SLIDE : Training Deep Neural Networks with Large Outputs on a CPU faster than a V100-GPU

Beidi Chen
beidi.chen@rice.edu

Tharun Medini
tharun.medini@rice.edu

James Farwell
james.c.farwell@intel.com

Sameh Gobriel
sameh.gobriel@intel.com

Charlie Tai
charlie.tai@intel.com

Anshumali Shrivastava
anshumali@rice.edu

Abstract

Deep Learning (DL) algorithms are the central focus of modern machine learning systems. As data volumes keep growing, it has become customary to train large neural networks with hundreds of millions of parameters with enough capacity to memorize these volumes and obtain state-of-the-art accuracy. To get around the costly computations associated with large models and data, the community is increasingly investing in specialized hardware for model training. However, specialized hardware is expensive and hard to generalize to a multitude of tasks. The progress on the algorithmic front has failed to demonstrate a direct advantage over powerful hardware such as NVIDIA-V100 GPUs. This paper provides an exception. We propose SLIDE (**S**ub-**LI**near **D**eep learning **E**ngine) that uniquely blends smart randomized algorithms, with multi-core parallelism and workload optimization. Using just a CPU, SLIDE drastically reduces the computations during both training and inference outperforming an optimized implementation of Tensorflow (TF) on the best available GPU. Our evaluations on industry-scale recommendation datasets, with large fully connected architectures, show that training with SLIDE on a 44 core CPU is more than 3.5 times (1 hour vs. 3.5 hours) faster than the same network trained using TF on Tesla V100 at any given accuracy level. On the same CPU hardware, SLIDE is over 10x faster than TF. We provide codes and scripts for reproducibility.

1 Introduction

Vast amounts of data powered by the exponential increase in computing capabilities have been instrumental in the success of DL. More notably, with the advent of the powerful Graphic Processing Unit (GPU) [14], training processes have been drastically accelerated. Nevertheless, we are now reaching a limit beyond which there are fewer hopes of obtaining better speedups in fundamental operations like matrix multiplication. Furthermore, the need for astronomical size neural networks and unprecedented growth in the data volumes have worsened this problem. As a result, the community is heavily investing in dedicated hardware to take DL further beyond this point [6].

Exploiting Adaptive Sparsity in Neural Networks: In popular frameworks like Tensorflow (TF), Sampled Softmax [5] is deployed to estimate the full softmax efficiently. While sampled softmax offers computational savings, it has high estimation bias [3]. This leads to poor convergence behavior which is empirically verified in our experiments in section 4. In this paper, we will exploit the idea of adaptive sparsity [3] or adaptive dropouts [1]. The idea stems from several recent observations [12, 11] that we can accurately train neural networks by selectively sparsifying most of the neurons, based on their activation, during every gradient update. [17] has also shown that selective sparsification can in-fact be superior in accuracy due to implicit regularization. However, selective sparsification does not directly lead to computational savings. [16] shows the first possibility of an algorithmically efficient solution by employing Locality Sensitive Hash (LSH) tables to identify a sparse set of neurons efficiently during each update. The proposed algorithm has an added advantage of making

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

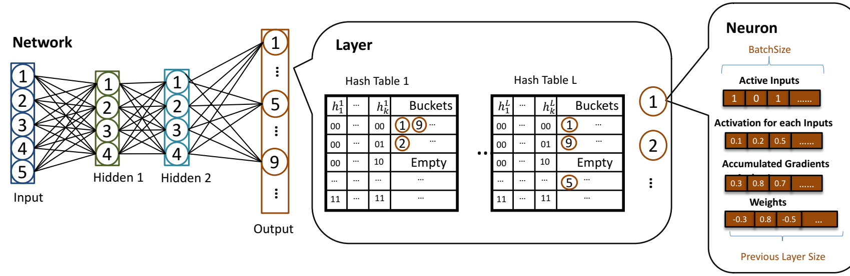

Figure 1: Architecture: The central module of SLIDE is Network. The network is composed of few-layer modules. Each layer module is composed of neurons and a few hash tables into which the neuron ids are hashed.

the gradient update HOGWILD style [15] parallel. Such parallelism does not hurt convergence because extremely sparse and independent updates are unlikely to overlap and cause conflicts of considerable magnitude. Despite all the niceness presented, current implementation of [16] fail to demonstrate that the computational advantage can be translated into a faster implementation when directly compared with hardware acceleration of matrix multiplication. In particular, it is not clear if we can design a system that can effectively leverage the computational advantage and at the same time compensate for the hash table overheads using limited (only a few cores) parallelisms. In this paper, we provide the first such implementation for large fully connected neural networks.

1.1 Our Contributions

Our main contributions are as follows:

- We show the first C++ OpenMP based system SLIDE with modest multi-core parallelism on a standard CPU that can outperform the massive parallelism of a powerful V100 GPU on a head-to-head time-vs-accuracy comparison.
- We make several novel algorithmic and data-structural choices in designing the LSH based sparsification to minimize the computational overheads to a few memory lookups only (truly $O(1)$). At the same time, it does not affect the convergence of the DL algorithm.
- We provide a rigorous evaluation of our system on two large benchmark datasets involving fully connected networks and show that SLIDE, on a modest CPU can be up to 3.5x faster, in wall clock time, than the best possible alternative with the best possible choice of hardware, at any accuracy. We perform a CPU-efficiency analysis of SLIDE using Intel VTune Performance Analyzer and show that memory-bound inefficiencies reduce for SLIDE with an increasing number of cores while it is the opposite for TF-CPU.
- Our analysis suggests that SLIDE is a memory-bound application, prone to some bottlenecks described in section 3. With careful workload and cache optimizations (eg. Transparent Hugepages) and a data access pattern (eg. SIMD instructions), we further speed up SLIDE by roughly 1.3x, making the overall speed up to 3.5x faster than TF-GPU and over 10x faster than TF-CPU.

2 Proposed System: SLIDE

Before introducing SLIDE in details, we define important notations: **1)** B: input batch size **2)** N_l^j: Neuron j in layer l **3)** x_l: inputs for layer l in the network **4)** w_l^a: weights for a^{th} neuron in layer l **5)** h_l: hash functions in layer l **6)** N_l^a: the set of active neurons in layer l for the current input.

Initialization: Figure 1 shows the modular structure of SLIDE. Every layer object contains a list of neurons and a set of LSH sampling hash tables. Each hash table contains ids of the neurons that are hashed into the buckets. During the network initialization, the weights of the network are initialized randomly. After weight initialization, $K \times L$ LSH hash functions are initialized along with L hash tables for each of the layers. For instance, the example network in Figure 1 maintains hash tables in two hidden layers as well as the output layer. The LSH hash codes $h_l(w_l^a)$ of the weight vectors of neurons in the given layer are computed according to the hash functions. The id a of the neuron are saved into the hash buckets mapped by the LSH function $h_l(w_l^a)$. This construction of LSH hash tables in each layer is a one-time operation which can easily be parallelized with multiple threads over different neurons in the layer independently.

Sparse Feed-Forward Pass with Hash Table Sampling: In the feed-forward phase, given a single training instance, we compute the network activation until the final layer, which gives us the output.

In SLIDE, instead of calculating all the activations in each layer, the input to each layer x_l is fed into hash functions to compute $h_l(x_l)$. The hash codes serve as a query to retrieve ids of active (or sampled) neurons from the matching buckets in hash tables. Only the activations of active neurons are calculated and passed on as the inputs to the next layer. The other activations, are directly treated as 0 and never computed.

The above-described operations are performed sequentially in every layer, starting from the very first layer where the input is the data itself. Even in the output layer, which has softmax activation, only neurons sampled from hash tables are treated as active neurons. For softmax, for every active neuron, we compute its output as $\sigma(N_o^k) = \frac{e^{x_o w_o^k}}{\sum_{N_o^a} e^{x_o w_o^k}}$. Note that the normalizing constant for softmax is no longer the sum over all neurons but only the active ones.

Sparse Backpropagation or Gradient Update: The backpropagation step follows the feed-forward step. After computing the output of the network, we compare it with the known label of the input and backpropagate the errors layer-by-layer to calculate the gradient and update the weights. Here we used the classical backpropagation message passing type implementation rather than vector multiplication based. For every training data instance, after updating the weights of any given neuron, the neuron propagates the partial gradients (using error propagation) back to only active neurons in previous layers via the connected weights. As a result, we never access any non-active neuron or any non-active weight, which is not part of the feed-forward process on a given input. The process ensures that we take full advantage of sparsity. Our computation over each input is only of the order of active neurons and weights rather than the total number of parameters. It should be noted that if we compute activation for $s < 1$ fraction of neurons in each layer (on an average), the fraction of weights that needs to be updated is s^2 only, which is a significant reduction when s is small (as is the case for our experiments).

OpenMP Parallelization across Training Instances in a Batch: For any given training instance, both the feed-forward and backpropagation operation are sequential as they need to be performed layer by layer. SLIDE uses usual Batch Gradient Descent with ADAM optimizer, where the batch size is generally in the order of hundreds. Each data instance in the batch runs in a separate thread and its gradients are computed in parallel. To ensure the independence of computation across different threads, every neuron stores two additional arrays, each of whose length is equal to the batch size. These arrays keep track of the input specific neuron activations and error gradients. Every input is assigned an id, which can be used as an index to locate its activation (or error gradient) on any neuron. Besides, we also have a bit array at each neuron to determine whether the particular input activates a neuron or not. This small memory overhead is negligible for CPUs as they have abundant memory. But it ensures that the gradient computation is independent across different instances in the batch.

The extreme sparsity and randomness in gradient updates allow us to asynchronously parallelize the accumulation step of the gradient across different training data without leading to a considerable amount of overlapping updates. SLIDE heavily capitalizes on the theory of HOGWILD [15] which shows that a small amount of overlap is tolerable. It does not hurt the convergence even if we resolve the concurrent updates randomly. Thus, after independently computing the gradients, each thread pushes the updates directly to the weights asynchronously. This asynchronous update avoids synchronization during batch accumulation which is otherwise sequential in the batch.

3 Threading Model and Platform Micro-architecture Optimization

Our experimental analysis shows that SLIDE is a memory-bound workload. We show that a careful workload optimization to design a threading model and a data access pattern to take into consideration the underlying platform architecture leads to a significant performance boost.

Cache Optimizations: A key metric for the identification of memory and cache performance bottlenecks in a multi-threaded application, e.g., SLIDE, is the number of data misses in the core private caches. This is a significant source of coherence traffic, potentially making the shared bus a bottleneck in a symmetric multiprocessor (SMP) architecture, thus increasing memory latency.

CPU caches are arranged into cache lines. Multiple threads updating data items that happen to co-locate into the same cache line (called false sharing) can also cause cache thrashing, since these updates need to be serialized to ensure correctness, leading to performance degradation. Much

previous work (e.g., [18]) have tried to detect and resolve the issue of false sharing for OpenMP multi-threads mainly using compiler optimizations and hardware performance counters. However, generally speaking, carefully allocating data structures and aligning them on cache line boundaries (e.g., by padding) significantly reduce the false sharing opportunities. We chose to use the later alternative for SLIDE.

Address Translation and Support for Kernel Hugepages: Virtual memory provides applications with a flat address space and an illusion of sufficiently large and linear memory. The addressed memory is divided into fixed-size pages, and a page table is used to map virtual pages to physical ones. The address lookup is accelerated using Translation Lookaside Buffers (TLBs).

Since SLIDE is a workload with a large memory footprint, the performance of virtual memory paging can suffer due to stagnant TLB sizes. TLB address translation is on the processors critical path. It requires low access times which constrain TLB size (and thus, the number of pages it holds). On a TLB miss, the system must walk the page table, which may incur additional cache misses. Recent studies show that workloads with large memory footprints can experience a significant performance overhead due to excessive page table walks [7, 2].

We employ Hugepages for SLIDE, which is a technology for x86-64 architectures to map much larger pages than the default 4KB normal-sized pages on the orders of 2 MB to 1 GB. Use of huge pages (Transparent Hugepages and libhugetlbfs [4]) increases TLB reach substantially, and reduces the overhead associated with excessive TLB misses and table walks.

Vector Processing, Software Pipelining, and Prefetching: We further use software optimization techniques to improve workload performance in SLIDE. In particular, we use Vector processing which is capable of exploiting data-level parallelism through the use of Single-Instruction-Multiple-Data (SIMD) execution, where a function is called with a batch of inputs instead of an individual input (e.g., the function to update a large matrix of weights in the back-propagation phase). The implementation uses SIMD instructions (e.g., Intel AVX [9]) to implement the update to multiple weights simultaneously. Implementing a software pipeline is an excellent way to hide memory latency for memory-bound workloads. Our implementation divides the processing of data items into stages of a pipeline, where explicit software prefetch stage (using, for example, x86 PREFETCHT0 instruction set) is followed by a processing stage(s). The data items that are accessed in the future are prefetched into the core caches in advance to the time when they are needed to get processed. In particular, for a vector processing of updating of N weights, a software implementation can prefetch weight W_{i+d} (where d is the depth of the pipeline) while updating weight W_i, as a result, when it is time to process weight W_{i+d} it is already in the CPU cache.

4 Evaluations

In this section, we're going to empirically investigate SLIDE's performance against TF-GPU with V100s and TF-CPU. Our first goal is to evaluate the basic structure of the system thoroughly. Hence for the first part, we do not include the optimization described in section 3. Later, we will show the overall gains obtained by leveraging Threading Model and Platform Micro-architecture in section 4.1.

Datasets: To show SLIDE's real advantage, we'll need large networks where even a slight decrease in performance is noticeable. Thus, the publicly available extreme classification datasets [10], requiring more than 100 million parameters to train due to their extremely wide last layer, fit this setting appropriately. For these tasks, most of the computations (more than 99%), is in the final layer. We employ two large real datasets; Delicious-200K and Amazon-670K and train feed-forward networks with large output spaces (200K and 670K respectively).

Infrastructure: All the experiments are conducted on a server equipped with two 22-core/44-thread processors (Intel Xeon E5-2699A v4 2.40GHz) and one NVIDIA Tesla V100 Volta 32GB GPU. The server has an Ubuntu 16.04.5 LTS system with the installation of TF-GPU 1.12. We compiled TF-CPU 1.12 from source with GCC5.4 in order to support FMA, AVX, AVX2, SSE4.1, and SSE4.2 instructions. This boosts the performance of TF-CPU by about 35%. SLIDE is written in C++ and compiled under GCC5.4 with OpenMP flag. The most exciting part is that SLIDE only uses vanilla CPU thread parallelism and yet outperforms TF-GPU (V100) by a large margin in performance.

Hyper Parameters: For both the datasets, we adopt the same model architecture in [19]. We choose the standard fully connected neural network with one hidden layer of size 128. We choose a batch size of 128 for Delicious-200K dataset and 256 for Amazon-670K dataset as the input dimension

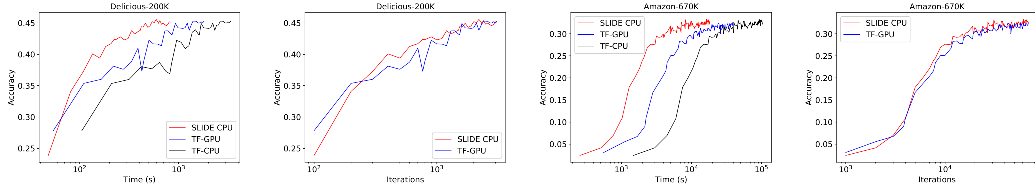

Figure 2: Comparison of SLIDE (in red) against TF-GPU (in blue) and TF-CPU (in black). The x-axis is plotted in log scale to accommodate the otherwise slow TF-CPU curve. We notice that the time required for convergence is 2.7x lower than that of TF-GPU.

for the former is very large. We run all algorithms until convergence. To quantify the superiority of SLIDE over other baselines, we also use the same optimizer, Adam [8] by varying the initial step size from $1e^{-5}$ to $1e^{-3}$ which leads to better convergence in all experiments. For SLIDE, we maintain the hash tables for the last layer, where we have a computational bottleneck of the models (owing to a large number of classes). For specific LSH setting, we choose Simhash, $K = 9$, $L = 50$ for Delicious dataset and WTA hash, $K = 8, L = 50$ for Amazon-670k dataset. We update the hash tables with an initial update period of 50 iterations with an exponential decay.

Main Results: We show the time-wise and iteration-wise comparisons for SLIDE vs TF GPU/CPU in Figure 2. Note that the x-axis is in log-scale, and all the curves have a long flat converged portion when plotted on a linear scale indicating clear convergence behavior. Red, blue and black lines represent the performance of SLIDE, TF-GPU, TF-CPU, respectively. We can see from the plots that SLIDE on CPU achieves any accuracy faster than TF on V100 demonstrating the superiority of SLIDE. TF-GPU is always faster than TF-CPU which is expected. It should be noted that these datasets are very sparse, e.g., Delicious dataset has only 75 non-zeros on an average for input features, and hence the advantage of GPU over CPU is not always straight-forward. SLIDE can be around 1.8 times faster than TF-GPU on Delicious 200k. On the larger Amazon 670k dataset, where we need more computations, the gains are substantially more. SLIDE is around 2.7 (2 hrs vs. 5.5 hrs) times faster than TF-GPU. Most of the computational benefits of SLIDE come from sampling a small subset of active neurons in the output layer. After a few iterations into the training process, the average number of neurons sampled in the output layer for Delicious-200K is ≈ 1000. Similarly, for Amazon-670K, we sample ≈ 3000 neurons. With fewer than 0.5% of active neurons, SLIDE outperforms TF-GPU on time by a huge margin on either dataset. It is interesting to note that even after compiling TF-CPU with AVX2 instructions, it is nowhere close to the performance of SLIDE or TF-GPU (SLIDE is 8x faster than TF-CPU). Therefore, it is exciting to note that without any rigorous optimization in our prototype, SLIDE outperforms both baselines using smart randomized algorithms with OpenMP parallelism.

For Iteration vs. Accuracy plots in Figure 2, we can observe that SLIDE achieves the same accuracy per iteration even though it adaptively selects neurons in some layers. This observation also confirms that adaptively selecting neurons and performing asynchronous SGD does not hurt the convergence from an optimization perspective. The plot also confirms that the advantage of SLIDE is not due to any bells and whistles in the optimization process as the convergence with iteration has very similar behavior. For this plot, we only show TF-GPU as TF-CPU would also lead to the same plot as the optimization algorithm is the same. Since SLIDE performs much fewer computations and memory accesses on the last layer, each iteration is faster than the baselines. This is the critical reason why SLIDE outperform other baselines when compared on wall-clock time.

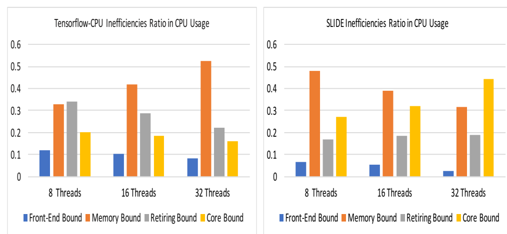

Figure 3: Inefficiencies in CPU Usage: Memory-bound inefficiencies (orange bars) are the most significant ones for either algorithm. For TF-CPU, memory-bound inefficiency rises with an increasing number of cores. For SLIDE, the memory bottleneck reduces with an increasing number of cores. Hence, SLIDE takes better advantage of higher CPU cores.

Inefficiency Diagnosis: We profile and analyze TF-CPU and SLIDE by a state-of-the-art parallel performance analyzer tool, the Intel VTune Performance Analyzer [13]. We observe that with 8,

16, 32 threads for the above tasks, for TF-CPU, the core utilization is generally low ($< 50\%$) and decreases with more threads;for SLIDE, the utilization is stable (around 80%) across all number of threads.

Figure 3 presents the distribution of inefficiencies in CPU usage for TF-CPU and SLIDE. Based on core utilization, the overall inefficiencies of TF-CPU is much more than those of SLIDE in general. Thus the distribution in figure 3 is based on those inefficiencies. It is obvious that being memory-bound is a major issue for all number of threads in the histogram. The biggest bottleneck is that the significant fraction of execution pipeline slots is stalled due to demand memory load and store. Observe that the higher the number of threads TF-CPU uses, the more memory-bound it gets.

On the other hand, the higher the number of threads SLIDE uses, the less memory-bound it becomes. Recall that the critical advantage of SLIDE is that it has a lot fewer active neurons and sparse gradient updates. Naturally, memory accesses are a lot fewer than TF-CPU due to very sparse memory accesses within each thread. Our choice of using extra arrays to separate the computations of each thread with asynchronous gradients updates (section 2) across all the threads ensures that simple OpenMP parallelism is sufficient to get near-peak utilization.

4.1 Doubling the Speedup with Threading Model and Platform Micro-architecture

For our experiments, we first install $Hugepages$ package for Ubuntu, which offers 2MB and 1GB pages. We pre-allocate 1000 2MB Hugepages and 10 1GB Hugepages which is found to be enough for both Delicious-200K and Amazon-670K datasets. Recall the issue of the false sharing for OpenMP mutli-thread. We reduce it by giving a provision to our data structures to align on cache line boundaries. Besides using Hugepages, we also used SIMD instructions (specifically, Intel-AVX) to facilitate per thread batching. In figure 4, we compare the benefit of aforementioned optimizations against an un-optimized SLIDE and TF-GPU. We notice that Cache-Optimized SLIDE (in green) is ≈ 1.3 times faster than basic SLIDE (in red). Since we already have a 2.7x speed-up over TF-GPU on Amazon-670K, it translates to 3.5x speedup over TF-GPU and a 10x speedup over TF-CPU.

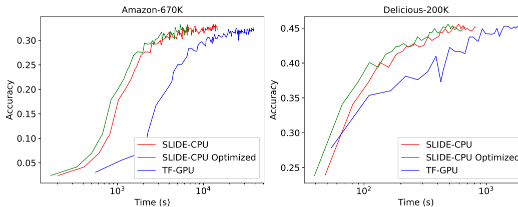

Figure 4: Impact of Hugepages and SIMD Optimization: The comparison of training time for optimized version of SLIDE against a plain version of SLIDE and TF-GPU. We can see that SLIDE-Optimized is roughly 1.3x faster than the un-optimized one on both datasets (x-axis is log scale)

4.2 Measuring the Impact of Transparent Hugepages

A direct benefit of employing Transparent Hugepages is the drastic reduction in TLB miss rate. In our implementation, TLB load miss rate for data reduces from 5.12% to 0.25%. Similarly, TLB load miss rate for instruction also decreases from 56.12% to 20.96%. Consequently, we expect a huge reduction in page table walks (PTW) incurred due to TLB misses. We corroborated this fact by observing a reduction in ratios of CPU cycles spent by PTWs caused by data and instruction TLB misses from 7.74% to 0.72% and 0.02% to 0.015% respectively. As mentioned in section 3, TLB misses cause expensive main memory reads. Using Hugepages, we reduce the memory reads caused by data and instruction TLB misses from $3,062,039/s$ to $749,485/s$ and $12,060/s$ and $11,580/s$ respectively. Finally, we also report the reduction in page faults (which can possibly occur when there is a TLB miss) from $32,548/s$ to $26,527/s$.

5 Conclusion

We provide the first evidence that a smart algorithm with modest CPU OpenMP parallelism can outperform the best available hardware NVIDIA-V100, for training large deep learning architectures. Our system SLIDE is a combination of carefully tailored randomized hashing algorithms with the right data structures that allow asynchronous parallelism. We show up to 3.5x gain against TF-GPU and 10x gain against TF-CPU in training time with similar precision on popular extreme classification datasets.

References

[1] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092, 2013.

[2] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture*, pages 237–248, 2013.

[3] Guy Blanc and Steffen Rendle. Adaptive sampled softmax with kernel based sampling. In *International Conference on Machine Learning*, pages 589–598, 2018.

[4] Jonathan Corbet. Transparent huge pages in 2.6.38. *http://lwn.net/Articles/423584/*, 2011.

[5] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1–10, 2015.

[6] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[7] V. Karakostas, O. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. Performance analysis of the memory management unit under scale-out workloads. In *International Symposium on Workload Characterization*, pages 1–12, 2014.

[8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] A. Kumar, D. Soltis, I. Esmer, I. Yoaz, and S. Kottapalli. The new intel xeon scalable processor(formerly skylake-sp). In *Hot Chips*, 2017.

[10] Himanshu Jain Yashoteja Prabhu Manik Varma Kush Bhatia, Kunal Dahiya. The extreme classification repository: Multi-label datasets code. `http://manikvarma.org/downloads/XC/XMLRepository.html#Prabhu14`.

[11] Alireza Makhzani and Brendan Frey. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*, 2013.

[12] Alireza Makhzani and Brendan J Frey. Winner-take-all autoencoders. In *Advances in neural information processing systems*, pages 2791–2799, 2015.

[13] Rama Kishan Malladi. Using intel® vtune performance analyzer events/ratios & optimizing applications. *http:/software. intel. com*, 2009.

[14] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. 2008.

[15] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[16] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454. ACM, 2017.

[17] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[18] B. Wicaksono, M. Tolubaeva, and B. Chapman. Detecting false sharing in openmp applications using the darwin framework. In *Lecture Notes in Computer Science*, pages 282–288, 2011.

[19] Ian En-Hsu Yen, Satyen Kale, Felix Yu, Daniel Holtmann-Rice, Sanjiv Kumar, and Pradeep Ravikumar. Loss decomposition for fast learning in large output spaces. In *International Conference on Machine Learning*, pages 5626–5635, 2018.