Stochastic Architectures for Probabilistic Computation

by

Eric Jonas

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Neuroscience

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

Author ....

Department of Brain and Cognitive Sciences
September 13, 2013

Certified by ....

Joshua B. Tenenbaum
Professor of Computational Cognitive Science
Thesis Supervisor

Accepted by ....

Matthew A. Wilson
Sherman Fairchild Professor of Neuroscience and Picower Scholar
Director of Graduate Education for Brain and Cognitive Sciences

Stochastic Architectures for Probabilistic Computation

by

Eric Jonas

Submitted to the Department of Brain and Cognitive Sciences
on September 13, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Neuroscience

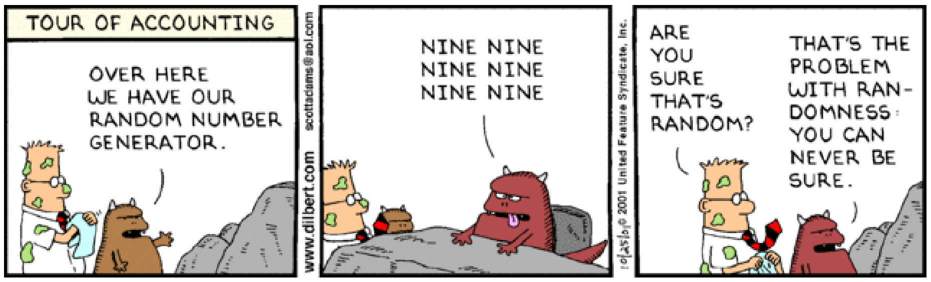
Abstract

The brain interprets ambiguous sensory information faster and more reliably than modern computers, using neurons that are slower and less reliable than logic gates. But Bayesian inference, which is at the heart of many models for sensory information processing and cognition, as well as many machine intelligence systems, appears computationally challenging, even given modern transistor speeds and energy budgets. The computational principles and structures needed to narrow this gap are unknown. Here I show how to build fast Bayesian computing machines using intentionally stochastic, digital parts, narrowing this efficiency gap by multiple orders of magnitude.

By connecting stochastic digital components according to simple mathematical rules, it is possible to rapidly, reliably and accurately solve many Bayesian inference problems using massively parallel, low precision circuits. I show that our circuits can solve problems of depth and motion perception, perceptual learning and causal reasoning via inference over 10,000+ latent variables in real time — a 1,000x speed advantage over commodity microprocessors – by exploiting stochasticity.

I will show how this natively stochastic approach follows naturally from the probability algebra, giving rise to easy-to-understand rules for abstraction and composition. I have developed a compiler that automatically generate circuits for a wide variety of problems fixed-structure problems. I then present stochastic computing architectures for models that are viable even when constrained by silicon area and dynamic creation and destruction of random variables. These results thus expose a new role for randomness and Bayesian inference in the engineering and reverse-engineering of computing machines.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor of Computational Cognitive Science


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com
scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

DILBERT: Scott Adams/Dist. by United Feature Syndicate, Inc.

Acknowledgments

Fourteen years is a long time to be a student, especially at a single institution. It's only a slight exaggeration to say that I spent the first half of my life wanting to go to MIT, and the second half living that dream. The successful defense of this thesis represents the completion of my career as an MIT student, but that career has only been possible through the tremendous support of others.

I only have one regret of my (lengthy) trajectory – it took took so long that my father is not here to witness this final victory. He never wavered in his encouragement of my curiosity and his support of my (sometimes dangerous) engineering pursuits. As a teenager, most of our conflicts were proxy battles around figuring out how to pay for MIT. On December 11, 1998 when I received my admissions letter, his first words were "we'll find a way." We found a way, Dad.

Above all this is a scientific work, and as a scientific work it would not have existed were it not for Vikash. Our collaboration over the past half-decade has been the most productive of my academic career. When he approached me in 2007 with the idea of embracing, not exterminating, stochasticity in our lives, I thought he was mad. Now I just think he was right. I know that his future students have a great deal to look forward to.

This trajectory was a little unorthodox – Vikash, Keith, Beau and I started Navia in 2007 and eventually ended up in Silicon Valley with the dream of building a probabilistic empire built on probabilistic hardware and software. It is only through their support – intellectual, emotional, and entrepreneurial – that this document exists. Credit must of course be given to Peter Thiel's venture capital group, Founders Fund, and their decision to invest in what would ultimately be a scientific success but a commercial failure. Their appetite for "science risk" brought the future a bit closer to the present, and gave me tremendous flexibility as a cofounder to work on this thesis.

Out of the ashes of Navia rose Prior Knowledge, and along with it the incredible support of Vimal, Cap, Max, and (again) Beau. The wild sixteen month ride of building a substantially-more commercially focused probabilistic enterprise was quite emotional and exhausting. During that time, progress on this work slowed to a standstill, and I was certain I would never graduate. Our acquisition by Salesforce.com was an incredibly exciting outcome for everyone involved, and relevant to this document, Salesforce has been amazing in supporting my renewed work on this thesis. Special thanks to Keith, Fayyaz, Joe, and Villi for greasing the wheels and clearing the IP issues necessary for this to happen.

My advisors, Matt Wilson and Josh Tenenbaum, both deserve immeasurable credit for putting up with me. I often felt like quite the black sheep in BCS – an engineer in scientists' clothing. Matt supported me year after year, even as I consumed more and more lab resources with no obvious benefit in sight, motivated by my own curiosity and engineering passion. As my research interests drifted, Josh was immeasurably generous in taking up the mantle of advisor, and I consider this work (as well as Navia and P(K)) to owe him an impossible scientific debt.

Practically, BCS can be a challenge to navigate, and I would certainly not be graduating were it not for the support, advocacy, and cat-herding performed by Denise. Stuart, Yarden, Tom, Jeenah, and Hubert made the experience – dare I say it – enjoyable at times, and proved that you can make and keep close friends in graduate school.

Finally, I can't overexpress the importance of Cap, Erica, and Vimal played during the past decade – as informal collaborators, scientific advisors, personal therapists, and family members. And rounding out this family, my sister Courtney, as well as Selena and of course Shelly. You have all helped me through my darkest moments, when I was sure I would never finish and all hope was lost. You all continue to motivate me to drag the future, kicking and screaming, into the present.

With all my thanks,

Eric Jonas
September, 2013

Anyone who considers arithmetical methods
of producing random digits is,
of course, in a state of sin.

JOHN VON NEUMANN

Contents

Chapter 1

Introduction

We live in a world of uncertainty, in which intelligent agents expend vast computational resources to understand, predict, and exploit their surroundings. We've long attempted to engineer computing systems that can accomplish the tasks that a four-year-old human excels at, and yet even in 2013 this dream still feels far off.

When IBM's question-and-answer system, Watson, successfully defeated the world's best Jeopardy players (Ferrucci et al. 2010), Watson was given the questions as ASCII text. Even after spending \$100 million, engineers at IBM were still unwilling to make their project dependent on voice recognition technology. Why are low-level perceptual tasks still so difficult for our computing systems?

1.1 Gordon Moore's Brain

Even with the relentless pursuit of Moore's law (G. E. Moore 1965), computing systems still come up short solving this class of perceptual problems. Yet the brain of Gordon Moore still, after 84 years, performs these tasks effortlessly using approximately 20 watts. The difference is staggering. Neural systems are massively parallel – the human brain has 10^{12} neurons with on average 1000 connections between them (Kandel, Schwartz, and Jessell 2000). Even our best multi-core GPUs today have at most 2688 cores (*Tesla Kepler GPU Accelerators* 2102). Neural systems are incredibly fault-tolerant, as anyone who has ever had a concussion or imbibed too much alcohol can attest. They are incredibly stochastic, and fundamentally slow – interneuron communication is limited to roughly 1 kbps. And yet Gordon Moore could understand this sentence spoken aloud, and Watson can't.

As neuroscientists, the question is, how do neurons in a brain do what supercomputers cannot? As machine learning researchers, how can we make those supercomputers do those tasks? And as computer architects, are there fundamentally better ways of building our computing machines to accomplish these goals?

1.2 The Impedance Mismatch

Stochastic algorithms are among the best-known solutions to many problems of optimization, estimation, and inference. Approximate inference methods based on drawing samples from probability distributions have impacted every field of applied mathematics (**Diaconis2008**). But even today, we run these broad stochastic algorithms on deterministic computing machines designed to *evaluate deterministic functions*, not sample from distributions.

On the hardware side, this determinism comes at a price – in terms of the millions of device-physicist man-years, the cost of new fabrication facilities, and the resulting yield problems. Long before we reach the quantum regime, classical silicon is quite stochastic (Shepard and Narayanan 1996). The smallest features in a contemporary $22nm$ process are only 50 silicon atoms across. And the view of "noise as the enemy" is taught at the youngest levels – undergrads learn about Johnson noise in resistive elements and treat it as something to be eliminated at all costs.

But what if there was a better way? What if we could eliminate the impedance mismatch between this rich class of stochastic algorithms and the underlying stochasticity of the substrates? Could we take inspiration from our neural systems, and build fundamentally stochastic circuits?

This thesis argues that there may be a better way – that by embracing, not mitigating, stochasticity as a design primitive, we can build machines that are fundamentally more brain-like, and that close the efficiency gap between neural and man-made computation.The secret lies in recognizing that the rules of probability, long known to be a generalization of traditional Boolean algebra, can guide the construction of these stochastic systems the same way Boolean algebra has for deterministic, digital systems.

We can go further – while computing systems are modeled formally on finite state machines evolving in time, computational statistics (Metropolis et al. 1953) has long recognized that Markov chains – stochastic state machines – can solve many of the most difficult challenges in computing. We will show that by taking this stochastic architecture discipline seriously and focusing on the construction of time-evolving Markov chains, we can further exploit parallelism and low bit precision to solve many problems of interest in machine learning.

1.2.1 Similar Approaches and where they are today

Engineers have been trying to build brain-like hardware since the 1980s (Lyon and Mead 1988). Neuromorphic approaches have spawned an entire cottage industry of engineers, computational neuroscientists, and ASIC designers building more brain-like computing systems. Yet, these systems never seem to have caught on, in spite of 30 years of work. We will review some of these efforts to build "neurons" in silicon, ranging from highly-abstract integrate-and-fire units to more biologically-plausible multicompartmental units. Yet none of these have had the impact that simple digital signal processors have achieved, when the latter simply offer hardware-based convolution and zero-overhead indexing (a far simpler engineering feat!).

We argue this lack of adoption is because these systems are fundamentally difficult to *engineer* with. They lack the sound engineering rules that we use effortlessly in all other areas of our computing stack. They are difficult to abstract, and they are difficult to glue together. Here we argue that the probability algebra provides such a set of engineering principles.

We are not talking about ways to build *deterministic* systems out of fundamentally stochastic, unreliable parts. While von Neumann explored this extensively at the circuit level in the middle part of the 20th century (Neumann 1956), his goal was to still make perfectly reliable systems with probabilistic version of finite automata.

We are not talking simply about accelerating random number generation. While there are many security applications that depend on cryptographically-secure sources of entropy, accelerators for this domain have long existed. Indeed, even Intel's latest Ivy Bridge architecture has dedicated hardware to produce cryptographically-strong random bitstreams (Taylor and Cox 2011).

And we are not simply seeking to accelerate a class of computation by hardware implementation. Various attempts have been made in the past to accelerate essentially deterministic algorithms for probabilistic inference, both using reconfigurable digital hardware (Lin, Lebedev, and Wawrzynek 2010) as well as silicon-level analog hardware (Vigoda 2003).

Instead we suggest that a transition towards a fundamentally more probabilistic model of computation, with stochasticity at the core, will let us reduce this impedance mismatch.

1.3 Organization of this thesis

The audience for this thesis is diverse. We hope electrical engineers and computer architects will find stochastic circuits to be a useful, coherent engineering framework for building more interesting machines. We hope machine learning researchers and computational statisticians will start to think how their data and inference challenges can be solved at the architectural level. And we hope those of us seeking to better understand the workings of actual neural systems will take the role of stochasticity *in computation* to heart.

This diversity in audience drives our organization. David Marr (David Marr 1982) suggested that any attempt to understand the operation of complex neural systems should begin by analyzing the system at three complementary levels:

1. **The Computational Level** : What are the functions that this system is trying to achieve?
2. **The Algorithmic Level** : What are the computational steps this system takes to achieve the computational challenges?
3. **The Hardware Level** : What are physical neural, biological, and chemical substrates on which this computation occurs?

To computer architects, this decomposition feels natural, and thus we attempt to organize the individual chapters along similar lines.

1. Computational Level : The inspiring computational challenge, from both cognitive and machine-learning perspectives.
2. Probabilistic Level : The underlying probabilistic model or model class we're using to address the computational challenge
3. Architectural Level : The ways that this probabilistic, stochastic approach impacts existing assumptions about circuit design and computer architecture

Chapter 2 reviews the basics of probability, Markov chains, and techniques for sampling from distributions. Exact sampling is contrasted with the use of the ergodic distributions of Markov chains, termed Markov chain Monte Carlo (MCMC), for drawing samples from intractable distributions. While a reader with a strong background in probability might be able to skip this section, importance is placed on several concepts (including the role of conditional independence). Computer architects wanting a deeper understanding may find the references included helpful. A brief explanation is given about the parameters and resources present in field-programmable gate arrays (FPGAs), on which we have prototyped all of our hardware.

In chapter 3, we will present the fundamental stochastic computing primitives and design principles that let us build inferential systems out of intentionally stochastic parts. By viewing the rules of probability through a lens analogous to Shannon's seminal treatment of the Boolean algebra (Shannon 1940), we can build systems that sample from discrete probability distributions. These systems give ample opportunities to exploit parallelism, low-bit-precision, and stochasticity in the underlying media.

We then turn to the problem of causal reasoning and inference (chapter 4), showing how we can build systems for inductive reasoning (Bayes networks) by using these stochastic circuits. Again taking a page from Shannon, we will present a compiler that will automatically compile any given discrete-state factor graph down to one of these circuits, and then show the functioning of these circuits on problems of medical diagnosis.

Vision, especially low-level vision, has been a topic of interest to the machine learning and artificial intelligence communities for years. In chapter 5, we show how we can combine stochastic and deterministic components to "virtualize" portions of the circuit, allowing us to solve much larger models than before. We explore the tradeoffs between space-parallel design and runtime, which enable the design of a stochastic video processor. We then use the video processor to solve stereo vision and optical flow problems in real-time.

Rational models of cognition have been proposed for explaining how animals acquire categorical knowledge. Without explicit training, we can understand the categories present in data, instinctively grouping objects like hand-written digits. In chapter 6, we build a system capable of category learning with a potentially unbounded number of categories, and use it in a perceptual clustering task. Whereas previous probabilistic models required knowing the number of random variables at the outset, the stochastic designs here are capable of much more dynamic behavior, creating new random variables as the data necessitate.

We conclude in chapter 7 by examining the future directions for building inductive, probabilistic systems on stochastic substrates. There is a rich universe of theory waiting to be formalized and a wide variety of novel substrates (DNA-based, spin-based, and more) waiting to be explored. The Markov chains we've built represent only a subset of known MCMC techniques for performing inference, many of which have direct architectural analogs.

Stochasticity is a fundamental fact of all physical systems, even when we use them for computation. Probabilistic inference is the ultimate goal of many neural and cognitive systems. We hope this thesis shows how, by starting to combine the two at the lowest levels, we can make progress on understanding, emulating, and going beyond what evolution accomplishes in a mere 20 watts.

Chapter 2

Probabilistic Modeling, Markov Chains, and Hardware

The world is stochastic, leaving us awash in uncertainty. Every day, we are required to find meaning in ambiguous, noisy, and incomplete data, both in the quantitative and qualitative aspects of our lives.

Here we review the universal formalism for working with this uncertainty, probability theory. We discuss how this formalism is used to model and make predictions about the world. We then turn to the task of performing inference – drawing conclusions from data modeled probabilistically, as well as the role of *sampling* from probability distributions, and how sampling enables us to build systems for inference[1].

We then review the basics of computer architecture design using field-programmable gate arrays, and the various resource tradeoffs involved. Building hardware systems, especially for probabilistic inference, provides a tremendous amount of flexibility, and a basic understanding of FPGA resource utilization will help evaluate those tradeoffs.

2.1 Probability

Probabilistic systems are now staple of undergraduate engineering curricula. Here we review the key concepts and terminology that are often overlooked or first forgotten by undergraduates. Nonetheless, this section is review, and can be skipped.

A question often omitted by by typical statistics and probabilistic curricula is : why probability? Probability theory arises uniquely out of a reasonable set of requirements for mathematically modeling uncertainty (Jaynes 2003), where we wish to represent degree of uncertainty by real numbers, seek a qualitative correspondence with common sense, and want the resulting system to posses mathematical consistency. That is to say, if we want a mathematical formalism for uncertainty which matches our intuition, the laws of probability are all there is.

2.1.1 Terminology

A probabilistic system consists of a state space Ω of possible outcomes. For a die, the space of outcomes are the faces of the die, $\Omega = \{1, 2, 3, 4, 5, 6\}$. A random variable X takes on

[1]What follows is a synthesis of (Liu 2008), (Kevin P Murphy 2012), and (Bishop 2006). The reader is encouraged to seek out these excellent texts for a more thorough understanding

a value x_i in the state space Ω with probability $P(X = x_i)$. Note we often speak of the probability mass function $p(x)$, where $p(x) = P(X = x)$. Note $p(x) \in [0, 1]$, and the total probability of all possible events must sum to one, $\sum_{x_i \in \Omega} p(x_i) = 1$.

In the event that the state space Ω is continuous (such as $\Omega = \mathbb{R}$), the terminology changes slightly, and $p(x)$ is referred to the probability density function, and $\int_\Omega p(x)dx = 1$. Note this density function can be greater than one, $p(x) \geq 0$. For the remainder of the text, we will often simplify as much of this formalism as possible to focus on the main points, often leaving the state space explicit and eliding the difference between a random variable X and a particular value x taken by that random variable[2].

The *joint distribution* of two random variables can be written $p(x, y)$, and is the probability $P(X = x, Y = y)$. We say two random variables are *independent* if the outcome of one does not inform us of the other; another way of expressing this is that their joint distribution factors into the product of the two distributions

$$p(A, B) = p(A) \cdot p(B) \tag{2.1}$$

The *marginal distribution* x is the distribution $p(x)$ when we integrate out the dependence on y from the joint. That is,

$$p(x) = \int p(x, y)\, dy \tag{2.2}$$

The conditional distribution $p(x|y)$ is the distribution of x when we know the value of y. Note that any joint probability distribution can be expressed as the product of its conditional and marginal:

$$p(x, y) = p(x|y)p(y) \tag{2.3}$$

Two variables are conditionally independent given a third if knowledge of the third, say z, decouples the dependence between x and y. This can be expressed as

$$p(x, y|z) = p(x|z)p(y|z) \tag{2.4}$$

This situation arises frequently in probabilistic modeling when z is the cause of both x and y. If you know z, then knowing more information about x doesn't tell you anything about y.

2.1.2 Bayes' Theorem

Bayes' theorem is perhaps the most talked-about and most misunderstood component of all of probability theory. It should not have its own name, as it is simply the consequence of the algebraic manipulation of probability distributions. But it is powerful – Bayes' rule can be thought of as probabilistic inversion, telling us how to compute the *posterior distribution* from the *likelihood* and the *prior distributions.*

When modeling probabilistic systems, we often have some set of hypotheses H and some collection of data D, and we want to reason about the probable hypotheses given the data. That is, we want to understand the *posterior distribution*, $P(H|D)$. The *prior distribution*

[2]The field of theoretical probability is rich, diverse, and has deep connections to many aspects of probabilistic modeling. Those interested in a formal treatment are highly encouraged to see chapter 2 and the appendix of (Schervish 1996)

on hypotheses, $P(H)$, is how much we believe in each hypotheses before seeing the data. The *likelihood* is our measurement model – how, given a particular hypothesis H, the data D would look. We express this as a conditional probability distribution $P(D|H)$.

Bayes theorem is simply:

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \tag{2.5}$$

Note we have described in some sense a forward generative model, where we have some beliefs about our hypotheses and we know how they give rise to data. Bayes' theorem let's us go from our data D, and this generative model, to a distribution on possible causes (hypotheses) of the data.

Note that the denominator is just a normalizing constant,

$$P(D) = \int_H P(D|H)P(H)dH$$

independent of any particular hypothesis. But this constant but requires us to integrate/sum over all possible hypotheses. For complex probabilistic models, this is often impossible. Dealing with this integral is one of the principle challenges of probabilistic inference.

Note that once we have the posterior distribution, we can then run our model "forward" – we can say "what's the probability of the next datapoint?" The *posterior predictive distribution* is is the distribution on a new datapoint, given a series of existing observations D. That is,

$$p(x^*|D) = \int_H p(x^*|h)p(h|D)\,dh \tag{2.6}$$

2.2 Probabilistic Modeling

Perhaps because they rarely venture outside, a canonical example model used by probablists involves reasoning about why ones lawn is wet. Is the grass wet because the sprinkler was on earlier during the day, or because it rained earlier during the day? And what does an observation that it is cloudy out have to do with anything?

We will build a circuit to answer this question later (see section 4.4), but for now we consider how to express the problem? An electrical engineer today who set out to build a digital system by writing down pages of Boolean expressions would be considered quite mad, yet unfortunately a great deal of probabilistic systems modeling begins with a lot of symbolic math on paper.

Fortunately, the past two decades have seen an attempt to develop better formalisms for expressing probabilistic models. Graphical models (Pearl 1988) attempted to express complex conditional independence structure graphically, a sort of "probabilistic schematic" of a problem. This in turn has lead to the rise of probabilistic programming languages – domain specific languages for expressing rich probabilistic models. Increasingly, these probabilistic programming languages also contain engines for performing probabilistic inference in them (Milch et al. 2005; Goodman et al. 2008; Mccallum, Schultz, and Singh 2009).

2.3 Inference

As Bayesians, we're often most interested in the posterior distribution, $P(H|D)$ – how likely is a hypothesis, given the data? More generally, given data and model, we want make inferences about various aspects of the model – hidden parameters, unobserved values, etc. In theory, Bayes' rule tells us the posterior, but often the denominator $P(D) = \int_H P(D|H)P(H)\,dH$ is intractable and renders the entire exercise, at first blush, impossible. Sampling methods, both exact and approximate, are described below that offer an incredible way around this problem.

2.3.1 Monte Carlo

Often we want to ask some question about a particular probability distribution $p(x)$ like "what is its average value?" or "what is the most likely value of x?". In the case of the average value, or expectation, we want to compute

$$E[x] = \int_{x \in X} x p(x)\,dx \tag{2.7}$$

If we can draw m independent samples from the distribution $p(x)$, we can approximate $E[x]$ by

$$\hat{\mathbf{E}}[x] = \frac{1}{m}(x_1 + x_2 + \cdots + x_m) \tag{2.8}$$

This is the basis of all Monte Carlo methods, and has the phenominal property that the error of this approximation grows as $O(m^{-1/2})$, regardless of the dimensionality of x (Liu 2008).

2.3.2 Sampling

How do we obtain the samples x_i needed to power the Monte Carlo machinery discussed above? And as engineers, we should be concerned with the mechanisms of generating these samples – their accuracy, their performance, and their complexity. We will refer to all systems that generate samples from probability distribution as "samplers".

The simplest sampler is the classic fair coin. You can think of a fair coin as a physical machine, a sampler constructed to give samples from the Bernoulli distribution

$$p(X) = \begin{cases} H & p = 0.5 \\ T & p = 0.5 \end{cases}$$

When you *flip* a fair coin a number of times, you are effectively generating samples from the above probability distribution.

Other physical examples of samplers include fair and biased dice. In fact, it is often trivial to sample from any probability distribution where we can explicitly enumerate the probabilities associated with each outcome in the state space. Of course, this enumeration approach does not work very well as the enumerable space grows large, or is infinite.

The prime mover

If you can draw samples from one distribution, you can often use that sampler to then draw samples from a totally different, *target* distribution. Indeed, most of the sampling approaches we discuss in this work are built on other samplers, exploiting their controlled stochasticity to sample from different target distributions. While we don't focus much on this particular compositional aspect of sampling methodology, it is nonetheless incredibly powerful.

But then where do the original random values come from – what is the prime sampler? For all of the circuits we discuss, we assume the simplest digital source of randomness possible : a random bit stream. This stream can be viewed as samples from a fair coin, and is the root source of entropy for all our operations.

There are many ways to generate this randomness – computer scientists use a pseudo-random approach, where we construct numerically-sensitive systems that evolve in time in ways that look empirically quite random. For all the work described in this thesis, this pseudorandomness is sufficient – higher quality randomness required for cryptography is unnecessary.

2.3.3 Rejection Sampling

This chapter is not a detailed survey of all possible methods of generating random variables, but we will highlight several classical approaches that we build on later in the text. Rejection sampling is one such approach.

Say we have a sampler for one probability distribution, $q(x)$, but wish to draw samples from another distribution, $p(x)$ with the same support. It often happens that we only know $p(x)$ up to a normalizing constant, that is we have $p^*(x)$

$$p(x) = \frac{1}{Z} p^*(x) \tag{2.9}$$

Von Neumann (Neumann 1951) showed that, if we can find a sampler that gives us samples from the distribution $q(x)$ where $Mq(x) \geq p^*(x)$ for some constant M, we can use this method to draw samples from $p(x)$ as follows:

- Draw a sample x' from $q(x)$ and also $u \sim Unif(0, 1)$
- Let
$$r \leq \frac{p^*(x')}{Mq(x')} \tag{2.10}$$
- If $r < u$ then accept x' as an IID sample from $p(x)$. Otherwise, repeat the procedure.

Rejection sampling says, in effect, if $p(x)$ is under the curve of $Mq(x)$, sample uniformly under the area of $Mq(x)$ and accept the values that are under the curve of $p(x)$.

We adapt a proof from (Liu 2008) to show that this method works. If I is an indicator function, where $I = 1$ when x' is accepted, then

$$P(I = 1) = \int P(I = 1|X = x')q(x)\, dx = \int \frac{Zp^*(x)}{Mq(x)} g(x)\, dx = \frac{Z}{M} \tag{2.11}$$

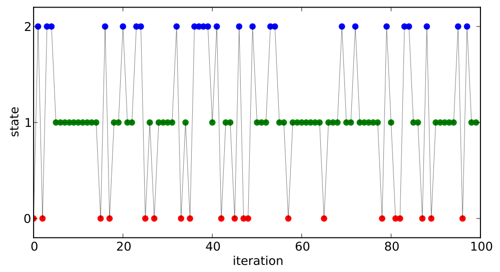
and so

$$p(x|I = 1) = \frac{Zp^*(x)}{Mq(x)} q(x)/P(I = 1) = p(x) \tag{2.12}$$

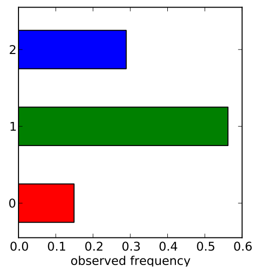
Note that the closer $Mq(x)$ is to $p^*(x)$, the closer r will be to 1 and the more likely the draw x' will be accepted. It's also possible for this procedure to repeat virtually indefinitely without ever producing accepting a sample.

$$\begin{pmatrix} 0.1 & 0.5 & 0.4 \\ 0.1 & 0.7 & 0.2 \\ 0.3 & 0.3 & 0.4 \end{pmatrix}$$

(a)



(b)



(c)

Figure 2-1: A Markov chain and its ergodic behavior. a.) The transition matrix for a three-state Markov chain. b.) an example run of this markov chain, showing switching between three states. c.) the empirical distribution on state values for the Markov chain after 10,000 iterations.

2.4 Markov-Chain Monte Carlo

Rejection sampling is inefficient to the point of being useless in high-dimensional settings. So what do we do when we want to sample from an unnormalized distribution which we can evaluate analytically, $p^*(x)$, of the sort that often comes up in performing Bayesian inference? Markov chain Monte Carlo (MCMC) tells us how to build a special type of probabilistic model – a Markov chain – that will help us out. But before digging into MCMC we must first review Markov Chains.

2.4.1 Markov Chains

Consider a sequence of random variables, X_0, X_1, $\cdots$, X_T indexed by the time variable t. Let X be defined on a finite state space $x \in 1 \ldots K$. The sequence of random variables is a Markov chain if the probability of x_{t+1} only depends on the value of x_t

$$p(x_{t+1}|x_1, x_2, \cdots, x_t) = p(x_{t+1}|x_t) \tag{2.13}$$

As a Markov chain evolves in time, it will transition from one state to another. Note that a Markov chain can quickly "forget" where it used to be. The distribution $p(x_{t+1}|x_t)$ can be viewed as a transition matrix or kernel, A^t:

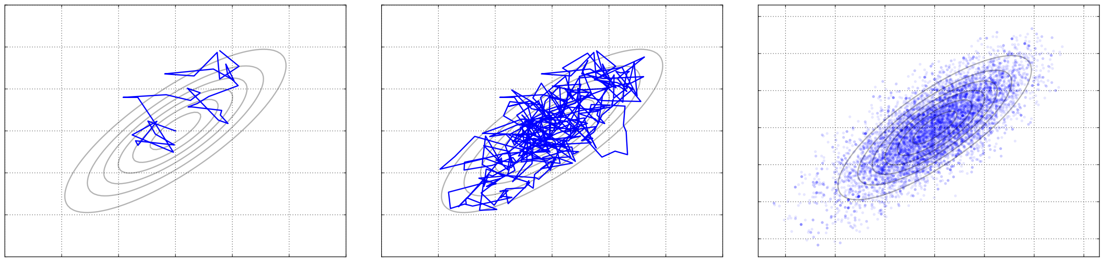
$$A^t(x', x) = p(x_{t+1} = x'|x_t = x) \tag{2.14}$$

For the rest of this document we will only discuss *homogeneous* Markov chains whose transition matrix A^t is the same for all t, and thus we will write simply as A.

If a Markov chain is irreducible and aperiodic, it will define a *stationary* distribution on its state space. That is, after running the markov chain for a long time, when we examine the history of visited states, we will get a probability distribution over p(x).

Continuous values

A Markov chain can also be defined over a continuous-valued state space, such as a random walk in 2D space. Let the state variables $\mathbf{x}_t = (x_t, y_t)$ with $x \in \mathbb{R}$, $y \in \mathbb{R}$.



(a) 50 iterations (b) 500 iterations (c) 10000 iterations

Figure 2-2: Metropolis-Hastings iterations for the two-dimensional Gaussian distribution specified in the text – true distribution contours are in grey. This plot shows an appropriately-scaled proposal distribution. (a) and (b.) connect adjacent points with lines to further indicate the path of the Markov chain. c.) shows 10000 with each point partially transparent.

Note the transition kernel A in this case is not an explict matrix as above. Rather, it's simply a function mapping the state at time t to the state at time $t+1$. This is a perfectly valid way to define a Markov chain.

2.4.2 Metropolis Hastings

In 1953 Metropolis showed (Metropolis et al. 1953) that one can construct a Markov chain with a desired stationary distribution, $p(x)$, with an exceptionally simple procedure, The algorithm, now know referred to as Metropolis-Hastings (after Hastings generalized it), requires two things:

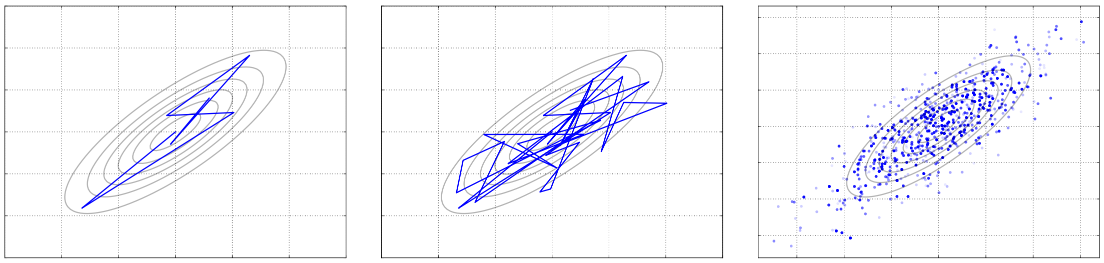
- A means of evaluating $p^*(x)$, the unnormalized version of the target density $p(x)$, for any point in your state space x.
- A *proposal* distribution, $q(x'|x)$, defined on the same state space as $p(x)$, from which you can both exactly sample a new point in the state space, $x' \sim q(x'|x)$ and evaluate the probability of any particular proposal value. The proposal distribution can depend on the current point in the state space, as indicated by the above.

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:

$$a = \min\left(1, \frac{p^*(x')}{p(x)} \frac{q(x|x')}{q(x'|x)}\right) \tag{2.15}$$

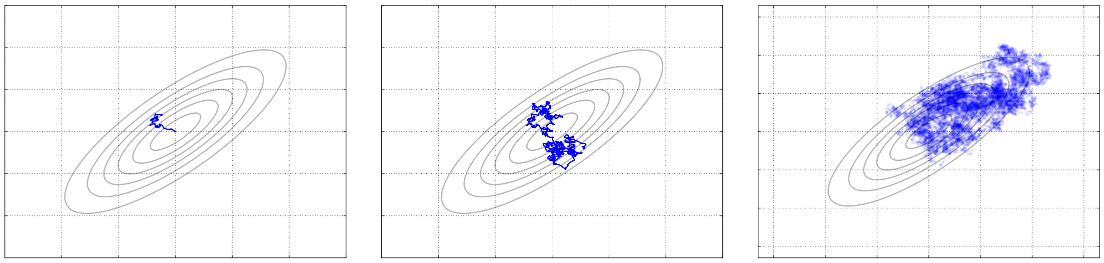
Note a is closer to one the larger the ratio $\frac{p^*(x')}{p^*(x)}$ – that is, the more probable the new state is, relative to the current state. There's a catch, though – the ratio $\frac{q(x|x')}{q(x'|x)}$ is higher the more likely we would be to propose the *current state* x if we were already at the new state x'. Dealing with this tradeoff is at the core of designing efficient MCMC systems.

Taking an example from (Liu 2008) (and figure inspiration from (Bishop 2006), we use Metropolis hastings to sample from a bivarite Gaussian distribution. This is a nice example



(a) 50 iterations (b) 500 iterations (c) 10000 iterations

Figure 2-3: MH for the same distribution as figure 2-3, with a proposal distribution that's too large ($\sigma = 4.0$). Note all the jumps that are accepted are massive, and even after 10000 iterations only a few points have been accepted.



(a) 50 iterations (b) 500 iterations (c) 10000 iterations

Figure 2-4: MH for the same distribution as figure 2-3, with a proposal distribution that's too small ($\sigma = 0.04$). Proposals are almost always accepted but the chain cannot move far enough, even after 10000 iterations, to accurately explore the space.

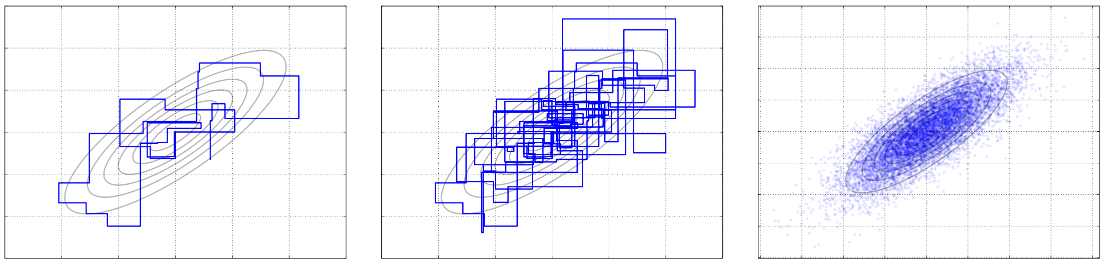
because we can exactly compute the contours and plot the results. Let $\mathbf{x} = (x_1, x_2)$ and

$$\mathbf{x} \sim \mathcal{N}\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right\} \tag{2.16}$$

Our proposal distribution $q(x'|x)$ is an isotropic two dimensional Gaussian, centered at the current state value x, with standard deviation $\sigma = 0.4$. Figure 2-2 shows the results of letting the markov chain run for 50, 500, and 10000 iterations.

MH lets you construct a Markov chain guaranteed to asymptotically produce samples from the target distribution of interest, works well in high dimensions, and is straight forward to implement. None the less, engineering good proposal distributions is tricky, and sometimes impossible.

Just as with rejection sampling, MCMC methods work better when you use a proposal distribution that is closely matched to your target distribution. In our simple example, this means using a Gaussian proposla whose size is roughly "on par" with the size of the target distribution. If the proposal distribution σ is too small, the Markov chain will accept nearly every proposal, but not get very far – it will remain stuck in a tiny space and fail to explore the entire distribution (figure 2-4). However, if the proposal distribution is too large, often the chain will propose ridiculous values, and they will rarely be accepted (figure 2-3).



(a) 50 iterations (b) 200 iterations (c) 10000 iterations

Figure 2-5: Gibbs sampling the same distribution as figure 2-3. Gibbs sampling changes one variable a time and always accepts the proposed state, thus all the moves are axis-aligned.

Gibbs Sampling

For a high dimensional problem, if we can figure out the conditional probability distribution for a variable and know how to sample directly from it, we can do better than basic MH. A derivative of MH termed Gibbs sampling (S. Geman and D. Geman 1984) lets us build a transition kernel that is always accepted.That is, maybe sampling directly from $p(x_1, x_2)$ is difficult, but we can sample from $p(x_1|x_2)$ and $p(x_2|x_1)$ with ease. Gibbs sampling allows the construction of a valid transition kernel where we simply sample from each conditional distribution over and over.

As an example, using the same bivariate Gaussian as the above, again with $\mathbf{x} = (x_1, x_2)$,

$$\mathbf{x} \sim \mathcal{N}\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right\} \tag{2.17}$$

we can analytically compute the conditional distributions

$$x_2^{(t+1)}|x_1^{(t)} \sim \mathcal{N}(\rho x_1^{(t)}, (1-p^2)) \tag{2.18}$$

$$x_1^{(t+1)}|x_2^{(t+1)} \sim \mathcal{N}(\rho x_2^{(t+1)}, (1-p^2)) \tag{2.19}$$

$$\tag{2.20}$$

To Gibbs sample, we first sample a new value $x_2^{(t+1)}$ conditioned on x_1^t, and then sample a new value $x_1^{(t+1)}$ conditioned on the just-sampled $x_2^{(t+1)}$. The results of this process can be seen in figure 2-5. Note that there is no accept/reject step nor derivation of a problem-appropriate proposal distribution! These are the benefits Gibbs sampling provides, as long as we can exactly sample from the conditional distributions.

2.4.3 MCMC Theory

Metropolis hastings tells us how to construct a Markov chain transition kernel K on a state space x which has two crucial properties (adopting from (Bonawitz 2008)):

- $p(x)$ is an *invariant* distribution of the transition kernel K. That is, repeated application of K to $p(x)$-distributed data will maintain the distribution $p(x)$.
- K is a "nice Markov chain" – it is irreducible (any point in the state space x can be

reached from any other point in the state space) and aperiodic (there are no cycles to get stuck in)

MCMC transition kernels compose quite well. If K_1 and K_2 are two different transition kernels that both have $p(x)$ as their stationary distribution, then their chained application K_2K_1 also has this stationary distribution. This means that we can first apply K_1, and then K_2. In fact, we can even randomly pick which to apply, and preserve $p(x)$.

In fact, even if K_1 and K_2 don't have $p(x)$ as their stationary distribution, K_1K_2 may. We make extensive use of this property to design (via MH) a kernel K_1 that samples from x_1 and another kernel K_2 that samples from x_2. The combined K_1K_2 (that is, the repeated application of one after another to the joint state space (x_1, x_2) has $p(x_1, x_2)$ as it's stationary distribution.

2.4.4 Runtimes and Asymptotics

The astute engineer should now be left with two questions: "How do I design good proposal distributions?" and "how long do I run my chain?"

Designing good proposal distributions is a crucial part of the art of using MCMC methods, especially when you can't exactly sample from the conditional distributions (and thus Gibbs sampling is off the table.) As seen above, even just getting the "scale" of the proposal distribution correct in a two-dimensional problem can be a challenge. Compositionality helps us tremendously – if K_1 is a MCMC kernel that works well in one regime, and K_2 works well in another, we can just compose them and get the best of both worlds. In the above 2-d Gaussian example, this would mean that we could apply MH three times per iteration, each with a different width of the isotropic gaussian proposal kernel.

The samples produced by MCMC Markov chains are not independent – there is obviously a long-run correlation. The better the MCMC kernel, the quicker this correlation disappears – the more efficnetly the kernel "moves" around the state space. Multimodality can be quite challenging to deal with. When the target distribution of interest has multiple modes (peaks) separated by very improbable regions, it is easy for an MCMC kernel to get stuck in one mode. Diagnosing this behavior is a challenge and an active area of research.

In practice, especially for many of the applications we consider, the primary challenge (and bulk of the computational effort) is spent finding a high-probability region from an initial point in the state space. This initial search phase is often referred to as "burn-in" in the literature, and it is recommended that chains be run for a long period of time initially, and then after that burn-in period, every subsequent Nth iteration is taken as a "sample".

2.5 Hardware, Bit Precision, and FPGAs

Computer architecture is part creativity, part trade-off optimization. Those tradeoffs almost always involve optimizing for cost, power, or performance – the old adage of "pick two" often applies. Engineers often use "silicon area" as a proxy for cost and power – smaller designs are both more inexpensive to manufacture and, with fewer transistors, consume less power.

We prototype all of our circuits on reconfigurable, field-programmable gate arrays (FPGAs), as opposed to making them in actual silicon. A FPGA consists of a dense network of undifferentiated flip flops and programmable look-up tables whose functionality and connectivity are specified at runtime by the loading of firmware. This is vastly preferable for development compared to synthesizing and fabricating actual silicon devices, as new designs can be prototyped and evaluated in days, not months.

To better understand how our devices perform with respect to existing architectures, we use "FPGA resources" (described below) as a proxy for silicon area. Our designs all target the Xilinx Virtex-6 series of FPGAs (*Virtex-6 Family Overview* 2012) using a commercial FPGA development board form Pico Computing (Module 2013). The FPGA consists of hundreds of thousands of "slices" of logic, each containing four look-up tables, eight flip-flops, various multiplexors, and fast carry logic (*Virtex-6 Configruable Logic Block User Guide* 2012). We primarily measure the following when performing our resource utilization experiments:

1. Flip Flops : stateful 1-bit synchronous logic
2. Look-up tables (LUTs): reconfigurable six input, one output purely combinational logic blocks capable of implementing any 6:1 function
3. Block RAMs : dedicated synchronous static RAM, optionally dual-ported.

We can only compare designs to each other, measuring a rough amount of the "quantity of stuff" needed, not true silicon area. Our task is complicated by the eagerness of the Xilinx synthesis tools to infer BlockRAM memory units for any structure that looks like a stateful, addressable memory. We have manually disabled this automatic inference to get more consistent, easier-to-compare estimates for the resources utilized by a particular design. This tends to overstate the amount "stateful logic" a design would require, resulting in our estimates being quite conservative. Our synthesized designs do use the BlockRAM for runtime performance, however.

2.5.1 Fixed Precision

Throughout the document we will repeatedly refer to various bit-precisions of the underlying circuit. All numerical values are implemented in traditional fixed-width twos-complement. An $m.n$ bit representation expresses values $\in \mathbb{R}$ with integers between -2^{m+n-1} and $2^{m+n-1} - 1$, implicitly divided by 2^n. The m are referred to in the text as the *integer bits*, and the n as the *fractional bits* (see figure 2-7).

This allows us to perform very easy analysis of the impact on dynamic rage for various bit precisions. For many circuits, we express the precision as a tuple with $(m.n, q)$ where $m.n$ is the fixed-point precision of the underlying log-probability calculation, and Q are the number of bits in $1.Q$ fixed-point (non-two's-complement) format used for sampling from the underlying discrete distribution.

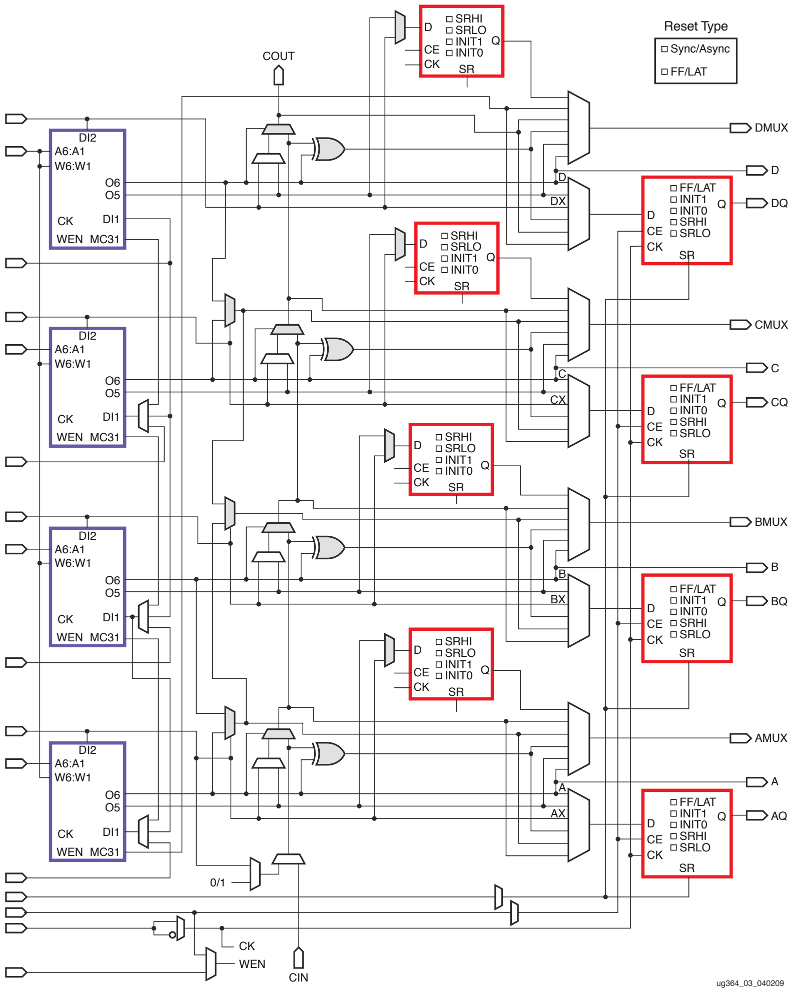


Figure 2-6: One SLICEM from a Virtex-6 FPGA (from (*Virtex-6 Configruable Logic Block User Guide* 2012)) with LUTs highlighted in blue and flip-flops highlighted in red.

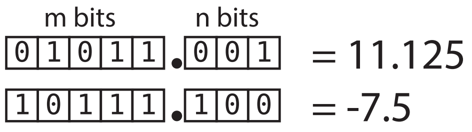


Figure 2-7: Examples of twos-complement bit precision for $m = 5$, $n = 3$ bits

Chapter 3

Stochastic Circuits for Inference

Previous attempts to work with neuron-like computing substrates have lacked a guiding set of principles that would enable designs to scale. In a sense, "engineering" with them has been very difficult. Yet we'd still like to build systems that exhibit and exploit many of the properties of neural systems – distributed, robust, low-power computation with the ability to handle the massive uncertainty present in the world.

Natively stochastic circuits offer these advantages, while being amiable to the construction of useful large-scale systems for solving problems of probabilistic inference. Our stochastic circuits produce samples from probability distributions conditioned on their inputs; this lets them obey abstraction and composition laws that closely mirror the probability algebra. Ultimately, this allows the construction of large-scale systems for probabilistic inference. Using these rules, we show how the conditional independence structure of many probabilistic problems provides ample opportunities for parallelization.

In this chapter we describe several basic stochastic circuits for producing exact samples from underlying probability distributions. We show how these circuit elements offer various performance, silicon resource, and complexity trade-offs. Various design idioms allow tremendous flexibility in exploring the space between larger-and-faster and slower-but-more-area-efficient architectures.

The construction of stochastic finite state machines (Markov chains) with the above stochastic elements result in architectures ideally suited for certain classes of approximate inference, namely Markov-Chain Monte Carlo. Crucially, the conditional independence structure of many probabilistic models, coupled with these sampling methods, allows for tremendous gains via parallelism.

The uncertainty present in most probabilistic problems dwarfs any rounding error or precision errors present in the computational substrate, allowing us to compute with very low bit precision. Lower bit precision allows us to build smaller, denser circuits, reducing silicon area and thus cost and power. Similarly, because our stochastic systems generally evolve over time, transient perturbations rarely impact long-term behavior. This means that we can work with both less reliable substrates and be more tolerant of manufacturing errors.

3.1 Primitives

Fundamentally, stochastic circuits are circuits that produce *samples* from a particular probability distribution, conditioned on their inputs (figure 3-1). This stochasticity requires a

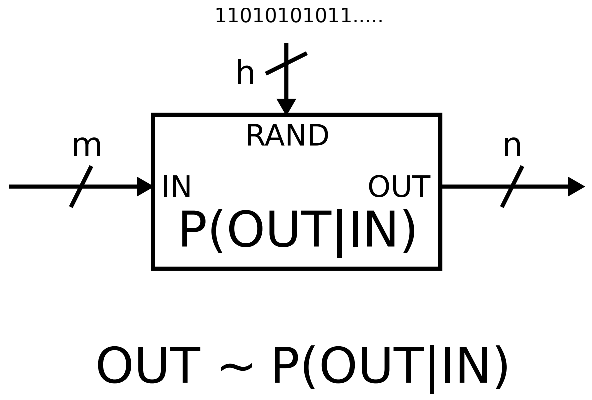


Figure 3-1: A typical stochastic circuit element. The output is a sample from a distribution conditioned on the input. Entropy is consumed in the process.

source of entropy, which as we show later can be extrinsic or intrinsic, and even of poor quality. In all subsequent discussions we make this entropy external and the input port explicit, as a reminder of its importance.

Just as a Boolean logic gate is completely characterized by its truth table, a stochastic logic circuit can be completely characterized by its conditional probability table, specifying the probabilities of all possible outputs given all possible combinations of inputs. This lets us easily show that stochastic logic recovers Boolean logic in the deterministic limit. Figure 3-2 shows a classical Boolean AND gate and the associated truth table. It also shows a stochastic logic element which implements deterministic AND functionality by having a conditional probability table with delta potentials on the set of accepted outputs. Thus, if $A = 1$ and $B = 1$, the probability of observing a 1 at the output is 1.0.

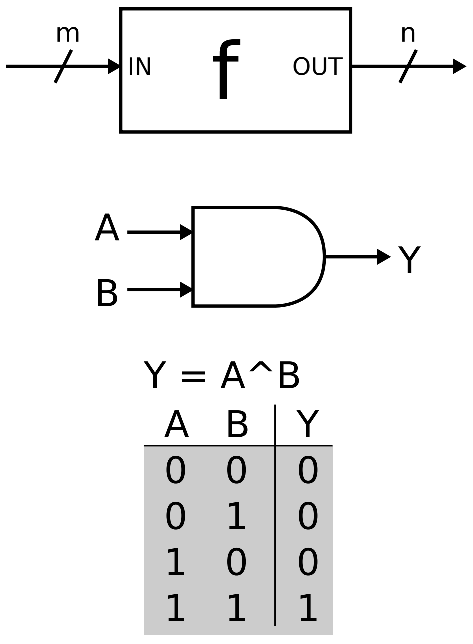
Boolean logic effortlessly supports the combination of arbitrary elements to build up more complex functions and circuits (*composition*) as well as the *abstraction* of complex circuit functions into black boxes fully specified by their truth tables. This has allowed the growth of circuit complexity from 7400-series quad NAND gates up through billion-transistor microprocessors today.

Stochastic logic elements crucially support very similar processes of abstraction and composition, which is not surprising given the deterministic reduction to Boolean logic. Consider two stochastic circuit elements, one producing samples $B \sim P(B|A)$ and one samples $C \sim P(C|B)$. If we chain the output from the first to the input of the second, we obtain sample from the joint distribution of $(B, C) \sim P(B, C|A)$ (figure 3-3).

Abstraction lets us replace the chain of stochastic logic elements with a single element, ignoring the details of the internal implementation. By discarding the B output, the abstracted unit produces samples from the marginal distribution $P(C|A) = \sum_B P(C, B|A)$.

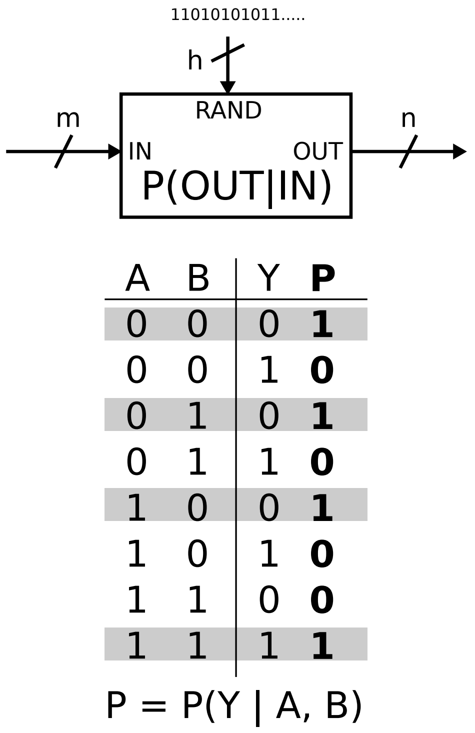
3.1.1 Clocking, timing, iterative state

Classical combinational digital logic is characterized by the propagation delay, T_{PD}, between changes at the inputs and the reflection of those changes on the output. Because our outputs are samples, we adopt an explicit clocking discipline, making all of our logic inherently synchronous. There are many ways of controlling synchronous logic systems, but the most common is a shared global clock and selective enable lines. We adapt this convention for



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

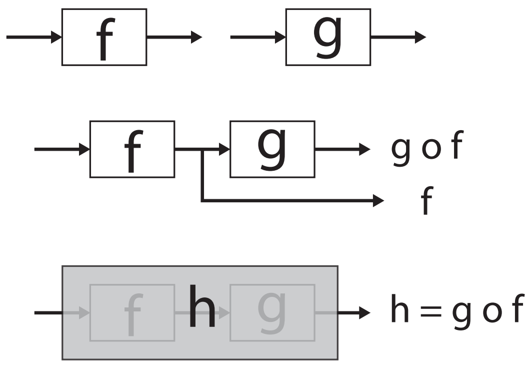
(a) Boolean AND gate



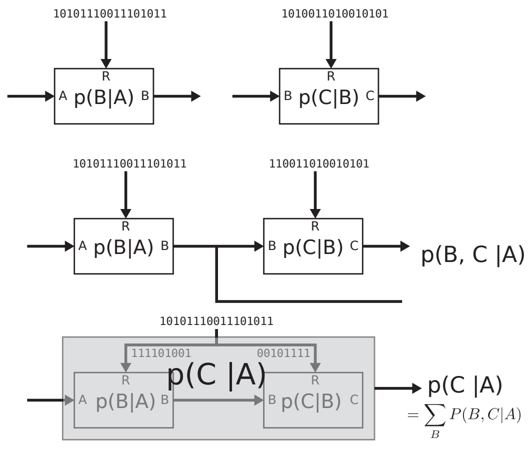
A	B	Y	P
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**0**
1	1	1	**1**

(b) Stochastic Logic AND gate

Figure 3-2: A classical Boolean gate, in this case an AND gate, performing logical conjunction, and completely specified by its truth table. A stochastic logic gate can perform the same deterministic function by only having probability mass on the allowed output.



(a) Boolean logic



(b) Stochastic Logic

Figure 3-3: (a.) Classical Boolean logic gates f and g support effortless composition : chaining the output of f into the input of g yields the operation $g \circ f$. The combined unit can be abstracted into a unit h which performs $g \circ f$ without regard to internal implementation. (b.) Stochastic logic elements support similar abstraction and composition, with combination resulting in samples from the joint $P(B, C|A)$ and abstraction resulting in samples from the marginal $P(C|A)$.

	$<$	$\leq$
bits	LUT	LUT
2	1	1
4	2	2
8	4	4
12	11	11
16	15	15

Table 3.1: Theta gate resource utilization as a function of the number of bits in the weight representation.

all of our stochastic gates unless otherwise noted.

For hand shaking, we also give most gates a "done" line, indicating that the sample is ready on the output. This allows for the efficient asynchronous chaining of variable-latency sampling systems, as well as the effortless construction of Markov chains (see below).

3.1.2 The Theta Gate

The simplest stochastic gate is the theta gate, which samples from a Bernoulli distribution conditioned on the inputs. The input lines are thus setting the weight parameter θ for the flip of a biased coin.

A theta gate can be implemented using a single comparator, where the input bits set the threshold of comparison. The theta gate produces exact samples from the indicated distribution and can generate one sample per cycle. By varying the input bitwidth, we can vary the encoding of the unit interval $[0, 1]$ – more bits allow finer specification of the weight θ. We can implement a theta gate with two possible comparison functions, $<$ and $\leq$, controlling which of two endpoints (0 or 1) are included in the encoding of θ.

Note that the the theta gate is the initial bridge between stochastic and deterministic logic, exploiting a deterministic comparison function coupled with entropy to produce stochastic outputs.

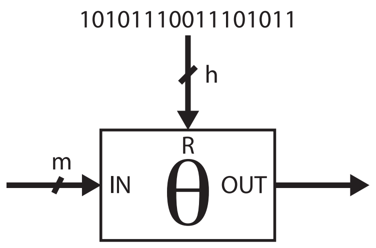
3.1.3 The Uniform Rejection Gate

A uniform rejection gate produces a sample uniformly from the integers $[0, k]$ via rejection sampling. Figure 3-5 shows one possible architecture. For the gate shown, MAX is the maximum possible value for k. MAX sets the envelope function for rejection sampling – the closer k is to MAX, the more efficient the sampler becomes (figure 3-5). On average we expect $1 + (1 - k/\text{MAX}) * \text{MAX}$ cycles per sample.

3.1.4 The CPT Gate

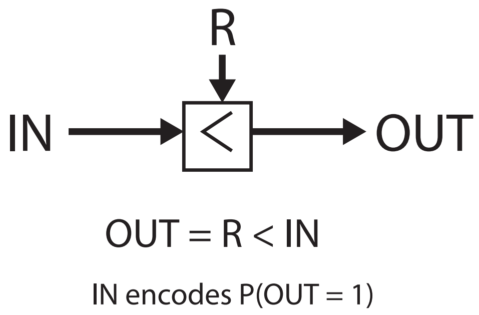
The Conditional Probability Table Gate (CPT Gate) lets you exactly sample from a discrete K-valued distribution, conditioned on the inputs X_i. This is equivalent to generating a collection of K-sided dice, and using the inputs to switch between them. Internal precision is specified to p bits.

Internally, the CPT Gate is implemented via a look-up table. The particular configuration of X_i values indexes into a ROM which encodes the CDF of the desired distribution.

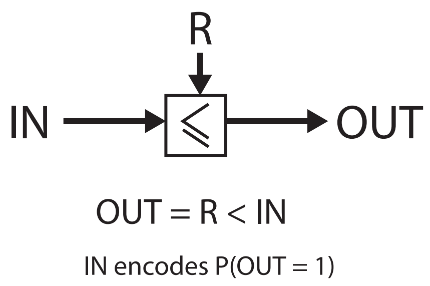


IN	OUT	P
0000	0	**1**
	1	**0**
0001	0	**15/16**
	1	**1/16**
⋮	⋮	⋮
1111	0	**1/16**
	1	**15/16**

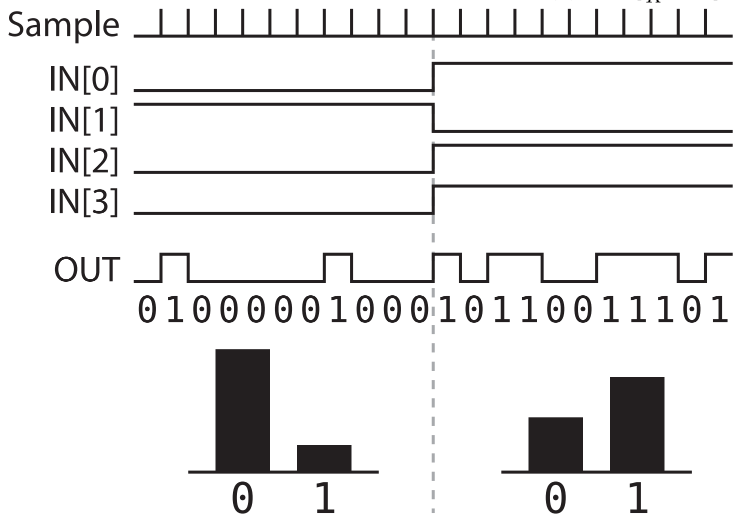
(a) Example Theta Gate



(b) $p \in [0, \frac{K-1}{K}]$

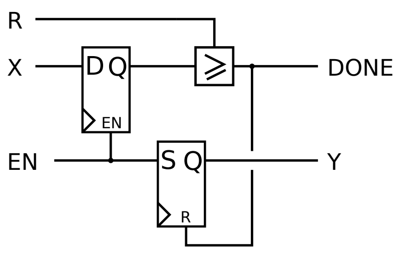


(c) $p \in [\frac{1}{K}, 1.0]$

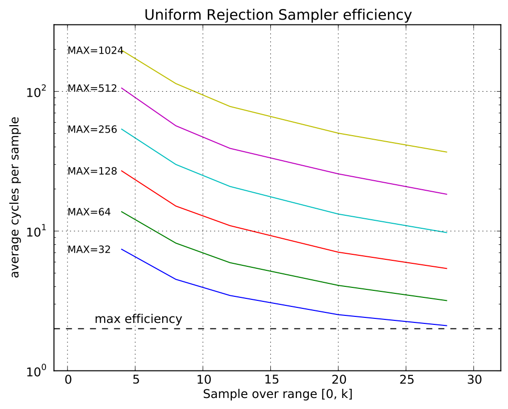


(d) Example output from theta gate over time

Figure 3-4: a.) An example theta gate, showing the input conditioning values (for $m = 4$ and the probability of $\{0, 1\}$ on the output. Two different implementation styles (b. and c.) control the endpoints. d. Sampling from a theta gate for two different input values

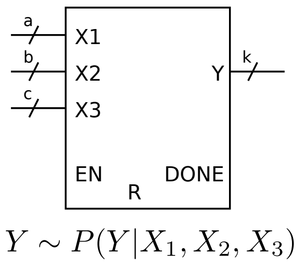


(a) Architecture



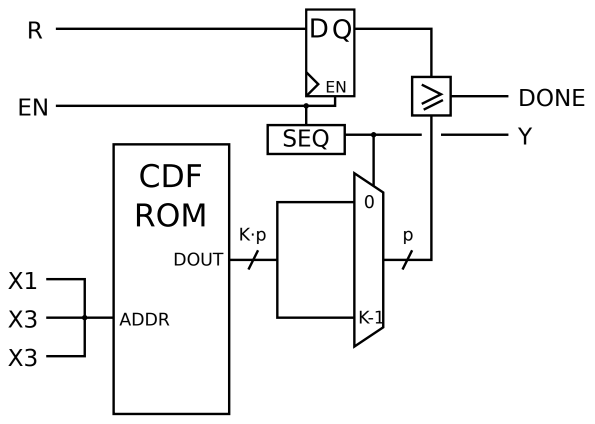
(b) Efficiency

Figure 3-5: Uniform Rejection Gate, sampling uniformly on $[0, k]$ for $k <$ MAX.



$Y \sim P(Y|X_1, X_2, X_3)$

(a) Symbol



(b) Architecture

Figure 3-6: CPT Gate, which produces exact samples from a K-valued probability distribution conditioned on the $\{X_i\}$ inputs.

The gate samples a random internal value, and then steps through the CDF. The gate takes an expected $\frac{K}{2}$ ticks per sample.

Table 3.2 shows how the silicon area and FPGA resource utilization vary as a function of the number of die sides ($k = \{2, 4, 8\}$), the internal bit precision $p = \{4, 8, 12\}$, and the total number of conditioning bits $bits = \{2, 4, 8\}$.

3.1.5 Stream Sampling Gate

In many problems, we do not know ahead of time the exact distribution we wish to sample from ahead of time – sometimes we do not even know the parametric form. The stream-sampling gate lets us sample from an arbitrary, normalized probability distribution. It does this by taking in a sequence of normalized probability values (in 1.q format) along with labels for each value, and generates an exact sample from the streamed-in distribution. The gate requires that the input distribution be normalized.

The gate works by sampling a random uniform value r at reset, and then computing

	$k = 2$		$k = 4$		$k = 8$	
	LUT	FF	LUT	FF	LUT	FF
$p = 4$						
bits=2	12	14	25	32	35	38
bits=4	12	15	23	37	40	58
bits=8	12	27	28	74	42	141
$p = 8$						
bits=2	15	23	30	43	72	63
bits=4	16	25	35	56	78	103
bits=8	16	49	35	128	80	268
$p = 12$						
bits=2	25	28	50	54	95	69
bits=4	27	35	57	78	109	138
bits=8	27	71	57	186	111	390

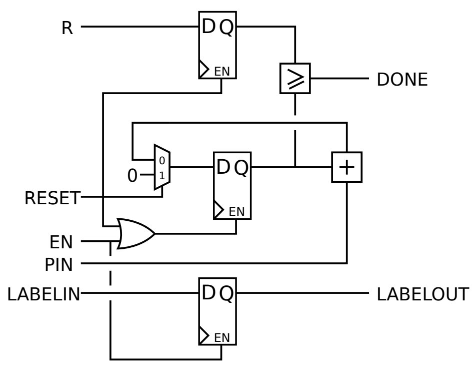
Table 3.2: FPGA resource utilization (slice LUTs and flip-flops) and for the CPT Gate as we vary the number of possible outcomes k, the internal bit precision p, and the number of conditioning bits "bits".

	$q = 4$		$q = 8$		$q = 12$	
	LUT	FF	LUT	FF	LUT	FF
2	33	16	43	30	57	46
4	41	18	49	30	65	48
6	49	20	55	30	73	50
8	57	22	61	30	81	52
10	65	24	67	30	89	54

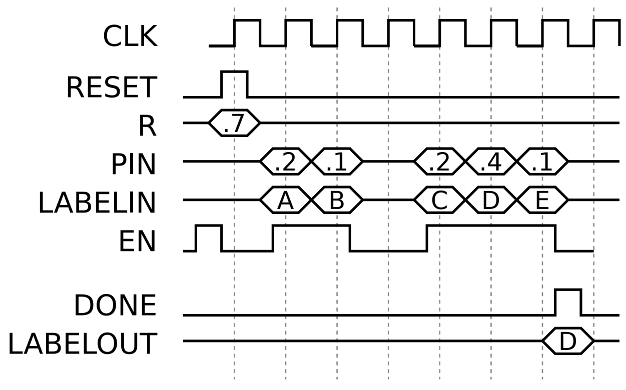
Table 3.3: Stream Sample gate resource utilization. Resource utilization increases roughly linearly with the number of bits q in the probability representation, and linearly in the number of bits in the label representation.

the CDF on the fly, returning the label i when when $r < \sum_i p_{in}$ (figure 3-7). The gate can also aborts the sampling process once a value has been sampled, even if it has not seen all values in the distribution.

The gate runs in $O(N)$ in the number of probability values in the underlying distribution. Note that the issue of precision-of-values and number-of-values are completely distinct – a probability distribution with 1000 possible outcomes, but where 999 of those outcomes have zero chance of occurring, only needs one bit to express the probability values themselves, but $\log_2 1000 \approx 10$ bits to label the values.

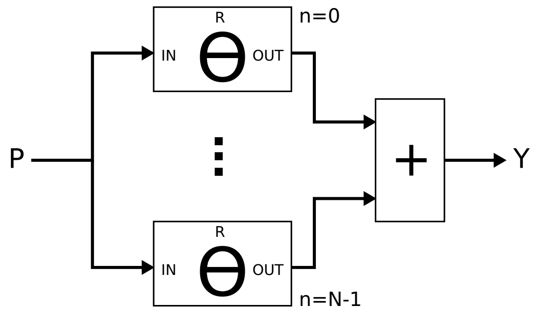


(a) Architecture

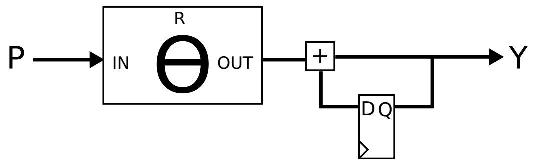


(b) Timing

Figure 3-7: Stream sample gate. a.) Accumulator-based architecture. b.) Timing



(a) Fully-parallel



(b) Serial

Figure 3-8: Binomial sampling circuit, with two implementations – 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.

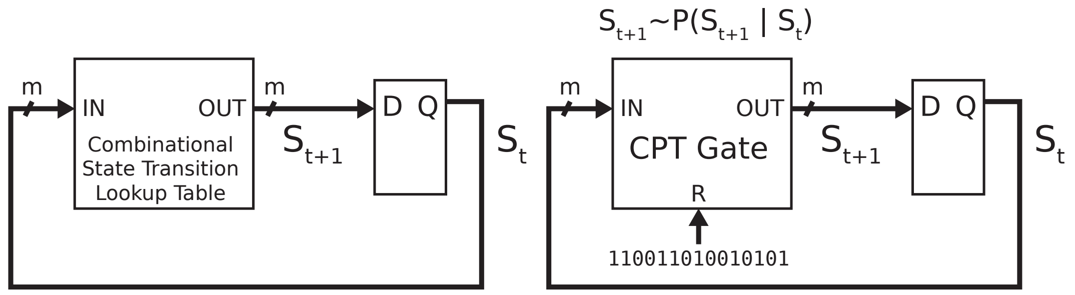
3.2 Design Idioms

3.2.1 Parallelism, trade-offs

Statistical 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 3-8), 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.

3.2.2 Stochastic FSMs

Finite state machines are a common engineering idiom in digital logic systems, and are used to control the evolution of a digital system through a series of states. The machine is in a single state at any given time, and that state at time t (along with, optionally, some additional inputs) determines the state of the machine at time $t + 1$. In what follows, our state machines will be examined as Moore machines (E. F. Moore 1956), in which the output of the state machine depends entirely on the current state of the system. Often such state machines are implemented via a combinational lookup table (the "state transition table") and a register for storing the current state (figure 3-9).


$S_{t+1} \sim P(S_{t+1} \mid S_t)$
m
IN
OUT
Combinational State Transition Lookup Table
m
S_{t+1}
D Q
S_t
m
IN
OUT
CPT Gate
R
110011010010101
m
S_{t+1}
D Q
S_t

(a) Deterministic Finite State Machine (b) Stochastic Finite State Machine (Markov chain)

Figure 3-9: a.) A traditional finite state machine b.) Stochastic finite state machines (Markov chains) can be constructed out of CPT gates by connecting the time-delayed output to the inputs, allowing $S_{t+1} \sim P(S_{t+1}|S_t)$.

Consider a stochastic CPT gate with an m-bit input and an m-bit output. We can replace the combinatiorial state transition table in the above FSM with a stochastic gate, thus making a stochastic finate state machine. At each clock period $X_t \sim p(X_t|X_{t-1})$, so the system evolves with Markovian dynamics – a Markov chain.

Approximate sampling with Markov chains

Of course, Markov chains form the basis of the widely-used Markov chain Monte Carlo (MCMC) approximate inference techniques. (see 2.4). MCMC tells us how to construct a Markov chain with any desired ergodic distribution – runing that markov chain over time will produce samples from that ergodic distribution.

Gibbs sampling is a MCMC method for producing joint samples from a distribution $p(a, b)$ if you can sample exactly from the conditional distributions

$$a \sim p(a|b) \tag{3.1}$$
$$b \sim p(b|a) \tag{3.2}$$
$$\tag{3.3}$$

We can do this by chaining two CPT gates together (figure 3-9). Thus we can use circuit elements which produce *exact samples* to build larger systems of Markov chains, which then give us (asymptotically) approximate samples from their ergodic (target) distribution.

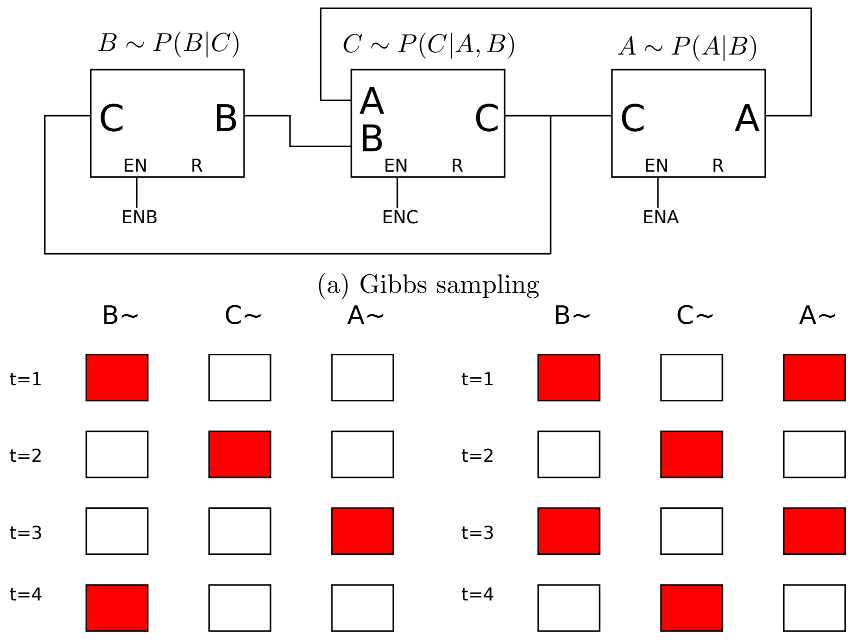
3.2.3 Approximate Sampling and Parallelization

Approximate sampling schemes such as Gibbs sampling make it easy to exploit the underlying conditional independence structure of a particular probability distribution.

Consider the distribution

$$p(a, b, c) = p(a|c)p(b|c)p(c) \tag{3.4}$$

To perform gibbs sampling, we must be able to sample from $a \sim p(a|c, b) = p(a|c)$,



(a) Gibbs sampling

(b) Serial

(c) Parallel

Figure 3-10: a.) A stochastic finite state machine enabling Gibbs sampling for the distribution $p(a, b, c) = p(a|c)p(b|c)p(c)$. b.) To operate the circuit, each sampler can be sequenced in turn. c.) Because of conditional independence, we can sample a and c simultaneously.

$b \sim p(b|c, a) = p(b|c)$, $c \sim p(c|a, b)$. If we connect three CPT gates appropriately (figure 3-10) we can construct a circuit to do just that. To Gibbs sample, we sample b, then c, then a in order.

Note that a and b are conditionally independent given c. Thus, as these two sites are conditionally independent, once we fix c we can sample them both *simultaneously*. This point bears repeating: conditional independence in the probabilistic model lets us sample in parallel. As seen in figure 3-10, we can sample A and B simultaneously, leading to a 33% reduction in total sampling time.

The ability to exploit parallelism like this will be a continuing theme of this thesis. Conditional independencies show up throughout probabilistic modeling, and along with them come opportunities for dense parallelism.

3.3 The Normalizing Multinomial Gate

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

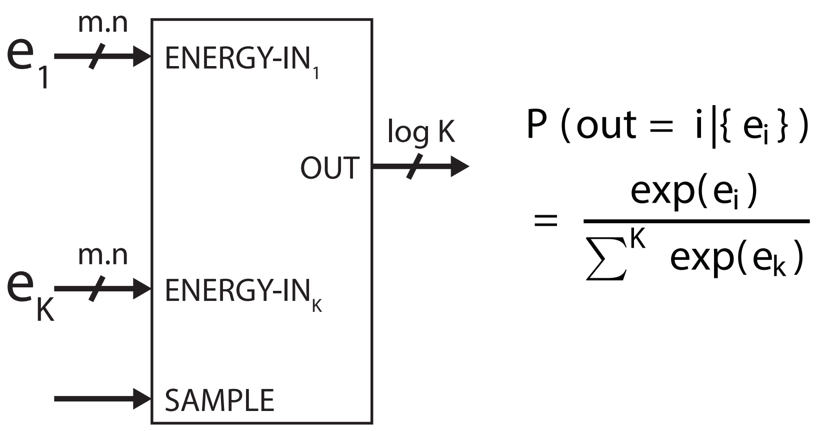


Figure 3-11: The normalizing multinomial gate: Accepts a variety of unnormalized values in log space representing $e_i = \log_2 p_i$.

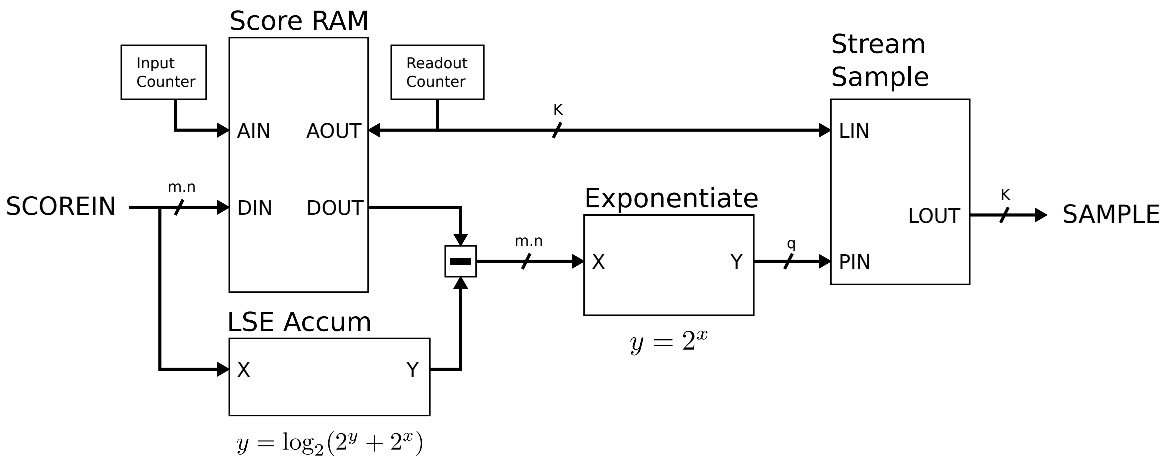


Figure 3-12: The normalizing multinomial gate: Accepts a variety of unnormalized values in log space representing $\log_2 p_i$. It performs the normalization step, converts into real space, and feeds into the stream sampling unit.

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 multinominial 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 $x \in \{1 \ldots K\}$. We can use the normalizing multinomial gate to then sample from $p(x|\cdot)$.

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 in section 3.5.

3.3.1 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 3-13.

3.3.2 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.

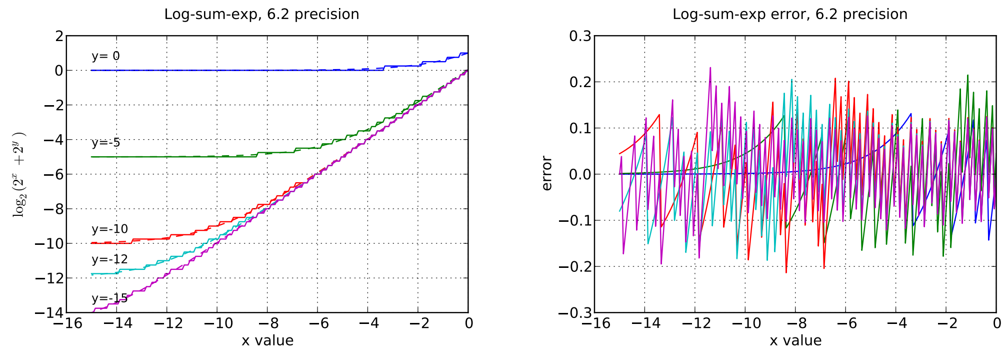
3.3.3 Random Starts to remove bias

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.

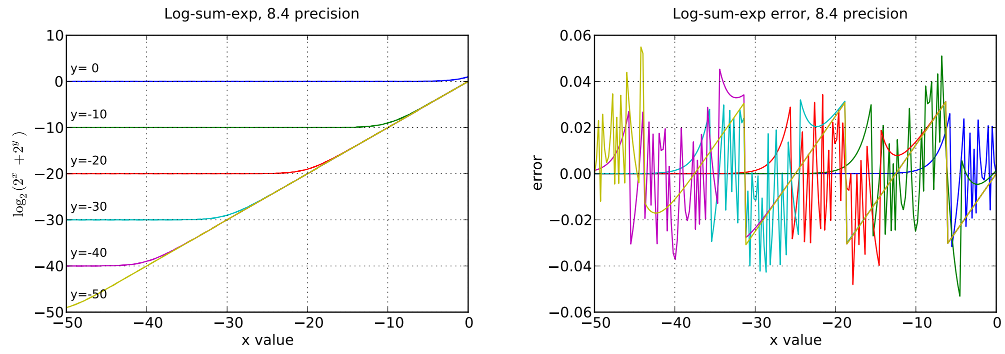
To remove this systematic bias we circularly permute the probability vector before computing the CDF and sampling. A circular permutation can be easily implemented by randomly sampling the starting position and taking care to wrap around at the end of the array.

3.3.4 Resources

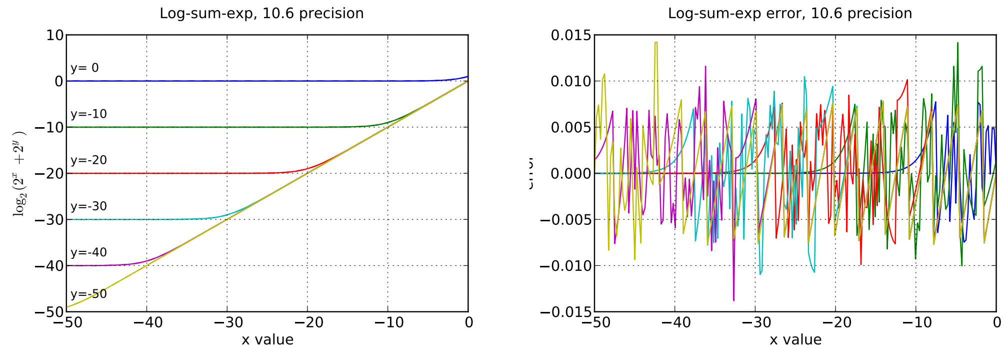
Figure 3-14 shows how look-up table and flip flop utilization vary as a function of both internal bit precision and the maximum arity for the multinomial sampler. Going from the smallest 6-bit, $k = 16$ unit to the largest 12-bit $k = 1024$ unit increases the combinational logic requirements by four times and doubles the amount of stateful silicon logic.



(a) m=6, n=2



(b) m=8, n=4



(c) m=10, n=6

Figure 3-13: "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.

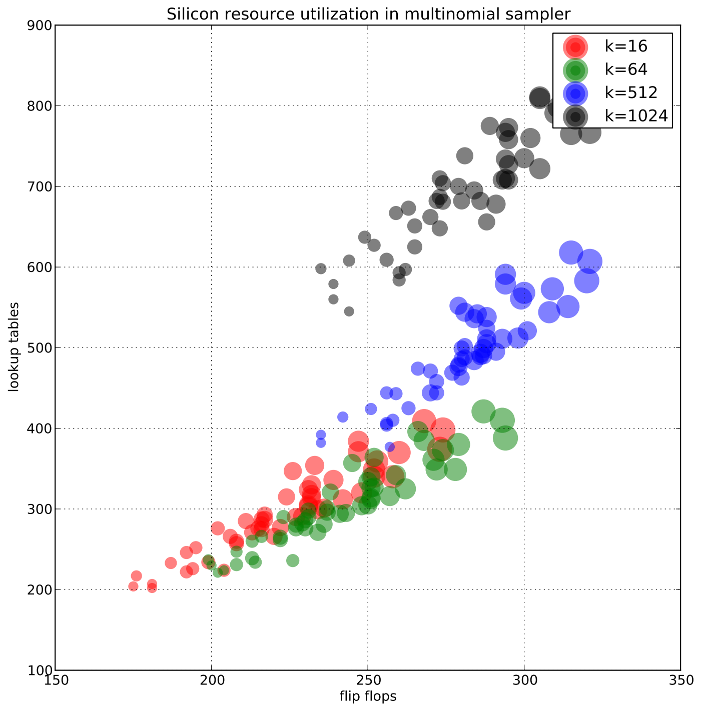


Figure 3-14: Resource utilization for the Multinomial Sampler, for samplers configured with several different values of K, and bit precisions varying as above. Larger circles indicate higher bit precisions, ranging from six bits to twelve.

3.4 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 $x_1, x_2, \ldots, x_n$, how close is $P(x_i)$ 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

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 3.4: 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.

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

```
$x \leftarrow$ SEED[0]
$y \leftarrow$ SEED[1]
$z \leftarrow$ SEED[2]
$w \leftarrow$ SEED[3]
$tmp \leftarrow x \oplus$ LEFT-SHIFT(x, 15)
$x \leftarrow y$
$y \leftarrow z$
$z \leftarrow w$
$w \leftarrow (w \oplus$ RIGHT-SHIFT(w,21)) $\oplus$ ($tmp$& 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 phenominal 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 multple PRNGs with different seeds.

3.5 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 3.4). 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 3-15, 3-16, and 3-17 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 3-15c, 3-16c, and 3-17c 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.

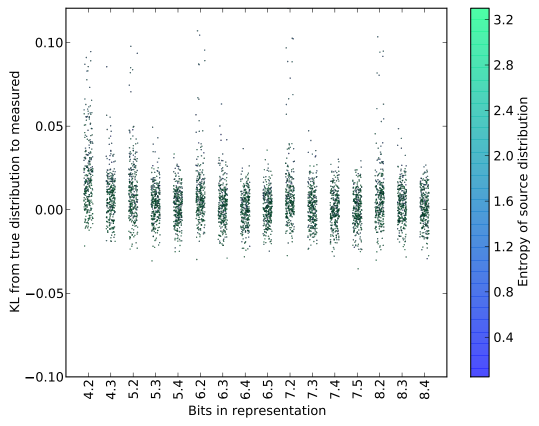
3.6 Resource Utilization

But performing large floating-point operations consume a massive quantity of silicon resources when compared to our stochastic sampling elements. As figure 3-18 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).

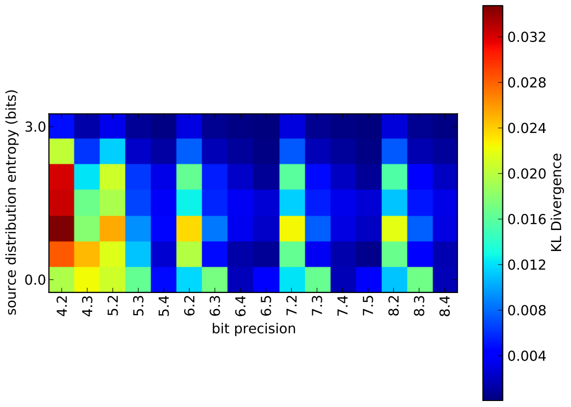
Also note that while the XORShift RNG takes up more silicon area than some of the other sampling elements, a single PRNG instance can supply entropy to dozens of stochastic circuit elements.

3.7 Next Steps

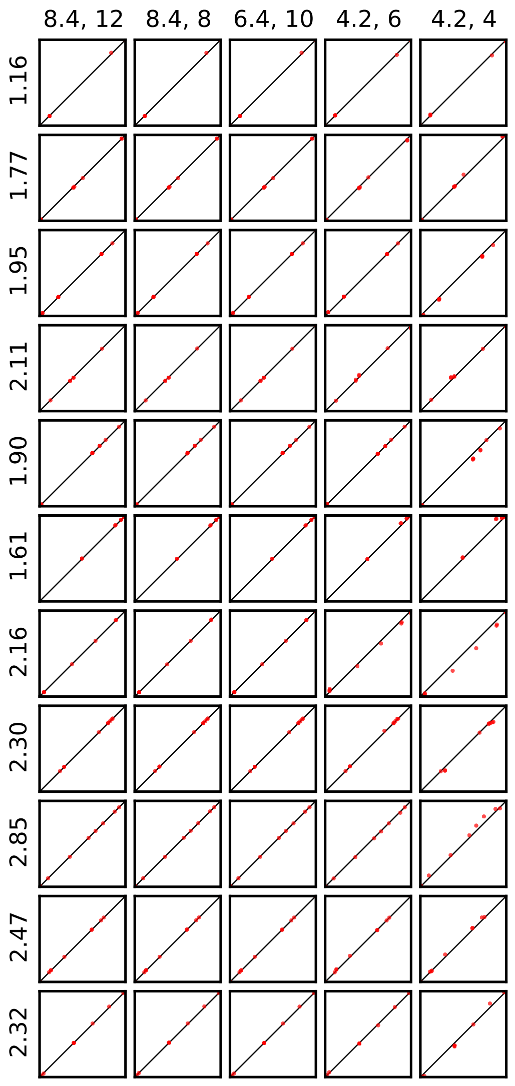
We've shown that digital stochastic logic recovers Boolean digital Logic in the deterministic limit, while preserving the principles of abstraction and composition. Various sampling primitives, ranging from the theta gate up through the generic multinomial sampler, enable



(a) KL vs bit precision

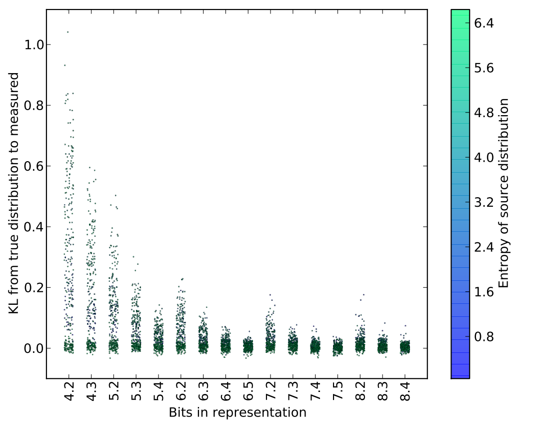


(b) Entropy vs bit-precision

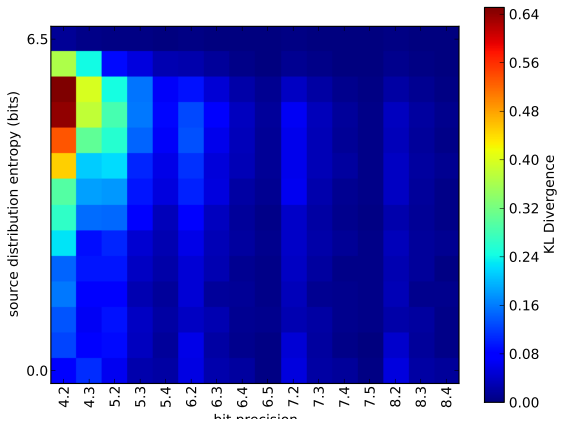


(c) QQ plot, true vs circuit

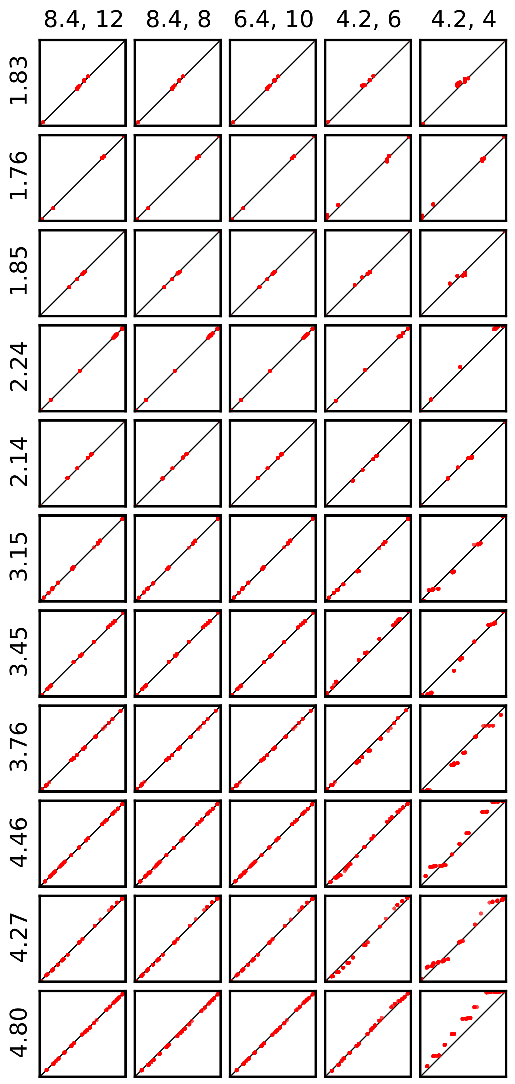
Figure 3-15: The effects of bit precision on KL divergence for a $K = 10$ multinomial sampling gate, a.) KL vs bit precision, b.) heatmap showing regions of entropy/bit-precision with high KL, and c.) example distribution QQ plots. Each column is a different bit precision (labeled at top) and each row 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.



(a) KL vs bit precision

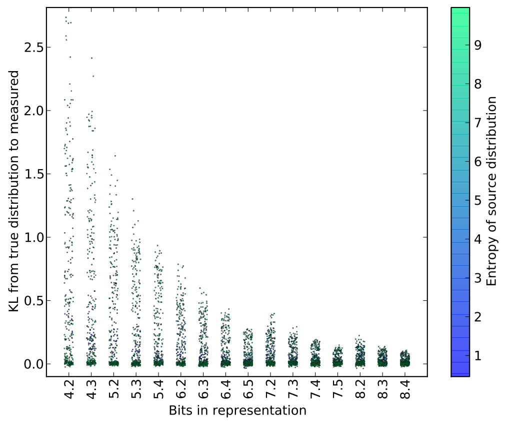


(b) Entropy vs bit precision

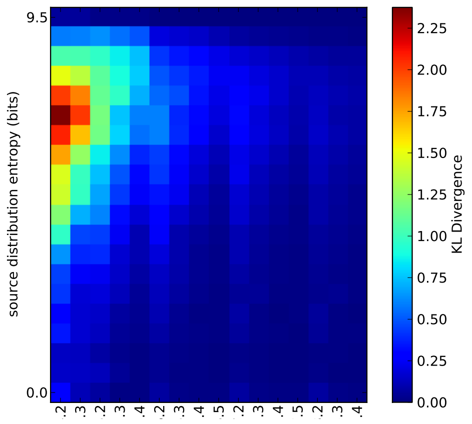


(c) QQ plots, true vs circuit

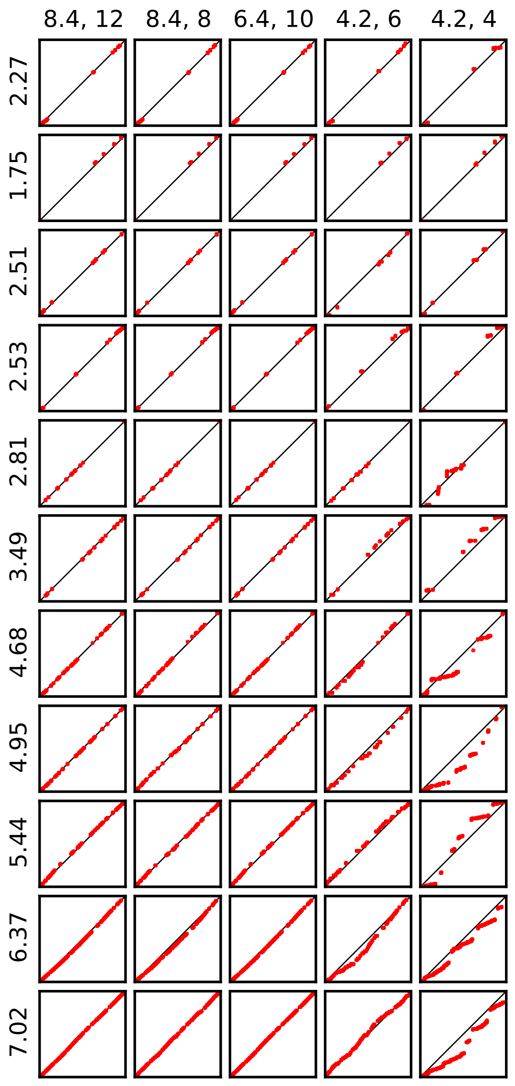
Figure 3-16: The effects of bit precision on KL divergence for a $K = 100$ multinomial sampling gate, a.) KL vs bit precision, b.) heatmap showing regions of entropy/bit-precision with high KL, and c.) example distribution QQ plots. Each column is a different bit precision (labeled at top) and each row 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.



(a) K=1000



(b) K=1000



(c) QQ plots for example distributions

Figure 3-17: 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.

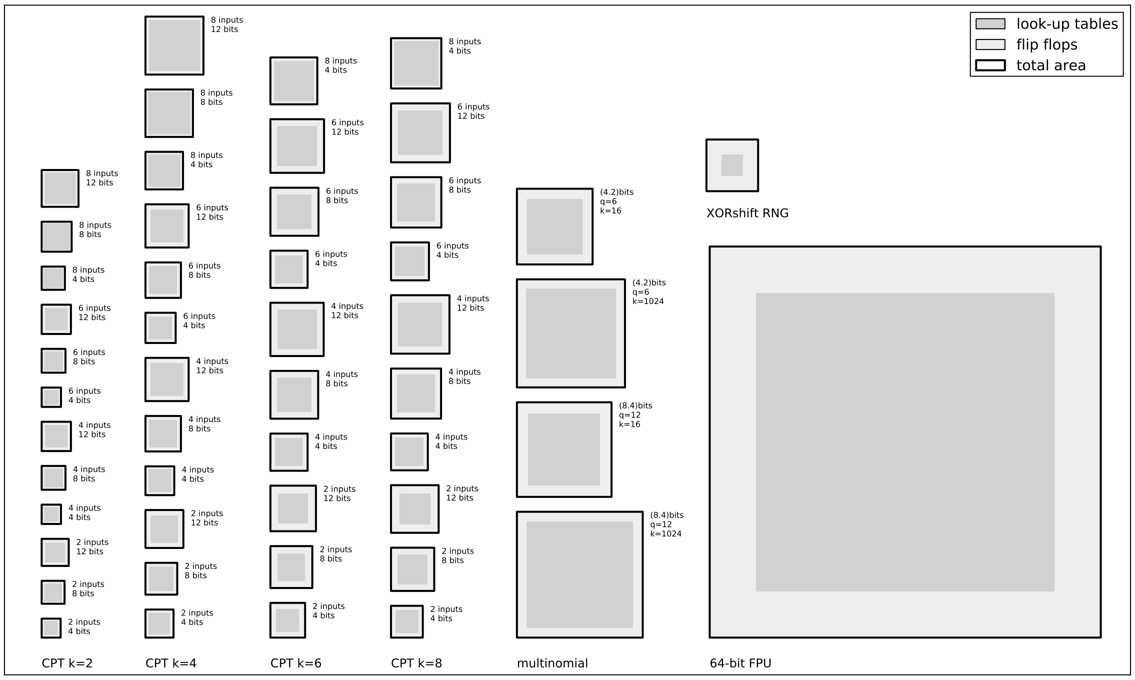


Figure 3-18: Comparing approximate silicon area between a 64-bit IEEE-754 floating point unit and various examples of our stochastic gates, including the Conditional Probability Table gate and the normalizing multinomial gate. Dark grey areas are the purely-combinational lookup tables, lighter areas are the stateful flipflops. See text for an explanantion of bit precision designations.

the construction of systems for exact inference and the construction of Markov chains with desired ergodic distributions.

In the following chapters, we'll show how various ways of expressing complex probability distributions and probabilsitic models can be automatically transformed into stochastic circuits, how resources can be shared between sampling units, and how we can even build systems for sampling which learn the number of random variables online.

Chapter 4

Causal Reasoning, and Automatic Circuit Construction

Causal reasoning – drawing conclusions about probable causes from their observed, ambiguous effects – is a hallmark capability of intelligent systems. Nowhere is this inductive learning more impressive than in children, where young toddlers are capable of rapidly building causal models of the world as they develop.

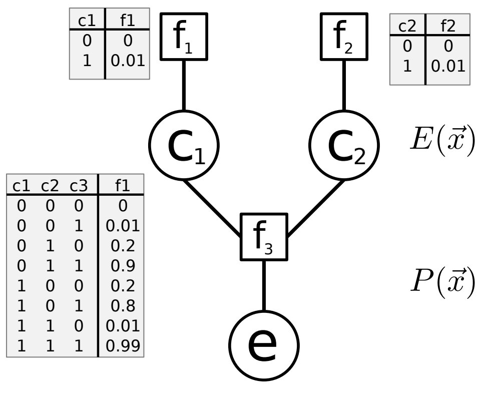
Developmental psychologists have argued recently that children build “causal maps” of the world (Gopnik, Glymour, et al. 2004), and model these maps as Bayesian networks. Bayesian networks are a formalism developed (Pearl 1988) to provide a graphical language for describing this process of inductive causal reasoning. Indeed, Bayesian models of child cognitive development have become something of a cottage industry in recent years (Gopnik and Joshua B. Tenenbaum 2007). Even rats have been shown to perform causal reasoning consistent with Bayes nets (Blaisdell et al. 2006).

4.0.1 Architectural Motivation

Because of our abstraction and composition laws, automatic transformation (compilation) from a high-level description of a probabilistic problem, such as a Bayes net, to a synthesizable circuit is possible. We built this compiler to argue that our approach is general – probabilistic models can be synthesized down to hardware using a discrete set of rules simple enough for a computer to implement.

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,



c1	f1
0	0
1	0.01

c2	f2
0	0
1	0.01

c1	c2	c3	f1
0	0	0	0
0	0	1	0.01
0	1	0	0.2
0	1	1	0.9
1	0	0	0.2
1	0	1	0.8
1	1	0	0.01
1	1	1	0.99

$$\begin{aligned} E(\vec{x}) &= \sum_i f_i() \\ &= f_1(c_1) + f_2(c_2) + f_3(c_1, c_2, e) \\ P(\vec{x}) &= \frac{1}{Z} e^{-E(\vec{x})} \\ &= \frac{1}{Z} e^{-f_1(c_1) - f_2(c_2) - f_3(c_1, c_2, e)} \end{aligned}$$

Figure 4-1: 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 $\frac{1}{Z}e^{-E(\mathbf{x})}$.

for causal medical diagnosis, and show how we can programmatically pick at compilation time which subset of the variables are fixed.

4.1 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 4-1). Discrete-state factor graphs were the first class of probabilistic models supported by Kevin Murphy's excellent Bayesian Network Toolbox (Kevin P. 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 (3.2.3) 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.

4.1.1 Discrete-output CPT-sampling gate

Ultimately, all sampling units for all nodes are compiled down into discrete-output conditional probability gates, described in section3.1.4. Neighboring variables are connected to the input lines of the CPT gate, and output samples are generated conditioned on these values.

Listing 4.1: 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])
```

4.2 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 4.1 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.

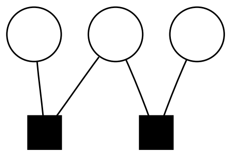
Factors are specified as python functions that return an energy (larger values are less likely). Note that the functions can perform arbitrary computation, as they are only evaluated in the course of compilation, generating a lookup table for later synthesis.

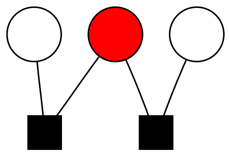
The compilation steps are as follows. We color the initial factor graph to identify parallelization opportunities. Nodes of the same color are conditionally independent, and thus we can do inference on them simultaneously. We then annotate the variables with the number of bits necessary to represent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

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.

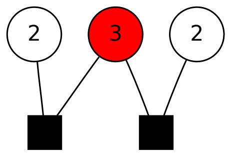
Taking the simple SDK from before, we annotate it with the "kernels" that will ultimately be performing inference. The primary kernel used is an "enumerated Gibbs" kernel which will perform Gibbs sampling of a particular target node. State variables labeled "observed" do not have kernels attached.

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

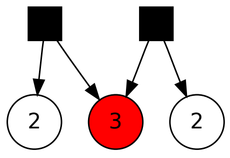

(a) Initial Factor graph



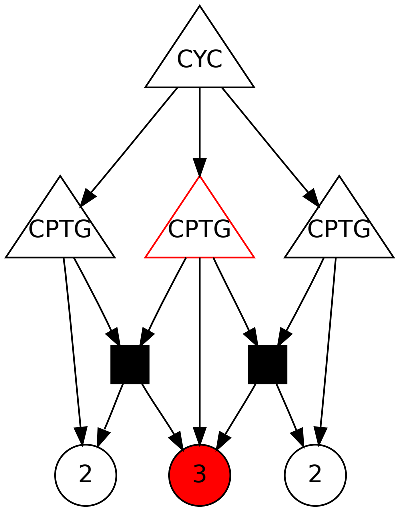
(b) Color



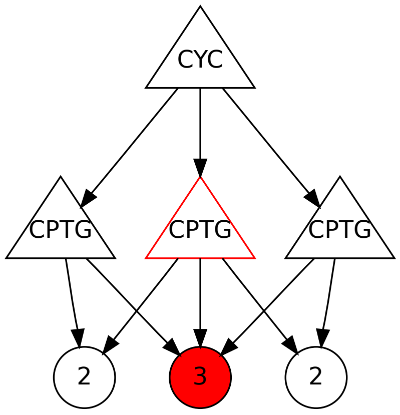
(c) Annotate with bit precision



(d) Convert to SD(K)



(e) Add Kernels



(f) Compile away Densities

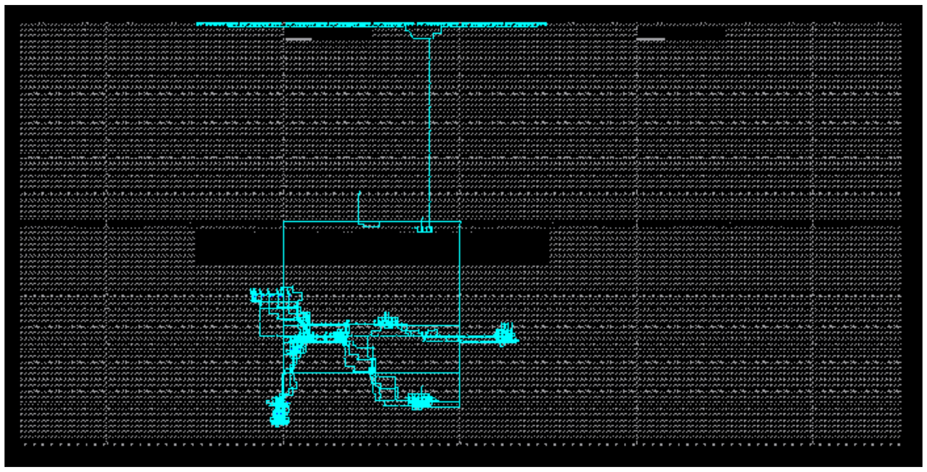

(g) FPGA Result

Figure 4-2: 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.

"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 3.1.4 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.

4.3 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.

4.3.1 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 3.4) 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.

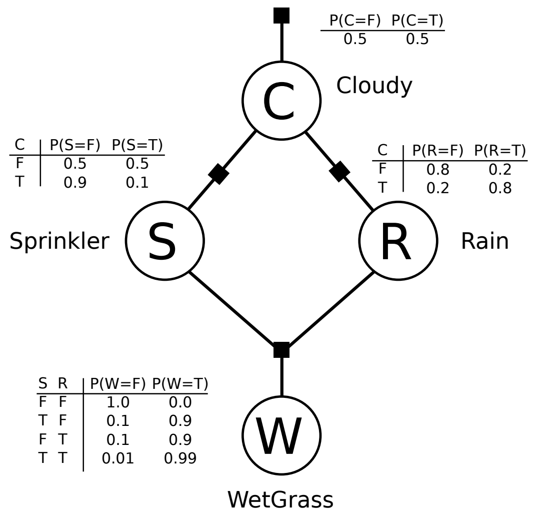


Figure 4-3: “rain” factor graph.

4.4 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.

4.4.1 Rain

We adopt Kevin Murphy’s (Kevin P. 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), sprinker (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 4-3) and describe it efficiently in Python (listing 4.2).

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 4.1, 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 4.1):

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

Listing 4.2: 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))
```

that the sprinker is on? Because of the rain, the posterior probability of the sprinkler being on goes down.

4.4.2 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.

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

The factors are thus all homogenenous, and of the form

$$f(x,x') = \begin{cases} 0 & \text{if } x = x' \\ J & \text{if } x \neq x' \end{cases} \tag{4.1}$$

The energy of the total ising system is thus

$$E(X) = \sum_{x,x' \in N(x)} f(x,x') \tag{4.2}$$

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 4-4 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.

4.4.3 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 diaagnoses 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 4-6) to a factor-graph representation (figure 4-7) 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 4-9, 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

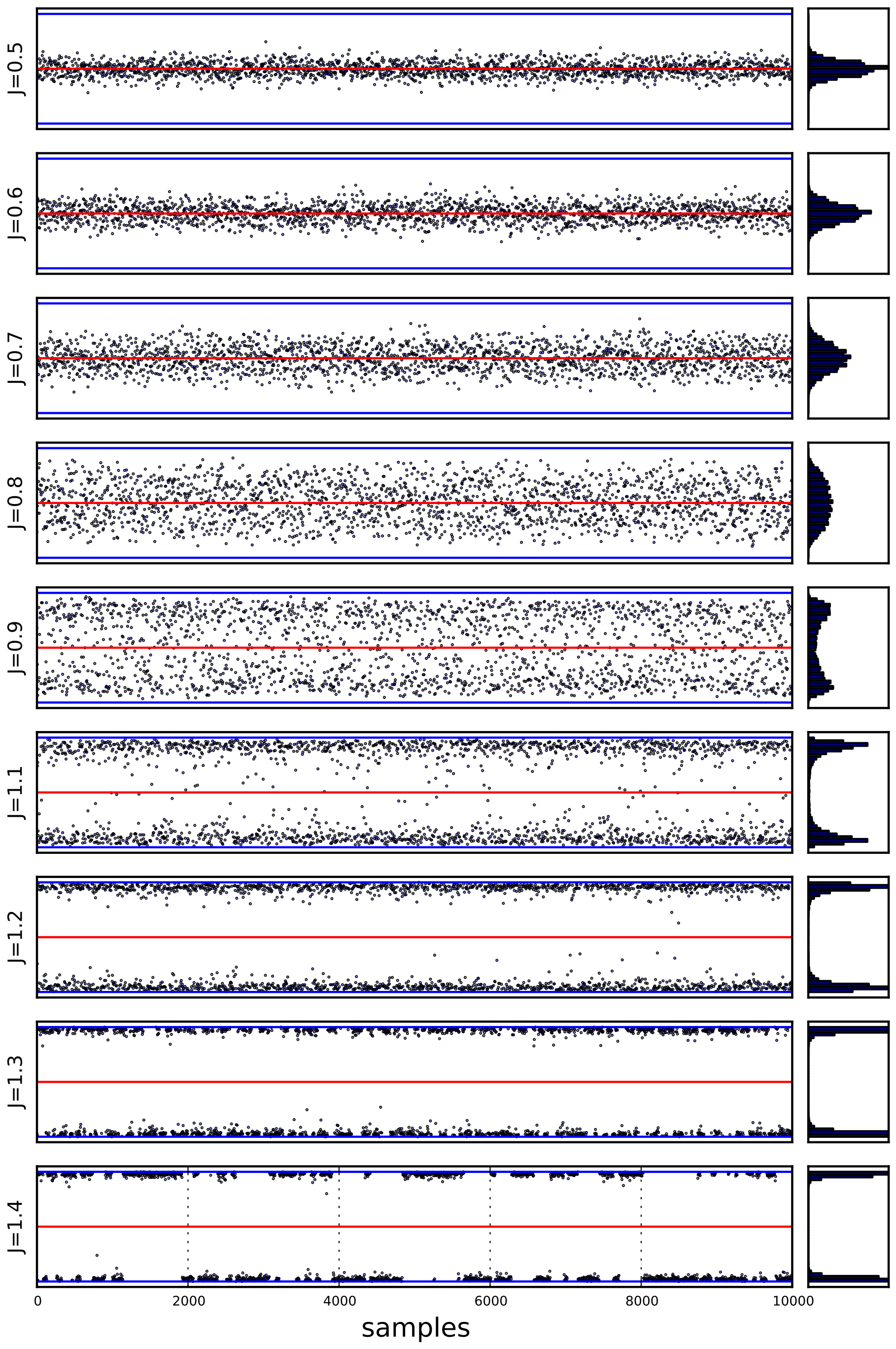
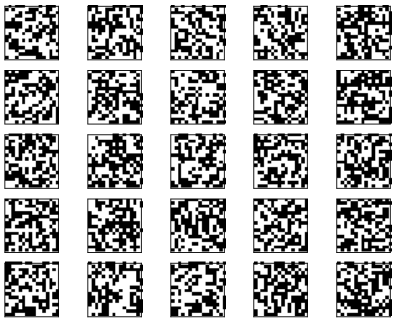
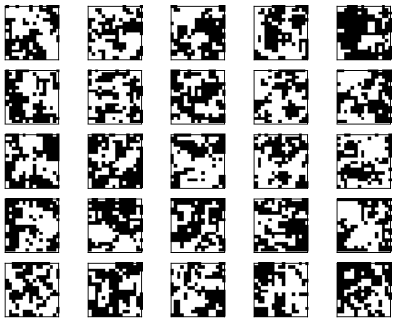
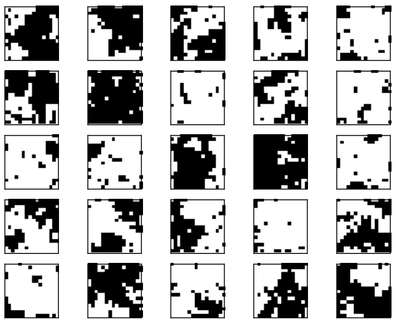


Figure 4-4: Samples from 8-bit 256-node Ising circuits with different coupling strengths. As the coupling strength increases, the distribution becomes more bimodal.


(a) J = 0.0


(b) J = 0.6


(c) J = 1.0

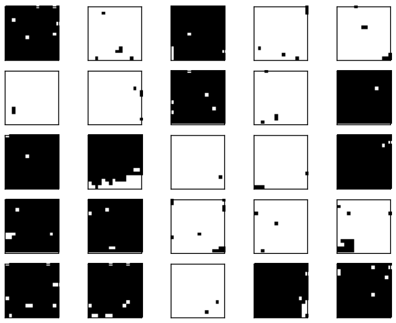

(d) J = 1.4

Figure 4-5: Example samples from the compiled Ising model for four different coupling strengths.

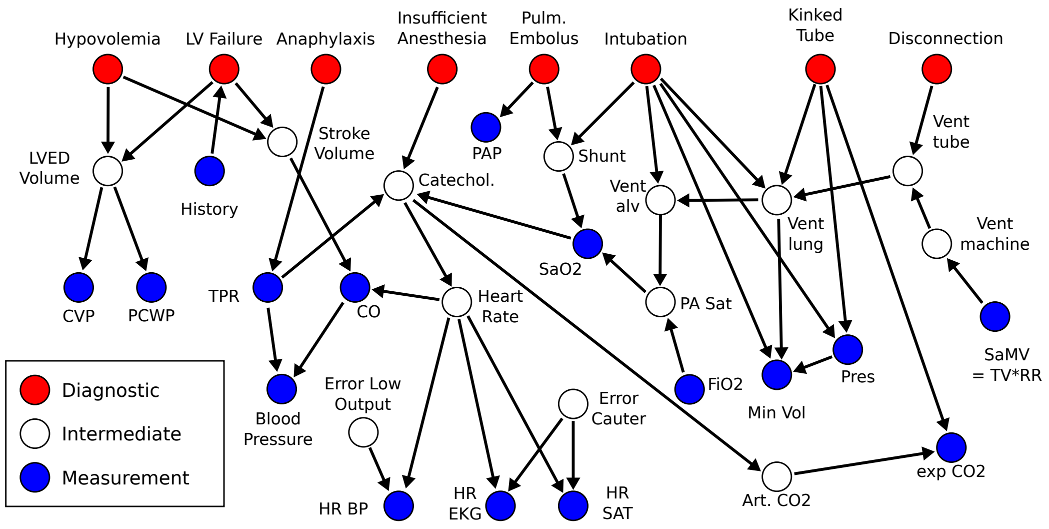


Figure 4-6: The ALARM (A Logical Alarm Reduction Mechanism) Network, with 8 diagnoses, 16 findings, and 13 intermediate variables.

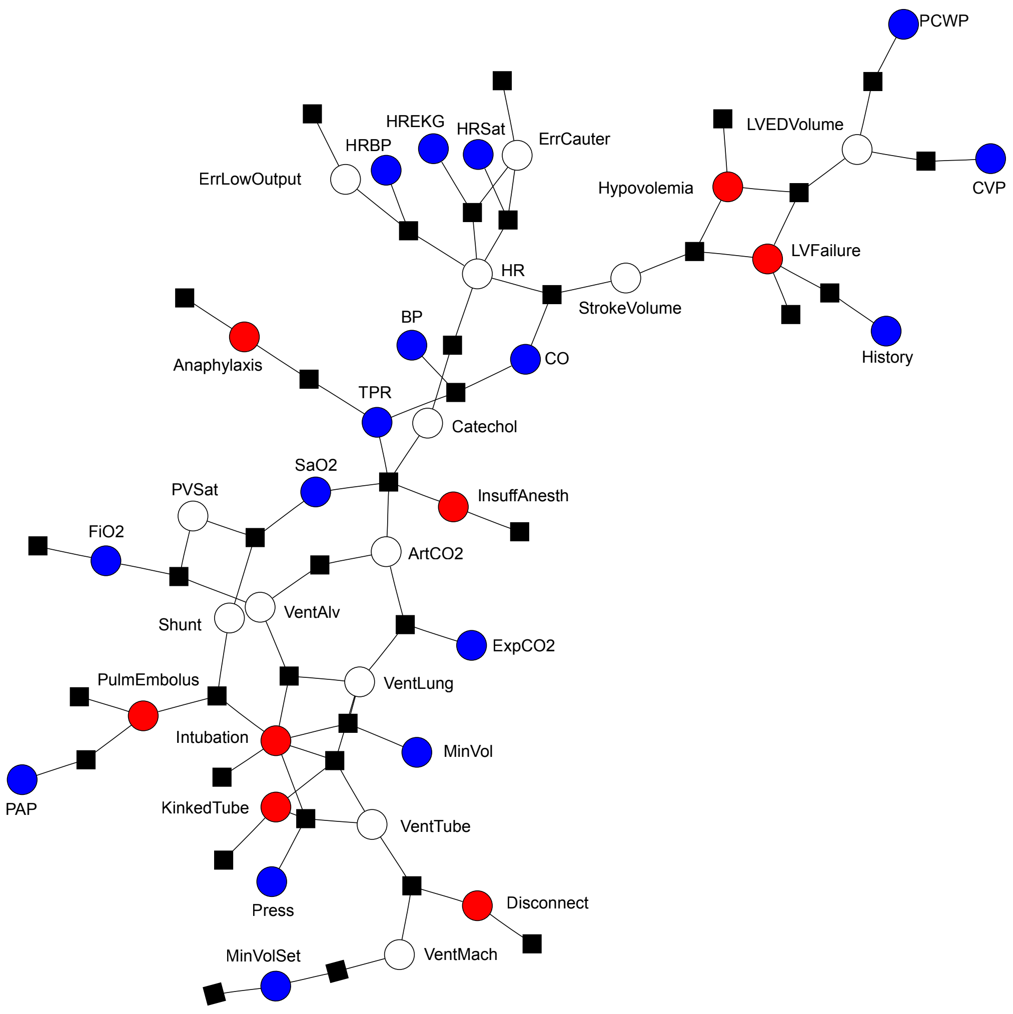


Figure 4-7: The ALARM Factor Graph generated from the Alarm Bayes Network.

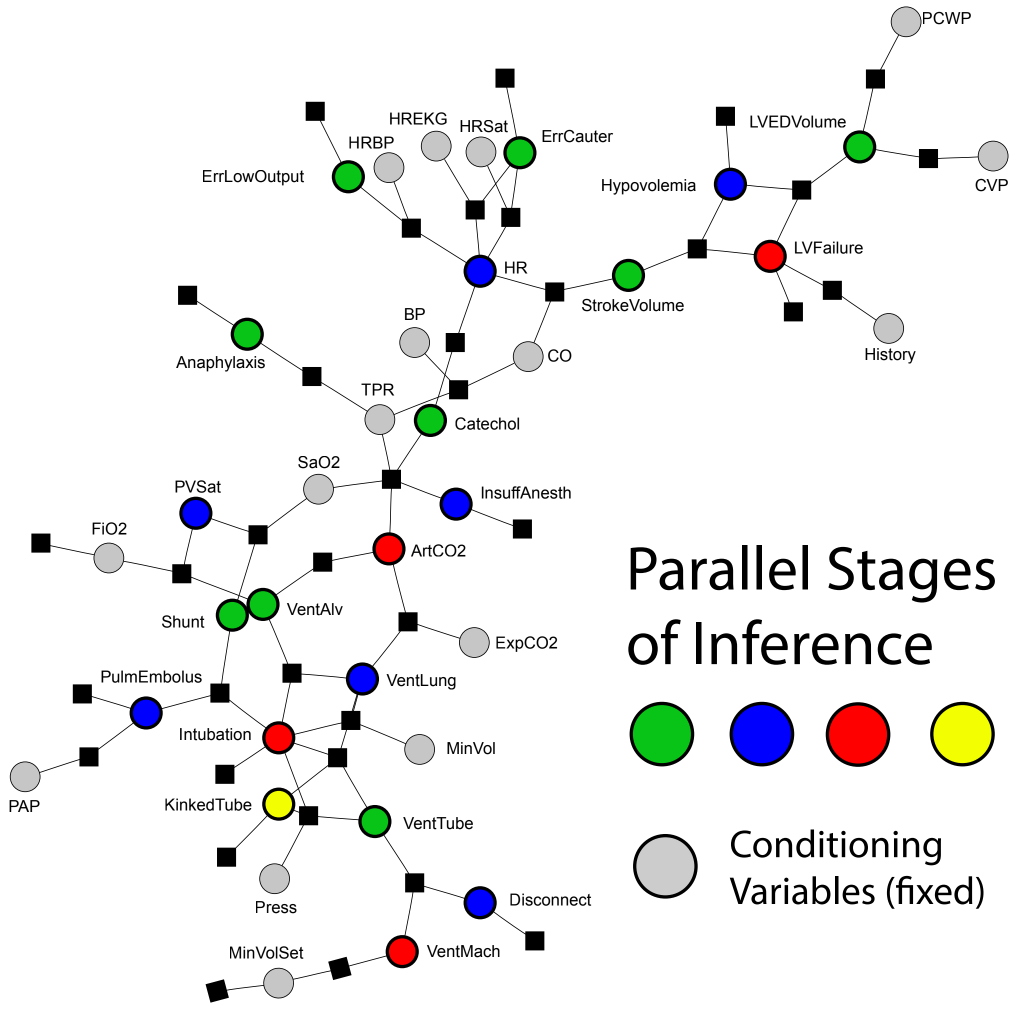


Figure 4-8: The ALARM factor graph from figure 4-7 colored for parallel execution.

measurements are in the “normal” range (figure 4-10, no diagnosis is suggested. Particular settings of measurements adjust the probabilities of particular symptoms.

4.5 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 complied 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 an 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.

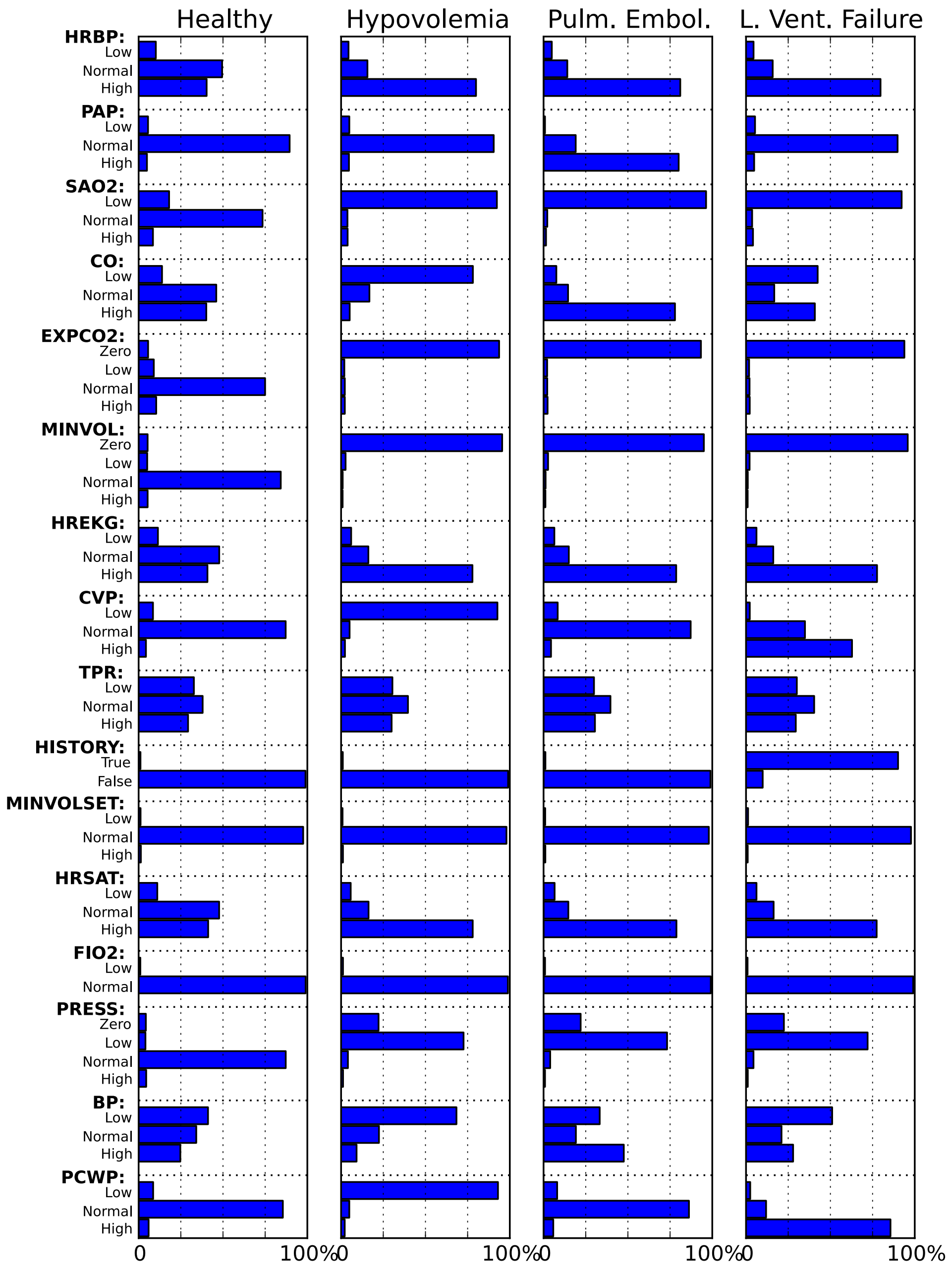


Figure 4-9: The ALARM Network: Marginal distributions for measured (symptom) variables given particular diseases.

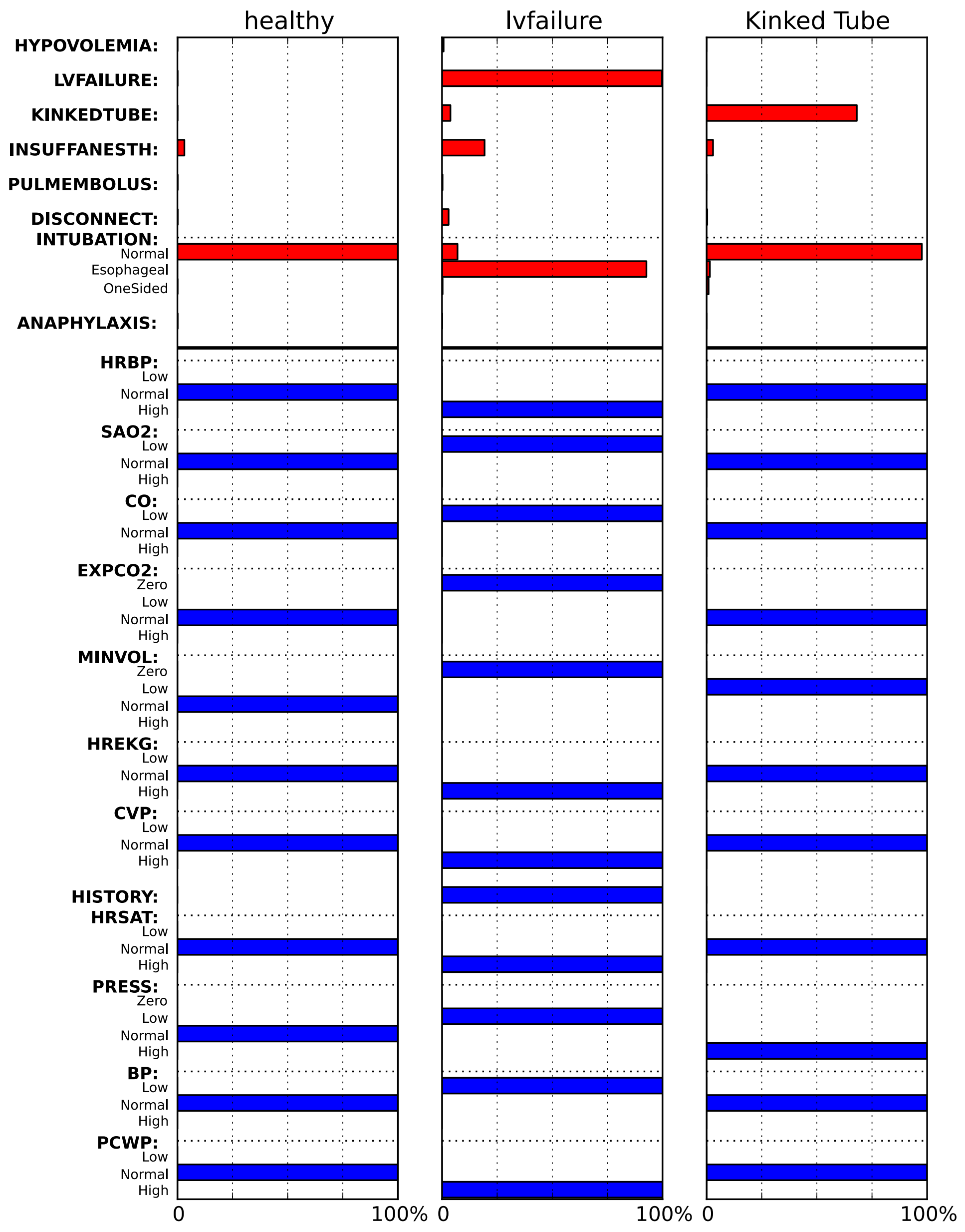


Figure 4-10: The ALARM Network: Marginal distributions for diagnoses based upon particular settings of the measured symptoms.

Chapter 5

Low-level vision, Lattice Factor Graphs, and Virtualization

Vision is a classic problems in machine intelligence, and many animals use vision as their primary sensory modality. Low-level vision tasks perform inference at the pixel or near-pixel level – examples include computing the motion field (optical flow), estimating depth from binocular images, (stereo correspondence), and removing noise in corrupted images. These problems have a similar structure: combining per-pixel information about the outside world, with a belief that there's some hidden (latent) cause that we need to infer, and that those causes vary smoothly. It's not surprising that they were originally tackled via soft constraint satisfaction (D Marr and T Poggio 1976), and more recently through probabilistic inference (Scharstein, Szeliski, and Zabih 2001). Here we model them using a class of factor graphs which exhibit per-pixel lattice structure, an approach which dates back to the original Gibbs sampling work of (S. Geman and D. Geman 1984).

The circuit architectures described previously have been fully space-parallel, with topologies and resource allocations that closely mirror the underlying probabilistic model structure. As a result, state values have been colocalized with their associated stochastic elements.

This approach does not scale well to probabilistic models with thousands of state nodes or mathematically-intensive density calculations, due to silicon area and power constraints. Models with substantial homogeneity, however, admit the possibility of segregating state and stochastic sampling elements. The reuse, or *virtualization* of particular subsets of stochastic elements for a given model, enables the construction of architectures for solving problems much larger than could be admitted via naive space-parallel designs.

We present a virtualized circuit for a particular class of probabilistic model : lattice factor graphs with discrete latent state values, per-pixel external field densities, and homogeneous latent pairwise densities. This class of factor graphs arises frequently in problems of low-level vision and video processing, including depth estimation from stereo images and optical flow. This model class generalizes the previously-described Ising model (4.4.2), which will be used as a point of architectural comparison.

First, we describe the detailed structure of the lattice Markov random field and the associated inference scheme. Then we describe the generic architecture, and how this architecture lends itself to model-specific stochastic circuits and a weakly-reconfigurable probabilistic processor for inference in factor graphs. Finally, we show performance, silicon resource, and quality results for the aforementioned problems.

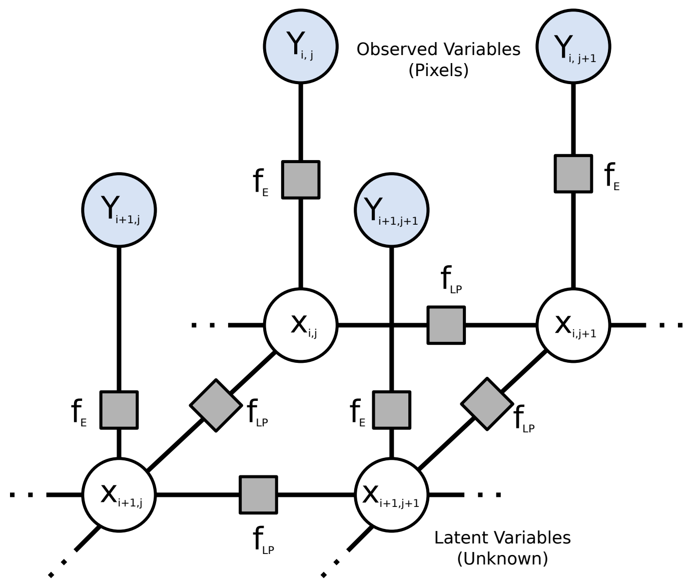


Figure 5-1: Low-level vision lattice factor graph. Some lattice of unobserved, *latent* state variables $X_{i,j}$ generate observed values $Y_{i,j}$ through an *external* potential f_E. The relationship between adjacent $X_{i,j}$ and $X_{i',j'}$ is constrained by the *latent, pairwise* potential f_{LP}. Note that in this formulation, both f_{LP} and f_E are homogeneous in the graph.

5.1 Low-level Vision Factor Graph

A low-level vision factor graph (figure 5-1) 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 similar values. Thus

$$P(X_{i,j} = x) \propto f_E(x, y_{i,j}) \sum_{x' \in \text{Neighbor}(X_{i,j}))} f_{LP}(x, x') \tag{5.1}$$

Here I am 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.

5.2 Resource virtualization and parallelization

5.2.1 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

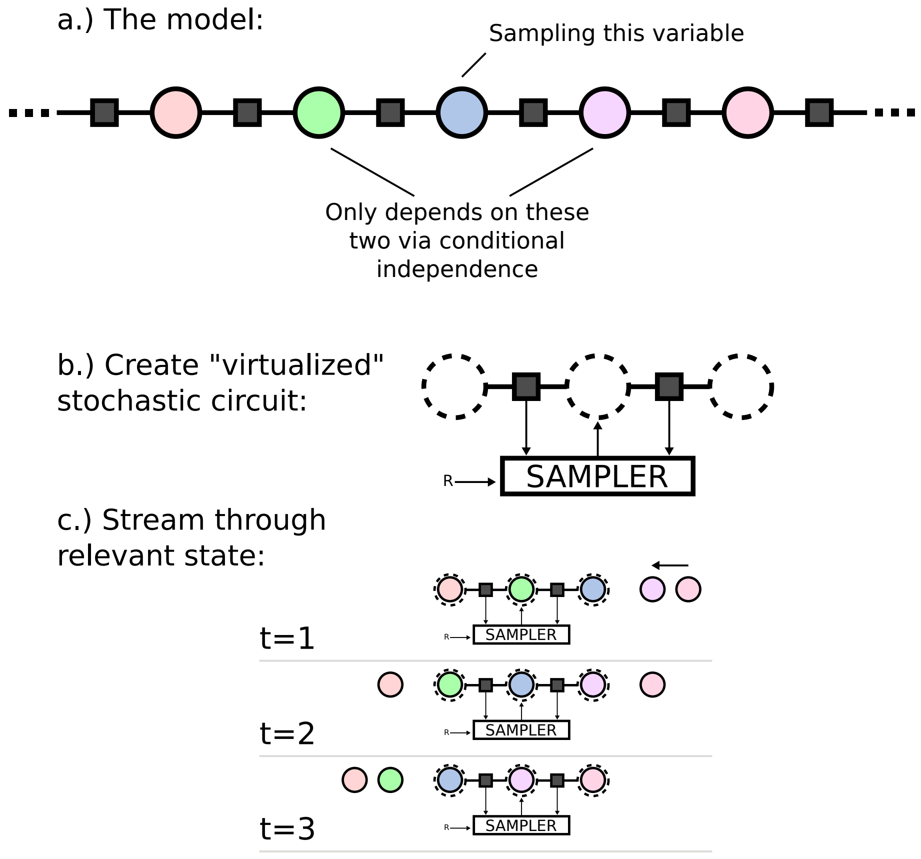

a.) The model:
Sampling this variable
Only depends on these two via conditional independence
b.) Create "virtualized" stochastic circuit:
SAMPLER
R
c.) Stream through relevant state:
t=1
t=2
t=3

Figure 5-2: Virtualization example for linear-chain factor graph. a.) The model consists of a repeating chain of variables connected pairwise by homogenenous 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 vitualized circuit then can sample values for the entire factor graph by streaming in the variable values, performing the sample, and writing the output.

5-2 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]

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

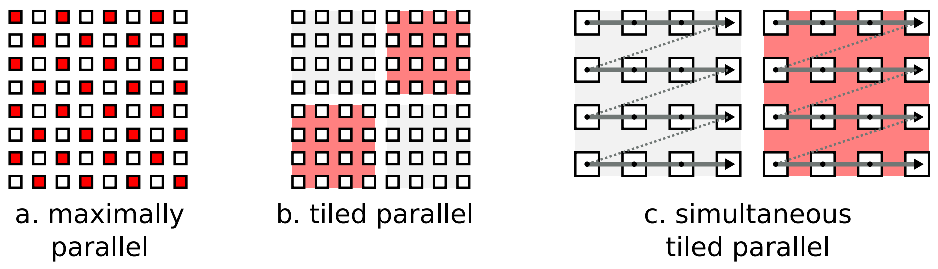


Figure 5-3: Parallelization via conditional independence. a. Maximally parallel, derived from the graph coloring. All red sites can be sampled simultaneously b. The naive coarsening, tiled parallel, in which inference in the same color of tile can be performed simultaneously c. Simultaneous tiled parallel, where we carefully time the sequential sampling within a tile to guarantee correctness, allowing sampling on tiles to happen simultaneously.

5.2.2 Parallelization

I have previously shown that factor graphs with conditional independencies provide extensive opportunities for paralellization. The granularity of that parallelization can be varied. In 5-3a, 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 5-3b, resulting in"tiles" of sequential serial inference, where inference can take place simultaneously in similarly-colored tiles.

I go one step further here, as shown in figure 5-3c. 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.

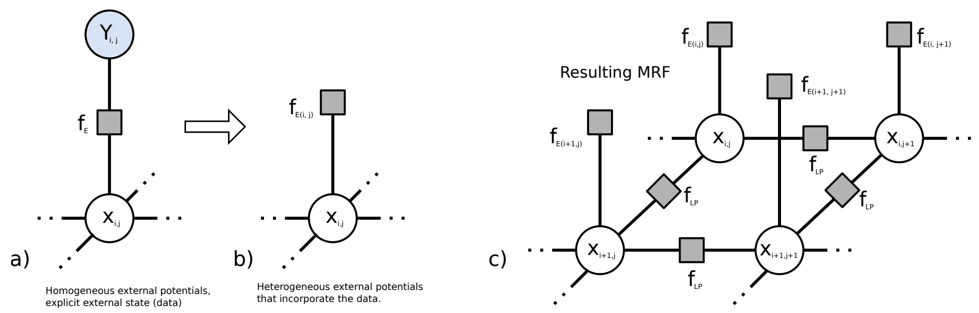
5.3 Circuit Architecture

Here I 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 (see section 2.4.2) on a subtile of the total lattice factor graph.

Reencoding the factor graph (figure 5-4), 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.

5.3.1 Gibbs Tile

The Gibbs Tile virtualizes a single normalizing multinomial gate stochastic element (section 3.3 over a rectangular subregion of the factor graph (figure 5-5), 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



a)
Homogeneous external potentials, explicit external state (data)
b)
Heterogeneous external potentials that incorporate the data.
c)
Resulting MRF

Figure 5-4: 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.

table density. The external field density is implemented as a runtime-reprogrammable SRAM, An XORShift pseudo-random number generator (section 3.4) 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. 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).

Algorithm 2 Gibbs Tile Operation

```
for all $v$ in VirtualizedSet do
    $n_i$ ← LookupNeighbors(v)
    offset ← ComputeOffset(v)
    newv ← GibbsCoreSample($n_i$, offset)
    v ← newv
end for
```

5.3.2 Pixel State Controller

The pixel state controller (PSC in figure 5-6) 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 particlar tile’s edge state for interruption-free lookup by neighboring tiles (labeled “Adjacent state IO” in figure 5-5).

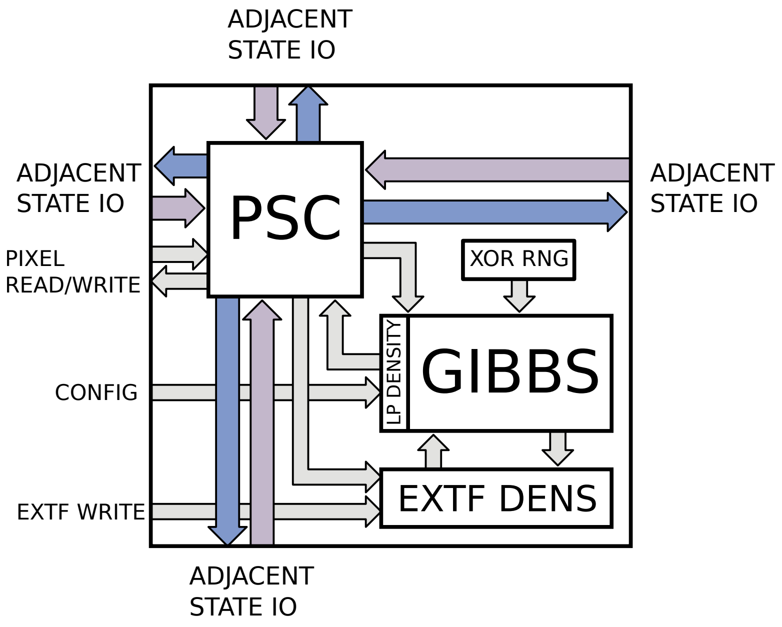


Figure 5-5: Gibbs tile, consisting of the pixel state controller, the Gibbs unit with local potential density, the external field density lookup table, and the PRNG. The tile communicates with neighboring tiles via adjacent state IO.

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

5.3.3 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.

5.3.4 Gibbs Core

Algorithm 3 Gibbs Core Sampling algorithm

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

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 (section 3.3) normalizes and samples from the resulting score ta-

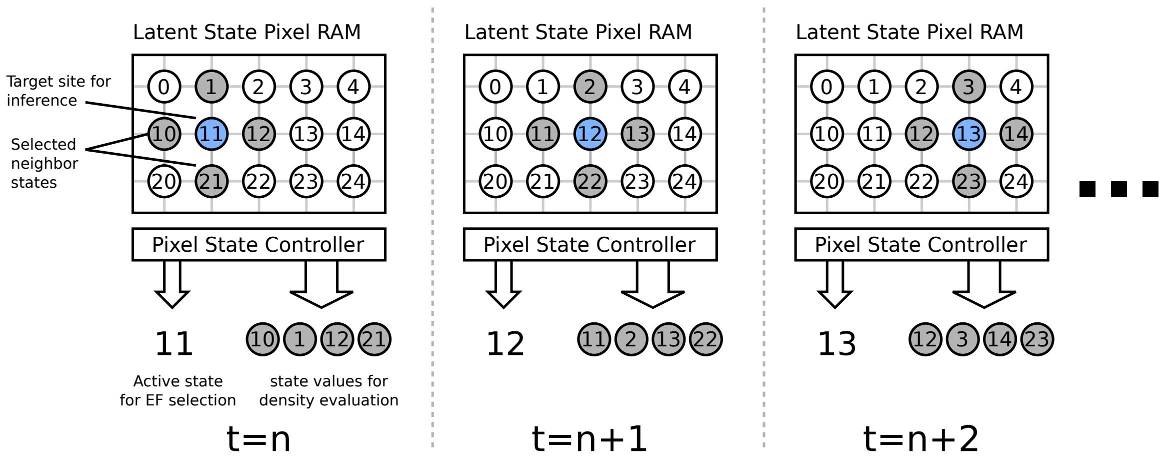

Latent State Pixel RAM
Target site for inference
Selected neighbor states
Pixel State Controller
11
Active state for EF selection
state values for density evaluation
t=n
12
t=n+1
13
t=n+2

Figure 5-6: 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.

ble. The Gibbs Core can also temper the resulting scores, allowing for annealing and tempering MCMC operations.

5.3.5 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.

5.3.6 Configuration Parameters

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

Figure 5-8 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.

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 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 5-9 shows empirically-measured tile performance for a $8x8$ 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.

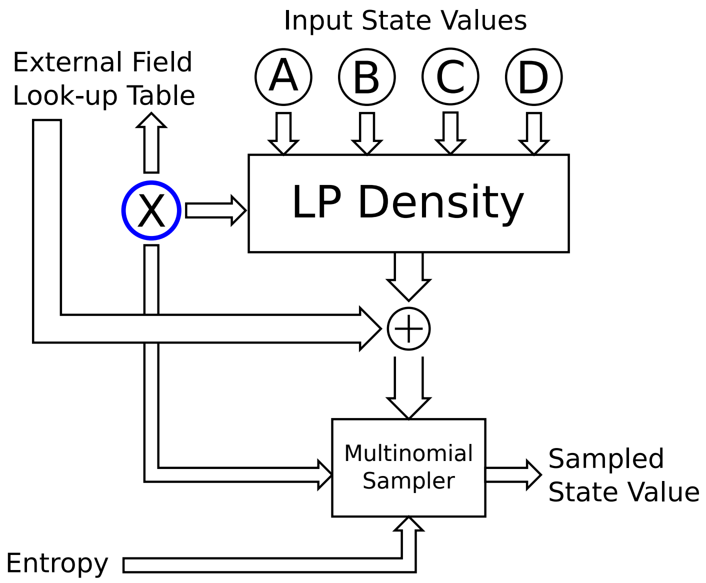


Figure 5-7: Gibbs core enumerates through possible values for this site's latent state variable, setting X to each value and evaluating the LP density. The score from the LP density and the external field lookup table are summed. The multinomial sampler takes these unnormalized scores and produces a sampled state value.

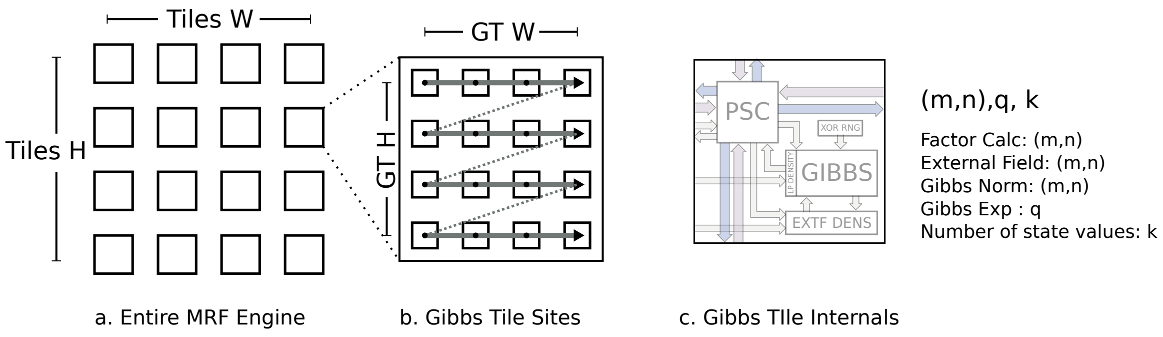


Figure 5-8: Stochastic Video Processor parameters. a.) Tiles H and Tiles W control the height and width of the engine, in tiles. b.) Each Gibbs Tile samples GT H by GT W sites. Within the tile, $(m, n), q$)-bit-precision computations are performed, with each variable taking on k possible values.

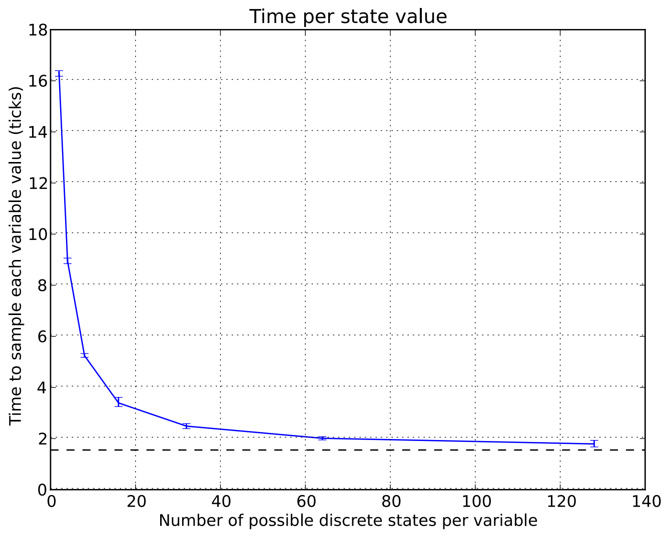


Figure 5-9: Performance overhead of architecture per state variable value. As the number of possible discrete states per variable increases, we approach the expected limit of 1.5 ticks per possible value (black dashed line).

Table 5.1: Resource utilization and performance for a 4-state 20x20 compiled Potts model.

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

5.4 Comparison to explicit compilation

In chapter 4, the compiler generated a fully space-parallel Ising models.. We can generate an equivalent fixed-function lattice factor graph engine [2] for k-state Potts models (Wu 1982) and measure performance, in terms of samples per second and silicon resources used.

To synthesize the engine with the Potts latent pariwise 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 5.1).

We can synthesize a variety of Potts lattice factor graph engines for comparison, all resulting in a 16,384-node MRF (Table 5.2). 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.

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

Table 5.2: 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

Table 5.3: 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

5.5 Stochastic Video Processor

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

5.5.1 Resources and Speed

5.6 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 5-10).

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 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.

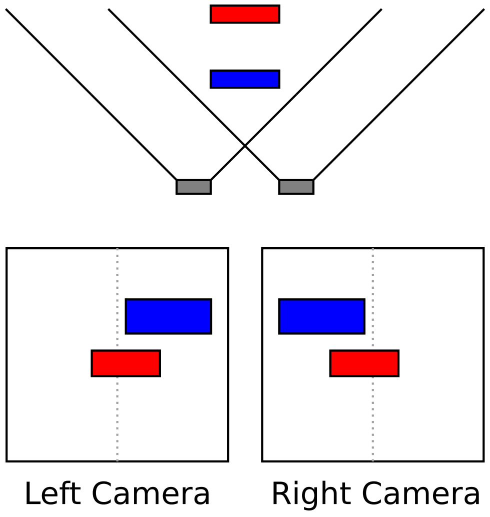


Figure 5-10: Stereo geometry calculations. There is substantial overlap between left and right camera views. Closer objects (blue) appear to shift more between adjacent frames than objects that are further away (red).

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:

$$f_{LP}(x, x') = \begin{cases} 0 & \text{if } x = x' \\ \rho_I(\Delta I) & \text{otherwise} \end{cases} \tag{5.2}$$

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

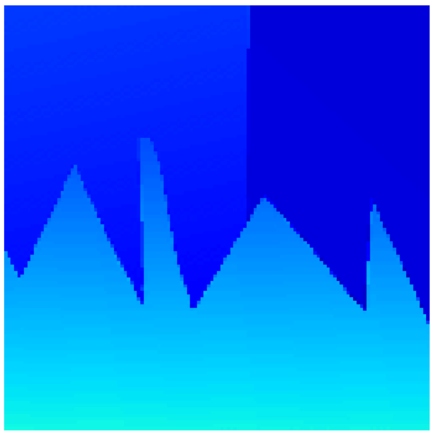
$$\rho_I(\Delta I) = \begin{cases} P \times s & \text{if } \Delta I < T \\ s & \text{otherwise} \end{cases} \tag{5.3}$$

where T is a threshold, s is the penalty for violating the smoothness constraint and P extra-penalizes small smootness 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.

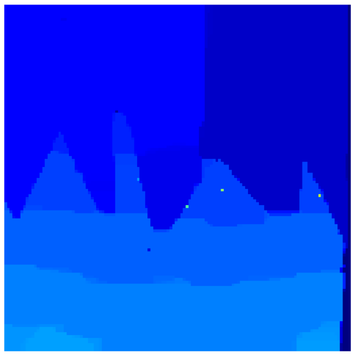
The external field density $F_e(x_{i,j}, y_{i,j})$ computes the Birchfield-Tomasi dissimilarity (Birchfield and Tomasi 1998), a smoothed between-pixel distance measure that is robust against aliasing.

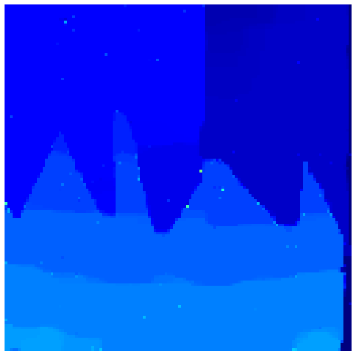
We use three rectified, intensity-calibrated image pairs from the stereo benchmark dataset created by Scharstein and Szeliski (Scharstein, Szeliski, and Zabih 2001). These images have known ground truth depth maps to enable evaluation of our MRF engine at both 8.4 and 6.2 bit precisions. The results (figures 5-11, 5-12, 5-13) are shown for the full 64-bit software engine as well as 8- and 12-bit MRF engines.

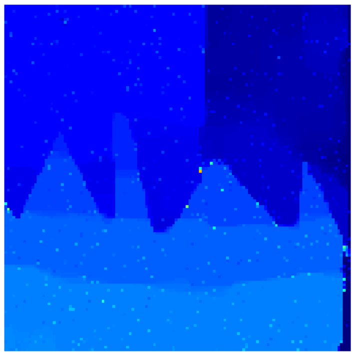
The stochastic video processor finds a total score almost as good as the version computed using Gibbs sampling using IEEE-754 64-bit floating point on a 2.8GHz Intel Xeon, literally three orders of magnitude faster. This is in spite of the clock rate of the video processor being only 125 MHz. For most examples,the quality of the 12-bit solution is nearly as good

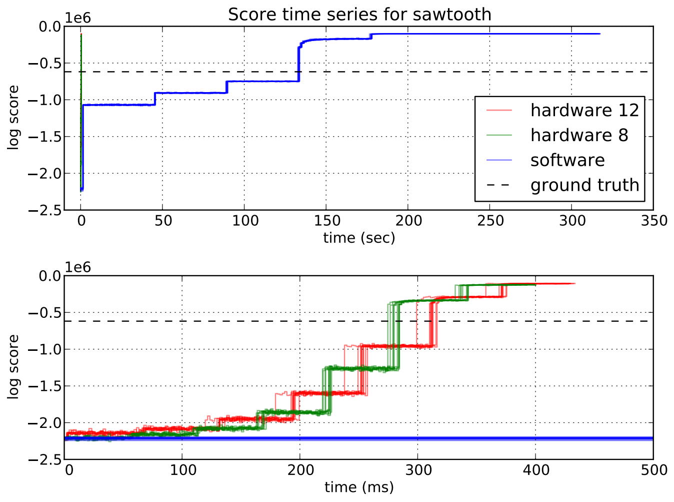

(a) Ground Truth

(b) Left Image


(c) software, 64-bit floating point

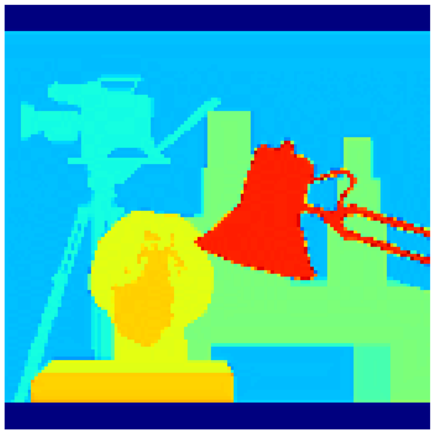

(d) hardware, 12 bits


(e) hardware, 8 bits

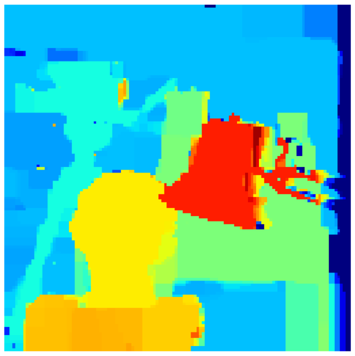


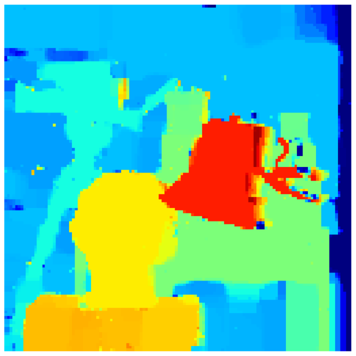
(f) log score vs time, hw vs sw

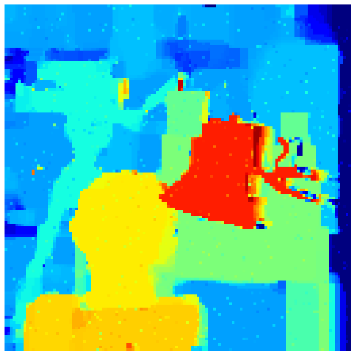
Figure 5-11: Middlebury "sawtooth" example stereo depth dataset. Top row is the ground truth disparity map, and subsequent rows are the empirical mean mean of 10 full annealing sweeps for the double-precision software, and 12-bit and 8-bit hardware, circuits.

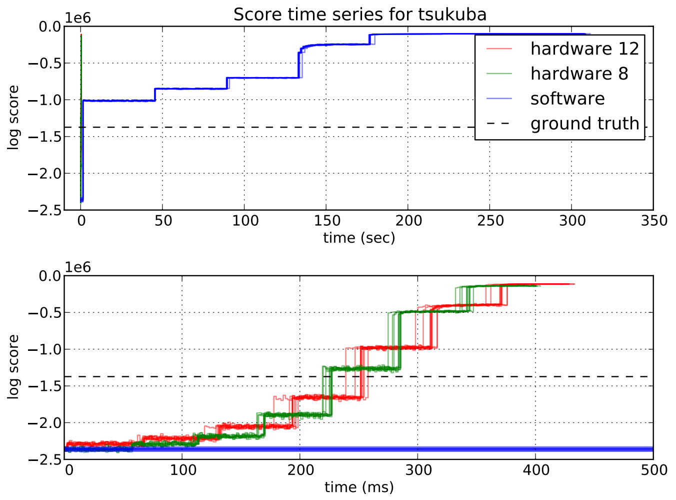

(a) Ground Truth

(b) Left Image


(c) software, 64-bit floating point

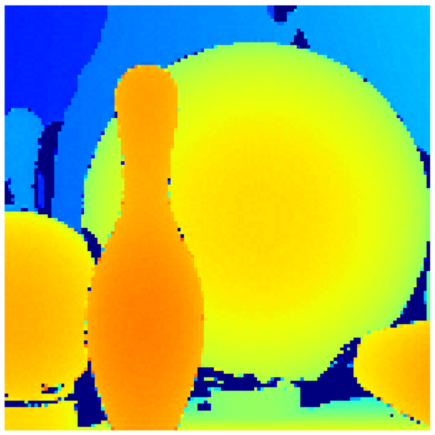

(d) hardware, 12 bits


(e) hardware, 8 bits



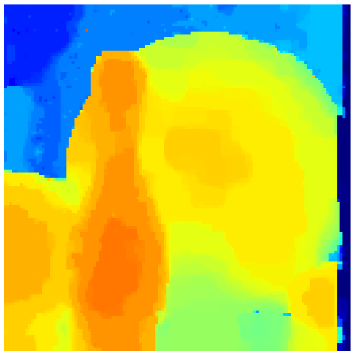
(f) log score vs time, hw vs sw

Figure 5-12: 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.

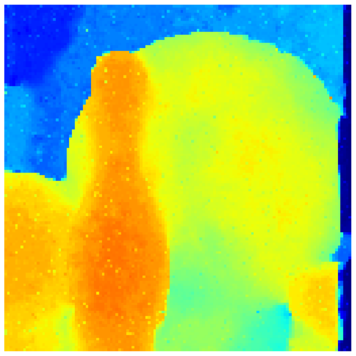


(a) Ground Truth

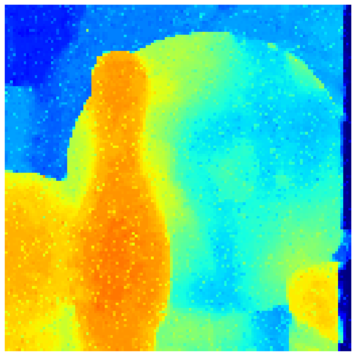
(b) Left Image



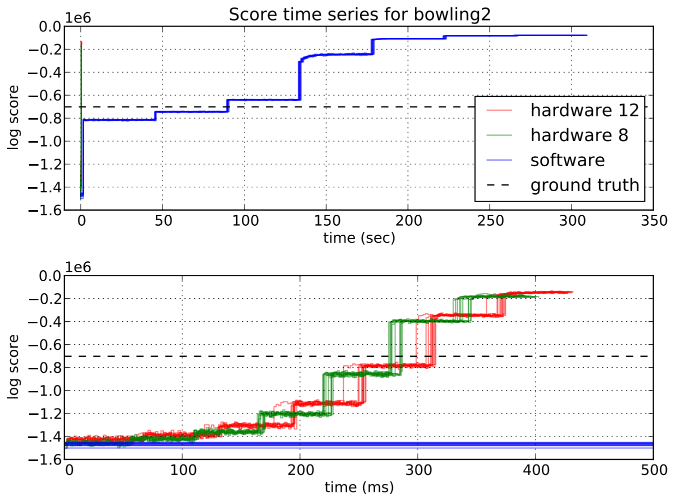
(c) software, 64-bit floating point



(d) hardware, 12 bits



(e) hardware, 8 bits



(f) log score vs time, hw vs sw

Figure 5-13: 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.

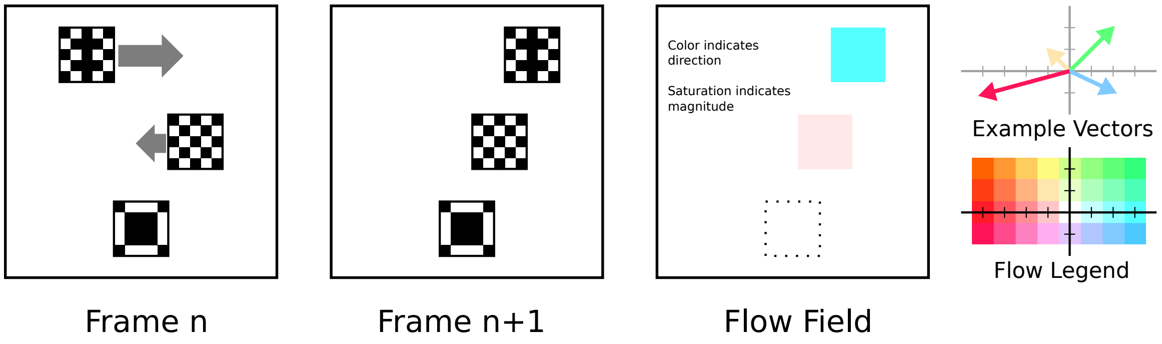


Figure 5-14: Optical flow uses the local pixel differences between frame n and $n+1$ to compute a dense flow field, indicating the direction and magnitude of each pixel's interframe motion. In the above, flow vector direction is indicated by color, and flow magnitude is indicated by saturation. Stationary objects are thus white.

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.

5.7 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 T. 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.

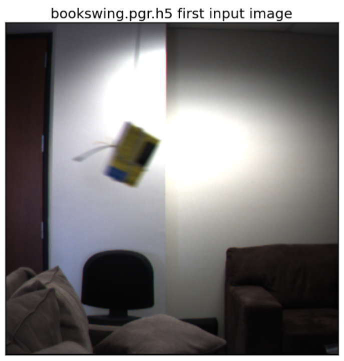
Markov random fields have been used successfully to estimate discontinuous optical flow (Heitz and Bouthemy 1993). A MRF for optical flow can be computed as follows: we discretize the possible flow vectors (in our case, $k \in 0 \ldots 31$) as the latent state variable values. Let flow vector value f_k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

$$f_{LP}(x_{ij}, x_{i'j'}) = -(|i' - i| + |j' - j|) \tag{5.4}$$

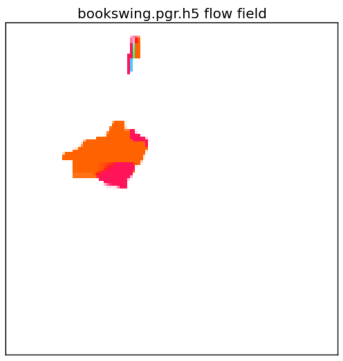
The associated latent pairwise potential is simply the Manhattan distance between the two latent state values 5.4. The range of motion is limited to the 32 nearest flow vectors surrounding the target point.

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 5-15, 5-16, 5-17). Adjacent frames were taken 10 ms apart.

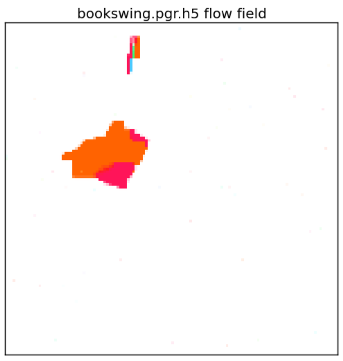
High-quality flow fields were obtained with as few as 300 gibbs scans per frame, giving a maximum frame rate of 32 fps. The dynamic ranges encountered in the optical flow calculations resulted in the 6.2 engine producing very poor quality results; in this case, dense optical flow is a problem that needs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



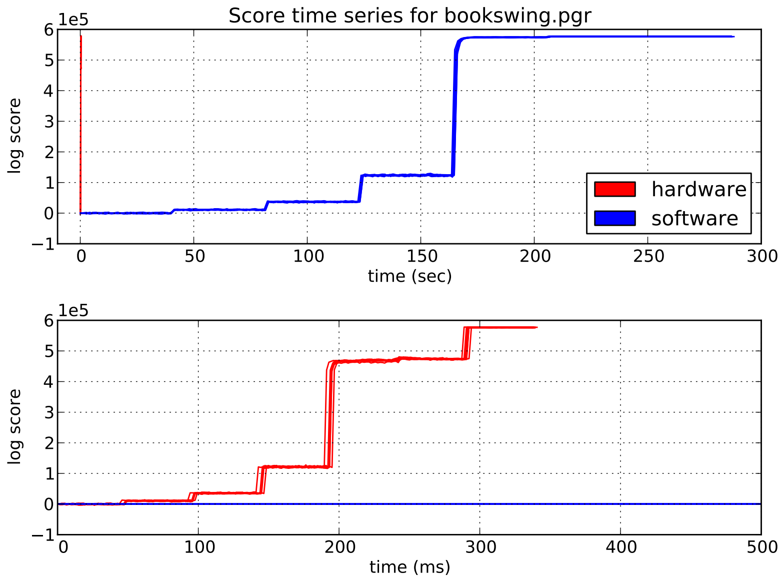
(a) Initial frame



(b) 64-bit software

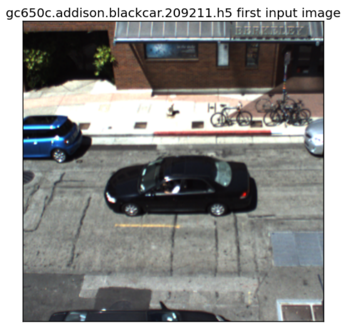


(c) 12-bit hardware

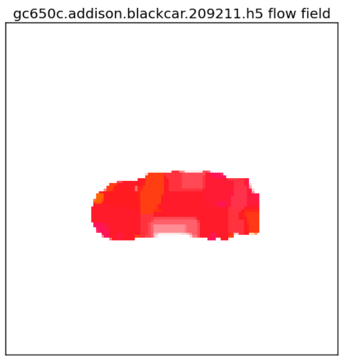


(d) log score vs time

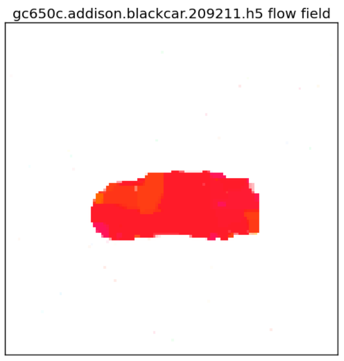
Figure 5-15: Optical flow results on the "bookswing" data



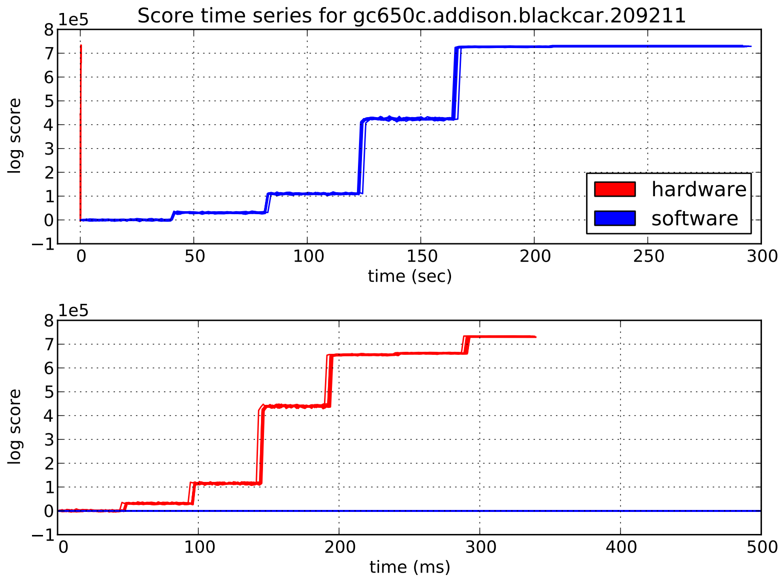
(a) Initial frame



(b) 64-bit software

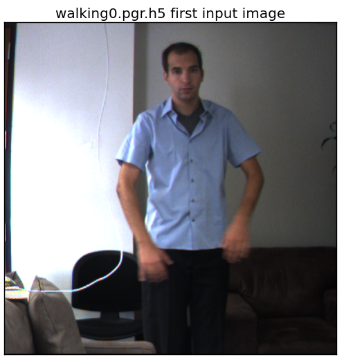


(c) 12-bit hardware

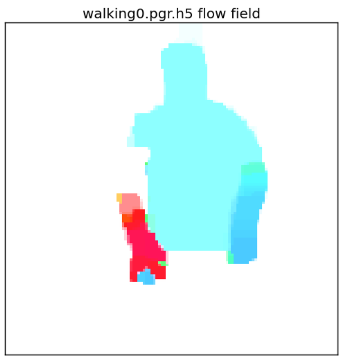


(d) log score vs time

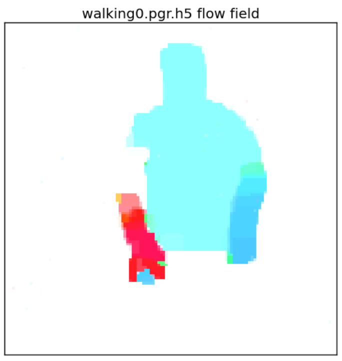
Figure 5-16: Optical flow results on the "blackcar" data



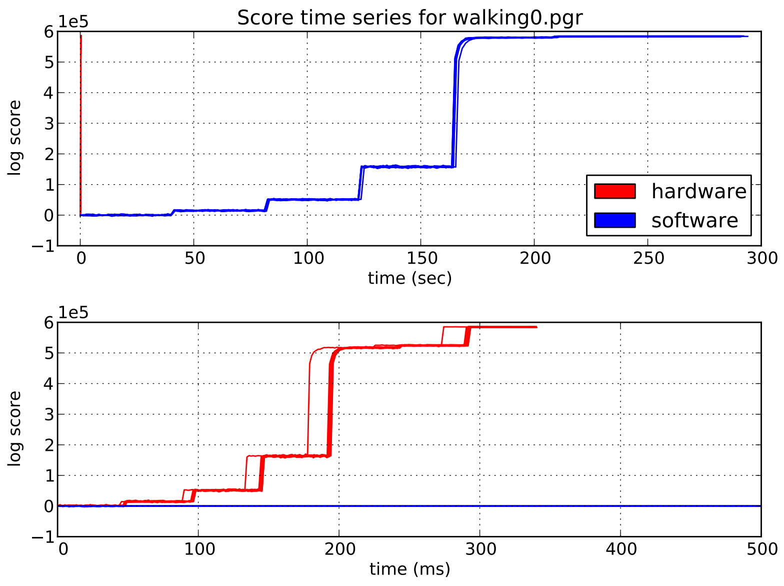
(a) Initial frame



(b) 64-bit software



(c) 12-bit hardware



(d) log score vs time

Figure 5-17: Optical flow results on the "eric" data

5.8 Conclusion

I've 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 5.3 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.

Chapter 6

Clustering, Dirichlet Process Mixture Models, and Dynamic Structure

Both human and other cognitive systems can infer underlying causes in data through identifying hidden "groupings" or clusterings of the data. These systems can identify the true number of clusters in the underlying data, while simultaneously allowing for cluster refinement in the presence of additional data. We'll show how a particular class of probabilistic model captures exactly this intuition, the nonparametric Dirichlet process mixture model.

Up to this point, all of our stochastic architectures have closely followed the statistical nature of the underlying probabilistic model. The Dirichlet Process mixture model exhibits substantially more dynamic structure. By using stochastic queues and storing drastically-reduced quantities of data (the "sufficient statistics"), it's possible to construct a more time-and-area-efficient stochastic circuit. There is a nice intuition – systems should remember summeries of the observed data, not the data in totality.

The stochastic clustering engine enables real-time clustering of large numbers of datapoints with a very large number of features, while closely exploiting the conditional independence of the underlying model. Again, we see that low bit precision and conditional independence provide considerable architectural wins.

The outline of this section closely follows the others; we will describe the computational problem; we will describe the probabilistic model; and then we will focus on the implementation.

6.1 Clustering as a cogntivie primitive

Discovering how humans categorize the world has fascinated both neuroscientists and cognitive scientists since the beginning. People form categories with data without direct supervision, and use this to infer the properties of previously-unseen objects (Joshua B Tenenbaum et al. 2011). Neuroscientists have identified category learning occuring across multiple modalities in multiple higher-order brain areas, including prefrontal and parietal cortex, basal ganglial structures, and even in higher-order sensory corticies (Seger and E. K. Miller 2010).

To explain the structure of human thought as a solution to computatioanl problems present in the environment, cognitive scientists have proposed solutions that take the form of probabilistic mixture models. Anderson (Anderson 1991) introduced a Rational Model of Categorization (RMC) in an attempt to explain how humans learn the number of underlying clusters representing each category.

Radford Neal (Neal 1998) was the first to recognize that RMC was nearly identitical to a Bayesian nonparametric mixture model known as the Dirichlet process mixture model, in which a probability model is specified over a potentially infinite number of clusters. While the DPMM had been celebrated in statistics for quite some time, Griffiths points out that in some sense Anderson "thus independently discovered one of the most celebrated models in nonparametric bayesian statistics, deriving this distribution from first principles" (Griffiths, Sanborn, et al. 2008).

Beyond this, thanks to the composable nature of probabilistic modeling (see 2), the dirichlet process is an effective component in probabilistic models of more complicated cogntivie processes (Joshua B Tenenbaum et al. 2011). As we examine the relationship between the probabilistic model, the clustering beahvior, and the stochastic architectures it suggests, it's good to remember that like an ALU, the clustering architecture we present here can serve as a dedicated component in more complex inference systems.

6.2 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 6-1 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

$$P(\mathbf{x_i}|\{\theta_k\}) = \prod_{d=1}^{D} P(x_i^d|\theta_k^d) \tag{6.1}$$

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 $\mathbf{c}$.

$$\mathbf{X} \sim P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\mathbf{c}|\{\pi_k\}) \tag{6.2}$$

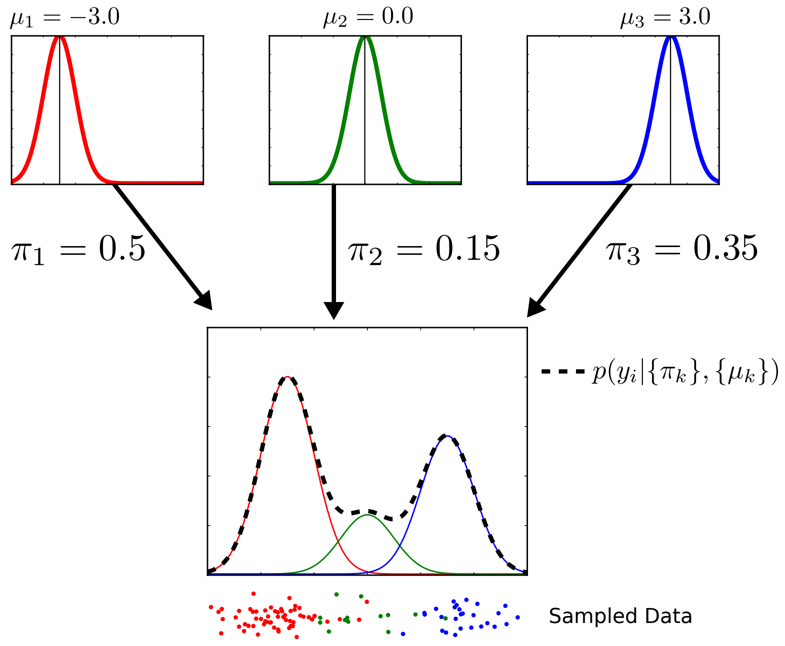


Figure 6-1: A Gaussian mixture model, illustrating the generative process for mixture model data. There are three latent classes from which the data are generated, with the first class generating a datapoint with $\pi_1 = 0.5$. The total probability of a data point y_i is the sum of the source probabilities weighted by their mixture weights. Samples from the resulting distribution are shown below.

6.2.1 Mixing weight prior

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

$$P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\mathbf{c}|\{\pi_k\})P(\{\pi_k\}) \tag{6.3}$$

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 $\frac{\alpha}{K}$:

$$p(\pi_1, \cdots, \pi_k|\alpha) \sim \text{Dirichlet}(\alpha/K, ..., \alpha/K) = \frac{\Gamma(\alpha)}{\Gamma(\alpha/K)^K} \prod_{j=1}^{K} \pi_j^{\alpha/k-1} \tag{6.4}$$

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

$$\begin{aligned} P(\mathbf{c}|\alpha) &= \int_{\Delta_K} \prod_{i=1}^{N} P(c_i|\pi) d\pi & (6.5)\\ &= \int_{\Delta_K} \frac{\prod_{k=1}^{K} \pi_k^{m_k+\alpha_k-1}}{D(\alpha_1, ..., \alpha_K)} d\pi & (6.6)\\ &= \frac{D(m_1 + \frac{\alpha}{K}, m_2 + \frac{\alpha}{K}, \cdots, m_k + \frac{\alpha}{K})}{D(\frac{\alpha}{K}, \frac{\alpha}{K}, \cdots, \frac{\alpha}{K})} & (6.7)\\ &= \frac{\prod_{k=1}^{K} \Gamma(m_k + \frac{\alpha}{K})}{\Gamma(\frac{\alpha}{K})^K} \frac{\Gamma(\alpha)}{\Gamma(N+\alpha)} & (6.8) \end{aligned}$$

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.

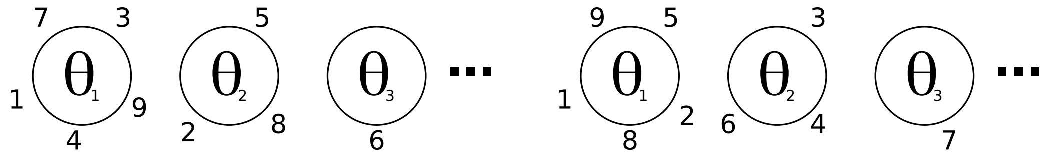
Note that in this equation above, individual class assignments are no longer independent, but what they are is *exchangable* – the probability of a particular assignment vector is the same as any other permutation of the assignment vector.

6.2.2 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 apparantly-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)

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



(a) Assignment from the CRP

(b) Equally-likely assignment from the CRP due to exchangability

Figure 6-2: 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.

$$\begin{aligned} P(\text{ occupied table k}|\text{previous customers}) &\quad \frac{m_k}{\alpha + i - 1} \quad (6.9) \\ P(\text{ next unoccupied table}|\text{previous customers}) &\quad \frac{\alpha}{\alpha + i - 1} \quad (6.10) \\ & \quad (6.11) \end{aligned}$$

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 *exchangable* (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.

6.2.3 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.

$$P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\{\theta_k\})P(\mathbf{c}|\{\pi_k\}) \quad (6.12)$$

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,

$$p(\theta|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \quad (6.13)$$

If we use the Beta distribution as the prior on Bernoulli(θ), conjugacy results in the

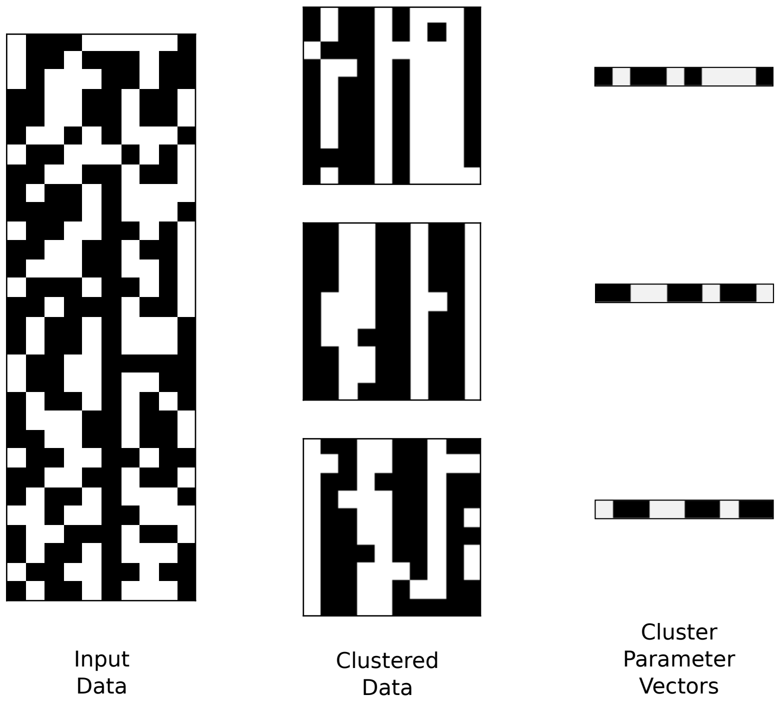


Figure 6-3: An example of clustering via a 10-dimensional binary mixture model. The input data is at left – each row is a data point in the 10-dimensional binary space. The resulting discovered clusters (middle) are associated with latent "parameter vectors", one for each cluster, which are estimated from the data.

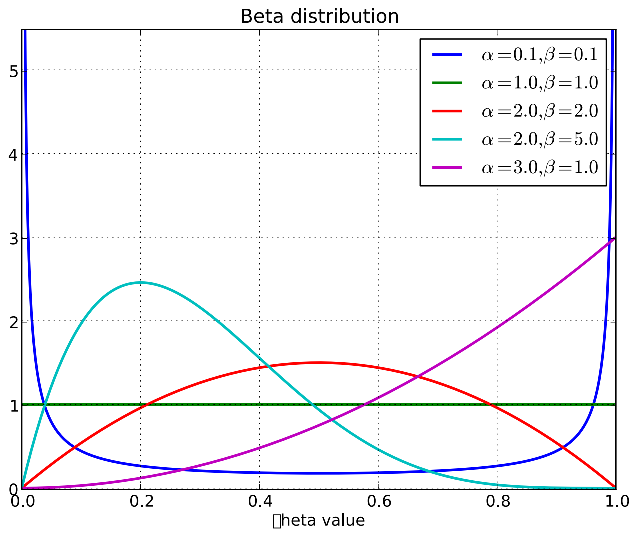


Figure 6-4: The two-parameter Beta distribution, the prior likelihood for the conjugate Beta-Bernoulli data model

posterior after observing $x = 1$ as:

$$p(\theta|x = 1, \alpha, \beta) = \frac{\Gamma(\alpha + \beta + 1)}{\Gamma(\alpha + 1)\Gamma(\beta)} \theta^{1+\alpha-1}(1 - \theta)^{\beta-1} \tag{6.14}$$

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

$$p(\theta|m, n, \alpha, \beta) = \frac{\Gamma(\alpha + \beta + m + n)}{\Gamma(\alpha + m)\Gamma(\beta + n)} \theta^{m+\alpha-1}(1 - \theta)^{n+\beta-1} \tag{6.15}$$

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

6.2.4 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,

$$P(\mathbf{c}|\mathbf{X}) \propto P(\mathbf{X}|\mathbf{c}, \{\theta_k\})P(\{\theta_k\})P(\mathbf{c}|\alpha) \tag{6.16}$$

The combination of conjugacy and exchangability allows us to exactly sample from the resulting conditional distribution on cluster assignments. As we've shown earlier in section 2.4.2, this lets us Gibbs sample (Rasmussen 2000).

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

$$P(c_i = k|\mathbf{c}_{-i}, \mathbf{X}) \propto P(\mathbf{X}|\mathbf{c})P(c_i = k|\mathbf{c}_{-i}) \tag{6.17}$$

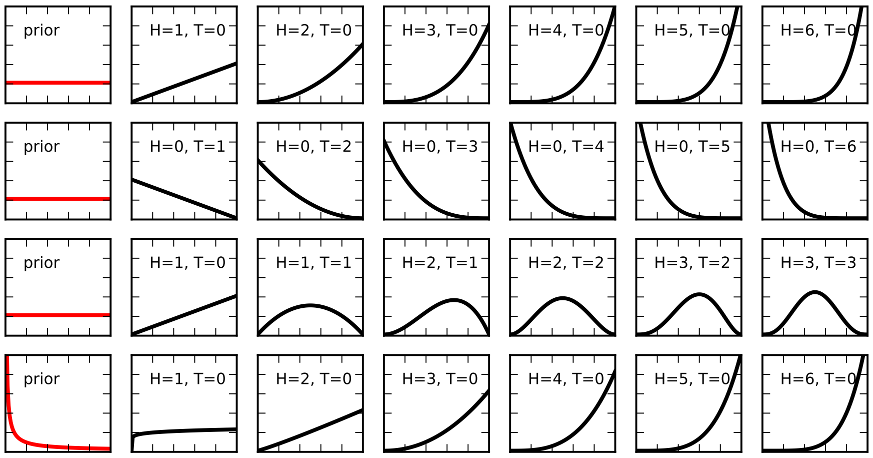


Figure 6-5: Sequential updates to the Beta-Bernoulli conjugate data model. Starting with no observed data and a prior distribution (left), additional observations shift the posterior distribution of θ in an intuitive way. Note on the last row that, even if the prior is wildly biased, sufficient data "overwhelms" the prior's effect on the posterior distribution.

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

$$P(c_i = \text{occupied class}k|\mathbf{c}_{-i}) = \frac{m_{k,-i}}{\alpha + N - 1} \tag{6.18}$$

$$P(c_i = newclass|\mathbf{c}_{-i}) = \frac{\alpha}{\alpha + N - 1} \tag{6.19}$$

$$\tag{6.20}$$

Via conjugacy above, we can easily compute $P(x_i|\mathbf{X}_{-i}, \mathbf{c})$. Thus the gibbs sampling algorithm for the dirichlet-process mixture model with conjugate likelihood is as follows:

6.3 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 underlying number of random variables can change dynamically as inference occurs. This is in contrast to 3, where every random variable had an equivalent stochastic circuit element, or 5, where every random variable was explictly known ahead of time, even though some 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 6-3 – a dense binary matrix where each row is a binary vector of observations. We perform nonparametric clustering to group the rows, identifying for each

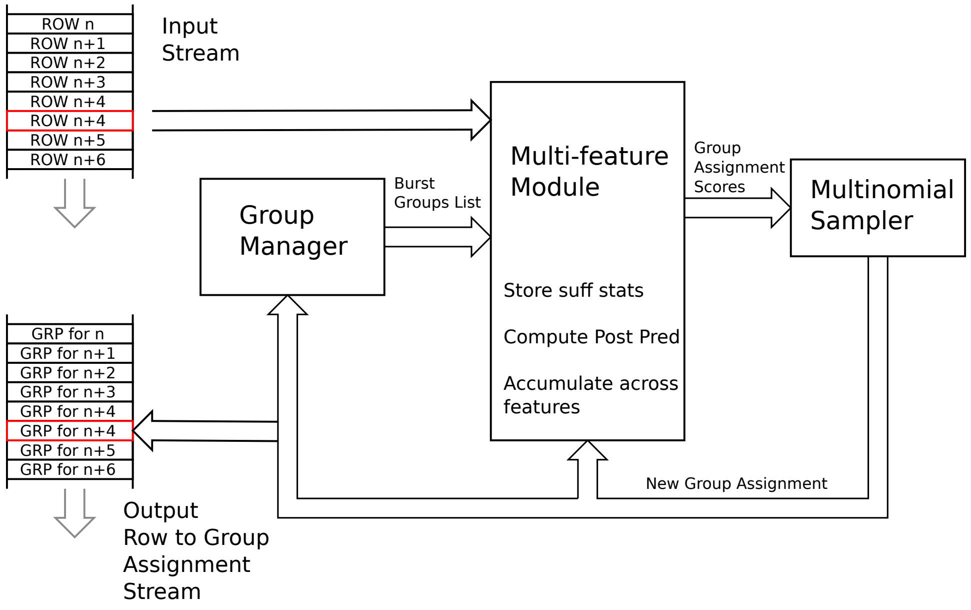


Figure 6-6: Stochastic circuit for inference in a Dirichlet-Process Mixture Model. Input data is streamed through (upper-left), and a distribution on group assignment is computed for each row by the multi-feature module. A sample is drawn from that distribution, the row is inserted in that group, and the new assigned group is streamed out.

grouping a canonical "parameter vector". The circuit learns both the parameter vectors and, importantly, the number of parameter vectors.

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 likelyhood 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 ben sampled, the sufficient statistics for that group are updated, and the group assignment is streamed out of the circuit.

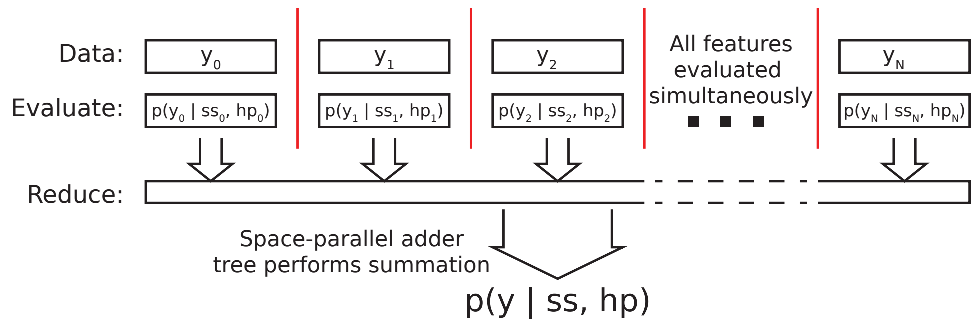


Figure 6-7: Schematic of feature-parallel evaluation of the cluster assignment probability for a given feature; all features are evaluated simultaneously, and the results are summed in parallel via an adder tree.

6.3.1 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.

6.3.2 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 6-9 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 statis (SUFFSTATS and the hyperparameters (HYPERS) and returns $\log P(y * |SS, HP)$. It is also heavily pipelined, with one tick per sample throughput.

6.3.3 Beta Bernoulli Component Model

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

$$p(x = 1|\{x_i\}, \alpha, \beta) = \frac{m + \alpha}{\alpha + \beta + m + n} \tag{6.21}$$

to compute the log score we must compute

$$\log p(x = 1|\{x_i\}, \alpha, \beta) = \log(m + \alpha) - \log(\alpha + \beta + m + n) \tag{6.22}$$

This requires an internal approximation to the log function, which we do via linear interpolation. The resulting approximation error is shown in Figure 6-10, which stays very

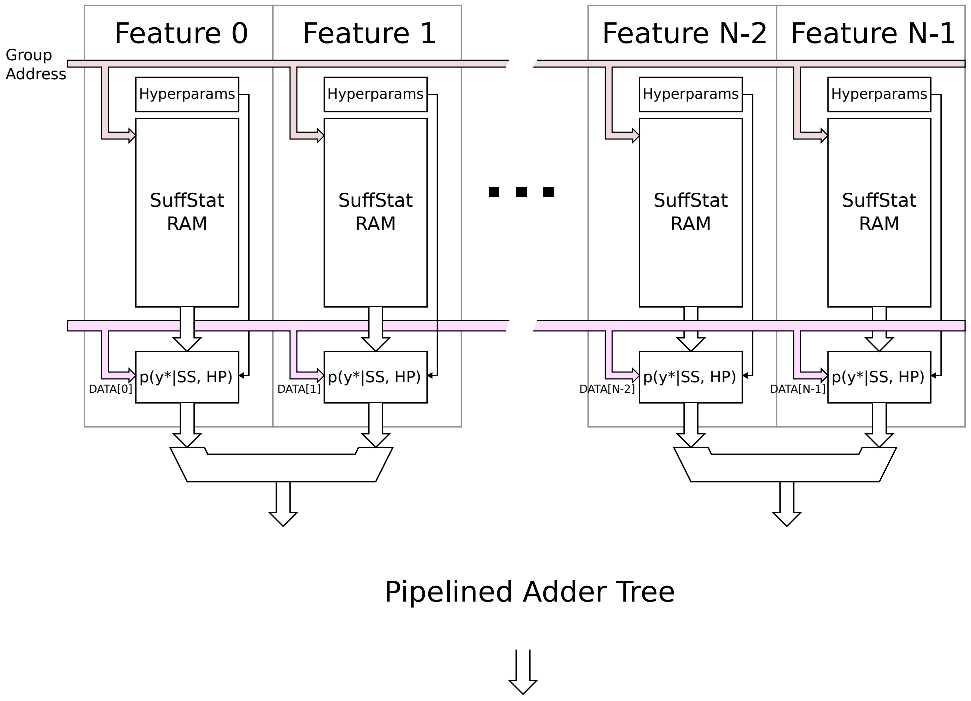


Figure 6-8: Feature Parallel evaluation architecture: A common group is selected across all features, and the posterior predictive is computed for each datum using its associated feature. As the features are independent, and the probabilities are in $\log_2$ space, a simple pipelined adder tree is used to accumulate the total score.

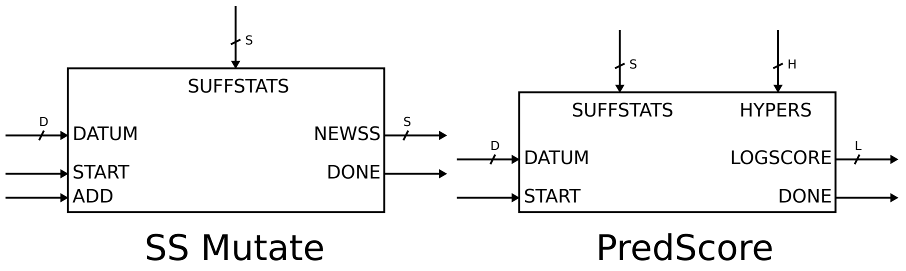


Figure 6-9: Every component model requires the implementation of two modules, a “SSMutate” module to compute updates to sufficient statistics based on the addition or removal of a datum to a group, and a “PredScore” module to compute the probability of a datapoint being generated by a particular group.

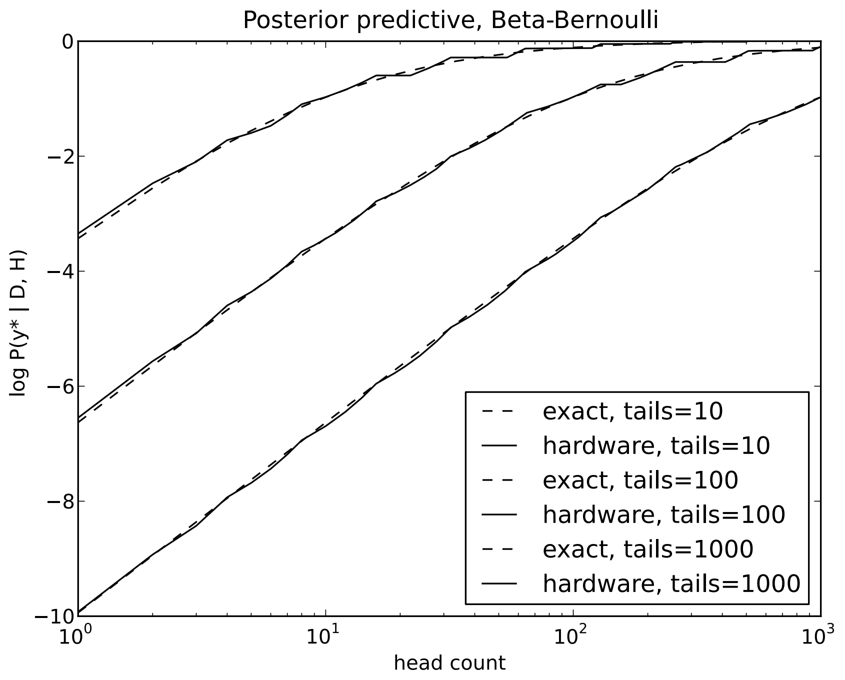


Figure 6-10: Beta Bernoulli posterior predictive hardware approximation results. The dashed line is the exact, floating point result, whereas the solid line is the answer generated by the PostPred module. The results are for $P(x = 1|x_i)$, and are shown as we systematically vary the number of observed heads from 1 to 1000, for three values of observed tails.

small even as we vary the sufficient statistics over a wide range.

6.3.4 Multi-Feature Module

Figure 6-7 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.

6.3.5 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.

Table 6.1: 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

The group manager does this (figure 6-11) 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 it's 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.

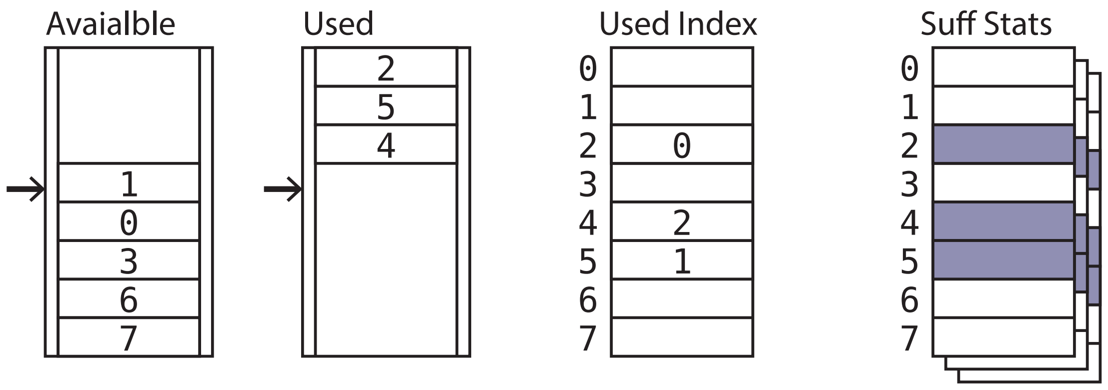
6.3.6 Streaming Inference

The streaming interface (Figure 6-12) 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.

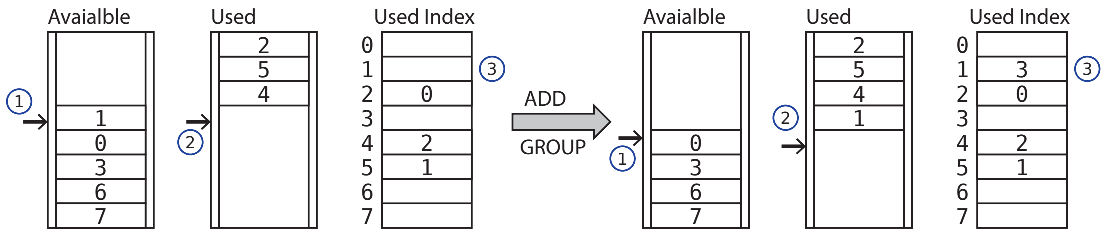
Data is written in a bit-packed format, and is pipelined – the next row of data can be written while the circuit is performing inference on a current row. Hyperparameters and other per-feature configuration information can be written via the feature-control interface.

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.

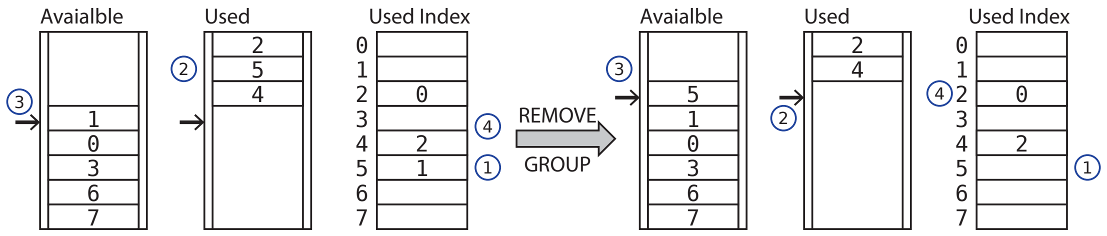
The specific bits of the command word are shown in table 6.1. To add a new datapoint and pick the right group for that datapoint based on the existing data , we set SAMP=1, ADD=1, GROUPSEL=1, DLATCH=1, NEWG=1. Once DONE is asserted, GRPOUT is the group assignment of this new datapoint. To perform generic 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.



(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 6-11: The group manager, which provides dynamic $O(1)$ creation, $O(1)$ deletion, and $O(K)$ bursting of group addresses.

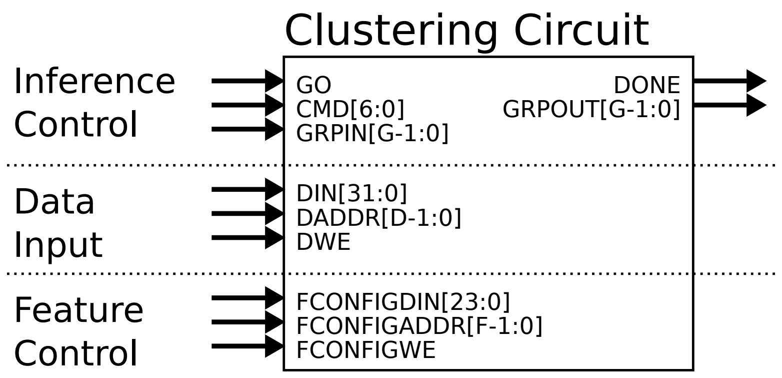


Figure 6-12: Interface for the Dirichlet Process Mixture Model Stochastic Circuit. Data is loaded asychronously via the data interface, and feature hyperparameters are set via feature control. Inference is controlled via a 7-bit command word.

6.4 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 theorietical predictions and an optimized software implementation on commodity hardware.

6.4.1 Resource Utilization

Table 6.2 shows the resource utilization for the hardware designs synthesized to measure KL, below. Table 6.3 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.

6.4.2 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 impracticle in the large data limit. Here we compare with $n = 10$.

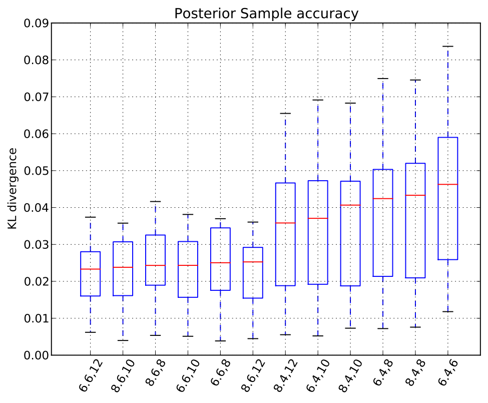
We randomly generate ten 16-feature datasets, and for each firmware bit precision we evaluate the KL between the true posterior distribution and the result of 100000 samples, with a sample taken after every 100 iterations of the core Gibbs steps.

Table 6.2: Resource utilization for 16-feature clustering circuit, maximum 1024 clusters, maximum 2^{16} datapoints per cluster.

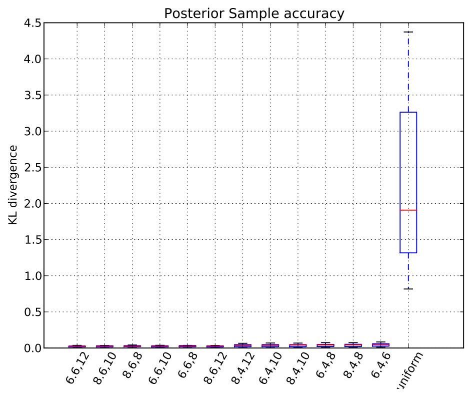
			Resources		
Features	Precision	Q bits	Slice FFs	Slice LUTs	BRAMS
16	4.2	4	17890	13660	56
16	4.4	4	18354	14929	56
16	4.4	6	18368	14940	56
16	4.4	8	18382	14955	56
16	6.2	4	18188	14041	56
16	6.2	6	18190	14048	56
16	6.4	4	18662	15581	56
16	6.4	6	18667	15465	56
16	6.4	8	18681	15480	56
16	6.4	10	18695	15500	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 6.3: Resource utilization for 256-feature clustering circuit, maximum 1024 clusters, 2^{16} datapoints per cluster.

			Resources		
Features	Precision	Q bits	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 6-13: Kullback-Liebler divergence between true, explictly enumerated distribution and collection of posterior samples.

6.4.3 Basic Inference

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 6-14.

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.

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 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 (6-15 shows the result – the circuit closely tracks the true number of groups in the data, although expresses uncertainty for each new datapoint.

6.4.4 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

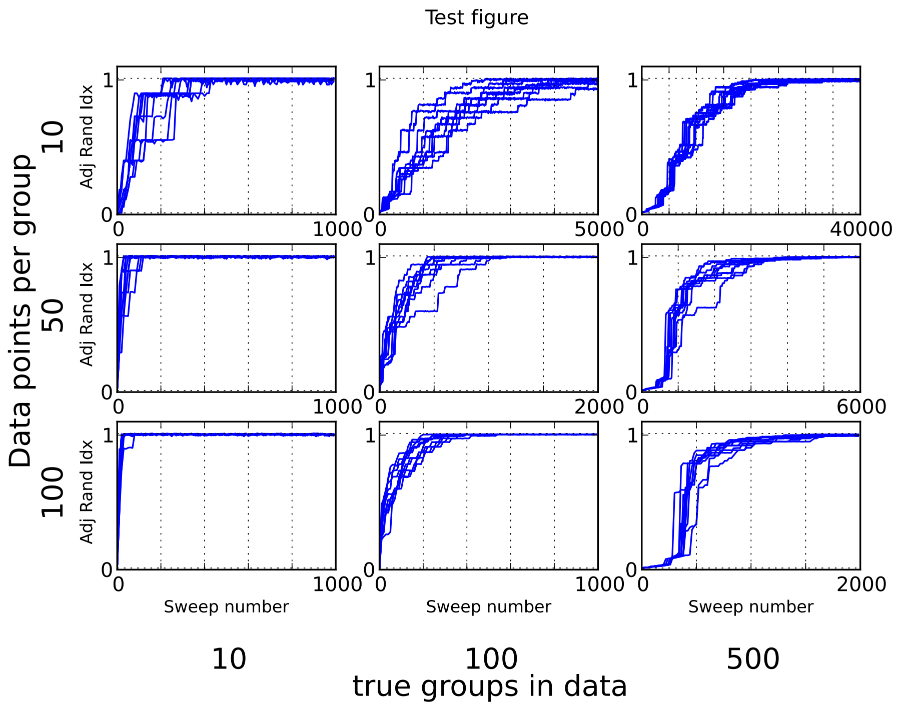


Figure 6-14: Recovery of ground truth for all-in-one-group initialization, across various numbers of true groups in data and rows per group. The adjusted Rand index (see text) measures cluster similiarty – an ARI of 1.0 means recovered clusters are identical to ground truth.

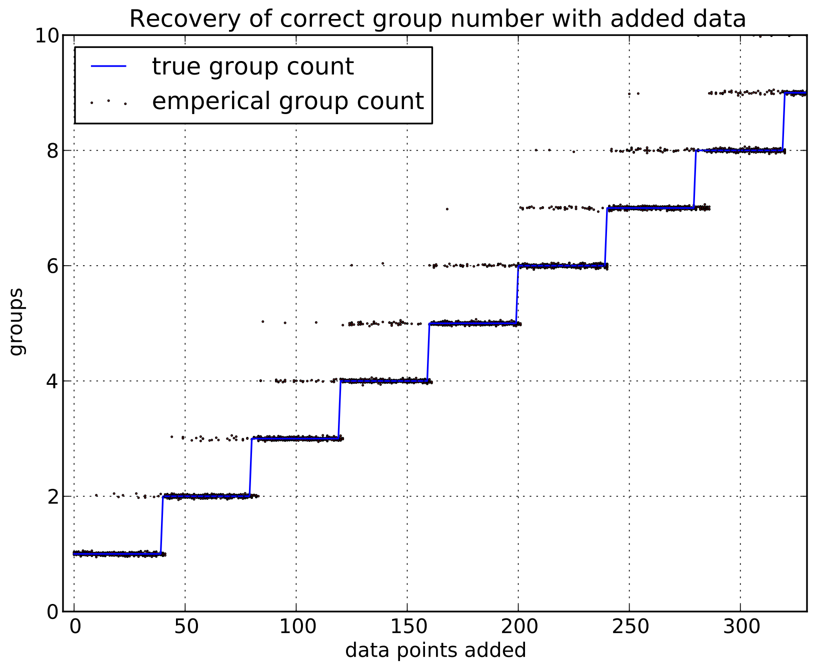


Figure 6-15: Adding new datapoints to the engine. Every set of 40 datapoints belongs to a new group (true group count is in blue). The model correctly estimates the number of latent groups in the data. Jitter has been added to the y-axis to enable density visualization.

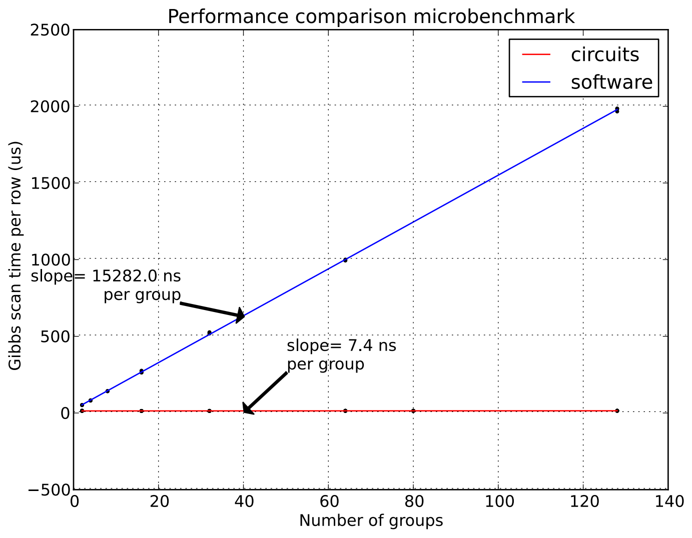


Figure 6-16: Comparison of performance between the stochastic circuit and a hand-optmized mixture model performing the same operations on a desktop computer. Here, we measure the time necessary to compute the group assignment distribution for a single row. The slope of each line is the time necessary to evaluate the probability of the row being assigned to the particular group.

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 can compare this with Gibbs sampling performance with a custom beta-Bernoulli DP mixture model implemented by hand in 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 6-16. 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 μs 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 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 6-17). The two slopes are in close agreement, however we can see that the PCI-E overhead adds 4 μs per row.

6.5 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 20 by 20-pixel binarized images of hand-written digits. They are often used as

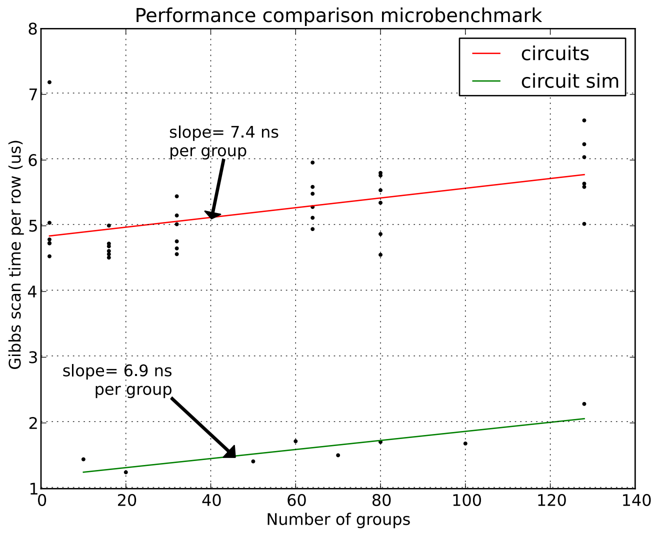


Figure 6-17: The circuit's performance in actual hardware, including PCI-E bus and host control overhead, is very close to the theoretical performance from VHDL simulation of the circuit without complex inferface hardware.

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 20x20 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.

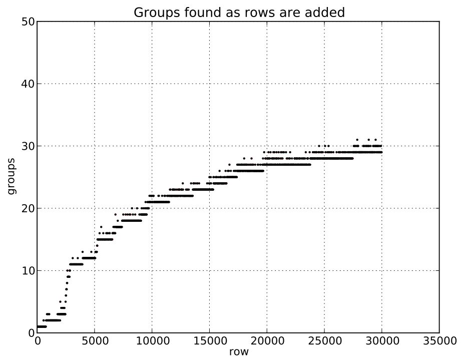
Clustering

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 6-18). All hyperparameters were set to 1.0.

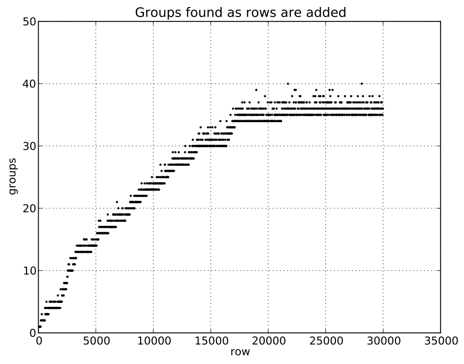
We organize the clusters by their "most common true class" in figure 6-19. The different "styles" of each digit are readily apparent, as is the perceptual ambiguity of certain styles of digits (8 and 3, for example).

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 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

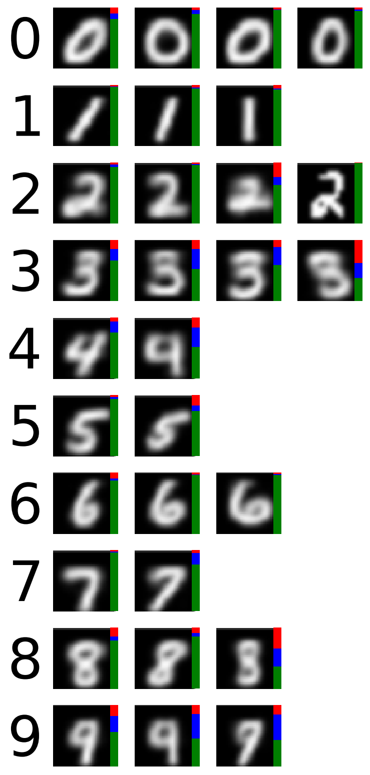


(a) 8.6,12-bit precision

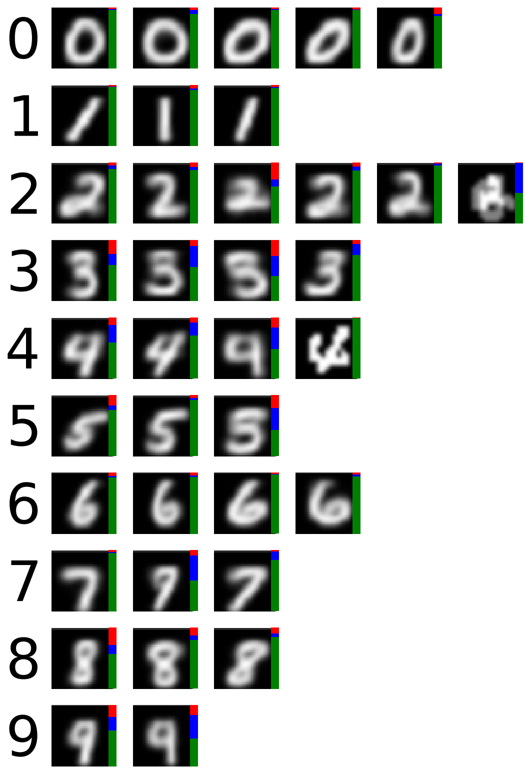


(b) 6,4,10-bit precision

Figure 6-18: 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.

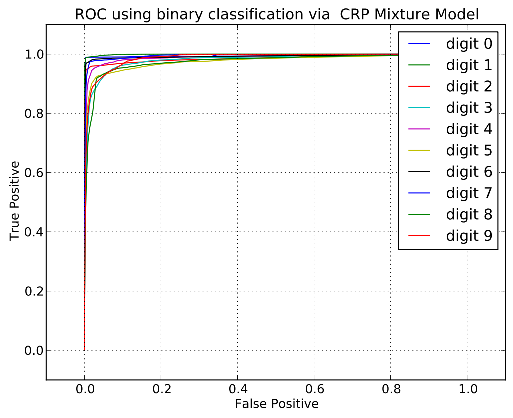


(a) 8.6,12-bit precision

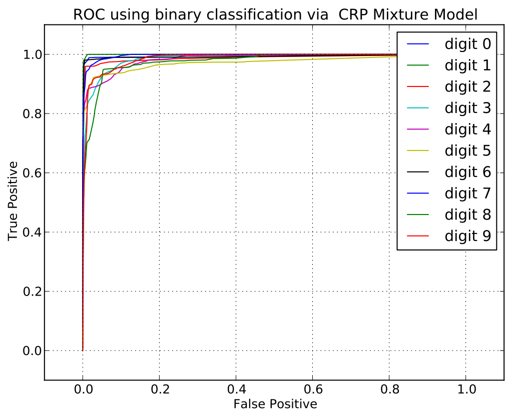


(b) 6.4,10-bit precision

Figure 6-19: 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.



(a) Low bit precision (6.4, 6)



(b) High bit precision (8.6, 12)

Figure 6-20: ROC curves for posterior predictive-based classification of test digits from the MNIST dataset for two different circit bit precision. Classification becomes more perfect as the line gets closer to the upper-left axes.

each of 1000 test rows, taking 100 samples per row.

The ROC curves show good performance in this prediction task (figure 6-20, 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 6.4: Area under the curve for the inclass-outclass ROCs.

6.6 Future Directions

I 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.

6.6.1 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.

As I've shown repeatedly, conditional independence gives rise to opportunities for parallelism. Here, the features are conditionally independent, and thus we can score them in parallel with minimal overhead. But 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.

6.6.2 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 – a dirichlet process "coprocessor", if you will, for models dependent on the hierarhcial dirichlet process.

And finally, for some applications, the inductive bias of the dirichlet process is not appropriate (J. W. Miller and Harrison 2012), in particular where we have very strong priors on the true (but still unknown) number of clusters. Recent work (J. W. Miller and Harrison 2012) on "Mixtures of Finite Mixture" suggests a more reasonable modeling approach in this case, although the exact architectural connections are a subject for further research.

Chapter 7

Conclusion and Future Directions

"Thus the nervous system appears to be using a radically different system of notation from the ones we are familiar with in ordinary arithmetics and mathematics: instead of the precise systems of markers where the position – and presence or absence – of every marker counts decisively in determining the meaning of the message, we have here a system of notations in which the meaning is conveyed by the statistical properties of the message. We have seen how this leads to a lower level of arithmetical precision but to a higher level of logical reliability: a deterioration in algorithmics has been traded for an improvement in logics." – von Neumann, 1956

We have covered a lot of ground while exploring the viability of stochastic architectures for probabilistic computation.

First, we outlined the composition and abstraction laws that arise when working with the probability calculus, and how migration from a universe of deterministic evaluation to one of stochastic sampling preserves the ability to engineer while giving considerable performance improvements.

Then we showed that this stochastic architectural approach was both well-defined an advantageous enough that we could build an automatic compiler – transforming graphical models into natively stochastic circuits to solve problems of inductive reasoning.

We then solved various problems of visual perception, in the process demonstrating how we could virtaulize portions of stochastic circuits, enabling the re-use of stochastic elements and scaling to thousands of random variables.Finally, we showed how dynamic architectures for stochastic inference could solve problems of clustering and categoriztaion.

7.1 Related Work

Having presented a wide variety of stochastic architectures, it is now useful to revisit some of the previous work done on building more brain-like computing systems and shifts in computer architecture.

7.1.1 The digital signal processor

The rise of the digital signal processor provides a good background from which to compare our own work and other work in neuromorphic-style computation. The rise of digital signal processing began in the middle part of the last century as it became desirable to implement more and more of a signal-processing chain digitally. At the time, existing embedded hardware was ill-suited for the streaming data volumes and multiply-accumulate operations necessary to perform tasks like filtering. Chips slowly arrived capable of performing particular tasks, such as TI's TMS5100 for linear-predictive coding and speech synthesis (made famous by the company's "Speak and Spell" product). TI's TMS32010 introduced many of the standard features of DSPs going forward, including a Harvard architecture and custom instructions including fast mutliply-accumulate (accelerating convolution).

I would argue that generic DSPs thrived because

- there was already extensive theoretical work on how to solve real world problems with digital signal processing.
- fabricating custom ASICs, even for high-volume consumer products, was just too expensive
- dedicated signal processing hardware enabled vendors to innovate at the algorithm, not circuit, level.

7.1.2 Existing Neuromorphic work

Early attempts in the 1980s to build hardware that worked more like the brain focused on developing ASICS with computing units similar to neurons. Carver Mead and others attempted this by operating various CMOS amplifier elements in the ultra-low power, subthreshold regime (Lyon and Mead 1988; Hutchinson, Koch, and Mead 1988). In one example circuit, they even worked with optical flow, casting it as an energy-minimization problem with smoothness constraints on top of low-power analog VLSI (Hutchinson, Koch, and Mead 1988).

While these circuits worked, they didn't point towards a larger set of design idioms for building more complex circuits. Ultimately, it was not clear how they would abstract or compose. This, coupled with their nascent programmability and the incredible costs of ASICs at the time, resulted in them staying a research project.

Work at the beginning of the last decade (Hahnloser et al. 2000) started to try and combine digital and analog approaches in silico to capture both the multistability and smoothly-varying properties of cortical networks. Around this time greater experimentation with spiking neural networks and spike-based computation began (Fusi et al. 2000), but with a focus more on capturing the physiological properties of neural systems, rather then creating a viable engineering substrate.

That said, there continues to be considerable research interest in neuromorphic-style computation on low-power analog VLSI substrates. Groups have been experimenting with low-power subthreshold probabilsitic CMOS (Chakrapani, George, and B. Marr 2008) which operates in the sub-threshold regime and can probabilistically give the "correct" answers to deterministic queries. They have subsequently built up low-power PCMOS circuits for tasks like h.264 decoding, with 4-10x power savings over traditional designs. It's important to note that, while the substrates themselves are more stochastic than engineers are used to, the goal of these efforts have been still to approximately execute existing, deterministic algorithms.

7.1.3 Analog computation

A more promising approach was outlined by Vigoda in 2003 (Vigoda 2003). They showed how continuous-time low-power analog circuits could perform various message-passing algorithms in binary factor graphs. There are many problems that can be solved as binary factor graphs, including the toy Ising model we presented in this thesis, as well as many schemes for encoding and decoding low-density parity check codes. To our knowledge, this line of work has resulted in the only commercially-viable line of probability processing, ultimately resulting in Vigoda's company Lyric Semiconductor being acquired by Analog Devices.

While the work presented in this thesis extends, both conceptually and in the examples given, beyond the binary factor-graph case currently covered by continuous time analog probabilistic logic, it remains to be seen if there will be similar commercial impact.

7.2 Future directions

There are still a lot of unanswered research questions at the interface of stochasticity and computation.

7.2.1 Other Substrates

For starters, we've focused entirely on stochastic digital logic – using the existing advances in CMOS to prove the viability of stochastic architectures. The development of analog, molecular, biological, and quantum substrates for computation have all been plagued by the existence of stochastic behavior. Existing work has largely focused on adding redundancy to these systems to reduce the probability of error in these very noisy systems. Is it instead possible to exploit the stochastic nature of these systems?

7.2.2 Other Monte Carlo Schemes

We've also spent the majority of our time focusing on a handful of MCMC techniques. There are many additional MCMC methods, all with different trade-offs for parallelization and bit precision.

Sequential Monte Carlo methods, such as particle filtering (Gordon, Salmond, and Smith 1993; Chen 2003) are also parallelizable and stochastic. Particle filters can be used to effectively solve nonlinear state-space models where we model some latent state x_t at time t in a Markovian fashion:

$$x_t \sim p(x_t|x_{t-1}) \tag{7.1}$$

$$y_t \sim p(y_t|x_t) \tag{7.2}$$

An ensemble of samples for $P(x_t)$, called "particles", are kept around and updated and reweighted sequentially with each new timestep. Particle filtering lacks the asymptotic guarantees of MCMC, but has the advantage of being constant-time and constant-space as it merely solves the *filtering* problem – estimating $p(x_t|x_{1:t-1}, Y_{1:t})$. One can think of particle filtering as the mating of a particular inference strategy and query (that is, a desire to solve the filtering problem) with the structure of a particular probabilistic model (sequential state-space models).

Based on our circuit designs, it should be possible to perform the resampling step of a particle filter using our multinomial Gibbs units, and sample new particles with custom-designed sampling circuits in a densely parallel fashion. The tradeoffs between serial and parallel evaluation need to be explored.

Approximate Bayesian Computation (ABC) is another Monte Carlo technique for performing Bayesian inference in models where computing the likelihood $p(y|x)$ is intractable – only forward simulation is viable (Bartlett 1963; Marin et al. 2011). Our stochastic sampling approach is a very natural fit for ABC methods, as we can build circuits to run massive, nested forward simulations in parallel. This area is still entirely unexplored, and remains a subject for future research.

7.2.3 The Neural connection

We've also sidestepped the issue of the exact relationship between probabilistic inference, MCMC, and neural systems. Recent results, including some by us, have shown that one can build sampling systems out of biophysically-realistic neurons (Mansinghka and Jonas, in submission). The implementation details and properties of sampling neural systems can be an entire PhD in itself.

Many researchers are increasingly thinking this way. Mounting evidence supports the "Bayesian Brain" hypothesis, as we've referenced throughout this document. Computational neuroscientists are attempting (Pouget et al. 2013) to build physiologically-realistic models of how neurons might perform this probabilistic inference. Sampling approaches do make an appearance, hinting at a possible connection between our stochastic sampling architectures and neural systems. Suggestions include spikes representing samples from binary distributions and membrane voltages representing samples from real-valued distributions (Fiser et al. 2010; Lee and Mumford 2003; Hoyer and Hyvarinen 2003).

7.3 Conclusion

There's ample ground for future work – in both advancing our circuit abstraction, building more scalable systems for inference, and discovering new substrates on which to build probabilistic computing systems. We're still quite far off from the energy efficiency and computational power of the brain, but we hope that the work contained herein represents an important first step.

Bibliography

[1] John R. Anderson. “The adaptive nature of human categorization.” In: *Psychological Review* 98.3 (1991), pp. 409–429. DOI: `10.1037//0033-295X.98.3.409`.

[2] MS Bartlett. “The Spectral Analysis of Point Processes.” In: *Journal of the Royal Statistical Society. Series B (. . .* 25.2 (1963), pp. 264–296.

[3] IA Beinlich et al. “The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks.” In: 38 (1989), pp. 247–256.

[4] Stan Birchfield and Carlo Tomasi. “A Pixel Dissimilarity Measure That Is Insensitive to Image Sampling.” In: *Pattern Analysis and Machine . . .* 20.4 (1998), pp. 401–406.

[5] C.M. Bishop. *Pattern Recognition and Machine Learning*. Vol. 4. 2. Springer New York, 2006.

[6] Aaron P Blaisdell et al. “Causal reasoning in rats.” In: *Science (New York, N.Y.)* 311.5763 (Mar. 2006), pp. 1020–2. DOI: `10.1126/science.1121872`.

[7] Keith A Bonawitz. “Composable probabilistic inference with BLAISE.” PhD. Massachusetts Institute of Technology, 2008, p. 190.

[8] LNB Chakrapani, Jason George, and Bo Marr. “Probabilistic Design : A Survey of Probabilistic CMOS Technology and Future Directions for Terascale IC Design.” In: *VLSI-SoC: Research . . .* 249 (2008), pp. 101–118.

[9] Zhe Chen. “Bayesian Filtering : From Kalman Filters to Particle Filters , and Beyond.” In: *Statistics* (2003).

[10] Persi Diaconis. “The Markov chain Monte Carlo revolution.” In: *Bulletin of the American Mathematical Society* 46.2 (Nov. 2008), pp. 179–205. DOI: `10.1090/S0273-0979-08-01238-X`.

[11] Thomas S. Ferguson. “A Bayesian Analysis of Some Nonparametric Problems.” In: *The Annals of Statistics* 1.2 (Mar. 1973), pp. 209–230. DOI: `10.1214/aos/1176342360`.

[12] David Ferrucci et al. “Building Watson: An overview of the DeepQA project.” In: *AI magazine* (2010), pp. 59–79.

[13] Jzsef Fiser et al. “Statistically optimal perception and learning: from behavior to neural representations.” In: *Trends in cognitive sciences* 14.3 (Mar. 2010), pp. 119–30. DOI: `10.1016/j.tics.2010.01.003`.

[14] Stefano Fusi et al. “Spike-Driven Synaptic Plasticity: Theory, Simulation, VLSI Implementation.” In: *Neural Computation* 12.10 (Oct. 2000), pp. 2227–2258. DOI: `10.1162/089976600300014917`.

[15] S Geman and D Geman. "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images." In: *IEEE transactions on pattern analysis and machine intelligence* 6.6 (June 1984), pp. 721–41.

[16] Noah D Goodman et al. "Church: a language for generative models." In: *Uncertainty in Artificial Intelligence* 22.April (2008), p. 23.

[17] Alison Gopnik, Clark Glymour, et al. "A theory of causal learning in children: causal maps and Bayes nets." In: *Psychological review* 111.1 (Jan. 2004), pp. 3–32. DOI: `10.1037/0033-295X.111.1.3`.

[18] Alison Gopnik and Joshua B. Tenenbaum. "Bayesian networks, Bayesian learning and cognitive development." In: *Developmental Science* 10.3 (May 2007), pp. 281–287. DOI: `10.1111/j.1467-7687.2007.00584.x`.

[19] NJ Gordon, DJ Salmond, and AFM Smith. "Novel approach to nonlinear/non-Gaussian Bayesian state estimation." In: *IEE Proceedings F: Radar and Signal Processing* 140.2 (1993), pp. 107–113.

[20] Thomas L Griffiths and Zoubin Ghahramani. "The Indian Buffet Process : An Introduction and Review." In: *Journal of Machine Learning Research* 12 (2011), pp. 1185–1224.

[21] Thomas L Griffiths, Adam N Sanborn, et al. "Categorization as nonparametric Bayesian density estimation." In: *The probabilistic mind Prospects for Bayesian cognitive science* (2008). Ed. by Nick Chater and Mike Oaksford, pp. 303–328.

[22] R H Hahnloser et al. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit." In: *Nature* 405.6789 (June 2000), pp. 947–51. DOI: `10.1038/35016072`.

[23] F. Heitz and P. Bouthemy. "Multimodal estimation of discontinuous optical flow using Markov random fields." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15.12 (1993), pp. 1217–1232. DOI: `10.1109/34.250841`.

[24] PO Hoyer and A Hyvarinen. "Interpreting Neural Response Variability as Monte Carlo Sampling of the Posterior." In: *neural information processing systems* 1 (2003).

[25] J. Hutchinson, C. Koch, and C. Mead. "Computing motion using analog and binary resistive networks." In: *Computer* 21.3 (Mar. 1988), pp. 52–63. DOI: `10.1109/2.31`.

[26] Ernst Ising. "Beitrag zur Theorie des Ferromagnetismus." In: *Zeitschrift fr Physik* 31.1 (Feb. 1925), pp. 253–258. DOI: `10.1007/BF02980577`.

[27] Edwin T Jaynes. *Probability Theory: The Logic of Science.* Ed. by G. Larry Bretthorst. Cambridge University Press, 2003.

[28] Eric R Kandel, James H Schwartz, and Thomas M Jessell. *Principles of Neural Science.* Ed. by Eric R Kandel, James H Schwartz, and Thomas M Jessell. Vol. 4. 22. McGraw-Hill, 2000. Chap. The Bodily, p. 1414. DOI: `10.1036/0838577016`.

[29] Y Lecun et al. *Gradient-based learning applied to document recognition.* Ed. by S Haykin and B Kosko. 1998. DOI: `10.1109/5.726791`.

[30] Tai Sing Lee and David Mumford. "Hierarchical Bayesian inference in the visual cortex." In: *Journal of the Optical Society of America. A, Optics, image science, and vision* 20.7 (July 2003), pp. 1434–48.

[31] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. "High-throughput bayesian computing machine with reconfigurable hardware." In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '10*. New York, New York, USA: ACM Press, 2010, p. 73. DOI: `10.1145/1723112.1723127`.

[32] Jun Liu. *Monte Carlo Strategies in Scientific Computing*. Corrected. Springer, 2008.

[33] David Lundgren. *FPU Double VHDL*. 2009.

[34] R.F. Lyon and C. Mead. "An analog electronic cochlea." In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.7 (July 1988), pp. 1119–1134. DOI: `10.1109/29.1639`.

[35] Jean-michel Marin et al. "Approximate Bayesian Computational methods." In: (Jan. 2011), pp. 1–28. arXiv: `1101.0955`.

[36] D Marr and T Poggio. "Cooperative computation of stereo disparity." In: *Science (New York, N.Y.)* 194.4262 (Oct. 1976), pp. 283–7.

[37] David Marr. *VISION*. Henry Holt and Company, 1982, p. 397.

[38] George Marsaglia. "Xorshift RNGs." In: *Journal Of Statistical Software* 8.14 (2003), pp. 1–6.

[39] George Marsaglia and Wai Wan Tsang. "Some difficult-to-pass tests of randomness." In: *Journal Of Statistical Software* 7.3 (2002), pp. 1–9.

[40] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." In: *Acm Transactions On Modeling And Computer Simulation* 8.1 (1998), pp. 3–30. DOI: `10.1145/272991.272995`.

[41] Andrew Mccallum, Karl Schultz, and Sameer Singh. "FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs." In: *Advances in Neural Information Processing Systems* 22 (2009). Ed. by Y Bengio et al., pp. 1–9.

[42] Nicholas Metropolis et al. "Equation of State Calculations by Fast Computing Machines." In: *The Journal of Chemical Physics* 21.6 (1953), p. 1087. DOI: `10.1063/1.1699114`. arXiv: `5744249209`.

[43] Brian Milch et al. "BLOG: Probabilistic Models with Unknown Objects." In: *International Joint Conference on Artificial Intelligence* 19 (2005). Ed. by F Giunchiglia and L Pack Kaelbling, p. 1352.

[44] Jeffrey W Miller and Matthew T Harrison. "Posterior consistency for the number of components in a finite mixture." In: *NIPS Workshop on Modern Nonparametric Machine Learning*. x. Lake Tahoe, NV, 2012, pp. 1–5.

[45] M-series Module. *Pico M-501 M-Series Module*. Tech. rep. Pico Computing, 2013, pp. –1.

[46] Edward F Moore. "Gedanken Experiments on Sequential Machines." In: *Automata Studies* (1956), pp. 129–153.

[47] Gordon E Moore. "Cramming More Components onto Integrated Circuits." In: *Electronics* (1965), pp. 114–119.

[48] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. Cambridge: The MIT Press, 2012.

[49] Kevin P. Murphy. "The Bayes Net Toolbox for MATLAB." In: *Computing Science and Statistics* 33 (2001). DOI: 10.1.1.25.1216.

[50] Radford M. Neal. *Markov Chain Sampling Methods for Dirichlet Process Mixture Models*. Tech. rep. University of Toronto, 1998.

[51] John von Neumann. "Probabilistic logics and the synthesis of reliable organisms from unreliable components." In: *Automata Studies* 34 (1956), pp. 43–99.

[52] John von Neumann. "Various Techniques Used in Connection with Random Digits." In: *Journal of Research of the National Bureau of Standards, Appl. Math. Series* 3 (1951), pp. 36–38.

[53] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988, p. 552.

[54] Alexandre Pouget et al. "Probabilistic brains: knowns and unknowns." In: *Nature Neuroscience* (Aug. 2013), pp. 1–9. DOI: 10.1038/nn.3495.

[55] William M. Rand. "Objective Criteria for the Evaluation of Clustering Methods." In: *Journal of the American Statistical Association* 66.336 (Dec. 1971), pp. 846–850. DOI: 10.1080/01621459.1971.10482356.

[56] CE Rasmussen. "The infinite Gaussian mixture model." In: *Advances in neural information processing systems* (2000).

[57] Stuart Russell and Peter Norvig. *Artificial Intelligence: a Modern Approach*. 3rd. Prentice Hall, 2009.

[58] D. Scharstein, R. Szeliski, and R. Zabih. "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms." In: *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*. 1. IEEE Comput. Soc, 2001, pp. 131–140. DOI: 10.1109/SMBV.2001.988771.

[59] Mark J. Schervish. *Theory of Statistics (Springer Series in Statistics)*. Springer, 1996, p. 702.

[60] Carol a Seger and Earl K Miller. "Category learning in the brain." In: *Annual review of neuroscience* 33 (Jan. 2010), pp. 203–19. DOI: 10.1146/annurev.neuro.051508.135546.

[61] Claude E Shannon. "A symbolic analysis of relay and switching circuits." M.S. Massachusetts Institute of Technology, 1940.

[62] K.L. Shepard and V. Narayanan. "Noise in deep submicron digital design." In: *Proceedings of International Conference on Computer Aided Design*. IEEE Comput. Soc. Press, 1996, pp. 524–531. DOI: 10.1109/ICCAD.1996.569906.

[63] M F Tappen and W T Freeman. *Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters*. 2003. DOI: 10.1109/ICCV.2003.1238444.

[64] Greg Taylor and George Cox. "Behind Intel's New Random-Number Generator - IEEE Spectrum." In: *IEEE Spectrum* (2011).

[65] Joshua B Tenenbaum et al. "How to grow a mind: statistics, structure, and abstraction." In: *Science (New York, N.Y.)* 331.6022 (Mar. 2011), pp. 1279–85. DOI: 10.1126/science.1192788.

[66] *Tesla Kepler GPU Accelerators*. Tech. rep. NVIDIA, 2102.

[67] a. Verri and T. Poggio. “Motion field and optical flow: qualitative properties.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11.5 (May 1989), pp. 490–498. DOI: `10.1109/34.24781`.

[68] Benjamin Vigoda. “Analog Logic : Continuous-Time Analog Circuits for Statistical Signal Processing.” PhD. Massachusetts Institute of Technology, 2003, p. 199.

[69] *Virtex-6 Configruable Logic Block User Guide*. Tech. rep. Xilinx, Inc., 2012, pp. 1–50.

[70] *Virtex-6 Family Overview*. 2012.

[71] Eric W. Weisstein. *Bell Number – from Wolfram MathWorld*. en.

[72] F. Y. Wu. “The Potts model.” In: *Reviews of Modern Physics* 54.1 (Jan. 1982), pp. 235–268. DOI: `10.1103/RevModPhys.54.235`.