2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Madrid, Spain, October 1-5, 2018

π-SoC: Heterogeneous SoC Architecture for Visual Inertial SLAM Applications

Jie Tang[1], Bo Yu[2], Shaoshan Liu[2], Zhe Zhang[2], Weikang Fang[3], Yanjun Zhang[3]

[1]South China University of Technology, China
[2]PerceptIn Inc, USA
[3]Beijing Institute of Technology, China

Abstract — **In recent years, we have observed a clear trend in the rapid rise of autonomous vehicles and robotics. One of the core technologies enabling these applications, Simultaneous Localization And Mapping (SLAM), imposes two main challenges: first, these workloads are computationally intensive and they often have real-time requirements; second, these workloads run on battery-powered mobile devices with limited energy budget. Hence, performance should be improved while simultaneously reducing energy consumption, two rather contradicting goals by conventional wisdom. Previous attempts to optimize SLAM performance and energy efficiency usually involve optimizing one function and fail to approach the problem systematically. In this paper, we first study the characteristics of visual inertial SLAM workloads on existing heterogeneous SoCs. Then based on the initial findings, we propose π-SoC, a heterogeneous SoC design that systematically optimize the IO interface, the memory hierarchy, as well as the the hardware accelerator. We implemented this system on a Xilinx Zynq UltraScale MPSoC and was able to deliver over 60 FPS performance with average power less than 5 W.**

I. INTRODUCTION

The rise of Simultaneous Localization And Mapping (SLAM) applications impose two main challenges on the computing systems: first, these workloads have complex pipelines and are computation intensive and they often have tight real-time requirements. For example, in SLAM, sensor data can rush in at a rate as high as 1 KHz, meaning that the computation pipeline needs to process sensor data to produce 1,000 position data in a second, it also means that the longest stage of the pipeline cannot take more than 1 ms to process. In addition, the samples form a time series such that the samples cannot be processed in parallel. Second, these workloads could run on battery-powered mobile devices with extremely limited energy budget.

In this paper, we make the following contributions: first we conduct a thorough study to understand the characteristics of a visual inertial SLAM system on an existing heterogeneous SoC. Then based on our initial findings, we propose π-SoC, a heterogeneous SoC architecture to systematically improve the system's performance and energy efficiency. In addition, we implement the design on a Xilinx Zynq UltraScale MPSoC to verify its effectiveness.

The rest of the paper is organized as follows: section II provides background information on visual inertial SLAM. Section III provides a thorough study on the software implementation and optimization of the visual inertial SLAM system on an existing heterogeneous mobile SoC. Based on the interesting observations derived from section III, section IV proposes π-SoC, a heterogeneous SoC architecture to systematically optimize the IO sub-system, the memory sub-system, as well as accelerator integration. Section V presents a real-world implementation of the π-SoC design on FPGA and demonstrates its advantages in performance as well energy efficiency.

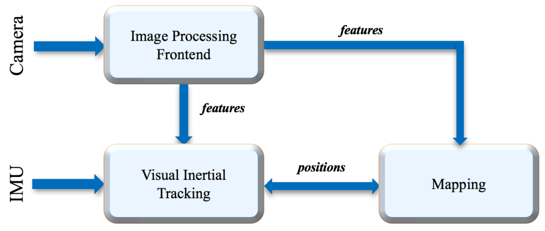


Fig. 1. Simplified Visual Inertial SLAM Pipeline

II. VISUAL INERTIAL SLAM

SLAM is the problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it [1]. It has been an active topic for many years [6] as it provides two fundamental components for many applications: where I am, and what I see. As shown in Figure 1, we have developed a visual-inertial SLAM system, PIRVS, with three major components: 1) highly optimized image processing front-end extracting image features and matching them with the 3D map; 2) EKF-based tightly-coupled visual-inertial tracking with IMU propagation and state update with 3D-2D feature correspondences; and 3) mapping with prior poses from the tracking thread. For more details regarding PIRVS and how PIRVS compares with other existing SLAM systems, please refer to [2]. In this section, we provide a summary of the PIRVS design.

A. Image Processing Front-End

The image processing frontend detects corner features spreading across the entire image. Each time an image arrives, we perform histogram equalization to adjust contrast to allow more features being detected to handle environments with low illumination. Then, we perform feature extraction using ORB [3] and apply spatial binning to ensure that the features are spread uniformly across the entire field-of-view. Once the features are detected, we establish the association of each feature to a 3D map point in the map. A 2D feature and a 3D map point is a match if the re-projection error given the current tracking pose is within a threshold. The match is ranked by the distance between the descriptors of the 2D feature and the 3D map point. The smaller the distance, the better the match is.

B. Tightly Coupled Visual-Inertial Tracking

Next we use the iterated Extended Kalman Filter (EKF) framework for tracking. Our filter state at time t is parameterized as $X_t = [q_G^{B_t}\ p_{B_t}^G\ v_{B_t}^G\ b_a\ b_g]$, where $q_G^{B_t}$ is the unit quaternion describing the rotation from the global frame $\{G\}$ to the IMU body frame $\{B\}$, $p_{B_t}^G$ and $v_{B_t}^G$ are the IMU position and velocity with respect to $\{G\}$, and finally b_a and b_g are gyroscope and accelerometer biases. Whenever a new IMU measurement (angular velocity w_t and linear acceleration a_t) is received, we propagate the state by the kinematic equations as follows:

$$\dot{R}_G^{B_t} = \left(R_{B_t}^G\right)^T = \left(R_{B_{t-1}}^G R_{B_t}^{B_{t-1}}\right)^T \quad (1)$$

$$R_{B_t}^{B_{t-1}} = f\left(\left(w_t - b_g\right)\Delta t\right) \quad (2)$$

$$\Delta v_{B_t} = \left(R_{B_t}^G\left(a_t - b_a\right) + g^G\right)\Delta t \quad (3)$$

$$\dot{v}_{B_t} = v_{B_{t-1}} + \Delta v_{B_t} \quad (4)$$

$$\dot{p}_{B_t} = p_{B_{t-1}} + v_{B_{t-1}}\Delta t + \frac{1}{2}\Delta v_{B_t}\Delta t \quad (5)$$

where $f(\cdot)$ is the function converting axis-angle rotation to rotation matrix, Δv_{B_t} is the velocity increment, and g^G represents the gravity correction in the global coordinate system. EKF update happens when measurements (a set of correspondences between 2D image features and 3D map points) from an image is received. Residual for each correspondence is the re-projection error of each 3D map point P^G in the global coordinate,

$$\begin{bmatrix} u \\ v \end{bmatrix} = \Pi\left(R_B^{C_i}\left(\dot{R}_G^{B_t}\left(P^G - \dot{p}_{B_t}^G\right)\right) + p_B^{C_i}\right) \quad (6)$$

where $(R_B^{C_i}, p_B^{C_i})$ is the transform from the IMU to the coordinate of the i-th camera obtained through offline calibration, and $R_G^{B_t}$ is the rotation matrix of the quaternion $q_G^{B_t}$. Π is the perspective projection of a 3D point to the image coordinate $\begin{bmatrix} u \\ v \end{bmatrix}$. The actual EKF residual is constructed by concatenating residuals from all the measurements.

C. Mapping

The last stage is mapping: we maintain a 3D sparse map in the global coordinate parallel to our visual-inertial tracking. The map consists of a set of 3D map points and a set of keyframes with constraints between the 2D observed features from the stereo camera and the 3D map points. A keyframe is inserted when the device sees a portion of the environment that is not covered by the map. Upon inserting a new keyframe, new map points are created from both spatial and temporal correspondences from 2D features in the keyframes that are not yet being associated to the 3D map. Temporal correspondences are obtained through our image-processing front-end, and spatial correspondences are established by matching features from the stereo images. Note, with our accurately calibrated stereo camera, we can efficiently search for spatial correspondences using epipolar constraint. Finally, we triangulate each set of corresponding 2D features into a 3D map point. Bundle adjustment [4] is applied to refine the 6-DoF pose of the keyframes and the 3D locations of the map points.

III. Implementation on Heterogeneous SoC

In the previous section we have introduced the algorithm design of PIRVS. In this section, we implement and optimize PIRVS on an existing mobile heterogeneous SoC to evaluate its performance and power consumption. Especially we try to understand whether software optimization can enable PIRVS to deliver reliable performance (preferable > 30 FPS) with stringent energy constraints.

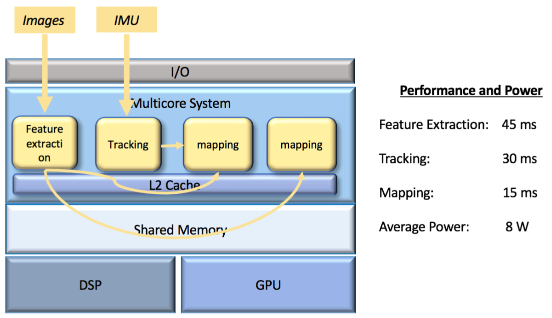
In our experiments, we implemented our SLAM system on a ARM v8 mobile SoC [5]. The SoC consists of a four-core CPU, running at 2 GHz, the four cores share a L2 cache. Besides the CPU, the SoC consists of a DSP, and a GPU, and the CPU communicates with the DSP and the GPU through shared memory. Note that the I/O subsystem is directly connected to the CPU. If acceleration is needed, the CPU would then dispatch workloads to DSP and GPU through shared memory.

Before implementing the SLAM system onto this mobile SoC, we first tried to understand the power consumption behavior of each computing unit. We implemented several stress tests to stress CPU, DSP, and GPU respectively: at its peak, each CPU consumes about 2.5 W, DSP consumes about 1.5 W, and GPU consumes about 2.3 W. This implies that if a DSP delivers similar performance compared to a CPU core, it would be more energy-efficient to use DSP.

A. Software Implementation on CPU

As a first step of implementing the SLAM system on mobile SoC, we implemented a straightforward CPU-only version. As shown in Figure 2, we implemented the feature extraction thread on core 0, the tracking thread on core 1, the mapping threads on cores 2 and 3. These different threads communicate with each other through the shared memory.

A typical flow works as follows: first, an image comes in and triggers the feature extraction thread, which then extracts feature points from the target image. Next, the tracking and the mapping threads pick up these features from shared memory. In addition, the tracking thread consumes IMU samples produces position updates.



Performance and Power

Feature Extraction:	45 ms
Tracking:	30 ms
Mapping:	15 ms
Average Power:	8 W

Fig 2. CPU implementation on mobile architecture

Delving into its performance, we found that the feature extraction stage takes 45 ms to finish, the tracking stage takes 30 ms, and the mapping stage takes 15 ms. On average, the overall system consumes 8 W when running this system. This system can only consume 15 to 20 images, and about 200 IMU samples per second. Ideally, for the SLAM system to meet real-time requirements, we would like to have it consume 30 images per second. Thus, the straightforward CPU implementation failed to meet performance requirement and incurred high power consumption.

B. Optimization with Heterogeneous Computing

Next we look at what we could do in the software layer to improve performance. We started by examining which stage could be easily parallelized: the tracking stages relies on a time-series of data, such that the new data has dependency on the previous data, and thus it is not straightforward to parallelize. We focus on the feature extraction thread.

As shown in Figure 3, we implemented our feature extraction stage on CPU, GPU, and DSP. Feature extraction takes 45 ms, 50 ms, and 20 ms respectively on CPU, GPU, and DSP. Note that using GPU even degrades performance, due to the high overhead to setup data and computing resources. In terms of energy consumption, each execution of feature extraction consumes 112 mJ, 115 mJ, and 30 mJ respectively on CPU, GPU, and DSP. Therefore, DSP seems a perfect accelerator for feature extraction, reducing execution time by half and improving energy efficiency by almost four-fold.

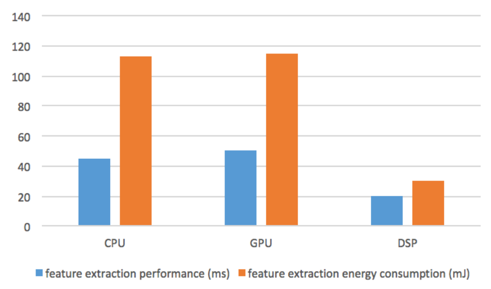
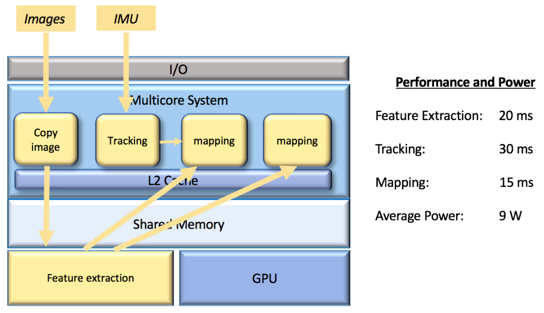


Fig 3. feature extraction performance and power

With DSP acceleration, let us examine the overall system performance. As shown in Figure 4, although we utilized DSP for feature extraction, we still required a core to relay data from I/O to DSP since DSP was not directly connected to the I/O system. This introduced two problems: first, we still needed to use a CPU thread to relay data, incurring extra power consumption. This was reflected in the average system power consumption, which increased to 9 W from 8 W due to the involvement of DSP. Second, this design had its impact on performance as well. As each time an image is received by the CPU, it needed to allocate an extract copy of it in memory then sent it to the DSP. By our measurement, this would take between 1 ms to 3 ms. Even worse, in a managed runtime environment, this could very likely introduce garbage collections, which freezes the whole system for over 100 ms, and thus stalling the whole SLAM pipeline, leading to the system losing track of itself.

Nonetheless, the SLAM system could now process 30 FPS, resulting in real-time performance, though this real-time performance could be interrupted by buffer copying and garbage collection operations, and the power consumption is not ideal. A demo of the implementation on heterogeneous mobile SoC is shown in [9].



Performance and Power

Feature Extraction:	20 ms
Tracking:	30 ms
Mapping:	15 ms
Average Power:	9 W

Fig 4. heterogeneous implementation on mobile architecture

C. Problems with Existing Heterogeneous SoCs

Through this study, we have fully utilized the heterogeneous computing resources available on the existing SoC. With heterogeneous computing optimization, we were able to achieve 30 FPS performance but power consumption was still high due to SoC architectural design constraints. We also derive three interesting observations: 1.) ***IO overheads***: the image IO pipeline usually involves several memory copies, leading to performance and energy consumption overheads; 2.) ***memory hierarchy overheads***:

although image data movement is on the critical path, in current SoC designs, , the image data does not have a dedicated data channel and needs to travel through the memory hierarchy with unpredictable latencies; 3.) ***accelerator needed for image processing frontend***: the comparisons of feature extraction performance on DSP vs. CPU confirm that accelerating the image processing frontend is an effective way to optimize performance and energy consumption.

IV. Π-SoC Heterogeneous Architecture

The results from the previous section indicate that the existing heterogeneous SoC architecture design is not optimized for SLAM applications. We propose π-SoC, a heterogeneous SoC architecture to systematically optimize the IO sub-system, the memory sub-system, and accelerator integration to improve performance and energy consumption for visual inertial SLAM.

A. Direct IO

As discussed in the previous section, in current implementations, the feature extraction accelerator (in this case the DSP) is not directly connected to the IO system, and a CPU core has to act as a relay to copy image data in memory for the accelerator to consume. Each incoming image is at least 300 KB, at 30 FPS, this means 10 MB of memory allocation per second. This not only wastes CPU resources, leading to extra power consumption, but also leads to memory problems, such as memory copy overheads and garbage collections.

To solve this problem, we propose direct IO such that the accelerator gets directly connected to the image sensor IO pins, allowing the accelerator to directly consume image data without the involvement of a CPU core. This way we free up one CPU core, and gets rid of the extra memory copying.

Some may worry that this design would introduce extra power consumptions. To find out the power overhead of driving the extra I/O pins, we implemented a test to repeatedly read data from a serial IO pin and measured its power consumption overhead, and the measured power consumption driving the pins is only 0.1 W.

B. ScratchPad Feature Buffer

With Direct IO, we could offload feature extraction tasks to the accelerator, however, the extracted features would go into memory, and later on the threads running on CPU cores would make a copy of the features then consume them. On mobile devices, the memory access latency is at the scale of 100 ns. To consume the extracted features, each second the CPU threads make at least 6000 memory access requests. Based on our profiling results, the extracted feature memory accesses account for about 20% of the total execution time of the update thread and the mapping thread.

From our initial study, we understood that from each image we generated at most 4 KB of feature data. So if we could design a low-latency ScratchPad memory [7] to directly feed the extracted features to the CPU cores, then we could significantly improve overall SLAM performance.

To understand the performance and power consumption of such ScratchPad memory, we utilized CACTI to analyze our design [8]. In this design, the size of the ScratchPad memory is 8 KB with two memory banks, one for each batch of image features, this way we leave enough swap space to store the features from two frames, thus making sure we do not have dropped frames if the CPU cores temporarily lag behind the DSP, and the ScratchPad memory should be implemented in at least 32 nm technology. The results are summarized in Table 1, this ScratchPad buffer consumes at most 0.15 W when active and it has a leakage power of only 0.002 W. In addition, each memory access now takes only 0.4 ns instead of 100 ns. Using this buffer, we can significantly improve the performance of the SLAM pipeline, by at most 20%.

Table 1. ScratchPad feature buffer performance and power

Access Time (ns)	Dynamic Power (W)	Leakage Power (W)
0.4	0.15	0.002

C. SLAM Trigger

Once the extracted features get written to the ScratchPad memory, we need a mechanism to notify the consumers, including the update thread and the mapping thread. Instead of having these two threads busy-waiting for the update, we implemented a notification mechanism, SLAM Trigger, and it works as follows: in the Feature Buffer controller we have a register to store the current filled feature memory bank ID. Once a bank is filled, the bank number is written in this register and the Feature Buffer controller would then interrupt the CPU to notify the update thread and the mapping thread about the incoming new features. Once these two threads are woken up, the CPU would lock the filled bank and then clear the filled feature memory bank ID while the accelerator is writing to the other bank. Once the CPU finishes consuming the feature buffer bank, it releases the bank and the threads go back to sleep. By swapping the two banks between the consumer and the producer, we effectively synchronize the two parties. If the consumer lags behind the producer, we can throttle the image frame rate to allow the consumer to catch up.

D. Integration into Π-SoC Architecture Design

Now let us systematically integrate these components into a new architecture, π-SoC, optimized for visual inertial SLAM workloads. As shown in Figure 5, in π-SoC, the accelerator is connected to the image sensor through Direct IO such that each time a new image comes in, it triggers the feature extraction task on accelerator, and the extracted feature gets written to the ScratchPad memory. Once the extracted features fill a ScratchPad bank, the ScratchPad controller interrupts the CPU to trigger SLAM.

In this design, we have both the mapping thread and the tracking thread running on Core 0, and the other mapping thread sharing Core 1. This is made possible since now we use a trigger mechanism for the mapping threads, such that the mapping thread is only active when data is available.

When the mapping thread is active, the incoming IMU data get buffered and later the tracking thread consume the IMU samples in batch. We verified that this technique does not impact the performance and accuracy of the SLAM system.

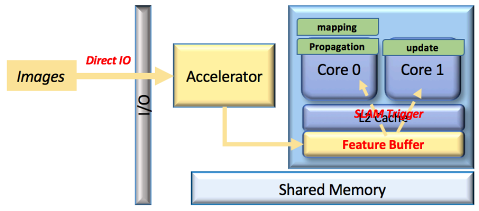


Fig 5. Π-SoC Heterogeneous Architecture

V. FPGA Implementation

In the previous section we have introduced the design of π-SoC, a systematic optimization of heterogeneous SoC for SLAM applications. In this section, we present a real-world implementation of π-SoC on a Xilinx Zynq UltraScale+ MPSoC FPGA [10] and demonstrate that it is a superior architecture compared to existing heterogeneous SoCs.

A. Π-SoC on FPGA

Figure 6 shows the implementation diagram of π-SoC on FPGA. In this implementation, direct IO and ScratchPad feature buffer are implemented in the *Feature Extractor* module, a specialized accelerator that consumes raw input images and extracts ORB features [3]. Other operations, such as mapping and tracking, are implemented on the ARM dual-core processor. A DMA is also in place to manage on-chip and off-chip data movements. On-chip system bus connects memory systems, Feature Extractor, ARM processors and DMA.

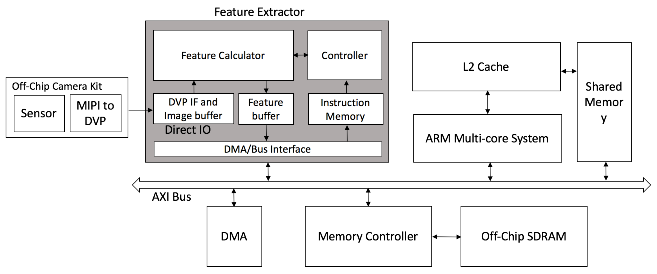


Fig 6. Π-SoC implementation on FPGA

B. Direct IO and ScratchPad feature buffer

To implement direct IO, we used an off-chip camera kits to provide Digital Video Port (DVP) interface to the FPGA. In the *Feature Extractor* module, *Direct IO* sub-module implements the DVP interface and stores the received images in on-chip image buffers. Then the *Feature Calculator* sub-module consumes images from the image buffer, extracts features from these images, and then stores features in the *Feature buffer*. When features are ready, DMA is then triggered to move features to the on-chip shared memory where the ARM dual-core system can directly access.

C. Feature Extraction Accelerator

Hardware implementation of ORB feature extraction accelerator is illustrated in Fig. 7. The accelerator consists of the *image resizing* unit, the *feature detection* unit, the *orientation computing* unit, and the *descriptor computing* unit. With the scale factor set to 1.2, the *resizing* unit implements bilinear interpolation and builds a 2-level image pyramid. The size of an input image is 640x480 and the size of a scaled image is 533x400. Then *RAM1* is used to store input images or scaled images. The *LB1* (line buffer 1) stores 31 lines of pixels. *RB1* (register bank 1) is used to buffer a patch of pixels (31 × 31 pixels) in *LB1*.

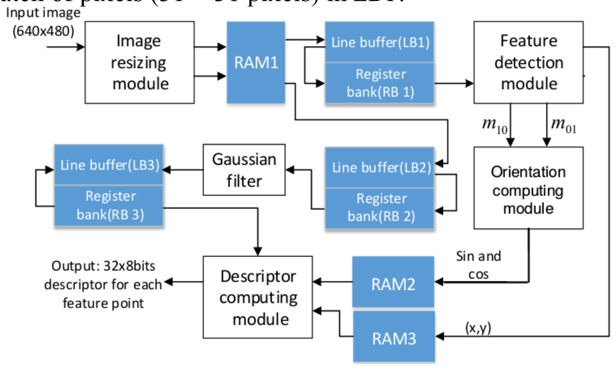


Fig. 7 Feature Calculator Sub-Module Implementation.

For a particular pixel in a patch, the *feature detection unit* first determine whether it is a feature point or not according to its intensity and the intensity of pixels around it. If a pixel is a feature point, the *feature detection unit* outputs its coordinate (x, y). Besides features' coordinates, the *feature detection unit* also determines features' orientation.

Let a feature point be the origin and in the center of a patch, the moments of the patch is defined in the following equation, where x, y are the coordinates of a pixel, $I(x, y)$ is the intensity of the pixel, and p and q are parameters, which is either 0 or 1. The coordinate of the intensity centroid of the patch is determined by $(m_{01}/m_{00}, m_{10}/m_{00})$. The orientation of the feature is defined by $\arctan(m_{01}/m_{10})$, which actually is the angle of the centroid. The *feature detection unit* outputs m_{01} and m_{10}, and the *orientation computing module* calculates the orientation of features. For more details regarding the ORB feature extraction acceleration design, please refer to [12].

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y)$$

D. Evaluation

With all the implementations in place, we used a Xilinx FPGA SoC (Zynq UltraScale XCZU9EG) [16] to evaluate this design's performance, energy consumption, as well as resource utilization, the results are listed in Tables 2 and 3.

Fist on performance, on FPGA, the max clock frequency of the hardware can reach 200 MHz. At this frequency, we can reduce the feature extraction latency down to 16 *ms* per frame, therefore effectively boosting the frame rate to 60 FPS, a 3X performance improvement over an ARM core.

Then on power consumption, *direct IO* and *feature buffer* consume only 46 *mW* and 77 *mW* respectively, where the *feature extractor* consumes 947 *mW*, making the whole accelerator system consume less than 1 *W*, 2.5X more power efficient compared to an ARM core. Combining the two, this implementation achieved close to 8X energy efficiency compared to the ARM baseline.

On resource utilization, the design consumes 62331 LUT and 351 BRAM units, accounting for only about 38% of the resources available on the FPGA chip, leaving significant spaces for more accelerators when needed.

These results verify that π-SoC is an effective architecture design that optimizes visual inertial SLAM performance and energy consumption.

Table 2. performance of the SLAM architecture

Feature Extraction (ms)	Tracking (ms)	Mapping (ms)
16	25	12

Table 3. power and resources of the SLAM architecture

	Direct IO	Feature buffer	Feature Calculator	Feature Extractor
Power (mW)	46	77	947	1070
LUT	504	32	61795	62331
BRAM	75	64	212	351

VI. RELATED WORK

Many works have been devoted to the design of real-time embedded visual SLAM systems. [11] and [12] present FPGA-based hardware to accelerate front-end parts of visual SLAM algorithms. Hardware and software co-designed systems have also been proposed to implement real-time visual SLAM application. [13] and [14] implement visual SLAM algorithm on FPGA+ARM SoCs in which computationally intensive tasks are implemented by FPGA fabric and the rest tasks run on ARM processors. More recently, an algorithm and hardware co-design methodology has been developed by [15], which explores algorithm and hardware design spaces and implement the whole VIO algorithm by special purpose hardware.

Instead of focusing only on accelerating one function in the visual inertial SLAM pipeline, we first explore the characteristics of visual inertial SLAM applications on existing heterogeneous SoC platforms, and then based on the observations, we propose π-SoC, a heterogeneous SoC design that systematically optimize the IO interface, the memory hierarchy, as well as the the hardware accelerator.

VII. CONCLUSIONS

In this paper, we have first conducted a thorough study to understand visual inertial SLAM performance and energy consumption on existing heterogeneous SoCs. The initial findings indicated that existing SoC designs are not optimized for SLAM applications, and systematic optimizations are required in the IO interface, the memory sub-system, as well as computation acceleration. Based on these findings, we proposed π-SoC, a heterogeneous SoC architecture optimized for visual inertial SLAM applications. Instead of simply adding an accelerator, we systematically integrated direct IO, feature buffer, and a feature extraction accelerator. To prove the effectiveness of this design, we implemented π-SoC on a Xilinx Zynq UltraScale MPSoC and was able to deliver over 60 FPS performance with average power less than 5 W. These results verify that π-SoC is capable to achieve performance and energy consumption optimization for visual inertial SLAM applications.

REFERENCES

[1] S. Thrun and J.J. Leonard, Simultaneous Localization and Mapping, Springer Handbook of Robotics, Springer, 2008.

[2] Z. Zhang, S. Liu, G. Tsai, H. Hu, C-C Chu, and F. Zheng, PIRVS: An Advanced Visual-Inertial SLAM System with Flexible Sensor Fusion and Hardware Co-Design, in Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2018

[3] E. Rublee, V. Rabaud, K. Konolige, G. Bradski. ORB: an efficient alternative to SIFT or SURF. IEEE International Conference on Computer Vision (ICCV).

[4] B. Triggs, P. McLauchlan, R. Hartley and A. Fitzgibbon, "Bundle Adjustment - A Modern Synthesis", International Workshop on Vision Algorithms, Springer, Berlin, 1999, pp. 298-372.

[5] ARM v8 Architecture: http://www.arm.com/products/processors/armv8-architecture.php

[6] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid and J. J. Leonard, "Past, Present, and Future of Simultaneous Localization and Mapping: Towards the Robust-Perception Age", IEEE Transactions on Robotics, 2016, 32(6), pp. 1309-1332.

[7] R. Banakar, S. Steinke, B.-S. Lee, M.Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In Proc. of the 10th International Symposium on Hardware/Software Codesign (CODES), May 2002.

[8] D. Tarjan, S. Thoziyoor, N.P. Jouppi, CACTI 4.0: cache modeling, area, delay, power, leakage power, HPL-2006-86

[9] Robot system running on a cell phone. https://youtu.be/PVvcmhGxJ6o. Accessed: 2017-11-20.

[10] Xilinx Zynq UltraScale+ MPSoC, https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html, Accessed: 2018-2-1

[11] K. Boikos, and B. Christos-Savvas. Semi dense SLAM on an FPGA SoC. 2016 26th International Conference on Field Programmable Logic and Applications (FPL) (2016): 1-4.

[12] W. Fang, Y. Zhang, B. Yu and S. Liu. FPGA-based ORB Feature Extraction for Real-Time Visual SLAM. In IEEE International Conference on Field Programmable Technology (FPT) 2017.

[13] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. Leutenegger, P.T. Furgale, and R. Siegwart. A Synchronized Visual-Inertial Sensor System with FPGA Pre-Processing for Accurate Real-Time SLAM. In IEEE International Conference on Robotics and Automation, 2014.

[14] G. Zhou, L. Fang, K. Tang, and H. Zhang. Guidance: A visual sensing platform for robotic applications. In IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2015.

[15] Z. Zhang, A. Suleiman, L. Carlone, V. Sze, and S. Karaman. Visual-Inertial Odometry on Chip: An Algorithm-and-Hardware Co-design Approach. Robotics: Science and Systems (RSS), 2017.