Accelerating deep neural network learning for speech recognition on a cluster of GPUs

Guojing Cong [1], Brian Kingsbury [1], Soumyadip Gosh [1], George Saon [1], Fan Zhou [2]

[1]IBM TJ Watson Research Center
1101 Kitchawan Road, Yorktown Heights, NY, 10598
[2]Georgia Institute of Technology, Atlanta, GA, 30332

November, 2017

Outline

- Introduction – application, data, neural network, sequential performance
- Acceleration strategies through parallelization
- Results
- Conclusion and future work

The application

- train neural networks for acoustic modeling for large-vocabulary continuous speech recognition
- use hybrid hidden Markov model/neural network (HMM/NN) approach
- neural network is trained to classify input feature vectors that summarize the acoustic characteristics of the speech signal over a period of 90–400 ms. into HMM states

The data

- 260-hour Switchboard American English telephone conversational task using the Mississippi State transcripts
- test data is the Switchboard part of the NIST 2000 Hub5 Evaluation data
- Acoustic feature vector, a “frame”, has 140 dimensions
- the training set contains 94M frames, and we use an alphabet of 32,000 context-dependent HMM states

The network

- a windowed, bidirectional long short-term memory (LSTM) architecture, each window has 21 frames
- 4 bidirectional LSTM layers with 512 LSTM units per direction per layer
- 32,000 HMM output states
- a 256-unit linear bottleneck layer between LSTM and output
- 30M parameters

The sequential Implementation

- implemented in Torch with CuDNN library
- runs stochastic gradient descent (SGD) for 14 epochs with initial learning rate $\gamma_1 = 0.025$ and minibatch size $B = 128$
- Validation at end of each epoch, and the average negative log-posterior estimate, or validation loss, is computed.
- achieves validation loss 1.63 and word error rate 10.6%
- learning rate reduced by half when the validation loss does not improve by 1%
- Each epoch takes > 2 hours, and training takes more than 1 day on P100 GPU

Notations

We use $\|\cdot\|_2$ to denote the ℓ_2 norm of a vector in $\mathbb{R}^d$; $\langle\cdot\rangle$ to denote the general inner product in $\mathbb{R}^d$. The following are key parameters of the algorithms.

- P denotes the number of learners;
- K denotes the length of the delay;
- B_n, $\bar{B}$, or B denotes the mini-batch size for the n-th update;
- γ_n, $\bar{\gamma}$, or γ denotes the step size for the n-th update;
- $\xi_{k,s}^{j}$ denotes the i.i.d. realizations of a random variable ξ generated by the algorithm on different processors and in different iterations, especially, $j = 1, ..., N$, $k = 1, ..., K$, and $s = 1, ..., B$.

Assumptions

- $F : \mathbb{R}^d \to \mathbb{R}$ is continuously differentiable, gradient function of F is Lipschitz continuous with Lipschitz constant $L > 0$
- The sequence of iterates $\{\mathbf{w}_j\}$ is contained in an open set over which F is bounded below by a scalar F^*.
- For any fixed parameter $\mathbf{w}$, the stochastic gradient $\nabla F(\mathbf{w}; \xi)$ is an unbiased estimator of the true gradient corresponding to the parameter $\mathbf{w}$
- There exist scalars $M \geq 0$ and $M_V \geq 0$ such that, for all $k \in \mathbb{N}$,

$$\mathbb{E}_\xi \left\| \nabla F(\mathbf{w}; \xi) \right\|_2^2 - \left\| \mathbb{E}_\xi \nabla F(\mathbf{w}; \xi) \right\|_2^2 \leq M + M_V \left\| \nabla F(\mathbf{w}) \right\|_2^2.$$

Acceleration through parallelization

- The cluster we use has 5 IBM Minsky nodes. Each node has 2 Power8 GPUs with 10 cores each, and 4 NVIDIA Tesla P100 GPUs. The interconnect between the nodes is Infiniband.
- The communication is implemented using CUDA-aware openMPI 2.0 through the mpiT library
- ASGD vs. our approach

ASGD

We run two popular ASGD implementations, *Downpour* and *EAMSGD*, for 28 epochs. As *Downpour* does not converge with $\gamma_1 = 0.025$, set $\gamma_1 = 0.01$ instead. Set $\gamma_1 = 0.025$ for *EAMSGD*.

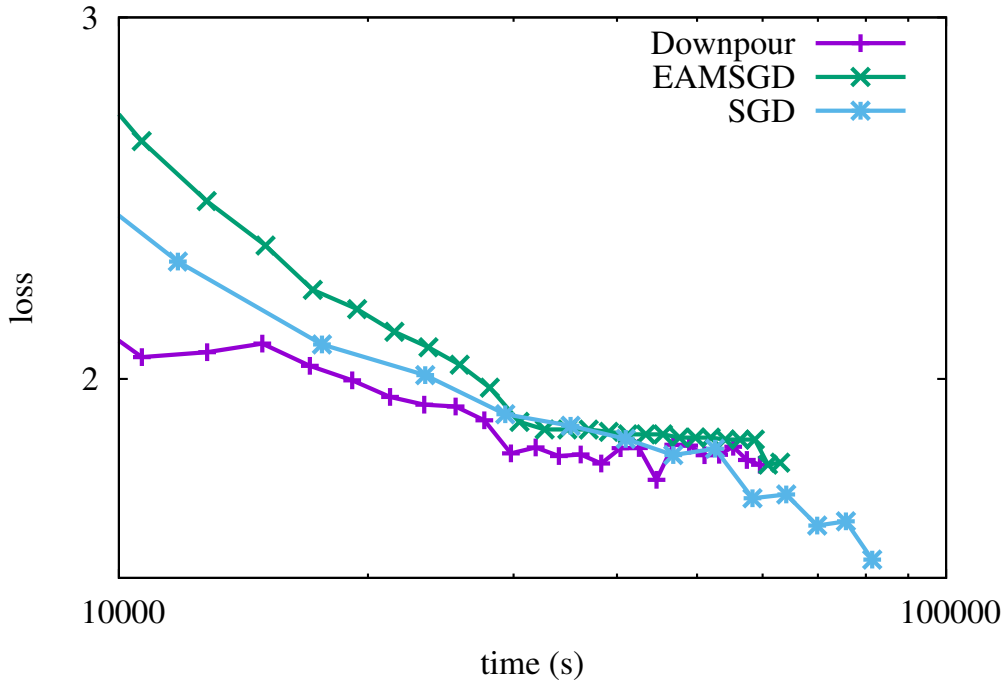

Figure: ASGD methods with 4 GPUs (log – log plot).

ASGD – 20 GPUs

$\gamma_1 = 0.001$ for *Downpour* and $\gamma_1 = 0.01$ for *EAMSGD*. $P = 20$ with 48 epochs. both *EAMSGD* and *Downpour* are about 4.5 times faster than SGD (run for 14 epochs). The loss for *Downpour* is 1.981, while the loss of *EAMSGD* remains above 7.

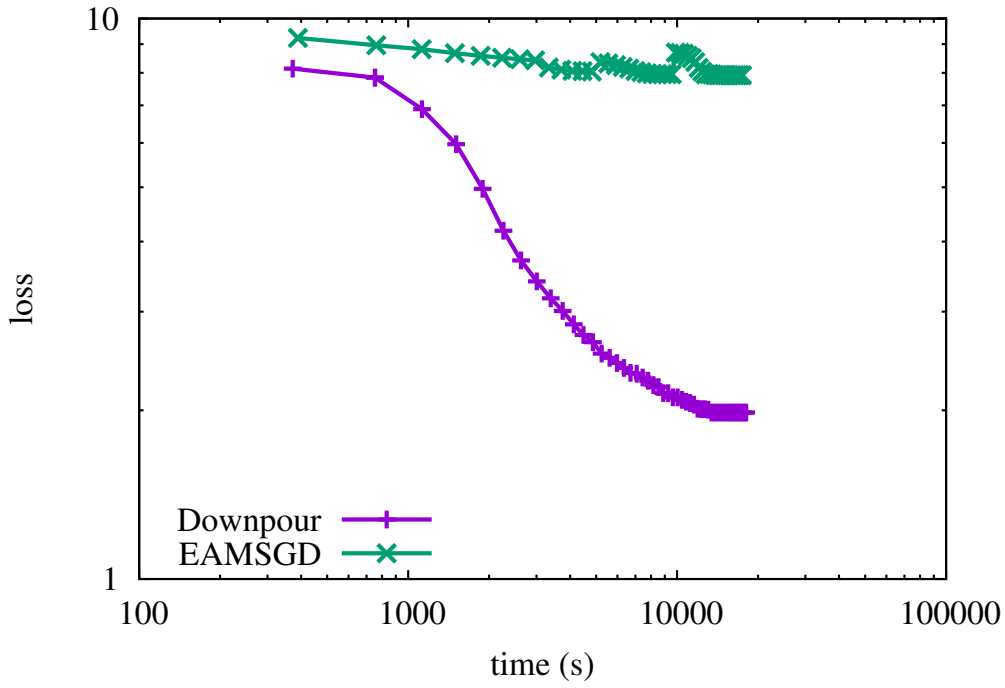

Figure: ASGD methods with 20 GPUs (log – log plot).

Convergence challenge for ASGD

- Scalability of ASGD is determined by the impact of P on its convergence rate guarantee, that is, the average expected squared gradient norm $\mathbb{E}\frac{1}{N}\sum_{n=1}^{N}\left\|\nabla F(\widetilde{\mathbf{w}}_n)\right\|^2$.
- After K updates, the bound on $\mathbb{E}\frac{1}{N}\sum_{n=1}^{N}\left\|\nabla F(\widetilde{\mathbf{w}}_n)\right\|^2$ is

$$\mathbb{E}\frac{1}{N}\sum_{n=1}^{N}\left\|\nabla F(\widetilde{\mathbf{w}}_n)\right\|^2 \quad \leq \quad \left[\frac{C_0(F(\widetilde{\mathbf{w}}_1)-F^*)}{\bar{\gamma}N}+\frac{C_1\bar{\gamma}^2L^2M^2P}{2\bar{B}}\right]$$

  where C_0 and C_1 are constants independent of P.
- As P increases, the convergence rate guarantee becomes larger (worse).

KAVG

We adopt a synchronous, P-learner K-step model averaging approach (KAVG). With KAVG, P learners run concurrently, and average their parameters every K steps.

Algorithm 1: KAVG

initialize $\widetilde{\mathbf{w}}_1$

for $n = 1, ..., N$ **do**

Processor P_j, $j = 1, \ldots, P$ do concurrently:

set $\mathbf{w}_n^j = \widetilde{\mathbf{w}}_n$;

for $k = 1, ..., K$ **do**

randomly sample a mini-batch of size B_n and update:

$$\mathbf{w}_{n+k}^j = \mathbf{w}_{n+k-1}^j - \frac{\gamma_n}{B_n} \sum_{s=1}^{B_n} \nabla F(\mathbf{w}_{n+k-1}^j; \xi_{k,s}^j)$$

end for

Synchronize $\widetilde{\mathbf{w}}_{n+1} = \frac{1}{P} \sum_{j=1}^{P} \mathbf{w}_{n+K}^j$;

end for

Analysis of KAVG

Theorem

With a fixed stepsize $\gamma_n = \bar{\gamma}$ and a fixed batch size $B_n = \bar{B}$ for all $n \in \mathbb{N}$ satisfying

$$\bar{B} \geq L\bar{\gamma}M_G(\bar{\gamma}LK + \frac{1}{P}).$$

the expected average squared gradient norm of F for KAVG satisfies the following bounds for all $N \in \mathbb{N}$:

$$\mathbb{E}\frac{1}{N}\sum_{n=1}^{N}\left\|\nabla F(\widetilde{\mathbf{w}}_n)\right\|^2 \leq \left[\frac{2(F(\widetilde{\mathbf{w}}_1)-F^*)}{\bar{\gamma}N(K+1)} + \frac{\bar{\gamma}LM}{\bar{B}}\left(\frac{1}{P} + \frac{\bar{\gamma}LK}{2}\right)\right] \cdot \left(\frac{\bar{\gamma}PLK+1}{\bar{\gamma}PL(K-1)+1}\right)$$

Impact of various parameters on convergence

- $\frac{2(F(\widetilde{\mathbf{w}}_1)-F^*)}{\bar{\gamma}N(K+1)}$ term is primarily impacted by N, and goes to zero as N increases. Increasing K and γ decreases this term.
- $\frac{\bar{\gamma}LM}{B}\left(\frac{1}{P}+\frac{\bar{\gamma}LK}{2}\right)$ term is independent of N, and is impacted by the choice of P, B, and K. Favors large B, large P, and small γ.

KAVG with 4 GPUs

We start with $P = 4$ GPUs, and run KAVG for 28 epochs. $\gamma_1 = 0.025$, and $B = 128$.

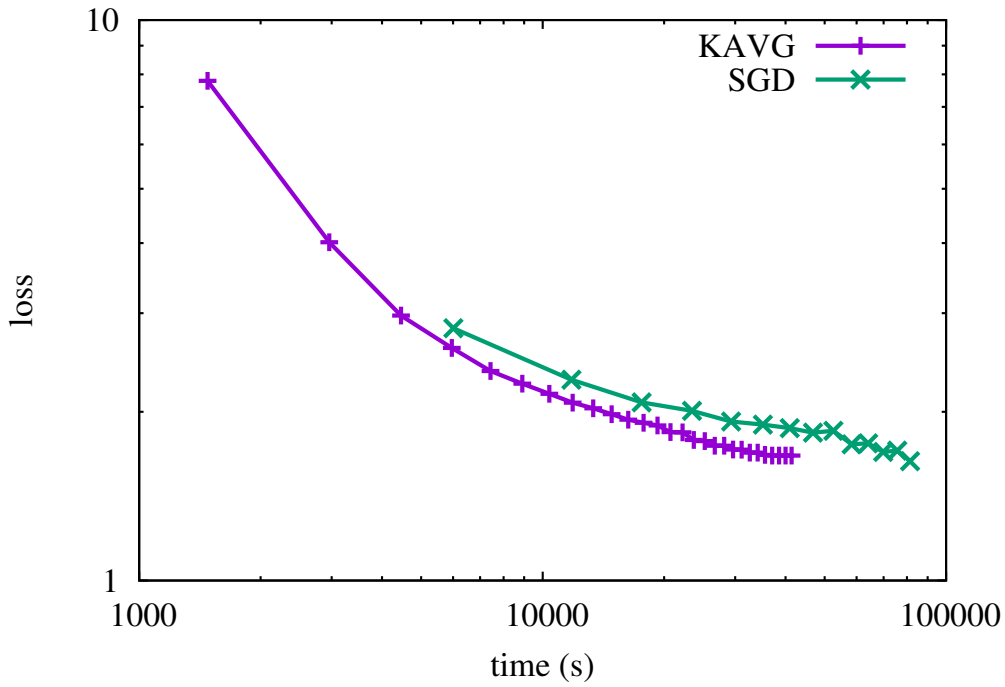

Figure: KAVG with 4 GPUs (log – log plot).

KAVG finishes 28 epochs in 41415 seconds, and the loss achieved is 1.65, close to the loss achieved by SGD

KAVG with 20 GPUs

$\gamma_1 = 0.025$ and $B = 128$.

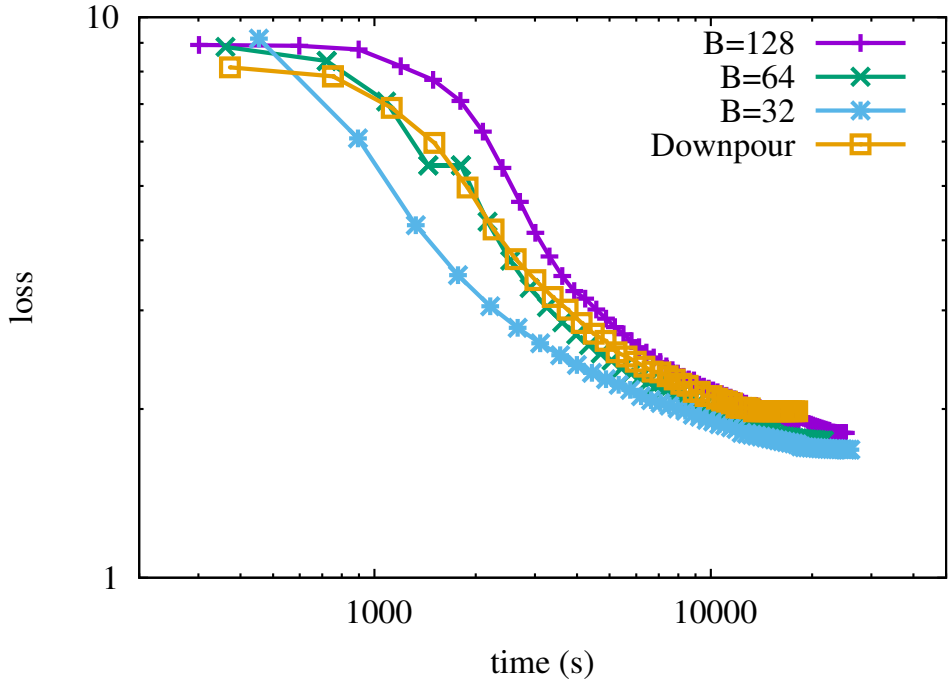

Figure: KAVG and *Downpour* with 20 GPUs (log – log plot).

the *B=128* line shows the evolution of validation loss over time for KAVG with minibatch size 128 for 77 epochs. The wall clock time taken is 25191 seconds. At a final loss of 1.81, the speedup is 3.23.

Explorations of B

- reduce *B* from 128 to 64. KAVG converges much faster with *B=64* than with *B=128*.
  - After 59 epochs, it converges and the final loss is 1.75. The speedup is 3.7.
  - The disadvantage of small *B* is that the GPU computing resources may not be fully utilized. Epoch time becomes longer. Yet the faster convergence rate makes it worthwhile
- further reduce B to *B=32*. Training converges even faster as shown by the *B=32* line. At the end of of 59 epochs, the loss is at 1.70. However, with *B=32*, more epochs do not reduce the loss below 1.70.

Adapting B and γ

Algorithm 2: Adapt(γ, B, B_{max}, γ_{min}, β_B, β_γ)

if validation loss stops improving **then**
  if $B < B_{max}$ **then**
    $B \leftarrow min(B \cdot \beta_B, B_{max})$;
  else
    if $\gamma > \gamma_{min}$ **then**
      $\gamma \leftarrow max(\gamma / \beta_\gamma, \gamma_{min})$;
    end if
  end if
end if
return (γ, B)

Impact of dynamic B

We use $B_{max} = 1024$, and start with $B = 48$. $\beta_\gamma = \beta_B = 1.2$, and γ_{min}=0.

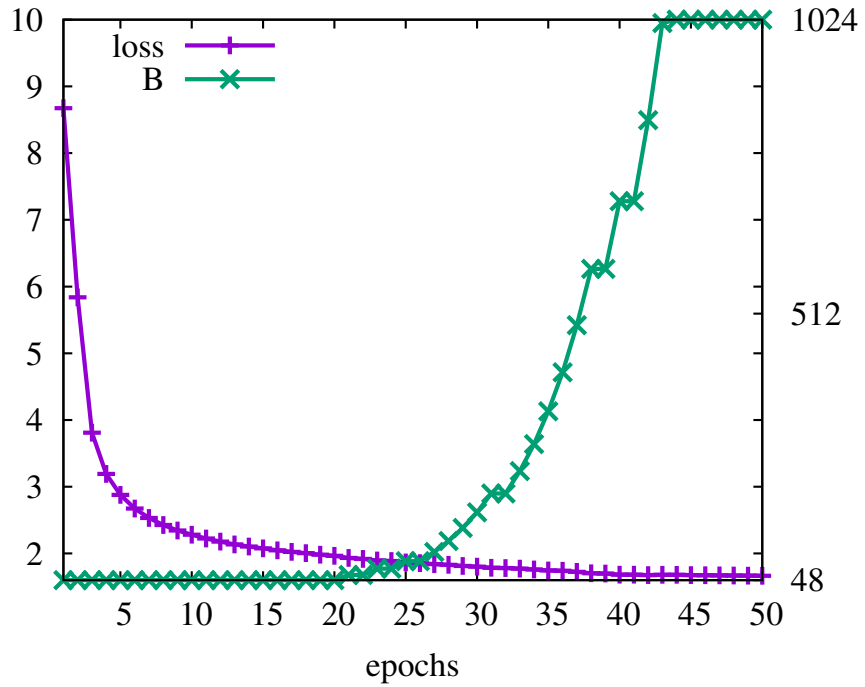

Figure: Adapting B and γ

B remains constant for about 20 epochs, then starts to grow fast and reaches 1024. After 50 epochs, the loss is at around 1.66 < 1.7. Time used is 16150 seconds.

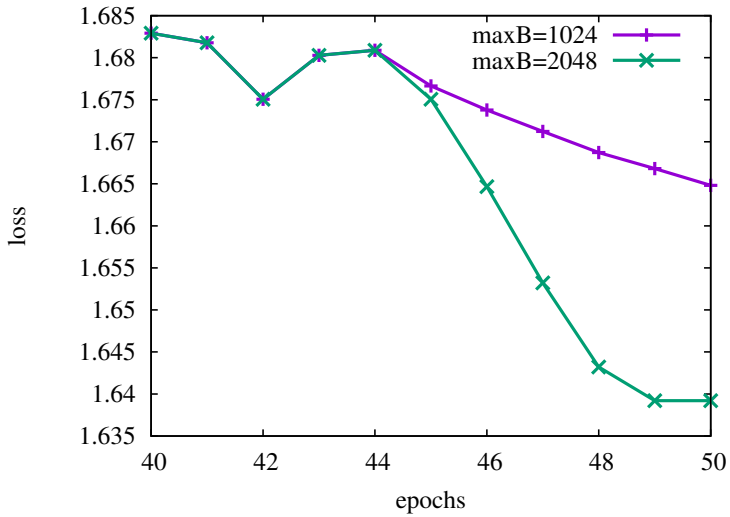

Figure: Impact of very large B_{max}. Results shown for epoch 40 and beyond

Evolution of average gradient norm

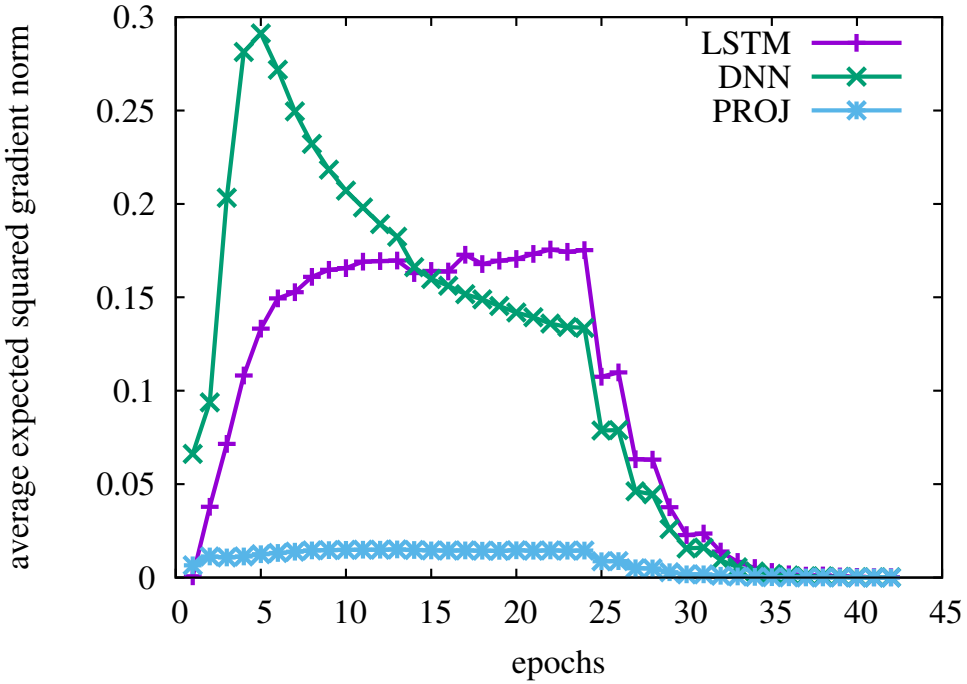

Figure: Evolution of expected average squared gradient norm for LSTM, PROJ, and DNN

Adam with batch size adaptation

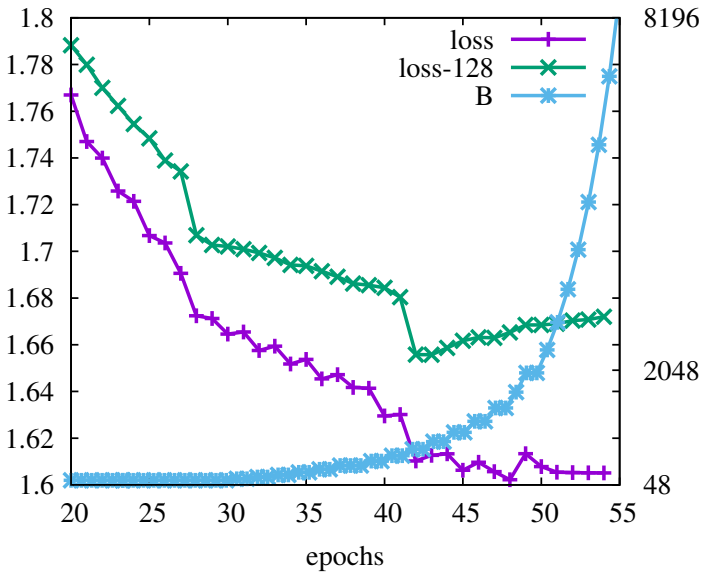

Figure: ADAM with batch size adaptation

KAVG performance

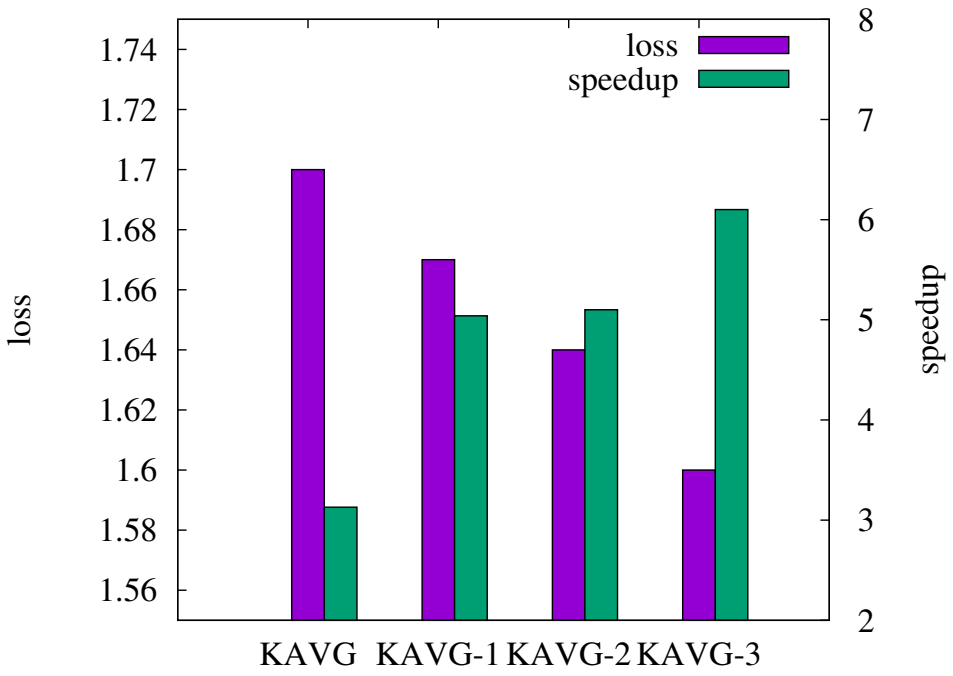

Figure: Performance of four KAVG implementations

Conclusion and future work

- KAVG performs better than ASGD for our application
- Adapting batch size is effective
- Adam is effective
- Further explore larger data sets