2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
September 24–28, 2017, Vancouver, BC, Canada

Design-Time Improvement Using a Functional Approach to Specify GraphSLAM with Deterministic Performance on an FPGA

Robin Appel, Hendrik Folmer, Jan Kuper, Rinse Wester and Jan Broenink

Abstract— **SLAM is a fundamental problem in robotics that can be solved by a set of algorithms that are known to have large computational complexity. GraphSLAM contains a rapidly growing system of equations which are often solved by sparse evaluation techniques. This paper proposes a technique to evaluate sparse equations on an FPGA by restricting the maximum amount of items in the system. The implementation is done using CλaSH which allows a transformation from mathematical descriptions to a hardware design. The results show a scalable hardware design that can be used to solve small and large systems with dynamic parallelism.**

I. INTRODUCTION

GraphSLAM is a modern graph-based technique to perform pose estimation for SLAM using probabilistic models in a graph representation [1]. Because of the high computational complexity of algorithms like GraphSLAM, the computations are often performed on high-end computing systems. These systems, often equipped with GPU and/or CPU are in many cases large and heavy and consume a lot of energy which makes them unsuitable to mount on mobile robots. Field programmable gate arrays (FPGAs) are a potential alternative hardware platform because of their reconfigurability, parallel nature, deterministic behaviour and lower power usage.

Designing hardware architectures for FPGAs using traditional low-level languages like VHDL or Verilog demands a lot of engineering effort. In this paper, CAES Language for Synchronous Hardware (CλaSH) is used to describe hardware in a way that is very close to the mathematics. CλaSH is a high-level hardware description language that borrows its syntax and semantics from the functional programming language Haskell [14]. In CλaSH the mathematical specifications are written down which describe relations between variables rather than instructions that describe the behaviour of values over time. From the mathematical specification the CλaSH-compiler can automatically a specification in a traditional HDL, like Verilog or VHDL, which can be synthesized for an FPGA.

We introduce an implementation of GraphSLAM on an FPGA created with CλaSH that completely runs on an FPGA. Implementation of an algorithm as complex as the GraphSLAM algorithm in an ordinary hardware description language is an extremely time consuming process which is very prone to design errors. Using CλaSH, the implementation is made from an abstract perspective that is closer to the mathematics compared to VHDL or Verilog, and is easier to transform into hardware than imperative programming languages.

The GraphSLAM algorithm implemented in this paper is environment independent which means it is not constrained to markers in a landscape or a sensor type. Another advantage is the scalable size of the state vector in GraphSLAM. This paper starts with an overview of GraphSLAM and some constraints to make the algorithm feasible for hardware implementation. With these constraints a functional design is created which uses all required resources in parallel. The amount of parallel resources such an implementation utilizes is more than most FPGAs offer. A shortage of resources can be solved by performing the calculations in different time steps.

II. RELATED WORK

SLAM is a collective name for a group of algorithms which have varying complexity and quality, the comparison should be made while considering the properties of the particular SLAM algorithm. SLAM algorithms on FPGAs often rely heavily on properties of the environment like in [2] and [3]. [7] presents an interesting approach of modifying the data structure of the pose and landmark graph to obtain efficient memory access management to solve the non-linear least square problem. They demonstrate that optimizing the data structure can decrease the execution time significantly for both single and multi-threading. [8] describes an FPGA implementation for a keypoint detection part of the SLAM algorithm, the FPGA is used as an accelerator. [9] presents a combination of an FPGA and mobile CPU. The FPGA is used as an accelerator layer between the CPU and image sensor to speed up the vision algorithm. Another proposed solution uses synthesized soft-cores in order to control accelerators on the FPGA instead of using dedicated control like in [4]. An implementation of SLAM on an FPGA is shown in [5] in which they use gene fitness based on running the ICP algorithm between the global map and local scans. A parallel implementation of GraphSLAM can be found in [10] showing an ICP-based implementation on a GPU with which they achieve fast computation times. [6] presents an FPGA-based architecture for the EKF algorithm that is capable of processing two-dimensional maps containing up to 1.8k features at real-time (14Hz). The presented architecture has 4 processing elements, containing floating point adders and multipliers, and is synthesized along with distributed memory containing data. The result is a 3-fold improvement over a Pentium M 1.6GHz, and a 13-fold improvement over an

The authors are all from: Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, 7500 AE Enschede, The Netherlands. {h.h.folmer, r.n.appel, j.kuper, r.wester, j.f.broenink}@utwente.nl

ARM920T 200MHz. We present in this work, using CλaSH, an application generic, environment independent and scalable GraphSLAM implementation on an FPGA able to achieve fast and predictable computation times.

III. Philosophy and Approach

Many of today's algorithms are implemented in imperative languages such as C, C++, or Java. Imperative languages describe a program as a series of instructions, they focus on how a computer executes a program. Imperative languages are designed for a standard CPU architectures and suitable to perform computations sequentially. To accelerate programs developers often start by parallelization of imperative sequential code. We propose to start with the mathematical descriptions, then implement these descriptions in a functional language like Haskell.

Haskell is a purely functional programming language. A crucial concept in Haskell is the presence of *higher-order functions*. Higher-order functions take functions or operators as arguments and apply those operators or functions on lists. A useful property of higher-order functions for hardware design is that they have an architectural structure. Examples of higher-order functions and their structure are shown in Table I. Note that the length of lists in Haskell is dynamic.

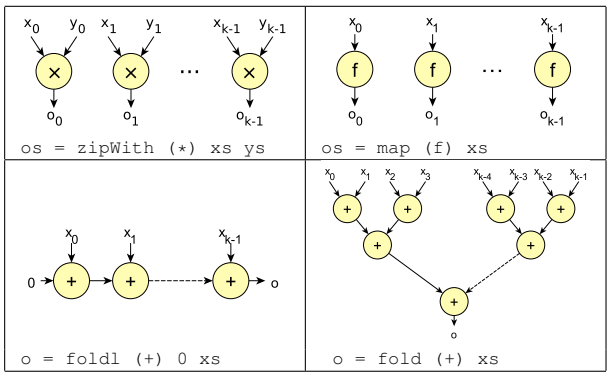
CλaSH is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. It provides a structural design approach to both combinational and synchronous sequential circuits. The CλaSH compiler transforms high-level mathematical descriptions to hardware architectures. In Haskell higher-order functions are applied on lists, but lists have dynamic lengths which is not suitable for hardware architectures. If an algorithm is written in Haskell using higher-order functions, reformulating it in CλaSH is only a matter of defining vector lengths and value representations.

The goal of this project was to create a hardware implementation for the GraphSLAM algorithm, therefore, we use CλaSH to automatically generate a hardware architecture from a mathematical specification. After the specification in CλaSH it is often not possible to synthesize the generated architecture due to hardware constraints of the target device, in our case the FPGA. Hardware constrains in terms of memory, area, clock frequency, combinational paths, IO. Due to these boundaries an area-time trade-off is needed in order to be able to execute the algorithm in hardware. Once the design is created it can be simulated and CλaSH can be used to automatically generate low-level hardware descriptions that fits the hardware. The design process can be summarized by the following steps:

A) Choose mathematical solutions/algorithm
B) Specify the math in Haskell and transform it to CλaSH
C) Determine hardware constrains
D) Create an area-time trade-off
E) Simulate and synthesize

Step A begins by specifying a mathematical definition of the solution for a problem, but already at this step it is important to keep in mind how well the target hardware

TABLE I
HIGHER ORDER FUNCTIONS AND THEIR STRUCTURE



could potentially handle the chosen algorithms. In case of FPGAs, regular structures which can be reused over time are preferred. Step B consists of specifying the mathematical descriptions in Haskell after which the description can be simulated using the Haskell interpreter. The Haskell code is easily transformed into a CλaSH description. If the specifications of the FPGA do not form a limit and the fully parallel design fits on an FPGA, then go directly to step E in which the HDL is generated with the CλaSH-compiler. In step C, determine the hardware constrains and which part of the system is going to be a bottleneck. Note that the bottlenecks are dependent on the chosen algorithm. In area-time trade-off, step D, start by looking at the overall structure of all the algorithms and determine regular parts of hardware which might be reused over time. The aim is to achieve a high utilization of the used hardware area. In this step, create an architecture which performs computations over time using memory and control. During every design step one simulates the design in CλaSH. In step E the CλaSH-compiler automatically generates a HDL, which is directly synthesizeable on an FPGA. These steps form a recipe for transforming complex algorithms into a hardware description and this approach has been used to describe GraphSLAM in hardware in the next section.

IV. GraphSLAM: Algorithm to FPGA

GraphSLAM is a SLAM algorithm that represents the environment as a graph. An operational GraphSLAM algorithm can be built from the following steps:

1) Creation of a system of linear equations consisting of the state vector and information matrix
2) Discover potential loop closure in the system
3) Include loop closure in the system of linear equations
4) Solve linear system of equations and update the state vector
5) Create a map of the environment

In this paper, we present the implementation of steps 1, 3, and 4. Step 2 is done manually because this paper focusses on the execution of the error convergence related parts of GraphSLAM, which are represented by steps 1, 3 and 4. Note that for this research only a pose graph is taken into consideration. Mapping is performed in step 5 which is not

part of the state estimation of GraphSLAM. Mapping with known poses is trivial, and having performed GraphSLAM, the poses in the state vector are known.

A. Algorithm

Steps 1 and 3 of the GraphSLAM algorithm construct the linear system that needs to be solved in order to perform error convergence based on loop closures. The created linear system is solved in step 4 and is described by (1). In this equation, $\Delta\vec{x}$ contains the state change that is applied to the current state vector $\vec{x}$. The relations between the states in the state vector are described by the information matrix ($\mathbf{H}$) and the errors that are found are represented in error vector $\vec{b}$. The sum of the errors can be minimized by iteratively solving the linear system from (1).

$$\mathbf{H}\ \Delta\vec{x} = -\vec{b} \tag{1}$$

The information matrix and error vector are created over time and their densities are dependent on the amount edges in the system. Each edge creates a matrix $\mathbf{H_{ij}}$ that is added to $\mathbf{H}$ which is shown in (2). Vector $\vec{b}$ is constructed by the same method, but only the edges that are constructed due to loop closing will inject values into $\vec{b}$.

$$\mathbf{H} = \sum_{ij} \mathbf{H}_{ij} \tag{2}$$

$$\vec{b} = \sum_{ij} \vec{b}_{ij} \tag{3}$$

Each partial matrix and vector is created by the presence of an edge between two nodes. Each partial matrix is a matrix where only four positions are non-zero with help of Jacobians. These Jacobian vectors will be 1 or -1 at positions *ii*, *jj*, *ij* and *ji* in the information matrix for an edge between $node_i$ and $node_j$ and all other items will be zero. A partial matrix is created in (4) and the same for the error vector in (5) where Ω_{ij} is a scalar that is defined as the probability of correctness of a transformation between nodes *i* and *j*.

$$\mathbf{H}_{ij} = \mathbf{J}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{J}_{ij} \tag{4}$$

$$\vec{b}_{ij} = \mathbf{J}_{ij}^T \mathbf{\Omega}_{ij} \vec{e}_{ij} \tag{5}$$

Equations (1) to (5) form a large part of the computations performed in the GraphSLAM algorithm which have been implemented on an FPGA.

B. Mathematical specification in Haskell or CλaSH

The information matrix usually contains a lot of zeroes, so storing it sparse will limit memory usage. The memory needed to store sparse data is dynamic, which means the amount of calculations will also be dynamic. Dynamic storage size of data and a dynamic amount of calculations is unwanted on an FPGA, which is primarily focussed on structured parallel calculations. In order to implement GraphSLAM efficiently on an FPGA, the algorithm should be deterministic. Memory size needed for storing matrices is growing quadratically with the number of poses (pose graph) when a traditional matrix notation is used. When using a sparse notation, the amount of items grows at an unpredictable rate with a maximum of the same quadratic growth that is seen with a traditional notation. In practice, the growth is never quadratic, and will be near linear growth. For pose graphs in GraphSLAM the amount of items that is needed to describe the matrix cannot be determined because it is dependent on the amount of loop closures. If a sparse notation is used, many calculations can be left out. A cumbersome property of sparse data is that the amount of memory that is necessary to store the result is not known in advance of the calculation because the amount of items in the result is dependent on the values of the indices of both inputs of the operation. An example is presented below showing a sparse vector addition where both vectors contain the same indices and the length of the output vector matches the size of the input vectors.

$$\begin{bmatrix}(0,\ 2)\\(1,\ 4)\\(2,\ 6)\\(3,\ 8)\\(4,\ 10)\end{bmatrix} + \begin{bmatrix}(0,\ 18)\\(1,\ 16)\\(2,\ 14)\\(3,\ 12)\\(4,\ 10)\end{bmatrix} = \begin{bmatrix}(0,\ 20)\\(1,\ 20)\\(2,\ 20)\\(3,\ 20)\\(4,\ 20)\end{bmatrix}$$

Another example addition is shown below in which there are exclusive indices which means the length of the output vector growths. The length of the resulting sparse vector can theoretically double by each addition if there is no match between any of the indices.

$$\begin{bmatrix}(0,\ 2)\\(1,\ 4)\\(2,\ 6)\\(3,\ 8)\\(4,\ 10)\end{bmatrix} + \begin{bmatrix}(2,\ 14)\\(3,\ 12)\\(4,\ 10)\\(5,\ 8)\\(6,\ 6)\end{bmatrix} = \begin{bmatrix}(0,\ 2)\\(1,\ 4)\\(2,\ 20)\\(3,\ 20)\\(4,\ 20)\\(5,\ 8)\\(6,\ 6)\end{bmatrix}$$

Since the amount of loop closures determines the amount of memory that is necessary for storing the sparse description of the matrix, restricting the number of items in the matrix should be considered. However, because loop closures can occur virtually anywhere, it is difficult to describe a model for the amount of loop closures since every environment can have a different structure. A way to restrict the amount of items in the information matrix is by restricting the amount of loop closures. Multiple techniques are known to reduce the amount of poses and edges in the system such as hierarchical optimizations [11], graph pruning [12] or optimization of the D-optimality of a graph [13]. Restricting the amount of loop closures can influence the resulting quality of the state vector and the map negatively. However, if loop closures is strategically done, the quality will remain sufficient. If this property is used, it is possible to have a fixed number of items in each vector of the information matrix and a deterministic number of calculations. For the GraphSLAM implementation presented in this paper, each pose can create one loop closure to another pose when its probability is above a specified threshold.

In GraphSLAM, edges can be created between consecutive poses for example by odometry sensors. Non-consecutive edges are formed by loop closures, which means they can connect virtually any two poses by an edge which means

non-zero items can appear in the information matrix anywhere.

$$\begin{bmatrix} d-1 & d & d+1 & in_1 & out_1 & \dots & in_n & out_n \end{bmatrix}^T \quad (6)$$

A possible structure of a fixed size sparse vector structure is shown in (6). Each pose, with exception for the first and the last, has been connected to the previous pose and next pose. In the information matrix, items describing consecutive relations between poses appear on the diagonal and directly above and below the diagonal, denoted by $d-1$, d , and $d+1$. The other non-zero items are created by loop closures. Outgoing loop closure means the current pose is closing the loop with an older pose (out_i) and incoming loop closure means a pose is connected to a newer pose in the graph (in_i). The idea of having reserved memory for loop closures is a method we created to ensure the maximum data size in a later phase. Using this structure for sparse vectors, GraphSLAM can be realized without extensive memory usage and unnecessary computations.

To solve the linear system, the conjugate gradient algorithm is used because it will leave the sparsity of the information matrix intact. The conjugate gradient is an iterative algorithm that converges towards the solution vector, Algorithm 1 shows the process of solving a linear system $\mathbf{A}\vec{x} = \vec{b}$. Also, the conjugate gradient algorithm consists mainly of vector operations which can easily be parallelized on an FPGA. A more extensive design choice of conjugate gradient and other choices are presented in [16].

Algorithm 1 Conjugate gradient algorithm to solve a linear system $\mathbf{A}\vec{x} = \vec{b}$

1: **initialize:**
2: $\vec{r}_0 = \vec{b} - \mathbf{A}\vec{x}_0$
3: $\vec{p}_0 = \vec{r}_0$
4: $k = 0$
5: **repeat until convergence of r:**
6: $\alpha_k = \frac{\vec{r}_k^T \vec{r}_k}{\vec{p}_k^T \mathbf{A} \vec{p}_k}$
7: $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$
8: $\vec{r}_{k+1} = \vec{r}_k - \alpha_k \mathbf{A} \vec{p}_k$
9: $\beta_k = \frac{\vec{r}_{k+1}^T \vec{r}_{k+1}}{\vec{r}_k^T \vec{r}_k}$
10: $\vec{p}_{k+1} = \vec{r}_{k+1} - \beta \vec{p}_k$
11: $k = k + 1$

The information matrix ($\mathbf{H}$) can be stored using a sparse notation because it has a restricted amount of non-zero items. All other vectors do not meet this requirement which means it is not advantageous to use a sparse notation to describe these vectors. Looking at the conjugate gradient algorithm in line 2 and 6, a multiplication between the matrix and a vector is performed (line 8 is the same multiplication as in line 6). The matrix-vector multiplication is the only operation between sparse and non-sparse data which consists of dot products. Dot products between vectors result in a single value which means that the addresses are only used to select the inputs and do not appear in the result of the operation.

Listing 1 shows a functional description of the conjugate gradient algorithm in CλaSH performing 5 iterations, note that Unicode is allowed in Haskell/CλaSH. The operator $\bullet$ is the function that multiplies the sparse information matrix with a non-sparse vector, The operator $\otimes$ does a sparse matrix vector multiplication and is defined in a similar fashion as the $\bullet$ operator which is described further in Section IV-D. The $\cdot$ operator does vector scaling. One can see that the functional CλaSH description from Listing 1 has many similarities in notation with Algorithm 1. This functional description can directly be transformed into a low-level hardware description in VHDL or Verilog. However, this implementation will synthesize the algorithm as a single combinational path, which does not fit in the FPGA for many applications.

```
cgi a b x0 = iterate d5 cgmIter (r0r0, r0, r0, x0)
  where
    r0   = b − (h ∗ x0)
    r0r0 = r0 • r0

    cgmIter state@(rr,p,r,x) = (rr',p',r',x')
      where
        hp  = h ⊗ p
        α   = rr  / (hp • p)
        x'  = x + (α · p)
        r'  = r − (α · hp)
        rr' = r' • r'
        β   = (rr' / rr )
        p'  = r' + (β · p)
```

Listing 1. Description of the conjugate gradient algorithm in CλaSH

C. Determining hardware constraints

FPGAs are constrained by area and time. Area determines how many operations the FPGA can perform in parallel, which can be operators created in logical elements or digital signal processors (DSPs). Time is defined by the clock frequency at which the FPGA operates, defined by the length of the combinational paths. Vector operations can potentially be performed in parallel, an important requirement is sufficient read and write speeds of memories. Memories have the restriction of only be able to read or write a single element of the width of the databus at once. In order to perform operations between sparse vectors from the information matrix and a non-sparse vector, the complete non-sparse vector should be available for computation at the same time sparse data is presented at an arithmetic unit. The non-sparse vector should completely be present because an index from the sparse vector can select any value from the non-sparse vector. Whenever the system size is large, the memory does not support a databus as wide as the complete non-sparse vector. Also, if this is possible, it would mean that a large number of multiplexers should be used to extract the correct item from the vector. In our GraphSLAM implementation, the unstructured memory operations ask for an alternative memory management to fetch the correct data in parallel. Because the used FPGA contains blockRAMs, multiple

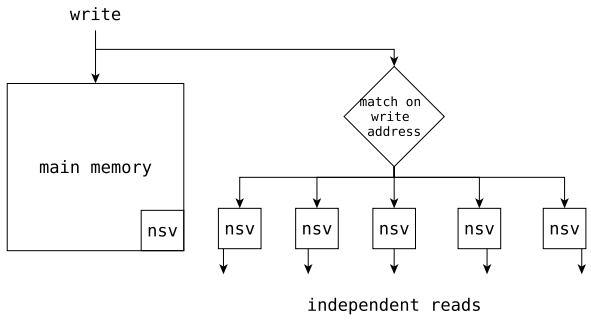


Fig. 1. Multiple copies of portion of memory to allow parallel access

TABLE II
RESOURCES TO PERFORM ONE CONJUGATE GRADIENT ITERATION

	scaleNSV	dotNSV	addNSV	subNSV	mulSMV	per CG iteration
adders		n-1	n	n	n*(m-1)	(m+5)*n-1
multipliers	n	n			n*m	(m+2)*n
occurrence per CG iteration	3	3	1	2	1	

memories can be synthesized in parallel. Our implementation uses large blockRAMs that contains all sparse and non-sparse vectors, and, next to those, multiple small blockRAMs that allow parallel access to small portions of the non-sparse vector. Each large vector will be cut down into multiple parts which makes it fit the capacity and databus of the blockRAM. Each small blockRAM contains a copy of the vectors that needs to be multiplied with the sparse vectors. Fig. 1 shows a structure that allows multiple reads from small blockRAMs which can be used to perform dot products between a sparse vector and a non-sparse vector in parallel.

The target FPGA for this project an Altera Cyclone V which contains 110k logic elements, 112 DSPs, and 5140 Kbits of blockRAM. As this project is performed in cooperation with an implementation of ICP, which is further described in [16], the FPGA resources are shared and only half of the FPGA resources can be used. Multiplications on an FPGA are performed preferably by DSPs since performing multiplications using logical elements is slower and utilizes many logical elements. Table II shows an overview of the resources that are required to perform one iteration of the conjugate gradient algorithm with a dynamic system size n and a sparse vector containing m items. The last column in the table represents the total number of resources for one conjugate gradient cycle. It has been deduced by multiplying the first 5 columns of the first two rows with the third row. For a system that has hundreds of poses, one conjugate gradient iteration requires thousands of multiplications. Given lack of so many DSPs, the multiplications should be run in multiple steps to perform these calculations. Because the conjugate gradient algorithm is iterative, the total number of computations is many times higher than this number, which means the amount of multiplications on DSPs is also a hardware constraint.

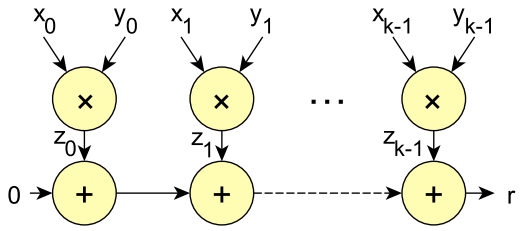


Fig. 2. Structure of a • product

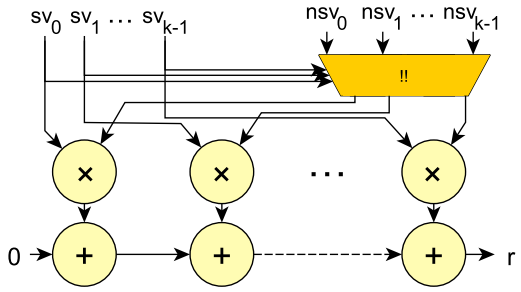


Fig. 3. Structure of the sparse • product

D. Time-area trade-off

Because of the hardware constrains a time-area trade-off must be made in order to fit the algorithm on the FPGA. The conjugate gradient algorithm contains a matrix-vector multiplication which consists of dot products. A structure of a dot trivial product is shown in Fig. 2 and the corresponding CλaSH code, using higher-order functions, in Listing 2. The parallel multiplications are performed by zipping (pairwise operate on two or more vectors) the values of both vectors with the multiply operator in the higher-order function *zipWith*. The sum is performed by the higher-order function *foldl* with the plus operator starting from zero, which is equivalent to the function *sum*. The multiplication part of the dot product has no data dependencies between the parts and can therefore be executed in parallel. The sum of the products has less parallel potential but since additions are associative the addition can be executed in parts. It is possible to perform operations in parallel without a clock transition which means the area of the FPGA is used. An operation can also be performed in different time-steps on the same area in which case the operations consumes a certain time. The trade-off between time and area has influence on the final computation times.

```
xs • ys = foldl (+) 0 (zipWith (*) xs ys)
```

Listing 2. CλaSH definition of the • operator

```
sv • nsv = foldl (+) 0 $ zipWith (*) xs ys
  where
    xs = map (snd) sv -- data from the sv
    inds= map (fst) sv -- list of indices
    ys = map (nsv!!) inds -- data from the nsv
```

Listing 3. CλaSH definition of the sparse • operator

The architecture in Fig. 2 performs a dot product on non-sparse vectors, in order to perform a dot product between a sparse and non-sparse vector, more components are needed. Fig. 3 and corresponding Listing 3 show the structure and

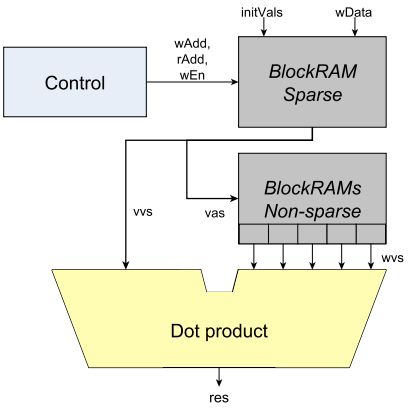


Fig. 4. Structure of dot product between sparse and non-sparse vector using BlockRAM

CλaSH code needed to perform a sparse dot product operation. Higher-order functions can be split up into parts while still being able to proof the correctness [15]. In case of the dot product the operation can be split up, and use memory components to store incoming data and partial results. The architecture of the dot product between a sparse and non-sparse vector is implemented using a multi-port memory as proposed in the previous subsection and an example architecture of a sparse vector with *5* items is shown in Fig. 4 and the corresponding code in Listing 4. Notice that blockRAM is sequential memory, which means the data is available one clock cycle later than it is requested. In this case, the addresses to read from the non-sparse memories are the indices that are read from the sparse memory.

```
(vas,vvs) = blockRam initValsS rdAddrS (wrAddrS, wrDataS)
wvs       = map (\rAddr ->
              blockRam initVals rdAddr (wrAddr, wrData)) vas
res       = vvs • wvs
```

Listing 4. CλaSH description of dot product between sparse- and non-sparse vector using a blockRAM

The architecture in Fig. 4 shows multiple blockRAMs, the structure with small non-sparse blockRAMs is the structure shown in Fig. 1. Each blockRAM has multiple arguments which are respectively: the initial values, read address, write address, and the data to write. Since the read address is the only thing that differs for each blockRAM, we inject the correct read address using the higher-order function *map* that passes the addresses from the sparse vectors (vas) to the blockRAMs. The dot product that is executed is replaced by an arithmetic block, which can be controlled to execute any specified arithmetic function which also appears in the final implementation. A structural overview of the final implementation is shown in Fig. 5. The multi-port memory is used to support parallel reads of sparse vectors. The dot product block in Fig. 4 has been replaced by an arithmetic logical unit (ALU) that can perform the vector operations such as dot products, vector additions and vector scaling so the conjugate gradient algorithm can be performed.

E. Simulate and synthesize

Fig. 6 shows an uncorrected state vector (path) on the left and a state vector that has been corrected after loop

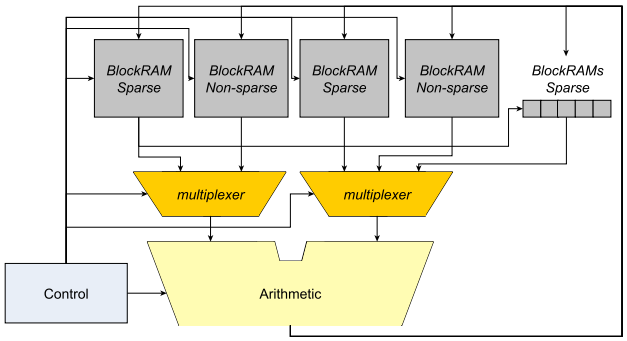


Fig. 5. Overview of the GraphSLAM vector processor

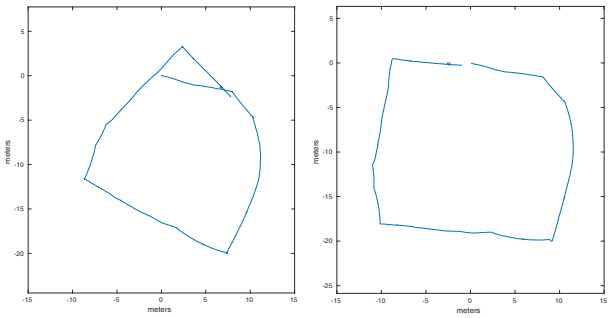


Fig. 6. Example input (left) and output state vector (right) of the GraphSLAM algorithm

closure by GraphSLAM on the right. The figures show the positions of the robot over time, no map has been drawn in these images. The input state vector consists of the first 336 poses of the Intel Research laboratory SLAM dataset which contains the first loop through the hallway of the building. The odometry data from the dataset has already been corrected by an ICP algorithm before using it in GraphSLAM which results in a better state estimation compared to only odometry. The raw odometry will result in a state estimation with converged error after loop closure too. However, finding a loop closure in an ICP corrected odometry state vector will be easier than in an uncorrected state vector. The image on the right shows a square figure in which the robot has moved, this square figure matches the structure of the building.

Equation (7) shows a formula used to calculate the number of clock cycles that is necessary to perform the conjugate gradient algorithm. It has been parametrized in the number of poses l, the number of items in a sparse vector sl, the width of the vector ALU w and the number of conjugate gradient iterations cg. This formula describes the number of clock cycles of the linear solver, it does not mention the rest of the GraphSLAM algorithm, since in a real life implementation the FPGA will mainly be idle with high utilization when loop closure occurs and errors are calculated.

$$cg \times \left\lceil \frac{3 \times l}{\lfloor \frac{w}{sl} \rfloor} + 13\frac{l}{w} + 2 \right\rceil \tag{7}$$

Table III shows three examples of parametrization of (7). The amount of parallelism is defined by w and the computation time heavily relies on how many conjugate

TABLE III
DIFFERENT PARAMETRISATIONS OF CONJUGATE GRADIENT AND THEIR COMPUTATIONS TIMES

System size (l)	sl	w	cg	clock cycles	time @ 50 MHz
336	5	8	10000	15,560,000	311 ms
960	5	32	20000	17,440,000	348 ms
3840	5	64	50000	87,100,000	1742ms

gradient iterations are performed. If the error in a graph is large, conjugate gradient will take more iterations. If the number of iterations is determined by whether the error has converged, often less iterations are necessary.

The vector ALU has been simulated and synthesized with an arithmetic unit that performs 8 computations in parallel, this is equal to the number of 27-bit DSPs the synthesis results show. The system has been simulated using 27-bit signed fixed-point data types with 15 integer- and 12 fractional bits. The utilization is low, but will scale up if a larger system is used. The synthesis results corresponding to this architecture are shown in Table IV.

TABLE IV
FPGA SYNTHESIS RESULTS

	Used	Available	Utilization
ALMs	2611	41,910	6%
BlockRAM(bits)	412,784	5,662,720	7%
DSPs	8	112	7%

V. CONCLUSIONS AND FUTURE WORK

Conclusions

Using CλaSH, we have implemented GraphSLAM on an FPGA by transforming mathematical descriptions to low-level hardware descriptions. The system uses a sparse data notation and puts constraints on the number of loop closures to restrict the amount of computations. The implementation makes use of an ALU and memories, which makes it appear like an ordinary processor. This processor has an ALU that performs operations on vectors and multiple memories to support sparse data. By altering parameters in the processor, more parallelization is added. Each application could have its own constrains on computation times, which can be met by scaling up the system, while respecting the limitations of the FPGA. Although quantification of design-time is a difficult matter, the level of abstraction CλaSH offers will always pay off. With applications like SLAM, a lot of operations are involved which require an extensive amount of bookkeeping. The correct functionality can easily be tested with the simulation environment in CλaSH.

Future work

By parametrizing hardware boundaries and algorithm in terms of resources and parallel potential, the time-area trade-off can be carried out automatically, if the design is parametrizable. In this paper we show a methodology that is used to implement the GraphSLAM algorithm on an FPGA and the time-area trade-off is done manually which is a time consuming process. Design time can be reduced significantly when optimization is carried out automatically.

ACKNOWLEDGEMENT

We would like to thank the STW (Stichting Toegepaste Wetenschappen) organization for supporting the project I-Care 11854 and Christiaan Baaij for the support on and help with CλaSH.

REFERENCES

[1] G. Grisetti and R. Kummerle and C. Stachniss and W. Burgard, "A Tutorial on Graph-Based SLAM", *Intelligent Transportation Systems Magazine*, pages 31-43, 2010.

[2] G. Spampinato, J. Lidholm, C. Ahlberg, F. Ekstrand, M. Ekstrm and L. Asplund, "An Embedded Stereo Vision Module for 6D Pose Estimation and Mapping", *Proceedings of the IEEE International conference on Intelligent Robots and Systems (IROS)*, pages 1626-1631, 2011

[3] R. Konomura and K. Hori, "FPGA-based 6-DoF Pose Estimation with a Monocular Camera Using Non Co-planer Marker and Application on Micro Quadcopter", *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 4250-4257, 2016

[4] M. Gu, K. Guo, W. Wang, Y. Wang and H. Yang, "An FPGA-based Real-time Simultaneous Localization and Mapping System", *International Conference on Field Programmable Technology (FPT)*, pages 200-203, 2015.

[5] G. Mingas, E. Tsardoulias and L. Petrou, "An FPGA implementation of the SMG-SLAM algorithm", *Microprocessors and Microsystems - Embedded Hardware Design*, pages 190-204, 2012

[6] V. Bonato, E. Marques, G.A.Constantinides, "A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robot", *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pp 576–579, 2007

[7] A. Dine, A. Elouardi, B. Vincke, S. Bouaziz, "Graph-based SLAM embedded implementation on low-cost architectures: A practical approach", *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp 4612-4619, 2015

[8] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. Leutenegger, P. Furgale, R. Siegwart, "A Synchronized Visual-Inertial Sensor System with FPGA Pre-Processing for Accurate Real-Time SLAM", *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp 431-437, 2015

[9] D. Honegger, H. Oleynikova, M. Pollefeys, "Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU", *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, pp 4930–4935, 2014

[10] A. Ratter, C. Sammut and M. McGill. "GPU Accelerated Graph SLAM and Occupancy Voxel Based ICP For Encoder-Free Mobile Robots", *International Conference on Intelligent Robots and Systems (IROS)*, pages 540-547, 2013

[11] G. Grisetti, R. Kummerle, C. Stachniss, U. Frese and C. Hertzberg, "Hierarchical optimization on manifolds for online 2D and 3D mapping", *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pages 273-278, 2010

[12] H. Kretzschmar, C. Stachniss, G. Grisetti, "Efficient Information-Theoretic Graph Pruning for Graph-Based SLAM with Laser Range Finders. *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 864-871, 2011

[13] K. Khosoussi, G. S. Sukhatme, S. Huang and G. Dissanayake, "Designing Sparse Reliable Pose-Graph SLAM: A Graph-Theoretic Approach", CoRR: abs/1611.00889, 2016

[14] C.P.R. Baaij, M. Kooijman, J. Kuper, W.A. Boeijink and M.E.T. Gerards, "CλaSH: Structural Descriptions of Synchronous Hardware using Haskell", *IEEE Computer Society*, pages 714-721, 2010

[15] R. Wester and J. Kuper, "A space/time tradeoff methodology using higer-order functions", *IEEE Computer Society*, pages 1-2, 2013

[16] R.N. Appel, H.H. Folmer, "Analysis, optimization, and design of a SLAM solution for an implementation on reconfigurable hardware (FPGA) using CλaSH" Master thesis, 2016, url: http://essay.utwente.nl/71550/