Evaluating Achievable Latency and Cost: SSD Latency Predictors

Olivia Weng and Andrew A. Chien

The University of Chicago, Chicago, IL 60637, USA
{oliviaweng, aachien}@uchicago.edu

Abstract. Cutting tail latencies at the millisecond level in internet services for good response times in data-parallel applications is possible by integrating MittOS, an OS/data center interface. Typically, MittOS analyzes white-box information of the internals of devices such as SSD's and decides if a given server can "fast reject" a service request. But commercial SSD's have a black-box design, so MittOS researchers have developed machine learning models to determine if requests to commercial SSD's can be rejected or not. When run on CPUs, however, these models cannot predict in the time it takes an SSD to fully process a request, defeating MittOS's fast-rejecting abilities. We demonstrate that ASICs such as the Efficient Inference Engine (EIE) accelerate the prediction times of these MittOS models well within the time it takes an SSD to complete a request at minimal cost, cutting SSD tail latencies. EIE achieves 2.01 μs inference latency while incurring minimal area costs (20.4 mm^2) and power costs (0.29 W). We show that integrating machine learning into the critical path of operating systems becomes cost-efficient and within reason.

Keywords: Model Compression · Hardware Acceleration · Scalability · Tail-tolerance.

1 Introduction

The problem of stragglers or "tail" is a widely studied problem in internet services that use parallelism to achieve low latency [5]; tail-tolerance is considered fundamental to providing good response for applications such as search, feeds, ads, and more that use large-scale parallelism to process large data sets in service of a single service request. MittOS proposes an OS/data center interface that frames a latency threshold with each request, giving each of the parallel servers an opportunity to "fast reject" the request. If the parallel servers can make these decisions well, MittOS can effectively reduce service tail latencies at the millisecond level in disks, SSD's, and the OS cache [11].

Tail latency refers to the trailing "tail" found in latency CDF graphs (around the 95th- or 99th-percentile latency) that is often induced by rare high-latency incidents. As cloud environments and resource sharing scale out, resource contention has become a major reason for such incidents [5]. While infrequent, when

coupled with high degrees of parallelism, these high-latency incidents will dominate performance, producing unacceptable latencies. "The tail latency problem" slows down overall system performance and amounts to bad or unreliable service—ultimately to the detriment of service users.

While prior work has successfully cut tail latencies with time-savings on the order of tens or hundreds of seconds [6], techniques for further reducing tail latencies on the millisecond level have been less effective. Attempts have doubled IO workload through cloning requests [2] or have had trouble handling requests with bursty noise [13]. MittOS [11] tackles these issues at the millisecond level through a new OS interface that is aware of application service-level objectives (SLOs). Given SLOs for IO requests to hard disks, the OS cache, and SSD's, MittOS analyzes the internals of these devices to predict if the SLOs will be met; if not, then MittOS will immediately reject these requests, returning `EBUSY` to the application. Fast prediction allows for fast rejection so that the application can stop waiting and retry its request on another node that is less busy. As a result, MittOS can reduce overall state-of-the-art [5] HDD IO request latencies at scale by 23%.

Most of the work to date with MittOS assumes "white-box" performance models for server latency; that is, these devices are assumed to expose their internal complexities, e.g., fullness of IO queues, sources of variation in latency, etc. However, this makes a common IO element, the commercial SSD, a challenge to incorporate in a MittOS system. Commercial SSD's generally employ a black-box design because the algorithms and structures within play a critical part in their competitive performance and cost. For such devices, we cannot build a "fast reject" responder based on knowledge of the internals, and this leaves MittOS with no way to predict whether a given SSD request's SLO deadline can be met.

Recently, MittOS researchers have explored the use of supervised machine learning to build black-box models for closed SSD's—and more generally subsystems of all kinds without explicit modelling. Their work produced a set of deep neural network models that are studied in this report. Within MittOS, these models can be used to predict which requests will have long latency, and thus provide the ability to "fast reject" for the black-box SSD's [7]. The elegance of the approach is that the models can be trained with a commercial SSD (for which we have no implementation information), using its performance to create labeled data. Others have reported model accuracy results [7], but here we focus on evaluating the viability of the DNN's as a system component in MittOS, studying prediction latency and cost (in power and silicon).

Because SSD's have much lower latency ($\sim$50 microseconds) when compared to traditional HDD's (5+ milliseconds) and even data center network latencies (100's of microseconds when loaded), predictors for systems built on SSD's must make decisions much more quickly (Figure 1). SSD's also support much higher IOPS rates than HDD's, so on a per-device basis, prediction (or "fast reject" decisions) must be made at a much higher rate. Together, these make prediction latency and cost critical factors. Prediction is useful for tail-tolerance only if it

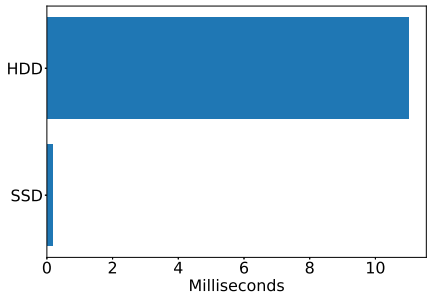


Fig. 1: Network RPC + device latency

can be done in approximately the latency for an SSD to serve an IO request, which is on the order of μs (a 4KB read takes 100 μs).

Table 1: Summary of the MittOS Team's Models

Model	Topology	# Layers	Max Kernel Size	# Weights	# Operations
precise_linear	Linear, fully-connected	8	352x352	583,520	1,167,040
good_custom	"storage-aware"	12	128x128	149,776	365,472
RAID5_aware	"storage-aware"	12	352x352	1,115,490	2,727,652

The MittOS team has produced a number of models (Table 1). One example is the precise_linear model (Figure 5a), which is a small linear neural network that achieves 99.94% accuracy. A second model from the team is the good_custom model (Figure 5b), which is a smaller but more complex model whose network topology is "storage-aware" in that it mimics how storage devices work, achieving 97.94% accuracy. These accuracy levels are high enough to be useful in a scalable MittOS system.

In this paper, we study achievable latencies and costs for running two inference models: precise_linear and good_custom. As discussed above, in order to be used in a MittOS system using SSD's, prediction latencies must be below $\sim$50 μs. Moreover, the cost dynamics of the storage industry require that the power and circuit costs of implementing these models be small on a per-SSD basis. To meet these stringent goals, we employ pruning to reduce the compute requirements of the models [10, 14]. We also employ a hardware accelerator, the Efficient Inference Engine (EIE) [9], an ASIC optimized to run compressed model inference.

Specific contributions of the paper include:

1. Accurate IO latency prediction models (precise_linear and good_custom) can be pruned to dramatically reduce their size and computation costs with little

reduction in accuracy. The costs for precise_linear and good_custom were reduced by 98% and 95% respectively while incurring accuracy reductions of only 2% and 6% respectively.
2. Using a hardware accelerator, the precise_linear DNN model achieves inference latencies of 25 μs and as low as 2.5 μs for the pruned network. On the same accelerator, the smaller good_custom model achieves 15 μs for the original, and below 2 μs latency for the pruned network. These latencies are comfortably lower than the 50 μs requirement for use in a MittOS system for SSD's.
3. With the same hardware accelerator, precise_linear requires less than 2/3 watts for a high inference rate (400,000/second), fast enough to support IO rates for the fastest SSD's. If the accelerator were implemented with a modern CMOS process, this requirement could be reduced to 0.1 watts—easily within the power limits of an SSD.

Overall, we conclude that both the specific DNN models and the general approach can achieve latencies and power low enough for use in SSD-based IO systems prediction for MittOS.

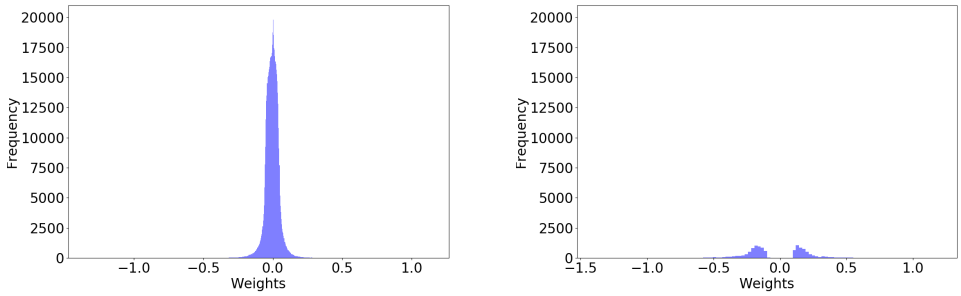
The rest of the paper is organized as follows: Section 2 describes how we compressed our models; Section 3 details the resulting inference latency and power consumption of the compressed models on the EIE and a CPU; Section 4 examines related work; and Section 5 summarizes our findings and discusses future directions of research.

2 Reducing Model Size

The precise_linear and good_custom models parameterize the logical block address (LBA), which are encoded in 8-bit binary format, of the current IO request and the LBAs of pending IO requests. The precise_linear model (Figure 5a) is a linear neural network that has eight fully-connected layers with ReLU activation functions. The whole network has 583k weights. The good_custom model (Figure 5b) is more complex because it features a "storage-aware" design. Its network topology mimics the structure of SSD storage partitioning and contention logic. This model has 12 dense layers with linear activation functions and additional Addition, Subtraction, and ReLU activation operators connecting the layers together. The kernel sizes are smaller than precise_linear's, so overall the model only has 150k weights.

2.1 The precise_linear model

To prune the precise_linear model, we used TensorFlow's Keras-based weight pruning API, which gradually removes unnecessary low-weight connections and then retrains the newly pruned model, repeating this process until a desired sparsity is achieved. Based on [14], the API looks for small weights because they have been shown to have little to no impact on inference when compared to



(a) Before pruning (b) After pruning to 98% sparsity

Fig. 2: Weight distributions of the precise_linear model

larger weights. When looking at the weight distribution of precise_linear (Figure 2a), we see that our model is made up of mostly such small weights, making it particularly amenable to pruning.

We pruned the model aggressively, removing 90%, 95%, 97%, and 98% of the connections to maximize size reduction. We stopped at 98% because further pruning caused the model to become disconnected, destroying its accuracy precipitously. The weight distribution of the 98% sparse pruned model in Figure 2b shows how low-magnitude weights are gradually removed.

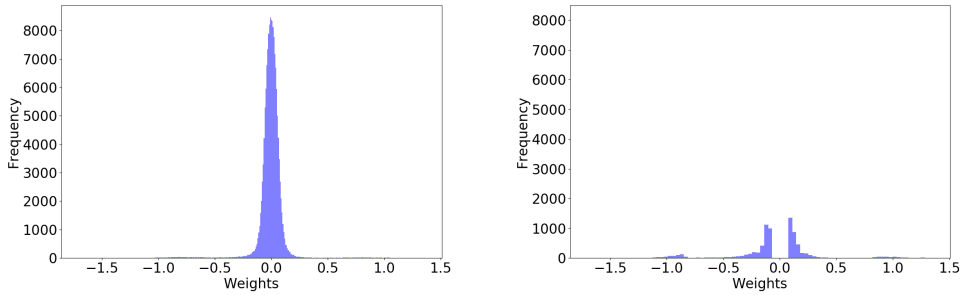
It further follows from these weight distributions that low-magnitude weights are, in fact, low-impact because the model does not suffer much loss in accuracy. In Table 2, we can see that even the largest loss in accuracy, which came from pruning to 98% sparsity, is only 1.91%.

Table 2: Pruned precise_linear model accuracies

Sparsity (%)	Accuracy (%)
0	99.94
90	99.87 (0.07 loss)
95	99.61 (0.33 loss)
97	98.82 (1.12 loss)
98	98.03 (1.91 loss)

2.2 The good_custom model

We used a similar method to prune the good_custom model. However, since the good_custom model is so small (∼150k weights), it was only pruned to 90% and 95% sparsity, as again more aggressive pruning would remove all of the connections between layers, destroying the model. In Figure 3, we can see the distribution of weights before and after pruning.



(a) Before pruning (b) After pruning to 95% sparsity

Fig. 3: Weight distributions of the good_custom model

However, pruning caused significant problems with good_custom. The resulting accuracy of pruned models was unstable and predicted every request as `NON-EBUSY`, effectively failing to classify requests correctly. All of the pruned versions started retraining at 92% accuracy and failed to improve. We believe this is because the original model was under-trained and not robust enough to undergo such aggressive pruning [14]. After pruning, we found that both 90% and 95% sparse models exhibited 92.18% accuracy, but this may not be a good representation of these pruned models' accuracy. Of the 100,000 requests predicted in our testing, 92,178 should be and were indeed predicted to be `NON-EBUSY`; however, the remaining 7,822 requests which should have been predicted to be `EBUSY` were all inaccurately predicted to be `NON-EBUSY`. As such, these accuracy results do not meaningfully reflect how well the pruned good_custom models predict signals since they simply infer all requests to be `NON-EBUSY`.

3 Latency and Power

The EIE is an ASIC specially designed to run models compressed via Deep Compression [10], which involves a combination of pruning, quantization and Huffman Encoding. Applying Deep Compression on neural networks creates models that exhibit irregular computation because of sparse matrix-vector multiplication, layers, and activations. The EIE exploits the shapes of these models, as its custom design specializes in handling irregular sparsity and weight sharing, reducing redundant computation and storage. For instance, recognizing that sparse vectors are full of zeros, EIE uses a compressed sparse column (CSC) format that seeks to store only non-zero values—the only values that have impact on inference.

EIE is made up of a collection of processing elements (PEs). During execution, each PE is given a portion of a matrix to multiply with a given vector. These PEs are controlled by a Central Control Unit. In Figure 4, we see the logical layout of a single PE, which has specialized units designed to handle

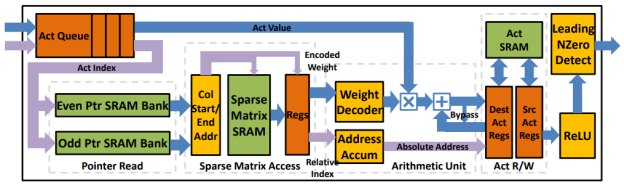


Fig. 4: Logical view of an EIE Processing Element [9]

CSC-formatted vectors and sparse calculations. For example, the Leading Non-Zero Detect unit manages decoding CSC-formatted vectors, which are especially useful given the activation sparsity often found in ReLU computation. Having an array of PEs exploits the parallelism found in many models, further accelerating computation. As such, EIE is well-suited to exploit the irregularities found in a compressed model, as it attempts to maximize the throughput of the computation of non-zero values while minimizing or, if possible, eliminating PE compute power wasted on processing zeros.

Having pruned our models, we looked into how they would perform on the EIE as well as a CPU. Performance on the EIE was calculated via estimating based on the EIE's reported peak throughput (1.6 GOPS) and examination of the sparsity of the weight matrices to account for workload GOPs and load imbalance. Our performance calculations assume the worst case sparsity for input and activation vectors (i.e., all vector elements are nonzero), and therefore provide upper-bounds on throughput and latency. The EIE utilizes its resources efficiently, so generally inference is compute limited. In this model, the remaining benefit of batching would come from dynamic sparsity of the input and activation vectors, which cannot be estimated, and appears to provide no increase. Likewise, the latency impact of batching would be a simple linear increase. The EIE scales near-linearly up until it hits significant load imbalance, when on average $\leq$1 rows are allocated per PE at a given time. According to [9], at this level of load imbalance, the EIE suffers from high variation in distributing the rows to the PEs, as many PEs remain idle. Area and power estimates are based on the EIE's reported area and power consumption per PE and any additional components (e.g., the Leading Non-Zero Detect unit).

The weight matrices of the precise_linear model are small (352 rows max), and performance drops around 16 and 32 PEs because increased sparsity and fewer weights to distribute among the PEs incur even more load imbalance. In spite of this, we can see that we achieve $<$10 μs latency easily with the EIE. As seen in Tables 3 and 5, at 98% sparsity, which has comparable accuracy as seen previously, the EIE can achieve a mere 2.01 μs/inference (Table 3) latency with 32 PEs while only incurring an area footprint of 20.4 mm^2 and dissipating 0.29 W (Table 5), which is significantly low-cost compared to allocating an entire CPU for inference. Because of load imbalance, however, it would be more cost-effective to use 8 PEs, which achieves 2.45 μs/inference for our 98% sparse model (Table

3). This way we have fewer idle PEs at a given time while further reducing area footprint to 5.10 mm^2 and power dissipation to 0.073 W (Table 5).

Table 3: Average latency of precise_linear models (μs/inference)

Sparsity (%)	CPU (i7-8700)	EIE (1PE)	EIE (2PEs)	EIE (4PEs)	EIE (8PEs)	EIE (16PEs)	EIE (32PEs)
90	14.1	81.1	40.5	20.3	10.1	5.07	2.53
95	13.0	40.7	20.3	10.2	5.08	2.54	2.31
97	12.4	24.7	12.4	6.18	3.09	2.81	2.55
98	13.4	17.3	8.63	4.32	2.45	2.23	2.01

Table 4: Average latency of good_custom models (μs/inference)

Sparsity (%)	CPU (i7-8700)	EIE (1PE)	EIE (2PEs)	EIE (4PEs)	EIE (8PEs)	EIE (16PEs)	EIE (32PEs)	EIE (64PEs)
0	7.49	251	126	62.8	31.4	15.7	7.85	3.93
90	7.08	25.5	12.7	6.37	3.19	1.99		
95	7.36	13.3	6.65	3.32	2.08	1.89		

Table 5: Peak throughput, area, and power of EIE

	CPU (i7-8700)	EIE (1PE)	EIE (2PEs)	EIE (4PEs)	EIE (8PEs)	EIE (16PEs)	EIE (32PEs)	EIE (64PEs)
Peak Throughput (GOPS)	82.8	1.6	3.2	6.4	12.8	25.6	51.2	102.4
Area (mm^2)	1406.25	0.638	1.28	2.56	5.10	10.2	20.4	40.8
Power (W)	65	0.0092	0.018	0.037	0.073	0.15	0.29	0.59

While the runtime latencies of the pruned good_custom models seem promising, their unstable accuracies give us pause. However, since the original, unpruned good_custom model is so small—its weight matrices are only 128 rows max—the EIE does not end up suffering from the load balance issues that the sparse ones encounter. As such, its runtime on 64 PEs (Table 4) is comparable to the runtime of the pruned precise_linear models. We did not calculate the runtimes for 32 and 64 PEs for the pruned sparse good_custom models, because at those points the load imbalance is so severe that the EIE becomes extremely volatile and the runtimes become unclear—perhaps running the EIE simulator on these settings may reveal these runtimes.

4 Related work

Model compression and pruning are widely-studied techniques to reduce size and computational cost. These techniques exploit the plentiful redundancy in trained

deep neural networks. Optimal Brain Damage [12] used pruning to reduce the number of connections to simplify network complexity. Magnitude-based pruning [14] during training further showed how large deep learning models—e.g., deep CNNs—can be pruned to 10x smaller size with minimal loss in accuracy. Deep Compression [10] builds on these methods, as discussed previously, and employs a combination of pruning, quantization and Huffman encoding to compress DNNs such as AlexNet by 35x. Our work takes advantage of these advances, exploiting pruning to reduce latency and power costs of DNN's for IO latency prediction.

Hardware accelerators for DNN's are widely available; incorporated in major mobile phone platforms from Apple, Samsung, Qualcomm, and Huawei. These accelerators typically support Int16 operations and rely on effective model compression techniques to achieve acceptable latency and power (e.g. Edge TensorFlow processing unit (TPU) [1]). The Edge TPU is a small $\sim$50mm^2 piece of silicon, and with support for 300x300 matrices, is designed for smaller networks. It computes exclusively on smaller, cheaper datatypes (Int8, Int16), aiming to accelerate quantized networks. Other accelerators include the EIE (which we studied), the Efficient Speech Recognition Engine (ESE) [8] designed to to accelerate sparse LSTM models used for speech recognition, and the Eyeriss systems [3, 4]. Any of these systems could be suitable for accelerating our IO latency predicting modules.

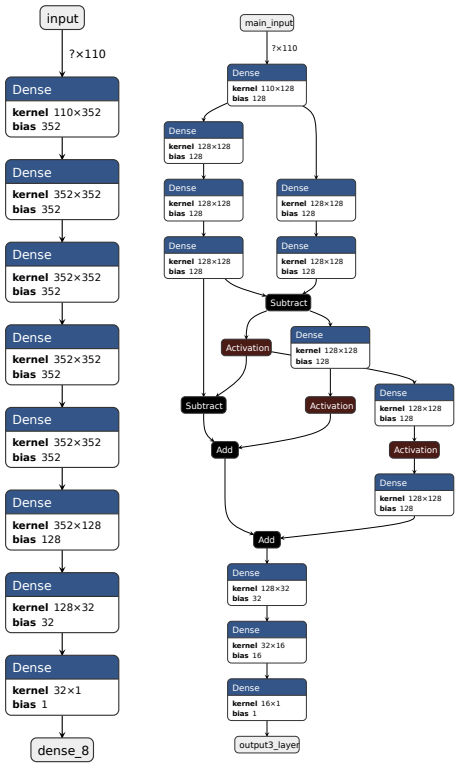
Our work builds on MittOS [11], an OS interface that cuts tail latencies at the millisecond level, extending it by replacing the explicit white-box models with a trained DNN. Specifically, we show that the trained DNN can be a practical substitute in MittOS systems that manage tail-latency IO systems built for SSD's, which operate at latencies of a few hundred seconds. The broader literature on tail-tolerance includes hedging [5], tied requests [5], snitching [13], and many more. However, we are not aware of other approaches within the context of MittOS that employ trained DNN's, and further none that explore the latency, area, and power costs to make such application viable in SSD-based IO-systems.

5 Summary and Future Work

Since the EIE can achieve such low latency with little area usage and power consumption, these MittOS neural networks become useful and effective tools in predicting SSD IO requests, bringing us one step closer to integrating MittOS into operating systems. It becomes unreasonable to assume that we must rely on the CPU to support operating systems, as introducing machine learning via ASICs into the critical path of operating systems becomes a reality. With 32 PEs, EIE can predict our 98% sparse precise_linear model in 2.01 μs, and with 64 PEs, EIE can predict the unpruned good_custom model in 3.93 μs. Thus, our models achieve latencies well below the $<$10 μs latency of even the fastest SSD reads and writes. As such, by means of ASICs like the EIE, introducing our MittOS models into the CPU or near the SSD controller to serve SSD requests, cutting their millisecond tail latencies, becomes practicable.

Not only is pruning an effective method of compression, quantization has been shown to reduce model size while maintaining accuracy [10]. Although EIE can be computed in float32 with little to no change in area and power (SRAM accesses dominate), it aims to quantize weights and compute in float16, so it would be worth exploring how robust precise_linear is in float16. It is also worth testing how precise_linear and good_custom would run on the EIE simulator to get a clearer idea of how much the load imbalance at a higher number of PEs affects performance. While our performance and power model for the EIE does not allow analysis of some potential benefits of batching, such study would be an interesting direction for future work.

6 Appendix



(a) The precise_linear model, with input 1x110

(b) The good_custom model, with input 1x110

Fig. 5: The MittOS models

References

1. Edge tpu. https://cloud.google.com/edge-tpu/, accessed: 2019-11-21
2. Ananthanarayanan, G., Ghodsi, A., Shenker, S., Stoica, I.: Effective straggler mitigation: Attack of the clones. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. pp. 185–198. nsdi'13, USENIX Association, Berkeley, CA, USA (2013), http://dl.acm.org/citation.cfm?id=2482626.2482645
3. Chen, Y., Emer, J.S., Sze, V.: Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices. arXiv:1807.07928 [cs.CV] (2019)

4. Chen, Y., Krishna, T., Emer, J.S., Sze, V.: Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits **52**(1), 127–138 (2016)
5. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM **56**(2), 74–80 (Feb 2013). https://doi.org/10.1145/2408776.2408794
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. pp. 137–150. San Francisco, CA (2004)
7. Han, M., Zhang, A., Roaffa, A., Chien, A.A., Hoffman, H., Gunawi, H.S.: Machine learning for operating systems: A case of using deep neural network for os-level i/o latency prediction. In: 13th USENIX Symposium on Operating Systems Design and Implementation (2018), (Poster)
8. Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., Yang, H., Dally, W.B.J.: Ese: Efficient speech recognition engine with sparse lstm on fpga. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. pp. 75–84. FPGA '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3020078.3021745, http://doi.acm.org/10.1145/3020078.3021745
9. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: Eie: Efficient inference engine on compressed deep neural network. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA) (2016). https://doi.org/10.1109/isca.2016.30
10. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv:1510.00149 [cs.CV] (2015)
11. Hao, M., Li, H., Tong, M.H., Pakha, C., Suminto, R.O., Stuardo, C.A., Chien, A.A., Gunawi, H.S.: Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. Proceedings of the 26th Symposium on Operating Systems Principles - SOSP 17 (2017). https://doi.org/10.1145/3132747.3132774
12. LeCun, Y., Denker, J.S., Solla, S.A., Howard, R.E., Jackel, L.D.: Optimal brain damage. Advances in neural information processing systems **2** (1989)
13. Suresh, L., Canini, M., Schmid, S., Feldmann, A.: C3: Cutting tail latency in cloud data stores via adaptive replica selection. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 513–527. USENIX Association, Oakland, CA (May 2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh
14. Zhu, M., Gupta, S.: To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv:1710.01878 [stat.ML] (2017)