Visual Inertial Odometry At the Edge: A Hardware-Software Co-design Approach for Ultra-low Latency and Power

Dipan Kumar Mandal[1], Srivatsava Jandhyala[1], Om J Omer[1], Gurpreet S Kalsi[1], Biji George[1], Gopi Neela[1], Santhosh Kumar Rethinagiri[1], Sreenivas Subramoney[1], Lance Hacking[2] , Jim Radford[2], Eagle Jones[2], Belliappa Kuttanna[2] and Hong Wang[1]

[1]Microarchitecture Research Lab, Intel Labs; [2] Emerging Growth Initiatives Group, Intel Corporation

***Abstract* — *Visual Inertial Odometry* (VIO) is used for estimating pose and trajectory of a system and is a foundational requirement in many emerging applications like AR/VR, autonomous navigation in cars, drones and robots. In this paper, we analyze key compute bottlenecks in VIO and present a highly optimized VIO accelerator based on a hardware-software co-design approach. We detail a set of novel micro-architectural techniques that optimize compute, data movement, bandwidth and dynamic power to make it possible to deliver high quality of VIO at ultra-low latency and power required for budget constrained edge devices. By offloading the computation of the critical linear algebra algorithms from the CPU, the accelerator enables high sample rate IMU usage in VIO processing while acceleration of image processing pipe increases precision, robustness and reduces IMU induced drift in final pose estimate. The proposed accelerator requires a small silicon footprint (1.3 mm^2 in a 28nm process at 600 MHz), utilizes a modest on-chip shared SRAM (560KB) and achieves 10x speedup over a software-only implementation in terms of image sample-based pose update latency while consuming just 2.2 mW power. In a FPGA implementation, using the EuRoC VIO dataset (VGA 30fps images and 100Hz IMU) the accelerator design achieves pose estimation accuracy (loop closure error) comparable to a software based VIO implementation.**

I. Introduction

Visual Inertial Odometry (VIO) is a well-known method for estimating the 3D pose and trajectory of an observer. VIO promises to be the most effective and accurate pose estimation solution for edge applications that cannot afford the form factor, power and cost budget of Lidar based 3D sensing and mapping, or those deployed in GPS-denied environments such as indoors or underwater. Drones and mobile robots traditionally use the pose and trajectory information to autonomously navigate through unknown environments. In a VR head mounted display (VR HMD), the estimated pose of the user's head is used as the starting point of the virtual reality scene rendering process.

More recently, many emerging applications like untethered lightweight AR/VR HMDs and pico drones are being deployed in energy-constrained (e.g. battery operated), thermally restrained (e.g. fan-less small enclosed form factor), weight sensitive, low-cost embedded platforms with limited compute budget. This introduces unforeseen challenges in implementing VIO capability without sacrificing pose estimate precision, latency and real-time throughput guarantees. In this work we present a hardware accelerator architecture that, along with a low power host micro-controller, accelerates all key compute intensive parts of a typical VIO algorithm, producing pose output at an extremely low latency but within the power and area budget affordable in deeply embedded SoC based edge applications.

II. Background and Motivation

Visual Inertial Odometry (VIO) estimates 3D pose (translation and orientation in six degrees-of-freedom a.k.a. 6DoF pose) of the observer using motion measurements from images of camera(s) and linear acceleration, angular velocity from Inertial Measurement Units (IMUs). In a typical VIO processing pipeline [1], noisy motion measurements from camera and IMU frontend are used in a maximum likelihood (ML) estimator backend to estimate the system dynamics or the state. This involves two complementary state estimation and update paths within a VIO system – (1) the *IMU update path,* operating at IMU sampling frequency, enables higher frequency state updates at lower compute cost; (2) the *vision update path,* operating at camera frame rate, provides accurate state updates and compensates for the drift in IMU measurements due to its inherent temperature dependent noise.

In particular, VIO methods like [2] with an Extended Kalman Filter (EKF, [3]) backend take a causal approach to state update, making the best pose estimate available immediately as each sensory input (image frame, or IMU sample) is processed, without waiting for a correction via local or global optimization based on past estimates. This helps lower pose update latency which is of prime importance to AR/VR HMD applications where estimation of 6DoF pose of the HMD is the first-order contributory component in the overall motion-to-photon latency which must be less than 20ms for motion sickness free immersive experience [4].

Processing time, CPU load, memory usage and output pose estimation accuracy of some popular and publicly available VIO methods are presented in [5]. [5] notes that in order to fit in the computational budget of the underlying hardware architecture, many of the methods need to be scaled down to a reduced set of algorithm parameters e.g. number of feature points tracked in the vision frontend, size of EKF state matrices etc. In real applications using low-power compute-limited hardware, this leads to either reduced precision and robustness of pose output or constraining of other essential compute needs of the application such as flight control in pico drones.

Some recent works have proposed real time implementation of Visual or Visual-Inertial Odometry using embedded architectures, primarily leveraging FPGAs. Visual Odometry system for autonomous rovers on the Mars are presented in [6] focusing primarily on FPGA-based hardware acceleration of image processing tasks. A stereo-based VIO system in [7] accelerates primarily the stereo matching part in hardware leaving other parts of the system e.g. EKF filter, to be run on the x86 host processor. Reference [8] has instead focused on accelerating the EKF filtering part on FPGA for meeting VIO performance need. A fully software based VIO implementation in small drones has been reported in [9] using

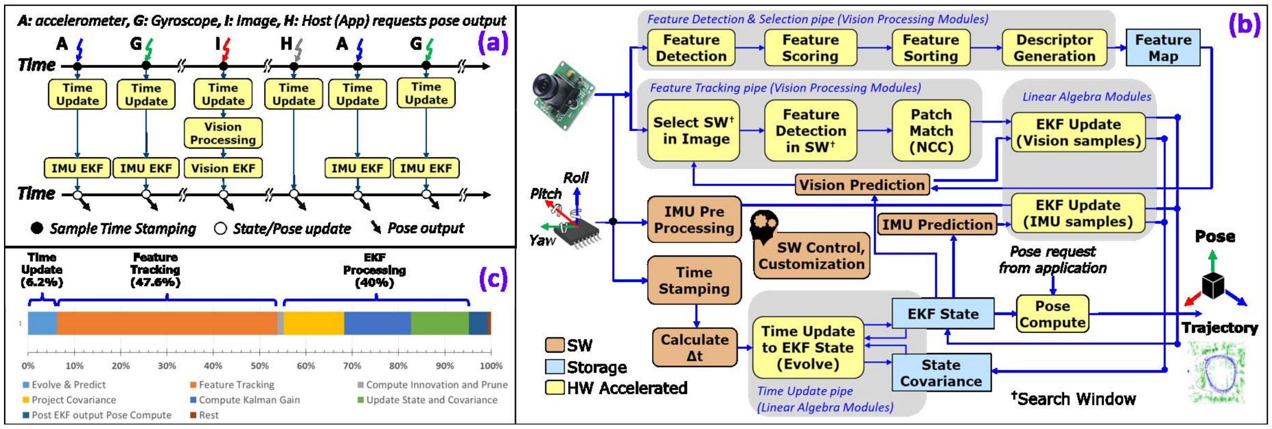

Figure 1: EKF based VIO System overview
(a) Sensor sample chronology (b) processing flow diagram and acceleration candidate mapping (c) execution profile of vision update path

Qualcomm Snapdragon platform without any effort towards acceleration. Reference [10] has reported a more complete focus on accelerating all key parts of the VIO processing. However, it relies on stereo based methods, incorporates IMU pre-integration to reduce IMU processing load and focusses on a keyframe based pose graph based iterative optimization technique.

In comparison, in our work, we focus on accelerating EKF based monocular VIO (that helps lowering system power, cost) with capacity to support direct processing of all IMU samples without pre-integration thus improving latency and accuracy of instantaneous pose updates, and ability to accelerate EKF processing with large number of tracked features and state size. We make accelerator architecture easy to integrate with a host CPU or micro-controller enabling software-based configuration, control and retain programmability for use-case specific adaptation. In particular, our contributions are:

- Analysis and identification of computational bottlenecks in typical EKF based VIO algorithm
- A fully on-the-fly (OTF) frontend accelerating all parts of image processing pipeline for VIO, including supporting multiple simultaneous cameras in OTF mode
- A flexible, scalable and power-efficient microarchitecture to accelerate all parts of the EKF pipeline
- Achieving, to the best of our knowledge, best-in-class processing latency (0.5 msec for vision update path), power dissipation (2.2 mW with VGA 30fps images, 100 Hz IMU) and area foot print (1.3 mm^2 in a 28nm process at 600 MHz).

III. VIO Algorithm Analysis

Figure 1 illustrates a representative processing pipeline of an EKF based VIO system. Arrival of each sensor sample (accelerometer, gyroscope samples from IMU and images from camera) initiates separate processing threads (**Figure-1a**), asynchronous to each other, to implement time update (a.k.a. the evolve step), preprocessing and EKF processing steps.

IMU preprocessing may include IMU pre-integration, innovation and state Jacobian computations. Image sample preprocessing is much more involved and as shown in **Figure-1b**, consists of two processing pipelines: (1) *Feature Detection and Selection pipeline* which detects features in the image, scores them as per their strength and sorts them to select few strongest feature points (usually having maximum repeatability) as part of the feature map of the scene; (2) *Feature Tracking pipeline* which uses the expected image feature locations (from vision prediction step) to select a search window for each feature point being tracked since the last frame and establishes the best matching position of the feature points (a.k.a. feature tracks or image observations).

Innovation (difference of predicted and observed values) of IMU and image measurements, the state, covariance matrices and innovation covariance are then used in the EKF update step to modify EKF state and covariance matrices (a.k.a. *the EKF Update pipeline*). When pose estimation is needed (e.g. on request from a host for use in an HMD rendering pipeline), the EKF state is updated to current time (time update) and output pose is computed from the latest EKF state.

Figure-1c shows an execution time profile of the whole vision update path on an ARM Cortex A9 processor (single core) at 1.2 GHz. Clearly, almost 94% of the processing time is spent on image feature tracking, EKF update and time update making them the prime target for acceleration and offload. Similar profile of the IMU update path shows time update and EKF update takes 96% of processing time making it possible to offload IMU update path with the same accelerator cores. Feature detection and selection pipe, even if called for only selected frames, takes up to 8-10 milli-sec depending on the scene content and thus is also a prime candidate for offload and acceleration. The chosen acceleration candidates are illustrated in **Figure-1b** overlaid on the processing flow.

IV. The Proposed Hardware Architecture

Figure 2 shows our proposed hardware architecture, denoted as VIO HWA, integrated in a typical embedded vision SoC (e.g. Intel Movidius Myriad® X) while **Figure 3** shows an internal block diagram of VIO HWA.

We partition the identified acceleration candidates into the VIO HWA as specialized modules interfacing an on-chip **Shared L2 SRAM** (**SL2**) sub-system. All module micro-architectures are made tolerant to the data fetch latency using intelligent data pre-fetch techniques leveraging decoupling FIFOs to hide latency of data access to SL2. The micro-controller host may setup and kick-off processing tasks by programming the memory mapped control registers and monitor the task by polling status registers or via task completion interrupts.

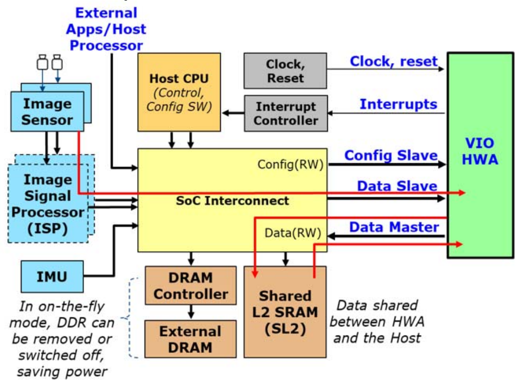

Figure 2: VIO HWA integration in an embedded SoC

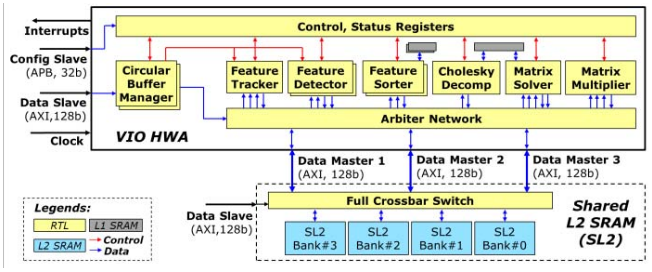

Figure 3: VIO HWA Block diagram

A. *Image Processing Modules*

The **Feature Detection** module implements FAST9 feature detection (with 16 pix per cycle throughput) using a novel *2D sliding window* pixel buffer. It feeds 16 parallel FAST9 corner detection datapath instances with their own 7x7 pixel windows in single cycle (i.e. approximately 470 GB/sec BW) by reusing fetched pixels in horizontal and vertical directions. Pixel reuse reduces SL2 pixel bandwidth (and hence memory access power) by 7x as compared to a CPU based software implementation. It also computes feature strength (i.e. the maximum threshold for which the feature point remains a FAST9 feature) using a binary search technique in hardware, reusing the same corner detection logic, in just 8 cycles. Thus, in most of the image regions we achieve a 16-input pixel per cycle throughput and when a corner is detected (much lower in number than image size) we get a throughput of 16 input pixels every 8 clock cycles.

The **Feature Sorter** module sorts the detected feature points w.r.t. to their strength and returns a list of configurable N-number of strongest features. Note that the detected feature points from the feature detector form a list of (key, value) pairs (x,y coordinate as value and strength as the key) and the sorting operation needs to sort the key-value pair - which is more complicated than straight forward sorting of a list of numbers. The Sorter employs histogram-based sorting technique. The histogram is built in parallel with feature detection and then the saved feature list is read again for comparison and selection is done based on the cumulative histogram score. This novel technique helps in getting a list of strongest feature points concurrent with feature detection, thereby reducing overall latency.

The **Feature Tracker** does prediction-based tracking. The predicted positions of feature points are calculated using the newest EKF state to define the search center of a (configurable) search window within which the best matching image patch center is computed as the tracking result, using the Normalized Cross Correlation (NCC) score over the 7x7 image patch (used as descriptors). Tracking latency directly contributes to vision path update latency. To reduce tracking latency, Feature Tracker (1) uses the high throughput FAST9 corner point selection logic used in Feature Detector to detect 'candidate FAST9 corner points' within the search window to prune down the number of NCC evaluations; (2) accelerates NCC based patch match by a novel approximate, fully integer based datapath. For NCC computation, we employ an unbiased rounding method on all intermediate subtraction, multiplication and accumulation of integer pixel values so as to maintain a fully integer-based processing pipeline for the whole NCC computation. Instead of doing the final square root, we use the ratio of the final squared terms (quantized to a 10b) as proxy for NCC score. This optimization, as per our exhaustive experiment over millions of real life images, shows no adverse impact on the ability to find the best patch match but enables a fully-pipelined NCC computation logic, 11x smaller, with only 16 cycle latency and 3 cycles per NCC throughput.

The image processing frontend (feature detection, sorting, pruning and tracking) supports **On-the-fly (OTF) image processing** (illustrated with red line in **Figure 2**) wherein the image data from sensor is directly fed to VIO HWA (without first being saved off to an external DRAM) and is maintained in a hardware managed circular buffer (using *Circular Buffer Manager*) in SL2. This technique saves system bandwidth to external DDR without requiring large (full frame) buffers in the on-chip SRAMs enabling large savings in dynamic power in the system.

B. *EKF processing modules*

Cholesky Decomposition (as in the *LAPACK/dpotrf* API, decompose matrix **A** as $\mathbf{LL^T}$ where **L** is lower triangular) is the first step in Kalman Gain computation in the EKF processing pipeline. The key challenge in accelerating this compute is to effectively parallelize compute (primarily dot-products) in spite of data dependency from serial compute (including divide and square-root of diagonal elements). We map the high bandwidth *parallel compute* (dot product via *MAC units*) onto multiple hardware threads enabling high-throughput element compute while precisely controlling data dependency using a high latency *serial compute logic block* and a *dependency clear* module. We adopt data fetch logic to do in-place computation saving 50% on-chip SRAM footprint. We support either linear or compressed triangular storage for the input and output data buffer(s), saving an additional 50% on-chip SRAM requirement. Furthermore, we use parallel element compute across multiple columns that reduces SL2 access bandwidth resulting in lower dynamic power. Overall, this semi-parallel compute micro-architecture achieves almost linear, 1.96x performance scaling using 2 MAC units.

The **Matrix Solver** uses Cholesky output to solve for (as in the *LAPACK/dpotrs* API, solving for **X** in **AX=B** where **A** is positive definitive and $\mathbf{A=LL^T}$) the Kalman Gain matrix in EKF processing pipeline. The forward substitution (**LY=B**, solve for **Y**) and backward substitution ($\mathbf{L^TX=Y}$, solve for **x**) steps are implemented reusing the same datapath but intelligently fetching appropriate matrix elements. We note that mathematically, to compute elements of **X** (say, $\mathbf{X}pq$ where p, q are row and column IDs), we need all the elements of q-th column that are computed before $\mathbf{X}pq$. We use *parallel compute block(s)* to handle the "within column" parallel execution. We employ a local SRAM buffer (reused across Cholesky and Matrix Solver) to store intermediate **X** values in column order reducing SL2 accesses and dynamic power. We also choose to make Cholesky block write out inverse of diagonal elements (i.e. $1/\mathbf{L}qq$) to SL2 such that Matrix Solver can directly use them as multiplicand (i.e. replace an expensive division operation by a cheaper multiplication). We use 4 parallel instances of such *Column Compute Block* (processing 4 columns of **X** simultaneously) achieving 4x increase in performance and 4x reduction of SL2 bandwidth and power.

Matrix Multiplier uses the Kalman Gain found by the Matrix Solver to update the EKF state and state covariance matrices (as in the BLAS/sgemm API, compute $\mathbf{C}_{out} = \alpha^*\mathbf{A}^*\mathbf{B} + \beta^*\mathbf{C}_{in}$). We aggressively leverage the inherent parallelism in Matrix-matrix multiplication operation by implementing a 2D compute element array (of 32 pipelined multiply-accumulate blocks) supported by efficient operand pre-fetch and caching. A programmable sequence logic breaks the input matrices into tiles, rightly sized to match compute capacity (32 SP MAC per cycle), and fetches them into operand specific caches for subsequent dispatch to the compute array. We keep the accumulator buffers near the 2D MAC array to save routing congestion and power dissipation. The operand caches support in-place transpose of the input matrix operand saving data movement and power. The data fetch interfaces supports compact triangular storage reducing SL2 footprint. The tightly-coupled hardware sequencer based title scheduling, aggressive operand pre-fetch and caching results in 99.8% utilization of all MAC blocks improving the processing efficiency in terms of performance per mm^2 per watt.

V. Results

We validated the proposed architecture on an Arria-10 FPGA (10AS066N3) for functionality, performance and pose estimation accuracy. Operating at 70 MHz in FPGA, for EuRoC VIO dataset (VGA 30fps images and 100Hz IMU), VIO HWA achieves very similar loop closure error compared to the reference software implementation on ARM Cortex-A9 (A9) processor at 1.2 GHz (**Table 1**).

We implemented VIO HWA in a 28nm process technology, with full timing closure to achieve a 600MHz silicon operating frequency and a bounding box area of 1.3mm2 with about 72% utilization of standard logic cells. VIO HWA running at 600 MHz, achieves 7-133 times speedup on vision path update for a typical average complexity image frame as shown for individual tasks accomplished by corresponding modules in **Table 2**. When combined with the intermediate small software control overheads, this results in about 0.5 milli-sec latency for the entire vision path update – an impressive 10x reduction compared to software on Cortex-A9 at 1.2 GHz. **Table 2** also shows dynamic power consumption by individual modules (tasks). When scaled to 30fps frame rate, due to tiny active duty cycle, we consume only about 2.2 mW of total average power (1.4 mW in VIO HWA and 0.8 mW in the SL2 sub-system) for a full vision path update.

Area, power, performance data is measured with VIO HWA configured to support detection of up to 8192 features while selecting the 256 strongest ones amongst them. The individual module-level performance and power is obtained for VGA 30fps images, tracking 128 feature points, doing Cholesky decomposition of 128x128 matrices, Matrix Solve for 128x100 matrices and matrix multiplication (SGEMM) of 128x100x128 matrices.

EuRoC Dataset Sequence Name	Number of Frames	Loop Closure Error (in cm)		
		Ground Truth	A9	VIO
MH_04_difficult	44704	26	89	85
V1_01_easy	64064	43	70	80
V1_02_medium	37621	01	92	101
V1_03_difficult	47298	39	124	128

Table 1: Accuracy comparison on EuRoC dataset

Module	Execution Time (in micro-sec)		Execution Speedup (times)	Dynamic power (in milli-Watt)
	A9	VIO		28nm, 1V, 25C, 600MHz
Cholesky	621	80.5	7.7	62.1
Mat Solver	1200	79.2	15.1	88.3
Mat Multiplier	828	31.9	25.9	127.3
Tracker	2637	48.5	54.4	31.9
Detector	11241	84.0	133.7	22.7
Sorter	104	01.4	74.3	6.55

Table 2: Module (task) level speedup and power reduction

VI. Conclusion

We proposed a novel hardware architecture for VIO that demonstrates best-in-class pose output latency and area, power efficiency published till date. This, we believe, opens up possibilities of new applications in low power embedded platforms deployed at the edge.

References

[1] M. Li and A. I. Mourikis, "High-precision, consistent ekf-based visualinertial odometry," The Int. J. of Robotics Ressearch., 2013

[2] E. S. Jones et al, "Visual-inertial navigation, mapping and localization: A scalable real-time causal approach," Int. J. Robotics Research., 2011

[3] R. E. Kalman, "A new approach to linear filtering and prediction problems," Journal of basic Engineering, vol. 82, 1960

[4] R. Kijima and K. Miyajima, "Measurement of head mounted display's latency in rotation and side effect caused by lag compensation by simultaneous observation; an example result using oculus rift dk2," IEEE Virtual Reality (VR), 2016

[5] J. Delmerico and D. Scaramuzza, "A benchmark comparison of monocular visual-inertial odometry algorithms for flying robots," Memory, 2018

[6] G. Lentaris et al., "Hw/sw codesign and fpga acceleration of visual odometry algorithms for rover navigation on mars," IEEE Transactions on Circuits and Systems for Video Technology, 2016

[7] K. Schmid and H. Hirschmuller, "Stereo vision and imu based real-time ego-motion and depth image computation on a handheld device," IEEE International Conference on Robotics and Automation (ICRA), 2013

[8] D. T. Tertei, J. Piat, and M. Devy, "Fpga design of ekf block accelerator for 3d visual slam," Computers and Electrical Engineering, 2016

[9] G. Loianno et al., "Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu," IEEE Robotics and Automation Letters, 2017

[10] Z. Zhang, A. Suleiman, L. Carlone, V. Sze, and S. Karaman, "Visualinertial odometry on chip: An algorithm-and-hardware co-design approach," Robotics: Science and Systems, 2017