FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM

Daniel Tortei, Jonathan Piat, Michel Devy

FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM

Daniel Törtei Tertei[1,2,3], Jonathan Piat[1,2] and Michel Devy[1]
[1]CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2]Univ de Toulouse, UPS, LAAS, F-31400 Toulouse, France
[3]Faculty of Technical Sciences, Department of Computing and Automation, 21000 Novi Sad, Serbia
Email: {dtertei, jpiat, devy}@laas.fr

***Abstract*—In hw/sw co-design FPGAs are being used in order to accelerate existing solutions so they meet real-time constraints. As they consume less power than a standard microprocessor and provide powerful parallel data processing capabilities, they remain a highly optimizable tool and object of research within an embedded system. In this paper we present an efficient architecture for matrix multiplication accelerator conceived as a systolic array co-processor to IBM's PPC440 processor on Virtex5 XC5VFX70T FPGA. Our design is afterwards synthesized and wired as a large-scale matrix multiplier required for an embedded version of a visual Simultaneous Localization and Mapping (SLAM) algorithm based on Extended Kalman Filter (EKF). This algorithm is implemented entirely as a System On a programmable Chip (SoC) design on the FPGA; an EKF epoch is executed at least 7.3 times faster than the pure software implementation, maintaining and correcting 20 points in the map. This optimization permits an EKF block throughput to be increased from 6.07Hz to 44.39Hz, which exceeds our real-time constraint of 30Hz.**

INTRODUCTION

A standard visual SLAM algorithm is required on a mobile system (here a robot) moving in an initially unknown environment. Simultaneously, it builds a map of this environment, estimating the positions of landmarks observed in images, and it estimates the robot position. All positions are expressed in a selected world reference frame which is generally the initial robot position. The robot must be equipped with a camera and besides it may also have proprioceptive sensors used to estimate its motion: odometry, Inertia Measurement Unit (IMU)...For outdoor applications, the SLAM algorithm can take advantage of position estimates given by a Global Positioning System (GPS).

For systems of limited size, typically for low cost aerial robots, a state-space based predictive algorithm such as Extended Kalman filter (EKF) [1] is most often applied for pose estimation. EKF runs in a two–part loop: i prediction of the robot position from the motion estimates and ii correction of the robot and landmarks positions with respect to the world reference frame. An additional loop, iii initialization of landmarks [2] is required due to physical displacement of the platform. In order to construct a reliable map of a complex environment, it is preferable to have many landmarks memorized in the map after loop iii as in that way, SLAM becomes more robust. On the other hand, introducing more landmarks increases the algorithm's computational complexity, having as a consequence in loop ii larger scale matrix-matrix multiplications.

Level of autonomy in autonomous driving systems and mobile robotics - such as [3] - is limited by power consumption of the platform used, which is most of the time a standard microprocessor-based hardware acrhitecture. As a solution to that issue, efficient embedded systems must be designed as to provide high computational power at lower power consumption rates [4]. Modern reconfigurable devices such as Field-Programmable Gate Arrays (FPGAs) consist of a large number of configurable slices, reconfigurable fabrics and embedded Digital Signal Processing (DSP) blocks that can be used for floating point applications. Compared to a standard PC and GPU-based platforms, computationally extensive algorithms may be also parallelized on FPGAs in floating-point precision [5]. Although there are as well embedded GPUs on the market today with similar power consumption to an FPGA, algorithm-specific memory management techniques along with reconfigurability give FPGAs an edge over embedded GPUs in terms of computational efficiency.

The EKF-based visual SLAM algorithm makes use of many matrix operations. So this paper presents mainly an efficient architecture of a matrix multiplication accelerator intended to be used in visual EKF-based mobile systems.

A. Related work and motivation

RT-SLAM [3] is a state of the art visual bearing-only SLAM algorithm with one camera to observe the environment and inertial sensors to estimate motion. It's map contains up to 20 Anchored Homogeneous Point (AHP)-parameterized landmarks [6]. Combined active search strategy and one-point RANSAC [7] are used to observe and match the landmarks. Iin this case, the real-time constraint considers a monocular SLAM with 20 consecutive landmark corrections. It's C++ implementation runs at 60Hz on a PC with an Intel i7 processor (only one core is used). Authors in [8] propose a simplified 3D SLAM algorithm with constant speed robot motion function (no inertia data) and Inverse-Depth Point landmark parameterization [9]. They present in [10] a coupled Virtex5 - Virtex6 FPGA [11] embedded architecture for C-SLAM that runs at 24Hz when having 20 IDP landmarks in the map and correcting half of them in each frame.

Our visual 3D SLAM algorithm is a monocular inertial SLAM with GPS and IMU odometry that observes and corrects 20 AHP points. Active search strategy is used as well to match the observed points and initialize new ones. As it is shown that pose estimation can make up to 84% of execution time of an EKF-based SLAM algorithm [12] and after the previous works

on the subject, our work is envisaged to improve the filtering part of the SLAM.

In Section I an analysis of EKF-SLAM complexity is given in order to deduce it's computational bottleneck. In Section II we introduce theoretical lower bounds on the latency of any multiplication algorithm with regard to its feasibility on an FPGA. A hardware architecture is proposed in Section III which is afterwards evaluated in terms of latency, resource usage and power consumption in Section IV. Experimental results are given in Section V followed by a discussion. Finally, Section VI concludes the paper.

I. EKF-SLAM COMPLEXITY

It is shown that computational requirements for an EKF algorithm depend on the number of features *N* retained in the map: $L_{EKF} = O(N^2)$ [13]. Equations of the EKF are given below.
Prediction loop:

$$\hat{x}^{(t)} = f(\hat{x}^{(t-1)}, \varpi^{(t)}) \tag{1}$$

$$P_1^{(t)} = F_x^{(t)} P_1^{(t-1)} F_x^{(t)T} + F_\omega^{(t)} Q F_\omega^{(t)T} \tag{2}$$

$$P_2^{(t)} = F_x^{(t)} P_2^{(t-1)} \tag{3}$$

Correction loop:

$$Z^{(t)} = H^{(t)} P_3^{(t)} H^{(t)T} + R \tag{4}$$

$$K^{(t)} = P_4^{(t)} H^{(t)T} (Z^{(t)})^{-1} \tag{5}$$

$$\hat{x}^{(t)} = \hat{x}^{(t)} + K^{(t)} (y^{(t)} - h(\hat{x}^{(t-1)})) \tag{6}$$

$$P^{(t+1)} = P^{(t)} - K^{(t)} Z^{(t)} K^{(t)T} \tag{7}$$

After each frame acquisition we choose the best twenty landmarks found by active search algorithm. Then we do a prediction step over P : 1 - 3. Matrices P_1, P_2, P_3 and P_4 are sub-matrices of the cross covariance matrix P - Table I, extracted in such a manner because of P's sparsity. Number of corrected landmarks is set to 20 and thus we run twenty correction loops in an EKF epoch, each with respect to features of a single observed landmark (see Table I). The entire P matrix is updated after each correction loop, as given in equations 4 - 7. Matrix dimensions according to a visual EKF-based SLAM algorithm are given in Table I. From Table I we deduce Table II which shows the total amount of Floating-Point Operations (FLOPs) that are executed in an EKF block in function of the number of retained landmarks *N* in the visual map.

Fact is that the most computationally expensive equations occur in the correction loop which makes 85% of all the FLOPs when $N = 20$ while only computing KZK^T makes up to 64% of time spent on all the equations in the EKF block. Highly correlated response is obtained after we run a code profiling tool [14] over the implemented EKF-SLAM code (which will be more detailed in section V). Thus, a significant speed-up can be made by leveraging the KZK^T tri-matrix multiplication in hardware and focusing on a design that efficiently manages operations around the cross-covariance matrix of the entire visual SLAM. The main reason of choosing to accelerate this specific part of EKF-SLAM instead of implementing the whole EKF block on the FPGA - as one related work suggests [15] - is because our entire SLAM algorithm is envisaged to run as a System on a programmable Chip (SoC) configuration on an FPGA which makes the area for possible acceleration logic limited in terms of resources. Our approach is to conceive a scalable solution that accelerates the most demanding parts of software.

Table I: Description and matrices dimensions according to our implementation of the EKF algorithm. *N* is the number of landmarks retained in the map, 7 is the number of AHP parameters of a landmark in the map, 6 is the number of GPS and IMU related variables, *c* is the number of corrections made in each EKF epoch and *r* is the state-space size of the robot.

Symbol	Dimension	Description
$\hat{x}$	$(7N + r) \times 1$	Robot and feature positions
f	-	Prediction function
P	$(7N + r) \times (7N + r)$	Cross covariance matrix
P_1	$r \times r$	Cross covariance matrix with respect to robot position
P_2	$r \times 7N$	Cross covariance matrix with respect to all landmarks
P_3	$(c + 1)7 \times (c + 1)7$	Cross feature-robot covariance matrix
P_4	$(7N + r) \times (c + 1)7$	Cross feature-feature and robot-robot covariance matrix
F_x	$r \times r$	Jacobian to system state
F_ω	$r \times 6$	Jacobian to system state perturbation
Q	6×6	Permanent perturbation
Z	$2c \times 2c$	Covariance innovation
H	$2c \times (c + 1)7$	Jacobian
R	$2c \times 2c$	Measurement noise
K	$(7N + r) \times 2c$	Filter gain
y	$2c \times 1$	Measured output
h	-	Observation function

Table II: Number of floating-point operations required for execution of equations in the EKF algorithm when $c = 1$ and $r = 19$. Equations 4 - 7 are executed 20 times for each of 20 landmark corrections.

Equation no.	FLOP
1	361
2	32300
3	4921N
4	868
5	420N + 1140
6	49N + 135
7	$196N^2$ + 1106N + 1558
Total	$3920N^2$ + 36421N + 106681

II. MATRIX MULTIPLICATION TRADEOFFS ON FPGAS

On a reconfigurable computing system the main tradeoff is between optimal speed and resource utilization. When considering matrix multiplication algorithms on FPGAs, we have to take into account their specific constraints as to latency *L*, total storage size in words *M* and memory bandwidth requirement *B*

denoted by number of I/O operations performed in each cycle. Based on a multiplication of two squared matrices of order *n*, authors in [16] explain these tradeoffs focusing on lower bound on achievable latency:

$$L \geq max(L_1, L_2) \geq max\left(\frac{n^3}{p}, \frac{n^3}{\sqrt{M}B}\right), (M \leq n^2) \quad (8)$$

where *p* is the number of Processing Elements (PE), L_1 is the latency according to computation time and L_2 is the latency dependent on I/O bandwidth. Furthermore, they conclude that an optimal latency is in this case of order $O(n^2)$ when having $p = O(n)$ PEs and $M = O(n^2)$ available storage size in words. However, for large scale matrices the latency is of order $O(\frac{n^3}{p})$. This is due to the fact that having FPGAs with limited resources it is hardly possible to instantiate that many PEs. A recent work [17] describes an architecture of linear array PEs, similar to those in [16], but achieving an optimal latency of order $O(n^2)$ by exploiting full duplex communication with the host processor and at the cost of having it involved during addition of intermediary values.

III. Architecture of the co-processor

In this section we start by describing our motivation behind a Systolic Array (SA) based platform for matrix multiplication. A computational flow is given following the dimensionality of our problem. Afterwards, the co-processor is put into the context of the SoC design as an Intellectual Property (IP) module.

A. Notion of systolic arrays on an FPGA

Systolic designs present a suitable solution when it comes to allocating resources given the specific problem size of our computational model. By identifying basic operations (floating-point multiplication and addition) we conceive a problem-specific computational unit called Processing Element (PE). Systolic array is an an N-dimensional grid of PEs that are fitted temporal data flows. Fitting is controlled by sequencers so that data streams entering the ports of the array are being processed in a pipeline. So, by exploiting structural properties of an SA we are able to lower the dimension of the computational model (which is three-dimensional for matrix-matrix multiplications) and obtain an optimal throughput at the cost of more complex control logic of data flows. Authors in [18] present useful mapping techniques inherent to an FPGA-based design. Because of speed issues, PEs are often organized in a linear list structure - a special case of a one-dimension systolic array in which they interconnect by short connection wires. In general, broadcasting data on a larger grid in an FPGA using a higher dimensional SA structure is not recommended because of speed degradation due to the size and routing complexity of FPGA-based floating-point units.

B. Computational model mapping

The tri-matrix multiplication is performed by taking into account the identity:

$$KZK^T = (Z^T K^T)^T K^T \quad (9)$$

as in that way the larger matrix K has to be transposed only once during the computation in order to avoid using buffers. We instantiate[1] two floating-point multipliers and a floating-point adder that form our PE (Figure 1-C). Computational flow is mapped onto four PEs that are not interconnected due to dimensionality of the problem: $A : (2, 2) \times (2, 159)$ and $B : (159, 2) \times (2, 159)$ as an individual PE structure computes a new result c_{uv} each clock cycle. Main multiplication operation is itself performed in two matrix-matrix multiplication stages: $A : Z^T K^T$ (Figure 1-A) and $B : (Z^T K^T)^T K^T$ (Figure 1-B). Thus we do not have any additional internal buffers and drivers instantiated in our SA which allows higher overall speed of the design. Main sequencer is a state-machine control logic unit which pre-buffers the input stream onto PEs - Figure 2. Its task is to provide the SA with input data at each clock cycle in order to maximize its throughput. For stage A it fetches matrices Z^T from First In First Out 1 (FIFO1)[2] and K^T from FIFO2. During both stages A and B elements of K^T are fetched and stored in a circular fashion using asynchronous read/write operation in FIFO2 structure. After initial SA latency it stores the result in FIFO1. At stage B it fetches input matrices from both FIFOs, loads them onto PEs, and after the results are being computed it stores them directly into four tri-port 1W/2R Block Random Access Memories (BRAMs)[3]. Along and after tri-matrix multiplication we need to perform two additional operations:

$$P^{(t)} - K^{(t)} Z^{(t)} K^{(t)T} \quad (10)$$

$$P_l^{(t+1)} = P_u^{(t+1)} \quad (11)$$

which will be more detailed in subsection III-C. The first is the floating-point subtraction and the second is a memory copy operation that updates the lower triangle of the P matrix as it has to be symmetric.

C. Co-processor as an embedded module in SoC

On Figure 3 we present the entire SoC with the co-processor attached as a Processor Local Bus (PLB) peripheral [4]. It functions as follows:

1) Processor PPC440 loads matrices K and Z onto a fast on chip memory unit - the default dual port 1W/1R BRAM - via its local bus from the external DDR2 SDRAM.
2) PPC440 sends a "go" signal to the IP.
3) Input sequencer loads matrices K and Z from the BRAM and stores them into IP's local FIFO structures. As they are read in element-by-element, they are transposed at the same time.
4) Main sequencer buffers Z^T and K^T and loads them onto four PEs in each clock cycle.
5) After initial delay, four PEs give each clock cycle 2 columns (4 elements) of the resulting $Z^T K^T$ matrix, and they are stored by main sequencer into $FIFO_1$. Nothing is stored into $FIFO_2$ as the K^T matrix is going to be reused again in second matrix multiplication $(Z^T K^T)^T K^T$.

[1] We make use of Xilinx IP Core Generator v13.1 to instantiate the floating-point arithmetic units.

[2] FIFOs are instantiated using Xilinx IP Core Generator v13.1 and implemented as dual-port BRAMs. Each FIFO structure contains 4 FIFOs of depth of 128 32-bit words.

[3] 1W/2R BRAMs are instantated on the FPGA. Read operation is done by multiplexing the BRAM controller's bus

[4] The system is designed using Xilinx Platform Studio v13.1

A: $a = Z^T,\ b = K^T$

```
for ( i = 0; i < n; i+=(p/2) ){
        c[0][i] = a[0][0]*b[0][i] + a[0][1]*b[1][i];
        c[1][i] = a[1][0]*b[0][i] + a[1][1]*b[1][i];
        c[0][i+1] = a[0][0]*b[0][i+1] + a[0][1]*b[1][i+1];
        c[1][i+1] = a[1][0]*b[0][i+1] + a[1][1]*b[1][i+1];
}
```

B: $a = [Z^T K^T]^T,\ b = K^T$

```
for ( i = 0; i < n; i+=p ){
        for ( k = 0; k < s; k++ )
                c[i][k] = a[i][0]*b[0][k] + a[i][1]*b[1][k];
                c[i+1][k] = a[i+1][0]*b[0][k] + a[i+1][1]*b[1][k];
                c[i+2][k] = a[i+2][0]*b[0][k] + a[i+2][1]*b[1][k];
                c[i+3][k] = a[i+3][0]*b[0][k] + a[i+3][1]*b[1][k];
}
```

C:

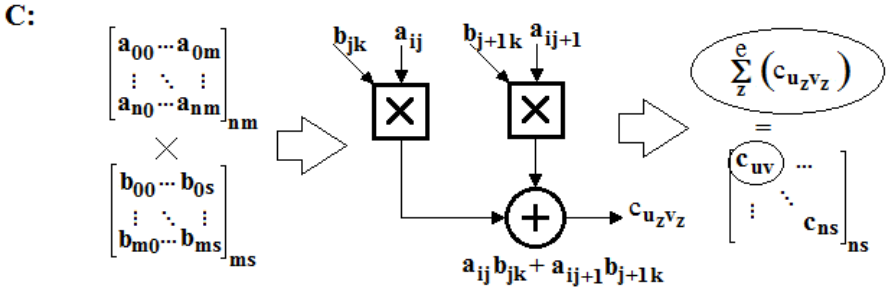

Figure 1: Our tiling of a systolic array with corresponding processing element structure, accommodated to our problem size I: n = m = 2, s = 159, and II: n = s = 159, m = k = 2. In both cases e = ceil(m/2) and p = 4 because we are instantiating two multiplying units in a PE and the SA consists of four PEs, respectively.

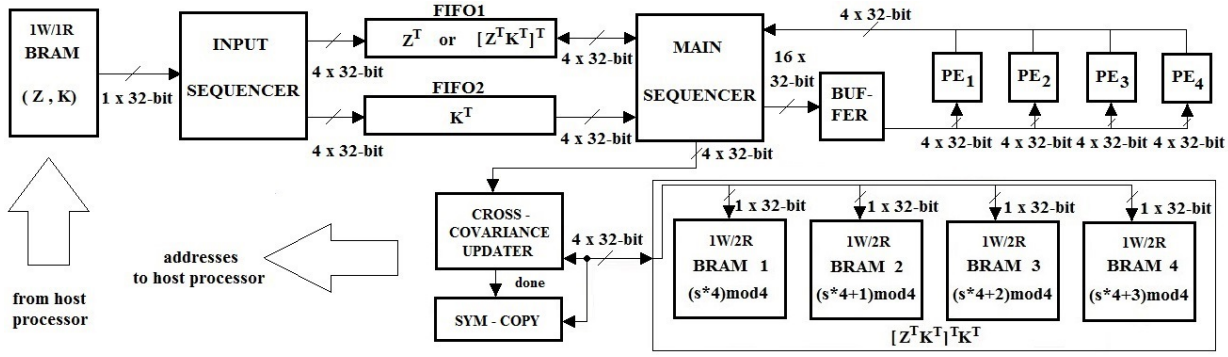

Figure 2: Tri-matrix multiplication architecture.

6) After having computed $Z^T K^T$, main sequencer buffers the elements for second multiplication and loads them onto PEs.
7) After initial delay, PEs compute 4 column elements of the resulting tri-matrix multiplication $(Z^T K^T)^T K^T$.
8) Before updating a corresponding column element in P matrix (Eq 10), the intermediary module "cross-covariance updater" pre-fetches its old value from corresponding BRAM and subtracts it from the new value. It contains four floating-point subtractors.
9) Main sequencer stores the subtracted four results into four BRAMs, each containing the $(4s + j) mod 4$ th element, where s is the current store cycle count and j is the number of the current row in a $mod 4$ cycle. There are 40 store cycles as the resulting square matrix is of dimension 159.
10) Also, out of the scope of the SA and as an additional submodule of the IP, we added a symmetry-copy BRAM controller that performs the equation 11.
11) The co-processor asserts a "done multiplication" signal. By polling, PPC440 gets notified of the execution of the operations in equation 7.

The resulting cross-covariance matrix P makes in total 102kB and after having finished the computations, we do not send it back via local processor bus into DDR2 SDRAM as it would generate large communicational delays. Instead, we preset pointer values in external memory on each corresponding element of the resulting matrix in the four output BRAMs.

IV. Design Evaluation

A. Multiplier Latency

After the analysis in section II we have:

$$L > L_0 + L_1 + L_2 \tag{12}$$

$$L_1 > max\left(\frac{n}{p}, \frac{n}{\sqrt{n}B}\right) \tag{13}$$

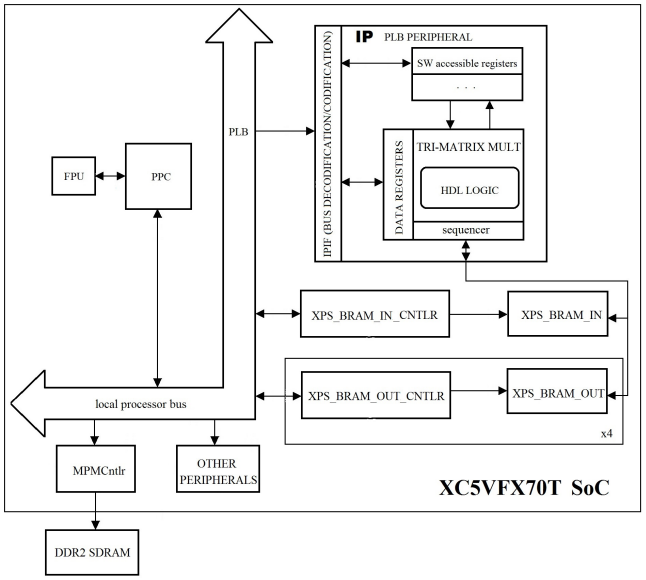

Figure 3: Our SoC implementation of the 3D EKF-SLAM algorithm.

$$L_2 > max\left(\frac{n^2}{p}, \frac{n^2}{\sqrt{n^2}B}\right) \quad (14)$$

L_1 and L_2 are latencies after $Z^T K^T$ and $(Z^T K^T)^T K^T$ matrix multiplications respectively. L_0 is the latency due to processor-IP communication bandwidth. Concerning the matrix multiplication part in the IP we have $M_1 = \sqrt{n}$ and $M_2 = \sqrt{n^2}$ because we have loaded all the corresponding matrices into two FIFO structures so that they could be fetched at each clock cycle. Furthermore, $B = 20$ owing to buffered input data and parallel storing of the results. We can clearly see that $L_1 > \frac{159}{4} = 39.75$ and $L_2 > \frac{159^2}{4} = 6320.25$ as we have $n = 159$ and $p = 4$ processing elements. All delays greater than 6360 clock cycles are due to L_0 and because of pushing the two input matrices onto FIFOs.

B. Resource usage

In Table III an overview over resource usage of the SA-based multiplier versus IP is given. The one-dimensional SA consists of 8 deeply pipelined multipliers and 4 adders. In terms of logical resources, our systolic array makes less than one fourth of the entire IP while the corresponding control logic, in consequence, makes more than three fourths: input sequencer, main sequencer, cross-covariance updater with four floating-point subtractors, a buffer and internal buffers with 32-bit delay lines - Figure 2. For speed issues, embedded multipliers (DSP48Es) are instantiated in each PE of the SA and in cross-covariance updater submodule. In Table IV we show the ratio of resource usage of the IP (with instantiated on-chip memory) versus all the available resources on FPGA XC5VFX70T. A 36k BRAM is used for the input BRAM. Eight 18k BRAMs are used for instantiation of the FIFO structures (4 × 18k per FIFO) and thirty-two 36k BRAMs are used to form the four memory banks[5] for storing 102kB of resulting data.

[5]A 32-bit 8192 word memory bank is implemented as 8 × 36k BRAM, instantiated in Xilinx Platform Studio v13.1. Each memory bank holds one fourth of the multiplication product, that is 6400 32-bit elements.

Table III: 1-D systolic array resource usage.

Resources	**Add**	**Mult**	**Array**	**% IP**
LUTs	238	100	1752	22.63
Slice registers	272	77	1704	18.15
DSP48Es	2	2	24	100
Latency	9	5	14	-
$F_{MAX}[MHz]$	367	295	255	-

Table IV: IP resource usage with memory.

Resources	**IP and BRAMs**	**% FPGA**
LUTs	8686	22
Slice registers	10198	25
DSP48Es	32	25
BRAMs [Kb]	1548	29
F_{MAX}[MHz]	119	-

C. Power consumption

The FPGA power consumption is estimated using Xilinx XPower Analyzer tool. Apart from device static power (leakage), our design consumes only 2.164 - 1.469 = 0.695 [W] which is much less when compared to embedded GPUs or microprocessors.

V. Experimental results and discussion

After having synthesized our design on the FPGA XC5VFX70T, it's maximal operating frequency is 119MHz. Thus its peak throughput is $((p \times 3)+4)FLOP \times 119MHz = 1.9GFLOPs$. Coupled to EKF-SLAM block it's operating frequency is that of processor's local bus (100MHz).

Using Gprof we obtain Table V which shows the cycles duration in each EKF block equation. We differentiate between the case when using and when not our accelerator in order to measure its impact. Below the horizontal line are the equations related to the correction loop, which is run twenty times in an EKF epoch. Some equations take longer to execute with acceleration because of host processor that is accessing parts of matrix P. The overall speed-up introduced by our co-processor is 44.39 / 6.07 = 7.31 times over - Table VI - in the EKF block. The SA multiplier reduced 38 times the number of

Table V: EKF equations cycles.

Equation no°	**Accelerator**	**No accelerator**
1	10000	10000
2	254500	250000
3	816000	750000
4	400	400
5	30100	27100
6	11200	11200
7	10500	399000
10	10	210000
11	6400	125000

Table VI: Co-processor efficiency at 100MHz.

Acc. IP	Optim.	KZK^T cycles	EKF [Hz]
Yes	O3	10500	44.39
No	O3	399000	6.07

Table VII: Design scalability for $F \geq 30Hz$.

N	GFLOPs	PEs	%DSPs	%LUTs	%SRs	%BRAMs
20-21	1.6	4	25	22	25	23
22-31	3.2	8	50	28	31	50
32-39	5	12	75	34	38	71
40-45	6.4	16	100	40	44	93

cycles for tri-matrix multiplications in equation 7. For code optimization settings we set the GNU Compiler Collection (GCC) optimization level to be the maximal - O3 - in order to measure the least performance increase. Moreover, a Floating Point Unit (FPU) is instantiated on the FPGA that speeds up floating-point operations using a direct communication chanel with PPC440 (as seen on Figure 3).

The input sequencer takes $2 \times 2 + 159 \times 2 = 322$ clock cycles and the computational latency is $L_1 + L_2 \sim 6535$ clock cycles[6] which leaves us for the host processor-accelerator communication latency $L_0 \sim 10500 - 6857 = 3643$ clock cycles. Computational latency is close to the predicted value in subsection IV-A from which we confirm that the algorithm is optimal according to the used processing power. The scalability of the design is given in Table VII focusing on maximum number of landmark corrections under the real-time constraint, the needed design performance in GFLOPs and resource constraints. Performance would be met if we instantiated multiple designs - each time 4 PEs anew - and thus slightly changed the control logic. The estimation is based on varying the parameter N as we keep correcting each time a single landmark ($c = 1$) in a loop.

VI. Conclusion

In this paper we proposed a tri-matrix hardware based accelerator that leverages matrix multiplications and updates the cross-covariance matrix in the correction loop of a 3D EKF-based SLAM algorithm. It has been entirely implemented on Virtex5 XC5VFX70T FPGA, supporting 3D-SLAM map sizes of 20 points represented with AHP parameters. With acceleration, an EKF epoch with 20 landmark corrections is executed at 44.39Hz. Moreover, the scalability of the design permits a real-time visual SLAM with up to 45 observed and corrected landmarks. Its performance per watt measure [19] is 1.9 / 2.164 = 0.88 [GFLOPs/W] which would not change substantially if we were to include additional logic (see Table VII) in order to implement the whole EKF block as an embedded design on a FPGA. Thus the achieved coefficient of ≈ 0.88 is, to our knowledge, above all other, embedded or not, state-of-the-art EKF block modules.

VII. Acknowledgement

This work has been performed by Daniel Törtei Tertei, paid by the FUI-AAP14 project AIR-COBOT, co-funded by BPI France, FEDER and the Midi-Pyrénées region.

[6]Validated by ChipScope Pro Analyzer

References

[1] H. Strasdat, J. Montiel, and A. Davison, "Real-time monocular slam: Why filter?" in *Proc. of IEEE International Conference on Robotics and Automation (ICRA)*, May 2010.

[2] J. Solà, A. Monin, M. Devy, and T. Lemaire, "Undelayed initialization in bearing only slam." in *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005, pp. 2499–2504.

[3] C. Roussillon, A. Gonzalez, J. Solà, J. Codol, N. Mansard, S. Lacroix, and M. Devy, "RT-SLAM : A Generic and Real-Time Visual SLAM Implementation." in *8th International Conference on Computer Vision Systems*, Sophia Antipolis (France), September 2011.

[4] D. Göhringer, M. Birk, Y. Dasse-Tiyo, N. Ruiter, M. Hübner, and J. Becker, "Reconfigurable MPSoC versus GPU: Performance, Power and Energy Evaluation," in *Proc. of IEEE International Conference on Industrial Informatics (INDIN)*, Lisbon, Jully 2011.

[5] K. Underwood and K. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, April 2004.

[6] J. Solà, T. Vidal-Calleja, J. Civera, and J. Montiel, "Impact of landmark parametrization on monocular EKF-SLAM with points and lines," *International Journal on Computer Vision*, 2011.

[7] M. Fischler and R. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," *Communications of the ACM*, June 1981.

[8] A. Gonzalez, J. Codol, and M. Devy, "A C-embedded Algorithm for Real-Time Monocular SLAM." in *18th International Conference on Electronics, Circuits and Systems*, Beyrouth, Liban, December 2011.

[9] J. Montiel, "Unified inverse depth parametrization for monocular slam." in *Proc. Robotics: Science and Systems (RSS)*, 2006.

[10] D. Botero, J. Piat, M. Devy, and J. Boizard, "An fpga accelerator for multispectral vision-based ekf-slam," *Proc. IROS Workshop on Smart CAmeras for roBOTic applications (SCaBot), Vilamoura (Portugal)*, October 2012.

[11] Xilinx. (2011) All Programmable FPGAs. [Online]. Available: http://www.xilinx.com/content/xilinx/en/products/silicon-devices/fpga/

[12] B. Vincke, A. Elouardi, and A. Lambert, "Implementation of EKF-SLAM on a Multi-Core Embedded System," in *IECON*, Montreal, Canada, October 2012.

[13] S. Thrun, W. Burgard, and D. Fox, "Probabilistic Robotics," *MIT Press*, 2005.

[14] J. Spivey, "Fast, accurate call graph profiling," *Oxford University Computing Laboratory*, September 2003.

[15] V. Bonato, E. Marques, and G. Constantinides, "A Floating-Point Extended Kalman Filter Implementation for Autonomous Mobile Robots," *Journal of VLSI Signal Processing*, July 2008.

[16] L. Zhuo and V. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems," in *IEEE Transactions on Parallele and Distributed Systems, Vol. 18, No. 4*, April 2007.

[17] Z. Jovanović and V. Milutinović, "FPGA accelerator for floating-point matrix multiplication," *IET Computers and Digital Techniques*, May 2012.

[18] A. Castillo-Atoche, D. Torres-Roman, and Y. Shkvarko, "Towards real time implementation of reconstructive signal processing algorithms using systolic array coprocessors," *Journal of Systems Architecture*, pp. 327–339, May 2010.

[19] G. Afonso, R. Ben Atitallah, A. Loyer, J. Dekeyser, N. Belanger, and M. Rubio, "A prototyping environment for high performance reconfigurable computing," in *Proc. of IEEE International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, Montpellier, France, June 2011.

[20] "IEEE Standard for Binary Floating-Point Arithmetic," *IEEE*, 1984.