Accepted Manuscript

Journal of Circuits, Systems, and Computers

Article Title:	FPGA-based Hardware Accelerator for an Embedded Factor Graph with Configurable Optimization
Author(s):	Indar Sugiarto, Cristian Axenie, Jorg Conradt
DOI:	10.1142/S0218126619500312
Received:	11 May 2017
Accepted:	22 April 2018
To be cited as:	Indar Sugiarto, Cristian Axenie, Jorg Conradt, FPGA-based Hardware Accelerator for an Embedded Factor Graph with Configurable Optimization, *Journal of Circuits, Systems, and Computers*, doi: 10.1142/S0218126619500312
Link to final version:	https://doi.org/10.1142/S0218126619500312

Journal of Circuits, Systems, and Computers
© World Scientific Publishing Company

FPGA-based Hardware Accelerator for an Embedded Factor Graph with Configurable Optimization

Indar Sugiarto[§][*]
Department of Electrical Engineering, Petra Christian University, Jl. Siwalankerto 121-131, Surabaya, 60234, Indonesia
[§]*indi@petra.ac.id*

Cristian Axenie[‡][†]
Neuroscientific System Theory, Karlstraße 45, 5. OG. München, 80333, Germany
[‡]*cristian.axenie@tum.de*

Jörg Conradt[¶]
Neuroscientific System Theory, Karlstraße 45, 5. OG. München, 80333, Germany
[¶]*conradt@tum.de*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

A factor graph can be considered a unified model combining a Bayesian network and a Markov random field. The inference mechanism of a factor graph can be used to perform reasoning under incompleteness and uncertainty, which is a challenging task in many intelligent systems and robotics. Unfortunately, a complete inference mechanism requires intense computations that introduces a long delay for the reasoning process to complete. Furthermore, in an energy constrained system such as a mobile robot, it is required to have a very efficient inference process. In this paper, we present an embedded factor graph inference engine that employs a neural-inspired discretization mechanism. The engine runs on a system-on-chip and is accelerated by its FPGA. We optimized our design to balance the trade-off between speed and hardware resource utilization. In our fully-optimized design, it can accelerate the inference process eight times faster than the normal execution, which is twice the speed-up gain achieved by a parallelized factor graph running on a PC. The experiments demonstrate that our design can be extended into an efficient reconfigurable computing machine.

Keywords: embedded factor graph; FPGA; hardware accelerator; probabilistic reasoning.

[*]Also affiliated with the Advanced Processor Technology - School of Computer Science at the University of Manchester, United Kingdom.
[†]Also affiliated with the IT R&D Division - HUAWEI Technologies German Research Center, Germany.

1. Introduction

In graph theory, there are basically two forms of models: directed and undirected graphs. Bayesian networks (BNs) fall into the first category whose the inference mechanism is considered a powerful tool that can be used for reasoning under uncertainty in many applications such as robotics. For example, a BN can be composed to link a set of variables that represent beliefs about sensory readings with other set of variables that carry control information for the robotic actuators. Another graphical model, known as the Markov network (or Markov random field, MRF), falls into the second category. The undirected graph such as MRF is better suited to express soft constraints between random variables [1], which is found to be very useful in many domains such as computer vision.

There is an increasing trend to merge/combine directed and undirected graphs into one unified formalism. This unification offers prospective treatments for applications, where the intrinsic problem in the application cannot be solved solely by a directed or an undirected graph. A special case of such a unified model is known as a factor graph (FG), which represents a function's factorization of several random variables [2]. Factor graphs also support general trends in the field of computational intelligence that spans from sequential processing to distributed processing [3].

For inference, a factor graph can use a message-passing algorithm known as sum-product algorithm or belief propagation (BP). A wide variety of algorithms developed in machine learning, signal processing, and digital communications can be derived as specific instances of this algorithm, including Pearl's belief propagation algorithm for Bayesian networks [2,4,5]. Unfortunately, the BP algorithm may demand an intense computing platform, which is impractical in many real world applications. To address this issue, we propose to use dedicated hardware to implement biologically-inspired solutions and strategies, which can also be extended for a broader class of probabilistic inference engine.

Many researchers have already proposed methods to improve the performance of graphical model computations by harnessing parallelism in modern computers [6,7,8,9,10], as well as using graphics processing units (GPUs). Silberstein et al. first demonstrated the potential of a GPU computation that impacts the performance of Bayesian networks for statistical fitting tasks using a BP approach [11]. Factor graphs have also already been implemented in a GPU [12,13]. However, to our knowledge, neither implementation nor investigation has yet been conducted on factor graphs using any dedicated hardware.

In this paper, we propose an embedded factor graph that will fit into a single system-on-chip (SoC). The SoC is also equipped with an FPGA module that can be used to provide an acceleration circuitry for the main processors of the SoC. Such an accelerator is very useful for real-time embeddes systems[14]. The main contributions of our work can be summarized as follows.

- We propose a neurally-inspired discrete factor graph useful for real-time probabilistic reasoning.

- We implemented the factor graph engine on an SoC. To our knowledge, no factor graph has been implemented on any dedicated hardware for real technical applications.
- We evaluate the optimization strategy on the system design and provide its performance measurement on a real application.

Fig. 1 shows the general overview of how we use the factor graph inference engine running on an SoC that can be used to control a robot. Our work is also driven by the flexibility requirement such that we can create many prototype networks for different application scenarios. Under this paradigm, it is more convenient to create a factor graph application as a user-friendly program, which is written in a standard C/C++ and is compiled in an operating system environment (e.g., embedded Linux). The program then calls the factor graph's library, which is synthesized on the FPGA part of the SoC to accelerate the computation. With this strategy, users of our embedded factor graph can concentrate on the modeling aspect to develop the best model for their specific application.

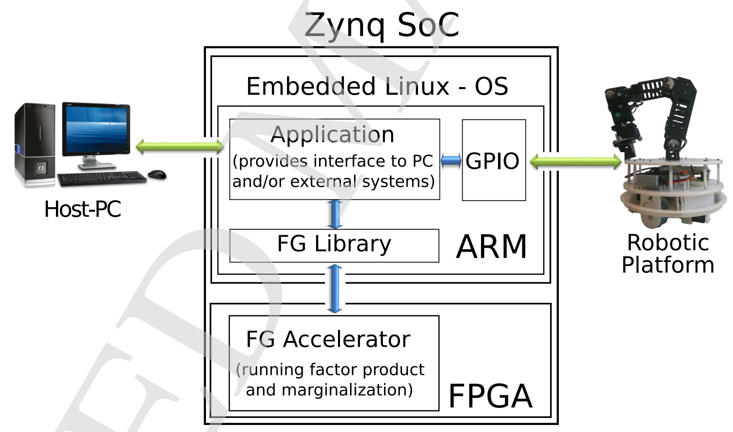


Fig. 1: The factor graph inference engine uses the FPGA part of the SoC as an accelerator. A higher level control algorithm that uses a Bayesian network or a factor graph runs on the microprocessor part of the SoC.

This paper is organized as follows. In Section 2, we provide a general overview of our factor graph framework and several related works within related domains. Afterwards, we describe our system's architecture and optimization strategy in Section 3. The paradigm introduced in Section 2 provides guidance on the evaluation of our proposed method that will be explained in detail in Section 4 for a selected example in the robotic domain. In Section 5, we provide a thorough discussion about the overall evaluation of our proposed method. Finally, in Section 6, we conclude our work and provide our vision for future work.

2. Review of Factor Graphs and Related Works

2.1. *Brief Overview of Factor Graphs*

A factor graph is a bipartite probabilistic graphical model which is composed of two types of nodes: variable nodes and factor nodes. A factor node can represent a conditional probability distribution or simply a functional relationship between variable nodes connected to it.

A factor graph has two important properties: the network's parameters and the network's structure. The parameters of the network depend on how values are encoded and decoded in the network; whereas the network structure is application specific (i.e., it can be inferred from the given tasks in the application).

A graphical model such as a Bayesian network can be transformed into a factor graph by adopting the conditional probability as the internal function of a factor node. As an illustration, consider a dynamic system that can be expressed in difference equations:

$$\begin{aligned} x_{k+1} &= \mathbf{A}x_k + \mathbf{B}u_k \\ y_k &= \mathbf{C}x_k + \mathbf{D}u_k \end{aligned} \tag{1}$$

The value of x is then calculated by summing points from initial/starting point $k = 0$ up to some value $0 < k \leq K$, and also by considering the value at x_0. Using the unrolling mechanism, Eq. (1) can be represented as a Bayesian network resulting in a chain as shown in Fig. 2a.

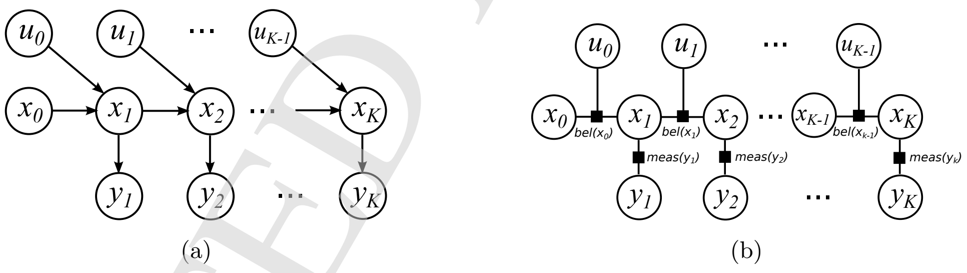


Fig. 2: (a) A Bayesian network representation for a dynamic system expressed in (1). (b) The resulting factor graph from the network shown in (a). In this representation, probabilistic random variables are denoted by circles, whereas the probabilistic relation between neighboring variables are captured in factor nodes and are denoted as solid squares.

The Bayesian network shown in Fig. 2a can be transformed into a factor graph by adding factor nodes per maximal-clique basis in order to avoid loops. The resulting factor graph is shown in Fig. 2b.

As we can see in Fig.2, basically there are two main probability functions. The first is the state transition probability, expressed as $bel(x_k) = p(x_k \mid x_{k-1}, u_k)$, which specifies how the system's internal and environmental states evolve over time as a function of control inputs u_k. The second is the measurement probability, expressed as $meas(y_k) = p(y_k \mid x_k)$, which specifies the probabilistic law according to which measurement y should be observed when the system is in the state x_k. In robotics, this measurement probability is useful for modelling not only the sensor measurement, but also for the noise which might present during the measurement. In this paper, we use this sensor measurement modelling as the example case (see Section 4.1).

Once we have defined the network structure, the parameters of each factor nodes must be obtained using a learning mechanism. In our work, we used an expectation maximization (EM) algorithm for learning. This algorithm uses inference processes iteratively as follows: each node in the graph generates an output message that will be updated consecutively after receiving messages from the neighboring nodes. We observe that this inference process is the most intense computation in a factor graph; hence, we focus our optimization strategy on this aspect. Section 3.3 describes in detail how we developed the algorithm and accelerated it in hardware.

2.2. *Encoding and Decoding Strategy*

Learning the factor graph's parameters are crucial since it involves the decision of how to encode messages' value. Our encoding strategy is based on the positional coding principle in population coding theory [15]. Here, a collection of neurons with similar characteristics will react in synchrony after stimuli [16]. We propose to use the population coding for the following reasons:

(1) By using a population of neurons with certain activation functions, the entire space can be represented compactly so that the loss of information due to quantization can be minimized. Here, we can think of a state in the discrete variable as a neuron in the population.
(2) The probability distribution produced by the population of neurons can be used to represent the uncertainty of sensory information. Since the sensor reading may be influenced by noise in the environment, it is beneficial to read the sensor data with some level of confidence encapsulated in the probability distribution.

Fig. 3 shows the basic idea of population coding that we use in our factor graph. Each neuron in that population has a specific characteristic that can be modeled using tuning curves, shown in Fig. 3b. The combined activation levels of those neurons resembles the probability density function of the given input stimulus.

In our work, we implement the population coding as follows. Consider the network in Fig. 2b; the smallest subset of the dynamic factor graph is composed of four nodes, e.g., $\{x_{K-1}, x_K, u_{K-1}, y_K\}$ (shown at the right most in Fig. 2b). For representing the joint probability $p(x_{K-1}, x_K, u_{K-1}, y_K)$ numerically, we can split the interval $I = [\text{min\_val}, \text{max\_val}]$ into i subintervals (called states) and then assign a probability value for each state and make sure that $\sum_i p_i = 1$. This way,

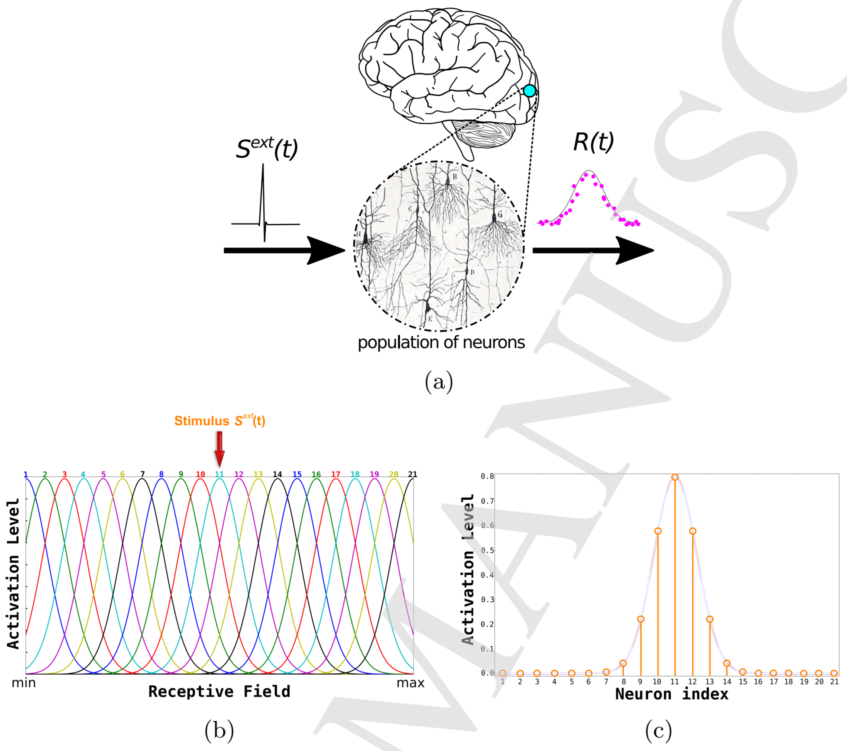


Fig. 3: The basic principle of population coding for encoding values in our discrete factor graph. (a) Conceptually, a population of neurons in a region of the brain will react in synchrony after the stimulus $S^{ext}(t)$, and produces an intrinsic response $R(t)$. (b) Neurons' characteristic modeled using several tuning curves. (c) The resulting encoded probability distribution.

each neuron will contribute to the population activity defined as [16]:

$$x(t) = \frac{\int_t^{t+\Delta t} \sum_j \sum_f \delta(t - t_j^f)\mathrm{d}t}{\Delta t {\cdot} N} \tag{2}$$

The population activity may vary rapidly and can reflect changes in the stimulus conditions nearly instantaneously. However, the activation model given by (2) is very coarse and leads to the local coding representation that yields similar characteristic to Kronecker delta discretization method. This discretization method is known to have a staircase effect [17]. To have a finer approximation, we propose to use a Gaussian function for tuning curves in a homogeneous neurons population:

$$x_i = \frac{1}{\sigma\sqrt{2\pi}} e^{-(s-\mu)^2/2\sigma^2} \tag{3}$$

where x_i is the i^th neuronal activation level of the population, corresponding to the neuron i in which the relative position of the neuron to the stimulus is represented by μ. Fig. 3b shows an example of the homogeneous tuning curves and illustrates how a real value input, which is regarded as an incoming stimulus, is encoded in a population code. The combined activation levels from each neuron then shape the overall distribution of the corresponding population (see Fig.3c).

To get the real value back from the population code representation, one might be tempted to use the winner-take-all approach since the winning neuron intuitively represents the most likely state that contributes the largest portion to the overall density. Eventhough this idea is a common perspective in the decision theory, it will not valid for a multi-modal probability density function such as the mixture Gaussian [18]. Furthermore, this approach does not always work especially in the case where the density function originates from data with a non-uniform distribution.

In our work, we use the maximum likelihood inference (MLI) in which the stimulus estimator $\hat{x}$ is obtained by maximizing the log-likelihood $p(\mathbf{r} \mid x)$, where $\mathbf{r}$ is the tuning curves function and x is the stimulus. For practical consideration, it is convenient to assume that the inter-correlation between the tuning curves can be neglected. Thus, solving this MLI will be the problem of approaching the stimulus estimator using the center-of-mass method [15]:

$$\hat{x} = \frac{\int_{-\infty}^{\infty} x \cdot p(x)\, dx}{\int_{-\infty}^{\infty} p(x)\, dx}$$

If the message containing the above information is normalized, then the denominator part can be removed. In the discrete form, the stimulus estimator is the expected value of the probability mass function:

$$\hat{x} = \sum_{i}^{n} x_i p_i(x) \tag{4}$$

where x_i is the center of the tuning curves and p_i is the activation level of the corresponding neuron. In other words, all neuron responses are integrated by using the weighted population average. This will yield an approximate inference since the real value computed using formula (4) will not be exact.

2.3. *Related Works*

Our discrete factor graph is an instance of a class in machine learning tools known as the probabilistic graphical model (PGM). Many PGM tools were designed to suit one of the forms of PGMs, either directed or undirected models [19]. Only a few of them include factor graphs in their libraries. However, many of those factor graph libraries use a standard discrete factor graph (e.g., using Kronecker delta discretization strategy) in their implementation, for example, the software package called libDAI [20], GTSAM [21,22] and BNT [19]. libDAI uses belief propagation for its inference, but GTSAM and BNT use variable elimination algorithms to perform the inference process. Those libraries implement factor graphs on standard computers (PCs) and only suitable for simulation purposes; hence, they cannot be used for practical purposes such as for real-time robot control. Furthermore, none of them implements population coding for encoding message values to run through a factor graph network.

Regarding the discretization strategy, the work by Mansinghka [23] has some similarities to ours. He created a stochastic digital circuit for Markov chain Monte Carlo (MCMC) sampling using FPGA. The main difference between Mansinghka's work and ours is that we implement the discretization for continuous variables using the population coding technique; whereas Manshinghka concentrates on the sampling algorithm from statistics. Nevertheless, both approaches work in the discrete domain because working with the propagation of continuous variable distributions may result in multidimensional integration which leads to intractable operations, especially for embedded systems with limited resources [24].

We are also aware that some researchers have already used factor graphs in a distributed computing framework to exploit Bayesian networks. For example, the work by Zhao [25] uses libDAI to implement discrete factor graphs and speed up the computation by using a parallelism framework called MapReduce [26]. Unfortunately, we could not test their method on a discrete factor graph using population coding because libDAI does not support population coding techniques.

Regarding the inference procedure, work by Andreas Steimer [27] also has some similarities to ours. However, he was focused on different levels of abstraction for implementing BP in neural substrates. His method was implemented using Liquid-State Machines (LSMs), and he applied it to Forney-style factor graphs. Our approach, on the other hand, uses BP on ordinary, but arbitrary, factor graphs with tuning-curve-based population coding. Furthermore, Steimer developed an abstract idea of hardware implementation called Interspike-Intervals-based processor; while we implemented our factor graph in real SoC hardware.

In this paper, we use a sensor fusion task as an example case. There are some efforts to incorporate the sensor fusion task into a factor graph [28,29,30]. However, their methods rely on the variable elimination mechanism for factor graphs. Our method, in contrast, uses a native- and hardware-based message passing method for the belief propagation.

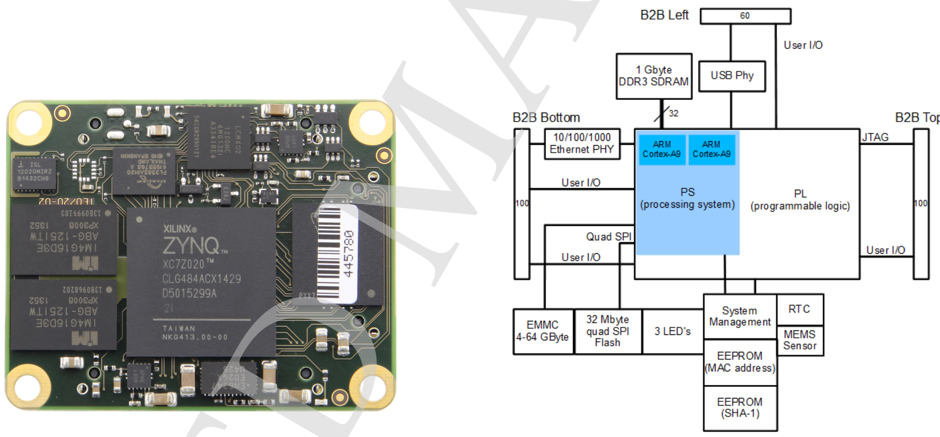
Finally, regarding the numerical and arithmetic implementation, we used the native single-precision floating point offered by the SoC vendor. However, our experience with floating-point arithmetic provided by the vendor for implementing our proposed method showed that the operation involving floating-point literals might not be optimized during synthesis. Thus, we had to inspect carefully the synthesis report produced by the FPGA synthesizer and looked for the mismatches and artefacts. This idea was inspired by the work of Tomasz Czajkowski on the optimization approach that uses functionally linear decomposition technique and dynamic power reduction [31].

3. System Architecture and Optimization Strategy

A system-on-chip is an integrated circuit (IC) that integrates all components of a microprocessor or other electronic systems into a single chip. SoCs offer extensive system level integration and flexibility, but they also impose a new challenge of integrating both concurrent and sequential programming paradigm. In the subsequent subsections, we describe how our embedded factor graph framework was developed.

3.1. *Selected SoC Platform*

In our work, we use a tailored module TE0720 produced by Trenz-Electronics (see Fig. 4). It is equipped with 8 Gbit DDR3 SDRAM and 32 Mbyte flash memories for configuration and operation.



(a) (b)

Fig. 4: The module TE0720 (GigaZee) shown in (a) physical appearance, and (b) block diagram. It has a Xilinx Z-7020 and several additional components such as a gigabit Ethernet transceiver (physical layer), 8 Gbit (1 Gbyte) DDR3 SDRAM and 32 Mbyte SPI Flash. (Source: www.trenz-electronic.de)

The TE0720 module is equipped with an SoC XC7Z020 from Xilinx Zynq-7000 family. This SoC is composed of two tightly coupled sub-systems: PS (processing system, i.e. the microprocessor core) and PL (programmable logic, i.e. the FPGA component). The PS sub-system consists of equivalently two ARM Cortex-9 processors, and the PL sub-system is equivalent with an FPGA Artix-7 from Xilinx. Table 1 summarizes the most important features of Xilinx Z-7020 that are relevant to the evaluation of our proposed design.

Table 1: Important features of Xilinx Z-7020.

Processor	Dual ARM Cortex-A9
Logic Cells (LC)	85K Logic Cells
BlockRAM (BRAM)	560 KB
DSP Slices	220

3.2. *FPGA-Accelerated Factor Graph Engine*

Running an intense computation on hardware can improve the overall performance of the system. We also agree that the most effective way to increase the efficiency is by exploiting application's characteristics in hardware [32]. In our work, we implemented this idea such that the main application will run on the PS component of the SoC, whereas the factor graph inference engine runs on the PL . The application will invoke some factor graph functionalities (as described in [33]), and the factor graph library manages all the nodes and the scheduler used for the message-passing. In addition, the PS component of the SoC will be responsible for communication with external devices such as the host PC (to support further data analysis) and/or the robot (which will be controlled by the factor graph).

During the inference process, the PS will send message values to the PL. In the PL, messages will be processed with the sum-product algorithm. This algorithm, which performs many for-loop-based computations, are implemented in a parallel fashion by pipelines and by unrolling a block of the code. Once the computations have been completed, the results will be sent back to the PS and will be delivered to the external devices or propagated to different nodes within the respective factor graph. Fig. 5 shows the block design of our embedded factor graph with an accelerator.

To communicate factor graph messages between the PS and the PL components, the AXI protocol was used. The factor graph messages from the PS were sent to the PL (and vice versa) in a form of an array. To facilitate computations with the array, internal memory units (either distributed block RAMs (BRAMS) or looked-up tables (LUTs)) must be included in the design. In most parts of our implementation, BRAMs were used instead of LUTs because of the high cost of LUTs usage (although BRAMs are a bit slower than LUTs).

The accelerator module (labeled as SumProductAcc shown in Fig. 5) facilitates the interrupt mechanism; when the factor product or the marginalization is completed, the PL will send an interrupt signal which will be intercepted by the accelerator module driver in the Linux running on the PS. The Linux kernel then notifies the factor graph program that the sum-product acceleration has been completed and the factor graph's message can be fetched from the PL. Fig. 5 also shows that the SumProductAcc module has a special type of input called factor_PORTA that facilitates a direct access to the ROM that contains the internal function of a factor node. This internal factor's function must be supplied at the beginning of

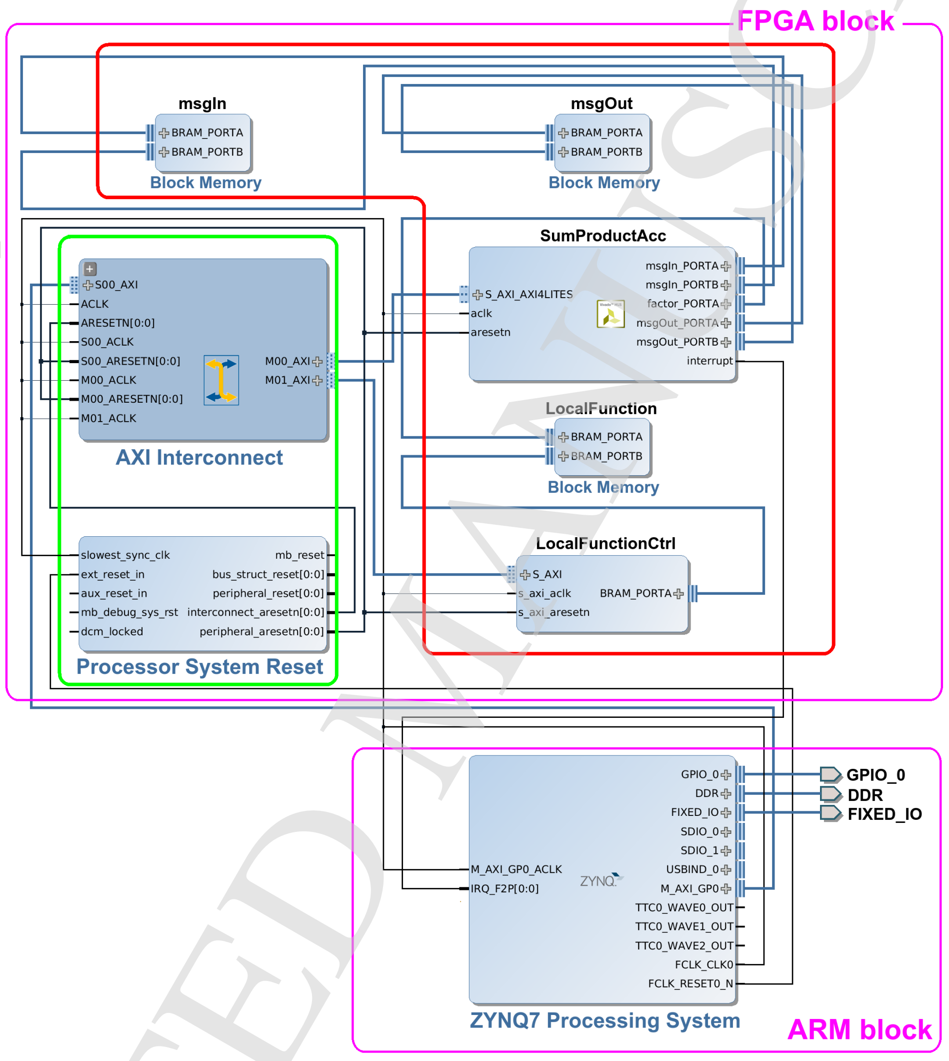


Fig. 5: Block design of the factor graph engine in a SoC with FPGA as the accelerator. All modules in the upper region are implemented in the FPGA while the lower region represents the ARM processor of the Zynq-7000. The modules within the red block are the main elements of the accelerator while the modules within the green block are supporting elements that connect the accelerator to the ARM processor.

the program execution during the loading of the device driver that was generated by the synthesizer. The number of the functions may vary (depending on the factor graph network being instantiated in the PS) and can be determined by the memory address that has been allocated for the SumProductAcc module. The size of each factor's function depends on the number of scope variables and the cardinality of each variable. In this paper, this cardinality is referred to as the number of states in the population coding.

3.3. *Optimization Strategy*

Regarding the area optimization with respect to the storage/memory allocation in internal resources of the FPGA, the main trade-off usually lies on the choice between using the basic logic gates (LUTs) or BRAMs. Although it is possible to use external memory, we prefer to avoid this method since external memory access is an expensive task in terms of FPGA resources. Controlling external memory needs explicit routing strategies in order to match the interfacing protocols and timing constraints required by the memory hardware [34]. In our work, we strive to optimize our design by only instantiating memory elements either on LUTs or BRAMs.

Table 1 shows that Z-7020 has limited BRAMs and it should only be used when the latency is not the main issue since the distribution of BRAM units within the chip is sparse. In contrast, the LUTs will provide the fastest response (i.e. lower latency), since LUT-based memory can be allocated right next to the computing cores. However, LUTs are the elemental logic units necessary for implementing the core elements of a factor graph. Many parts in our algorithm require accesses to memory units in a form of an array. The Array is a basic construct to express a memory access in Xilinx Vivado-HLS. The optimization strategy for arrays includes reshaping and partitioning. By optimizing the array (either reshaping or partitioning), the data transmission bottleneck can be avoided. Fortunately, Xilinx Vivado-HLS provides a convenient way to handle this array optimization that helps us to inspect and analyse the resource usage/consumption for later optimization.

Regarding the speed optimization, our approach is mainly based on the idea of exploiting the "unbounded" parallelism paradigm in the sense that we can parallelize any task, in any degree, in a resourceful FPGA. In our work, we used two important optimization scenarios: unrolling and pipeline. The unrolling mechanism provides intrinsic/naïve task parallelisms for achieving high-speed performance; whereas pipelining mechanism provides behaviouristic parallelism by breaking down a sequential process into sub-operations, and then pushed them into a series of independent processes as hardware blocks. We give an example of how to use these unrolling and pipelining in our algorithm-1.

Using these scenarios, there was a trade-off during each loop's iteration on which we had to make a balance between the hardware state and the hardware resources. We used two metrics to measure the efficiency of our optimization approach: clock latency (which indicates the success of our speed-based optimization) and resource consumption (which measures how well our area-based optimization has been carried on by the synthesizer).

In addition to these two optimization scenarios, there was another important aspect that needed to be considered. When implementing our factor graph framework in a PC, we used double-precision floating point values so that we could get the best or the smoothest result for the inference using belief propagation. Unfortunately, this double-precision was very expensive in terms of hardware resource usage in the FPGA. Hence, we used single-precision floating point values. Although basically we can use any number of bits, the Xilinx synthesizer restricts the use of such an approach and only optimizes a design that uses 32-bit representation. As a result, we could not perform any further optimization in this floating point aspect.

As an alternative to floating point, we could also use the fixed point format, which is faster than floating point for some operations. However, we found that the fixed point arithmetic produced coarse results, which will be less useful in real robotics applications. Moreover, its dynamic range is also limited. For example, it is very difficult to get a large real value number (higher than 100.0) and a very low probability value (less than 0.000001) at the same code using the fixed point arithmetic. Hence, we argue that floating point is the best choice for our factor graph implementation in the SoC, even though it consumes many its FPGA logic resources. Furthermore, we can see the trend of SoCs (and FPGAs) becoming denser and relatively cheaper; hence, this issue will not become a problem in the future.

All of the main core modules of our factor graph engine were developed using Vivado-HLS. Only a small subset of logic elements, which are small but frequently used, were written purely in VHDL in order to reduce total latency and to achieve higher area optimization. Beside this Vivado-HLS, Vivado Suite from Xilinx also has an SDK (Software Development Kit) framework which is very useful to create an embedded application. We used Vivado SDK for developing our factor graph-based controller which runs under embedded Linux system. The diagram in Fig. 6 shows the workflow of our SoC-based embedded factor graph design.

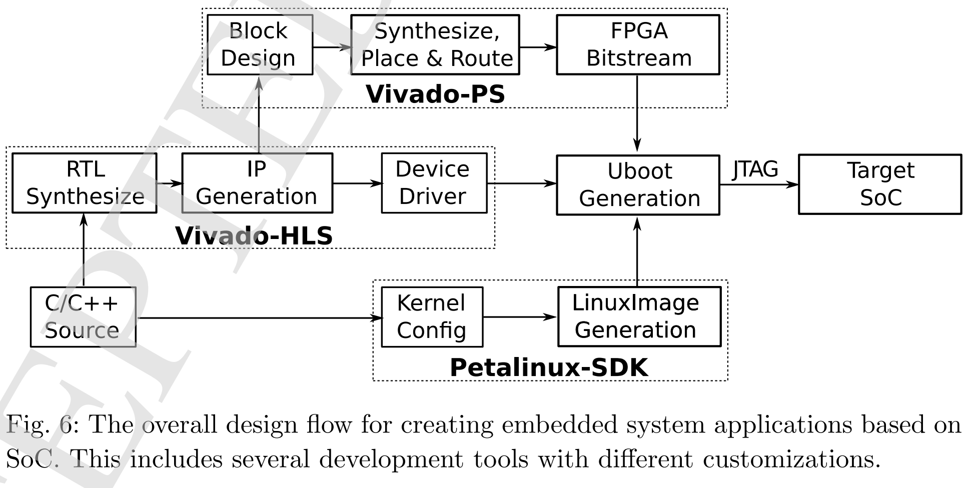


Fig. 6: The overall design flow for creating embedded system applications based on SoC. This includes several development tools with different customizations.

In our work, we used Petalinux as our embedded Linux platform and made the system to run in kernel space; hence, the driver will be loaded automatically when the Petalinux starts. This will be a preferable choice for most future users of our factor graph module due to its simplicity and flexible instantiation. Furthermore, this will facilitate various experiments regarding the learning process of the instantiated factor graph module. For example, the EM algorithm for our factor graph requires unique scheduling mechanism that will be easily accommodated by using system-wide timing mechanism for the updating process.

Regarding the belief propagation setting for EM, the network will propagate messages iteratively during which the network's parameters are regularly updated. Iteration in EM consists of two steps: the expectation update for the log-likelihood given the old parameters and the observed data, and the maximization procedure to update the parameters. Normally, the expectation update of the log-likelihood is computed as follows:

$$E[\log p(\mathbf{X})|\mathbf{Y},\theta] = \sum_x p(\mathbf{X}|\mathbf{Y},\theta)\log p(\mathbf{X}|\theta) = E[\log \prod_i (\frac{1}{Z}\prod_a f(X^i))|\mathbf{Y},\theta] \quad (5)$$

Here f denotes an internal function of a factor node in a factor graph and $X^i = x$ indicates a specific variable configuration (i.e. state) for this function. Hence, $f(X^i = x)$ corresponds to a single parameter of that function.

In our work, we used a modified version of an EM algorithm. It is inefficient to calculate $f(X^i = x)$ by enumerating all configurations of the arguments. Rather, we need to find a configuration with the maximum probability value and then spread the distribution according to the population code's variance. Since the EM algorithm is an iterative approach and we used population coding instead of Kronecker delta function, we used Kullback-Leibler (KL) formula to measure the divergence level of the newly learned parameters. This KL divergence has the following basic form:

$$D_{KL}(p \parallel q) = \sum_x p(x)\ln\frac{p(x)}{q(x)} = \sum_x p(x)\ln p(x) - \sum_x p(x)\ln q(x) \quad (6)$$

By using Jensen's inequality theorem, we know that $D_{KL}(p \parallel q) \geq 0$ with equality *iff* $p = q$. We applied the KL measure on the difference between the new probability distribution ($p(x)$) and the old probability distribution ($q(x)$) about some threshold value as the stopping criteria. The iteration was stopped when this KL measure was fulfilled or when the $MAX\_ITERATION$ value was reached. The algorithm for learning the parameters using this EM approach is shown below.

As we can see in algorithm-1, there are two loops in which we can apply the unrolling and pipelining. The outer loop (started at line-2 and terminated at line-16), the algorithm will be fed with certain data points in which the parameter Θ will be accumulated. Normally, this is a sequential process, since the resulting Θ will be used for comparison using (6). However, we can break down this process into sub-tasks and then pipeline them. This is possible because fetching data and storing the result are independent of the computation itself. This pipelining mechanism is

Algorithm 1: Estimate factor parameter θ using EM

```
1  $\Theta \leftarrow uniformly\ distribute$;
2  forall sample in $X$ do
3      [oRead] $\phi, \phi_{old}, \phi_{new} \leftarrow uniformly\ distribute$;
4      [oComp] for $i = 0$ to MAX_ITERATION do
5          for $j = 0$ to $k$ do
6              [iRead] get sample data;
7              [iComp] compute product($\phi^k_{new}$);
8              [iWrite] $\phi \leftarrow \sum \phi^k_{new}$;
9          end
10         compute $diff = KL(phi_{old}, phi)$;
11         if $diff \leq THRESHOLD$ then
12             $\phi_{old} \leftarrow \sum \phi$
13         end
14     end
15     [oWrite] $\Theta \leftarrow \phi_{old}$
16 end
17 normalize $\Theta$;
18 return $\Theta$;
```

shown in Fig.7a. We denote this outer loop tasks as oRead, oComp, and oWrite respectively.

The inner loop (started at line-5 and terminated at line-8) within an intermediate loop (started at line-4 and terminated at line-14) is responsible for computing the factor product. This is a completely independent computation due to independence between neurons in the population coding; hence, we apply unrolling on this section. The number of parallel hardware instantiations depends on the cardinality of the factor graph. This unrolling mechanism is shown in Fig. 7b. We denote the computing blocks inside this inner loop as iRead, iComp, and iWrite respectively.

In our work, we used Xilinx Vivado HLS (High-Level Synthesis) to create modules using C++ syntax. For example, we created a module SumProductAcc (shown in Fig. 5) that provides the acceleration for the sum-product computation. Using Vivado HLS, the unrolling and pipelining mechanisms can be conveniently implemented. Fig. 8 shows how the product operation in line-7 of algorithm-1 was implemented.

Shown on Fig. 8, the `#pragma` directive with `HLS PIPELINE` command is inserted at the beginning of the inner loop, which instructs the synthesizer to pipeline the loop with minimum Initiation Interval. On the other hand, the #pragma directive with `HLS unroll` command is inserted at the beginning of outer loop for unrolling the outer loop. Loop unrolling creates more operations in each loop iteration to achieve higher throughput and system performance.

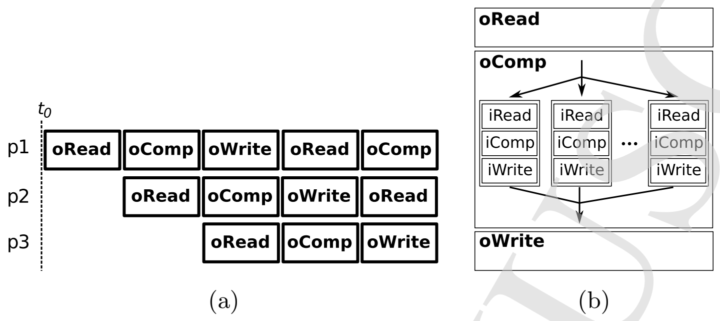


Fig. 7: Unrolling and pipelining mechanism for the algorithm-1, exhibiting: (a) pipelining outer part, and (b) unrolling inner part. We used 3-stage pipeline (p1 to p3) since we only had three separable processes, and we used 10 to 20 instantiations (due to current hardware limitation) for the unrolling.

```
#define MAX_VAR 4
#define UROL_FACTOR 4

extern int nodeIdx;
extern float msgIn[];
extern float msgOut[];
extern float factor[];

void fproduct(int cardinality, int nScope)
{
    int N = cardinality * nScope;
    int ScopeDim[MAX_VAR] = {cardinality};
    int checkInScope;
    for(int j=0; j<cardinality; j++)
    {
#pragma HLS unroll factor=UROL_FACTOR
        for(int i=0; i<N; i++)
        {
#pragma HLS PIPELINE
            int stride[MAX_VAR] = {1};
            for(int k=0; k<nScope; k++)
            {
                stride[k] = cardinality * stride[k-1];
            }
            checkInScope = k/stride[i] % ScopeDim[nodeIdx];
            if(checkInScope)
            {
                msgOut[j] = msgIn[j] * factor[i];
            }
        }
    }
}
```

Fig. 8: A snippet of a code that shows how unrolling and pipelining are used to implement a factor product used in algorithm-1.

4. Experiments and Evaluation

In this section, we evaluate the performance of our proposed implementation of a discrete factor graph inference engine, and use a test case in a robotic domain. To keep this paper concise, we use only the sensor fusion part from our whole robot model to give an intuitive example of how elegant a factor graph can be used to deal with uncertainty in the real world.

4.1. *Example Case: Sensor Fusion for Robot Navigation*

The robot is equipped with several sensors; two of them are a gyroscope and a compass. Here, we want to use those sensors to give information about the orientation of the robot (i.e., the heading of the robot). The use of both gyroscope and compass is important, particularly when the robot works in an outdoor environment where there is no overhead sensor that gives information about absolute robot's pose. The experimental data for this example case is based on our previous work that used our robot shown in Fig. 9a. Readers are referred to [35] for the detailed information about the data collection and its pre-processing.

The factor graph models of the sensor fusion network are shown in Fig. 9b and Fig. 9c. The goal of this sensor fusion is to combine measurements from a set of different sensors to improve the quality of the perception about the state of the world.

In our experiment, the robot was placed in a room with an overhead camera tracking system that gave the "ground-truth" data, which was also useful for calibrating the robot's sensors. Fig. 10a shows how the data was collected and Fig. 10b shows the corresponding sensor data when the robot was driven to follow a certain pattern (shown in Fig. 10a).

Further analysis of the data reveals important information regarding the sensors' characteristic during robot movement. Since the overhead camera tracker provides the absolute robot's pose with high resolution, we regarded this information from robot tracker as the "ground-truth". Fig. 10c and Fig. 10d show the correlation between the gyroscope and the camera tracker, as well as between the compass and the camera tracker. As can be seen from those figures, both sensors produce noisy measurements and distorted linearity. Using the gyroscope alone or the compass alone in an outdoor environment will put the robot in a higher risk of instability. By fusing the information from those unreliable sensors, the robot is expected to infer its correct pose.

Our sensor fusion network works as follows. The gyroscope data is denoted as sensor-G, the compass data as sensor-C and the direct/absolute measurement from the camera tracker as sensor-T. The factor graph network corresponding for fusing these three sensors is shown in Fig. 9b. For the training of the model, we used the data collected in [35]. During the training phase, the factor graph learned the joint probability distribution as the internal function of its factor nodes using the EM algorithm shown in Algorithm-1.

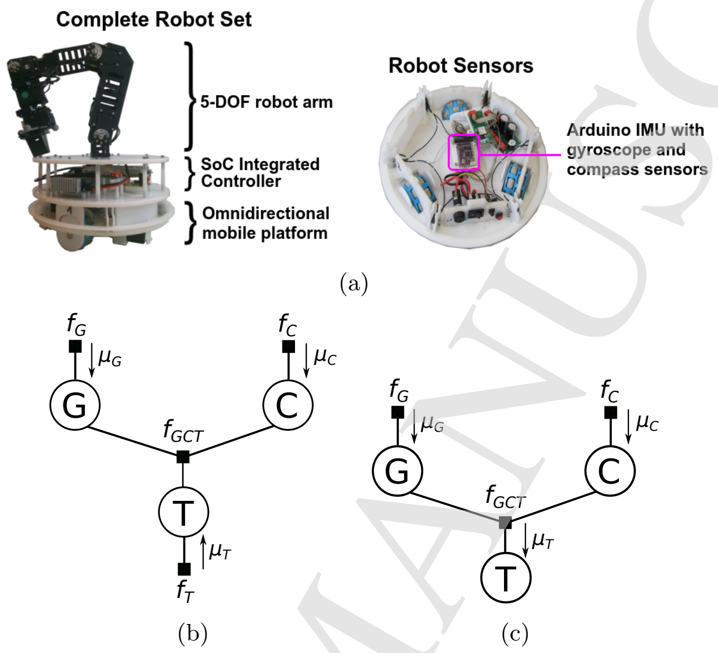


Fig. 9: (a) Our mobile manipulator, which is equipped with several sensors. (b) The factor graph network for the sensor fusion task. (c) The network during inference phase. The evidence enters the network through factors f_G and f_C, and propagate until reaching the output node T.

Once the training phase was completed, the model was used for reasoning by simply unclamping the target variable. The inference procedure estimated the correct belief about the sensory reading by fusing the data from the available and connected sensors. To put it differently, this is basically a reversal process to estimate the "ground-truth". In this case, we removed the input factor node for the variable-T and let the messages from variable-G and variable-C flowed through the network. As a result, the network inferred the expected robot's pose. This inference process is depicted in Fig. 9c.

The final output from the inference process was obtained by marginalizing the messages running towards node-T. To evaluate our model, we generated a test set containing two sensor values (sensor-G and sensor-C) with a distorted sinusoidal shape. The distortions were introduced by generating Mackey-glass chaotic data and were added to the sensor values. In order to simulate the real sensor characteristics, both simulated sensor data were adjusted with a correlation factor of 0.9. Those simulated sensor data were fed to the fusion network which produced the result shown in Fig. 11. It shows that the estimated robot's orientation has some degree of confidence level (depicted as the variance along the estimated result). As

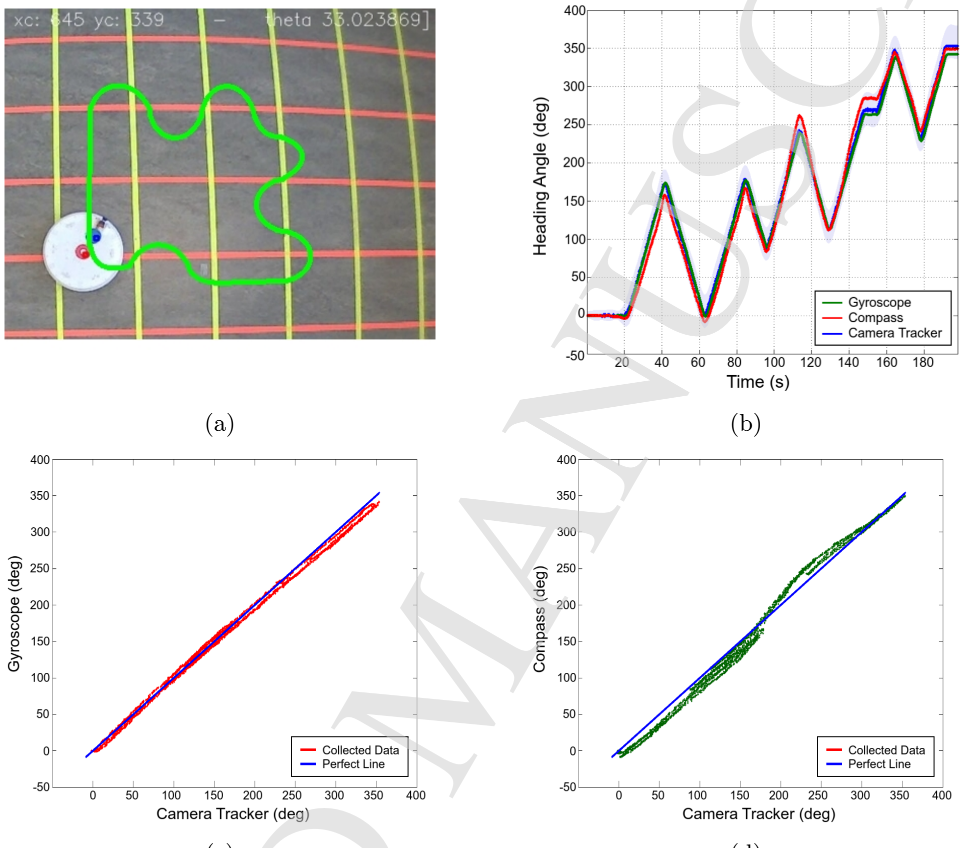


Fig. 10: The robot experiment to collect data for the sensor fusion network. (a) The robot followed certain trajectory during which the sensory data were collected. (b) Collected data that corresponds to the robot trajectory in (a). (c)-(d) Using the absolute values from the camera-tracking system as the "ground-truth", data from gyroscope and compass are evaluated. It can be seen that the robot has an unreliable sensory reading that introduces distorted linearity.

a comparison, another inference using a standard Kalman filter for sensor fusion was also performed. As we can see in Fig. 11, our sensor fusion network produced a smoother trajectory than the standard Kalman filter at some regions. We argue that this smoother feature is a result of a better generalization on data with bigger fluctuation disparity, whereas Kalman filter performance is very close to an averaging technique.

In this example, we demonstrate that our factor graph network can produce a smooth estimation of robot's orientation given noisy sensor readings. By fusing the information from two unreliable sensors, the robot will be provided with more

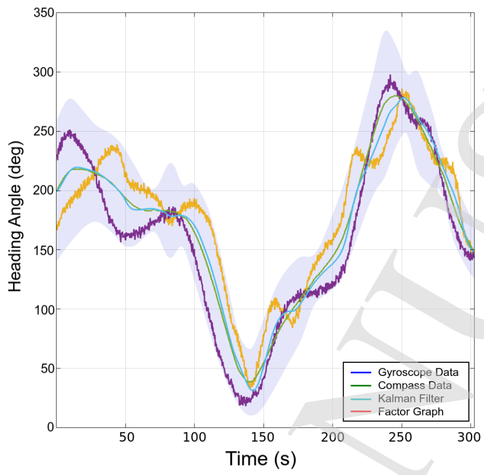


Fig. 11: Sensor fusion results. Using pseudo-random data as sensor values, the network is tested for its capability to reason under uncertainty. The network tries to estimate the underlying "ground-truth" of the robot's orientation from the given sensor reading.

reliable information about its state and its environment. With this excellent result, we believe that it can be extended into a more complex task that includes more sensor modalities. We argue that this is an elegant approach because the network can be easily and intuitively extended without sophisticated additional nodes in the structure. This is also favorable for low-level hardware implementation of belief propagation in our approach. For example, we can add one more measurement for the robot heading from the odometry sensor of the robot, e.g., from the robot wheel encoder (see Fig. 9a). The network will now have four variable nodes, as shown in Fig. 12.

4.2. *Performance Evaluation*

We measured the efficiency of our method using the standard metric commonly used in FPGA-based designs; i.e. the clock latency for measuring the speed optimization result and the resource consumption in percentage for measuring the area optimization result. It has been described in Section 3 that the optimization on one aspect (e.g. the area optimization) will affect the other aspect due to tightly coupled resource constraint of the FPGA. In this paper, we are interested in exploring the balance of these aspects in order to find the best solution for our factor graph modules.

For evaluation purpose, we use the networks shown in Fig. 9b and Fig. 12. All of the variable nodes in the networks are observed; each node will have its own corresponding factor input (e.g., G will be connected to f_G, C will be connected

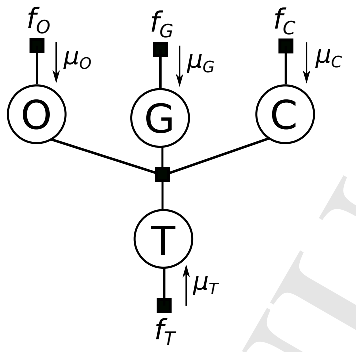


Fig. 12: The new sensor fusion network. The new measurement unit is added as a variable node, whereas the factor node in the center now contains the joint probability distribution of all four nodes.

to f_C, etc.) As we have explained in Section 3, we used unrolling and pipeline techniques for optimizing our design. Application of the unrolling and pipeline mechanisms requires the loop to be perfect or semi-perfect. A perfect loop means that the loop bound is constant, whereas a semi-perfect loop might have a variable bound and needs to apply an exit check protocol. In both circumstances, we need to specify the cardinality of the variables in the source code before synthesizing it. Table 2 to Table 5 provides comparisons of our framework implementation scheme with and without optimization.

Table 2: Clock latency comparison between the optimized and unoptimized design of the network with three variables (shown in Fig. 9b).

	Unoptimized		**Optimized**	
States	min	max	min	max
10	688	76129	675	34400
20	2574	284244	2530	120978

Table 3: Clock latency comparison between the optimized and unoptimized design of the network with four variables (shown in Fig. 12).

	Unoptimized		**Optimized**	
States	min	max	min	max
10	9462	442182	9279	160052
20	37848	1768728	36984	628728

Table 4: FPGA resources consumption (in %) in the optimized and unoptimized design of the network with three variables (shown in Fig. 9b).

	Unoptimized			**Optimized**		
States	BRAM	FF	LUT	BRAM	FF	LUT
10	16	4	11	16	8	20
20	32	7	24	32	14	42

Table 5: FPGA resources consumption (in %) in the optimized and unoptimized design of the network with four variables (shown in Fig. 12).

	Unoptimized			**Optimized**		
States	BRAM	FF	LUT	BRAM	FF	LUT
10	48	9	27	48	19	51
20	96	17	52	96	39	83

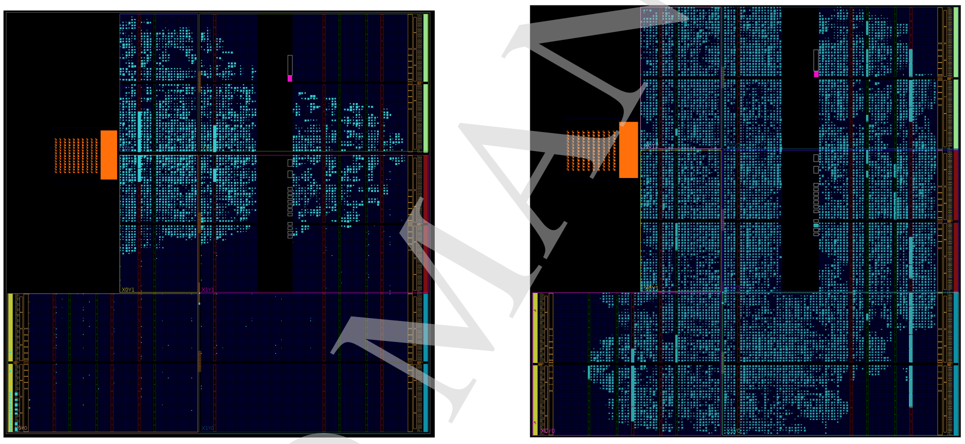
In Table 2 and Table 3, the values in the min and max columns reflect the minimum and maximum clock latencies that are required to move from one state to the next state in the FSM (finite state machine) implementation of the algorithm. From these values, we can estimate how long it will take for the algorithm to run. These values are only estimations based on the given clock frequency in the synthesizer program (Vivado-HSL) and not the real clock frequency of the hardware. For example, in our design we usually specify the clock frequency to be 100 MHz; the value of 688 means that it takes 6.88 μs to complete the execution. In real hardware implementation using SoC Zyng-7000, where the FPGA's frequency clock can be set up to 628 MHz using an external clock source, the latency value of 688 is estimated to be completed in 1.1 μs. Likewise, for a maximum latency value of 1768728, it will take roughly 17.687 ms with 100 MHz clock systems.

In Table 2 and Table 3, we can see that the minimum values do not differ much for both the optimized and the unoptimized designs, revealing the fact that there are some parts of the code that cannot be further optimized. Usually, these values are related to the inter-block data exchange in the code. The maximum values, on the other hand, show a notable difference between the optimized and the unoptimized designs. Dividing the maximum value obtained from the unoptimized design by the value from the optimized design showed an average speed-up ratio of 2.53. We also observe that the clock latency is heavily affected by the number of states used to encode a factor graph's message, which increases exponentially.

Table 4 and Table 5 show the optimization efficiency of the design with respect to the number of FPGA resources consumed by the design. BRAMs are the distributed memory units mainly used for instantiating arrays in our design. The FFs (flip-flops) and LUTs are the main constituents of the configurable logic block in the FPGA for implementing the arithmetic and logic operations.

Table 4 shows how much resources were utilized for the network shown in Fig. 9b.

With only three variables connected to a factor node, we could optimize the design for up to two independent networks, either using 10 or 20 states for the variable's cardinality. Fig. 13 shows the internal routing and mapping of FPGA resources for the fully optimized design used by the network in Fig 9b. As can be seen in Fig. 13a or Fig. 13b, there are still some free spaces for the network to use a higher number of states. In our experiment, the network with 50 states was still synthesizable in an unoptimized version with the maximum latency of about 214056 clock cycles (which corresponds to the predicted execution time of about 0.32 ms in real hardware). The resource consumption is also increased up to 65% for LUTs and 28% for the FFs.



(a) (b)

Fig. 13: Inside the chip: the factor graph accelerator program will be translated and mapped into FPGA resources (BRAM, DSP, FF, and LUT) and scattered all over the chip to match the routing policy of the synthesizer. In (a), the floorplan was produced for the fully-optimized design with 15 states, while in (b) it was produced by using 20 states.

Table 5 reveals the fact that the network shown in Fig. 12 requires more resources than the network shown in Fig. 9b. It can be seen in the table that if the unrolling and pipeline mechanisms are not used, two independent networks with 10 states for each variable can be created. However, if the optimization mechanisms are implemented, only one network with 20 states can be created. In the full optimized design, almost all of the resources are consumed by the network shown in Fig. 12.

For the evaluation of the run-time execution, we performed a complete inference test using the network shown in Fig. 9b and using the dataset from [35]. The inference task, in this case, was computing the marginal probability of variable T given input data for variable G, and C. For this test case, we created the network and ran the belief propagation on it using only one processor of the SoC. We fed the dataset to the network, collected the inference result and sent it to the host PC for evaluation. Next, we modified the program to use the accelerator, re-created the network and re-ran the belief propagation using the same dataset as before. The resulting data from the inference was also sent to the host PC.

The combined result of these two runs is shown in Fig. 14. As expected, the accelerator can speed up the factor graph computation with a ratio almost reaching 8-times higher than the normal run when the variable's cardinality is 25. This demonstrates that the optimization strategy in our module was implemented successfully. Unfortunately, we could not test the four-variable network shown in Fig. 12 with greater than 20 states using our current hardware, but we argue that our accelerator can be extended further, given a denser FPGA part of the SoC (e.g., the moderate SoC chip in the Zyng-7000 family, that is the Z-7035, has a capacity as much as four times of our Z-7020). This is a contrast with the factor graph without an accelerator, which runs only on the microprocessor of the SoC. Without the accelerator, we can use any number of variable's cardinality, but at the cost of slow performance.

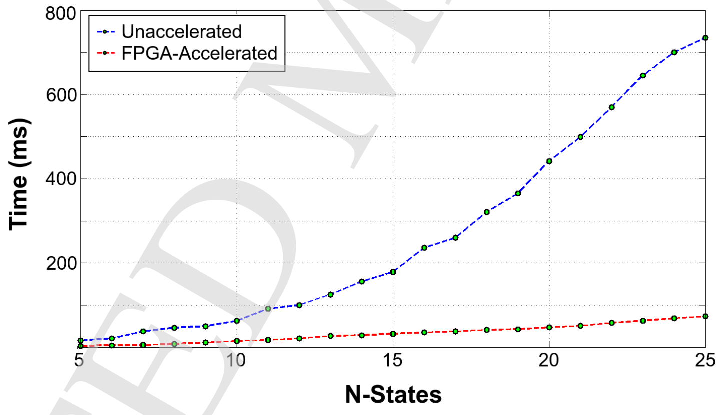


Fig. 14: Comparison of inference execution performance between accelerated mode and non-accelerated mode by FPGA in the SoC.

To emphasize the excellence of our method, we tested our design against different parallelism approaches running on a standard computer (PC). In particular, we are interested in implementing our population-coding-based factor graph in a multi-core PC as well as using a GPU (graphics processing unit). The current trend in such parallelism platforms enables us to quickly implement our algorithm and gain benefit of parallelism such that we can get the faster result for analysis. However, parallelizing our factor graph framework in a PC-based machine is not our goal. For example, we did not utilize the full multigrid threading of the GPU card to implement our factor graph node-by-node. Instead, we just used the GPU as an accelerator for some of the most intensive computations in the belief propagation algorithm.

Table 6 and Fig. 15 show the comparison of our factor graphs running on different platforms. We can see that our SoC-based factor graph engine outperforms the standard parallelism approaches running on PCs in terms of speed-up gain per watt (our SoC board runs on power less than 5 Watt, whereas our PC needs at least a 650 Watt power supply). On a PC, the factor graphs were implemented using three different parallelism strategies: Matlab's Parallel Toolbox, OpenMP, and GPU-CUDA. The PC has Intel-i5 running at 3.30 GHz with 16GB SDRAM-DDR3 running at 1.3 GHz. It was also equipped with a graphic card GeForce GTX-650 with 384 CUDA-cores and 1 GB DDR5 running at 1 GHz. As we can see from the table and the figure, the SoC implementation outperforms the other implementations but with one inevitable disadvantage: using our current SoC hardware (XC7Z020), it cannot run with a very high number of states. We can achieve 50 states in a partial optimization configuration by using only unrolling without pipelining, and when we use both unrolling and pipeline mechanisms, we can only achieve 25 states (see Fig. 13b). However, the speed-up gain of the fully-optimized design far exceeded the other parallelism strategies.

One important aspect that we observed when using these PC-based parallelism platforms was about the data preparation. We found that the speed-up gain was heavily influenced by the way the data were prepared apart from the algorithm itself. We believe that this is because the underlying parallelism method has its own mechanisms for handling potential software bugs introduced by the concurrency process, and we have to follow its rules. Problems such as race condition and mutual exclusion in the OpenMP scenario need to be handled properly in order to make sure that the results are consistent with the standard/normal way of running the algorithm on a PC. Also, communication and synchronization between the different threads are the most difficult tasks to handle in the first place to get the best performance of the program running in parallel. These issues contribute to the phenomenon related to the maximum possible speed-up of a single parallelized program known as the Amdahl's law. Furthermore, for the GPU-CUDA version, the overhead of transferring small data from the host to the device hinders the powerfulness of a Single instruction, multiple data (SIMD) capability of such a multi-core graphic card.

5. Discussion

The experimental result and its evaluation in Section 4 showed that our accelerator module worked impressively and can arguably be extended into a more powerful module in a denser chip. Currently, our hardware cannot handle a large network with high variable's cardinality, but the acceleration result does not scale down with the size of the network's parameters.

Table 6: Comparison of the full inference time (in second) during the training phase of the network shown in Fig. 9b on different platforms.

N-States	Std-PC	Matlab	OpenMP	GPU	Std-SoC	Acc-SoC
5	4.946	5.339	4.633	5.573	20.770	3.100
10	8.155	5.923	5.856	5.906	54.020	7.400
15	13.445	5.991	6.042	6.004	101.250	12.500
20	22.167	7.055	6.518	6.921	152.150	17.900
25	36.547	11.654	7.833	10.699	232.960	25.600

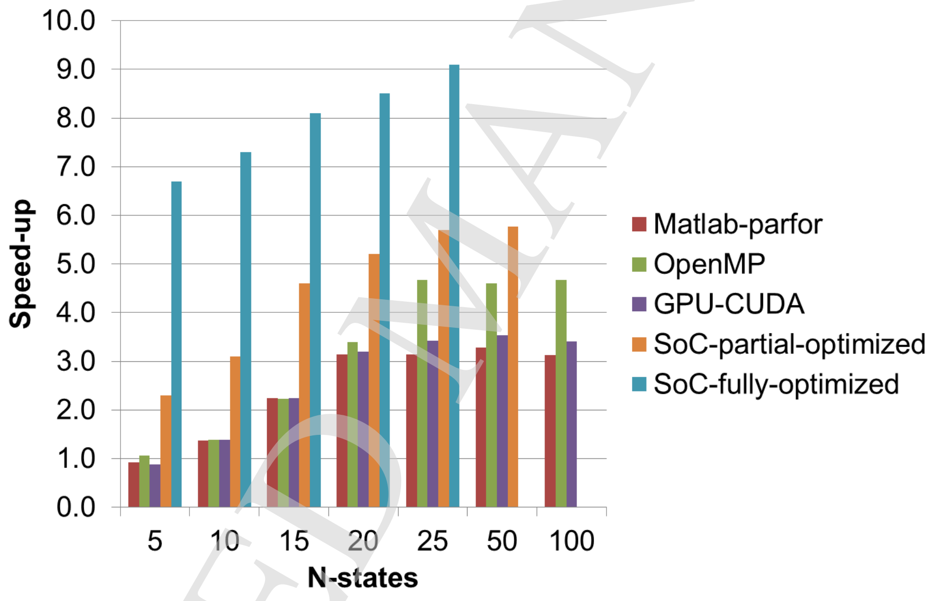


Fig. 15: The performance comparison of our factor graphs for the network shown in Fig. 9b. The PC-version of our factor graphs were implemented using three different platforms: Matlabs Parallel Toolbox, OpenMP, and GPU-CUDA. We consider our method of implementing factor graphs on an SoC as a true fine-grained parallelism since it can optimally make use of the abundance FPGA resources.

The experiment showed that the accelerator running on a single SoC module can produce optimal results for networks with a factor node having three scope variables. Adding more connected variables to the corresponding factor node requires the module to be slightly modified because we have to allocate more memory space in the LUTs instead of the BRAMs. Also, our current implementation of the accelerator still needs bridging access via the microprocessor to the external RAM where the factor parameters are stored. This configuration slows down the performance. Another solution might be the use of a direct memory access (DMA) to the external memory. This is an interesting idea that needs to be explored further in our future work. However, the DMA access from the PL component requires the use of a special intellectual property (IP) core for handling this mechanism. To our knowledge, the IP core for using DMA via AXI bus will consume a considerable amount of FPGA resources (up to 10%) which is impractical for our current hardware. Considering this trade-off, we decided to rule out this idea in our current implementation.

Deploying programs in an embedded system, especially the ones with intrinsic parallelism, requires different treatments and explicit considerations. The experimental result and our analysis in the previous section reveal that our proposed framework exemplifies the important aspects of hardware-software co-design paradigm, which are: flexibility and platform-friendliness. Our framework proves to be flexible enough and reconfigurable for robotic applications such as a sensor fusion without too many modifications to the framework. Its flexibility is also demonstrated by its simple transitional step from a standard graphical model-based application to our embedded factor graph.

Readers might observe that our method does not produce a ready-to-go design with which a non-hardware developer can use without touching too much the design entry in the hardware domain. This is a common circumstance in SoC- and FPGA-based systems which always require resynthesizing and, eventually, regenerating the bitstream of the design. It does not mean that once we have generated the bitstream, the hardware-side development is done. We still need to make sure that the application programmer can use our hardware. Indeed, the bitstream generation triggers the next step in embedded system design: developing the driver which is accessible through a simple API (application programming interface) to make it more developer friendly.

Finally, we also agree with the conclusion made by Juan Carlos et.al in [36] stating that in general, FPGA-based design for real robotic application is difficult due to two main reasons: difficulty in changing platform's functionality (very often requires specialized person) and tools system dependency. However, we also believe that our embedded factor graph framework on SoC has a prospective future, because we see an increasing trend to bring the SoC and FPGA design into a heterogeneous computing platform (such as SystemC, OpenCL, etc.) and also an increasing effort to bring the embedded Linux kernel into the Linux mainstream. This, in turn, will make the future optimization and further development for broader applications easier. Furthermore, the price per chip for SoC technology also shows decreasing tendency, which makes an SoC-based solution a very good choice in the future.

6. Conclusion

In this paper, we describe our work on developing an embedded factor graph that can be used to perform reasoning, which is very useful in many robotic applications. We exemplify the use of our embedded factor graph in a real robotic experiment, where the factor graph is used for a sensor fusion to reinforce the robot navigation . We incorporate population coding that mimics the brain-style information processing for encoding values of factor graph's parameters. Such an encoding strategy offers two benefits: compact representation and reliability under uncertainty. To achieve a high-performance result, we implemented the framework on a dedicated hardware. We propose to use the resourceful FPGA in an SoC as an accelerator for a factor graph-based program running on the ARM processor of the SoC. As an accelerator, the FPGA is responsible for transforming the sequential nature of the sum-product computation in a belief propagation algorithm into a parallel processing. The result showed that the accelerator can speed up the computation, eight times faster than the normal run of the factor graph. We incorporated optimization strategies from the perspective of the hardware designer to achieve high efficiency and flexibility. To achieve high scalability and interoperability, we encapsulated our factor graph modules and provide them as embedded Linux modules. With this setup, the future users can conveniently use our factor graph engine to perform many experiments and explorations in order to find the best model for their specific application. From the experimental result and our analysis, we are confident that we have already built an important and fundamental framework for a more powerful inference system. For our future work, we envision to extend our embedded factor graph into a massively-distributed computing engine.

Acknowledgement

This work was supported partially by DAAD (Deutscher Akademischer Austauschdienst e.V.) under the grant A/10/76323.

References

1. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2006).
2. F. Kschischang, B. Frey and H.-A. Loeliger, Factor graphs and the sum-product algorithm, *IEEE Transactions On Information Theory* **47**(2) (2001) 498–519.
3. H.-A. Loeliger, An introduction to factor graphs, *Signal Processing Magazine, IEEE* **21**(Jan 2004) 28–41.
4. J. Yedidia, W. Freeman and Y. Weiss, Constructing free-energy approximations and generalized belief propagation algorithms, *IEEE Transactions on Information Theory* **51**(7) (2005) 2282–2312.
5. B. J. Frey and N. Jojic, A comparison of algorithms for inference and learning in probabilistic graphical models, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**(9) (2005) 1–25.
6. H. Guo and W. Hsu, A survey of algorithms for real-time bayesian network inference, in *AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems 2002*, (Edmonton, Alberta, Canada, 2002).

7. V. Namasivayam, A. Pathak and V. Prasanna, Scalable parallel implementation of bayesian network to junction tree conversion for exact inference, in *The 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06)*, (Ouro Preto, Minas Gerais, Brasil, 2006).
8. V. Sudhakar and C. Murthy, Efficient mapping of backpropagation algorithm onto a network of workstations, *IEEE Transactions on Systems, Man, and Cybernetics, Part B* **28** (1998) 841–849.
9. S. Aluru and N. Jammula, A review of hardware acceleration for computational genomics, *IEEE Design Test* **31**(February 2014) 19–30.
10. A. Papadopoulosa, I. Kirmitzogloub, V. Promponasb and T. Theocharides, Fpga-based hardware acceleration for local complexity analysis of massive genomic data, *INTEGRATION, the VLSI Journal* **46**(June 2013) 230–239.
11. M. Silberstein, A. Schuster, D. Geiger, A. Patney and J. Owens, Efficient computation of sum-products on gpus through software-managed cache, in *Proceedings of the 22nd annual international conference on Supercomputing (ICS'08)*, (ACM, New York, NY, USA, 2008), pp. 309–318.
12. N. Piatkowski, Parallel algorithms for gpu accelerated probabilistic inference, in *NIPS 2011: workshop on parallel and large-scale machine learning*, (Sierra Nevada, Spain, 2011).
13. R. Nasre, M. Burtscher and K. Pingali, Morph algorithms on gpus, in *the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13)*, (Shenzhen, China, 2013), pp. 147–156.
14. M. Pandey, J. Ubhi and K. Raju, *Journal of Circuits, Systems and Computers* **25**(4) (2016).
15. S. Wu, S. Amari and H. Nakahara, Population coding and decoding in a neural field: a computational study, *Neural Computation* **14**(5) (2002) 999–1026.
16. W. Gerstner and W. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity* (Cambridge University Press, 2002).
17. I. Sugiarto, P. Maier and J. Conradt, Reasoning with discrete factor graph, in *IEEE International Conference on Robotics, Biomimetics, and Intelligent Computational Systems (ROBIONETICS), 2013*, (IEEE, November 2013), pp. 170–175.
18. S. Russell and P. Norvid, *Artificial Intelligence: A Modern Approach, 3rd Ed.* (New Jersey: Prentice Hall, 2010).
19. K. Murphy, The bayes net toolbox for MATLAB, *Computing Science and Statistics* **33** (2001) p. 2001.
20. J. Mooij, libDAI: A free and open source C++ library for discrete approximate inference in graphical models, *Journal of Machine Learning Research* **11**(August 2010) 2169–2173.
21. F. Dellaert and M. Kaess, Square root sam: Simultaneous location and mapping via square root information smoothing, *International lJournal of Robotics Research (IJRR)* **25**(12) (2006) 1181–1213, Special issue on RSS 2006.
22. M. Kaess, A. Ranganathan and F. Dellaert, isam: Incremental smoothing and mapping, *IEEE Transactions on Robotics (TRO)* **24**(Septmber 2008) 1365–1378.
23. V. Manshinghka, *Natively Probabilistic Computation*, PhD thesis, Department of Brain & Cognitive Sciences, Massachusetts Institute of TechnologyJune 2009.
24. F. Palmieri, Learning non-linear functions with factor graphs, *IEEE Transactions on Signal Processing* **61**(17) (2013) 4360–4371.
25. Y. Zhao, J. Xu and Y. Gao, A parallel algorithm for bayesian network parameter learning based on factor graph, in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI2013)*, (Washington DC, USA, 2013), pp. 506–512.

26. J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters, *Communication of the ACM* **51**(January 2008) 107–113.
27. A. Steimer, *Neurally Inspired Models of Belief-Propagation in Arbitrary Graphical Models*, PhD thesis, ETH Zürich, Switzerland2012.
28. S. Lange, N. Sunderhauf and P. Protzel, Incremental smoothing vs. filtering for sensor fusion on an indoor uav, in *IEEE International Conference on Robotics and Automation (ICRA) 2013*, (Karlsruhe, Germany, 2013), pp. 1773–1778.
29. H. Chiu, X. Zhou, L. Carlone, F. Dellaert, S. Samarasekera and R. Kumar, Constrained optimal selection for multi-sensor robot navigation using plug-and-play factor graphs, in *IEEE International Conference on Robotics and Automation (ICRA) 2014)*, (Hongkong, China, 2014).
30. V. Indelman, S. Wiliams, M. Kaess and F. Dellaert, Information fusion in navigation systems via factor graph based incremental smoothing, *Robotics and Autonomous Systems* **61**(August 2013) 721–738.
31. T. Czajkowski, *Physical Synthesis Toolkit for Area and Power Optimization on FPGAs*, PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Canada2008.
32. B. Mesman, Q. Zhao, N. Busa and K. Leijten-Nowak, Reconfigurable instruction-set application-tuning for dsp, *Journal of Circuits, Systems and Computers* **12**(3) (2003) 333–351.
33. I. Sugiarto and J. Conradt, Discrete belief propagation network using population coding and factor graph for kinematic control of a mobile robot, in *IEEE International Conference on Computational Intelligence and Cybernetics (CYBERNETICSCOM) 2013*, (Yogyakarta, Indonesia, 2013), pp. 136–140.
34. S. Aqueel and K. Khare, Design and fpga implementation of ddr3 sdram controller for high performance, *International Journal of Computer Science & Information Technology (IJCSIT)* **3**(4) (2011) 101–110.
35. C. Axenie and J. Conradt, Cortically inspired sensor fusion network for mobile robot egomotion estimation, *Robotics and Autonomous Systems* **71** (2015) 69–82, Emerging Spatial Competences: From Machine Perception to Sensorimotor Intelligence.
36. J. Eugenio and M. Estrada, Hardware/software FPGA architecture for robotics applications, in *the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications, ARC'09*, (Springer-Verlag, Berlin, Heidelberg, 2009), pp. 27–38.