IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

High-Speed Stereo Visual SLAM for Low-Powered Computing Devices

Ashish Kumar†, Jaesik Park‡, *Member, IEEE*, Laxmidhar Behera†, *Senior Member, IEEE*

***Abstract*—We present an accurate and GPU-accelerated Stereo Visual SLAM design called Jetson-SLAM. It exhibits frame-processing rates above 60FPS on NVIDIA's low-powered 10W Jetson-NX embedded computer and above 200FPS on desktop-grade 200W GPUs, even in stereo configuration and in the multiscale setting. Our contributions are threefold: (*i*) a Bounded Rectification technique to prevent tagging many non-corner points as a corner in FAST detection, improving SLAM accuracy. (*ii*) A novel Pyramidal Culling and Aggregation (`PyCA`) technique that yields robust features while suppressing redundant ones at high speeds by harnessing a GPU device. `PyCA` uses our new Multi-Location Per Thread culling strategy (`MLPT`) and Thread-Efficient Warp-Allocation (`TEWA`) scheme for GPU to enable Jetson-SLAM achieving high accuracy and speed on embedded devices. (*iii*) Jetson-SLAM library achieves resource efficiency by having a data-sharing mechanism. Our experiments on three challenging datasets: KITTI, EuRoC, and KAIST-VIO, and two highly accurate SLAM backends: Full-BA and ICE-BA show that Jetson-SLAM is the fastest available accurate and GPU-accelerated SLAM system (Fig. 1).**

***Index Terms*—Aerial Systems: Applications; SLAM; Embedded Systems for Robotic and Automation**

Supplementary Material

Code: https://github.com/ashishkumar822/Jetson-SLAM
Video: See attachment.

I. Introduction

A Centimeter-accurate local positioning system is crucial for complex robotic and autonomous flight systems to execute navigation, control, and visual servoing tasks precisely [1]. Visual odometry (VO) can be employed for this purpose, but it discards older environmental observations and lacks global consistency [2]. This causes pose-estimation drifts over time, although an agent navigates in the same area.

In contrast, visual SLAM offers drift-free localization and mapping, which enables the precise execution of autonomous tasks. In this context, *Stereo visual SLAM* is particularly interesting due to its high metric accuracy and low-cost sensor demands. However, its compute-intensive frontend (feature detection-extraction-matching, stereo-matching) and backend (graph optimization, loop-closure, localization and mapping) quickly exhaust low-powered devices. Also, the limited computing power of these devices forces a SLAM system to drop intermediate frames [3], resulting in reduced frame rate and tracking failure [2].

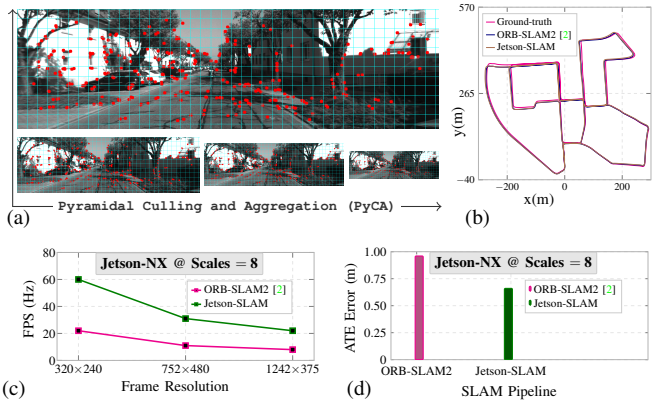

Figure 1: (a) Output of Jetson-SLAM's GPU-accelerated and resource-efficient Frontend–Middle-end design, (b) the output trajectory, (c) Frames-Per-Second benchmarking on Jetson-NX embedded computer, and (d) SLAM performance on a KITTI sequence.

This situation becomes difficult due to shared computing resources in the presence of co-existing modules such as data acquisition, control, grasping, and compute-intensive deep networks. It can easily cause a catastrophic system failure, e.g., the divergence of the control system from the desired trajectory due to delays in position feedback. For these reasons, UAV-based autonomous manipulation [1] executed SLAM on a remote computer. [4] explored network computing for SLAM but is unsuitable for isolated robotic systems.

Although there have been efforts to speed up VO systems, SLAM still remains untouched. For instance, recent [3] only benchmarks runtime of existing VO systems [5], [6], [7] onto Jetson devices. In VO, restricting feature count by dividing an image into grids and picking one feature per grid is commonly used. [8] is such an approach for GPU. However, its design is limited only to monocular VO, preventing its use in SLAM. Its unconfigurable scale factor of two in a multiscale setting reduces the image resolution aggressively, resulting in fewer features which are further trimmed to only one feature per-grid irrespective of the number of scales. It leads to inadequate feature points in SLAM, causing tracking failures (see video). Moreover, it is inefficient for smaller grids when deployed onto embedded computing devices.

ORB-SLAM2 [2], ICE-BA [9] are highly accurate SLAM systems. However, their complex CPU-only workload turns them slower and exhausts the low-powered devices. SLAM-Core [10] is CPU-efficient SLAM but is not open source and is not benchmarked alongside deep networks. Despite these

Manuscript received: August 18, 2023; Accepted October 16, 2023. This paper was recommended for publication by Editor Javier Civera upon evaluation of the Associate Editor and Reviewers' comments.

Jaesik Park was supported by IITP grant funded by the Korea government(MSIT) (NO.2021-0-01343 AI Graduate School Program (Seoul National University) & (RS-2023-00227993: Detailed 3D reconstruction for urban areas from unstructured images)

†EE, Indian Institute of Technology (IIT), Kanpur, India. {krashish,lbehera}@iitk.ac.in ‡CSE & IPAI, Seoul National University (SNU), Republic of Korea. jaesik.park@snu.ac.kr

Digital Object Identifier (DOI): see top of this page.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

limitations, a high-speed SLAM system is the need of the current time with a huge scope in modern autonomous systems.

In visual SLAM, the speed is mainly governed by the frontend load, which varies with image resolution and doubles in stereo mode (of our interest). Also, the number of features yielded by the frontend affects the backend load. Thus, we develop a GPU-accelerated frontend from scratch which produces a sufficiently smaller number of impactful features at high-speed by harnessing the on-chip GPU of the embedded computers. However, in this endeavor, the scarcity of GPU cores appears as a bottleneck that limits the maximum achievable speed, and an inefficient use of the cores degrades the GPU throughput. We tackle this issue via our algorithmic and system development contributions, resulting in a high-speed, accurate, and resource-efficient GPU-accelerated SLAM system available to date, called *Jetson-SLAM*. Our contributions are:

a) Bounded Rectification: prevents misclassification of non-corners as corners in FAST features [11], and improves SLAM accuracy by producing impactful corners (Sec. II-A).

b) Pyramidal Culling and Aggregation (`PyCA`): It yields high-quality multiscale features via our Multi-Location Per-Thread (`MLPT`) culling, and Thread Efficient Warp-Allocation (`TEWA`) to deliver high speeds (2000 FPS) and high computing efficiency even in the scarcity of GPU cores (Sec. II-C).

c) Frontend–Middle-end–Backend Design of Jetson-SLAM: We develop a new SLAM component called *Middle-end* that houses stereo-matching, feature matching, feature-tracking, and performs data-sharing to avoid CPU-GPU memory-transfer overhead of duplicating-and-accessing intermediate results needed across SLAM components (Sec. III).

Despite we contribute in the frontend and system design, the middle-end and backend performance also gets boosted. It turns Jetson-SLAM efficient and accurate while reaching above 60FPS @432×240, even at eight scales in stereo mode on Jetson-NX alongside VGG [12] deep neural network. The high speed minimizes tracking failures during camera rotations (video), and facilitates developing autonomous UAVs which still rely on external positioning systems [1].

Next, we discuss our algorithmic contributions (Sec. II), and system development contributions (Sec. III). Experiments are presented in Sec. IV, with conclusions in Sec. V.

II. Methodology

A. Bounded Rectification for Corner Detection

In FAST [11] detection, the number of consecutive dark (N_d) and bright pixels (N_b) are computed for a segment around each image pixel, known as Bresenhem circle [11] of radius 3 and a length $N_{seg}=16$ pixels. If there exists P_{min} brighter or darker pixels, the center pixel is labeled as a corner (Eq. 1).

$$L_p=\begin{cases}\texttt{bright}, & I_c-I_p<-\varepsilon\\ \texttt{similar}, & -\varepsilon<I_c-I_p<\varepsilon\\ \texttt{dark}, & I_c-I_p>\varepsilon\end{cases} \tag{1}$$

$$L_c=\begin{cases}\texttt{Corner}, & N_b,\ N_d\geq P_{min}\\ \texttt{Non-corner}, & \text{otherwise}\end{cases} \tag{2}$$

where, L_p is the pixel label, I_c and I_p are the intensities of the center pixel and any pixel on the segment respectively. ε is often set to 20, and L_c denotes the center pixel label.

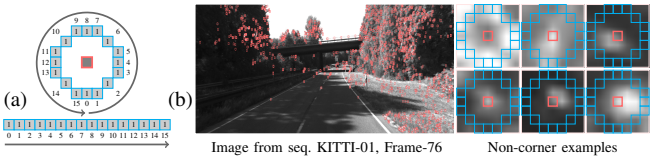

Figure 2: (a) A non-corner but all bright pixels on its Bresenhem circle [11], (b) Real-image examples of such points tagged as a corner. Each rectangle (□) denotes a pixel in a 7×7 patch of the image.

We note that the above process misclassifies many non-corner points as corners. Such points are essentially statistical outliers having a shot-noise-like or tiny blob-like appearance. Interestingly, they satisfy Eq. 1, 2, but can not be used as map points due to their highly similar appearance which confuses the stereo-matcher. For instance, all segment pixels in Fig. 2a are bright and $N_b>P_{min}$, yet it is not a corner. Fig. 2b shows a real-world case. This issue needs to be resolved since the error in the frontend propagates to the other SLAM components. Thus we aim to discard such points at the detection stage itself because it reduces the number of features reaching the stereo-matcher which lowers the matching time. To this end, we propose a key enhancement called *Bounded Rectification* which not only improves the corner quality but also improves the metric accuracy of Jetson-SLAM significantly.

Rectification: we propose to rewrite Eq. 2 which performs upper and lower bounded rectification (Eq. 3).

$$L_c=\begin{cases}\texttt{Corner}, & P_{min}\leq N_b,N_d\leq P_{max}\\ \texttt{Non-corner}, & \text{otherwise}\end{cases} \tag{3}$$

This formulation rejects the cases similar to Fig. 2a by upper and lower bounding N_b or N_d because for the center pixel to be a corner, all of its surrounding pixels can not be `bright` or `dark` simultaneously. This can be assured *iff* $P_{max}<16$, which is the drawback of Eq. 2. P_{min} is generally set to 9 whereas P_{max} can be set anywhere $\in(P_{min},16)$. We use $P_{max}=13$ in our case which does not restrict corner diversity but can be adjusted based on the nature of the visual scenes.

Bounded rectification-based FAST detection outputs a Corner Response Matrix (CRF-matrix, Sec. III-1), which is utilized by our `PyCA` technique. Next, we brief GPU terminologies for better grasping of the upcoming text.

B. GPU Fundamentals

NVIDIA GPUs comprise *streaming multiprocessors* (SM), each having multiple GPU *cores*, and an on-chip *shared memory* with low memory access cost (MAC). Off-chip *global memory* is also present but has a higher MAC. A GPU performs computing in *warp* consisting of 32 *threads* that are concurrently executed on an SM. A *block* comprises many such warps which all are executed on the same SM even if the other SMs are sitting idle. It is so because the threads of a block residing onto an SM may need to communicate with each other, and if warps are executed on different SMs, communication can only be achieved via global memory which has higher MAC, in contrast to the shared memory which has lower MAC but is inaccessible to the other SMs.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

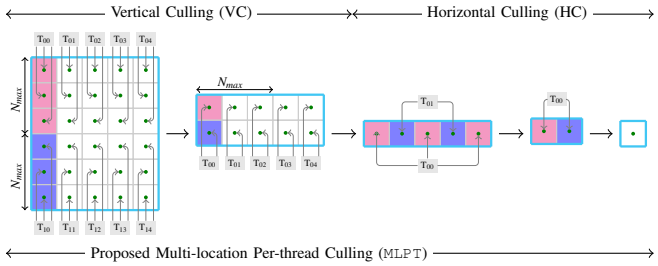

Figure 3: Feature Culling (FC) for a 6×5 cell. T_{ij} is a CUDA-thread of CUDA kernel [13]. A '•' indicates the corner strength of a pixel.

C. Pyramidal Culling and Aggregation (PyCA)

PyCA detects robust features at high speeds (in μS) on GPU via its Feature Culling (FC) and Pyramidal Feature Aggregation (PFA) steps. FC harvests the strongest corners by suppressing the weaker ones, mimicking a culling behaviour, while PFA harvests robust features from the output of FC applied at multiple scales. PyCA remains aware of GPU core scarcity, which, if left unaddressed, reduces the throughput.

1) Feature Culling (FC): It divides the CRF-matrix into non-overlapping cells of a size (c_h, c_w) pixels, and then performs vertical, and horizontal culling in each cell (Fig. 3).

a) Vertical Feature Culling: In this step, a cell is traversed vertically, and the maximum response is recorded for each column. To achieve that, we propose Multi-Location Per Thread (MLPT) culling in which we divide the vertical cell dimension c_h into chunks (Multi-Location), and then process each of them with a single thread (Per-Thread). The total number of threads required in this process is given by:

$$N_t = \min(1,\ \lceil c_h/N_{max} \rceil), \tag{4}$$

where N_{max} is the maximum location that a single thread processes. It is adjustable per the needs. Based on our analysis, it can be set in the range $[1,10]$ for common cell sizes.

Now each of the N_t threads stores its result in shared memory which is utilized by the first thread (T_{0j}) of each column. The thread T_{0j} finds the strongest response among N_t values and stores the result for horizontal feature culling.

For the above operation, $\log_2$-reduction [14] can also be used, however, it processes only two locations per thread, thus requiring a huge number of threads per cell which incurs GPU kernel-launch overhead. Moreover, it is repeated $\lceil \log_2(c_h) \rceil$ times which increases the number of warps required and thus increased runtime to process multiple cells. Also, culling an entire column using one thread becomes slower and inefficient for larger cells in a scenario of core scarcity.

In contrast, MLPT avoids these issues by using fewer threads regardless of the cell size and also conforms to memory coalescing (Fig. 4). This can be verified via Table I, that $\log_2$-reduction requires more warps which induce overhead, even for one cell on Jetson-NX-like embedded devices having core scarcity (Table III). Since multiple cells exist in an image, MLPT can run faster by the order of milliseconds.

b) Horizontal Feature Culling: Now feature culling is performed in the horizontal direction over c_w responses, stored in the shared memory by the previous step. In this case, memory is consecutive, therefore memory coalescing can not

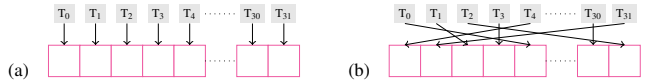

Figure 4: (a) coalesced, and (b) non-coalesced memory access. '□' denotes contiguous memory block, and a 'T_i' denotes i^{th} warp thread. In coalesced access, 32 threads read in one machine-cycle, whereas in non-coalesced access, the memory transactions are serialized [13].

Table I. MLPT vs $\log_2$-reduction. N_w: number of warps.

Cell-size	Culling Scheme	N_{max}	N_t	Total Threads	N_w	Time (μS RTX-2070	Time (μS Jetson-NX
20×32	• $\log_2$	–	10	320	10	4.9μS	11.4μS
	• MLPT	5	4	**160**	**5**	**4.6**μS	**8.7**μS
50×32	• $\log_2$	–	25	800	25	3.2μS	20.4μS
	• MLPT	5	4	**320**	**10**	**2.9**μS	**16.3**μS
100×32	• $\log_2$	–	50	1600	50	3.8μS	14.0μS
	• MLPT	10	10	**320**	**10**	**2.6**μS	**10.6**μS

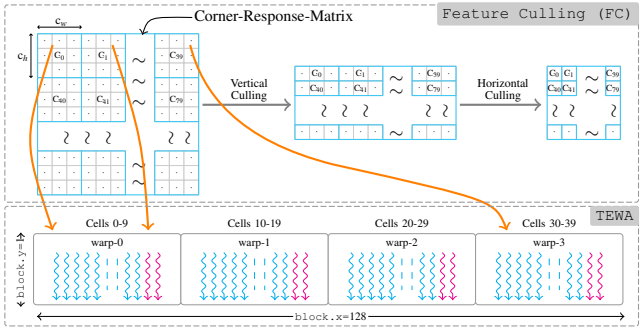

Figure 5: Illustration of FC + (TEWA) scheme. FC is applied over the CRF-Matrix which produces the strongest corner in a cell. A 'C_i' is a cell, and a '↓' and a '↓' denote a working and an idle/wasted thread in a warp respectively for a 3×3 cell-size.

be achieved, but memory transactions can be minimized. To achieve that, a thread in MLPT always accesses an element at a stride of N_{max} instead of a consecutive location, unlike vertical culling. This combines the memory transactions of N_t threads of a warp into one. The remaining process is the same as the previous i.e. MLPT performed over c_w locations which produce the strongest corner in the cell if present.

c) Thread Efficient Warp-Allocation (TEWA): A naïve way to allocate GPU for FC is to assign a block size equal to the cell size. However, a job requiring less than 32 warp threads (W) leads to a wastage of leftover threads since they do not involve in the computations but are still part of a warp. This leads to poor throughput on Jetson devices due to core scarcity. The multiscale scenario becomes more challenging due to smaller cells at lower scales. For instance, a block consists of 9 threads for a 3×3 cell which if executed in a warp, will lead to a wastage of 23 threads, indicating severely low *warp-efficiency*, defined as:

$$\eta_w = \frac{N_{ta}}{WN_w} \tag{5}$$

where, N_{ta}, N_w, W are the number of active threads, number of warps required, and threads per warp respectively.

Hence, we propose a thread-efficient warp allocation scheme (TEWA) that offers high warp efficiency regardless of the cell size. In this scheme, we assign multiple cells to a single thread block by virtually partitioning the warps into chunks, each chunk handling one cell. We fix x dimension (block.x) of the thread block to 128, and find the maximum number of the cells that can be fit into block.x, while y dimension

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

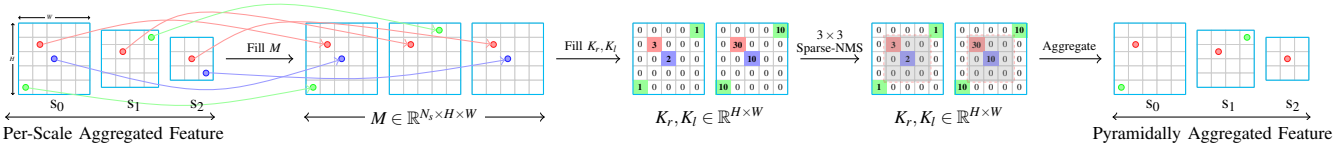

Figure 6: Pyramidal Feature Aggregation (PFA). Steps of a single 5×5 image at scale factor 1.2 and three scales. A '●' indicates a survived point in the PFA process, a '●' indicates a suppressed point, and a '●' indicates an unaffected point.

Table II. Thread-Efficient Warp allocation (TEWA) efficiency (Eq. 5).

Cell-size	Warp allocation Scheme	N_{ta}	N_w	η_w
3×3	• Single block per cell	9	1	28%
	• TEWA	30	1	**94%**
5×5	• Single block per cell	25	1	78%
	• TEWA	30	1	**94%**
7×7	• Single block per cell	49	2	76%
	• TEWA	28	1	**88%**

(`block.y`) of the block is set to N_t. Together both of these numbers are utilized to launch the GPU kernels. It might be the case that a few threads in the block might still sit idle because the warp-size W is not an integer multiple of cell-size, however, the wastage is minimal in TEWA.

Fig. 5 depicts the TEWA design, where, FC for the entire CRF-matrix occurs in one CUDA kernel-launch call that avoids additional launch overhead. Moreover, multiple cells are processed in a single warp which reduces the number of warps and cores required. This leads to high throughput in FC contrary to the naïve allocation (Table II), eventually reducing the runtime and latency, even in GPU core scarcity.

2) Pyramidal Feature Aggregation (PFA): It is the final step of PyCA that processes the features obtained by performing FC at each scale of the image pyramid in a multiscale setting. PFA reduces the overall feature count, which lowers the feature and stereo-matching calculations, thus lowering the overall runtime without affecting the SLAM accuracy.

PFA is motivated by two reasons: *First*, the multiscale setting is crucial in SLAM to obtain robust features which govern tracking, localization, and mapping accuracy. Hence, features that project to a common pixel at the native scale (s_0), need to be prioritized because of their uniqueness and consistency across scales. *Second*, the projected features may fall into the vicinity of each other, which confuses the feature-matcher and stereo-matcher, and also increases the computation time of the frontend (feature extraction), middle-end (stereo matching, tracking) and backend (optimization).

PFA is carried out in three steps (Fig. 6). Firstly, we project all keypoints to s_0, i.e. $x_0 = x_n \times \zeta^n$, and copy their responses in a 3D matrix $M \in \mathbb{R}^{N_s \times H \times W}$ initialized with zeros. Here x_0 denotes s_0 correspondent of a coordinate x_n at n^{th} scale with scale-factor ζ, and N_s, H, W denote the number of scales, image height and width respectively at the s_0. Secondly, we compute two metrics for each keypoint, i.e. the sum of the corner responses across scales if a keypoint is detected at multiple scales (k_r), and the total number of levels at which it is detected (k_l). These scores are stored as two matrices $K_r, K_l \in \mathbb{R}^{H \times W}$. Finally, we perform non-maximal suppression (NMS) over the K_r, K_l matrices, but only sparsely in a $q \times q$ window ($q = 3$ adjustable) around each keypoint, saving a lot of computations. The keypoint is suppressed if its k_r and k_l scores are smaller than any keypoint in the window.

PFA is crafted such that it runs on GPU while avoiding CPU-GPU memory transfer. On the contrary, if run on CPU, its operation unnecessarily consumes CPU and requires CPU-GPU memory transfer, because the input to PFA resides on GPU. It also keeps the CPU occupied, and reduces the memory-transfer bandwidth due to small-sized data transfer, a point of consideration for Jetson devices.

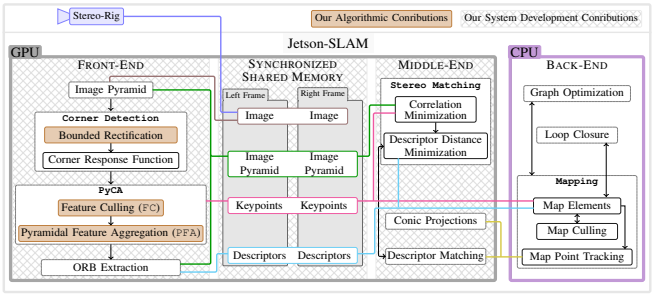

Figure 7: Jetson-SLAM design, highlighting our contributions i.e. bounded rectification, PyCA, Synchronized Shared Memory (SSM), and Middle-end. Jetson-SLAM utilizes them to achieve high speeds and resource efficiency in multi-scale stereo setting. The lines '▬', '▬', '▬', '▬' depict the consumption of shared SSM objects among various SLAM components, interconnected via the lines '▬', '→'.

III. System Integration

In this section, we describe our system development contributions i.e. our new Frontend–Middle-end Jetson-SLAM design (Fig. 7), and strategic integration to optimize information flow. These are crucial to achieving resource efficiency despite the frontend achievements because SLAM components now are multi-device residents (CPU and GPU)

1) μ-Sec. Efficient FAST Detection: We use two 16-bit integers B_b and B_d whose each bit denotes one of the 16 locations on the Bresenhem path (Sec. II-A), and is computed via Eq. 1. A '1' bit corresponds to $L_p = \texttt{bright}$ in B_b and $L_p = \texttt{dark}$ in B_d, whereas a '0' bit signifies $L_p = \texttt{similar}$. To speed up the process, we construct a lookup table where all the 16-bit combinations are pre-calculated to be a corner or a non-corner based on the proposed bounded rectification technique (Eq. 3). We use the sum of absolute differences ($|I_c - I_p|$) over the circle as the corner response [15].

2) Streamlined MultiScale Detection & Extraction: Our frontend is dedicated to high-speed SLAM, however, it can be used as a VO frontend which only performs detection. In contrast, SLAM computes descriptors as well for the map elements, making SLAM slower than VO. The multiscale setting is more compute-intensive, and the scarcity of GPU cores prevents the concurrent execution of multiple jobs. Thus we employ CUDA-streams [13] for faster execution such that when GPU is released for one scale, its CPU work begins, while at the same time, GPU is allocated to another scale.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

Table III. GPU devices specifications.

Attribute	RTX-2070	Jetson-NX
GPU grade	Desktop/Laptop	Edge/Embedded
GPU cores	2304	384
Clock	1620 MHz	1100 MHz
Memory bandwidth	448 GB/s	59.7GB/s
Compute performance	7.4 TFLOPs	1 TFLOPs

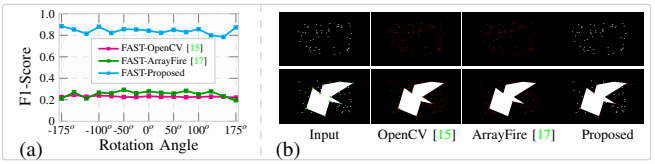

Figure 8: Evaluation of Bounded Rectification. (a) Repeatability score. (b) Qualitative results. *Top*: only corners where baselines do not face issues. *Bottom*: non-corners and corners where only bounded rectification suppresses non-corners but the baselines [15], [17] fail. Green dots are ground truth corners and red dots are detections.

For extraction, ORB descriptor [16] is used due to its speed and uniqueness but existing implementations extract serially for each scale [15], [17]. Since we have the multiscale key points ready, we perform the extraction for all scales at once. To do so, we parallelize Gaussian filtering via CUDA streams, which is essential for robustness [2]. Then the ORB extraction, which is quite a time-consuming step and prevents memory coalescing, is performed at once for all scales, leading to high-speed multiscale detection and extraction (Fig. 10d).

3) Middle-end: Despite the accuracy, stereo visual SLAM [2] poses a high computing burden for Jetson-like devices. In addition, unlike VO, conic projections of the map points in feature tracking are also time-consuming [2]. Hence to attain high throughput, we parallelize both of them, which now form the middle-end. However, naïvely doing so results in inefficiency since stereo matching requires the descriptors and images to be present in the GPU memory. As they are also used by many SLAM components, creating their multiple copies is not desirable for a resource-constraint platform.

Thus, we design an information flow that allows data sharing between the frontend and the middle-end (Fig. 7). It saves memory consumption by preventing duplication, which in turn avoids CPU-GPU data transfer overhead. Achieving this task is programmatically complex, however we tackle it via our synchronized shared memory, as discussed below.

4) Synchronized Shared Memory (`SSM`): We adopt synchronized memory primitives from [18], and on top of which, we build synchronized shared memory (`SSM`) that wraps CPU-GPU transfers and memory allocation/de-allocation calls. Since feature count across frames keeps varying, stereo-matching and tracking demand variable memory. In such cases, `SSM` reduces dynamic memory allocation/de-allocations calls by performing them only when the requested memory exceeds the current size. Also when CPU-GPU memory is accessed, `SSM` on its own transfers the underlying data to the destination device, reducing the framework's complexity.

IV. Experiments

We evaluate our contributions on a desktop GPU: NVIDIA RTX-2070 (200W) and an embedded device: Jetson-NX (10W). Please see Table III for the GPU specifications.

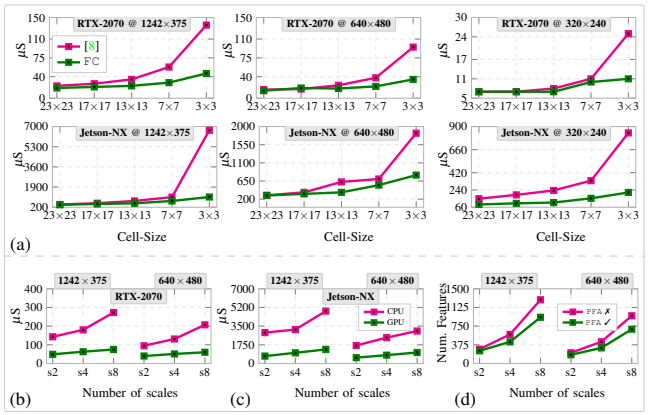

Figure 9: Evaluation of `PyCA`'s components. (a) Runtime of Feature Culling (`FC`) *vs* [8] for different cell-size, resolution and GPUs since smaller cells are a concern in multiscale SLAM (Sec. II-C1c). Our `FC` handles them easily, as notable via its linear profile compared to exponentially growing [8] at smaller cells. (b)-(c) `PFA`'s runtime on CPU and GPU and (d) `PFA`'s effect on features count.

A. Bounded Rectification For Corner Detection (Sec. II-A)

Due to the lack of a labeled corner dataset, we use synthetic data following [19]. We perform rotation transformation as it is the most challenging case [19] to analyze Repeatability (F1-score) of bounded rectification. Fig. 8a shows bounded rectification drastically improves the corner quality while outperforming [15], [17]. Fig. 8b shows a qualitative analysis.

B. Pyramidal Culling and Aggregation (`PyCA`) (Sec. II-C)

We evaluate `PyCA` comprehensively since it governs the front-end speed. However, due to the lack of aligned baselines, we evaluate its feature culling (`FC`), and pyramidal feature aggregation (`PFA`) steps separately.

1) Feature Culling (`FC`): [8] is loosely comparable with `PyCA` but in `FC` mode only. We compare the wall-time (obtained via NVIDIA profiler) for which `FC` and [8] hold the computing resources. It was done for different resolutions, cell sizes, and GPUs since these variables govern the computations required and the available computing power. See Fig. 9a.

Notably, `FC` has a roughly linear timing profile w.r.t the cell size, whereas [8] grows exponentially. The smaller cells are a major concern in the multiscale setting and core scarcity because they consume more time and cause GPU wastage. However, our `FC` can handle them easily, as evident that the runtime of `FC` is drastically lower than [8] in smaller cells, irrespective of the GPU. This advantage is attributed to our resource-efficient `MLPT` and `TEWA` schemes.

2) Pyramidal Feature Aggregation `PFA`: Due to the lack of matching baselines, we evaluate `PFA` by studying its runtime on CPU and GPU, and its effect on the number of features by varying resolution and scales, as claimed in Sec. II-C2.

In the runtime analysis (Fig. 9b, 9c), we observe a huge gap in the CPU and GPU modes regardless of the GPU device, thanks to `PFA`'s parallelizable and sparse-NMS design. While the other analysis (Fig. 9d) shows that `PFA` significantly reduces the number of features regardless of the resolution.

Conclusively, the high-speed feature culling (`FC`) and feature aggregation via `PFA` drastically improves the Jetson-SLAM's speed, even in the multiscale stereo mode on low-powered

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

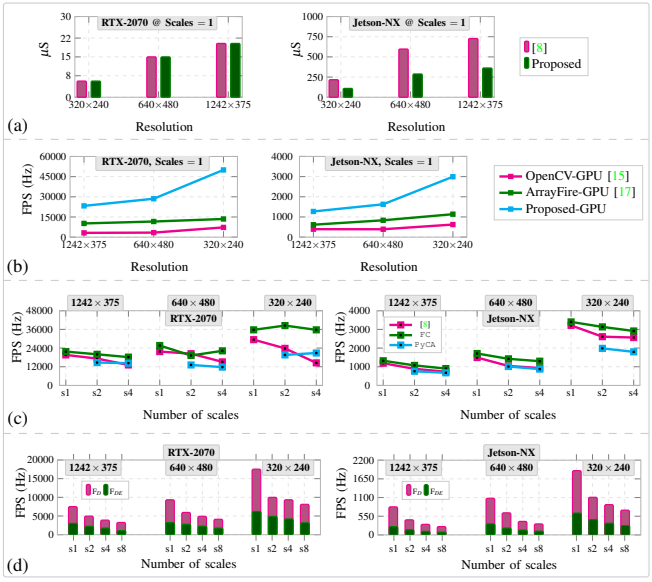

Figure 10: Evaluation of FAST Detection. (a) Performance of our GPU kernel for computing Corner Response Function (CRF) *vs* [8]. (b) The proposed FAST detection *vs* open-source single-scale baselines [15], [17]. (c) multi-scale baseline [8] comparable with `FC` only since [8] does not perform feature aggregation. However, `PyCA` i.e. `FC`+`PFA` is shown for reference. (d) Throughput of our `PyCA` based frontend across resolutions, scales and GPUs. 'F_D': Detection frame rate, 'F_{DE}': Detection-Extraction frame rate.

devices having core scarcity. The smaller number of features lowers the Jetson-SLAM's middle-end and backend runtime by $\sim 2-4$ms even on high-resolution KITTI images, without affecting the SLAM accuracy. This allows timely allocation of the computing resources to other onboard compute-intensive sub-systems, such as deep neural networks.

C. High-Speed FAST Detection & Extraction

1) µ-Second Efficient CRF Computations (Sec. III-1): Computing CRF-matrix is the foremost step in FAST detection. Hence we compare the GPU kernel performance to compute CRF-matrix, enhanced with bounded rectification against [8] (Fig. 10a). On the RTX GPU, similar runtime is observed due to many cores (2304). However, on Jetson-NX with only 384 cores, ours runs 50% faster regardless of the resolution, thanks to the simplified GPU kernel that avoids warp divergence and guarantees coalesced access to image data. This contrasts to [8], which performs additional address extraction steps in the look-up tables, leading to warp divergence and slower speeds in case of GPU core scarcity.

2) High-Speed FAST Detection: We compare frame rate of our `PyCA`-based FAST detection with single-scale [17], [15] and multi-scale [8] baselines (See Fig. 10c, 10b). Notably, `PyCA` is quite faster with predominant gains on Jetson-NX.

3) High-Speed Multiscale Detection-Extraction (Sec. III-2): Since, FAST detection and Extraction are also useful in many applications other than VO or SLAM, such as sparse optical flow or calibration etc., we present the throughput of our FAST detection and ORB extraction modules across resolutions and scales for their use in such applications. The timings include image upload to GPU, image-pyramid, CRF, `PyCA`, ORB extraction, and keypoint download to CPU.

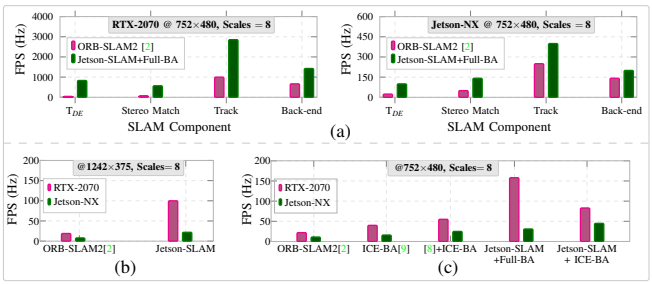

Figure 11: SLAM throughput analysis. (a) Frame-rates of SLAM components on EuRoC [21] resolution. Notice how the frontend efficiency also transfers to the backend. (b) SLAM frame rate at KITTI resolution [20], and (c) SLAM frame rate EuRoC resolution.

Table IV. SLAM Performance on KITTI [20] . '✗': Tracking Failure.

Approach	RMSE ATE (m)		
	KITTI-00	KITTI-01	KITTI-02
• ORB-SLAM [2]	0.70m	1.39m	0.76m
• [8]+Full-BA [2]	✗	✗	✗
• Jetson-SLAM+Full-BA [2]	0.66m	1.96m	0.96m

From Fig. 10d, detection on Jetson-NX at a high resolution of 1242×375 can run at 1000FPS for single scale and 250FPS for eight scales, and the detection-and-extraction can run at 250 FPS for single scale and 80FPS for eight scales. For a smaller resolution of 320×240 (popular in UAV), our method runs at 2000FPS for single scale detection and at 800FPS for eight scales, while for detection-and-extraction, it reaches 250FPS even for eight scales which is huge. It fulfils our goal and claim of using the computing resources for a short duration which is a key requirement in the modern robotic autonomy solutions having many sub-systems [1].

D. SLAM Runtime Analysis of Jetson-SLAM

We evaluate Jetson-SLAM with two backends: Full-BA [2], ICE-BA [9], and three datasets: KITTI [20], EuRoC [21], KAIST-VIO [3]. KITTI dataset is collected via a self-driving test bed, while EuRoC and KAIST-VIO are collected via a UAV flying indoors. These datasets have several events of saturation [20], severely low lighting and low texture [21], and rapid rotations [3], thus sufficient to rigorously analyze a frontend and a SLAM system. Following [2], we report RMSE Absolute Trajectory Error (ATE) and frame rate.

1) Effect of `PyCA` on the Middle-end and the Backend: Fig 11a shows that Jetson-SLAM with Full-BA backend is 85% faster on RTX-2070 GPU, whereas 65% faster on Jetson-NX @752×480 for eight scales and stereo mode, showing major achievement of this work. Interestingly, the backend still uses CPU and has not been altered to run faster, nonetheless, `PyCA` improves the backend's speed, justifying that `PyCA` produces robust features. The overall frame rate is bottlenecked by the backend, especially on Jetson-NX, which opens future possibilities to speed up the backend.

2) Throughput Analysis of Jetson-SLAM over Datasets: We also analyze the SLAM frame rate on high-resolution KITTI and EuRoC images with different backends. See Fig 11b, 11c. Notably, Jetson-SLAM achieves an average speed-up of 80% on RTX-2070 and 67% on Jetson-NX, even at eight scales and stereo mode, which is huge (see video).

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

Table V. SLAM Performance of VO/VIO/SLAM pipelines on EuRoC [21]. boldface denotes Top-3 scores. '✗': Tracking Failure.

Approach	RMSE ATE (m)		
	MH01	MH02	MH03
• VINS-Stereo [22]	0.54m	0.46m	0.33m
• MSCKF-Stereo [23]	0.42m	0.45m	0.23m
• VINS-Stereo + IMU [22]	0.24m	0.18m	0.23m
• ICE-BA [9]	**0.05**m	**0.04**m	0.10m
• [8]+ICE-BA [9]	0.16m	**0.06**m	0.16m
• [8]+Full-BA [9]	✗	✗	✗
• SVO-GTSAM [24]	**0.05**m	**0.03**m	**0.12**m
• KIMERA-VIO-FULL [25]	**0.04**m	0.07m	**0.12**m
• KIMERA-RPGO [25]	0.08m	0.09m	**0.11**m
• Jetson-SLAM+ IMU + ICE-BA [9]	**0.07**m	**0.04**m	**0.07**m
• Jetson-SLAM+ Full-BA [2]	**0.04**m	**0.03**m	**0.03**m

Table VI. SLAM Performance on KAIST-VIO [3] on Jetson-NX. Baselines results are borrowed from [3].

Approach	KAIST-VIO Sequence											
	circle			infinite			square			rotation		
	normal	fast	head	normal	fast	head	normal	fast	head	normal	head	
• VINS-Fusion [22]	0.06m	0.12m	**0.08**m	0.05m	0.09m	0.12m	0.17m	0.07m	0.19m	0.11m	0.28m	
• MSCKF-Stereo [23]	0.12m	0.19m	0.21m	0.32m	0.17m	0.60m	0.10m	0.30m	0.30m	0.10m	0.29m	
• KIMERA-VIO [25]	0.12m	0.07m	0.28m	0.05m	0.14m	1.08m	0.17m	0.19m	1.57m	0.17m	0.74m	
• VINS-Fusion + IMU [22]	0.11m	0.10m	0.13m	0.08m	0.08m	0.12m	0.21m	0.13m	0.20m	0.16m	0.10m	
• VINS-Fusion + GPU [22]	0.09m	0.13m	0.11m	0.09m	0.05m	0.14m	0.12m	0.11m	0.15m	0.12m	0.11m	
• ORB-SLAM2 [2]	0.09m	0.11m	0.13m	0.08m	0.10m	0.12m	0.09m	0.09m	0.16m	0.17m	0.21m	
• Jetson-SLAM	**0.014**m	**0.017**m	0.12m	**0.017**m	**0.016**m	**0.09**m	**0.016**m	**0.017**m	**0.04**m	**0.07**m	**0.09**m	

E. SLAM Metric Performance of Jetson-SLAM

1) KITTI [20] Dataset: See Table IV. Jetson-SLAM performs better in seq. KITTI-00. It has a marginally higher error in other sequences that can be traded for speed, and occurs due to the reduced number of features. This is also evident from trajectories in Fig. 12a where Jetson-SLAM remains close to the ground truth. We also test [8] with Full-BA backend, but observe tracking failures due to its incapability to produce sufficient map points (see video).

2) EuRoC [21] Dataset: We compare Jetson-SLAM with existing VO/VIO/SLAM pipelines in Table V. Jetson-SLAM achieves the lowest error on all sequences with Full-BA backend while remains among Top-3 with ICE-BA backend. This indicates `PyCA` produces reliable features at high speed, turning Jetson-SLAM faster and accurate. Notably, Jetson-SLAM does not fail in any sequence even without an IMU.

We also show the trajectory analysis in Fig. 12b. It indicates a high overlap of Jetson-SLAM with the ground truth, regardless of the backend, validating the small ATE.

3) KAIST-VIO [3] dataset: Table VI shows the analysis on all the 11 sequences of this dataset. Notably, Jetson-SLAM achieves the lowest error except `circle_head`, where it remains among Top-3 with only a marginal ATE difference.

In this dataset, *rapid heading movement* is the most challenging aspect for SLAM systems to handle without using an Inertial-Measurement-Unit (IMU). Nonetheless, Jetson-SLAM, even without using an IMU, outperforms several baselines relying on IMU. The primary reason is the Jetson-SLAM's capability to process each frame quickly.

For reference, we also show trajectories by Jetson-SLAM in Fig. 12c, which closely overlaps with ground truth.

F. SLAM-Focused Ablation Study of Jetson-SLAM

1) Bounded Rectification: It governs the quality of the features reaching the SLAM backend. Therefore, it is crucial to analyze its effect on the SLAM accuracy. See Table VII. Notably, bounded rectification significantly lowers the SLAM error, verifying the robustness of the detected corners.

2) `PyCA` Cell-Size: Table VIII shows that smaller cells at the native scale lead to a large number of features because smaller cells become even smaller at lower resolutions in the multiscale setting. It impacts the middle-end and backend performance due to stereo and feature matching ambiguities, as evident from 15×15 cell having the highest feature count and error. On the contrary, the cell-size 32×32 results in 8×8 cells at the 8^{th} scale with a scale factor of 1.2 but have the lowest feature count, ATE, and runtime. This demonstrates `PyCA`-based frontend yielding fewer but high-quality features.

3) Effect of Number of Scales: Table IX shows that too few scales (i.e. 2) at large cell-size results in insufficient points which leads to SLAM failure. Smaller cell size helps but increases the number of features and hence the runtime. On the contrary, more scales with larger cell-size results in sufficiently fewer key points, lower ATE, and a high frame-processing rate of Jetson-SLAM.

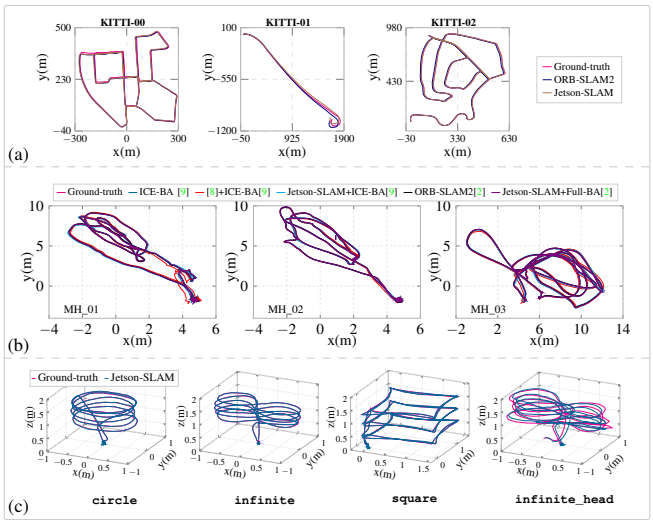

Figure 12: **Zoom in**. Trajectory output on different datasets. (a) KITTI [20], (b) EuRoC [21], and (c) KAIST-VIO [3].

Table VII. Effect of bounded rectification on SLAM performance.

Approach	Dataset	Sequence	RMSE ATE (m)	
			Bounded Rectification	
			✗	✓
Jetson-SLAM +ICE-BA [9]	EuRoC [21]	MH01	0.11m	0.07m
		MH02	0.07m	0.04m
		MH03	0.16m	0.07m
Jetson-SLAM +FULL-BA [2]	KAIST-VIO [3]	`infinite_fast`	0.10m	0.09m

Table VIII. Effect of `PyCA` cell-size on SLAM for challenging `infinite_head` sequence of KAIST-VIO. GPU: RTX-2070.

Cell Size	Average Feature Count	RMSE ATE (m)	FPS (Hz)
15×15	1800	0.20m	52
20×20	1385	0.13m	62
25×25	925	0.10m	90
32×32	**600**	**0.09**m	200

Table IX. Effect of number of scales on SLAM for seq. `infinite_head` [3]. GPU: RTX-2070. '✗': Tracking Failure.

Number of scales	Cell Size	Average Feature Count	RMSE ATE (m)	FPS (Hz)
2	32×32	✗	✗	✗
2	10×10	1200	0.09m	66
4	32×32	✗	✗	✗
4	20×20	800	0.13m	125
6	32×32	700	0.11m	166
8	32×32	**600**	**0.09**m	**200**

G. Jetson-SLAM Resource Utilization Analysis

Jetson-SLAM utilizes roughly 15% CPU, 40% GPU, and 10% RAM on Jetson-NX, which is quite low, owing to `PyCA`, `MLPT`, `TEWA`, and data-sharing. It allows other sub-systems to utilize GPU, such as deep networks, discussed next.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2024, Yokohama, Japan. Cite as RA-L paper.

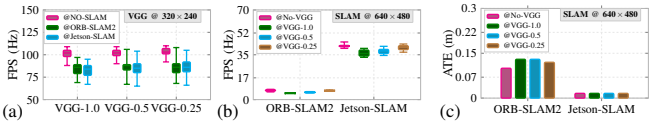

Figure 13: Effect of SLAM on the FPS of a deep network VGG [12] on Jetson-NX, stereo-mode, eight scales @640×480, `infinite_fast` sequence [3], and (b)-(c) Effect of VGG onto the FPS and accuracy of ORB-SLAM2 [2] *vs* Jetson-SLAM.

H. Jetson-SLAM Co-existing with Deep Neural Networks

Modern autonomy requires deep networks co-existing with other sub-systems. We show that Jetson-SLAM marginally affects the runtime of a deep network without sacrificing its own runtime and accuracy, a major achievement of our work.

To verify that, we choose VGG [12] deep network, popular in robotics applications, and construct its three variants: VGG-1.0, VGG-0.5, and VGG-0.25. The variant VGG-1.0 has five stages with $\{2,2,2,4,4\}$ layers, and $\{32,64,128,256,256\}$ channels, while the others are scaled w.r.t the 1.0 variant.

Fig. 13a depicts the effect of ORB-SLAM2 and Jetson-SLAM onto the frame rate of VGG. Despite using GPU, Jetson-SLAM incurs a frame-rate drop similar to the CPU only ORB-SLAM2, indicating GPU efficiency of Jetson-SLAM. This experiment was conducted at a resolution of 640×480 which is quite high, thus for smaller resolutions, Jetson-SLAM will result in negligible frame-rate drop.

Fig. 13b, 13c shows how VGG affects SLAM performance. Interestingly, ORB-SLAM2 faces a drop in frame rate which is already running below 10FPS and faces higher ATE error due to its failure to process the frames in time. On the contrary, Jetson-SLAM does not face a drop in ATE but has a minimal drop in frame rate that is still well-above real-time (30FPS).

This experiment shows the utility of Jetson-SLAM to develop complex UAV autonomy solutions. We use Jetson-SLAM in a UAV @432×240 with VGG-1.0, and do not observe FPS drop in VGG or Jetson-SLAM.

V. Conclusion & Future Work

We present a resource-efficient and accurate GPU-accelerated Jetson-SLAM for low-powered computing devices. We proposed *Bounded Rectification* to prevent non-corners from being classified as corners in the FAST corner detection process, and *Pyramidal Culling and Aggregation* `PyCA` which yields high-quality features at very high speeds in multiscale and stereo setting. `PyCA` is based on our Feature Culling (`FC`), Pyramidal Feature Aggregation (`PFA`), Multi-Location Per-Thread (`MLPT`) culling, and Thread Efficient Warp Allocation (`TEWA`) techniques. We also design *Middle-end* in visual SLAM and develop a Jetson-SLAM library that utilizes synchronized shared memory to achieve resource efficiency. Jetson-SLAM exhibits a very high frame rate, suitable for modern autonomous robotic systems having several sub-systems. Jetson-SLAM outperforms many prominent SLAM pipelines by a large margin even in multi-scale and stereo settings on Jetson devices.

References

[1] A. Kumar, M. Vohra, R. Prakash, and L. Behera, "Towards deep learning assisted autonomous uavs for manipulation tasks in gps-denied environments," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1613–1620, IEEE, 2020.

[2] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and rgb-d cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[3] J. Jeon, S. Jung, E. Lee, D. Choi, and H. Myung, "Run your visual-inertial odometry on nvidia jetson: Benchmark tests on a micro aerial vehicle," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5332–5339, 2021.

[4] A. J. B. Ali, M. Kouroshli, S. Semenova, Z. S. Hashemifar, S. Y. Ko, and K. Dantu, "Edge-SLAM: Edge-assisted visual simultaneous localization and mapping," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 1, pp. 1–31, 2022.

[5] C. Forster, Z. Zhang, M. Gassner, M. Werlberger, and D. Scaramuzza, "SVO: Semidirect visual odometry for monocular and multicamera systems," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 249–265, 2016.

[6] L. Von Stumberg and D. Cremers, "DM-VIO: Delayed marginalization visual-inertial odometry," *IEEE Robotics and Automation Letters*, 2022.

[7] H. Rebecq, G. G. Bonet, and D. Scaramuzza, "Simultaneous localization and mapping with an event camera," 2021. US Patent 11,151,739.

[8] B. Nagy, P. Foehn, and D. Scaramuzza, "Faster Than FAST: GPU-accelerated frontend for high-speed VIO," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4361–4368, IEEE, 2020.

[9] H. Liu, M. Chen, G. Zhang, H. Bao, and Y. Bao, "ICE-BA: Incremental, consistent and efficient bundle adjustment for visual-inertial SLAM," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1974–1982, 2018.

[10] SLAMCore in *https://www.slamcore.com/*.

[11] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 2, pp. 1508–1515, Ieee, 2005.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[13] CUDA in *https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*.

[14] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[15] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[16] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE transactions on robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[17] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, "Arrayfire-a high performance software library for parallel computing with an easy-to-use API," *AccelerEyes, Atlanta*, vol. 106, 2015.

[18] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[19] Y. Zhang, B. Zhong, and X. Sun, "A benchmark for the evaluation of corner detectors," *Applied Sciences*, vol. 12, no. 23, p. 11984, 2022.

[20] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE conference on computer vision and pattern recognition*, pp. 3354–3361, IEEE, 2012.

[21] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The euroc micro aerial vehicle datasets," *The International Journal of Robotics Research*, vol. 35, no. 10, pp. 1157–1163, 2016.

[22] T. Qin, J. Pan, S. Cao, and S. Shen, "A general optimization-based framework for local odometry estimation with multiple sensors," *arXiv preprint arXiv:1901.03638*, 2019.

[23] K. Sun, K. Mohta, B. Pfrommer, M. Watterson, S. Liu, Y. Mulgaonkar, C. J. Taylor, and V. Kumar, "Robust stereo visual inertial odometry for fast autonomous flight," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 965–972, 2018.

[24] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "IMU preintegration on manifold for efficient visual-inertial maximum-a-posteriori estimation," Georgia Institute of Technology, 2015.

[25] A. Rosinol, M. Abate, Y. Chang, and L. Carlone, "Kimera: an open-source library for real-time metric-semantic localization and mapping," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1689–1696, IEEE, 2020.