $C_u$ and $C_v$, this implies that its storage could be omitted completely. Hence, it would be possible to save storage space on the GPU during both training and inference time. This section investigates whether it is possible to to perform the cost volume separation in this memory saving fashion. An implementation is described and its memory and timing results are evaluated.

## 5.1 Theoretical Savings

This section investigates how much memory can theoretically be saved by omitting storage of the 4D correlation volume. The first important aspect of this is how much memory is consumed by $C$ and also by the 4D correlation volume pyramid.

$$\begin{aligned} \text{storage}(C^l) &= H \cdot W \cdot (H/2^l) \cdot (W/2^l) \cdot 4 \text{ Bytes} \\ &= (HW)^2 \cdot (1/4^l) \cdot 4 \text{ Bytes} \end{aligned} \tag{5.1}$$

Equation (5.1) shows the memory consumption for storing the 4D correlation volume $C^l$ at the pyramid level $l$ for an image with dimensions $(H_{\text{full}}, W_{\text{full}}) = (8H, 8W)$. Both 2D locations in the image features of frame one and frame two that correspond to one value in the 4D correlation volume contribute their dimension sizes. Each value in $C^l$ of shape $(H, W, H/2^l, W/2^l)$ is a floating point number that requires 4 Bytes of storage.

$$\begin{aligned} \text{storage}(\bigcup_{l=0}^{N-1} \{C^l\}) &= \sum_{l=0}^{N-1} \text{storage}(C^l) \\ &= (HW)^2 \cdot 4 \text{ Bytes} \cdot \sum_{l=0}^{N-1} \left(\frac{1}{4}\right)^l \end{aligned} \tag{5.2}$$

$$\lim_{N \to \infty} \text{storage}(\bigcup_{l=0}^{N-1} \{C^l\}) = \frac{4}{3} \cdot (HW)^2 \cdot 4 \text{ Bytes} \tag{5.3}$$

To compute the storage of the pyramid, the storage consumption of all levels is accumulated. For $N = 4$ levels, the sum amounts to around 1.328, which is close to the value of $1.\overline{3}$ that the sum converges to for $N \to \infty$.

When considering an image of size $(320, 448)$, then $(H, W)$ would be $(40, 56)$ and the 4D correlation volume pyramid would consume at least 26.8MiB. Similarly, for a larger image size of $(512, 1024) = 8(64, 128)$ the memory requirement would be 357.9MiB.

At inference time, the storage savings should almost reach the amount of the size of the 4D correlation volume pyramid. Only the additional storage requirement of the image feature pyramid $F_2^l$ for levels 1-3 has to be deducted, since it is replaced in the original implementation by pooling $C$ directly.

In contrast to the inference time, during training time intermediary results of the forward pass from the PyTorch functions have to be stored for use in the backward pass. During the backward pass, additionally the gradient of the 4D correlation volume with respect to the loss $\frac{\partial L}{\partial C^l}$ has to be stored. This leads to additional storage requirements of the same size as the pyramid, doubling the overall amount.

## 5.2 Feasibility Considerations for Varying Model Structures

To investigate whether it is feasible to omit storing $C$ as an intermediary result, its data dependencies are examined. Furthermore, operations required to perform the calculation are identified and described.

As mentioned in Section 3.7 and Section 4.2, the 4D correlation volume depends on the features of the first and second image as well as guidance for cost volume aggregation. The maximum, average and attention-based channels of $C_u$ and $C_v$ as well as the motion features depend on the 4D correlation volume pyramid $\bigcup_{l=0}^{3}\{C^{A,l}\}$.

| Implementation | Paper | |
|---|---|---|
| $F_1, F_2 \rightarrow C$ | $F_1, F_2 \rightarrow C$ | (5.4) |
| $C, W_{\text{NLF}}^C \rightarrow C^A$ | **no aggregation** | (5.5) |
| $C^A \rightarrow \bigcup_{l=0}^{3}\{C^{A,l}\}$ | $C \rightarrow \bigcup_{l=0}^{3}\{C^l\}$ | (5.6) |
| $\bigcup_{l=0}^{3}\{C^{A,l}\} \rightarrow C_u, C_v$ | $\bigcup_{l=0}^{3}\{C^l\} \rightarrow C_u, C_v$ | (5.7) |
| $\ldots, \bigcup_{l=0}^{3}\{C^{A,l}\} \rightarrow F_{\text{motion}}$ | $\ldots \rightarrow F_{\text{motion}}$ | (5.8) |

To omit storage of the correlation volume, its values need to be computed on-demand. In this context, on-demand computation means that one index of $C$ is computed only at the point in time when it is needed.

After computing $C$ (Equation (5.4)), it is aggregated in the implementation (Equation (5.5)). The aggregation is performed using a non-local filter. It may introduce dependencies from each index of the aggregated correlation volume $C^A$ to many indices of $C$. This can result in the need for cascading on-demand computations of the correlation volumes. First, one index of the aggregated 4D correlation volume would be requested, leading to on-demand computations of the indices of $C$ that are dependencies of this index. Due to that, the computation of $C_u$ and $C_v$ in Equation (5.7) becomes very costly, since all indices of $C^A$ are requested at least once. The second problem of the implementation lies in the fact that the aggregated 4D correlation volume pyramid is used in

the computation of the motion features (Equation (5.8)). As described in Section 3.6, the paper that introduced RAFT provides an implementation for the 4D correlation volume lookup with on-demand computation of the motion features. However, despite the availability of the on-demand lookup implementation, the same problem of cascading on-demand computation would still persist in case $C^A$ is used. Therefore, omitting $C$ as an intermediary result is likely not feasible for the provided implementation.

Following the model structure from the paper, both the aggregation of $C$ and its usage in the motion features are not present. The dependencies for this structure are shown in the right column of Equations (5.4)–(5.8). Because of the reduced structure which results in fewer on-demand computations of $C$, the model is far more suitable for omitting the storage of the 4D correlation volume. Methods for the model described in the paper to compute the maximum, average and attention-based channels of the 3D correlation volume without storing the 4D correlation volume will be investigated in the next section.

## 5.3 Alternative 3D Correlation Volume Computation

This section describes how the 3D correlation volumes are computed in the paper and how this computation was implemented. The goal is to compute the 3D correlation volumes $C_u$ and $C_v$ from the image features $F_1$ and $F_2$ without storing the 4D correlation volume $C$ in a time-efficient manner.

$$C^l(i,j,u,v) = F_1^0(i,j) \cdot F_2^l(u,v)$$

| | | | |
|---|---|---|---|
| $C_u^{\text{avg},l}(i,j,u)$ | $= \frac{1}{\lvert V\rvert} \sum_{v \in V} C^l(i,j,u,v)$ | $C_v^{\text{avg},l}(i,j,v)$ | $= \frac{1}{\lvert U\rvert} \sum_{u \in U} C^l(i,j,u,v)$ |
| $C_u^{\text{max},l}(i,j,u)$ | $= \max_{v \in V} C^l(i,j,u,v)$ | $C_v^{\text{max},l}(i,j,v)$ | $= \max_{u \in U} C^l(i,j,u,v)$ |
| $\hat{A}_u$ | $= \sigma(\phi_u(C_v^{\text{avg},l}, C_v^{\text{max},l}))$ | $\hat{A}_v$ | $= \sigma(\phi_v(C_u^{\text{avg},l}, C_u^{\text{max},l}))$ |
| $C_u^{k+2,l}(i,j,u)$ | $= \hat{A}_u^k(i,j) \cdot C^l(i,j,u,:)$ | $C_v^{k+2,l}(i,j,v)$ | $= \hat{A}_v^k(i,j) \cdot C^l(i,j,:,v)$ |
| $C_u^l$ | $= \text{concat}(C_u^{\text{max},l}, C_u^{\text{avg},l}, C_u^{3:K,l})$ | $C_v^l$ | $= \text{concat}(C_v^{\text{max},l}, C_v^{\text{avg},l}, C_v^{3:K,l})$ |

The equations above show how each channel of the 3D correlation volumes $C_u$ and $C_v$ is calculated. Each channel depends directly on all indices of $C$. By replacing each occurrence of $C(i,j,u,v)$ by its equation $F_1(i,j) \cdot F_2(u,v)$, the usage of $C$ as an intermediary result is eliminated. This means that all usages of intermediary results are replaced by their corresponding on-demand computation. At each level $l$ of the correlation volume pyramid, on-demand computation of indices is performed as described in RAFT [TD20], see Section 3.6.2.

There is no need to modify the equation for the attention $\hat{A}_u$ and $\hat{A}_v$ because it receives and produces data with 3D storage requirement with respect to image height and width. The input consists of the maximum and average channels of the 3D correlation volume and the output is the attention of shape $(H,W,H,K)$ or $(H,W,W,K)$. Hence this operation can be implemented using the PyTorch operations `softmax` and `Conv3d`, which do not allow for direct control over the storage allocated for intermediary results.

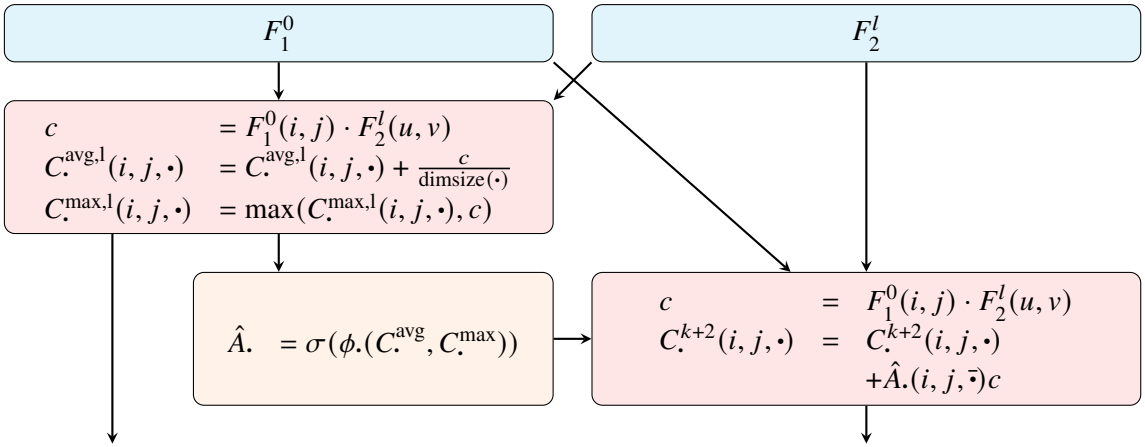


**Figure 5.1:** Approach for minimizing the number of on-demand computations of $C$. Blue boxes: image features input, red boxes: CUDA operation, each index of $C$ is computed only once, orange box: attention computed using PyTorch convolutions and softmax.

By contrast, the computation of the maximum, average and attention-based channels is implemented using custom CUDA functions. This is necessary, since the computation of each correlation volume index has to be performed on-demand and full control over storage of intermediary results is required. Another important aspect is the computation time. For each of the $HWHK$ indices of $C_u$, $W$ on-demand computations of $C$ are required. Every on-demand index computation takes the scalar product over two vectors of size $F$. In total, this leads to a time complexity in $O((HW)^2KF)$.

### 5.3.1 Speedup Strategies

Because the time complexity is quadratic in the number of pixels and multiplied by a large factor of 256 for the image feature size $F$, it is important to speed this computation up as much as possible. To accelerate the custom operations, two different approaches were combined. First the number of computations of each index of $C$ is minimized. Second indices of $C_u$ and $C_v$ are computed in parallel on the GPU using CUDA.

Figure 5.1 shows how the number of on-demand computations of indices of $C$ is reduced. The image features $F_1^0$ and $F_2^l$ in the blue boxes serve as input to the channels computation. In the upper left red box, each index of $C$ is computed once. Each index contributes to the maximum and average channels $C_u^{\text{max},l}$, $C_u^{\text{avg},l}$, $C_v^{\text{max},l}$ and $C_v^{\text{avg},l}$ by updating them directly. For the maximum channel, the current maximum is compared to the on-demand correlation value. If the new value is larger, the channel value is overwritten. In case of the average channel, each correlation value is divided by the corresponding displacement dimension size and added to the current intermediate average result. After the results of the maximum and average channels are finalized, the attention for $K-2$ attention-based channels is computed using a convolutional layer and a softmax operation. Finally, the attention-based channels are computed, receiving the results of all previous operations as input. All indices of $C$ are computed once more. Each value is reused for each of the $K-2$ additional channels $C_u^{k+2}$ and $C_v^{k+2}$. Updates are performed by adding the attention-weighted correlation value to the intermediary channel results.

Further speedup is achieved by parallelizing the computation and running it on the GPU. Executing the custom function on the GPU has the additional benefit that all PyTorch tensors of the model and intermediary results are already stored on the GPU. Therefore, input and output Tensors of the function do not need to be moved to and from the GPU Custom PyTorch functions can be implemented in CUDA. Each custom function can consist of multiple executions of CUDA kernels. A CUDA kernel is a function that is executed in parallel, where the same code runs with different parameters in each thread, as described in Section 3.9. Two custom functions were implemented, each with with only one kernel execution. The calculation steps are distributed as shown in Figure 5.1. One function computes the maximum and average channels, while the other computes the attention-based channels.

### 5.3.2 Implementation of the Forward Pass

This section describes how the forward pass of the custom functions is implemented. To understand the custom functions better, Algorithm 5.1 shows the context they appear in. The algorithm describes the calculation of the 3D correlation volumes $C_u$ and $C_v$, which is similar to listing 4.1 in the previous Chapter 4. However, implementations of the maximum and average channels, as well as the attention-based channels have been contracted into functions. Each of these two functions is a custom PyTorch function that runs a CUDA kernel on the GPU. For the first function, the frame features of frame one at level zero and frame two at level $l$ are required as inputs to compute the maximum and average channels. In case of the function for the attention-based channels, attention weights are passed additionally. These two custom function calls are repeated for every level in the 4D correlation volume pyramid. At each level, both displacement dimensions are halved, so the computation steps are quartered with every level increase.

**Algorithm 5.1** Computation of the 3D Correlation Volume: Overview

**Require:** maxPyramidLevel $\geq 0$
**Require:** $K \geq 2$
$F_1^0 \leftarrow \text{FrameFeatureNet}_1(I_1)$
$F_2^0 \leftarrow \text{FrameFeatureNet}_2(I_2)$
**for** $l \leftarrow 1, \ldots,$ maxPyramidLevel **do**
    $F_2^l \leftarrow \text{Pool2d}(F_2^{l-1}, \text{kernel=2, stride=2})$
**end for**
**for** $l \leftarrow 0, \ldots,$ maxPyramidLevel **do**
    $(C_u^l)^{1:2}, (C_v^l)^{1:2} \leftarrow \text{MaxAvg}(F_1^0, F_2^l)$ // Call function for max and avg channels
    $A_u \leftarrow \phi_u((C_v^l)^{1:2})$
    $A_v \leftarrow \phi_v((C_u^l)^{1:2})$
    $(C_u^l)^{3:K}, (C_v^l)^{3:K} \leftarrow \text{Compress}(F_1^0, F_2^l, A_u, A_v)$ // Call function for attention-based channels
    $(C_u^l)_{\text{up}}^{1:K} \leftarrow \text{Upsample}((C_u^l)^{1:K})$
    $(C_v^l)_{\text{up}}^{1:K} \leftarrow \text{Upsample}((C_v^l)^{1:K})$
**end for**
$C_u \leftarrow \text{Concatenate}((C_u^{0:\text{maxlevel}})_{\text{up}}^{1:K})$
$C_v \leftarrow \text{Concatenate}((C_v^{0:\text{maxlevel}})_{\text{up}}^{1:K})$

After showing the context in which the custom functions appear, the implementation of the first two functions will be described in this block. Algorithm 5.2 provides a sketch of the GPU kernel of the custom function for the maximum and average channels. The provided pseudocode is optimized to have good cache coherency of loading data from global GPU memory under the constraints of sharing each index computation of $C$ to update the channels of $C_u$ and $C_v$.

As already mentioned, the kernel is invoked by a custom PyTorch function. This function allocates memory for the output $C_u^{\max}$, $C_u^{\text{avg}}$, $C_v^{\max}$ and $C_v^{\text{avg}}$ of the kernel. Alongside the previously described inputs to the custom function, the output arrays are passed to the kernel. All threads have access to the kernel arguments which are located in the global memory of the GPU. The custom function also specifies special template parameters that indicate the number of blocks and the number of threads in each dimension.

$$
\begin{aligned}
&\underline{\text{blockDim} \in \mathbb{N}^3} && \underline{\text{threadDim} \in \mathbb{N}^2} \\
&\text{blockDim}.x = \text{batchSize} \\
&\text{blockDim}.y = \left\lceil \frac{H}{H_{\text{block}}} \right\rceil && \text{threadDim}.x = H_{\text{block}} = u_{\text{blockMax}} = 4 \\
&\text{blockDim}.z = \left\lceil \frac{W}{W_{\text{block}}} \right\rceil && \text{threadDim}.y = W_{\text{block}} = v_{\text{blockMax}} = 8
\end{aligned}
$$

They were chosen as can be seen above, with three block dimensions and two thread dimensions. Variables $H_{\text{block}}$ and $W_{\text{block}}$ are compile time constants which determine the size of the patch of image $I_1$ that is processed by each block. Based on the thread dimensions, block dimensions are chosen such that the set of all blocks covers all pixels of all first frames in the batch with one thread each. If the spatial image dimension sizes height $H$ and width $W$ are not divisible by $H_{\text{block}}$ and $W_{\text{block}}$ respectively, which will lead to edge block threads that are outside the image. These threads are handled by performing out-of-bounds checks on data loading operations between global memory and shared memory. Each thread has to be able to identify in where it is located in each thread. For this reason, kernels have built-in variables *threadIdx* and *blockIdx* as well as *blockDim*. In the implementation the variables are assigned as follows.

$$
\begin{aligned}
&\underline{\text{Block-Local Thread Index } t \in \mathbb{N}^2} && \underline{\text{Global Thread Index / Pixel Index } p \in \mathbb{N}^2} \\
&t.x = \text{threadIdx}.x && p.x = \text{blockIdx}.y \cdot \text{threadDim}.x + \text{threadIdx}.x \\
&t.y = \text{threadIdx}.y && p.y = \text{blockIdx}.z \cdot \text{threadDim}.y + \text{threadIdx}.y
\end{aligned}
$$

The idea for parallelizing the computation is that each thread is responsible for computing some of the indices of the output $C_u(b, i, j, u, v)$ and $C_v(b, i, j, u, v)$. Each thread can be assigned a global index $(b, p.x, p.y)$. In this case, the thread is responsible for output indices $(i, j) = (p.x, p.y)$ of the output for the $b$-th image pair of the batch. By assigning index responsibility zones, data read and write conflicts between the threads can be avoided.

Because there is one thread for all indices $(b, i, j)$, each thread needs to iterate only over the displacement indices $(u, v)$ and the image feature channel indices $f$. These indices are partitioned into blocks as well. However, in contrast to $(i, j)$, they are all processed by the same thread. Iterative processing in blocks of a small constant size has the advantage of enabling the use of the highly limited and fast shared memory for storing intermediate results. Such shared storage is denoted by the subscript $\bullet_{\text{cache}}$.

**Algorithm 5.2** Custom GPU kernel for Maximum and Average Channels

1: **for** $v_{\text{block}} \leftarrow 0 \dots v_{\text{blockMax}}, u_{\text{block}} \leftarrow 0 \dots u_{\text{blockMax}}$ **do**
2: $v_{\text{offset}} \leftarrow v_{\text{block}} \cdot W_{\text{block}}$
3: $u_{\text{offset}} \leftarrow u_{\text{block}} \cdot H_{\text{block}}$
4: sync()
5: $C_{\text{cache}}(t.x, t.y, :, :) \leftarrow 0$
6: $C^{\max}_{v,\text{cache}}(t.x, t.y, :, :) \leftarrow -\infty, \quad C^{\text{avg}}_{v,\text{cache}}(t.x, t.y, :, :) \leftarrow 0$
7: $C^{\max}_{u,\text{cache}}(t.x, t.y, :, :) \leftarrow -\infty, \quad C^{\text{avg}}_{u,\text{cache}}(t.x, t.y, :, :) \leftarrow 0$
8: **for** $f_{\text{block}} \leftarrow 0 \dots f_{\text{blockMax}}$ **do**
9: $f_{\text{offset}} \leftarrow f_{\text{block}} \cdot f_{\text{blockSize}}$
10: $F_{1,\text{cache}}(t.x, t.y) \leftarrow F_1(b, p.x, p.y, f_{\text{offset}} : f_{\text{offset}} + f_{\text{blockSize}})$
11: $F_{2,\text{cache}}(t.x, t.y) \leftarrow F_2(b, u_{\text{offset}} + t.x, v_{\text{offset}} + t.y, f_{\text{offset}} : f_{\text{offset}} + f_{\text{blockSize}})$
12: sync()
13: **for** $u \leftarrow 0 \dots H_{\text{block}}, v \leftarrow 0 \dots W_{\text{block}}$ **do**
14: $C_{\text{cache}}(t.x, t.y, u, v) \leftarrow \sum_{f=0}^{f_{\text{blockSize}}} F_{1,\text{cache}}(b, t.x, t.y, f) \cdot F_{2,\text{cache}}(b, u, v, f)$
15: **end for**
16: sync()
17: **end for**
18: **for** $u \leftarrow 0 \dots H_{\text{block}}, v \leftarrow 0 \dots W_{\text{block}}$ **do**
19: $C^{\max}_{v,\text{cache}}(t.x, t.y, v) \leftarrow \max(\bullet, C_{\text{cache}}(t.x, t.y, u, v))$
20: $C^{\max}_{u,\text{cache}}(t.x, t.y, u) \leftarrow \max(\bullet, C_{\text{cache}}(t.x, t.y, u, v))$
21: $C^{\text{avg}}_{v,\text{cache}}(t.x, t.y, v) \leftarrow \bullet + C_{\text{cache}}(t.x, t.y, u, v)/H$
22: $C^{\text{avg}}_{u,\text{cache}}(t.x, t.y, u) \leftarrow \bullet + C_{\text{cache}}(t.x, t.y, u, v)/W$
23: **end for**
24: $C^{\max}_{v}(b, p.x, p.y, v_{\text{offset}} : v_{\text{offset}} + W_{\text{block}}) \leftarrow \max(\bullet, C^{\max}_{v,\text{cache}}(t.x, t.y, 0 : W_{\text{block}}))$
25: $C^{\max}_{u}(b, p.x, p.y, u_{\text{offset}} : u_{\text{offset}} + H_{\text{block}}) \leftarrow \max(\bullet, C^{\max}_{u,\text{cache}}(t.x, t.y, 0 : H_{\text{block}}))$
26: $C^{\text{avg}}_{v}(b, p.x, p.y, v_{\text{offset}} : v_{\text{offset}} + W_{\text{block}}) \leftarrow \bullet + C^{\text{avg}}_{v,\text{cache}}(t.x, t.y, 0 : W_{\text{block}})$
27: $C^{\text{avg}}_{u}(b, p.x, p.y, u_{\text{offset}} : u_{\text{offset}} + H_{\text{block}}) \leftarrow \bullet + C^{\text{avg}}_{u,\text{cache}}(t.x, t.y, 0 : H_{\text{block}})$
28: **end for**

The algorithm can be divided into two phases for each block of $(u, v)$ displacement indices. In the first phase, a subvolume of $C$ is computed and stored in $C_{\text{cache}}$. During the second phase, $C_{\text{cache}}$ is used to update the corresponding subvolume of $C^{\max}_{v,\text{cache}}$, $C^{\text{avg}}_{v,\text{cache}}$, $C^{\max}_{u,\text{cache}}$ and $C^{\text{avg}}_{u,\text{cache}}$. Line one starts the iteration over all displacement blocks denoted by $(u_{\text{block}}, v_{\text{block}})$. Variables $u_{\text{offset}}$ and $v_{\text{offset}}$ are helper variables assigned with the minimum index in the current displacement block. The subvolume processed by thread $i, j$ in this iteration is $(i, j, u_{\text{offset}} : u_{\text{offset}} + u_{\text{blockMax}}, v_{\text{offset}} : u_{\text{offset}} + v_{\text{blockMax}})$. When considering all threads in the thread block, then the subvolume is $(i_{\text{offset}} : i_{\text{offset}} + H_{\text{block}}, j_{\text{offset}} : j_{\text{offset}} + W_{\text{block}}, u_{\text{offset}} : u_{\text{offset}} + u_{\text{blockMax}}, v_{\text{offset}} : v_{\text{offset}} + v_{\text{blockMax}})$.

Phase one is started by synchronizing all threads, to prevent overwriting the results of the previous iteration while some threads are still using them (line 4). When all threads in the block have arrived at line 4, overwriting the caches begins by initializing all values of the subvolume intermediary 4D correlation volume result to zero (line 5). After that, the subvolume results for the 3D correlation volume are reset. Maximum channel values are initialized with $-\infty$ while average channel ones are initialized to 0 (lines 6,7). Similarly to the displacement indices, the features are processed in

blocks to update the subvolume $C_{\text{cache}}$. Line 8 initiates the iteration over all feature blocks. For each block, the minimum feature index $f_{\text{offset}}$ in the block is computed. Next, the image features are loaded from global memory to the shared memory (line 10,11). Because threads in the same block share the values of $F_{2,\text{cache}}$, synchronization is performed in line 12 to prevent threads from advancing before all threads have completed loading their part of the features. Finally, each thread updates its responsibility range of $C_{\text{cache}}$ by iterating over all $(u, v)$ displacements in the block for its specific $(i, j) = (t.x, t.y)$ index. This operation is performed completely in the cache, resulting in the minimum possible delay for loading and storing the values in the arrays. After the updates for all blocks of image features have finished on all threads in the thread block, the subvolume $C_{\text{cache}}$ now contains the values of $C$ for its specific index range. Phase one ends by synchronizing all threads to ensure that all values of $C_{\text{cache}}$ are updated to completion (line 16). Synchronization in this case also prevents threads from overwriting the feature caches before all threads have finished the previous iteration.

In the second phase the entries of the 3D correlation cache are updated and the results are written back to the full 3D correlation volume in global memory. The first step is to iterate over all displacements in the current displacement block and update $C^{1:2}_{u,\text{cache}}(t.x, t.y, u)$ as well as $C^{1:2}_{v,\text{cache}}(t.x, t.y, v)$. Line 19 shows the update for $C^{\max}_{v,\text{cache}}$. For every index $v$, the maximum over all cached values for index $u$ is computed. In this context, the $\cdot$ symbol is a shorthand for the value on the left side of the assignment. Following $C^{\max}_{v,\text{cache}}$, $C^{\max}_{u,\text{cache}}$ is calculated in the same fashion with switched indices. The average channels are computed in a similar fashion. However the $u$ or $v$ indices are divided by the height or width of $I_2$ at level $l$. Then the results are accumulated (lines 21,22). These operations only involve cached arrays, therefore data loading times are minimized. Finally, the second step writes the update back to global memory. In lines 24-27, the update is performed in a similar fashion as lines 19-22. However, cached block results are used to update the global result by computing the maximum between the partial cache maximum and the current global maximum for each index. The global average is updated by adding the contribution of the cached block result.

By repeating phase one and phase two, eventually all displacement blocks are processed by every thread. Hence, every subvolume of $C$ has been computed once and used to update the 3D correlation volume channels $C^{1:2}_u$ and $C^{1:2}_v$. After all updates are complete, the 3D correlation channels in global memory have arrived at their final result.

A second custom function is required to compute the attention-based channels of the 3D correlation volume. The kernel for this function can be implemented similarly to the kernel for the maximum and average channels. Phase one of Algorithm 5.2 can be utilized in the same manner, since the attention based channels calculation also requires $C$. The only exception is that the caches of the maximum and average 3D correlation volume channels are replaced with caches for the attention based channels that are initialized to zero. In contrast, the second phase has to be modified heavily, by introducing a new step that loads the attention weights $A^{k+2}_u$ and $A^{k+2}_v$ for each displacement block from global memory to the cache. In addition, loops over the attention-based channels with index $k$ are added to the in-cache 3D correlation volume update and cache to global memory update. Furthermore, the in-cache update equation is $C^{k+2}_v(t.x, t.y, v) \leftarrow \cdot + (C(t.x, t.y, u, v) \cdot A^{k+2}_v(t.x, t.y, u))$ in this case. For the cache to global memory update, the operation of incrementing the previous global result is used as previously for the average channels.

### 5.3.3 Custom Backward Pass

To be able to train the model without storing $C$, the two custom functions need to have a backward pass function that complements their forward pass function. This section shows the derivatives of the equations for the forward pass. Furthermore, it shows how the backward pass was implemented.

The first step is to derive the equations for the backward pass. Each one of the maximum, average and attention-based channels of $C_u$ and $C_v$ requires derivatives for every input $F_1$, $F_2$ and $A^k$.

The PyTorch backward function receives the gradient of the loss $L$ with respect to the outputs of the corresponding forward function. It is expected to return the gradients of its inputs with respect to $L$.

$$\begin{aligned} X_k &= g_k(\dots, i_p, \dots) \\ \left(\frac{\partial L}{\partial i_p}\right)_{g_k} &= \frac{\partial L}{\partial X_k} \frac{g_k(\dots, i_p, \dots)}{\partial i_p} \end{aligned} \tag{5.9}$$

This means that the derivative can be computed by applying the chain rule as shown in Equation (5.9). In this equation, $X_k$ is the output tensor of function $g_k$ which receives $i_p$ as an input tensor.

$$(C_u^{\max})_{i,j,u} = \max_{v \in V}\left((F_1^0)_{i,j} \cdot (F_2^l)_{u,v}\right) = (F_1^0)_{i,j} \cdot (F_2^l)_{u,\tilde{v}} \text{ with } \tilde{v} = \operatorname*{argmax}_{v \in V}\left((F_1^0)_{i,j} \cdot (F_2^l)_{u,v}\right) \tag{5.10}$$

Equation (5.10) shows the maximum channel of the 3D correlation volume $C_u$. If the index $\tilde{v}$ that maximizes the dot product is known for every index $(i, j, u)$ of $C_u$, then the maximum operation can be removed and only the dot product of $F_1^0(i, j)$ and $F_2^l(u, \tilde{v})$ remains.

$$\frac{\partial (C_u^{\max})_{i,j,u}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}}\left((F_1^0)_{i,j} \cdot (F_2^l)_{u,\tilde{v}}\right) = (F_2^l)_{u,\tilde{v}} \tag{5.11}$$

Next, the operation is derived by its first input $F_{i,j}^0$. The resulting derivative for every index $(i, j, u)$ is shown in Equation (5.11).

$$\left(\frac{\partial L}{\partial (F_1^0)_{i,j}}\right)_{C_u^{\max}} = \frac{\partial L}{\partial (C_u^{\max})_{i,j}} \frac{\partial (C_u^{\max})_{i,j}}{\partial (F_1^0)_{i,j}} = \sum_{u \in U} (F_2^l)_{u,\tilde{v}} \frac{\partial L}{\partial (C_u^{\max})_{i,j,u}} \tag{5.12}$$

Now the equation for the derivative of $C_u^{\max}$ is known. However the backward function calls for the derivative of the loss $L$. By applying the chain rule as Equation (5.9) and accumulating over indices $u$ the resulting Equation (5.12) describes the contribution of $C_u^{\max}$ to the derivative of the loss with respect to the input $F_1^0$ for every index $(i, j)$.

$$\frac{(C_u^{\max})_{i,j,u}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}}\left((F_1^0)_{i,j} \cdot (F_2^l)_{u,\tilde{v}}\right) = [v = \tilde{v}_{i,j,u}](F_1^0)_{i,j} \tag{5.13}$$

$$\left(\frac{\partial L}{\partial (F_2^l)_{u,v}}\right)_{C_u^{\max}} = \frac{\partial L}{\partial (C_u^{\max})_u} \frac{\partial (C_u^{\max})_u}{\partial (F_2^l)_{u,v}} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} [v = \tilde{v}_{i,j,u}](F_1^0)_{i,j} \frac{\partial L}{\partial (C_u^{\max})_{i,j,u}} \tag{5.14}$$

Another input in the calculation of $C_u^{\max}$ is $F_2^l$. Equation (5.13) and (5.14) show the corresponding derivatives for this input. Since the derivative of $(F_2^l)_{u,\tilde{v}}$ with respect to $(F_2^l)_{u,v}$ is only non-zero in case $v = \tilde{v}_{i,j,u}$, the condition $[v = \tilde{v}_{i,j,u}]$ is added. If the equality holds, it evaluates to one, otherwise it evaluates to zero.

| Model | Settings | | | GPU(MiB) | | Time(s/it) | | |
|---|---|---|---|---|---|---|---|---|
| **Method** | **MemSave** | **Patch Size** | **BS** | **T(12)** | **I(12,32)** | **T(12)** | **I(12)** | **I(32)** |
| RAFT | - | (320, 448) | 12 | 16877 | 1475 | - | - | - |
| NoAgg | No | (320, 448) | 12 | 22777 | 1472 | 2.56 | 0.19 | 0.32 |
| NoAgg | Yes | (320, 448) | 12 | 21894 | 1447 | 9.34 | 0.21 | 0.33 |
| RAFT | - | (512, 1024) | 1 | 6846 | 2195 | - | - | - |
| NoAgg | No | (512, 1024) | 1 | 9205 | 2305 | 2.32 | 0.62 | 0.78 |
| NoAgg | Yes | (512, 1024) | 1 | 8160 | 1799 | 8.31 | 0.79 | 0.95 |

**Table 5.1:** Results for the alternative memory saving implementation compared to the original one. Consumed GPU memory and time per iteration were measured during both training and inference time. BS: batch size, T: training, I: inference.

Similarly to $C_u^{\max}$, derivatives of the channel for the other correlation volume $C_v^{\max}$ need to be determined for every input. In addition, the derivatives for the average channels $C_u^{\text{avg}}$, $C_v^{\text{avg}}$ and the attention-based channels $C_u^{k+2}$, $C_v^{k+2}$ need to be determined for every input.

The implementation of the backward pass calculates other derivatives such as $C_v^{\max}$, $C_u^{\text{avg}}$, $C_u^{k+2}$ and many more. Since the number of the derivatives is large and they are similar, they will not be described in this chapter. All derivatives are listed alongside their corresponding forward equations in the appendix A.

Because the backward pass is computationally more expensive than the forward pass, it is important to apply the same speedup techniques as in the forward pass. Therefore, the backward functions of the forward pass were also implemented using custom PyTorch functions that are running CUDA kernels. As in the forward pass, two kernels are set up. The first one computes derivatives for maximum and average channels, while the second one is responsible for the derivatives of the attention based channels. Indices $\tilde{v}(i,j,u)$ and $\tilde{u}(i,j,v)$ in $C$ of the maximum $C_u^{\max}(i,j,u)$, $C_v^{\max}(i,j,v)$ are needed during the backward pass. Therefore, at training time the kernel of the forward pass is modified to compute the argmax $\tilde{v}(i,j,u)$ and $\tilde{u}(i,j,v)$ alongside the maximum channel. During the forward pass, all relevant inputs and outputs of the forward pass required for the backward pass such as $F_1^0(i,j,u)$, $F_2^l(i,j,u)$, $\tilde{v}(i,j,u)$ and $\tilde{u}(i,j,v)$ are stored. A similar strategy to utilize shared memory as in the forward pass is followed for some parts of the backward pass implementation.

As a result of the large number of derivatives, implementing and testing the kernels is laborious. Thus the kernels are not as well optimized compared to the forward pass. The implementation of the backward pass equations will not be discussed further.

## 5.4 Results and Discussion

By providing a description of the operations for calculating $C_u$ and $C_v$ without storing $C$, it was shown in Section 5.3 that it is possible to omit the storage of the 4D correlation volume. Now that the alternative implementation of the forward and backward pass have been introduced, the results will be presented to show that sizeable storage savings are achieved while keeping computation

times feasible. Table 5.1 shows the results of the implementation with and without the alternative memory-saving 3D correlation volume computation. Experiments were performed with the *NoAgg* model, which corresponds to the last ablation step in Chapter 4. For comparison, the same experiment with regard to storage was performed using the default *RAFT* model that stores the 4D correlation volume. In this case, *MemSave: Yes* indicates that the alternative, memory saving implementation was used. Results at training time are listed with 12 refinement iterations, while at inference time both results for 12 and 32 iterations are listed. Furthermore, at training time, the specified batch size was used for the measurement. However, for measurements at inference time a batch size of 1 was used, irrespective of the batch size specified in the table. The storage consumption and timing results were measured on a single NVIDIA TITAN RTX GPU. To measure the storage consumption, a simplified version of the training and inference procedure that is compatible with both RAFT and Separable Flow was executed. After each training or inference iteration, the memory consumption on the GPU was measured using the function `torch.cuda.mem_get_info`. The largest memory consumption reported by this function over all iterations is the final memory consumption displayed in the table. To measure the training speed, the timestamps after every 100 training iterations in log files from training were analyzed. Finally, inference speeds were measured using the PyTorch benchmark class `torch.utils.benchmark.Timer` to run and average the times of full forward passes of the models.

The first three rows contain results for a small image size of $(320, 448)$ and a large batch size of 12. From the second to third row, 884MiB of memory are saved during training. Dividing by the batch size, around 74MiB are saved for each image in the batch. At inference time on the other hand, only 25MiB of memory are saved for each image, which is only a third of the saving at training time. Similarly, the gap between the relative increase in training and inference time is large. While the training time increases by a factor of around 3.6, the inference time only increases by 1.1 times the previous value.

Compared to the results for *RAFT* in the first row, the memory consumption during training for the Separable Flow models called *NoAgg* is much higher. The difference during training is at least 5017MiB However, at inference time the additional storage consumption between *RAFT* and *NoAgg* without memory saving is negligible with difference of 3MiB in favor of *NoAgg*. Therefore, the difference is also in favor of *NoAgg* when memory saving is enabled and the difference amounts to 28MiB.

To highlight the increased savings for larger image sizes, the experiment was repeated with a larger image size of $(512, 1024)$ and a batch size of 1. Consequently, the number of pixels increased from 143360 to 524228 by an approximate factor of 3.7. First the *NoAgg* models will be compared. A larger reduction of 1045MiB in GPU storage consumption can be observed between original and alternative implementation for the single image in the batch. At inference time the reduction is halved with a decrease of 506MiB. However, the relative decrease of around 22% is especially noticeable in this case because the absolute memory consumption during inference is only 2305MiB or 1799MiB depending on the implementation. Similarly to the small image size, the time per iteration during training is increased by a factor of around 3.6. During inference, the difference in time per iteration increased by a factor of 1.5 for 12 refinement iterations and 1.2 for 32 iterations, which is much larger than for the smaller image size.

Comparing the *NoAgg* models with *RAFT* on the large image size shows similar results at training time than for the smaller image. RAFT consumes at least 1314MiB less than the *NoAgg* models. During inference, *RAFT* consumes 110MiB less than *NoAgg* without memory saving, whereas

before the difference was negligible. When using memory saving with *NoAgg* the results are once more in favor of the Separable Flow model by a margin of 396MiB.

In Section 5.1 the expected memory savings were discussed from a theoretical standpoint. For training and inference on the small image pairs, storage consumption differences of around 26.8MiB and 53.6MiB were expected. The results showed 25MiB and 74MiB, thus the training has a larger memory difference than expected. The prediction may overlook an occasion that requires additional storage of the same size as the 4D correlation pyramid, which would lead to a more accurate prediction of 80.4MiB especially when subtracting the size of the image feature pyramid $F_2^l$.

Based on the results, the most useful application for the alternative implementation may be for the inference of image pairs with a high pixel count. It can be observed that the storage savings increase dramatically when raising the pixel count. At the same time, the inference time for a large number of 32 refinement iterations only increases by a factor of 1.2. This may be a good trade-off in applications where the global GPU memory is limited and no training will be performed. During training with large batch sizes, the memory of the GPU can be exhausted rapidly. However, due to applying data augmentations such as cropping and limited resolution of the images in the training dataset, usually the image size is not as large during training compared to inference. Furthermore, the large increase of training time caused by the alternative implementation may lead to a trade-off for saving memory that will be rarely worthwhile.

The memory consumption of Separable Flow during training with and without memory saving seems especially large compared to RAFT. Even when omitting the storage of the 4D correlation volume, Separable Flow models still consume much more memory than RAFT. Since the implementation of Separable Flow is based on RAFT, the additional memory requirement has to be caused by expensive additional operations such as 3D cost volume aggregation introduced by Separable Flow. In contrast to training time, the *NoAgg* model with memory saving consumes significantly fewer memory than *RAFT* during inference. This may be attributed to the large storage savings of the memory saving implementation for large images. In addition, expensive operations with regard to the storage size of intermediary results may be overshadowed during inference since such results can be deleted as soon as they have served as input to the next operation. It has to be noted that the default *RAFT* model was used, which stores the 4D correlation volume. Therefore, RAFT may achieve storage requirements that are lower than *NoAgg* with memory saving enabled, when using the alternative implementation. The alternative implementation enables RAFT to omit the storage of 4D correlation volume.

This chapter investigated the claim made by the paper that omitting the storage of the 3D correlation volume is possible. By describing the equations to accomplish this, it was shown that this is possible. After that, an implementation of the equations in the form of a GPU kernel was presented. The purpose of the implementation was to examine whether the computation is feasible with respect to the time taken. The results showed that the training time increased by around 3.6 times and the inference time by between 1.1-1.5 of the original when using the alternate implementation. Thus, it is feasible to perform training and inference with the alternate implementation and save around 1GiB of memory during training for an image size of $(512, 1024)$ and batch size of 1. Compared to RAFT however, the memory consumption during training is much larger. Therefore, the presented memory saving approach does not lead to an advantage during training with regard to memory consumption compared to RAFT.

# 6 Combining Separable Flow with Global Motion Aggregation

This section describes how Global Motion Aggregation can be integrated with Separable Flow to improve the estimation quality. Global Motion Aggregation was already introduced in Section 3.8, where its structure is explained in detail. In the next section, the approach of integrating GMA with Separable Flow will be described. After that, the results of GMA in combination with the original model and final ablation step model are presented.

## 6.1 Approach

To combine GMA with a method that is similar to RAFT, the initialization phase and the iterative refinement phase of the method need to be modified. The initialization phase computes the inputs for the refinement phase such as the context features and the 4D correlation volume $C$. During the motion refinement phase, motion features are computed and the recurrent update step is performed.

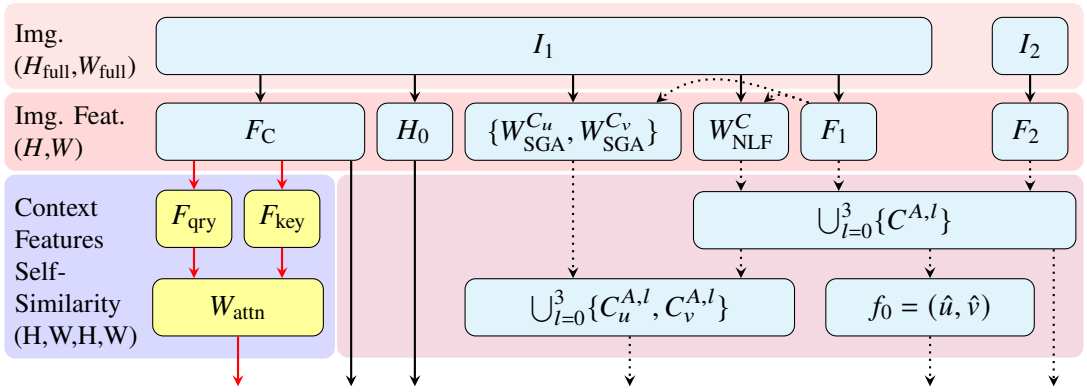


**Figure 6.1:** Data dependencies of the Separable Flow initialization phase with GMA additions highlighted in yellow.

Figure 6.1 shows the initialization phase of Separable Flow. Because the computation of the 3D and 4D correlation volume pyramids as well as the initial flow is not relevant to GMA, the depiction of their data dependencies is greatly simplified. All data dependencies relevant to GMA such as the self similarity of the projected context features $W_{\text{attn}}$ are highlighted in yellow. This self similarity

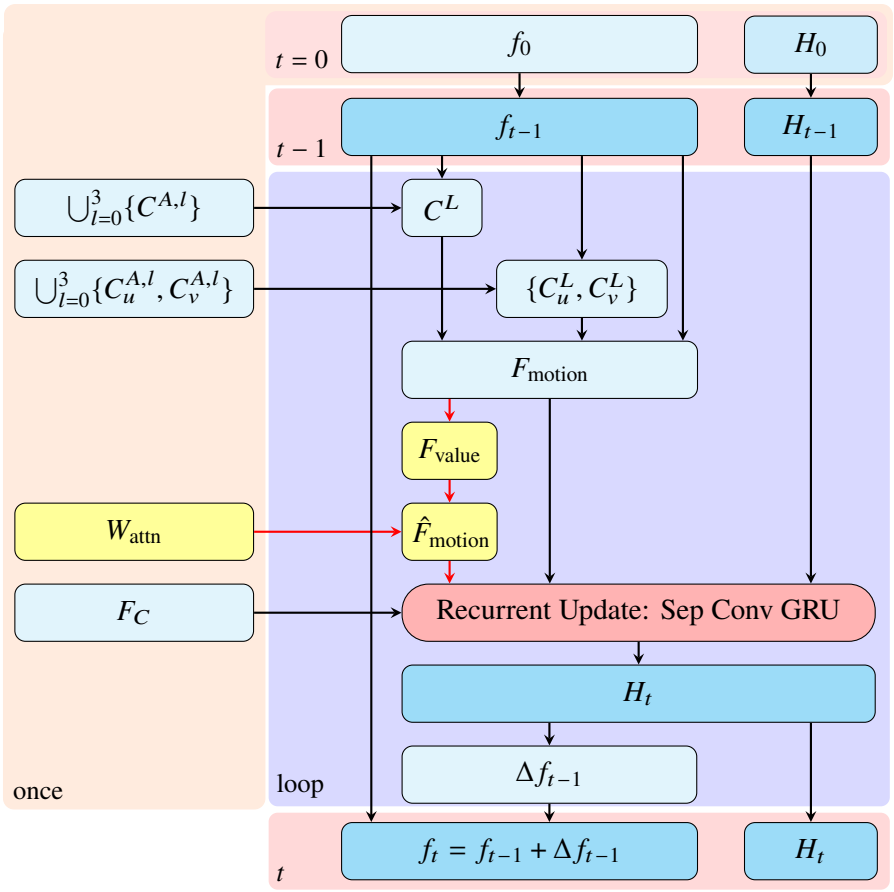


**Figure 6.2:** Data dependencies of the Separable Flow refinement phase with additions of GMA highlighted in yellow.

is computed only once per image pair and it is responsible for globally weighting the contributions of the motion features.

$$\begin{array}{rcl|rcl} x, y & \in & \{0, 1, \ldots, HW-1\} & (F_{\text{qry}})_x & = & W_{\text{qry}}(F_C)_x \\ W_{\text{qry}}, W_{\text{key}} & \in & \mathbb{R}^{\text{channels}(F_C) \times \text{channels}(W_{\text{qry}})} & (F_{\text{key}})_y & = & W_{\text{key}}(F_C)_y \end{array} \tag{6.1}$$

The key features $F_{\text{key}}$ and query features $F_{\text{qry}}$ for the attention mechanism are computed using a 2D convolutional layer with a kernel size of one. It is applied to the context features, which corresponds to a matrix multiplication for each pixel. Thus the context features $F_C$ at each pixel $x$ are projected to arrive at $F_{\text{key}}$ and $F_{\text{qry}}$.

$$(\tilde{W}_{\text{attn}})_{x,y} = \frac{(F_{\text{qry}})_x \cdot (F_{\text{key}})_y}{\sqrt{\text{channels}(F_{\text{qry}})}} \qquad (W_{\text{attn}})_x = \text{softmax}\big((\tilde{W}_{\text{attn}})_x\big) \tag{6.2}$$

Finally the self similarity is computed by taking the scalar product between the query and key features of all pixels and normalizing them by the square root of the size of the query features. The softmax operation is applied to have the similarities for pixel $x$ and every other pixel $y$ sum to one.

| Model | | | Sintel (train) | | KITTI (train) | |
|---|---|---|---|---|---|---|
| **Method** | **Chairs It.** | **Data** | **clean** | **final** | **epe** | **F1** |
| RAFT | 100K | C+T | 1.43 | 2.71 | 5.04 | 17.4 |
| SepFlow | 100K | C+T | 1.30 | **2.59** | **4.60** | **15.9** |
| GMA | 100K | C+T | 1.30 | 2.74 | 4.69 | 17.1 |
| Repo | 100K | C+T | 1.31 | 2.68 | 5.08 | 17.0 |
| NoAgg | 100K | C+T | 1.81 | 3.33 | 10.50 | 32.0 |
| Provided + GMA | 100K | C+T | **1.27** | 2.67 | 4.79 | 16.6 |
| NoAgg + GMA | 100K | C+T | 1.96 | 3.64 | 11.24 | 34.0 |

**Table 6.1:** Ablation results for integrating GMA into Separable Flow. Only 2-view initialization results are listed. Bold: Best overall results, Underlined: Best results of models trained in this thesis.

The iterative refinement phase of Separable Flow is depicted in Figure 6.2. After each iteration, the motion features are globally aggregated using the attention from the initialization step.

$$(F_{\text{value}})_x = W_{\text{value}}(F_{\text{motion}})_x \qquad (\hat{F}_{\text{motion}})_x = \sum_{y=0}^{HW-1} (W_{\text{attn}})_{x,y} \cdot (F_{\text{value}})_y \tag{6.3}$$

Value features $F_{\text{value}}$ are projected from the motion features. The motion features are then aggregated in a weighted sum over the elementwise product of $(W_{\text{attn}})_x$ and $F_{\text{value}}$ for each pixel $x$.

## 6.2 Results and Discussion

This section discusses the results of integrating GMA with Separable Flow. Table 6.1 shows the results of adding GMA to two different models of Separable Flow. The implementation allows to combine GMA with any of the ablation steps. In this case the original model *Provided* and the final ablation step *NoAgg* with 100K training iterations were chosen. These models were chosen because the original model performed the best, while the final ablation step is closest to the model described in the paper. All listed models were trained with the C+T schedule and evaluated on the Sintel and KITTI training datasets.

The first three rows contain results for the methods RAFT, Separable Flow and GMA from their respective paper. Separable Flow has the best reported results among those methods. Rows four and five display the results of the *Provided* and *NoAgg* model. Finally, the last two rows show the results for the previous two models combined with GMA. For the final ablation model *NoAgg* the results are worse with than without GMA, thus showing the worst overall results. In contrast, the original model performs better with GMA. When evaluating on the training datasets, it even outperforms GMA and has the best overall results on Sintel clean.

| Model | | | Sintel(test) | |
|---|---|---|---|---|
| **Method** | **Chairs It.** | **Data** | **clean** | **final** |
| RAFT (2-view) | 100K | C+T+S+K+H | 1.94 | 3.18 |
| RAFT (warm-start) | 100K | C+T+S+K+H | 1.61 | 2.86 |
| SepFlow (2-view) | 100K | C+T+S+K+H | 1.50 | 2.67 |
| GMA (warm-start) | 100K | C+T+S+K+H | 1.39 | 2.47 |
| Provided + GMA (2-view) | 100K | C+T+S+K+H | 1.55 | 2.79 |

**Table 6.2:** Sintel refinement results for integrating GMA into Separable Flow.

Because the *Provided* model with GMA performed well during the ablations, the model was finetuned for Sintel with the training schedule C+T+S+K+H for an additional 100K iterations and evaluated on the Sintel test dataset. The results are displayed in Table 6.2. Rows one through four show the results of RAFT, Separable Flow and GMA with added information about the flow initialization. The flow for the RAFT and GMA methods can either be zero-initialized (2-view) or initialized with the previous forward-projected flow estimate (warm-start). Only the warm-start result was reported for GMA on Sintel test. Therefore, in contrast to the previous table, results will also be listed with warm-start instead of 2-view only. Since Separable Flow uses motion regression to arrive at an initial flow estimate, it does not use the previous result. Therefore it is a 2-view method. On Sintel test, the results of GMA are superior to all other methods. The reported results for Separable Flow are worse than GMA, but better than RAFT. *Provided + GMA* ranks between RAFT with warm-start and Separable Flow.

In conclusion, the ablation step for the original model with GMA seemed promising. However, the finetuned model for Sintel could not compete with the results of GMA warm-start, which is not a completely unexpected result for the following reasons. The reported results for Separable Flow during the ablations were the best overall, outperforming GMA. However, on the test dataset GMA yielded better results than Separable Flow. Because the provided original model is a modified version of the model structure and training schedule described in Separable Flow, it may also experience the same performance drop on the test dataset compared to GMA. Furthermore, the usage of warm-start initialization may be the deciding factor that gives GMA the advantage over Separable Flow and the *Provided* model which use 2-view initialization.

# 7 Conclusion and Outlook

This thesis investigated the publication and implementation of the deep optical flow estimation approach Separable Flow. A comparison of the model structure, training regiment as well as the reported results was performed. The published method of separating a 4D cost volume into two 3D cost volumes promises state of the art estimation quality with the potential of reducing the required computational and memory resources.

Training and evaluating the implementation on the ablation schedule produced results inferior to the reported ones. Several differences between the publication and implementation were discovered and described. The implemented training schedule differed in the number of training iterations and image augmentation patch size from the reported one. By moving to the published number of training iterations, the ablation results could be greatly improved over the previous results. However, these results were still slightly worse than the reported ones.

Several discrepancies of the model structure were discovered as well. To start with, the implementation still used the 4D correlation volume for motion features although the publication stated that it is replaced by the 3D correlation volumes. Another difference was discovered in the implementation of the 3D correlation volumes themselves. They did not have learned channels, as described in the publication. Finally, a custom aggregation method was present in the implementation, which was absent from the paper.

Every structural change of the model was reverted and evaluated in a series of ablation steps. The 4D correlation volume motion features and aggregation were removed and the learned 3D correlation volume channels implemented. This experiment showed that especially the removal of the 4D correlation volume motion features worsened the results, which highlights the importance of using 4D correlation features during motion refinement. After that, the estimation quality was recovered slightly by adding the learned 3D correlation volume channels. However, none of the ablation steps yielded better results than the original model implementation with increased training iterations. Compared to RAFT which uses only 4D correlation features, the superior performance of this model on the ablation schedule indicates potential benefits of additionally including the 3D correlation features. While the ablation steps lowered the parameter count from 8.35M to 7.56M parameters, the reported parameter count of 6.0M parameters could not be reached. This suggests that it was not possible to revert all structural changes.

To verify the claim that the 4D correlation volume does not need to be stored, the correlation volume separation was implemented with on-demand computation of 4D correlation volume indices. For efficiency reasons, the PyTorch CUDA extension mechanism was used. The parallel GPU separation implementation of the forward and backward pass was described. It was shown that the use of this implementation is possible and feasible at both inference and training time. For a large image size, more than one gigabyte of GPU memory could be saved during training and 500 megabytes at inference time. The training time was increased by a factor of around 3.6 whereas the inference

time only increased slightly. Therefore, the alternative implementation may be especially suitable for inference of large images, while training may be performed with the regular implementation to save time.

Finally, Global Motion Aggregation was added as a submodule to Separable Flow to improve the estimation quality. The results on the ablation training schedule were promising, achieving better results than GMA and the best overall results on the Sintel clean dataset. However, when finetuning and testing on Sintel, GMA dominated while the combined method was still better than RAFT with warm-starting.

## Outlook

**Additional Experiments** The publication of Separable Flow is not very specific with respect to the parameters of its modules' layers such as convolutional layers. Hence, with the availability of computing resources to perform many ablations, further experiments may be conducted to improve the estimation quality or reduce the parameter count. For example, the paper does not specify the dimension of the parameters of the 3D convolutional layer that is used to learn the aggregation weights for the self adaptive 3D correlation volume channels. As the implementation does not include those channels, there is no specification for these layers. In consequence, they were chosen in this thesis as the standard kernel size of $(3, 3, 3)$. Further experiments may show improved results for different kernel sizes. Another experiment could be conducted with a model that has aggregation networks with fewer parameters. Especially the size and number of layers between the aggregated 3D correlation volumes and the motion regression of the initial flow could be reduced. To this end, the suggested modification in Section 4.2.4 may be tested by training and evaluating additional ablation steps. This could reduce the parameter count and fit the published description better.

**Backward Pass Optimization** The training speed is still much slower when using the alternative 3D correlation volume implementation, especially when considering the difference of the inference speed. This can be attributed to the backward pass which was implemented as a parallel CUDA program that runs on the GPU but was optimized not optimized to the same degree as the forward pass. Because of the large amount of derivative equations that need to be implemented, it was too tedious to optimize this fully by using shared memory in all cases. Therefore, the backward pass can be optimized further by expanding the use of shared memory to all parts of the implementation. Moreover, the computation time may be decreased by running one thread for each of the displacement dimensions, while the inner loops iterate over the dimensions of the frame one features. This is especially useful for the implementation of some of the derivatives because data conflicts and synchronization overhead can be avoided. A faster backward pass may enable the training of Separable Flow models on inexpensive GPUs with less memory and acceptable training times.

# A Equations of the Backward Pass

This chapter lists the equations for the forward and backward pass of the 3D correlation volume computation with on-demand 4D correlation volume. Each of the loss derivatives with respect to one input were implemented as part of a PyTorch CUDA extension.

### Forward equations for the maximum channels $C_u^{\max}$ and $C_v^{\max}$

$$(C_u^{\max})_{i,j,u} = \max_{v \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) = (F_1^0)_{i,j} \cdot (F_2^l)_{u,\tilde{v}} \tag{A.1}$$

$$(C_v^{\max})_{i,j,v} = \max_{u \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) = (F_1^0)_{i,j} \cdot (F_2^l)_{\tilde{u},v} \tag{A.2}$$

$$\text{with } \tilde{v}_{i,j,u} = \operatorname*{argmax}_{v \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) \text{ and } \tilde{u}_{i,j,v} = \operatorname*{argmax}_{u \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) \tag{A.3}$$

### Derivative and backward equations for the maximum channel $C_u^{\max}$

$$\frac{\partial (C_u^{\max})_{i,j,u}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,\tilde{v}} \right) = (F_2^l)_{u,\tilde{v}} \tag{A.4}$$

$$\left( \frac{\partial L}{\partial (F_1^0)_{i,j}} \right)_{C_u^{\max}} = \frac{\partial L}{\partial (C_u^{\max})_{i,j}} \frac{\partial (C_u^{\max})_{i,j}}{\partial (F_1^0)_{i,j}} = \sum_{u \in U} (F_2^l)_{u,\tilde{v}} \frac{\partial L}{\partial (C_u^{\max})_{i,j,u}} \tag{A.5}$$

$$\frac{(C_u^{\max})_{i,j,u}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,\tilde{v}} \right) = [v = \tilde{v}_{i,j,u}] (F_1^0)_{i,j} \tag{A.6}$$

$$\left( \frac{\partial L}{\partial (F_2^l)_{u,v}} \right)_{C_u^{\max}} = \frac{\partial L}{\partial (C_u^{\max})_u} \frac{\partial (C_u^{\max})_u}{\partial (F_2^l)_{u,v}} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} [v = \tilde{v}_{i,j,u}] (F_1^0)_{u,v} \frac{\partial L}{\partial (C_u^{\max})_{i,j,u}} \tag{A.7}$$

### Derivative and backward equations for the maximum channel $C_v^{\max}$

$$\frac{\partial (C_v^{\max})_{i,j,v}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{\tilde{u},v} \right) = (F_2^l)_{\tilde{u},v} \tag{A.8}$$

$$\left( \frac{\partial L}{\partial (F_1^0)_{i,j}} \right)_{C_v^{\max}} = \frac{\partial L}{\partial (C_v^{\max})_{i,j}} \frac{\partial (C_v^{\max})_{i,j}}{\partial (F_1^0)_{i,j}} = \sum_{v \in V} (F_2^l)_{\tilde{u},v} \frac{\partial L}{\partial (C_v^{\max})_{i,j,v}} \tag{A.9}$$

$$\frac{(C_v^{\max})_{i,j,v}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{\tilde{u},v} \right) = [u = \tilde{u}_{i,j,u}] (F_1^0)_{i,j} \tag{A.10}$$

$$\left( \frac{\partial L}{\partial (F_2^l)_{u,v}} \right)_{C_v^{\max}} = \frac{\partial L}{\partial (C_v^{\max})_v} \frac{\partial (C_v^{\max})_v}{\partial (F_2^l)_{u,v}} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} [u = \tilde{u}_{i,j,v}] (F_1^0)_{i,j} \frac{\partial L}{\partial (C_v^{\max})_{i,j,v}} \tag{A.11}$$

### Forward equations for the average channels $C_u^{\text{avg}}$ and $C_v^{\text{avg}}$

$$(C_u^{\text{avg}})_{i,j,u} = \frac{1}{|V|} \sum_{v \in V} (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \tag{A.12}$$

$$(C_v^{\text{avg}})_{i,j,v} = \frac{1}{|U|} \sum_{u \in U} (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \tag{A.13}$$

### Derivative and backward equations for the average channel $C_u^{\text{avg}}$

$$\frac{\partial (C_u^{\text{avg}})_{i,j,u}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}} \left( \frac{1}{|V|} \sum_{v \in V} (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) = \frac{1}{|V|} \sum_{v \in V} (F_2^l)_{u,v} \tag{A.14}$$

$$\left( \frac{\partial L}{\partial (F_1^0)_{i,j}} \right)_{C_u^{\text{avg}}} = \frac{\partial L}{\partial (C_u^{\text{avg}})_{i,j}} \frac{\partial (C_u^{\text{avg}})_{i,j}}{\partial (F_1^0)_{i,j}} = \frac{1}{|V|} \sum_{u \in U} \left( \sum_{v \in V} (F_2^l)_{u,v} \right) \frac{\partial L}{\partial (C_u^{\text{avg}})_{i,j,u}} \tag{A.15}$$

$$\frac{\partial (C_u^{\text{avg}})_{i,j,u}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}} \left( \frac{1}{|V|} \sum_{\hat{v} \in V} (F_1^0)_{i,j} \cdot (F_2^l)_{u,\hat{v}} \right) = \frac{1}{|V|} (F_1^0)_{i,j} \tag{A.16}$$

$$\left( \frac{\partial L}{\partial (F_2^l)_{u,v}} \right)_{C_u^{\text{avg}}} = \frac{\partial L}{\partial (C_u^{\text{avg}})_u} \frac{\partial (C_u^{\text{avg}})_u}{\partial (F_2^l)_{u,v}} = \frac{1}{|V|} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} (F_1^0)_{i,j} \frac{\partial L}{\partial (C_u^{\text{avg}})_{i,j,u}} \tag{A.17}$$

### Derivative and backward equations for the average channel $C_v^{\text{avg}}$

$$\frac{\partial (C_v^{\text{avg}})_{i,j,v}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}} \left( \frac{1}{|U|} \sum_{u \in U} (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) = \frac{1}{|U|} \sum_{u \in U} (F_2^l)_{u,v} \tag{A.18}$$

$$\left( \frac{\partial L}{\partial (F_1^0)_{i,j}} \right)_{C_v^{\text{avg}}} = \frac{\partial L}{\partial (C_v^{\text{avg}})_{i,j}} \frac{\partial (C_v^{\text{avg}})_{i,j}}{\partial (F_1^0)_{i,j}} = \frac{1}{|U|} \sum_{v \in V} \left( \sum_{u \in U} (F_2^l)_{u,v} \right) \frac{\partial L}{\partial (C_v^{\text{avg}})_{i,j,v}} \tag{A.19}$$

$$\frac{\partial (C_v^{\text{avg}})_{i,j,v}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}} \left( \frac{1}{|U|} \sum_{\hat{u} \in U} (F_1^0)_{i,j} \cdot (F_2^l)_{\hat{u},v} \right) = \frac{1}{|U|} (F_1^0)_{i,j} \tag{A.20}$$

$$\left( \frac{\partial L}{\partial (F_2^l)_{u,v}} \right)_{C_v^{\text{avg}}} = \frac{\partial L}{\partial (C_v^{\text{avg}})_v} \frac{\partial (C_v^{\text{avg}})_v}{\partial (F_2^l)_{u,v}} = \frac{1}{|U|} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} (F_1^0)_{i,j} \frac{\partial L}{\partial (C_v^{\text{avg}})_{i,j,v}} \tag{A.21}$$

**Forward equations for the attention-based channels $C_u^{k+2}$ and $C_v^{k+2}$**

$$(C_u^{k+2})_{i,j,u} = \sum_{v \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) (A_u^k)_{i,j,v} \tag{A.22}$$

$$(C_v^{k+2})_{i,j,v} = \sum_{u \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) (A_v^k)_{i,j,u} \tag{A.23}$$

**Derivative and backward equations for the attention-based channel $C_u^{k+2}$**

$$\frac{\partial (C_u^{k+2})_{i,j,u}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}} \left( \sum_{v \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) (A_u^k)_{i,j,v} \right) = \sum_{v \in V} (F_2^l)_{u,v} (A_u^k)_{i,j,v} \tag{A.24}$$

$$\left( \frac{\partial L}{\partial (F_1^0)_{i,j}} \right)_{C_u^{k+2}} = \frac{\partial L}{\partial (C_u^{k+2})_{i,j}} \frac{\partial (C_u^{k+2})_{i,j}}{\partial (F_1^0)_{i,j}} = \sum_{u \in U} \left( \sum_{v \in V} (F_2^l)_{u,v} (A_u^k)_{i,j,v} \right) \frac{\partial L}{\partial (C_u^{k+2})_{i,j,u}} \tag{A.25}$$

$$\frac{\partial (C_u^{k+2})_{i,j,u}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}} \left( \sum_{\hat{v} \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,\hat{v}} \right) (A_u^k)_{i,j,\hat{v}} \right) = (F_1^0)_{i,j} (A_u^k)_{i,j,v} \tag{A.26}$$

$$\left( \frac{\partial L}{\partial (F_2^l)_{u,v}} \right)_{C_u^{k+2}} = \frac{\partial L}{\partial (C_u^{k+2})_u} \frac{\partial (C_u^{k+2})_u}{\partial (F_l^2)_{u,v}} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \left( (F_1^0)_{i,j} (A_u^k)_{i,j,v} \frac{\partial L}{\partial (C_u^{k+2})_{i,j,u}} \right) \tag{A.27}$$

$$\frac{\partial (C_u^{k+2})_{i,j,u}}{\partial (A_u^k)_{i,j,v}} = \frac{\partial}{\partial (A_u^k)_{i,j,v}} \left( \sum_{\hat{v} \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,\hat{v}} \right) (A_u^k)_{i,j,\hat{v}} \right) = (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \tag{A.28}$$

$$\left( \frac{\partial L}{\partial (A_u^k)_{i,j,v}} \right)_{C_u^{k+2}} = \frac{\partial L}{\partial (C_u^{k+2})_{i,j}} \frac{\partial (C_u^{k+2})_{i,j}}{\partial (A_u^k)_{i,j,v}} = \sum_{u \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) \frac{\partial L}{\partial (C_u^{k+2})_{i,j,u}} \tag{A.29}$$

**Derivative and backward equations for the attention-based channel $C_u^{k+2}$**

$$\frac{\partial (C_v^{k+2})_{i,j,v}}{\partial (F_1^0)_{i,j}} = \frac{\partial}{\partial (F_1^0)_{i,j}} \left( \sum_{u \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) (A_v^k)_{i,j,u} \right) = \sum_{u \in U} (F_2^l)_{u,v} (A_v^k)_{i,j,u} \tag{A.30}$$

$$\left( \frac{\partial L}{\partial (F_1^0)_{i,j}} \right)_{C_v^{k+2}} = \frac{\partial L}{\partial (C_v^{k+2})_{i,j}} \frac{\partial (C_v^{k+2})_{i,j}}{\partial (F_1^0)_{i,j}} = \sum_{v \in V} \left( \sum_{u \in U} (F_2^l)_{u,v} (A_v^k)_{i,j,u} \right) \frac{\partial L}{\partial (C_v^{k+2})_{i,j,v}} \tag{A.31}$$

$$\frac{\partial (C_v^{k+2})_{i,j,v}}{\partial (F_2^l)_{u,v}} = \frac{\partial}{\partial (F_2^l)_{u,v}} \left( \sum_{\hat{u} \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{\hat{u},v} \right) (A_v^k)_{i,j,\hat{u}} \right) = (F_1^0)_{i,j} (A_v^k)_{i,j,u} \tag{A.32}$$

$$\left( \frac{\partial L}{\partial (F_2^l)_{u,v}} \right)_{C_v^{k+2}} = \frac{\partial L}{\partial (C_v^{k+2})_v} \frac{\partial (C_v^{k+2})_v}{\partial (F_l^2)_{u,v}} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \left( (F_1^0)_{i,j} (A_v^k)_{i,j,u} \frac{\partial L}{\partial (C_v^{k+2})_{i,j,v}} \right) \tag{A.33}$$

$$\frac{\partial (C_v^{k+2})_{i,j,v}}{\partial (A_v^k)_{i,j,u}} = \frac{\partial}{\partial (A_v^k)_{i,j,u}} \left( \sum_{\hat{u} \in U} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{\hat{u},v} \right) (A_v^k)_{i,j,\hat{u}} \right) = (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \tag{A.34}$$

$$\left( \frac{\partial L}{\partial (A_v^k)_{i,j,u}} \right)_{C_v^{k+2}} = \frac{\partial L}{\partial (C_v^{k+2})_{i,j}} \frac{\partial (C_v^{k+2})_{i,j}}{\partial (A_v^k)_{i,j,u}} = \sum_{v \in V} \left( (F_1^0)_{i,j} \cdot (F_2^l)_{u,v} \right) \frac{\partial L}{\partial (C_v^{k+2})_{i,j,v}} \tag{A.35}$$

# Bibliography


[BBPW04] T. Brox, A. Bruhn, N. Papenberg, J. Weickert. “High accuracy optical flow estimation based on a theory for warping”. In: *European conference on computer vision*. Springer. 2004, pp. 25–36 (cit. on p. 19).

[BKH16] J. L. Ba, J. R. Kiros, G. E. Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016) (cit. on p. 23).

[BWSB12] D. J. Butler, J. Wulff, G. B. Stanley, M. J. Black. “A naturalistic open source movie for optical flow evaluation”. In: *European Conf. on Computer Vision (ECCV)*. Ed. by A. Fitzgibbon et al. (Eds.) Part IV, LNCS 7577. Springer-Verlag, Oct. 2012, pp. 611–625 (cit. on pp. 15, 16, 27, 39).

[BYPC16] N. Ballas, L. Yao, C. Pal, A. C. Courville. “Delving Deeper into Convolutional Networks for Learning Video Representations.” In: *ICLR (Poster)*. 2016 (cit. on p. 25).

[CGCB14] J. Chung, C. Gulcehre, K. Cho, Y. Bengio. “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *NIPS 2014 Workshop on Deep Learning, December 2014*. 2014 (cit. on p. 24).

[CGN14] H. Chao, Y. Gu, M. Napolitano. “A survey of optical flow techniques for robotics navigation applications”. In: *Journal of Intelligent & Robotic Systems* 73.1 (2014), pp. 361–372 (cit. on p. 15).

[CMBB14] K. Cho, B. van Merriënboer, D. Bahdanau, Y. Bengio. “On the Properties of Neural Machine Translation: Encoder–Decoder Approaches”. In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. 2014, pp. 103–111 (cit. on p. 24).

[CT98] R. Cutler, M. Turk. “View-based interpretation of real-time optical flow for gesture recognition”. In: *Proceedings Third IEEE International Conference on Automatic Face and Gesture Recognition*. IEEE. 1998, pp. 416–421 (cit. on p. 15).

[DFI+15a] A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazırbaş, V. Golkov, P. v.d. Smagt, D. Cremers, T. Brox. “FlowNet: Learning Optical Flow with Convolutional Networks”. In: *IEEE International Conference on Computer Vision (ICCV)*. 2015. URL: http://lmb.informatik.uni-freiburg.de/Publications/2015/DFIB15 (cit. on pp. 15, 39).

[DFI+15b] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. Van Der Smagt, D. Cremers, T. Brox. “Flownet: Learning optical flow with convolutional networks”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2758–2766 (cit. on pp. 16, 17).

[DV16] V. Dumoulin, F. Visin. “A guide to convolution arithmetic for deep learning”. In: *ArXiv e-prints* (Mar. 2016). eprint: 1603.07285 (cit. on p. 22).

[GLU12] A. Geiger, P. Lenz, R. Urtasun. “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012 (cit. on pp. 15, 16, 27).

[Hir08] H. Hirschmuller. “Stereo Processing by Semiglobal Matching and Mutual Information”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.2 (2008), pp. 328–341. DOI: 10.1109/TPAMI.2007.1166 (cit. on p. 18).

[Hoc91] S. Hochreiter. “Untersuchungen zu dynamischen neuronalen Netzen”. In: 1991 (cit. on p. 23).

[HRB+12] A. Hosni, C. Rhemann, M. Bleyer, C. Rother, M. Gelautz. “Fast cost-volume filtering for visual correspondence and beyond”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.2 (2012), pp. 504–511 (cit. on p. 18).

[HS81] B. K. Horn, B. G. Schunck. “Determining optical flow”. In: *Artificial intelligence* 17.1-3 (1981), pp. 185–203 (cit. on p. 15).

[HS97] S. Hochreiter, J. Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 24).

[HZRS16] K. He, X. Zhang, S. Ren, J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 27).

[IS15] S. Ioffe, C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456 (cit. on p. 23).

[JCL+21] S. Jiang, D. Campbell, Y. Lu, H. Li, R. Hartley. “Learning to Estimate Hidden Motions with Global Motion Aggregation”. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society. 2021, pp. 9752–9761 (cit. on pp. 3, 4, 16, 19, 35, 36).

[JMRB22] A. Jahedi, L. Mehl, M. Rivinius, A. Bruhn. “Multi-Scale Raft: Combining Hierarchical Concepts for Learning-Based Optical Flow Estimation”. In: *2022 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2022, pp. 1236–1240 (cit. on p. 16).

[KNH+16] D. Kondermann, R. Nair, K. Honauer, K. Krispin, J. Andrulis, A. Brock, B. Gussefeld, M. Rahimimoghaddam, S. Hofmann, C. Brenner, *et al.* “The HCI Benchmark Suite: Stereo and Flow Ground Truth With Uncertainties for Urban Autonomous Driving”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2016, pp. 19–28 (cit. on p. 39).

[LVW+20] Y. Lu, J. Valmadre, H. Wang, J. Kannala, M. Harandi, P. Torr. “Devon: Deformable volume network for learning optical flow”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2020, pp. 2705–2713 (cit. on p. 18).

[MG15] M. Menze, A. Geiger. “Object Scene Flow for Autonomous Vehicles”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015 (cit. on pp. 15, 16, 27, 39).

[MIH+16] N. Mayer, E. Ilg, P. Häusser, P. Fischer, D. Cremers, A. Dosovitskiy, T. Brox. "A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation". In: *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*. arXiv:1512.02134. 2016. URL: http://lmb.informatik.uni-freiburg.de/Publications/2016/MIFDB16 (cit. on pp. 15, 39).

[NVI22] NVIDIA. *CUDA C++ Programming Guide*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 11/02/2022) (cit. on p. 36).

[ODo05] P. O'Donovan. "Optical flow: Techniques and applications". In: *International Journal of Computer Vision* 1 (2005), p. 26 (cit. on p. 15).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on pp. 27, 31).

[RB17] A. Ranjan, M. J. Black. "Optical flow estimation using a spatial pyramid network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4161–4170 (cit. on p. 16).

[SSB14] H. Sak, A. W. Senior, F. Beaufays. "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition". In: (2014) (cit. on p. 24).

[SYLK18] D. Sun, X. Yang, M.-Y. Liu, J. Kautz. "PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018 (cit. on p. 17).

[TBF+16] D. Tran, L. Bourdev, R. Fergus, L. Torresani, M. Paluri. "Deep End2End Voxel2Voxel Prediction". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2016 (cit. on p. 16).

[TBKP12] M. Tao, J. Bai, P. Kohli, S. Paris. "SimpleFlow: A Non-iterative, Sublinear Optical Flow Algorithm". In: *Computer Graphics Forum*. Vol. 31. 2pt1. The Eurographs Association & John Wiley & Sons, Ltd. Chichester, UK. 2012, pp. 345–353 (cit. on p. 15).

[TD20] Z. Teed, J. Deng. "RAFT: Recurrent All Pairs Field Transforms for Optical Flow". In: *Europe Conference on computer Vision (ECCV)*. 2020 (cit. on pp. 3, 16, 18, 27, 30, 31, 53).

[UVL16] D. Ulyanov, A. Vedaldi, V. S. Lempitsky. "Instance Normalization: The Missing Ingredient for Fast Stylization". In: *CoRR* abs/1607.08022 (2016). arXiv: 1607.08022. URL: http://arxiv.org/abs/1607.08022 (cit. on p. 23).

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 25).

[WH18] Y. Wu, K. He. "Group normalization". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19 (cit. on p. 23).

[WZD+20] J. Wang, Y. Zhong, Y. Dai, K. Zhang, P. Ji, H. Li. “Displacement-invariant matching cost learning for accurate optical flow estimation”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. 2020, pp. 15220–15231 (cit. on p. 18).

[XRK17] J. Xu, R. Ranftl, V. Koltun. “Accurate optical flow via direct cost volume processing”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 1289–1297 (cit. on pp. 17, 18).

[YR19] G. Yang, D. Ramanan. “Volumetric correspondence networks for optical flow”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 2019, pp. 794–805 (cit. on p. 17).

[ZPB07] C. Zach, T. Pock, H. Bischof. “A duality based approach for realtime tv-l 1 optical flow”. In: *Joint pattern recognition symposium*. Springer. 2007, pp. 214–223 (cit. on p. 19).

[ZPYT19] F. Zhang, V. Prisacariu, R. Yang, P. H. Torr. “GA-Net: Guided Aggregation Net for End-to-end Stereo Matching”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 185–194 (cit. on pp. 18, 34, 46).

[ZQY+20] F. Zhang, X. Qi, R. Yang, V. Prisacariu, B. Wah, P. Torr. “Domain-invariant Stereo Matching Networks”. In: *Europe Conference on Computer Vision (ECCV)*. 2020 (cit. on p. 18).

[ZWPT21] F. Zhang, O. J. Woodford, V. A. Prisacariu, P. H. Torr. “Separable Flow: Learning Motion Cost Volumes for Optical Flow Estimation”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021, pp. 10807–10817 (cit. on pp. 3, 4, 16, 18, 19, 31, 43, 50, 51).

[ZZL+20] T. Zhang, H. Zhang, Y. Li, Y. Nakamura, L. Zhang. “Flowfusion: Dynamic dense rgb-d slam based on optical flow”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 7322–7328 (cit. on p. 15).

All links were last followed on November 25, 2022.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature