Efficient All-Pairs Correlation Volume Sampling for Optical Flow Estimation

Supplementary Material

Block Size	Row-Major	Patch-Major	Improvement
1^2	1.6 ± 0.4		–
2^2	2.5 ± 0.5	2.3 ± 0.5	8%
4^2	6.7 ± 1.1	4.2 ± 0.7	37%
8^2	20.6 ± 4.4	8.6 ± 1.4	58%
16^2	28.2 ± 8.0	27.1 ± 1.2	4%

Table 3. Percentage of sampled correlation volume cells depending on the block size and layout, measured over the *SINTEL* [1] training dataset.

8. Method Details

In this section, we provide more information on obtaining the block mask, the sampling patterns, as well as complexity analysis and implementation details of our method.

8.1. Obtaining the Block Mask

In Figure 7 we provide a conceptual visualization of how the block mask is computed, which is then used for the correlation volume analysis and to determine which blocks need to be computed. Note that in practice it is obtained through splatting into the block mask directly, or computed implicitly through the inter-thread voting.

8.2. Correlation Volume Sampling Patterns

In Table 3, we show that averaging the number of sampled values per row-major block increases the ratio of cells that need to be computed, but is still significantly lower than the full matrix for sufficiently small block sizes. In the second column, we show that by averaging rows in our patch-major manner, sparsity is significantly increased without any computational overhead. At the largest measured block size of 16^2, it becomes too large to maintain high level sparsity.

8.3. Implementation Details

In this subsection, we provide more details on our implementation of the fused correlation volume sampling algorithm, split in 3 parts according to our algorithmic design: 1) computing implicit computation mask; 2) joint computation of the next correlation block; 3) correlation value sampling. A simplified pseudocode is provided in Algorithm 1, kernel implementation is provided in Listings 2-4. It computes a single-level correlation whereas for multi-level pyramid an average-pooled F^2 is used for each level. We set $B = 8$ as a trade-off between being able to use efficient warp MMA operations for block computations and limiting the shared memory usage.

Computing Implicit Block Mask. To determine the blocks which need to be computed, in each thread we iterate over the $[-r, r]^2$ neighborhood around the target pixel and compute indices of blocks the value needs to be sampled from. In practice, it is not necessary to iterate over all offsets, as for any 3 consecutive offsets, at least 2 land in the same block. Therefore, we only iterate over offsets with a stride of the block size B.

Indices of all blocks that a local thread needs to compute are stored in a register array in strictly increasing order, and we store an index pointer to the first block. To determine the next block to compute across all threads in a thread block, we perform a parallel reduction of the local indices with atomic minimum operation to a shared memory scalar, initialized larger than the size of the row. If the current local block matches the next global block, the index pointer is advanced. If the global next block is larger than the total number of blocks, all blocks have been processed and the kernel can exit.

Computation of the Next Block. Having agreed on the next block that needs to be computed, all threads participate in its computation, distributing loading of the input values and performing matrix-matrix computations across all threads. At once, we perform `64x64x32` product with `16x8x16` MMA instructions, `bf16` inputs and `fp32` accumulation. The final correlation values are kept in registers for faster processing.

Processing of the Correlation Block. Only threads that voted for the currently computed block sample correlation values by computing the target coordinates for each offset, and splitting them into the block index and local coordinates within the block. Values that can be extracted from the block are added to a global final output buffer, initialized as zeros.

8.4. Component Analysis

To analyze the impact of different steps of our algorithm to the total runtime, we approximate the time spent in each step by adding early kernel exits, and observe the aggregated runtime increase over the whole dataset. The runtime breakdown is provided in Tab. 4.

8.5. Memory Complexity Analysis

Here we show that our algorithm achieves linear time and space complexity in the number of pixels.

Assuming equal-size feature maps, let $P = H \times W$ be the number of feature pixels per image, B the block size,

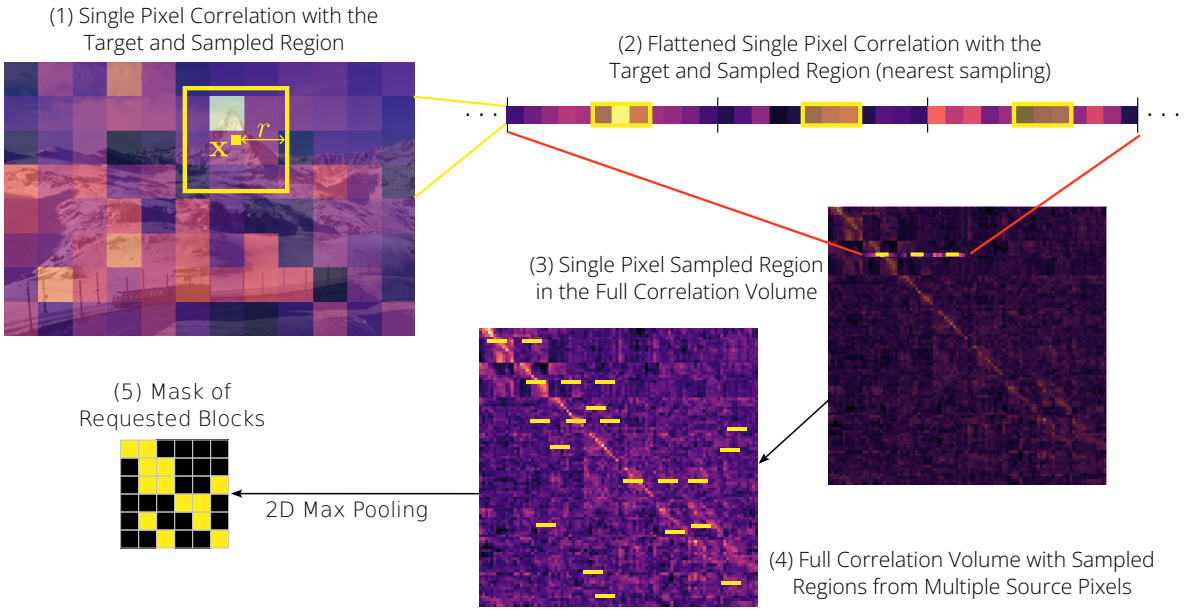


Figure 7. Conceptual visualization of the block mask computation.

Correlation Volume Width =	**256px**	**512px**
Preprocessing	11.3%	9.5%
Computing Per-Thread Blocks	**32.7%**	2.6%
Next-Block Voting	0.2%	0.3%
Matrix-Matrix Multiplication	0.4%	**25.5%**
Sampling Blocks and Writing Outputs	**55.3%**	**62.2%**
Total Runtime, ms	25.9	93.9

Table 4. Approximate runtime breakdown of the steps of our method.

and $K = 2r + 1$ the lookup region size. We also assume that both input tiles are fully stored in shared memory. In practice, larger channel sizes D is split into chunks of 32 elements.

As all lookup offsets are continuous, each source pixel can sample from $(K+1)^2$ pixels. In the worst case, the first pixel is in a separate block, and, in total, they spread across $(1 + \left\lceil \frac{(K+1)-1}{B} \right\rceil)^2 = \left\lfloor \frac{K+2B-1}{B} \right\rfloor^2$ blocks. With our hyperparameters $r = 4, B = 8$, it results in at most $3^2 = 9$ blocks per pixel.

The matrix-matrix multiplication for computing each tile requires B^4D operations, sampling it requires $\min(B^2, K^2)$ operations (if the block is smaller than the lookup region, at most all values are sampled). The total time complexity across all steps of the algorithm is therefore:

$$
\begin{aligned}
&P\left\lfloor \frac{K+2B-1}{B} \right\rfloor^2 \Big(\underbrace{1}_{\text{Obtain mask}} + \underbrace{B^4D}_{\text{Compute a tile}} + \underbrace{\min(B^2, K^2)}_{\text{Sample correlation}} \Big) \\
\leq & P(K+2B-1)^2 \frac{1}{B^2}(1 + B^4D + \min(B^2, K^2)) \\
= & P(K+2B-1)^2 (\frac{1}{B^2} + B^2D + \min(1, \frac{K^2}{B^2})) \qquad (3) \\
\in & O(P(K+B)^2 B^2 D),
\end{aligned}
$$

which is linear in the number of input pixels and feature dimensionality, and quadratic in the lookup radius. In practice, the number of requested blocks is lower due to smoothness of the optical flow and multiple source pixels requesting the same block, as shown in Table 3.

In addition to storing the inputs $O(PD)$ and outputs $O(PK^2)$, we also need to use registers and shared memory to store the block indices and inputs and outputs of one computed tile per thread block, requiring:

$$
\begin{aligned}
&P\left\lfloor \frac{K+2B-1}{B} \right\rfloor^2 + \frac{P}{B^2}\Big(2B^2D + B^4\Big) \\
\leq & P((\frac{K-1}{B^2})^2 + 2) + P(2D + B^2) \qquad (4) \\
\in & O(P(K^2 + D + B^2)),
\end{aligned}
$$

which, as inputs, is also linear in the number of pixels and feature dimensionality, quadratic in the lookup radius.

9. Extended Evaluation

In this section we provide extended evaluation results.

Algorithm 1 Single-Level Correlation Volume Sampling

Require: Flattened input features $F^{1,2} \in \mathbb{R}^{H \times W \times D}$, lookup centroids $\mathbf{X} \in \mathbb{R}^{N \times H \times W \times 2}$, block size B, lookup radius r, output buffer $\mathbf{O} \in \mathbb{R}^{N \times H \times W \times (2r+1)^2}$.

$bH, bW \leftarrow \lceil H/B \rceil, \lceil W/B \rceil$ ▷ Setting the number of blocks
$\bar{F}^{1,2} \leftarrow$ rearrange($\bar{F}^{1,2}$, '$(bH \times B_h) \times (bW \times B_w) \times D \rightarrow (bH \times bW) \times (B_h \times B_w) \times D$') ▷ Patch-major reshaping
`nBlocks` $\leftarrow bH \cdot bW$ ▷ The total number of target blocks
for $i = 0, 1, \ldots, N-1$ **do** ▷ For every lookup iteration
  On Device block $by \in [bH \cdot bW]$, thread $t \in [B^2]$
    $\mathbf{x} \leftarrow \mathbf{X}[i][by \cdot B^2 + t]$ ▷ Get the respective target coordinates
    `blocks` $\leftarrow$ [ ] ▷ Empty list of blocks to compute
    for all $\mathbf{dx}' \in \{-r, -r+1, ..., r, r+1\}^2$ **do** ▷ Find blocks for all offsets
      `blockId` $\leftarrow \lfloor (\mathbf{x} + \mathbf{dx})/B \rfloor$
      if $0 <=$ `blockId` $< \{bH, bW\}$ `and (empty(blocks) or blockId >` $\mathbf{x}$ `blocks[-1])` **then**
        `blocks.append(blockId)`
      end if
    end for
    `blocks.append(nBlocks)` ▷ Add end marker

    while True **do**
      `nextBlock` $\leftarrow$ inter-thread minimum of `blocks.front()` ▷ Vote for the next block to compute
      if `nextBlock >= nBlocks` **then**
        break ▷ All blocks have been processed
      end if

      `corrTile` $\in \mathbb{R}^{B^2 \times B^2} \leftarrow (\bar{F}^1[by])(\bar{F}^2[$`nextBlock`$])^T$ ▷ Matrix-matrix multiplication of the block

      if `nextBlock = blocks.front()` **then** ▷ This thread requested the current block
        `blocks.pop(0)` ▷ Advance local block pointer
        for all $\mathbf{dx} \in \{-r, -r+1, ..., r\}^2$ **do** ▷ Sample from the computed block
          $\mathbf{x}' \leftarrow \mathbf{x} + \mathbf{dx} -$ `nextBlock` $\cdot B$ ▷ Local coordinates within the block
          $\mathbf{O}[i][by \cdot B^2 + t][\mathbf{dx}] \leftarrow$ **sample** `corrTile`$[t][\mathbf{x}']$
        end for
      end if
    end while
  End Device
end for

9.1. Correlation Sampling Measurements

In Table 10, we provide full benchmarking results, corresponding to all runtime and memory usage plots. We further analyze the impact of the number of iterations, input feature dimensions, and GPU model. By default, $(512 \times 224)^2$ correlation volume, 256 feature channels, and 32 flow update iterations are used.

9.1.1. Number of Iterations

In Figure 8, we show the impact of number of flow update iterations on the runtime and memory usage. All methods show approximately linear runtime increase and constant peak memory consumption. The gap between our method and the default implementation slightly narrows with increasing number of iterations, as it has a fixed correlation volume precomputation cost, while in further steps only sampling is performed.

9.1.2. Feature Dimensionality

In Figure 9, we report the results at different feature $F^{1,2}$ dimensionality. The runtime of all methods increases approximately linearly with the feature dimension, with slower rate for our method and the default implementation.

As both our method and on-demand sampling stores down-sampled replicas of the target features for multi-level computations, they show an increase of memory with larger dimensions. However, the absolute increase is small and typically negligible.

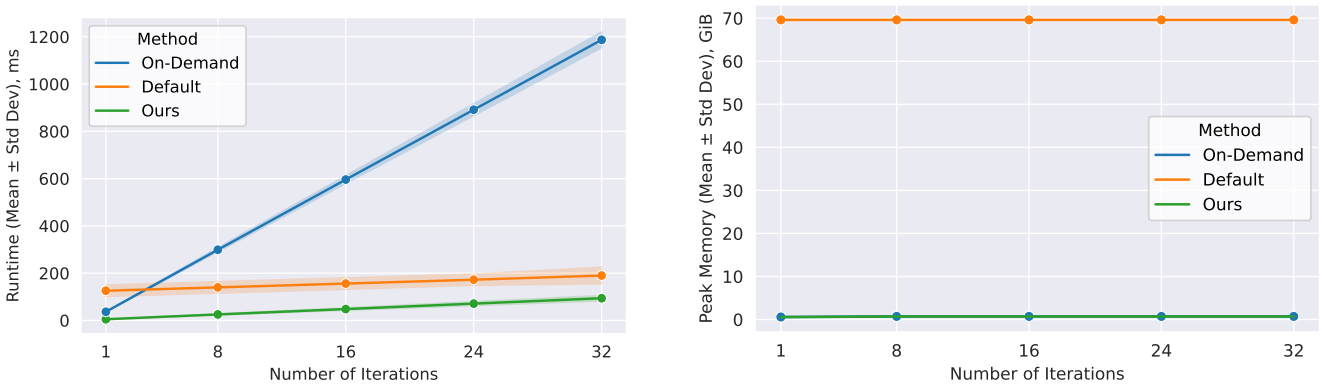


Figure 8. Runtime and peak memory consumption by the number of flow update iterations.

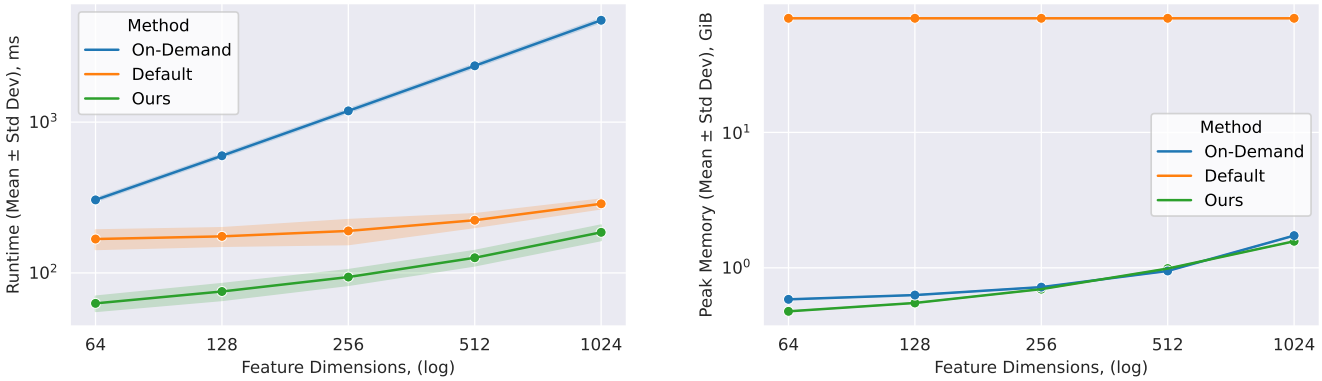


Figure 9. Runtime and peak memory consumption by input feature dimensionality.

Shot	Start Frame	End Frame
010_0050	101	196
040_0040	101	264
060_0130	101	175

Table 5. List of rendered CHARGE sequences.

9.1.3. Hardware

We perform additional experiments with different GPU models - *NVIDIA RTX 3090*, *NVIDIA RTX 4090*, and *NVIDIA A100* - and show results in Figure 10. It can be observed that hardware has little impact on the relative performance between different methods.

Note that for some GPU models, such as the *A100*, at $128px$ correlation volume width, the overhead of precomputing the full cost matrix becomes less significant, slightly outperforming our method. However, it still obtained at the expense of much higher memory requirements.

9.2. Dataset Generation Details

For the data generation, we use the CYCLES renderer to generate 335 frames from 3 sequences of the BLENDER movie CHARGE as listed in Table 5. Sample frames from each of the sequences can be seen in Figures 13-15.

To improve the quality of the rendered motion vectors, we remove lens flares and volumes, and disable motion blur to decrease noise levels. Following Mehl *et al*. [7], we render super-resolved motion vectors at $16K$ resolution with four ground truth values for each pixel. The *Python* script, used to apply scene adjustments, is provided in Listing 1.

9.3. Extended Results

In Table 9, we further analyze the improvements from our method by considering alternative strategies to reduce computational costs. First, we consider *SEA-RAFT* cascaded variants with more iterations at the lower resolution, but observe significant reduction in flow quality. Second, instead of using our correlation sampler, we run the default implementation on internal processing resolutions it is feasible on 80GB GPU, before switching to the on-demand sampler. While small improvement over on-demand sampler is achieved, it is significantly outperformed by our algorithm.

In Table 11, we provide the full results of all evaluated methods on the *Charge* $8K$ dataset. The first part of the table contains methods that do not employ correlation volume sampling and do not benefit from performance improvements using our efficient volume sampling algorithm.

In Figure 11, we visually plot the 1px error results and

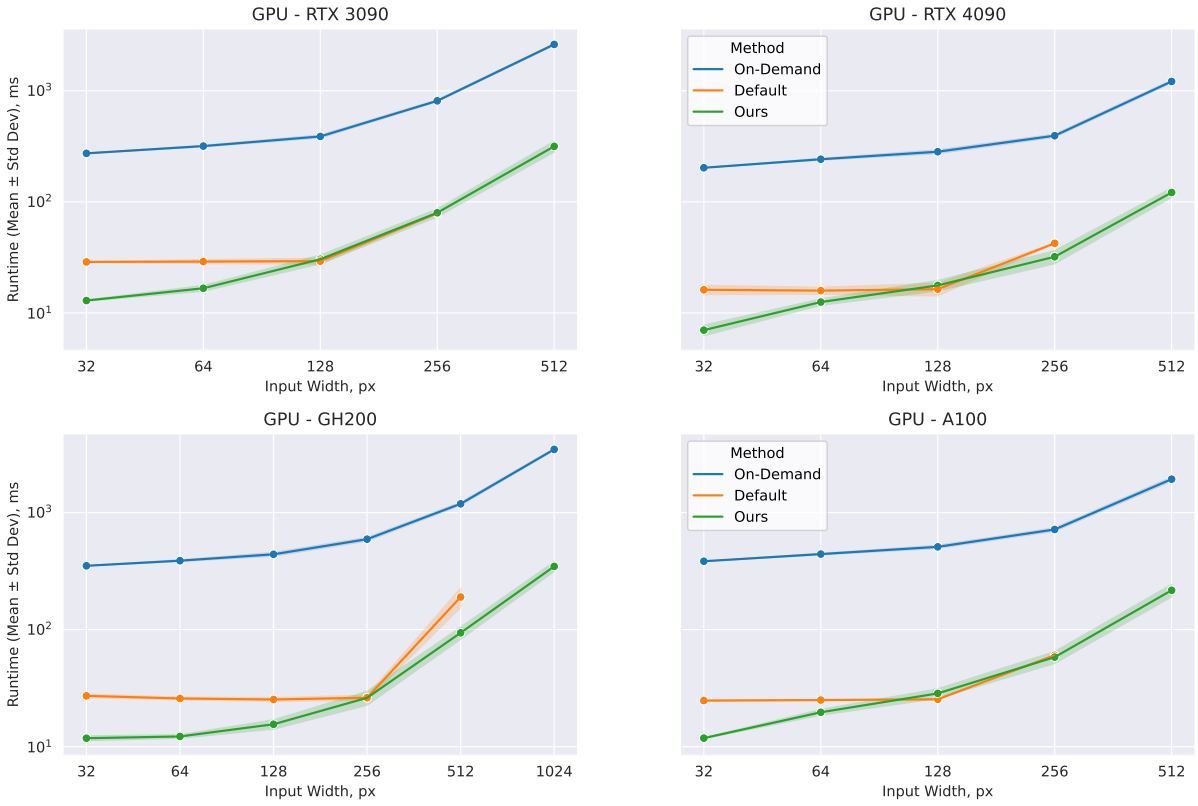


Figure 10. Runtime depending on input image size and different hardware.

runtime depending on the input scaling resolution. It can be observed that several method degrade in quality when used with the highest resolution inputs.

In Figures 13-15, we provide additional qualitative comparisons.

To obtain results of all methods, we extend PTLFlow [9], and use checkpoints fined-tuned on the Sintel dataset, unless otherwise noted in Table 11. *FlowFormer* results were obtained with their default tiling technique. We do not run *WAFT* on the full resolution due to its very slow runtime ($\approx$ 2 min / image).

9.4. Additional Datasets

We run similar evaluations on the *Kubric-8K* [10] dataset, which also contains ultra-high-resolution synthetic ground truth, but is obtained from generated scenes with moving objects. The respective results are provided in Figure 12 and Tables 6 and 7. Unlike the *Charge* dataset, where several methods benefit from fine-grained detail reconstruction at the highest resolution, on *Kubric-8K* most methods reach their lowest 1px error at quarter resolution. This could be explained from the use of less-detailed objects in the dataset. However, our algorithm still provides runtime and memory improvements across different evaluations scales (Table 7).

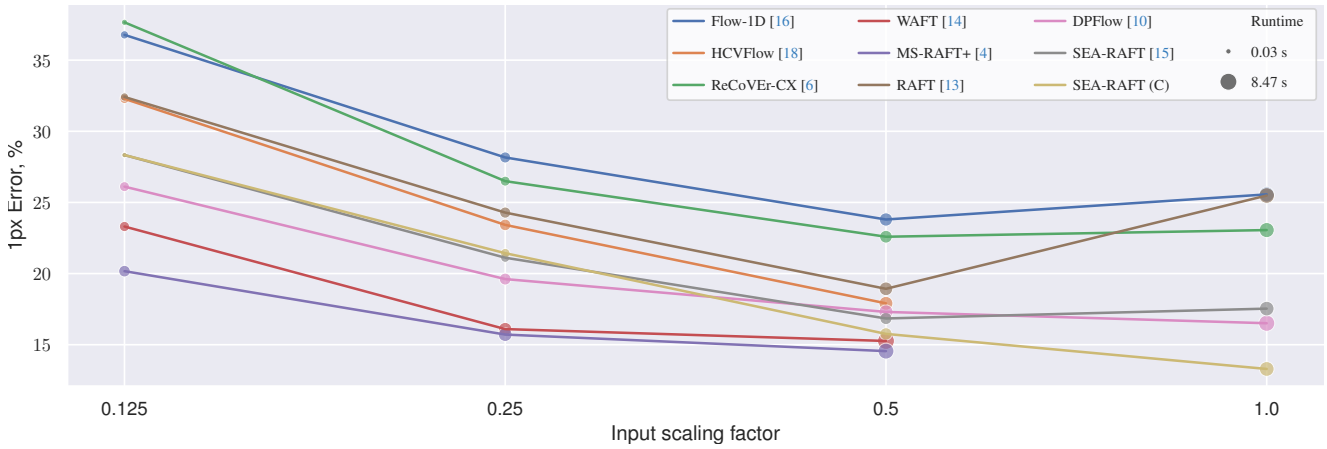
In addition to the *8K* datasets used in most experiments, we perform end-to-end runtime measurements on *Spring* [7], *Sintel* [1] and *KITTI* [8] datasets and report results in Table 8. Despite these datasets having a much lower resolution than our target, making the fast default implementation feasible, our method can still provide significant runtime or memory improvements. As the cascaded inference is not applicable on low-resolution inputs of these datasets, and we focus on performance improvements, we refer to their official benchmarks for full numerical results.

References

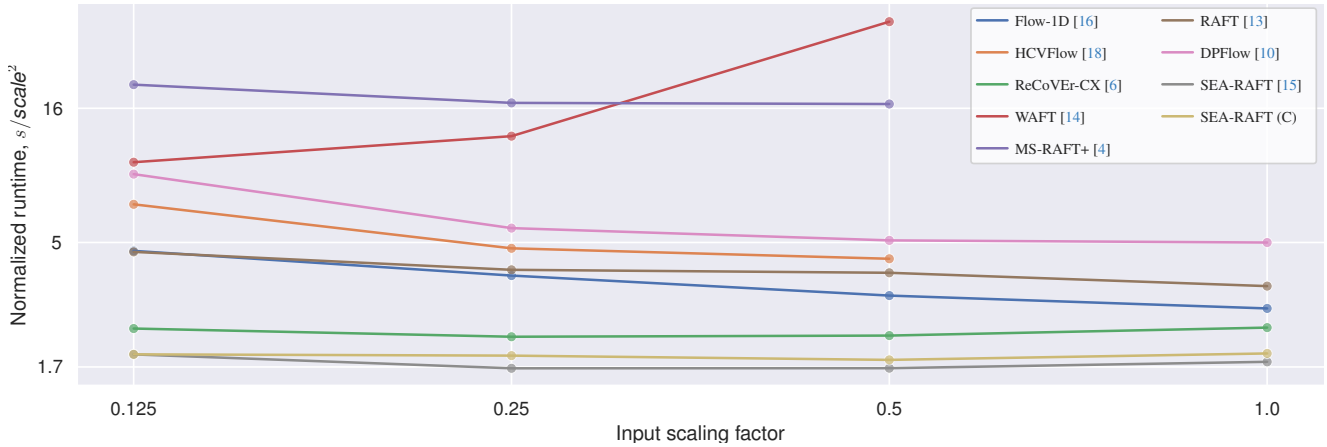
[1] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In *European Conf. on Computer Vision (ECCV)*, pages 611–625. Springer-Verlag, 2012. 1, 5

[2] Zhaoyang Huang, Xiaoyu Shi, Chao Zhang, Qiang Wang, Ka Chun Cheung, Hongwei Qin, Jifeng Dai, and Hongsheng Li. FlowFormer: A Transformer Architecture for Optical Flow. In *Computer Vision – ECCV 2022*, pages 668–685. Springer Nature Switzerland, Cham, 2022. 9, 16

[3] Azin Jahedi, Maximilian Luz, Marc Rivinius, and Andrés Bruhn. CCMR: High Resolution Optical Flow Estimation via Coarse-To-Fine Context-Guided Motion Reasoning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 6899–6908, 2024. 9, 16



(a) Comparison of 1px error.



(b) Comparison of runtime, normalized by the scaling factor.

Figure 11. Quality and runtime comparison of selected methods on the CHARGE dataset depending on the input scaling resolution.

[4] Azin Jahedi, Maximilian Luz, Marc Rivinius, Lukas Mehl, and Andrés Bruhn. MS-RAFT+: High Resolution Multi-Scale RAFT. *International Journal of Computer Vision*, 132 (5):1835–1856, 2024. 6, 8, 9, 10, 16

[5] Shihao Jiang, Yao Lu, Hongdong Li, and Richard Hartley. Learning Optical Flow From a Few Matches. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16592–16600, 2021. 9, 16

[6] Simon Kiefhaber, Stefan Roth, and Simone Schaub-Meyer. Removing cost volumes from optical flow estimators. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 79–89, 2025. 6, 8, 9, 16

[7] Lukas Mehl, Jenny Schmalfuss, Azin Jahedi, Yaroslava Nalivayko, and Andrés Bruhn. Spring: A high-resolution high-detail dataset and benchmark for scene flow, optical flow and stereo. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023. 4, 5

[8] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3061–3070, 2015. 5

[9] Henrique Morimitsu. Pytorch lightning optical flow. https://github.com/hmorimitsu/ptlflow, 2021. 5

[10] Henrique Morimitsu, Xiaobin Zhu, Roberto M. Cesar, Xiangyang Ji, and Xu-Cheng Yin. DPFlow: Adaptive Optical Flow Estimation with a Dual-Pyramid Framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17810–17820, 2025. 5, 6, 8, 9, 16

[11] Xiaoyu Shi, Zhaoyang Huang, Dasong Li, Manyuan Zhang, Ka Chun Cheung, Simon See, Hongwei Qin, Jifeng Dai, and Hongsheng Li. FlowFormer++: Masked Cost Volume Autoencoding for Pretraining Optical Flow Estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1599–1610, 2023. 9, 16

[12] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan 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*, pages 8934–8943, 2018. 9, 16

[13] Zachary Teed and Jia Deng. RAFT: Recurrent All-Pairs

Field Transforms for Optical Flow. In *Computer Vision – ECCV 2020*, pages 402–419, Cham, 2020. Springer International Publishing. 6, 8, 9, 16

[14] Yihan Wang and Jia Deng. WAFT: Warping-Alone Field Transforms for Optical Flow, 2025. 6, 8, 9, 16

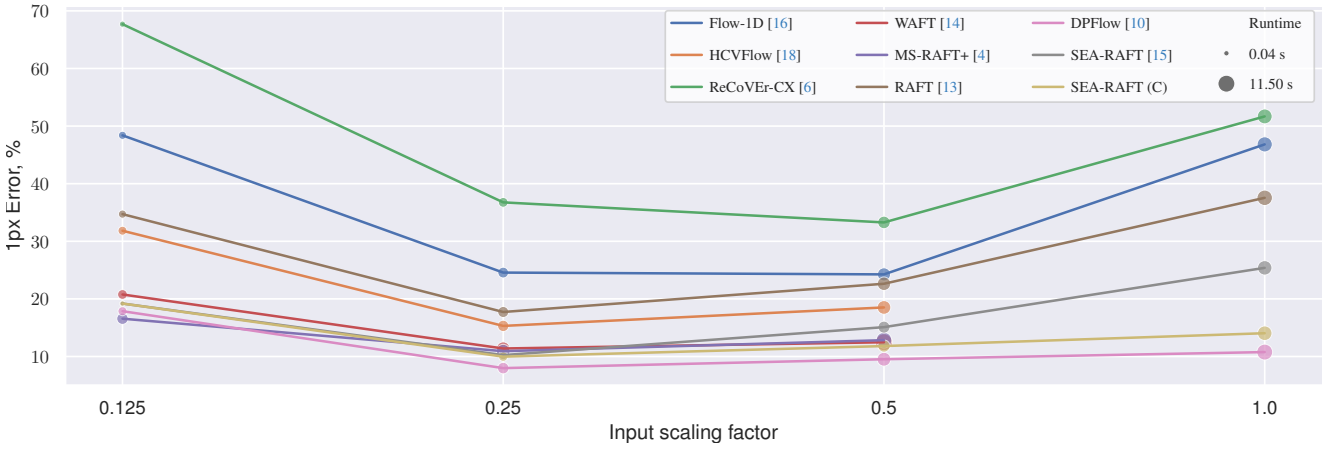
[15] Yihan Wang, Lahav Lipson, and Jia Deng. SEA-RAFT: Simple, Efficient, Accurate RAFT for Optical Flow. In *Computer Vision – ECCV 2024*, pages 36–54. Springer Nature Switzerland, Cham, 2025. 6, 8, 9, 10, 16

[16] Haofei Xu, Jiaolong Yang, Jianfei Cai, Juyong Zhang, and Xin Tong. High-Resolution Optical Flow From 1D Attention and Correlation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10498–10507, 2021. 6, 8, 9, 16

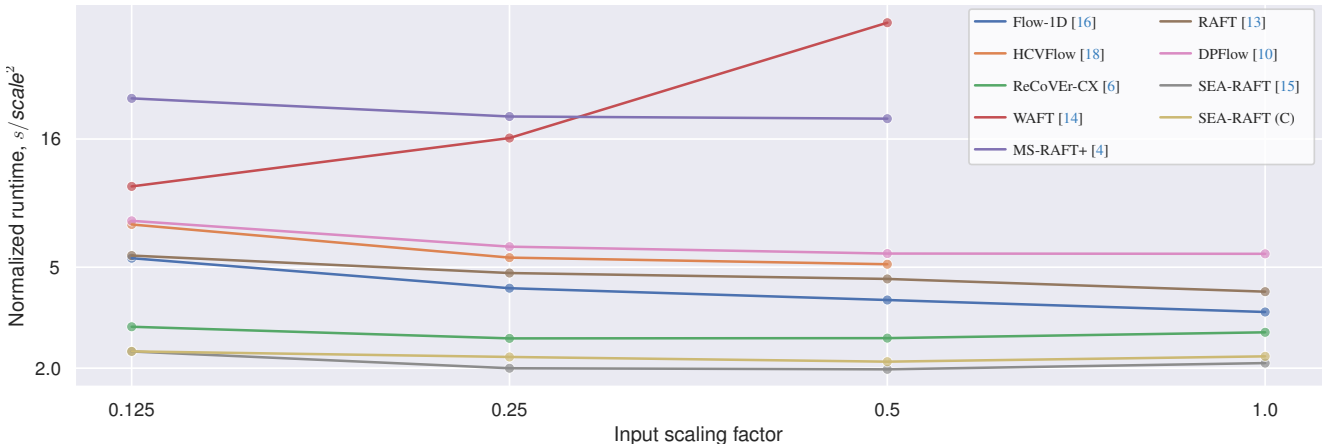
[17] Haofei Xu, Jing Zhang, Jianfei Cai, Hamid Rezatofighi, and Dacheng Tao. GMFlow: Learning Optical Flow via Global Matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8121–8130, 2022. 9, 16

[18] Yang Zhao, Gangwei Xu, and Gang Wu. Hybrid Cost Volume for Memory-Efficient Optical Flow. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 8740–8749, New York, NY, USA, 2024. Association for Computing Machinery. 6, 8, 9, 16

[19] Zihua Zheng, Ni Nie, Zhi Ling, Pengfei Xiong, Jiangyu Liu, Hao Wang, and Jiankun Li. DIP: Deep Inverse Patchmatch for High-Resolution Optical Flow. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8925–8934, 2022. 9, 16



(a) Comparison of 1px error.



(b) Comparison of runtime, normalized by the scaling factor.

Figure 12. Quality and runtime comparison of selected methods on the ***Kubric-8K*** dataset depending on the input scaling resolution.

	Best-Accuracy Input Width	1px error % ↓	EPE px ↓	LM - 1px error % ↓	LM-EPE px ↓	Best Runtime, s	Without Ours, s	Our Improvement
GMFlow [17]	2048 - 1/4	73.4	5.08	84.4	15.36	0.94		n/a
PWC-Net [12]	4096 - 1/2	19.7	21.79	44.5	98.94	0.22		n/a
Flow-1D [16]	4096 - 1/2	24.3	7.80	49.3	31.45	0.93		n/a
DIP [19]	2048 - 1/4	14.9	6.44	33.5	27.09	0.96		n/a
FlowFormer [2]	2048 - 1/4	15.6	4.10	33.7	15.92	1.79		n/a
FlowFormer++ [11]	2048 - 1/4	14.7	4.06	32.5	14.87	1.80		n/a
SCV [5]	2048 - 1/4	22.8	4.41	39.8	17.29	3.48		n/a
HCVFlow [18]	2048 - 1/4	15.3	4.34	36.2	17.61	0.34		n/a
ReCoVEr-CX [6]	4096 - 1/2	33.3	20.33	83.2	93.02	0.66		n/a
WAFT [14]	2048 - 1/4	11.4	4.81	30.8	20.17	1.01		n/a
MS-RAFT+ [4]	2048 - 1/4	10.9	4.46	22.0	18.95	1.23	1.65	−26%
RAFT [13]	2048 - 1/4	17.7	4.81	38.8	19.73	0.30	0.31	−4%
CCMR [3]	2048 - 1/4	13.5	4.79	26.2	20.07	1.29	1.60	−20%
DPFlow [10]	2048 - 1/4	**8.0**	3.16	**21.0**	**11.22**	0.38	0.42	−11%
SEA-RAFT [15]	2048 - 1/4	10.2	4.25	23.7	17.92	**0.12**	0.17	−25%
SEA-RAFT (Cascaded) 1/2	4096 - 1/2	11.8	3.95	21.3	14.75	0.53	0.87	−39%
SEA-RAFT (Cascaded)	2048 - 1/4	10.0	**3.04**	22.5	11.78	0.14	0.18	−22%

Table 6. Quantitative evaluation of optical flow estimation methods on the ***Kubric-8K*** dataset. For each method, we list the resolution that can be run with $80GB$ of GPU memory and obtains the best 1px error. For a full comparison with other methods we also list our 1/2 results. We report the 1px outlier rate, endpoint-error (EPE), both metrics for pixels with large motion (LM, magnitude over 128px) and the best runtime across all variants with and without our improvements. We highlight the best (in **bold**) and second-best (underlined) method.

		Default		**On-Demand Sampling**		**Ours**	
Method	Input Width	Runtime	Memory	Runtime	Memory	Runtime	Memory
RAFT	960 - 1/8	0.09 (−13%)	2.54 (+9%)	0.61 (+510%)	2.33 (=)	0.10	2.33
	1920 - 1/4	0.31 (+4%)	9.98 (+200%)	0.99 (+235%)	3.33 (=)	0.30	3.33
	3840 - 1/2	OOM		3.34 (+197%)	7.31 (=)	1.12	7.31
	7680 - 1/1	OOM		12.49 (+212%)	23.26 (=)	4.00	23.26
MS-RAFT+	960 - 1/8	OOM		0.52 (+43%)	3.73 (−1%)	0.36	3.76
	1920 - 1/4	OOM		1.65 (+34%)	8.77 (−1%)	1.23	8.89
	3840 - 1/2	OOM		6.46 (+34%)	28.75 (−2%)	4.81	29.25
DPFLow	960 - 1/8	0.13 (+8%)	2.61 (+13%)	0.16 (+33%)	2.31 (=)	0.12	2.31
	1920 - 1/4	0.42 (+12%)	10.48 (+244%)	0.45 (+19%)	3.05 (=)	0.38	3.05
	3840 - 1/2	OOM		1.59 (+13%)	6.12 (=)	1.41	6.12
	7680 - 1/1	OOM		6.25 (+11%)	18.45 (=)	5.65	18.45
SEA-RAFT	960 - 1/8	0.04 (+2%)	2.60 (+17%)	0.10 (+175%)	2.23 (=)	0.04	2.23
	1920 - 1/4	0.17 (+33%)	10.07 (+271%)	0.21 (+70%)	2.71 (=)	0.12	2.71
	3840 - 1/2	OOM		0.77 (+55%)	4.66 (=)	0.49	4.66
	7680 - 1/1	OOM		3.13 (+50%)	12.46 (=)	2.09	12.46

Table 7. Runtime (s) and peak memory usage (GiB) of the full optical flow method end-to-end evaluation on ***Kubric-8K*** dataset depending on the correlation computation variant at different scales of the inputs. We report the relative difference compared to our method in parentheses. OOM indicates that the method requires more than $80GB$ of memory and fails with an out-of-memory error.

Method	Dataset	Default		On-Demand Sampling		Ours	
		Runtime	Memory	Runtime	Memory	Runtime	Memory
RAFT	*KITTI 2015*	0.08 (−29%)	0.50 (+41%)	0.60 (+425%)	0.35 (=)	0.11	0.35
	Sintel	0.08 (−16%)	0.53 (+57%)	0.58 (+518%)	0.34 (=)	0.09	0.34
	Spring	0.32 (+7%)	8.22 (+423%)	0.98 (+227%)	1.57 (=)	0.30	1.57
MS-RAFT+	*KITTI 2015*	OOM		0.48 (+34%)	1.64 (−2%)	0.36	1.67
	Sintel	OOM		0.46 (+39%)	1.57 (−2%)	0.33	1.60
	Spring	OOM		1.68 (+34%)	7.01 (−2%)	1.25	7.14
DPFLow	*KITTI 2015*	0.12 (+11%)	0.57 (+63%)	0.15 (+39%)	0.35 (=)	0.11	0.35
	Sintel	0.12 (+9%)	0.53 (+56%)	0.14 (+33%)	0.34 (=)	0.11	0.34
	Spring	0.43 (+13%)	8.72 (+579%)	0.45 (+17%)	1.29 (=)	0.39	1.29
SEA-RAFT	*KITTI 2015*	0.04 (−33%)	0.55 (+115%)	0.10 (+81%)	0.26 (=)	0.05	0.26
	Sintel	0.03 (=)	0.58 (+64%)	0.09 (+184%)	0.34 (−4%)	0.03	0.35
	Spring	0.17 (+34%)	8.31 (+773%)	0.21 (+66%)	0.95 (=)	0.13	0.95

Table 8. Runtime (s) and peak memory usage (GiB) of the full optical flow end-to-end evaluation depending on the correlation computation method on three additional datasets - *KITTI 2015*, *Sintel*, and *Spring*.

		Metrics				Runtime (s) / Memory (GiB)			
		1px error	EPE	LM-1px	LM-EPE	Default	On-Demand	Mixed	Ours
MS-RAFT+ [4]	1/8	20.2	1.94	64.1	22.57	OOM	0.44 / 4.4	0.38 / 9.2	0.30 / 4.4
	1/4	15.7	**1.64**	41.1	19.85	OOM	1.40 / 8.7	1.35 / 9.9	1.04 / 8.8
	1/2	14.5	1.92	34.6	32.03	OOM	5.47 / 25.8	5.45 / 25.8	4.11 / 26.2
SEA-RAFT [15] 4LR + 0HR (default)	1/8	28.3	2.30	81.5	21.51	0.03 / 3.5	0.09 / 3.2	0.03 / 3.5	0.03 / 3.3
	1/4	21.1	2.01	54.5	19.18	0.13 / 8.9	0.19 / 3.6	0.13 / 8.9	0.10 / 3.6
	1/2	16.8	4.94	39.8	39.83	OOM	0.63 / 5.2	0.62 / 5.2	0.42 / 5.2
	1/1	17.5	17.65	49.2	107.43	OOM	2.63 / 11.8	2.63 / 11.8	1.78 / 11.8
SEA-RAFT 3LR + 1HR	1/8	31.3	2.67	84.8	27.48	0.02 / 3.5	0.04 / 3.2	0.02 / 3.5	0.02 / 3.2
	1/4	25.9	2.25	62.9	20.29	0.12 / 8.9	0.15 / 3.6	0.12 / 8.9	0.09 / 3.6
	1/2	20.7	1.93	44.3	18.41	OOM	0.44 / 5.2	0.40 / 5.2	0.35 / 5.2
	1/1	17.9	1.83	37.0	17.04	OOM	1.71 / 11.8	1.64 / 11.8	1.42 / 11.8
SEA-RAFT 6LR + 2HR	1/8	29.6	2.42	82.3	22.90	0.03 / 3.5	0.05 / 3.2	0.02 / 3.5	0.02 / 3.3
	1/4	23.1	2.03	56.1	17.82	0.13 / 8.9	0.22 / 3.6	0.13 / 8.9	0.10 / 3.6
	1/2	18.0	1.84	39.0	17.31	OOM	0.58 / 5.2	0.49 / 5.2	0.39 / 5.2
	1/1	15.0	1.82	33.2	17.05	OOM	2.19 / 11.8	2.02 / 11.8	1.58 / 11.8
SEA-RAFT 4LR + 4HR (Cascaded)	1/8	28.3	2.30	81.5	21.51	0.03 / 3.5	0.09 / 3.2	0.03 / 3.5	0.03 / 3.3
	1/4	21.4	1.86	54.1	**16.15**	0.15 / 8.9	0.25 / 3.6	0.15 / 8.9	0.12 / 3.6
	1/2	15.8	1.89	36.8	18.55	OOM	0.71 / 5.2	0.65 / 5.2	0.45 / 5.2
	1/1	**13.3**	2.70	**31.6**	21.53	OOM	2.88 / 11.8	2.77 / 11.8	1.89 / 11.8

Table 9. Analysis of alternative inference strategies on CHARGE8K dataset, with larger number of iterations at low 1/4 resolution (LR) than high resolution (HR), and mixed correlation sampler, running default implementation up to the highest feasible scale before switching to the on-demand sampler.

```
import bpy

# Choose to render RGB or Flow pass
render_flow = False

for scene in bpy.data.scenes:

    width = 1024
    aspect_ratio = width / scene.render.resolution_x
    height = int(round(scene.render.resolution_y * aspect_ratio))

    scale = 16 if render_flow else 8
    scene.render.resolution_x = width * scale
    scene.render.resolution_y = height * scale

    scene.render.use_motion_blur = False
    scene.render.engine = "CYCLES"
    scene.cycles.device = "CPU"

    n_samples = 1 if render_flow else 1024
    scene.cycles.samples = n_samples
    scene.cycles.adaptive_min_samples = n_samples > 1
    scene.cycles.adaptive_threshold = 0.01
    scene.cycles.use_adaptive_sampling = True
    scene.cycles.denoiser = "OPENIMAGEDENOISE"
    scene.cycles.use_denoising = (not render_flow)

    # Removing very rare but strong fireflies
    scene.cycles.sample_clamp_direct = 50.0

    # Removing lens flares and overlays
    scene.use_nodes = False

    collection = bpy.data.collections.get("flares")
    if collection:
        bpy.data.collections.remove(collection)

    for view_layer in scene.view_layers:
        view_layer.use_pass_combined = True


# Removing volumes
for mat in bpy.data.materials:
    if mat.use_nodes:
        for node in mat.node_tree.nodes:
            if node.bl_static_type == "OUTPUT_MATERIAL" and node.is_active_output:
                for link in node.inputs["Volume"].links:
                    mat.node_tree.links.remove(link)
```

Listing 1. Blender scene setup script.

Figure 13. Qualitative comparison across different evaluation scales on a frame from `010_0050` shot. Image from Charge by Blender Studio.

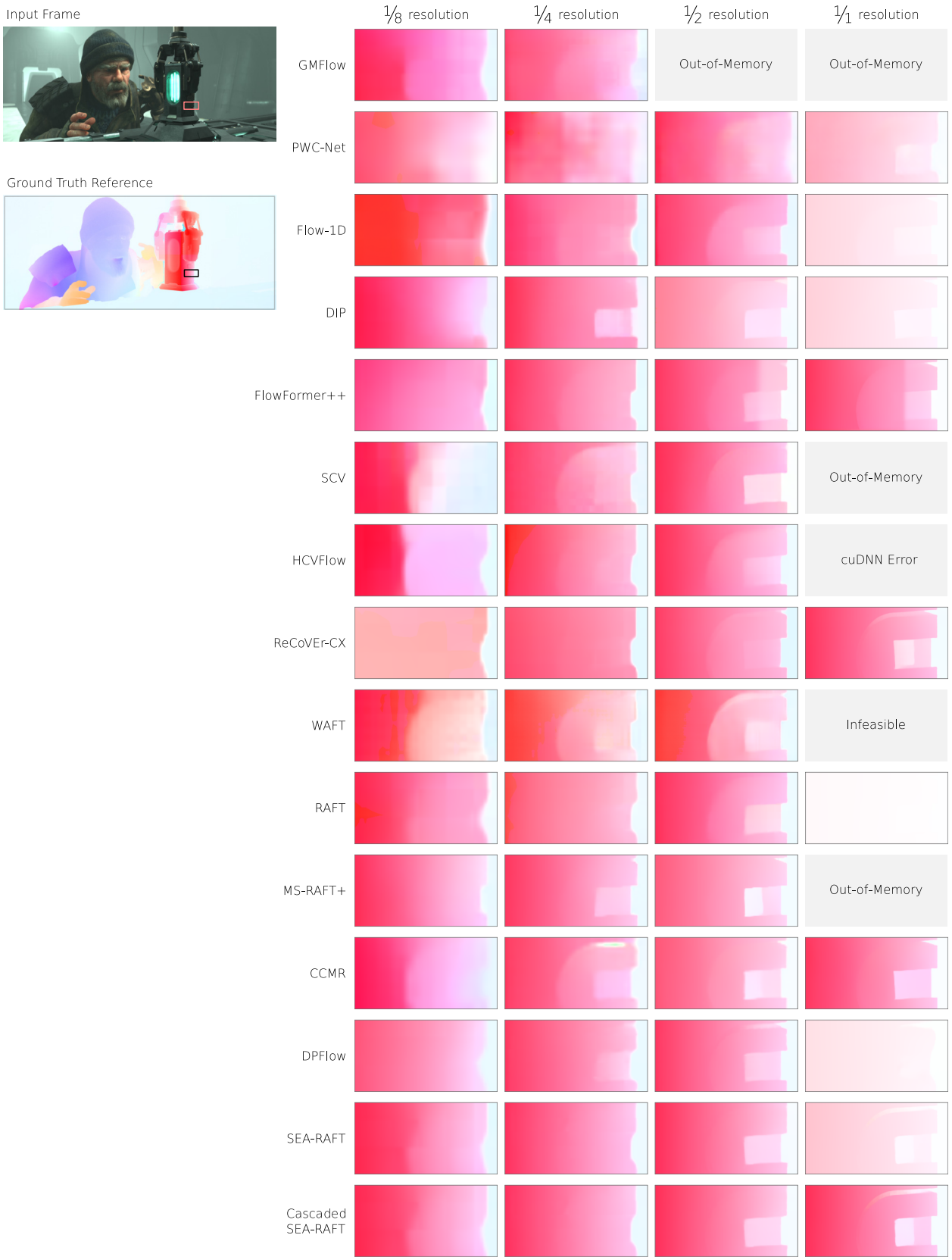


Figure 14. Qualitative comparison across different evaluation scales on a frame from `040_0040` shot. Image from Charge by Blender Studio.

Figure 15. Qualitative comparison across different evaluation scales on a frame from `060_0130` shot. Image from Charge by Blender Studio.

Variable	Value	Runtime (mean ± std), ms				Peak Memory (mean ± std), MB			
		Default	On-Demand	Ours	Improvement	Default	On-Demand	Ours	Improvement
Input Width GH200	32	27 ± 1	351 ± 0	12 ± 1	96.6%	4 ± 0	3 ± 0	3 ± 0	13.3%
	64	26 ± 1	388 ± 6	12 ± 0	96.8%	27 ± 1	12 ± 0	12 ± 0	55.3%
	128	25 ± 1	440 ± 15	16 ± 1	96.5%	291 ± 0	47 ± 0	46 ± 0	84.3%
	256	26 ± 1	592 ± 20	26 ± 4	95.6%	4091 ± 0	184 ± 0	178 ± 0	95.6%
	512	190 ± 35	1187 ± 33	94 ± 11	92.1%	71288 ± 0	737 ± 0	712 ± 0	99.0%
	1024	OOM	3462 ± 62	347 ± 32	90.0%	OOM	2944 ± 0	2829 ± 0	–
	2048	OOM	13522 ± 288	1334 ± 133	90.1%	OOM	11764 ± 0	11274 ± 0	–
Input Width A100	32	25 ± 0	384 ± 0	12 ± 0	96.9%	4 ± 0	3 ± 0	3 ± 0	13.3%
	64	25 ± 0	442 ± 6	20 ± 1	95.5%	27 ± 0	12 ± 0	12 ± 0	55.3%
	128	25 ± 0	509 ± 15	29 ± 3	94.4%	291 ± 0	47 ± 0	46 ± 0	84.3%
	256	60 ± 0	717 ± 23	58 ± 7	91.9%	4090 ± 0	184 ± 0	178 ± 0	95.6%
	512	OOM	1928 ± 73	217 ± 26	88.8%	OOM	737 ± 0	712 ± 0	–
Input Width RTX 4090	32	16 ± 2	203 ± 0	7 ± 1	96.5%	4 ± 0	3 ± 0	3 ± 0	13.3%
	64	16 ± 1	242 ± 4	13 ± 1	94.8%	27 ± 0	12 ± 0	12 ± 0	55.3%
	128	16 ± 2	283 ± 11	18 ± 2	93.7%	291 ± 0	47 ± 0	46 ± 0	84.3%
	256	42 ± 0	394 ± 12	32 ± 4	91.9%	4090 ± 0	184 ± 0	178 ± 0	95.6%
	512	OOM	1213 ± 34	121 ± 12	90.0%	OOM	737 ± 0	712 ± 0	–
Input Width RTX 3090	32	29 ± 0	273 ± 1	13 ± 0	95.3%	4 ± 0	3 ± 0	3 ± 0	13.3%
	64	29 ± 1	318 ± 5	17 ± 1	94.8%	27 ± 0	12 ± 0	12 ± 0	55.3%
	128	29 ± 2	388 ± 11	30 ± 3	92.2%	291 ± 0	47 ± 0	46 ± 0	84.3%
	256	79 ± 0	812 ± 15	80 ± 6	90.2%	4090 ± 0	184 ± 0	178 ± 0	95.6%
	512	OOM	2615 ± 49	316 ± 35	87.9%	OOM	737 ± 0	712 ± 0	–
Number of Iterations GH200	1	126 ± 24	37 ± 0	5 ± 0	86.3%	71288 ± 0	599 ± 0	574 ± 0	99.2%
	8	140 ± 24	299 ± 8	25 ± 2	91.5%	71288 ± 0	737 ± 0	712 ± 0	99.0%
	16	156 ± 24	596 ± 16	48 ± 5	91.9%	71288 ± 0	737 ± 0	712 ± 0	99.0%
	24	172 ± 24	891 ± 25	71 ± 8	92.0%	71288 ± 0	737 ± 0	712 ± 0	99.0%
	32	190 ± 35	1187 ± 33	94 ± 11	92.1%	71288 ± 0	737 ± 0	712 ± 0	99.0%
Feature Dimensionality GH200	64	168 ± 25	305 ± 8	63 ± 7	79.4%	71288 ± 0	599 ± 0	489 ± 0	99.3%
	128	175 ± 24	598 ± 16	75 ± 10	87.4%	71288 ± 0	644 ± 0	563 ± 0	99.2%
	256	190 ± 35	1187 ± 33	94 ± 11	92.1%	71288 ± 0	737 ± 0	712 ± 0	99.0%
	512	224 ± 23	2362 ± 68	126 ± 14	94.7%	71288 ± 0	971 ± 0	1011 ± 0	98.6%
	1024	287 ± 21	4734 ± 145	186 ± 22	96.1%	71288 ± 0	1771 ± 0	1608 ± 0	97.7%

Table 10. Full correlation volume sampling isolated benchmarking results depending on one variable. We report the runtime and peak memory usage of each implementation, as well as memory improvement over default implementation and runtime improvement over the on-demand sampling method. OOM indicates that the method fails with an out-of-memory error.

		Metrics				Runtime		
		1px error	EPE	LM-1px	LM-EPE	Default	On-Demand	Ours
GMFlow [17]	1/8	43.9	2.95	94.3	24.46	0.09 ± 0.00	n/a	
	1/4	47.0	2.74	86.3	28.61	0.71 ± 0.00	n/a	
PWC-Net [12]	1/8	42.8	3.89	93.6	47.94	0.07 ± 0.04	n/a	
	1/4	30.3	3.32	78.6	47.27	0.10 ± 0.05	n/a	
	1/2	24.4	3.26	61.4	57.14	0.21 ± 0.07	n/a	
	1/1	26.0	6.65	76.1	128.30	0.64 ± 0.10	n/a	
Flow-1D [16] Highres	1/8	36.8	3.14	95.2	37.06	0.07 ± 0.01	n/a	
	1/4	28.2	2.39	83.7	29.80	0.23 ± 0.05	n/a	
	1/2	23.8	2.23	71.3	31.58	0.79 ± 0.02	n/a	
	1/1	25.6	5.86	69.2	57.55	2.82 ± 0.01	n/a	
DIP [19]	1/8	28.3	2.44	84.4	28.92	0.28 ± 0.01	n/a	
	1/4	22.5	2.23	60.6	33.65	0.84 ± 0.00	n/a	
	1/2	20.8	2.76	46.0	56.97	2.93 ± 0.01	n/a	
	1/1	19.8	4.00	51.2	85.62	11.57 ± 0.06	n/a	
FlowFormer [2]	1/8	33.2	2.62	89.0	26.13	0.79 ± 0.00	n/a	
	1/4	25.0	2.19	68.0	22.20	1.81 ± 0.01	n/a	
	1/2	19.2	2.31	52.1	33.86	5.06 ± 0.03	n/a	
	1/1	16.9	3.22	56.4	58.16	16.57 ± 0.06	n/a	
FlowFormer++ [11]	1/8	32.0	2.53	88.6	22.44	0.80 ± 0.00	n/a	
	1/4	24.1	2.00	66.5	20.04	1.82 ± 0.01	n/a	
	1/2	18.9	2.46	51.9	32.26	5.09 ± 0.03	n/a	
	1/1	16.8	3.41	55.3	55.70	16.68 ± 0.05	n/a	
SCV [5]	1/8	35.3	2.60	85.8	29.57	0.94 ± 0.08	n/a	
	1/4	23.7	2.01	61.0	23.19	3.08 ± 0.21	n/a	
	1/2	19.1	2.83	46.1	46.42	14.66 ± 0.88	n/a	
HCVFlow [18]	1/8	32.3	2.61	91.0	28.68	0.11 ± 0.00	n/a	
	1/4	23.4	1.99	70.9	25.37	0.30 ± 0.03	n/a	
	1/2	17.9	2.08	54.4	32.92	1.09 ± 0.01	n/a	
ReCoVEr-CX [6] Mixed	1/8	37.7	3.20	97.3	43.77	0.04 ± 0.00	n/a	
	1/4	26.5	2.67	95.1	40.80	0.14 ± 0.00	n/a	
	1/2	22.6	3.01	94.6	58.71	0.56 ± 0.00	n/a	
	1/1	23.1	4.98	99.6	114.81	2.39 ± 0.01	n/a	
WAFT [14]	1/8	23.3	2.05	83.1	21.35	0.16 ± 0.03	n/a	
	1/4	16.1	1.93	62.0	27.44	0.79 ± 0.01	n/a	
	1/2	15.3	2.74	62.6	65.14	8.47 ± 0.07	n/a	
MS-RAFT+ [4]	1/8	20.2	1.94	64.1	22.59	OOM	0.44 ± 0.00	0.31 ± 0.00
	1/4	15.7	1.64	41.1	19.85	OOM	1.41 ± 0.01	1.05 ± 0.00
	1/2	14.5	1.92	34.6	32.03	OOM	5.51 ± 0.02	4.15 ± 0.02
RAFT [13]	1/8	32.4	2.63	90.3	26.73	0.07 ± 0.00	0.56 ± 0.01	0.09 ± 0.00
	1/4	24.3	2.01	69.8	22.16	0.25 ± 0.00	0.89 ± 0.01	0.25 ± 0.00
	1/2	18.9	5.90	49.8	36.41	OOM	2.62 ± 0.03	0.96 ± 0.01
	1/1	25.5	59.94	71.3	266.20	OOM	10.47 ± 0.13	3.43 ± 0.04
CCMR [3]	1/8	25.9	2.14	76.1	24.40	0.34 ± 0.02	0.47 ± 0.02	0.33 ± 0.01
	1/4	17.5	1.88	50.5	27.71	OOM	1.35 ± 0.04	1.09 ± 0.02
	1/2	15.8	2.16	39.7	42.86	OOM	5.23 ± 0.02	4.39 ± 0.01
	1/1	18.0	4.08	45.7	76.96	OOM	21.09 ± 0.15	17.58 ± 0.02
DPFlow [10]	1/8	26.1	2.00	82.4	18.88	0.14 ± 0.01	0.15 ± 0.01	0.14 ± 0.02
	1/4	19.6	**1.60**	57.8	**13.83**	0.40 ± 0.01	0.41 ± 0.01	0.35 ± 0.01
	1/2	17.3	1.71	40.3	14.29	OOM	1.41 ± 0.00	1.27 ± 0.00
	1/1	16.5	1.92	34.4	18.03	OOM	5.48 ± 0.02	5.00 ± 0.02
SEA-RAFT [15]	1/8	28.3	2.30	81.5	21.51	0.03 ± 0.00	0.09 ± 0.00	0.03 ± 0.00
	1/4	21.1	2.01	54.5	19.18	0.13 ± 0.00	0.18 ± 0.00	0.11 ± 0.00
	1/2	16.8	4.94	39.8	39.83	OOM	0.62 ± 0.00	0.42 ± 0.00
	1/1	17.5	17.65	49.2	107.43	OOM	2.63 ± 0.02	1.78 ± 0.01
SEA-RAFT (Cascaded)	1/8	28.3	2.30	81.5	21.51	0.03 ± 0.00	0.09 ± 0.00	0.03 ± 0.00
	1/4	21.4	1.86	54.1	16.15	0.15 ± 0.00	0.25 ± 0.00	0.12 ± 0.00
	1/2	15.8	1.90	36.8	18.58	OOM	0.72 ± 0.00	0.45 ± 0.00
	1/1	**13.3**	2.70	**31.6**	21.53	OOM	2.89 ± 0.02	1.91 ± 0.01

Table 11. Full qualitative evaluation of optical flow estimation methods on the CHARGE-8K dataset. We split the results depending on the evaluation scale and report endpoint-error (EPE), 1px outlier rate, as well as metrics for pixels with large motion. Best result of each metric is highlighted in **bold**, best performing scale for each method is underlined. OOM indicates that the method fails with an out-of-memory error, while n/a - not applicable. We also underline the scale of the row corresponding to the results in Table 1 in the main paper.

```python
from dataclasses import dataclass
import cutlass
from cutlass import cute
from cutlass.cute.nvgpu import warp

@dataclass
class Gemm:
    """Helper class to perform tiled GEMM"""
    thr_copy: cute.TiledCopy
    rMat1: cute.Tensor
    rMat2: cute.Tensor
    rC: cute.Tensor
    tCrC: cute.Tensor
    gmem_copy: cute.TiledCopy
    smem_copy1: cute.TiledCopy
    smem_copy2: cute.TiledCopy
    tM1gM1: cute.Tensor
    tM1sM1: cute.Tensor
    tM2sM2: cute.Tensor
    mma: cute.TiledMma
    tSsMat1: cute.Tensor
    tSrMat1: cute.Tensor
    tSsMat2: cute.Tensor
    tSrMat2: cute.Tensor

    @classmethod
    def make(cls, gMat1: cute.Tensor, sMat1: cute.Tensor,
             sMat2: cute.Tensor, sCorr: cute.Tensor,
             gmem_copy: cute.TiledCopy,
             mma: cute.TiledMma):
        tidx, _, _ = cute.arch.thread_idx()
        thr_copy = gmem_copy.get_slice(tidx)
        tM1gM1 = thr_copy.partition_S(gMat1)
        tM1sM1 = thr_copy.partition_D(sMat1)
        tM2sM2 = thr_copy.partition_D(sMat2)
        thr_mma = mma.get_slice(tidx)
        rMat1 = thr_mma.make_fragment_A(thr_mma.partition_A(sMat1))
        rMat2 = thr_mma.make_fragment_B(thr_mma.partition_B(sMat2))
        rC = cute.make_fragment(thr_mma.partition_shape_C(sCorr.shape), sCorr.element_type)
        tCrC = thr_mma.partition_C(sCorr)
        smem_atom = cute.make_copy_atom(warp.LdMatrix8x8x16bOp(num_matrices=4), gMat1.element_type)
        smem_copy1 = cute.make_tiled_copy_A(smem_atom, mma)
        smem_copy2 = cute.make_tiled_copy_B(smem_atom, mma)
        s1, s2 = smem_copy1.get_slice(tidx), smem_copy2.get_slice(tidx)
        tSsMat1, tSrMat1 = s1.partition_S(sMat1), s1.retile(rMat1)
        tSsMat2, tSrMat2 = s2.partition_S(sMat2), s2.retile(rMat2)
        return cls(thr_copy, rMat1, rMat2, rC, tCrC, gmem_copy, smem_copy1, smem_copy2,
                   tM1gM1, tM1sM1, tM2sM2, mma, tSsMat1, tSrMat1, tSsMat2, tSrMat2)

    def compute_tile(self, gMat2: cute.Tensor, n_steps: int):
        tM1gM2 = self.thr_copy.partition_S(gMat2)

        self.rC.fill(0.0)

        for ko in range(n_steps):
            cute.copy(self.gmem_copy, self.tM1gM1[None, None, ko], self.tM1sM1[None, None, 0])
            cute.copy(self.gmem_copy, tM1gM2[None, None, ko], self.tM2sM2[None, None, 0])
            cute.arch.cp_async_commit_group()
            cute.arch.cp_async_wait_group(0)
            cute.arch.sync_threads()

            cute.copy(self.smem_copy1, self.tSsMat1, self.tSrMat1)
            cute.copy(self.smem_copy2, self.tSsMat2, self.tSrMat2)

            cute.gemm(self.mma, self.rC, self.rMat1, self.rMat2, self.rC)

            cute.arch.sync_threads()

        cute.autovec_copy(self.rC, self.tCrC)
        cute.arch.sync_threads()
```

Listing 2. CuTe DSL GEMM helper class.

```python
def _floor(v: cutlass.Numeric):
    return cute.Int32(cutlass._mlir.dialects.math.floor(v.ir_value()))


def _atomicMin(p: cutlass.Pointer, v: cutlass.Int32):
    return cutlass._mlir.dialects.nvvm.atomicrmw(
        cutlass.cutlass_dsl.T.i32(),
        cutlass._mlir.dialects.nvvm.AtomicOpKind.MIN,
        p.llvm_ptr, v.ir_value(),
    )


def _map_index(i: int, h: cutlass.Constexpr, bh: cutlass.Constexpr, w: cutlass.Constexpr, bw: cutlass.Constexpr):
    bw_i = i % bw
    bh_i = (i // bw) % bh
    w_i = (i // (bw * bh)) % w
    h_i = i // (bw * bh * w)
    return h_i * bh + bh_i, w_i * bw + bw_i


@cute.jit
def compute_coefficients(coords, bi, by, tidx, scale, h_block, cs, r, dtype):
    ch, cw = coords.shape[2:]
    cy, cx = _map_index(by * h_block + tidx, cute.ceil_div(ch, cs), cs, cute.ceil_div(cw, cs), cs)
    is_valid_row = (cy < ch) & (cx < cw)
    x_floor, y_floor = cute.Int32(0), cute.Int32(0)
    nw, ne, sw, se = dtype(0.0), dtype(0.0), dtype(0.0), dtype(0.0)
    if is_valid_row:
        x_start = coords[bi, 0, cy, cx] * scale - r // 2
        y_start = coords[bi, 1, cy, cx] * scale - r // 2
        x_floor, y_floor = _floor(x_start), _floor(y_start)
        xc, yc = x_start - x_floor, y_start - y_floor
        nw, ne = (1 - xc) * (1 - yc), xc * (1 - yc)
        sw, se = (1 - xc) * yc, xc * yc
    return (cy, cx), is_valid_row, (x_floor, y_floor), (nw, ne, sw, se)


@cute.jit
def sample_block(sCorr, tidx, rCorr, mat2x, mat2y,
                 x_floor, y_floor, nw, ne, sw, se,
                 output, bi, cy, cx, cs, r, dtype):
    cute.autovec_copy(sCorr[tidx, None], rCorr)

    # Only iterate over parts that are in the current tile
    rx_start = max(mat2x * cs - x_floor - 1, 0)
    rx_end = min(rx_start + cs + 1, r)
    ry_start = max(mat2y * cs - y_floor - 1, 0)
    ry_end = min(ry_start + cs + 1, r)

    for ry in range(ry_start, ry_end):
        y = y_floor - mat2y * cs + ry
        for rx in range(rx_start, rx_end):
            x = x_floor - mat2x * cs + rx
            value = dtype(0.0)
            if (y >= 0) & (y < cs):
                if (x >= 0) & (x < cs):
                    value += rCorr[y * cs + x] * nw
                if (x + 1 >= 0) & (x + 1 < cs):
                    value += rCorr[y * cs + x + 1] * ne
            if (y + 1 >= 0) & (y + 1 < cs):
                if (x >= 0) & (x < cs):
                    value += rCorr[(y + 1) * cs + x] * sw
                if (x + 1 >= 0) & (x + 1 < cs):
                    value += rCorr[(y + 1) * cs + x + 1] * se
            output[bi, ry + rx * r, cy, cx] += value
```

Listing 3. Kernel helper functions.

```python
@cute.kernel
def kernel(
    coords: cute.Tensor,  # [B 2 H W]
    mat1: cute.Tensor,  # patch-major [B (h w bh bw) C]
    mat2: cute.Tensor,  # patch-major [B h w (bh bw) C]
    output: cute.Tensor,  # output [B (2r+1)^2 H H]
    sMat1_layout: cute.ComposedLayout,  # with Swizzle<2,3,3> o 0 o (8,32):(32,1)
    sMat2_layout: cute.ComposedLayout,  # with Swizzle<2,3,3> o 0 o (8,32):(32,1)
    corr_layout: cute.Layout,  # [bh, bw]
    gmem_tiled_copy_mat: cute.TiledCopy,  # with cute.nvgpu.cpasync.CopyG2SOp
    tiled_mma: cute.TiledMma,  # MmaF16BF16Op
    coord_scaler: cutlass.Numeric,  # float, scaler for coords on lower resolutions
    cell_size: cutlass.Constexpr, radius: cutlass.Constexpr, k_block: cutlass.Constexpr,
):
    (by, bi, _), (tidx, _, _) = cute.arch.block_idx(), cute.arch.thread_idx()
    h_block, w_block = corr_layout.shape[:2]
    dtype = output.element_type
    _, targetH, targetW, _, channels = mat2.shape

    smem = cutlass.utils.SmemAllocator()
    next_block = smem.allocate_tensor(cutlass.Int32, cute.make_layout((1,)))
    sMat1 = smem.allocate_tensor(mat1.element_type, sMat1_layout, byte_alignment=16)
    sMat2 = smem.allocate_tensor(mat1.element_type, sMat2_layout, byte_alignment=16)
    sCorr = cute.make_tensor(cute.recast_ptr(sMat1.iterator, dtype=dtype), corr_layout)

    gMat1 = cute.local_tile(mat1[bi, None, None], (h_block, channels), (by, 0))
    gMat1 = cute.make_tensor(gMat1.iterator.align(16), gMat1.layout)
    gemm = Gemm.make(gMat1, sMat1, sMat2, sCorr, gmem_tiled_copy_mat, tiled_mma)
    rCorr = cute.make_fragment(cute.make_layout((w_block,)), dtype)
    cute.arch.sync_threads()

    (cy, cx), is_valid_row, (x_floor, y_floor), (nw, ne, sw, se) = compute_coefficients(
        coords, bi, by, tidx, coord_scaler, h_block, cell_size, radius, dtype)

    # Find the blocks this thread requires and prepare the vote
    rp1 = radius + 1
    nr = rp1 // cell_size + 2  # Maximum number of blocks this thread can request
    blocks = cute.make_fragment(cute.make_layout((nr * nr + 1,)), cutlass.Int32)
    max_block = targetH * targetW
    blocks.fill(max_block)
    block_i = 0
    if is_valid_row:
        for i in range(nr * nr):
            rx, ry = i % nr, (i // nr) % nr
            x = (x_floor + min(rx * cell_size, rp1 - 1)) // cell_size
            y = (y_floor + min(ry * cell_size, rp1 - 1)) // cell_size
            if (x >= 0) & (x < targetW) & (y >= 0) & (y < targetH):
                val = y * targetW + x
                if block_i == 0 or blocks[block_i - 1] < val:
                    blocks[block_i] = val
                    block_i += 1
    if tidx == 0:
        next_block[0] = max_block
    cute.arch.sync_threads()

    block_i = 0
    next_local = 0
    while next_local < max_block:
        # Vote for the next block
        if is_valid_row and blocks[block_i] < max_block:
            _atomicMin(next_block.iterator, blocks[block_i])
        cute.arch.sync_threads()
        next_local = next_block[0]

        if next_local < max_block:
            mat2y, mat2x = next_local // targetW, next_local % targetW
            gMat2 = cute.local_tile(mat2[bi, mat2y, mat2x, None, None], (w_block, channels), (0, 0))
            gMat2 = cute.make_tensor(gMat2.iterator.align(16), gMat2.layout)
            cute.arch.sync_threads()
            if tidx == 0:  # Resetting the vote
                next_block[0] = max_block

            gemm.compute_tile(gMat2, n_steps=(channels + k_block - 1) // k_block)

            if next_local == blocks[block_i]:  # If this thread requested the block, sample it and advance to the next
                sample_block(sCorr, tidx, rCorr, mat2x, mat2y, x_floor, y_floor,
                    nw, ne, sw, se, output, bi, cy, cx, cell_size, radius, dtype)
                block_i += 1
```

Listing 4. Kernel implementation using CuTe DSL, reformatted for space constraints.