A Quantitative Analysis of Optimizations on the Tegra X1 for Probabilistic Sampling Inference

Menna El-Shaer, Keith Redmill, and Fusun Ozguner

Abstract—Processing large amounts of data is computationally expensive for real-time systems in terms of process execution time and resources, specifically memory space. As a result, optimizing with respect to computation time is important for real-time systems. This research presents the costs and benefits of using different types of GPU memories, data formats and memory access patterns to perform inference on a set of images taken from multiple cameras (viewpoints) while driving through traffic intersections. Analysis of the data shows that faster runtimes can be better achieved by ensuring data or instruction coalescence and vectorization than choosing a particular memory model implementation. Contribution of this research is to understand GPU optimization techniques and provide a comparison between different runtime profiles of the same inference algorithm.

Index Terms—Probabilistic Inference; Graphical Processing Units; Deep Learning

1 Introduction

Graphical Processing Units (GPUs) have been instrumental in deep learning applications over the past few years. The massive parallelism inherited in GPUs allows for practical training and inference on the large amount of data required. Nevertheless, system memory is finite and response times of real-time control applications, such as those used for automated driving and ADAS are fast, which require fast sensory data processing. Large volume of data and rapid processing calls for a tradeoff between the perception algorithm's optimal execution performance and the bias of its outcome. In this work, we study optimization techniques on an embedded processor NVIDIA Tegra X1. This processor is used in most autonomous vehicles and robotics applications, which make it suitable for real-time testing. In Section 2, we introduce the optimization techniques studied and the metrics we used to analyze the algorithms performance. We study an example of a generative probabilistic sampling method used for deep learning inference – Gibbs Sampling, which is a Markov Chain Monte Carlo method that constructs 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 [1], which makes them suitable for general inference problems. In Section 3 we detail the experiments done and the results are presented in Section 4.

2 Background

2.1 Inference Algorithm

Inspired by statistical mechanics, the Restricted Boltzmann Machine [2], [3], [4], a probabilistic graphical model's energy with bipartite connections between its stochastic variables $v \in [0,1]^V$ and $h \in \{0,1\}^H$ shown in Figure 1 is defined with model parameters θ as:

$$E(v,h) = -bv - ch - h^T W v \,;\, \theta \,\epsilon\, \{b, c, W\} \tag{1}$$

Interactions between nodes are represented by the weight matrix W, while b and c are bias terms for visible nodes v and hidden nodes h respectively.

The joint probability distribution between the variables is modeled as:

$$p(v,h) = \frac{e^{-E(v,h)}}{Z} \tag{2}$$

where Z is the partition function or probability normalizing constant.

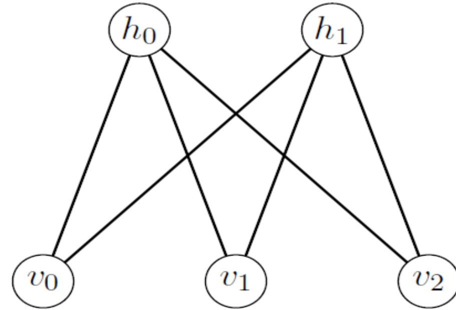

Fig. 1. A graphical representation of a Restricted Boltzmann Machine

Given input variables $v \in [0,1]^V$, model parameters θ can be learned by maximizing the log-likelihood $F(v)$ as shown in equations 3 and 4.

$$F(v) = -\,log(p(v,h)) = -\,log(\sum_h e^{-E(v,h)}) \tag{3}$$

$$-\frac{\partial log(p(v))}{\partial \theta} = \frac{\partial F(v)}{\partial \theta} - \sum_v p(v) \frac{\partial F(v)}{\partial \theta} \tag{4}$$

The second negative term in equation 4 which is the expectation over all possible input data configurations is intractable in practice, to make computations feasible we will approximate the likelihood by fixing the number of

model samples to a finite N as shown in equation 5 and use repeated sampling i.e. Markov Chain Monte Carlo (MCMC) which results in N samples drawn from a distribution similar to the marginal distribution $p(v)$.

$$-\frac{\partial log(p(v))}{\partial\theta} \approx \frac{\partial F(v)}{\partial\theta} - \frac{1}{N}\sum_{v \epsilon N}\frac{\partial F(v)}{\partial\theta} \quad (5)$$

The bipartite structure of the RBM allows for repeated alternating Block Gibbs Sampling [2], [3], [4] between the two layers as shown in equation 6.

$$\begin{aligned} p(h_j = 1 \,|\, v) &= g(c_j + \sum_i v_i w_{ij}) \\ p(v_i = 1 \,|\, h) &= g(b_i + \sum_j h_j w_{ij}) \end{aligned} \quad (6)$$

Such approximation defined by Hinton in [5], known as constrastive divergence (equation 7) has been shown to work well with $N = 1$.

$$\triangle w_{ij} = \lambda \,.\, (E(v_i, h_j)_{\text{data}} - E(v_i, h_j)_{\text{reconstruction}}) \quad (7)$$

where λ is the learning rate of the stochastic gradient descent of equation 5.

2.2 The Heterogeneous Computing Model

GPUs are an example of heterogeneous processors, where two separate different architecture processors are connected via a PCIe bus. Each system has its own DRAM and resources. The GPU is comprised of a scalable array of *Streaming Multiprocessors* or SMs. Parallelism is achieved using NVIDIA's Single Instruction Multiple Thread (SIMT) model, where multiple threads execute the same instruction in a group called *warp* [6], [7], [8]. The SMs are responsible for processing thread blocks of the kernel grid launched by the CPU co-processor which the programmer gets to control. Faster runtimes is achieved by concurrent memory access, processing several elements per thread as well as increasing the GPU's occupancy as a measure of hardware utilization or how much of the device's resources are being used to execute a specific kernel. In the SIMT model, this is equivalent to increasing concurrent warps or the number of warps runnning simultaneously on the SMs.

2.3 Optimization Techniques

2.3.1 *Memory Types*

A GPU has several types of memory, each characterized by their size, latency and throughput. Due to the principle of data locality [9], computing performance is greatly affected by type of memory storage used for a task. Table 1 illustrates the different types of memory found in a GPU, along with their caching behavior and data lifetime.

TABLE 1
Summary of different memory types on the GPU

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	

(a) Global Memory

- *Pagable Memory:* The traditional heterogeneous memory model.
- *Pinned and Unified Memory*: Unified Memory offers a single-pointer-to-data model that is conceptually similar to zero-copy memory, where the physical location of memory is pinned in CPU system memory such that a program may have fast or slow access to it depending on where it is being accessed from. Unified Memory, on the other hand, decouples memory and execution spaces so that all data accesses are fast [8].

(b) Constant Memory is cached read-only memory in respect of the GPUs memory view, it can be declared at compile time or defined at runtime as read only by the host. There is separate block for constant memory; it is a form of addressing global memory. Advantages of using global memory is distributing or broadcasting data in a single cycle to all the threads in a warp. For compute capability 2.0 and higher GPU devices, constant memory fetch speed is almost as fast as L1 cache speeds [10].

2.3.2 *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. *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 [8].

2.3.3 *Data Layout*

Memory Access Order: 1. Data is read and written row major 2. Data is read and written column major

2.3.4 *Data Format*

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 [11]. In addition, storing half-precision data requires less memory usage for transfer and computations than full-precision floating-point data.

2.4 Performance Metrics

2.4.1 *Total Execution Time*

Measured using timers and CUDA events [8].

2.4.2 *Global Memory Bandwidth (Data Throughput)*

Two approaches are usually taken together to maximize memory bandwidth: hiding memory latency by increasing the number of warps executing concurrently along with coalescing and aligning memory accesses [12]. The measured bandwidth a kernel actually achieves when reading and writing global memory is known as *Effective Bandwidth* and is calculated using:

$$\text{Bandwidth (GB/s)} = \frac{(\text{bytes read} + \text{bytes written}) * 10^{-9}}{\text{time}} \tag{8}$$

3 EXPERIMENTS

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 implementation: all variables are FP32, defined in pageable global memory and are read and written in a column major order. Table 2 shows all experimental test cases done 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 2
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

3.1 Implementation Details

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.

4 RESULTS AND DISCUSSION

Total execution times for each of the cases in Table 2 were measured using cudaEvents and resulting times are shown in Table 3. Total execution time was defined as the sum of memory transfers and one iteration of 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.

TABLE 3
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 4
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

Times in Tables 3, 4 and 5 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, which translates to a theoretical bandwidth of 0.208 GB/s.

From Table 3, 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 unified memory choice is not as a significant 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.

From Table 6, 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 Section 1, 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 4 and 5.

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 5
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 6
Global Memory Metrics

	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

5 Conclusion

The goal of this work is to quantitively analyze the performance of different GPU optimization techniques regularly discussed in the literature on a classical probabilistic inference problem applied to sampling images from a network trained to uncover the probability distribution of real-world traffic scenes. We studied the application of four optimization techniques and experimentally tested seven different combinations of memory types, access patterns, data layout and format on the sampling algorithm. The results of our experiments showed that more optimization benefit can be achieved by ensuring data coalescence and vectorization than choosing the memory type which is consistent with SIMD parallelization. One type of memory that we didn't study in our implementation was the use of shared memory for sharing data among threads in a block. Since this is the only on-chip memory in a GPU, it will likely lead to the most performance benefit provided that any overhead produced from thread dependence and synchronization operations is minimal. Although runtimes of our implementation are not real-time ready, understanding the effect of the different techniques can help towards achieving real-time performance.

TABLE 7
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

Acknowledgments

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

References

[1] "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).
[2] R. R. Salakhutdinov, "Learning deep generative models," *Annual Review of Statistics and its Application*, vol. 2, pp. 361–385, April 2015.
[3] 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.
[4] 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.
[5] G. E. Hinton and S. Osindero, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, 2006.
[6] J. Sanders and E. Kandrot, *CUDA by example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.
[7] 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.
[8] "CUDA C programming guide," Available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2018/04/16), NVIDIA Corporation.
[9] P. J. Denning, "The locality principle," *Communication Networks and Computer Systems*, p. 4367, 2006.
[10] S. Cook, "Chapter 6 - memory handling with cuda," in *CUDA Programming*, ser. Applications of GPU Computing Series, S. Cook, Ed. Boston: Morgan Kaufmann, 2013, pp. 107 – 202.
[11] M. Harris, "Mixed-precision programming with CUDA 8," Available at https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/ (2018/05/18).
[12] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Wrox, 2014.