An Experimental Evaluation of Probabilistic Deep Networks for Real-time Traffic Scene Representation Using Graphical Processing Units

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Mennat Allah Ahmed Mohammed El-Shaer, B.Sc.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2019

Dissertation Committee:

Fusun Ozguner, Advisor

Keith Redmill, Co-Advisor

Xiaorui Wang

Abstract

The problem of scene understanding and environment perception has been an important one in robotics research, however existing solutions applied in current Advanced Driving Assistance systems (ADAS) are not robust enough to ensure the safety of traffic participants. ADAS development begins with sensor data collection and algorithms that can interpret that data to guide the intelligent vehicle's control decisions. Much work has been done to extract information from camera based image sensors, however most solutions require hand-designed features that usually break down under different lighting and weather conditions.

Urban traffic scenes, in particular, present a challenge to vision perception systems due to the dynamic interactions among participants whether they are pedestrians, bicyclists, or other vehicles. Object detection deep learning models have proved successful in classifying or identifying objects on the road, but do not allow for the probabilistic reasoning and learning that traffic situations require. Deep Generative Models that learn the data distribution of training sets are capable of generating samples from the trained model that better represent sensory data, which leads to better feature representations and eventually better perception systems. Learning such models is computationally intensive so we decide to utilize Graphics Processing chips designed for vision processing. In this thesis, we present a small image dataset collected from different types of busy intersections on a university campus along with our CUDA implementations of training a Restricted Boltzmann Machine on NVIDIA GTX1080 GPU, and its generative sampling inference on an NVIDIA Tegra X1 SoC module.

We demonstrate the sampling capability of a simple unsupervised network trained on a subset of the dataset, along with profiling results from experiments done on the Jetson TX1 platform. We also include a quantitative study of different GPU optimization techniques performed on the Jetson TX1.

Acknowledgments

This work was partially funded by NSF grant number CNS-1446735.

Vita

July, 2009	Bachelor of Science in Computer Engineering - Ain Shams University, Cairo, Egypt

Publications

Research Publications

Menna El-Shaer, Keith Redmill and Fusun Ozguner "A Quantitative Analysis of Optimizations on the Tegra X1 for Probabilistic Inference" *Under Review* 2018

Menna El-Shaer "A Collection of Five Types of Traffic Intersection Images Collected Using Point Grey Cameras and Jetson TX1" *Dataset* 2018

Menna El-Shaer "Real-time Inference of Generative Models on TX1 at Traffic Intersections" *NVIDIA's GPU Technology Conference (GTC) in San Jose, CA* 2018

Dongfang Yang, John M. Maroli, Linhui Li, Menna El-Shaer, Bander A. Jabr, Keith Redmill, Fusun Ozguner and Umit Ozguner "Crowd Motion Detection and Prediction for Transportation Efficiency in Shared Spaces" *Proceedings of the Third International Workshop on Science of Smart City Operations and Platforms Engineering (SCOPE), Cyber-Physical Security Systems Week in Porto, Portugal* 2018

Menna El-Shaer "Real-time Mapping at Traffic Intersections" *NVIDIA's GPU Technology Conference (GTC) in San Jose, CA* 2017

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

Page

Abstract ii

Acknowledgments iv

Vita v

List of Tables ix

List of Figures x

1. Introduction 1

1.1 Scene Perception in Automated Driving and ADAS applications 1
1.1.1 Related Work 2
1.2 Parallel Computing 4
1.2.1 Types of Parallelism 5
1.3 Real-time Sensor Data Processing 6
1.3.1 Why we need Parallel (Multicore) Computing for Perception Systems 6
1.4 Organization of this Thesis 7

2. Scene Representation Learning 9

2.1 Generative Models 9
2.2 Energy-based Generative Models 10
2.2.1 Restricted Boltzmann Machines 11
2.2.2 Deep-Belief Networks 13
2.2.3 Deep Boltzmann Machines 17
2.3 Evaluating Generative Models 19

3. Scene Reconstruction 23

3.1 Classic Inference Algorithms 23
3.1.1 Markov Chain Monte Carlo (MCMC) 23
3.1.2 Variational Inference 24
3.2 Differentiable Inference Algorithms 24
3.2.1 Variational Autoencoders (VAEs) 24
3.2.2 Generative Adversarial Networks (GANs) 25
3.2.3 Invertible Density Estimation Models 26
3.3 Conclusion 27

4. Scene Interpretation and Object Classification 28

4.1 Naive Bayes Classification 28
4.2 Probabilistic Markov Networks 29

5. Embedded Vision Processing on SoCs 32

5.1 Hardware Architectures for Embedded Vision Applications 34
5.1.1 Hardware Accelerators Examples 34
5.2 High-Level Software Frameworks 36
5.3 Conclusion 37

6. Heterogeneous Processing GPU Models 39

6.1 The CUDA Programming Model 40
6.1.1 Single Instruction Multiple Thread (SIMT) Execution Model . . . 41
6.1.2 Memory Organization (excluding registers) 41
6.2 Heterogeneous Memory Systems 41
6.2.1 Unified Memory 41
6.2.2 Pinned Memory and Data Transfers 45
6.2.3 Unified Memory Optimizations on the TX1 47
6.3 Performance Analysis and Profiling 48
6.3.1 Parallel Execution 49
6.3.2 Global Memory Bandwidth 49
6.3.3 Instruction Throughput 52
6.3.4 Occupancy 53

7. Experiments 55

7.1 Image Representation and Reconstruction 55
7.1.1 Dataset 1: Bike Path Images 55

7.1.2 Dataset 2: Urban Traffic Intersections 57
7.1.3 Model Training 59
7.1.4 Model Inference 64
7.1.5 Profiling on the TX1 70
7.2 Optimization Study 73
7.2.1 Implementation 73
7.2.2 Results and Discussion 75

8. Conclusion 81

Bibliography 83

Appendices 91

A. Data Collection and Processing Setup 91

A.1 The Development Environment 91
A.1.1 NVIDIA TX1 Quick Overview 91
A.2 NVIDIA Tegra X1 Device Specifications 93

List of Tables

Table **Page**

6.1 Summary of different memory types on the GPU 43

6.2 Kernel Execution Times 47

7.1 Hyperparameters Constants in training our experimental networks 65

7.2 Summary of different GPU memory types used in this study 74

7.3 Test Cases 74

7.4 Total Execution Times (ms) 77

7.5 Visible to Hidden Execution Times (ms) 77

7.6 Hidden to Visible Execution Times (ms) 77

7.7 Gibbs Sampling Execution Times (ms) 78

7.8 Global Memory Throughput 78

7.9 Global Memory Load Efficiency 79

7.10 Global Memory Store Efficiency 79

7.11 Optimization Benefit for each case in terms of time 80

List of Figures

Figure **Page**

1.1 Variation trends of processor clock speeds and sizes. Images courtesy of Nature 530, 144147 (11 February 2016), (a) Intel (b) SIA/SRC 5

2.1 Types of Generative Models 10

2.2 A graphical representation of a Restricted Boltzmann Machine 12

2.3 Three single-layered RBMs 14

2.4 Three-layer Deep Belief Network 15

2.5 Three-layer Deep Boltzmann Machine with no within-layer connections . . . 18

2.6 Algorithm steps used to compute PIQUE perception scores of images 22

4.1 Representing an image as a Markov network 30

5.1 A very high-level view of a typical embedded processing system used in machine vision applications 33

6.1 The Heterogeneous Computing Model 40

6.2 The Parallel Thread Execution Model 42

6.3 Profiling example showing 99.63% of processing time spent in device synchronization on the Tegra X1 to ensure memory coherence 46

6.4 CUDA API flags used to change behavior of managed memory that can be used for data prefetching 48

6.5 Global Memory Access Patterns. (a-c) Aligned memory access: the first address of access is a multiple of 32. Bus utilization is 100% in (a,b) and 3.125% in (c) for 4-byte words (d,e) Misaligned memory access: access is spread across two 128-byte transactions. Bus utilization is 50% (f) Scattered memory access: Warp requests six 4-byte words scattered across global memory; over-fetching is avoided here since L2-cached accesses are always 32 bytes. 51

7.1 Experiment Overview 56

7.2 Experimental Setup used to collect real-world images along a gravel bike path using an electric golf cart 57

7.3 Sample images recorded from an .svo video file using ZED SDK 1.0 58

7.4 Sample images recorded from the collected dataset of traffic intersections . . 60

7.5 Initial weight matrix displayed as 360 x 960 image. This weight matrix is for the first hidden neuron in the RBM layer of 200 hidden neurons 61

7.6 Computed weight matrices computed after training using the bike path dataset 64

7.7 Computed weight matrices computed after training using the urban intersections dataset 65

7.8 Generated Samples from the trained model in section 7.1.3 67

7.9 Image quality (PIQUE) scores (%) computed for five different image samples generated using different number of hidden nodes for both datsets respectively 68

7.10 Images used for quality evaluation from figure 7.9. The images in the first column are from the Bike Path Dataset generated using 5, 20, 100, 200 and 400 hidden nodes respectively. The images in the second column are from the Intersection Dataset generated using 5, 10, 20, 40 and 45 hidden nodes respectively. 69

7.11 Implementation of Gibbs sampling kernels 70

7.12 A plot of vis_to_hid kernel duration times ran with block sizes of 32, 64, 128 and 1024 threads 71

7.13 How the number of warps vary per Maxwell SM with different block sizes. There are 2 SMs on the Maxwell GPU architecture 71

7.14 Function Unit utilization of the vis_to_hid kernel 72

7.15 Types of instructions and their percentage of execution in the vis_to_hid kernel 72

A.1 NVIDIA's Maxwell GPU Architecture 92

A.2 TX1 Device Specifications 93

Chapter 1: Introduction

The problem of scene understanding and environment perception has been an important one in robotics research, however existing solutions applied in current Advanced Driving Assistance systems (ADAS) are not robust enough to ensure the safety of traffic participants. ADAS development begins with sensor data collection and algorithms that can be interpreted to guide the intelligent vehicle's control decisions. Much work has been done to extract information from camera based sensors, however most solutions require hand-designed features that usually break down under different lighting and weather conditions.

1.1 Scene Perception in Automated Driving and ADAS applications

Scene perception remains one of the hardest problems facing intelligent vehicle navigation and self-driving cars. If cars are meant to replace human drivers, then we at least need to incorporate human cognition and perception into the problem. Human cognition is very complex and involves many processing layers and is not the subject of our study. However, neuroscientists have been modeling the human brain for years and we should be able to use this knowledge to our advantage. Part of the problem lies in understanding how this sensory data is represented. My goal in this work is to study and develop hierarchical internal representations of the data similar to how cognition in the human brain is represented.

1.1.1 Related Work

Most of the work that has been done to understand scenes in images has been through parsing the input image to detect objects or to extract information using pre-learned feature representations. Thus, the success of classification algorithms depends heavily on how the choice of features matches the sensory input data [1]. In the context of traffic scenes; where extracting lane marks on the road, traffic signs, vehicles, and pedestrians is important; traditional feature descriptors as SIFT [2], SURF [3], BRIEF [4], HOG [5], [6] and LBP [7], [6] were previously used. Haar [8] and Gabor [9] wavelets are another popular feature-based approach. Random Forest classification was explored in [10] using multimodal data from RGB cameras and LiDARs to detect pedestrians. Appearance-based cues, such as color and texture information were used in [11] and [12] respectively. Contextual information, such as 3D road geometry and vanishing points, were used as priors in a Bayesian framework, in addition to low-level cues in [13]. A probabilistic model for urban scene intersections using vehicle tracklets, vanishing points, semantic labels, scene flow, and occupancy grid as observations or evidence was developed in [14].

A popular image representation in the intelligent vehicle literature has been Stixels; where an image is segmented into thin, vertical, stick like rectangles of superpixels. Seeking a compact scene representation for real-time automative applications: in [15], a stixel model for street scenes was defined by solving an energy minimization problem where a column stixel segment is described by the number of stixels in a segment; the bottom-to-top row vertical extent of the stixel; its semantic class label; its color and depth attributes. In [16], instead of parsing the whole scene to output a driving decision, a mapping was done to use a set of 13 affordance indicators trained using a deep convolutional network.

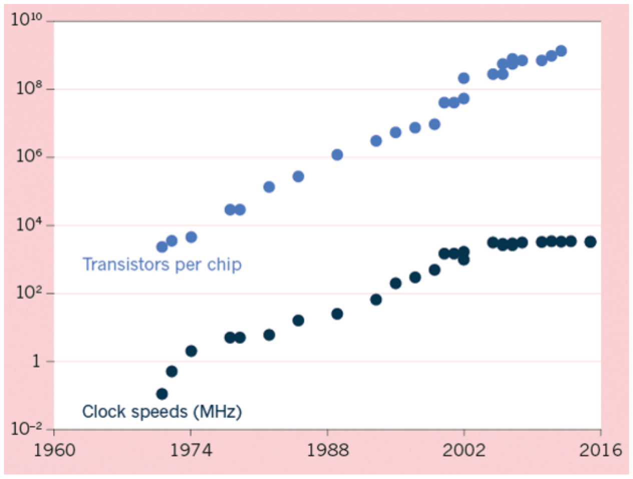
Deep Convolutional Neural Networks (CNNs) have been used in almost image segmentation and object classification tasks (e.g. *semantic labeling*) for the past few years. The recent availability of GPUs and open-source deep learning libraries such as Caffe [17], Torch [18], Tensor Flow [19], [20], PyTorch [21], [22] and MXNet [23] has spread their use. However, choosing network parameters for best detection performance on a specific dataset is more of an art than a science as their mechanics are not well understood. CNNs are formed of multiple hierarchical *Convolutional layers* that learn filters which are activated on detecting some specific feature at some spatial position in the input. Convolutional layers only require partial connections between the neurons within a local receptive field where the learned weights are shared among those neurons. A set of hyperparameters control these connections by, for example, determining the depth of the layer and the number of filters (feature maps) learned. As a result, these connections model the correlation pattern within the input; *Pooling layers* reduce the amount of data by applying a non-linear function after the convolutional layer to obtain a higher-resolution representation and reduce dimensionality.

Maps: In the context of automated driving, maps have been pivotal. Incorporating satellite imagery with street views helped create rich maps with an accuracy of several meters. In the DARPA Urban Challenge in 2007, detailed map information enabled autonomous driving for several miles. For behavior and trajectory planning, the vehicle has to localize itself within the environment created by these 2D maps. Localization based on visual sensors can help create more detailed 3D maps than using GNSS-IMU localization alone in traffic environments [24].

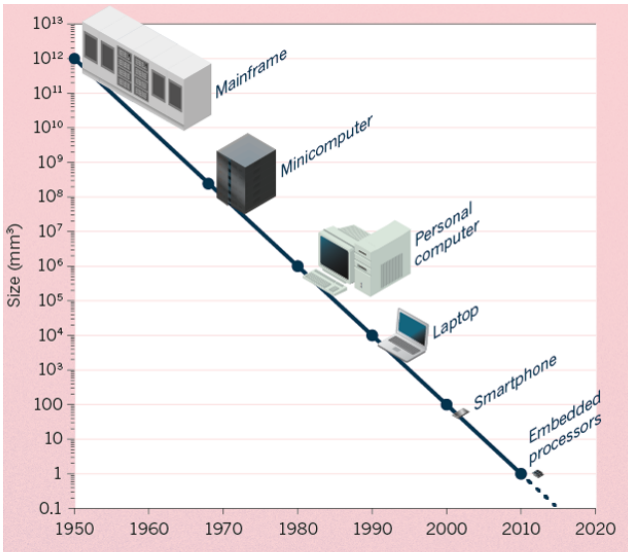
Generative Models: The recent success of deep learning in the autonomous vehicle industry is largely attributed to how well those systems can classify objects in the traffic environment. Usually the most studied network architectures e.g. Convolutional Neural Networks (CNNs) are discriminative learning models where a vast amount of human-annotated, i.e. labeled, data is needed for training and learning features in the data. The system then learns from the labeled examples and can detect or classify those objects on its own. While this is important in solving a subset of situational awareness problems of self-driving cars, identifying the class of an object in the environment, e.g. classifying the object that runs in front of the car as a child or a squirrel, is irrelevant if the car is supposed to react in the same way. Thus, training these deep networks in the supervised manner used is not very useful in challenging urban traffic environments that need reasoning built into the learning process. A class of deep networks that are better suited for reasoning tasks are generative learning models that don't require labeled examples and generally learn the joint distribution of the training data. Thus, by training these networks with different traffic situations, they learn a traffic model with specific parameters related to the traffic situation e.g. a moving object is in front of the car, which is then used to control the vehicle to take the appropriate action e.g. reduce speed to avoid collision.

1.2 Parallel Computing

The needs for faster processing times, along with stagnating clock speeds (see figure 1.1) due to power budgets [25], has led to the rise of parallel multicore systems usage able to increase performance at the same clock speed.



(a) Clock speeds plateauing over the years as the number of transistors in chips increase



(b) Variation of processor sizes over the years

Figure 1.1: Variation trends of processor clock speeds and sizes. Images courtesy of Nature 530, 144147 (11 February 2016), (a) Intel (b) SIA/SRC

1.2.1 Types of Parallelism

- Instruction-level Parallelism A well-known technique of concurrent execution of independent instructions. It exploits the implicit parallel operations that could be done in a loop or otherwise.

- Thread-level Parallelism A software capability where a program is divided in to several units that could be executed in parallel; this kind of parallelism could be on a single-core or multicore system.

- Task-level Parallelism (Function Parallelism) A method of dividing an application into multiple tasks such that their execution could be distributed across processing cores.

- Data-level Parallelism According to Flynn's taxonomy [26], Single-Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) are types of data parallelism where data is distributed across threads executing the same (or different) instructions in parallel.

1.3 Real-time Sensor Data Processing

Since the evolution of General Purpose GPU computing (GPGPU) over the last decade, several research studies have applied GPUs in different intelligent vehicle ADAS applications: [27], [28] and [29] used GPU architectures to classify pedestrians on the road. [30] computed occupancy grids on the GPU to detect road boundaries, while [31] and [32] were interested in detecting traffic signs using a GPU implementation. A hybrid GPU-FPGA architecture was used in [33] where the feature extraction was implemented on an FPGA, and the SVM-based pedestrian classification was done on the GPU. A GPU-based implementation of HOG pedestrian detectors was developed in [34] and [35]. Real-time depth information was computed on the NVIDIA Tegra X1 [36] at 42 fps, using 4-path semi-global matching of stereo views in [37].

In addition, a complete end-to-end computer system [38] that performs learning, inference and vehicle control decisions is usually sought after, and has been popular in the last two years.

1.3.1 Why we need Parallel (Multicore) Computing for Perception Systems

Data from real-world sensors such as cameras and LiDARs are known to be bandwidth heavy. In addition, almost all developed perception algorithms require lots of computation, whether to achieve convergence or accuracies acceptable for real-life deployment. The current resurgence in neural network research is mainly attributed to the success of parallel computing architectures, the availability of programming models and development libraries, and the ease of their application. Even though tremendous success has been achieved in the last few years in object classification systems, that does not translate to better intelligence in real-life traffic situations, and using those systems in a world where the safety of traffic participants is dependent on real-time decisions made by the intelligent system is not just measured by how accurate the system can recognize objects in the scene.

1.4 Organization of this Thesis

This thesis addresses the general problem of real-world scene understanding from the embedded computing point of view for autonomous driving and assistance applications. Our scene understanding strategy can be divided into three parts: We start with scene *representation* in chapter 2 where we explore how to best represent sensor signal data in an unsupervised learning way using deep generative neural networks. Scene *reconstruction* using learned feature representations using classic statistical inference algorithms is discussed in chapter 3. An overview of recent developments in generative learning is also introduced in chapter 3. *Interpreting* scenes is crucial in machine vision applications and even more so to ensure the safety of traffic participants e.g. drivers, pedestrians, bicyclists. Chapter 4 continues the theme of probabilistic graphical models to infer objects and moving scene characteristics such as velocities.

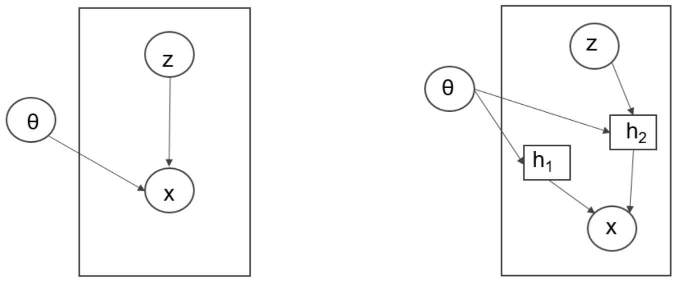
We introduce embedded vision computing and newly developed hardware architectures for deep learning and inference in chapter 5, followed by a discussion of NVIDIA's parallel processing models including the CUDA programming model and GPU architectures in chapter 6. We finish by conducting experiments to implement generative modeling on NVIDIA's embedded GPU architectures using our own collected traffic scene dataset. Program profiling and performance evaluation is also discussed in chapter 7, as well as quantitative analyses of program optimization techniques on embedded GPUs, and their results.

Chapter 2: Scene Representation Learning

2.1 Generative Models

Cognitive Perception: Computational neuroscientists model cognitive processes as non-linear interactions among a large number of simple, neuron-like processing units that form a neural network [39]. However, shallow architectures of the neural networks used cannot capture the complex processes presented; which is why several "*deep*" layers are required and are being used. Deep neural architectures, in contrast with the shallow architectures, have been argued to better represent complex sensory data [40]. Most of the learning of these non-linear interactions is done in an unsupervised manner; hence the modeling of those interactions or functions is "*generative*" – which models the latent (hidden) causes of the data – rather than "*discriminative*" where a certain classification or regression function is computed. Generative representations are good for finding meaningful latent representations for the data which can be helpful not just for classification tasks down the road, but are good for analytical reasoning that comprise a big part of intelligence.

Figure 2.1 classifies generative models into two categories: one that models observed data directly in an unsupervised manner, while the other kind of models assumes hidden layers (or variables) that are latent causes for the data. The second interpretation fits the general description of deep autoencoder networks, where input data is encoded as vectors and hidden representations are learned to best represent the raw data in a compressed state (or with lower dimensionality). To summarize, generative models always model the joint distribution of the data, in contrast with discriminative models that find the conditional distribution of some target variable given the observed evidence.



(a) Modeling observed data x as unknowns z distributed by a model with parameters θ

(b) A latent variable model where h_1 and h_2 are hidden variables

Figure 2.1: Types of Generative Models

2.2 Energy-based Generative Models

A building block of the more complex networks to follow is a Boltzmann Machine [41]. A 2-layer BM will consist of a visible layer and a fully-connected single hidden layer. Network activations are governed by an energy function defined as in equation 2.1. Inspired by statistical mechanics, the network state changes probabilistically to reach equilibrium a ”global energy minimum” gradually using a process known as simulated annealing [42]. A certain temperature parameter allows for increasing and decreasing the energy to prevent the optimization from getting stuck at a local minimum.

$$P(v,h) = \frac{e^{-E(v,h)}}{Z} \tag{2.1}$$

2.2.1 Restricted Boltzmann Machines

Removing the within-layer lateral connections in the bipartite graph of Boltzmann machines gives rise to a variant – the Restricted Boltzmann Machine (RBM). This special network structure allows for efficient learning and inference as the hidden layer nodes are conditionally independent given the visible layer nodes and vice-versa. This speeds up the learning and inference dramatically since there will be no need to use maximum likelihood estimation to compute the model parameters since this requires learning the Markov chain until convergence which is usually exponential in running time.

Assuming binary hidden and visible input variables; i.e. $v \in \{0,1\}^V$ and $h \in \{0,1\}^H$, the energy of the network can be expressed as:

$$\begin{aligned} E(v,h) &= -v^T W h - b^T v c^T h \\ &= -\sum_{i=1}^{V}\sum_{j=1}^{H} W_{ij} v_i h_j - \sum_{i=1}^{V} b_i v_i - \sum_{j=1}^{H} c_j h_j \end{aligned} \tag{2.2}$$

Since modeling binary inputs is not always appropriate; if we have real-valued inputs $v \in \mathbb{R}^{\mathbb{V}}$, the Gaussian-Bernoulli variant [43] is usually used. In this case, the energy is given as:

$$E(v,h) = -\sum_{i=1}^{V}\sum_{j=1}^{H} W_{ij} \frac{v_i}{\sigma} h_j - \sum_{i=1}^{V} \frac{(v_i - b_i)^2}{2\sigma^2} - \sum_{j=1}^{H} c_j h_j \tag{2.3}$$

The variance σ^2 is usually a predetermined parameter [44], although it can also be learned.

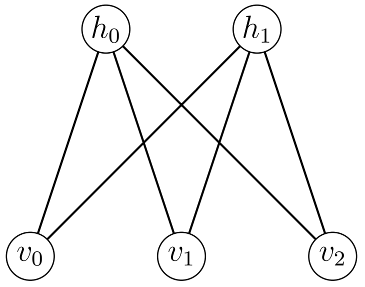


Figure 2.2: A graphical representation of a Restricted Boltzmann Machine

Markov Chain Monte Carlo sampling algorithms:

Usually, when one wants to find a model that best fits certain data, one would want to find the model parameters that can explain the data with minimum error. This error is usually measured as the Kullback-Liebler divergence between the two distributions and defined for discrete distributions D and M as the expectation of the logarithmic distance taken over the distribution D.

$$d_{KL}(D||M) = \sum_i D(i) log \frac{D(i)}{M(i)} \tag{2.4}$$

A faster, more tractable way – instead of using Maximum Likelihood Estimation – to find the model parameters is to form a Markov chain of the data variables states, collect samples from that target distribution and repeat till the chain best approximates the target (data) distribution. This is the general idea of several MCMC sampling algorithm variations including Gibbs sampling. In Gibbs sampling, described by equations (2.5 and 2.6), this Markov chain is constructed such that a variable is sampled at a certain step, given the values of all other variables at that step. We can use the structure of the graph to our advantage here as the conditional independences between the variables can help speed up the process when used.

$$\begin{aligned} P(h|v;\theta) &= \prod_j p(h_j|v) \\ p(h_j = 1|v) &= g(\sum_i W_{ij} v_i + b_i) \end{aligned} \tag{2.5}$$

$$\begin{aligned} P(v|h;\theta) &= \prod_i p(v_i|h) \\ p(v_i = 1|h) &= g(\sum_j W_{ji} h_j + c_j) \end{aligned} \tag{2.6}$$

2.2.2 Deep-Belief Networks

A deep belief network [40] is a hierarchical probabilistic generative model composed of one undirected layer (an RBM) and multiple directed layers, a sigmoidal-belief network. The network is usually greedily trained layer-by-layer [40], [45] until the complex structure of the input sensory data is captured within the hidden units. Stacking layers allows for a better representation of the data as a single RBM is only 2-layer deep. Figures 2.3 and 2.4 show a deep belief network formed by stacking three RBM layers.

Learning is defined here as maximizing the likelihood of the observed data. In general, learning Boltzmann machines requires sampling from the joint distribution of the visible and hidden layer nodes to compute variable correlations. Given the hidden layer nodes, we can sample the visible layer nodes in parallel since they are then conditionally independent. This variation of sampling is called block Gibbs sampling.

Bottom-Up Layer Greedy Training: Let θ^i be the parameters of layer i in the network.

1. Fit the input data to the first RBM layer, i.e. find the parameters θ^1.

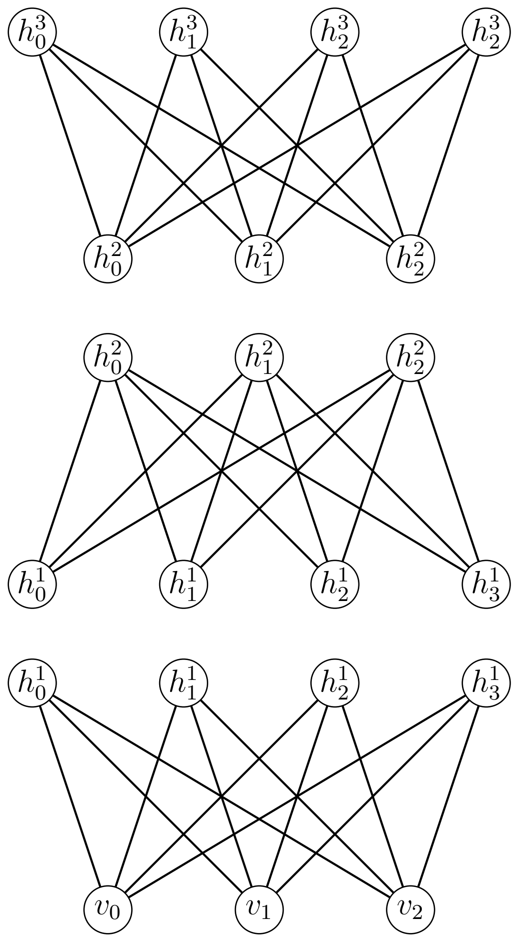


Figure 2.3: Three single-layered RBMs

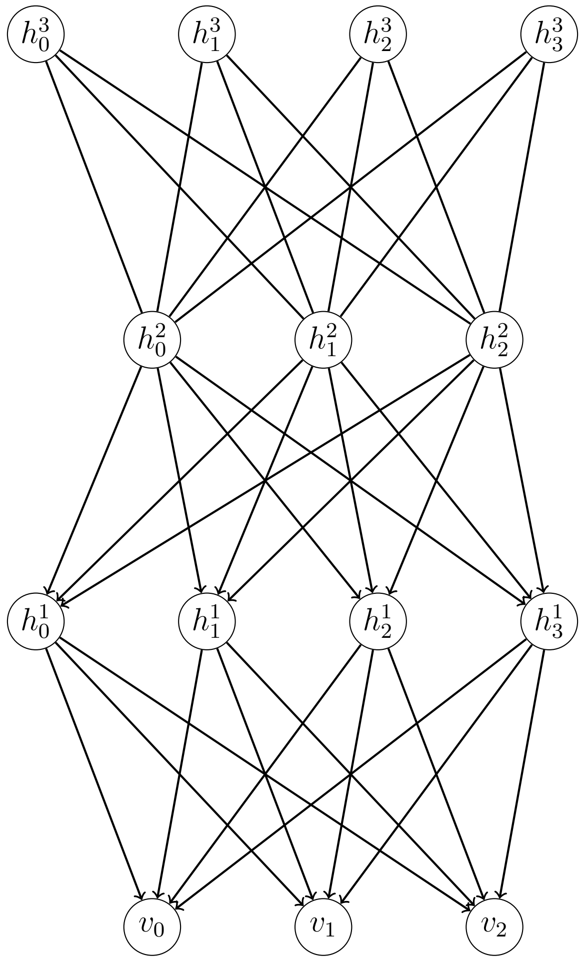


Figure 2.4: Three-layer Deep Belief Network

2. Use θ^1 as initial θ^2 to ensure that the 2-layer network is at least as good as the single RBM (θ^2 = transpose(θ^1)). Sample h^1 from the approximate posterior distribution $Q(h^1|v)$, which is the true distribution initially $P(h^1|v;\theta^1)$, and use as training data for θ^2.

3. Use θ^2 as initial θ^3 as θ^3 = transpose(θ^2)). Sample h^2 from the distribution $Q(h^2|h^1)$ $= P(h^2|h^1;\theta^2)$ and use as training data for θ^3.

4. Repeat recursively for the remaining layers till all θ^i are learned.

Analysis: For any $Q(h^{j-1}|h^j)$, the log-likelihood $logP(h^j;\theta^j)$:

$$\begin{aligned} logP(h^j;\theta^j) &= \sum_{h^{j-1}} logP(h^j,h^{j-1};\theta^j) \\ &= log\sum_{h^{j-1}} P(h^j,h^{j-1};\theta^j)\frac{Q(h^{j-1}|h^j)}{Q(h^{j-1}|h^j)} \end{aligned} \tag{2.7}$$

Using Jensens Inequality [46] gives a variational lower bound for the likelihood:

$$\begin{aligned} &\geq \sum_{h^{j-1}} Q(h^{j-1}|h^j)log\frac{P(h^j,h^{j-1};\theta^j)}{Q(h^{j-1}|h^j)} = \sum_{h^{j-1}} Q(h^{j-1}|h^j)logP(h^j,h^{j-1};\theta^j) \\ &\quad + \sum_{h^{j-1}} Q(h^{j-1}|h^j)log\frac{1}{Q(h^{j-1}|h^j)} \\ &\qquad = \sum_{h^{j-1}} Q(h^{j-1}|h^j)[logP(h^j,h^{j-1};\theta^j) \\ &\quad +logP(h^{j-1};\theta^j)] + H(Q(h^{j-1}|h^j)) \end{aligned} \tag{2.8}$$

Fixing θ^{j-1} and maximizing the variational lower bound is equivalent to maximizing the likelihood when sampling h^{j-1} from $Q(h^{j-1}|h^j)$ as it maximizes $\sum_{h^{j-1}} Q(h^{j-1}|h^j)logP(h^{j-1};\theta^j)$; which means that the next layer will learn a better posterior model given h^{j-1}, and so on.

Inference: To infer the values of the top-hidden variables, a single bottom-up pass is used where sampling is done from a fully-factorized approximate posterior distribution:

$$\tilde{Q}(h^1, h^2, ..., h^L|v) = \prod_{i=1}^{L} \tilde{Q}(h^i|v) \tag{2.9}$$

where L is the number of network layers – instead of the non-factorized multimodal form :

$$Q(h^1, h^2, ..., h^L|v) = Q(h^1|v)Q(h^2|h^1)...Q(h^L|h^{L-1}) \tag{2.10}$$

2.2.3 Deep Boltzmann Machines

Another version of deep graphical models that can learn complex internal representations, the Deep Boltzmann Machine (DBM) [47], is different from the DBN in the sense that it is completely undirected, i.e. a Markov Random Field. This allows for top-down feedback after the initial bottom-up pass during training and inference. As a result, the DBM can improve on the learned intermediate features which could result in better representations.

The energy of the DBM, the probability over the visible layer and the conditional distributions in figure 2.5 can be expressed respectively as:

$$E(v, h^1, h^2, h^3) = -v^T W^1 h^1 - h^{1T} W^2 h^2 - h^{2T} W^3 h^3 \tag{2.11}$$

$$P(v) = \frac{\sum_{h^1,h^2,h^3} e^{-E(v,h^1,h^2,h^3)}}{Z} \tag{2.12}$$

$$p(h^1_j = 1|v, h^2) = g(\sum_i W^1_{ij} v_i + \sum_k W^2_{jk} h^2_k) \tag{2.13}$$

$$p(h^2_k = 1|h^1, h^3) = g(\sum_j W^2_{jk} h^1_j + \sum_m W^3_{km} h^3_m) \tag{2.14}$$

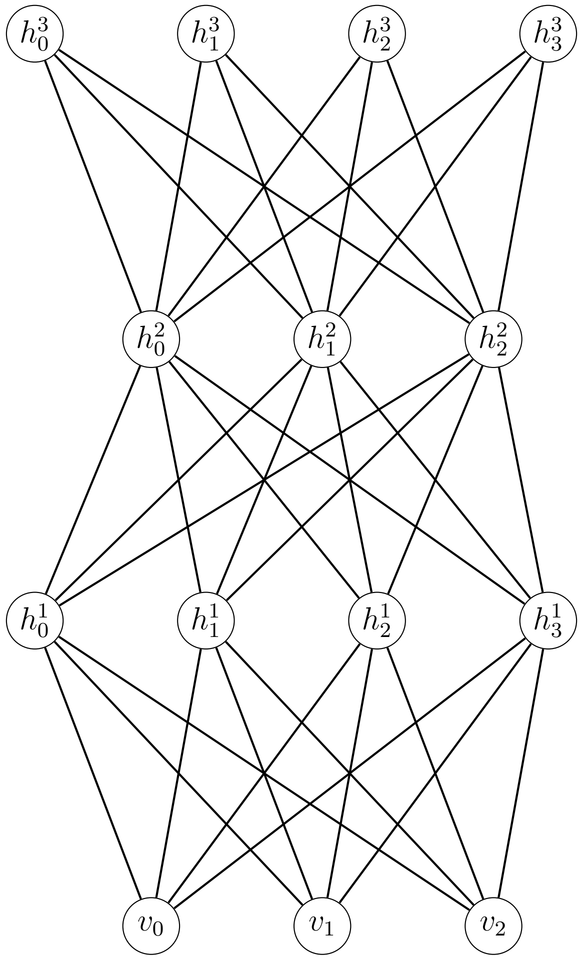


Figure 2.5: Three-layer Deep Boltzmann Machine with no within-layer connections

$$p(h^3_m = 1|h^2) = g(\sum_k W^3_{km} h^2_k) \tag{2.15}$$

$$p(v_i = 1|h^1) = g(\sum_j W^1_{ij} h^1_j) \tag{2.16}$$

Learning DBMs is done through maximizing the log-likelihoods with respect to the model parameters W^1, W^2, and W^3 by computing the derivatives $\frac{\partial logP(v)}{\partial W^1}$, $\frac{\partial logP(v)}{\partial W^2}$, and $\frac{\partial logP(v)}{\partial W^3}$.

Exact computation of the derivatives requires computing the data and model expectations which is exponential in the number of hidden, and hidden and visible variables respectively. A variational approach using mean-field inference is used to approximate the data expectations while MCMC sampling is used for model expectations [47].

2.3 Evaluating Generative Models

Representation results shown by generative models should be evaluated directly with respect to the applications for which they were intended [48]. One could use performance metrics to quantitatively analyze the performance of probabilistic models such as *fidelity* and *coverage*. The model's fidelity is described by how much of the synthesized generated data points resemble the actual training data points, while coverage quantifies the data distribution a generated sample represents. Average log-likelihood or KL-Divergence criteria are default criteria used to optimize generative models. The following three distance measures can be used to assess the similarity between two sets of data points based on the nearest neighbor concept of metric spaces: one being the training dataset and the other is the synthesized or the generated sample data. It is to be noted that only the third metric can be used as a measure for log-likelihoods and KL-Divergence criteria.

- Chamfer Distance Transform defined as: the sum of closest point distances between sets X and Y

$$ch(X,Y) = \sum_{x \in X} \min_{y \in Y} \|x - y\| \tag{2.17}$$

- Hausdorff Distance defined as: the maximum distance of set X to the closest point in set Y

$$h(X,Y) = \max_{x \in X} \min_{y \in Y} \|x - y\| \tag{2.18}$$

- Wasserstein Metric can be used as a distance metric between two probability distributions P and Q defined as:

$$L(P,Q) = \frac{\sum\limits_{i=1}^{m} \sum\limits_{j=1}^{n} f_{i,j} d_{i,j}}{\sum\limits_{i=1}^{m} \sum\limits_{j=1}^{n} f_{i,j}} \tag{2.19}$$

  Define P and Q as two cluster distributions of points p_i and q_j respectively, where $1 \leq i \leq m$ and $1 \leq j \leq n$; the optimal flow f that minimizes the distance d between the two distributions P and Q is to be found using the optimization problem: $\min \sum\limits_{i=1}^{m} \sum\limits_{j=1}^{n} f_{i,j} d_{i,j}$, where $d_{i,j}$ is the distance between clusters p_i and q_j, subjected to the following constraints:

  (a) $f_{i,j} \geq 0; 1 \leq i \leq m, 1 \leq j \leq n$

  (b) $\sum\limits_{j=1}^{n} f_{i,j} \leq 1; \sum\limits_{i=1}^{m} f_{i,j} \leq 1$

  (c) $\sum\limits_{i=1}^{m} \sum\limits_{j=1}^{n} f_{i,j} = \min\{m, n\}$

For image data, a subjective evaluation based on the visual fidelity of samples is usually appropriate. [49] developed an image quality metric using cues from human perception of distortions by assigning a score from 0 to 100% to blocks in an image. Such score represents the perceptual quality of the image; a higher score indicates a poorer quality image and vice versa. Algorithm steps used to compute the perception score are summarized in figure 2.6. Natural scene statistics of the input image are first computed and a normalization operation is applied to each pixel intensity value as in equation 2.20. The image is then segmented into 16×16 non-overlapping blocks that are labeled either spacially-active (SA) or not (U) using the variance parameter of the block. For each spatially-active block, a Noticable Distortion Criterion (NDC) and Noise Criterion (NC) are computed. The block distortion is then quantified using its variance based on the NDC and NC values computed for that block. (We refer the reader to [49] for calculation details.) Finally, the PIQUE score is computed as shown in equation 2.21.

$$\mu(i,j) = \sum_{k=-3}^{3} \sum_{l=-3}^{3} w_{k,l} I_{k,l}(i,j)$$

$$\sigma(i,j) = \sqrt{\sum_{k=-3}^{3} \sum_{l=-3}^{3} w_{k,l} (I_{k,l}(i,j) - \mu(i,j))^2} \tag{2.20}$$

$$\hat{I}(i,j) = \frac{I(i,j) - \mu(i,j)}{\sigma(i,j) + 1}$$

where $I(i,j)$ is the intensity value at pixel (i,j) and $w_{k,l}$ is a 2D symmetric Gaussian weighting function

$$PIQUE = \frac{\sum_{k=1}^{N_{SA}} D_{sk} + 1}{N_{SA} + 1} \tag{2.21}$$

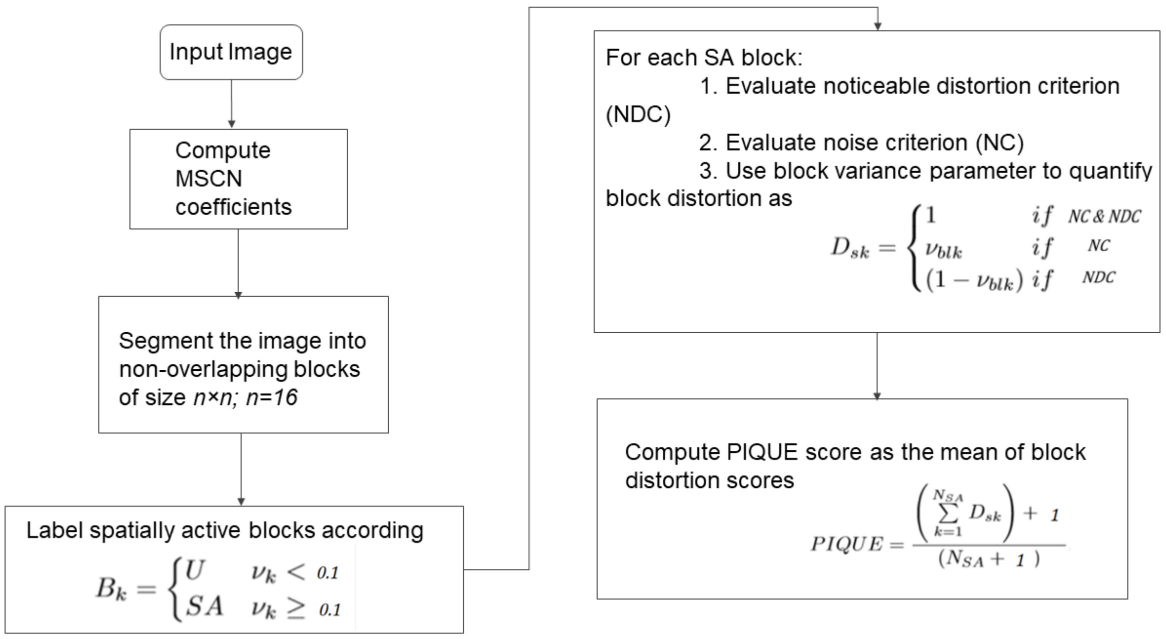

Input Image
Compute MSCN coefficients
Segment the image into non-overlapping blocks of size n×n; n=16
Label spatially active blocks according
$B_k = \begin{cases} U & \nu_k < 0.1 \\ SA & \nu_k \geq 0.1 \end{cases}$
For each SA block:
1. Evaluate noticeable distortion criterion (NDC)
2. Evaluate noise criterion (NC)
3. Use block variance parameter to quantify block distortion as
$D_{sk} = \begin{cases} 1 & if\ NC\,\&\,NDC \\ \nu_{blk} & if\ NC \\ (1-\nu_{blk}) & if\ NDC \end{cases}$
Compute PIQUE score as the mean of block distortion scores
$PIQUE = \frac{\left(\sum_{k=1}^{N_{SA}} D_{sk}\right) + 1}{(N_{SA} + 1)}$

Figure 2.6: Algorithm steps used to compute PIQUE perception scores of images

where N_{SA} is the number of spacially active blocks in the image and D_{sk} is the block distortion parameter defined in equation 2.22.

$$D_{sk} = \begin{cases} 1 & , NC\, and\, NDC \\ \nu_{block} & , NC \\ 1 - \nu_{block} & , NDC \end{cases} \tag{2.22}$$

Evaluation metrics that use statistical features of an image such as PIQUE don't require a reference training image set. In addition, they correlate better with subjective human quality scores [49] than using log-likelihood metrics when judging image samples, as these metrics are not indicative of the quality of samples produced [48].

Chapter 3: Scene Reconstruction

We formulate reconstruction here as an inference problem: finding the maximum likelihood distribution of the observed data from sensor signals, in other words, finding the exact inference is intractable, and approximating the solution, either by variational approximations or having enough samples can recover important information about the distribution. What follows is a brief discussion of two classical inference methods that have been previously studied extensively over the years, and generally require an optimization approach to do inference i.e. finding likelihood or Maximum A Posteriori (MAP) estimations, followed by a quick review of more recent inference algorithms that fall under the umbrella of *differentiable* inference algorithms, where tuning the inference procedure can be done end-to-end, since they have differentiable loss functions.

3.1 Classic Inference Algorithms

3.1.1 Markov Chain Monte Carlo (MCMC)

MCMC methods rely on constructing a Markov chain with a stationary probability distribution as the unknown distribution of interest, i.e. running the chain for a few steps will generate samples that approximate the unknown probability distribution. These methods are very general and can theoretically approximate any distribution of interest [50].

3.1.2 Variational Inference

Instead of using MCMC methods: constructing a Markov chain over the hidden variables that represents the posterior distribution, running it to convergence (equilibrium) and collecting samples that approximate that posterior, one can define a parametric distribution that approximates the posterior and find its best set of parameters. In this case, the inference becomes an optimization problem. Since this optimization problem is also intractable just like the MCMC methods, stochastic optimization methods (e.g. stochastic gradient descent) is preferred. In this case, the minimum of the cost function is reached by using noisy estimates of the gradient. Also, representing the cost function as a sum of several independent terms allows for faster gradient computations; a technique called *mean-field approximations* that is generally used [50].

3.2 Differentiable Inference Algorithms

3.2.1 Variational Autoencoders (VAEs)

Defined in 2013 in [51], variational autoencoders (VAEs) assume local latent variables z for datapoints x. As usual, inference is defined as finding the posterior distribution $p(z|x)$. Using variational inference, we can approximate the posterior with variational distributions $q_\lambda(z|x)$. Instead of minimizing the KL-divergence to find the best family of distributions λ that approximate the posterior, since it's intractable; an Evidence Lower BOund is usually maximized using equation 3.1:

$$ELBO(\lambda) = E_q[log(p(x, z))] - E_q[log(q_\lambda(z|x))] \tag{3.1}$$

For a VAE, the ELBO for each x_i can then be written as equation 3.2 and λ is found using Stochastic Gradient Descent algorithms [51].

$$ELBO_i(\lambda) = E_{q_\lambda(z|x_i)}[log(p(x_i|z))] - KL(q||p) \tag{3.2}$$

We then form two networks: an *encoder* network that encodes input data x with parameters λ, followed by a *decoder* network that outputs the likelihood data distribution $p(x|z)$ given the latent variables from the encoder network [51]. Decoder network parameters are found using variational Expectation Maximization [51], i.e. finding the model parameters by maximizing the likelihood of the data with respect to the parameters, in a variational way i.e. maximizing the ELBO with respect to the model parameters [51].

3.2.2 Generative Adversarial Networks (GANs)

Generative Adversarial Models [52] have been gaining interest in the past couple of years in deep learning research. While discriminative models are very good at object classification, they fail at reasoning tasks that generative models can accomplish. A GAN model where both a discriminative and a generative model are trained simultaneously is defined [52]. The goal of the generative model (as the case with all generative models) is to capture the data distribution, while the goal of the discriminative model is to estimate that a drawn sample is more likely to have come from the training data rather than generated from its adversary i.e. by the generative model.

This translates to two simultaneous optimization scenarios, where one model D tries to maximize the probability that it assigns the correct label to samples generated from G, and

G tries to minimize the success of D i.e. $log(1-D)$. This results in the following optimization scenario:

$$min_G max_D = E_X[log(D(x))] + E_Z[log(1 - D(G(z)))] \tag{3.3}$$

Minibatch Stochastic Gradient Learning is used to learn network parameters by alternating ascents and descents to both discriminator and generator networks respectively [52].

3.2.3 Invertible Density Estimation Models

They are classes of probabilistic generative models that are invertible. Instead of starting out with observations x, one starts from the latent space and samples z. Latent samples are then passed through a *deterministic* function $g(.)$, which results in exact sampling. One can argue that the inference is exact also, as function g has to be chosen bijective, where there is one-to-one correspondence between both spaces. Thus, given any observed input x, one can exactly infer z. Finally, assuming the generative function g is known and bijective, and given random variable z, the likelihood function for x can be expressed in terms of g and z, giving an exact likelihood computation as shown in equation 3.4.

$$P_X(x) = P_Z(f(x))det(\partial(f(x))/\partial(x^T)) \tag{3.4}$$

The challenging tasks to solve now are computing the Jacobian $\partial(f(x))/\partial(x^T)$ as well as its determinant when the number of input variables is large. In [53] and [54], function f was learned by stacking individual bijection layers y in an affine way (see equation 3.5), and propagating through the layers in a certain alternating pattern that makes the Jacobian of the transformation f a triangular matrix, thus its determinant can be computed easily by multiplying the diagonal elements.

$$\begin{aligned} y_{1:d} &= x_{1:d} \\ y_{d+1:D} &= x_{d+1:D} \bigodot exp(h(x_{1:d})) + k(x_{1:d}) \end{aligned} \tag{3.5}$$

3.3 Conclusion

The above mentioned methods all solve the problem of reconstructing data from hierarchical features learned by generative modeling of the data, i.e. finding a latent distribution that we assumed to have generated the data. They mostly rely on the Maximum-Likelihood principle, and use either sampling (MCMC) or variational distributions and lower bounds to approximate the intractable inference problem. VAE adds neural networks (similar to an autoencoder), with a reconstruction cost or loss function, while GANs avoid maximum-likelihood altogether and use a discriminative model instead. Invertible density estimations rely on finding invertible transformations that can relate the input data and latent variables. It is our future goal to further study these models on real-world data in real-time environments.

Chapter 4: Scene Interpretation and Object Classification

The main task of machine vision is to understand real-world scenes, as a result, much work has been done in this area. Since the problem is usually under-determined [55], there are too many scene attributes in natural images to be solved for, most methods involve statistical estimation or even optimization theory. Prior statistical feature models are often handcrafted or tweaked for a better fit to the data. Some approaches use synthetic scenes using simulations and graphics techniques to control for some scene attributes like shading and color models [56], [57]. Others implement statistical models and use learning-based approaches to estimate the model's parameters [58]. This chapter introduces three different scene interpretation models that are founded in Bayesian statistical methods.

4.1 Naive Bayes Classification

We define an image as a random field of independently and identically distributed 4D random variables or *pixels*, each pixel belongs to a discrete object class with prior distribution α which represents the probability distribution of each object class given RGBD values. We then define a probabilistic model to represent the image using pixel RGB values $\{R, G, B\}$, and depth measures $\{D\}$ calculated from the stereo pair of cameras, as evidence (observations). Assuming all observations to be conditionally independent given the pixel class, using the chain rule of conditional probability, we can write the joint distribution as:

$$P(pixel, R, G, B, D; w) = \alpha.P(R|pixel; w).P(G|pixel; w).P(B|pixel; w).P(D|pixel; w) \tag{4.1}$$

where w is the hidden-visible weight activations in the deep neural network, i.e. image pixel (4D) distribution.

For an object detection/classification task, one might wish to find the posterior $P(pixel = class|R, G, B, D; w)$. Using the Bayesian inference framework, posterior integrals are intractable [59] which leads to adopting approximate methods of estimating that posterior (please refer to chapter 3 for inference methods). Thus, the classification problem becomes an inference problem [60].

4.2 Probabilistic Markov Networks

Here, the problem is formulated as given two images from a video sequence, infer the velocities of the moving objects in the scene. This problem is especially interesting in automated driving and ADAS, since motion is the main component of these applications, and inferring current and future states of moving participants in the traffic scene is often advantageous.

To define this *optical flow* estimation problem, we first model two consecutive image frames as an undirected graph $G(V, E)$, with observations $I_j \epsilon V$, and unknown scene quantities or explanations $h_j \epsilon V$, where $j \epsilon \{1, 2, 3, ..., 2N\}$; and N is the number of image pixels equal to the image size $N = m \times n$. Nodes represent states evolving over time such that I_j is observed dependent on hidden state h_j at the same time step. Edges in the graph represent statistical dependencies between nodes. An example is given in figure 4.1.

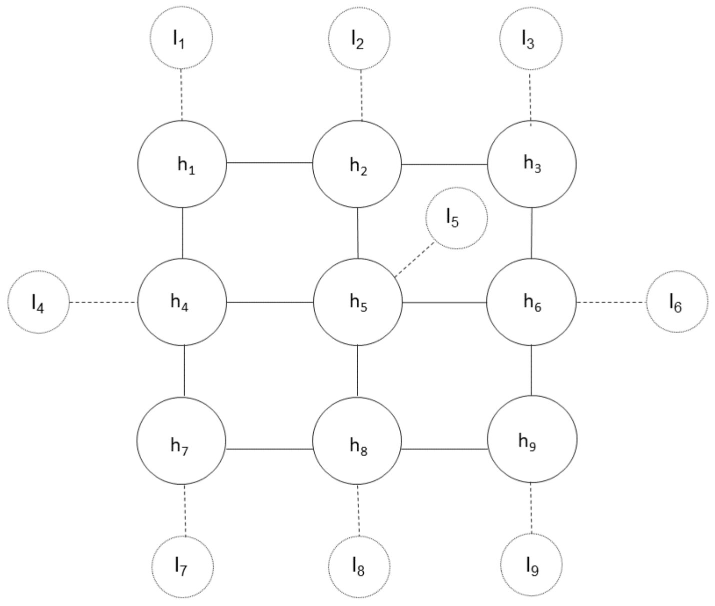


Figure 4.1: Representing an image as a Markov network

The undirected graph in figure 4.1 represents a Markov network, where the Markov property [50] is satisfied between the nodes. The graph represents the joint probability distribution $P(h, I)$ given in equation 4.2 [61].

$$P(h_1, h_2, ..., h_{2N}, I_1, I_2, ..., I_{2N}) = \prod_{\substack{\text{neighboring} \\ i,j}} \Psi(h_i, h_j) \prod_k \Phi(h_k, I_k) \tag{4.2}$$

The scene-scene graph potential Ψ and image-scene potential Φ, also known as *compatibility functions* [61], [55] can both be learned from the graph statistics e.g. co-occurence histograms of the training data or modeled as mixture of gaussians [61]. It is to be noted that for real-world scenes, learning an exact representation for the potentials is intractable due to the large number (almost infinite) of scene/image pairs available.

Image-Scene Estimation

Recall that an "image" is a concatenated set of pixels from consecutive time frames, and a "scene" is the projected velocity at those pixels. To generate scene estimates, the likelihood of different optical flow vectors at a pixel is calculated using the directional image derivative $\nabla I \cdot v = -\frac{\delta I}{\delta t}$, although a least squares estimate is usually used since the equality is not exact for real-world images [55].

MAP Estimation using Belief Propagation

Now, we go back to the Markov assumption and figure 4.1 to use an example to estimate the hidden states given all observations. This translates to a Maximum A Posteriori (MAP) problem that can be easily solved by belief propagation in undirected graphs e.g. Markov Random Fields [61], [55], [50], [62]. MAP estimation at node j is given in equation 4.3

$$\hat{h_j}^{\mathrm{MAP}} = \underset{h_j}{\arg\max}\, \Phi(h_j, I_j) \prod_k M_j^k \tag{4.3}$$

where M_j^k is the message sent from node k to node j and is given by equation 4.4

$$M_j^k = \max_{h_k} \Psi(h_k, h_j)\Phi(h_k, I_k) \prod_{l \neq k} \tilde{M_l^k} \tag{4.4}$$

where $\tilde{M_l^k}$ is the message from the previous iteration. Repeated iterations of calculating equations 4.3 and 4.4 for every hidden node until convergence will give the estimated scene value at the specified image node. The complexity of this MAP elimination algorithm is bounded by the size of the largest fully-connected loop in the graph, the graph size, and is exponential in the image size [50].

Chapter 5: Embedded Vision Processing on SoCs

The past two decades have seen tremendous advances in machine vision applications in consumer products from smartphones and video analytics solutions to automotive safety systems. In contrast to traditional computer vision applications in industrial settings and automated manufacturing; implementation is usually constrained by cost, size and power consumption of system components. In addition, vision processing does not start with an image in the framebuffer [63], the entire real-time image processing pipeline starting from image acquision to output needs to be considered. Figure 5.1 shows the components that typically constitute a real-time vision processing system. The pipeline usually starts with the photoelectric conversion of input light rays into voltage using CMOS transistors which is passed through an ADC as digital pixels followed by their packet representation to be transferred for application processing using standard interfaces e.g. PCIe, USB, GigE. Application processing is done on multicore homogeneous or heterogeneous processing elements that are built on a single chip with their inter-network interfaces. Integrating said components with memory and advanced peripherals produces a System-on-Chip (SoC), that are predominant in the embedded systems industry at present and have replaced the older Digital Signal Processors (DSPs) since they offer more computational power.

Most FPGAs have a soft core that includes an SoC in addition to the reconfigurable fabric. This aids in creating powerful vision systems since they combine the flexibility of

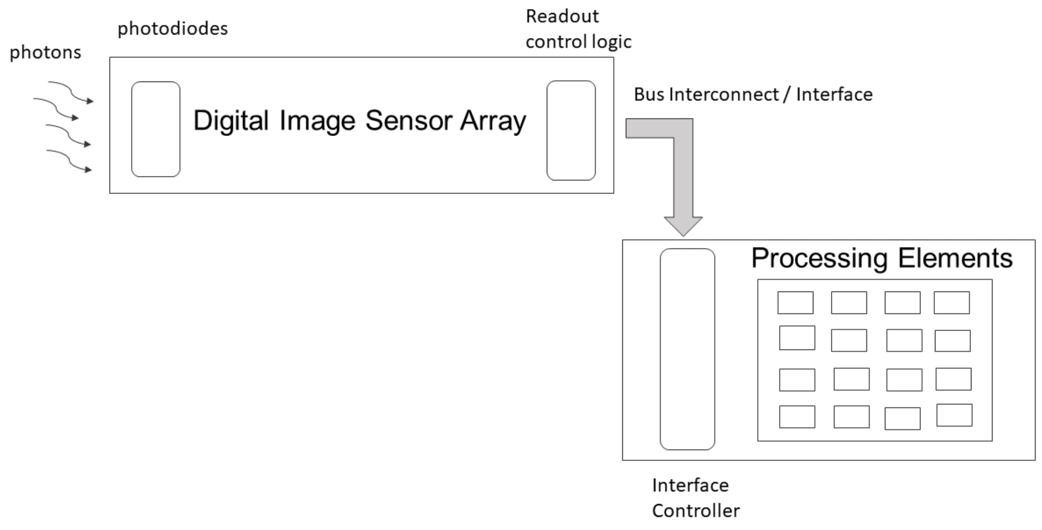


Figure 5.1: A very high-level view of a typical embedded processing system used in machine vision applications

adding custom hardware to the embedded peripherals to accelerate time-critical algorithms and frees precious programmable LUTs for application acceleration.

Beyond general purpose processors, Application Specific Integrated Circuit (ASIC) designs have existed since the 1980s [64] and have evolved with Moore's Law and Very Large Scale Integration (VLSI) technology into chips with more than one million transistors contain deep pipelining structures and massively connected parallel compute resources, e.g. Network-on-Chip (NoC) designs that facilitate high performance computing applications such as machine vision. Low manufacturing costs and power budgets helped the SoC evolution in the embedded industry, where all system components are on the same die, thus reducing power consumption with high compute resources.

5.1 Hardware Architectures for Embedded Vision Applications

Traditionally, low-level vision architectures utilized 1D arrays or 2D meshes of processing elements for applications such as edge detection and image smoothing operations. The addition of DSP cores to multiprocessors on an SoC, e.g. Texas Instruments' DaVinci family of processors [65], helped in implementing more complex applications like video processing. Nowadays, all systems have dedicated accelerator cores in addition to main CPUs where application specific vision algorithms are implemented. Dedicated cores could be GPUs, video processing units or even FPGA-implemented accelerator cores. Examples of different types of hardware accelerator chips used in machine vision are described next. By no means this list is exhaustive as the field is evolving rapidly and new architectures are designed everyday.

5.1.1 Hardware Accelerators Examples

A number of accelerators and vision and deep learning specific purpose accelerators have emerged in the past few years. Below are some of the most prominent.

- Google Tensor Processing Units [66]

  Their deep learning capabilities are accelerated by a Matrix Mutliply and Accumulate (MMA) 256x256 array of 8-bit multipliers.

- NVIDIA's Tesla V100 Chip [67]

  The 640 Tensor cores on this chip are also designed to accelerate MMA operations.

- Mobileye's EyeQ [68]

  This family of processors is designed specifically for automotive driving and ADAS applications.

- Intel's Knights Mill Chip (New generation Xeon Phi) [69]

  The individual cores on this many-core chip are smaller where inner loops can fit in L1-instruction caches. As a result, the performance of cores per socket is decreased but the number of cores is greater, which is good for compute intensive applications like deep networks. Its design is said to be a midway between a server CPU and a hardware accelerator.

- ThinCI's Chip [70]

  This chip incorporates small processors and a thread scheduler analogous to a CPU with execution units and an instruction scheduler. The new feature here is that the processors can stream data to each other instead of having to load/store from RAM each time a computation is needed.

- Data-Flow Engines

  One of the most sophisticated accelerator types. The main design is focused on how to map data graphs onto the data flow processing nodes to maximize computation speed and minimize Inter Process Communication overhead and synchronizations. They are basically many-core coprocessors featuring a network on a chip scratchpad memory model, suitable for a dataflow programming model, which should be suitable for many machine learning tasks. Examples are the Adapteva Epiphany processors [71], and Wave Computing's DPU [72].

- Movidius Vision Processing Units (VPU) [73]

  A multicore processor family with features fairly consistent with vision processing units that handle SIMD instructions and datatypes suitable for video with an on-chip DMA between scratchpad memories.

- GPU-based Accelerators

  Examples are NVIDIA's Tegra family of processors [74], and AMD's Radeon Instinct accelerators[75].

- Eyeriss [76]

  An FPGA-based accelerator for deep convolutional neural networks with low real-time energy consumption, that utilizes data reuse to avoid unnecessary reads and computations, and data compression to reduce memory bandwidth.

- Xilinx Automotive (XA) Spartan series [77]

  FPGAs designed specifically for ADAS applications.

- Synopsys' DesignWare EV5X processors [78], [79]

  A family of embedded vision processors with CNN capabilities.

- Intel's Nervana Neural Network Processor (NNP) [80]

  Another ASIC developed specifically for deep learning computations and memory operations.

- Neuromorphic-based Processors

  Integrated circuits design based on spiking neuron elements instead of traditional boolean logic gates, where a neuron fires a weighted range of values, in response to input stimuli within a certain period of time. Examples are the TrueNorth processors [81], and Intel's Loihi [82].

5.2 High-Level Software Frameworks

Despite the ease of directly using general parallel computing frameworks such *OpenMP* for shared-memory architectures, and *MPI* for distributed memory systems, on different multicore architectures to accelerate applications, that abstraction is usually not the best approach taken when designing embedded software. To get the most out of multicore systems, using programming models mapped to the system architecture that can well expose all types of parallelism on that system is recommended. When designing parallel programs, one should start with *partitioning* and breaking up computation among the different Processing Elements (PEs), either according to different functional steps in the algorithm, i.e. functional decomposition, or according to data to be processed, i.e. data decomposition [83]. After partitioning, *mapping* task groups to the available PEs is done. When mapping, data dependencies between tasks should be considered as well as reducing communication needs between PEs through the type of memory used: shared memory is usually fastest. Acceleration is thus achieved with increasing the total amount of work done per unit time, i.e. application *throughput*, or decreasing turnaround times and overheads, i.e. *latency.* The availabilty of profiling tools that can provide accurate space (memory) and time measurements can aid in the optimization process by providing feedback on the performance and adjusting accordingly.

5.3 Conclusion

New architectures are being developed everyday to speed up the processing of the huge number of pixel data collected from image sensors/cameras. Even though one might initially achieve fast processing times using the appropriate hardware or software; we usually optimize for acceleration by finding the best software implementation for a specific hardware implementation. Studying the interactions between hardware and software collectively at that optimization stage is suggested to achieve real-time performance requirements for many of the big data vision applications. Despite the availability of different accelerator types, a thorough study of an application's algorithm implementation on that accelerator instead of a generic solution can improve the algorithm's real-time performance moving forward.

Chapter 6: Heterogeneous Processing GPU Models

The definition of a multiprocessor system varies according to context. A traditional multiprocessor implies the use of more than one CPU to execute a task. Symmetric Multiprocessing systems adopt using a *homogeneous* set of CPUs with the same instruction set architecture, that share the main memory, while assymmetric systems could be either homogeneous or heterogeneous with their separate memory space.

This chapter discusses another type of heterogeneous multiprocessing: the usage of a co-processor as a hardware accelerator in addition to a host processor. Both processors can have the same architecture e.g. Intel MIC, or different architectures e.g. NVIDIA GPUs.

Multithreading on CPUs and GPUs Parallel processing on CPUs is not the same as that on GPUs. Even though both architectures use multiple threads to parallelize the execution of tasks, *Context Switching* on CPUs is much more expensive and slower than on GPUs. GPU threads are designed to maximize throughput by supporting a large number of threads in a streaming multiprocessor using schedulers (please refer to section 6.1.1 for more details) to allocate resources to active threads, in comparison with CPU cores that rely on separate copies of resources for each thread. This results in the capability of GPUs running thousands more threads concurrently than CPUs [84].

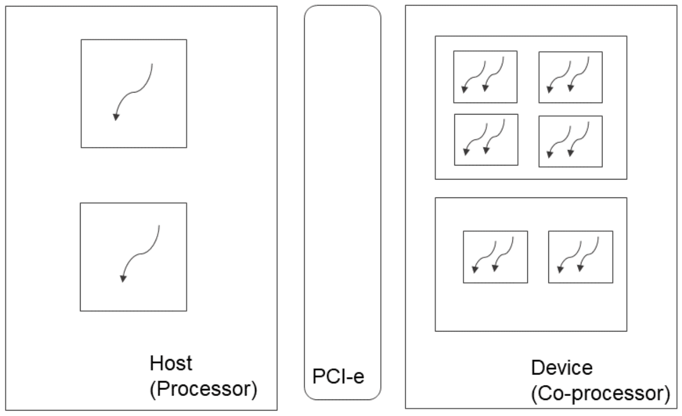


Figure 6.1: The Heterogeneous Computing Model

6.1 The CUDA Programming Model

The CUDA programming model assumes the heterogeneous processing model, where parallel code executes on a device separate from the main processor. Each processor has its own memory, with a high bandwidth PCIe bus used for data transfer between both memories. Parallel code is launched in a kernel by the host and runs concurrently on several *Streaming Multiprocessors* or SMs on the GPU. From a programmer's point of view, the functional tasks are in parallel code that is run in a grid of blocks of multiple threads, each of which runs the whole function of tasks on a part of the input data. Figure 6.1 shows the general heterogeneous computing model on which the CUDA framework is based.

6.1.1 Single Instruction Multiple Thread (SIMT) Execution Model

Unlike a CPU, a GPU is built of a scalable array of multiprocessors called *Streaming Multiprocessors* (SM) [85]. Streaming Multiprocessors can run one or more blocks of threads concurrently utilizing thread-level parallelism while instruction-level parallelism is exploited within single threads by pipelining instructions [85]. NVIDIA SMs partition thread blocks to groups of 32 threads called *warps*. Threads within a warp execute the same instruction, however each thread has its own register state to enable independent branching. This execution model called the Single Instruction Multiple Thread (SIMT) model is similar to Single Instruction Multiple Data (SIMD) models on CPUs but they can provide much lower latencies without lowering the throughput (especially in indirect memory accesses), as well as easier (albeit slower) divergence control e.g. SIMT does not use flag registers for conditional branching [85].

6.1.2 Memory Organization (excluding registers)

A GPU has several types of memory, each characterized by their size, latency and throughput. Due to the principle of data locality [86], computing performance is greatly affected by type of memory storage used for a particular task. Table 6.1 illustrates the different types of memory found in a GPU, along with their caching behavior and data lifetime.

6.2 Heterogeneous Memory Systems

6.2.1 Unified Memory

CUDA 6.0 introduced a managed memory model where a coherent memory image is shared across all processing elements [87]. In this model, all processors see a common memory

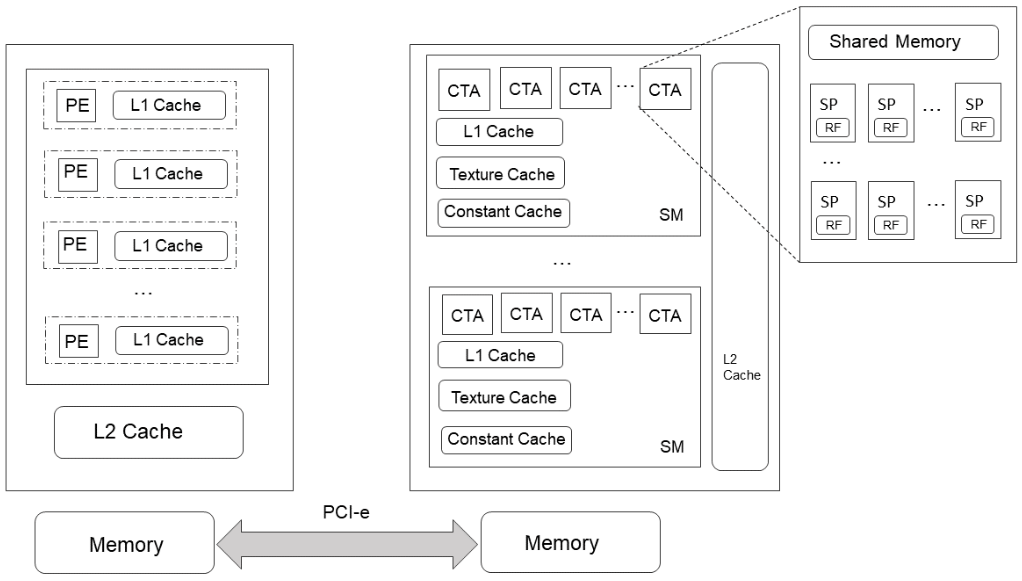


Figure 6.2: The Parallel Thread Execution Model

address space and no explicit data transfer is required between processors which is usually done using the traditional runtime API routines *cudaMemcpy()* [85]. This model should be useful for applications running on Tegra devices since both the CPU and GPU are on the same physical silicon chip and share the same DRAM.

The Tegra X1 SoC on the Jetson development board is a good application case for the unified memory model compared to the ones on the Drive PX system since the Jetson system doesn't have a discrete GPU unit.

Pinned Memory vs. Unified Memory

Even though the CPU and the GPU on Tegra devices share physical DRAM, accessing and caching this memory can be done in various ways. Thus, it is important to select the appropriate memory type for each application to ensure efficient program execution.

Table 6.1: Summary of different memory types on the GPU

Global Memory	Shared Memory	Local Memory	Constant Memory	Texture Memory
Off-chip DRAM	On-chip DRAM	Off-chip DRAM (Allocated in global memory)	Off-chip ROM	Off-chip ROM
Uncached	Behaves like L1 cache for a block on an SM	Uncached	Cached per SM	Cached per SM
High latency, low throughput	Low latency, high throughput	High latency	On a cache miss, the cost is one memory read	Accessed through a dedicated read-only cache
Host and device access	Within-block thread access	Individual thread access	Located in device memory and accessed through a special read-only cache	Located in device memory
Kernel Persistent	Lifetime of block	Lifetime of thread		

On traditional heterogeneous memory systems e.g. CPU with dGPUs, CPU host memory is pageable by default and not directly accessible by the GPU. This is the case when explicit data transfers using *cudaMemcpy()* routines are needed for the GPU to access CPU data memory. Although host memory is cached, data transfers can still slow down the application significantly. An alternative method is to pin host memory to the GPU through the Pinned (Non-pageable) Memory Model. In this method, data is allocated in pinned host memory instead, thus saving the CUDA driver the time of allocating a temporary pinned host data arrays before transferring them to the GPU, as in traditional pageable memory models.

Pinned Memory models are also known as Zero Copy Memory models since data allocated in host memory is accessible to device memory using the allocated pointers.

It is to be noted that Tegra X1 is compute capability 5.3, which is not cache coherent, thus data is not cached on the CPU which increases data access times [88].

On the other hand, unified memory is cached on all compute capability devices, which introduces a slight overhead on non-cache coherent devices like the TX1. Despite that overhead, it is still preferable to use unified memory over pinned memory for large arrays that are frequently accessed [88]. We aim to demonstrate that in sections 6.2.2 and 6.2.3.

What memory model to use?

The CUDA for Tegra guide [88] says that zero-copy or pinned memory is preferable to use in low latency applications, and that is due to the non-cached behavior of zero-copy memory. Memory selection is dependent on required kernel execution times, data transfers, data locality and latency.

Page Migration and Data Coherence

Unified Memory works in a similar way to Unified Virtual Addressing (UVA) in the sense that all system memory has a single virtual address space. However, unified memory automatically migrates data from one physical location to another [87], [89], [85] based on where the data needs to be accessed. From a programmer's standpoint, to use unified memory, one has to allocate dynamic memory using *cudaMallocManaged().* Behind the scenes, this translates to the following steps [89]:

1. New pages are allocated on the GPU

2. Old pages are unmapped from the CPU

3. Data is copied from CPU to GPU

4. New pages are mapped on the GPU

5. Old pages are freed from the CPU

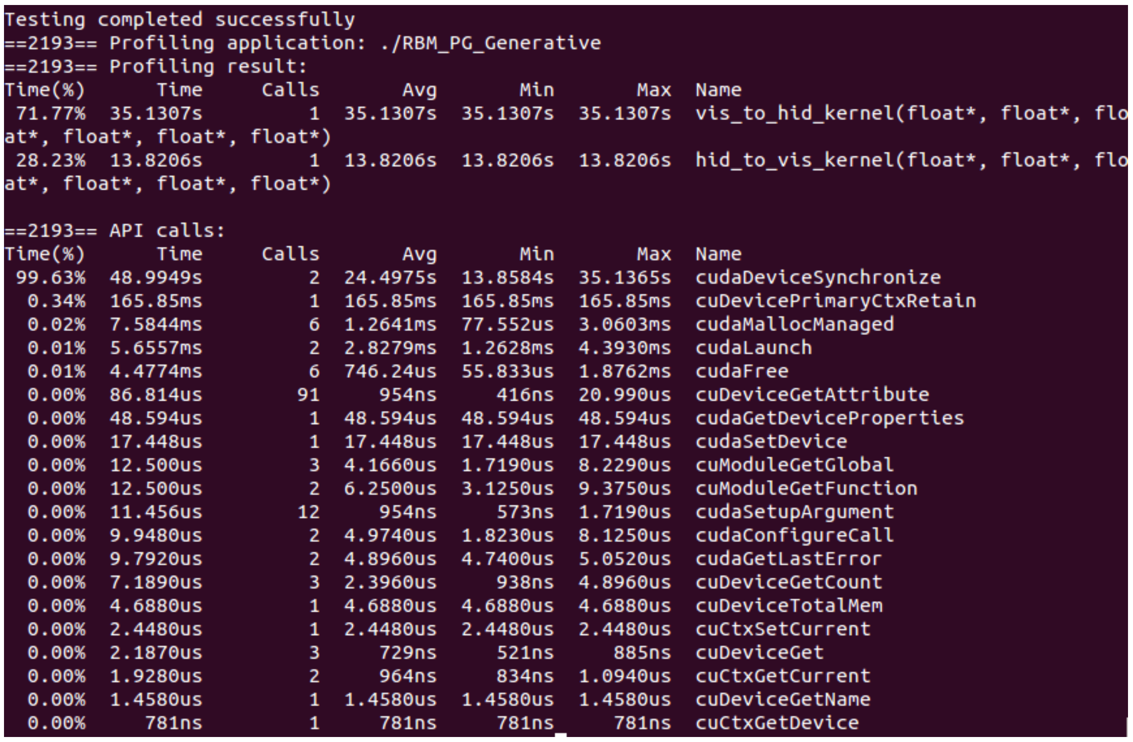
On older GPU architectures, e.g. Maxwell in the TX1, managed memory has to be synchronized between the CPU and the GPU before launching any kernels. Newer architectures of compute capability greater than 6.x implement a page faulting mechanism where data is migrated on demand when a page fault occurs [89], [85]. On page faulting, the Translation Lookaside Buffer (TLB) for the corresponding SM is locked and any new translations are suspended until all faults are handled by the driver. This ensures a consistent and coherent view of memory for each SM.

When profiling the generative sampling application on the TX1, as shown in figure 6.3, most if not all processing time is spent in synchronizing between CPU and GPU, which is typical on old Tegra SoC architectures.

On-demand page migration is an interesting feature that can help improve unified memory performance, albeit it is not supported on Maxwell architectures and thus won't be explored further here.

6.2.2 Pinned Memory and Data Transfers

On dGPU systems, page-locked or pinned memory is sometimes used to increase application performance by reducing data transfers and overlapping them with kernel executions. However on iGPU systems such as the TX1, using pinned memory can degrade performance with repetitive memory access patterns due to the non-caching behavior on the TX1 [88]. Table 6.2 shows the duration times for a kernel performing addition on a vector of one million



```
Testing completed successfully
==2193== Profiling application: ./RBM_PG_Generative
==2193== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 71.77%  35.1307s         1  35.1307s  35.1307s  35.1307s  vis_to_hid_kernel(float*, float*, flo
at*, float*, float*, float*)
 28.23%  13.8206s         1  13.8206s  13.8206s  13.8206s  hid_to_vis_kernel(float*, float*, flo
at*, float*, float*, float*)

==2193== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.63%  48.9949s         2  24.4975s  13.8584s  35.1365s  cudaDeviceSynchronize
  0.34%  165.85ms         1  165.85ms  165.85ms  165.85ms  cuDevicePrimaryCtxRetain
  0.02%  7.5844ms         6  1.2641ms  77.552us  3.0603ms  cudaMallocManaged
  0.01%  5.6557ms         2  2.8279ms  1.2628ms  4.3930ms  cudaLaunch
  0.01%  4.4774ms         6  746.24us  55.833us  1.8762ms  cudaFree
  0.00%  86.814us        91     954ns     416ns  20.990us  cuDeviceGetAttribute
  0.00%  48.594us         1  48.594us  48.594us  48.594us  cudaGetDeviceProperties
  0.00%  17.448us         1  17.448us  17.448us  17.448us  cudaSetDevice
  0.00%  12.500us         3  4.1660us  1.7190us  8.2290us  cuModuleGetGlobal
  0.00%  12.500us         2  6.2500us  3.1250us  9.3750us  cuModuleGetFunction
  0.00%  11.456us        12     954ns     573ns  1.7190us  cudaSetupArgument
  0.00%  9.9480us         2  4.9740us  1.8230us  8.1250us  cudaConfigureCall
  0.00%  9.7920us         2  4.8960us  4.7400us  5.0520us  cudaGetLastError
  0.00%  7.1890us         3  2.3960us     938ns  4.8960us  cuDeviceGetCount
  0.00%  4.6880us         1  4.6880us  4.6880us  4.6880us  cuDeviceTotalMem
  0.00%  2.4480us         1  2.4480us  2.4480us  2.4480us  cuCtxSetCurrent
  0.00%  2.1870us         3     729ns     521ns     885ns  cuDeviceGet
  0.00%  1.9280us         2     964ns     834ns  1.0940us  cuCtxGetCurrent
  0.00%  1.4580us         1  1.4580us  1.4580us  1.4580us  cuDeviceGetName
  0.00%     781ns         1     781ns     781ns     781ns  cuCtxGetDevice
```

Figure 6.3: Profiling example showing 99.63% of processing time spent in device synchronization on the Tegra X1 to ensure memory coherence

Table 6.2: Kernel Execution Times

GPU Device	Zero-copy Memory	Unified Memory
Tegra X1 (cc 5.3)	$17.846\,ms$	$17.856\,ms$
GTX 1080 (cc 6.1)	$700.424\,\mu s$	$3.064\,ms$

elements using both memory models independently. As expected, the zero-copy memory implementation on the TX1 was not as fast as the corresponding implementation on the GTX 1080.

6.2.3 Unified Memory Optimizations on the TX1

As stated in section 6.2.1, using unified memory on the TX1 requires cache coherence and synchronization steps. These steps are usually taken care of by the CUDA driver. However, we can assist the driver by providing some hints on where to access this data; this idea is called *Data Prefetching*, and can decrease latencies produced from synchronization operations of managed memory.

Even though the TX1 does not support concurrent managed memory access of the CPU and the GPU so that the *concurrentManagedAccess* flag is equal to 0 always when calling *cudaDeviceGetAttribute()*, concurrent access can be achieved through the usage of *streams* [88], [85]. CUDA Streams were introduced to allow concurrent execution of kernels. Since streams running in parallel are independent, managed memory allocations could be explicitly set for each stream. The state of the flag in the *cudaStreamAttachMemAsync()* API function causes data to be fetched to the CPU or the GPU, thus calling that function before launching a host operation or kernel with the correct flag set saves the Unified Memory system from having to migrate all data between host and device. Figure 6.4 demonstrates how to use such

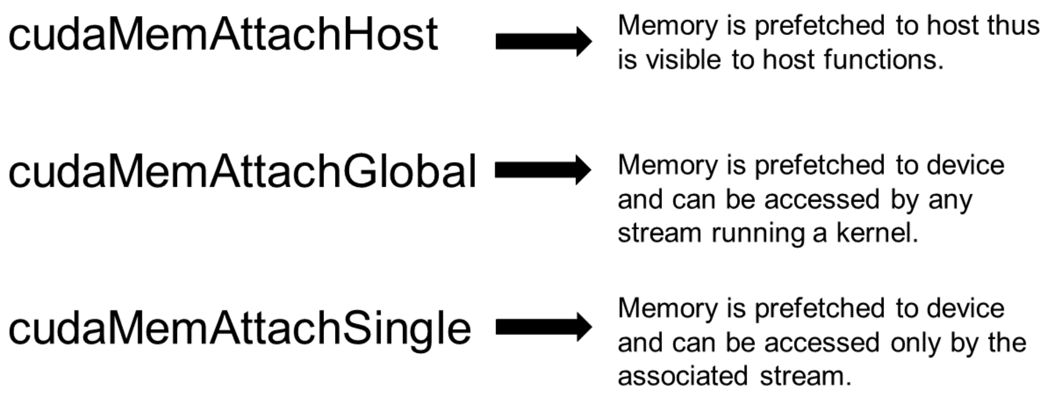

cudaMemAttachHost
Memory is prefetched to host thus is visible to host functions.
cudaMemAttachGlobal
Memory is prefetched to device and can be accessed by any stream running a kernel.
cudaMemAttachSingle
Memory is prefetched to device and can be accessed only by the associated stream.

Figure 6.4: CUDA API flags used to change behavior of managed memory that can be used for data prefetching

flags [88]. It is to be noted however that the default managed allocation, allocating memory using *cudaMallocManaged()* without attaching it to a specific stream, makes it visible to all kernels that are running.

6.3 Performance Analysis and Profiling

It is always essential in all hardware and software system designs to measure the performance of the system to assess whether the specific application could be implemented more efficiently given the system's resources. One of the most recommended approaches to optimization is the **APOD** approach [90] where optimization is a cycle of **A**ssessing the application's requirements and analyzing the performance of the current design; **P**arallelizing the solution with the goal of achieving those requirements; **O**ptimizing the parallel solution to improve performance; **D**eploying the optimized parallel implementation on real-world systems; and repeating the cycle until a satisfactory performance is achieved. This section will introduce performance strategies and metrics that are used in that cycle.

6.3.1 Parallel Execution

Perhaps one of the naive methods of achieving less kernel execution time is to divide the work across as many threads as possible. However, as seen in the next few sections, this is not always the best optimization strategy. In the SIMT model as discussed in section 6.1.1, threads are grouped into warps for global memory access and general instruction execution. This model works well if all threads in the warp are executing the same set of instructions, i.e. there is no branch divergence (see section 6.3.3). Problems arise when some threads in the warp are stalled since they are waiting on the rest of the warp threads to execute their path. This defeats the whole parallelism idea and is called *thread divergence.* Redundant use of barrier synchronization functions like *__syncthreads()* can also lead to wasted time since all threads in the same block have to finish execution before the next instruction is serviced. Thus the use of barrier synchronization and atomic functions should only be used for data and memory coherence.

6.3.2 Global Memory Bandwidth

Two approaches are usually taken to maximize memory bandwidth: hiding memory latency by increasing the number of warps executing concurrently along with coalescing and aligning memory accesses [84]. The measured bandwidth a kernel actually achieves is known as *Effective Bandwidth* and is calculated using:

$$Bandwidth\,(GB/s) = \frac{(Number\,of\,bytes\,read + Number\,of\,bytes\,written) * 10^{-9}}{Total\,time\,taken} \tag{6.1}$$

Global Memory Access Patterns

How a warp accesses the memory, whether it is reading or writing, can greatly affect a kernel's performance. When all threads in a warp read from or write to the same contiguous segment, the access is said to be *coalesced*; which is almost always desirable to reduce the number of memory transactions required to service the warp. Figure 6.5 shows six different memory access load/read patterns. *Load Efficiency or Bus Utilization* is calculated as number of bytes requested by a warp divided by the total number of loaded bytes. If each thread accesses a word size greater than 4-bytes (FP32 value), warp memory requests are split into independent 128-byte transactions [85].

Caching Behavior

Caching is always done in L2 for devices of cc 5.x. Only data that is read-only can be cached in the unified L1/texture cache. Cache line is 128 bytes, and L2-cached memory accesses are serviced with 32-byte transactions. On cache hit, requests are serviced at the throughput of L2, when L2 misses, global memory DRAM throughput is achieved.

It is therefore recommended to have block sizes be a multiple of the warp size i.e. 32 threads, so the memory accessed by warps is aligned to L2-cache lines.

The Effect of Block size on Memory Bandwidth

In section 6.1, it was explained how the CUDA model is well-suited to multidimensional data arrays by mapping the data onto grid blocks and the programmer has the freedom to choose appropriate grid and block sizes that will speed up the application. From the hardware point of view, those blocks are not multidimensional at all and are organized into groups that run on the SMs called warps. Warps are one dimensional, hence those multidimensional grids and blocks are flattened to run on the SMs. Although the order of thread execution is

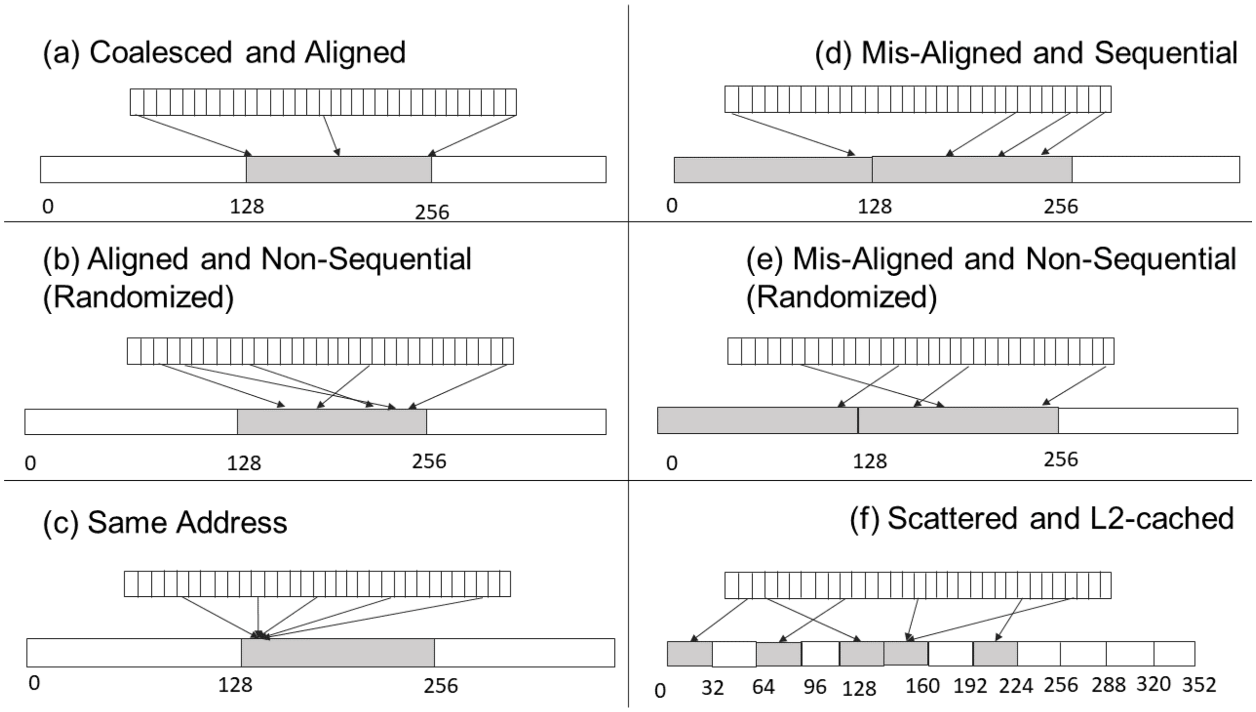

(a) Coalesced and Aligned
0
128
256
(d) Mis-Aligned and Sequential
0
128
256
(b) Aligned and Non-Sequential (Randomized)
0
128
256
(e) Mis-Aligned and Non-Sequential (Randomized)
0
128
256
(c) Same Address
0
128
256
(f) Scattered and L2-cached
0
32
64
96
128
160
192
224
256
288
320
352

Figure 6.5: Global Memory Access Patterns. (a-c) Aligned memory access: the first address of access is a multiple of 32. Bus utilization is 100% in (a,b) and 3.125% in (c) for 4-byte words (d,e) Misaligned memory access: access is spread across two 128-byte transactions. Bus utilization is 50% (f) Scattered memory access: Warp requests six 4-byte words scattered across global memory; over-fetching is avoided here since L2-cached accesses are always 32 bytes.

non-deterministic and controlled by the CUDA scheduler, each thread gets a unique id by the programmer based on how those blocks are flattened i.e. whether they are row-major order – like C arrays – or column-major order. As a result, a block size of (16, 16) will not execute in the same way as a block size of (8, 32), even though both block sizes expose the same amount of parallelism i.e. each block has 256 threads. A kernel of block size (8, 32) will have a higher effective memory bandwidth than the (16, 16) sized kernel [84]. That is due to the way global memory is accessed and how requests to DRAM are serviced, as described before.

6.3.3 Instruction Throughput

Whether the instruction is an arithmetic, load or a branch instruction, every executed instruction consumes processing time and bandwidth, thus redundancy in instruction usage should be avoided. In addition, not all arithmetic instructions are created equal: Single-Precision Floating Point (FP32) instructions have higher throughput than Double-Precision Floating Point (FP64) instructions, and in vision applications one can get away with using Half-Precision Floating Point (FP16) or even INT8 instructions without losing much accuracy [91]. Using intrinsics are also more efficient than using regular functions for standard arithmetic operations [90].

Instruction serialization or encoding is sometimes a metric to consider. Serialization percentage can be measured by comparing the number of instructions executed to the number of instructions issued. The NVIDIA profiling tool *nvprof* provides such metrics.

Another factor that can affect instruction throughput is thread granularity and thread reuse [92], [93]. Using fewer threads by making them do more work on the same dataset can often result in eliminating redundant memory load operations [92], and thread creation and destruction cost [93]. Thread reuse, however, consumes more resources such as registers and shared memory – a trade-off discussed in section 6.3.4, and might not be efficient for small datasets since not enough parallelism is used.

Warp and Branch Divergence is another important performance metric provided by *nvprof* that affects instruction throughput when the number of active threads in a warp is low [84]. The concept of branch divergence in parallel algorithm design is well-studied, and nonetheless applies to the **SIMT** execution model. Little warp divergence is indicated by a reported high branch efficiency percentange, and is a representation of good control flow in the design. Unrolling loops also helps avoid thread synchronization and loop control executions that can affect throughput as well [84].

6.3.4 Occupancy

Occupancy is a measure of hardware utilization; how much of the device's resources are being used to execute a specific kernel. The Best Practices Guide [90] defines it as the ratio of the number of active warps per SM to the maximum possible number of warps the device can run concurrently. While higher occupancy does not guarantee better performance, low occupancy is usually a sign of high memory latency that can degrade a kernel's performance.

Resource utilization is described in terms of block size, i.e. number of threads per block; number of thread blocks running on an SM; number of registers available for a thread block to use; and the amount of shared memory used by a thread block. All four factors are interdependent and increasing one can lead to resource under-utilization, which affects the occupancy ratio. Experimentation is needed here to find the best possible utilization. A plateau is usually reached at 50%, and increasing it further does not translate to better kernel performance. In [94], it was even shown that in some cases, it is possible to hide latencies using fewer warps and Instruction-Level Parallelism (ILP) techniques.

Chapter 7: Experiments

7.1 Image Representation and Reconstruction

Two sets of images were used in this experimental section. The first dataset was collected to test the experimental setup, while the second dataset was collected to model and reconstruct traffic scene intersections. Algorithm implementation and testing were done on both datasets, and are discussed in sections 7.1.1 and 7.1.2 respectively. An overview of the experiment done is shown in figure 7.1.

7.1.1 Dataset 1: Bike Path Images

System Setup

Figure 7.2 shows the experimental setup to collect image data using an electric golf cart as the moving vehicle. Images were captured using one StereoLabs ZED camera [95] mounted on the bottom right-side of the windshield connected using USB 3.0 to an NVIDIA Jetson TX1 development board [96]. Images were recorded using ZED SDK 1.0.

Data Collection

We drove the golf cart along the gravel bike path and captured images of walking, running and jogging pedestrians along with some on bicycles. A collection of seven sample videos in .svo format were recorded. Example images from a sample video are shown in figure 7.3.

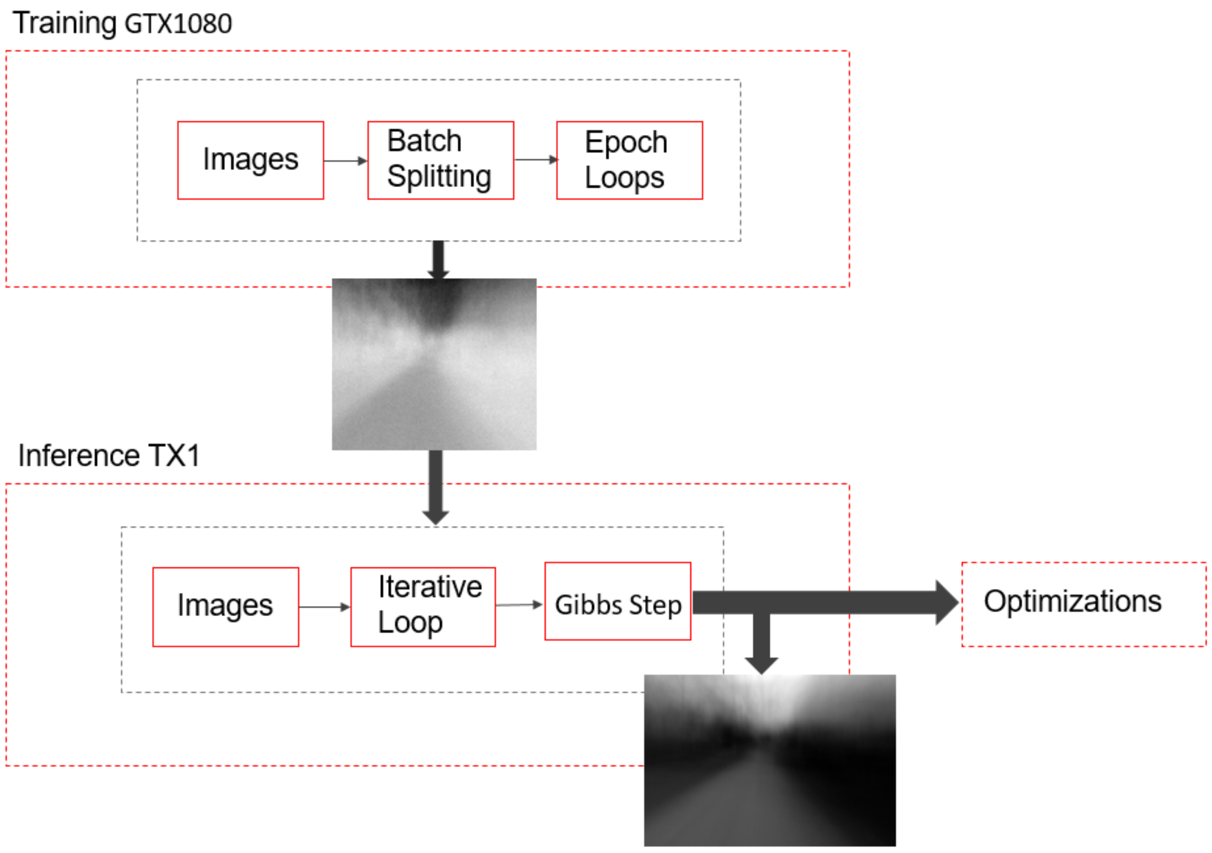


Figure 7.1: Experiment Overview

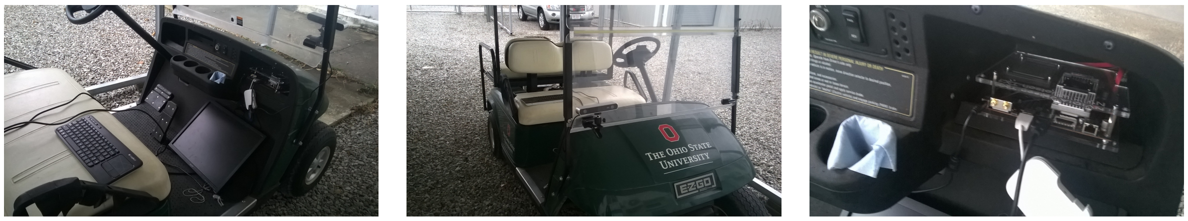


Figure 7.2: Experimental Setup used to collect real-world images along a gravel bike path using an electric golf cart

7.1.2 Dataset 2: Urban Traffic Intersections

System Setup

Images were captured using two Point Grey GigE Flea3 cameras [97] mounted on the car's dashboard angled 45° to the left and right, connected using Intel Network GigE PCIe Adapter 82576 to an NVIDIA Jetson TX1 development board [96]. Image acquisition was done synchronously using both cameras.

Data Collection

Images of real-life traffic intersections were collected while driving around campus. We focused on including images of the following five types of intersections in the dataset:

- Four-way stop sign controlled
- Traffic light signal controlled
- Cross-walks without stop signs
- Roundabout-type intersections
- Three-way intersections

Figure 7.3: Sample images recorded from an .svo video file using ZED SDK 1.0

The total number of images in the dataset were 5859 images from each camera. Sample images from this dataset can be seen in figure 7.4. We divided the dataset into independent training and test sets in the typically used ratio of 5:1.

7.1.3 Model Training

All image frames in an .svo video were used to train a Restricted Boltzmann Machine as an example of a probabilistic one-layer unsupervised network. Images were cropped from 720×1280 to 360×960 to remove redundant pixels e.g. sky regions. 2D image data used were normalized grayscale values stored as FP64 data type of 8 bytes. This was done to best match the RBM model that uses probability values for its input visible layer. One container for all training data was created indexed by the frame number as the container's row pointer. This speeded up the copying process since images were copied as in blocks of their rows instead of individual pixels.

Before training the network, a set of initial weights and biases had to be computed.Theoretically, any random weights and biases will do but since usually there are thousands of visible and hidden neurons, any hints on where to start the search in the weight space could lead to faster convergence in the Markov chain. The procedure used for initializing weights was similar to the training procedure: all training data was divided into batches, and each complete round constituted an epoch. We followed the procedure in [98]: initial weights were allowed to vary using the harmonic mean of the weight matrix dimensions; this was done using the fact that summing the dot product of neuron activations and a weight vector depends on the length of that vector. Weights were initialized using the following formula:

$$W_{ij}^{Initial} = \frac{4 * rand(0,1)}{\sqrt{\sqrt{N_V * N_H}}} * (rand(0,1) - 0.5) \tag{7.1}$$

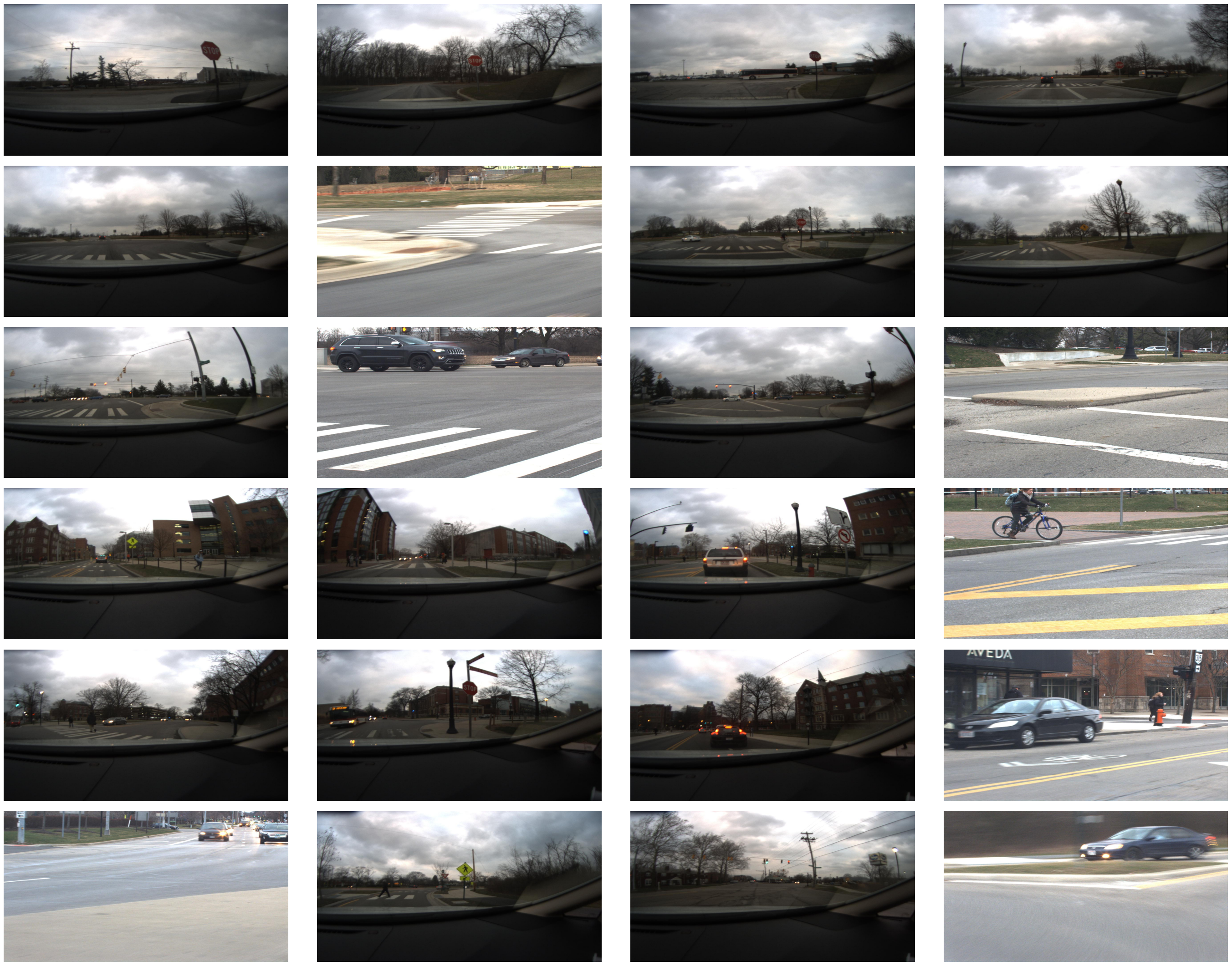


Figure 7.4: Sample images recorded from the collected dataset of traffic intersections

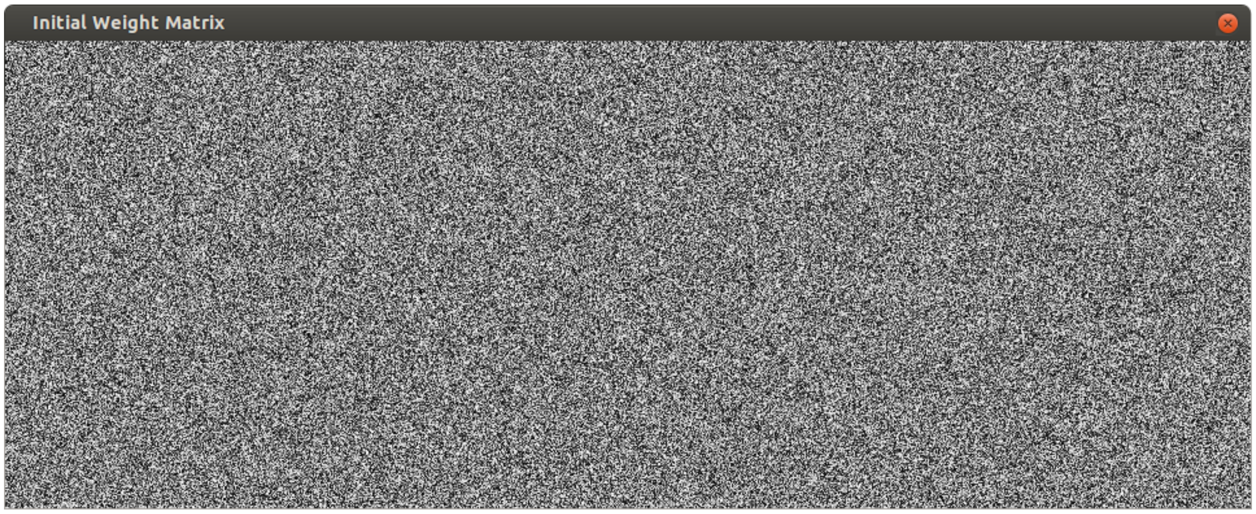


Figure 7.5: Initial weight matrix displayed as 360 x 960 image. This weight matrix is for the first hidden neuron in the RBM layer of 200 hidden neurons

. To compensate for a potential unbalance in the random weight set, we add an initial bias $a_j = -\sum_i \bar{x}_i W_{ji}$, where $\bar{x}_i$ is the mean of the input dataset, to each hidden neuron such that for the average training set, its net input is zero because the distribution is centered. To ensure a somewhat small reconstruction error, we set an initial bias b_i for the visible neuron activation that depends on the average activation of the hidden layer q using equation 7.2, and an initial bias for the hidden neuron activation as in equation 7.3.

$$b_i = log(\frac{\bar{x}_i}{1-\bar{x}_i}) - q\sum_j W_{ji} \tag{7.2}$$

$$c_j = log(\frac{q}{1-q}) - a_j \tag{7.3}$$

Figure 7.5 displays the weight matrix of the first hidden neuron as an image.

Alternatively, one could follow the recipe described in [99], and use small random values chosen from a distribution $N(0, 0.01)$, zero hidden biases, and bias for visible unit i as $log[p_i/(1-p_i)]$; where p_i is the proportion of training samples that turn unit i on.

Training was done in batches on an NVIDIA GTX-1080 [100] which is based on the Pascal GPU architecture. Since the card follows the typical heterogeneous memory architecture, data has to be copied into device memory space before processing. Training hyperparameters were declared in global device memory, while pointers to dynamic device memory were used for data and weight arrays. At the beginning of each training epoch, data was shuffled using a standard Fisher-Yates shuffle [101] and indices were copied to the device. There are usually some serial correlations in the data so shuffling helps to vary the contents of batches throughout training epochs.

A training epoch thus becomes a loop through all batches of the following:

1. Start batch (loop)
    - 1.1. Get visible unit values for the current batch from the data
    - 1.2. Compute the hidden probabilities without sampling from the visible layer values as in equation 2.5
    - 1.3. Start Markov chain (loop)
        - 1.3.1. Sample the computed hidden activation values
        - 1.3.2. Compute using equation 2.6 the visible layer values from the hidden activation values sampled in the previous step
        - 1.3.3. Compute the reconstruction error for the current chain by subtracting the difference of the computed visible layer values from the original provided in the dataset

1.3.4. Use the latest visible layer probability values to get the hidden layer probabilities according to equation 2.5

1.4. End Markov chain (loop)

1.5. Update Parameters and Error

1.5.1. Update the visible bias vector using the current momentum and learning rate values

1.5.2. Update the hidden bias vector using the current momentum and learning rate values

1.5.3. Update the weight matrix using the current momentum and learning rate values

1.5.4. Add current batch error to previous batch error for a single epoch error

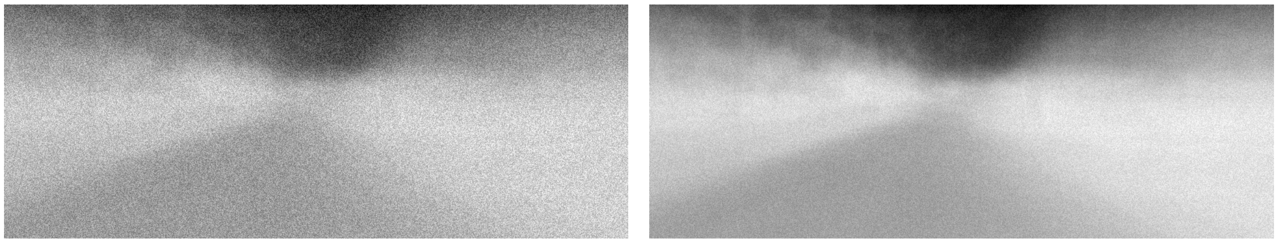
1.6. Compute weight gradients and gradient differences for adjusting the learning rate heuristically

2. Repeat for next batch (loop) until all batches in the epoch are done

3. Test for convergence by measuring the largest weight gradient computed during the current epoch relative to the largest weight magnitude

4. Adjust momentum and learning values independently for the next epoch – if needed to prevent large deviations when near convergence

Each training step was implemented on the device as a CUDA kernel. Columns of working vectors were mapped to grid blocks along x-dimension, while rows were mapped to the y-dimension. Each element in the working vector was assigned to a thread, launching the kernel with a number of threads per block that is a multiple of the warpsize, i.e. 32 threads,


(a) Weight matrix relating h_0, the first hidden neuron to the input. A total of 5 hidden neurons were used in this network

(b) Weight matrix relating h_0, the first hidden neuron to the input. A total of 200 hidden neurons were used in this network

Figure 7.6: Computed weight matrices computed after training using the bike path dataset

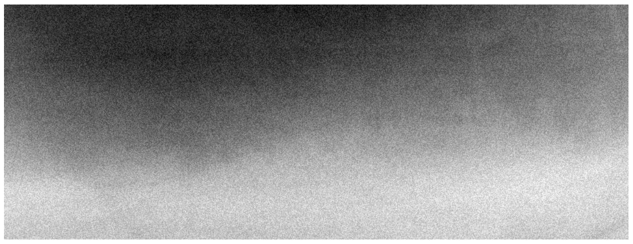
to ensure coalesced memory reads/writes and general concurrent instruction execution. Parallel reduction using shared memory was used to compute the weight gradients and weight gradient differences using partial sums, and the maximum weight magnitude using partial maximums.

A total number of 2110 images from the first dataset were used in training, while 4883 combined images from both cameras were used from the second dataset. Each dataset was trained separately. Final weights and biases were copied back to the host and displayed. Figures 7.6 and 7.7 shows sample weight matrices from trained networks on both datasets.

We used the recommended values for setting training hyperparameters from [98] and [99]. Table 7.1 shows their descriptions and values used.

7.1.4 Model Inference

To test the trained model on new image samples, we performed a round of Gibbs sampling. Test images were provided from another recorded .svo video from the first dataset, preprocessed in the same manner as the training images, and fed to the trained network in a random order, to assess the generative ability of the training by generating feature samples.


(a) Weight matrix relating h_0, the first hidden neuron to the input. A total of 20 hidden neurons were used in this network

(b) Weight matrix relating h_1, the second hidden neuron to the input. A total of 20 hidden neurons were used in this network

Figure 7.7: Computed weight matrices computed after training using the urban intersections dataset

Table 7.1: Hyperparameters Constants in training our experimental networks

Hyperparameter		
Name	Description	Value
Markov Chain Start	Beginning index of markov chain	1
Markov Chain End	Ending index of markov chain	4
Markov Chain Rate	Exponential Smoothing Rate of markov chain	0.5
Mean Field	Use meanfield activations = 0; use random sampling = 1	1
Greedy Mean Field	if == 0 ? use for training	1
Max Epochs	Maximum number of epochs for training	5
Max No Improvments	Converge if this many epochs without ratio improvements	5
Convergence Criterion	Convergence heuristic	0.5
Learning Rate	Stochastic Gradient Descent Rate	0.1
Momentum Start	Network's momentum start value	0.005
Momentum End	Network's momentum end value	1
Weight Penalty	Cost[a] assigned to weights	0.005
Sparsity Penalty	Sparsity[b] cost	0.1
Sparsity Target	Sparsity target value	0.1

[a]introduced parameter to discourage large weights associated with poor mixing rates

[b]average probability of a neuron being active

Inference was done on the Tegra X1 SoC [96], which is based on the older Maxwell GPU architecture. In contrast to how we used the system memory for training; no explicit data copies were done on the TX1. Network hyperparameters were defined as static variables in managed device memory space; working vectors and data were dynamically allocated on the TX1's unified memory. A one-step Gibbs sampling is the process of first sampling the hidden layer activations using the input visible neurons followed by sampling the visible layer from the computed hidden values. As stated in multiple previous work [43], [102], [103] we found the one-step contrastive divergence, i.e. one-step sampling, to be sufficient.

Gibbs sampling was done on the device using two CUDA kernels, one for each sampling phase. Thread mapping was done in a similar case to our training implementation. 2D image data were treated as 32-bit floats (FP32).

The same inference procedure was done for the second dataset except that images from both cameras were used, which enriched the data, but meant that a sampling step could contain two images captured simultaneously.

Samples from both datasets are shown in figure 7.8 respectively. No data copying between processing systems was needed here to display sample results, since both the CPU and GPU share the memory space. Total number of test images in both datasets were 817 and 976 images respectively.

Generative Capability: Images in figure 7.8 represent the hidden features found after training the model with 20 hidden units, which is equivalent to representing the data in a 20 hidden (latent) dimensions. Grayscale images (0 to 255) are mapped from the 0 to 1 probability space of the RBM. Visible neurons in the RBM represent discrete input pixel values of images while hidden neurons represent the lower dimensional feature space learned

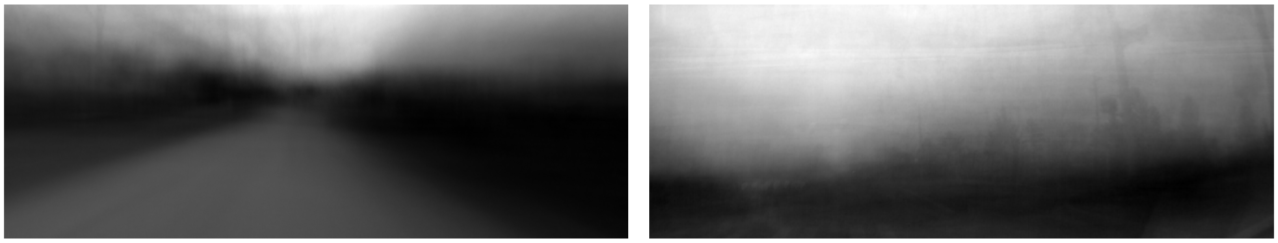


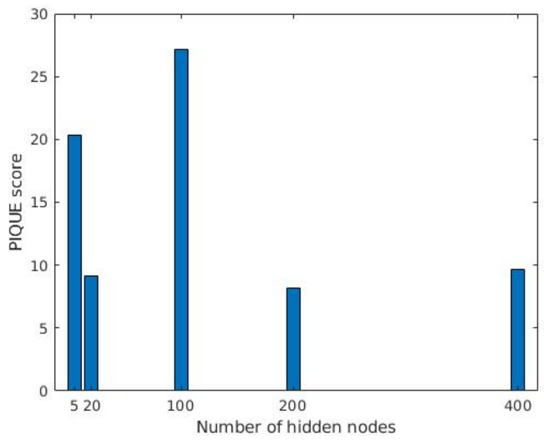
Figure 7.8: Generated Samples from the trained model in section 7.1.3

by the model. The goal of generative models is to find the representation space that best describes real-world image features, so the hypothesis is the more hidden neurons used in the implementation, the better image representation we get. It is to noted that the CUDA API function we used *cudaMallocManaged()* could only allocate up to half the size of the physical RAM (at the time of writing), which restricted the number of hidden neurons used.

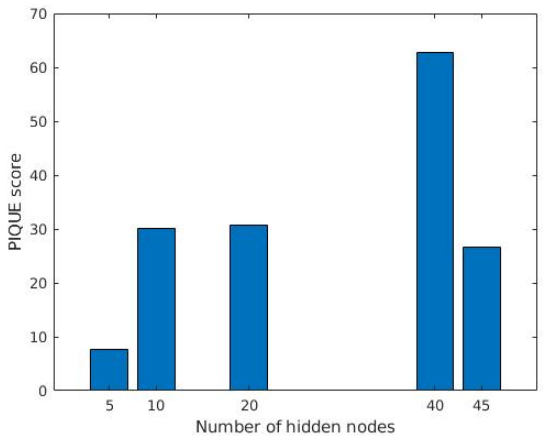
Image Perception Evaluation: To test the generative capability of the model, we used the Perception-based Image QUality Evaluator (PIQUE) metric defined in section 2.3 and [49]. PIQUE calculates the quality score of an image using a block-wise distortion map of local features that are extracted from perceptually significant spatial regions. We refer the reader to figure 2.6 and [49] for more details.

Our average perception scores were 28.3107% and 42.9174% for each dataset respectively. We measured the mean score for 20 different generative samples from each dataset independently.

Image Quality versus Network Size: To get an idea of how the image perception quality varies with the network size, we retrained both datasets with different numbers of



(a) Bike Path Dataset



(b) Intersection Dataset

Figure 7.9: Image quality (PIQUE) scores (%) computed for five different image samples generated using different number of hidden nodes for both datsets respectively

hidden units, reconstructed the images and measured their perception scores using the same quality evaluator. Figure 7.9 shows the results.

Figure 7.10 shows the images used in the evaluation process. The images generated from the first dataset show more variation between individual pixels when changing the hidden nodes representation than the second dataset, thus having a low distortion value or a low score on the evaluator scale. We hypothesize two possible explanations for this behavior. First, the range of hidden nodes used in the generator network is larger in the first dataset compared to the range used in the second dataset, thus enabling the network to represent more features. Second, the first datset originally had more correlated images, as a result modeling the images of the first dataset using the network was more biased and thus it is easier to capture their distribution.

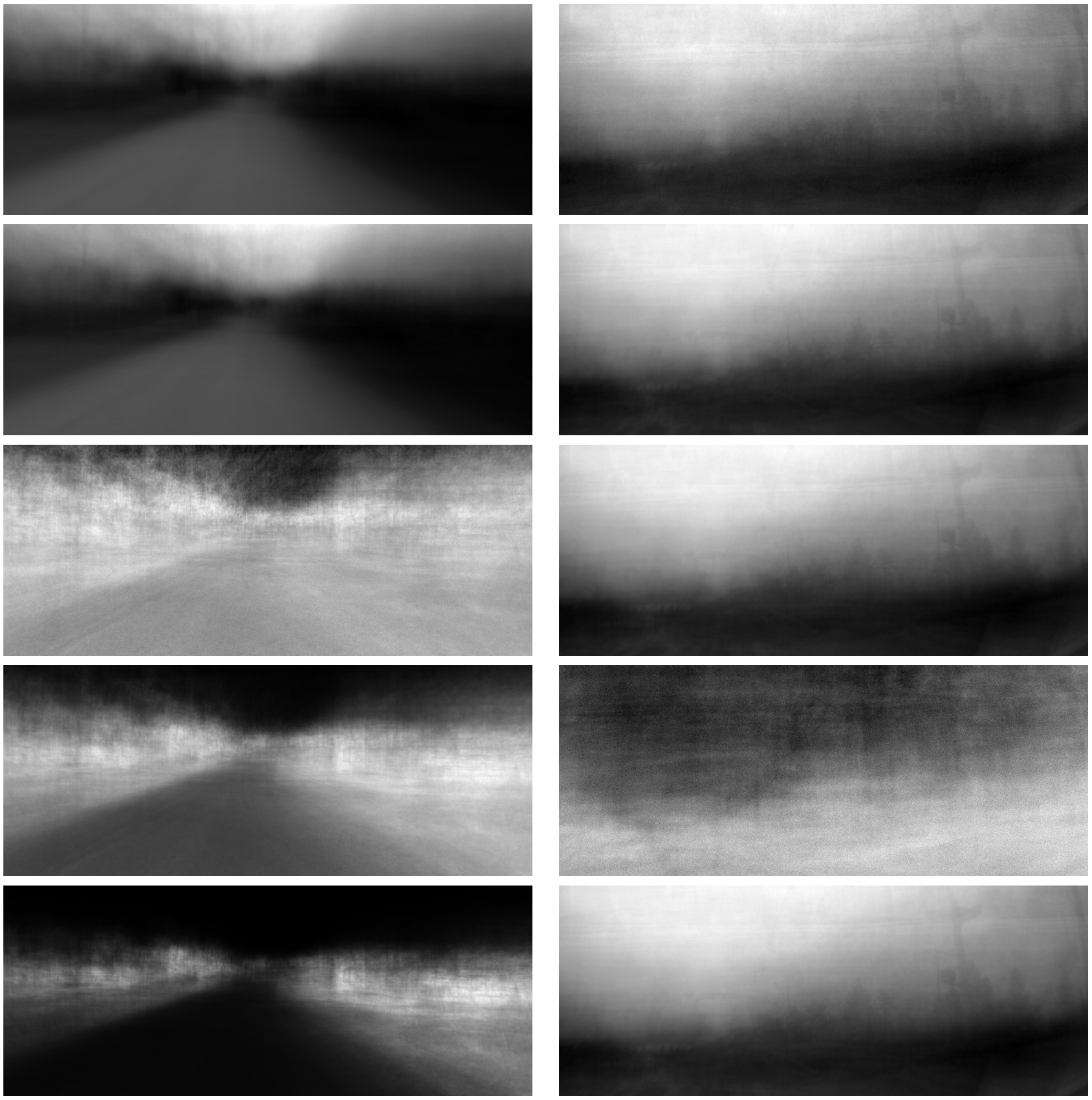
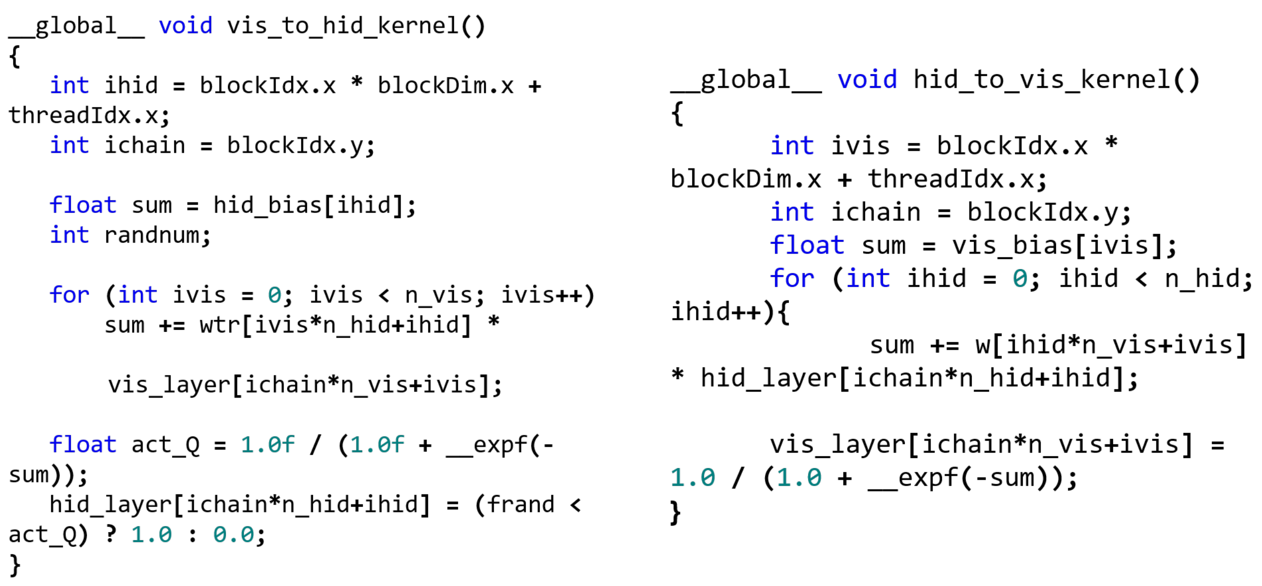


Figure 7.10: Images used for quality evaluation from figure 7.9. The images in the first column are from the Bike Path Dataset generated using 5, 20, 100, 200 and 400 hidden nodes respectively. The images in the second column are from the Intersection Dataset generated using 5, 10, 20, 40 and 45 hidden nodes respectively.



```
__global__ void vis_to_hid_kernel()
{
   int ihid = blockIdx.x * blockDim.x +
threadIdx.x;
   int ichain = blockIdx.y;

   float sum = hid_bias[ihid];
   int randnum;

   for (int ivis = 0; ivis < n_vis; ivis++)
       sum += wtr[ivis*n_hid+ihid] *

       vis_layer[ichain*n_vis+ivis];

   float act_Q = 1.0f / (1.0f + __expf(-
sum));
   hid_layer[ichain*n_hid+ihid] = (frand <
act_Q) ? 1.0 : 0.0;
}
```

```
__global__ void hid_to_vis_kernel()
{
        int ivis = blockIdx.x *
blockDim.x + threadIdx.x;
        int ichain = blockIdx.y;
        float sum = vis_bias[ivis];
        for (int ihid = 0; ihid < n_hid;
ihid++){
                sum += w[ihid*n_vis+ivis]
* hid_layer[ichain*n_hid+ihid];

        vis_layer[ichain*n_vis+ivis] =
1.0 / (1.0 + __expf(-sum));
}
```

Figure 7.11: Implementation of Gibbs sampling kernels

7.1.5 Profiling on the TX1

Figure 7.11 shows our implementation of Gibbs sampling as CUDA kernels on the TX1. We expect our *vis_to_hid* kernel to be more computationally expensive than the *hid_to_vis* kernel, since we there are more visible than hidden neurons. We measured *vis_to_hid* kernel runtimes with four different block sizes of 32, 64, 128 and 1024 threads, and plotted them in figure 7.12. Figure 7.13 shows a profiler plot of the number of warps issued per SM on the Maxwell GPU with different block sizes. Figures 7.14 and 7.15 show the utilization of each function unit, and the percentage of execution of each instruction type respectively. We follow with a discussion of the optimization strategies utilized in our implementation.

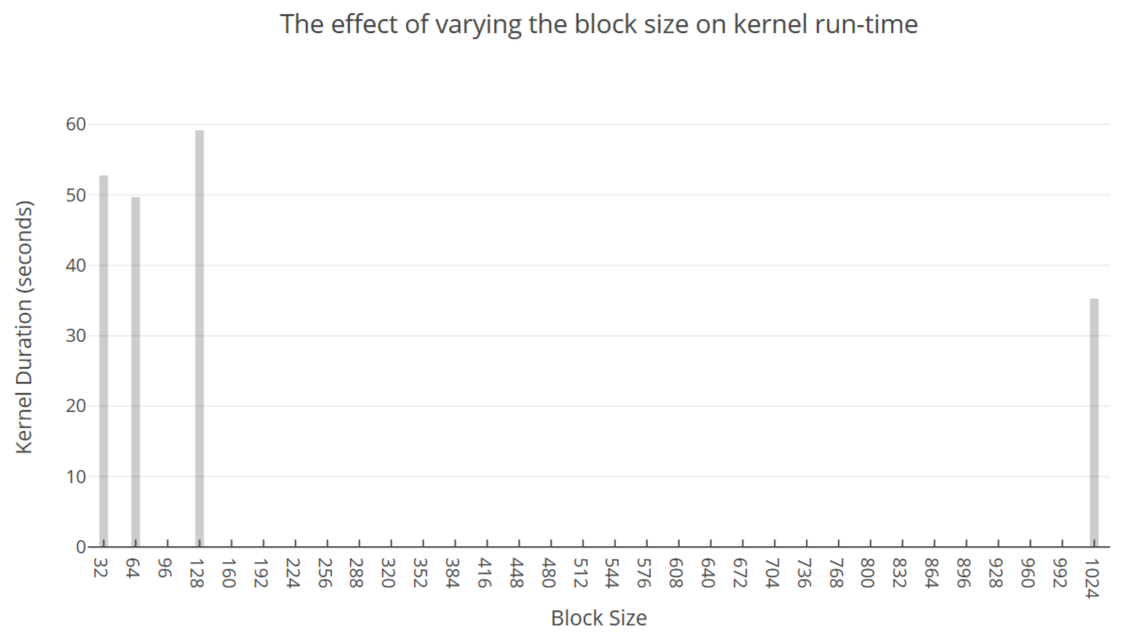


Figure 7.12: A plot of vis_to_hid kernel duration times ran with block sizes of 32, 64, 128 and 1024 threads

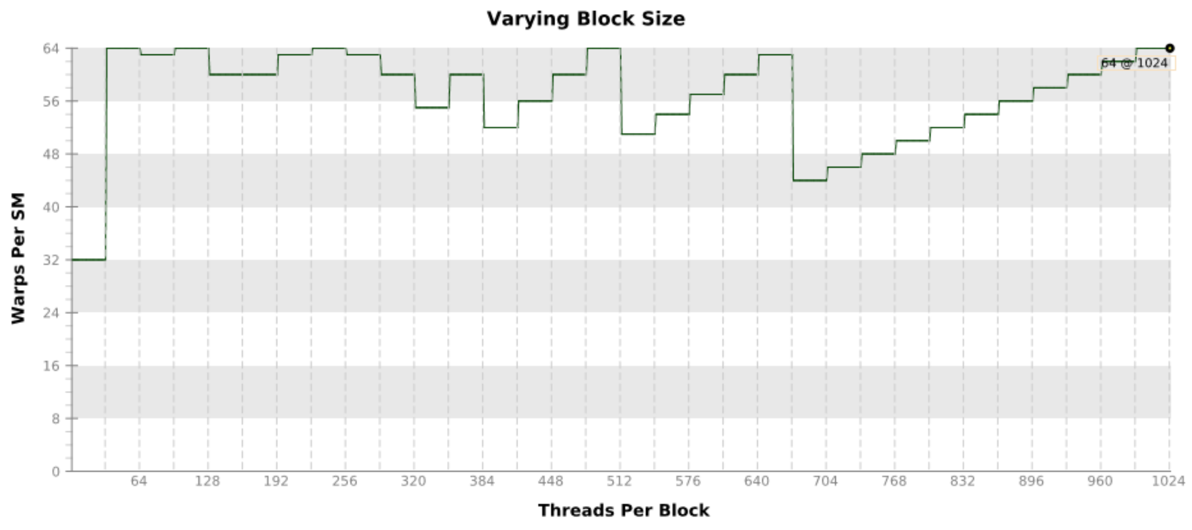


Figure 7.13: How the number of warps vary per Maxwell SM with different block sizes. There are 2 SMs on the Maxwell GPU architecture

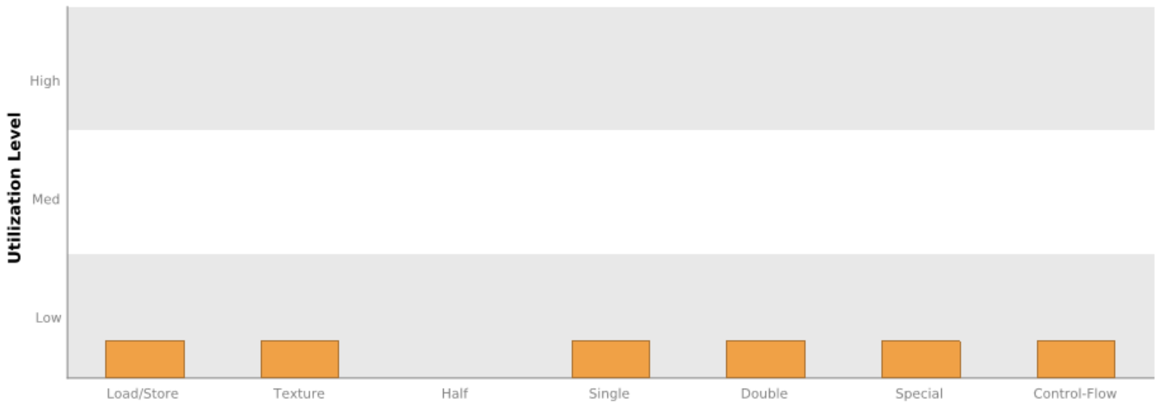


Figure 7.14: Function Unit utilization of the vis_to_hid kernel

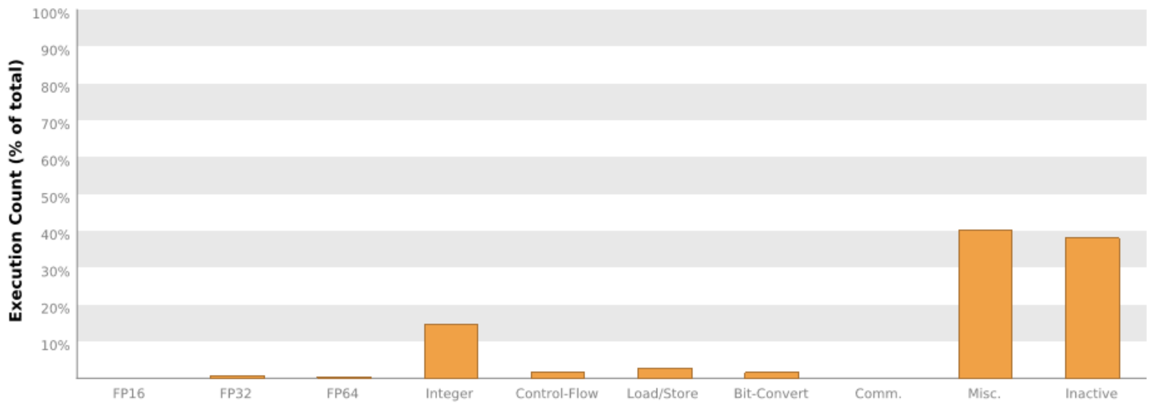


Figure 7.15: Types of instructions and their percentage of execution in the vis_to_hid kernel

Global Memory Bandwidth Calculations: We calculated the effective memory bandwidth used by the *vis_to_hid* kernel by calculating the number of bytes read and written by the kernel over its run time. A memory bandwidth of 2.35GB/s was achieved.

We utilized the unified memory model in our inference implementation. This avoided unnecessary data copies between host and device, in addition to utilizing the L2- cache, in contrast with zero-copy implementations where caching behavior is disabled [88]. Row-major order global memory reads and writes were coalesced, which ensured no unnecessary memory transactions were issued [85], [90], [84]. Achieved occupancy was 23.4% versus 31.2% for the 32 thread block kernel.

Active warps: Warp execution efficiency, the average percentage of active threads in the executed warps, for the *vis_to_hid* kernel was 62.5% due to the presence of intra-warp divergence in the if-branching statements.

7.2 Optimization Study

This research presents the costs and benefits of using different types of GPU memories presented in table 7.2 and data formats and memory access patterns previously discussed in chapter 6, to perform the same inference algorithm from section 7.1.4 on the dataset of images described in section 7.1.2. For all experiments in this section, we assume a fixed block size of 1024 threads.

7.2.1 Implementation

We implemented our algorithm using different optimization techniques used for compute-bound and memory-bound applications, and measured performance accordingly. We defined our baseline measurements as those resulting from an unoptimized version of the algorithm

Table 7.2: Summary of different GPU memory types used in this study

Global Memory	Local Memory	Constant Memory
Off-chip DRAM	Off-chip DRAM (Allocated in global memory)	Off-chip ROM
Uncached	Uncached	Cached per SM
High latency, low throughput	High latency	On a cache miss, the cost is one memory read
Host and device access	Individual thread access	Located in device memory and accessed through a special read-only cache
Kernel Persistent	Lifetime of thread	

implementation: all variables are FP32, defined in pageable global memory and are read and written in a column major order. Table 7.3 shows all experimental test cases run on 40 random images selected from our traffic intersection scene dataset. One step Gibbs sampling was done on all 40 images simultaneously and execution times were measured using CUDA Events.

Table 7.3: Test Cases

Case number	Weights	Image Data	Access Pattern	Format
Zero (Baseline)	Pageable	Pageable	Column	FP32
One	Pageable	Pageable	Row	FP32
Two	Constant	Pageable	Column	FP32
Three	Constant	Unified	Column	FP32
Four	Constant	Pinned	Row	FP32
Five	Constant	Unified	Row	FP32
Six	Constant	Unified	Row	FP16

Image data and trained parameters are to be read from ROM (SSD) into the standard OpenCV/C++ convention of row-based order. To test row-major ordering vs column-major ordering, a transposition operation was done on the image data and stored independently in global memory. We performed the transposition using the cuBLAS library GEMM function cublasSgeam(). Using this function with managed (unified) memory allocation and initialization was not successful unless host-device synchronization was called before data re-access on host.

7.2.2 Results and Discussion

Total execution times for each of the cases in table 7.3 were measured using cudaEvents and the resulting times are shown in table 7.4. Total execution time was defined as the sum of memory transfers and one iteration of a Gibbs sampling kernel run. One sampling iteration is defined as the total time taken by both kernels (visible-to-hidden and hidden-to-visible), kernel launching times by the host and host-device synchronizations before and after each kernel launch. The times in tables 7.4, 7.5, 7.6 and 7.7 were measured as the average of 5 runtimes for each test case.

The Tegra X1 chip has a 64-bit DRAM interface with memory clock DDR rate of 13MHz (by checking CUDA 8.0 SDK's *deviceQuery*), which translates to a theoretical bandwidth of 0.208 GB/s. Actual memory transactions can be different from load/store throughput, which is why we decided to calculate the global memory efficiency as stated in section 6.3.2. Tables 7.9 and 7.10 show the results of global memory efficiency obtained for each case.

From table 7.4, we can see that the baseline case took the longest, which agrees with our hypothesis that this is the least optimal case, followed by the zero-copy (case 4). Case 1 (pageable) took the least amount of time, less than cases 4 (unified) and 5 (zero-copy) which is surprising on the TX1 given memory duplication. We posit the question whether there is significant overhead to use those memory models on the TX1 such that the unified memory choice is not as significant an optimization technique as hypothesized.

Data layout and memory access patterns proved an important optimization technique. Even though cases 3 and 5 share memory and data types, row-major data ordering is faster. This agrees with NVIDIA's SIMT model of warp access.

Using constant memory to store network coefficients and parameters proved faster than traditional pageable global memory. Case 2 performed much faster than the baseline case.

As shown in table 7.8, the highest throughput was obtained in case 4 (pinned memory) for both kernels, while the lowest was case 2 for the visible-to-hidden kernel and case 5 for the hidden-to-visible kernel. Referring to sections 7.1.4 and 7.1.5, the network has more visible than hidden neurons and with mapping the same number of CUDA threads to our resulting neurons, the hidden-to-visible kernel (5746ms) executed faster (on-average) than the visible-to-hidden kernel (37834.7ms), which was expected as shown in tables 7.5 and 7.6.

The most optimization benefit was gained from case 6, by using half-precision floats instead of full-precision numbers. Using half-precision operations allowed us to double the throughput by vectorizing floating point instruction operations.

Table 7.4: Total Execution Times (ms)

Case number	Time (milliseconds)
Zero (Baseline)	50929.7
One	38633.2
Two	44499.8
Three	42582.1
Four	46352.5
Five	39072.4
Six	28336.5

Table 7.5: Visible to Hidden Execution Times (ms)

Case number	Time (milliseconds)
Zero (Baseline)	45146.6
One	32684.6
Two	38737.2
Three	36672.5
Four	40408.2
Five	33359.6
Six	23007.3

Table 7.6: Hidden to Visible Execution Times (ms)

Case number	Time (milliseconds)
Zero (Baseline)	5637.67
One	5819.69
Two	5644.01
Three	5841.66
Four	5881.62
Five	5651.93
Six	5185.46

Table 7.7: Gibbs Sampling Execution Times (ms)

Case number	Time (milliseconds)
Zero (Baseline)	50787.7
One	38509.1
Two	44383.6
Three	42516.7
Four	46294.7
Five	39014.5
Six	23347.4

Table 7.8: Global Memory Throughput

	Throughput (MB/s)	
Case number	vis to hid	hid to vis
Zero (Baseline)	82.6	121.21
One	89	124
Two	74.1	117.72
Three	82.6	114.12
Four	95.23	125.65
Five	92.21	107.99
Six[a]	95.236	125.653
Six[b]	99.43	486.66

[a]half number of threads

[b]same number of threads

Table 7.9: Global Memory Load Efficiency

	Global Load Efficiency (%)	
Case number	vis to hid	hid to vis
Zero (Baseline)	36.1	56.5
One	36.1	56.5
Two	36.1	56.5
Three	36.1	56.5
Four	36.1	56.5
Five	62.5	36.1
Six[a]	36.1	56.5
Six[b]	36.1	56.5

[a]half number of threads

[b]same number of threads

Table 7.10: Global Memory Store Efficiency

	Global Store Efficiency (%)	
Case number	vis to hid	hid to vis
Zero (Baseline)	83.3	100
One	83.3	100
Two	83.3	100
Three	83.3	100
Four	83.3	100
Five	12.5	83.3
Six[a]	83.3	100
Six[b]	83.3	100

[a]half number of threads

[b]same number of threads

Table 7.11: Optimization Benefit for each case in terms of time

Case number	Optimization Benefit (%)
One	24.14
Two	12.62
Three	16.39
Four	8.9
Five	23.28
Six	44.36

Chapter 8: Conclusion

This research addressed the problem of computational problems associated with real-time scene understanding and environment perception in the context of traffic scenes for different Advanced Driving Assistance systems (ADAS) and automated driving applications. It concerned finding suitable representations for image data by using probabilistic generative methods to model the hidden or latent variables in the data. The claim here was that if we could find the optimal space representation, classification (e.g. labeling vehicles, pedestrians and other objects in the traffic scene), regression tasks and inference become easier and more accurate. Traffic image data from camera sensors have proven to be complex and thus require multiple stages for best feature extraction. This, in addition to the amount of available sensory data, become bottlenecks for the real-time processing requirement of traffic applications.

We introduced a probabilistic framework for traffic scene understanding comprised of unsupervised representation of sensory data using deep neural networks and reconstruction and interpretation of traffic scenes using probabilistic inference methods.

The availability of parallel computing architectures like GPUs has helped improve the work complexity of these feature learning algorithms. Multi-layer supervised feed-forward convolutional neural networks have proved successful in various object detection tasks using traffic image data, however not much has been done to investigate the performance of unsupervised probabilistic networks in the same context. The goal of this research was to study these approaches, their behavior and real-time performance on the NVIDIA Tegra X1 SoC which is computational processor in the DrivePX automotive platform designed for ADAS and automated driving applications.

To summarize, the contributions of this work can be listed as follows:

1. Collection of real-world images of traffic intersections and a dataset that could always be expanded in the future.

2. Implementing and training a probabilistic unsupervised generative network on GPUs, utilizing and studying its architecture for best computational processing power.

3. Implementing an inference model on embedded SoCs and studying performance optimization strategies.

4. Analyzing of different GPU optimization techniques for real-time inference applications.

Although runtimes of our implementation are not real-time ready, understanding the effects of the different techniques can help towards achieving real-time performance.

Bibliography

[1] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[2] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, 1999, pp. 1150–1157 vol.2.

[3] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-Up Robust Features (SURF)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, Jun. 2008.

[4] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary Robust Independent Elementary Features," in *Proceedings of the 11th European Conference on Computer Vision: Part IV*, ser. ECCV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 778–792.

[5] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, June 2005, pp. 886–893 vol. 1.

[6] A. González, G. Villalonga, G. Ros, D. Vázquez, and A. M. López, "3D-guided multiscale sliding window for pedestrian detection," in *Pattern Recognition and Image Analysis.* Springer International Publishing, 2015, pp. 560–568.

[7] M. Heikkila and M. Pietikainen, "A texture-based method for modeling the background and detecting moving objects," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, p. 657662, 2006.

[8] C. Papageorgiou and T. Poggio, "A trainable system for object detection," *International Journal of Computer Vision*, vol. 38, no. 1, pp. 15–33, 2000.

[9] Z. Sun, G. Bebis, and R. Miller, "On-Road vehicle detection using Gabor filters and Support Vector Machines," in *International Conference on Digital Signal Processing*, 2002, pp. 200–2.

[10] A. González, G. Villalonga, J. Xu, D. Vázquez, J. Amores, and A. M. López, "Multiview random forest of local experts combining RGB and LiDAR data for pedestrian detection," in *2015 IEEE Intelligent Vehicles Symposium (IV)*, June 2015, pp. 356–361.

[11] L. W. Tsai, J. W. Hsieh, and K. C. Fan, "Vehicle detection using normalized color and edge map," *IEEE Transactions on Image Processing*, vol. 16, no. 3, pp. 850–864, March 2007.

[12] M. Heikkila and M. Pietikainen, "A texture-based method for modeling the background and detecting moving objects," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 657–662, April 2006.

[13] J. M. Alvarez, T. Gevers, and A. M. López, "3D scene priors for road detection," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 2010, pp. 57–64.

[14] A. Geiger, M. Lauer, C. Wojek, C. Stiller, and R. Urtasun, "3D traffic scene understanding from movable platforms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 5, pp. 1012–1025, May 2014.

[15] M. Cordts, T. Rehfeld, L. Schneider, D. Pfeiffer, M. Enzweiler, S. Roth, M. Pollefeys, and U. Franke, "The stixel world: A medium-level representation of traffic scenes," *Image and Vision Computing*, vol. 68, pp. 40 – 52, 2017.

[16] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deep driving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 2722–2730.

[17] Y. Jia and et al., "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference of Multimedia*, 2014, pp. 675–678.

[18] K. Ronan, Clément and Soumith. [Online]. Available: https://github.com/torch/torch7 (6/26/2018)

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *ArXiv e-prints*, Mar. 2016.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283.

[21] [Online]. Available: https://pytorch.org/docs/stable/index.html (6/26/2018)

[22] [Online]. Available: https://github.com/pytorch (6/26/2018)

[23] [Online]. Available: http://mxnet.ai/ (6/26/2018)

[24] B. Ranft and C. Stiller, "The role of machine vision for intelligent vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 8–19, March 2016.

[25] N. Z. Haron and S. Hamdioui, "Why is CMOS scaling coming to an END?" in *2008 3rd International Design and Test Workshop*, Dec 2008, pp. 98–103.

[26] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sept 1972.

[27] R. Benenson, M. Mathias, R. Timofte, and L. V. Gool, "Pedestrian detection at 100 frames per second," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, June 2012, pp. 2903–2910.

[28] L. Zhang and R. Nevatia, "Efficient scan-window based object detection using GPGPU," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1–7.

[29] V. Campmany, S. Silva, A. Espinosa, J. Moure, D. Vázquez, and A. López, "GPU-based pedestrian detection for autonomous driving," *Procedia Computer Science*, vol. 80, pp. 2377 – 2381, 2016.

[30] F. Homm, N. Kaempchen, J. Ota, and D. Burschka, "Efficient occupancy grid computation on the GPU with LiDAR and Radar for road boundary detection," in *2010 IEEE Intelligent Vehicles Symposium*, June 2010, pp. 1006–1013.

[31] P. Muyan-Ozcelik, V. Glavtchev, J. M. Ota, and J. D. Owens, "Chapter 32: Real-time speed-limit-sign recognition on an embedded system using a GPU," in *GPU Computing Gems Emerald Edition.* Boston: Morgan Kaufmann, 2011, pp. 497 – 515.

[32] M. Luca and et al., "GPU implementation of a road sign detector based on Particle Swarm Optimization," *Evolutionary Intelligence*, vol. 3, no. 3, pp. 155–169, December 2010.

[33] S. Bauer, S. Khler, K. Doll, and U. Brunsmann, "FPGA-GPU architecture for kernel SVM pedestrian detection," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, June 2010, pp. 61–68.

[34] C. Wojek, G. Dorkó, A. Schulz, and B. Schiele, "Sliding-windows for rapid object class localization: A parallel technique," in *Proceedings of the 30th DAGM Symposium on Pattern Recognition.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 71–81.

[35] V. A. Prisacariu and I. Reid, *fastHOG - a real-time GPU implementation of HOG*, 2009.

[36] *Tegra X1 Technical Reference Manual*, Nov 2015.

[37] D. Hernandez-Juarez, A. Chacón, A. Espinosa, D. Vázquez, J. Moure, and A. López, "Embedded real-time stereo estimation via semi-global matching on the gpu," *Procedia Computer Science*, vol. 80, pp. 143 – 153, 2016, international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[38] M. Bojarski and et al., "End-to-end learning for self-driving cars," NVIDIA Corporation, Tech. Rep., 2016.

[39] D. E. Rumelhart and J. L. McClelland, Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations.* Cambridge, MA, USA: MIT Press, 1986.

[40] Zorzi and et al., "Modeling language and cognition with deep unsupervised learning: a tutorial overview," *Frontiers in Psychology*, vol. 4, p. 515, 2013.

[41] H. Ackley, E. Hinton, and J. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive Science*, pp. 147–169, 1985.

[42] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[43] R. R. Salakhutdinov, "Learning deep generative models," *Annual Review of Statistics and its Application*, vol. 2, pp. 361–385, April 2015.

[44] V. Nair and G. Hinton, "Implicit mixtures of Restricted Boltzmann Machines," in *Proceedings of the 21st International Conference on Neural Information Processing Systems*, ser. NIPS'08. USA: Curran Associates Inc., 2008, pp. 1145–1152.

[45] G. E. Hinton and S. Osindero, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, p. 2006, 2006.

[46] J. L. W. V. Jensen, "Sur les fonctions convexes et les inégalités entre les valeurs moyennes," *Acta Math.*, vol. 30, pp. 175–193, 1906.

[47] R. Salakhutdinov and G. Hinton, "Deep boltzmann machines," in *Proceedings of the 12th International Conference of Artificial Intelligence and Statistics*, 2009, pp. 448–455.

[48] L. Theis, M. Bethge, and A. V. D. Oord, "A note on the evaluation of generative models," *Proceedings of the Fourth International Conference on Learning Representations (ICLR)*, 2016.

[49] N. Venkatanath, D. Praneeth, B. M. Chandrasekhar, S. S. Channappayya, and S. S. Medasani, "Blind image quality evaluation using perception based features," *2015 Twenty First National Conference on Communications (NCC)*, 2015.

[50] "6.438 Algorithms for Inference. fall 2014," Massachusetts Institute of Technology: MIT OpenCourseWare, Available at http://ocw.mit.edu under Creative Commons BY-NC-SA (Accessed September 2, 2017).

[51] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *Computing Research Repository*, vol. abs/1312.6114, 2013.

[52] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680.

[53] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real NVP," *CoRR*, vol. abs/1605.08803, 2016.

[54] L. Dinh, D. Krueger, and Y. Bengio, "NICE: Non-linear Independent Components Estimation," 10 2014.

[55] R. P. N. Rao, B. A. Olshausen, and M. S. Lewicki, *Probabilistic models of the brain: perception and neural function.* MIT Press, 2002.

[56] D. Kersten, A. J. Otoole, M. E. Sereno, D. C. Knill, and J. A. Anderson, "Associative learning of scene parameters from images," *Applied Optics*, vol. 26, no. 23, pp. 4999–5006, Jan 1987.

[57] D. Kersten, *Transparency and the cooperative computation of scene attributes.* MIT Press, 1991.

[58] M. I. Jordan, *Learning in graphical models.* MIT Press, 2002.

[59] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *Journal of Machine Learning Research*, vol. 14, 06 2012.

[60] D. P. Kingma, D. J. Rezende, S. Mohamed, and M. Welling, "Semi-supervised learning with deep generative models," *Computing Research Repository*, vol. abs/1406.5298, 2014.

[61] W. Freeman and E. Pasztor, "Learning low-level vision," *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 1999.

[62] U. Schmidt, Q. Gao, and S. Roth, "A generative perspective on mrfs in low-level vision," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 2010, pp. 1751–1758.

[63] K. Branislav, S. S. Bhattacharya, and S. Chai, *Embedded computer vision.* Springer, 2010.

[64] S. K. Tewksbury, *Application Specific Integrated Circuits (ASICs)*, 1996.

[65] "Digital video processors products." [Online]. Available: http://www.ti.com/processors/dsp/media-processors/digital-video/products.html

[66] [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture (7/12/2018)

[67] [Online]. Available: www.nvidia.com/v100 (7/12/2018)

[68] [Online]. Available: https://www.mobileye.com/our-technology/evolution-eyeq-chip/ (7/12/2018)

[69] [Online]. Available: https://ark.intel.com/products/series/132784/Intel-Xeon-Phi-72x5-Processor-Family (7/12/2018)

[70] [Online]. Available: https://thinci.com/about_us.html (7/12/2018)

[71] [Online]. Available: http://www.adapteva.com/epiphanyiv/ (7/12/2018)

[72] [Online]. Available: https://wavecomp.ai/technology (7/12/2018)

[73] [Online]. Available: https://www.movidius.com/vision-processing-units (7/12/2018)

[74] [Online]. Available: http://www.nvidia.com/object/tegra.html (7/12/2018)

[75] [Online]. Available: https://www.amd.com/en/graphics/servers-radeon-instinct-mi (7/12/2018)

[76] Y. H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan 2016, pp. 262–263.

[77] *XA Spartan-6 Automotive FPGA*, Xilinx, 12 2012, v1.3.

[78] *DesignWare EV52 and EV54 Processors*, Synopsis, 2015.

[79] J. Campbell and V. Kazantsev, *Using an Embedded Vision Processor to Build an Efficient Object Recognition System*, May 2015.

[80] [Online]. Available: https://ai.intel.com/intel-nervana-neural-network-processor/ (7/12/2018)

[81] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, vol. 113, no. 41, pp. 11 441–11 446, 2016.

[82] [Online]. Available: https://newsroom.intel.com/editorials/intels-new-self-learning-chip-promises-accelerate-artificial-intelligence/ (7/12/2018)

[83] G. Barlas, "Chapter 2 - multicore and parallel program design," in *Multicore and GPU Programming*, G. Barlas, Ed. Boston: Morgan Kaufmann, 2015, pp. 27 – 54.

[84] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Wrox, 2014.

[85] "CUDA C programming guide," Available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2018/04/16), NVIDIA Corporation.

[86] P. J. Denning, "The locality principle," *Communication Networks and Computer Systems*, p. 4367, 2006.

[87] M. Harris, "Unified memory in CUDA 6," Available at https://devblogs.nvidia.com/unified-memory-in-cuda-6/ (2018/04/27).

[88] "CUDA for Tegra," NVIDIA Corporation, Tech. Rep., Feb 2018.

[89] N. Sakharnykh, "Maximizing unified memory performance in CUDA," Available at https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/ (2018/04/27).

[90] "CUDA C best practices guide," Available at http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (2018/04/16), NVIDIA Corporation.

[91] M. Harris, "Mixed-precision programming with CUDA 8," Available at https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/ (2018/05/18).

[92] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands On Approach*, 2nd ed. San Francisco, CA: Elsevier Science and Technology, 2012.

[93] M. Harris, "CUDA Pro Tip: Write flexible kernels with grid-stride loops," Available at https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (2018/05/18).

[94] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2016.

[95] "Stereolabs ZED camera." [Online]. Available: {https://www.stereolabs.com/zed} (6/7/2018)

[96] "NVIDIA Jetson TX1 development platform." [Online]. Available: {https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules} (6/7/2018)

[97] "Point Grey FLEA3 cameras." [Online]. Available: https://www.ptgrey.com/flea3-13-mp-color-gige-vision-cs-mount-sony-icx445-camera (6/24/2018)

[98] T. Masters, *Deep Belief Nets in C++ and CUDA C*, 1st ed. CreateSpace Independent Publishing Platform, 2015.

[99] G. Hinton, "A practical guide to training restricted boltzmann machines," University of Toronto, Tech. Rep., 2010.

[100] "NVIDIA GTX1080 graphics cards series." [Online]. Available: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/ (6/19/2018)

[101] R. Durstenfeld, "Algorithm 235: Random permutation," *Commun. ACM*, vol. 7, no. 7, pp. 420–, July 1964.

[102] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, ser. NIPS'06. Cambridge, MA, USA: MIT Press, 2006, pp. 153–160.

[103] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, pp. 1–55, 01 2009.

[104] "NVIDIA Tegra X1: NVIDIA's new mobile superchip," NVIDIA Corporation, Tech. Rep., Jan 2015.

Appendix A: Data Collection and Processing Setup

A.1 The Development Environment

Two machines were used for development, each with its own compatible binaries and libraries for developing and testing the algorithms in question.

The Jetson System was flashed with Linux For Tegra (L4T) release 24.2.1 which runs a sample file system derived from Ubuntu v16.04 LTS. Flashing was done through an Ubuntu Linux x86_64 v14.04 desktop system.

The Host System was a hexcore Intel Core i7-6700K CPU which needed aarch64 compilation libraries for cross-compilation on the Jetson.

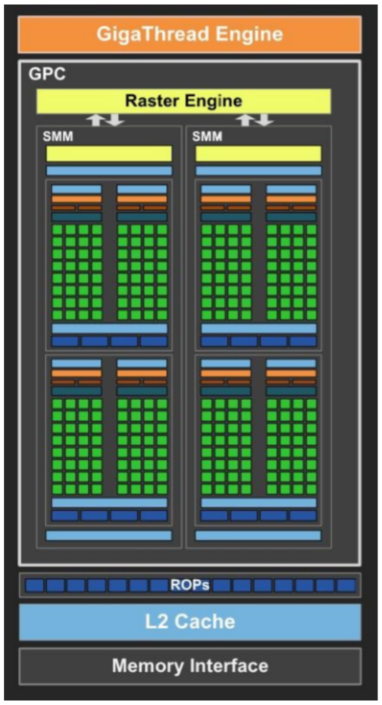
CUDA 8.0 was used in the experiments on both machines.

The image processing part was done on the Jetson due to SDK compatibility issues with the ZED camera, CUDA 8.0 version and the Ubuntu 14.04 host machine.

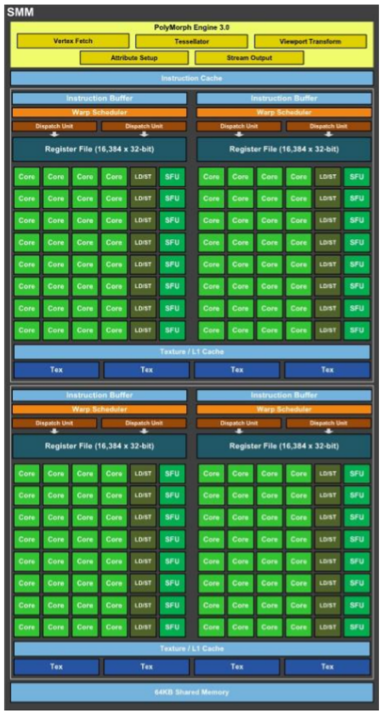
A.1.1 NVIDIA TX1 Quick Overview

The NVIDIA Tegra X1 SoC [104] features four 64-bit ARM Cortex A57 CPU core architecture; four 32-bit ARM Cortex A53 CPU cores; and a Maxwell GPU architecture with 256 cores built on 20nm TSMC.

The four high performance A57 CPU cores share a 2MB L2 cache, with a 48KB L1 instruction cache and a 32KB data cache each. The more power efficient A53 cores share a 512KB L2 cache with a 32KB instruction and data caches.



(a) Maxwell Tegra X1 Architecture Overview

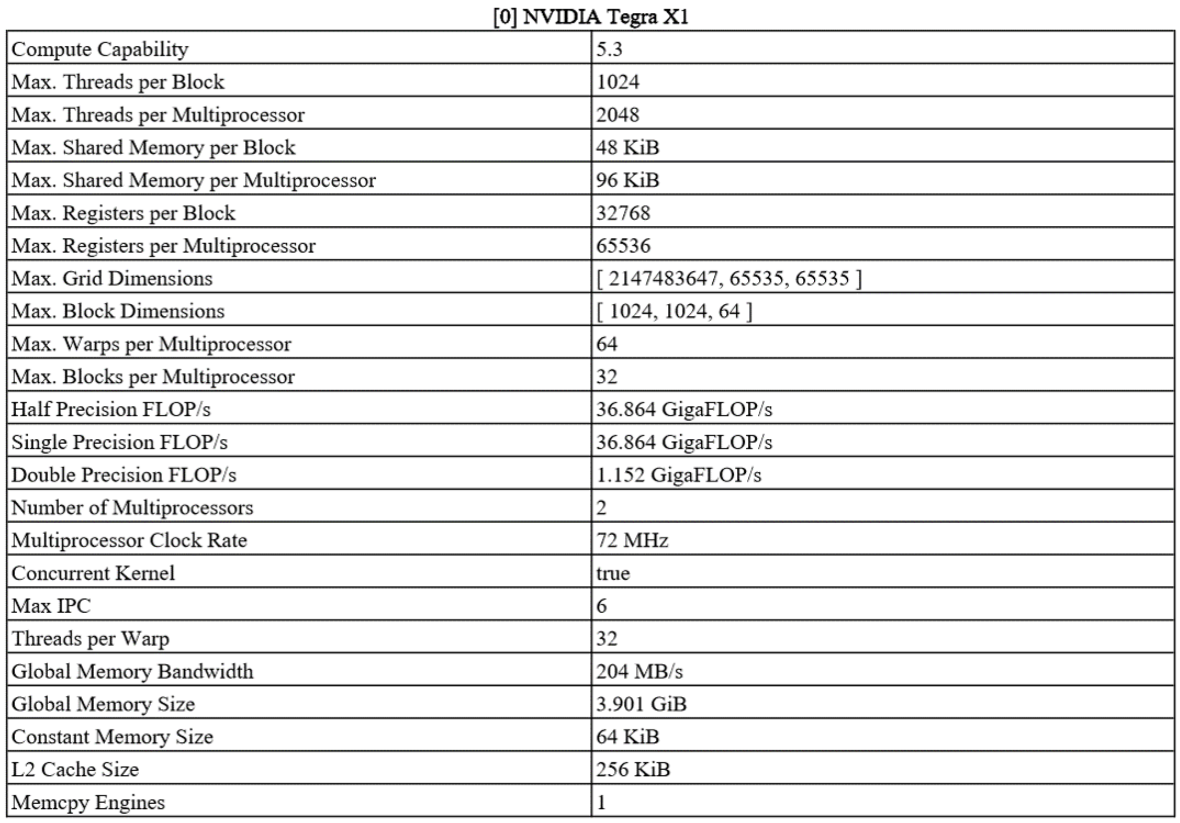


(b) Maxwell Streaming Multiprocessor Architecture

Figure A.1: NVIDIA's Maxwell GPU Architecture

The Tegra X1 SoC has two Maxwell Streaming Multiprocessors (SMM) with 128 CUDA Cores (Single Precision Floating Point ALU) each, partitioned into four distinct 32-core blocks. This organization aligns with the warp size for a more efficient datapath. The texture/L1 cache memory is shared between each pair while the total amount of shared memory available is 64KB [36].

A.2 NVIDIA Tegra X1 Device Specifications



[0] NVIDIA Tegra X1

Compute Capability	5.3
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	32768
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Half Precision FLOP/s	36.864 GigaFLOP/s
Single Precision FLOP/s	36.864 GigaFLOP/s
Double Precision FLOP/s	1.152 GigaFLOP/s
Number of Multiprocessors	2
Multiprocessor Clock Rate	72 MHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
Global Memory Bandwidth	204 MB/s
Global Memory Size	3.901 GiB
Constant Memory Size	64 KiB
L2 Cache Size	256 KiB
Memcpy Engines	1

Figure A.2: TX1 Device Specifications