A Real-Time Stitching Method for Dual-View Street-View Images Based on a TensorRT Hybrid Inference Architecture

Wengui Tian
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: 2983443454@qq.com

Haoyu Zhao
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: 2922135660@qq.com

Yawen Wang
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: wangyawen@xatu.edu.cn

Jiayu Bai
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: 1959825177@qq.com

***Abstract*—Deep image stitching networks deliver strong alignment quality, yet their inference latency keeps them below the frame rate that dual-view street-view applications demand. This paper reports a segmented hybrid inference architecture built on TensorRT. The partition criterion is the numerical sensitivity of each sub-graph: compute-bound but numerically stable modules are handed to TensorRT, while sensitive modules stay in PyTorch. Using the UDIS2 unsupervised deep image stitching algorithm as its base, the architecture re-cuts the two-stage pipeline of the Warp geometric alignment network and the Composition fusion network into a compute-bound segment and a sensitivity-critical segment that run on different backends. The ResNet-50 backbone and the correlation computation module are exported as a TensorRT FP16 engine, which lets the half-precision units of the GPU Tensor Cores carry the bulk of the arithmetic. Direct linear transformation solving, homography warping and thin-plate spline interpolation remain in the PyTorch FP32 environment, so that matrix inversion never runs under a low-precision floating-point representation. The Composition fusion network is deployed as a separate TensorRT FP16 engine. On an NVIDIA RTX 4060 Laptop GPU the end-to-end latency falls from 213.9 ms for the original PyTorch pipeline to 39.7 ms, giving a frame rate of 25.2 FPS and a speedup of about 5.4. Fidelity at the Composition output reaches 51.3 dB peak signal-to-noise ratio, and a pixel-level audit of the final panorama places 99.88 percent of the canvas within one grey level of the gold-standard render. Speed and image quality are therefore obtained together rather than traded against each other.**

Keywords-Deep Learning; Image Stitching; Tensorrt Acceleration; Mixed-Precision Inference; UDIS2; Real-Time Vision System

I. INTRODUCTION

Large-scale collection and immediate processing of street-view data sit at the base of smart-city and autonomous-driving systems. Security surveillance, environmental perception and urban three-dimensional reconstruction all depend on several cameras working together, their video streams being merged into one wide-field panorama that extends what a single camera can see [1], [2]. The traditional route relies on hand-crafted descriptors such as SIFT, SURF and ORB for keypoint matching and homography estimation [3]. Robustness falls away under strong illumination change, motion blur and weak texture, and the planar assumption behind a single homography leaves the method poorly suited to non-planar scenes with pronounced parallax [4].

Deep learning reopened the problem. The UDIS family proposed by Nie et al. [5] trains without ground-truth annotation and cascades a Warp network with a Composition network, the first handling geometric alignment and the second appearance fusion; on wide-baseline, high-parallax material the resulting quality is far above that of

feature-based pipelines. UDIS2 [6] builds on that design and couples direct linear transformation (DLT) regression with a thin-plate spline (TPS) transformation, so that global projective motion and local non-rigid deformation are modelled at once. It reports leading scores on several public benchmark datasets.

Deployment cost is what keeps UDIS2 out of real-time systems. Inside the Warp network sit a dual-branch ResNet-50 backbone, a correlation computation layer (CCL), a DLT regression module and two geometric transformations, namely homography warping and TPS warping. On an NVIDIA RTX 4060 Laptop GPU a single frame takes about 167 ms. The Composition network is lighter, being a U-Net mask predictor, but the combined end-to-end latency still passes 200 ms and misses the 25 FPS real-time bar.

NVIDIA TensorRT [7] is a mature industrial inference optimizer that shortens latency several-fold through operator fusion, memory reuse and FP16 or INT8 quantization. Exporting the whole UDIS2 computational graph as one TensorRT engine nevertheless runs into three obstacles. The DLT step of the Warp network performs a matrix decomposition, which is fragile under a low-precision floating-point representation. The 172×172 inversion inside the TPS transformation is served by the Newton-Schulz approximation that TensorRT substitutes, and the error it accumulates cannot be bounded. Finally, the Composition network reacts sharply to whatever the Warp stage hands it, so a small numerical drift upstream is amplified into visible damage downstream.

The architecture proposed here answers those three obstacles. Backend allocation follows the numerical sensitivity of each sub-module of the computational graph: compute-bound and numerically stable modules go to TensorRT FP16, sensitive modules stay in PyTorch FP32, and the trade-off between inference speed and numerical precision is settled module by module rather than for the graph as a whole. The contributions are threefold.

First, a segmented hybrid inference architecture for UDIS2 is proposed. The ResNet-50 backbone feature extraction and the CCL correlation computation run as a TensorRT FP16 engine, whereas DLT solving, homography warping and TPS warping stay in PyTorch FP32. Hardware acceleration is thus applied where it is safe and withheld where it is not.

Second, a four-path accuracy verification framework centred on the peak signal-to-noise ratio (PSNR Gate) is established. Following the controlled-variable method, it separates the error sources of the Warp stage and quantifies what each of them contributes to the final stitching quality.

Third, the system is implemented and measured on an NVIDIA RTX 4060 Laptop GPU. End-to-end latency falls to 39.7 ms (25.2 FPS) at a PSNR of 51.3 dB, which shows that real-time deployment is reachable without giving up high-fidelity visual quality.

The paper is organized as follows. Section II reviews the stitching techniques and the inference acceleration methods related to this study. Section III sets out the design principles and key techniques of the proposed hybrid architecture. Section IV reports the experimental configuration, the quantitative results and the ablation analysis. Section V concludes and outlines further work.

II. Related Work

Image stitching aligns several images captured from different viewpoints and fuses their appearance into a single panorama of wider field of view. Along the evolution of the technical routes, existing methods fall into two families: feature-matching pipelines of the traditional kind, and end-to-end deep networks.

A. Traditional Image Stitching Based on Feature Matching

The classical stitching pipeline has four stages: feature detection, feature matching, geometric model estimation and image fusion [3]. The SIFT descriptor of Lowe [8] served for years as the standard registration feature because of its scale and rotation invariance. Bay et al. [9] brought a Hessian-based acceleration into SURF and cut computation time while holding matching accuracy. On the modelling side, RANSAC [10] is the

usual choice for estimating a homography robustly from matches contaminated by outliers. All of this works in richly textured planar scenes with small parallax. Matching reliability collapses in weakly or repetitively textured material and in scenes whose depth varies steeply, and the rigid planar assumption of the homography then produces obvious ghosting and geometric distortion [4].

B. Image Stitching Based on Deep Learning

To get past those limits, researchers turned to end-to-end stitching with deep neural networks. DeTone et al. [11] proposed the first deep homography network, regressing the four-point homography parameters directly from an image pair with a VGG-style architecture. Nguyen et al. [12] then removed the dependence on ground-truth annotation through unsupervised training. Both remain tied to a global homography model and handle complex three-dimensional scenes poorly.

The UDIS algorithm of Nie et al. [5] splits stitching into two networks trained separately, geometric alignment (Warp) and appearance fusion (Composition), which avoids the gradient conflict of multi-objective learning. UDIS2 [6] adds a cascade inside the Warp network: a four-point DLT first estimates the global homography for coarse alignment, then a TPS transformation over a dense control mesh performs local non-rigid registration. The Composition network learns a pixel-wise fusion mask with a U-Net encoder-decoder, removing the seam and compensating exposure difference by a soft blending strategy. Alignment capability in wide-baseline scenes is strong. More recent work along this line carries the same warping machinery into online video stitching, where temporal stability becomes an additional objective [13], which pushes model capacity up rather than down. Research on these networks has concentrated almost entirely on stitching quality; the inference cost of their cascaded structure is a new bottleneck once real-time deployment is required, and systematic acceleration work for them remains scarce. That gap is the starting point of the present study.

C. Deep Learning Inference Acceleration Techniques

Inference speed decides whether a deep model is usable in a time-critical setting. Current acceleration routes cover model compression through pruning and knowledge distillation [14], quantization to FP16 or INT8 [15], and computational graph optimization inside dedicated inference engines. Operator fusion, constant folding, memory-pool management and mixed-precision scheduling let TensorRT [7] cut latency by a factor of two to five with negligible accuracy loss. An independent benchmark of five mainstream backends on an edge platform confirms both halves of that picture: TensorRT gives the lowest latency of the group, and it also depends most heavily on the graph being static [16]. Networks carrying dynamic control flow, large-scale matrix inversion or non-standard custom operators frequently hit unsupported operators, accuracy degradation or outright numerical divergence when they are exported end to end [17].

Hybrid inference offers a way around that dependence. The computational graph is cut into sub-graphs, and each sub-graph is placed on the backend that suits it, a TensorRT engine or the native PyTorch runtime, so that hardware acceleration and per-operator accuracy requirements are both respected. Graph-partitioning compilers already implement the mechanism. Torch-TensorRT compiles the convertible part of a module into TensorRT engines and returns the remainder to PyTorch [18], and Apache TVM [19] performs the analogous split against its own code generator. On embedded hardware, Jeong et al. mapped the layers of a single network across the GPU and the deep learning accelerators of a Jetson board and reported throughput gains of 101 to 680 percent over GPU-only execution [20], [21]. What all of these work's partition, however, is one homogeneous perception network. None of them addresses a cascade in which a learned sub-network alternates with a closed-form geometric solver, which is exactly the shape of a deep stitching pipeline. Taking the real-time deployment requirement of UDIS2 as its target, this paper proposes a segmented hybrid inference architecture oriented to image stitching scenarios.

III. Design and Key Techniques of the Hybrid Inference Architecture

This section follows the design logic of segmentation by numerical sensitivity, then heterogeneous backend deployment, then accuracy gating verification. The computational characteristics and numerical sensitivity of each module of the UDIS2 base algorithm are analysed first (Section III-A). The inference pipeline is then partitioned and allocated to the two backends of TensorRT FP16 and PyTorch FP32, and the performance bottleneck inside the partition is optimized (Section III-B). The components are integrated into a unified end-to-end pipeline (Section III-C), the accuracy loss introduced by the partition is verified source by source through a four-path PSNR gating framework (Section III-D), and the system engineering implementation closes the section (Section III-E). Fig. 1 shows the overall architecture.

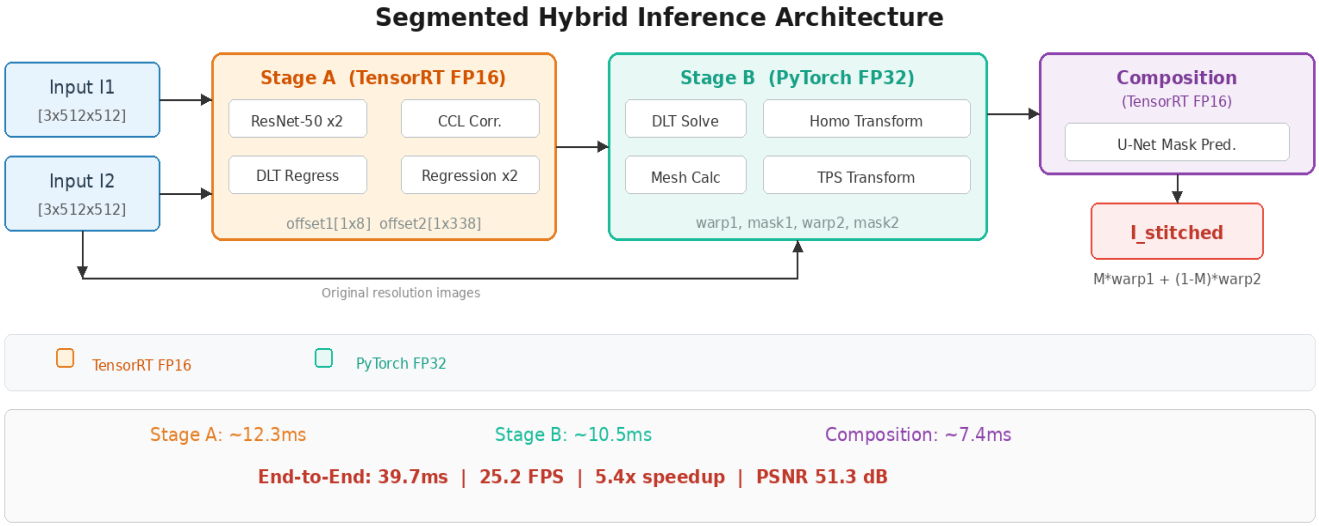


Figure 1. Overall system flow of the TensorRT-based segmented hybrid inference architecture.

A. Overview of the UDIS2 Base Algorithm

UDIS2 adopts a Warp-Composition two-stage cascaded architecture to accomplish the stitching task [6]. Given the input image pair $I_1 \in {}^{(3\times H\times W)}$ and $I_2 \in {}^{(3\times H\times W)}$ from the left and right cameras, the Warp network estimates the geometric transformation parameters and generates a spatially aligned pair of warped images, on the basis of which the Composition network learns pixel-wise fusion weights that produce the final seamless stitching result.

1) Warp Geometric Alignment Network: The Warp network undertakes the task of estimating the geometric transformation relationship between the input image pair and is internally composed of the following key computational modules connected in sequence.

a) Dual-branch backbone feature extraction: After normalization, the input images are fed separately into two weight-sharing ResNet-50 backbone networks, which extract the multi-scale deep features $F_1, F_2 \in {}^{(C\times H'\times W')}$.

b) Correlation computation layer (CCL): This layer performs a cross-correlation operation on the backbone features and establishes the pixel-level correspondence between the two images, and the computation process is

$$C(i,j) = \frac{<F_1(i), F_2(j)>}{\|F_1(i)\| \|F_2(j)\|} \tag{1}$$

where $<,>$ denotes the inner product operation, and i and j index the spatial positions on the two feature maps respectively. The correlation volume C encodes the cross-view pixel matching probability distribution and provides dense correspondence information for the subsequent regression of the geometric parameters.

c) DLT homography matrix estimation: The

correlation features are passed through a fully connected regression sub-network which outputs an eight-dimensional parameter vector offset $t_1 \in {}^{8}$ encoding the displacements of the four corners of the reference image. The 3×3 homography matrix H is solved from the corner correspondences by means of the direct linear transformation (DLT) algorithm [22]

$$Ah = 0, H = reshape\left(h, 3\times 3\right) \tag{2}$$

where A is the 8×9 coefficient matrix formed by the coordinates of the four pairs of corner points, and h is the nine-dimensional vectorized representation of H, which is obtained by performing singular value decomposition (SVD) on A and extracting its null space.

d) TPS thin-plate spline transformation: In order to characterize the local non-rigid deformation that cannot be expressed by the global homography matrix, the network further regresses a 338-dimensional parameter vector offset $t_2 \in {}^{338}$, which encodes the offsets of the individual nodes of the $(grid_{h+1})\times(grid_{w+1})$ dense control mesh. The TPS transformation establishes a smooth mapping from the control points to the target positions by solving the following system of linear equations

$$\begin{cases} K\cdot w + P\cdot a = v \\ P^T\cdot w = 0 \end{cases} \tag{3}$$

where $K \in {}^{(n\times n)}$ is the kernel matrix composed of the values of the radial basis function between control points, whose elements are defined as $K_{ij} = U(\| p_i - p_j \|)$, with $U(r) = r^2 ln(r)$ being the TPS kernel function; $P \in {}^{(n\times 3)}$ is the matrix of homogeneous coordinates of the control points; w and a correspond to the non-affine and affine transformation parameters respectively; and v is the vector of target point coordinates. The solving process requires an inversion operation to be performed on an augmented matrix of dimension $(n+3)\times(n+3)$. Under the default configuration of UDIS2 the number of control points is n=169, corresponding to a matrix dimension of 172×172.

2) Composition Fusion Network: The Composition network adopts a U-Net encoder-decoder structure [23] and accepts as input the pair of warped images $warp_1, warp_2 \in {}^{(3\times Ho\times Wo)}$, output by the Warp network. The two warped images are concatenated along the channel dimension to form a six-channel tensor, which, after being processed by a five-layer downsampling encoder and a four-layer upsampling decoder, yields a single-channel fusion mask $M \in [0,1]^{(1\times Ho\times Wo)}$. The final stitched image is given by the weighted fusion formula

$$I_{stitch} = M \odot warp_1 + \left(1 - M\right) \odot warp_2 \tag{4}$$

where $\odot$ denotes the elementwise Hadamard product. The fusion mask M adaptively regulates the contribution ratio of the two warped images at each pixel position: when M approaches 1, the output pixel is dominated by $warp_1$, which corresponds to taking the left-branch view as the reliable source at that position; when M approaches 0, the output is dominated by $warp_2$ and the right-branch view is adopted instead; and in overlapping regions where the quality of the two branches is comparable, M takes intermediate values, which yields a smooth colour transition and compensates the exposure difference.

Experiments further reveal how sharply the Composition network reacts to the output of the Warp stage. Once a systematic coordinate deviation enters the warped images, fusion amplifies it instead of absorbing it: a mean pixel deviation of 0.66 in $warp_1$ drives the PSNR after Composition fusion down from 110 dB to 10.7 dB, as reported in Table V of Section IV-C. This behaviour sets a hard accuracy constraint on the geometric post-processing segment of the hybrid architecture, and it is the reason that segment is kept in FP32 throughout what follows.

B. Design of the Segmented Hybrid Inference Architecture

Directly exporting the complete UDIS2 Warp network as a single TensorRT engine faces two core obstacles. On the one hand, the SVD decomposition in DLT solving is liable to trigger numer-

ical instability under FP16 representation. On the other hand, the inversion of the 172×172 matrix in the TPS transformation is internally replaced by the Newton-Schulz iterative approximation algorithm in TensorRT, and the accumulated error thus incurred reaches a catastrophic level in the $offset_2$ dimension—a mean deviation of 9.96 and a relative error of 148%. Accordingly, this paper splits the inference process of the Warp network into two execution stages, which are executed in the TensorRT engine and in the PyTorch runtime respectively.

1) Stage A: TensorRT Acceleration Segment:

Stage A covers the computationally intensive but numerically stable sub-graph of the Warp network, including the dual-branch ResNet-50 backbone network, the CCL correlation computation layer and the two-level regression sub-networks. These modules take standard convolution, fully connected layers and ReLU activation as their principal computational primitives, and are thus naturally suited to the operator fusion and FP16 quantization optimization of TensorRT.

In the export stage, the above sub-graph is converted into an ONNX intermediate representation by means of the torch.onnx.export interface of PyTorch (opset 17, input shape [1,3,512,512]×2, outputs $offset_1 \in {}^{(1\times 8)}$ and $offset_2 \in {}^{(1\times 338)}$. Statistics show that the ONNX model contains 1326 operator nodes with a file size of 297.72 MB. The ONNX model is subsequently compiled into a serialized engine by means of the trtexec tool or the TensorRT Python API, with FP16 mode enabled during the build.

As regards the safety analysis of FP16 precision, the numerical range of the Conv-BN-ReLU computation chain of ResNet-50 under FP16 is entirely controllable; although the Softmax normalization in the CCL involves exponential operations, the backbone features have already been brought into a safe numerical range after normalization and will not give rise to overflow. It has been verified that the mean absolute deviation between the $offset_1$ output of the Stage A engine and the PyTorch FP32 baseline is 0.034, with a relative error of only 0.05%.

2) Stage B: PyTorch Exact Computation Segment:

Stage B covers the numerically sensitive geometric post-processing sub-graph of the Warp network, including offset parameter scaling, DLT homography matrix solving, rigid mesh initialization, dynamic canvas computation, homography transformation and TPS transformation. All these operations are retained for execution in the PyTorch FP32 environment, which keeps bit-level consistency with the inference results of the original version.

The complete computational flow of Stage B can be divided into six steps.

In Step 1, the $offset_1$ output by Stage A is reshaped into a 4×2 matrix h_motion and multiplied by the scaling factor $(W_{orig}/512,\ H_{orig}/512)$ that adapts it to the original image resolution; $offset_2$ is likewise reshaped into a mesh_motion tensor of size $(grid_{h+1})\times(grid_{w+1})\times 2$ and subjected to the same scaling.

In Step 2, taking the four corner coordinates of the original image as the source points src_p and $src_p + h_{motion}$ as the target points dst_p, the DLT algorithm is invoked to solve the homography matrix $H \in {}^{(3\times 3)}$.

In Step 3, a uniform rigid mesh rigid_mesh is initialized and mapped to the target plane through the inverse transformation of H to obtain ini_mesh, upon which mesh_motion is superimposed to give the TPS control mesh mesh.

In Step 4, the dynamic canvas extent is computed from the corner coordinates of mesh (the min/max values of all corner points being taken), which determines the spatial size (H_{out},W_{out}) of the output image. A fixed canvas size is not adopted here, because different input images and different offset parameters cause the canvas extent to vary accordingly, and a fixed size would introduce additional pixel offset error.

In Step 5, the homography transformation is performed, mapping I_1 into the output canvas space through the transformation matrix, which gives $warp_1$.

In Step 6, the TPS transformation is performed, mapping I_2 into the output canvas space through thin-plate spline interpolation between the normalized control meshes, which gives $warp_2$.

The flow above reproduces the logic of the latter half of the original build_output_model function of UDIS2. Given the gold offset as input, that is, the offset produced by the original PyTorch implementation, the deviation of $warp_1$ in Stage B is 0 and that of $warp_2$ is 1.85×10^{-6}, corresponding to a PSNR of 110 dB, which is the bit-exact level.

At the implementation level, Stage B must further overcome the performance bottleneck of the original TPS transformation: the TPS transformation of the original UDIS2 completes the coordinate mapping by means of a pixel-wise Python loop and, together with the torch.inverse inversion of the 172×172 matrix, incurs a latency of approximately 30 ms per invocation, which is the principal source of time consumption of the entire inference pipeline. In response to this problem, two optimization strategies are adopted in this paper.

First, the pixel-wise coordinate mapping is replaced by the PyTorch-native F.grid_sample function. F.grid_sample performs the resampling of all pixels in parallel on the GPU by means of bilinear interpolation, thereby fundamentally eliminating the loop overhead at the Python level.

Second, the exact solution of the control point equation system by torch.inverse is retained. The matrix inversion operation is itself of limited time consumption (approximately 2 ms), but its result directly determines the spatial mapping accuracy of the TPS transformation and is therefore not suitable for approximate simplification.

After these two changes, a single invocation of the TPS transformation drops from 30 ms to roughly 8 ms, a speedup of 3.75 times at no cost in PSNR. Stage B keeps FP32 precision and is no longer the performance bottleneck of the pipeline.

3) Composition TensorRT Engine:

The composition network is a standard U-Net structure consisting of five DownBlock layers and four UpBlock layers, with a total of 115 ONNX nodes and a model size of 34.54 MB. Its computational primitives are dominated by convolution, pooling and upsampling, and involve no numerically sensitive operations such as matrix inversion, so that it can be safely deployed in TensorRT FP16 mode.

One technical detail requiring attention during the export process is that the forward method of the Composition network formally accepts four parameters ($warp_1$, $warp_2$, $mask_1$, $mask_2$), but a source code audit confirms that $mask_1$ and $mask_2$ are not used inside the network. Accordingly, only the two inputs $warp_1$ and $warp_2$ are retained in the ONNX export, which reduces invalid data transfer. The problem of non-fixed input shapes caused by the dynamic canvas is solved by setting a dynamic shape profile (specifying the minimum, optimal and maximum shapes) at the TRT build stage.

C. Integration of the End-to-End Inference Pipeline

The complete end-to-end inference pipeline connects the above three components in series into a unified forward computation process, as shown in Fig. 2.

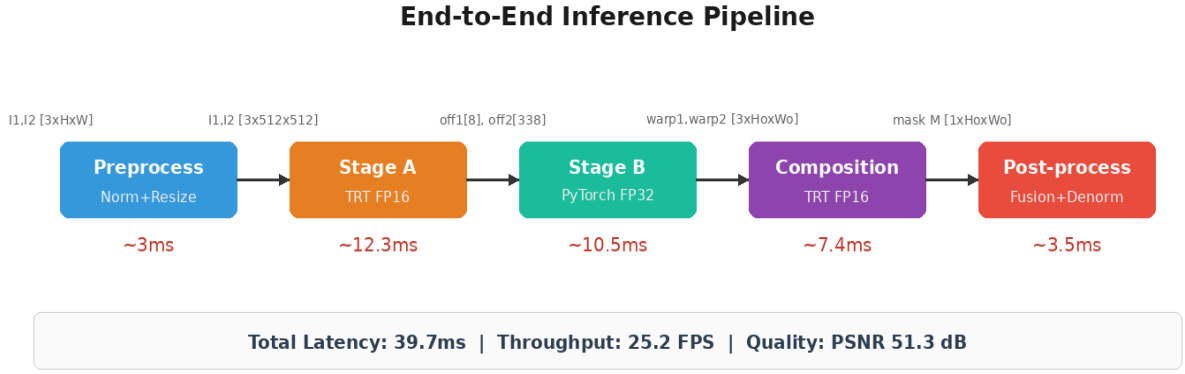


Figure 2. Flow of the end-to-end inference pipeline and latency distribution of each stage.

Given the input image pair (I_1, I_2), the inference process passes through the following five stages in sequence.

1) Image preprocessing:

I_1 and I_2 are normalized to the interval [-1, 1] and scaled to a resolution of 512×512, which forms the standard input tensors of the TensorRT engine.

2) Stage A inference (TRT FP16):

Asynchronous inference is executed by means of preallocated GPU buffers, outputting offset₁ and offset₂. A CUDA stream together with torch.cuda.synchronize() is used to ensure synchronization with the subsequent PyTorch computation.

3) Stage B computation (PyTorch FP32):

The $offset_1$ and $offset_2$ are received, and the geometric post-processing flow described in Section III-B2 is executed at the original resolution, outputting $wrap_1$, $mask_1$, $wrap_2$ and $mask_2$.

4) Composition inference (TRT FP16):

The $wrap_1$ and $wrap_2$ are fed into the Composition TRT engine, which outputs the fusion mask M.

5) Image fusion and post-processing:

The final stitched image I_stitch is computed according to (4), denormalized to [0, 255] and converted into uint8 format for output.

In the above pipeline, the GPU buffers of the Stage A engine and of the Composition engine are preallocated at the system initialization stage, so that no repeated application for and release of GPU memory is required at runtime, which eliminates the additional overhead of CUDA memory management. The data transfer between Stage A and Stage B is always completed within GPU memory—the offset tensors are passed directly as CUDA tensors, involving no data movement between the CPU and the GPU.

D. Accuracy Evaluation Method

To quantify the accuracy loss that the hybrid inference architecture introduces at each stage, this paper designs a four-path PSNR verification framework (PSNR Gate), as shown in Fig. 3 and Table I. This framework adopts the controlled-variable method, replacing each component of the inference pipeline one by one with either the original PyTorch implementation or the TensorRT engine, so as to isolate the independent contribution of each error source to the final stitching quality.

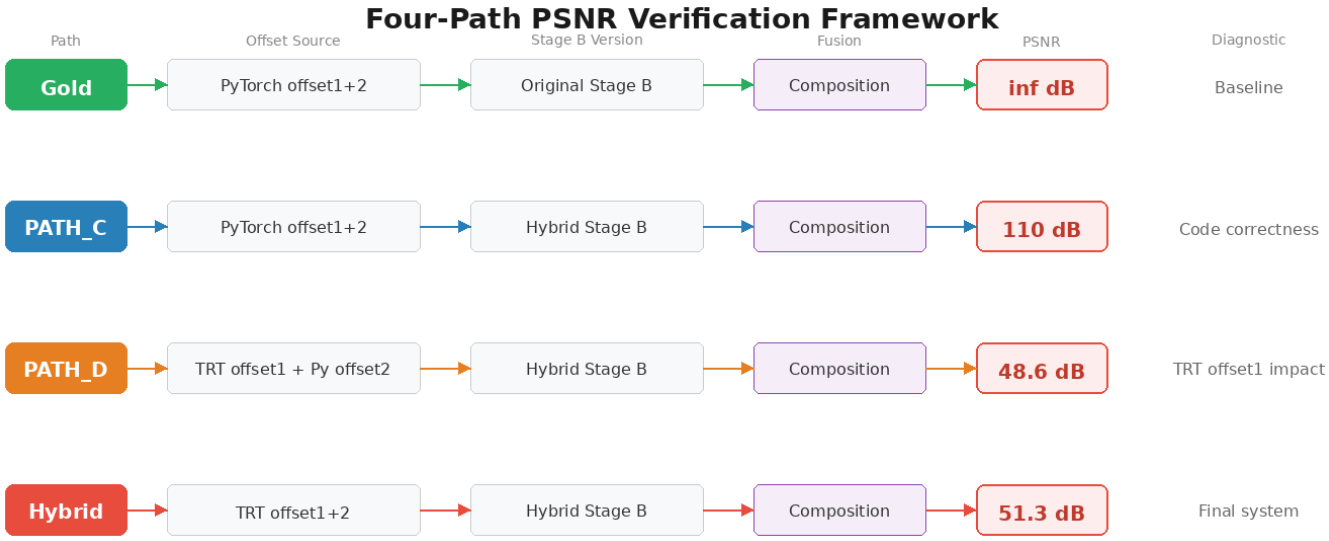


Figure 3. Schematic diagram of the four-path PSNR accuracy verification framework.

TABLE I. PATH CONFIGURATIONS OF THE FOUR-PATH ACCURACY VERIFICATION FRAMEWORK

Path	Source of $offset_1$	Source of $offset_2$	Stage B version	Diagnostic significance
Gold	Original PyTorch	Original PyTorch	Original function	Gold-standard baseline
PATH_C	Original PyTorch	Original PyTorch	Hybrid architecture	Correctness of Stage B code
PATH_D	TRT FP16	Original PyTorch	Hybrid architecture	Effect of TRT precision on $offset_1$
Hybrid	TRT FP16	TRT FP16	Hybrid architecture	Final accuracy of the complete system

The PSNR is defined by the formula

$$PSNR = 10log_{10}(\frac{MAX^2}{MSE}) \tag{5}$$

where MAX is the maximum value of the pixel values (MAX = 255 for an 8-bit image) and MSE is the mean squared error between the reference image and the test image

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2 \tag{6}$$

In which x_i is the i-th pixel value of the gold-standard output of the original PyTorch implementation, $\hat{x}_i$ is the corresponding pixel value output by the hybrid inference architecture, and N is the total number of pixels of the image. A PSNR ≥ 50 dB is generally considered to indicate that the difference between the reference image and the test image is indistinguishable to human visual perception.

E. System Engineering Implementation

The front end of the system is built as a web visualization console by means of the Gradio framework, which provides the user with two working modes, namely real-time stitching and offline stitching. The real-time stitching mode takes dual USB or RTSP cameras as input sources, invokes the above hybrid inference pipeline in a loop and stitches frame by frame, and presents the results in real time in the form of a three-view preview (left-branch input, right-branch input and stitched output). The offline stitching mode is oriented to the production requirements of high-quality material and supports the export of stitching results for single images and for batch videos.

The system does not depend on any database middleware; configuration persistence, calibration caching and the runtime mutex lock are all managed by means of the file system. At the time of first deployment, the calibration module acquires the camera pose parameters and writes them into a cache file, which is loaded directly at subsequent start-ups, thus dispensing with the cold start overhead of repeated computation. The runtime mutex lock is implemented on the basis of a file locking mechanism; when the process PID recorded in the lock file is detected to have terminated, the lock is automatically judged to be residual and is cleared, so that GPU resource conflicts caused by concurrent multi-tasking are effectively prevented.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

This section evaluates the proposed method systematically from four dimensions: inference speed, stitching accuracy, ablation analysis and visualization effect.

A. Experimental Environment and Parameter Configuration

All the experiments were completed on a local workstation, and the detailed hardware and software configurations are given in Table II.

TABLE II. EXPERIMENTAL ENVIRONMENT CONFIGURATION

Category	Detailed configuration
Operating system	Windows 10 (64-bit)
GPU	NVIDIA GeForce RTX 4060 Laptop (8 GB VRAM)
CUDA	CUDA 11.8
Python	3.10.20 (Miniconda)
Deep learning framework	PyTorch 2.0.0+cu118
Inference engine	TensorRT 8.6.1.6
ONNX opset	17
Test images	512×512 (Stage A input) / original resolution (Stage B)

The test images are pairs of dual-camera frames captured in real street-view scenes. Every latency figure is the minimum over 50 consecutive runs, which excludes the fluctuation caused by thermal throttling on a laptop and reflects the true peak

performance of the hardware.

B. Comparative Analysis of Inference Speed

Table III gives the stage-by-stage latency comparison between the original PyTorch inference pipeline and the hybrid inference architecture proposed in this paper.

As can be seen from Table III, the hybrid inference architecture compresses the end-to-end inference latency from 213.9 ms to 39.7 ms, an overall speedup of approximately 5.4 times. Among these, the Warp stage is reduced from 166.9 ms in the original version to 22.8 ms (a speedup of 7.3 times), which mainly benefits from the full utilization of the Tensor Cores by the ResNet-50 backbone network and the CCL operation on the TensorRT FP16 engine. The latency of the Composition stage is reduced from 43.2 ms to 7.4 ms (a speedup of 5.8 times), and the standard convolutional structure of U-Net likewise obtains a significant throughput improvement after TensorRT optimization.

TABLE III. COMPARISON OF INFERENCE LATENCY (UNIT: MS, MIN VALUES TAKEN)

Inference stage	Original PyTorch / ms	Hybrid architecture / ms
Preprocessing (normalization + resize)[a]	—	3.0
Warp Stage A (feature extraction + CCL + regression)	—	12.3
Warp Stage B (DLT + homography + TPS)	—	10.5
Warp, complete	166.9	22.8
Composition	43.2	7.4
Post-processing (fusion + denormalization)	3.8	3.5
Inter-stage synchronization and data transfer	—	3.0
End-to-end total	213.9	39.7
Frame rate / FPS	4.7	25.2
Speedup ratio	1.0×	5.4×

Note:a. In the original PyTorch pipeline the normalization and resizing of the input pair are executed inside the Warp forward pass and are therefore already contained in the 166.9 ms; in the hybrid architecture they are timed separately because they produce the input tensors of the TensorRT engine. The synchronization row covers the CUDA stream synchronization and the tensor hand-over between the TensorRT engines and the PyTorch runtime, which are not attributable to any single stage. Each column therefore sums exactly to the reported end-to-end total.

Of the 10.5 ms spent in Stage B, roughly 8 ms belongs to the optimized TPS transformation described in Section III-B2. Custom CUDA operators, or a further tuning of the grid_sample invocation, leave a margin of 2 to 3 ms for compression in this part.

C. Accuracy Verification and Ablation Analysis

Table IV summarizes the detailed data of the four-path PSNR accuracy verification. Compared with the experimental configuration given in Table I, Table IV reports the measured PSNR and the mean absolute error (MAE) of $warp_I$ over the same four paths, thereby directly relating the "pixel deviation at the geometric level" to the "image fidelity after fusion".

The data in Table IV may be interpreted at three levels. The PSNR of PATH_C is 110 dB, which indicates that the implementation of Stage B attains bit-level numerical consistency with the original build_output_model function, and confirms the correctness of the code of the geometric post-processing module in the hybrid architecture. The PSNR of PATH_D is 48.6 dB, which reflects the influence of TRT FP16 on the accuracy of off-set_I—the mean absolute deviation of $offset_I$ is only 0.034 (a relative error of 0.05%), but after geometric amplification through DLT solving and homography transformation, a mean error of 0.004 is produced at the pixel level of $warp_I$, and the PSNR falls from 110 dB to 48.6 dB. Although this value is slightly lower than the red-line setting of 50 dB, the two are completely indistinguishable at the level of human visual perception. The PSNR of the Hybrid path is 51.3 dB, which is the overall accuracy of the final system. This value is higher than the 48.6 dB of PATH_D, the reason being that the fusion mask of the Composition network exerts a certain spatial smoothing effect on local errors—part of the warp error is suppressed by the mask weights during the fusion process. The final PSNR of 51.3 dB satisfies the accuracy red-line requirement of 50 dB.

1) Ablation Experiment on Accuracy Sensitivity: To reveal how strongly each error source affects the final accuracy, this paper designs a progressively refined ablation experiment following

the controlled-variable method, the results of which are shown in Table V.

TABLE IV. RESULTS OF THE FOUR-PATH PSNR ACCURACY VERIFICATION

Path	$offset_1$	$offset_2$	Stage B	PSNR/ dB	$warp_1$ MAE
Gold	PyTorch	PyTorch	Original version	∞	0
PATH _C	PyTorch	PyTorch	Hybrid architecture	110.0	0
PATH _D	TRT FP16	PyTorch	Hybrid architecture	48.6	0.004
Hybrid	TRT FP16	TRT FP16	Hybrid architecture	51.3	0.004

Table V carries a warning for engineering practice. When Stage B uses the rewritten fast geometric transformation functions (_fast_homo_transform and _fast_tps_transform), the numerical error of the offset itself stays modest, yet the coordinate-system deviation introduced by the rewritten code, a $warp_1$ deviation of 0.66, collapses the PSNR to 10.7 dB. Comparing the second and third rows of the table isolates the cause: raising the warp deviation from 0.004 to 0.66 costs 37.9 dB of fidelity, so the amplifier is the Composition network rather than the offset error. On the strength of that measurement, Stage B retains the original UDIS2 functions torch_homo_transform and torch_tps_transform unchanged, which removes the systematic deviation of rewritten code altogether.

TABLE V. ABLATION ANALYSIS OF ERROR SOURCES

Ablation configuration	offset deviation	warp deviation	PSNR/ dB	ΔPSNR
Full PyTorch (Gold)	0	0	∞	—
$offset_1$ by TRT only	0.034	0.004	48.6	from ∞ to 48.6
$offset_{1+2}$ by TRT + rewritten functions	9.96 ($offset_2$)	0.66	10.7	-37.9
$offset_{1+2}$ by TRT + original functions	0.034 ($offset_1$)	0.004	51.3	+40.6

Note: ΔPSNR in the table is computed with the configuration of the preceding row as the baseline and therefore reflects the marginal accuracy variation brought about by each introduced change.

D. Comparison with Existing Methods

Table VI compares the proposed method horizontally with several representative image stitching acceleration schemes.

Although exporting UDIS2 in full as a TRT FP32 engine compresses the latency to 73 ms, the Newton-Schulz approximation of the TPS matrix inversion leaves a PSNR of only 10.7 dB, with severe visual distortion present. Enabling FP16 on top of that makes the exponential of the Softmax layer diverge to NaN and the engine becomes entirely unusable. The traditional baseline in Table VI was re-implemented on the same machine with OpenCV 4.8.0 [24]: SIFT keypoints are extracted at the original resolution with default parameters, matched by a FLANN-based nearest-neighbour search under the Lowe ratio test at 0.75, and the homography is estimated by findHomography with RANSAC at a reprojection threshold of 3.0 px, after which the overlap is merged by multi-band blending. The procedure runs entirely on the CPU and takes roughly 350 ms per pair; it also fails on large-parallax material. PSNR is marked as not applicable for this baseline, since its output canvas geometry differs from that of UDIS2 and no pixel-wise correspondence with the gold-standard render exists. Against all of these, the segmented hybrid inference architecture holds 51.3 dB at 39.7 ms and is the only scheme in the table that meets the real-time target (≥ 25 FPS) and the accuracy red line (≥ 50 dB PSNR) at the same time.

TABLE VI. PERFORMANCE COMPARISON WITH EXISTING METHODS

Method	Latency/ms	FPS	PSNR/dB	Acceleration strategy
UDIS2, original PyTorch	213.9	4.7	∞ (baseline)	None
UDIS2, full TRT FP32	73.0	13.7	10.7	End-to-end TRT
UDIS2, full TRT FP16	—	—	NaN	Numerical divergence
SIFT+RANSAC, traditional method	~350	~2.9	N/A	CPU parallelism
Proposed method (Hybrid)	39.7	25.2	51.3	Segmented hybrid inference

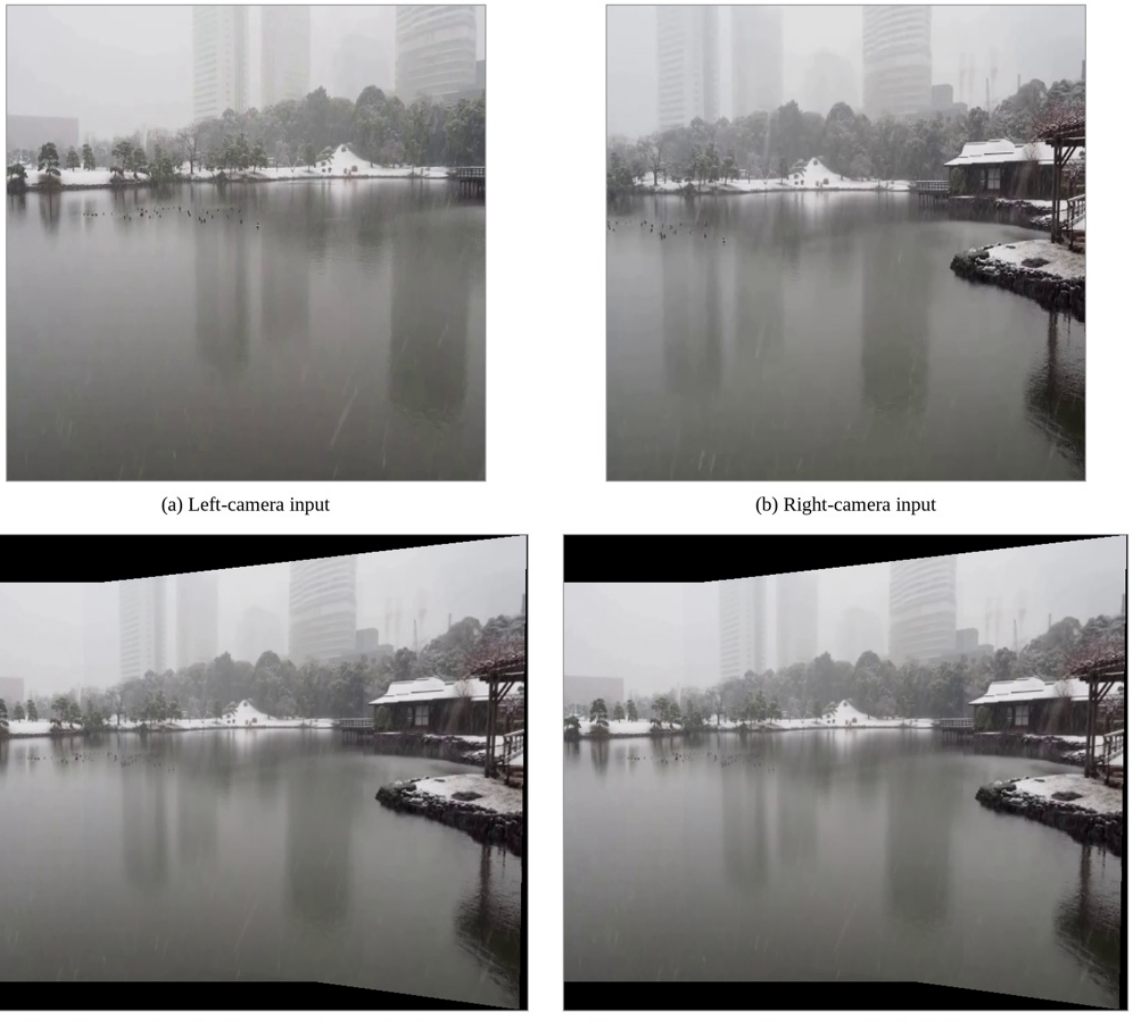
E. Analysis of Visualization Results

Fig. 4 shows the stitching result in a typical street-view scene. Fig. 4(a) and Fig. 4(b) are the raw left-branch and right-branch camera inputs, Fig. 4(c) is the gold-standard render produced by the original PyTorch inference, and Fig. 4(d) is the

output of the hybrid inference architecture proposed here.

Fig. 4(c) and Fig. 4(d) are visually indistinguishable. The seam region transitions smoothly in both, and neither shows ghosting, colour shift or geometric distortion. Magnified inspection of high-frequency detail in the overlap, such as building edges and railing structures, likewise turns up nothing perceptible, which matches the fidelity level implied by a PSNR of 51.3 dB.

Visual agreement of that kind is easy to assert and hard to falsify, so the two renders were also differenced pixel by pixel; Fig. 5 reports the outcome. The residual map of Fig. 5(a) saturates at two grey levels, and the histogram of Fig. 5(b) uses a logarithmic ordinate. Of the 437,040 pixels on the canvas, 84.63 percent are bit-identical between the two pipelines and 99.88 percent lie within a single grey level. Deviations above five levels occur at sixteen isolated pixels, all of them on the boundary between the valid canvas and the black padding, where a sub-pixel shift of the sampling grid flips a pixel between image content and background. The residual that does exist concentrates in the TPS-warped branch on the right of the canvas and along the horizontal shoreline edge, which agrees with the error path traced in Section IV-C: the drift of offset$_1$ is amplified by geometric warping rather than by the fusion step itself.



(a) Left-camera input

(b) Right-camera input

(c) Gold-standard result of the original PyTorch inference

(d) Stitching output of the proposed hybrid inference architecture

Figure 4. Comparison of stitching results in a typical street-view scene: (a) left-camera input; (b) right-camera input; (c) gold-standard result of the original PyTorch inference; (d) stitching output of the proposed hybrid inference architecture.

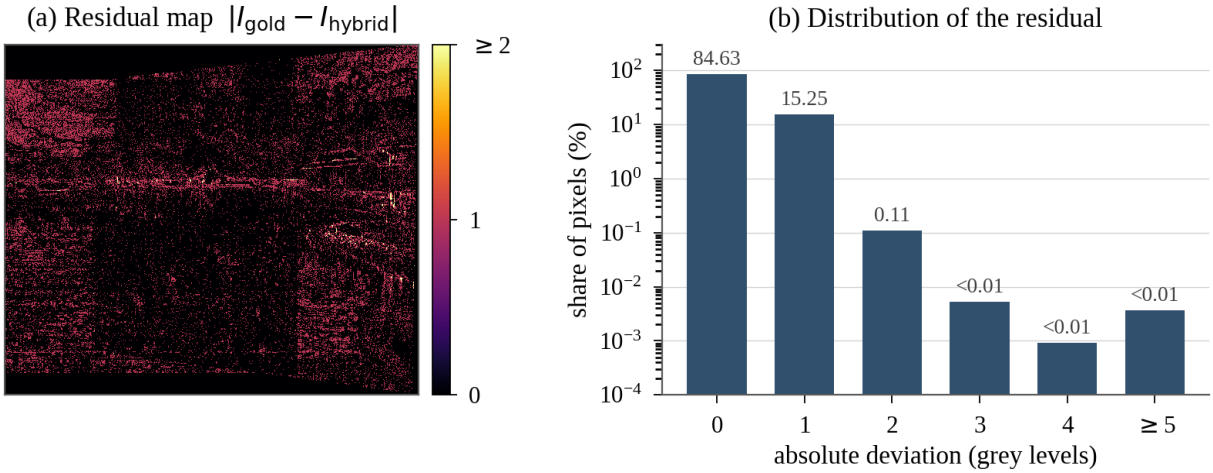


Figure 5. Pixel-level residual between the gold-standard render and the output of the hybrid inference architecture for the scene of Fig. 4: (a) absolute residual map, the colour scale saturating at two grey levels; (b) distribution of the absolute residual over the whole canvas, logarithmic ordinate.

V. Conclusions and Outlook

This paper addresses the inference-latency bottleneck that blocks real-time deployment of deep-learning-based dual-view street-view image stitching and proposes a segmented hybrid inference architecture built on TensorRT. Three ideas organize the design: partition by numerical sensitivity, deploy on heterogeneous backends, and gate the result on accuracy. Following the sensitivity profile of the UDIS2 computational graph, the inference pipeline is divided into a TensorRT FP16 acceleration segment covering backbone feature extraction and the CCL operation, and a PyTorch FP32 exact computation segment covering DLT solving, homography warping and TPS warping. Geometric accuracy is thereby preserved while the hardware is still worked hard. The Composition fusion network is deployed as its own TensorRT FP16 engine, and the four-path PSNR gating framework verifies the accuracy cost of the partition source by source, which yields an end-to-end pipeline that is efficient and trustworthy at the same time.

On the NVIDIA RTX 4060 Laptop GPU platform the end-to-end inference latency falls from 213.9 ms to 39.7 ms, a speedup of approximately 5.4 times, at a frame rate of 25.2 FPS, which clears the benchmark requirement for real-time video processing. The four-path PSNR verification framework puts the final stitching quality at 51.3 dB, above the 50 dB threshold beyond which the human eye cannot separate two images, and the pixel-level audit of Section IV-E places 99.88 percent of the canvas within one grey level of the gold-standard render. Speedup ratio and fidelity are therefore both obtained.

One piece of generally applicable engineering experience emerged along the way. Because the Composition fusion network is so sensitive to what the Warp stage produces, the geometric post-processing module must stay numerically identical to the original algorithm. A fast function rewritten for the sake of performance must not enter the pipeline before pixel-wise verification has been carried out, since a difference that looks negligible at the offset level can still be amplified by the geometric transformation into catastrophic loss of stitching quality. Comparable acceleration work should treat this as a design rule rather than a precaution.

Four directions are open for further work. Custom CUDA operators could accelerate the matrix inversion and the coordinate mapping inside the TPS transformation and compress the latency of Stage B further. The applicability of INT8 quantization to Stage A deserves study, since it would raise the energy-efficiency ratio on edge computing devices. The hybrid architecture also extends naturally to panoramic stitching with more than two cameras, which would widen the field-of-view coverage of the system. Finally, temporal consistency constraints would suppress inter-frame flicker in video stitching and improve the visual stability of long sequences.

ACKNOWLEDGMENT

This article is supported by “Xi'an Technology University's National-level College Students' Innovation and Entrepreneurship Training Program in 2025 (Project Number: 202510702022)”.

REFERENCES

[1] M. Brown and D. G. Lowe, "Automatic panoramic image stitching using invariant features," International Journal of Computer Vision, vol. 74, no. 1, pp. 59-73, 2007.

[2] R. Szeliski, "Image alignment and stitching: A tutorial," Foundations and Trends in Computer Graphics and Vision, vol. 2, no. 1, pp. 1-104, 2006.

[3] J. Zaragoza, T. J. Chin, M. S. Brown, and D. Suter, "As-projective-as-possible image stitching with moving DLT," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 2339-2346.

[4] C. C. Lin, S. U. Pankanti, K. N. Ramamurthy, and A. Y. Aravkin, "Adaptive as-natural-as-possible image stitching," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1155-1163.

[5] L. Nie, C. Lin, K. Liao, S. Liu, and Y. Zhao, "Unsupervised deep image stitching: Reconstructing stitched features to images," IEEE Transactions on Image Processing, vol. 30, pp. 6184-6197, 2021.

[6] L. Nie, C. Lin, K. Liao, S. Liu, and Y. Zhao, "Parallax-tolerant unsupervised deep image stitching," in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2023, pp. 7399-7408.

[7] NVIDIA Corporation, "NVIDIA TensorRT developer guide," NVIDIA Developer Documentation, 2024.

[8] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, vol. 60, no. 2, pp. 91-110, 2004.

[9] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded up robust features," in European Conference on Computer Vision. Springer, 2006, pp. 404-417.

[10] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," Communications of the ACM, vol. 24, no. 6, pp. 381-395, 1981.

[11] D. DeTone, T. Malisiewicz, and A. Rabinovich, "Deep image homography estimation," arXiv preprint arXiv:1606.03798, 2016.

[12] T. Nguyen, S. W. Chen, S. S. Shivakumar, C. J. Taylor, and V. Kumar, "Unsupervised deep homography: A fast and robust homography estimation model," IEEE Robotics and Automation Letters, vol. 3, no. 3, pp. 2346-2353, 2018.

[13] L. Nie, C. Lin, K. Liao, Y. Zhang, S. Liu, R. Ai, and Y. Zhao, "Eliminating warping shakes for unsupervised online video stitching," in European Conference on Computer Vision. Springer, 2024, pp. 390-407.

[14] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in Advances in Neural Information Processing Systems, 2015, pp. 1135-1143.

[15] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704-2713.

[16] I. J. Ratul, Y. Zhou, and K. Yang, "Accelerating deep learning inference: A comparative analysis of modern acceleration frameworks," Electronics, vol. 14, no. 15, article 2977, 2025.

[17] S. Migacz, "8-bit inference with TensorRT," in GPU Technology Conference, 2017, vol. 2, no. 4, p. 5.

[18] NVIDIA Corporation, "Torch-TensorRT: A compiler for PyTorch models targeting NVIDIA GPUs," NVIDIA Developer Documentation, 2024.

[19] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018, pp. 578-594.

[20] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha, "Deep learning inference parallelization on heterogeneous processors with TensorRT," IEEE Embedded Systems Letters, vol. 14, no. 1, pp. 15-18, 2022.

[21] E. Jeong, J. Kim, and S. Ha, "TensorRT-based framework and optimization methodology for deep learning inference on Jetson boards," ACM Transactions on Embedded Computing Systems, vol. 21, no. 5, article 51, pp. 1-26, 2022.

[22] R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision. Cambridge University Press, 2003.

[23] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in International Conference on Medical Image Computing and Computer-Assisted Intervention. Springer, 2015, pp. 234-241.

[24] G. Bradski, "The OpenCV library," Dr. Dobb’s Journal of Software Tools, vol. 25, no. 11, pp. 120-125, 2000.