

REPORT DOCUMENTATION PAGE			Form Approved OMB NO. 0704-0188		
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA, 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 14-02-2019		2. REPORT TYPE Final Report		3. DATES COVERED (From - To) 15-Apr-2013 - 14-Apr-2016	
4. TITLE AND SUBTITLE Final Report: Stochastic Computing Machines Enabled by DNA Self-Assembly			5a. CONTRACT NUMBER W911NF-13-1-0096		
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBER		
6. AUTHORS			5d. PROJECT NUMBER		
			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAMES AND ADDRESSES Duke University C/O Office of Research Support 2200 W. Main St., Ste. 710 Durham, NC 27705 -4677			8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES) U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211			10. SPONSOR/MONITOR'S ACRONYM(S) ARO		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S) 63907-LS-DRP.2		
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Chris Dwyer
a. REPORT UU	b. ABSTRACT UU	c. THIS PAGE UU			19b. TELEPHONE NUMBER 919-660-5275

RPPR Final Report
as of 01-Apr-2019

Agency Code:

Proposal Number: 63907LSDRP

Agreement Number: W911NF-13-1-0096

INVESTIGATOR(S):

Name: Chris Dwyer
Email: c.dwyer@duke.edu
Phone Number: 9196605275
Principal: Y

Organization: **Duke University**

Address: C/O Office of Research Support, Durham, NC 277054677

Country: USA

DUNS Number: 044387793

EIN: 560532129

Report Date: 14-Jul-2016

Date Received: 14-Feb-2019

Final Report for Period Beginning 15-Apr-2013 and Ending 14-Apr-2016

Title: Stochastic Computing Machines Enabled by DNA Self-Assembly

Begin Performance Period: 15-Apr-2013

End Performance Period: 14-Apr-2016

Report Term: 0-Other

Submitted By: Alvin Lebeck

Email: alvy@duke.edu

Phone: (919) 660-5551

Distribution Statement: 1-Approved for public release; distribution is unlimited.

STEM Degrees:

STEM Participants:

Major Goals: The overall goal of this project is to determine optimal configuration and dynamics of spatially arranged chromophores on a self-assembled DNA grid, based on the principle of energy migration. Photonic wires are molecular devices that convey excited-state energy from an input to an output. The photonic wire in this project focuses on an array of chromophores engaging in multi-step FRET. This wire borrows from Resonance Energy Transfer and Energy migration principles predicted by Forster's theory. Energy migration works on the principle of homo-transfer, which involves identical dye molecules (chromophores) exchanging energy. Their band gap is such that there is a large enough singlet spread, and a small stokes shift with overlapping absorption and emission spectra, which results in a back-and-forth exciton diffusion. Hetero-transfer, on the other hand, uses a Donor-Acceptor configuration in which a Donor dye molecule excites an acceptor whose absorption spectrum overlaps with the donor's emission spectrum. Each subsequent acceptor then becomes a donor to the next-in-line acceptor.

Accomplishments: See uploaded report.

Training Opportunities: Nothing to Report

Results Dissemination: Nothing to Report

Honors and Awards: Nothing to Report

Protocol Activity Status:

Technology Transfer: Nothing to Report

PARTICIPANTS:

Participant Type: Graduate Student (research assistant)

Participant: Craig Laboda

Person Months Worked: 8.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

RPPR Final Report
as of 01-Apr-2019

Other Collaborators:

Participant Type: Graduate Student (research assistant)

Participant: Raul Vyas

Person Months Worked: 9.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: Graduate Student (research assistant)

Participant: Siyang Wang

Person Months Worked: 9.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: Graduate Student (research assistant)

Participant: Arjun Rallapalli

Person Months Worked: 8.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: Undergraduate Student

Participant: Chuanze Wu

Person Months Worked: 1.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: Postdoctoral (scholar, fellow or other postdoctoral position)

Participant: Vishwa Nellore

Person Months Worked: 1.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: PD/PI

Participant: Christopher Dwyer

Person Months Worked: 2.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

RPPR Final Report
as of 01-Apr-2019

National Academy Member: N
Other Collaborators:

Participant Type: Co PD/PI

Participant: Alvin Lebeck

Person Months Worked: 3.00

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Funding Support:

Nanoscale Resonance Energy Transfer-based Devices for Probabilistic Computing

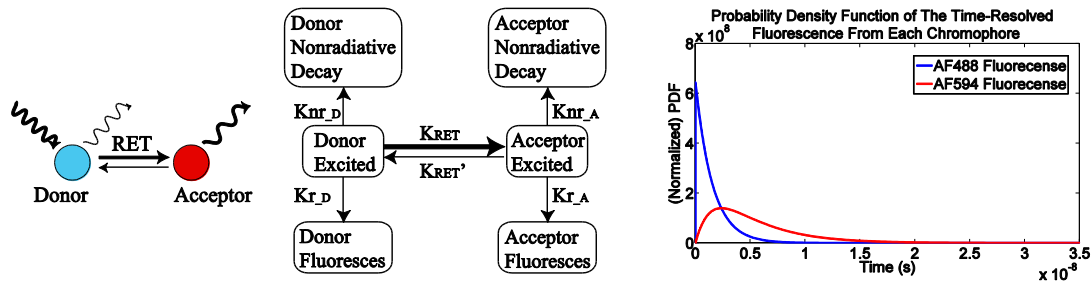
In this work, resonance energy transfer between chromophores is proposed as an enabling technology for probabilistic computing functional units. As a natural Continuous Time Markov Chain, resonance energy transfer networks can physically implement efficient samplers with arbitrary probability distributions and has great potential for accelerating probabilistic algorithms.

1. Phase-type distribution

Phase-type distributions can form a dense set in the field of all positive-valued distributions and approximate any positive-valued distribution. The Poisson process and Exponential distribution naturally exist in nanoscale processes (e.g., chemical reactions, intermolecular energy transfer, luminescence, and electrostatics) and, more importantly, can be flexibly convolved and mixed to form different instances of phase-type distributions, thus they provide unique opportunities for nanoscale physical implementation and approximating general distributions.

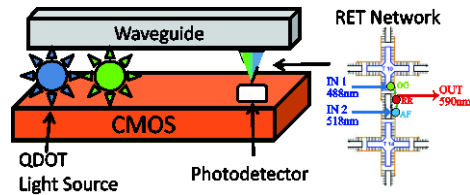
2. Molecular implementation using Resonance Energy Transfer Networks

RET networks can be designed to achieve different phase-type distributions. In a RET network, the time to RET transfer, after the donor is excited, follows an exponential distribution between each chromophore pair, therefore the sequence of RET transfers and the time from the exciton entering the chromophore network until leaving the network (i.e., decaying) follows a phase-type distribution. The RET transfer between a chromophore pair with a transfer rate of k_{RET} physically implements a phase transition with a transition rate of $\lambda = k_{RET}$ in a phase-type distribution, and the geometry of the chromophore network controls how these phases are convolved and mixed to form the phase-type distribution. Specifically, the RET transfer rate between each chromophore pair and the decay rate of each relaxation pathway constitute the transition matrix Q of the absorbing CTMC that defines the phase-type distribution. The figures below illustrate a RET network with two chromophore and its corresponding CTMC model and fluorescence distributions which are phase-type distributions.



Meanwhile, RET networks can be conveniently and economically fabricated using DNA self-assembly with sub-nanometer precision. Hence RET technology becomes a natural substrate for implementing phase-type distributions.

3. Using a RET circuit to sample from a general continuous or discrete distribution



4. Suitable probabilistic algorithms

Repeated random sampling from known distributions is the core of Monte Carlo methods which are widely used for optimization, numerical integration and sampling from a target probability distribution. Specifically, Monte Carlo Markov Chain (MCMC), a subset of Monte Carlo methods, is a critical approach to solving Bayesian Inference problems. When a large number of samples are required, directly generating samples from RET-based functional units can potentially accelerate Monte Carlo methods in the absence of the overhead required to computationally generate samples from a continuous or discrete distribution (e.g., inverse transform sampling and rejection sampling). In addition, designing hardware based on generating samples for probabilistic algorithms may also bring savings in energy efficiency and circuit area. Moreover, the natural entropy used to generate random samples from a RET circuit makes it a natural, or 'true', random number generator which may suffer less from distributional bias.

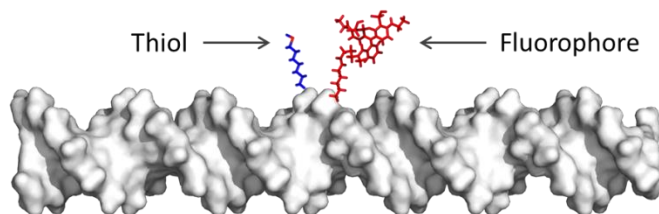
A large number of probabilistic algorithms can potentially benefit from RET circuits, including: Bayesian Networks, Probabilistic Cellular Automata, Random Neural Networks, etc. In addition, we found a novel f -Divergence based method for model comparison which is suitable for RET network based sampling. Unlike the likelihood-based method, this method is mainly based on sampling and requires minimal numerical computation.

Localized Triplet Reduction

When excited in the presence of thiolated reducing agents, certain fluorophores can enter a dark, long-lived reduced triplet state.¹ This molecular state can be used to modulate the flow of excitons in RET networks, as molecules in the reduced triplet state cannot accept excitons from neighboring fluorophores.² In addition, this state can last for hours at a time in the absence of oxygen, offering a form of non-volatile molecular memory.³ These unique properties make the reduced triplet state a promising candidate for building integrated molecular circuits.

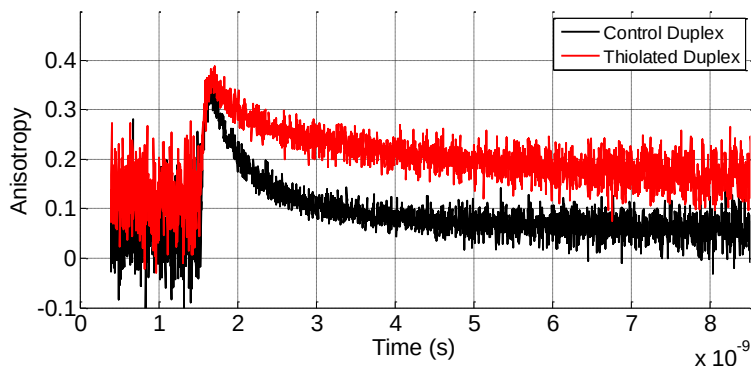
Thus far, all of the examples from the literature that utilize the reduced triplet state do so via bulk reduction in which the reducing agent is simply added in bulk to the solvent containing the fluorophores. With bulk reduction, any excited fluorophore in a RET network has a non-zero probability of transitioning into the reduced triplet state. This is not ideal for building integrated molecular circuits composed of many different fluorophores since certain portions of circuitry should never be subject to modulation, *e.g.* wires connecting circuit components. To fully integrate this mechanism into RET logic, we need the ability to choose which fluorophores enter the reduced triplet state. This concept is called localized triplet reduction.

One method for localizing triplet reduction is to place the fluorophore and reducing agent in close proximity to one another using a DNA duplex as a substrate, as illustrated in the figure below. By tethering the reducing agent to the DNA, rather than allowing the molecule to freely diffuse in solution, the reducing agent should be restricted to only interacting with its neighboring fluorophore.



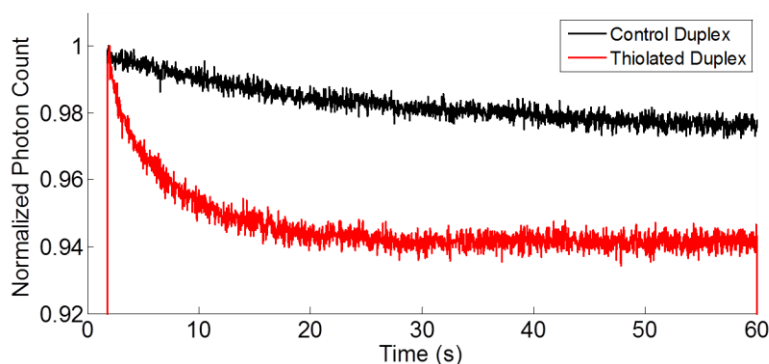
Cartoon illustrating the concept of localized triplet reduction.

To test this concept, we designed a DNA duplex that places a thiol group directly next to ATTO 532, a commonly bulk reduced fluorophore. Two versions of the duplex were then assembled: a thiolated duplex containing both the reducing agent and the fluorophore, and a control duplex containing only the fluorophore. Fluorescence anisotropy from the two duplexes, shown below, indicates that the rotation of ATTO 532 is more hindered, *i.e.* has a higher limiting anisotropy, when attached to the thiolated duplex than the control duplex. This result suggests that, when attached to the thiolated duplex, ATTO 532 physically interact with the local thiol group as intended.



Fluorescence anisotropy measurements of the thiolated and control duplexes.

Despite this physical interaction, the reduced triplet state was not observed upon excitation of the thiolated duplex. We did find, however, that the thiolated duplex consistently photobleaches at a faster rate than the control, as illustrated by the fluorescence time traces below. This is most likely due to the external heavy atom effect. The presence of the sulfur atom in the local thiol group increases the intersystem crossing rate of ATTO 532, which ultimately leads to increased photobleaching. While negative, this unexpected result demonstrates that local thiol groups could be used to modulate photobleaching pathways of RET networks.



Fluorescence time traces from the thiolated and control duplexes.

In the initial localization experiments described above, the fluorophore and thiol group were placed directly next to one another. To determine whether there is a particular separating distance that enables reduction, four additional fluorophore-thiol positions were probed. All four of these positions resulted in similar photobleaching trends to the initial position, but none of them showed any signs of

triplet reduction. Currently, we believe that this lack of localized reduction is due to geminate recombination, which forces the electron-hole pair to recombine immediately after the reduction process. This mechanism proceeds with incredibly high efficiency for bulk reduction and is likely to be even more efficient when the diffusional volumes are constrained.⁴ To test this theory, the DNA duplex is being redesigned to include a sequence of three G bases that act as a hole trap due to their low oxidation potential.⁵ This will help physically separate the electron and hole after the reduction process, thereby increasing the lifetime of the reduced triplet and allowing for a longer observational time window.

References:

1. van de Linde, S.; Krstic, I.; Prisner, T.; Doose, S.; Heilemann, M.; Sauer, M. Photoinduced formation of reversible dye radicals and their impact on super-resolution imaging. *Photoch Photobio Sci* 2011, 10, 499-506.
2. Fujii, R.; Nishimura, T.; Ogura, Y.; Tanida, J. Nanoscale energy-route selector consisting of multiple photo-switchable fluorescence-resonance-energy-transfer structures on DNA. *Opt Rev* 2015, 1-6.
3. Nishimura, T.; Ogura, Y.; Tanida, J. A Nanoscale Set-Reset Flip-Flop in Fluorescence Resonance Energy Transfer-Based Circuits. *Applied Physics Express* 2013, 6, 3.
4. Holzmeister, P.; Gietl, A.; Tinnefeld, P. Geminate recombination as a photoprotection mechanism for fluorescent dyes. *Angew Chem Int Ed Engl* 2014, 53, 5685-8.
5. Kawai, K.; Osakada, Y.; Takada, T.; Fujitsuka, M.; Majima, T. Lifetime regulation of the charge-separated state in DNA by modulating the oxidation potential of guanine in DNA through hydrogen bonding. *Journal of the American Chemical Society* 2004, 126, 12843-12846.

Design and a computer-aided simulation of a RET wire

The overall goal of this project is to determine optimal configuration and dynamics of spatially arranged chromophores on a self-assembled DNA grid, based on the principle of energy migration. Photonic wires are molecular devices that convey excited-state energy from an input to an output. The photonic wire in this project focuses on an array of chromophores engaging in multi-step FRET. This wire borrows from Resonance Energy Transfer and Energy migration principles predicted by Forster's theory [1]. Energy migration works on the principle of homo-transfer, which involves identical dye molecules (chromophores) exchanging energy. Their band gap is such that there is a large enough singlet spread, and a small stokes shift with overlapping absorption and emission spectra, which results in a back-and-forth exciton diffusion. Hetero-transfer, on the other hand, uses a Donor-Acceptor configuration in which a Donor dye molecule excites an acceptor whose absorption spectrum overlaps with the donor's emission spectrum. Each subsequent acceptor then becomes a donor to the next-in-line acceptor.

Previous studies have used hetero-FRET with a cascade of energy transfer events [2]. Although highly directional, there are significant energy losses, as the emission energy is red-shifted at each step. Work has also been done on quantifying Forster's approach on energy migration by using multiple identical chromophores engaging in a multi-step homo-FRET [3]. This approach merits low energy losses, but there is a loss in uni-directionality, since the exciton is free to hop onto any chromophore. Attempts for a wire that offers both uni-directionality and low energy losses have also been established, by using identical mediator dyes to facilitate energy transfer between the donor and the acceptor spatially arranged on double-stranded DNA [4]. But the mediators used in this study are intercalating dye molecules, which position themselves in between base-pairs. Major downsides to this are reduced precision and controllability, since intercalation is arbitrary and there is no information on the number of mediator dye molecules per DNA construct. From a design perspective towards building RET circuits, there is a need for a design that not only has low energy losses and directionality, but also offers a highly precise pattern of spatial arranged chromophores. And therefore, using a complex self-assembled DNA grid to

arrange an array of chromophores engaging in a multi-step hetero and homo-FRET for the Donor-Mediator-Acceptor configuration achieves low energy losses, directionality, and offers site-specific modifications and sub-nm precision in placement.

Modeling and Simulation –

In the first quarter of this year, we focused on analytical modeling of the Energy Migrating Photonic wire, to not only yield the best possible configuration for the chromophore network, but also to simulate the design space to come up with a set of experimentally tunable parameters. These parameters would be especially useful in providing an active feedback toward the improvement of experiments. The tool used for simulating the RET network is called RETLab [5]. The results of the simulations below depict pair-wise as well as end-to-end transfer efficiency values for the Donor-Mediator-Acceptor configuration, each at a 2 nm separation.

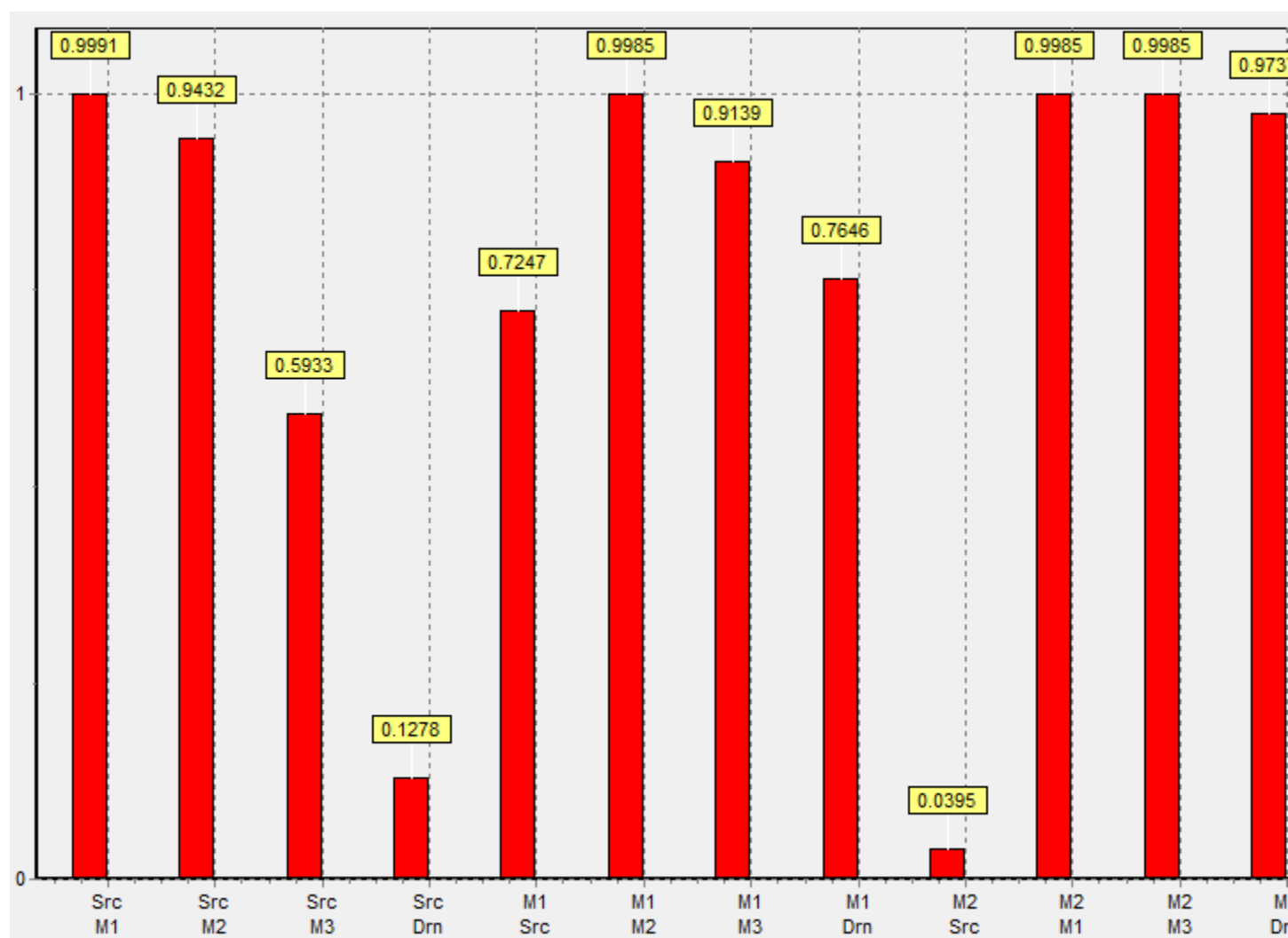


Figure: The bars represent Pair-wise TE for each chromophore combination in the RET network

As seen in the above figure, pair-wise Transfer Efficiencies are not a true indicator of the overall energy transfer of the wire. Therefore, a way to calculate the overall end-to-end transfer

efficiency was determined by performing a time-resolved simulation at the donor excitation wavelength (See Figure below). The ratio of the total number of fluorescence events from the Donor and excitation events from the Acceptor are a measure of the true End-to-End Transfer Efficiency of the system. Simulations were performed for various laser pulse widths as well as continuous lasers. The reasoning behind this was that because exciton diffusion within the homo-FRET section of the wire is bi-directional, there's a need to direct the energy transfer more towards the Acceptor, which is achievable only when the laser is on throughout the simulation time.

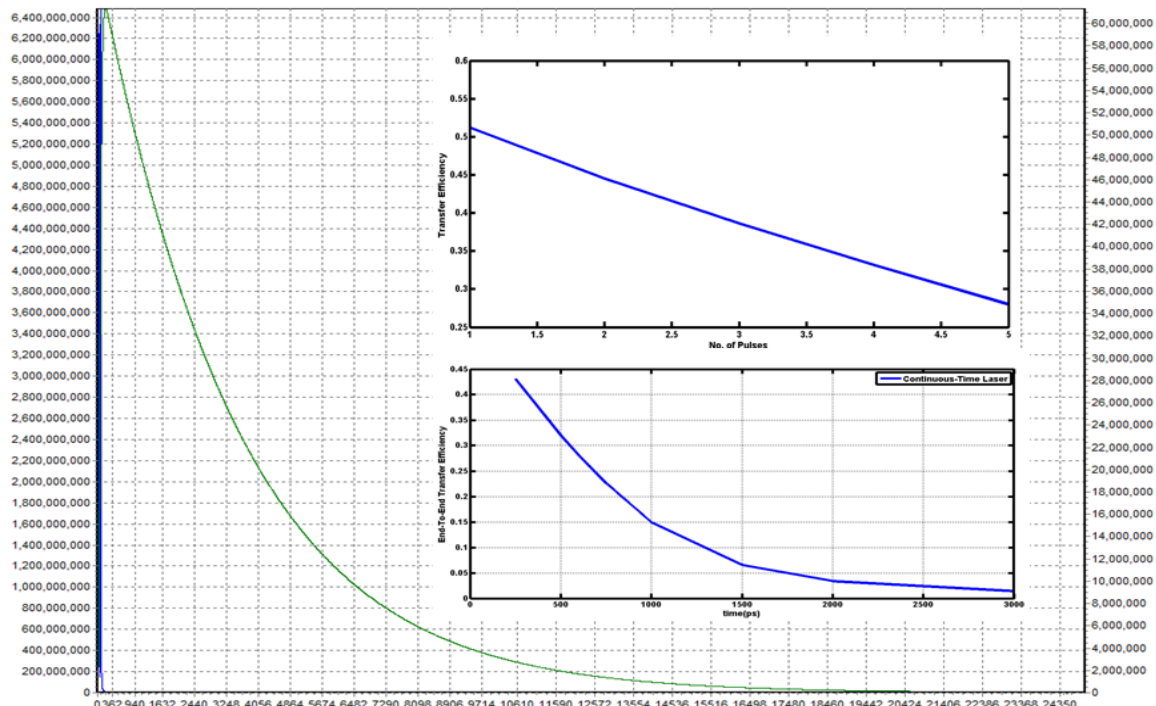


Figure: Time-resolved Simulation of the RET wire, End-to-End TE for different laser pulses; and for a Continuous laser

If we look closer at the fluorescence decay of the Acceptor, we can find that it is analogous to the voltage across the resistor in an RC circuit (figure below).

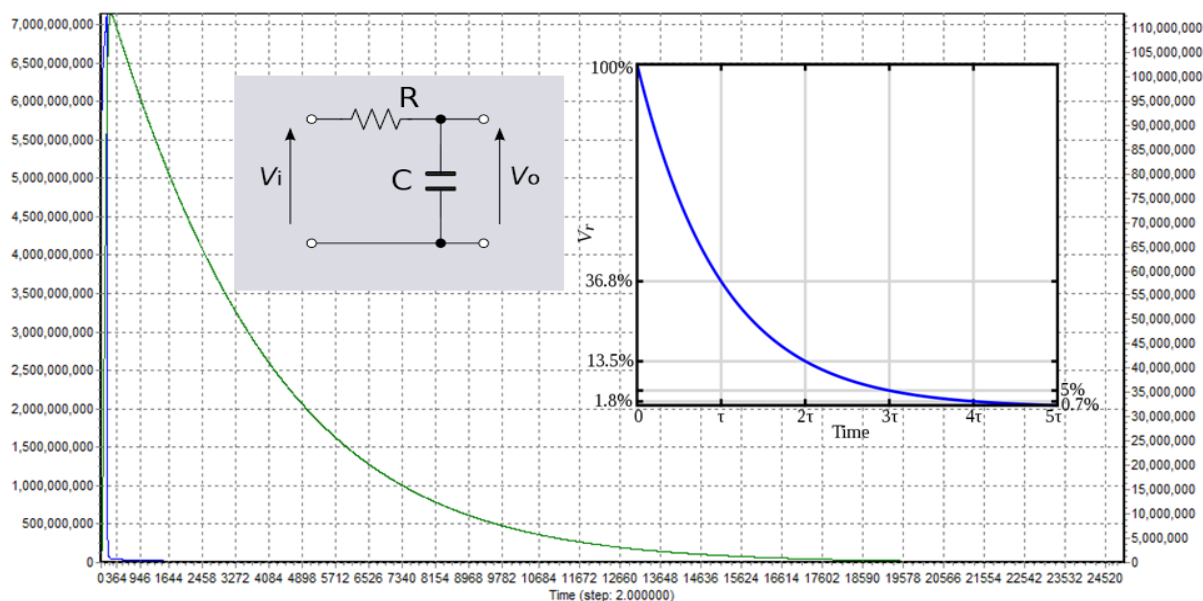


Figure: Comparisons between Acceptor Fluorescence decay and Voltage across R in a RC Circuit

Continuing with analogies, the time constant of an RC circuit can be determined by the time it takes for the capacitor to have discharged to (100 - 63.2) 36.8% of the initial stored charge. Similarly, the fluorescence lifetime of a RET network can be described as time it takes for the excited population to decay to 36.8% of the original population. The figure below plots the fluorescence lifetime of the Acceptor for different laser pulse widths.

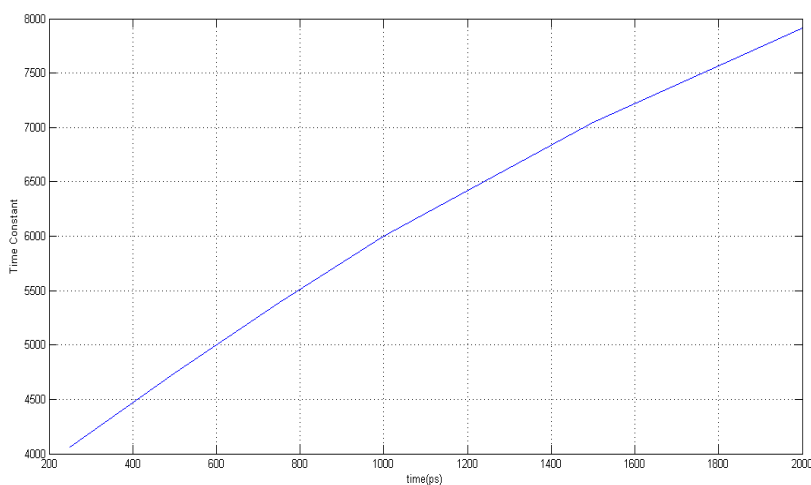
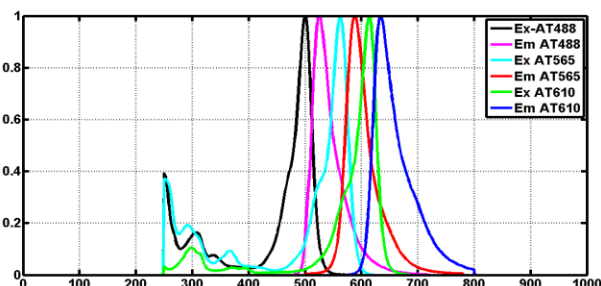


Figure: Times for which total Acceptor Fluorescence is 36.8% of its original value plotted for different laser pulse widths.

Initial Experiment set up – Dye and site location Selection –

Selection was based on good alignment between the Ex-Em spectra of the Donor-Mediator-Acceptor and a small Stokes shift between the mediators (as shown in figure below), large Forster Radii between the mediators, good Quantum Yield, high transfer efficiency (between Donor-mediators and between mediators-acceptor) and Longer Lifetimes.

ATTO DYES	Lifetime (ps)	Extinction Coefficient	Quantum Yield (%)	Transfer Efficiency (at 5 nm separation)
AT488 (Donor)	4,100	90,000	80	0.833 (to AT565)
AT565 (Mediator)	4,000	120,000	89	0.9065 (to AT610)
AT610 (Acceptor)	3,200	150,000	69	



InSequoia Design Studio (a CAD and grid-enabled optimization tool) was used to visualize and select locations on DNA Tile and Grid nanostructures (See figure below). The sites for donor-mediator-Acceptor were chosen to be at least 2.2 nm apart, for an overall wire length of 10.9 nm. A primary amino modifier group was used on Thymidine to attach an NHS ester-modified dye molecule.

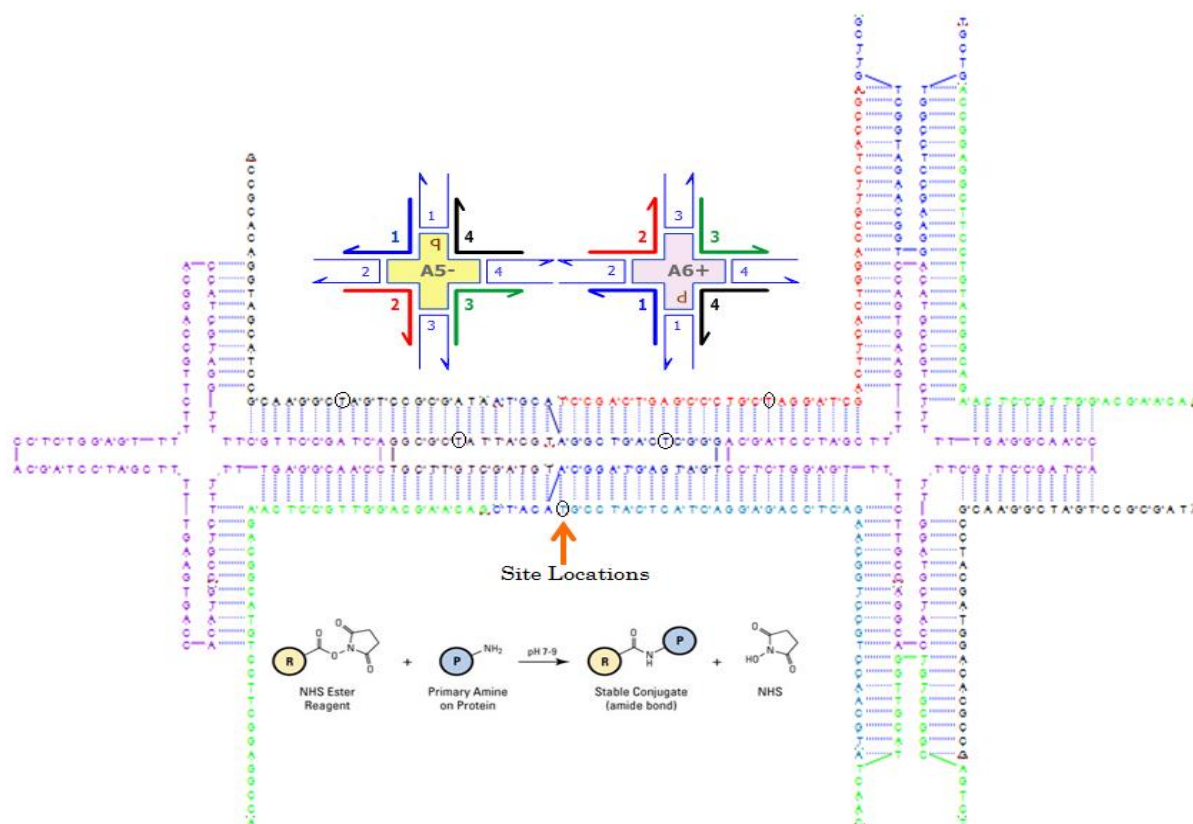


Figure: Depicting Site Locations on the DNA Nanostructures, with attachment chemistry

In the next quarter, we intend to focus on the experimental part of the project, which includes putting together self-Assembled DNA grids, attaching chromophores on these grids, and running steady-state fluorescence measurements to reaffirm the different RET processes along the wire.

References –

1. Förster, Th. (1948), "Zwischenmolekulare Energiewanderung und Fluoreszenz". *Ann. Phys.*, 437: 55–75.
2. Spillmann, Christopher M., et al. "Extending FRET cascades on linear DNA photonic wires." *Chemical Communications* 50.55 (2014): 7246-7249.
3. Hannah, K. C.; Armitage, B. A. *Acc. Chem. Res.* 2004, 37, 845–853.
4. Hannestad, Jonas K., Peter Sandin, and Bo Albinsson. "Self-assembled DNA photonic wire for long-range energy transfer." *Journal of the American Chemical Society* 130.47 (2008): 15889-15895.
5. Mottaghi, Mohammad D., Arjun Rallapalli, and Chris Dwyer. "RETLab: A fast design-automation framework for arbitrary RET networks." *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014.

Supplementary Material

We provide the following supplementary material:

1. Additional illustrations of combinational stochastic circuits.
2. Additional examples of stochastic transition circuits, elaborating on the connection with the building blocks of Markov chain Monte Carlo algorithms.
3. Additional empirical data that bears on the claim that extremely low bit precision (and therefore low circuit area and high computation density) is achievable with minimal reduction in accuracy.
4. Implementation details for our depth, motion perception, and perceptual learning circuits, as well as our compiler.
5. Mathematical background relevant for the Poisson-spiking implementation of stochastic digital circuits.

Binomial sampling circuits.

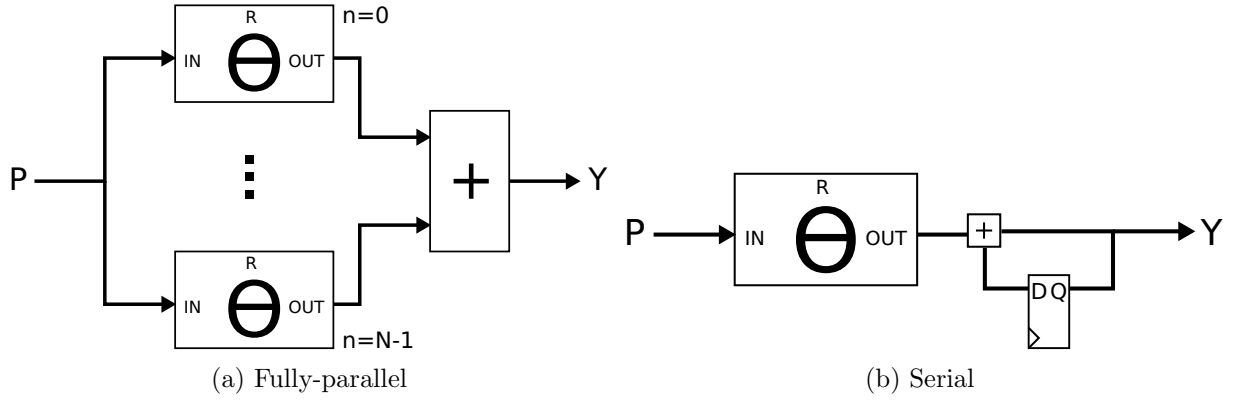


Figure 1: Two implementations of a binomial sampling circuit: a faster fully space-parallel design and a slower more area-efficient bit-serial one. Both produce samples from a binomial distribution with N possible output values and a weight of p .

Conditional independence gives the designer tremendous flexibility in making trade-offs between silicon area and the time required to produce a sample. Consider a Bernoulli distribution (figure 1), the distribution on the number of heads h from flipping N biased coins with weight p . Because the coins flips are independent, we can flip them simultaneously (via theta gates) and sum the result. Alternatively, we can accumulate the flips of a single theta gate, generating the sample N cycles later.

Stochastic Transition Circuits

The stochastic transition circuit specification we describe, requiring only ergodic convergence to $P(S|X)$ by repeated iteration of T , can accomodate a wide range of implementations. One approach to designing useful transition circuits is to draw from the literature on Markov chain Monte Carlo algorithms.

1 Iteration and composition

Markov chain transition kernels obey composition rules. If our state space is $p(x_A, x_B) \in \mathbb{R}^\#$, and we have a kernel $T_{A|B}$ that will ergodically generate samples from $p(x_A|x_B)$ and a kernel $T_{B|A}$ which will ergodically sample from $p(x_B|x_A)$, then $T = T_{A|B}T_{B|A}$ will ergodically produce samples from $p(x_A, x_B)$ (as long as no states have zero probability). Similarly, $T = \alpha T_{A|B} + (1 - \alpha)T_{B|A}$ for $\alpha \in (0, 1)$, a stochastic mixture of two kernels, also preserves invariance and extends the domain of ergodic convergence. References for these and other composition laws are provided in the main text.

We can construct transition circuits that ergodically produce samples from the indicated conditional probability distributions using many methods — such as Gibbs sampling and Metropolis-Hastings — and combine them to sample from larger, more complex distributions.

Figure 2a shows a typical stochastic transition circuit, which produces an output sample S_{t+1} conditioned on inputs X . The only requirement is that $P(S|X)$ be left invariant by the transition $P(S_{t+1}|S_t)$.

2 Discrete Gibbs Sampling

Gibbs sampling is a technique which produces a Markov chain whose ergodic distribution is the indicated target distribution of $p(x_1, \dots, x_N)$ provided we can produce samples from the probability distribution of one variable conditioned on the others.

That is, to construct a Markov chain to sample from $p(x_A, x_B, x_C)$ we need to iteratively draw samples

$$x_A \sim p(x_A|x_B, x_C) \tag{1}$$

$$x_B \sim p(x_B|x_A, x_C) \tag{2}$$

$$x_C \sim p(x_C|x_A, x_B) \tag{3}$$

$$\tag{4}$$

Note that if the state space of x is discrete, then it is easy to sample from x by computing $p(x_A, x_B, x_C)$ with x_B and x_C fixed for all values of x_A , and then normalizing and sampling the resulting distribution.

Figure 2c shows T_{gibbs} , a typical stochastic transition circuit allowing gibbs sampling. Note that the current value of S , that is, S_t , does not appear anywhere. Rather, this circuit samples directly from $P(S|X)$. Figures 2d,e, and f show example implementations for T_{Gibbs} highlighting the trade-offs. Figure 2d shows the explicit, internal mathematics necessary to, in parallel, accumulate and normalize the energies before producing a sample. Figure 2e simply looks up the table of appropriate probabilities conditioned on the inputs. This is the fastest option, but consumes space that grows exponentially in the number of bits needed to describe S and S (assuming there are no symmetries which can be used to compress the table). Figure 2f is a serial sampling gate, which we describe in further detail in the subsequent supplemental section. It stores the intermediate energy values, normalizes them in a second pass through the energy ram, and produces a sample. This design is optimized for sampling from discrete distributions with large state spaces (such as 1024 or more possible output symbols).

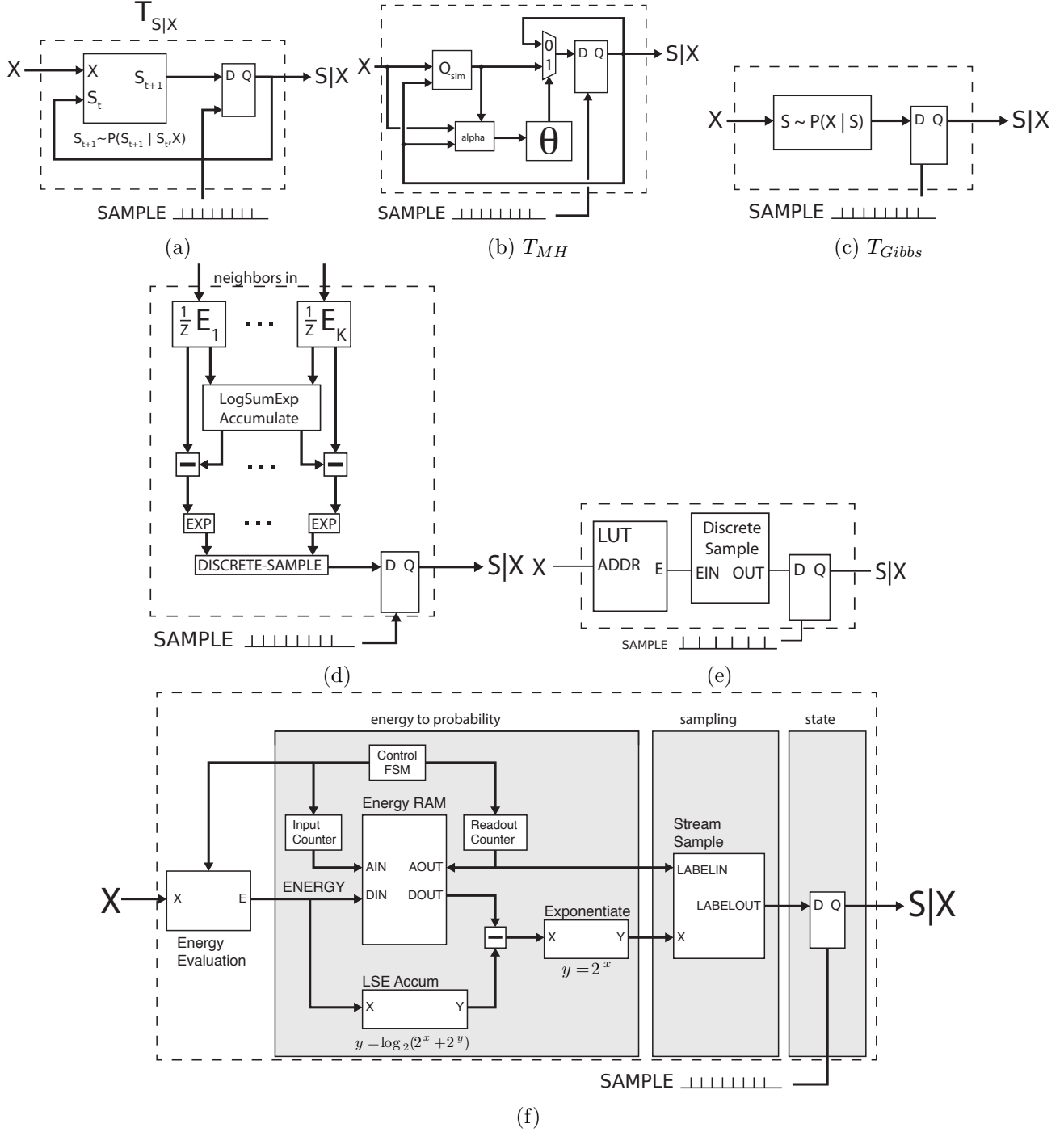


Figure 2: a.) Stochastic transition circuits take a set of conditioning inputs X and produce a sample S_{t+1} . The only requirement is that $P(S|X)$ be left invariant by the transition $P(S_{t+1}|S_t)$. b.) A M-H transition circuit involves a proposal distribution Q_{sim} to propose a new value for S_{t+1} , accepting or rejecting based upon the MH acceptance ratio. c.) Gibbs sampling transition circuits produce a value S_{t+1} conditioned on X only – they ignore the current value of S . d.) Space-parallel gibbs transition circuit with mathematical internals. e.) Lookup-table based gibbs transition circuit f.) Serial gibbs transition circuit stores the energy value associated with each possible output state, and then normalizes and samples.

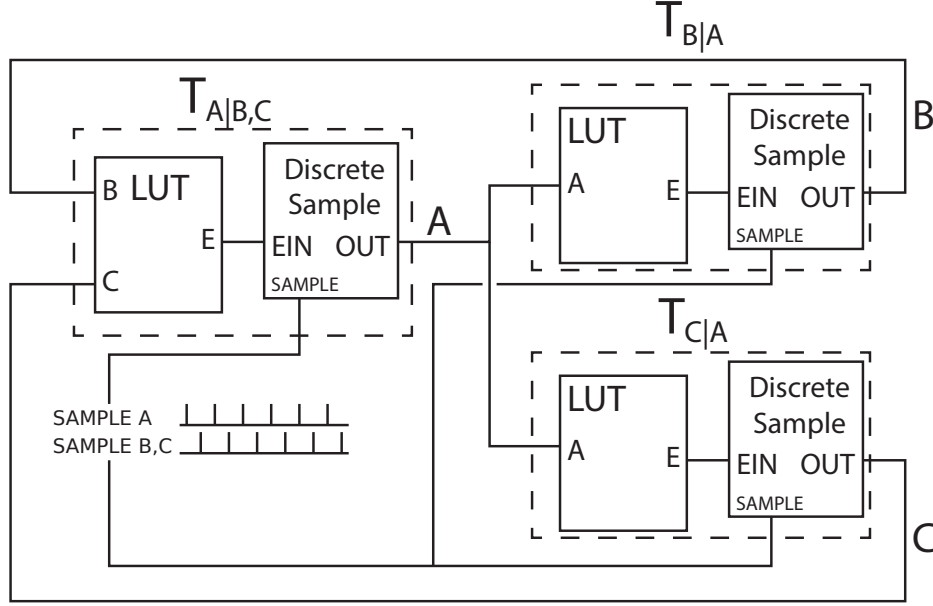


Figure 3: Three transition circuits combined to produce samples from $p(a, b, c)$. The conditional independence in the problem allows for B and C to be sampled in parallel.

3 Metropolis-Hastings

If sampling from the conditional distributions is not possible, an ergodic Markov chain can still be constructed using a collection of sampling elements that approximately sample from the conditional distributions. This still allows us to ergodically sample from $p(x_1, x_2, x_3)$ (continue the above example). The algorithm to do this, called Metropolis-Hastings, is the root of most MCMC schemes, and works by proposing new values and then “accepting” or “rejecting” them according to specific criteria.

Colloquially, proposal distribution $q(x'|x)$ is used to sample a “new” value for the state, x' , and then MH tells to accept this “new” value of the state with probability a , computed via:

$$\alpha = \min \left(1, \frac{p^*(x')}{p(x)} \frac{q(x|x')}{q(x'|x)} \right) \quad (5)$$

A MH transition kernel for a single variable x_1 can be chained with kernels for x_2 and x_3 , via compositionality. Figure 2b shows T_{MH} , an idealized Metropolis-Hastings transition circuit, in which Q_{sim} samples a new candidate state value, and the evaluation of the acceptance probability weights a theta gate, determining acceptance or rejection.

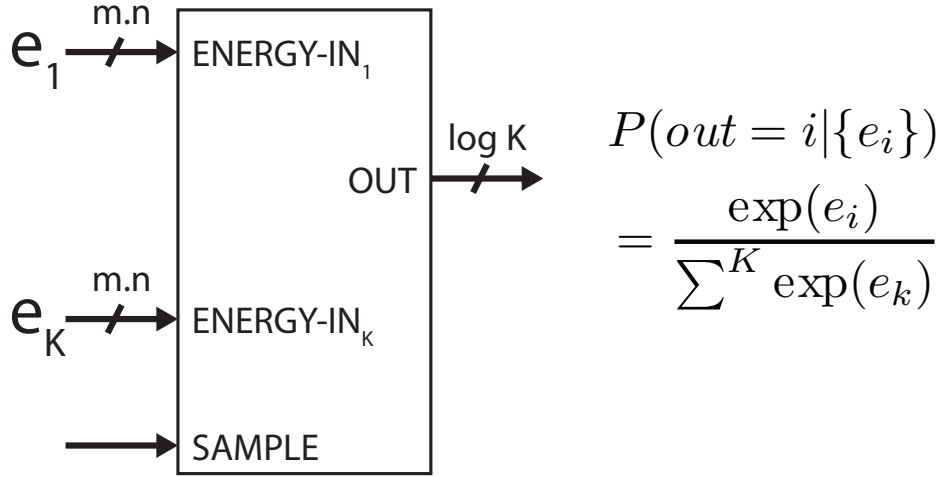


Figure 4: The normalizing multinomial gate: Accepts K unnormalized (energy) values representing $e_i = \log_2 p_i$ and samples from the normalized distribution

Sampling at very low bit precision

If X and Y are independent random variables, then $P(x, y) = p(x)p(y)$. We've seen above how independence is common feature of many probabilistic models, and how it enables us to exploit parallelism at varying granularities.

Probabilistic systems often compute the probability of many independent events, resulting in the multiplication of a large number of values $p \in [0, 1]$. To avoid overflow, and facilitate computation, it is useful to express all these calculations logarithmically – here we use a $\log_2$ encoding for most of our values. This lets us replace expensive high-dynamic-range multiplications with more-efficient additions with reduced dynamic range demands. There's a cost, however – addition becomes more expensive, as we must first convert the log-space representation back into real values before performing the addition. Similarly, if we want to sample from a (normalized) list of log values, we must exponentiate and accumulate.

The first circuit element exploiting this encoding is the normalizing multinomial gate, which takes an unnormalized stream of $K \log_2$ encoded energies, normalizes them to a probability distribution, and draws a sample from this probability distribution. Thus we can draw exact samples from any unnormalized vector of energies. This frequently arises when attempting to Gibbs sample over a discrete variable, where we can evaluate $\log_2 p^*(x|\cdot)$ for some $x \in \{1 \dots K\}$. We can use the normalizing multinomial gate to then sample from $p(x|\cdot)$.

```
for i in 0..{K-1}:
    score[i] = p*(x=i)

S = \sum_0^{K-1} exp(score[i])

for i in 0..{K-1}:
    probs[i] = score[i]/S

return MultinomialSample(probs)
```

To then gibbs sample, our circuit must take, as input, the values of the neighbor states that it is

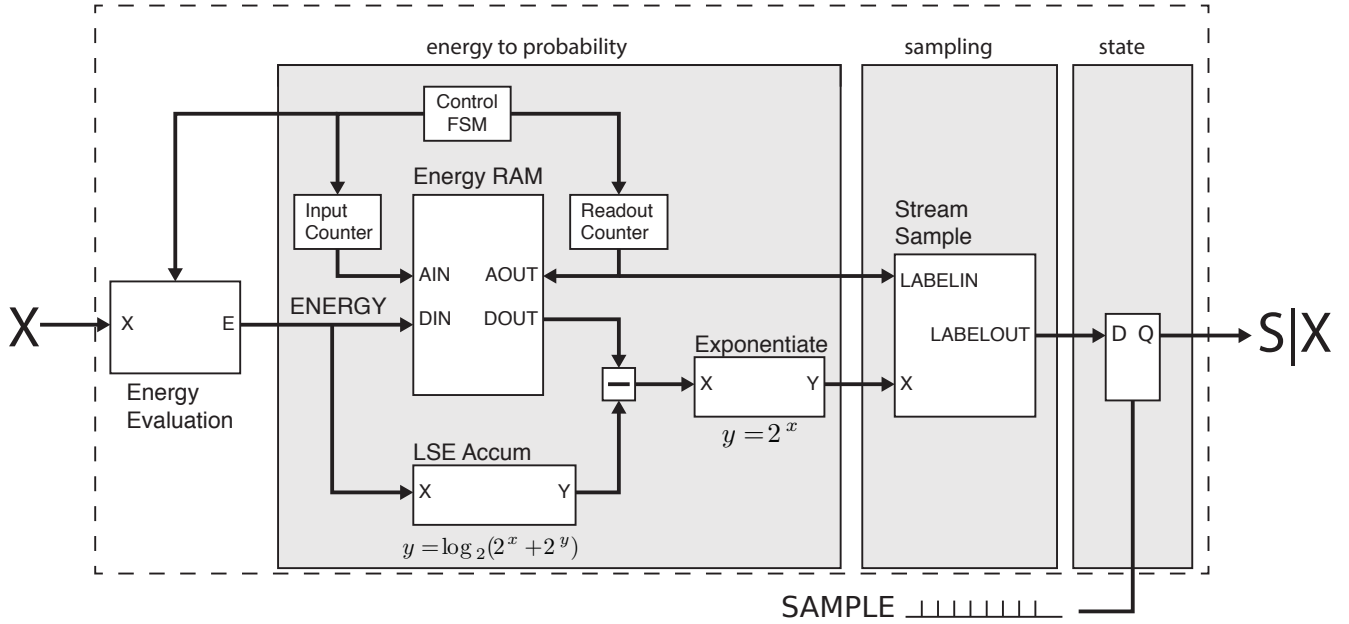


Figure 5: Implementation of the normalizing multinomial gate

conditioning on $-$ to sample from $p(x_1|x_2, x_3)$ we must take the values of x_2 and x_3 as inputs.

Normalization takes place with very finite-precision arithmetic using a variety of mathematical approximations that, at first glance, seem rather crude. Scores are saved internally in a small RAM, while simultaneously being accumulated via a log-sum-exp accumulator. Once all values are seen, the unit reads out the saved values, subtracts off the normalizing sum, exponentiates the log value, and feeds the result into the stream sampler to produce the eventual output. The accuracy of the results is discussed later.

3.1 Functional approximations within

To approximate the addition of two numbers (the log of the sum of the exponentiation of the two values, or “log sum exp”), we use the familiar (exact) trick where $Z = \max(x, y)$ and $W = \min(x, y)$, and then we return $Z + \log_2(1.0 + 2^{W-Z})$. This both allows increased dynamic range and lets us work with a smaller lookup table for $f(\Delta) = \log_2(1.0 + 2^\Delta)$. The approximation unit compares the two inputs, computes the delta, and then returns the larger Z plus the lookup-table-generated correction.

The resulting approximation is extremely accurate, as show by the plots of values and errors in figure 6.

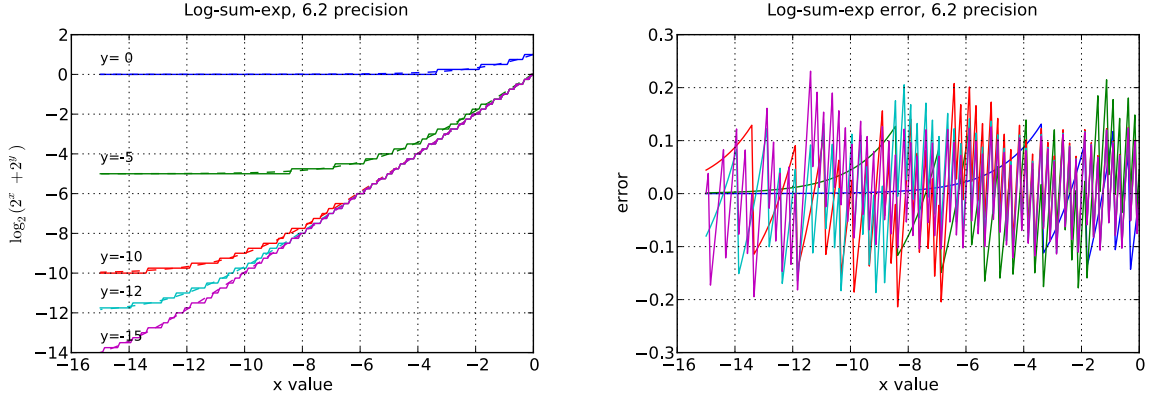
3.2 Exponentiation

We must exponentiate the resulting, normalized scores to sample from them. The similarity-across-scales of exp makes it very easy to use a limited-size lookup table.

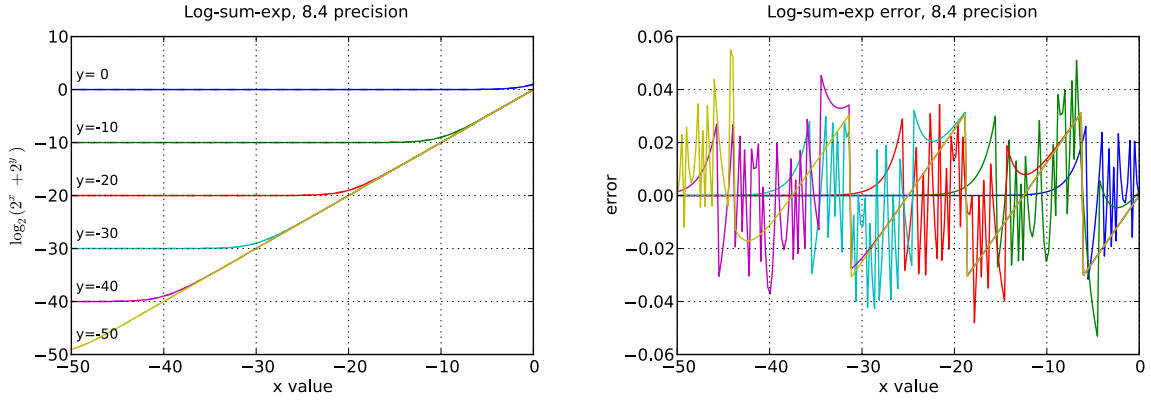
3.3 Random Starts to remove bias

Numerical errors for large energy vectors can accumulate, resulting in an underestimation of the total probability mass of the distribution; that is, $\sum p_i < 1$. As a result, we sometimes frequently end up with too much probability mass assigned to the final state possibility k .

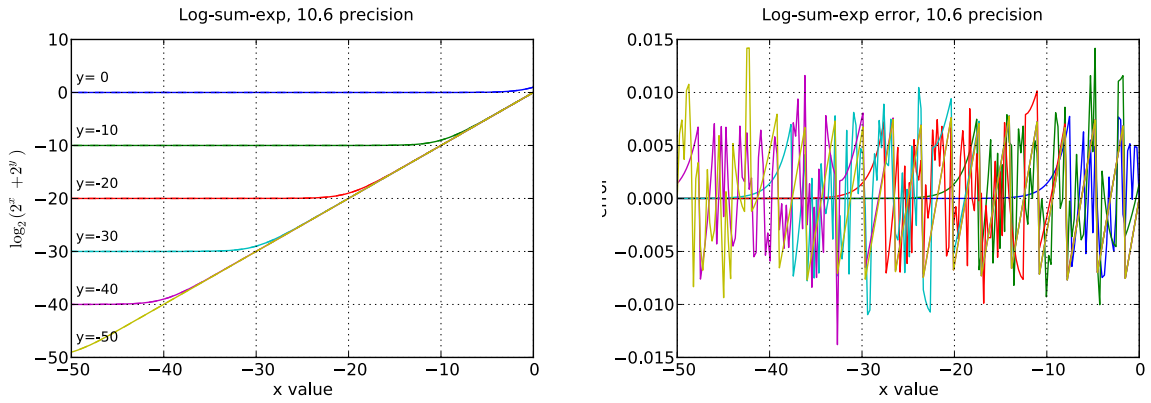
To remove this systematic bias we circularly permute the probability vector before computing the CDF and sampling. A circular permutation can be easily implemented by randomly sampling the starting



(a) $m=6, n=2$



(b) $m=8, n=4$



(c) $m=10, n=6$

Figure 6: “Log Sum Exp” ($\log_2(2^x + 2^y)$) approximation. Each curve is a parametric varying of x for a fixed value of y , and we plot the results of the approximation unit (solid line) and the true (floating-point-estimated, dashed line) value. The lines overlap so well that we also plot their differences (the error) to the right. Colors are consistent across figures. Each row is a different bit precision.

position and taking care to wrap around at the end of the array.

3.4 Resources

Figure 7 shows how look-up table and flip flop utilization vary as a function of both internal bit precision and the maximum arity for the multinomial sampler. Going from the smallest 6-bit, $k = 16$ unit to the largest 12-bit $k = 1024$ unit increases the combinational logic requirements by four times and doubles the amount of stateful silicon logic.

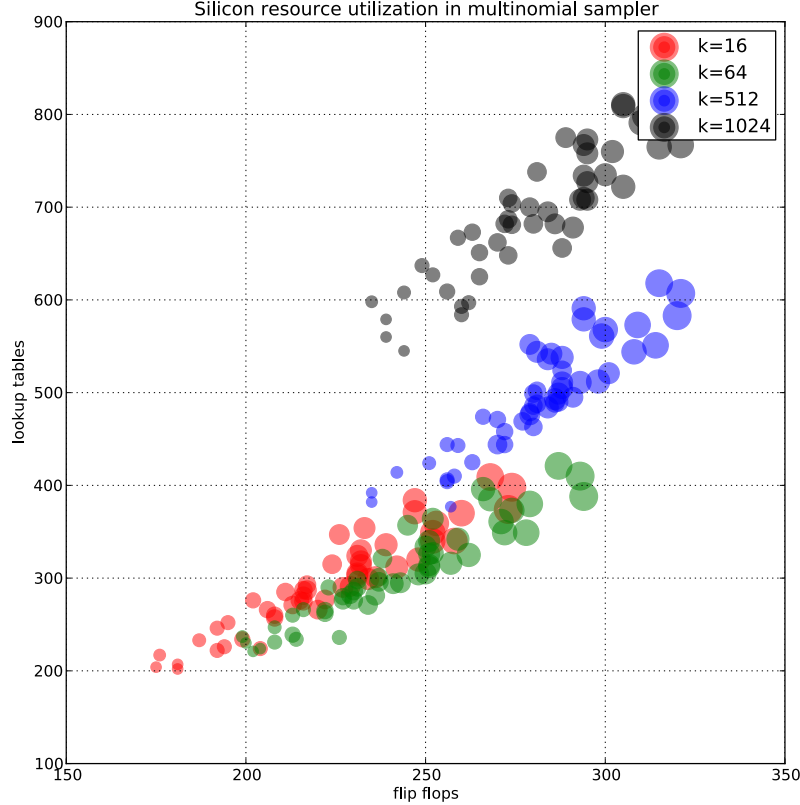


Figure 7: Resource utilization for the Multinomial Sampler, for samplers configured with several different values of K , and bit precisions varying as above. Larger circles indicate higher bit precisions, ranging from six bits to twelve.

4 Entropy Sources

There are many possible sources of entropy for the stochastic logic gates. High-quality entropy (suitable for cryptographic application) could be generated internally in silicon implementations, the result of amplification of atomic-level phenomena, including Johnson noise and shot noise. It could be provided externally, from other sources of natural entropy, such as radioactive decay.

But for all of the applications and elements we’ve identified, cryptographic randomness is overkill – we only need pseudorandomness. Assessing the quality of randomness from a pseudorandom source is a notoriously challenging problem (see Marsaglia and Tsang (2002)). In general, we care about both the marginal distribution of the samples from a PRNG – for $x_1, x_2, \dots, x_n$, how close is $P(x_i)$

N	$p = 0.5$	$p = 0.50002$
$N = 10$	5	5
$N = 100$	50	50
$N = 1000$	500	500
$N = 10000$	5000	5000
$N = 50000$	25000	25001

Table 1: Two coins with probability of heads specified to 16 bits, differing only in the LSB. It takes an average of 50000 flips to even detect the differences in weightings.

to $Uniform(0, 1)$ – and any possible long-running correlations between x_t and x_{t+k} . Of course, any PRNG with a finite latent state space will ultimately exhibit periodicity – the output will “wrap around”, resulting in $x_t = x_{t+T}$ for a PRNG with period T .

The classic Mersenne Twister (Matsumoto and Nishimura 1998) pseudorandom generator has a phenomenally long period ($2^{19937} - 1$) but pays for it by carrying a massive “state” of nearly $20kB$. This period is overkill for many applications, including those we care about.

George Marsaglia introduced the “Xorshift” family of random number generators, which use substantially fewer state bits (Marsaglia 2003) but still pass all of the known empirical tests for PRNGs. Here we implement the 128-bit XORShift, which has a total of 128 state bits and a period of $2^{128} - 1$. 1 shows the pseudocode for the 128-bit RNG; the entropy is output via the state variable w .

Algorithm 1 XORShift RNG 128

```

 $x \leftarrow \text{SEED}[0]$ 
 $y \leftarrow \text{SEED}[1]$ 
 $z \leftarrow \text{SEED}[2]$ 
 $w \leftarrow \text{SEED}[3]$ 
 $tmp \leftarrow x \oplus \text{LEFT-SHIFT}(x, 15)$ 
 $x \leftarrow y$ 
 $y \leftarrow z$ 
 $z \leftarrow w$ 
 $w \leftarrow (w \oplus \text{RIGHT-SHIFT}(w, 21)) \oplus (tmp \& \text{RIGHT-SHIFT}(tmp, 4))$ 
return  $w$ 

```

The underlying implementation is exceptionally tiny, consuming a mere 160 slice flip flops and 33 lookup tables on our target Virtex-6 FPGA. We have validated the output of our PRNG exactly matches the equivalent software implementation. It can operate at up to 200 MHz, delivering 6.4 Gbps of randomness. Note even at that phenomenal rate, it will take 5×10^{22} years to wrap around. The PRNG is free-running, and multiple independent sources of entropy can be created by either time-division multiplexing the output of one single PRNG or instantiating multiple PRNGs with different seeds.

5 The effects of bit precision

Small differences in the distributions underlying samplers are difficult to resolve without a very large collection of samples. This can be seen by considering two weighted coins whose probability of heads agrees to the 16th least-significant bit (Table 1). It takes roughly 2^{16} samples from these distributions to detect any difference in the encoded distribution.

Of course, digital signal processing engineers have been taking questions of bit precision seriously for decades. The available dynamic range is often limited by the underlying sensor technology – modern

high-end scientific cameras top out at 12 bits of intensity per pixel. Professional studio-quality audio systems exceed the dynamic range of the human ear at a mere 24 bits.

This stands in contrast to the historic focus on linear-algebra-based methods in data analytics, scientific computation, and machine learning, which have lead to a strong demand for more and more IEEE-754 floating point units from hardware vendors. Purveyors of scientific computing such as nVidia are only taken seriously once their hardware supports 64-bit floating point.

But when sampling from distributions, we can be incredibly insensitive to low bit precision. We can measure this property more precisely by using the multinomial sampling unit. We use the multinomial sampling unit because it has the most internal arithmetic computation, and thus should be *most* sensitive to bit precision errors. It also forms a core part of subsequent circuits.

We create three samplers representing three weighted dice, with $k = 10, 100$, and 1000 sides, and parametrically vary the entropy of their underlying discrete distribution from 0 to $\log_2 K$ bits.

Figures 8, 9, and 10 shows the results as we vary the number of bits in a representation, using $m.n$ encoding. We encode the true distribution in the circuit, and then compute an empirical distribution from a bag of 100000 samples generated by the synthesized circuit. In all cases, the KL divergence from the encoded distribution to the empirical distribution is remarkably low for all encodings with $m \geq 6$ bits.

As the representation becomes bit-starved, we see that the KL still stays low in two regimes : very high and very low entropy distributions. This makes intuitive sense – for maximally-entropic source distributions (that is, uniform), encoding the array of identical values is easy. Similarly, for minimal-entropy distributions with all mass concentrated on a single value, encoding the distribution is easy.

Figures 8c, 9c, and 10c show QQ plots of true versus recovered-from-hardware distributions, for distributions with varying entropies (listed at left). Again, the distributions look almost perfect, except for medium-to-high-entropy distributions in very low bit-precision regimes.

6 Resource Utilization

But performing large floating-point operations consume a massive quantity of silicon resources when compared to our stochastic sampling elements. As figure 11 shows, the area consumption by sampling elements varies with their flexibility, the arity of their output, and the dynamism and precision of their internal representations. We synthesized a 64-bit IEEE-754 FPU (Lundgren 2009) with the Xilinx toolchain. Note that this is a very conservative estimate for the number of silicon resources, as the Xilinx synthesis tools used some of the embedded multiplier blocks, whereas all the comparison units were tested entirely with slices and flipflips (no BRAMs or DSP48s were allowed).

Also note that while the XORShift RNG takes up more silicon area than some of the other sampling elements, a single PRNG instance can supply entropy to dozens of stochastic circuit elements.

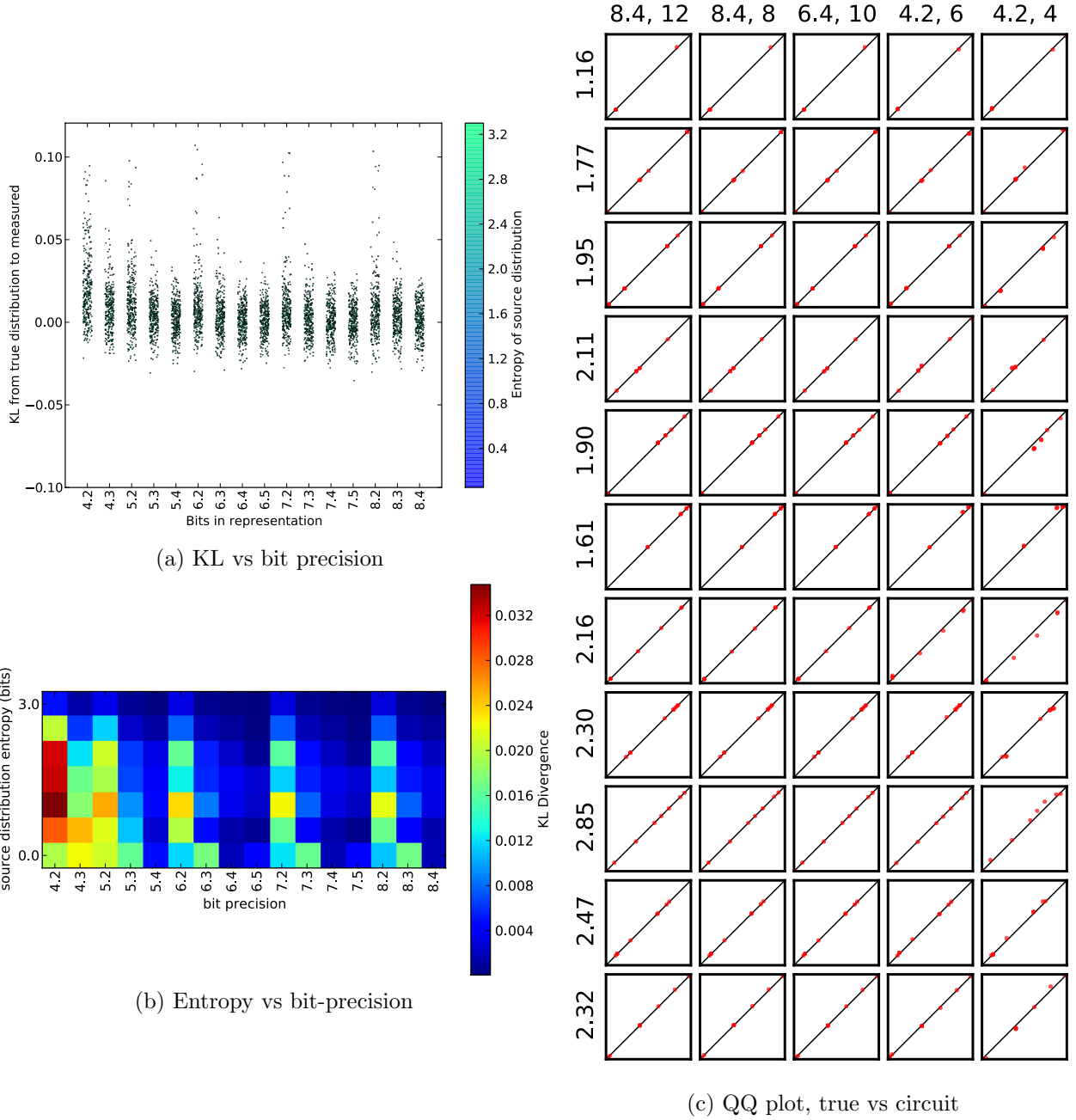


Figure 8: The effects of bit precision on KL divergence for a $K = 10$ multinomial sampling gate, a.) KL vs bit precision, b.) heatmap showing regions of entropy/bit-precision with high KL, and c.) example distribution QQ plots. Each column is a different bit precision (labeled at top) and each row is for a different input entropy. The QQ plot itself compares the true CDF (x-axis) with the empirical (y-axis). Perfect agreement results in all points lying on the $y = x$ line.

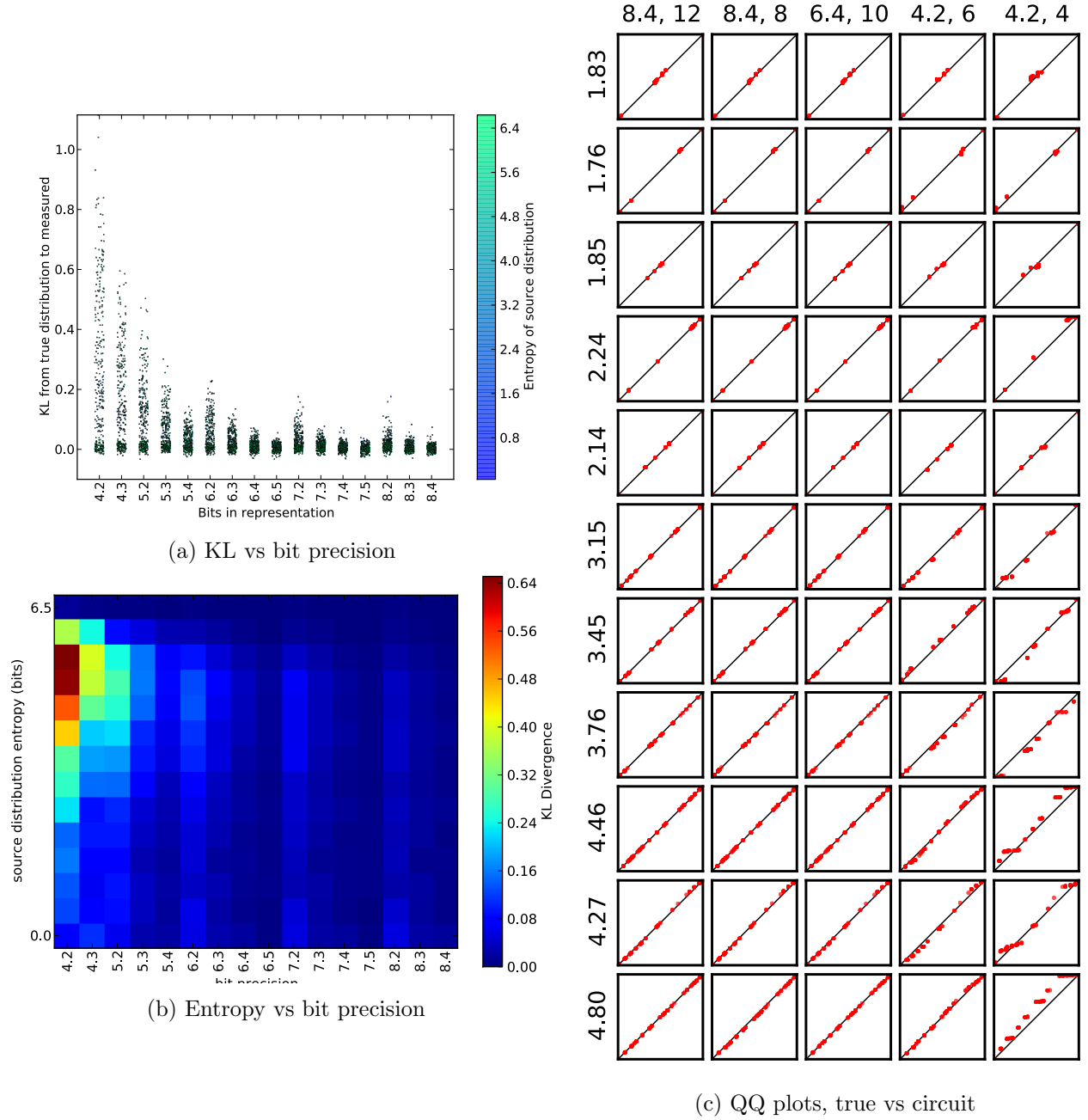


Figure 9: The effects of bit precision on KL divergence for a $K = 100$ multinomial sampling gate, a.) KL vs bit precision, b.) heatmap showing regions of entropy/bit-precision with high KL, and c.) example distribution QQ plots. Each column is a different bit precision (labeled at top) and each row is for a different input entropy. The QQ plot itself compares the true CDF (x-axis) with the empirical (y-axis). Perfect agreement results in all points lying on the $y = x$ line.

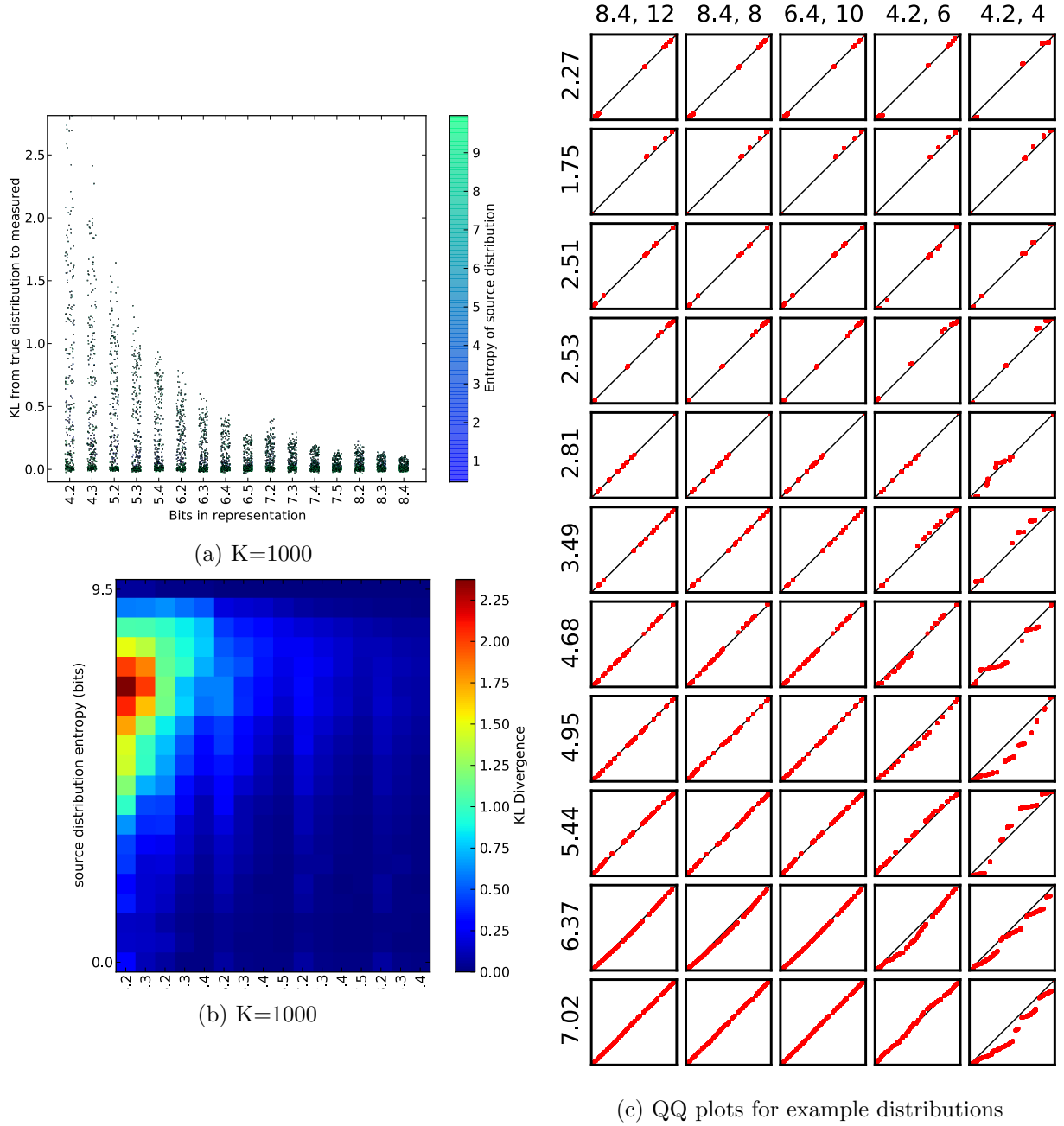


Figure 10: The effects of bit precision on KL divergence for a $K = 1000$ multinomial sampling gate, a.) KL vs bit precision, b.) heatmap showing regions of entropy/bit-precision with high KL, and c.) example distribution QQ plots. Each column is a different bit precision (labeled at top) and each row is for a different input entropy. The QQ plot itself compares the true CDF (x-axis) with the empirical (y-axis). Perfect agreement results in all points lying on the $y = x$ line.

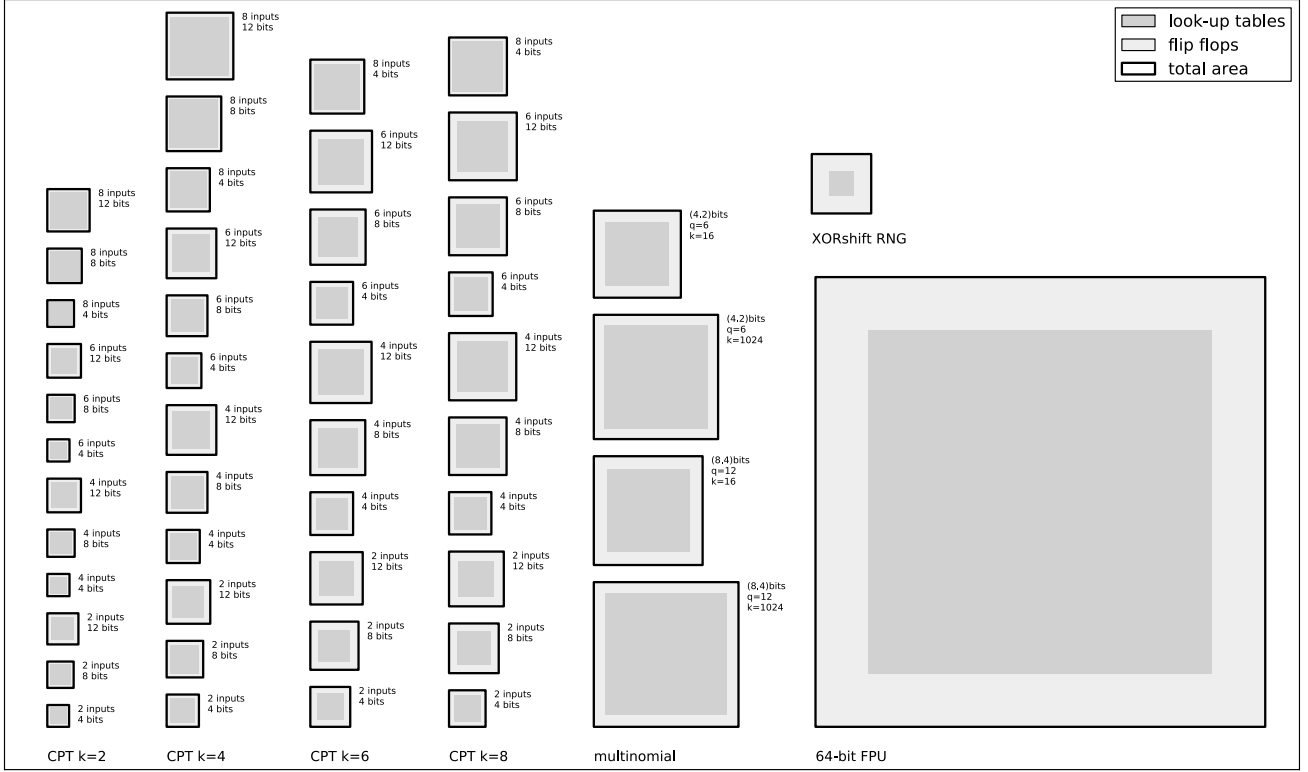


Figure 11: Comparing approximate silicon area between a 64-bit IEEE-754 floating point unit and various examples of our stochastic gates, including the Conditional Probability Table gate and the normalizing multinomial gate. Dark grey areas are the purely-combinational lookup tables, lighter areas are the stateful flipflops. See text for an explanation of bit precision designations.

Implementation details for vision applications

This section gives implementation details for our vision applications, beginning with the formulation of factor graph models for low-level vision problems, and moving through the performance details of a *stochastic video processor* based on stochastic digital circuits. This video processor is reprogrammable in software to emulate arbitrary discrete square-lattice factor graphs; frames can be streamed in and latent samples streamed out in real time.

7 Low-level Vision Factor Graph

A low-level vision factor graph (figure 12) is a probabilistic model for image processing problems where per-pixel data $Y_{i,j}$ is used to estimate some unknown (latent) per-pixel random variable $X_{i,j}$. The latent state variables are arranged in a square lattice. The factor $f_E(x_{i,j}, y_{i,j})$ dictates the probabilistic relationship between the observed variables and the latent ones. Additionally, a “latent, pairwise” factor, $f_{LP}(x, x')$ constrains the relationship between a latent state variable and its lattice neighbors. Typically, this factor serves as a smoothness prior, favoring configurations where adjacent latent variables have

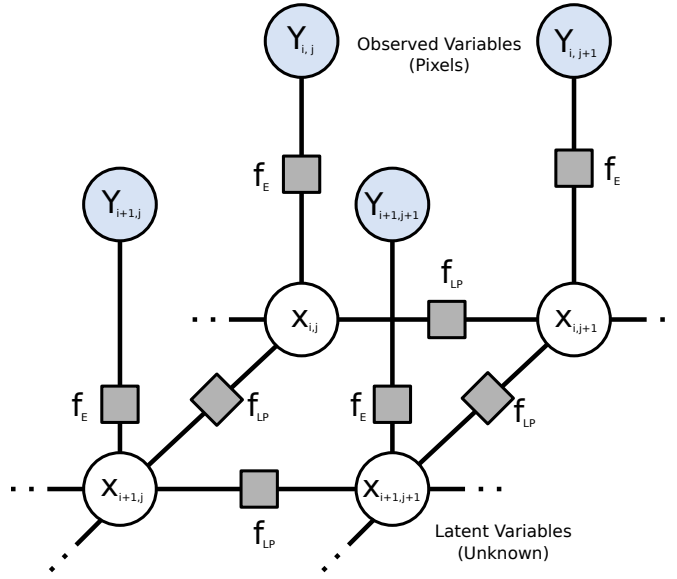


Figure 12: Low-level vision lattice factor graph. Some lattice of unobserved, *latent* state variables $X_{i,j}$ generate observed values $Y_{i,j}$ through an *external* potential f_E . The relationship between adjacent $X_{i,j}$ and $X_{i',j'}$ is constrained by the *latent, pairwise* potential f_{LP} . Note that in this formulation, both f_{LP} and f_E are homogeneous in the graph.

similar values. Thus

$$P(X_{i,j} = x) \propto f_E(x, y_{i,j}) \sum_{x' \in \text{Neighbor}(X_{i,j})} f_{LP}(x, x') \quad (6)$$

Here we are only considering the case of *homogeneous* densities – that is, the functional form for all f_E are the same and all f_{LP} are the same.

8 Resource virtualization and parallelization

8.1 Virtualization

The homogeneity and regular lattice structure of the factor graphs here suggests an opportunity for resource sharing. Only the state values $X_{i,j}$ and $Y_{i,j}$ differ between adjacent pixels.

To explore the opportunity for virtualization, consider a simpler factor graph. Figure 13 shows a simple discrete-state factor graph with a chain-topology. In this factor graph, the state variables all have the same domain, and the pairwise potentials are all identical.

Rather than creating a dedicated stochastic circuit to perform sampling at every site, we can create a *virtualized* stochastic circuit. This virtualized circuit can produce a sample for $X_i \sim X_i | X_{i-1}, X_{i+1}$. Thus we can load the X_{i-1} and X_{i+1} values from someplace else, sample a new value for X_i .

Thus, all of the relevant structure of this factor graph has been captured in the resulting virtualized circuit, and the relevant state variables can simply be streamed in serially, in effect “sliding” the virtualized circuit down the graph. A similar scheme can be used in a square-lattice factor graph (such as those we’re working with here), or indeed any factor graph with highly regularized structure.¹

¹This regularity is not uncommon in models which use a great deal of data, such as image processing and time series.

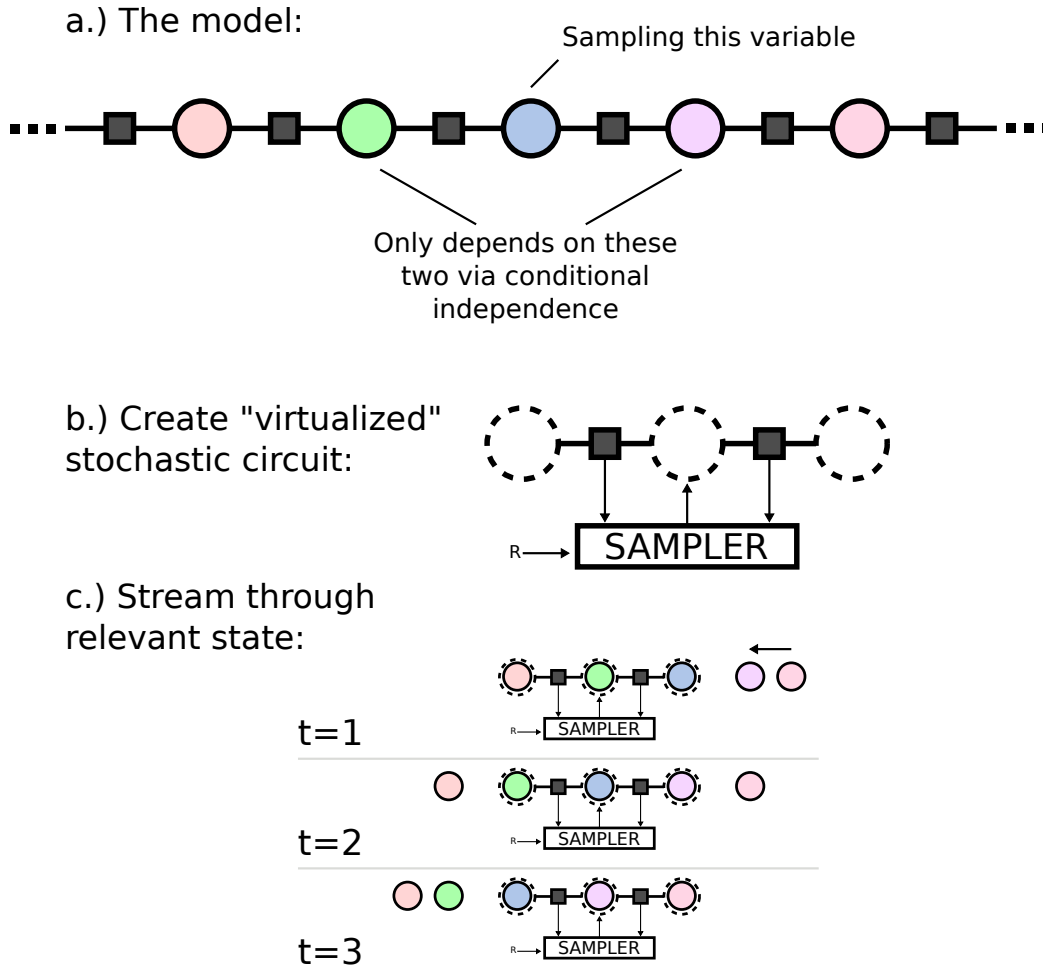


Figure 13: Virtualization example for linear-chain factor graph. a.) The model consists of a repeating chain of variables connected pairwise by homogenous factors. Sampling a new value for a particular variable requires conditioning on the neighbor state values and evaluating the connected factors. b.) We can create a virtualized stochastic circuit which contains the factors, and allows state values to be read in and out, sampling a new value for the center variable. c.) The virtualized circuit then can sample values for the entire factor graph by streaming in the variable values, performing the sample, and writing the output.

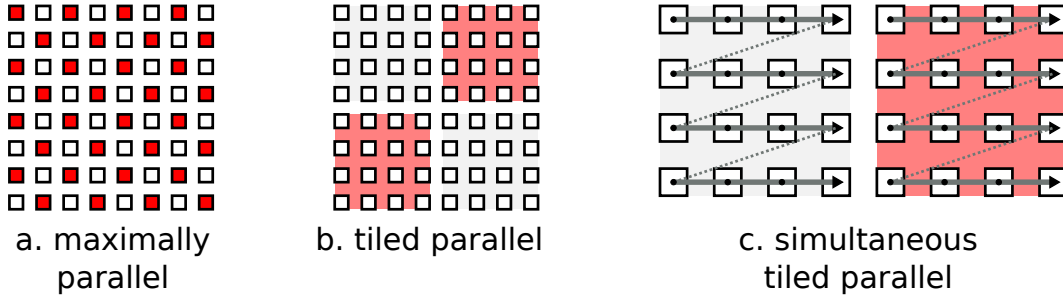


Figure 14: Parallelization via conditional independence. a. Maximally parallel, derived from the graph coloring. All red sites can be sampled simultaneously b. The naive coarsening, tiled parallel, in which inference in the same color of tile can be performed simultaneously c. Simultaneous tiled parallel, where we carefully time the sequential sampling within a tile to guarantee correctness, allowing sampling on tiles to happen simultaneously.

8.2 Parallelization

We have previously shown that factor graphs with conditional independencies provide extensive opportunities for parallelization. The granularity of that parallelization can be varied. In 14a, the variables in a square lattice factor graph are shown, colored for parallelism – sites with the same color can be sampled simultaneously. It’s entirely possible to “coarsen” this parallelism, as seen in figure 14b, resulting in “tiles” of sequential serial inference, where inference can take place simultaneously in similarly-colored tiles.

We go one step further here, as shown in figure 14c. By carefully controlling the sequential scan of particular random variables *within* a tile, we can be sure that no two adjacent sites between tiles are the target of inference simultaneously, a condition which would result in invalid inference. This allows serial inference to occur for all tiles simultaneously.

9 Circuit Architecture

Here we describe a tiled architecture for efficient inference in low-level vision factor graphs which exploits parallelism and virtualization to make larger models practical. The resulting “Lattice Factor Graph Engine” consists of an array of lattice-interconnected *Gibbs tiles*, each of which performs Gibbs sampling on a subtile of the total lattice factor graph.

Reencoding the factor graph (figure 15), replaces homogeneous external field potentials and observed data states with heterogeneous external field potentials that incorporate the per-pixel data. This enables the computation of the arbitrary external field relationships off-line.

9.1 Gibbs Tile

The Gibbs Tile virtualizes a single normalizing multinomial gate stochastic element over a rectangular subregion of the factor graph (figure 16, performing sequential Gibbs sampling on this region of the graph. The Gibbs tile stores the requisite state for the variables in this portion of the graph in the Pixel State controller, which also handles scheduling and coordinates communication with adjacent tiles. Each Gibbs Tile can be synthesized with a particular latent pairwise density, including the above-mentioned lookup table density. The external field density is implemented as a runtime-reprogrammable SRAM, An XORShift pseudo-random number generator (section 4) provides the needed entropy.

This bears repeating. The lookup-table density allows for run-time reconfiguration, and thus we could potentially build an ASIC capable of performing inference in an arbitrary lattice-structured factor graph.

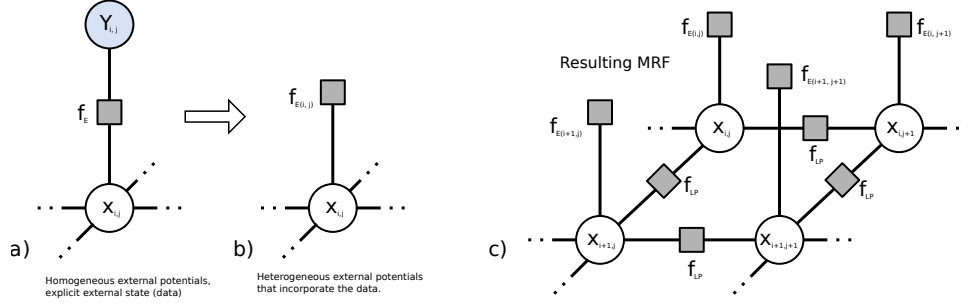


Figure 15: Re-encoding of the factor graph from using homogeneous external potentials and heterogeneous data (a.) to just using heterogeneous external potentials (b.), without loss of generality. c.) shows the resulting factor graph that is actually used.

While the external field density (also encoded as an LUT) can be easily updated with single-frame latency, configuring the LUT pairwise density can take several frame cycles.

To perform Gibbs sampling on its region of the factor graph, the gibbs tile sequentially iterates through sites, looking up the relevant adjacent state bits and then having the Gibbs Core Sampler produce a sample from the appropriately-conditioned distribution.

Algorithm 2 Gibbs Tile Operation

```

for all  $v$  in VirtualizedSet do
   $n_i \leftarrow \text{LOOKUPNEIGHBORS}(v)$ 
   $\text{offset} \leftarrow \text{COMPUTEOFFSET}(v)$ 
   $\text{newv} \leftarrow \text{GIBBSCORESAMPLE}(n_i, \text{offset})$ 
   $v \leftarrow \text{newv}$ 
end for

```

9.2 Pixel State Controller

The pixel state controller (PSC in figure 17) coordinates sampling over a virtualized region of per-pixel latent state, storing the latent state in RAM internal to the PSC. The PSC drives the Gibbs core, and stores the resulting sampled value.

Most importantly, the PSC coordinates the state virtualization, sequentially scanning through “active” states one pixel at a time, looking up the neighboring latent states, and then presenting them in a unified way to the gibbs unit.

The lattice structure of the factor graph results in adjacent tiles needing to see the “edge” latent pixels of their neighboring tiles. The PSC contains dual-ported edge RAMS that store buffered copies of this particular tile’s edge state for interruption-free lookup by neighboring tiles (labeled “Adjacent state IO” in figure 16).

For off-device IO, the internal state variables in the PSC are readable and write-able through an external port.

9.3 External Field Density RAM

We encode the per-pixel external field density as a lookup table in a dense SRAM. The PSC selects the relevant region of this RAM that corresponds to the lookup table for the particular active site.

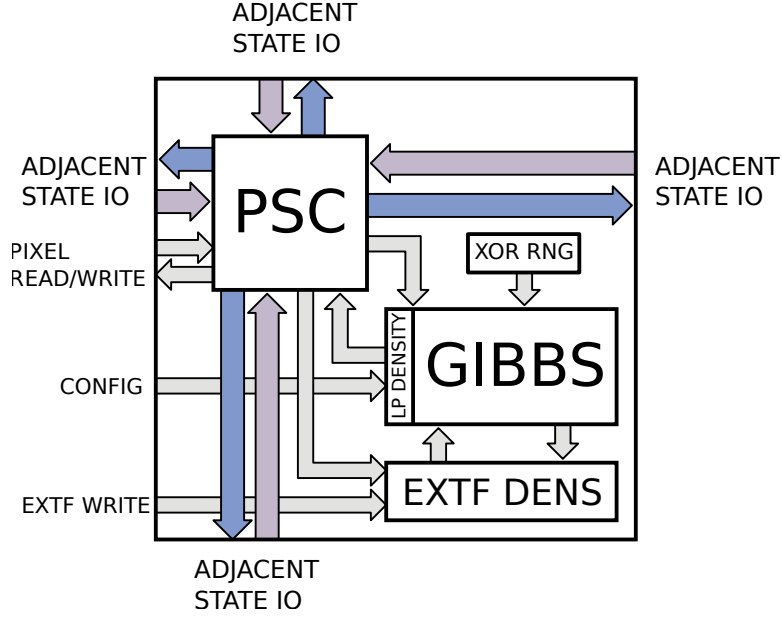


Figure 16: Gibbs tile, consisting of the pixel state controller, the Gibbs unit with local potential density, the external field density lookup table, and the PRNG. The tile communicates with neighboring tiles via adjacent state IO.

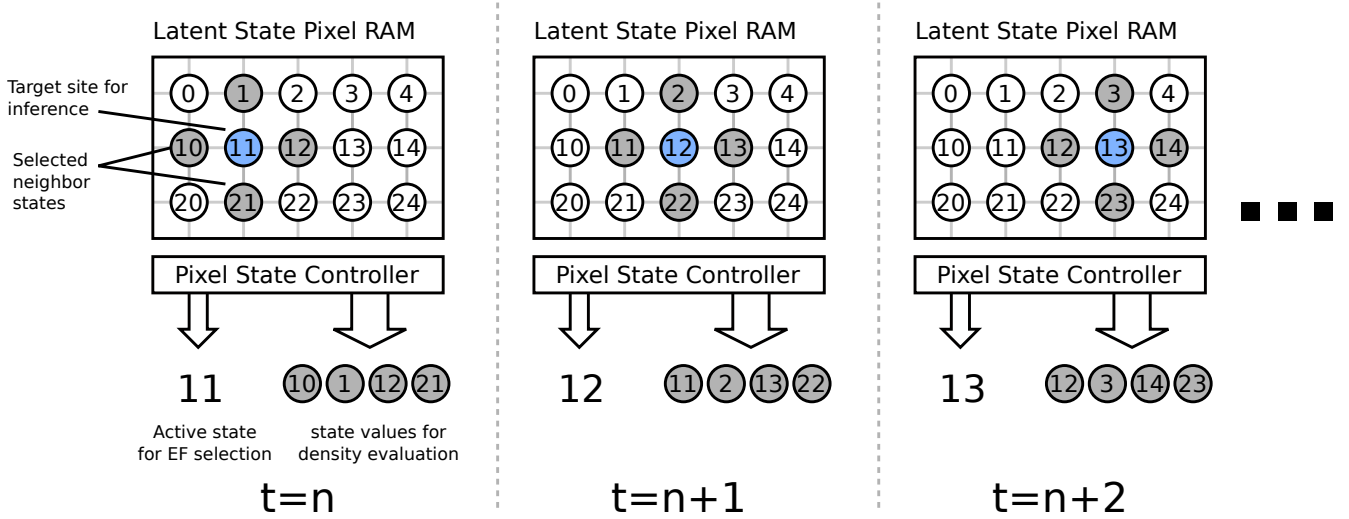


Figure 17: Pixel state controller behavior. At time $t = n$, the PSC centered on node 11 presents the neighborhood state values to the downstream core. At subsequent times, the active neighborhood shifts to the right.

Algorithm 3 Gibbs Core Sampling algorithm

```

for  $x = 0$  to  $K$  do
  extfscore  $\leftarrow$  extfram(offset +  $x$ )
  lp_score  $\leftarrow$  density(neighborvals,  $x$ )
  totalscore  $\leftarrow$  extfscore + lp_score
  multinomial-sampler.add(totalscore)
end for
return multinomial-sampler.sample()

```

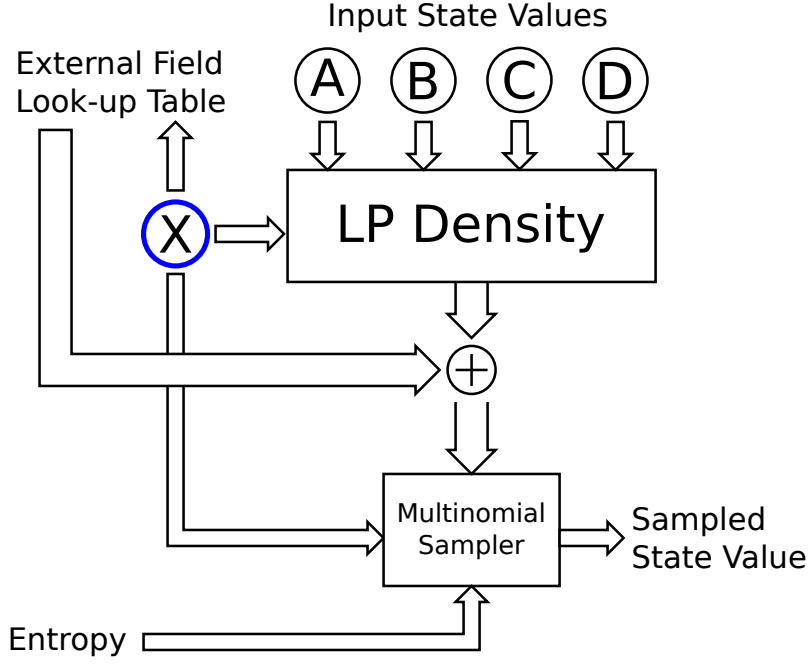


Figure 18: Gibbs core enumerates through possible values for this site’s latent state variable, setting X to each value and evaluating the LP density. The score from the LP density and the external field lookup table are summed. The multinomial sampler takes these unnormalized scores and produces a sampled state value.

9.4 Gibbs Core

Once the PSC has selected an active site $x_{i,j}$ and looked up the relevant neighboring values, the Gibbs Core Sampler computes the full conditional score for all values of $x_{i,j}$ (algorithm 3). For each of those values x^k we compute $S_{i,j}(x^k) = f_E(x^k) + \sum_{x' \in \text{neighbors}} f_{LP}(x^k, x')$. The Multinomial sampler normalizes and samples from the resulting score table. The Gibbs Core can also temper the resulting scores, allowing for annealing and tempering MCMC operations.

9.5 Latent Pairwise densities for specific models

The latent pairwise density is a pipelined, fixed-latency arithmetic primitive that performs $\sum_{x' \in N} f_{LP}(x^k, x')$. The LP density module has configuration registers which enable the setting of specific constants within the density. Each input has an optional enable which selectively includes that term in the resulting computation.

9.6 Configuration Parameters

The stochastic video processor is parametrized to allow the exploration of design tradeoffs and to generate application-specific engines targeted for certain problem domains.

Figure 19 shows the relevant parameters – we can vary the number of Gibbs tiles in either dimension, the number of sites within a tile, and various internal precision calculations within a tile.

9.6.1 Tile Efficiency

Gibbs sampling a site is $O(K)$, where each site can take on K possible values. For the stereo circuit described below, for example, there are $K = 32$ discrete depth values. Since we must evaluate each

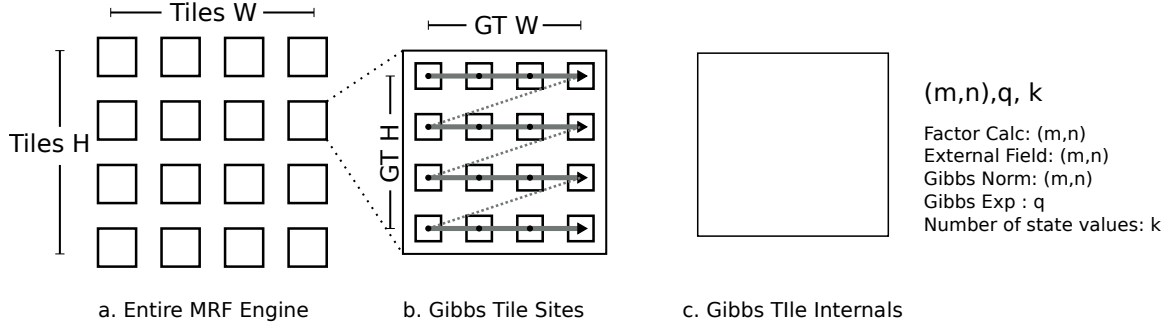


Figure 19: Stochastic Video Processor parameters. a.) Tiles H and Tiles W control the height and width of the engine, in tiles. b.) Each Gibbs Tile samples GT H by GT W sites. Within the tile, $(m,n),q$ -bit-precision computations are performed, with each variable taking on k possible values.

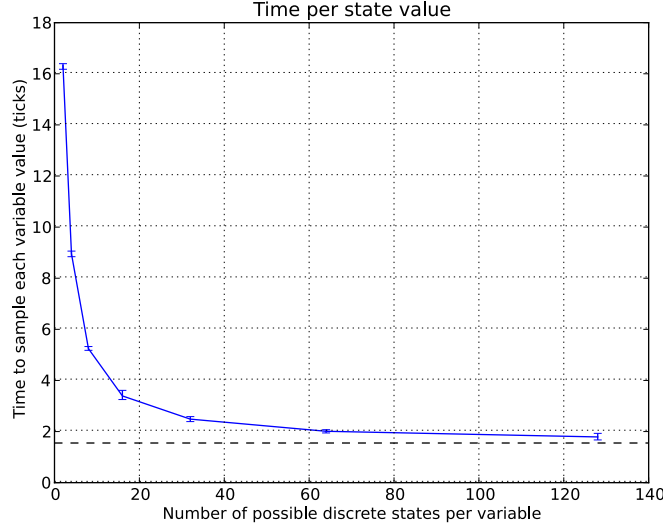


Figure 20: Performance overhead of architecture per state variable value. As the number of possible discrete states per variable increases, we approach the expected limit of 1.5 ticks per possible value (black dashed line).

possible state value before discretely sampling, we must take at least K ticks. The sampling step then needs $E[K]$ ticks to sample a value, suggesting a lower-bound on $K \cdot E[K]$, or $\approx 1.5K$ ticks per site.

Figure 20 shows empirically-measured tile performance for a 8×8 tile as the number of possible state values increases. Since there is constant startup and handshaking overhead, low-state-value variables tend to be more inefficient.

10 Comparison to explicit compilation

For reference, we compare the performance of our lattice markov random field architecture to the performance of circuits generated by our compiler. We focus on fully space-parallel Potts models, where we can generate an equivalent fixed-function lattice factor graph engine² for k -state Potts models (Wu 1982) and measure performance, in terms of samples per second and silicon resources used.

²Although the Lattice Factor Graph engine comes along with external field support.

Table 2: Resource utilization and performance for a 4-state 20x20 compiled Potts model.

Configuration				Performance	Resources
Vars	Bits	Scans/sec	Clock (MHz)	Slice FFs	BRAMS
400	5	1468535	125.0	76917	55710
					280

Table 3: Resource utilization and performance for a 4-state 128x128 Potts Lattice Factor Graph circuit.

Configuration (total vars=16384)			Performance		Resources		
Vars/Tile	Tiles	Bits	Scans/sec	MHz	Slice FFs	Slice LUTs	BRAMS
256	64	(6,2),8	13673	125.0	29559	35265	96
256	64	(8,4),12	13667	125.0	34231	44293	96
128	128	(6,2),8	27243	125.0	53936	64039	160
128	128	(8,4),12	27225	125.0	63280	82091	160
128	128	(6,2),8	27228	125.0	53952	64296	160
128	128	(8,4),12	27232	125.0	63296	82348	160
64	256	(6,2),8	54074	125.0	102457	120546	288
64	256	(8,4),12	54060	125.0	121145	156646	288

To synthesize the engine with the Potts latent pairwise potential, we create HDL representing $f_{LP}(x, x')$ and the module is replicated and synthesized.

The compiler can only fit a 400-node 4-state Potts model in our target Virtex-6 LX240 FPGA, but achieves 1.45 million full gibbs-scans per second with 5-bit precision (table 2).

We can synthesize a variety of Potts lattice factor graph engines for comparison, all resulting in a 16,384-node MRF (Table 3). We can vary the number of variables per tile – more state values per tile results in an engine that consumes fewer FPGA resources, but only scans at 13k scans/second. Or, we can use more FPGA resources, and a larger number of smaller tiles to sample at up to 55k scans per second. Note that while the compiler-generated factor graph is 25 times faster than the lattice engine, the lattice engine is solving a model 40 times larger.

11 Stochastic Video Processor

The virtualized circuit engine can already be loaded with data and the external field configurations at runtime. Here we extend the runtime reconfigurability to include *all* factors in the model, replacing the above fixed-function pairwise factors with a generic lookup table.

11.1 Resources and Speed

12 Depth estimation for Stereo Vision

The primate visual system uses stereopsis to estimate object depth, exploiting the image difference between the right and left eyes. Objects that are very close to the observer appear to be located at different horizontal positions on the eyes. Farther-away objects differ less in their separation (or *disparity*) between the eyes, with the far background identical for both eyes (figure 21).

The MRF model from Tappen and Freeman (2003) infers disparity, and thus distance from the camera, using a low-level vision markov random field. The latent variables $x_{i,j}$ are the disparity between the left

Table 4: Resource utilization and performance for a 32-state 128x128 Lookup-table MRF circuit.

Configuration			Performance		Resources		
Vars/Tile	Tiles	bits	Scans/sec	Clock (MHz)	Slice FFs	Slice LUTs	BRAMS
128	128	(4,2),6	14468	125.0	55874	352	69758
128	128	(6,2),8	14389	125.0	59714	352	78848
128	128	(8,2),10	14364	125.0	64066	352	91010
128	128	(6,4),8	12266	125.0	62402	352	88322
128	128	(8,4),12	12248	125.0	67266	352	100228

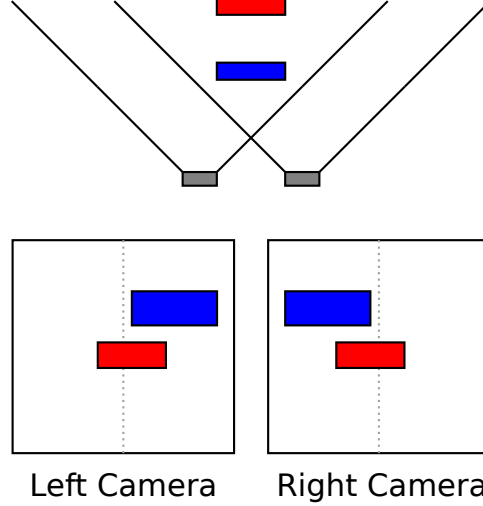


Figure 21: Stereo geometry calculations. There is substantial overlap between left and right camera views. Closer objects (blue) appear to shift more between adjacent frames than objects that are further away (red).

and the right image; the larger the value $x_{i,j}$, the greater the separation of the object between the left and right frames, and thus the closer the image is to the cameras.

Thus we must define two functions. The first is the latent pairwise factor, $f_{LP}(x, x')$ between two adjacent nodes. Tappen and Freeman use a Potts-model-style factor, where:

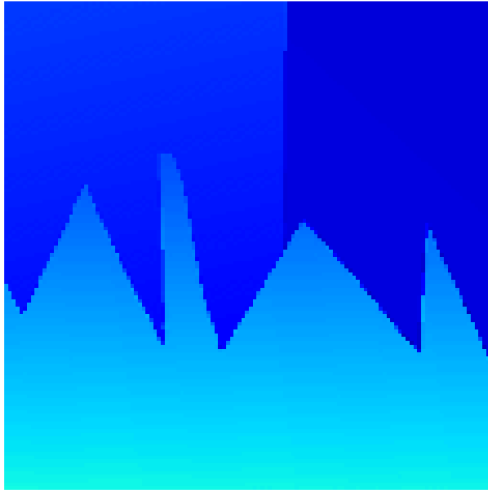
$$f_{LP}(x, x') = \begin{cases} 0 & \text{if } x = x' \\ \rho_I(\Delta I) & \text{otherwise} \end{cases} \quad (7)$$

where $\rho_I(\Delta I)$ is a function of $\Delta I = |x - x'|$,

$$\rho_I(\Delta I) = \begin{cases} P \times s & \text{if } \Delta I < T \\ s & \text{otherwise} \end{cases} \quad (8)$$

where T is a threshold, s is the penalty for violating the smoothness constraint and P extra-penalizes small smoothness violations. The intuition here is simple: objects are in general a fixed distance away from the camera, and thus the disparity is in general constant; abrupt jumps in disparity are to be expected, as there are different objects in the scene. Smoothly-varying disparities, however (as once might expect from a sphere) are uncommon and should be penalized accordingly.

The external field density $F_e(x_{i,j}, y_{i,j})$ computes the Birchfield-Tomasi dissimilarity (Birchfield and Tomasi 1998), a smoothed between-pixel distance measure that is robust against aliasing.



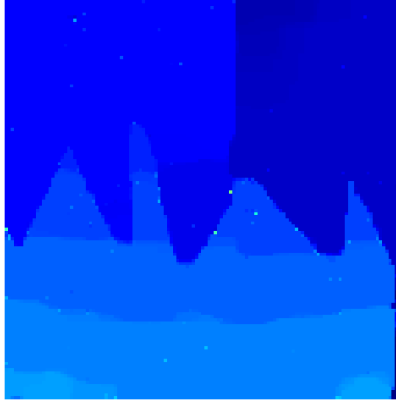
(a) Ground Truth



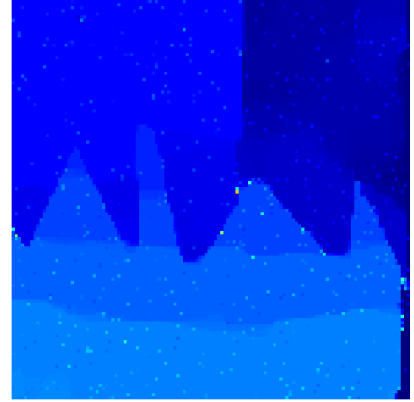
(b) Left Image



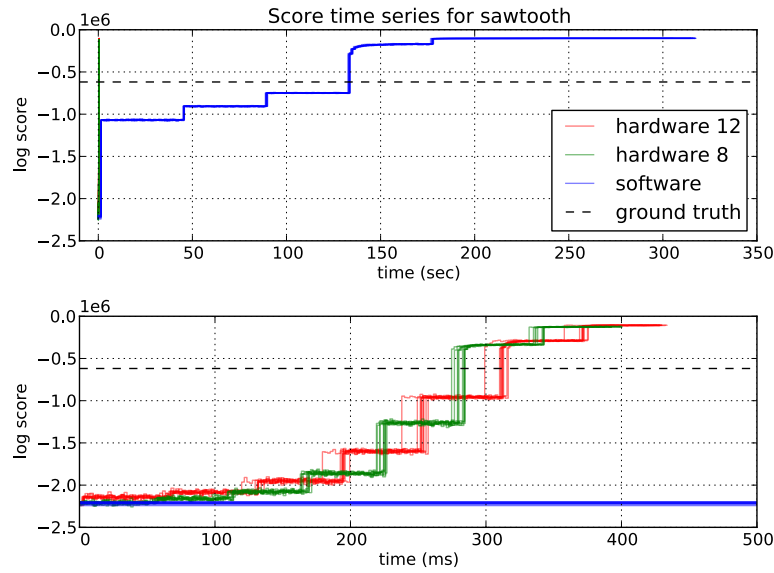
(c) software, 64-bit floating point



(d) hardware, 12 bits

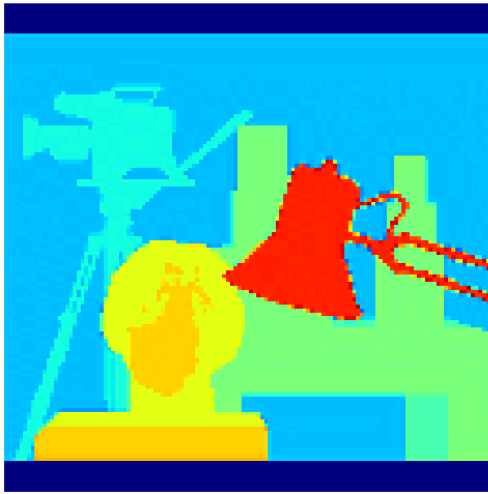


(e) hardware, 8 bits

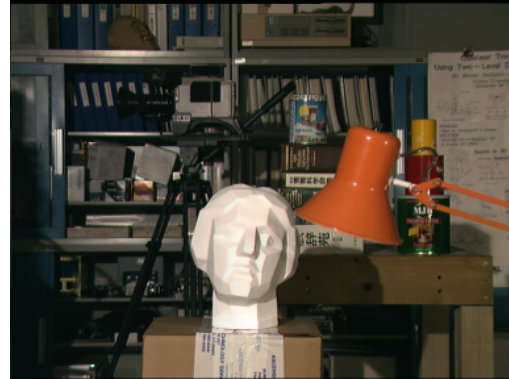


(f) log score vs time, hw vs sw

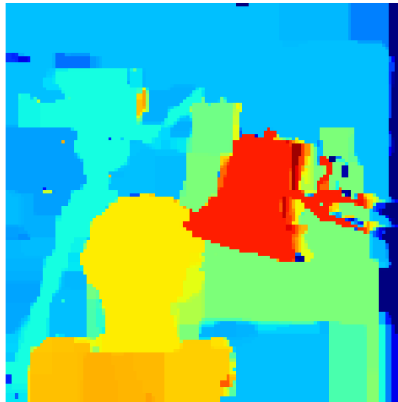
Figure 22: Middlebury “sawtooth” example stereo depth dataset. Top row is the ground truth disparity map, and subsequent rows are the empirical mean mean of 10 full annealing sweeps for the double-precision software, and 12-bit and 8-bit hardware, circuits.



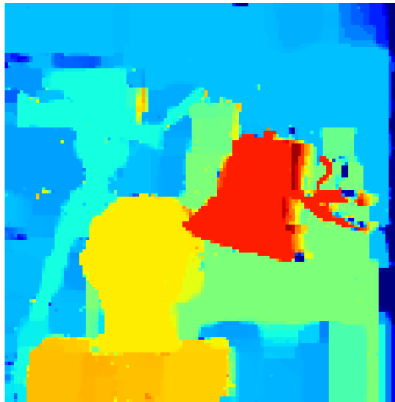
(a) Ground Truth



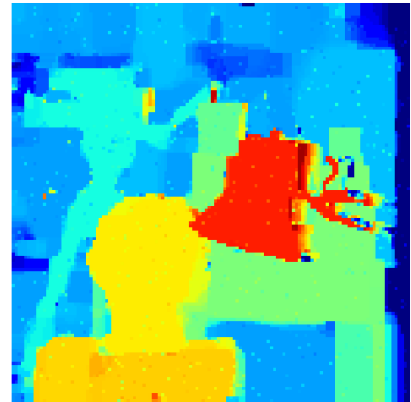
(b) Left Image



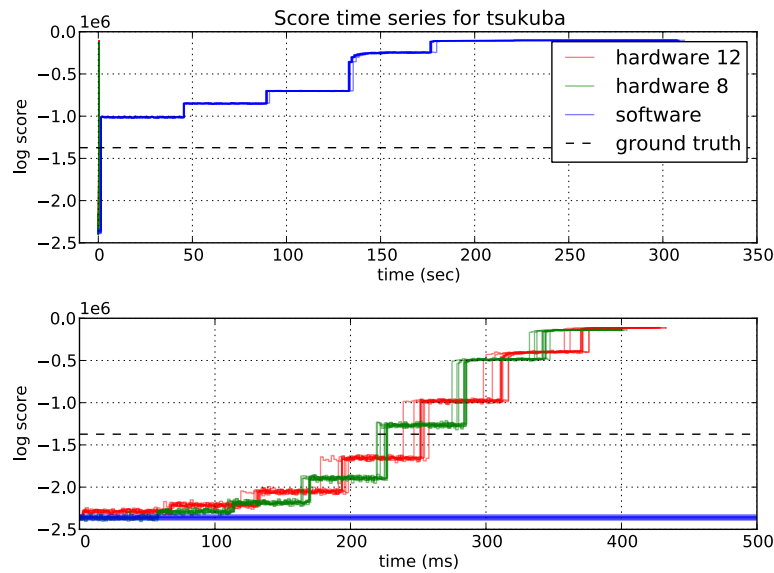
(c) software, 64-bit floating point



(d) hardware, 12 bits

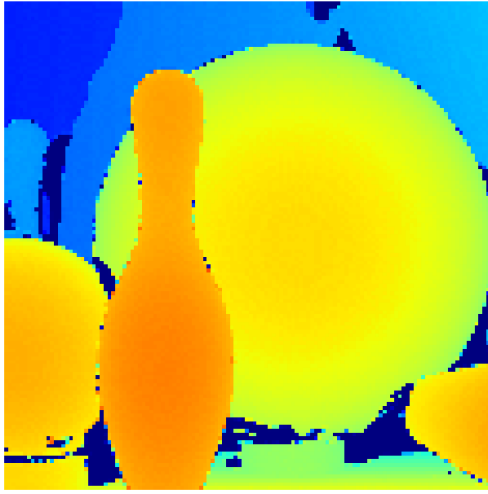


(e) hardware, 8 bits



(f) log score vs time, hw vs sw

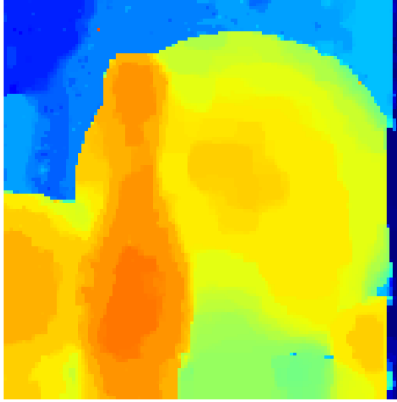
Figure 23: Middlebury “Tsukuba” example stereo depth dataset. Top row is the ground truth disparity map, and subsequent rows are the empirical mean mean of 10 full annealing sweeps for the double-precision software, and 12-bit and 8-bit hardware, circuits.



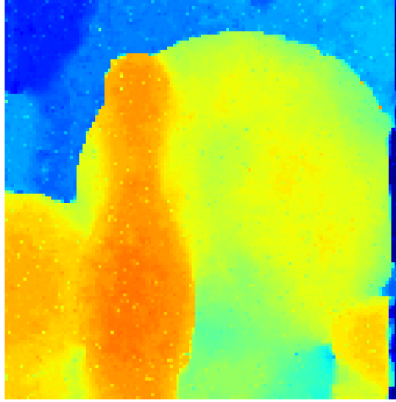
(a) Ground Truth



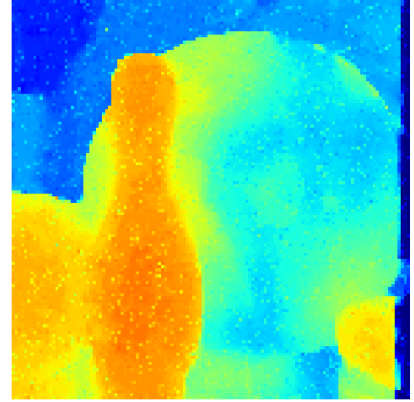
(b) Left Image



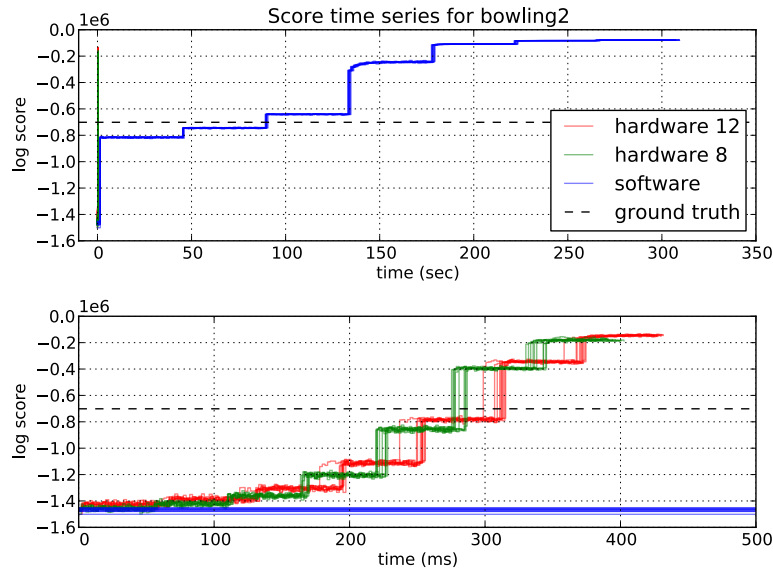
(c) software, 64-bit floating point



(d) hardware, 12 bits



(e) hardware, 8 bits



(f) log score vs time, hw vs sw

Figure 24: Middlebury “Bowling2” example stereo depth dataset. Top row is the ground truth disparity map, and subsequent rows are the empirical mean mean of 10 full annealing sweeps for the double-precision software, and 12-bit and 8-bit hardware, circuits.

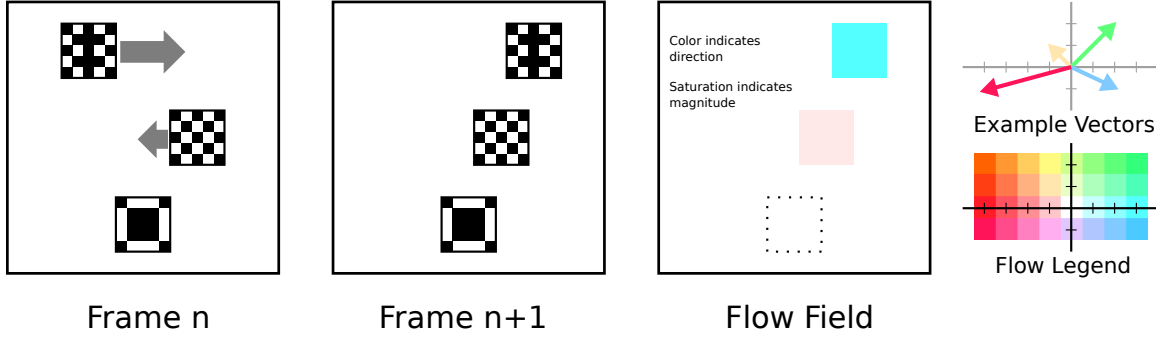


Figure 25: Optical flow uses the local pixel differences between frame n and $n + 1$ to compute a dense flow field, indicating the direction and magnitude of each pixel’s interframe motion. In the above, flow vector direction is indicated by color, and flow magnitude is indicated by saturation. Stationary objects are thus white.

We use three rectified, intensity-calibrated image pairs from the stereo benchmark dataset created by Scharstein and Szeliski (Scharstein, Szeliski, and Zabih 2001). These images have known ground truth depth maps to enable evaluation of our MRF engine at both 8.4 and 6.2 bit precisions. The results (figures 22, 23, 24) are shown for the full 64-bit software engine as well as 8- and 12-bit MRF engines.

The stochastic video processor finds a total score almost as good as the version computed using Gibbs sampling using IEEE-754 64-bit floating point on a 2.8GHz Intel Xeon, literally three orders of magnitude faster. This is in spite of the clock rate of the video processor being only 125 MHz. For most examples, the quality of the 12-bit solution is nearly as good as the floating-point version, although the limited dynamic range of the 8-bit version results in some areas (like the center of the bowling ball) failing to find ground truth.

13 Dense Optical Flow for Motion Estimation

The visual system is also required to estimate the motion of objects in the visual scene. This can be accomplished by computing the *optical flow field*, associating with each pixel a flow vector indicating the relative motion between the frame at time t and $t + 1$. The optical flow field also helps in parsing the 3D structure of the environment, estimating object boundaries, and computing the motion of the sensor. While Verri and Poggio (Verri and Poggio 1989) showed that the optical flow field is not the same as the true 2-D projected motion field, it is often close enough for computer vision applications.

Markov random fields have been used successfully to estimate discontinuous optical flow (Heitz and Bouthemy 1993). A MRF for optical flow can be computed as follows: we discretize the possible flow vectors (in our case, $k \in 0 \dots 31$) as the latent state variable values. Let flow vector value f_k indicates that pixel $x_{i,j}^t$ in frame t has moved to location $(i + F_k^x, j + F_k^y)$ at time $t + 1$. To compute the external field, we compare the neighborhood around source pixel $x_{i,j}^t$ in frame t with all k neighborhoods in frame $t + 1$.

$$f_{LP}(x_{ij}, x_{i'j'}) = -(|i' - i| + |j' - j|) \quad (9)$$

The associated latent pairwise potential is simply the Manhattan distance between the two latent state values. The range of motion is limited to the 32 nearest flow vectors surrounding the target point.

We compare inference time and posterior sample quality for three real-world datasets, captured in an office environment using a Prosilica GC650c gigabit ethernet color-CMOS camera under uncontrolled lighting conditions (figures 26, 27, 28). Adjacent frames were taken 10 ms apart.

High-quality flow fields were obtained with as few as 300 gibbs scans per frame, giving a maximum frame rate of 32 fps. The dynamic ranges encountered in the optical flow calculations resulted in the 6.2 engine producing very poor quality results; in this case, dense optical flow is a problem that needs the 8.4 engine.

14 Conclusion

We have shown how virtualization of state and inference elements in a stochastic circuit gives rise to more resource-efficient circuits for models with homogeneity and large amounts of state. This makes it possible to perform Bayesian inference on low-level vision problems in real-time with limited silicon resources.

The homogeneity present in the original model for f_{EF} is eliminated by the transformation outlined in section 9. The parametrized f_{LP} densities of the static circuit could be configured on a site-by-site basis, again with the configuration information living in the PSC (and thus runtime-reconfigurable).

The overall architecture of virtualizing over a region of the graph, and then selectively enabling parts of the resulting density calculation, suggests an engine for graphs with a more general topology. Each tile would be responsible for performing inference on some subregion of the graph, communicating with its neighbors, via message-passing the sampled state values, and selectively enabling and disabling the relevant densities.

Currently the engine described performs inference at every latent state site, making some applications (such as problems of filling-in missing regions (Scharstein, Szeliski, and Zabih 2001)) impossible. It would be easy to add an additional configuration bit at each site in the Pixel State Controller to selectively enable inference on a per-pixel basis.

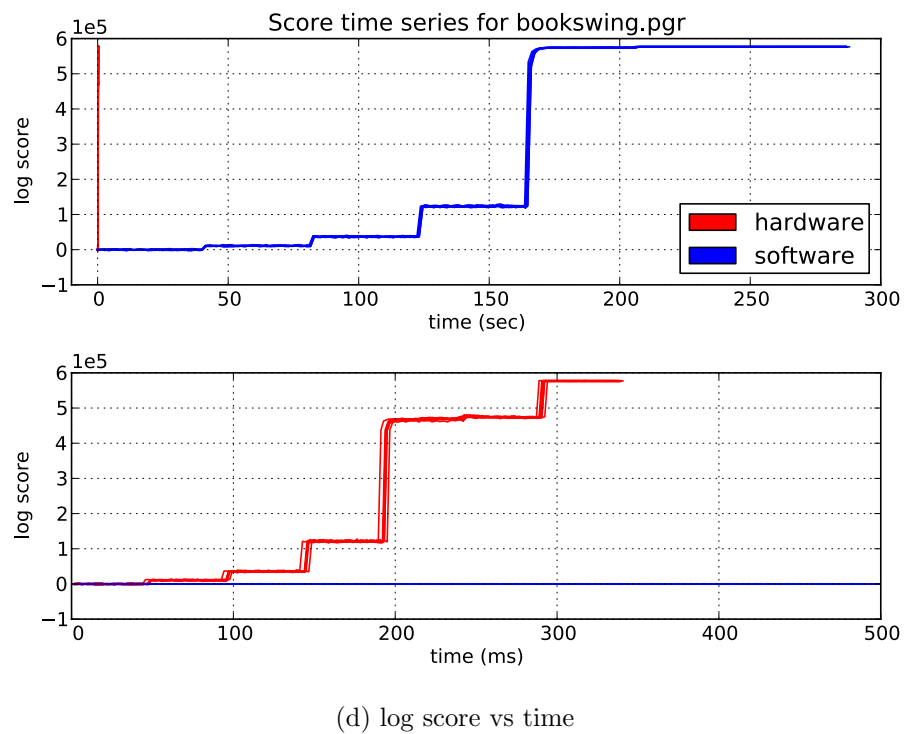
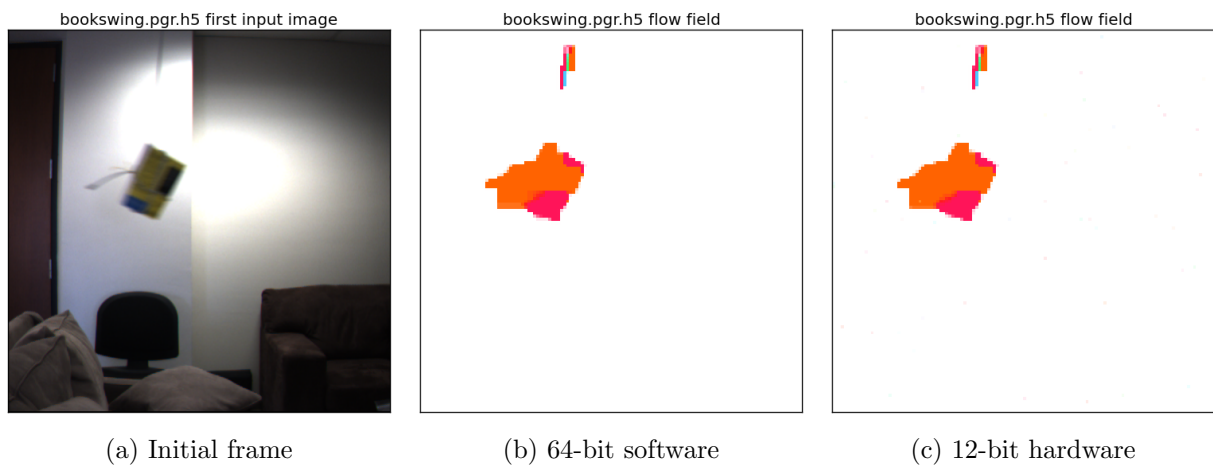


Figure 26: Optical flow results on the “bookswing” data

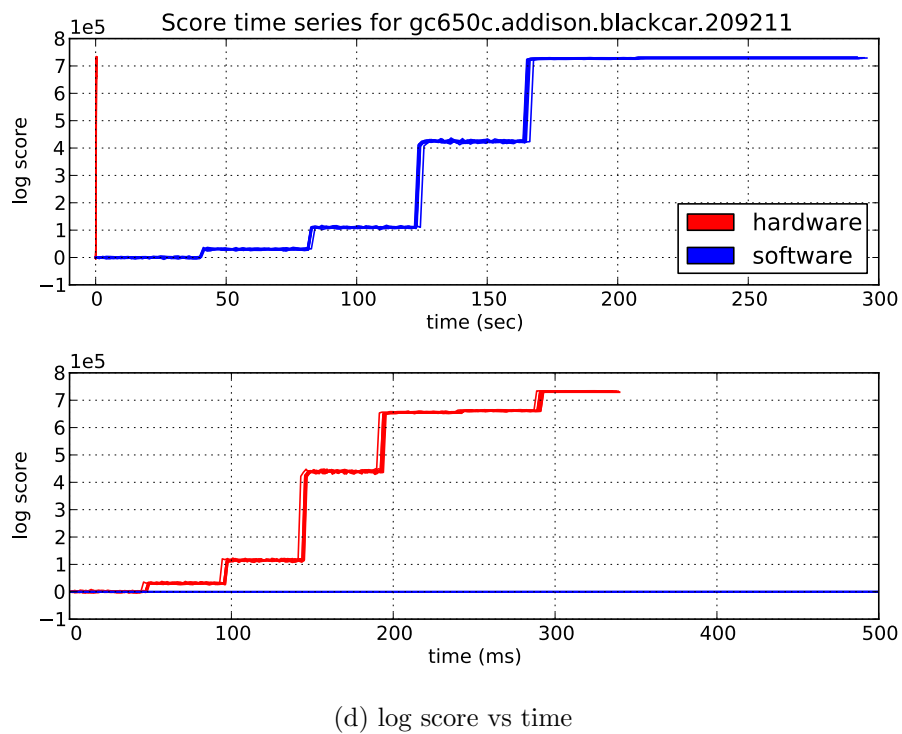
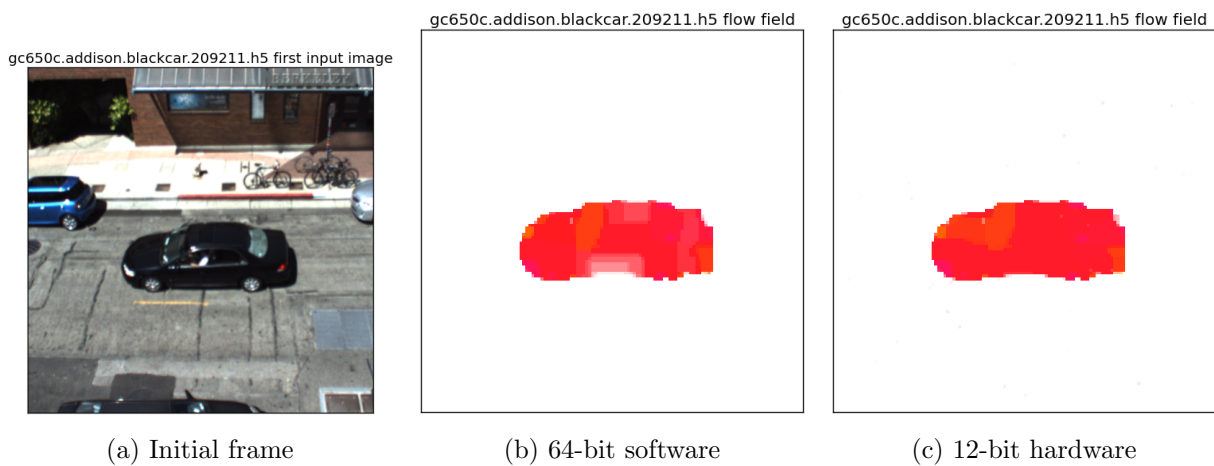


Figure 27: Optical flow results on the “blackcar” data

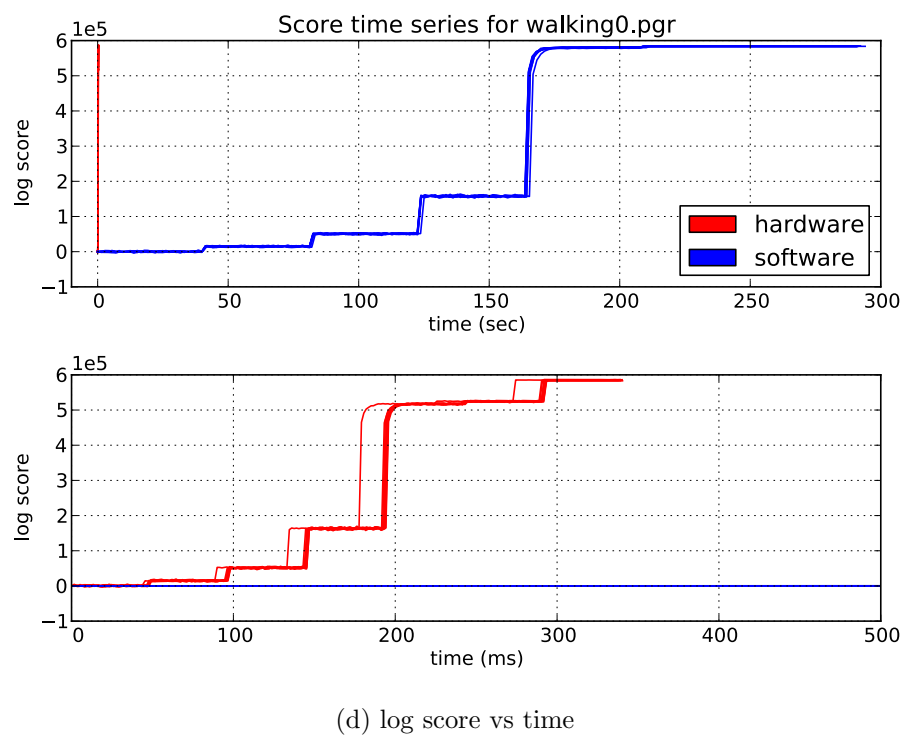
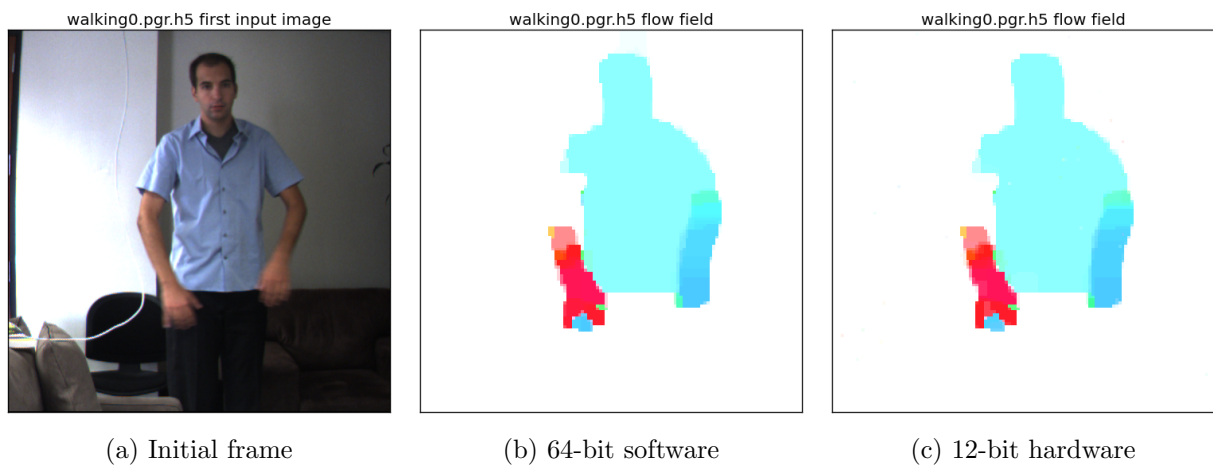


Figure 28: Optical flow results on the “eric” data

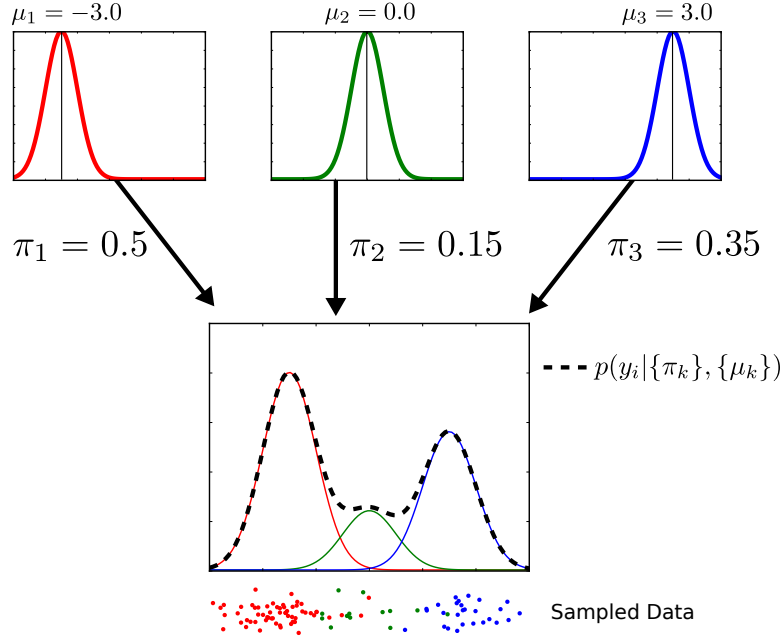


Figure 29: A Gaussian mixture model, illustrating the generative process for mixture model data. There are three latent classes from which the data are generated, with the first class generating a datapoint with $\pi_1 = 0.5$. The total probability of a data point y_i is the sum of the source probabilities weighted by their mixture weights. Samples from the resulting distribution are shown below.

Implementation details for perceptual learning

This section describes the Dirichlet process mixture model, which underpins our perceptual learning circuit, along with the inference scheme and circuit architecture we have developed. The Dirichlet process mixture model contains data-dependent conditional independencies that cannot be represented in a single fixed-structure probabilistic graphical model. It also contains detailed performance data clarifying the role played by conditional independence and parallelism in driving circuit efficiency.

15 Dirichlet Process Mixture Model

Probabilistically we view the “clustering” problem using a mixture model (Bishop 2006). Mixture models assume there are some number of hidden (latent) causes of our data, each cause having some distinct properties. When we observe the data, we don’t know which cause generated the data. When clustering we assume each cluster came from its own hidden cause. We will initially describe the model when the number of hidden causes is known, and then show how we extend it to the unknown case via the Dirichlet process.

The generative process for a mixture model is as follows. Assume there are K possible sources of data, each source having some associated set of parameters θ_k . That is, data from source k is distributed as $x_i \sim F(\theta_k)$, where $F(\cdot)$ is some known parametric distribution.

Each data point x_i is generated by first picking one of these sources with probability π_k ($\sum_k \pi_k = 1$) and then drawing $x_i \sim F(\theta_k)$, where F is often termed the “likelihood”. The $\{\pi_k\}$ are called *mixture weights*. Note that each x_i is drawn independently. Figure 29 shows an example of this generative process for three clusters of data from $N(\mu_k, 1)$ distributions.

This model easily extends to the multidimensional case, where each dimension is an independent “feature”. That is, D -dimensional data vector $\mathbf{x}_i$ is generated from cluster k such that the likelihood is

$$P(\mathbf{x}_i|\{\theta_k\}) = \prod_{d=1}^D P(x_i^d|\theta_k^d) \quad (10)$$

We can imagine there exists a vector $\mathbf{c}$ keeping track of the source of x_i – if x_i is drawn from cluster k , $c_i = k$. Thus “clustering” a dataset is attempting to compute the cluster assignment vector $\mathbf{c}$.

$$\mathbf{X} \sim P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\mathbf{c}|\{\pi_k\}) \quad (11)$$

15.1 Mixing weight prior

If we don’t know the mixing weights $\{\pi_k\}$ ahead of time, we can assign them a prior distribution.

$$P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\mathbf{c}|\{\pi_k\})P(\{\pi_k\}) \quad (12)$$

The derivation below is closely copied from (Griffiths and Ghahramani 2011). A natural fit for a prior over the mixing weights is the symmetric Dirichlet prior with concentration parameter $\frac{\alpha}{K}$:

$$p(\pi_1, \dots, \pi_K|\alpha) \sim \text{Dirichlet}(\alpha/K, \dots, \alpha/K) = \frac{\Gamma(\alpha)}{\Gamma(\alpha/K)^K} \prod_{j=1}^K \pi_j^{\alpha/K-1} \quad (13)$$

Given a particular assignment vector $\mathbf{c}$, we can integrate over all possible values of $\{\pi_k\}$ to arrive at

$$P(\mathbf{c}|\alpha) = \int_{\Delta_K} \prod_{i=1}^N P(c_i|\pi) d\pi \quad (14)$$

$$= \int_{\Delta_K} \frac{\prod_{k=1}^K \pi_k^{m_k + \alpha/K - 1}}{D(\alpha_1, \dots, \alpha_K)} d\pi \quad (15)$$

$$= \frac{D(m_1 + \frac{\alpha}{K}, m_2 + \frac{\alpha}{K}, \dots, m_K + \frac{\alpha}{K})}{D(\frac{\alpha}{K}, \frac{\alpha}{K}, \dots, \frac{\alpha}{K})} \quad (16)$$

$$= \frac{\prod_{k=1}^K \Gamma(m_k + \frac{\alpha}{K})}{\Gamma(\frac{\alpha}{K})^K} \frac{\Gamma(\alpha)}{\Gamma(N + \alpha)} \quad (17)$$

where we are using the shorthand $m_k = \sum_{i=1}^N \delta(c_i = k)$ is the number of objects assigned to class k .

Note that in this equation above, individual class assignments are no longer independent, but what they are is *exchangeable* – the probability of a particular assignment vector is the same as any other permutation of the assignment vector.

15.2 Dirichlet Process Prior

In all the examples above, we have assumed the number of latent classes, K , is fixed. Through various derivations outside the scope of this text, we can show that one infinite limit of the above dirichlet prior is the *Dirichlet Process*, also known as the Chinese Restaurant Process.

The Chinese Restaurant Process is named for the apparently-infinite seating capacity of many Chinese restaurants, and describes a particular algorithm for assigning mixing weights. The stochastic process defines a distribution of customer seatings at a restaurant with an infinite number of tables (Ferguson 1973)

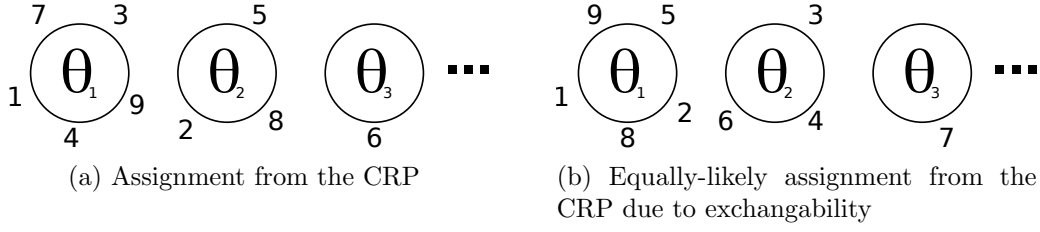


Figure 30: Two draws from the CRP with equal probability. Note that only the total number of customers at each table, and not their identity, affects the probability of the distribution. With each table (cluster) is associated some latent parameter θ_i .

In the CRP, N customers sit down, with the first customer taking a seat at the first table. The i th customer chooses a table at random, with

$$P(\text{occupied table } k | \text{previous customers}) = \frac{m_k}{\alpha + i - 1} \quad (18)$$

$$P(\text{next unoccupied table} | \text{previous customers}) = \frac{\alpha}{\alpha + i - 1} \quad (19)$$

$$(20)$$

where m_k is the number of customers sitting at table k . A crucial feature of the CRP is that the resulting distribution of table assignments is *exchangeable* (Schervish 1996) – the assignment vector is invariant to any permutation of the labels. This means that the probability of particular table seating arrangement is the same regardless of the order the customers arrived in. Thus, for customer i , the precise arrival ordering of customers 1 through $i - 1$ does not affect $p(c_i = k)$, only the total number of customers at each table does.

For a CRP mixture model, each table k corresponds to a mixture component, and has associated with it a set of cluster parameters θ_K .

15.3 Conjugacy

The θ_k are all often drawn from some base prior, or *hyperprior*, distribution. That is, the generative process has an additional step of first drawing K θ_k values from a prior distribution.

$$P(\mathbf{X} | \mathbf{c}, \{\theta_k\})P(\{\theta_k\})P(\mathbf{c} | \{\pi_k\}) \quad (21)$$

If a prior distribution and the likelihood exhibit *conjugacy* (Bishop 2006), then the posterior distribution on the parameter of interest (in this case, the θ_k) takes the same functional form as the prior. Important quantities of interest, such as the posterior distribution for $p(\theta_k | \{y_i\})$, and the posterior predictive distribution $p(y^* | \{y_i\})$ can then be computed based upon a reduced representation of the data, the *summary statistics*.

The Bernoulli distribution $p(x = 1 | \theta) = \theta^x (1 - \theta)^{1-x}$ is the distribution of a biased coin with heads probability θ . A conjugate prior for θ , the probability that $x = 1$, is the two-parameter beta distribution,

$$p(\theta | \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1 - \theta)^{\beta-1} \quad (22)$$

If we use the Beta distribution as the prior on Bernoulli(θ), conjugacy results in the posterior after observing $x = 1$ as:

$$p(\theta | x = 1, \alpha, \beta) = \frac{\Gamma(\alpha + \beta + 1)}{\Gamma(\alpha + 1)\Gamma(\beta)} \theta^{1+\alpha-1} (1 - \theta)^{\beta-1} \quad (23)$$

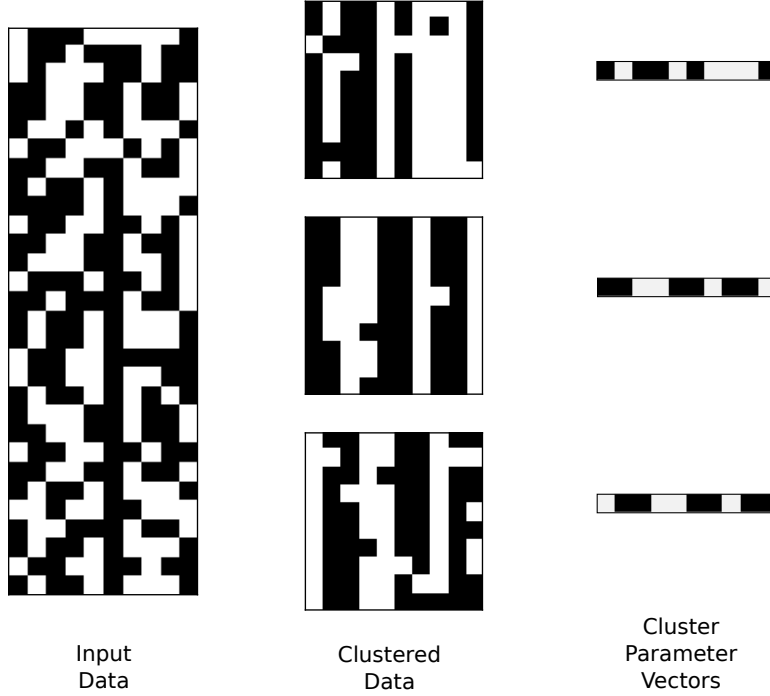


Figure 31: An example of clustering via a 10-dimensional binary mixture model. The input data is at left – each row is a data point in the 10-dimensional binary space. The resulting discovered clusters (middle) are associated with latent “parameter vectors”, one for each cluster, which are estimated from the data.

The observations y_i are drawn independently, and as a result, if m are the number of datapoints with $y_i = 1$ in the dataset and n are the number of datapoints with $y_i = 0$ then we can see that

$$p(\theta|m, n, \alpha, \beta) = \frac{\Gamma(\alpha + \beta + m + n)}{\Gamma(\alpha + m)\Gamma(\beta + n)} \theta^{m+\alpha-1} (1 - \theta)^{n+\beta-1} \quad (24)$$

A more detailed derivation can be found in (Bishop 2006).

15.4 Gibbs Sampling

Using a conjugate likelihood model with known parameters and the CRP as a prior on class assignments, we are thus interested in sampling from the posterior distribution on class assignments,

$$P(\mathbf{c}|\mathbf{X}) \propto P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\{\theta_k\})P(\mathbf{c}|\alpha) \quad (25)$$

The combination of conjugacy and exchangeability allows us to exactly sample from the resulting conditional distribution on cluster assignments. This lets us Gibbs sample using the scheme from (Rasmussen 2000). We briefly review some of the key terms necessary here.

Let $\mathbf{c}_{-i}$ be the assignments of all objects except for the object of current interest, $\mathbf{c}_i$. Thus

$$P(c_i = k|\mathbf{c}_{-i}, \mathbf{X}) \propto P(\mathbf{X}|\mathbf{c})P(c_i = k|\mathbf{c}_{-i}) \quad (26)$$

The CRP above readily provides $P(c_i = k|\mathbf{c}_{-i})$,

$$P(c_i = \text{occupied class } k|\mathbf{c}_{-i}) = \frac{m_{k,-i}}{\alpha + N - 1} \quad (27)$$

$$P(c_i = \text{newclass}|\mathbf{c}_{-i}) = \frac{\alpha}{\alpha + N - 1} \quad (28)$$

$$(29)$$

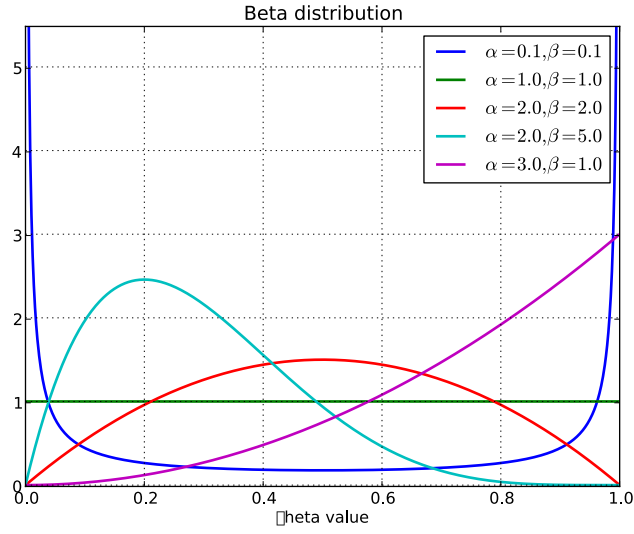


Figure 32: The two-parameter Beta distribution, the prior likelihood for the conjugate Beta-Bernoulli data model

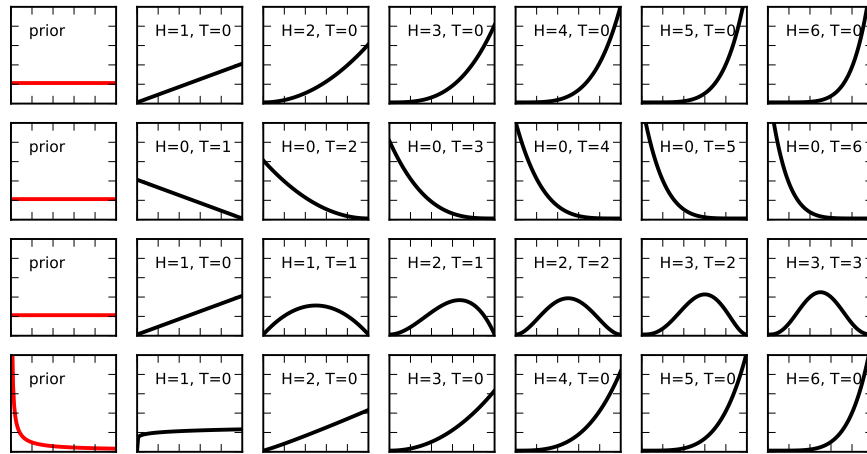


Figure 33: Sequential updates to the Beta-Bernoulli conjugate data model. Starting with no observed data and a prior distribution (left), additional observations shift the posterior distribution of θ in an intuitive way. Note on the last row that, even if the prior is wildly biased, sufficient data “overwhelms” the prior’s effect on the posterior distribution.

Via conjugacy above, we can easily compute $P(x_i|\mathbf{X}_{-i}, \mathbf{c})$. This defines all the terms we need to Gibbs sample assignments under the Dirichlet process mixture model.

16 Architecture

Here we present a stochastic architecture for efficient data streaming and sampling in the Dirichlet-process mixture model for binary (Bernoulli-distributed) data and a conjugate Beta distribution prior. There are two features which make this system stand out from our previous stochastic architectures:

First, the architecture is dynamic – this is the first case where the set of possible random values changes as inference unfolds. This is in contrast to our simplest proposals for transition circuits and compiler generated circuits, where every random variable has an equivalent, dedicated stochastic circuit element, or our lattice Markov Random Field engine, where the set of random values is explicitly known ahead of time, even though some random variables were virtualized.

Second, this is the first example of an architecture where the data is streamed through the core system. Everything we’ve discussed up to this point has required that all data be present in the circuit, at once for inference to occur. Here, we relax that assumption, instead streaming a row of data at a time through the system, thus allowing for far larger datasets, enabling “big-data” style applications.

To better understand the operation of the circuit, we stream in a dataset that looks like the input data in figure 31 – a dense binary matrix where each row is a binary vector of observations. We perform nonparametric clustering to group the rows, identifying for each grouping a canonical “parameter vector”. The circuit learns both the parameter vectors and, importantly, the number of parameter vectors.

16.0.1 Terminology

To that end, we will use HP when referring to the hyperparameters for the data model (for the Beta-Bernoulli data model, $HP = (\alpha, \beta)$), and SS when referring to the sufficient statistics (for the Beta-Bernoulli model, $SS = (m, n)$). Note that our model is multidimensional – here we describe the conjugate likelihood for a single feature.

The conjugacy of the likelihood means that we only need to store the sufficient statistics for a group, allowing us to cluster very large amounts of data using relatively few on-device state bits. Sufficient statistics are stored in constant-time-access SRAM on-device. The finiteness of this RAM means that we can only cluster datasets with up to K_{MAX} clusters, but this is rarely a problem in practice – while the Dirichlet process mixture model accommodates a potentially infinite number of latent clusters, most real-world datasets have far fewer.

Overall operation is as follows – data is streamed in one row at a time. We sample an assignment from that new row based on previously seen data, by Gibbs sampling all possible latent group assignments. This sampling, as well as the dynamic creation and destruction of new latent groups, is handled by the Group Manager. A multi-feature module stores the sufficient statistics for all groups and computes the $P(c_i = k|\{y_{-i}\}, \{HPs\})$ necessary for the resulting sampling step. Once a row’s group assignment has been sampled, the sufficient statistics for that group are updated, and the group assignment is streamed out of the circuit.

16.1 Parallelism

Conditional independence allows us to compute cluster assignments in parallel. The multi-feature module computes the probability that the current row y_i was generated by a particular cluster k for all features in parallel. The resulting scores are simply added via a pipelined adder tree.

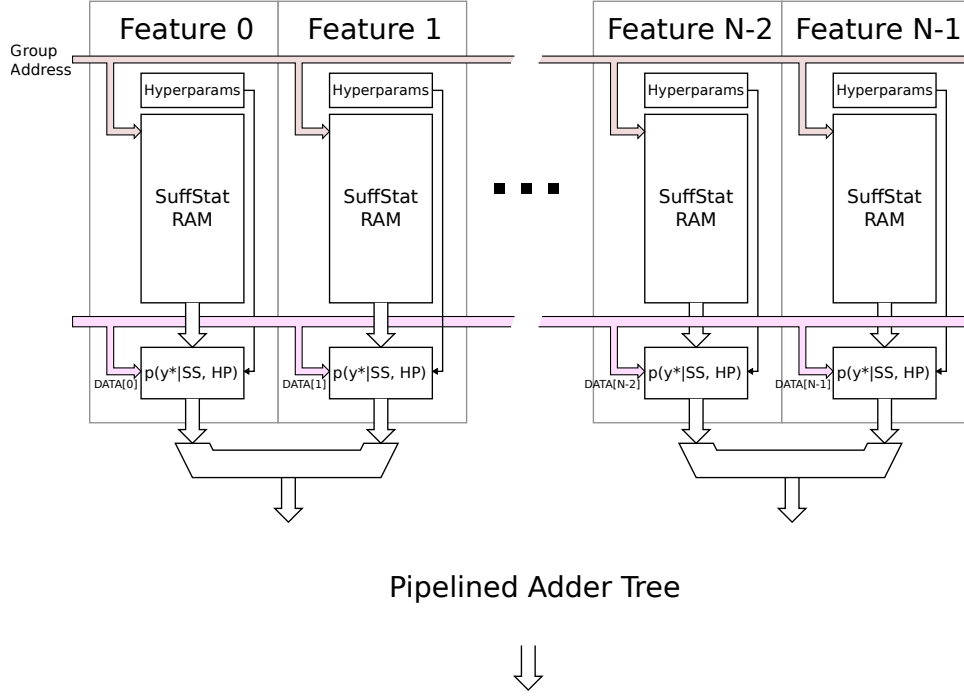


Figure 36: Feature Parallel evaluation architecture: A common group is selected across all features, and the posterior predictive is computed for each datum using its associated feature. As the features are independent, and the probabilities are in $\log_2$ space, a simple pipelined adder tree is used to accumulate the total score.

16.2 Component Models

To evaluate the probability of a data point being generated by a particular group, we need to compute $p(y^*|SS, HP)$. Thus we must store those sufficient statistics someplace stateful. We must also implement component-model-specific hardware to update those sufficient statistics when a data point is either added to or removed from a group.

Figure 37 shows the interface for the sufficient-statistics mutation module “SSMutate” and the posterior predictive evaluation module “PredScore”. SSMutate is heavily pipelined (hence the START and DONE signals) and returns the updated values on NEWSS based on whether the data point is being added to the group (ADD = 1) or removed from the group (ADD = 0). PredScore takes in the current values for the sufficient statistics (SUFFSTATS and the hyperparameters (HYPERs) and returns $\log P(y^*|SS, HP)$. It is also heavily pipelined, with one tick per sample throughput.

16.3 Beta Bernoulli Component Model

Having shown above that the posterior predictive $p(x = 1|D, HP)$ for a beta bernoulli component model is

$$p(x = 1|\{x_i\}, \alpha, \beta) = \frac{m + \alpha}{\alpha + \beta + m + n} \quad (30)$$

to compute the log score we must compute

$$\log p(x = 1|\{x_i\}, \alpha, \beta) = \log(m + \alpha) - \log(\alpha + \beta + m + n) \quad (31)$$

This requires an internal approximation to the log function, which we do via linear interpolation. The

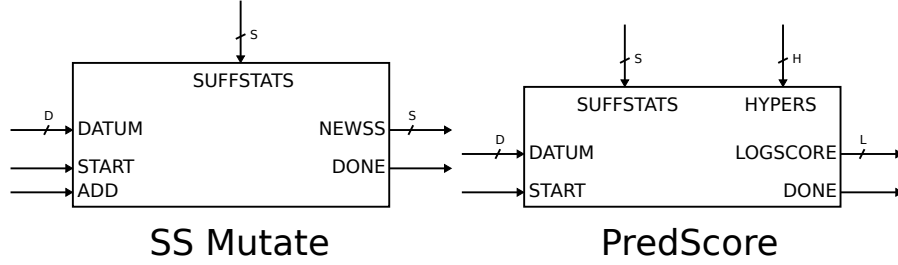


Figure 37: Every component model requires the implementation of two modules, a “SSMutate” module to compute updates to sufficient statistics based on the addition or removal of a datum to a group, and a “PredScore” module to compute the probability of a datapoint being generated by a particular group.

resulting approximation error is shown in Figure 38, which stays very small even as we vary the sufficient statistics over a wide range.

16.4 Multi-Feature Module

Figure 35 shows the internal parallel-evaluation process of the multi-feature module. At compilation time, the specific feature configuration (number of features, feature type, bit-precision) parameterizes this module. For each element in the data vector y_i we compute the posterior predictive $p(y_i|SS_i, HP_i)$ using the above-described components. A massive pipelined parallel adder tree performs the reduction. Carefully keeping track of the pipeline stages enables deep pipelining and thus single-cycle evaluation of a given row belonging to each group.

16.5 Group Manager

The group manager is responsible for tracking which entries in sufficient-statistics RAM are in use, and enabling the dynamic creation and deletion of groups as the inference process dictates.

Even if the multi-feature module is aggressively pipelined to allow for single-cycle throughput, the sufficient statistics must be delivered rapidly enough such that the score can be evaluated with no gaps. Thus, the group manager must be able to burst out a list of all addresses of groups currently in use.

The group manager does this (figure 39) by keeping two stacks: an “available” stack of addresses not in use, a “used” stack of addresses currently in use, and a look-up table mapping between addresses and locations in the “used” stack. Thus creating a new group is $O(1)$: pop an address A from the available stack, push A onto the used stack, and write the current “used” stack pointer at A in the lookup table. Group deletion is also $O(1)$: to delete group A , look up its location in the used stack via the look-up table; copy the top entry from the used stack over that address, updating its entry in the LUT along the way. Then push A onto the available stack. Bursting is $O(K)$, as we simply read out the entries in the used stack.

16.6 Streaming Inference

The streaming interface (Figure 40) to the clustering circuit enables rapid clustering without necessitating the circuit keep all of the data locally; rather the only state stored on the device are the sufficient statistics.

Data is written in a bit-packed format, and is pipelined – the next row of data can be written while the circuit is performing inference on a current row. Hyperparameters and other per-feature configuration information can be written via the feature-control interface.

Inference is controlled by asserting **GO** with a command word and an input group. The command word serves as an opcode, enabling particular aspects of the circuit to allow for initialization, inference, data addition and removal, and prediction.

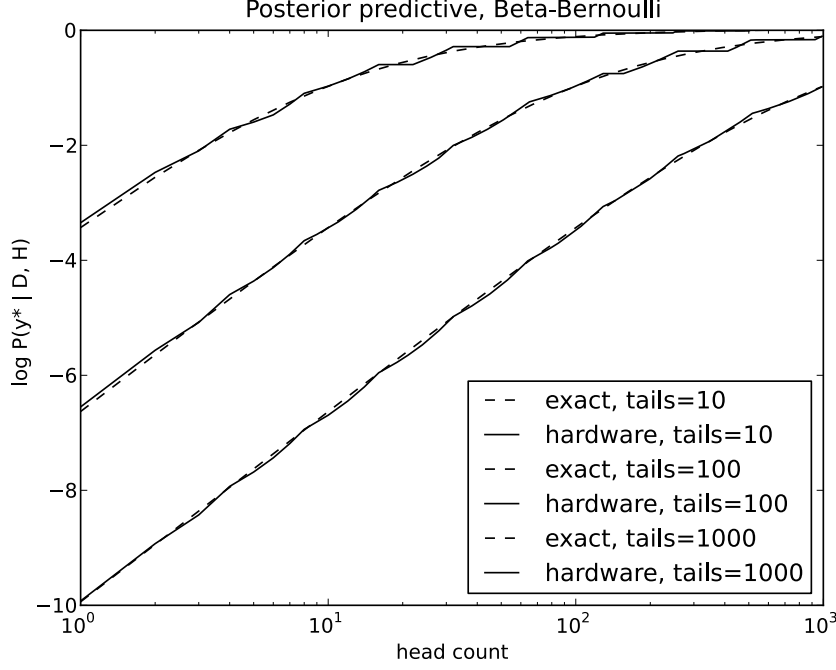


Figure 38: Beta Bernoulli posterior predictive hardware approximation results. The dashed line is the exact, floating point result, whereas the solid line is the answer generated by the PostPred module. The results are for $P(x = 1|x_i)$, and are shown as we systematically vary the number of observed heads from 1 to 1000, for three values of observed tails.

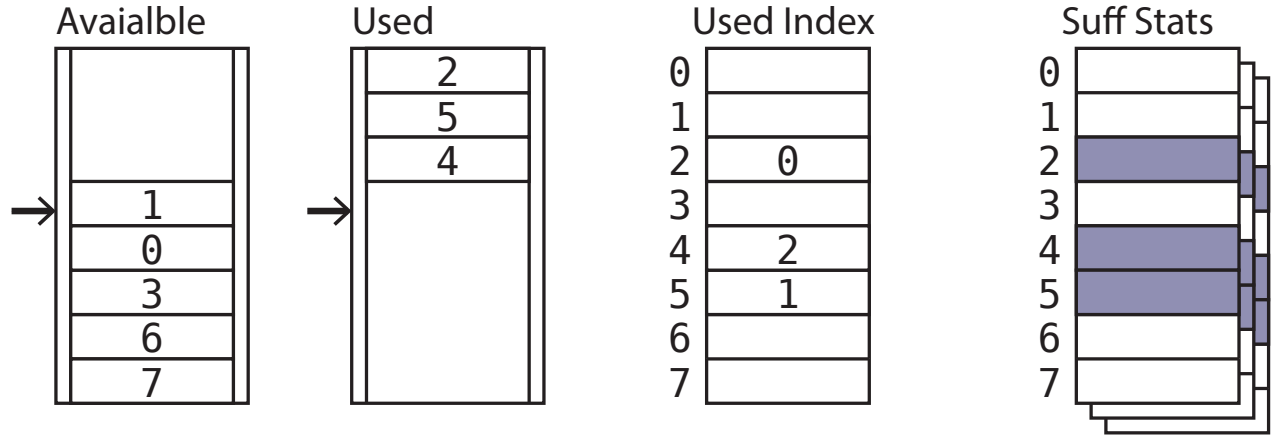
The specific bits of the command word are shown in table 5. To add a new datapoint and pick the right group for that datapoint based on the existing data, we set SAMP=1, ADD=1, GROUPSEL=1, DLATCH=1, NEWG=1. Once DONE is asserted, GRPOUT is the group assignment of this new datapoint. To perform generic inference without adding or removing data, set REM=1, SAMP=1, ADD=1, GROUPSEL=1, DLATCH=1, and NEWG=1. This will remove the datapoint, perform inference (creating new groups as necessary), and then assign the datapoint to the resulting group.

17 Results

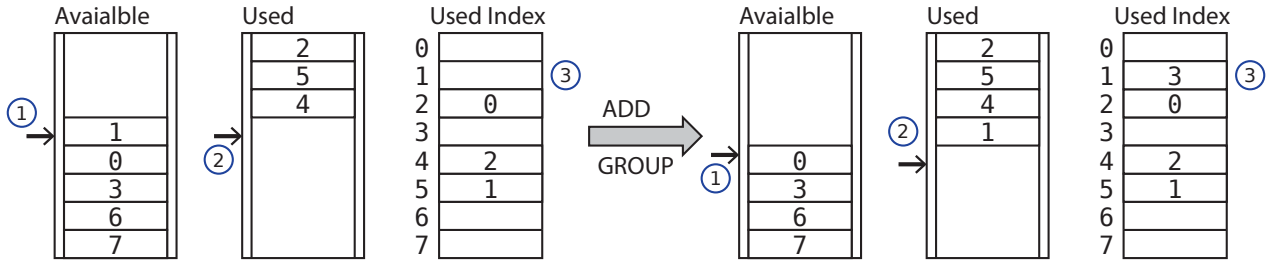
We validate the resulting circuits through a series of tests assessing the impact of bit precision, including explicitly comparing the posterior distribution with an exact enumeration, testing behavior with synthetic and incremental data. Runtime performance is compared against theoretical predictions and an optimized software implementation on commodity hardware.

17.1 Resource Utilization

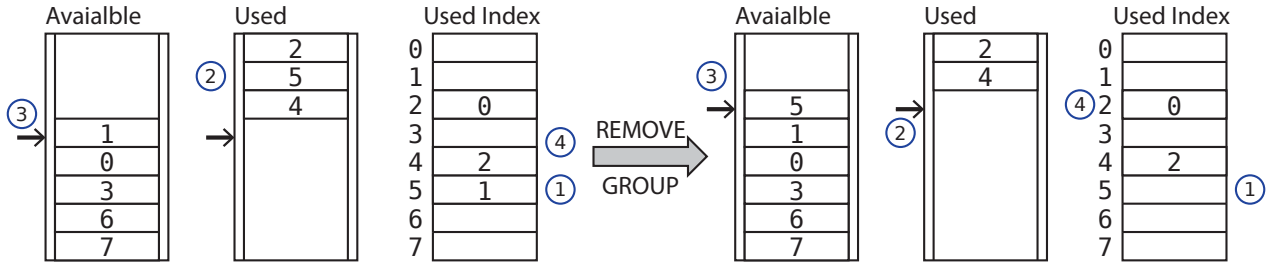
Table 6 shows the resource utilization for the hardware designs synthesized to measure KL, below. Table 7 shows the resource utilization for the 256 feature circuit used in all other experiments. The internal score calculation is expressed in our standard $m.n$ fixed point format, and is listed under the “precision” column. The number of bits used by the Gibbs sampler internally for sampling is Q . Each of these circuits can handle 1024 possible groupings and a maximum 2^{16} datapoints per group, and runs at 125 MHz.



(a) Group manager for tracking sufficient statistics. Available is a stack of unused entries in sufficient statistics RAM; used is a stack of the in-use sufficient statistics ram locations. The “used index” enables lookup from a sufficient-statistics location to a location in the used stack, enabling $O(1)$ group removal.



(b) Creating a new group. 1. the location for the sufficient statistics is determined from the available queue, which is 2. pushed onto the “used” stack. That stack position is saved in the correct location in the “used index”.



(c) Removing a group. To remove a group (in this case, group 5), we first look up its location in the “used” stack via the used index. We then 2. remove it from the used stack, 3. push it back onto the available stack, and move the top of the used stack down to the location previously held by 5. This necessitates 4. an update to the used index.

Figure 39: The group manager, which provides dynamic $O(1)$ creation, $O(1)$ deletion, and $O(K)$ bursting of group addresses.

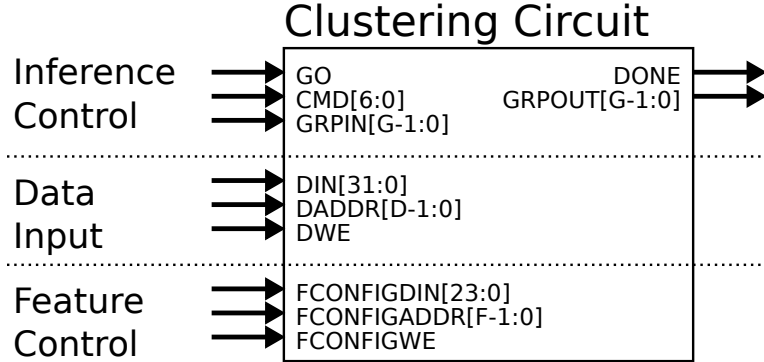


Figure 40: Interface for the Dirichlet Process Mixture Model Stochastic Circuit. Data is loaded asynchronously via the data interface, and feature hyperparameters are set via feature control. Inference is controlled via a 7-bit command word.

Table 5: Command word bits – see text for examples of common settings

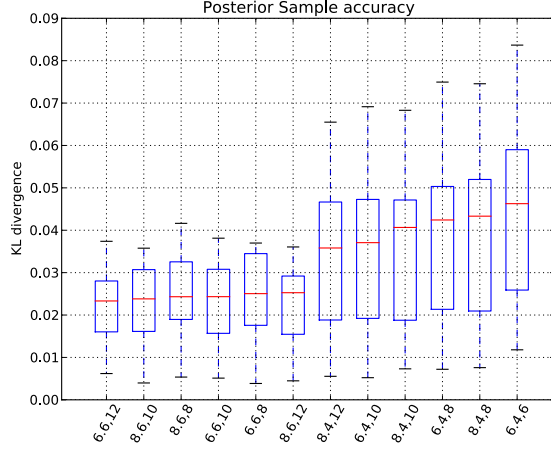
Bits	Name	Description
0	REM	Remove the current data point from the group indicated by GRPIN
1	SAMP	Perform a "sampling" step, that is, determine which group we should assign this data point to
2	ADD	Add this datapoint to a group (generally after sampling)
4:3	GROUPSEL	When we add the group, which group source do we use, the input (= 0) (GRPIN) or the one you just sampled (= 1) or the new one we generated (= 2)
5	DLATCH	Latch the data – make the data in the shadow register set "live" . Generally this is set to 1.
6	NEWG	Attempt to create a new group: this should always be 1 when sampling, but otherwise can be set to 1 to attempt to assign (force) a datapoint to a new group

Table 6: Resource utilization for 16-feature clustering circuit, maximum 1024 clusters, maximum 2^{16} datapoints per cluster.

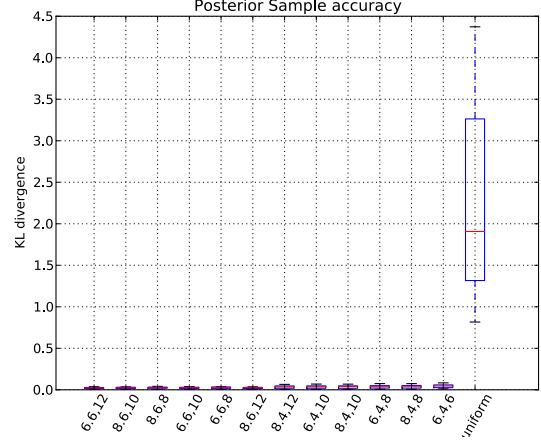
Features	Precision	Q bits	Resources		
			Slice FFs	Slice LUTs	BRAMS
16	4.2	4	17890	13660	56
16	4.4	4	18354	14929	56
16	4.4	6	18368	14940	56
16	4.4	8	18382	14955	56
16	6.2	4	18188	14041	56
16	6.2	6	18190	14048	56
16	6.4	4	18662	15581	56
16	6.4	6	18667	15465	56
16	6.4	8	18681	15480	56
16	6.4	10	18695	15500	56
16	6.6	8	19170	16946	56
16	6.6	10	19184	16970	56
16	6.6	12	19200	16988	56
16	8.2	6	18498	14327	56
16	8.2	8	18508	14340	56
16	8.4	6	18984	15861	56
16	8.4	8	18989	15762	56
16	8.4	10	19003	15780	56
16	8.4	12	19019	15801	56
16	8.6	8	19478	17215	56
16	8.6	10	19492	17240	56
16	8.6	12	19508	17258	56

Table 7: Resource utilization for 256-feature clustering circuit, maximum 1024 clusters, 2^{16} datapoints per cluster.

Features	Precision	Q bits	Resources		
			Slice FFs	Slice LUTs	BRAMS
256	6.4	6	77332	121128	296
256	6.4	8	77346	121147	296
256	6.4	10	77360	121162	296
256	6.6	8	84548	142893	296
256	6.6	10	84562	142917	296
256	6.6	12	84578	142935	296
256	8.4	8	81982	125040	296
256	8.4	10	81996	125064	296
256	8.4	12	82012	125086	296
256	8.6	8	89197	146778	296
256	8.6	10	89211	146812	296
256	8.6	12	89227	146829	296



(a) KL for different bit precisions



(b) KL for different bit precisions, with comparison to completely random clustering

Figure 41: Kullback-Liebler divergence between true, explicitly enumerated distribution and collection of posterior samples.

17.2 Explicit posterior samples

The sampling system embodied in our circuit should produce samples from the distribution $P(c|x_i, HPs)$. As we’ve done in previous chapters, here we compare the distributions from the sampler and the true known distribution.

In the case of our Dirichlet process mixture model, the size of the posterior space grows very quickly. The number of clusters possible in a dataset with n rows follows the Bell Numbers (*Bell Number – from Wolfram MathWorld*). Thus explicit enumeration and scoring of this dataset quickly becomes impracticable in the large data limit. Here we compare with $n = 10$.

We randomly generate ten 16-feature datasets, and for each firmware bit precision we evaluate the KL between the true posterior distribution and the result of 100000 samples, with a sample taken after every 100 iterations of the core Gibbs steps.

17.3 Basic Inference

17.3.1 Recovering Ground Truth

We create several synthetic datasets with known ground truth, and vary the number of true underlying groups and the number of rows per group. We initialize the data to a single group, as this requires the engine to do the most work to break symmetry and find a robust clustering. We set the hyperparameters to match their ground truth settings – for the Beta prior to be $\alpha = 0.1$, $\beta = 0.1$

To measure the similarity between a found clustering and the ground truth, we use the adjusted Rand index (ARI, (Rand 1971)). ARI ranges from 0 to 1.0, with 1.0 being identical clusterings. For each group/row configuration, we create ten data sets, and perform inference on them. The results are plotted in Figure 42.

As we can see from the figure, synthetic datasets with fewer numbers of datapoints per group are in some sense “harder” to cluster – a stable clustering equivalent to ground truth takes many more sweeps.

17.3.2 Incremental addition of data

The streaming interface enables the incremental addition of data and continual reevaluation of the clustering of existing data. The nonparametric mixture model always places non-zero probability mass on a

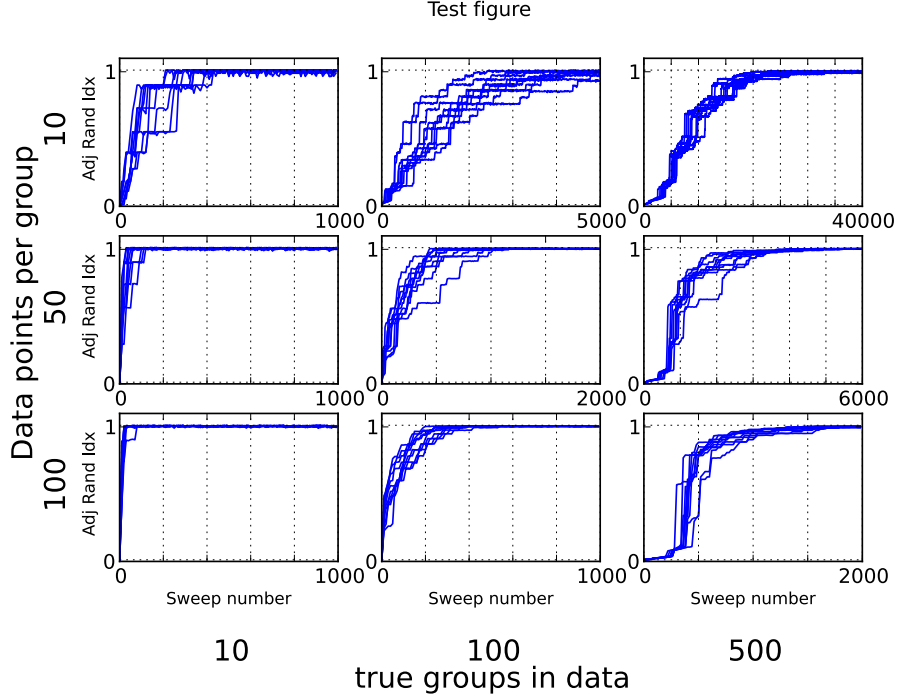


Figure 42: Recovery of ground truth for all-in-one-group initialization, across various numbers of true groups in data and rows per group. The adjusted Rand index (see text) measures cluster similarity – an ARI of 1.0 means recovered clusters are identical to ground truth.

new datapoint belonging to a new group. To test if our circuit correctly recapitulates this behavior, we generated a series of 40 datapoints for each of 10 groups, and then added them one row at a time and observed the resulting clustering. Figure (43 shows the result – the circuit closely tracks the true number of groups in the data, although expresses uncertainty for each new datapoint.

17.4 Performance vs software

We can directly compare the time necessary to sample a new assignment for a row, given the existing data and group structure. This is the fundamental operation that the model performs. For the 256-feature circuit, we create a variety of synthetic datasets and disable the final mutation step, such that the number of groups remains constant throughout inference. The time taken is the same, however the final write-enable has been disabled. We compare this Gibbs sampling performance with a custom beta-Bernoulli DP mixture model implemented by hand in highly optimized C++.

We expect a linear relationship between the time required for a row sample and the number of latent groups, which we see in Figure 44. By examining the slope, we can compute the marginal time necessary for a row operation. The circuit takes 7.4 ns per operation, whereas the software implementation takes 15.3 μ s on a 2.8GHz Intel Xeon CPU. For large datasets which fully take advantage of the circuits extensive pipelining, this suggests a two-thousand-fold speed increase – 250x from parallelism, and another 8 from dedicated hardware. We can compare the performance of the circuit to the simulated performance with PCI-E overhead removed (figure 45). The two slopes are in close agreement, however we can see that the PCI-E overhead adds 4 μ s per row.

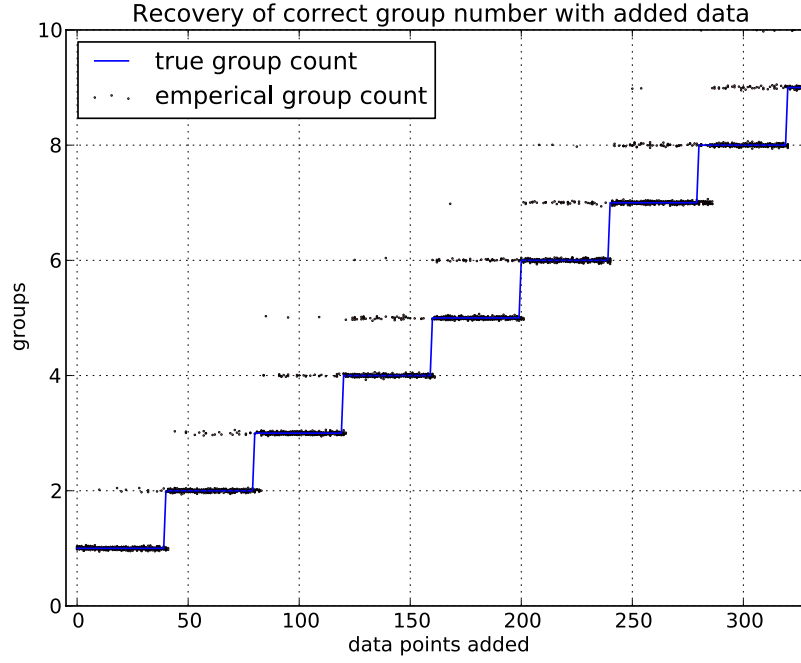


Figure 43: Adding new datapoints to the engine. Every set of 40 datapoints belongs to a new group (true group count is in blue). The model correctly estimates the number of latent groups in the data. Jitter has been added to the y-axis to enable density visualization.

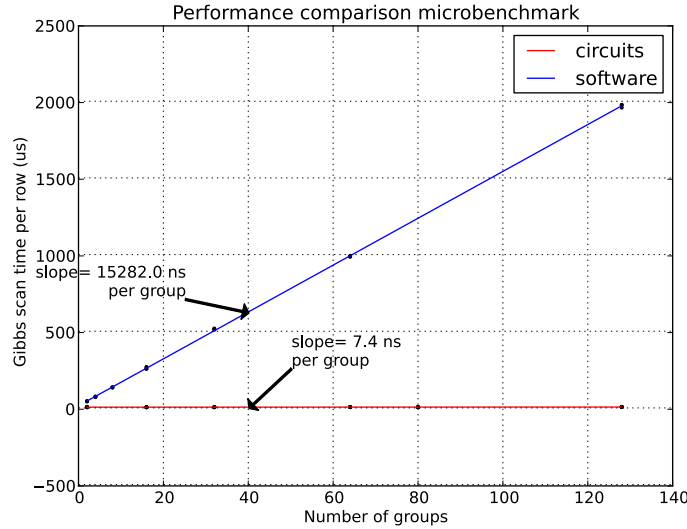


Figure 44: Comparison of performance between the stochastic circuit and a hand-optimized mixture model performing the same operations on a desktop computer. Here, we measure the time necessary to compute the group assignment distribution for a single row. The slope of each line is the time necessary to evaluate the probability of the row being assigned to the particular group.

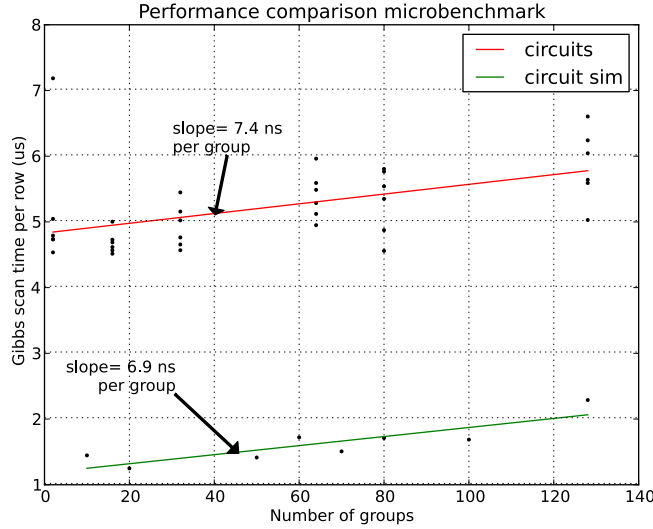


Figure 45: The circuit’s performance in actual hardware, including PCI-E bus and host control overhead, is very close to the theoretical performance from VHDL simulation of the circuit without complex interface hardware.

18 Perceptually-plausible clustering of handwritten digits

For example, humans are good at recognizing handwritten digits, even when they are drawn in a variety of styles. Here we use a database of handwritten digits to demonstrate perceptual clustering, which produces human-interpretable results even at low bit-precision.

The MNIST hand-written digit database (Lecun et al. 1998) consists of size-normalized and centered 28 by 28-pixel binarized images of hand-written digits. They are often used as a benchmark for supervised learning methods. The original dataset consists of 60k labeled training images and 10k labeled test images.

We use 20k images from the training set (2000 per digit) and 1000 images from the test dataset (100 per digit). We downsample each 28x28 image to 16x16 and treat them as flat (one-dimensional) binary vectors.

This encoding throws away much of the spatial information in the image; that is, our model has now knowledge of pixel locality – feature F_{33} might be for a pixel at (3, 4) and feature F_{34} for a pixel at (4, 4), but the model does not exploit this relationship. As an additional consequence, our performance would be exactly the same were the pixels randomly permuted.

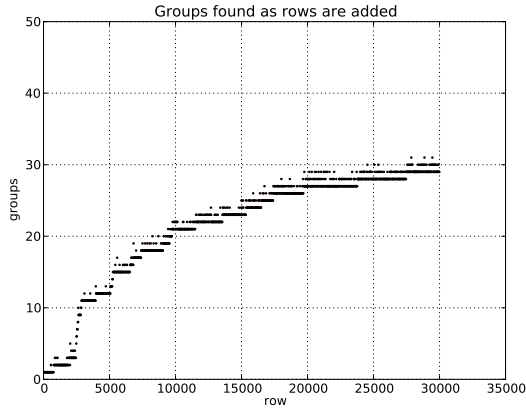
18.0.1 Clustering

We use both (8.6), 12-bit and (6, 4), 10-bit circuits on the original MNIST dataset. We perform 4 gibbs scans for each new row added, and can see how, in the presence of more data the circuit finds more plausible clusters (Figure 46). All hyperparameters were set to 1.0.

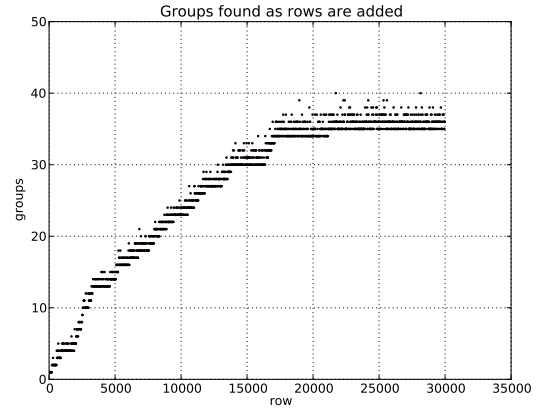
We organize the clusters by their “most common true class” in figure 47. The different “styles” of each digit are readily apparent, as is the perceptual ambiguity of certain styles of digits (8 and 3, for example).

18.0.2 Prediction

While we’ve spent the entire time discussing clustering in an unsupervised context, when we know ground truth for each data point, it’s possible to use the system to make supervised predictions. Our streaming

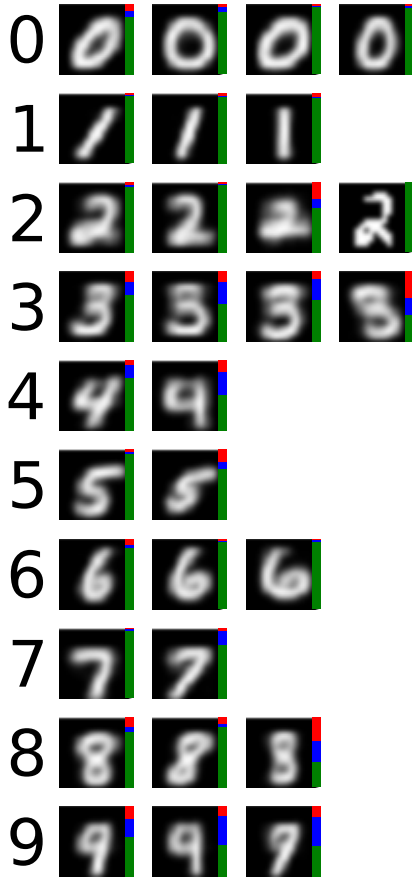


(a) 8.6,12-bit precision

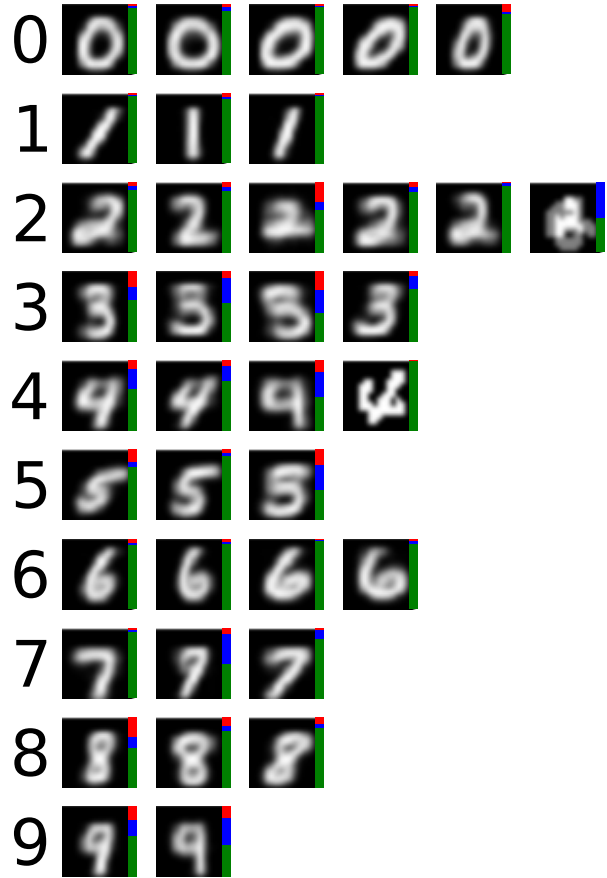


(b) 6.4,10-bit precision

Figure 46: The number of cluster groups found in the MNIST dataset as we add new digits; the model continually finds subtypes of clusters in the presence of more and more data.



(a) 8.6,12-bit precision



(b) 6.4,10-bit precision

Figure 47: The clusters found in 20000 example digits, organized by the most common class present in that cluster. Various different “styles” of writing each digit are found. Bars at right indicate (green) the fraction of the cluster made up by the most common digit, (blue) the fraction in the second-most-common digit, and (red) the remaining digits in the cluster.

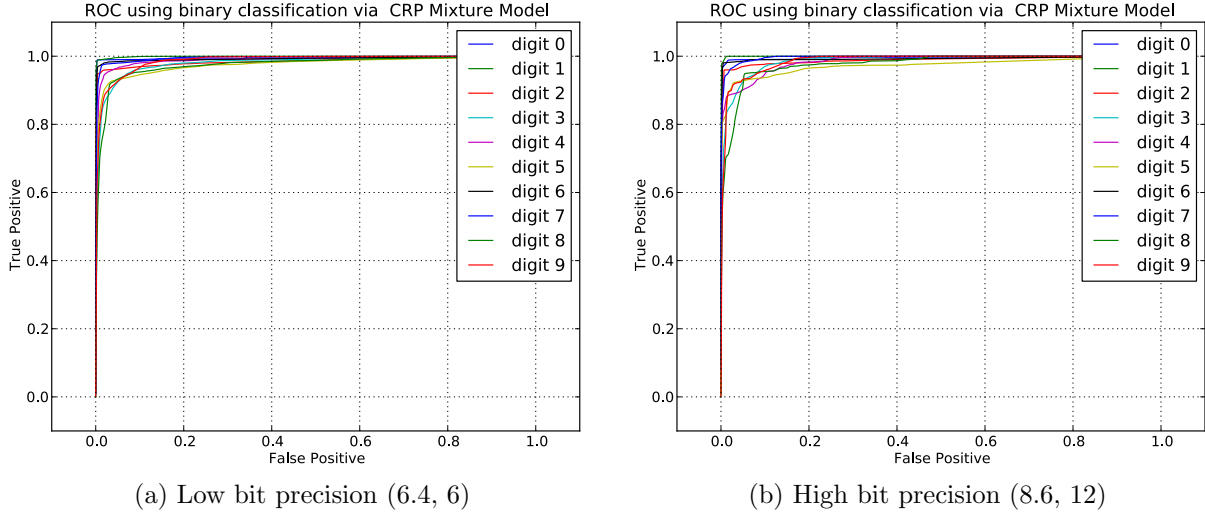


Figure 48: ROC curves for posterior predictive-based classification of test digits from the MNIST dataset for two different circuit bit precision. Classification becomes more perfect as the line gets closer to the upper-left axes.

CRP interface let's us evaluate the probability of cluster assignment for any new test row. We then compute in-class vs out-of-class ROC curves for each of 1000 test rows, taking 100 samples per row.

The ROC curves show good performance in this prediction task (figure 48, and highlight the expected challenges in disambiguating simialr digits (such as 3 and 8).

precision			Digits									
m	n	q	0	1	2	3	4	5	6	7	8	9
6	4	6	0.993	0.996	0.985	0.979	0.988	0.976	0.991	0.992	0.974	0.985
6	4	8	0.994	0.996	0.988	0.983	0.984	0.976	0.993	0.993	0.974	0.982
6	4	10	0.995	0.996	0.988	0.972	0.982	0.980	0.993	0.994	0.966	0.984
6	6	8	0.994	0.996	0.987	0.980	0.987	0.973	0.993	0.994	0.972	0.983
6	6	12	0.995	0.995	0.988	0.984	0.989	0.977	0.992	0.994	0.973	0.985
8	4	8	0.994	0.996	0.986	0.977	0.988	0.976	0.993	0.993	0.977	0.986
8	4	10	0.994	0.995	0.990	0.980	0.983	0.974	0.993	0.994	0.972	0.980
8	4	12	0.994	0.996	0.990	0.972	0.988	0.977	0.992	0.995	0.974	0.982
8	6	8	0.994	0.997	0.988	0.984	0.988	0.981	0.994	0.994	0.981	0.987
8	6	10	0.995	0.995	0.989	0.978	0.986	0.976	0.994	0.995	0.972	0.986
8	6	12	0.995	0.997	0.988	0.984	0.983	0.976	0.994	0.994	0.975	0.985

Table 8: Area under the curve for the inclass-outclass ROCs.

19 Future Directions

We have constructed a stochastic circuit with dynamic structure for Dirichlet-process mixture model clustering, along the way showing substantial performance gains even in spite of fairly extreme arithmetic functional approximation.

19.1 Architectural Improvements

The circuit presented here can potentially expand to models with thousands of features – the only limit as currently constructed is the depth of the pipelined adder-tree. We have also seen that conditional

independence gives rise to opportunities for parallelism, key to the efficiency advantages enjoyed by this circuit. Here, the features are conditionally independent, and thus we can score them in parallel with minimal overhead.

We have not gone as far as possible in exploiting conditional independence in this model: the posterior predictive $P(c_i = k|y_i)$ for a given group is conditionally independent of all other groups. This would allow computation $P(c_i = k)$ for all k in parallel as well, giving us an architecture whose parallelism would scale with the underlying *latent conditional independence* of the data.

19.2 Model and Inference

While we focus on the Beta-Bernoulli conjugate model class, we can obviously extend the circuits to other conjugate models, such as the Normal-Inverse-Gamma model for real-valued data, by replacing the SSMutate and PredScore modules. Heterogeneous collections of features are also possible.

Right now, our inference scheme requires externally setting the hyperparameters. We’ve determined internally (unpublished) that hyperparameter inference is often the key to extracting best-in-class performance from probabilistic models, and can even accelerate the mixing times of the underlying Markov chains. It would be reasonable to add additional state-controller logic to implement various forms of hyperparameter inference, such as slice sampling for both the CRP and per-feature hyperparameters. As we mentioned above, it would also be possible to incorporate this circuit as part of a larger, more complex probabilistic model.

Automatic Circuit Construction via a Compiler

One stringest test of putative composition and abstraction laws — or primitives and design rules more generally — is whether they can be exploited to produce useful designs mechanically. This section describes our approach to doing so. We have built a system that uses our abstraction and composition laws to effect the automatic transformation (compilation) from a high-level description of a probabilistic problem, such as a Bayes net, to a synthesizable circuit.

Here we present a compiler for discrete-state factor graphs. At a high level, this compiler takes two inputs, a factor graph and a list of variables to perform inference on, and generates an optimized circuit for inference. This compiler is capable of transforming arbitrary-topology discrete-state factor graphs into synthesizable densely-parallel circuits capable of performing inference at millions of samples per second. The compiler automatically identifies the conditional independence structure in the model to exploit opportunities for parallelism.

We then compile three example probabilistic models. First we compile the classic pedagogical “rain” model, showing how even at ridiculously-low bit precision the resulting circuit closely approximates the results obtained by exact marginalization. We then turn our attention to the much larger, highly bimodal undirected Ising model from statistical physics. We compile multiple Ising models, at varying coupling strengths, and recover the correct qualitative behavior. We then show compilation of a real-world Bayes network, ALARM, for causal medical diagnosis, and show how we can programmatically pick at compilation time which subset of the variables are fixed.

20 What can we compile

Our compiler supports arbitrary-topology factor graphs with discrete-valued state variables. The potentials must be representable as conditional probability tables (figure 49). Discrete-state factor graphs were the first class of probabilistic models supported by Kevin Murphy’s excellent Bayesian Network Toolbox (Murphy 2001), and can be applied to a wide range of problem domains.

Discrete-state factor graphs also provide ample opportunity to explore the viability of automatic parallelization. Our compiler creates a dedicated stochastic circuit element for each random variable, a choice that allows for maximal parallelization at the expense of consuming greater silicon resources. Only limited silicon resources constrain the number of random variables (size of the factor graph) we can support at the moment.

The resulting circuit performs massively-parallel Gibbs sampling (??) on the resulting graph. Gibbs sampling is viable in discrete-state factor graphs as exact sampling from $p(x_i|x_{-i})$ is easy – simply tabulate the scores for each possible setting of x_i and then exactly sample from the resulting table.

The dynamic nature of the compiler makes targeting a reconfigurable platform like FPGAs a natural fit, although all of the generated HDL is synthesizable for ASIC targets. Most of the performance numbers in this chapter are generated by targeting a Xilinx Virtex-6 LX240T FPGA, unless otherwise indicated.

20.1 Discrete-output CPT-sampling gate

Ultimately, all sampling units for all nodes are compiled down into discrete-output conditional probability gates, described in section???. Neighboring variables are connected to the input lines of the CPT gate, and output samples are generated conditioned on these values.

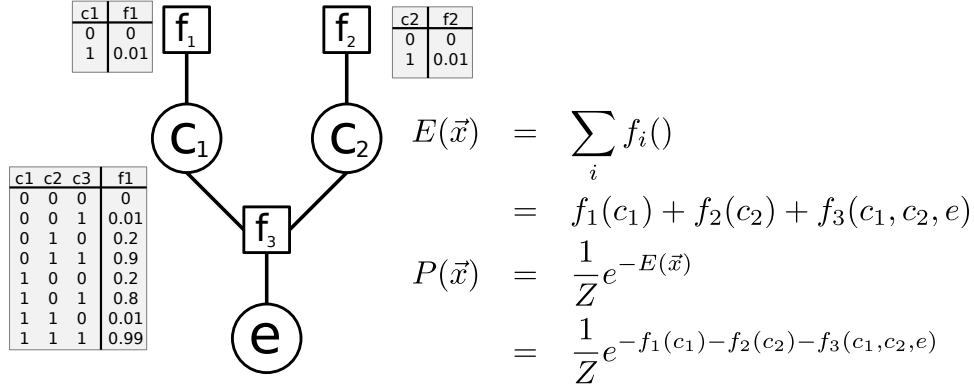


Figure 49: Discrete-state factor graph with factors expressed as conditional probability tables (CPTs). The total energy of the model is $E(\mathbf{x})$ summing over all factors, and the probability of any particular state is $\frac{1}{Z} e^{-E(\mathbf{x})}$.

Listing 1: Code to express a simple 3-node chain factor graph. Factor is defined in line 1, variables are created in lines 9-11, and the factors are wired up in lines 13-14

```

1 def factor(x1, x2):
2     if x1 == x2:
3         return 0
4     else:
5         return 16
6
7 fg = fglib.FactorGraph()
8
9 v1 = fg.add_variable((0, 3))
10 v2 = fg.add_variable((0, 7))
11 v3 = fg.add_variable((0, 3))
12
13 fg.add_factor(factor, [v1, v2])
14 fg.add_factor(factor, [v2, v3])

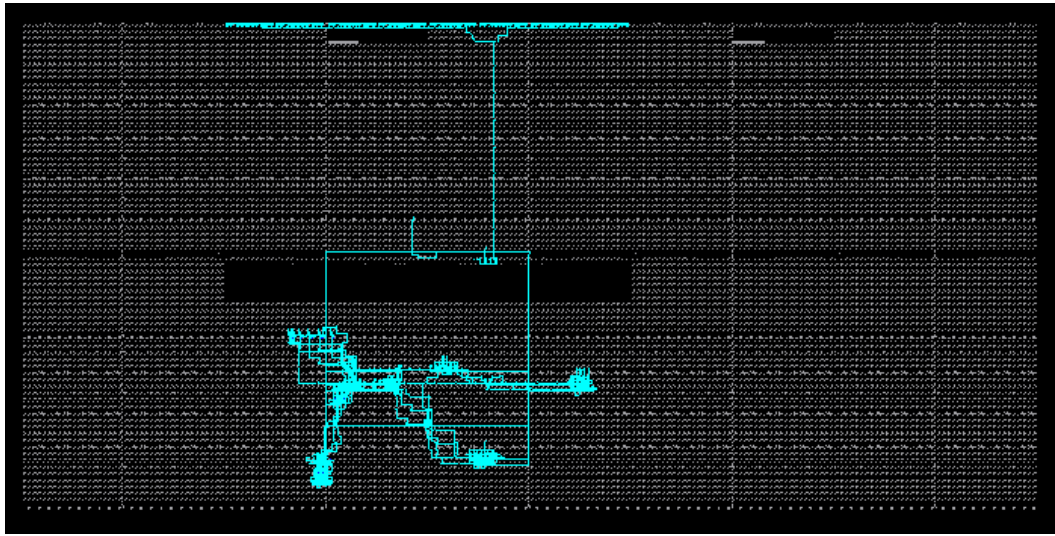
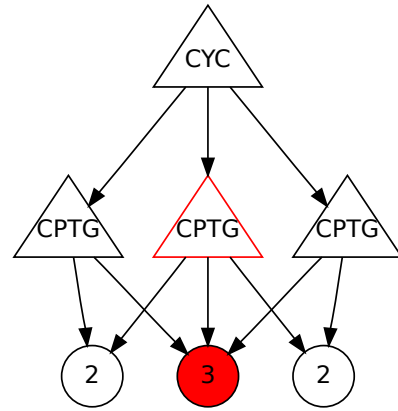
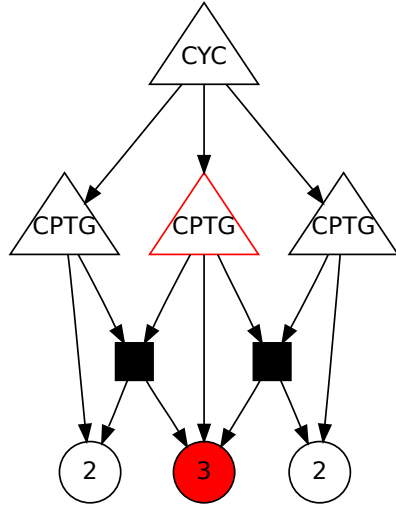
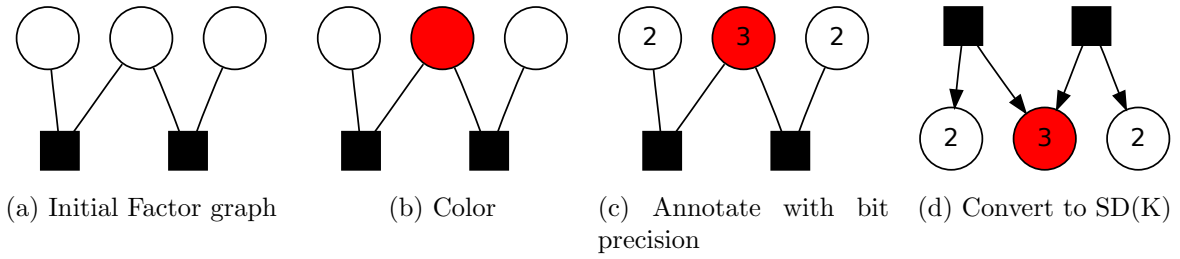
```

21 The compiler passes

The compiler begins with a factor graph description in Python, where a simple graph library allows a user to construct the graph by specifying variables, factors, and their topology. Listing 1 shows the construction of a simple three-variable two-factor graph. Variables are created in the graph and a handle is returned for further manipulation; the user specifies the (inclusive) range of possible values for the variable. Each variable can also be created as “observed”, which causes the compiler to *not* target this variable for inference. Observed variables are data – measurements about the world that we wish to condition on.

Factors are specified as python functions that return an energy (larger values are less likely). Note that the functions can perform arbitrary computation, as they are only evaluated in the course of compilation, generating a lookup table for later synthesis.

The compilation steps are as follows. We color the initial factor graph to identify parallelization opportunities. Nodes of the same color are conditionally independent, and thus we can do inference on them simultaneously. We then annotate the variables with the number of bits necessary to represent them, derived from their user-supplied range information.



(g) FPGA Result

Figure 50: Compilation passes. a.) shows the original factor graph, which we perform graph coloring on (b.) to identify conditionally-independent random variables that are amenable to simultaneous sampling. We (c.) convert to an SDK and add the sampling kernels (d.) and then compile away the densities (e.) leaving a collection of interconnected CPT gates. (f.) shows the visualized netlist in hardware.

We then convert the factor graph into a form (Bonawitz 2008) which explicitly represents the variables as states, the factors as densities, and includes the stochastic FSMs doing inference (the kernels). This “State, Density, Kernel” (SDK) form allows reasoning about the precise flow of inference and kernel structure. In the SDKs pictured, circles are state variables, squares are densities which score those state variables, and triangles are the kernels which perform mutation and control other kernels.

Taking the simple SDK from before, we annotate it with the “kernels” that will ultimately be performing inference. The primary kernel used is an “enumerated Gibbs” kernel which will perform Gibbs sampling of a particular target node. State variables labeled “observed” do not have kernels attached.

A kernel inherits the coloring of its target state variable. Thus all kernels of a given color can be executed simultaneously. All of the Gibbs kernels are driven by a single master “mixture” kernel at the root of the SDK. The mixture kernel randomly selects one color of kernel to execute at a time, effectively implementing “random scan gibbs” as described by (Liu 2008).

We then perform a graph transform on the SDK to compile away the densities, and replace the resulting kernels with CPT gates as follows (the densities are of course the original factors in the factor graph). For a given target node we:

- Consider all possible state values of the state nodes in the target node’s Markov blanket, and build up a giant lookup table mapping from the possible input state space to the distributions on the output state.
- We use this table to create a Conditional Probability Table Gate (as described in section ?? with its conditioning inputs as the neighboring state variables.
- The resulting CPT Gate is a SDK kernel – it’s a stochastic unit which mutates the value of the target variable and preserves the total ergodic distribution of the markov chain.

The compiler then simply wires up these CPT gates and connects their enable lines appropriately.

22 Performance

A kernel for a state with k possible values will take $k + o$ cycles to sample a new value, where o is the overhead associated with handshaking. When we tell all the CPTs for a given graph color to “sample”, we schedule for worst-case performance. If k_{MAX} is the maximum number of possible values for a state variable, *all* kernels are given $k_{MAX} + o$ cycles to complete. In practice, this has limited impact on performance, for two reasons:

- the airities we’re working with are generally small – 2, 3, 4 possible states
- Since all the nodes for a given color are sampled in parallel, we can only move on when the last of these is done sampling. Even if the time to sample is $E[k/2] + o$, it’s likely that at least one kernel will need the full $k + o$ cycles, stalling the completion of the cycle.

22.1 IO and entropy

We enable programmatic IO with the resulting compiled circuits by chaining all the state variables together in a single long shift register, akin to JTAG. The shift register is latched to allow inference to continue to occur during the readout. Compilation metadata is saved post-compilation to allow readout from python to match the factor graph node labels in the original source code.

Entropy is provided to each kernel via an associated XORshift RNG (see 4) which is given a unique seed at compile time. Note that this is a dramatically-inefficient use of entropy – we are using roughly one-thousandth of the entropy provided by each PRNG. it is possible to multiplex the output of the PRNGs to share them between different subsets of CPT gates and save silicon.

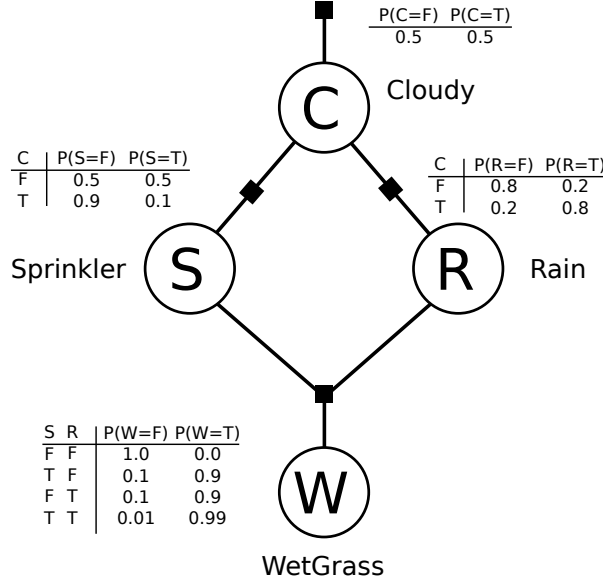


Figure 51: “rain” factor graph.

23 Example Models

We present three compiled models. The classic Rain example Bayes network is a careful walk-through of the possible queries we can make on such a Bayes net, and shows how even at very low bit precision we recover the correct answers. The Ising model from statistical physics demonstrates massively parallel execution of a very large model. We conclude with the ALARM causal medical diagnosis network, highlighting how compilation can enable different subsets of nodes to be “observed”, and thus conditioned on.

23.1 Rain

We adopt Kevin Murphy’s (Murphy 2001) modification of the classic “Rain” example from Artificial Intelligence, A Modern Approach (Russell and Norvig 2009) as our initial model. The Bayesian Network originally presented has four boolean nodes: cloudy (C), rain (R), sprinkler (S), and wet grass (G). When it’s cloudy, it’s more likely to rain, and you’re less likely to turn on the sprinkler. Both the sprinkler and rain can cause the grass to be wet. We can trivially convert this Bayes net into a factor graph (figure 51) and describe it efficiently in Python (listing 2).

We compile the network at three different bit precisions and generate 10,000 samples of the full joint distribution, $P(C, S, W, R)$, and use those samples to answer queries. We compare our empirical results with exact results obtained via belief propagation. Based on table 9, we see that even at 5 bits, we very accurately recover posterior values for queries. Merely 5 bits are enough to accurately encode the resulting joint distribution and efficiently sample from it.

The queries are as follows (see table 9):

1. $P(C)$: Probability of cloudy (this probability is explicitly coded in a factor, so this serves as a sanity check)
2. $P(S|W)$: Given that the grass is wet, what is the probability the sprinkler was on?
3. $P(S|W, R)$: Given that the grass is wet and it is raining, what is the probability that the sprinkler is on? Because of the rain, the posterior probability of the sprinkler being on goes down.

Listing 2: Source code for rain factor graph, defining the three potentials and wiring up the graph

```

fg = fglib.FactorGraph()

cloudy = fg.add_variable((0, 1), observed=False)
sprinkler = fg.add_variable((0, 1), observed=False)
rain = fg.add_variable((0, 1), observed=False)
wet_grass = fg.add_variable((0, 1), observed=False)

assignments = {'cloudy' : cloudy,
               'sprinkler' : sprinkler,
               'rain' : rain,
               'wet_grass' : wet_grass}

def sprinkler_pot(cloud, sp):
    if cloud:
        if sp: return to_energy(0.1)
        return to_energy(0.9)
    else:
        return to_energy(0.5)

fg.add_factor(sprinkler_pot, [cloudy, sprinkler])

def rain_pot(cloud, ra):
    if cloud:
        if ra: return to_energy(0.8)
        return to_energy(0.2)
    else:
        if ra: return to_energy(0.2)
        return to_energy(0.8)

fg.add_factor(rain_pot, [cloudy, rain])

def grass_pot(sp, ra, wg):
    if sp and ra:
        if wg: return to_energy(0.99)
        return to_energy(0.01)
    if sp == 0 and ra == 0:
        if wg: return to_energy(0.0001)
        return to_energy(0.9999)

    # else:
    if wg: return to_energy(0.9)
    return to_energy(0.1)

fg.add_factor(grass_pot, (sprinkler, rain, wet_grass))

```

Query	BP	5-bits	8-bits	12-bits
$P(C)$	0.5	0.4855	0.5065	0.4983
$P(S W)$	0.4298	0.4535	0.4320	0.4309
$P(S W, R)$	0.1945	0.2160	0.2045	0.1935

Table 9: Rain Factor Graph. Measured values for various bit-precisions of rain model

23.2 Ising Model

The Ising model (Ising 1925) is a probabilistic model of ferromagnetism in statistical mechanics, and is frequently used as a benchmark model for probabilistic methods due to its extremely bimodal nature. The model consists of binary variables which represent the spins of magnetic domains. Each spin can be either “up” or “down”, and only interacts with its nearest neighbors.

Adjacent spin variables contribute to the total model energy only when they have different values; that is, an “up” variable next to a “down” variable is a higher-energy state than two “up” or two “down” juxtaposed variables. J controls the magnitude of the difference between these two energy states. In statistical mechanics, higher-energy configurations are less probable – nature seeks out lower-energy states.

The factors are thus all homogenous, and of the form

$$f(x, x') = \begin{cases} 0 & \text{if } x = x' \\ J & \text{if } x \neq x' \end{cases} \quad (32)$$

The energy of the total ising system is thus

$$E(X) = \sum_{x, x' \in N(x)} f(x, x') \quad (33)$$

where $x' \in N(x)$ is the set of all nodes that are adjacent to x .

We compile nine different 256-node Ising factor graphs, systematically varying the coupling strength J from 0.5 to 1.4. Figure 52 shows both the evolution of the sampler over time, as well as the resulting histogram of the number of “up” vs “down” states. When the coupling strength is very low, each binary variable is effectively independent, and as we expect the sum of states histogram looks roughly Gaussian. As the coupling strength increases, bimodality emerges, with the “all up” and “all down” configurations being dramatically preferred.

We’re thus able to compile large factor graphs and perform efficient probabilistic inference programmatically. The programmatic nature of the compiler has the benefit of making it easy to explore different points in the parameter space.

23.3 ALARM

ALARM (“A Logical Alarm Reduction Mechanism”, (Beinlich et al. 1989)) is a Bayesian network for patient monitoring, encoding the probabilities of a differential diagnosis with 8 possible diagnoses based on 16 measurements.

Alarm diagnoses are mutually exclusive, but not encoded as such. Measurements are often continuous, but for the purpose of the network they are encoded categorically, e.g. “low, normal, high”. The network also makes inferences on 13 intermediate nodes, connecting diagnoses to measurements.

We go from the original Bayes net (figure 54) to a factor-graph representation (figure 55) which we then compile with 12-bit precision. We compile two different target networks: One with the diagnoses observed, and one with the measurements observed.

Compilation with the diagnoses observed lets us understand the relationship between diseases and evidence. As seen in figure 57, a healthy person has the majority of measurements in the “normal” column, although for some variables (such as Total Peripheral Resistance, TPR), there is a roughly uniform distribution on measurements. Hypovolemia, pulmonary embolism, and left ventricular failure create different symptom profiles.

Compilation with the measurements observed allows us to use the network as it might be in a clinical setting – measurements are made and diagnoses are suggested. When all measurements are in the “normal” range (figure 58, no diagnosis is suggested. Particular settings of measurements adjust the probabilities of particular symptoms.

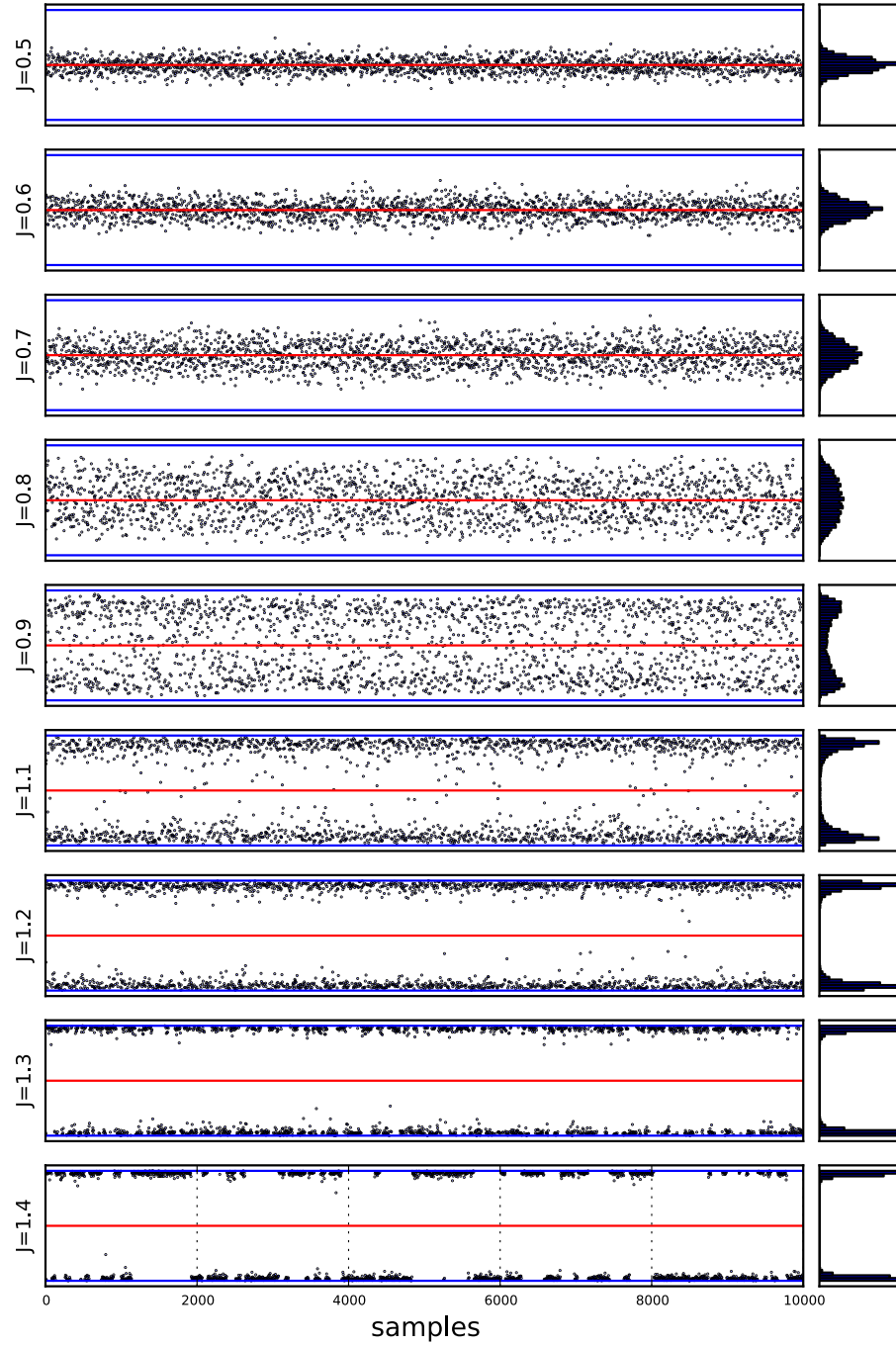


Figure 52: Samples from 8-bit 256-node Ising circuits with different coupling strengths. As the coupling strength increases, the distribution becomes more bimodal.

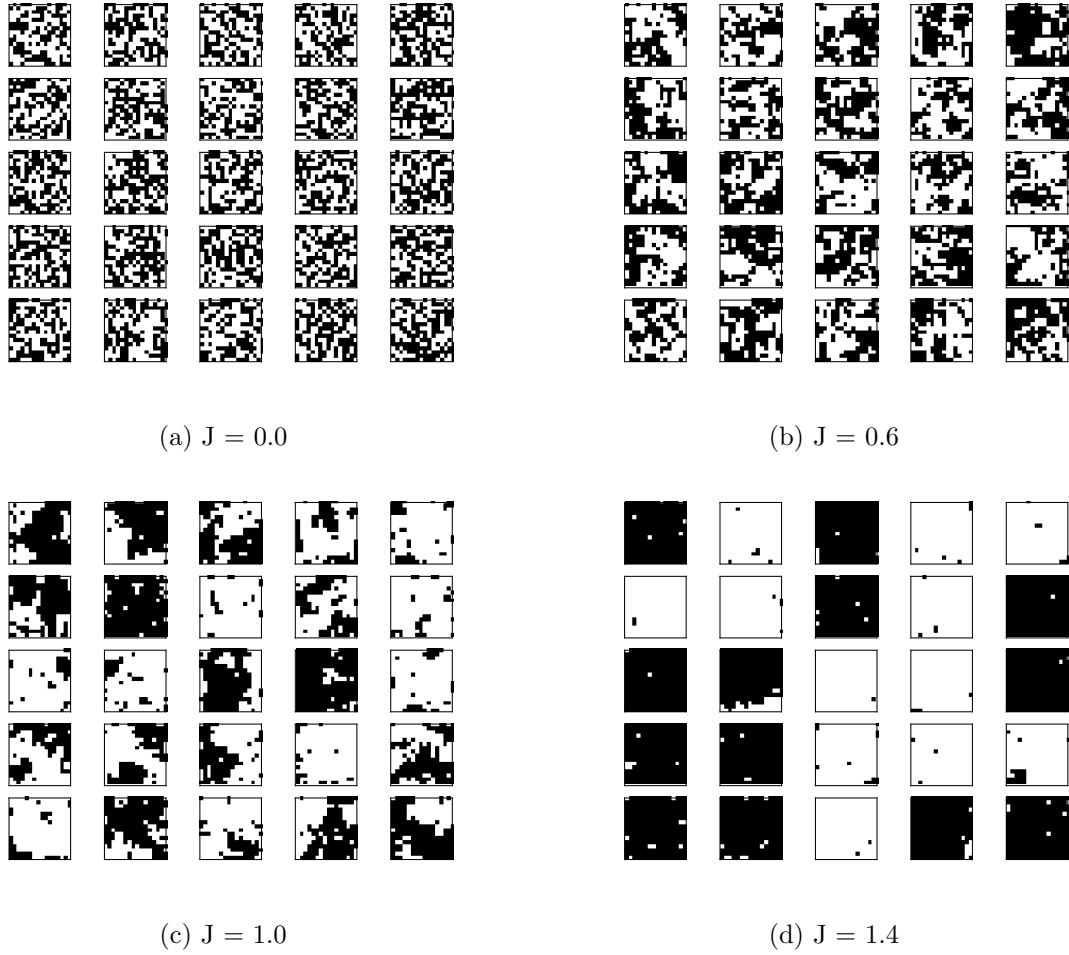


Figure 53: Example samples from the compiled Ising model for four different coupling strengths.

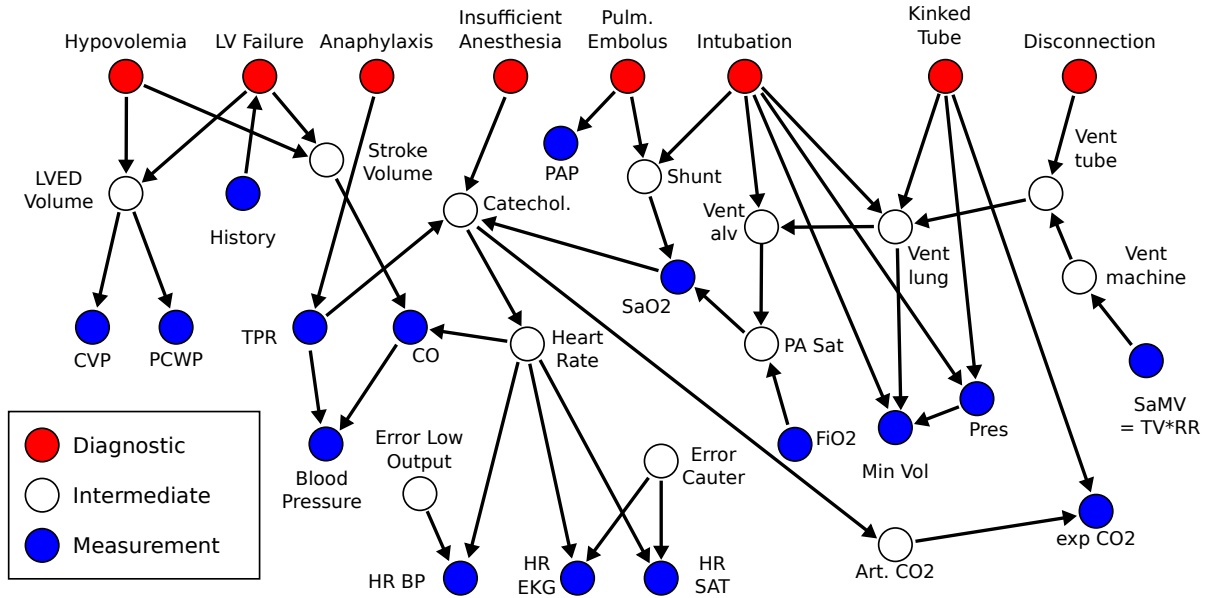


Figure 54: The ALARM (A Logical Alarm Reduction Mechanism) Network, with 8 diagnoses, 16 findings, and 13 intermediate variables.

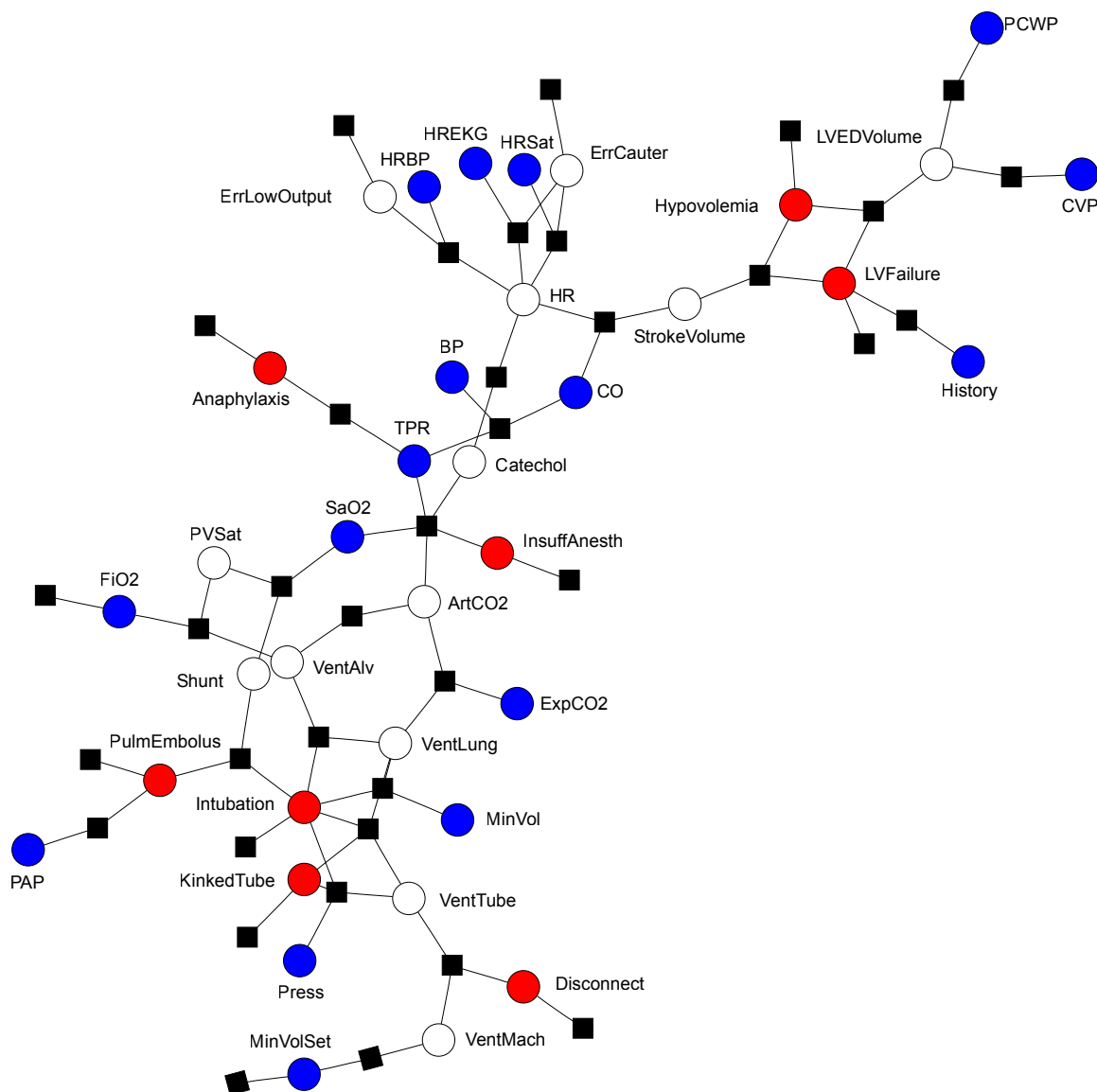


Figure 55: The ALARM Factor Graph generated from the Alarm Bayes Network.

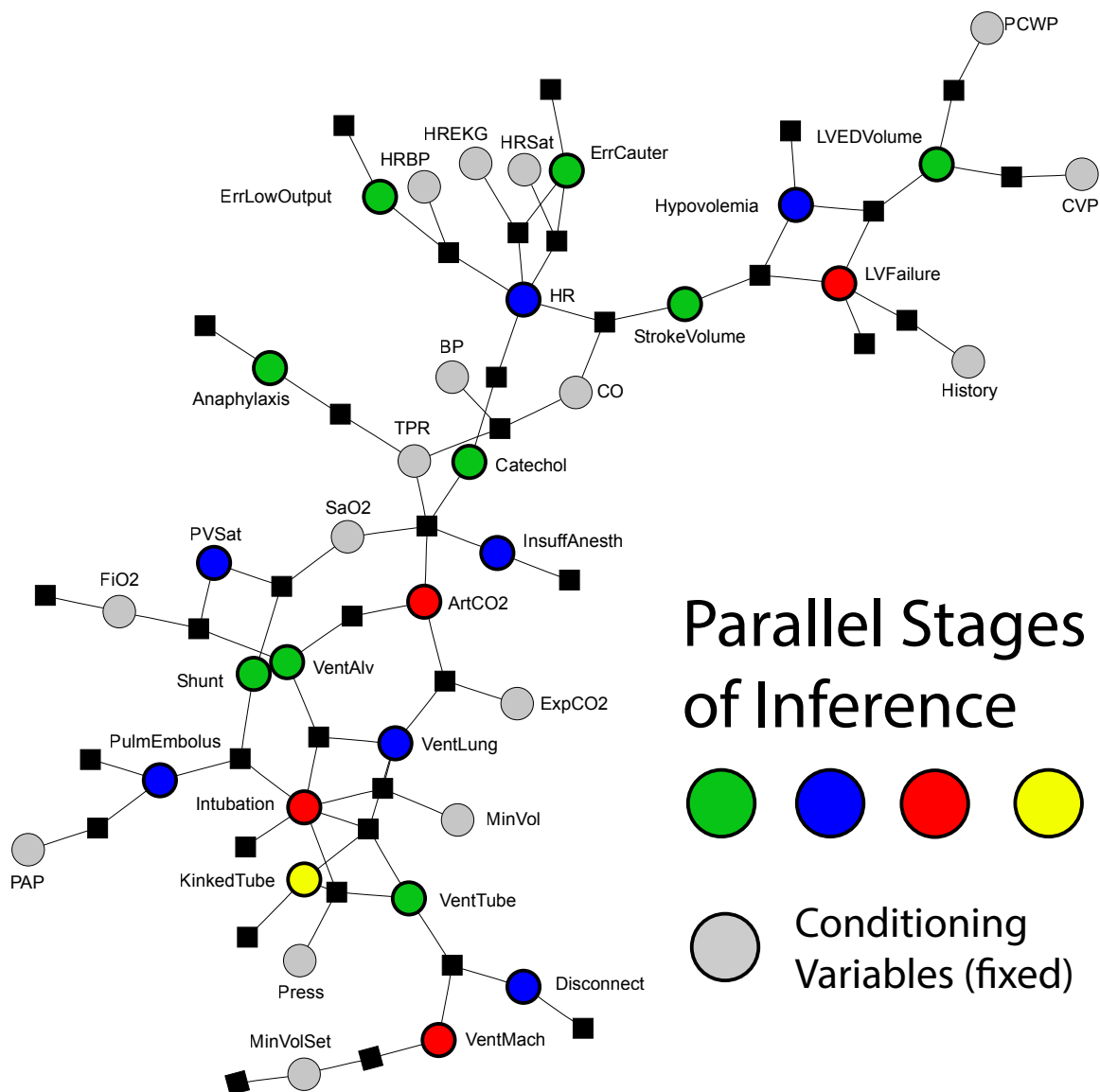


Figure 56: The ALARM factor graph from figure 55 colored for parallel execution.

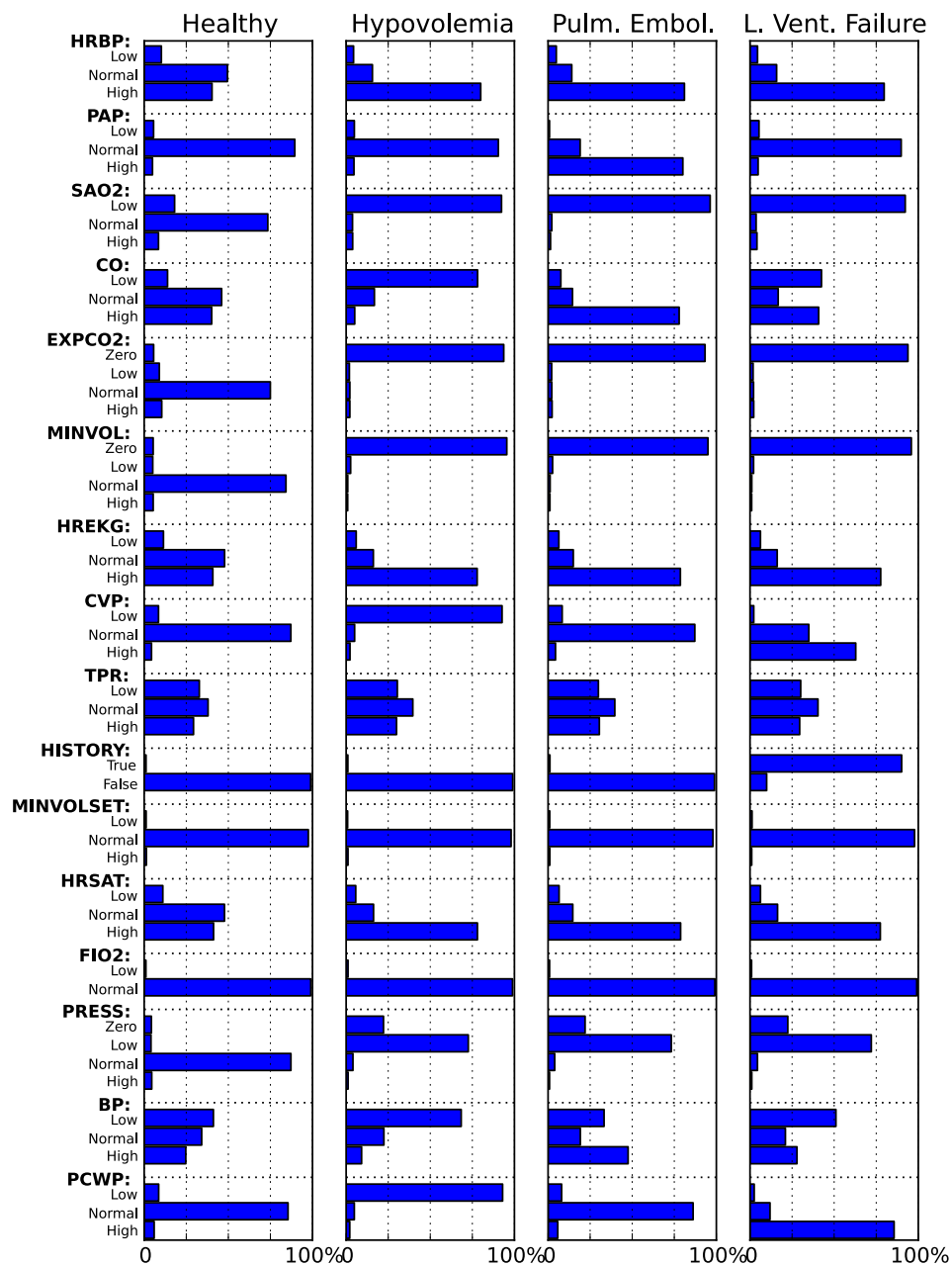


Figure 57: The ALARM Network: Marginal distributions for measured (symptom) variables given particular diseases.

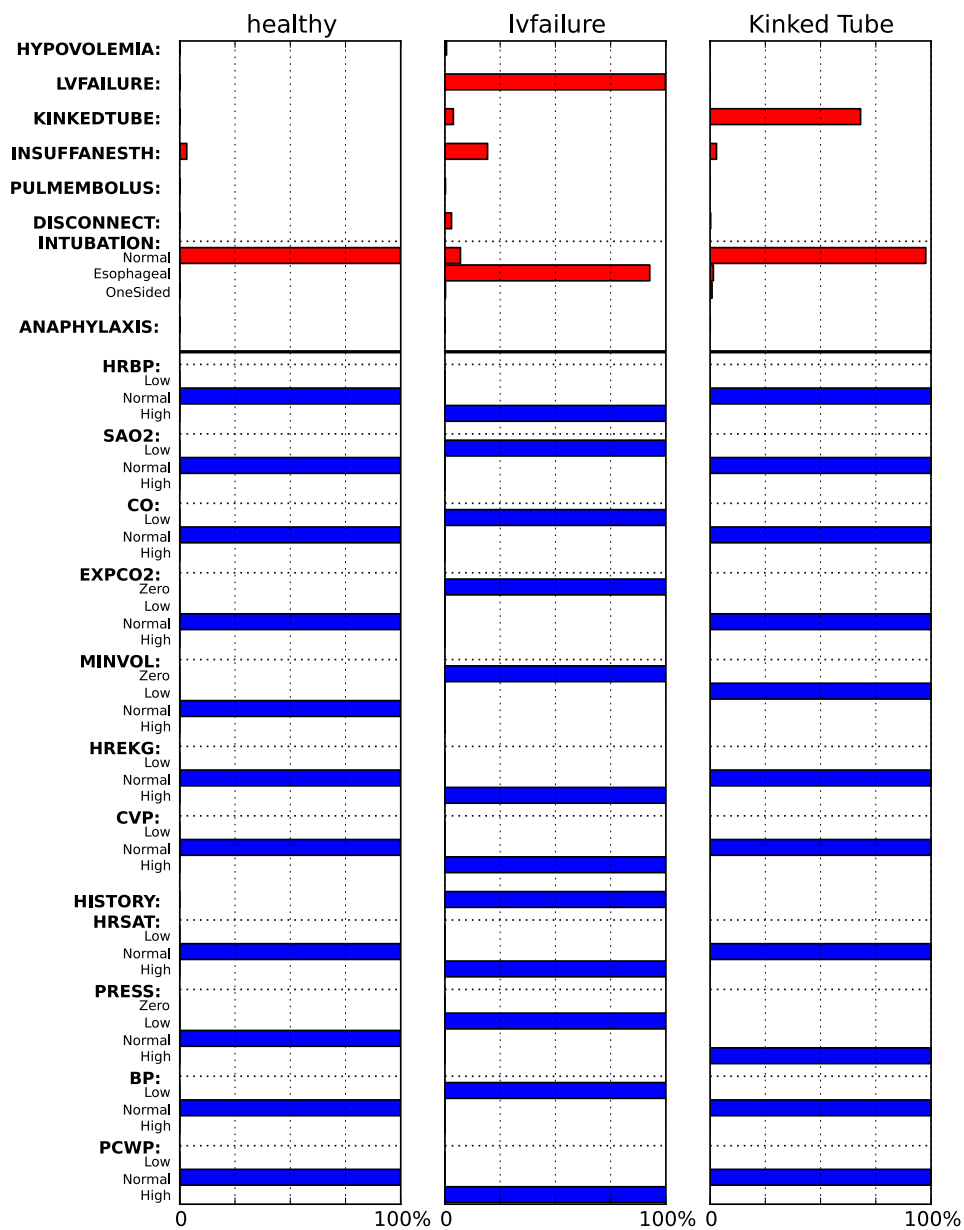


Figure 58: The ALARM Network: Marginal distributions for diagnoses based upon particular settings of the measured symptoms.

24 Future Directions

Causal reasoning is a crucial capability in systems that try and make sense of the noisy data they observe in the world, and can be modeled as a Bayes net. Here we have shown how a generalization of Bayes nets, discrete-state factor graphs, can be programmatically compiled into stochastic circuits. The compiler we built takes compact descriptions of factor graphs in python, and generates synthesizable RTL. The resulting circuits enable rapid inference, allowing for posterior exploration across a wide range of models.

We’ve also shown how having a compiler allows for the automatic exploration of a variety of models and parameters in the problem space. With the rise of probabilistic programming languages, one can imagine a day when arbitrary probabilistic programs can be compiled down to efficient circuits.

Right now, the proof-of-concept compiler we’ve built leaves open the option for many performance optimizations. As our timing is all based on the worst-case time of the slowest sampler in a particular graph subset, future versions can adopt better handshaking to achieve closer-to-optimal runtime. Right now we give each stochastic gate its own PRNG, a waste of resources that could easily be ameliorated by multiplexing the output of the RNGs.

Any of our stochastic gates could be incorporated into the compilation step, as fully blowing out the CPT table for every state variable tends to be somewhat space-inefficient. More compact special-purpose gates would enable much larger graphs. Similarly, we could create more runtime-configurable gates, allowing for greater runtime flexibility in the underlying model. The next chapter suggests a particular scheme for run-time reconfiguration, *virtualization*, that allows for much larger graphs, albeit with reduced flexibility in topology.

Spiking Implementations of Stochastic Digital Circuits

To implement stochastic digital circuits using spiking elements with Poisson firing statistics, we will leverage two basic identities of the Poisson process:

Proposition 1 *Let X_1 and X_2 be exponentially distributed random variables with rates λ_1 and λ_2 respectively. Then $P(X_1 < X_2) = \frac{\lambda_1}{\lambda_1 + \lambda_2}$.*

Proposition 2 *Let X_1 and X_2 be exponentially distributed random variables with rates λ_1 and λ_2 respectively. Then $Z = \min(X_1, X_2)$ has an exponential distribution with rate $\lambda_Z = \lambda_1 + \lambda_2$.*

The proofs are elementary³. Given these identities, if we have a discrete random variable X with K possible outcomes where $Pr[X = k] = \frac{\lambda_k}{\lambda_Z}$ for $k < K$, we can simulate from X by running K Poisson processes and detecting which one spiked first. We can thus implement a DISCRETE-SAMPLE gate by exponentiating each energy e_i to obtain a corresponding unnormalized probability λ_i and using a bank of Poisson processes. The crucial observation is that this is the core building block for the space-parallel implementation of the Gibbs sampling transition circuit we describe earlier.

It is straightforward to make a semi-synchronous spiking network out of these elements. The circuit we simulate, over binary random variables A , B and C , can be viewed as a factor graph with the following potentials:

$$F_A = \{ (0) : 2, \\ (1) : 1 \}$$

$$F_{AB} = \{ (0, 0) : 1, \\ (0, 1) : 2, \\ (1, 0) : 1, \\ (1, 1) : 1 \}$$

$$F_{AC} = F_{AB}$$

This results in spiking neurons A_0 and A_1 for variable A , B_0 and B_1 for variable B , and C_0 and C_1 for variable C . We simulate the network semi-synchronously, in accord with the dynamic discipline we rely on for stochastic transition circuits: local simulation is done asynchronously, with fast lateral inhibition between X_0 and X_1 , but A_0 and A_1 are suppressed while B and C are updating, and vice versa. We leave a treatment of fully asynchronous networks and clocking schemes to future work.

³See <http://www.columbia.edu/~ks20/stochastic-I/stochastic-I-PP.pdf> for a typical presentation