FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM
Daniel Tortei, Jonathan Piat, Michel Devy

To cite this version:

Daniel Tortei, Jonathan Piat, Michel Devy. FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM. International Conference on ReConFigurable Computing and FPGAs (ReConFig), Dec 2014, Cancun, Mexico. 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14) 2014, <10.1109/ReConFig.2014.7032523>. <hal-01354873>
FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM

Daniel Törtei Tertei¹, ², ³, Jonathan Piat¹, ² and Michel Devy¹
¹CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
²Univ de Toulouse, UPS, LAAS, F-31400 Toulouse, France
³Faculty 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 customizable 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 architecture. 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. In 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:**

$$\dot{x}^{(t)} = f(\dot{x}^{(t-1)}, \omega^{(t)}) \tag{1}$$

$$P^{(t)}_1 = F_x^{(t)} P^{(t-1)}_1 F_x^{(t)T} + F_\omega^{(t)} Q F_\omega^{(t)T} \tag{2}$$

$$P^{(t)}_2 = F_x^{(t)} P^{(t-1)}_2 F_x^{(t)T} \tag{3}$$

**Correction loop:**

$$Z^{(t)} = H^{(t)} P^{(t)}_3 H^{(t)T} + R \tag{4}$$

$$K^{(t)} = P^{(t)}_4 H^{(t)T} (Z^{(t)})^{-1} \tag{5}$$

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

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

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 $K Z K^T$ 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 $K Z K^T$ 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.

## 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 $B$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$(7N + r) \times 1$</td>
<td>Robot and feature positions</td>
</tr>
<tr>
<td>$f$</td>
<td>-</td>
<td>Prediction function</td>
</tr>
<tr>
<td>$P$</td>
<td>$(7N + r) \times (7N + r)$</td>
<td>Cross covariance matrix</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$r \times r$</td>
<td>Cross covariance matrix with respect to robot position</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$r \times 7N$</td>
<td>Cross covariance matrix with respect to all landmarks</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$(c + 1)7 \times (c + 1)7$</td>
<td>Cross feature-robot covariance matrix</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$(7N + r) \times (c + 1)7$</td>
<td>Cross feature-feature and robot-robot covariance matrix</td>
</tr>
<tr>
<td>$F_x$</td>
<td>$r \times r$</td>
<td>Jacobian to system state</td>
</tr>
<tr>
<td>$F_\omega$</td>
<td>$r \times 6$</td>
<td>Jacobian to system state perturbation</td>
</tr>
<tr>
<td>$Z$</td>
<td>$6 \times 6$</td>
<td>Permanent perturbation</td>
</tr>
<tr>
<td>$H$</td>
<td>$2c \times 2c$</td>
<td>Covariance innovation</td>
</tr>
<tr>
<td>$R$</td>
<td>$2c \times 2c$</td>
<td>Jacobian</td>
</tr>
<tr>
<td>$K$</td>
<td>$(7N + r) \times 2c$</td>
<td>Measurement noise</td>
</tr>
<tr>
<td>$y$</td>
<td>$2c \times 1$</td>
<td>Filter gain</td>
</tr>
<tr>
<td>$h$</td>
<td>-</td>
<td>Measured output</td>
</tr>
<tr>
<td>$-$</td>
<td>-</td>
<td>Observation function</td>
</tr>
</tbody>
</table>

| 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 |
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:

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

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 \( O(n^3) \) 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 \( O(\frac{n^2}{p}) \). 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:

\[
KZK^T = (Z^TK^T)^TK^T \quad (9)
\]

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: \( A : Z^TK^T \) (Figure 1-A) and \( B : (Z^TK^T)^TK^T \) (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:

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

\[
P^{(t+1)} = P^{(t+1)} \quad (11)
\]

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 \( Z^TK^T \) 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 \( (Z^TK^T)^TK^T \).

\(^1\)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 instantiated 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: \( a = Z^T, \quad b = K^T \)

\[
\begin{aligned}
  & \text{for } (i = 0; i < n; i += (p/2)) \\
  & \quad \{ \\
  & \quad \quad c[0][i] = a[0][0]*b[0][i] + a[0][1]*b[1][i]; \\
  & \quad \quad c[1][i] = a[1][0]*b[0][i] + a[1][1]*b[1][i]; \\
  & \quad \quad c[i+1][i-1] = a[0][0]*b[0][i-1] + a[0][1]*b[1][i-1]; \\
  & \quad \quad c[i+1][i] = a[1][0]*b[0][i] + a[1][1]*b[1][i]; \\
  & \quad \} \\
\end{aligned}
\]

B: \( a = [Z^T K^T]^T, \quad b = K^T \)

\[
\begin{aligned}
  & \text{for } (i = 0; i < n; i += p) \\
  & \quad \{ \\
  & \quad \quad \text{for (k = 0; k < s; k++)} \\
  & \quad \quad \quad \{ \\
  & \quad \quad \quad \quad c[i][k] = a[i][0]*b[0][k] + a[i][1]*b[1][k]; \\
  & \quad \quad \quad \quad c[i+1][k] = a[i+1][0]*b[0][k] + a[i+1][1]*b[1][k]; \\
  & \quad \quad \quad \quad c[i+2][k] = a[i+2][0]*b[0][k] + a[i+2][1]*b[1][k]; \\
  & \quad \quad \quad \quad c[i+3][k] = a[i+3][0]*b[0][k] + a[i+3][1]*b[1][k]; \\
  & \quad \quad \} \\
  & \quad \} \\
\end{aligned}
\]

C:

\[
\begin{align*}
  & \begin{bmatrix}
    a_{00} & \ldots & a_{0n} \\
    \vdots & \ddots & \vdots \\
    a_{m0} & \ldots & a_{mn}
  \end{bmatrix}_{nm} \times \\
  & \begin{bmatrix}
    b_{00} & \ldots & b_{0s} \\
    \vdots & \ddots & \vdots \\
    b_{ms} & \ldots & b_{ms}
  \end{bmatrix}_{ms} \\
  & \Rightarrow \begin{bmatrix}
    c_{00} & \ldots & c_{0s} \\
    \vdots & \ddots & \vdots \\
    c_{ns} & \ldots & c_{ns}
  \end{bmatrix}_{ns}
\end{align*}
\]

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

6) After having computed \( Z^TK^T \), 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 \( (Z^TK^T)^TK^T \).

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 \( (4s+j)\mod4 \) th element, where \( s \) is the current store cycle count and \( j \) is the number of the current row in a \( \mod4 \) 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:

\[
L > L_0 + L_1 + L_2 \quad (12)
\]

\[
L_1 > \max \left( \frac{n}{p}, \frac{n}{\sqrt{nB}} \right) \quad (13)
\]
of resulting data.

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, its maximal operating frequency is 119MHz. Thus its peak throughput is ((p x 3)+4)FLOP x 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.

<table>
<thead>
<tr>
<th>Equation no</th>
<th>Accelerator</th>
<th>No accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>254500</td>
<td>250000</td>
</tr>
<tr>
<td>3</td>
<td>816000</td>
<td>750000</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>30100</td>
<td>27100</td>
</tr>
<tr>
<td>6</td>
<td>11200</td>
<td>11200</td>
</tr>
<tr>
<td>7</td>
<td>10500</td>
<td>399000</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>210000</td>
</tr>
<tr>
<td>11</td>
<td>6400</td>
<td>125000</td>
</tr>
</tbody>
</table>

3A 32-bit 8192 word memory bank is implemented as 8 x 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.
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 channel 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 $L_1 + L_2 \approx 6535$ clock cycles\(^8\) which leaves us for the host processor-accelerator communication latency $L_0 \approx 10500 - 6857 = 3643$ 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.

### Table VI: Co-processor efficiency at 100MHz.

<table>
<thead>
<tr>
<th>Acc. IP</th>
<th>Optim.</th>
<th>$KZK^T$ cycles</th>
<th>EKF [Hz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>O3</td>
<td>10500</td>
<td>44.39</td>
</tr>
<tr>
<td>No</td>
<td>O3</td>
<td>399000</td>
<td>6.07</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>GFLOPs</th>
<th>PEs</th>
<th>%DSPs</th>
<th>%LUts</th>
<th>%SRs</th>
<th>%BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-21</td>
<td>1.6</td>
<td>4</td>
<td>25</td>
<td>22</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td>22-31</td>
<td>3.2</td>
<td>8</td>
<td>50</td>
<td>28</td>
<td>31</td>
<td>50</td>
</tr>
<tr>
<td>32-39</td>
<td>5</td>
<td>12</td>
<td>75</td>
<td>34</td>
<td>38</td>
<td>71</td>
</tr>
<tr>
<td>40-45</td>
<td>6.4</td>
<td>16</td>
<td>100</td>
<td>40</td>
<td>44</td>
<td>93</td>
</tr>
</tbody>
</table>

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 44.39MHz. 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 $\approx 0.88$ 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.

REFERENCES


\(^8\)Validated by ChipScope Pro Analyzer