A High-Performance System-on-Chip Architecture for Direct Tracking for SLAM

Konstantinos Boikos
Department of Electrical and Electronic Engineering
Imperial College London
Email: k.boikos14@imperial.ac.uk

Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
christos-savvas.bouganis@imperial.ac.uk

***Abstract*—Simultaneous Localization and Mapping or SLAM, is a family of algorithms that solve the problem of estimating an observer's position in an unknown environment while generating a map of that environment. SLAM algorithms that produce high quality dense maps require powerful hardware platforms. In the simultaneous solution of these two problems, Localization, also known as Tracking, is the one that is latency sensitive and needs a sustained high framerate. This work focuses on providing an efficient, high-performance solution for Direct Tracking using a high bandwidth streaming architecture, optimized for maximum memory throughput. At its centre is a Tracking Core that performs non-linear least-squares optimization for direct whole-image alignment. The architecture is designed to scale with the available hardware resources in order to enable its use for different performance/cost levels and platforms. An initial implementation tested with a Zynq System-on-Chip can process and track more than 22 frames/second with an embedded power budget and achieves a $5\times$ improvement over previous work on FPGA SoCs.**

I. Introduction

Building machines with the ability to see and understand the space around them is an important milestone on the way to advanced, truly autonomous robots. In the last couple of decades there has been significant research and progress on this front. One of the main elements has been a family of algorithms and systems under the name Simultaneous Localization and Mapping, or SLAM. Their purpose is to provide a solution to the problem of online environment mapping, using a sequence of observations, while tracking the observer's position and pose in that environment.

SLAM is split into its two major components. Mapping, which is the task of fusing observations into a coherent model of the environment, and Tracking, which is the online pose estimation of the sensor used and, by extension, the platform it is attached to. They are both computationally and data intensive and the output of each one is strongly dependent upon the output of the other. Tracking, which is the front end of these algorithms, compares the incoming data from the sensor, with the constructed model of the environment to estimate the current pose of the system. At the same time, the estimated model will be updated on Mapping, with regard to the incoming data from the Tracking task.

Although both tasks are interdependent, Mapping can happen in the background and is less susceptible to a slower operation. A low framerate and high latency for Tracking will result in an autonomous robot failing to track its movement and, hence, a much slower and less robust operation. Custom hardware accelerators mapped onto an FPGA, acting as co-processors to a mobile CPU, can achieve both high performance and low latency while, at the same time, significantly increasing power efficiency for this task.

Many state-of-the-art SLAM algorithms, called Dense SLAM in the literature, are now able to provide a complete reconstruction of the environment with a high quality and localization accuracy. On the other hand, they are highly complex and data-intensive and require at least high-end desktop-grade hardware and often GPU acceleration [1] to process all of the available information at a framerate of $30\,\mathrm{frames/s}$ or more. Lately, works described as Semi-Dense have emerged. These use a smaller set of high quality observations in an attempt to achieve both better computational efficiency and a dense point cloud, useful for applications in robotics. However, they still require high-end multicore CPUs to process $30\,\mathrm{frames/s}$ with a Tracking resolution of 320×240 [2].

Providing this level of functionality and quality to a mobile low-power platform would open the way to emerging applications. Examples of this are search and rescue using autonomous UAVs[3] and intelligent household robotics[4]. Moreover, wearable technology, such as Augmented Reality, would benefit as it also needs low-latency, high performance environment awareness. These applications require fast on-board processing, however, the application domain requires lightweight and low-power hardware which cannot offer the necessary performance to run advanced state-of-the-art SLAM algorithms in real time.

This field is still characterized by significant volatility, with algorithms that change significantly every few years. The use of FPGAs will allow efficient heterogeneous architectures to be explored and used in the field while, reducing the cost of changes to follow the state-of-the-art in the software. Such architectures, if scalable, can also be used as a base for larger and more powerful machines, such as self-driving cars. There the high performance and low latency would provide safer and more reliable operation at higher speeds in more complex environments.

In this work, we propose a custom hardware architecture that provides a high frame-rate, low latency solution for Direct photometric Tracking using a monocular camera. Tracking is performed using a semi-dense map provided by a state-of-the-art Semi-Dense mapping task running as software, based on the work by J. Engel et al. [2]. To the best of our knowledge, this is the first high performance hardware architecture for Direct photometric Tracking for SLAM on an FPGA-SoC.

II. Background

Historically, SLAM and robotics have used various sensors to provide mapping and navigation. This includes laser scanners, sonar, a Kinect rig, stereo or monocular cameras, omnidirectional cameras and, lately, work has been published on using event-based cameras[5]. These sensors each provide their own advantages and disadvantages. This work focuses on a monocular camera as a sensor, since it has a low weight, size and energy consumption. This makes it especially attractive for lightweight mobile robotics, and as a sensor it can provide robust operation in large-scale unknown and unstructured outdoor environments[6].

Camera observations are noisy and the inherent ambiguities of monocular vision mean that the pose and map cannot be calculated precisely, but are modelled with probabilistic frameworks, with high computational requirements. The density of the data coming from a visual sensor also creates an incoming data rate of tens to hundreds of megabits per second that needs to be processed in real time, leading to significant memory and computation requirements. As a result the most advanced state of the art SLAM algorithms require at least desktop-grade CPUs and often a high-end GPU for acceleration to run at the camera rate of 30Hz or more.

The computational and memory requirements of SLAM algorithms scale with the amount of information processed, which depends on the resolution of the camera and the amount of that information that is actually used for the tracking and the environment model. On one end of the spectrum we have SLAM algorithms that process only a small set of observations, such as corners in the image, and use a sparse point cloud representation for the map, referred to as Sparse SLAM. A state-of-the-art example of these is ORB-SLAM [7]. As shown in that work, using high quality features achieves an accurate solution for tracking. However, high-quality features are computationally demanding to use and some state-of-the-art feature detectors also require a machine learning model to be retrained for each new type of dataset. Moreover, the map produced consists of a sparse selection of 3D points, which conveys limited information about the environment, so it's suited to robust odometry, in a known type of environment.

One the other end of the spectrum is Dense SLAM, which tries to use all of the available information coming from RGB or RGB-D cameras, and reconstruct as much of the space around the camera as possible, such as [1]. Between the two solutions is Semi-dense SLAM[2], which uses a smaller set of high quality observations compared to Dense SLAM, in an attempt to achieve both improved efficiency and a denser point cloud, which is much more effective for applications in robotics.

A Semi-Dense map is faster to compute and has more parameters to tune for different scenarios of operation. Simultaneously, the larger amount of information recovered in comparison to Sparse SLAM offers a more complete model of the environment, and can be post-processed to be used in conjunction with obstacle recognition, avoidance and other intelligent behaviour.

Finally, within the continuum of Sparse to Dense, SLAM algorithms can also be grouped based on their Tracking method, in the categories of Direct and Feature based. Direct tracking methods use the pixel intensity information in the image to perform whole-image alignment, by optimizing the pose based on the photometric error of aligning two images. Feature based methods focus on detecting a specific type of feature, such as corner or line, and generate a descriptor so that it can be matched across frames. They then optimize the camera pose using the geometric error of these point to point correspondences.

III. Related Work

The platforms in the embedded space have significant constrains in power and performance, and are not able to handle a large part of the state-of-the-art in Semi-Dense and Dense SLAM research. As a result, SLAM implementations in the embedded space mostly target Sparse SLAM algorithms, and they are often aggressively optimized to reduce computational requirements [8]. Tracking using a sparse selection of features, besides resulting in a less rich reconstruction, will be less robust and versatile in comparison to direct tracking as only information conforming to the feature type will be used.

Another approach is to limit mapping and reconstruction, and use lightweight SLAM front-ends as Visual Odometry, such as in [9]. This provides the option of performing sparse computationally efficient odometry on-board paired with remote computation on a base station to achieve dense map reconstruction[10]. However, this comes with its own drawbacks, such as increased power consumption and latency, high bandwidth requirements and a reduced area of operation.

There has not been much work on advanced state-of-the-art SLAM algorithms on FPGAs. Honegger et al. [11] proposed a system combining a mobile CPU and an FPGA for SLAM. In that, they describe a system architecture in which the FPGA is used as a preprocessor for feature detection, placed between the camera and the processing system using the frame grabber of the mobile CPU. However, the architecture described in that work isn't able to revisit data from the main memory or provide a more fine-grained cooperation between the cores. This means the applications of this architecture are restricted to algorithms that require no feedback from the software and process incoming frames only once. Another similar work is [12], in which the authors present an accelerator for LSD-SLAM on an FPGA-SoC and identify and describe bottlenecks and opportunities for accelerating such algorithms with this platform. However, a bottleneck in bandwidth and memory latency of their proposed design resulted in a lower performance than expected.

In contrast to the approach of Honegger et al.[11], in this paper a core running complex state-of-the-art Tracking for SLAM is investigated. It also uses a dataflow architecture and distributed asynchronous blocks, aiming to allow the memory system and the custom hardware pipelines to function at peak efficiency. It is designed to work as a co-processor with an advanced mobile CPU on a System-on-chip with more fine-grained division of tasks to fully take advantage of this heterogeneous platform. The goal of this work is to aim for a robust high-performance expandable baseline platform to be used for state-of-the-art SLAM in mobile robots.

IV. LSD-SLAM

LSD-SLAM[2] is used as the base algorithm for this work. It is a state-of-the-art Semi-Dense SLAM method, combining Direct photometric Tracking which works well in a variety of environments, with a semi-dense map as output. LSD-SLAM uses Keyframes to represent the map, a special subset of the camera frames essentially created by selecting a camera frame and fusing it with depth and depth variance information for a subset of its pixels. A new Keyframe is created by propagating the recovered information from the current one every time the camera frame's distance to it exceeds a certain threshold.

LSD-SLAM performs Tracking by directly optimizing on the image pixel intensities, in a process called direct whole-image alignment, to estimate the camera pose with respect to an existing Keyframe. It then uses that pose estimate to recover the depth of points on the camera frame from their frame-to-frame correspondences.

A subset of the camera pixels is defined by selecting the ones with a gradient above a certain threshold. They are referred to as Keypoints as they become part of the Keyframe, and have associated with them a depth and depth variance. During tracking, the Keyframe closest in distance to the current camera frame is used to generate matches between the Keypoints, which correspond to 3D points in the world and the same points captured in the current camera frame.

These matches are used to calculate the error of the current camera pose $\boldsymbol{\xi}$. The per-point errors are used to form a variance-normalized sum to be subsequently minimized, as in Equation 1:

$$E_p(\boldsymbol{\xi}) = \sum_{\boldsymbol{p}\in\Omega_{Di}} \left\| \frac{r_p^2(\boldsymbol{p},\boldsymbol{\xi})}{\sigma_{r_p}^2(\boldsymbol{p},\boldsymbol{\xi})} \right\|_\delta \qquad (1)$$

The operator $\|.\|_\delta$ is the Huber norm, a normalizing factor that reduces the effect of larger residuals on the optimization process. The residual r is calculated for the subset of pixels p which contain a depth value D_i, by subtracting the intensity of each KeyPoint, from the intensity of its projection on the camera frame based on the current pose estimate. This intensity is further refined by interpolating the 4 pixels surrounding that point on the camera frame, based on the floating point coordinates of the projection.

Finally, the sum is minimized using a modified Gauss-Newton optimization method, with a second-order approximation of the error function, where J is the residual vector of the error E, and the Hessian of E is approximated with $\boldsymbol{J}^T\boldsymbol{J}$. The update step of the optimisation function is found by solving a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where

$$\boldsymbol{A} = \boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J} \; and \; \boldsymbol{b} = -\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{r} \qquad (2)$$

This process is iterative, and repeats for a maximum number of steps or until convergence. As mentioned in [2], to improve the convergence radius of this method, the optimization is performed on different pyramid levels of the original image. A first step happens on a resolution of 30x40 pixels, and then the iterative process is repeated until convergence in successive levels from Level 3 (60x80) to Level 1 (240x320). This results in faster convergence and a more robust and accurate performance.

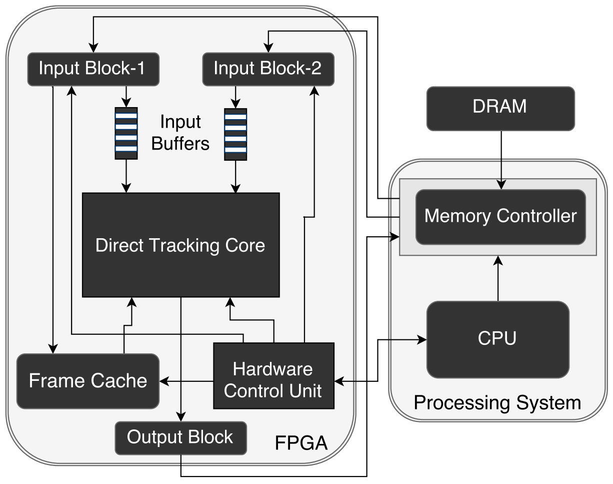

Fig. 1. System Architecture

V. Architecture

A. System Architecture

The system architecture was designed with an FPGA SoC in mind, a System-on-Chip that contains a mobile CPU and an FPGA fabric. The CPU and FPGA share the same off-chip DRAM, and they operate on the same memory space. The Direct Tracking Core on the reconfigurable fabric is connected directly to the memory controller through two dedicated high performance ports, where the accelerator acts as a master, providing a high throughput and low latency stream of reads from the DRAM. It is also connected to a general purpose port, where the core acts as a slave, to enable control and parameter set-up from the software running on the CPU.

B. Memory DMA and Streaming Dataflow

An important aspect of this architecture is the movement of data inside the core, as well as to and from memory. The communication with the off-chip DRAM, is routed through two dedicated hardware ports. Data reads are handled by two input blocks each with its own FIFO buffers, to allow the input system to absorb memory inefficiencies and delays. There is

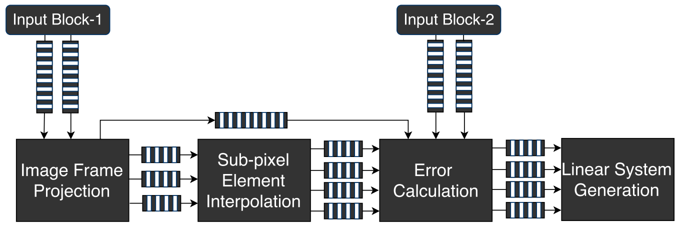

Fig. 2. Tracking Core Architecture

also an output block in the core, with its own master controller, to handle writing the output back to DRAM after the core has finished processing all elements.

Throughout this core, the communication between hardware blocks is done through FIFOs, as seen in Fig. 1 and 2. Firstly this allows blocks to function asynchronously, reducing delays, and improving the utilization of the pipeline. Secondly, this decouples the internal tracking pipeline from the input blocks, and absorbs the small gaps that can arise between memory requests, due to memory traffic and the latency of the memory system. For the same reason, the input blocks can be configured to run with at a faster rate than the rest of the core, in order to always have data available and push the utilization of the tracking pipeline close to 100%.

C. Direct Tracking Core

The tracking core is designed as a set of hardware blocks that operate asynchronously, with the same processing rate, connected by sets of FIFO buffers. It was also designed to scale well in terms of resources. By changing a few parameters, and with minimal modifications, the proper architecture can target a much larger FPGA, or be scaled down to co-exist with other cores or post processing, depending on the application.

The inputs for the core, streamed from the main memory, are in the form of a 5 element vector consumed for each Keypoint. Additionally, for each Keypoint, a 4x4 window is fetched from the currently tracked frame. The entire tracked frame is preloaded on the FPGA, in a partitioned memory described in Section V-D, to avoid the off-chip memory latency, which for successive random accesses would accumulate and become a very significant bottleneck.
In the following subsections, the blocks that constitute the Tracking Core are described in more detail.

1) Image Frame Projection: This hardware block is responsible for projecting a stream of reference points with coordinates (x,y,z) from the current Keyframe on the plane of the camera frame whose pose is currently being estimated. The projection is done based on the current estimate of the frame's pose using the pinhole camera model. The block finally generates a pair of floating-point image frame coordinates (x',y') that are streamed to the Sub-pixel Element Interpolation.

2) Sub-pixel Element Interpolation: This block receives the stream of floating-point coordinates (x',y'), belonging on the image plane of the current frame, generated from the Image Frame Projection block. Its functionality is first to find the block of 4 pixels surrounding that planar point and calculate their intensity gradients dx and dy. It then proceeds to calculate an interpolated value for the intensity and the gradients of the point (x',y'), using the values of these 4 pixels, based on the decimal digits of the coordinates. The output is a stream (I, dx, dy) of the interpolated values for the intensity and gradients.

The gradients for a pixel with coordinates (u, v) are calculated as :

$$dx = \frac{1}{2}(intensity(u+1,\ v) - intensity(u-1,\ v))$$

$$dy = \frac{1}{2}(intensity(u,\ v+1) - intensity(u,\ v-1))$$

To calculate these gradients, this block needs to fetch a 4x4 window of pixel values for every iteration, with an address and offset generated on the fly. The memory partitioning and window access pattern, that will be explained in more detail in subsection V-D was designed to allow low latency access for these 16 values, with the option of scaling to a single-cycle access architecture if more memory ports and blocks are used.

3) Error Calculation: This block receives multiple streams and is responsible for the calculation of most of the required values for the generation of the Linear System that is to be solved and the estimation of the total photometric error of the current pose estimate. The input streams include the (x,y,z) coordinates of the reference points, re-projected from the frame of reference of the new frame, the interpolated intensity and gradients streamed from the Interpolation block and finally a stream of the intensity and depth variance of the Keyframe reference points.

The first set of outputs of this block is the residual and a weight factor for each Keypoint, that will be used along with the (x,y,z) coordinates and the gradients in the next block. It also calculates the sums of the errors and squared errors while processing all Keypoints, as well as sums of the pixel intensities, and point quality characteristics for each Keypoint that is being tracked. The architecture accumulates these variables as partial sums in a number of distributed registers to account for the latency of the adder pipeline. After all Keypoints finish, these are summed, and streamed to the Output block to be written to the DRAM and later used by the rest of the system.

4) Linear System Generation: Finally, this block receives the (x,y,z) Keypoint coordinates, intensity and gradients, streamed on after being used on the Error Calculation block, as well as the error and weights that are calculated there. Its functionality is to generate the 6x6 matrix $\boldsymbol{A} = \boldsymbol{J}^T \boldsymbol{J} * w$ and the vector $\boldsymbol{b} = \boldsymbol{J} * r * w$ for the optimization method described in Section IV.

As matrix $\mathbf{A}$ is symmetric, only 21 out of 36 elements need to be computed. Taking this into account, a partitioned memory of size 21 was used to hold the temporary sums. This memory shares the same distributed register architecture such as the one mentioned in the Error Calculation block, to allow single-cycle accumulation of results. At the end of the computation, these are added and repacked into matrix

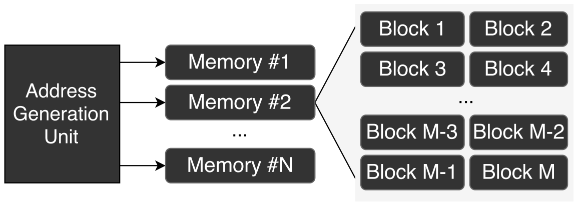

Fig. 3. Frame Cache Architecture

form to be written back to DRAM. Finally, large number of multiply and add operations per element naturally creates a requirement for a larger number of computational units to achieve the maximum throughput.

D. Frame Cache Partitioning

The Sub-pixel Interpolation block requires access to a 4x4 window for each tracked reference point. An architecture is developed for the frame cache to achieve low latency access to windows of this size. This was then generalized for window of a larger size. Using the assumption that the image width is a multiple of 4, an assumption that holds in most widely used resolutions in the literature, the row elements are stored in memory partitioned with a factor of 4 in a cyclic fashion. Using the available FPGA 2-port BRAMs, 8 successive elements can be accessed in a single cycle. Then, successive image rows in the image cache would be stored in different memories, in order to have the ability to perform parallel accesses per row.

This partitioning means that pixels directly underneath one another will be stored in the memory block with the same index, but belonging to the next memory. This results in a very simple address generation pattern, and it can scale well by increasing the number of memories, or memory blocks per memory. As seen in Fig. 3, we partition the Cache in a number of memories N, depending on how many rows need to be accessed in parallel. Then the number of memory blocks per row memory, M, can be calculated by the number of pixels we need to access simultaneously per row, then increased to the next closest multiple of 4.

The address generation is performed as follows. First, the address of the top-left corner of the window is expressed as $base * 4 + offset$, where

$$base = index/4, \ and \ offset = remainder(index, \ 4)$$

Thus, since $offset < 4$, the elements of each row are in the range: $address \in [base * 4, \ base * 4 + 7]$. Subsequently, 7 values are loaded in local registers instead of 4, starting from the address $[base * 4]$ which will always be aligned on a boundary of 4. Then, by using the value of $offset$, the right elements are stored in 4 registers to be used in the next phase of the pipeline. This design trades more BRAM ports for lower latency access, with simpler overall multiplexing.

VI. Evaluation

The design presented was implemented and synthesized using Vivado HLS and Vivado (v.2015.4) on a Zedboard, with a Zynq-7020 SoC. Due to the resource constraints of the FPGA fabric on this smaller chip, from the theoretic performance design points, one with a throughput of one Keypoint per 3 cycles was selected. On this board, the post-implementation resource usage was 80.8% for the LUTs (43028) and 80% for the DSPs (176). It also requires 78.93% of BRAMs (110.5) and 54.84% of FFs (58352). A frequency of 111MHz is achieved at this board. The estimated dynamic power consumption of the SoC for this configuration is 2.7 W for the CPU and FPGA combination.

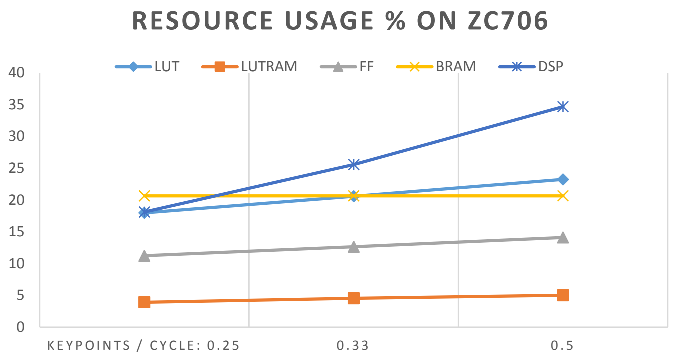

Fig. 4. Performance/Resource Scaling targeting a larger FPGA

A. Custom Core Performance

At this design point and frequency the acceleration achieved by the core was $40\times$ in comparison to a hand-optimized software version running on the dual core ARM-Cortex A9 of the same board, as shown in Fig. 5. This corresponds to a processing time of $9.49\,\mathrm{ms}$ in comparison to $383.9\,\mathrm{ms}$ for the software version. When accounting for the cost of copying the data to be processed to a dedicated area of the DRAM and the setup cost, the total latency for all iterations increases to $18.34\,\mathrm{ms}$ shown in Fig. 6, with the acceleration coming to $21\times$.

B. Resource Scaling

Fig. 4, shows post-implementation resource usage for different performance points, ranging from 1 Keypoint per 4 cycles to 1 Keypoint every 2, targeting the larger FPGA on the Zynq ZC706 board. The architecture scales well with resources. In fact, the significant change is floating point arithmetic units, which mostly affect DSP usage, scaling almost linearly with the required throughput, and some additional LUT requirements and FF requirements for the arithmetic units and to provide more pipeline registers. The design implemented on the Zedboard is the one corresponding to a throughput of one Keypoint per 3 hardware cycles, because of the available DSPs.

C. Running as part of a SLAM Pipeline

The two ARM cores on the Zynq are running a distribution of Ubuntu Linux as an operating system. This provides the necessary libraries for the open source version of LSD-SLAM. This was then modified so that the tracking frame function,

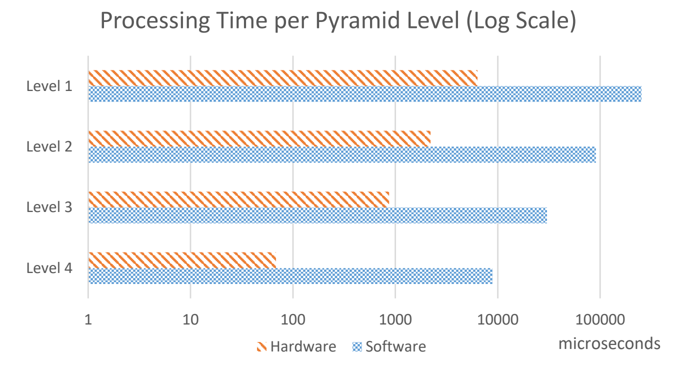

Fig. 5. Processing Time Per Level

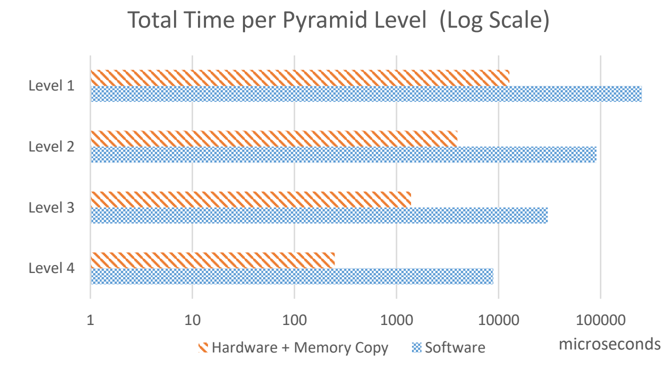

Fig. 6. Total Time per Level including Memory Copy

with the exception of some control and the copying of some data, is completely taken over by the FPGA core. The DRAM area used by the FPGA is marked as an I/O peripheral making it uncached and unbuffered. This allows simpler sharing of data between the two cores and the FPGA but it can slightly reduce the efficiency of writes and reads to this memory from the Linux OS. However the effect is not significant as only writes are performed from the software, and these are mostly streams of sequential stores.

Running a full SLAM system, the performance achieved for the entire Tracking function was on average $22.7\,\mathrm{frames/s}$. This was tested on a sequence of camera frames from the "Desk Sequence" provided by the authors of LSD-SLAM [2] stored on flash memory. Before a new frame can be processed the FPGA needs to wait for the software system to generate the required input data from the map. This is a source of delay that happens as a result of interfacing with the second half of the algorithm, as will be discussed in the next section. The software running on the dual-core ARM Cortex-A9 in comparison achieved $2.25\,\mathrm{frames/s}$ on average, an order of magnitude slower, using both cores to process in parallel at a frequency of 667MHz. The achieved framerate, when accounting for the software overhead in the test system, corresponds to a processing time of $44\,\mathrm{ms}$. This brings the acceleration from $21\times$ to $10\times$ for the full system to run in comparison to the software version and to $5\times$ in comparison to previous work on an FPGA-SoC[12].

VII. Performance Analysis

The two input ports of the core are connected through an AXI4 interface to two of the "high performance" (HP) ports of the Zynq, where the accelerator acts as Master, and can sustain a high throughput for reads and writes. In addition, an AXI4-Lite interface to one of the "General Purpose" ports where the core acts as a slave is used to receive control and parameter signals from the Software. Data is moved using the maximum bus width, in this case 64-bits, and unpacked in the hardware into two 32-bit floating point values.

The HP ports can scale up to a theoretical 1200 MB/Sec of read bandwidth each, at the maximum frequency, and with a continuous stream of 64-bit requests. In a test with a dummy producer/consumer, a sustained read and write throughput up to 1000 MB/sec each way has been tested successfully for a single HP port. However there are sources of inefficiency, in the protocol overheads, the frequency of the master connected to this port, and the overheads of the memory system.

Firstly, the ports have to work at 64 bits to have full bandwidth, and at the full 150MHz that it can operate on. However, the frequency is dependent on the rest of the Tracking Core as they share the same clock. Additionally, the data used is single-precision floating-point and therefore 32-bit. We need to process a set of 5 for each iteration of the Tracking Core blocks. This means that a percentage of the total bandwidth will be wasted, because of the mismatch of the memory transfer rate, and the values, and corresponding Bytes that can be consumed every cycle from the hardware. Another bottleneck, that becomes much more apparent in smaller boards, is the available computational units. Having fewer LUTs and DSPs to use will result in an architecture with a lower throughput. However, by running the inputs with a faster rate than the Tracking core, the pipeline utilization will get closer to 100%, leading to more efficient use of the available compute performance.

Moreover, currently the camera frames reside in the DRAM, and will have to be copied on-chip entirely before processing begins. This process is necessary because the access requests in the frame are random, dictating that the full frame needs to be available before processing can begin. Once copied on-chip it is stored on a static set of DRAMs, and is re-used as long as we are performing different optimization steps on the same pyramid level, which means that overall this copy happens only once per Level. However, it still manages to increase the total time necessary, on average, for the computation.

Finally, when running a full SLAM system, there is a source of delays on the software side because, before every frame can be processed, the software creates the different pyramid levels of the Keyframe depth map on the fly. This process turned out to be a significant bottleneck on our current test system, taking more than $25\,\mathrm{ms}$.

VIII. Conclusions

This work demonstrates a high performance state-of-the-art tracking architecture, with embedded-grade power consumption, by utilizing an SoC with reconfigurable logic. The proposed architecture is utilized in a full unmodified state-of-the-art SLAM, with an estimated $2.7\,\mathrm{W}$ of total power. The FPGA performed Direct Semi-Dense Tracking acting as a co-processor to a dual-core ARM Cortex-A9, with the full system achieving more than $22\,\mathrm{frames/s}$. The architecture is scalable, so it can target FPGAs with different resource profiles and is designed for significantly faster operation, thus paving the way to a complete embedded SLAM solution.

Acknowledgments

The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1) is gratefully acknowledged. The authors would also like to thank Stylianos Venieris, from the Department of Electrical and Electronic Engineering, Imperial College London for the interesting discussions and the rest of the Circuits and Systems group for their feedback.

References

[1] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, "DTAM: Dense tracking and mapping in real-time," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2320–2327.

[2] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM," in *European Conference on Computer Vision (ECCV)*, September 2014.

[3] T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. L. Grixa, F. Ruess, M. Suppa, and D. Burschka, "Toward a fully autonomous UAV: Research platform for indoor and outdoor urban search and rescue," *IEEE robotics & automation magazine*, vol. 19, no. 3, pp. 46–56, 2012.

[4] R. B. Rusu, N. Blodow, Z. Marton, A. Soos, and M. Beetz, "Towards 3D object maps for autonomous household robots," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE, 2007, pp. 3191–3198.

[5] H. Kim, S. Leutenegger, and A. J. Davison, "Real-time 3D reconstruction and 6-DoF tracking with an event camera," in *European Conference on Computer Vision*. Springer, 2016, pp. 349–364.

[6] M. Achtelik, M. Achtelik, S. Weiss, and R. Siegwart, "Onboard IMU and monocular vision based control for MAVs in unknown in-and outdoor environments," in *Robotics and automation (ICRA), 2011 IEEE international conference on*. IEEE, 2011, pp. 3056–3063.

[7] 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.

[8] B. Vincke, A. Elouardi, and A. Lambert, "Real time simultaneous localization and mapping: towards low-cost multiprocessor embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2012, no. 1, pp. 1–14, 2012.

[9] C. Forster, M. Pizzoli, and D. Scaramuzza, "SVO: Fast semi-direct monocular visual odometry," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 15–22.

[10] J. Sturm, E. Bylow, C. Kerl, F. Kahl, and D. Cremers, "Dense tracking and mapping with a quadrocopter," in *UAV-g 2013*, 2013.

[11] D. Honegger, H. Oleynikova, and M. Pollefeys, "Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2014, pp. 4930–4935.

[12] K. Boikos and C.-S. Bouganis, "Semi-dense SLAM on an FPGA SoC," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016, pp. 1–4.