Asynchronous Adam

Alex Renda

1. Abstract

In this paper, we analyze the convergence rates of variants on asynchronous versions of Adam. In addition to the shared parameter vector, Adam requires two additional vectors estimating the first and second moments of the gradient. Unlike the parameters, however, moments are estimated by an exponential moving average, complicating their parallelization. We analyze both a version of Adam in which all workers update shared moment estimation vectors, and a version in which each worker has its own (identical in expectation). Experimental results suggest that both versions converge, but that shared moment vectors converge faster; in the following sections, we analyze their rates, and suggest practical methods for getting a faster wall-clock convergence rate than serial Adam.

2. Background

There are countless variants of stochastic gradient descent (SGD) for first-order function optimization, each with various tradeoffs in terms of speed, memory requirements, convergence rate, etc. Due to their stochastic nature, almost all of these algorithms have some inherent amount of variance or error, while still provably converging in expectation. Several techniques exist to exploit this error tolerance, adding some amount of additional noise (and thereby reducing the theoretical rate) in exchange for increasing throughput and energy efficiency, or decreasing wall-clock time. Such techniques include low precision computation (Gupta et al., 2015) and lockless parallelization (Niu et al., 2011). (De Sa et al., 2015) proved that both of these techniques still allow for convergence on naïve SGD, and provided a framework for analyzing the convergence rate of other parellel stochastic optimization algorithms. This result has also been shown for a parallel version of SVRG (J. Reddi et al., 2015). Further, it has been shown (Mitliagkas et al., 2016) that asynchrony has a connection to momentum, which is known to both increase converge rate while also reducing test error in the nonconvex case. However, asynchronous convergence has not yet been shown for Adam (Kingma & Ba, 2015), a popular stochastic optimization algorithm that accelerates convergence by estimating moments of the gradient distribution. Given that Adam has been shown to achieve worse test error than even SGD with momentum (Wilson et al., 2017), it would be useful to parallelize Adam while maintaining its convergence, to get a fast optimizer with good test error.

3. Algorithms

We experimented with two parallel algorithms, both modifications of the original Adam algorithm presented in (Kingma & Ba, 2015), based on the parallelization approach in (Niu et al., 2011). Due to the nature of estimating the moment vectors, both of these algorithms have another potential source of conflicts among worker updates: the calculating of the moment vectors. The HOGWILD! algorithm assumes that all workers share a singular parameter vector, which is an assumption we will keep (future work could see if this assumption could be modified too). However, we propose two different algorithms for parallelizing Adam: Shared (Algorithm 1), and Private (Algorithm 2).

Moment ownership The key distinction between the two algorithms is the ownership of the moment estimation vectors. In Shared, the moment estimation vectors are shared between each worker, meaning there is potential for conflicts between updates, not only to the parameter vector but also to the moment vectors. In Private, the moment estimation vectors are private to each worker, meaning that each worker's moment estimate will be less accurate than theoretically possible, given the gradient samples the algorithm has seen so far, but will be free from conflicting updates. Intuitively, we expect the Shared algorithm to perform better in the sparse case, where conflicting updates are less likely, and the Private algorithm to perform better in the dense case.

Exponential moving average Unfortunately, this bifurcation of algorithms is not yet enough to address all of the issues with parallelizing Adam for sparse problems. This is due to the exponential moving averages in m and v. Since the exponential moving average requires decaying our old estimation, and in the sparse condition we rarely get new estimates, we effectively lose information by decaying all values every iteration. One potential solution to this takes advantage of the sparsity assumption, namely that each individual gradient will be sparse. We therefore slightly redefine our exponential moving average to be the moving

Algorithm 1 Shared Async Adam

Require: α_0: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
Require: $f(\theta)$: Stochastic objective function
Require: θ_0: Initial parameter vector
Require: T: Number of threads
Require: i: Thread index
Init Shared: $\theta \leftarrow \theta_0$
Init Shared: $m \leftarrow 0$
Init Shared: $v \leftarrow 0$
Init Private: $t_i \leftarrow 0$
while θ not converged **do**
  $t = t_i \cdot T + \texttt{randi}(0, T)$
  $g = \nabla_t f_t(\theta)$
  $\alpha = \alpha_0 \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$
  Update Private: $t_i \leftarrow t_i + 1$
  Sparse Update Shared:
    $\forall g_j \neq 0,\ m_j \leftarrow \beta_1 \cdot m_j + (1 - \beta_1) \cdot g_j$
  Sparse Update Shared:
    $\forall g_j \neq 0,\ v_j \leftarrow \beta_2 \cdot v_j + (1 - \beta_2) \cdot g_j^2$
  Sparse Update Shared:
    $\forall g_j \neq 0,\ \theta_j \leftarrow \theta_j - \alpha \cdot m_j/(\sqrt{v_j} + \epsilon)$
end while

Algorithm 2 Private Async Adam

Require: α_0: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
Require: $f(\theta)$: Stochastic objective function
Require: θ_0: Initial parameter vector
Require: T: Number of threads
Require: i: Thread index
Init Shared: $\theta \leftarrow \theta_0$
Init Private: $m_i \leftarrow 0$
Init Private: $v_i \leftarrow 0$
Init Private: $t_i \leftarrow 0$
while θ not converged **do**
  Update Private: $t_i \leftarrow t_i + 1$
  $g = \nabla_t f_t(\theta)$
  $\alpha = \alpha_0 \cdot \sqrt{1 - \beta_2^{t_i}}/(1 - \beta_1^{t_i})$
  Sparse Update Private:
    $\forall g_j \neq 0,\ m_{i,j} \leftarrow \beta_1 \cdot m_{i,j} + (1 - \beta_1) \cdot g_j$
  Sparse Update Private:
    $\forall g_j \neq 0,\ v_{i,j} \leftarrow \beta_2 \cdot v_{i,j} + (1 - \beta_2) \cdot g_j^2$
  Sparse Update Shared:
    $\forall g_j \neq 0,\ \theta_j \leftarrow \theta_j - \alpha \cdot m_{i,j}/(\sqrt{v_{i,j}} + \epsilon)$
end while

average over *non-zero samples*, and only update the corresponding entry in the parameter vector upon finding such a non-zero sample. It is possible, although unconfirmed in this work, that the two behaviors lead to effectively equivalent results — commonly used implementations of serial Adam (Abadi et al., 2013) operate under the dense assumption even in sparse conditions.

Shared time conflict In our implementation of Shared, we found frequent conflicts to the shared t variable in the original Adam algorithm. These conflicts caused highly undesireable and unpredictable behavior, especially in the early iterations of the algorithm. Using C++'s std :: atomic caused an unacceptable slowdown. As such, we chose to use the random behavior shown above, which should give the same result. We believe that there is a better solution to this issue (std :: atomic integers should be lockless, so the observed slowdown does not make sense), but in the interest of time, we chose to use the described implementation.

4. Experiments

We experimented with our parallel implementations of Adam, comparing them to each other, and to other stochastic serial algorithms. We compare Shared and Private Adam against serial Adam, SVRG (Reddi et al., 2015), and simple SGD.

4.1. Setup

We ran our experiments on a single node of a cluster, with a Intel Xeon CPU E5-2620 v3 @ 2.40GHz. This CPU has 2 sockets, 6 cores per socket, and 2 threads per core, leading to a maximum of 24 threads. When not specified, our experiments use 12 threads, one per physical core. Our entire dataset is roughly 102MB, larger than all caches but capable of fitting in main memory. We use minibatches of size 16, which fit in the L2 cache. The parameter vector comfortably fits in the L1d cache.

We did not experiment on GPUs, which could change the outcome due to an apparent lack of sequential consistency guarantees on GPUs (Braibant, 2013). This could cause interesting behavior, since the HOGWILD! model assumes sequential consistency. We instead leave these experiments for future work.

4.2. Sparse Logistic Regression

For our first experiment, we intentionally chose a problem that would be relatively hard, but that a parallel version of Adam should excel at. Specifically, we tested our algorithms on an L2 regularized ($\lambda = 0.002$) binary logistic regression problem, with sparsely selected data points. This is a sparse, convex problem, which should be relatively easy for both serial and parallel Adam. For our data, we randomly generated $n = 10,000$ data points with 4096

dimensions each. Each feature in each datapoint was 0 with probability 0.99, and otherwise drawn from $\mathcal{N}(0, 1)$. These points were labeled according to a dense randomly generated hyperplane (also drawn from $\mathcal{N}(0, 1)$), with

$$p(y_i = 1|x_i) = \frac{1}{1 + \exp -w^T x_i}$$

The test dataset was identically generated, and labeled according to the same hyperplane.

Hyperparameters A large amount of time was dedicated to choosing ideal but fair hyperparameters for our experiments. This is hugely important to ensure that comparisons between algorithms with different convergence rates and different amounts of noise are fair, as clearly running them all with the same step size does not give an accurate comparison of their tradeoffs. However, we also wanted to make sure not to overtune: a component of the tradeoff of each algorithm is their complexity to tune, and manually fiddling with hyperparameters until a desired result is achieved is disingenuous at best.

To balance these competing objectives, we chose to run a grid search with the same number of choices over the hyperparameter space for each algorithm. We chose to search over $\alpha \in \{0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0, 10.0\}$ for the stepsize, and $\beta_1 \in \{0.9, 0.99, 0.999\}$ and $\beta_2 \in \{0.9, 0.99, 0.999\}$ for Adam. We consistently found that for Adam, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ worked best across all of our experiments, so we elide discussion of those parameters.

4.3. Results

The results of our comparisons between the two algorithms and their serial counterpart can be seen in Figure 1.

Wall clock time On this problem, we were not able to achieve significantly faster convergence in wall-clock time than serial Adam. The initial descent for both Shared and Private is faster than that of the serial version, with the Shared being marginally faster than Private. Both Shared and Private bottom out earlier than serial though, suggesting that their additional noise makes them converge to a larger noise ball around the optimum.

Iteration count In both parallel algorithms, the actual convergence rate in terms of iteration count is worse — this is true for any step size up to and including that of the serial version (see Figure 2). This is entirely expected: any wall clock time improvement was gained at the cost of additional noise in the algorithm. Given that the convergence rate of the algorithm in part depends on the noise of the gradient samples (Theorem 4.1 in (Kingma & Ba, 2015) contains a term proportional to the standard deviation of the gradient distribution), we therefore expect the convergence rate to decrease as a function of the induced noise. It is in fact somewhat surprising that the convergence rate in terms of iteration count decreased as little as it did (in Figure 2): for a high number of threads, we would expect a fair amount of conflict between workers. This is especially true since we used minibatching, which by its nature will increase the chance of a conflict. With our batch size of 16 and $p(x_i = 0) = 0.99$, the probability of a conflict between two threads at an individual dimension is $(1 - 0.99^{16})^2 \approx 0.022$, which means that at least one conflict between each 4096-dimensional gradient sample is almost guaranteed. However, we believe that this same effect may at the same time diminish the additional noise. Batching naturally reduces the variance of the gradient samples, which will in turn reduce the variance of their moment estimates. As our batch size approaches n, we would expect to see the variance between moment estimates go to zero, and as such, significantly reduce the impact of any conflicts.

Test error As predicted, the test error is also somewhat better than expected, considering the gap in training loss (especially noticeable in the wall clock time plots). While our results unfortunately do not show a lower testing error, this is also somewhat expected since this is a convex problem in which the training and testing points are drawn from the exact same distribution. It would be interesting future work to experiment with this on a nonconvex problem. The "asynchrony begets momentum" argument implies a lower testing error in the nonconvex case, as do these results. However, without a more rigorous experiment, we cannot yet fully reach that conclusion to resolve the issues referenced in (Wilson et al., 2017).

4.3.1. Algorithm Choice

These results seem to imply relatively similar performance for the Shared and Private algorithms on this problem. Shared and Private appear to have roughly identical rates in terms of iteration counts, while Shared appears to converge faster in terms of wall clock time. Given that Shared and Private should take roughly the same wall clock time per iteration, it is somewhat surprising to see Shared's wall clock time being faster than that of Private. We postulate that this is due to cache effects: assuming the caches are per-CPU and not explicitly per thread, the use of different moment vectors could cause conflicts and evictions, whereas a single shared one would not.

Still, these results are along the lines of what we expect to see for a sparse problem, although our originally hypothesis was that Shared would perform even better in this case. It would be interesting to experiment on a dense problem as well, where we would expect Shared to perform worse.

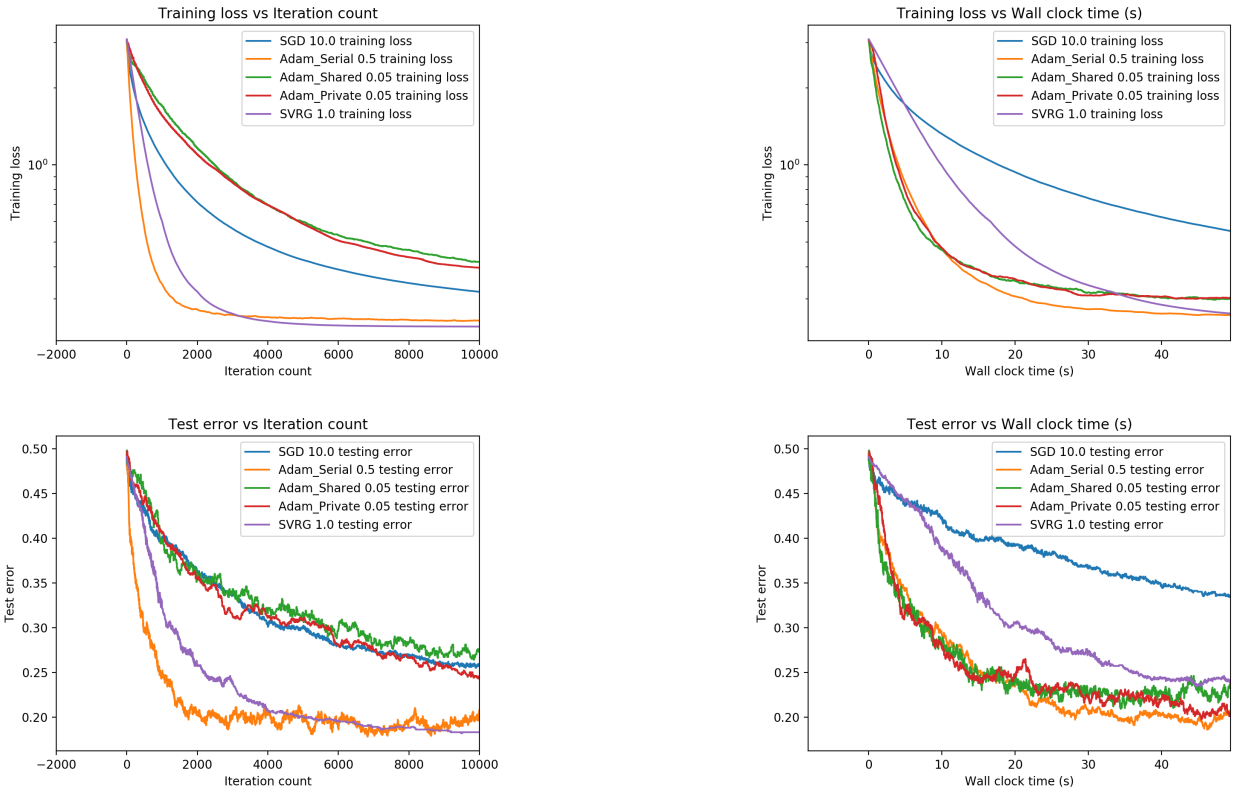

Figure 1. Training loss and test error for different algorithms

Figure 2. Training loss across algorithms for the same step size

Preliminary results actually suggest the opposite, but are not well explored enough yet to include in this report.

Ultimately, we can make no conclusion about a general comparison between these algorithms based on these results. Shared appears to converge marginally faster, potentially due to cache effects, but this may not be a universal truth, especially as this is a sparse problem likely to favor Shared over Private.

4.3.2. Thread Count

4.4. Results

In addition to experimenting between the algorithms, we experimented with different thread counts for the better performing parallelized algorithm, Shared. Across all experiments, we found exactly the expected results: for a given step size below some threshold, increasing the thread count decreases the wall clock time to convergence, while simultaneously decreasing the convergence rate. Increasing the thread count above that threshold has a strictly worse effect, decreasing both the wall-clock and iteration count convergence time.

Identical small step size Our preliminary results in this category, presented in Figure 3, were unduly exciting: we found that for a relatively small step size, increasing the thread count results in a nearly linear increase in convergence rate. This makes total sense: we take smaller steps, meaning that the delay from stale updates is less significant, while also not having any of the additional error from larger α, meaning that the algorithm has more room to ab-

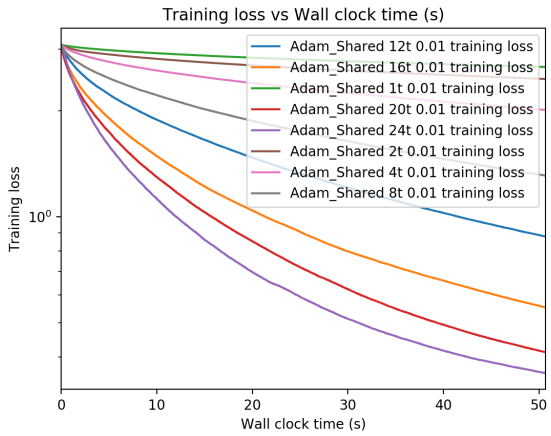

Figure 3. Training loss for the same small step size

sorb error from the parallelism.

Optimal step size Unfortunately, these results do not hold up as nicely when hyperparameter tuning is applied, as shown in Figure 4. In this experiment, we selected the best performing $\alpha \in \{0.01, 0.030.05, 0.075, 0.1, 0.3, 0.5, 0.75, 1\}$ for each thread count, and plotted the results against each other.

Regardless, this result is probably the most important of this paper: we show that for tuned algorithms, there is still a possibility of getting an improvement in some situations. We found that for this problem, the version with 2 threads performed better than the single threaded version with the same step size, at no cost in noise ball size. However, thread count also clearly cannot just be naïvely increased: in this problem, increasing beyond 2 threads actually *hurt* convergence rate, even with hyperparameter optimization.

Identical big step size Finally, for completeness, we ran experiments on the other end of the step size spectrum, increasing the thread count for a constant large value of α. These results can be seen in Figure 5. These results are roughly as expected, where constantly increasing the number of threads actually results in significantly worse performance in wall clock time.

5. Future Work

These preliminary results suggest many possible future directions of exploration to compare the two proposed algorithms against each other and their serial counterpart.

Theoretical results The most pressing area for more work is the theoretical results behind the convergence of asynchronous Adam. Many of the claims in this paper are conjecture based on intuition — however, given that Adam itself has been proved to converge, and that there exists a framework for analyzing the convergence of asynchronous stochastic algorithms (De Sa et al., 2015), the convergence of both Shared and Private seem likely, and seem like tractable proofs. Moreover, theoretical results about these may give insights into which algorithm behaves better in practice, or potential terms to add to accelerate each of their convergence.

Experiments Beyond theoretical results, the conclusions of this paper should be extended with more experiments. HOGWILD! is proven to work on sparse problems, and is known to work well on dense problems anyway. That may be the case with asynchronous Adam, but it is hard to claim that without both theoretical results and more experiments. Specifically, the asynchronous versions of Adam should be tested on:

- Sparser problems, since our analysis above showed that there is a high chance of a conflict on the problem we experimented on
- Dense problems, to better compare the algorithms to each other, and to see if like HOGWILD! we are still able to get speedups
- Nonconvex problems, such as a neural net, to potentially give a counterexample to the results in (Wilson et al., 2017) about Adam test error

Regardless, we do believe that our experimental results are promising, in that at least for sparse strongly convex problems, asynchronous Adam can converge with a faster wall clock time than serial Adam.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mané, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viégas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. Tensorflow. https://github.com/tensorflow/tensorflow/blob/6df
2013.

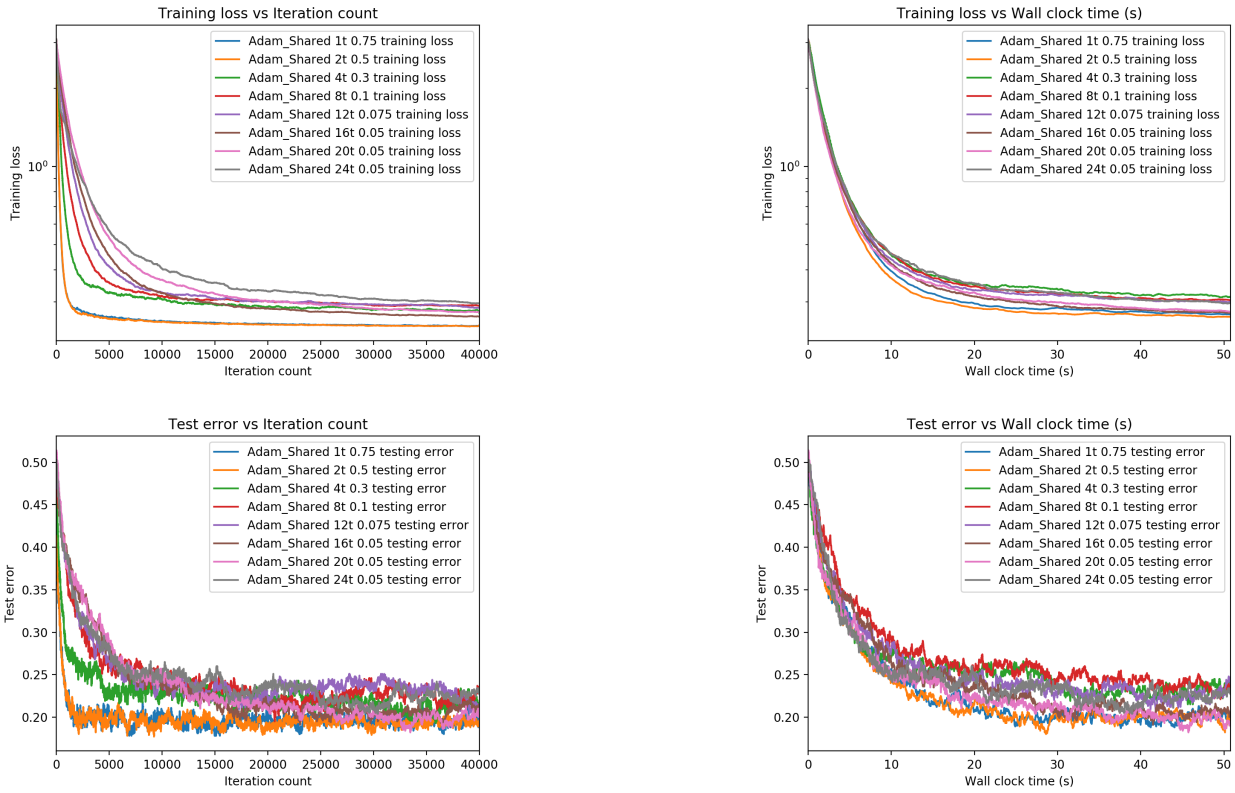

Figure 4. Training loss and test error for optimal step size for different numbers of threads

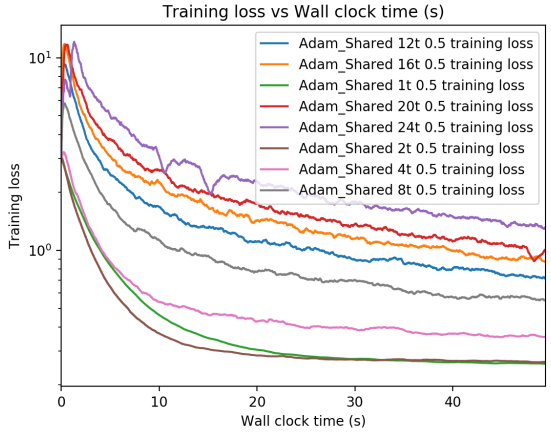

Figure 5. Training loss for the same large step size

Braibant, Thomas. Gpu memory model. http://gallium.inria.fr/blog/gpu_memory_model/, 2013.

De Sa, Christopher, Zhang, Ce, Olukotun, Kunle, and Ré, Christopher. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 2674–2682, 2015. URL http://papers.nips.cc/paper/5717-taming-the-wild-a-unified-analysis-of-hogwil

Gupta, Suyog, Agrawal, Ankur, Gopalakrishnan, Kailash, and Narayanan, Pritish. Deep learning with limited numerical precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 1737–1746. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045303.

J. Reddi, Sashank, Hefny, Ahmed, Sra, Suvrit, Poczos, Barnabas, and Smola, Alexander J. On variance reduction in stochastic gradient descent and its asynchronous variants. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 28*, pp. 2647–2655. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5821-on-variance-reduction-in-stochastic-gradient pdf.

Kingma, Diederik P. and Ba, Jimmy Lei. Adam: A

method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Mitliagkas, Ioannis, Zhang, Ce, Hadjis, Stefan, and Ré, Christopher. Asynchrony begets momentum, with an application to deep learning. In *Allerton*, pp. 997–1004. IEEE, 2016.

Niu, Feng, Recht, Benjamin, Re, Christopher, and Wright, Stephen J. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pp. 693–701, USA, 2011. Curran Associates Inc. ISBN 978-1-61839-599-3. URL http://dl.acm.org/citation.cfm?id=2986459.2986537.

Reddi, Sashank J., Hefny, Ahmed, Sra, Suvrit, Póczos, Barnabás, and Smola, Alexander J. On variance reduction in stochastic gradient descent and its asynchronous variants. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 2647–2655, 2015. URL http://papers.nips.cc/paper/5821-on-variance-reduction-in-stochastic-gradient-descent-and-its-asynchronous-variants.

Wilson, Ashia C., Roelofs, Rebecca, Stern, Mitchell, Srebro, Nathan, and Recht, Benjamin. The marginal value of adaptive gradient methods in machine learning. *CoRR*, abs/1705.08292, 2017. URL http://arxiv.org/abs/1705.08292.