Cek Plagiarism Modular SoC Paper

by Indar Sugiarto

Submission date: 14-Sep-2025 11:50PM (UTC+0700)
Submission ID: 2750414735
File name: Modular_design_SoC.pdf (3.53M)
Word count: 10508
Character count: 54824

Microprocessors and Microsystems 60 (2018) 53–64

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Modular design of a factor-graph-based inference engine on a System-On-Chip (SoC)

Indar Sugiarto[a,a,b], Jörg Conradt[c]

[a] Department of Electrical Engineering, Petra Christian University, Indonesia
[b] School of Computer Science, The University of Manchester, United Kingdom
[c] Neuroscientific System Theory, Technische Universität München, Germany

ARTICLE INFO

Keywords:
Discrete factor graph
Population coding
System-on-Chip
Re-configurable computing

ABSTRACT

Factor graphs are probabilistic graphical frameworks for modeling complex and dynamic systems. They can be used in a broad range of application domains, from machine learning and robotics, to signal processing and digital communications. One important aspect that makes a factor graph very useful and very promising to be applied widely is its inference mechanism that is suitable for performing a complex model-based reasoning. However, its features have not fully explored and factor graphs are still used mainly as modeling tools that run on standard computers. Whereas in real applications such as robotics, one needs a practical implementation of such a framework. In this paper, we describe the development of a factor-graph-based inference engine that runs on a System-on-Chip (SoC). Running natively on a low level hardware, our factor graph engine delivers highest performance for real-time applications. We designed the embedded architecture so that it conveys important aspects such as modularity, scalability, flexibility and platform-friendly framework. The proposed architecture has customizable levels of parallelism as well as re-configurable modules that are extensible to accommodate large networks. We optimized the design to achieve high efficiency in terms of clock latency and resources consumption. We have tested our design on Xilinx Zynq-7000 SoCs and the implementation result demonstrates that the proposed framework can potentially be extended into a massively distributed probabilistic computing engine.

1. Introduction

Probabilistic graphical models (PGMs) can be viewed as a unification of graph theory with probability theory into a new formalism for multivariate statistical modeling [1]. It provides a convenient way of integrating perception and action as well as learning and planning which are basic requirements for almost all artificial intelligence (AI) manifestations. This formalism can be characterized as a graph of relations between the involved state variables (i.e., the inferred data) and the observations (i.e., the evidence) [2,3]. In control systems and robotics, PGMs are best suited for higher level control algorithms, although it offers a convenient method for multi-level integration, from low-level control to high-level rule-based planning in the domain of relational statistics [4–6].

The graph in PGMs may be directed (e.g., a Bayesian or Belief network), or undirected (i.e., generally known as a Markov random field). Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to express soft constraints between random variables [7]. In robotics, for example, it is common to use a computer vision technique as a means of gaining information about the state of the world, and to use some form of Kalman filters to infer its own internal states to trigger the robot motion. In this scenario, the undirected graph is the right model for handling the vision processing, whereas the directed graph is the right one for modeling the robot motion through some paths or trajectories.

There is an increasing trend in this decade to merge/combine both directed and undirected graphs into one unified formality. One of the emerging models resulting from such a unification is called the factor graph model. In general, a factor graph represents function's factorizations of several random variables [8]. In its original form, it is an undirected graphical model, but in its inference, it behaves in the same way as a directed model. Because of this inheritance aspect, for a factor graph that has an underlying functionality originating from a Bayesian network, it is usually possible to use an inference mechanism such as belief propagation (BP).

Factor graphs have attracted the attention of many researchers and engineers in recent years because of the wide variety of algorithms that

* Corresponding author.
E-mail addresses: indi@petra.ac.id (I. Sugiarto), conradt@tum.de (J. Conradt).

https://doi.org/10.1016/j.micpro.2018.04.002
Received 8 March 2016; Received in revised form 6 April 2018; Accepted 11 April 2018
Available online 12 April 2018
0141-9331/

have been developed in AI, signal processing, and digital communications, can be expressed using factor graphs. These developments can be derived as specific instances of the BP algorithm running on factor graphs, including the forward/backward algorithm, the Viterbi algorithm, the iterative "turbo" decoding algorithm, Pearl's belief propagation algorithm for Bayesian networks, the Kalman filter, and a certain fast Fourier transform (FFT) algorithm [8–10].

Unfortunately, to our knowledge, all existing factor-graph-capable frameworks are developed to run on standard personal computers (PCs) [11,12]. This, in turn, will conceal the generality of factor graphs. To extend the applicability of factor graphs and to provide practical tools for real technical system applications such as in robotics, we propose an implementation of belief propagation engine for factor graphs on dedicated hardware. Such hardware will provide a convenient way to harness parallelism to achieve high performance computation in low power mode, which is a very important factor that determines the successful operation of many robotic platforms.

In this paper, we focused on the exploration and development of a system that works best with exact inference in a discrete form of factor graphs. We are also aware of the trade-off between fidelity and granularity, usually expressed as a cost function, that rises naturally almost in every digital machine involving discrete-event systems and that there is no single optimal solution that can be applied for every situation [13,14].

For the demonstration purpose, we use the Zynq-7020 from Xilinx as our system-on-chip (SoC) platform, which is internally composed of two tightly coupled sub-systems: the PS (processing system, i.e., microprocessor core) and the PL (programmable logic, i.e., FPGA fabric). The PS sub-system consists of two ARM Cortex-9 processors and the PL sub-system is equivalent to the FPGA Artix-7 from Xilinx. This SoC offers flexibilities because of the fact that their FPGA resources can be structured and organized to mimic the true parallelism in complex computations. This attribute is very useful for performing intense multiplication operations in a factor graph. We can effectively distribute the computation into concurrent calculations in every slice of the FPGA in the SoC device. We refer to our factor graph framework on an SoC as an "embedded factor graph". To our knowledge, it is still uncommon to see a factor graph in any embedded systems.

The novelty of this work is the exploration of the implementation of probabilistic inference using message-passing-based methods for factor graphs natively in low-level hardware. Such fundamental probabilistic inference hardware, which takes into account the uncertainty and randomness into its computational platform, will produce more powerful, flexible and efficient building blocks for a more complex computational intelligence machine. In summary, the major contributions presented in this paper are as follows:

1. Implementation of belief propagation for SoC-based factor graphs that integrates both software-based control and real-time hardware-based processing.
2. Introduction to a reconfigurable computing framework for general PGMs. The factor graph's core modules in the FPGA part are implemented as IPs that can be re-configured to accommodate various requirements for research on PGMs.
3. Our design readily accommodates the requirements to support technical systems with PGM-based cognitive capabilities.

This paper is organized as follows. After providing selected reviews on existing factor graph implementations in Section 2, we describe our development paradigm in Section 3. The paradigm provides guidance on the development of our proposed embedded framework, which is explained in detail in Section 4. In Section 5, we provide an analysis of the implementation results. A thorough discussion about the overall evaluation of our proposed methods is presented in Section 6. Finally, we conclude our work and propose recommendations for future work.

2. Review of factor graphs and related work

2.1. On discrete factor graphs

Factor graphs using belief propagation (BP) are a fascinating topic of research and some studies have already been conducted to explore the many potential applications of such methods in specific fields, such as in communications and signal processing [15], and control systems [16]. Recently, this probabilistic graphical approach has attracted more attentions from researchers who work in the field of machine learning (ML) and cognitive intelligence [4,5,17,18].

BP is a common method for performing inference in PGMs. For factor graphs, the algorithm called sum-product [8] resembles a similar message-passing algorithm found in the original Bayesian network proposed by Judea Pearl [19]. There are two types of messages that are transmitted among nodes in a factor graph network: the message sent by a variable node to a factor node (denoted as $\mu_{x\to f}(x)$ and the message sent by a factor node to a variable node (denoted as $\mu_{f\to x}(x)$). These messages are computed based on the following equation:

$$\mu_{x\to f}(x) = \prod_{h\in n(x)\setminus\{f\}} \mu_{h\to x}(x) \tag{1}$$

$$\mu_{f\to x}(x) = \sum_{\sim\{x\}} \left(f(X) \prod_{y\in n(f)\setminus\{x\}} \mu_{y\to f}(y) \right) \tag{2}$$

where $X = n(f)$ is the set of arguments of the function f and $\sim\{x\}$ is the "not-sum" or *summary* indicating the variables that being summed over. The flow of such messages as a propagation of nodes' beliefs is illustrated in Fig. 1.

To integrate the factor graph into the hardware level, we need to determine how to discretize probabilistic values operated by the factor graph. In the BP setting, such probabilistic values are treated as messages propagating through the factor graph network. In our work, we discretize such messages using a population coding principle. In this paradigm, a group of neurons are activated in a way such that they produce neural responses with certain probabilistic distribution when given stimuli [20–24]. The number of neurons reflects the number of states (or cardinality) of discretized random variables involved in the network.

A neuron may receive spikes from its interconnected neurons in the previous layer because of incoming stimulus:

$$x(t) = \frac{\int_t^{t+\Delta t} \sum_j \sum_f \delta(t - t_j^f)\mathrm{d}t}{\Delta t \cdot N} \tag{3}$$

Such a neuron may be characterized by certain tuning curves as its activation function. One commonly used tuning curve in homogeneous population codes is the Gaussian function:

$$x_i = \frac{1}{\sigma\sqrt{2\pi}} e^{-(s-\mu)^2/2\sigma^2} \tag{4}$$

where x_i is the ith 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. 2 shows an example of the

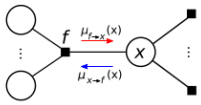


Fig. 1. Illustration of BP in a factor graph. The red arrow indicates a factor-to-variable message expressed in (2), whereas the blue arrow indicates a variable-to-factor message expressed in (1). Both messages take the same variable-x as their argument. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

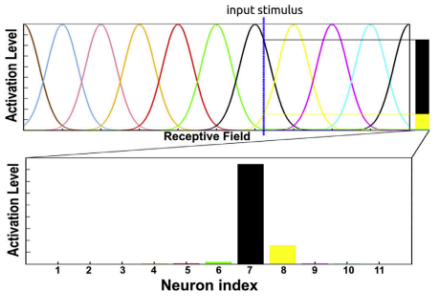


Fig. 2. The Gaussian tuning curves to represent neuronal activation levels in a homogeneous population comprising of 11 neurons. When an input stimulus arrives at the receptive field (shown as a blue-dashed line), each neuron fires. The measured activation levels from all neurons (along the blue-dashed line) are then combined to produce the overall probability distribution which is depicted as a discrete probability mass function. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

homogeneous tuning curves and illustrates how a real value input is encoded in a population code.

More detailed information about how we implement population codes for factor graphs can be found in [25]. In our experience, the precision of the factor graph inference result mainly depends on the number of neuron used to encode the messages. This demonstrates that our population code implementation result agrees with the common consensus in computational neuroscience; that is, the population size can scale exponentially with the number of neurons to accurately estimate the underlying likelihood function [26].

2.2. Related works

Many PGM tools that are available as software packages were originally designed to suit one of the forms of PGMs, either directed or undirected models [11]. Recently, some of those software packages also include factor graphs in their libraries and use a factor graph as an inference engine. For example, a software package called GTSAM mainly uses the elimination algorithm to make an inference [27,28]. A similar mechanism is also provided by the BNT software in it's factor graph library [11]. Another software package called libDAI implements various inference methods including a message-passing algorithm [29]. Unfortunately, those software packages and many other libraries implement factor graphs only for PCs and serve as a simulation platform for studying certain aspects of AI or ML. Hence, they cannot be used directly in real technical applications such as robotics. Furthermore, none of them implement population codes for encoding message values to run through a factor graph network. Our factor graph framework, in contrast, is readily targeting real-time applications that extend a technical system into a dynamic system with cognitive capabilities [30].

With regard to the discretization strategy, similar research has also been conducted by Mansinghka, who created a stochastic digital circuit using FPGA to build fault-tolerant machines for sampling, which enables Markov chain Monte Carlo (MCMC) to run effectively [31]. The main difference between Mansinghka's work and ours is that we implement the discretization for continuous variables using a population coding principle developed by the computational neuroscience community. On the other hand, Manshinghka concentrates on the sampling algorithm from statistics and provides little details on the higher level of intelligence abstraction. 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 [32].

One underlying source of motivation for our work is to achieve high performance computing through parallelism. Many researchers have already proposed methods to improve the performance of graphical model computations by harnessing parallelism in modern computers [12,33–36]. Additionally, although not so surprisingly, the trend of exploiting graphics processing units (GPUs) has attracted many researchers to start deploying their PGMs on 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 [37]. Factor graphs have also already been implemented in a GPU [38,39]. However, to our knowledge, no investigations have yet been conducted on factor graphs using any dedicated hardware.

We are also aware that there is an effort to exploit Bayesian networks using factor graphs in a distributed computing framework [40]. They use libDAI [29] to implement discrete factor graphs and speed up the computation by using a parallelism framework called MapReduce [41]. Unfortunately, we could not test their method on a discrete factor graph using population code because libDAI does not support population code techniques.

There is also an attempt to implement BP that is inspired by biophysical properties of neuronal networks. Work by Andreas Steimer (in [42]) focuses 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.

3. Embedded framework development

Working with SoCs entails two perspectives. On one hand, SoCs offer extensive system level integration and flexibility; on the other hand, these devices impose a new challenge to the integration of both concurrent and sequential programming paradigms. Because of these different paradigms, we cannot directly implement a PC-based factor graph program into an SoC. We need to implement the distributed computing mechanism in a very low hardware level, down to the signal and bus topology that must be designed and implemented within the chip itself.

Also, developing an SoC-based embedded system always involves at least two different programming platforms: the software platform for the central processing unit and the software platform for the programmable logic (FPGA) unit. In this work, all of the main cores of our embedded factor graph were developed using Vivado-HLS (High Level Synthesis), which transforms a C, C++ or SystemC design specification into a Register Transfer Level (RTL) implementation which in turn can be synthesized into the FPGA. Only the glue-logic components and small but frequently used non-behavior-based logic blocks are written purely in VHDL to decrease total latency and achieve high area optimization. Vivado HLS can also generate the drivers for an embedded Linux kernel running on the PS part of the SoC. The diagram in Fig. 3 shows the work-flow of our SoC-based factor graph framework.

3.1. Design considerations

Deploying programs in dedicated hardware requires different treatments and explicit considerations. Our framework was developed with the following criteria.

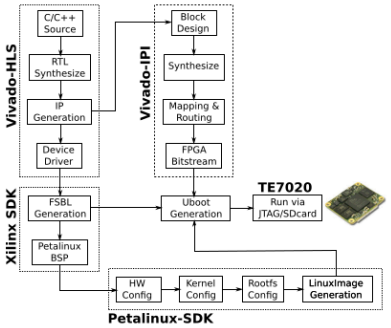


Fig. 3. The overall design flow for creating the framework of SoC-based factor graph engine. In this diagram, the abbreviations mean: SDK = Software Development Kit, BSP = Board Support Package, FSBL = First Stage Boot Loader, RTL = Register Transfer Level, IPI = Intellectual Property Integrator.

Scalability. The framework should be able to work with a variable number of chips, allowing us to conveniently resize the networks later.

Cross-boundary. The framework should be able to seamlessly connect the separated elements of a factor graph, i.e., inter-chips and intra-chips.

Modularity. Having an environment that offers a high degree of flexibility, the modules in the SoC-based embedded factor graph should be flexible enough to be reconfigured or remapped into a new, possibly denser, SoC chip.

Flexibility. The framework should be flexible enough to be reconfigured for many general purpose applications without too many modifications to the framework.

In addition to the aforementioned hardware-oriented paradigms, our overall framework also includes application programs running on a host-PC. For example, we developed a robot controller program based on factor graphs, which utilizes a graphical user interface to facilitate an interaction between the human operator and the robot, as described in [30]. Hence, we added the following criteria for our design considerations.

Continuity. The framework should be able to be used in a simple transitional step from its original PC-based framework. This means that when we have finished simulating a factor graph network on a PC (using our PC-based framework), the network should be able to be translated into the SoC-based embedded version seamlessly. Minor modifications may be required, but it should not be a burden to the application developers. To fulfil this aspect, we put the BP scheduler to run on the ARM-core instead of the FPGA part; hence, making the porting from PC to SoC easier for user.

Platform-friendly. The modules should be flexible enough to be re-synthesized in the new version of the development environment, both for the FPGA's bitstream generation and the Linux kernel reconfiguration. This co-development paradigm requires a careful design of both hardware and software for the resulting embedded system to work efficiently.

3.2. Optimization strategy

The optimization strategies that we apply in our design are closely related to two issues: the numerical accuracy and the speed-area trade-off.

With regard to numerical accuracy, we decided to use single precision floating-point rather than fixed-point; this is because fixed-point arithmetic introduces difficulties for probabilistic computations. This is often the case when a factor graph is used in an application that mixes a wide range of real-value numbers with probability values. In such cases, it is very difficult to compromise between large real-value numbers (greater than 100.0) and very low probability values (less than 0.000001) using fixed-point arithmetic. We argue that floating-point is the best choice for the implementation of our factor graph in the FPGA, because it can represent real numbers with a much wider dynamic range, which allows the data to be used through long sequences of calculations (e.g., in a BP computation).

Fortunately, Vivado-HLS provides a convenient way to use floating-point arithmetic in a C-style code. However, even with the abundance of logic resources in Zynq-7000, we still need to be cautious when implementing floating-point operations. This is because floating-point numbers inherently contain two main artifacts: accumulation of rounding errors and improper handling of subnormals [43]. In our implementation, we carefully inspected the synthesis report produced by the FPGA synthesizer and looked for mismatches and artifacts.

Regarding the trade-off between area and speed optimization, we are aware that there is no rule or theory on how the optimization can be best achieved with respect to the defined area and desired speed. This is because not all SoC/FPGA chips share the same internal supporting components (e.g., distributed RAM and DSP blocks). Hence, the re-implementation from one chip to another (from different families) will always produce different results. In this work, however, we placed an emphasize on speed over area optimization. The argument is that the optimization of speed is a more generic issue than area optimization for at least two reasons:

1. In general, the basic motivation for using dedicated hardware is to speed up the computation process. We often assume that the chip vendor will create/produce chips with all of the necessary elemental units
2. Many of the optimization options provided by the synthesizer vendor are more related to the speed issue, which the system designer can play with.

It turns out that many parts in our algorithm require accesses to memory units in the form of an array. Our optimization strategy for arrays includes reshaping and partitioning [44,45]. In addition to this strategy, we also consider the utilization ratio of look-up tables (LUTs)) and block RAMs (BRAMs). Although it is possible to use external memory, we prefer to avoid this method, because external memory access is an expensive task in terms of allocating FPGA resources. Controlling external memory requires explicit routing strategies to match the interfacing protocols and timing constraints required by the memory hardware [46–48]. In our design, the use of BRAMs is maximized as long as latency does not pose any issues, because the locations of BRAM units within the chip are usually sparse.

With regard to speed optimization, our approach is mainly based on the idea of exploiting the "unbounded" parallelism paradigm [49,50]. Our design was focused more on the function-parallelism rather than on data-parallelism. We achieved this by combining both unrolling and pipeline mechanisms, which means that parallelism happens in a loop (statically-scheduled instruction-level parallelism) and between loops (dynamic task-level parallelism). Fig. 4 shows an example of how the unrolling and pipelining are employed by using Vivado HLS directives. More about the use of unrolling and pipelining in the module of our embedded factor graph are described in the next section.

```
#define NVAR 3          // factor's scope
#define CARD 2          // variables' cardinality
#define N 2
#define NPARAM (NVAR*CARD)
int CARDLIST[NVAR] = {CARD};
float msg_in[CARD];
float msg_out[CARD];
float myFactor[NPARAM];

extern int varIdx;

void factorProduct()
{
  int i, j;
  for (j=0; j<CARD; j++){
#pragma HLS unroll factor=N
    for (i=0; i<NPARAM; i++) {
#pragma HLS PIPELINE
      if(inScope(j,i,varIdx))
        msg_out[j] = msg_in[j] * myFactor[i];
    }
}

int inScope(int j, int i, int idx)
{
  int i, s[NVAR] = {1};
  for (i=1; i<NVAR; i++)
    s[i] = CARD * s[i-1];
  return (i/s[j]) % CARDLIST[idx])
}
```

Fig. 4. A snippet of a code that shows how unrolling and pipelining are used in our embedded factor graph.

These unrolling and pipelining require more resources in exchange for increases in speed. We use 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 out by the synthesizer).

4. Hardware architecture design

In this work, we aimed to optimize the embedded factor graph by exploiting all available resources in the FPGA. The factor graph inference engine was implemented completely in the FPGA part of the SoC, while the scheduling for the belief propagation (BP) is kept running in the ARM core in the SoC. The reason behind this strategy is that, we want to explore the challenge of cyclic factor graph in the future. It is known that cyclic factor graphs are very challenging since there is not guarantee that they will converge. Hence, we make our SoC-based factor graph engine widely open for researches in fundamental probabilistic graphical model.

4.1. Tested hardware

As a test case, our work was implemented on a Zynq-7000 from Xilinx Inc. The SoC has two parts: a processing system (PS), which contains one or more microprocessors, and a programmable logic (PL) component, which is normally an FPGA. We used a tailored module TE0720-2IF produced by Trenz-Electronics GmbH. It contains an SoC XC7Z020 with the following features: dual core ARM Cortex-A9 microprocessor, and an Artix-7 FPGA with 85K logic cells and 560 KB BlockRAM respectively. It also provides a large number of configurable I/Os in its form factor of $5 \times 4\,\text{cm}^2$, which is suitable for embedded system applications such as mobile robotics.

Fig. 5 shows the general overview of our design partitioning. As we explain in Section 5 that the SoC XC7Z020 has limited resources to implement the full network with high resolution, we also simulate the network using SoC XC7Z030 to gain some insight on the performance of our design. We used SoC XC7Z030 as an alternative simulation because this SoC is still supported by the synthesizer Vivado-Webpack from Xilinx. Table 1 summarizes the internal FPGA resources of both SoCs XC7Z020 and XC7Z030. The XC7Z020 is equipped with Artix FPGA, whereas the XC7Z030 is equipped with Kintex FPGA.

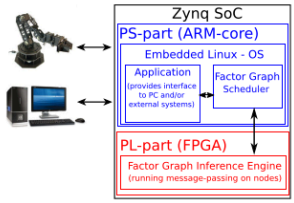


Fig. 5. General overview and mapping of the modular design of our SoC-based factor graph engine. The FPGA part of the SoC works as the main component for the factor graph inference engine, whereas the microprocessor part of the SoC works as a supervisory controller and provides interface to external system.

Table 1
Summary of important resources of the FPGAs inside the SoC XC7Z020 and XC7Z030.

	XC7Z020	XC7Z030
Logic Cells	85K	125K
Look-Up Tables (LUTs)	53,200	78,600
Flip-Flop (FF)	106,400	157,200
BRAM	4,9Mb	9,3Mb
DSP48E (Slices)	220	400

4.2. Modular design of factor graph engine

To achieve real-time performance, we developed a fully embedded factor graph framework. In this mode, we maximized the exploitation of the FPGA's abundance of logic fabrics and its routing channels. The routing management is the key to successfully implementing an efficient embedded factor graph. Inside the FPGA chip, there are numerous signal lines that interconnect the logic blocks within the chip. These lines can be arranged and managed to create buses, which can later be customized for our embedded factor graph. In our work, we used the AXI protocol to encapsulate the buses that communicate the factor graph messages between the PS and PL components.

We identified the following core modules that are necessary to build a complete discrete factor graph based on population coding: factor and variable nodes, a message encoder/decoder, and a scheduler. In this work, we regarded the factor/variable node and the message encoder/decoder as the most generic parts of the system, which could be used in almost any scenario, whereas the scheduler is a flexible module, which can be tailored to support the application scenario. For some applications where the network has loops, the scheduler will play an important role. Conceptually, we can implement many scheduling strategies in the scheduler (synchronous, asynchronous, residual, etc.). However, we have only tested the asynchronous option and defined it as the default scheduling strategy for tree-like networks.

Factor and variable node controller

Our implementation of factor and variable node modules follows the same mechanism described in [25]. We optimized the module with the following default setting: unrolled into N instances (where N is the variable's cardinality) and pipelined with one initiation interval (i.e., fully pipelined without resource sharing for operators). The unrolling

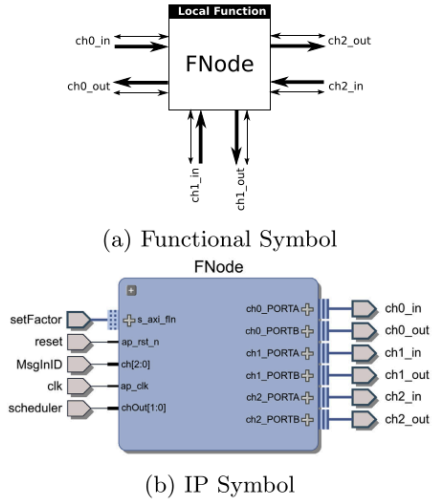


Fig. 6. FNode symbol representation.

optimization is done by creating multiple independent operations of for-loops rather than a single collection of operations, whereas pipelining is used to increase throughput by performing concurrent operations.

The basic algorithm for the factor and the variable node controllers is the same because both employ the same message-passing algorithm in Eqs. (1) and (2). However, they differ only in the internal factor product operation, which only happens with the factor node. We first created the factor node controller and then decreased its operation (i.e., removed the internal factor component) to create the variable node controller.

Fig. 6a shows the functional symbol of a factor node controller (marked with the name FNode). It represents a three-channel bidirectional module that can connect up to three variable nodes. Fig. 6b shows the resulting IP-block representation already included in the Xilinx Vivado IP repository after its successful synthesis. The internal factor's function can be supplemented in the controller using a simple call function, which is already encapsulated in the common AXI interface together with the main function of the module. The internal structure of this FNode module is shown in Fig. 7.

Within the FNode module, there are four sub-modules: setFactor, UpdateFMsg, NodeMsgIO, and LocalFactor. Apart from these four sub-modules, the FNode module requires several instances of BRAMs (which could also be replaced with distributed memory using LUTs) to facilitate data array processing. The sub-modules setFactor and LocalFactor work together and are responsible for managing the internal factor's function that corresponds with the factor node. The value of this internal factor's function can be set by using a standard call function, which is already available in the device driver to make it convenient when using the embedded Linux running in the PS component of the SoC. The sub-module NodeMsgIO is responsible for receiving and delivering messages from/to external variable nodes. It contains three bidirectional channels and has a small table which tracks the messages' exchange during the transaction. Whenever a new message arrives, this table will be updated and the sub-module UpdateFMsg

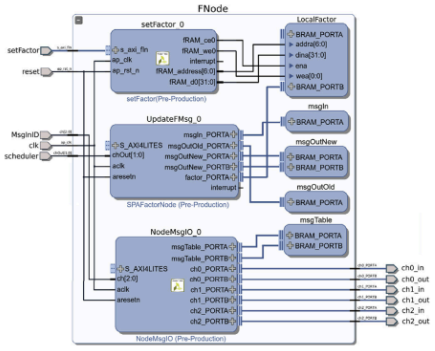


Fig. 7. Internal block diagram of module FNode shown in Fig. 6b.

will be notified when incoming messages are delivered and ready to be used to compute the output message. The sub-module UpdateFMsg is the core module where we implemented the sum-product algorithm. This sub-module also has an internal memory to save old messages, which can be used during the training with an MLE or EM algorithm in a message-passing scenario for example. This sub-module also has scheduler input signals that can be used to control the scheduling of the message-passing exclusively on this FNode instance. By default, we implemented asynchronous scheduling, which is the standard scheduling strategy for an acyclic factor graph.

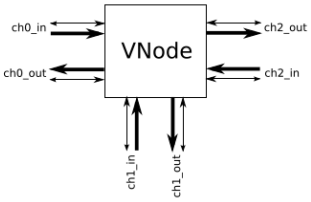
From the FNode module, we derived the VNode module which is responsible for handling variable nodes in factor graphs. In principle, the VNode module works in a similar way as the FNode module, but without the functionality to set up an internal function. The functional symbol and the IP symbol of the VNode module are shown in Fig. 8, whereas its internal structure is shown in Fig. 9.

There is one more input signal that is present in both FNode and VNode modules and is responsible for assigning an ID to each message. This input signal is called MsgIn ID and is preserved for future improvement of our embedded factor graph. Using this signal, the node can identify if this message is the same message circulating in the network or if it is a new message. This mechanism is useful to resolve cycling errors in a cyclic factor graph. However, we did not fully explore this mechanism yet and simply set its counter to 0; this means that all incoming messages are always assumed to be new. Consequently, if our embedded factor graph is going to be used to implement a cyclic network, then the software running in the PS must check manually to determine if there is a cycling error in the network.

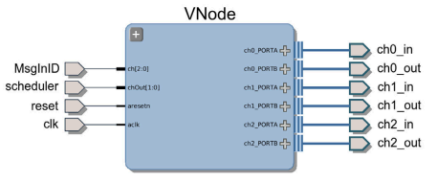
Message encoder and decoder

We implemented a special module, called NodeIO, that is responsible for encoding and decoding messages. This module implements our population coding algorithm. With regard to the structural complexity of the design, this module is simpler than the FNode or VNode modules and contains only a single bidirectional channel. The functional and the IP symbols of this NodeIO module are shown in Fig. 10.

This module should only be connected to a VNode module because it represents the input factor of the corresponding variable node. Once this module is connected to a VNode, we can send a value to the respective variable node via the kernel's function `getStates()`. This module then encodes the value into a probabilistic vector and sends it to the variable node via its output channel. Afterward, that vector will be propagated in the network as a message. At some point in time, the



(a) Functional Symbol



(b) IP Symbol

Fig. 8. VNode symbol representation.

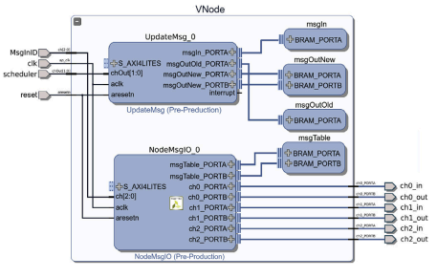


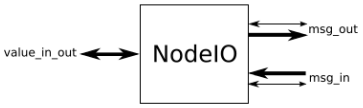
Fig. 9. Internal block diagram of module VNode shown in Fig. 8b.

NodeIO module might receive a message from its variable node. When a new message arrives, this module starts decoding the message and when it is completed, it triggers an interrupt to inform the hardware driver that a new value is ready to be picked up. The program in the PS then able to retrieve this value by calling the `getRealVal()` function of the kernel.

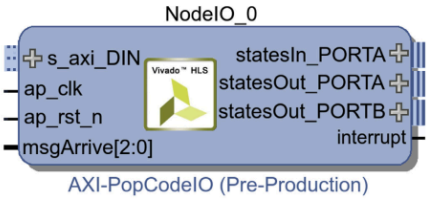
The sending of a value to this NodeIO module will trigger the BP automatically. This message-passing circulation will stop in a definite time if the network is an acyclic one. Reading a message arriving at this NodeIO corresponds to the reading of the marginal value of the corresponding variable node. There is no guarantee, however, that the message circulation will converge (or not) in a cyclic network. To stop the circulation, the software needs to inform all the nodes in the network by sending a value of -1 to the scheduler of each node.

Example network

Here, we provide an example of how to use our embedded factor graph. An example of a network, shown in Fig. 11(a), has five variable nodes, two factor nodes and four input nodes. This network has been used to control the omnidirectional mobile robot in our previous work



(a) Functional Symbol



(b) IP Symbol

Fig. 10. Node-IO symbol representation.

by kinematic mapping from robot's body velocity to its wheels velocity [30]. Here variables A, B, and C represent robot velocity in X and Y directions as well as its rotational velocity respectively, whereas variable D represent a wheel velocity of the robot.

Assuming that we have learned the factor's parameters, we can send those factor's parameters to the factors f_{ABH} and f_{CDH} by calling `setFactor()` through the kernel's driver prior to running the BP on the network. We translated the network in Fig. 11(a) into the symbolic diagram presented in Fig. 11(b).

As seen in Fig. 11, building the factor graph network using our framework is straightforward: given a structural description of a network, we then create its symbolic representation. The next step is to draw such a diagram in the Vivado IPI editor by fetching all of the corresponding IP blocks from the repository and connecting them.

5. Experimental results

We evaluate the modules used in our design in terms of their clock's latency characteristics, as well as their consumption of FPGA resources. It is important to note that the optimization using the pipeline mechanism works only with perfect or semi-perfect loops; hence, we need to specify the cardinality of the variables in the source code before synthesizing it.

We compare the overall estimated performance on two scenarios: fully optimized and unoptimized designs. For the fully optimized design, both unrolling and pipeline were maximized. Fig. 12 shows the latency characteristics of those modules for both scenarios.

In general, it is obvious that the optimized design that involve both unrolling and pipelining will reduce the clock latency; hence, it will speed-up the computation. However, the area coverage (i.e., the resource consumption) will be increased as its consequence.

The chart in Fig. 12 is obtained by using cardinality value of five; hence, the design with optimization is unrolled five times. We observe that we gained the speed-up at about six to seven times instead of five since we also employ one stage pipeline on the design.

For the NodeIO module, the encoder section generally has a higher profile than the decoder section. This is not surprising because in the encoder part, the sampling of each tuning curve takes more time and resources due to the use of an exponential function to compute the Gaussian distribution. Compared with the other modules (FNode and VNode), the clock latency is much lower, revealing the fact that the

Table 2
Summary of FPGA's resources consumption of the main modules in our embedded factor graph with full optimization (unrolling and pipelining).

	NodeIO			FNode				VNode			
	Encoder	Decoder	Σ	MsgIO	Factor Product	Sum	Σ	MsgIO	Factor Product	Sum	Σ
BRAM (%)	5	5	10	2	14	12	28	2	12	10	24
DSP48E (%)	7	3	10	0	2	2	4	0	2	2	4
FF (%)	4	3	7	1	17	10	28	1	14	10	29
LUT (%)	12	3	15	1	9	6	16	1	7	4	12

encoding and decoding of a message will not create a bottleneck for the entire system. For the FNode and VNode modules, their latencies did not differ much because those modules essentially originate from the same source. With an average clock latency of 160529, it takes about 0.24 ms for the node to compute an output message when we run the system with a clock frequency of 667 MHz.

Using unrolling and pipelining on the design will increase the speed operation, but they also increase the resource consumption. We measure the FPGA resource utilization on the following components: Look-up Tables (LUTs), Flip-flops (FFs), Block RAMs (BRAMs), and DSP Slices (DSPs). These four components are the major resources that will be utilized in order to implement any functions on the FPGA. Hence, measuring the utilization on these components will inform us how efficient our design is synthesized. We measure resource utilization on both scenarios: without optimization and with optimization (unrolling plus pipelining).

Comparing the usage of FPGA resources (BRAMs, DSPs, FFs, and LUTs), the FNode and VNode modules still show similar characteristics. For the optimized design with five states per node, the NodeIO module consumes 10% BRAMs, 7% FFs, and 15% LUTs. Table 2 shows the optimized design using five states for the variable's cardinality.

For the network shown in Fig. 11 with four states for variables' cardinality, the NodeIO modules consumes 25% BRAMs, 14% FFs, and 37% LUTs altogether. The FNode modules consumes 36% BRAMs, 36% FFs, and 22% LUTs altogether. The VNode modules consumes 39% BRAMs, 20% FFs, and 36% LUTs altogether. This configuration is the maximum setting that can be achieved using our hardware. Fig. 13a shows the floorplan of the implementation result with this maximum configuration.

If we increase the cardinality to be 5, then we get the following result: the NodeIO modules consumes 40% BRAMs, 21% FFs, and 60% LUTs; the FNode modules consumes 56% BRAMs, 56% FFs, and 32% LUTs; and the VNode modules consumes 80% BRAMs, 20% FFs, and 60% LUTs. This excessive amount of resource consumption could not be synthesized with our current hardware (TE0720 with an SoC XC7Z020). Hence, we simulated the network using a denser SoC chip such as XC7Z030. Fig. 13b shows the floorplan of the implementation with five states for variables' cardinality. It can be seen that there are still plenty of spaces left that can be used for extension. Using denser SoC XC7Z030, we could synthesize modules with cardinality up to 10. However, we could not use high number of unrolling factor. In our design, the maximum unrolling factor is 7 in order to achieve cardinality up to 10.

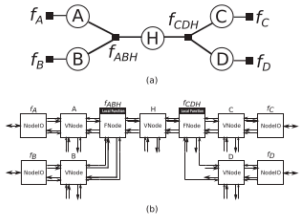


Fig. 11. An example of how to construct a factor graph network using core modules of our factor graph framework. (a) The original network consists of two factor nodes, five variable nodes, and four IOs. (b) The network in (a) is reconstructed using the core elements of our factor graph framework and later can be implemented in the FPGA.

The second approach in our design is conducted by using a simple module design (without unrolling and pipeline optimization). Table 3 summarizes the synthesis result for the design with five states for variable's cardinality.

Comparing Table 3 to Table 2, we can observe that our unoptimized design requires fewer FPGA resources. With this setting, we can implement the network shown in Fig. 11 on our hardware TE0720 without any problem. Fig. 14 shows the floorplan of this design. Comparing to the Fig. 13a for the same network but with a bit higher cardinality (five states rather than four states in this case), the unoptimized design yields area coverage about 75% fewer than the optimized design.

For five states per node, the total amount of resources used by the synthesized network are as follows. NodeIO modules require 24% BRAMs, 8% FFs, and 16% LUTs. The total amount of resources used by the FNode modules were: 22% BRAMs, 12% FFs, and 8% LUTs. The total amount of resources used by the VNode modules were: 40% BRAMs, 25% FFs, and 15%. Hence, to implement the network in Fig. 11 using SoC XC7Z020 with five states for cardinality, it requires a combination of 86% BRAMs, 45% FFs, and 39% LUTs. These can fit into our small density SoC, but with additional delay of about 3 ms for the network to generate the output message.

To test the speed-up gain for having such a full optimization, we ran the belief propagation using the network shown in Fig. 11 on two designs: the optimized version and the unoptimized version. The result is shown in Fig. 15. The performance on the SoC XC7Z020 was measured

Table 3
Summary of FPGA resource consumption without optimization (unrolling nor pipelining) for the design that uses only five states for each variable's cardinality. The report was generated for SoC XC7Z020.

	NodeIO			FNode				VNode			
	Encoder	Decoder	Σ	MsgIO	Factor Product	Sum	Σ	MsgIO	Factor Product	Sum	Σ
BRAM (%)	3	3	6	1	5	5	11	1	5	4	10
DSP48E (%)	2	1	3	0	1	1	2	0	1	1	2
FF (%)	1	1	2	0	4	2	6	0	3	2	5
LUT (%)	3	1	4	1	2	1	4	1	1	1	3

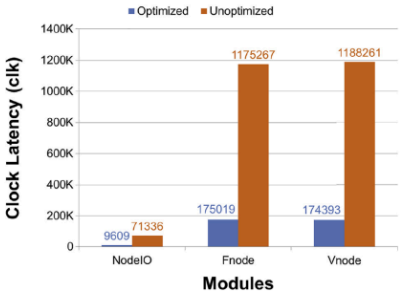


Fig. 12. Clock latency (in number of clocks) of three main modules.

in real-time, whereas the performance on the SoC XC7Z030 was estimated. In order to estimate the performance of the factor graph as if it ran on a real XC7Z030, we compared the latency of each module when they were synthesized for both XC7Z020 and XC7Z030. The latency values were aggregated according to the number of modules instantiated in the network. We then ran a program under Petalinux on TE7020 that utilized the network, and measured the execution time of the program until it produced the output. By using these values, we performed a regression to estimate how long it will take if the same network run on a XC7Z030 device.

Fig. 15a indicates that our TE0720 hardware limits the capability of our design to be synthesized with full optimization options. Up to four states for the cardinality is synthesizable for the network shown in Fig. 11. However, running the network without full optimization is still acceptable in our case because the actual robot that will be controlled by our hardware usually runs at much slower speed. The sensory systems of the robot in [30] are sampled within 100ms interval; hence, in this circumstance, our unoptimized design will work properly even with high number of states (i.e., 10 states).

From Fig. 15b, we can observe that the optimized version can give speed-up gain up to five times for the network with cardinality up to seven states. Afterwards, the speed-up gain remains to be constant at about three times. This happens because we used a fix number for unrolling factor, which is 7. Beyond this number, the network could not be synthesized on SoC XC7Z030. We believe that if we used denser SoC (such as XC7Z045 or XC7Z100), then we can increase the unrolling factor again, which will result in increasing speed-up gain accordingly.

With these results, we are convinced that our proposed factor graph engine has achieved its goal. Furthermore, it opens the possibility to be extended into a massively distributed probabilistic computing engine.

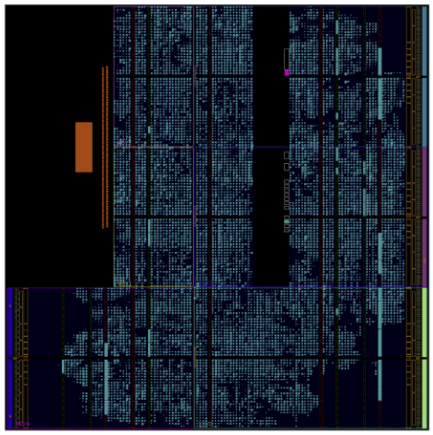
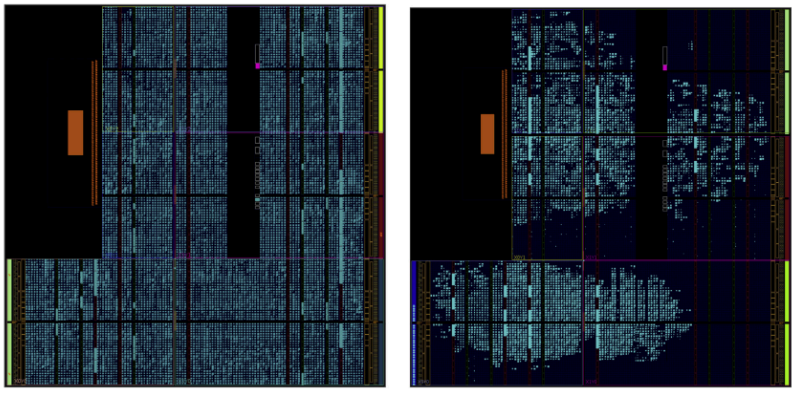
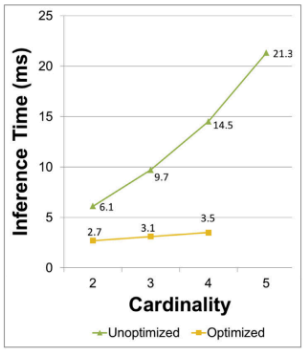


Fig. 14. Floorplan view of the design without optimization for the network shown in Fig. 11. This was implemented on SoC XC7Z020, and we used cardinality value of five for the network.

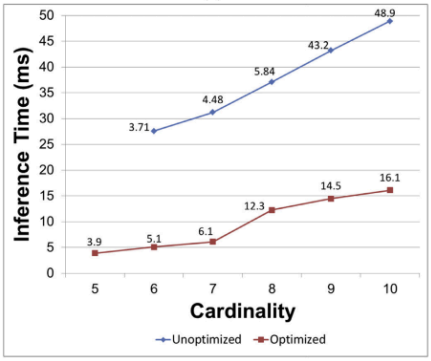


(a) (b)

Fig. 13. (a)Floorplan view for the design using optimization with four states per node implemented on SoC XC7Z020. (b)Floorplan view for the design using optimization with five states per node implemented on SoC XC7Z030. Using higher density SoC allows us to achieve greater resolution.



(a)



(b)

Fig. 15. Runtime performance of the network shown in Fig. 11 using two different design configurations: an optimized and an unoptimized implementation. (a) Real implementation result on XC7Z020. (b) Simulation result using XC7Z030.

6. Discussion

In this paper, we proposed the construction of a full factor graph inference engine as a part of embedded factor graph framework running natively on the FPGA part of an SoC. From the implementation result, we observed that implementing the whole factor graph nodes on the FPGA is a resource-intensive approach. Even with low variable cardinality and without optimization, it almost consumes all of the FPGA's main resources. Furthermore, the trade-off between area and speed optimization is a really challenging choice. Even though this burden can be easily overcome by employing higher density SoC (or FPGA), we prefer to find a practical solution using our existing SoC platform.

In our hardware experiment, we can only specified up to four states as the default variable's cardinal value for the design that include full optimization (both unrolling and pipelining are on). This cardinal value is the highest value we can achieve with TE0720-2IF hardware because of the limited resources of the XC7Z020. For evaluation purpose, we simulated and synthesized our design also for XC7Z030. This was to prove that our design is scalable and that high resolution is easily achievable using higher density SoC. Using XC7Z030, we can have resolution up to 10 states for each variable; this resolution is high enough to achieve high precision control for mobile robot platform used in [30].

In our design, we did not use external memory for storing messages, rather we simply maximize the utilization of the existing BRAMs and LUTs. The main reason for this choice is that accessing the external memory independently requires more resources, which in turn will not leave enough resources for the factor graph modules themselves. Controlling external memory needs explicit routing strategies in order to match the interfacing protocols and timing constrains required by the memory hardware. If we rely on the PS for intermediating between the FPGA and the external memory chips, then we have to implement the DMA protocol. Unfortunately, this DMA mechanism (using AXI-DMA IP in the Vivado library) is not a good choice for our current hardware because it consumes a considerable amount of FPGA resources.

To achieve higher scalability and adaptablity, the following approaches can be applied. (1) The NodeIO module can be modified such that one instance of this module will be used by all nodes in the network. A gating mechanism to determine which node the encoded message will be delivered to could be used to incorporate this approach. Of course, a new delay effect will be introduced to the system, but we can ignore it in a larger network because the total delay will be dominated by delays in the message-passing computation. (2) There are nodes without intense computation in its microarchitecture, especially for second order variable nodes. Therefore, it is beneficial if we do not create a complete working node by deriving it from the FNode module. Rather, we can create a simple message forwarding module. With this approach, we will have different modules for variable nodes; the choice of using a specific module depends on the degree of the corresponding variable. (3) The optimization strategy can be set to a moderate level to maintain a balance between the area and the speed.

In this paper, we present the design only using a single SoC system. However, we have laid a foundation to create more complex systems by providing interconnection and interface between modules that run on different chips. This will be our future work, where we want to have a massive distributed factor graph engine that spans across multiple SoC chips.

7. Conclusions

Bringing a high abstraction level concept such as factor graphs down to a hardware level is a fascinating but challenging task. In this paper, we described our work on developing an embedded factor graph framework that is implementable on an SoC. Initially, we started developing our approach by utilizing the FPGA in the SoC only as an accelerator for a factor graph framework that runs on the ARM processor of the SoC. As the accelerator, the FPGA is responsible for transforming the sequential nature of the sum-product algorithm into a parallel fashion. We continued exploring the framework by extending it into a more complex design that maximized the use of the FPGA resources. Using this fully embedded platform, the entire factor graph can run on the FPGA component of the SoC. To measure the effectiveness of our design, we used two metrics: clock latency and resource consumption. From the implementation of our approach, we gained some insights about the nature of the trade-off between speed-and-area optimization for our factor graph. Although our approach requires extensive FPGA resources, its architectural design matches the general implementation of a factor graph; hence, it is very flexible and may be extended for a larger factor graph network. From the implementation and experimental results, our approach is proven to be a flexible design with high customizability and impressive performance. With these results, we are confident that we have already built an important fundamental framework for a powerful embedded factor graph that opens many possibilities for further exploration. We envision future

applications of our embedded factor graph in the domain of cognitive intelligence, especially in the direction of a massively distributed computing engine.

Acknowledgement

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

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.micpro.2018.04.002.

References

[1] M. Wainwright, M. Jordan, Graphical models, exponential families, and variational inference, Found. Trends Mach. Learn. 1 (1–2) (2008) 1–305, http://dx.doi.org/10.1561/2200000001.
[2] I. Cox, J. Leonard, Modeling a dynamic environment using a bayesian multiple hypothesis approach, Artif. Intell. 66 (0) (1994) 311–344.
[3] M. Montemerlo, S. Thrun, Simultaneous localization and mapping with unknown data association using FastSLAM, IEEE Int. Conf. Robotics and Automation, Taipei, Taiwan. (2003).
[4] M. Toussaint, C. Goerick, A bayesian view on motor control and planning, From Motor Learning to Interaction Learning in Robots, Berlin: Springer, 2010.
[5] M. Toussaint, N. Plath, N. Jetchev, Integrated motor control, planning, grasping and high-level reasoning in a blocks world using probabilistic reasoning, IEEE International Conference on Robotics and Automation (ICRA) 2010, Anchorage, Alaska, (2010).
[6] A. Hommersom, P.J. Lucas, Using bayesian networks in an industrial setting: making printing systems adaptive, 19th European Conference on Artificial Intelligence (ECAI2010), Lisbon, Portugal, (2010).
[7] C. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.
[8] F. Kschischang, B. Frey, H.-A. Loeliger, Factor graphs and the sum-product algorithm, IEEE Trans. Inf. Theory 47 (2) (2001) 498–519.
[9] J. Yedidia, W. Freeman, Y. Weiss, Constructing free-energy approximations and generalized belief propagation algorithms, IEEE Trans. Inf. Theory 51 (7) (2005) 2282–2312.
[10] B.J. Frey, N. Jojic, A comparison of algorithms for inference and learning in probabilistic graphical models, IEEE Trans. Pattern Anal. Mach. Intell. 27 (9) (2005) 1–25.
[11] K. Murphy, The bayes net toolbox for MATLAB, Comput. Sci. Stat. 33 (2001) 2001.
[12] H. Guo, W. Hsu, A survey of algorithms for real-time Bayesian network inference, AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems 2002, Edmonton, Alberta, Canada, (2002).
[13] P. Ramadge, W. Wonham, The control of discrete event systems, Proc. IEEE 77 (1) (1989) 81–98, http://dx.doi.org/10.1109/5.21072.
[14] W. Savich, M. Moussa, S. Areibi, The impact of arithmetic representation on implementing MLP-BP on FPGAs: a study, IEEE Trans. Neural Networks 18 (1) (2007) 240–252.
[15] H.-A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, F. Kschischang, The factor graph approach to model-based signal processing, Proc. IEEE 95 (6) (2007) 1295–1322, http://dx.doi.org/10.1109/JPROC.2007.896497.
[16] D.-N. Ta, M. Kobilarov, F. Dellaert, A factor graph approach to estimation and model predictive control on unmanned aerial vehicles, International Conference on Unmanned Aircraft Systems (ICUAS) 2014, (2014), pp. 181–188, http://dx.doi.org/10.1109/ICUAS.2014.6842254.
[17] E. Rückert, G. Neumann, M. Toussaint, W. Maass, Learned graphical models for probabilistic planning provide a new class of movement primitives, Front. Comput. Neurosci. 6 (97) (2013).
[18] B. Williams, M. Toussaint, A. Storkey, Modelling motion primitives and their timing in biologically executed movements, in: J. Platt, D. Koller, Y. Singer, S. Roweis (Eds.), Advances in Neural Information Processing Systems 20, Curran Associates, Inc., 2008, pp. 1609–1616.
[19] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, 1988.
[20] S. Deneve, P. Latham, A. Pouget, Reading population codes: a neural implementation of ideal observers, Nat. Neurosci. 2 (8) (1999) 740–745.
[21] S. Wu, S. Amari, H. Nakahara, Population coding and decoding in a neural field: a computational study, Neural Comput. 14 (5) (2002) 999–1026.
[22] R. Ince, R. Senatore, E. Arabzadeh, F. Montani, M. Diamond, S. Panzeri, Information-theoretic methods for studying population codes, Neural Netw. 23 (6) (2010) 713–727.
[23] E. Doi, M. Lewicki, A simple model of optimal population coding for sensory systems, PLoS Comput. Biol. 10 (8) (2014) 1–14, http://dx.doi.org/10.1371/journal.pcbi.1003761.
[24] J. Beck, A. Pouget, K. Heller, Complex inference in neural circuits with probabilistic population codes and topic models, Adv. Neural Inf. Process. Syst. 25 (2012) 3068–3076.
[25] I. Sugiarto, P. Maier, J. Conradt, Reasoning with discrete factor graph, IEEE International Conference on Robotics, Biomimetics, and Intelligent Computational Systems (ROBIONETICS) 2013, (2013), pp. 170–175. doi:10.1109ROBIONETICS.2013.6743599 .
[26] A. Mathis, A. Herz, M. Stemmler, Resolution of nested neuronal representations can be exponential in the number of neurons, Phys. Rev. Lett. 109 (2012) 018103, http://dx.doi.org/10.1103/PhysRevLett.109.018103.
[27] F. Dellaert, M. Kaess, Square root SAM: simultaneous location and mapping via square root information smoothing, Int. J. Rob. Res. (IJRR) 25 (12) (2006) 1181–1213. Special issue on RSS 2006
[28] M. Kaess, A. Ranganathan, F. Dellaert, ISAM: incremental smoothing and mapping, IEEE Trans. Rob. (TRO) 24 (6) (2008) 1365–1378, http://dx.doi.org/10.1109/TRO.2008.2006706.
[29] J. Mooij, Libdai: a free and open source c++ library for discrete approximate inference in graphical models, J. Mach. Learn. Res. 11 (2010) 2169–2173.
[30] I. Sugiarto, J. Conradt, Discrete belief propagation network using population coding and factor graph for kinematic control of a mobile robot, IEEE International Conference on Computational Intelligence and Cybernetics (CYBERNETICSCOM) 2013, Yogyakarta, Indonesia, (2013), pp. 136–140. doi:10.1109CyberneticsCom.2013.6865797
[31] V. Manshinghka, Natively Probabilistic Computation, Ph.D. thesis, Department of Brain & Cognitive Sciences, Massachusetts Institute of Technology, 2009.
[32] F. Palmieri, Learning non-linear functions with factor graphs, IEEE Trans. Signal Process. 61 (17) (2013) 4360–4371.
[33] V. Namasivayam, A. Pathak, V. Prasanna, Scalable parallel implementation of Bayesian network to junction tree conversion for exact inference, The 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06), Ouro Preto, Minas Gerais, Brasil, (2006).
[34] V. Sudhakar, C. Murthy, Efficient mapping of backpropagation algorithm onto a network of workstations, IEEE Trans. Syst., Man, Cybern., Part B 28 (1998) 841–849.
[35] S. Aluru, N. Jammula, A review of hardware acceleration for computational genomics, IEEE Design Test 31 (1) (2014) 19–30, http://dx.doi.org/10.1109/MDAT.2013.2293757.
[36] A. Papadopoulosa, I. Kirmitzogloub, V. Promponasb, T. Theocharides, FPGA-based hardware acceleration for local complexity analysis of massive genomic data, Integr., VLSI J. 46 (3) (2013) 230–239.
[37] M. Silberstein, A. Schuster, D. Geiger, A. Patney, J. Owens, Efficient computation of sum-products on GPUs through software-managed cache, Proceedings of the 22nd annual international conference on Supercomputing (ICS'08), ACM, New York, NY, USA, 2008, pp. 309–318, http://dx.doi.org/10.1145/1375527.1375572.
[38] N. Piatkowski, Parallel algorithms for GPU accelerated probabilistic inference, NIPS 2011: workshop on parallel and large-scale machine learning, Sierra Nevada, Spain, (2011).
[39] R. Nasre, M. Burtscher, K. Pingali, Morph algorithms on GPUs, the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13), Shenzhen, China, (2013), pp. 147–156.
[40] Y. Zhao, J. Xu, Y. Gao, A parallel algorithm for bayesian network parameter learning based on factor graph, 2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI2013), Washington DC, USA, (2013), pp. 506–512.
[41] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113, http://dx.doi.org/10.1145/1327452.1327492.
[42] A. Steimer, Neurally Inspired Models of Belief-Propagation in Arbitrary Graphical Models, ETH Zürich, Switzerland, 2012 Ph.D. thesis.
[43] IEEE, IEEE Standard for Binary Floating-Point Arithmetic, The Institute of Electrical and Electronics Engineers, Inc., 2008.
[44] P. Zhao, S. Cui, Y. Gao, R. Silvera, J. Amaral, Forma: a framework for safe automatic array reshaping, ACM Trans. Programm. Lang. Syst. 30 (1) (2007).
[45] Y. Asher, N. Rotem, Automatic memory partitioning: increasing memory parallelism via data structure partitioning, IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2010, (2010), pp. 155–161.
[46] N. Margolus, An FPGA architecture for DRAM-based systolic computations, The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, (1997), pp. 2–11, http://dx.doi.org/10.1109/FPGA.1997.624599.
[47] E. Marchi, M. Cervetto, M. Tenorio, A DDR3 memory based time interleaving FPGA implementation for ISDB-T standard, Programmable Logic (SPL), 2011 VII Southern Conference on, (2011), pp. 1–5, http://dx.doi.org/10.1109/SPL.2011.5782616.
[48] S. Aqueel, K. Khare, Design and FPGA implementation of DDR3 SDRAM controller for high performance, Int. J. Comput. Sci. Inf. Technol. (IJCSIT) 3 (4) (2011) 101–110.
[49] D. Capalija, T. Abdelrahman, An architecture for exploiting coarse-grain parallelism on FPGAs, International Conference on Field-Programmable Technology 2009 (FPT 2009), (2009), pp. 285–291, http://dx.doi.org/10.1109/FPT.2009.5377658.
[50] P. Banerjee, M. Haldar, A. Nayak, V. Kim, D. Bagchi, S. Pal, N. Tripathi, A behavioral synthesis tool for exploiting fine grain parallelism in FPGAs, in: S. Das, S. Bhattacharya (Eds.), Distributed Computing, Lecture Notes in Computer Science, 2571 Springer Berlin Heidelberg, 2002, pp. 246–256, , http://dx.doi.org/10.1007/3-540-36385-8_25.

Indar Sugiarto is a lecturer in the Department of Electrical Engineering at Petra Christian University. Currently he works as a Research Associate in the Advanced Processor Technology Group of the School of Computer Science at the University of Manchester. He holds B.Sc. in Electrical Engineering from Institut Teknologi Sepuluh Nopember, Indonesia, M.Sc. degrees in Information and Automation Engineering from Universität Bremen, Germany, and Ph.D. in Electrical and Computer Engineering from Technische Universität München, Germany. His main research interests are computational intelligence, Bayesian machine learning, reconfigurable computing, and robotics.

Jörg Conradt is a Junior Professor at the Technische Universität München in the Faculty of Electrical Engineering and Information Technology, Institute of Automation and Control Engineering. The laboratory is affiliated with TUM's Competence Center on NeuroEngineering and the Munich Bernstein Center for Computational Neuroscience. He holds an M.S. degree in Computer Science/Robotics from the University of Southern California, a Diploma in Computer Engineering from TU Berlin and a Ph.D. in Physics/Neuroscience from ETH Zurich. His research group on Neuroscientific System Theory (http://www.nst.ei.tum.de) investigates key principles by which information processing in brains works, and applies those to real-world interacting technical systems.

Cek Plagiarism Modular SoC Paper

ORIGINALITY REPORT

6%	%	6%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

MATCH ALL SOURCES (ONLY SELECTED SOURCE PRINTED)

2%

★ Xiwei WU, Bing XIAO, Cihang WU, Yiming GUO, Lingwei LI. "Factor graph based navigation and positioning for control system design: A review", Chinese Journal of Aeronautics, 2022

Publication

Exclude quotes	Off	Exclude matches	< 1%
Exclude bibliography	Off		