A Comparison of Stochastic Gradient MCMC using Multi-Core and GPU Architectures

Sergio Hernández [1]
shernandez@ucm.cl

José Valdés [2]

Matias Valdenegro [3]

***Abstract*—Deep learning models are traditionally used in big data scenarios. When there is not enough training data to fit a large model, transfer learning repurpose the learned features from an existing model and re-train the lower layers for the new task. Bayesian inference techniques can be used to capture the uncertainty of the new model but it comes with a high computational cost. In this paper, we compare the run time performance of an Stochastic Gradient Markov Chain Monte Carlo method using two different architectures, GPU and multi-core CPU. As opposed to the widely usage of GPUs for deep learning, we found significant advantages from using modern CPU architectures.**

***Index Terms*—Bayes methods, Monte Carlo methods, learning (artificial intelligence)**

I. Introduction

As the amount of stored data is increased. analytical techniques for processing large-scale datasets are also required. In the other hand, big datasets are not only described by the number of observations but also through its dimensionality (number of features). Therefore. modern scalable Bayesian inference techniques must be able to improve performance on a single machine (scale up) as well as distributing performance across several machines or CPU cores (scale out) [1].

In particular. Bayesian inference is used for computing posterior distributions of the model parameters. In order to obtain such posterior estimates. Markov Chain Monte Carlo (MCMC) methods approximate intractable integrals as an expectation using a finite number of samples. However, this expectation requires the full dataset in order to guarantee convergence. Splitting a big dataset into smaller batches and running parallel chains is a simple alternative for scaling out Bayesian inference. Nevertheless, each chain will have its own posterior approximation and there is no standard method for recombining the results into a single posterior distribution.

The Stochastic Gradient Langevin Monte Carlo (SGLD) algorithm is an approximation technique that fully exploits data subsets [2]. The method replaces gradients and likelihood evaluations required by standard MCMC by stochastic gradients based on data subsets (mini-batches). Some implementations can be found in software libraries such as Edward [3], ZhuSuan [4] and the SGMCMC package [5]. All of these implementations are based on TensorFlow[1] as a backend and therefore inherit the scalability of GPU computation. However, there is little research done on the scalability of SGLD using modern CPU architectures. Therefore, in this paper we present a novel comparison of SGLD implementations on GPUs and CPUs.

[1] Laboratorio de Procesamiento de Información Geoespacial, Universidad Católica del Maule. Chile.

[2] Centro de Innovación en Ingeniería Aplicada. Universidad Católica del Maule. Chile.

[3] German Research Center for Artificial Intelligence, Robotics Innovation Center. Germany.

A. Related work

Deep learning research has attracted a lot of attention from practitioners and industry. Therefore, most modern models require specific hardware accelerators such as GPUs and multi-core CPUs. Benchmark suites such as BenchIP [6] and BenchNN [7] have been proposed for the task of hardware evaluation for neural networks. However, there is little research done on the scalability of SGLD using modern CPU architectures. Therefore, in this paper we present a novel comparison of SGLD implementations on GPUs and CPUs.

II. Stochastic Gradient MCMC

Stochastic Gradient Descent (SGD) is an optimization technique traditionally used for out-of-core training deep learning models. Let $\mathbf{D} = \{\mathbf{d_1}, \ldots, \mathbf{d_N}\}$ represent a typical dataset used for classification problems, as a set of tuples $\mathbf{d} = (\mathbf{x}, y)$ that contains features $\mathbf{x} \in \mathbb{R}^D$ and labels $y = \{1, \ldots, K\}$. Conversely, the stochastic optimization procedure iterates over mini-batches of size $B \ll N$ and computes a point estimate $\theta^* = \operatorname{argmax} p(\mathbf{D}|\theta)$.

Deep learning models such as convolutional neural networks may exploit model parallelism for fully connected layers and data parallelism for convolutional layers. The former training technique applies parallelism across the model dimension (number of features) and the latter technique applies parallelism across the data dimension (number of examples). In a model parallel setting, different workers are trained on the same batch and they recombine results as soon as possible. In the other hand, in a data parallel setting, each worker is trained on different batches of data [8].

In the Bayesian framework, we consider the unknown parameter θ as a random variable. The Stochastic Gradient Langevin Monte Carlo (SGLD) algorithm uses an stochastic gradient approximation to generate samples from the posterior distribution $p(\theta|\mathbf{D})$. The SGLD algorithm generate proposals using:

[1] https://www.tensorflow.org/

$$\theta_{t+1} = \theta_t + \frac{\epsilon_t}{2}\left(\nabla \log p(\theta_t) + \frac{N}{B}\sum_{i}^{B} \nabla \log p(\mathbf{d_i}|\theta_t)\right) + \eta_t \tag{1}$$

where $\eta_t \sim \mathcal{N}(0, \epsilon_t)$ and $\epsilon_t \mapsto 0$ is a time-decaying step size.

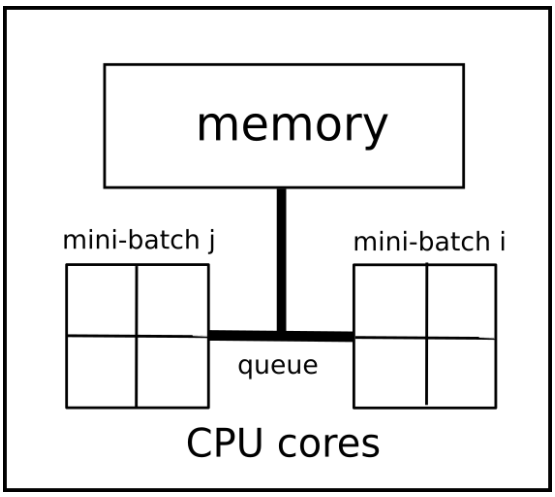
It is important to notice that using SGLD on a fully connected layer requires dense matrix multiplications for computing the log-likelihood and the log-prior. Therefore, we present two implementations using two different architectures.

A. GPU SG-MCMC

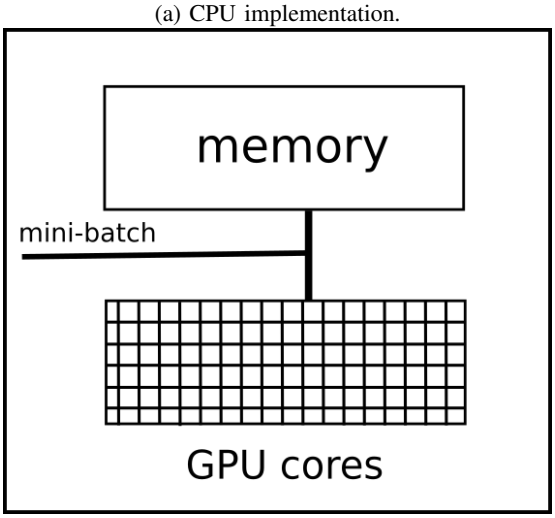
Graphic Processing Units (GPus) are many-core processors that offer massively parallel computation. GPUs are well suited for data parallel operations such as computing the gradient in convolutional neural networks and Monte Carlo methods [9]. In order to implement the SGLD algorithm we use CuPy, a Python library for GPU computation[2]. The library includes optimized GPU array manipulation and linear algebra routines, which are compatible with other CPU array manipulation routines such as Numpy.

B. CPU SG-MCMC

Modern CPUs are equipped with multiple cores which are well suited to single instruction multiple data operations. The Python programming language is built around a Global Interpreter Lock, which prevents the execution of multiple threads at once. Nevertheless, parallel Markov can be spawn using all available cores. Moreover, data parallelism can be achieved by sharing a data queue that feeds each process with a different data mini-batch. Figure 1 depicts the proposed GPU and CPU implementations of the SGLD algorithm.



(a) CPU implementation.



(b) GPU implementation.

Fig. 1: SG-MCMC implementation on CPU and GPU. In CPU, a data queue feeds multiple Markov chains. In GPU, a single mini-batch is fed to a Markov chain that updates model parameters.

III. Experimental results

Serial versions of the SGD and SGLD algorithms are tested on CPU and compared with parallel implementations on GPU. All experiments are run on a server with an Intel Xeon E5-2620 CPU with 12 cores using a single fully connected softmax layer (which is widely used in transfer learning). The server is also equipped with an NVIDIA TITAN X GPU and both implementations were developed using Python 3.4.9.

A. Synthetic data

In this section, the goal is to demonstrate performance and scalability of the SGLD algorithm. The GPU and CPU implementations are tested using randomly generated multi-class classification problems. Different combinations on the number of features D and number of instances N were tested. The number of classes was set to $K = 3$, the number of epochs was set to $E = 2 \times 10^4$ and the size of the mini-batches was set to $B = 50$ for all examples.

Firstly, a serial version of the SGD algorithm is tested on the CPU and compared with a parallel implementation on the GPU. All experiments are run on a Server with an Intel Xeon E5-2620 CPU with 12 cores using a single fully connected softmax layer (which is widely used in transfer learning). The server is also equipped with an NVIDIA TITAN X GPU and both implementations were developed using Python 3.4.9. Table I shows the run time for the different values of D and N. According to [8], convolutional layers contain 90% of the computation and 5% of the parameters, while fully connected layers contain 95% of the parameters and $5\% - 10\%$ of the computation. Consistently, since there is a fully connected layer operating on a small batch of data, there is no speedup and the GPU implementation is about 10x slower than the CPU implementation.

Secondly, the run time behavior of a multi-core CPU implementation of SGLD is shown in Table II. Similar to SGD, run time is scales with the number of data points N and

[2]https://cupy.chainer.org/

D	N	CPU Time [s]	GPU Time [s]	Speed-Up
10	1000	11.98	124.85	10.42
10	10000	119.90	1245.98	10.39
10	50000	594.71	6186.05	10.40
10	100000	1195.62	12261.716	10.25
50	1000	13.67	123.13	9.00
50	10000	135.96	1230.19	9.04
50	50000	689.89	6137.79	8.89
50	100000	1487.26	12240.06	8.22
100	1000	15.71	123.53	7.86
100	10000	156.94	1231.121	7.84
100	50000	806.78	6176.91	7.65
100	100000	1770.83	12394.13	6.99

TABLE I: Run time comparison of CPU and GPU implementations of SGD for a softmax regression problem

features D. This is not the case of the GPU implementation of SGD, where the number of features seems not to affect the run time due to the massive parallel processing capabilities of the GPU.

D	N	CPU Time [s]	GPU Time [s]	Speed-Up
10	1000	24.75	294.04	11.87
10	10000	245.22	2946.67	12.01
10	50000	1186.81	14518.66	12.23
10	100000	2380.18	29010.43	12.18
50	1000	25.70	288.02	11.48
50	10000	246.64	2885.16	11.69
50	50000	1230.02	14467.73	11.76
50	100000	2496.15	28930.81	11.59
100	1000	30.95	288.21	9.31
100	10000	279.28	2903.20	10.39
100	50000	1450.57	14531.93	10.01
100	100000	2796.20	29117.73	10.41

TABLE II: Run time comparison of CPU ans GPU implementations of SGLD for a softmax regression problem

B. Bayesian transfer learning

Training deep neural networks requires a vast amount of labeled data in order to achieve optimal performance. However, it becomes harder to collect data for any specific task and manually label examples as required for supervised classification problems. Transfer learning can alleviate the need of such amount of data by utilizing knowledge from previously learned tasks and applying this to the new task. In this experiment, we perform Bayesian transfer learning for training a plant diseases dataset using a previously trained deep neural network.

The Plant Village dataset contains 54306 images with combinations of 14 crops and 26 diseases (see Figure 2), leaving a total number of $K = 38$ classes. The images consist of 256×256 pixels in the RGB color space [10]. The VGG16 deep neural network with weights trained on the Imagenet dataset is used to extract features from the plants diseases images. The original class distribution is unbalanced, therefore random over-sampling is used to achieve a uniform class distribution, leaving a total number os $N = 203500$ training samples and 5700 test samples. Moreover, images are resized to 150×150 pixels and minmax normalization is used on each one of the examples.

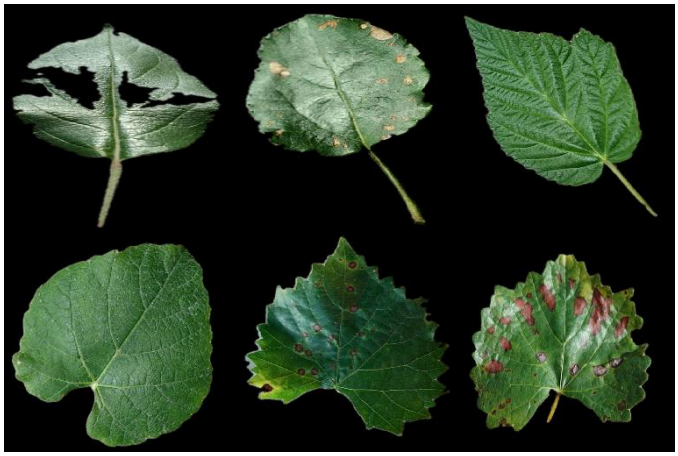
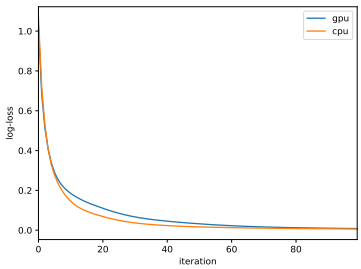
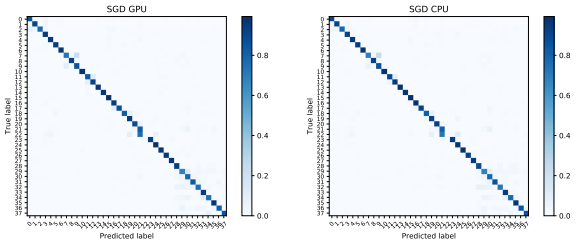


Fig. 2: Leaf images from healthy and infected plants from the Plant Village dataset



(a) SGD log-loss.



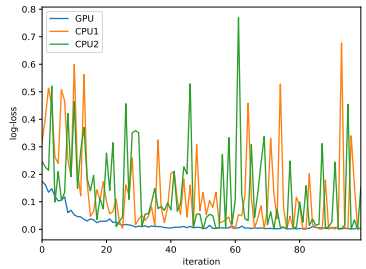
(b) SGD GPU confusion matrix.

(c) SGD CPU confusion matrix.

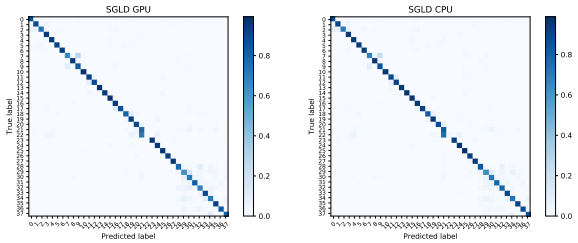
Fig. 3: SGD performance comparison between the CPU and GPU implementations.

Firstly, the SGD algorithm is used to train a total number of 100 epochs using the softmax regression model with mini-batch size $B = 50$. Figure 3a shows the log-loss function for both, the GPU and CPU implementations. Furthermore, the model accuracy is evaluated using the final model parameters and the results can be seen in Figures 3b and 3c.

Secondly, GPU and multi-core CPU implementations of the SGLD algorithm are used to obtain posterior estimates of the model parameters. As opposed to SGD, most MCMC techniques such as SGLD requires a burn-in period to enter a high probability region. In this case, both implementations are tested using 10 burn-in epochs and mini-batch size $B = 50$. After that, each method runs 100 epochs and the model param-



(a) SGLD log-loss.



(b) SGLD GPUconfusion matrix.

(c) SGLD CPU confusion matrix.

Fig. 4: SGLD performance comparison between the CPU and GPU implementations.

eters are kept as samples from the posterior distribution. Figure 4a shows the log-loss function for both implementations and Figures 4b and 4c show the achieved performance.

The performance of the SGLD model is evaluated using the mean of the posterior samples and the results can be seen on Table III.

	Precision	Recall	$F-1$ score	Time[s]
GPU	0.86	0.85	0.84	2113.63
CPU	0.84	0.83	0.83	16816.57

TABLE III: SGLD performance on the Plant Village test set.

IV. Conclusions

SGD is a serial algorithm that can be effectively computed using data parallel operations for large scale deep learning models such as convolutional neural nets. Although SGLD is an stochastic gradient MCMC method based on SGD, the algorithm can benefit from data parallel operations in a multi-core CPU architecture when the model is shallow (e.g. a single fully connected layer). The speedups achieved with synthetic data are not compatible with other real data sets such as transfer learning data sets, which is mainly due to the size of the data batches. Therefore, hybrid approaches based on multi-core CPU and multi-GPU could also be beneficial in real life scenarios.

References

[1] Elaine Angelino, Matthew James Johnson, and Ryan P Adams. Patterns of scalable Bayesian inference. *Foundations and Trends in Machine Learning*, 9(2-3):119–247, 2016.

[2] Max Welling and Yee W Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688, 2011.

[3] Dustin Tran, Matthew W Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, and Alexey Radul. Simple, distributed, and accelerated probabilistic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7598–7609. Curran Associates, Inc., 2018.

[4] Jiaxin Shi, Jianfei. Chen, Jun Zhu, Shengyang Sun, Yucen Luo, Yihong Gu, and Yuhao Zhou. ZhuSuan: A library for Bayesian deep learning. *arXiv preprint arXiv:1709.05870*, 2017.

[5] Jack Baker, Paul Fearnhead, Emily B Fox, and Christopher Nemeth. sgmcmc: An r package for stochastic gradient markov chain monte carlo. *arXiv preprint arXiv:1710.00578*, 2017.

[6] Jin-Hua Tao, Zi-Dong Du, Qi Guo, Hui-Ying Lan, Lei Zhang, Sheng-Yuan Zhou, Ling-Jie Xu, Cong Liu, Hai-Feng Liu, Shan Tang, Allen Rush, Willian Chen, Shao-Li Liu, Yun-Ji Chen, and Tian-Shi Chen. Benchip: Benchmarking intelligence processors. *Journal of Computer Science and Technology*, 33(1):1–23, Jan 2018.

[7] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam. Benchnn: On the broad potential application scope of hardware neural network accelerators. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 36–45, Nov 2012.

[8] X. Li, G. Zhang, K. Li, and W. Zheng. Chapter 4 - deep learning and its parallelization. In Rajkumar Buyya, Rodrigo N. Calheiros, and Amir Vahid Dastjerdi, editors, *Big Data*, pages 95 – 118. Morgan Kaufmann, 2016.

[9] Anthony Lee, Christopher Yau, Michael B Giles, Arnaud Doucet, and Christopher C Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Journal of computational and graphical statistics*, 19(4):769–789, 2010.

[10] S. P Mohanty, D. P. Hughes, and M. Salathé. Using deep learning for image-based plant disease detection. *Frontiers in plant science*, 7:1419, 2016.