RELAXING COHERENCE
FOR MODERN LEARNING APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Joo Hwan Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2017

RELAXING COHERENCE FOR MODERN LEARNING APPLICATIONS

Approved by:

Date Approved: December 1, 2016

Dedicated to God and my family.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank the people who have guided, helped and supported my Ph.D. journey at Georgia Tech.

First of all, I would like to thank my advisor Dr. Hyesoon Kim for her guidance, support and encouragement over the years. Over these years, she has provided me with valuable advice on all aspects of the research while allowing me the freedom to pursue research topics that I am interested in. Her insightful feedback directly pointed out the deficiencies I needed to improve. She is very caring and generous with her help. I am extremely fortunate to have her as my advisor, and all the things that I learned from her will continue to influence me for the rest of my life.

I would like to thank other members of my dissertation committee, Dr. Richard W. Vuduc, Dr. Hadi Esmaeilzadeh, Dr. Le Song, and Dr. Nuwan Jayasena for their valuable feedback and comments on improving the dissertation. I would also like to thank Dr. Sudhakar Yalamanchili for serving on my dissertation proposal committee.

I thank my friends and my colleagues from HPArch, Minjang Kim, Sunpyo Hong, Nagesh Lakshminarayana, Jaekyu Lee, Jaewoong Sim, Jen-Cheng Huang, Pranith Kumar, Dilan Manatunga, Hyojong Kim, Nimit Nigania, Lifeng Nai, Prasun Gera, and Ramyad Hadidi for their feedback. I also thank Jungju Oh, Ching-Kai Liang, and Sunjae Park for many coffee breaks I enjoyed.

I would like to thank for the wonderful researchers outside of Georgia Tech. I thank Jiayuan Meng from Argonne National Lab, Jesse Beu, Ganesh Dasika, Dam Sunwoo, and Myeongjae Jeon at ARM research, Dongping Zhang from AMD research for their help on my research.

Above all, I am deeply thankful to my parents Kang Hyun Lee and Hyun In Shin, and my brother Kyung Hwan Lee for their love, support, sacrifices and patience. They have given me unconditional love and always supported my decisions throughout my life for which I am very grateful. Without their love, I could not have finished my Ph.D. journey. I thank God for having them as my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The main objective of this research is to efficiently execute learning (model training) of modern machine learning (ML) applications. The recent explosion in data has led to the emergence of data-intensive ML applications whose key phase is learning that requires significant amounts of computation. A unique characteristic of learning is that it is iterative-convergent, where a consistent view of memory does not always need to be guaranteed such that parallel workers are allowed to compute using stale values in intermediate computations to relax certain read-after-write data dependencies. While multiple workers read-and-modify shared model parameters multiple times during learning, incurring multiple data communication between workers, most of the data communication is redundant, due to the stale value tolerant characteristic. Relaxing coherence for these learning applications has the potential to provide extraordinary performance and energy benefits but requires innovations across the system stack from hardware and software.

While considerable effort has utilized the stale value tolerance on distributed learning, still inefficient utilization of the full performance potential of this characteristic has caused modern ML applications to have low execution efficiency on the state-of-the-art systems. The inefficiency mainly comes from the lack of architectural considerations and detailed understanding of the different stale value tolerance of different ML applications. Today's architecture, designed to cater to the needs of more traditional workloads, incurs high and often unnecessary overhead. The lack of detailed understanding has led to ambiguity for the stale value tolerance thus failing to take the full performance potential of this characteristic. This dissertation presents several innovations regarding this challenge.

First, this dissertation proposes Bounded Staled Sync (BSSync), hardware support for the bounded staleness consistency model, which accompanies simple logic layers in the memory hierarchy, for reducing atomic operation overhead on data synchronization intensive workloads. The long latency and serialization caused by atomic operations have a significant impact on performance. The proposed technique overlaps the long latency atomic operation with the main computation. Compared to previous work that allows stale values for read operations, BSSync utilizes staleness for write operations, allowing stale-writes. It reduces the inefficiency coming from the data movement between where they are stored and where they are processed.

Second, this dissertation presents StaleLearn, a learning acceleration mechanism to reduce the memory divergence overhead of GPU learning with sparse data. Sparse data induces divergent memory accesses with low locality, thereby consuming a large fraction of total execution time on transferring data across the memory hierarchy. StaleLearn transforms the problem of divergent memory accesses into the synchronization problem by replicating the model, and reduces the synchronization overhead by asynchronous synchronization on Processor-in-Memory (PIM). The stale value tolerance makes possible to clearly decompose tasks between the GPU and PIM, which can effectively exploit parallelism between PIM and GPU cores by overlapping PIM operations with the main computation on GPU cores.

Finally, this dissertation provides a detailed understanding of the different stale value tolerance of different ML applications. While relaxing coherence can reduce the data communication overhead, its complicated impact on the progress of learning has not been well studied thus leading to ambiguity for domain experts and modern systems. We define the stale value tolerance of ML training with the effective learning rate. The effective learning rate can be defined by the implicit momentum hyperparameter, the update density, the activation function selection, RNN cell types, and learning rate adaptation. Findings of this work will open further exploration of asynchronous learning including improving the findings laid out in this dissertation.

CHAPTER 1
INTRODUCTION

1.1 The Problem: Data Communication Overhead of Learning

The recent explosion in data has led to the emergence of machine learning (ML). Multiple real-world problems have effectively been modeled into ML applications such as self-driving cars [1], voice recognition [2], and even the arts [3]. ML applications are expected to become more popular in the future; thus research efforts on efficiently processing ML applications have gained significant importance.

The key phase of ML is learning (model training). ML applications inductively formulate models by examining the patterns in a data set rather than using the hypothetical-deductive method. The learning starts with some guess as to the problems solution and proceeds through some iterations until the computed solution converges. The data-centric learning is the most performance limited phase. Learning requires significant amounts of computation processing large quantities of data, and thus easily takes several days or even months with sequential execution on a single node.

Therefore, a considerable amount of efforts [4, 5, 6, 7, 8, 9, 10, 11] has focused on distributed learning to shorten the training process. Most computations in learning are performed on each data item in the training data set, utilizing high-performance of hundreds of machines. Thanks to those efforts, training an approximate learning app is easier and efficient than developing an exact solution. Many easy-to-use tools [4, 12, 13, 14] have accelerated the domain experts to develop specific applications based on the frame.

Although these efforts have helped the wide application of ML applications, there still exists inefficiency with the significant amount of data communications during learning. The irregular and data-intensive nature of learning lead to high and often unnecessary data communication overhead under-utilizing the performance potential of parallel learning.

In this dissertation, we focus on reducing the data communication overhead of learning applications, which can provide extraordinary performance and energy benefits. The solution to the data communication bottleneck of modern learning applications is to utilize the stale value tolerant characteristic of iterative-convergent learning. The key property of learning is that parallel workers are allowed to compute using stale values in intermediate computations to relax certain read-after-write data dependencies without hurting the convergence guarantee or the final solution quality. While multiple workers read-and-modify shared model parameters multiple times during learning, incurring multiple data communication between workers, most of the data communication is redundant.

While considerable effort [15, 16, 17, 18, 19, 20] has utilized the stale value tolerance on distributed learning trying to reduce the inter-node synchronization cost among multiple nodes, still inefficient utilization of the full performance potential of this characteristic has caused modern ML applications to have low execution efficiency on the state-of-the-art systems. The inefficiency mainly comes from the lack of architectural considerations and a systematic method to understand the different stale value tolerance of different ML applications. Today's architecture, designed to cater to the needs of more traditional workloads, is not perfectly suited for learning applications. The lack of systematic method has caused the ambiguity on this characteristic thus limiting the scalability of learning with lots of data communications, most of which is redundant due to the stale value tolerance.

1.2 The Contributions

In this dissertation, we propose to utilize the stale value tolerant characteristics from the architectural perspective and provide a detailed understanding of the different stale value tolerance of different ML applications. Utilizing the stale value tolerance requires careful designs across the system stack from hardware and software. We design a set of mechanisms for a variety of performance challenges. The highlights of the proposed techniques are as follows.

1.2.1 Atomic Operation Overhead Reduction

First, we observe that the significant performance inefficiency for modern learning applications comes from a data synchronization overhead, which is implemented with atomic operations on a single-node machine such as shared memory processors. In parallel learning applications, atomic operations are typically used to ensure the convergence of lock-free algorithms. However, atomic operations occupy a large portion of overall execution time and become the biggest inefficiency within a node. The inefficiency comes from non-overlapped synchronization with the main computation thus wasting computing time.

For reducing atomic operation overhead, this dissertation proposes Bounded Staled Sync (BSSync), an effective hardware mechanism for the bounded staleness consistency model [21], which accompanies simple logic layers in the memory hierarchy. BSSync reduces the overhead of non-overlapped data communication, the serialization, and cache utilization inefficiency. The proposed technique overlaps the long latency atomic operation with the main computation. The overlapping is only achievable by utilizing the iterative-convergent characteristic. Compared to previous studies that allow staleness for read operations, BSSync utilizes staleness for write operations, allowing stale-writes. BSSync reduces the inefficiency coming from the data movement between where they are stored and where they are processed. With BSSync, atomic operations are asynchronously executed in parallel with the main computation.

1.2.2 GPU Memory Divergence Reduction

Second, we focus on the divergent memory accesses of GPU learning with sparse data. While the GPU is widely used for learning applications due to the enormous amount of parallelism available in data-centric learning [22, 23, 24, 25, 26, 27], the current system does not perfectly utilize the full performance potential of the state-of-art GPU architecture for learning applications with sparse data. Real-world data is sparse in nature, so the values are scattered in the memory. The memory access pattern of the GPU learning is divergent with low temporal locality, thus leading the GPU to suffer from the low utilization of GPU compute units waiting for memory.

Although a considerable effort has been devoted to reducing the divergent memory overhead of GPU, the stale value tolerance of learning provides a unique opportunity to achieve the full potential in modern GPUs. This dissertation presents StaleLearn, a learning acceleration mechanism to reduce the memory divergence overhead of GPU learning by rearranging the memory access pattern based on the stale value tolerant characteristic. Based on the stale value tolerance, StaleLearn transforms the problem of divergent memory accesses into the synchronization problem by replicating the model, and reduces the synchronization overhead by asynchronous synchronization on Processor-in-Memory (PIM). The stale value tolerance enables a clear task decomposition between the GPU and PIM, which can effectively exploit parallelism between PIM and GPU cores by overlapping PIM operations with the main computation on GPU cores.

1.2.3 Systematic Method to Define the Stale Value Tolerance

Finally, we observe that the lack of detailed understanding of the different stale value tolerance of different ML applications has led to ambiguity for domain experts and modern systems. The current ML training suffers from low scalability thus leading to inefficient utilization of millions of computing machines that are currently available in the era of cheap compute. Reducing data communication affects the progress of parallel learning such that different ML applications exhibit different stale value tolerance. The stale value tolerance is determined by the complex function of multiple factors and the complicated impact has led today's learning applications to suffer from large data communication overhead. For efficient execution of learning applications, a thorough investigation to define the different stale value tolerance of different ML learning applications is needed.

This dissertation presents a methodology to provide a detailed understanding of the stale value tolerance. We propose to define the stale value tolerance of ML application with the effective learning rate. The effective learning rate is different from the explicit learning rate specified by the programmer and can be changed by multiple design choices of training neural network. The effective learning rate is affected by i) the accumulated number of local updates from all workers when performing reduction operations that form implicit momentum update, ii) the different update density on different model parameters and different learning rates on different model parameters depending on iii) activation function selection, vi) RNN cell types, and v) learning rate adaptation.

1.3 Thesis Statement

Relaxing coherence for modern learning applications based on stale value tolerant characteristic can increase the execution efficiency, which can be effectively exploited with low-cost architectural modifications and the detailed understanding of the stale value tolerance.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides the background of modern ML applications and summarizes related work. Chapter 3 discusses BSSync, a practical hardware support for reducing atomic operation overheads. Chapter 4 proposes StaleLearn, a learning acceleration mechanism that transforms divergent memory accesses to more GPU-friendly regular/sequential accesses without the data synchronization overhead. Chapter 5 proposes our systematic method to define the stale value tolerance of different ML applications considering different application characteristics. Chapter 6 concludes this dissertation.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter, we first provide a background of the stale value tolerance convergence of modern ML applications (Section 2.1). We then describe the prior state of the art utilizing the characteristic (Section 2.2) and the bounded staleness consistency model we utilize in this dissertation (Section 2.3). After discussing proposals on learning to have the better quality solution (Section 2.4), we then discuss prior work on memory optimization for data-intensive applications (Section 2.5), and processor-in-memory (Section 2.6).

2.1 Stale Value Tolerant Convergence

Most learning applications can be mapped into linear algebra with matrices and vectors. Equation 2.1 shows the general formula of learning using the loss function L [9, 7, 28]. Assuming learning a model for classification, x (matrix) and y (vector) represent the input data for training, and w is the computed model. Each row of the x matrix represents different data points corresponding to multiple features (column) and the y vector represents the classification result. The loss function captures the quality of the model based on the error between the predictions and the ground truth on the y vector. The further away the prediction from the ground truth is the larger the penalty. The output of the training is w^*, the best model that minimizes the loss function.

$$w^* = \underset{w}{\operatorname{argmin}} \sum_{i=1}^{N} L(f_w(x_i), y_i) \tag{2.1}$$

Most learning approaches are iterative, which start with some guess as to the problem's solution and proceed through a number of iterations until the computed solution converges. Figure 2.1 shows an example of the iterative execution of stochastic gradient descent (SGD) [7]. Learning starts with selecting a training data point from the training data x. Learning computes Δw based on the current value of model w. Then, the model update is performed applying the computed Δw to the model w via read-modify-write (RMW) operation that is mostly implemented with atomics. Convergence check decides whether the solution is converged by analyzing the loss function value. Although Figure 2.1 does not show explicitly, there are data dependencies among training data points. w is continuously updated after computing Δw, which results in computing Δw being dependent on the value of w, the outcome of previous data points.

```
Repeat until convergence:
•   Draw a training sample $x_i$ from input x
•   Compute $\mathbf{\Delta} w_i = F(w, x_i)$
•   Read-modify-write $w = w + \mathbf{\Delta} w_i$
```

Figure 2.1: Iterative execution of stochastic gradient descent (SGD).

Since real-world learning involves big data with many data points, data parallel implementation where each worker works on its own data partition while sharing model parameters, is widely used. The iterative-convergent characteristic allows parallel workers in data parallel implementation to compute using stale values in their intermediate computations, without requiring that all workers always have a consistent view of memory [15, 16, 17, 18, 11]. The stale value tolerant characteristic is widely used in parallel learning that parallelizes inherently-sequential learning algorithms such as SGD.

In parallel SGD, each worker partitions training data and reads and modifies the shared model parameter. Read accesses to the current model parameter w and updates to w are overlapped without serializing them thus allowing data races. When multiple workers concurrently access the same model parameter w, w might not have the latest updates, and Δw is computed using a stale value of w. Unlike classical applications that are transaction-centric and deterministic with strict consistency requirements, a transient error in parallel

SGD does not affect the convergence guarantee, and the solution quality.

The stale value tolerant convergence is discovered not only on ML applications but also on multiple graph algorithms, which are known to require the exact computation. Multiple studies [20, 29, 30, 4] have utilized the stale value tolerance of iterative graph algorithms. For instance, in single source shortest path (SSSP) algorithm that computes the shortest path from a given source vertex to each vertex in a graph, the solution converges to the correct solution when updated values of costs to reach vertices by each worker are eventually observed by other workers.

2.2 Utilizing Stale Value Tolerance

Multiple studies [9, 7, 19, 20, 14, 8, 28, 8] have proven the loss of intermediate accuracy (or missing some updates) during learning does not affect the convergence guarantee and the solution quality. Zinkevich et al. [9] prove that stale values do not degrade the solution quality of convex problems. Agarwal and Duchi [7] prove that in smooth stochastic problems, the accuracy loss due to stale values is negligible. SSP [19] proves that the progress of convex problems is not affected much when bounding staleness degree with the iteration count granularity and the steps taken during each iteration are small enough. Recht et al. [8] propose Hogwild!, a lock-free updating scheme performing SGD updates in parallel without locking the parameters for sparse data, utilizing the findings that most gradient updates modify only small parts of the decision variables.

The stale value tolerant characteristic of learning has been mainly utilized in distributed learning to reduce the inter-node communication overhead [15, 16, 17, 18, 19, 20]. The stale value tolerant design is used because it is infeasible to guarantee the consistent view of the entire model on large-scale clusters. Multiple distributed learning platforms focus on throughput, including the Map-Reduce frameworks such as Hadoop [31], Spark [32], and graph-based platforms such as GraphLab [4]. Low et al. [6] focus on special communication patterns and propose Distributed GraphLab. GraphLab programs structure the computation as a graph, where data can exist on nodes and edges in which all communication occurs along the edges. They utilize the findings that if two nodes on the graph are sufficiently far apart, they may be updated without synchronization. Ahmed et al. [5] propose a parallel framework for inference in latent variable models utilizing a best-effort model for updating shared data. ASPIRE [20] proposes a relaxed consistency model and consistency protocol coordinating the use of a best-effort refresh policy. Tensorflow [14] extends DistBelief [28] with an asynchronous stochastic gradient descent procedure and distributed batch optimization procedures for large scale deep neural network learning.

2.3 Bounded Staleness Consistency Model

The bounded staleness consistency model [21] is a variant of the relaxed consistency models in which data read by a worker may be stale, missing some recent updates. The degree of staleness, the delay between when an update operation completes from a worker and when the effects of that update are visible to other workers, is defined under the user-specified threshold. The bounded staleness consistency model allows reads to use stale values unless they are too stale. The staleness is bounded so that it cannot be larger than the user-specified threshold. The bounded staleness consistency model has been widely used for its combined benefits of communication latency tolerance and minimization of the negative impact of stale values on convergence on multiple-node configurations [15, 16, 17, 18, 19, 20, 11].

Staleness can be measured in multiple ways. Interweave [33] supports delta coherence that keeps track of the number of modifications (versions). The Stale Synchronous Parallel (SSP) model [19] defines version numbers in terms of the number of iterations. The version number v of a datum represents that the value of a particular datum has been read by the worker at iteration v. In the SSP model, staleness is defined using the version number and the current iteration of the worker. If the worker is at iteration i, the staleness of the data with version number v is equal to $i - v$.

The bounded staleness consistency model changes the validity of the data and defines stale-hit/miss [20]. Stale-hit means that the staleness is less than or equal to the staleness threshold, and stale-miss refers to the case when the staleness is larger than the threshold. The bounded staleness consistency model affects the behavior of the read operation. Different workers can have different views of a shared datum; the read operation is only guaranteed to obtain the value whose staleness is less than or equal to the staleness threshold s. In the SSP model, when a worker reads a shared datum d at iteration i, the worker is guaranteed to see the effect of all updates on d from the first iteration to iteration $i-s$.

Most parallel learning methods concurrently process a certain number of training examples that forms a batch (an iteration). The number of missing updates within a batch is different depending on learning methods even with the same staleness degree in the SSP model. There are three variants of gradient descent method that compute the gradient with different amount of training data: batch gradient descent (BGD), stochastic gradient descent (SGD), and mini-batch gradient descent (MBGD). In BGD method, the gradient is computed by processing entire training dataset, while the gradient is computed by processing each training example in SGD method. MBGD method computes the gradient for every mini-batch of training examples. The number of missing updates within a batch is the largest when using SGD method, and the smallest when using BGD method.

2.4 Solving the Difficulty of Training

Lots of ML efforts have targeted to solve the difficulty of training of deep neural network (DNN). Sutskever et al. [34] analyze the impact of weight initialization and the momentum schedule in momentum-based stochastic gradient descent on DNN. Glorot and Bengio [35] propose to utilize non-squashing activation functions to reduce the saturating gradient problem. Pascanu et al. [36] suggest a gradient norm clipping strategy dealing with exploding gradients and maintaining a soft constraint for the vanishing gradients problem. Recent studies suggest residual learning [37, 38] for the training of networks that are substantially deeper than previously used, which learns residual functions regarding the layer inputs. On residual learning, each layer fine tunes the output from a previous layer by adding a learned residual to the input. The same inputs do travel through paths and a different number of layers so that the inputs of a lower layer is made available to a node in a higher layer.

Some number of studies propose different methods for learning instead of back-propagation. Compared to back-propagation based learning, where all layers of the network are locked waiting for other layers before they can be updated, they try to transform this synchronous execution into the asynchronous execution of different layers. Baxter and Bartlett [39] propose a reinforcement learning algorithm for computing approximated gradient of the average reward from a single sample path of a controlled partially observable Markov decision process. The method of auxiliary coordinates (MAC) [40] replaces the original problem involving a deeply nested function with a constrained problem dealing with a different function in an augmented space without nesting. Ollivier and Charpiat [41] propose the NoBackTrack algorithm utilizing an unbiased random estimate of the gradient of the loss function. The recent work by Jaderberg et al. [42] offers decoupled neural interfaces (DNI). DNI uses the synthetic gradient produced using only local information in place of true back-propagated error gradients. DNI decouples the sub-graphs, and the update of them can be performed independently and asynchronously.

2.5 Memory Optimization

The irregular and data-intensive nature of learning often lead to memory bottleneck with long memory latency. To reduce the memory latency, several prefetching mechanisms have been proposed. Ryoo et al. [43] suggest binding prefetch on GPUs. Yan et al. [44] propose a compiler-based approach performing software-based prefetching on registers with a fixed prefetch distance. Lee et al. [45] implement the Many-Thread Aware Prefetcher (MTAP), a hardware prefetcher performing intra-warp and inter-warp prefetching. Lakshminarayana and Kim [46] propose a tag-based hardware prefetcher that identifies target load pair and injects prefetch instruction to prefetch data into spare registers with a dynamic prefetch distance.

While the GPU is widely used for learning applications due to the enormous amount of parallelism available data-centric learning, the current system does not perfectly utilize the full performance potential of the state-of-art GPU architecture for learning applications with sparse data. A considerable effort has been done to mitigate the overhead of divergent memory accesses from GPU applications with a sparse data structure. Tarjan et al. [47] propose diverge on miss, allowing individual threads in a warp to continue while other threads are waiting for memory. Meng et al. [26] propose dynamic warp subdivision to create warp-splits, extra schedulable entities that do not require extra register file space in the case when there are not enough warps to hide memory latency. Chatterjee et al. [48] propose warp-aware scheduling schemes that coordinate scheduling decisions across multiple memory channels to reduce the DRAM latency divergence on irregular GPU workloads. Rhu et al. [49] propose LAMAR that adjusts access granularity depending on the locality of GPU applications to reduce the memory bandwidth utilization inefficiency.

Data reorganization is a widely used operation to reduce the overhead of scattered memory accesses. Wu et al. [50] perform a complexity analysis of data reorganization to minimize non-coalesced memory accesses on GPUs and proposed new data reorganization methods. Greathouse and Daga [51] address the challenge of irregular memory access on SpMV applications due to the CSR format by streaming data into the local scratchpad memory and then dynamically assigning rows to each compute unit. To increase efficiency, multiple studies have proposed hardware-assisted data reorganization [52, 53, 54] and recent efforts [55, 56] utilize PIM technology. Dymaxion [52] allows programmers to optimize memory mappings and uses GPU's high memory bandwidth to overlap with slow PCI-E transfer. Gou et al. [54] propose the extended Single-Affiliation Multiple-Stride (SAMS), a parallel memory scheme to maintain multiple layouts of the same data for SIMD processors. Micro-pages [53] co-locates chunks of cache blocks from different OS pages in a row-buffer. HAMLeT [55] is a 3D-stacked accelerator that performs data layout transformation by exploiting the locality and parallelism within the 3D-stack. Akin et al. [56] propose a permutation-based mathematical framework for data reorganization.

Although these efforts can be used to reducing the memory bottleneck of learning, still, the lack of architectural consideration of the stale value tolerance has led to under-utilizing the unique opportunity to achieve the full performance potential of multi-threaded architecture for learning applications. So, in this dissertation, we propose several architectural innovations that reduce the unnecessary hardware data communication overhead between computing units and memory.

2.6 Processor-in-Memory

The recent emergence of 3D-stacking technology enabled high performance by incorporating different technologies: a logic and memory layer that are manufactured with different processes [57, 58, 59]. So, multiple vendors such as Micron, are revisiting the concept of processing data where the data lives, utilizing processor-in-memory (PIM) [60] technology. Hybrid Memory Cube technology [61] has simple in-memory atomic operations. The Automata processor [62] directly implements non-deterministic finite automata in hardware to implement complex regular expression. Chu et al. [63] propose a high level, C++ template-based programming model for processor-in-memory that abstracts out low-level details from programmers. Nai and Kim [64] evaluates instruction offloading for graph traversals on HMC 2.0. Kim et al. [65] study the energy aspect of processor-in-memory for HPC workloads.

Scatter/gather operations are one of the popular operations on PIM and already have been implemented in hardware [66]. Several studies utilize the operations to reduce the overhead of remote memory accesses. Ahn et al. [67] propose the scatter-add mechanism, which scatters a set of data values to a set of memory addresses and adds each data value to each referenced memory location for parallel global accumulation. Fang et al. [68] propose the active memory operation, in which select operations can be sent to and executed on the home memory controller of data to reduce remote memory access. Erez et al. [69] develop techniques for mapping the applications onto a stream processor whose memory system supports complex addressing modes including scatter/gather.

CHAPTER 3
BSSYNC: REDUCING ATOMIC OPERATION OVERHEADS

3.1 Overview

In this chapter, we propose Bounded Staled Sync (BSSync), hardware support integrating logic layers on the memory hierarchy to reduce the atomic operation overhead in parallel learning applications. BSSync offloads atomic operations onto logic layers on memory devices to fully utilize the performance potential of parallel learning applications. Atomic operations are now overlapped with the main computation to increase execution efficiency. BSSync revisits the hardware/software interfaces to exploit parallelism.

We identify and quantify the bottleneck of atomic operations and evaluate how our proposal increases the execution efficiency. In summary, the key contributions in this chapter are as follows:

1. We evaluate parallel learning applications within a single node, unlike previous works that focus on communication latency between nodes.

2. We observe that the atomic operation overhead causes inefficiencies within a single node for parallel learning applications. We quantify how much the overhead contributes to the overall execution time.

3. Compared to previous works that utilize staleness only for read operations, BSSync utilizes staleness also for writes, allowing stale-writes that accompany simple logic layers at the memory hierarchy.

4. We propose hardware support that reduces the atomic operation overhead. Our evaluation reveals that our proposal outperforms a baseline implementation that utilizes the asynchronous parallel programming model by 1.33x times.

3.2 Background: Asynchronous Parallelism

Exploiting asynchronous parallelism has the benefit of addressing the issue of workload imbalance between threads that introduce a significant overhead for iterative-convergent learning applications [19]. For the Bulk Synchronous Parallel (BSP) model [70], all threads must execute the same iteration at the same time, and barrier synchronization is used at every iteration to guarantee that all running threads are in the same phase of execution. On the contrary, with asynchronous execution, threads can perform computation instead of waiting for other threads to finish [71].

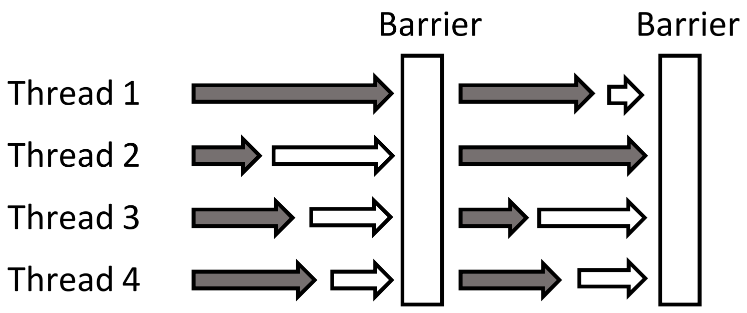
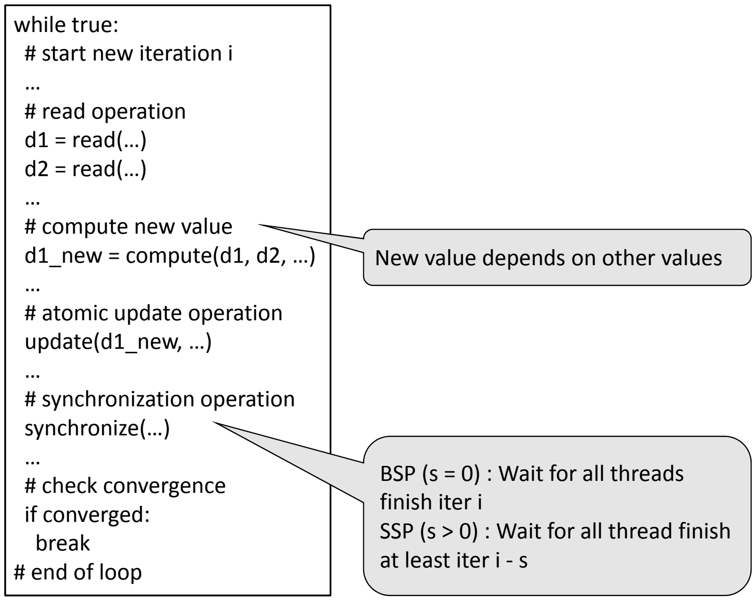


Figure 3.1: Straggler problem of the BSP model.

Figure 3.1 illustrates the problem of wasted computing time for the BSP model due to straggler threads. The white arrows represent the wasted computing time and the gray arrows represent when each thread performs computation. With barrier synchronization, each thread waits for the others at every iteration. As such, even when only a single straggler thread has not reached the barrier, other threads must stay idle as they wait for the straggler thread to finish before they can start the next iteration, thereby leading to wasted computing time. Relaxing the barrier can reduce the stall time, which leads to significant speedups for a variety of iterative-convergent learning applications.

The performance of iterative-convergent learning applications is determined by how much the solution has progressed within a certain time, which is the product of both 1) the number of iterations per time and 2) the progress per each iteration. A small iteration difference between threads can lead to the large progress per iteration. Relaxing the barrier yields more iterations per time but lowers the progress per iteration, therefore increasing the number of iterations required to converge to the final solution.[1]



```
while true:
  # start new iteration i
  …
  # read operation
  d1 = read(…)
  d2 = read(…)
  …
  # compute new value
  d1_new = compute(d1, d2, …)
  …
  # atomic update operation
  update(d1_new, …)
  …
  # synchronization operation
  synchronize(…)
  …
  # check convergence
  if converged:
    break
# end of loop
```

New value depends on other values
BSP (s = 0) : Wait for all threads finish iter i
SSP (s > 0) : Wait for all thread finish at least iter i - s

Figure 3.2: Stages of iterative-convergent workloads.

The Stale Synchronous Parallel (SSP) model [19] is a type of programming/execution model that makes use of asynchronous parallelism. Figure 3.2 shows a pseudo-code example of parallel iterative-convergent learning applications utilizing the SSP model. At a high level, a loop iteration consists of five stages of operation. First, a loop iteration starts with read operations fetching inputs (stage 1), followed by the computation on the inputs to generate new data (stage 2). The read operations may fetch stale values, and the stale values can be used for computation. Then, after executing an atomic update operation to store a new computation result (stage 3), a synchronization operation is performed (stage 4). Unlike the barrier operation in the BSP model, the synchronization operation is used to guarantee that the iteration counts of different threads are within the specified ranges. With the user-specified staleness threshold s, the fast thread should stall if not all threads have

[1] Iterative-convergent applications continue to iterate while the computed solution keeps changing from iteration to iteration. Convergence check determines whether or not the solution has converged (unchanged).

progressed for enough iterations; the thread should wait for slower threads until they finish iteration $i - s$. At the end of the iteration, a convergence check is performed (stage 5). If the values have not converged, the threads proceed to another iteration. The computed results from the iteration are used as inputs for the other threads and for the later iterations from the thread. This process continues until the computed values converge.

The SSP model provides the benefit of both the BSP and asynchronous execution models for iterative-convergent learning applications. It alleviates the overhead of straggler threads because threads can perform computation instead of waiting for other threads to finish. At the same time, the bound on staleness enables a faster progress toward the final solution within an iteration. We utilize the SSP model in our evaluation for asynchronous parallel workloads. Figure 3.3 compares the performance of the BSP model and the SSP model with a staleness threshold of two. The staleness threshold is selected through our experiments to find the value that provides the best speedup. Figure 3.3 shows that the SSP model outperforms the BSP model by 1.37x times.

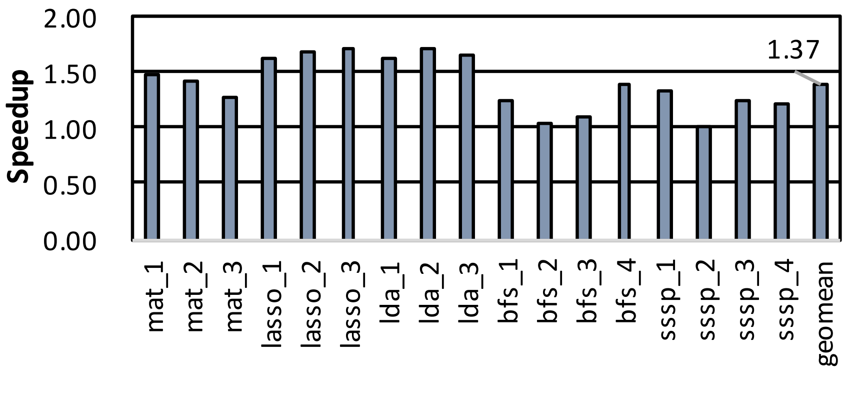


Figure 3.3: Speedups of the SSP model over the BSP model.

3.3 Motivation: Atomic Operation Overhead

Atomic read-modify-write operations capable of reading, modifying, and writing a value to the memory without interference from other threads are frequently used in workloads where threads must sequentialize writes to a shared variable. They provide serializability so that memory access appears to occur at the same time to every thread. Atomic operations affect how threads see updates from other threads for the shared memory address.

Atomic update operations are used in parallel learning applications for reduction operations. Reduction is used to combine the result of computation from each thread. Atomic operations enable different threads to update the same memory address in parallel code. For example, to implement Matrix Factorization, atomic-inc/dec operations are used. Atomic-inc/dec reads a word from a memory location, increments/decrements the word, and writes the result back to the memory location without interference from other threads. No other thread can access this address until the operation is complete. If atomicity is not provided, multi-threaded systems can read/write in shared memory thus inducing the data race. The data race can lead to reduction operation failure, which can slow down progress per iteration and even break the convergence guarantee.

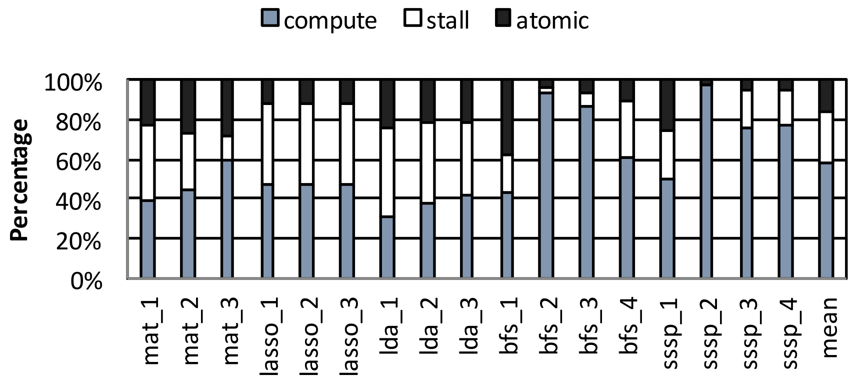


Figure 3.4: Portion of each pass on the BSP Model.

While previous studies show that exploiting asynchronous parallelism could improve performance significantly by reducing inter-node communications, the atomic operation overhead also has a huge impact on performance within a single node. Figures 3.4 and 3.5 show the execution breakdown of the BSP model and the SSP model with a staleness threshold of two on a single node. We measure the execution time of different stages from each thread and use the sum of all threads as the execution time of that stage for the

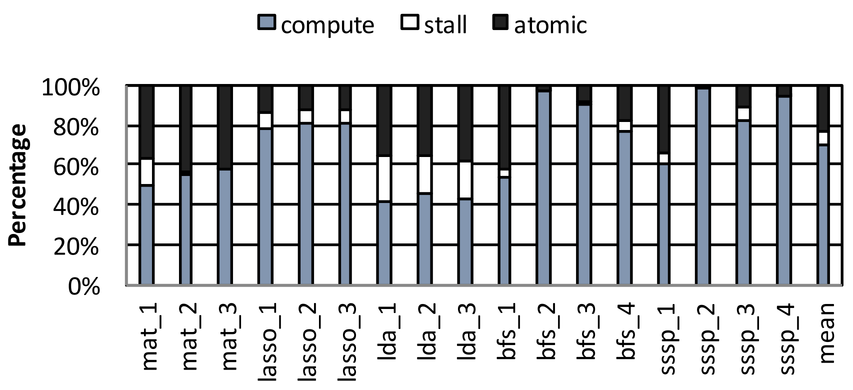


Figure 3.5: Portion of each pass on the SSP Model.

workload.[2]

On the BSP models, a major portion of the execution time is not spent on the main computation; only 58% is spent for the main computation, 16% for atomic update operations, and 26% for stall time. As previously explained, stall occurs because of the imbalanced progress of each thread in terms of iterations. The main reason for the thread imbalance is due to the sparse nature of the data and value-dependent computation in ML workloads; that is, different threads have non-uniform workload distributions. For example, in Breadth First Search (BFS), each vertex has a different number of neighbors. The task is typically partitioned so that each thread processes a disjoint subset of vertices; therefore different threads can execute a different number of instructions.

The stall time is reduced on the SSP model by allowing asynchronous execution; on the SSP model as an average, stall time is reduced to 7% from 26% of the execution time on the BSP model. The atomic update operation overhead now becomes the dominant performance bottleneck on the SSP model. On average, the atomic operation overhead consists of 23% of execution time on the SSP model. Still, 30% of the time is wasted, not performing the main computation.

Figure 3.6 shows the pseudo-code of atomic update operations using the compare-and-swap (CAS) operation.[3] The CAS operation is provided as a single instruction in many

[2]See Section 3.5 for detailed explanations of workloads and the hardware.

[3]We omit the Load-Linked/Store-Conditional (LL/SC) implementation that incurs similar cost as CAS

architectures such as x86. The atomic operation consists of four steps. First, a normal memory load operation is performed. This load operation does not allow reading the stale value unlike the load operation fetching the value for the main computation (stage 2 in Figure 3.2). Second, the new value to store (new_val) is calculated by the program using the value computed from the main computation (val) and the loaded value (old). Third, a check operation is performed to decide whether to perform a CAS operation. Fourth, the CAS operation for the same memory location is performed, which compares the memory contents with the original loaded value. Only when the values match, does the swap operation occur, updating memory.

```
void atomic_operation(int* address, int val){
  // normal memory load operation
  int old = *address;

  // compute new value to store
  int new_val = compute(old, val);

  int assumed;
  do {
    assumed = old;
    // decide whether to perform CAS operation
    if(good_to_compare_and_swap(assumed, new_val)){
      // CAS operation
      old = compare_and_swap(address, assumed, new_val);
    }else{
      return;
    }
  }while(assumed != old);
}
```

Figure 3.6: Pseudo code of atomic update operation.

The memory-intensive atomic operation incurs high overhead with non-overlapped multiple transactions on the lower level of the memory hierarchy. The transactions of reading, modifying, and writing a value back to memory are done by the core, and the data movement is not overlapped with the main computation, thus increasing execution time. Fetching data from the lower level of the memory hierarchy to the L1 cache increases the latency of the atomic operation, which can become worse with large data since the data should be fetched from DRAM rather than shared cache. Other cores trying to modify the

implementation for brevity.

same cache line can cause the repetition of the memory load and CAS operation, which increases the L1 access, which will be mostly cache misses.

The increased latency resulting from invalidation also increases the overhead, and the invalidation traffic will also be problematic on the shared cache with many cores. When multiple threads try to modify the same line, a lot of invalidations result since different threads will send the invalidation.

Also, all threads that try to access the same location are sequentialized to assure atomicity. Possible collisions can cause poor performance as threads are sequentialized. As the atomics serialize accesses to the same element, the performance of atomic instructions will be inversely proportional to the number of threads that access the same address. As more cores become available on-chip, performance degradation resulting from serialization will increase.

While overlapping atomic operations with computation are possible with launching extra workers, launching a background thread is neither realistic nor beneficial for paralell learning applications since launching more threads for computation is typically more helpful than launching background threads. When executing highly parallel data-intensive applications on many cores, despite the reduced stall time with asynchronous parallelism, strict consistency with atomic operations has caused slow execution. Therefore, we conclude that the performance of learning applications is atomic operation bound.

3.4 Solution

Now, we propose *BSSync*, hardware support for offloading the atomic update operation onto logic layers on memory devices. After we describe the key idea of our proposal, we explain how data communication is performed with our hardware implementation.

3.4.1 Key Idea

BSSync reduces the atomic update operation overhead by utilizing the characteristic of the iterative-convergent learning applications that allows the use of stale values in the computation. Compared to previous studies utilizing the characteristic only for read operations, BSSync utilizes the characteristic also for write operations to allow stale-writes to minimize the adverse impact of long latency atomic operations. BSSync is based on the following two ideas regarding the iterative-convergent algorithms and state-of-art hardware implementations.

- First, atomic update operations in iterative-convergent learning applications are for other threads to see the updates within a certain staleness bound. The atomic update stage is separate from the main computation, and it can be overlapped with the main computation. Since learning applications do not enforce strict data dependence constraints (threads can miss a certain number of updates to use stale values), the update operation can be performed asynchronously.
- Second, atomic update operations are a very limited, pre-specified set of operations that do not require the full flexibility of a general-purpose core. The hard-wired implementation of atomic update operations on the memory hierarchy can be more efficient.

A key observation is that offloading atomic update operations to asynchronously execute in parallel with the CPU core can eliminate the overhead of atomic update operations. The atomic update operation in Figure 3.6 can be offloaded, and the operation can be immediately retired from the core side but the logic layer can guarantee the atomicity. Atomic operations only need to guarantee the atomicity of the update. Assuming the atomicity is provided, that is, if the update from a core is not over-written by other updates, it is all right when the atomic update operation is performed asynchronously so that other threads can observe the recent value later.

Asynchronous execution of atomic operations reduces the overhead of long latency reading the value from the lower level of the memory hierarchy. It reduces the overhead by transforming blocking operations into non-blocking operations. The CPU core does not wait for the atomic operations to finish but proceeds to the main computation, enabling high performance. Asynchronous execution of atomic operations also reduces the overhead of redundant computation for retrying in the case of conflict and the serialization overhead that blocks cores from proceeding for computation.

Instead of general-purpose cores, BSSync utilizes simple logic layers at the memory hierarchy to perform atomic operations. The CMOS-based logic layer can provide an efficient implementation for atomic operations and can help the CPU core to be latency tolerant. While the atomic operation is basically read-modify-write, reading the value from the thread performing the atomic operation is not needed since the value will not be used by the thread. The read can probably incur fetching the value from the lower level of the memory hierarchy. Since the data will be invalidated before being used, it is inefficient to fetch the data to be located near the core expecting short latency with a cache hit. The second generation of Hybrid Memory Cube (HMC) also supports simple atomic operations [61]. The implementation is quite straight-forward without the requirement of full flexibility of the CPU core so that it can be a hard-wired implementation.

We use the staleness definitions of the SSP model [19]. The version number v represents that the value of a particular datum has been read by the thread at iteration v. The staleness of the data with version number v is defined as $i - v$, if the thread is at iteration i. We redefine the meaning of a cache hit/miss as in ASPIRE [20]. With the user-defined staleness threshold s, the read request to a datum can be

- Stale-hit: datum in cache and staleness $\leq$ s
- Stale-miss: datum in cache and staleness $>$ s
- Cache-miss: datum not in cache

3.4.2 Structure

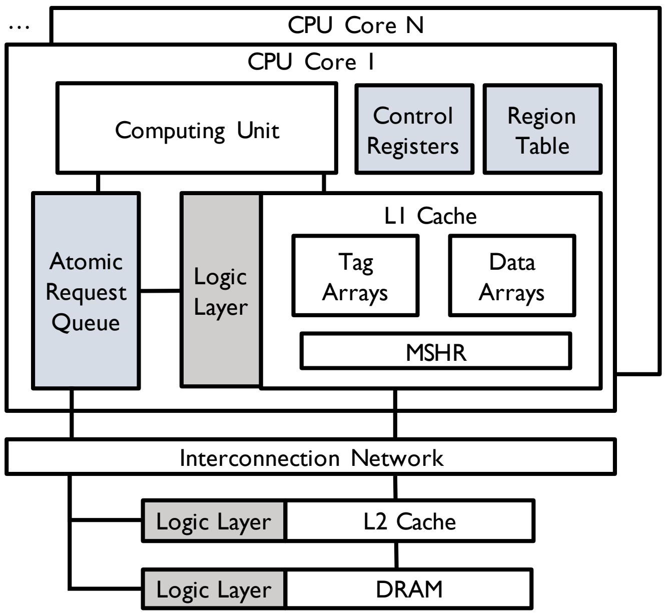


Figure 3.7: Hardware extension of BSSync.

Figure 3.7 shows the hardware extension of BSSync. Each core is extended with a region table, control registers (one iteration register and one threshold register per core), and an atomic request queue (ATRQ). The cache hierarchy consists of per-core private L1 data caches and an inclusive shared L2 cache. The traditional directory-based coherence mechanism is extended to control the degree of coherence. Logic layers are extended on each level of memory hierarchy: L1 data cache, L2 cache, and DRAM.[4]

BSSync changes the conventional hardware/software interfaces to exploit parallelism between the host core and the logic layer at the memory hierarchy. The programmer needs to define the staleness threshold, specify the loop that is associated with the staleness of data, and provide the thread progress in terms of iteration counts of the loop. The programmer also needs to modify the code to invoke the assembly-level atomic instruction and annotate the shared memory object that allows bounded staleness consistency. While this requires certain changes, these changes are mostly straight forward so they can be easily

[4] To coordinate atomic operation execution on DRAM, DRAM controller is extended.

done by the programmer.

The information provided by the programmer is used by the BSSync hardware. The region table for each core contains the address range of annotated memory objects, one entry for each memory object. The iteration register tracks the progress of each thread storing the iteration count that the core is currently executing. The threshold register stores the staleness threshold, which is provided by the programmer.

ISA is extended to pass the programmer-provided information to the BSSync hardware. BSSync includes the special instructions to set and clear the region table, the threshold register, and the iteration register. During the initialization phase of the learning application, the special instruction inserted by programmer is called to set the region table, and the threshold register. Before starting a new iteration, each thread calls the special instruction that updates the iteration register for the thread with the iteration number that the thread will start. The value stored in the region table and control registers are cleared when the special instruction inserted by programmer is called during the termination phase of the learning application.

BSSync also includes the fixed-function atomic instructions executing on the logic layer. We extend the opcode to encode the atomic operation. The format of the atomic instruction follows the format of the current load and store instructions where the size of the operation is encoded.[5]

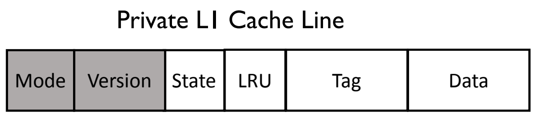


Figure 3.8: Cache tag extension of private L1 data cache line.

BSSync utilizes different cache protocols, depending on the type of memory object. The decision for which protocol to use is made at the cache line granularity on the private L1 data cache. Figure 3.8 shows the cache tag extension of the L1 cache line. The tag

[5] While we consider the bounded operand as the single word of data, the operation might operate on multiple words using pre-defined vectors as SIMD extensions are supported, which is straight-forward.

entry for each L1 cache line is extended to include additional bits for tracking (a) coherence mode, and (b) version number of the cache line. The mode bit is used to define the coherence mode of the cache line. The coherence mode is bi-modal: Normal (N = 0), and Bounded staled (B = 1). The normal line follows the conventional coherence protocols: write-back policy with write-allocate, and the MESI coherence protocol, but the bounded staled line follows a different protocol described in later sections. The version bits in the L1 cache line are used to track the time until this cache line is valid, provide the time when the cache line should be invalidated and get the new update from the lower level of the memory hierarchy.

The ATRQ decouples the atomic operations from the conventional coherence tracking structures. The ATRQ is for the computing unit to send the atomic operation requests to logic layers at the memory hierarchy. It is placed between the computing units and the logic layer on the L1 data cache. It is also connected to the shared L2 cache and DRAM through an interface to the interconnection network. This interface is similar to the one in many cache bypass mechanisms. The ATRQ shares an interface to the core's MMU and uses the physical address translated from the MMU to send the request to the logic layers of the shared L2 cache and DRAM.

3.4.3 Differentiating Memory Instructions

In BSSync, there are two different ways to handle memory instruction: memory accesses for normal memory objects and memory accesses for objects allowed to read stale values. The CPU core identifies the memory access type by using a region table and sends the memory request to different memory units.

The normal memory accesses are handled the same way as conventional directory-based MESI protocols. When a L1 cache miss occurs, a new miss status-holding register (MSHR) entry and cache line are reserved for the line if there is no prior request to the same cache line. The MSHR entry is released when the cache line arrives from the L2 cache.

When invalidation occurs, the invalidation requests are sent to all sharers and the L1 cache tag array and MSHR table are read upon receiving the request.

On the contrary, memory accesses for the objects allowed to read stale values do not follow conventional protocols. The accesses are further decomposed into the memory read requests for the objects, and the atomic reduction operation requests for those objects.

3.4.4 Handling Read Requests

The data that are modified through atomic operations are also read by each thread for computation. Each thread needs to fetch recent changes on the shared data into each thread's private L1 data cache. When a core makes a read request for the data on its private L1 cache, it checks whether the data is too stale (the staleness is larger than the staleness threshold). The version number of the cache line is used to decide stale-hit/miss; therefore it is used to define the limit until the line is used. If it is a stale-hit, no further action is required; the core keeps accessing the line in the L1 cache.

On the contrary, in the case of a stale-miss, the read request is blocked and the line in the lower level of the memory hierarchy should be fetched into the private L1 cache even if the line resides in the L1 cache. The cache line is invalidated from the L1 cache and the fetch request is sent to the shared L2 cache.[6] The version bit in the tag array is updated with the current iteration count that is stored in the iteration register of the core.

When the data is not cached in the L1 cache, thus incurring a cache miss, the core brings the data from the lower level of the memory hierarchy into the private L1 cache and updates the mode bit to one (bounded staled) and updates the version bit with the current iteration count. The behavior is same as if the version number in the L1 cache were $-\infty$.

[6]The fetch request goes to the L2 cache directly instead of looking for the cache line in other core's L1 cache, since the shared L2 cache accumulates more number of updates than thread-private L1 caches. This design choice helps CPU threads to observe the recent value earlier than fetching the value from other L1 caches. The detailed explanation of how atomic operation is handled is provided in later sections.

3.4.5 *Handling Atomic Operation Requests*

Figure 3.9 shows how BSSync handles the atomic request. First, the core issues an atomic operation request into the ATRQ (1). In BSSync, the ATRQ holds the information of the requests that are not completed. The atomic reduction operation is a non-blocking operation when there exists available ATRQ entry so that the operation completes immediately after the core simply puts the request into the ATRQ.[7] The logic layers on the cache and the DRAM perform atomic operations and notify the ATRQ to release its entry when the atomic operation is finished (2).

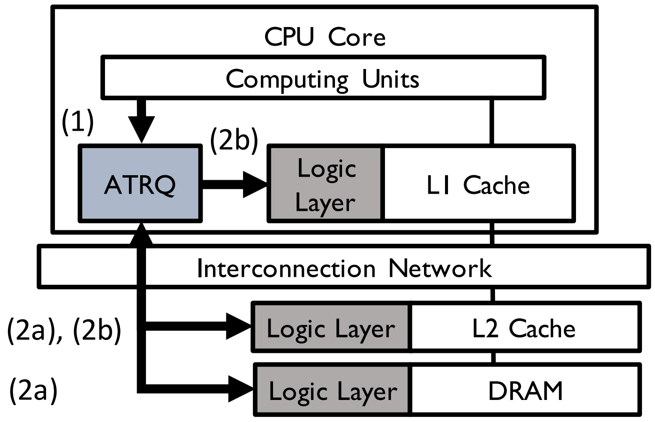


Figure 3.9: Handling atomic operation request.

The processing of the atomic operation request depends on whether or not the datum resides in the L1 data cache. In the case of an L1 cache miss (2a), the ATRQ diverts the atomic operation requests to bypass the L1 D-cache and directly sends requests through the interconnection network into the lower level of the memory hierarchy. If the line resides in the L2 cache, the logic layer in the L2 cache performs the atomic operation, sets the L2 cache line as ”dirty” (changing the state bits as ”modified”) and notifies the ATRQ. In the case of an L2 cache miss, the logic layer on the DRAM performs the operation and notifies the ATRQ.

When the line resides in the L1 cache (2b), the ATRQ sends the atomic operation re-

[7]The core stalls when ATRQ is full.

quest to both logic layers on the private L1 cache and the shared L2 cache. Since we assume inclusive cache, the line resides on the L2 cache if it resides on the L1 cache. The state bits of the L1 cache line still remain as ”clean,” unlike conventional coherence protocols, but changes only the state bits on the L2 cache line. The ATRQ entry is released when the logic layer at the L2 cache notifies the ATRQ after finishing the atomic operation.

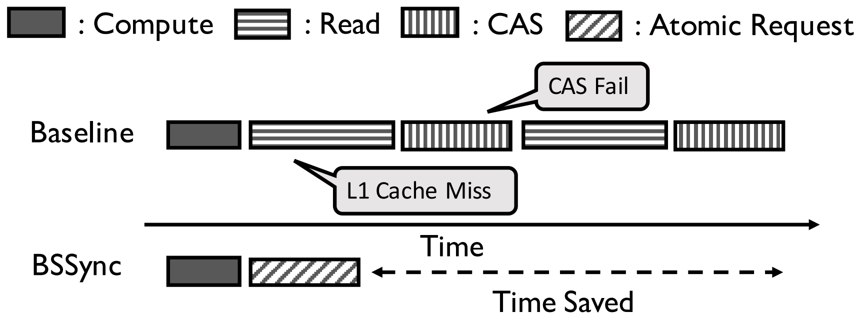


Figure 3.10: Comparison to conventional protocol

Figure 3.10 compares how BSSync changes the way the atomic operation is handled. While the atomic operation is completed by sending the atomic operation request into the ATRQ with BSSync, conventional implementation requires memory load and the CAS operation. The memory load can miss on the L1 cache, thus fetching the cache line from the lower level of the memory hierarchy. The CAS operation can fail incurring the repetition of the memory load and CAS.

BSSync supports two different types of atomic operations: atomic-inc/dec and atomic-min/max. The value accompanied by the atomic operation request is different depending on the type. For atomic-inc/dec, the ATRQ entry holds the delta for inc/dec, and the logic layer performs the inc/dec operation with the delta. For atomic-min/max, the logic layer compares the value for the datum in the request and the one in the memory hierarchy and stores the minimum/maximum value from the comparison.

It should be noted that our mechanism is different from the studies that bypass the private L1 cache to reduce the contention at the private L1 cache. Bypassing can help since it can reduce the contention of invalidation and serialization resulting from multiple writers and reduce cache thrashing that evicts the lines that may be reused shortly. However, bypassing loses the opportunity of the shorter access latency when the requested data resides in the L1 cache. When only fetching data from the L2 cache, the latency of accessing data will increase. On the contrary, BSSync increases the reuse at the private L1 cache reducing cache thrashing and invalidation by allowing multiple values for the same data and not using the L1 cache for the no-reuse data.

Our mechanism is also different from write-through policy where all writes directly go to the shared cache. BSSync eliminates the memory load within atomic operations for the low-reuse data, not the store operation, which is required for whatever policy is used for write.

3.4.6 Handling Evictions

BSSync changes how eviction is handled. When eviction occurs, the memory hierarchy uses the dirty bit (state bit as modified) to identify if write-back needs to be performed. Since the state bits of the L1 cache line of annotated memory objects are always ”clean,” the line is just evicted from the private cache without performing write-back on the L2 cache. Since we assume inclusive cache and perform atomic operation on both the L1 and L2 caches for L1 cache hit (Figure 3.9 (2b)), the updated value on the L1 cache should have already been applied to the L2 cache.

When the cache lines in the shared L2 cache are evicted, BSSync performs similar tasks as in the conventional protocol. The dirty lines on the L2 cache are written back to the DRAM. The atomic operation performed on the L2 cache sets the L2 cache line as ”dirty”, so that the new value can be stored onto DRAM when eviction occurs.

3.5 Experimental Methodology

3.5.1 Benchmarks and Inputs

We evaluate five data-intensive applications with different inputs: Least Squares Matrix Factorization (MAT), LASSO regression (LASSO), Latent Dirichlet Allocation (LDA), Breadth First Search (BFS), and Single Source Shortest Path (SSSP). MAT, LASSO, and LDA are from Petuum [19] and utilize the atomic-inc/dec operation. BFS and SSSP are taken from implementations provided by Harish et al. [72, 73] and utilize the atomic-min operation. MAT learns two matrices, L and R, such that $L * R$ is approximately equal to an input matrix X. Given a design matrix X and a response vector Y, LASSO estimates the regression coefficient vector β, where coefficients corresponding to the features relevant to Y become non-zero. LDA automatically finds the topics, and the topics of each document from a set of documents using Gibbs sampling. BFS expands the Breadth First Search tree from a given source node, and SSSP calculates the shortest path for each vertex from the given source vertex. We port original workloads into pthread workloads utilizing all available threads.

Tables 3.1 and 3.2 show the input for our evaluation. Each data set has varying properties. For each workload, we measure the workload completion time. Each thread iterates on a loop until the solution converges. We sum the execution time of each iteration and use it for performance comparison.

Table 3.1: Inputs for MAT, LASSO, and LDA from Petuum [19].

Workload	**Input**
Matrix Factorization (MAT)	729 X 729 matrix rank : 9 (1), 27 (2), 81 (3)
LASSO Regression (LASSO)	50 samples X 1M features lambda : 0.1 (1), 0.01 (2), 0.001 (3)
Latent Dirichlet Allocation (LDA)	20-news-groups data set topics : 4 (1), 8 (2), 16 (3)

Table 3.2: Input graphs for BFS, and SSSP from DynoGraph [74].

Graph	Vertices	Edges	Characteristic
coAuthorsDBLP (1)	299067	977676	Co-authorship network generated by DBLP
PGPgiantcompo (2)	10680	24316	PGP trust network
cond-mat-2003 (3)	30460	120029	Co-authorship network of condensed matter publications
ny_sandy (4)	44062	77348	Live Twitter event

3.5.2 Hardware Configurations

Table 3.3: Baseline hardware configuration.

Number of x86 cores	64
x86 core	x86 instruction set(user space) Out of order execution, 2.4 GHz
On-chip caches	MESI coherence, 64B line, LRU replacement L1I cache: 16 KB, 8-way assoc, 3 cycles L1D cache: 16 KB, 4-way assoc, 2 cycles L2 cache: 2 MB, 4 bank, 16-way assoc, 27 cycles
DRAM	Single controller, 4 ranks/channel, 8 banks/rank Closed page policy Latency : 100 cycles

For evaluating our hardware mechanism, we utilize ZSIM [75]. Table 3.3 shows the baseline hardware configuration on the ZSIM simulator. Not just the core but also the memory model are modeled in detail. Each core runs x86 instructions and consists of the execution pipeline, the private L1 instruction and data caches. In BSSync, each core is also extended to include the region table, and the control registers. The coherence mode and version number for each cache line are integrated by extending tag arrays to track the status of the cache line in the caches. The caches use the LRU replacement policy. A simple logic layer performing atomic operations is integrated on the caches and DRAM. We assume the single-cycle latency of performing the atomic operation when the request reaches the logic layer and assume the same cache access latency for the baseline and BSSync.[8]

[8]While the hardware extension of BSSync can potentially increase the latency of atomic operation and cache accesses, our evaluation revealed that the benefit of reducing contention by far transcends the overhead.

The overhead of the hardware extension of BSSync is negligible. The region table has 4 entries whose size depends on the number of annotated memory objects. In many iterative-convergent learning applications, these regions are contiguous memory regions and not many objects exist in most applications. The length for control registers and version bits on L1 cache tag arrays is set to 4 bits, which depend on the maximal staleness threshold with the use of modulo operation, which incurs negligible overhead. The ATRQ has 64 entries per core and we did not see resource contention for the ATRQ in the evaluation.

3.6 Experimental Results

3.6.1 Overall Performance

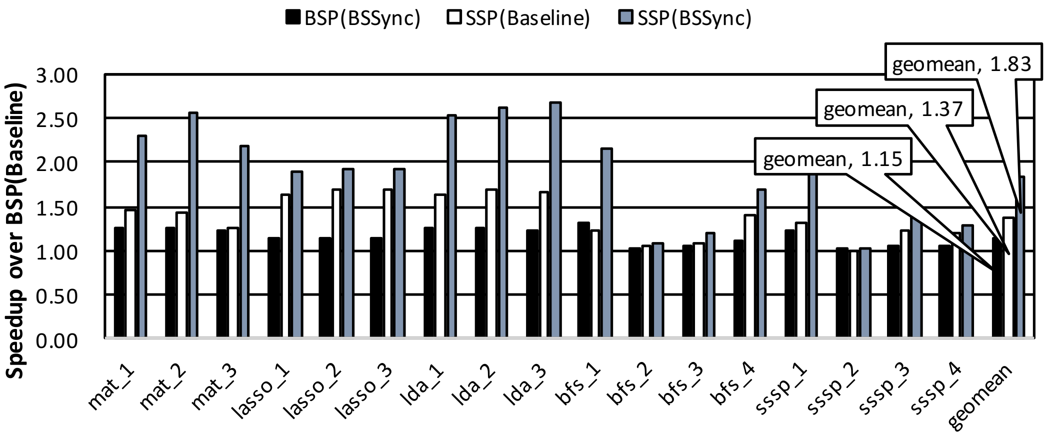


Figure 3.11: Speedup of BSSync.

Figure 3.11 shows the speedup of BSSync with the BSP model and the SSP model with a staleness threshold of two.[9] Not only with the SSP model, but also with the BSP model, BSSync reduces the atomic operation overhead. On average, BSSync outperforms the baseline implementation of SSP model by $1.83 \div 1.37 = 1.33$x times and the one of the BSP model by 1.15x times.

BSSync shows greater benefit on the SSP model for two reasons. First, it is because the portion of the atomic operation for overall execution time is greater on the SSP model. On

[9]The number after application name corresponds to different inputs shown in Tables 3.1 and 3.2.

the BSP model, stall time still consumes a large portion of the execution time and has less benefit. Second, it is because BSSync can better overlap atomic operation execution with the main computation on the SSP model than on the BSP model. On the BSP model, the computation result of an iteration should be visible to other threads before starting the next iteration, so atomic operations are not fully overlapped with the main computation.

The benefit of BSSync varies depending on workloads with regard to words per instructions. In general, if a thread touches more shared data per instruction, the degree of benefit from BSSync increases. MAT and LDA show a higher ratio for atomic operations than LASSO; therefore, they have more benefit with BSSync. LDA shows greater benefit with increasing topic counts. BFS and SSSP show the highest benefit with coAuthorsDBLP (1) input, which is the largest graph in our input sets.

3.6.2 Analysis

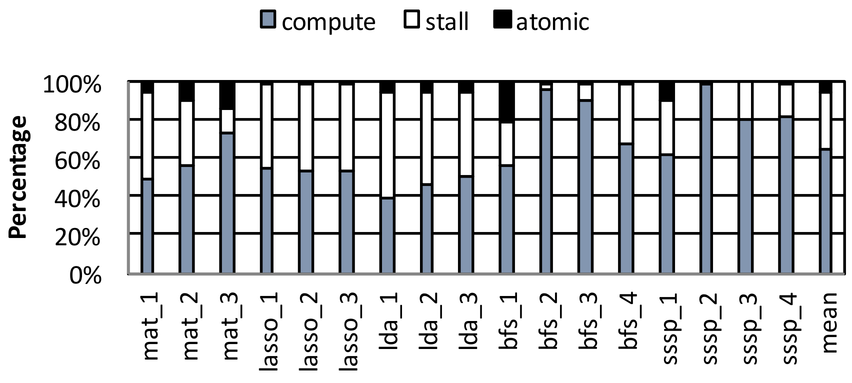


Figure 3.12: Portion of each pass on the BSP Model with BSSync.

Figures 3.12 and 3.13 show the reduced portion of the atomic pass in total execution time. Figure 3.12 shows that the atomic operation overhead consists of 5% of the execution time on the BSP model with BSSync. In Figure 3.13, the atomic operation overhead is reduced to 2.3% from 23% of the baseline implementation of the SSP model. Now, 89% of the time is spent on the main computation with BSSync for the SSP model.

A major portion of performance improvement of BSSync can be attributed to reduced

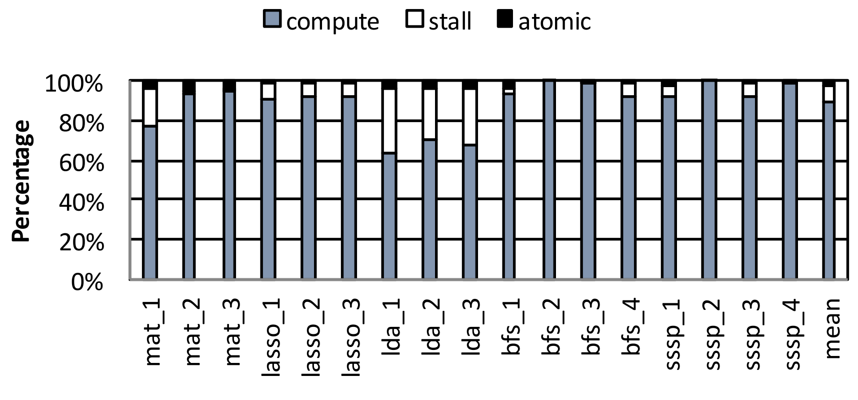


Figure 3.13: Portion of each pass on the SSP Model with BSSync.

number of memory loads. Performing atomic operations at the core incurs a large number of memory loads, which inefficiently handles data with high contention. Contention from multiple cores can incur the repetition of atomic operations and therefore more memory loads. Figure 3.14 shows how BSSync reduces the number of memory loads by offloading atomic operations to the logic layer for the SSP model with a staleness threshold of two. BSSync reduces the number of loads by 33%. Benchmarks such as LDA show less than half the number of memory loads from the baseline with BSSync.

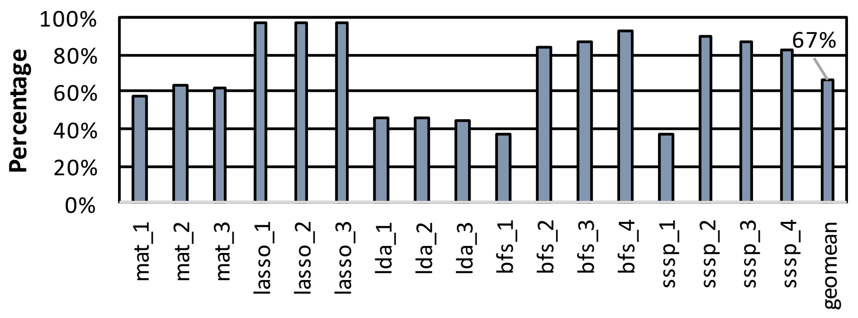


Figure 3.14: Reduced memory loads on the SSP model.

Contention from multiple cores to write the same cache line increases memory access latency. The memory latency is increased due to the round trip time to invalidate other cores and receive their acknowledgments, and for requesting and receiving synchronous write-backs. BSSync reduces memory access latency by reducing on-chip communication, reduces the upgrade misses by reducing exclusive request, and reduces sharing misses by reducing invalidations. Figure 3.15 shows the reduced invalidation traffic with BSSync for the SSP model with a staleness threshold of two. Invalidation traffic is reduced to 43% of baseline implementation. On the majority of the benchmarks, the invalidation traffic is reduced more than 50% from the baseline, which translates into a lower completion time.

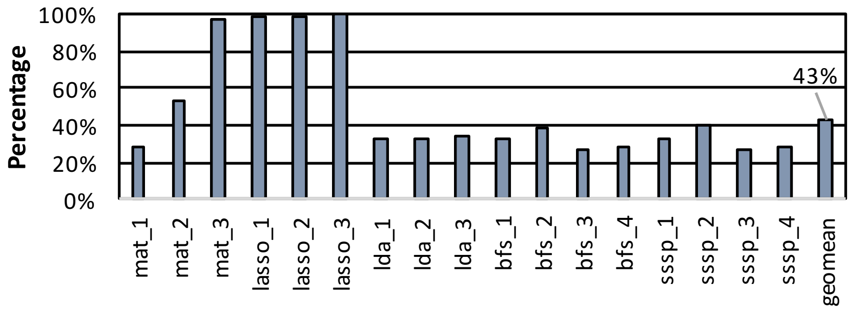


Figure 3.15: Reduced invalidation traffic on the SSP model.

BSSync also increases the L1 private cache utilization efficiency. Performing atomic operations at the core can incur a large number of accesses to the lower level of the memory hierarchy due to memory loads for data accessed in atomic operation. Unnecessary fetches of a word that will be invalidated before being used can lead to reducing effective L1 cache size and the cores can suffer from expensive communication to the lower level of the memory hierarchy such as the L2 cache and DRAM. Access to the L2 cache and DRAM is costly since there is additional latency due to the time spent in the network, and the queuing delays.

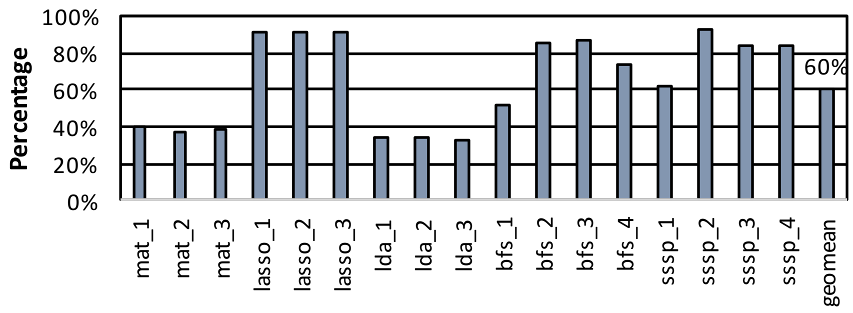


Figure 3.16: Reduced L2 cache read accesses with BSSync.

Figures 3.16 and 3.17 show a reduced number of read accesses to the L2 cache and DRAM. While BSSync involves accesses for atomic operations bypassing to the L2 cache

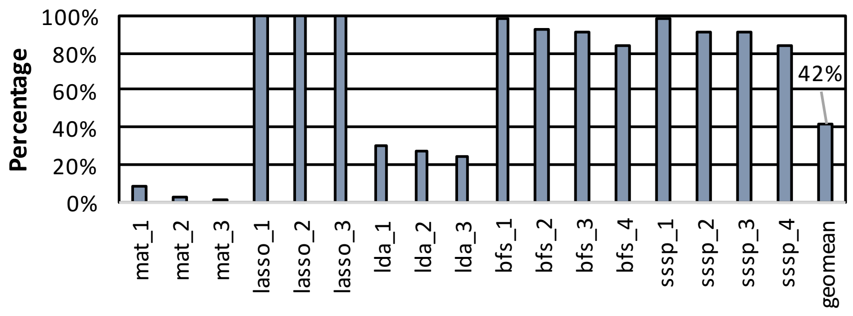


Figure 3.17: Reduced DRAM read accesses with BSSync.

or DRAM, BSSync reduces the capacity misses of the L1 cache thus reducing evictions of the other L1 cache lines to reduce read accesses to the L2 cache and DRAM. In Figures 3.16 and 3.17, BSSync reduces the number of L2 cache read accesses by 40% and reduces DRAM read accesses to 42% of the baseline implementation. Benchmarks like MAT and LDA benefit by reduced L2 cache read accesses and DRAM read accesses. BFS and SSSP also benefit from lower L2 cache access frequency with coAuthorsDBLP (1) input.

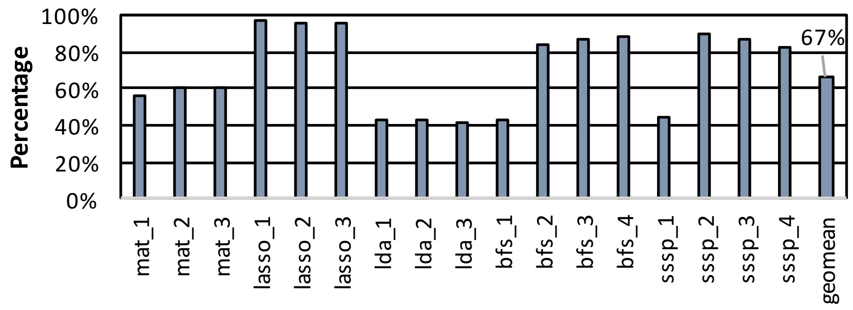


Figure 3.18: Total memory waiting cycles on the SSP model.

Here, we model how BSSync reduces memory waiting cycles at the cores. An application's execution time can be partitioned into computing cycles, memory waiting cycles, and synchronization cycles. The speedup of BSSync comes from overlapping atomic operations with the main computation, thus reducing execution time. Figure 3.18 shows how memory waiting cycles are reduced with BSSync. On average, the memory waiting cycles are reduced by 33%.

3.6.3 Discussion

Comparison with Other Studies: Compared to other studies focusing on inter-node communication latency, BSSync tries to improve execution efficiency within a single node. The atomic operation overhead due to contention has a huge impact on performance within a single node, while it is less important on distributed platforms thus leading most distributed learning studies only to focus on stale-reads. To reduce the atomic operation overhead within a single node, BSSync also utilizes the stale value tolerant characteristic for writes. BSSync can easily be combined with other studies to further improve overall performance.

Thread Migration: So far we have assumed that a thread executes on a pre-defined physical core and that the thread is not switched to other cores so that each physical core tracks the iteration count. If a thread migrates between cores, modification is required such that all hardware should use the thread id instead of the physical core id. However, overall, it is a minor modification and will not affect the benefit of our mechanism.

3.7 Summary

The importance of parallel learning applications for various application domains has been growing in the big data era. While previous studies focus on communication latency between nodes, the long latency and serialization caused by atomic operations have caused the workloads to have low execution efficiency within a single node.

In this chapter, we propose BSSync, an effective hardware mechanism to overcome the overhead of atomic operations consisting of non-overlapped data communication, serialization, and cache utilization inefficiency. BSSync accompanies simple logic layers at the memory hierarchy offloading atomic operations to asynchronously execute in parallel with the main computation, utilizing staleness for write operations. The performance results show that BSSync outperforms the asynchronous parallel implementation by 1.33x times.

CHAPTER 4
STALELEARN: LEARNING ACCELERATION WITH ASYNCHRONOUS SYNCHRONIZATION BETWEEN MODEL REPLICAS ON PIM

4.1 Overview

Many GPU learning application have low execution efficiency due to sparse data. Sparse data induces divergent memory accesses with low locality, thereby consuming a major fraction of total execution time on transferring data across the memory hierarchy. The learning involves applying simple computation for a massive amount of data, leading to a high data-to-compute ratio. The divergent memory accesses leads the GPU to suffer from the low utilization of GPU computing units waiting for memory. Although considerable effort has been devoted to reducing the divergent memory overhead, iterative-convergent learning provides an unique opportunity to achieve full potential in modern GPUs allowing different threads to continue computation using stale values.

We find that relaxing the consistency can play a pivotal role in parallel GPU learning. While considerable effort [15, 16, 17, 18, 19, 20] has utilized the stale value tolerance on distributed learning trying to reduce the inter-node synchronization cost among multiple nodes, there has been no work that brings the characteristic to reduce the hardware communication cost between the GPU and the memory. The lack of architectural considerations has led modern GPU learning to have low execution efficiency.

The stale value tolerance enables to utilize existing PIM technology [60] for the memory bottleneck of GPU learning. The recent advance in the 3D-stacking technology has led to a variety of PIM proposals [57, 58, 59, 76, 62, 77] as a technique for addressing the memory bottleneck. Despite its potential for memory bottleneck, the difficulty of task decomposition between the GPU and PIM has led to the inefficient utilization of the potential.

The stale value tolerance enables clear task decomposition, which can effectively exploit parallelism between PIM and GPU cores.

Therefore, we propose *StaleLearn*, a learning acceleration mechanism that reduces the memory divergence overhead by *transforming* the problem into the *synchronization* problem. Based on the stale value tolerant characteristic of learning, StaleLearn transforms divergent memory accesses of GPU learning into more GPU-friendly regular/sequential accesses by *model replication* and *asynchronous synchronization* on PIM. The stale value tolerance enables the memory access pattern conversion and overlapping low locality synchronization operation on PIM with the main computation on GPU cores.

We demonstrate the benefit of the proposed scheme for representative GPU learning applications with sparse data. We identify and quantify the memory bottleneck of GPU learning and evaluate how our proposal increases the execution efficiency of learning on the GPU architecture. In summary, the key contributions of our work are as follows:

1. We observe that the divergent memory accesses in GPU learning are the major cause of low execution efficiency. We find that the stale value tolerant characteristic provides a unique opportunity to increase the performance of the memory-bound GPU learning with the reduced synchronization requirement.

2. We propose StaleLearn, the first work utilizing the stale value tolerance to reduce the divergent memory access overhead of GPU learning. Unlike distributed learning studies focusing on inter-node synchronization, our solution brings the stale value tolerant characteristic to reduce the hardware communication cost between the GPU and the memory.

3. StaleLearn is the first work to reduce the memory divergence of GPU learning with existing PIM operations. StaleLearn transforms the problem of divergent memory access into data synchronization problem by model replication and reduces the synchronization overhead by asynchronous synchronization on PIM.

4.2 Transforming Memory Divergence into Data Synchronization

4.2.1 Categories of ML applications

GPU learning has become popular with a large amount of parallelism found in learning. Data-centric learning requires significant amounts of computation. Therefore, considerable efforts have focused on shortening the learning time by utilizing an enormous amount of parallelism of learning and utilizing GPUs [22, 23, 24, 25, 26, 27].

However, GPU is not the best choice for all ML applications. The best hardware for learning is different depending on the characteristic of training data. Different ML applications utilize training data with the different degree of parallelism and sparsity of features. Figure 4.1 shows the scatter plot of LIBSVM dataset [78] and UCI machine learning repository [79], where the x-axis represents the degree of parallelism and the y-axis represents the degree of sparsity. The degree of parallelism is defined as the number of data points. The degree of sparsity is defined as the number of features divided by the average number of features in each data point, such that the large degree of sparsity represents when each data point has only a small subset of features.

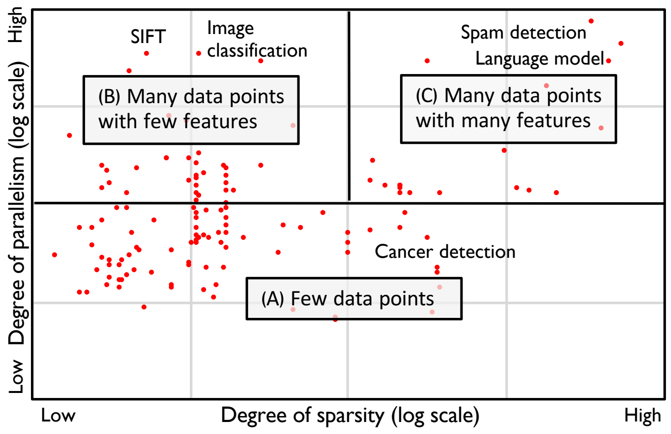


Figure 4.1: Categories of ML applications.

We categorize ML applications into three categories as shown in Figure 4.1: a small degree of parallelism (A-category), a large degree of parallelism with a small degree of sparsity (B-category), a large degree of parallelism with a large degree of sparsity (C-category). For A-category applications, GPU is not the best hardware for learning since the degree of the parallelism is low. On the contrary, for B-category applications, the GPU is effective for exploiting regular parallelism with thousands of threads and large memory bandwidth.[1] Most GPU learning platforms [13, 80, 14] have focused on this category of ML applications, such as the image classification task where the convolutional neural network (CNN) is heavily used [81].

However, little effort has been devoted to applications in the C-category. Figure 4.1 shows a non-negligible number of ML applications fall into this category. While the number of features is large in multiple real-world problems such as the natural language processing having more than millions of features [82, 83], the training data is typically sparse on these problems. While the large degree of parallelism leads to consider GPUs for these applications, the large degree of sparsity of data has prevented GPUs from perfectly utilizing the performance potential of the state-of-art GPU architecture. The increasing interest in the analysis of sparse data such as those in social networks mandates to solve the inefficiency of GPU learning for these applications. So, in this chapter, we focus on GPU learning with sparse data.

4.2.2 Memory Bottleneck of Sparse Data

While a large number of features are widely used to achieve more precise results, in multiple real-world ML applications, data is sparse in nature, where each data point has only a small subset of features. Due to the sparsity, sparse data representations are used in many implementations. Assuming learning a model for classification, x (matrix) represents the training data, and w is the computed model. Figure 4.2 shows an example of a sparse data representation, the compressed sparse row format for the learning application. Here, each row can be mapped as a data point, and columns can be mapped as features.

[1]Chapter 3 covers applications in this category.

Instead of maintaining a matrix whose values are mostly empty, three vectors are maintained: row_offset vector, column_index vector, and data vector. The data vector stores the valid features from the x matrix. Integer values are used for indexing memory objects so that the column_index vector stores the column indices of all the valid columns, and the row_offset vector stores the index of the column_index vector where the first valid column index of each row is stored. The model w is represented as a weight vector whose size is proportional to the number of features.

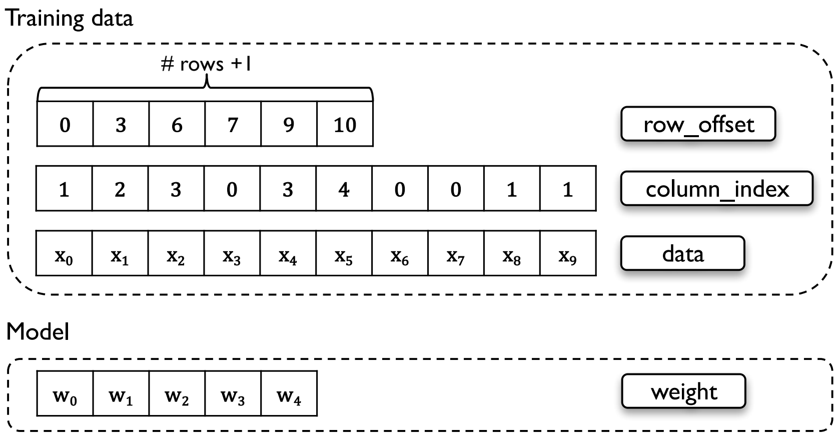


Figure 4.2: Compressed sparse row format.

In many ML applications, the GPU suffers from the divergent memory accesses on the weight vector during learning. Figure 4.3 shows an example of weight vector accesses for widely used data parallel implementation, where the data structure is organized to make each thread process a disjoint subset of data. In this example, each thread is assigned to one row (i.e., one element in row_offset vector) and each thread iterates the column for the given row reading the dedicated region of the column_index vector to find the row's valid columns. Weight vector accesses are dependent on column_index vector accesses, which leads to scattered, divergent accesses. Figure 4.3 shows that the thread processing row 1 accesses the weights w_0, w_3, and w_4. While the column_index vector accesses are sequential accesses, weight vector accesses are dependent on the column_index vector and become divergent accesses.

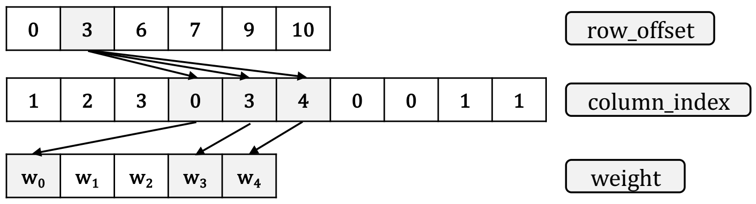


Figure 4.3: Weight vector access pattern.

The divergent weight vector access cause the GPU cores to stall while waiting for the low locality data to be fetched from the lower level of memory hierarchy. Fetching these data with no temporal/spatial locality only results in frequent capacity misses and high DRAM accesses. GPU learning cannot take the benefit the large memory bandwidth of GPU if GPU generates only divergent memory accesses.

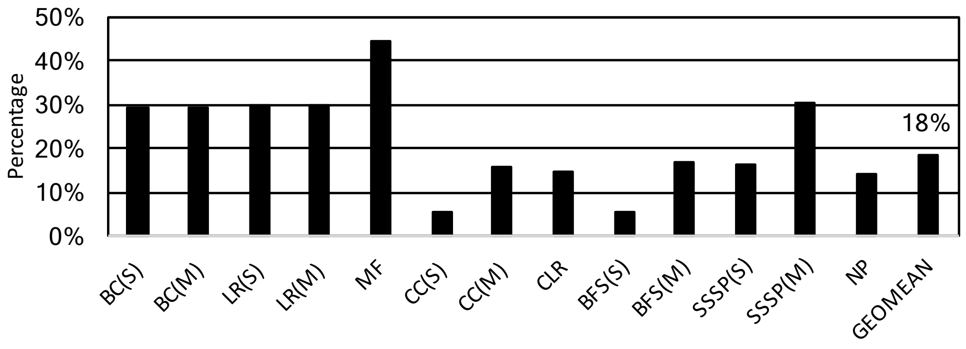


Figure 4.4: Cache line usage before eviction.

To evaluate the memory divergence during learning, we evaluate the ratio of how much cache space is wasted to bring unused data within a cache block.[2] Divergent memory accesses cause only a small portion of the cache line to be used, and most of the fetched data is not used at all before being evicted without exploiting spatial locality. Figure 4.4 shows that only 18% of data within a cache line is used before eviction, thus wasting 82% of cache space.

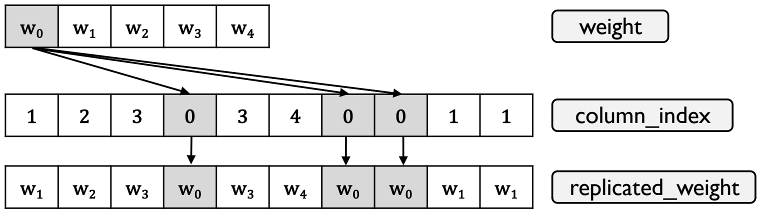

w_0 w_1 w_2 w_3 w_4
weight
1 2 3 0 3 4 0 0 1 1
column_index
w_1 w_2 w_3 w_0 w_3 w_4 w_0 w_0 w_1 w_1
replicated_weight

Figure 4.5: Model replication.

4.2.3 Model Replication

We propose to transform divergent memory accesses of GPU learning into more GPU-friendly sequential accesses using *model replication*. Model replication creates multiple replicas of a single weight, such that each GPU thread maintains its private copy of a single weight. Figure 4.5 shows an example of the model replication. The programmer defines an additional vector, the replicated_weight vector, which is the replica of the weight vector in Figure 4.3. The replication of the weight is performed along the column_index vector. So, the single weight at index 0 on the weight vector is replicated into multiple locations on the replicated_weight vector, whose indices are the same as the column_index vector entries with the value of 0.

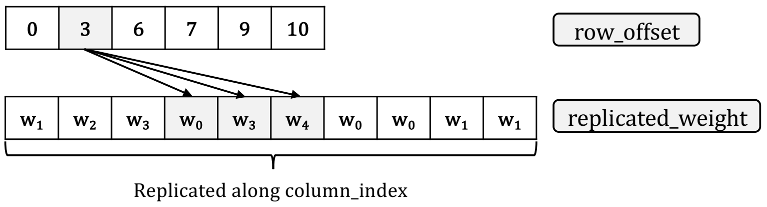

0 3 6 7 9 10
row_offset
w_1 w_2 w_3 w_0 w_3 w_4 w_0 w_0 w_1 w_1
replicated_weight
Replicated along column_index

Figure 4.6: Memory access pattern conversion.

Model replication enables the memory access pattern that improves the locality of GPU memory accesses to increase cache utilization, which results in reducing the long latency of data transfer from DRAM. The scattered accesses to the weight vector from the thread processing the row 1 in Figure 4.3 become the sequential accesses to replicas on the replicated_weight vector as shown in Figure 4.6 so that the GPU core performs sequential mem-

[2]Detailed configuration for the evaluation is provided in Section 4.4.

ory accesses instead of divergent memory accesses.

However, model replication introduces the new overhead of data synchronization between model replicas. While replication has the advantage of reducing the memory divergence overhead, this new overhead can reduce the benefit. In fact, model replication can suffer from a large number of synchronizations during the iterative execution. The same weight value is replicated in multiple places and they have to be updated together, whenever the weight value is changed. Iterative learning application produces hundreds of intermediate results, which means hundreds of data synchronization. Even worse, conventional strict consistency prevents from reading different values from a single weight, which results in serializing all GPU threads.

4.3 Solution

4.3.1 Utilizing Stale Value Tolerance

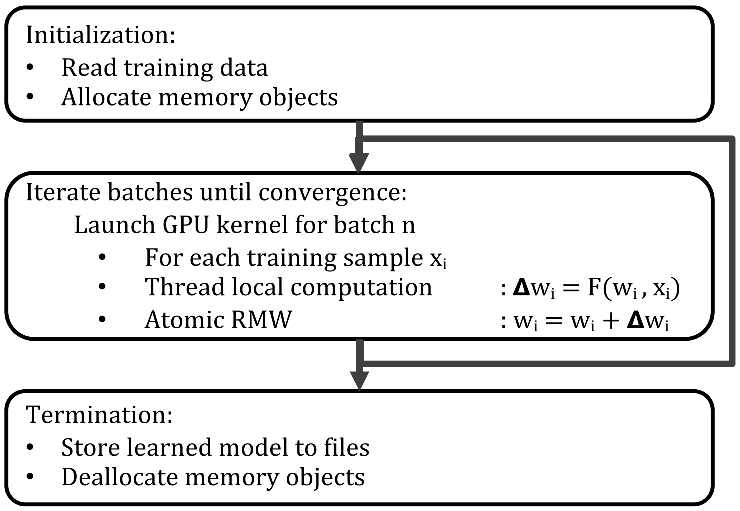
We find that relaxing the consistency can play a pivotal role in improving the performance of iterative GPU learning. The iterative-convergent characteristic allows GPU threads to use stale values in their intermediate computations, without requiring that all threads always have a consistent view of memory [15, 16, 17, 18, 11]. Unlike classical applications that are transaction-centric and deterministic with strict consistency requirements, using stale values in iterative GPU learning does not affect the correctness of applications, thus relaxing strict read-and-write data dependencies.

Our proposal, *StaleLearn*, takes advantages of this stale value tolerant characteristic of GPU learning to reduce the data synchronization overhead of model replication. The stale value tolerant characteristic allows multiple replicas to have different intermediate values, relaxes synchronization requirement, thereby allowing them to synchronize in the coarse-grained manner. Here, the coarse-grained synchronization effectively eliminates the biggest downside of model replication.

The coarse-grained synchronization enables clear task decomposition between the GPU and PIM. StaleLearn takes advantages of existing PIM technologies for the coarse-grained synchronization and exploits parallelism between PIM and GPU cores. GPU threads do not need to stall during the data synchronization on PIM; instead, they can continuously access the stale replicated data and continuously learn. The synchronization is not on the GPU execution critical path, enabling high performance.

4.3.2 Mechanism Overview

The key ideas of StaleLearn are 1) the thread local computation with stale model replicas, and 2) the asynchronous synchronization on PIM. The asynchronous synchronization consists of 1) non-blocking atomic RMW operation on the original weight and 2) replica synchronization that applies the recent value of original weight to the replicas.



Initialization:
• Read training data
• Allocate memory objects
Iterate batches until convergence:
Launch GPU kernel for batch n
• For each training sample x_i
• Thread local computation : $\Delta w_i = F(w_i, x_i)$
• Atomic RMW : $w_i = w_i + \Delta w_i$
Termination:
• Store learned model to files
• Deallocate memory objects

Figure 4.7: Learning of the baseline GPU implementation.

Figures 4.7 and 4.8 compare how StaleLearn changes the GPU learning for the example of the parallel SGD (highlighted in red). GPU learning iteratively invokes GPU kernel, where each kernel invocation forms a batch (iteration). For StaleLearn, the initialization phase of learning involves additional memory allocation for model replicas, and the finalization phase involves deallocating this model replica memory object. Following provides the overview of how iterative learning phase changes with StaleLearn.

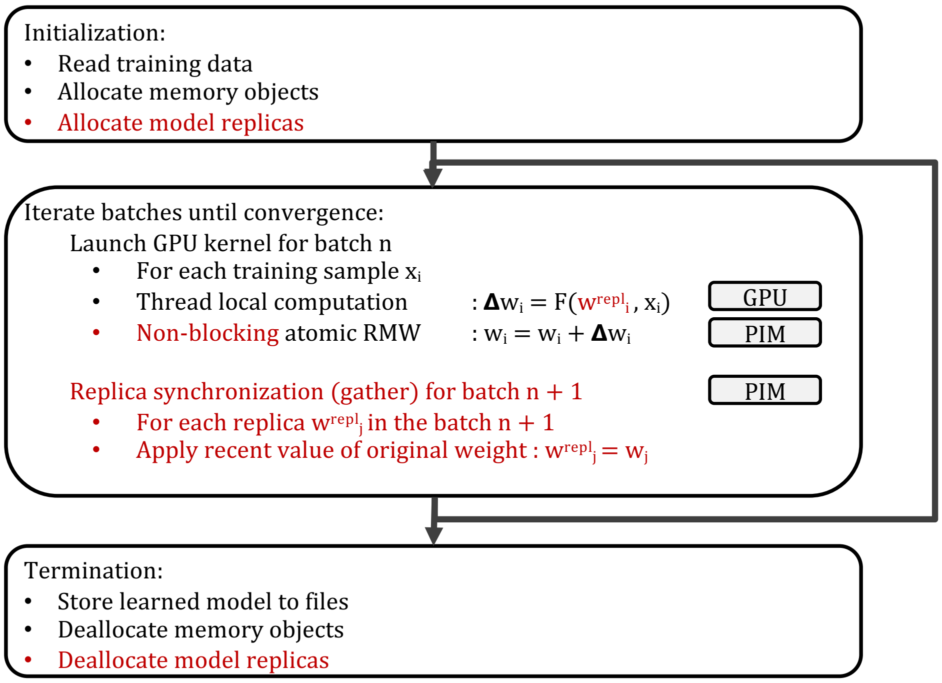

Initialization:
Read training data
Allocate memory objects
Allocate model replicas
Iterate batches until convergence:
Launch GPU kernel for batch n
For each training sample x_i
Thread local computation : Δw_i = F(w^repl_i, x_i)
Non-blocking atomic RMW : w_i = w_i + Δw_i
GPU
PIM
Replica synchronization (gather) for batch n + 1
PIM
For each replica w^repl_j in the batch n + 1
Apply recent value of original weight : w^repl_j = w_j
Termination:
Store learned model to files
Deallocate memory objects
Deallocate model replicas

Figure 4.8: Learning of StaleLearn.

1. Thread local computation: Each thread in the GPU kernel calculates Δw by accessing the its own replicated weight, w^{repl}. The divergent accesses on the original weight, w, are replaced by the sequential accesses to the thread-private replicated weight, w^{repl}.

2. Non-blocking atomic RMW: The non-blocking atomic RMW updates the original weight w using the calculated delta Δw. The GPU kernel offloads the atomic RMW to PIM with an instruction granularity.

3. Replica synchronization: Replica synchronization is a sequence of gather operations that applies the recent values on the original weight to replicas for the next batch. The programmer offloads the gather operation to PIM with a function granularity. The operation is overlapped with GPU kernel execution instead of waiting for the completion of the GPU kernel similar to software pipelining.[3]

[3]To simplify the example, we do not show the prologue and epilogue of the replica synchronization.

4.3.3 Utilizing PIM Technology

StaleLearn offloads the atomic RMW operation and the gather operation to PIM, which are handled by the atomic unit [61] and the gather unit [66]. The memory system consists of multiple channels, and each channel has dedicated atomic units and gather units. Memory controllers, atomic units, and gather units are located on the logic layer of the 3D stacked memory.

The communication of GPU and PIM is performed by extending the memory command with PIM commands as in most PIM proposals [84, 57, 85, 86]. The possible PIM command types are $atomic$, $gather\_read$, and $gather\_write$. All PIM commands contain 1) the command type, and 2) the size of the memory object. The size of the memory object is maintained since the granularity of the memory command is the cache line granularity.

4.3.4 Non-blocking Atomic RMW

On StaleLearn, the model update is performed by atomic RMW operations on the original weight. While divergent memory accesses during thread local computation are eliminated by accessing thread-private replicas, performing RMW operation on the original weight on the GPU is still expensive since it involves divergent memory accesses.[4] StaleLearn utilizes that the atomic RMW operation can be overlapped with GPU computation for the same reason as asynchronously updating replicas. Since reads from GPU threads are performed on stale model replicas, the atomic operation is not on the GPU execution critical path, which enables to perform the atomic operation asynchronously so that other threads can observe the recent value later. RMW operations are mostly memory operations with very simple computation, they can be easily offloaded to PIM. On StaleLearn, the RMW operation on the original weight w is non-blocking, such that GPU threads do not wait for the completion of the operation on PIM.

[4]We perform RMW operations on the original weight since it reduces the amount and the complexity of data synchronizations than performing the operation on replicas. A detailed discussion is provided in Section 4.3.7.

Programming Interface: On StaleLearn, the programmer modifies the code to call the new non-blocking atomic RMW builtin *PIMAtomic*. Here, T refers to the data type (e.g., double). The builtin function takes three arguments. The argument *dest* defines the destination of the RMW operation. The argument *op* defines the type of RMW operation to be performed, and the argument *val* defines the value to be used when performing the RMW operation.

```
PIMAtomic(T* dest, OP op, T val)
```

Available op argument values (*enum OP*) are as follows.

- *INC/DEC*: Increase/decrease the value at *dest* by *val*.
- *MIN/MAX*: When *val* is smaller/larger than the value at *dest*, store *val* at *dest*.

Figure 4.9 shows an example of GPU kernel modification for the example of parallel SGD. The code modification is negligible, which provides an easy integration of existing GPU code to utilize the PIM technology.

```
// Replicas are stored along the column from "start" to "end"
for (int i = start ; i < end ; ++i) {
  ...
  // Compute w_delta using the replicated weight
  double w_delta = compute_delta(w_repl[i] , ...) ;
  ...
  int c_id = column_index[i] ;
  // Non-blocking RMW on the original weight
  PIMAtomic(&(w[c_id]) , INC , w_delta) ;
}
```

Figure 4.9: GPU kernel modification for StaleLearn.

PIM Offloading: The non-blocking atomic RMW operation is initiated by GPU ISA extension. The compiler changes the *PIMAtomic* builtin into the new PIM atomic instruction. Instead of being executed on the GPU core, the PIM atomic instruction is offloaded by sending the atomic command to the memory.[5] The PIM atomic instruction retires just after sending the atomic command to the memory without waiting for the completion of the atomic operation, thereby decoupling the PIM execution from GPU computation via fire-and-forget.

4.3.5 Atomic RMW and Replica Synchronization

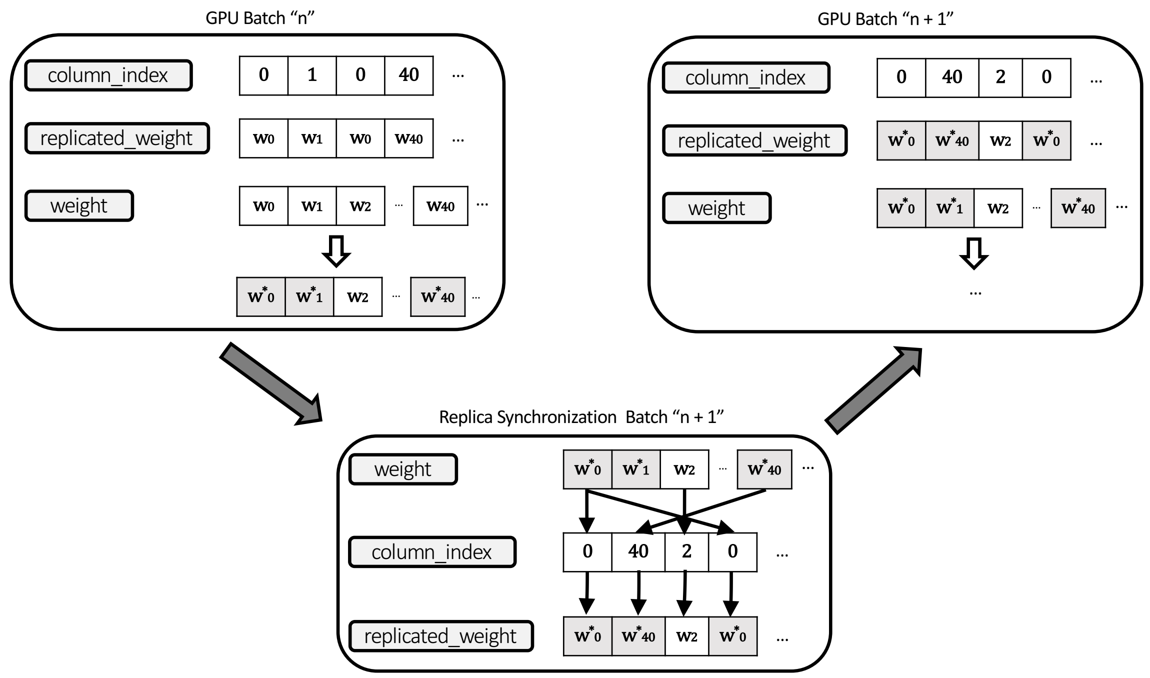


Figure 4.10: Coordination of non-blocking atomic RMW and replica synchronization.

Figure 4.10 shows how non-blocking atomic RMW and replica synchronization are coordinated. Each GPU batch reads the portion of the column_index vector thus the portion of the replicated_weight vector and performs atomic RMW on the original weight vector. In Figure 4.10, within the GPU batch "*n*", different GPU threads read thread-private stale model replicas for the batch regardless of the updates performed on the original weight during the batch. The original weight accumulates deltas from non-blocking atomic RMW

[5]Our baseline GPU system maintains write-no-allocate/write-evict L1 cache and supports write-back/write-through for shared L2 cache. Atomic commands are enqueued utilizing write-through on the L2 cache.

operation such that the values of original weights, w_0, w_1 and w_{40} are updated to the new values, w_0^*, w_1^* and w_{40}^*. After the GPU batch ”n” is over, the replica synchronization writes the value of the original weight to replicas for the next batch. So the value of multiple replicas replicating the same original weight become consistent again, such as w_0^* in Figure 4.10.

4.3.6 Replica Synchronization

Programming Interface: On StaleLearn, the replica synchronization is a sequence of gather operations that applies recent changes on the original weight to replicas. StaleLearn provides *PIMGather* API to offload the replica synchronization to PIM by passing the range of column_index vector for the next batch.

Figure 4.11 shows a pseudo code of the PIMGather API that performs a sequence of gather operations. A gather operation can be broken into three memory operations. First, the operation accesses the column_index vector to find the index on where the original weight is located. Then, the operation reads the value of the original weight. Lastly, the operation writes the recent value of the original weight into the replica on the replicated_weight vector.

```
// Column indices and replicas are stored
// between “batch_start” to ”batch_end”
for (int i = batch_start ; i < batch_end ; ++i) {
  // 1. Read the column_index
  int c_id = column_index[i] ;

  // 2. Read the original weight
  double w_value = w[c_id] ;

  // 3. Write to the replica
  w_repl[i] = w_value ;
}
```

Figure 4.11: Replica synchronization.

Gather Operation on PIM: Gather operation is one of the popular operations on PIM and already has been implemented in hardware [66]. Here, we explain how the gather unit performs the replica synchronization.

The gather unit performs the replica synchronization by sending memory commands to the corresponding channel. It utilizes the start address of 1) the weight, 2) the replicated weight, and 3) the column_index vector to identify the destination channel to send the memory command. The range of column_index vector given by the programmer is partitioned into each gather unit that has the column_index vector entries on its own channel. When multiple gather units are allocated for a single channel, they partition the address range of each channel.

The gather operation is performed by executing a sequence of memory commands: read, gather_read and gather_write commands. The gather_read command maintains the addresses of 1) the replicated weight and 2) the original weight. The gather_write command maintains 1) the address of the replicated weight and 2) the value of the original weight. Following explains the memory command sequence for the gather operation.

1. The gather unit first sends the read command to its own channel that has the column_index vector entries to find the locations of the original weight. The address of the original weight is computed using 1) the start address of the original weight vector, 2) the size of each weight value depending on the data type, and 3) the index value in the column_index vector.

2. After identifying the addresses of the original weight, the gather unit sends the gather_read command (with the replica address) to the channel where the original weight is located.

3. The channel that receives the gather_read command reads the value of the original weight and send back the gather_write command (with the value of the original weight) to the channel that holds the replica. The channel that receives the gather_write command writes the replicated weight with the value of the original weight.

Overlapping PIM execution with GPU: In a naive implementation, the programmer can sequentially perform replica synchronization for batch *n* + 1, after the GPU kernel launch for batch *n*. Simply offloading the replica synchronization on PIM can reduce the benefit of our proposal and cause GPU cores to wait for the completion of the PIM execution. In order to reduce PIM execution latency, StaleLearn exploits parallelism between PIM and GPU cores and overlaps the replica synchronization on PIM with the main computation on GPUs. The overlapped execution enables effective latency hiding of PIM execution, and GPU cores can continuously perform computation without waiting for the synchronization to finish.

For asynchronous synchronization, StaleLearn utilizes the bounded staleness consistency model [21], which has been widely used in distributed learning by guaranteeing the reasonable progress within an iteration with a user-defined staleness threshold. The staleness threshold is used to control the degree of staleness such that the GPU cores can access the recent updates on model weights. We define the degree of staleness with batch count, similar to the study by Lee et al. [87]. The staleness degree is equal to *m* when all the replicas for the batch *n* are synchronized with all updates until the batch *n* - *m*. The staleness degree 1 means that the replicated weights for the current batch are synchronized with the previous batch's update to the original weight.

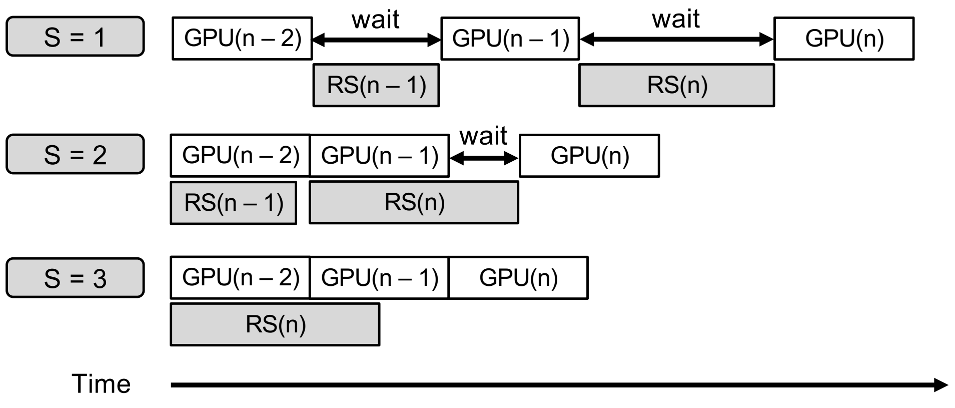


Figure 4.12: Overlapping replica synchronization.

Figure 4.12 shows how StaleLearn overlaps the replica synchronization (RS) on PIM with different staleness thresholds. The GPU performs computation for the new batch only when the replica synchronization for the batch is finished. With the staleness threshold *s*, StaleLearn starts the replica synchronization for the batch *n* after finishing execution of the batch *n* - *s*. The larger threshold can reduce GPU's waiting time, while it can lead to missing more number of updates.

4.3.7 Discussion

RMW Operation on the Original Weight: We perform RMW operations on the original weight since it reduces the amount and the complexity of replica synchronization than performing the operation on replicas. When performing RMW operations on replicas, the replica synchronization for the new batch requires accessing all replicas replicating the same weight from all previous batches, since the updates for the same weight are scattered across multiple replicas. Compared to that, the replica synchronization for the new batch only needs to access the original weight when performing RMW operation on the original weight.

Performing RMW operations on replicas also increases the complexity of replica synchronization. Since each replica should know the location of other replicas, it requires maintaining a book-keeping structure storing this information, which increases the memory footprint. When performing RMW operations on the original weight, each replica only needs to know the location of the original weight, which is already available from the column_index vector in the compressed sparse row format.

Memory footprint of Replication: The decision of maintaining replicas *clean* helps to reduce the memory footprint cost of replication. Not all replicas need to be allocated on memory since all updates are accumulated on the original weight. StaleLearn requires only replicas for *s* batches to be allocated with the staleness threshold of *s*, which leads to the negligible memory footprint overhead from replication as we will discuss in later sections.

4.4 Experimental Methodology

4.4.1 Workloads

Table 4.1 shows our evaluated workloads. We categorize several learning applications into two main categories: parallel gradient descent and graph primitives.[6] Three applications are evaluated for the gradient descent method, a popular method in supervised learning and recommender systems: binary classification (*BC*), linear regression (*LR*), and low-rank matrix factorization (*MF*). BC and LR learn weight vectors for high dimensional data with a large number of features. MF learns two matrices, L and R, such that LR is approximately equal to an input matrix X. We also evaluate popular graph primitives that share multiple characteristics with model learning: connected component (*CC*), graph coloring (*CLR*), breadth first search (*BFS*), single source shortest path (*SSSP*), and number of path (*NP*). Multiple studies [20, 29, 30] have proven the stale value tolerance of iterative graph primitives including GraphLab [4] that transforms ML applications into graph algorithms.

Table 4.1: Evaluated workloads.

Workload	**Learning Algorithm**	**Input**
BC	SGD, Mini-batch GD	news20.binary [88]
LR	SGD, Mini-batch GD	news20.binary [88]
MF [19]	SGD	news20.binary [88]
CC [20]	Single writer, Multiple writer	LBDC-1000K [89]
CLR [46]	Single writer	coAuthorsDBLP [74]
BFS [72]	Single writer, Multiple writer	LBDC-1000K [89]
SSSP [72]	Single writer, Multiple writer	LBDC-1000K [89]
NP [20]	Single writer	coAuthorsDBLP [74]

We focus on data-parallel implementations of these applications: MF is parallelized across non-zero features, and other applications are parallelized across training data points (or vertices). For BC and LR, we evaluate both stochastic gradient descent (S) and mini-batch gradient descent (M) with the batch size of 2048 that equals to the number of available

[6]The OpenMP version of applications are available at https://github.com/gthparch/stale_workload, which can be downloaded and be used for other evaluations.

GPU threads. For CC, BFS, and SSSP we evaluate both single writer implementations (S) and multiple writer implementation using atomics (M). Multiple writer implementation can reduce the number of iterations before convergence than single writer implementation by performing more writes per iteration.

We use real data sets with the compressed sparse row format. coAuthorsDBLP has 5.03 columns per row and 388788 weight values. LBDC-1000K has 28.8 columns per row and 1000000 weight values. news20.binary has 456 columns per row and 1355192 weight values.

4.4.2 Hardware Configuration

For evaluating our proposal, we utilize an in-house execution-based simulator. We have developed a lightweight simulator to evaluate how utilizing stale value affects convergence, completion time of learning, and the solution quality. We (have to) execute the application until it converges to see the final value, which forbids us from using a conventional cycle-level simulator. The simulator is a combination of an execution-based functional simulator with a timing model mimicking NVIDIA-like GPU: 32 threads in a warp execute in lockstep, branch/memory divergence is modeled, and GPU cores execute multiple warps in a round-robin fashion while other warps are waiting for memory. Both the core and the memory hierarchy are modeled in detail.

Table 4.2 shows the hardware configuration. Each core consists of the execution pipeline and the private L1 data cache.[7] The L2 cache is shared by all cores and supports both write-through and write-back policies. Both the L1 and L2 cache are non-blocking and utilize LRU replacement policy. The memory model is based on the HBM model [90] with the timing parameters from Ramulator [91]. Both the logic layer and the memory layer operate at a 0.5 GHz cycle. The memory access granularity and cache block size are 64 bytes. The logic layer of HBM has four atomic units and four gather units per channel. The

[7]We assume infinite instruction cache without cache misses and L2 cache contention.

Table 4.2: hardware configuration.

GPU core	8 GPU core, 1.6 GHz 8 32-wide warps, 32-wide SIMD width
L1 D cache	16 KB, 2-way assoc, 2 cycles Write-no-allocate/write-evict
L2 cache	128 kB, 16-way assoc, 16 cycles Write-no-allocate, write-back/write-through
Main Memory	HBM, 0.5 GHz, 64B access 8 channels, 1 rank per channel 1000 MT/s, 119.2 GB/s, CL-RCD-RP : 7-7-7
PIM units	4 atomic/gather units per channel
LAMAR [49]	4 sectors per cache block, 4 sub-ranks Static decision by programmer

ISA extension is modeled by special instructions to call functions on the simulator.

We compare StaleLearn against LAMAR [49] with the sector cache [92] and the sub-ranked memory system [93], which adjusts access granularity depending on the locality of memory accesses to increase the memory bandwidth utilization efficiency. The sector cache partitions cache block into four sectors and the sub-ranked memory reduces the minimum DRAM access granularity by four times. We implement LAMAR on the baseline HBM memory model. In our evaluation, LAMAR performs fine-grained accesses on model parameters, while it performs coarse-grained access for other memory objects, which is the optimal access granularity regarding memory bandwidth utilization.

4.4.3 Measurement

The iterative learning is a fine-tuning procedure to gain better-quality solutions, which requires StaleLearn to guarantee the same quality solution of the baseline implementation. For each workload, we measure the learning completion time,[8] which is defined as the time until the solution converges to the same quality solution to that of the baseline implementation. We execute the baseline implementation until there is no progress regarding the solution quality.

[8]The execution time of inference is determined by the size of the model and input, which is not affected by our proposal.

4.5 Experimental Results

4.5.1 Overall Performance Improvement

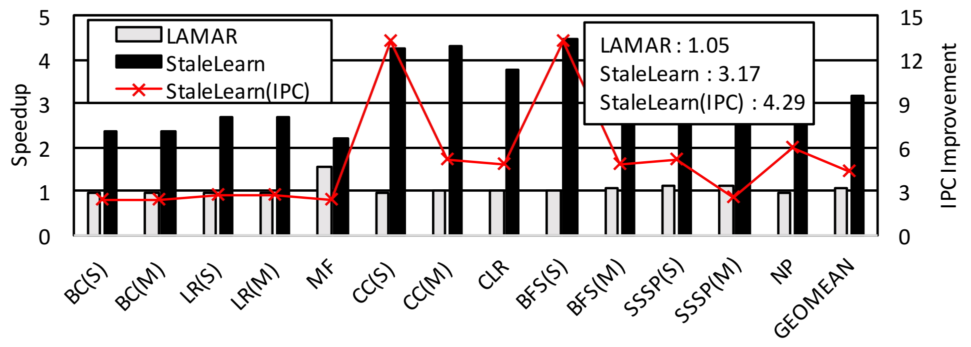


Figure 4.13: Speedup of LAMAR and StaleLearn.

StaleLearn significantly improves the performance of GPU learning. Figure 4.13 shows the performance improvement with LAMAR and StaleLearn for all evaluated workloads with the best staleness threshold for each workload. On average, StaleLearn outperforms the baseline implementation by 3.17 times, while LAMAR only improves the performance by 1.05 times.[9] The large IPC[10] improvement by StaleLearn (4.29 times as shown in Figure 4.13) leads to the overall speedup even though the loss of intermediate accuracy by allowing stale values slightly increases the number of iterations before converging to the same quality solution of the baseline implementation. (discussed in Section 4.5.3).

While the optimal staleness threshold is different for each workload, Figure 4.14 shows that StaleLearn can achieve similar benefit with a single staleness threshold across all workloads. The average benefit of using the same staleness threshold 3 is 3.07, which is similar to 3.17 with a different staleness threshold for each workload.

4.5.2 Reducing Data Communication Cost

Reducing Cache Misses: StaleLearn reduces the hardware data communication cost be-

[9]In the chapter, for all workloads, the baseline is the original workload, but still using HBM as main memory.

[10]The IPC is the number of warp instruction per cycle.

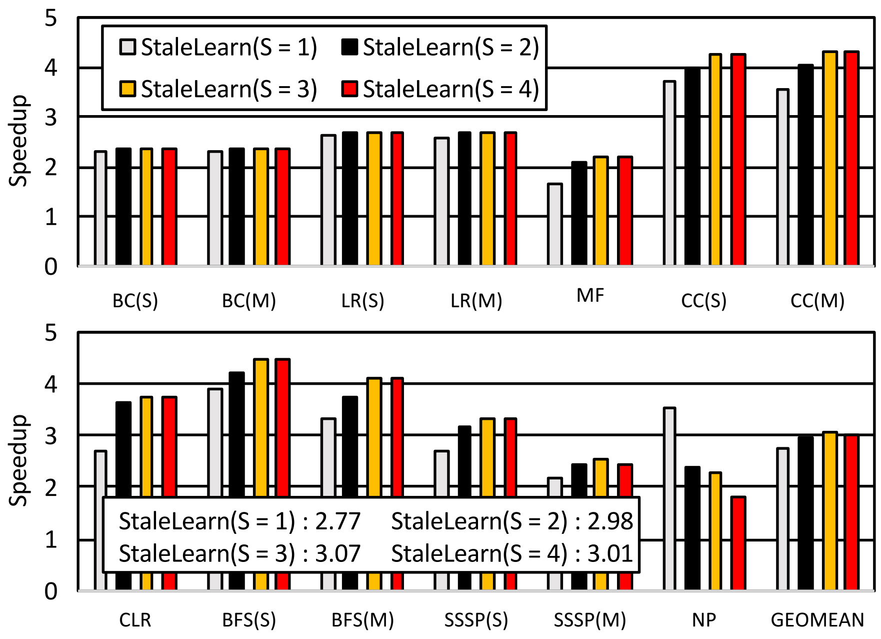


Figure 4.14: Speedup of StaleLearn with different staleness thresholds.

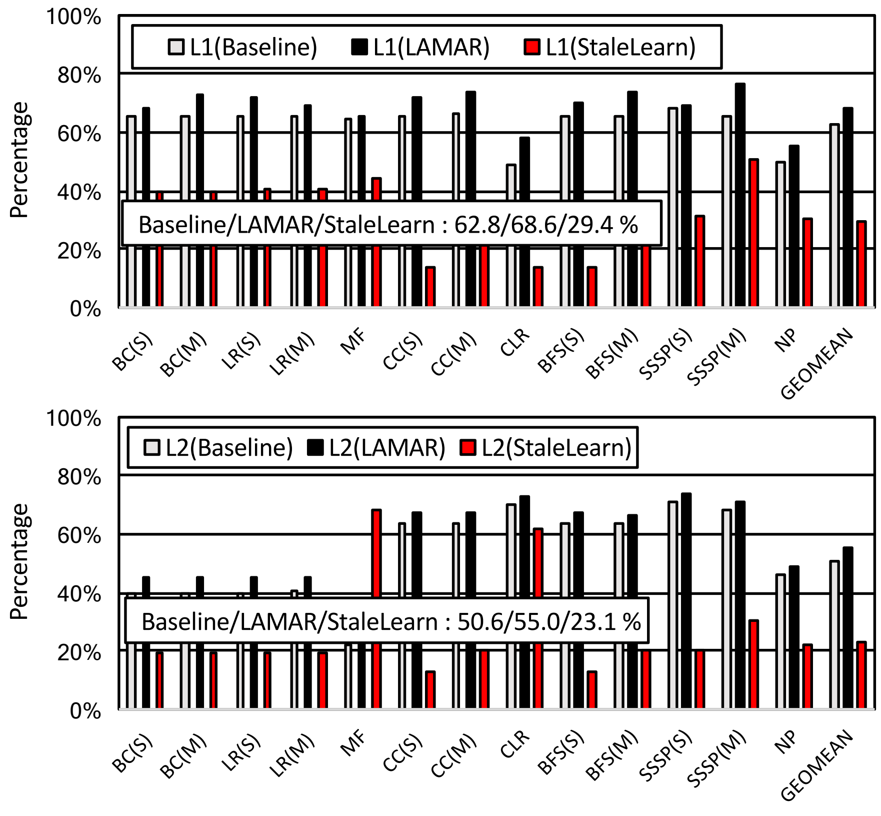


Figure 4.15: Comparison of cache miss ratio of different implementations.

tween the GPU core and memory by better cache utilization. StaleLearn reduces the number of cache misses by reducing the number of divergent memory accesses. Figure 4.15 compares the cache miss ratio of different implementations. On average, StaleLearn reduces the L1/L2 cache miss ratio from 62.8/50.6% of the baseline to 29.4/23.1%, while LAMAR increases the cache miss ratio to 68.6/55.0%. Unlike LAMAR that under-utilizes the spatial and temporal locality of cache (while it is small so that reducing access granularity can potentially increase bandwidth utilization efficiency), StaleLearn increases the locality of the cache accesses, causing subsequent cache accesses to be cache hits. Each GPU thread has an inner loop accessing different locations of the weight vector. With replication, the inner loop accesses are transformed from scattered memory accesses to sequential accesses. Therefore, when the inner loop advances to the next memory location, the required cache line will be present in the cache from the previous access, resulting in a cache hit.

Reducing Cache Accesses: Transforming divergent memory accesses into sequential accesses not only reduces the cache miss ratio but also leads to better coalescing. Since the size of the weights accessed by a GPU thread is mostly smaller than a cache line size, with StaleLearn, the memory locations that neighboring threads in a warp access reside in the same cache line, resulting in a coalescing degree increase.[11] Figure 4.16 shows the reduced number of cache accesses with StaleLearn. Unlike LAMAR that increases the L1/L2 caches accesses to 105.8/115.5% of the baseline by under-utilizing cache locality, better coalescing enabled by StaleLearn reduces the number of L1 cache accesses to 70.7% of the baseline. StaleLearn also reduces the L2 cache accesses to 52.3% of the baseline implementation with a reduced L1 cache miss ratio and better coalescing.

Reducing DRAM Traffic: The reduced number of cache accesses and misses lead to the overall DRAM traffic reduction. Figure 4.17 shows that StaleLearn reduces the amount of DRAM traffic to 53.2% of the baseline implementation while LAMAR only reduces it

[11]The coalescing degree is the average of #active_threads / #memory_events per warp load instruction. E.g., 32 active threads that are satisfied by 8 cache transactions would have a degree of 4.

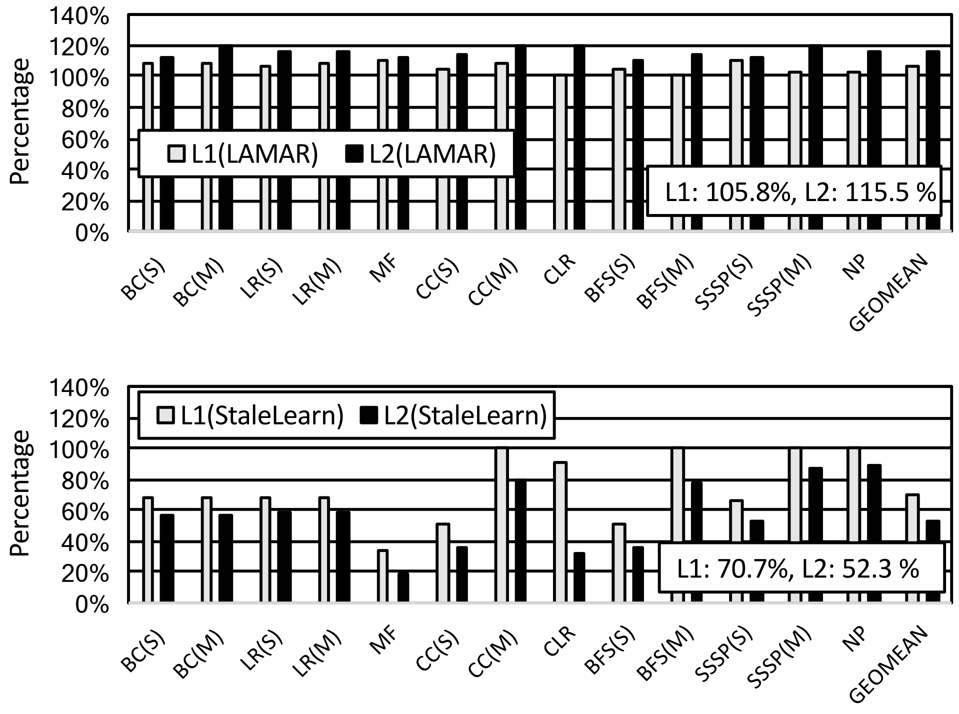


Figure 4.16: Reduction of cache accesses with StaleLearn.

to 79.3%. While StaleLearn involves additional DRAM transactions with asynchronous synchronization, the amount of the DRAM traffic for synchronization on PIM (33.2%) is smaller than the reduced amount of DRAM traffic from the GPU (100 - 19.1 = 80.9 %), thus reducing overall DRAM traffic.

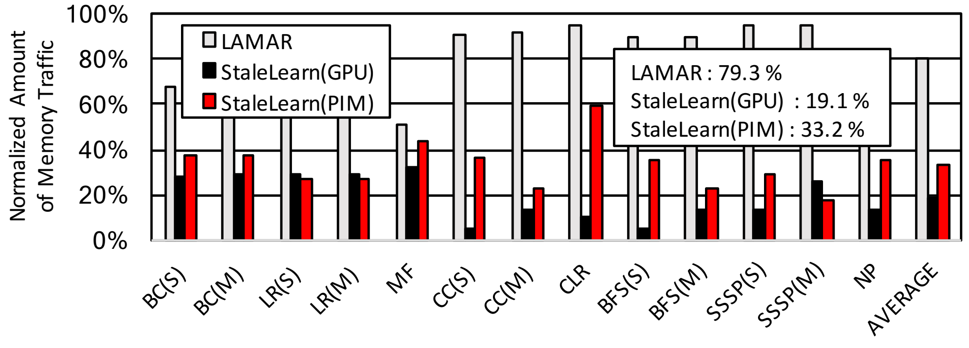


Figure 4.17: DRAM traffic reduction with StaleLearn.

Overlapping PIM execution with GPU: The reduction of hardware data communication cost is enabled by asynchronous synchronization. Overlapping PIM execution with GPU execution enables hiding the long synchronization operation latencies. Figure 4.18

shows the reduced replica synchronization time via overlapping the replica synchronization with GPU computation. We set the total execution time of the baseline implementation as 100%. Even though the GPU execution time is increased from 18.4% of the baseline to 23.3% via overlapping due to contention on memory, the replica synchronization time is reduced to 0.2% from 8.4% of non-overlapped execution of replica synchronization. Thus, the overall execution time is reduced from 26.8% to 23.5% by overlapping.

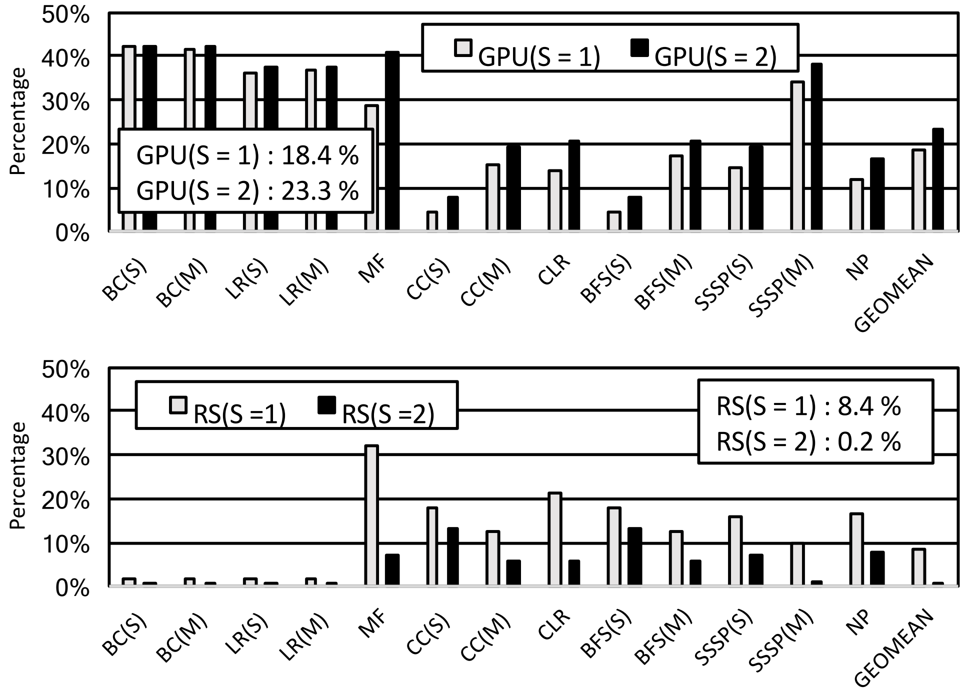


Figure 4.18: Reduced replica synchronization time via overlapping.

4.5.3 Stale Value Tolerance

While utilizing stale values can potentially lead to performance degradation by increasing the number of iterations before convergence, our evaluation reveals that the increase of iteration count is minimal. Figure 4.19 shows that the iteration count is only increased by 5% on average, with the best staleness threshold for each workload. For different staleness thresholds from one to four, StaleLearn only requires 5/9/10/13% more iterations before converging to the same quality solution of the baseline implementation.

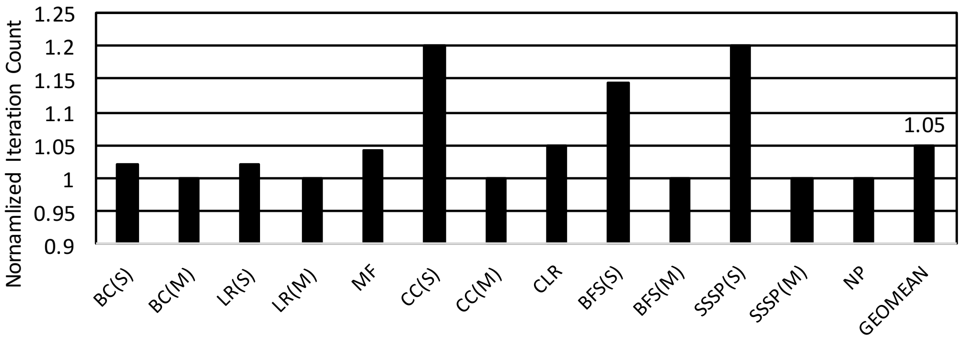


Figure 4.19: Iteration count with StaleLearn.

The small increase is due to the limited number of missing updates with sparse model updates. The sparse nature of training data leads each iteration to access and modify only a fraction of the entire model. As it is unlikely GPU will miss a large number of updates, the progress per iteration is not affected much. Figure 4.20 compares the number of missing updates per weight of the baseline implementation and that of StaleLearn with the staleness threshold of four, for all evaluated workloads except BC (S) and LR (S). On average, the number of missing updates per weight is only increased from 0.42 to 1.29.[12] The sparsity is also found on multiple writer implementations of CC, BFS, and SSSP that perform more writes per iteration than single writer implementation. This sparse characteristic has been utilized in multiple parallel learning studies [15, 16, 17, 18, 19, 20] such as Hogwild! [8] that performs lock-free model updates. Hogwild! assumes that useful information will not be overwritten by other threads due to this sparse nature of model updates.

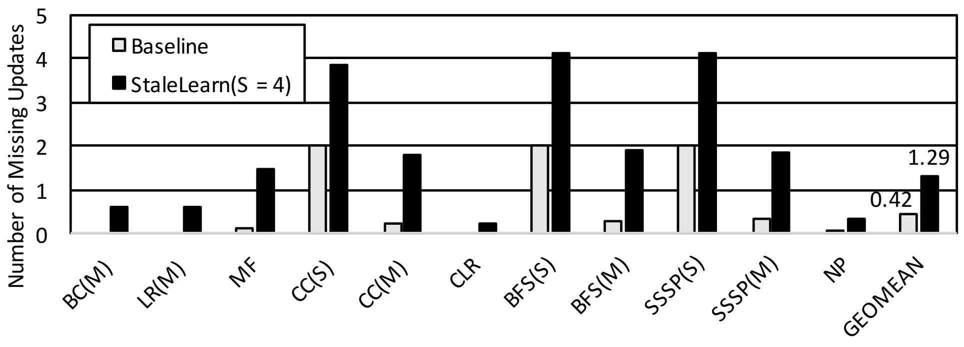


Figure 4.20: Comparison of the number of missing updates per weight.

[12]The baseline also misses updates due to parallel execution.

We also find that the number of iteration is not much affected for the SGD implementations of BC, and LR that miss more updates than other workloads. For BC(S)/LR(S), the number of missing updates is increased from 54.4/53.8 of the baseline implementation to 110.6/111.7 of StaleLearn with the staleness threshold four. The small increase of iteration count for these workloads is due to the convex property of the problem [9], which has been widely utilized in distributed learning studies [7, 19, 94, 95, 96].

4.5.4 Comparison

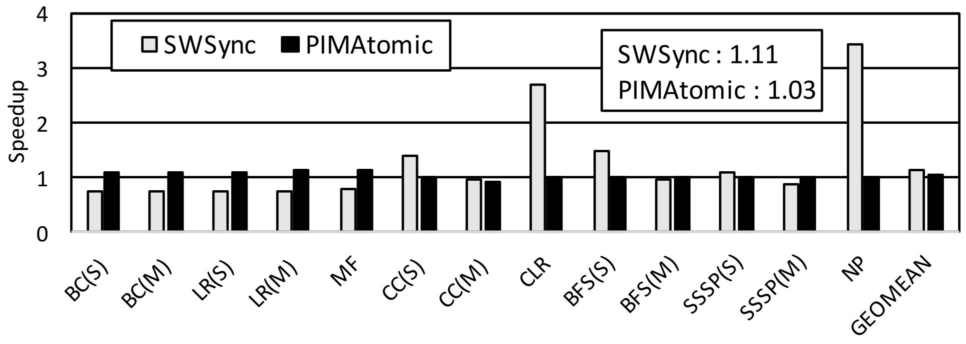


Figure 4.21: Speedup of SWSync and PIMAtomic.

While a software-only mechanism utilizing helper kernels for replica synchronization is also possible. it is neither realistic nor beneficial for improving the performance of learning. Figure 4.21 shows that the software-only approach performing the replica synchronization on GPU only improves the performance by 1.11 times. It is because the replica synchronization itself is memory intensive with low locality causing GPU cores to stall. The GPU needs to fetch different memory objects from memory only to write back the values in a different order, which can further increase the cache contention and lead to under-utilizing GPU resources.

Our evaluation also reveals that simply offloading atomic RMW operation to PIM does not help performance much. Figure 4.21 shows that non-blocking atomic RMW operation without replication (PIMAtomic) only improves the performance by 1.03 times. For most evaluated workloads, the performance bottleneck is the memory access to the current model parameter, not the RMW operation for the model update. The memory access during RMW operation would be a cache hit, while it would be a cache miss for the memory access to the current model parameter. Also, just offloading atomic RMW operation to PIM can increase the amount of DRAM traffic for the workload such as BFS and SSSP where the update is not always performed, while StaleLearn can check replicas to identify if the update is needed.

4.5.5 Energy Reduction

We also model the energy consumption of StaleLearn. The power and energy consumption of different architecture components is summarized in Table 4.3. We use the modeling equations from the study by Kim et al. [65], which we omit for brevity. We decompose the overall energy consumption into five categories: GPU_Logic, GPU_Cache, HBM_Logic,[13] HBM_Memory_Static, and HBM_Memory_Dynamic.[14]

Table 4.3: Power and energy parameters for each component.

Host	GPU core active power	10 W per core
	GPU uncore power	1 W per channel
	SRAM leakage power	32.4 nW per byte
	L1 access energy	0.494 nJ per access
	L2 access energy	3.307 nJ per access
Memory	HBM logic(core) power	640 mW per channel
	HBM logic(uncore) power	2.890 W
	HBM memory background power	0.479 W
	HBM memory access energy	28.034 nJ per access
Transfer	TSV_XFER energy	0.624 pJ per byte

Figure 4.22 compares the energy consumption of each application. The energy consumption is normalized to the baseline implementation. Each bar further details the energy consumed by each architectural component. The energy savings are realized across all evaluated workloads with StaleLearn. On average, StaleLearn greatly reduces the overall energy consumption to 43.2% (16.1/5.1/5.4/0.3/16.2%) of the baseline implementation,

[13]HBM logic(core) power is turned on only for StaleLearn.
[14]TSV data transfer energy is included.

while LAMAR only reduces the energy to 92.0% (50.0/9.1/6.0/1.0/25.8%).

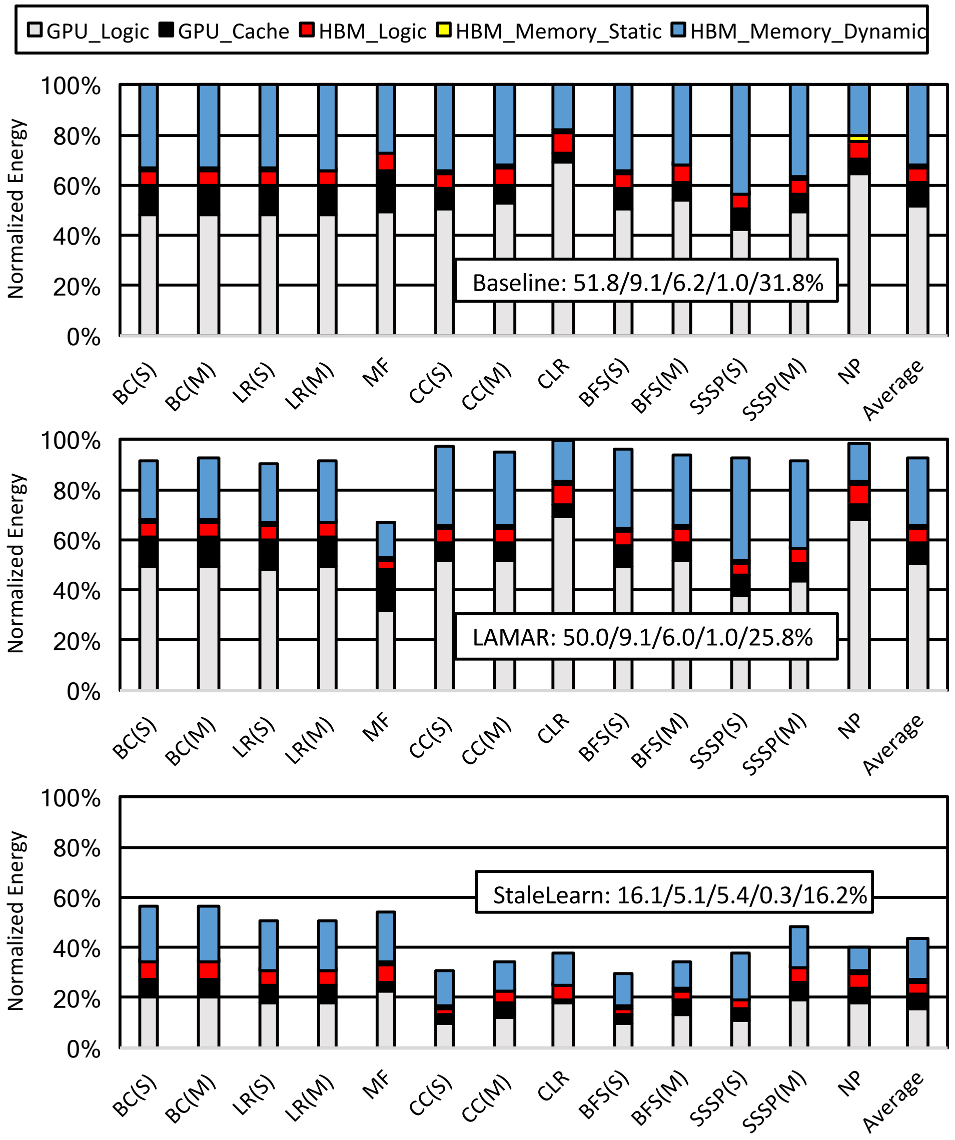


Figure 4.22: Energy consumption of different implementations normalized to the baseline.

The energy saving is obtained by the reduced execution time and reduced DRAM accesses. The reduced execution time of StaleLearn clearly reduces the static energy consumption. Even though the amount of work is increased by allowing stale values as represented by the increased iteration count as shown in Figure 4.19, because the overall execution time is reduced significantly, the overall energy consumption is reduced. StaleLearn also reduces the dynamic energy of HBM_Memory since it reduces the amount of overall

DRAM traffic as shown in Figure 4.17.

4.5.6 Discussion

Application to DNN: Multiple ML applications suffer from low locality memory accesses during learning. While we evaluate relatively simple kernels due to simulation time concern, these kernels are found in multiple state-of-art ML applications such as the deep neural network (DNN). In fact, the sparse model access is universal on the embedding layer of DNN. The embedding layer maps a large number of low-level features to the small number of high-level features, and this mapping is also learned during learning. On multiple domains, a large number of low-level features are used such as the recurrent neural network (RNN) based language model [2, 97] that maintains the embedding layer whose size equals to the number of vocabularies. The memory access for this layer during learning is divergent since each sentence only holds a limited number of words. Multiple ML studies utilize the sparse characteristic such as Hogwild! [8], a lock-free updating scheme assuming non-overlapped access to the same model parameter from different threads.

Staleness Threshold Selection: While the staleness threshold is set by programmers, it does not increase the programmer's burden too much. With sparse data, the increase of the iteration count with StaleLearn is minimal due to the limited number of missing updates. So, the staleness threshold can be easily determined by simple profiling to measure replica synchronization time. Our evaluation also revealed that StaleLearn still achieves large benefit with a single staleness threshold across all workloads.

Even when the progress per iteration is affected by utilizing stale values, the programmer's burden is minimal. Since many ML applications share the common learning pattern with minor differences (i.e. only differing on the model size), the staleness threshold for a new application can be easily computed with simple projection from other applications.

In fact, ML algorithms are rarely hyperparameter-free. ML algorithms already require careful tuning of a large set of hyperparameters such as the learning rate, and the regularization strength of a gradient-based optimization, which significantly affect the convergence of ML applications. This hyperparameter tuning is considered as the optimization of an unknown black-box function, and it often requires a time-consuming grid search procedure [98, 99, 100]. Lots of efforts have been devoted to efficient hyperparameter selection including the work by Bergstra and Bengio [101] proposing the random search for hyperparameter optimization. Multiple proposals propose to utilize bayesian optimization to automate the hyperparameter selection [102, 103, 104, 105, 106].

Replication Overhead: The overhead of replication on memory footprint is negligible. We replicate only the model parameter suffering from divergent memory accesses. For example, we replicate only the weight vector, not the training data, for BC/LR. For these objects, the number of replicas is proportional to the number of valid features of each training data point, which is limited due to the sparse nature of real-world data. Not all replicas need to be allocated on memory, but only replicas for s batches need to be allocated with the staleness threshold of s. In our evaluation, the memory footprint is increased only by 1.7/3.4/5.1/6.7% from the baseline implementation when we sequentially increase the staleness threshold from one to four. Most ML applications involve multiple properties that are not replicated, such as user profile, which we exclude in our evaluation.

Code Modification: The code modification is simple only requiring to change the weight vector access in GPU kernel and call PIMGather API function for replica synchronization. Multiple ML applications share similar memory access patterns so that the code modification will be similar to them. As most ML applications utilize common framework, the code modification only needs to be done for the framework, and multiple applications can benefit from StaleLearn without any code modification.

4.6 Summary

GPU learning is memory bound with divergent memory accesses, thus causing GPU cores to stall. While considerable effort has been devoted to reducing the overhead of divergent memory accesses, inefficient utilization of the unique, stale value tolerant characteristic of learning applications has prevented previous studies from achieving the full performance potential of the GPU architecture.

In this chapter, we propose StaleLearn, an effective learning optimization to overcome the memory divergence overhead. StaleLearn replicates the model and performs asynchronous synchronization on PIM. The efficient task decomposition between the GPU and PIM enables StaleLearn to exploit parallelism between PIM and GPU cores. Our evaluation shows that StaleLearn accelerates representative GPU learning applications by 3.17 times with existing PIM proposals.

CHAPTER 5
SYSTEMATIC METHOD TO REDUCE REDUNDANT COMMUNICATIONS UTILIZING STALE VALUE TOLERANCE

5.1 Overview

The stale value tolerant characteristic allows parallel workers to execute asynchronously by reducing redundant data communications to improve the scalability of modern data-intensive learning. However, the current learning suffers from low scalability thus leading to inefficient utilization of millions of computing machines that are currently available in the era of cheap compute.

The current ML studies lack the detailed understanding of how utilizing stale values affects the progress of parallel learning. Different ML applications exhibit different degrees of stale value tolerance, and the stale value tolerance is determined by the complex function of multiple factors. The complicated impact of utilizing stale values on the progress and the solution quality of parallel learning has caused previous studies to make ambiguity for domain experts thus limiting the scalability of parallel learning.

For this challenge, this dissertation proposes a systematic method to define the stale value tolerance of ML application. We study the stale value tolerance of the recurrent neural network (RNN), which is becoming a standard of deep learning. Our proposal defines the stale value tolerance with the effective learning rate, which is different from the explicit learning rate specified by the programmer and can be changed by multiple design choices of training neural network. Definition of the stale value tolerance of different ML applications with the effective learning rate can provide insights and lead to potential software optimizations with further benefit to future hardware choices. In summary, the key contributions in this chapter are as follows:

1. We observe that the redundant data communications in parallel learning have reduced the scalability. We find that, while maintaining model replicas with relaxed synchronization has the potential to improve the scalability, the ambiguity regarding how it affects the progress of learning has led to inefficient utilization of this potential.

2. We devise a systematic methodology to define the stale value tolerance of different ML applications with the effective learning rate. The effective learning rate is decided by i) implicit momentum hyperparameter, ii) the update density, iii) the activation function selection, iv) RNN cell types, and v) the learning rate adaptation.

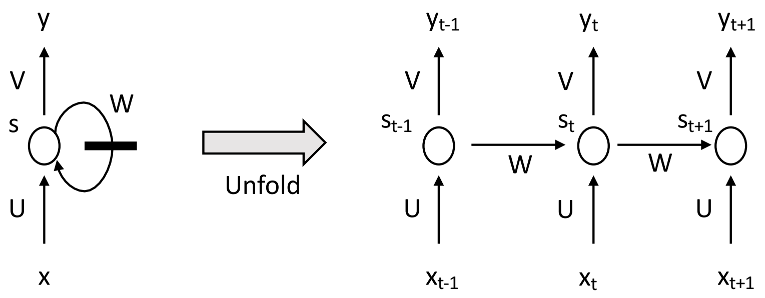
5.2 Reducing Redundant Communication

In this section, we provide the necessary background on reducing redundant communications with the example of the RNN. First, we provide the background of RNN. Then, we discuss the gradient-based back-propagation through time (BPTT). After discussing how to reduce data communication utilizing the stale value tolerance, we examine the performance potential and challenge of reducing data communications.

5.2.1 RNN Basic

Developing an efficient ML model has become a hot topic recently. RNN is becoming a standard of deep learning due to its equal representation power as long layer sequence of DNN, and the benefit of less number of weights that helps to reduce the memory footprint. RNN is now widely used not only in the domain where it has been used such as NLP and speech [34, 97] but also in computer vision where the deep convolutional neural network (CNN) has been widely used [107, 108, 109].

Figure 5.1 shows the diagram of RNN for the language model. It consists of three layers: x (input), s (recurrent), and y (output) layers. The hidden layer s is recurrent and unfolded multiple times in the time sequence to have the comparable representation strength as DNN. Here, sigmoid activation function is used at the recurrent layer and softmax function is used for the output layer. During learning, three weight vectors are learned: U, W, and V vector. U is the weight vector for the relationship between the input and hidden layer, and V is the weight vector for the relationship between the hidden and output layer. Without the recurrent weights W, this model would be a bi-gram neural network language model.



$$s_t = \text{sigmoid}(Ux_t + Ws_{t-1})$$
$$y_t = \text{softmax}(Vs_t)$$

Figure 5.1: RNN for language model and recurrence.

Figure 5.2 represents the dimension of layers and weights of the RNN shown in Figure 5.1. The size of the input layer x and the output layer y is proportional to the vocabulary size C (usually around 10K - 200K). Hidden layer s is orders of magnitude smaller (H, usually around 50 - 1000 neurons). The size of U and V weight vectors is $R^{C \times H}$, and the size of W weight vector is $R^{H \times H}$. So the RNN learning needs to learn $2HC + H^2$ parameters. For the case of English Penn Treebank benchmark (PTB) [110] with C = 10000 and H = 256, the size of U and V vector is 20 MB, and the size of W vector is 512 KB. Limited vocabulary is usually a good idea since it helps to have enough training data points for each word.

The model weights in RNN are learned by SGD-based back-propagation through time (BPTT). BPTT is the same as SGD-based back-propagation for DNN, except that the recurrent layer is unfolded multiple times. After unfolding, RNN looks quite similar to a

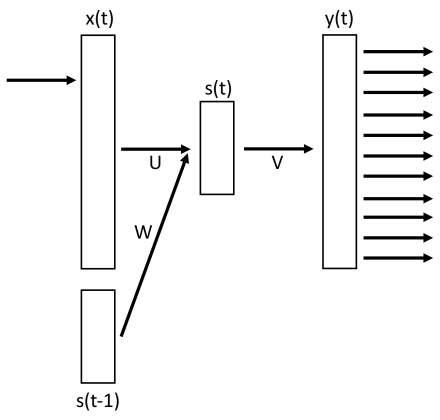


Figure 5.2: Dimension of layers and parameters of RNN.

regular multi-layer feed-forward network. The only difference is that each layer has two different inputs (the previous state and the current input), and the recurrent weight w is shared between each layer.

5.2.2 Stale Value Utilization via Replication

The problem of parallel RNN learning is the significant data communication overhead. While the training data is partitioned, multiple workers access (read-and-modify) common model parameter. The iterative learning produces hundreds of intermediate results which leads to hundreds of data communication between parallel workers thus leading to coherence overhead on coherent systems: lots of invalidation traffic is incurred increasing the memory access latency. Collisions with different workers can cause serialization, thus reducing the benefit of parallelization.

However, the most of the data communication is redundant due to the stale value tolerant characteristic of iterative-convergent ML training. The stale value tolerant characteristic enables to reduce the number of data communication, which has led a lot of distributed learning proposals [4, 5, 6, 7, 8, 9, 10, 11] to utilize the characteristic by model parameter replication.

Figure 5.3 shows how replication reduces the redundant communication for learning performed in a map-reduce manner. The parallel RNN learning that partitions training data consists of three stages: local update, reduce, and scatter. During the local update stage, each worker computes and updates local copies of model replicas. While the model parameter values are continuously modified, the updates are performed on local model replicas without data communication between workers. After processing partitioned data, the updates from different workers are aggregated and reduced(gathered) to the global copy of parameters during the reduce stage. Then, the scatter stage scatters the new model values with aggregated results to model replicas for further learning.

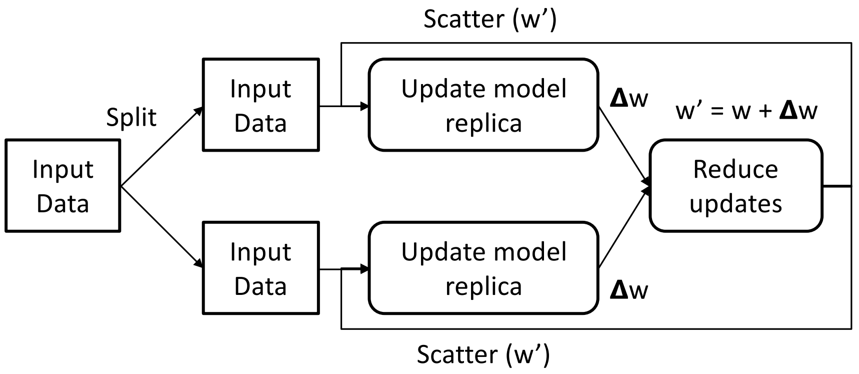


Figure 5.3: Data communication reduction via replication.

It should be noted that there is no data communication between workers between reduction operations, so each worker accesses the stale value of model parameters missing updates from other workers. The multiple data communications accessing the same data with the strict consistency requirement are now transformed into single reduction operation with the relaxed consistency requirement.

5.2.3 Motivation: Performance Potential of Reducing Redundancy

The reduced data communication can increase the scalability of parallel RNN learning. Here, we evaluate the performance implication on coherence overhead on NUMA system. We evaluate the parallel RNN implementation with the PTB [110] training data.[1] The evaluation was done on a multi-core computing platform, consisting of AMD Opterons 6370P

[1] Detailed configuration will be provided in Section 5.3.

with 64 threads on 8 NUMA nodes. We measure execution time per iteration. Since we focus on iterative execution of parallel model training, we ignore initial sequential execution time and file I/O time. For the memory stats, we use performance counter to measure socket-to-socket communication.

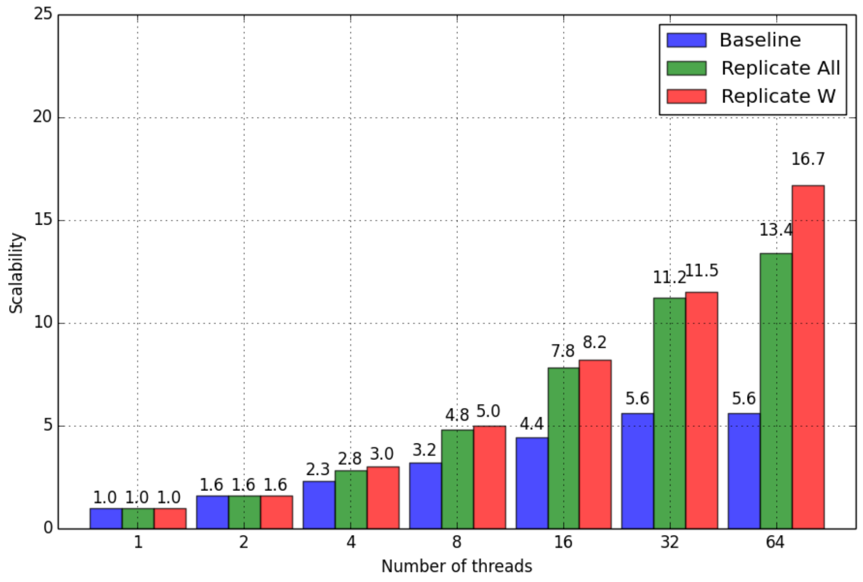


Figure 5.4: Better scalability with replication.

Figure 5.4 shows the better scalability with replication: replicating all vectors and W vector improves the performance by 13.4 and 16.7 times respectively with 64 threads, while the baseline implementation only improves the performance by 5.6 times.[2] The better scalability leads to the larger speedup with parallelization: for 64 threads, replicating all vectors improves the performance by 2.4 times and replicating W vector improves the performance by 3.0 times over baseline implementation.

The better scalability is due to the reduced coherence traffic. Figure 5.5 compares the number of cache block commands[3] of different implementations, which represent the number of requests made to the system for cache line transfers or coherency state changes. For the baseline implementation, the number is increased to 61.8 times when the number

[2]The better scalability of replicating W vector over replicating all vectors is due to the increased working size of replicating all vectors, while most of frequent data communication comes from the dense updates on W vector.

[3]The value of NBPMCx0EA performance counter

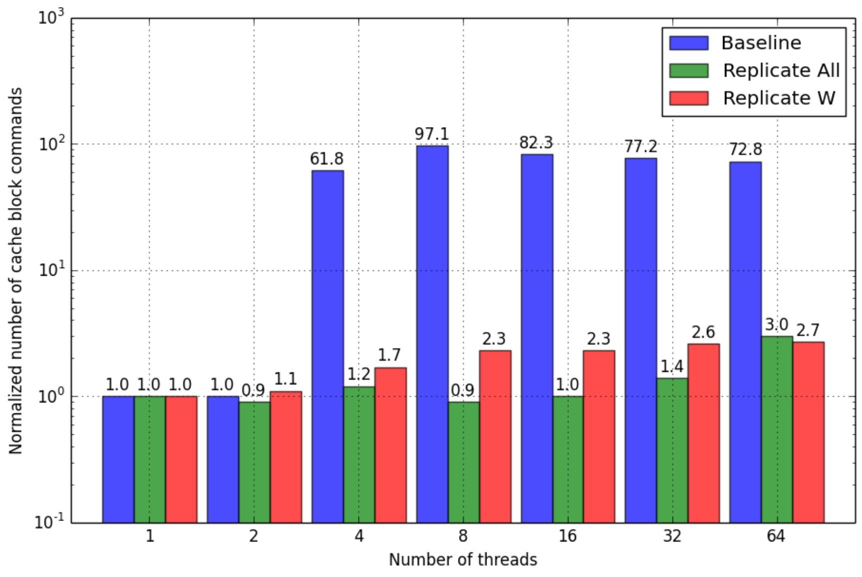


Figure 5.5: Reduced cache block commands.

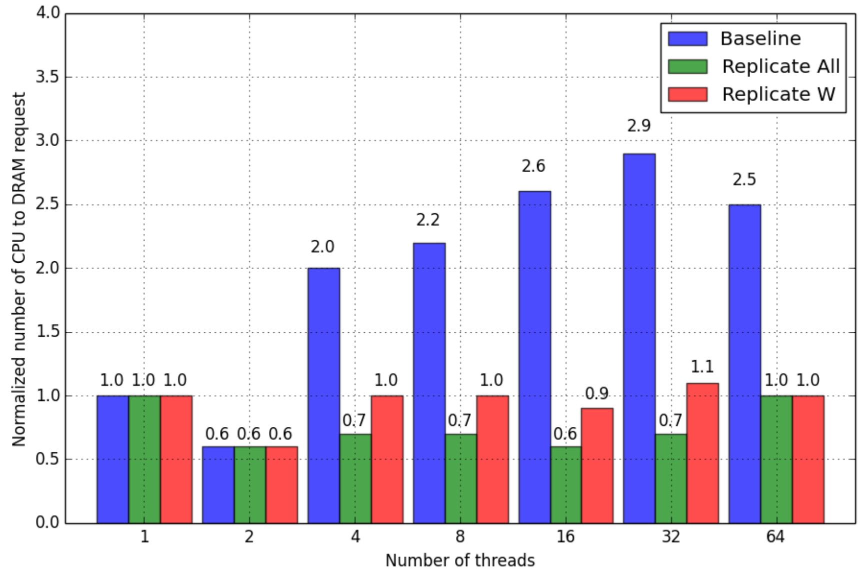


Figure 5.6: Reduced CPU to DRAM request.

of threads only reaches 4. Compared to that, for the implementations with replication, the number is only increased to 3.0 times (All), and 2.7 times (W) when the number of threads is 64. Figure 5.6 compares the number of CPU to DRAM requests (all DRAM reads and writes generated by cores),[4] which represents the processor data affinity in NUMA systems. While the value is increased for the baseline implementations, the number is unaffected

[4]The value of NBPMCx1E0 performance counter

with replications.

5.2.4 *Challenge: Impact of Replication on Progress*

While replication can reduce the redundant data communication, the errors in intermediate calculations should be limited. Too much reduction of data communication can lead to slower progress per each iteration thus increasing the number of iterations before converging to the same quality solution of the baseline implementation without model replicas. Due to the iterative execution characteristic, the performance of learning is the product of both 1) the number of iterations per time and 2) the progress per each iteration, which means how much the solution has progressed within a certain time. Thus, we evaluate how replication can affect the progress of learning.

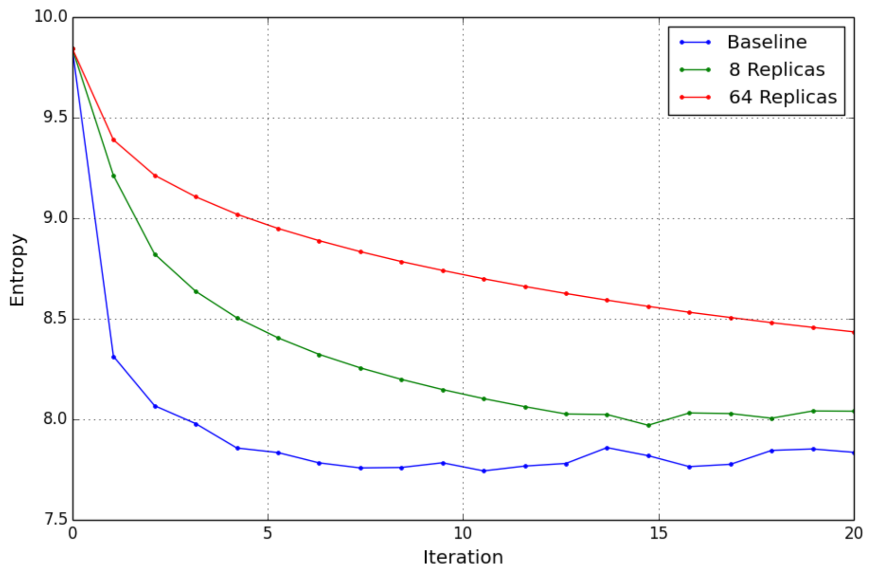


Figure 5.7: Slower progress with more number of replicas.

Unfortunately, the replication significantly affects the progress of deep RNN learning. Figure 5.7 shows the slower progress with more number of replicas, where the x-axis represents the iteration, and they y-axis represents the entropy of the RNN. Here, the lower entropy means higher solution quality and we apply arithmetic mean of replica values across all workers on the reduction phase. Not just the progress per iteration is reduced, but also the final solution quality can be affected since most of the learning methods decide whether the solution is converged by looking at the progress of last iterations. Further, when the progress is small, most learning methods lower the learning rate, which can further lower the progress rate and the final solution quality.

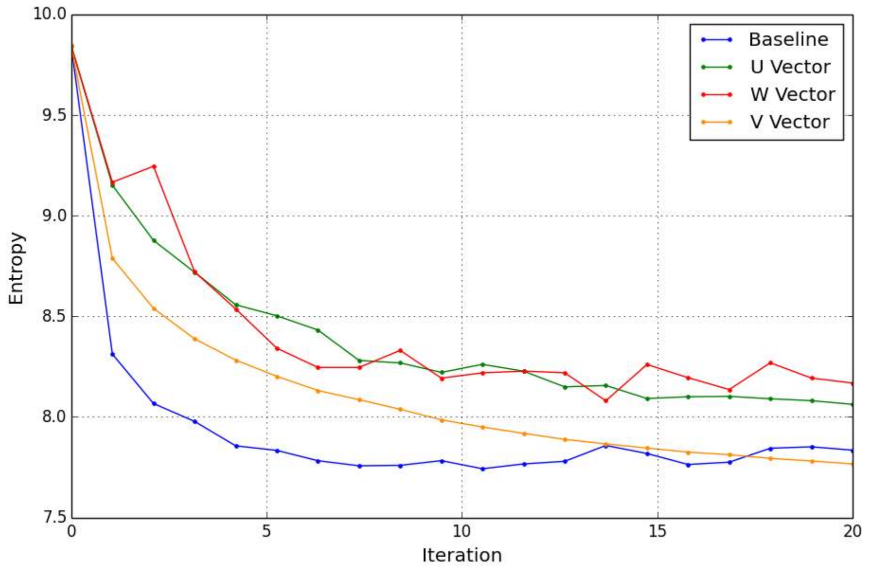


Figure 5.8: Different stale value tolerance on different model parameters.

To make the problem more complex, different weights have different stale value tolerance even within a single neural network. Figure 5.8 compares the different stale value tolerance on different layers in RNN. While replicating V vector can reach to similar solution quality to that of the baseline, replicating U and W vector only reaches to the significantly lower quality solution. The finding implies that we need to understand the stale value tolerance based on application characteristic. So for this challenge, we devise a systematic method to understand stale value tolerance in learning.

5.3 Understanding Stale Value Tolerance

In this section, we devise the systematic method to define the stale value tolerance of different learning applications. We use RNNLM implementations of faster-rnnlm [111] that supports flexible configuration such as different activation functions, different RNN cell types, and the learning rate adaptation. The implementation supports data parallel learning with p-threads. Instead of decaying the learning rate, we use the fixed learning rate, whose value equals to the initial learning rate of baseline implementation (0.1). For other parameters, we use default parameters unless specified otherwise.

$$
\begin{aligned}
w_{new} &= w_{old} + \sum_{i=1}^{n} \Delta w_i \text{ (delta reduction)} \\
w_{new} &= \frac{1}{n} \times \sum_{i=1}^{n} w_i \text{ (average reduction)} \\
&\text{where n is worker\_count, and } \Delta w_i = w_i - w_{old}
\end{aligned} \tag{5.1}
$$

For reduction operation, we control the reduction frequency by performing reduction after each worker processes *n* sentences (whose values are 1, 5, 40, 100, 200, 400), instead of performing reduction after processing all partitioned data. We evaluate two reduction method: delta and average reduction as shown in Equation 5.1. Delta reduction accumulates delta from each worker and adds to shared model parameter, where delta is computed by comparing old value(w_{old}) to new value on each replica (w_i). For average reduction, arithmetic mean of weights on all workers is calculated and applied at reduction phase. Average reduction reduces the learning rate by the "number of workers" from delta reduction.

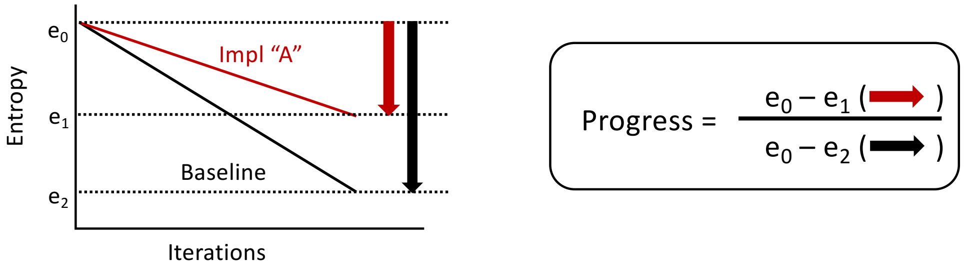


Figure 5.9: Progress of solution quality.

To understand the stale value tolerance, we compare the progress of solution quality (entropy). We measure how much the solution quality has been improved with the same number of iteration from different implementations. Figure 5.9 shows how we define progress. When the entropy is reduced from e_0 to e_2 for the baseline implementation and the entropy is reduced from e_0 to e_1 for the implementation ”A”, the progress of the implementation ”A”” is defined as $\frac{e_0-e_1}{e_0-e_2}$. The large entropy reduction means larger progress, such that ”progress = 1” means the same entropy reduction as the baseline, and the ”progress = 0” means learning nothing.

$$LR_{eff} = \delta \times LR_{orig} \tag{5.2}$$

In this study, we define the stale value tolerance of model parameter replication with the effective learning rate (LR_{eff}). As shown in Equation 5.2, the model parameter replication changes the original learning rate of the baseline (LR_{orig}). The δ value is decided by multiple factors of learning and replication. In this study, we evaluate how δ is changed depending on each factor of learning.

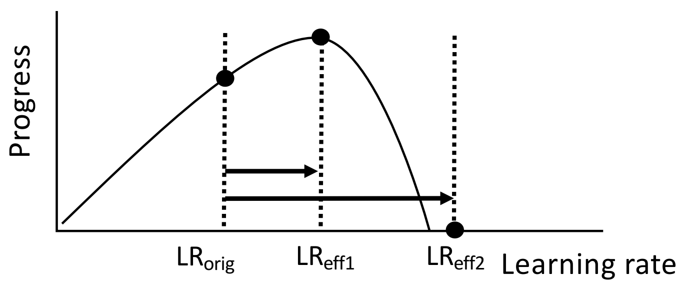


Figure 5.10: Progress and learning rate.

Figure 5.10 shows our assumption regarding progress and learning rate. The effective learning rate affects the progress of neural network learning. For the first case (LR_{eff1}), the larger learning rate with replication increase the progress. However, too large learning rate leads to learning nothing with fluctuations (LR_{eff2}). With this assumption, by measuring progress, we find how much the learning rate is changed. While this assumption can be too naive, our empirical evaluation revealed that this simple method works surprisingly well providing reasonable explanations regarding the different stale value tolerance of different ML applications and even different model parameters within a single neural network.

To verify our findings are consistent with different inputs, we use two inputs. We perform model fitting for different neural network configurations with the English Penn Treebank [110], whose training data has 10k words of vocabulary with 929589 training words (represented as lines). We measure the progress of different implementations and each execution of RNN learning is considered as a single data point in model fitting. For testing to verify the consistency of our findings, we use the One Billion Word Benchmark [112], with around 0.8 billion words in the training corpus and 793471 words in the vocabulary (represented as dots).

5.3.1 Effect of the Accumulated Number of Local Updates

Reducing redundant communication by performing reduction operation between replicas can be viewed as performing implicit momentum updates regardless of whether or not the objective function is convex [113]. Momentum update [114] is widely used on multiple learning problems due to its stability, unlike conventional gradient descent update that suffers from fluctuation with high variance. In this section, we call conventional momentum updates as explicit momentum updates to differentiate it from implicit momentum updates with replication.

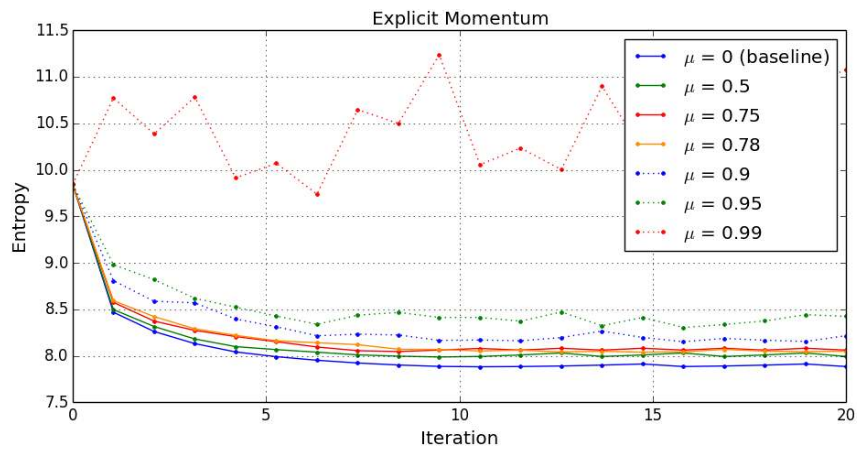
$$\begin{aligned} & w_{t+1} - w_t = \mu(w_t - w_{t-1}) + \alpha \nabla L(w_t) \\ & \text{where } \mu = \frac{n-1}{n} \end{aligned} \tag{5.3}$$

Equation 5.3 shows the formula of explicit momentum update for the weight w at the iteration $t+1$ for the loss function L with the momentum hyperparameter μ and learning rate α. The momentum hyperparameter μ is to increase updates for dimensions whose gradients consistently point in the same directions, which is defined by the number of accumulated updates (gradient) when performing the update, n. In other words, μ effectively multiplies the learning rate by $\frac{1}{1-\mu}$ times by accumulating $\frac{1}{1-\mu}$ SGD updates. Different from the SGD update, where the gradient directly integrates the position, the gradient only directly influences the velocity with momentum updates. By smoothing the weight updates, momentum makes deep learning with SGD both more stable and faster.

Similarly, we can define the implicit momentum hyperparameter. We use the same Equation 5.3 but with the different definition of n. Here, n represents the accumulated number of local updates performed on model replicas from all workers between reduction operations. For example, when two workers are executed in parallel, and each worker modifies the specific model parameter by two times between reduction operations, the value of n becomes four. The implicit momentum hyperparameter can be adjusted by changing the reduction operation frequency, such that more frequent reductions reduce the momentum hyperparameter by reducing the accumulated number of local updates n. Our definition of implicit momentum hyperparameter extends the similar definition in the work from Mitliagkas et al. [113] that defines the hyperparameter with the number of replicas to provide a better understanding of stale value tolerance of different model parameters with different update frequency.

Out evaluation revealed that the implicit momentum update affects the progress of RNN similar to the explicit momentum update. Figures 5.11 and 5.12 show the similar behavior of explicit and implicit momentum updates (performing delta reduction with different reduction frequency). For the W vector, the solution quality fluctuates for the explicit momentum hyperparameter of 0.99. Similar fluctuation is found for implicit momentum when performing reduction after each worker processes more than 200 sentences. Replicating the U vector exhibits similar behavior, exhibiting same progress with the baseline implementation when performing frequent reduction, and exhibiting slower progress after a certain threshold (100).

The slower progress and fluctuation with infrequent reduction is due to too large learn-



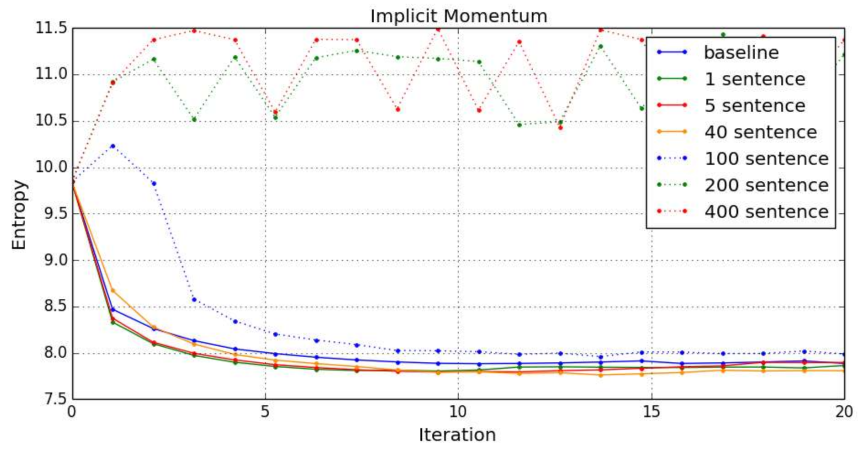


Figure 5.11: The effect of momentum updates on progress for W vector.

ing rate with the large momentum. Too large learning rate is problematic for progress since it brings too much kinetic energy, unable to settle down into the deeper part of the loss function, which hinders progress and cause the loss function to fluctuate chaotically or even to diverge. So the programmer needs to control the learning rate by frequent reduction since infrequent reduction leads to too large delta when performing reduction operation.

Even with the same reduction frequency, the number of local updates per weight between reduction operations is different depending on vectors as shown in Table 5.1 The number of local update for U vector is limited due to sparse accesses on this vector, while that of W vector is large even with frequent reduction operations due to dense accesses.

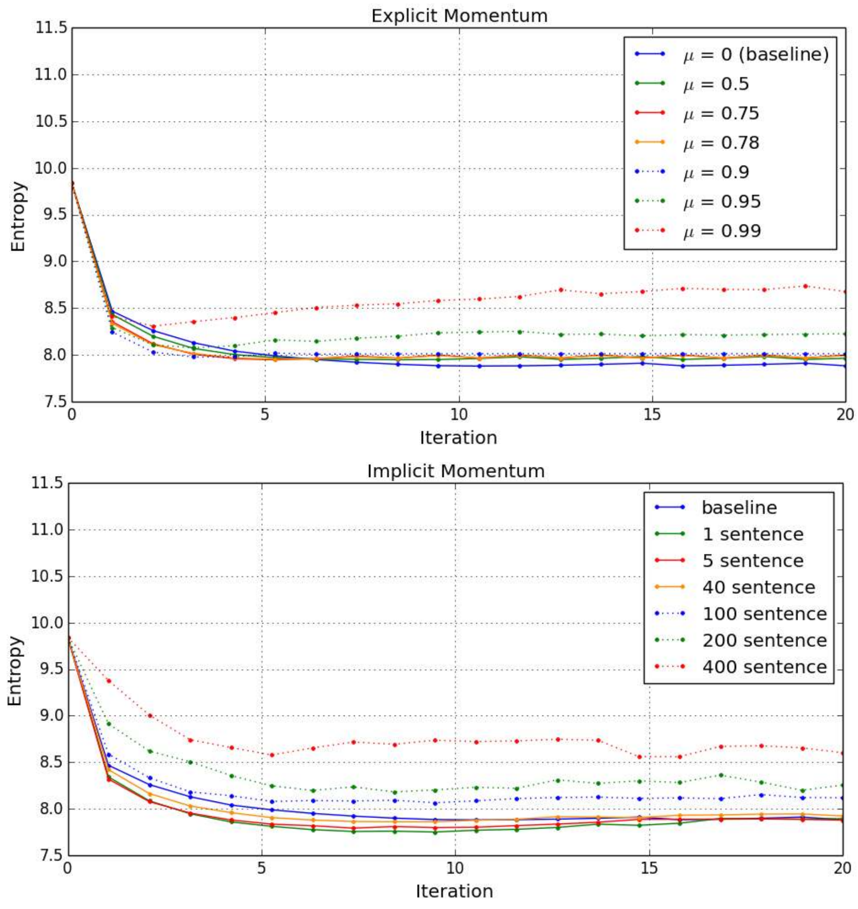


Figure 5.12: The effect of momentum updates on progress for U vector.

Table 5.1: Accumulated number of local updates with the different reduction frequency.

Sentence	W vector	U vector	V vector
1	1351	0.14	1.34
5	6753	0.71	6.69
40	54021	5.66	53.49
100	135053	14.15	133.72
200	270107	28.29	267.44
400	540214	56.58	534.89

5.3.2 Effect of Update Density

While implicit momentum hyperparameter provides a metric to understand the effect of reducing data communication on progress of RNN learning, the same implicit momentum hyperparameter can also exhibit totally different progress behavior when replicating dif-

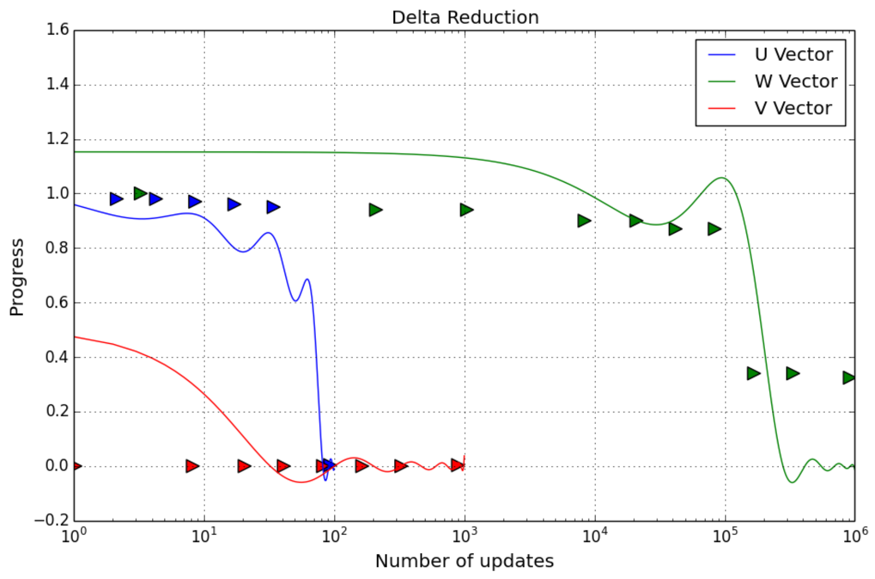


Figure 5.13: The effect of update density on progress with delta reduction.

ferent layers of the neural network. Figure 5.13 shows that the stale value tolerance when replicating U and V vectors is much worse than when replicating W vector: while replicating W vector does not degrade the progress until the accumulated number of local updates from all workers become 10^5, replicating U and V vector degrade the progress when the accumulated number is relatively small. This behavior is counter-intuitive when we only consider the implicit momentum update.

The effective learning rate when performing reduction operations depends on the update density on each layer of the neural network. The effective learning rate is inversely proportional to the update density on the vector. While more number of accumulated local updates might represent increasing the learning rate to result in saturation of the vector, the contribution of each update on the effective learning rate is different on different vectors with the different update density. For W vector, where the dense update is performed, the importance of each update is reduced, while the importance is increased for U and V vector where the sparse update is performed.

The different update density affects the reduction granularity. Figure 5.14 shows how the reduction granularity affects the progress when performing the delta reduction, where

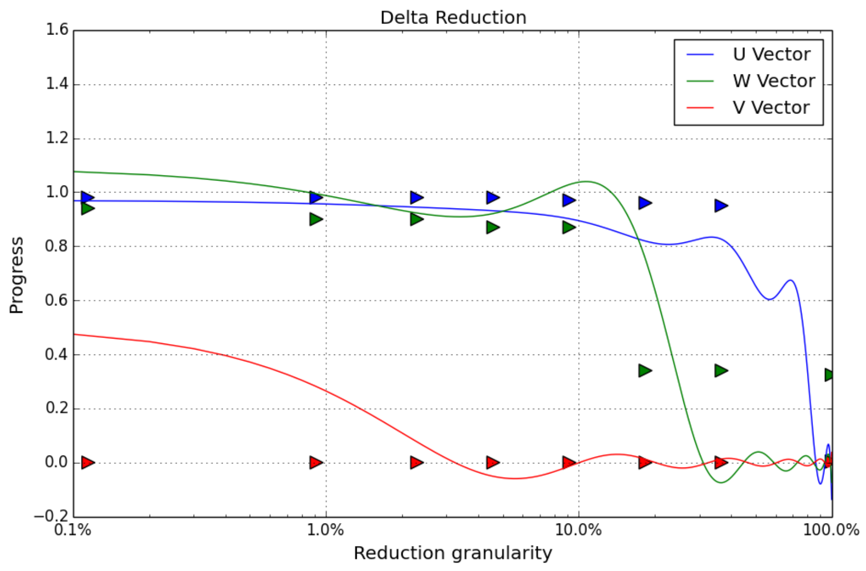


Figure 5.14: The effect of reduction granularity on progress with delta reduction.

100% means performing reduction after processing entire training data, and lower value represents more frequent reduction operation. While the replicating W vector accumulates more number of local updates than other vectors with the same reduction granularity, the programmer can maintain similar reduction granularity as other vectors. This finding is opposite to the assumptions of previous distributed learning proposals such as Hogwild! [8], that relaxes synchronization assuming non-overlapped accesses due to sparse model accesses.

5.3.3 *Effect of Activation Functions*

Figures 5.13 and 5.14 show that the stale value tolerance of replicating V vector is much worse than replicating U vector, which cannot be explained only considering the implicit momentum hyperparameter and update density. This different effect can be understood by different learning rates of different layers in the neural network. Different learning rates increase the difficulty of learning with increasing number of layers and have limited the number of layers on many so-called deep neural network. While more number of layers should help to have a more accurate model [115], the larger learning error with more layers has led the solution quality to become lower after a certain number of layers.

The different learning rates of different layers in the neural network is a fundamental characteristic introduced by back-propagation [35]. Let's assume a multi-layer network with a single neuron in each layer with n as the layer length, w as the weight, b as the bias, and L as the loss function. The output of each layer, o_j is $\sigma(z_j)$, where $z_j = w_j o_{j-1} + b_j$. In this case, the gradient for the first bias ($\frac{\delta L}{\delta b_1}$) becomes $\sigma'(z_1) \times \prod_{i=2}^{n} w\sigma'(z_i) \times \frac{\delta L}{\delta o_n}$. The gradient for the first bias includes the product of w_j and $\sigma'(z_j)$ from all the later layers. Since the value of w_j and $\sigma'(z_j)$ are different on different layers, the back-propagation naturally leads to different learning rates of different layers in the deep neural network.

Different activation functions affect the learning rates of different layers in different ways. Activation function in a neural network is used to approximate functions universally, by producing non-linear combinations of the weighted inputs. The selection of activation function plays a critical role in learning rates of different layers since the derivatives of activation functions affect the magnitude of gradients.

To evaluate the effect of the activation function for the effective learning rate, we evaluate three different activation functions: sigmoid (baseline), Rectified Linear Unit (ReLU), and truncated ReLU (ReLU-trunc). Figure 5.15 shows how the activation function affects the progress of RNN learning when performing the delta reduction. The effect of activation function on progress is different on different layers. While the similar progress behavior is exhibited for W vector (The reduction frequency dominates progress), the behavior is quite different for U and V vectors. While ReLU and ReLU-trunc activation functions require the less frequent reduction operations than sigmoid when replicating the V vector, they require more frequent reduction operation when replicating the U vector.

Different learning rates due to activation function selection significantly affect the reduction frequency to maintain reasonable progress per iteration. When using sigmoid activation function that squashes the input space into a small region, early layers learns slower than the later layer since gradient contributions from "far away" steps vanishes as proceed-

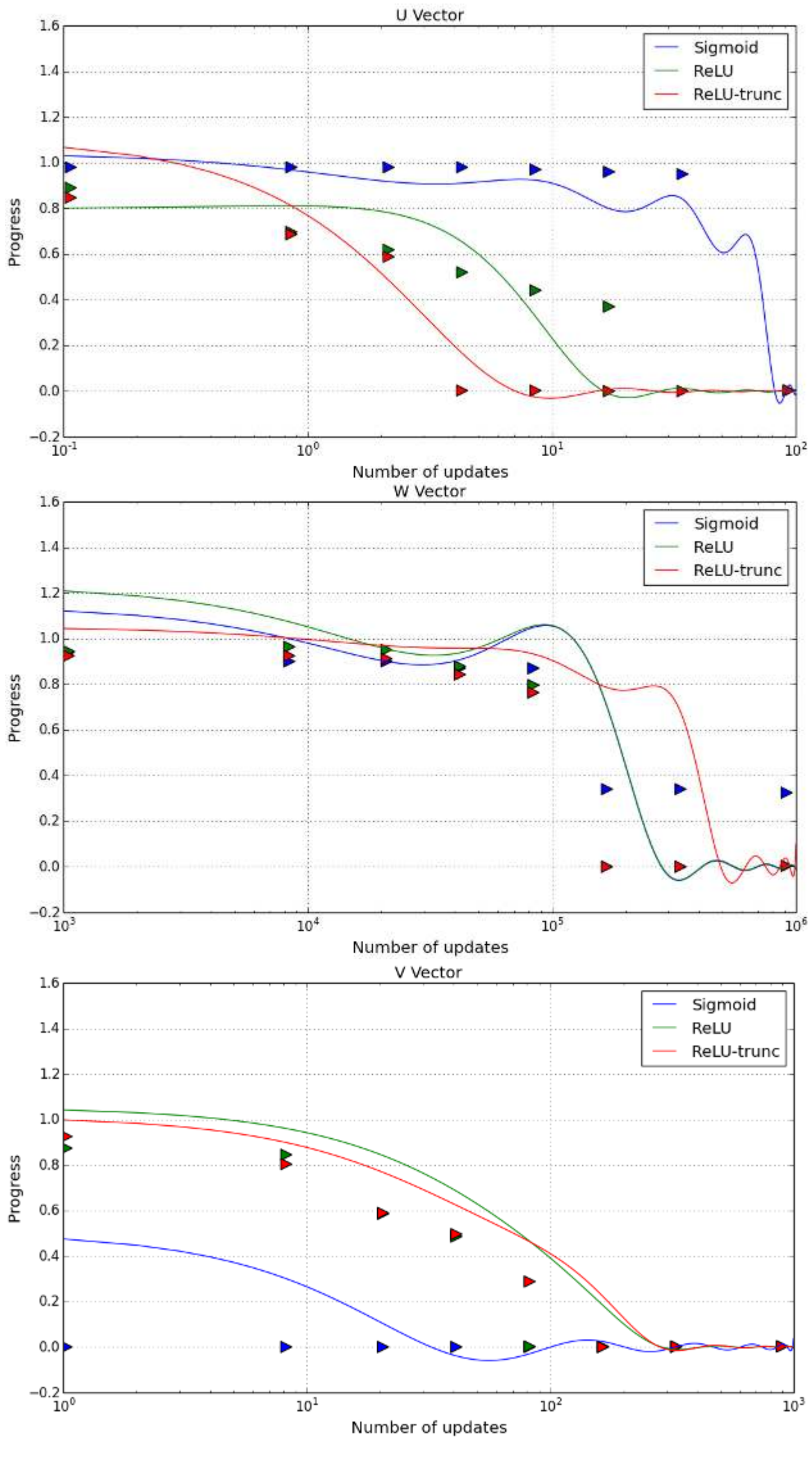


Figure 5.15: The effect of activation function on progress with delta reduction.

ing to early layers during back-propagation. So the large delta with delta reduction for the V vector leads to the output vector saturation, substantially slowing down learning and affecting progress [35]. When the output vectors are saturated, the sigmoid activation function has the zero gradient and drives other gradients in previous layers towards zero, thus learning nothing. ReLU and ReLU-trunc activation functions work better than sigmoid activation since ReLU and ReLU-trunc activation functions reduce the vanishing gradient. Compared to sigmoid activation, early layers learn at faster learning rate when using ReLU activation function, since the derivative of the activation function is a constant of either 0 or 1.

On the contrary, when replicating the U vector, sigmoid activation enables better progress than ReLU and ReLU-trunc. When using ReLU and ReLU-trunc activation functions, the large delta with delta reduction for the U vector can lead to input vector saturation making too large steps thus slowing down progress [116]. The reduced learning rate of early layers with sigmoid activation function helps to reduce the reduction frequency since it reduces the large delta coming from the delta reduction than ReLU and ReLU-trunc activation functions.

The progress with average reduction can also be understood with different learning rates depending on activation functions. Figure 5.16 shows how the activation function affects the progress when performing the average reduction. When replicating the V vector, the average reduction enables similar progress behavior regardless of activation function, since the small α of the average reduction reduces the V vector saturation by balancing the learning rate of V vector to those of other vectors. On the contrary, when replicating the U vector, ReLU and ReLU-trunc function activation functions enable better progress (1.2) than baseline, while the quality is lowered with sigmoid activation (0.9). It is because they maintain faster learning rates on the U vector, the early layer of the neural network.

In summary, the effect of activation function on the learning rate is different depending on the location of the layer: whether it is the early layer or the later layer in the neural network. Following summarizes the findings.

- The activation function affects the learning rates of different layers of the neural network in different ways. The sigmoid activation function reduces the learning rates

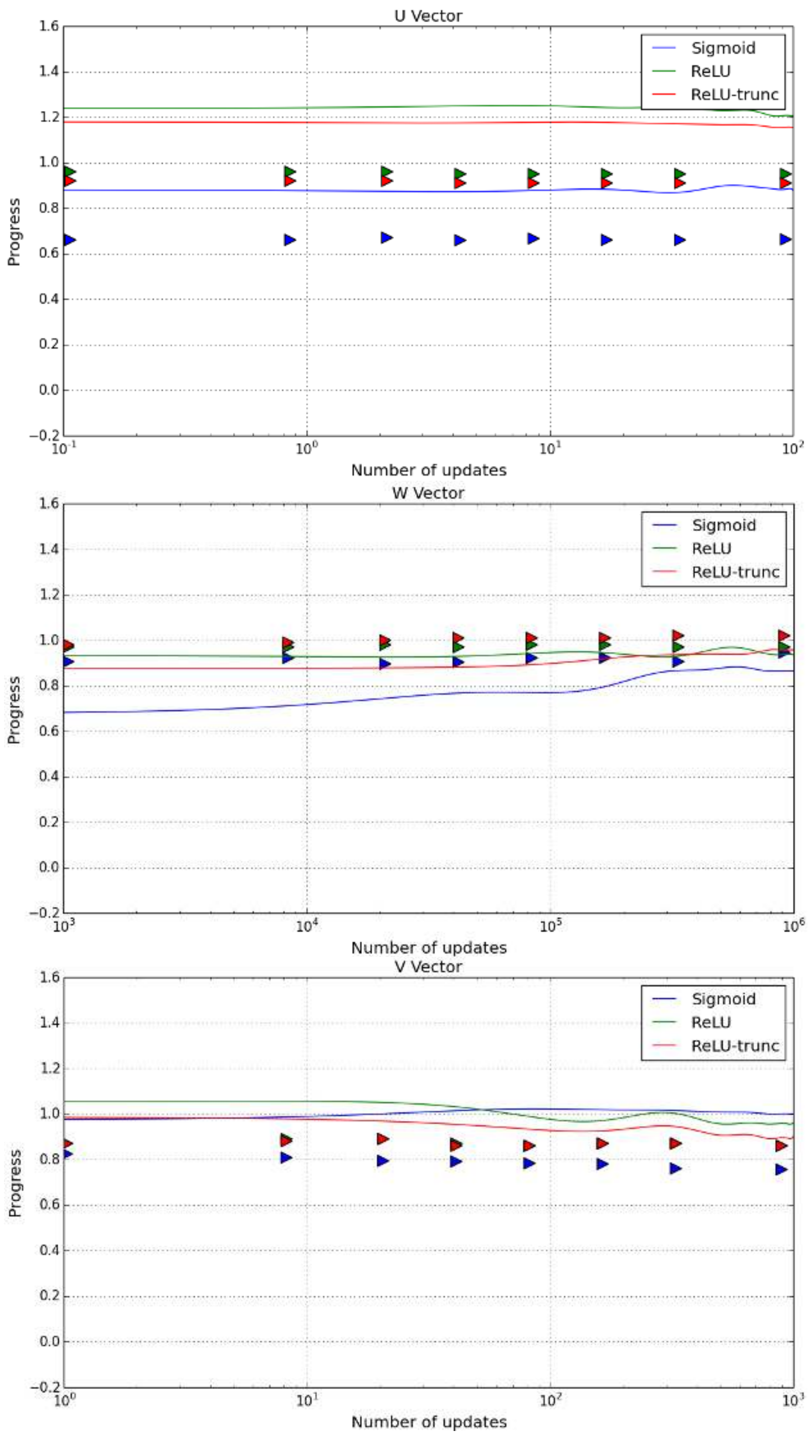


Figure 5.16: The effect of activation function on progress with average reduction.

of early layers, while the ReLU and ReLU-trunc activation increase the learning rates of early layers.

- When replicating the model weights, the programmer needs to control the size of delta (effective learning rates) to enable balanced learning on different layers. The delta reduction can lead to too large deltas thus slowing down learning. While ReLU activation can reduce the vanishing gradient problem on early layers when replicating the weight vector at later layers, the larger step than sigmoid activation when replicating the weight vector at early layers can also slow down learning, so programmer needs to balance the effective learning rate on both early layers and later layers.

5.3.4 Effect of RNN Cells

Considerable efforts have been made to reduce the unstable gradient problem on deep RNN by replacing basic RNN cells with different types of RNN cells including Long short-term memory (LSTM) [117] and Gated Recurrent Units (GRU) [118] architectures. These RNN cells enable to learn longer dependencies; which is hard with simple RNN cells that suffer from different learning rates of different layers in the deep RNN.

$$\begin{aligned}
z &= sigmoid(x_t U^z + s_{t-1} W^z) \\
r &= sigmoid(x_t U^r + s_{t-1} W^r) \\
h &= tanh(x_t U^h + (s_{t-1} \circ r) W^h) \\
s_t &= (1-z) \circ h + z \circ s_{t-1}
\end{aligned} \tag{5.4}$$

Equation (5.4) shows the equation of GRU gating mechanism. A GRU cell has two gates, an update gate z and a reset gate r. They are called gates because they maintain the weighted average of the new value and the previous value instead of completely replacing cell contents. The update gate defines how much to keep the previous hidden state value, and the reset gate determines how to compute the candidate hidden state h based on the current input and the previous hidden state. The candidate state h is computed using a similar equation as in vanilla RNN with same parameters U and W. However, instead of directly using the previous hidden state value, the product of the value of reset gate and the previous hidden state value is used when calculating h. Given the candidate hidden state, the output hidden state s_t is computed, which is the weighted average of h and s_{t-1}, with the ratio defined by the value of the update gate z.

To evaluate the how different RNN cell types affect the progress of RNN learning when performing reduction operation, we evaluate four different RNN cell types: basic RNN cells with sigmoid activation and three different GRU cell types (GRU, GRU-bias, GRU-insyn). GRU is the GRU cell that uses identity matrices for input transformation without bias. GRU-bias is GRU with bias terms. GRU-insyn [119] follows the equation of Equation (5.4).

GRU cells balance the learning rate of different layers in a neural network. Figures 5.17 compares how the different RNN cells affect the progress when performing the delta reduction. On many cases, the behavior of GRU cells is similar to that of the ReLU activation function, which is intuitive since both proposals target to reduce the saturating (vanishing) gradient problem of the deep RNN. Similar to the RNN with ReLU activation, when we replicate the U vector and perform delta reduction, the RNN with the GRU cell requires more frequent reduction operation than the RNN with sigmoid activation. It is because the combination of delta reduction and GRU cells lead to too strong gradient on the U vector, which is similar to the behavior of the RNN with ReLU activation function.

But, there are also unique characteristics coming from GRU cells. Different GRU cells have different learning rates: GRU-insyn with the largest learning rate, GRU in the middle, and the GRU-bias with the lowest learning rate. It is because the input transformation matrix increases the learning rates while the bias term reduces the learning rate by increasing the symmetric bias. Figure 5.17 shows that when replicating the V vector, the low learning rates of GRU-bias reduces the output vector saturation thus requiring less frequent reduction operations, while the higher learning rate of GRU-insyn requires frequent reduction operations. Figure 5.18 shows that GRU-insyn always exhibits the highest progress when

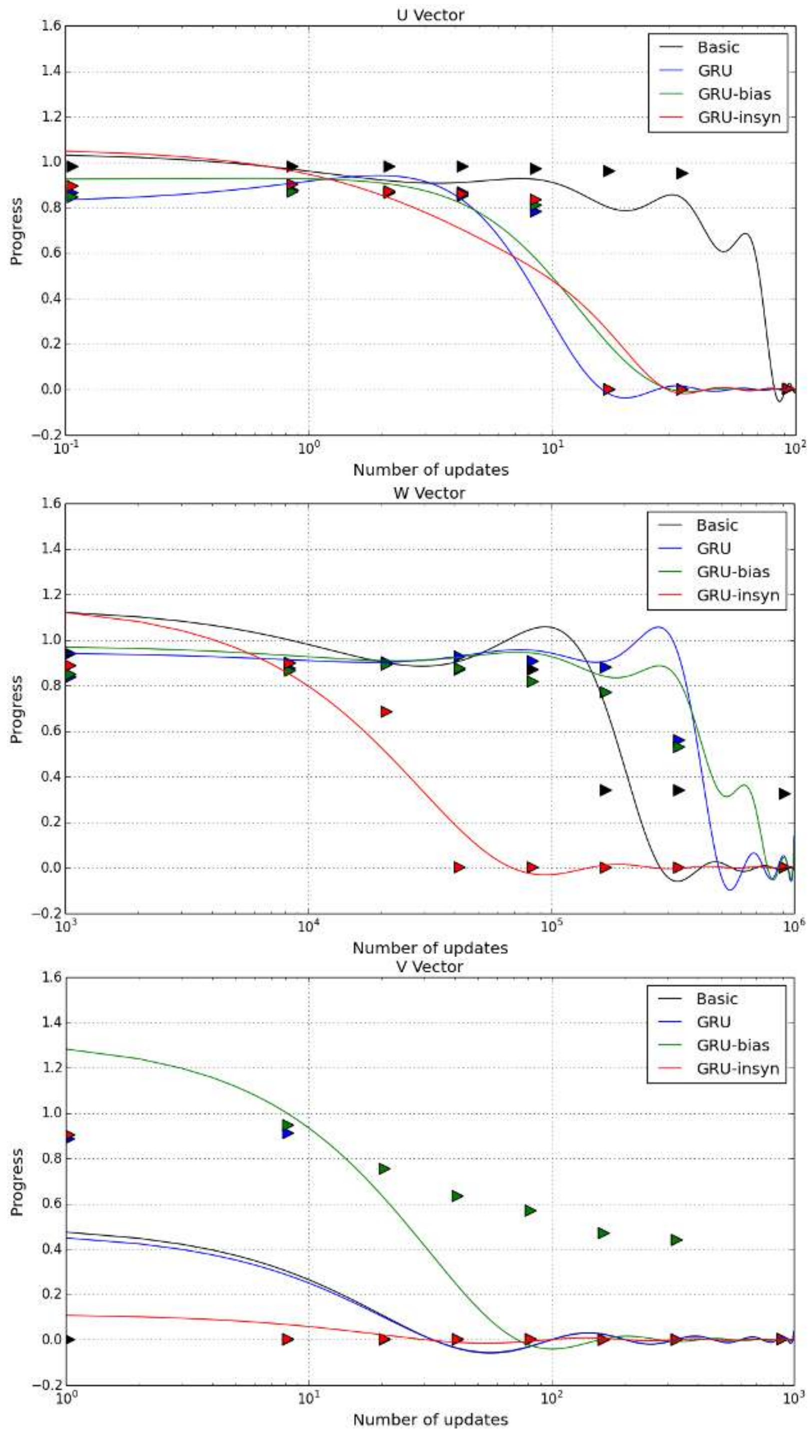


Figure 5.17: The effect of RNN cells on progress with delta reduction.

performing the average reduction on U vector due to the higher learning rate.

It should be noted that the balanced learning of GRU cells can also lead to too large deltas when performing the reduction operation on early layers. While the balanced learn-

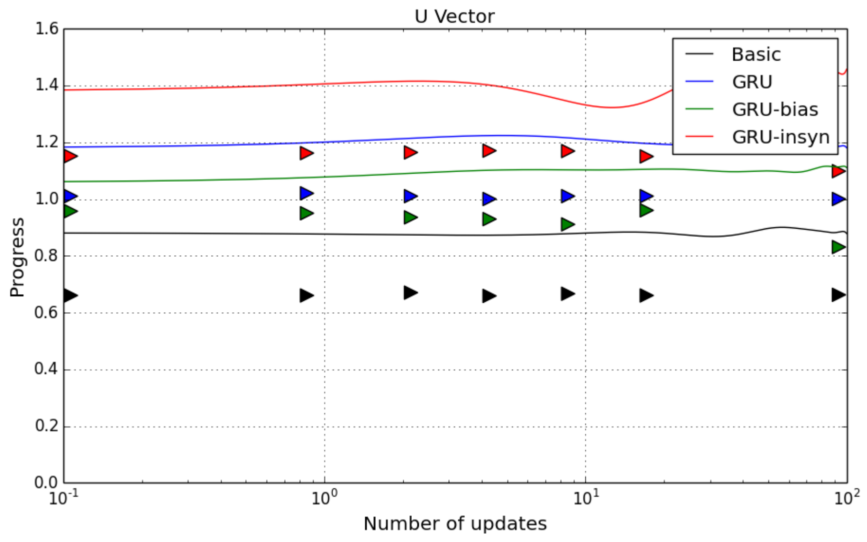


Figure 5.18: The effect of RNN cells on progress with average reduction on the U vector.

ing of GRU-bias enables better progress than sigmoid activation when performing delta reduction for the V vector, GRU-bias performs worse when performing delta reduction for the U vector as shown in Figure 5.17. In fact, the small learning rate of U vector due to vanishing gradient of sigmoid activation function reduces the delta when performing delta reduction for the U vector.

Following shows the how the different RNN cells affect the progress behavior regarding reduction operations.

- GRU cells reduce saturating gradient problem via gating, thus balancing the learning rates of different layers.
- The different RNN cells have different learning rates. The input matrix on GRU-insyn cell increases the learning rate while the bias term reduces the learning rate.

5.3.5 *Effect of Learning Rate Adaptation*

The basic SGD method mostly reduces the learning rate by initializing the learning rate to a relatively large value and dropping it when the loss begins to reach an apparent plateau. While it is simple, the approach has a downside of getting trapped in saddle points where the gradient is zero in all dimensions. The monotonic learning rate decay can lead deep learning to be trapped and stop learning too early. The optimal magnitude of the gradient can be very different for different weights and can change during learning, which makes it hard to choose a single global learning rate. Therefore, multiple studies propose to adapt learning rate for each individual parameter.

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma){g_t}^2 \\ w_{t+1} &= w_t + \alpha \frac{g_t}{\sqrt{E[g^2]_t + \varepsilon}} \end{aligned} \tag{5.5}$$

Equation 5.5 shows the equation of RMSProp [120], a popular, adaptive learning rate method. Given the computed gradient g_t for a weight, RMSprop computes the moving average of squared gradients for that weight ($E[g^2]_t$) with decay rate γ whose value is typically set as [0.9, 0.99, 0.999]. RMSprop divides the learning rate for the weight by the moving average, so the weight that receives high gradients will have the learning rate reduced, while the weight that receives small updates will have the learning rate increased. RMSprop will increase the learning rate, enabling to escape from the saddle point. The smoothing term ε avoids division by zero.

Figure 5.19 shows that RMSprop increases the learning rate when performing the average reduction for the W vector. It is because each worker adapts learning rate well, adjusting delta size on reduction operation. RMSProp limits the too large delta per each reduction operation and increases the learning rate for the weights that receive small updates.

5.3.6 Usage and Future Work

Usage: With a large amount of data examined during learning, parallel learning is imperative to reduce the learning time. As a result, multiple distributed learning studies utilize the stale value tolerant characteristic to reduce data communications between parallel workers. Most of ML application domains can benefit from the stale value tolerance since they share

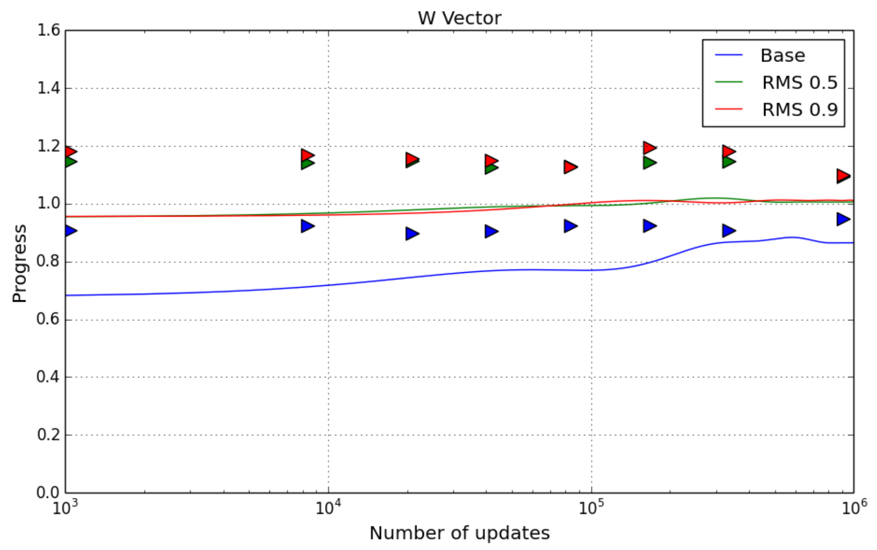


Figure 5.19: The effect of RMSProp on progress with average reduction for W vector.

the same learning method, the gradient-based learning method.

As shown in our study, utilizing stale values reduces the data communication overhead of parallel learning but may also lower the progress of each iteration. Any parallel learning framework should take this trade-off into account. However, conventional parallel learning studies implicitly select a staleness degree by try and error without considering the tradeoffs from different staleness degrees. Due to the vast decision space between gains and losses of different staleness degrees, a naive selection of the staleness degree can potentially lead to an unacceptable slowdown of progress. In particular, the optimal selection of design choices for neural network learning with a single worker can yield slower progress when the application is deployed for the real usage, which requires utilizing multiple workers for faster learning.

Compared to previous efforts, our study provides a metric to understand the different stale value tolerance with a single term, the learning rate and suggests a limit for stale value utilization. With this simple term from our study, we can build an optimizer that would tune the degree of staleness by navigating the tradeoffs to deliver high performance while lowering the loss of progress. The optimizer can dynamically identify whether each individual data communication reduction will lead to an undesirable slowdown of progress.

Furthermore, our study enables to statically predict the performance and the progress of parallel learning based on the progress of single worker even before parallelization. The prediction can be used for deciding the optimal number of parallel workers without wasting computing resources. Also, the prediction can even be used to change the design choices of neural network learning before deploying the application to run on multiple nodes, which significantly reduces the application development time.

Future Work: The metric provided in this study is simple and can be easily verified with examining different types of ML applications. While we expect the similar behavior of what we observed from RNN, different ML applications can potentially exhibit totally different behavior to suggest new interesting models. In the future work, we will examine other types of neural network architecture and different application domain for verification of our findings.

5.4 Summary

Efficient execution of modern ML training has gained significant importance. While considerable effort has focused on distributed platforms utilizing the stale value tolerant characteristic of training, the lack of a systematic method to define the stale value tolerance on different applications has caused ambiguity for domain experts thus limiting the scalability of parallel learning. For this challenge, we define the stale value tolerance of model parameter replication with the effective learning rate, which is proportional to delta size when performing reduction operations. While different training data might exhibit different progress with the same effective learning rate, we assume that degree of the learning rate change is consistent depending on the design choices of training neural network. In summary, the key findings in this chapter are as follows:

1. The effective learning rate is proportional to the accumulated number of local updates and inversely proportional to the update density on the model parameters.

2. Different stale value tolerance of neural network with different activation functions is due to different learning rates of different layers depending on activation function selection. The sigmoid activation function reduces the learning rates of early layers, while the ReLU and ReLU-trunc activation increase the learning rates of early layers. When performing reductions with replicated model parameters, the programmer needs to control the effective learning rates to enable balanced learning on different layers.

3. Different RNN cells affect the stale value tolerance in different ways. GRU cells reduce saturating gradient problem via gating, thus balancing the learning rates of different layers. While the balanced learning of GRU cells can lead to faster progress with large learning rate, it can also lead to too large deltas when performing the reduction operation on the model weights at early layers. The input transformation matrix in GRU cells increases the learning rate while the bias term reduces the learning rate by increasing the symmetric bias.

4. The learning rate adaption can increase the stale value tolerance with replication, since it limits the too large delta per each reduction operation, and increases the learning rate for the weights that receive small updates.

CHAPTER 6
CONCLUSION

The key phase of ML is learning that is iterative-convergent. Iterative-convergent learning allows the consistent view of memory does not need to always be guaranteed, allowing different threads to compute using stale values. Relaxing coherence for these learning applications has the potential to provide extraordinary performance and energy benefits. So, in this dissertation, we have proposed several innovations for efficient utilization of the full performance potential of the stale value tolerance. We focus on the major performance challenges of parallel learning: the atomic operation overhead with dense model updates, and divergent memory access overhead with sparse model updates. We also reduce the ambiguity for the stale value tolerance that has limited utilizing this characteristic. Here is the summary of the techniques proposed in this dissertation.

1. Chapter 3 provides Bounded Staled Sync (BSSync), an effective hardware mechanism to overcome the overhead of atomic operations on iterative-convergent machine learning training. BSSync reduces the overhead of non-overlapped data communication, the serialization, and cache utilization inefficiency. The proposed technique overlaps the long latency atomic operation with the main computation. Compared to previous work that allows staleness for read operations, BSSync utilizes staleness for write operations. Atomic operations are asynchronously executed in parallel with the main computation. The performance results show that BSSync outperforms the asynchronous parallel implementation by 1.33x times.
2. Chapter 4 presents StaleLearn, an effective learning optimization to overcome the memory divergence overhead of the GPU learning with sparse data. We find that relaxing the coherence can play a pivotal role in parallel GPU learning by transform-

ing memory divergence problem into synchronization problem. StaleLearn performs model replication and asynchronous synchronization on PIM. The reduced synchronization requirement enables StaleLearn to exploit the parallelism between PIM and GPU cores by overlapping PIM operations with the main computation on GPU cores. StaleLearn accelerates representative GPU learning applications by 3.17 times with existing PIM proposals.

3. Chapter 5 presents a systematic methodology providing a detailed understanding of the different stale value tolerance of different ML application. While reducing data communication can reduce the redundancy of data communication, it can reduce the progress of learning. We define the stale value tolerance with the effective learning rate, which is proportional to the size of delta when performing reduction operations. The effective learning rate, the delta size is dependent on the reduction method and reduction frequency, the type of model parameter, and multiple design choices of training neural network. While different training data might exhibit different behavior regarding progress since the optimal learning rate is different depending on training data, we assume that the design choices of training neural network affect the effective learning rate in a consistent direction. Our empirical evaluation revealed that this simple method works well providing reasonable explanations regarding the different stale value tolerance of different ML applications.

The trend of increasing amount of data and increasing number of ML applications will increase the importance of reducing learning time via parallel learning. To efficiently reduce the learning time with parallel execution, utilizing the stale value tolerance of learning is imperative. This dissertation serves as a starting point to investigate techniques to utilize this unique characteristic for more accurate and wider ML applications.

REFERENCES

[1] Google, *Google Self-Driving Car Project*, `https://www.google.com/selfdrivingcar`.

[2] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, *et al.*, "The Kaldi speech recognition toolkit," in *ASRU '11*, 2011.

[3] A. Mordvintsev, C. Olah, and M. Tyka, *Inceptionism: going deeper into neural networks*, `http://googleresearch.blogspot.co.uk/2015/06/inceptionism-going-deeper-into-neural.html`.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "GraphLab: A New Framework For Parallel Machine Learning," in *UAI '10*, 2010.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable Inference in Latent Variable Models," in *WSDM '12*, 2012.

[6] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," in *VLDB '12*, 2012.

[7] A. Agarwal and J. Duchi, "Distributed Delayed Stochastic Optimization," in *NIPS '11*, 2011.

[8] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *NIPS '11*, 2011.

[9] M. Zinkevich, J. Langford, and A. J. Smola, "Slow Learners are Fast," in *NIPS '09*, 2009.

[10] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent," in *KDD '11*, 2011.

[11] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch, "LazyBase: Trading Freshness for Performance in a Scalable Database," in *EuroSys '12*, 2012.

[12] Microsoft, *Azure*, https://azure.microsoft.com.

[13] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *MM '14*, 2014.

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: large-scale machine learning on heterogeneous systems*, 2015.

[15] R. Zajcew, P. Roy, D. L. Black, C. Peak, P. Guedes, B. Kemp, J. Loverso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala, "An OSF/1 UNIX for Massively Parallel Multicomputers," in *USENIX Winter*, 1993.

[16] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective Distributed Scheduling of Parallel Workloads," in *SIGMETRICS '96*, 1996.

[17] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, "The Impact of System Design Parameters on Application Noise Sensitivity," *Cluster Computing*, 2013.

[18] D. Terry, "Replicated Data Consistency Explained Through Baseball," *Commun. ACM*, 2013.

[19] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS '13*, 2013.

[20] K. Vora, S. C. Koduru, and R. Gupta, "ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM," in *OOPSLA '14*, 2014.

[21] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session Guarantees for Weakly Consistent Replicated Data," in *PDIS '94*, 1994.

[22] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *ISCA-39*, 2012.

[23] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "SIMD Re-Convergence At Thread Frontiers," in *MICRO-44*, 2011.

[24] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *PPoPP '11*, 2011.

[25] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO 40*, 2007.

[26] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA '10*, 2010.

[27] M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," in *ISCA-39*, 2012.

[28] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *NIPS 25*, 2012.

[29] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *USENIX ATC'14*, 2014.

[30] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous Large-Scale Graph Processing Made Easy.," in *CIDR*, 2013.

[31] Apache, *Hadoop*, http://hadoop.aparch.org.

[32] Apache, *Spark*, https://spark.apache.org.

[33] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. L. Scott, "Exploiting high-level coherence information to optimize distributed shared state," in *PPoPP '03*, 2003.

[34] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *ICML-13*, 2013.

[35] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *AISTATS10*, 2010.

[36] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *ICML-30*, 2013.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv*, 2015.

[38] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training Very Deep Networks," *arXiv*, 2015.

[39] J. Baxter and P. L. Bartlett, "Direct gradient-based reinforcement learning," in *ISCAS 2000*, 2000.

[40] M. Á. Carreira-Perpiñán and W. Wang, "Distributed optimization of deeply nested systems," in *AISTATS*, 2014.

[41] Y. Ollivier and G. Charpiat, "Training recurrent networks online without backtracking," *arXiv*, 2015.

[42] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, and K. Kavukcuoglu, "Decoupled Neural Interfaces using Synthetic Gradients," *arXiv preprint arXiv:1608.05343*, 2016.

[43] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPU," in *CGO-6*, 2008.

[44] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *PLDI '10*, 2010.

[45] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching for GPGPUs," in *MICRO-43*, 2010.

[46] N. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on GPUs," in *HPCA '14*, 2014.

[47] D. Tarjan, J. Meng, and K. Skadron, "Increasing Memory Miss Tolerance for SIMD cores," in *SC '09*, 2009.

[48] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC '14*, 2014.

[49] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architecture," in *MICRO '13*, 2013.

[50] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU," in *PPoPP '13*, 2013.

[51] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format," in *SC '14*, 2014.

[52] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems," in *SC '11*, 2011.

[53] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, “Micro-pages: increasing DRAM efficiency with locality-aware data placement,” in *ASPLOS ’10*, 2010.

[54] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev, “SAMS multi-layout memory: providing multiple views of data to boost SIMD performance,” in *ICS ’10*, 2010.

[55] B. Akin, J. C. Hoe, and F. Franchetti, “HAMLeT: Hardware accelerated memory layout transform within 3D-stacked DRAM,” in *HPEC ’14*, 2014.

[56] B. Akin, F. Franchetti, and J. C. Hoe, “Data Reorganization in Memory Using 3D-stacked DRAM,” in *ISCA ’15*, 2015.

[57] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “TOP-PIM: Throughput-oriented Programmable Processing in Memory,” in *HPDC ’14*, 2014.

[58] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch, “Sonic Millip3De: A Massively Parallel 3D-stacked Accelerator for 3D Ultrasound,” in *HPCA ’13*, 2013.

[59] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, “A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing,” in *3DIC ’13*, 2013.

[60] R. C. Murphy, P. M. Kogge, and A. Rodrigues, “The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems,” in *IMS ’00*, 2001.

[61] J. Jeddeloh and B. Keeth, “Hybrid memory cube new DRAM architecture increases density and performance,” in *VLSIT ’12*, 2012.

[62] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, “An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing,” *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[63] M. L. Chu, N. Jayasena, D. P. Zhang, and M. Ignatowski, “High-level Programming Model Abstractions for Processing in Memory,” in *WoNDP ’13*, 2013.

[64] L. Nai and H. Kim, “Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals,” in *MEMSYS ’15*, 2015.

[65] H. Kim, H. Kim, S. Yalamanchili, and A. F. Rodrigues, “Understanding Energy Aspects of Processing-near-Memory for HPC Workloads,” in *MEMSYS ’15*, 2015.

[66] Intel, *Intel xeon phi coprocessor vector microarchitecture*, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture, Intel.

[67] J. H. Ahn, M. Erez, and W. J. Dally, “Scatter-Add in Data Parallel Architectures,” in *HPCA ’05*, 2005.

[68] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker, “Active memory operations,” in *ICS ’07*, 2007.

[69] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally, “Executing irregular scientific applications on stream architectures,” in *ICS ’07*, 2007.

[70] L. G. Valiant, “A Bridging Model for Parallel Computation,” *Commun. ACM*, 1990.

[71] L. Liu and Z. Li, “Improving Parallelism and Locality with Asynchronous Algorithms,” in *PPoPP ’10*, 2010.

[72] P. Harish and P. J. Narayanan, “Accelerating Large Graph Algorithms on the GPU using CUDA,” in *HiPC ’07*, 2007.

[73] P. Harish, V. Vineet, and P. J. Narayanan, “Large Graph Algorithms for Massively Multithreaded Architectures,” International Institute of Information Technology, Tech. Rep., 2009.

[74] *DynoGraph*, https://github.com/sirpoovey/DynoGraph, Georgia Institute of Technology, 2014.

[75] D. Sanchez and C. Kozyrakis, “ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems,” in *ISCA-40*, 2013.

[76] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, “A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM,” in *WoNDP ’13*, 2013.

[77] S. Lloyd and M. Gokhale, “In-Memory Data Rearrangement for Irregular, Data-Intensive Computing,” *Computer*, 2015.

[78] C.-C. Chang and C.-J. Lin, “LIBSVM: a library for support vector machines,” *TIST*, 2011.

[79] M. Lichman, *UCI machine learning repository*, 2013.

[80] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv*, May 2016.

[81] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS ’12*, 2012.

[82] F. Sha and F. Pereira, “Shallow Parsing with Conditional Random Fields,” in *NAACL ’03*, 2003.

[83] J. Mayfield, P. McNamee, and C. Piatko, “Named Entity Recognition Using Hundreds of Thousands of Features,” in *CONLL ’03*, 2003.

[84] R. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, D. Sylvester, D. Blaauw, and T. Mudge, “Centip3De: A 64-Core, 3D Stacked Near-Threshold System,” *Micro, IEEE*, 2013.

[85] M. Scrbak, M. Islam, K. Kavi, M. Ignatowski, and N. Jayasena, “Processing-in-Memory: Exploring the Design Space,” in *ARCS 2015*, 2015.

[86] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-in-memory Accelerator for Parallel Graph Processing,” in *ISCA ’15*, 2015.

[87] J. H. Lee, J. Sim, and H. Kim, “BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models,” in *PACT ’15*, 2015.

[88] S. S. Keerthi and D. DeCoste, “A Modified Finite Newton Method for Fast Solution of Large Scale Linear SVMs,” *J. Mach. Learn. Res.*, 2005.

[89] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, “GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions,” in *SC ’15*, 2015.

[90] JEDEC, “Jesd235 high bandwidth memory (hbm) dram,” 2013.

[91] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: a fast and extensible dram simulator,” *IEEE Computer Architecture Letters*, 2015.

[92] J. S. Liptay, “Structural Aspects of the System/360 Model 85: II the Cache,” *IBM Syst. J.*, 1968.

[93] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, “Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency,” in *MICRO 41*, 2008.

[94] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, “High-performance distributed ML at scale through parameter server consistency models,” *arXiv*, 2014.

[95] G. Mann, R. Mcdonald, M. Mohri, N. Silberman, and D. D. Walker, “Efficient large-scale distributed training of conditional maximum entropy models,” in *NIPS ’09*, 2009.

[96] R. McDonald, K. Hall, and G. Mann, “Distributed Training Strategies for the Structured Perceptron,” in *HLT ’10*, 2010.

[97] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocky, “Rnnlm-recurrent neural network language modeling toolkit,” in *ASRU ’11*, 2011.

[98] K. Miller, M. P. Kumar, B. Packer, D. Goodman, and D. Koller, “Max-Margin Min-Entropy Models,” in *AISTATS ’12*, 2012.

[99] C.-N. J. Yu and T. Joachims, “Learning structural svms with latent variables,” in *ICML ’09*, 2009.

[100] M. P. Kumar, B. Packer, and D. Koller, “Self-paced learning for latent variable models,” in *NIPS ’10*, 2010.

[101] J. Bergstra and Y. Bengio, “Random Search for Hyper-parameter Optimization,” *J. Mach. Learn. Res.*, 2012.

[102] N. Srinivas, A. Krause, M. Seeger, and S. M. Kakade, “Gaussian process optimization in the bandit setting: no regret and experimental design,” in *ICML ’10*, 2010.

[103] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *NIPS ’11*, 2011.

[104] A. D. Bull, “Convergence Rates of Efficient Global Optimization Algorithms,” *J. Mach. Learn. Res.*, 2011.

[105] J. Snoek, H. Larochelle, and R. P. Adams, “Practical bayesian optimization of machine learning algorithms,” in *NIPS ’12*, 2012.

[106] J. Bergstra, D. Yamins, and D. Cox, “Making a Science of Model Search: Hyper-parameter Optimization in Hundreds of Dimensions for Vision Architectures,” in *ICML ’13*, 2013.

[107] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, “Long-term Recurrent Convolutional Networks for Visual Recognition and Description,” *arXiv*, 2014.

[108] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, “Recurrent models of visual attention,” *arXiv*, 2014.

[109] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: A neural image caption generator,” *arXiv*, 2014.

[110] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, “Building a large annotated corpus of English: The Penn Treebank,” *Computational linguistics*, 1993.

[111] Bakhtin, Anton and Edrenkin, Ilya, *Faster RNNLM (HS/NCE) toolkit*, https://github.com/yandex/faster-rnnlm.

[112] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, and P. Koehn, “One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling,” *arXiv*, 2013.

[113] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, “Asynchrony begets Momentum, with an Application to Deep Learning,” *arXiv*, 2016.

[114] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” *USSR Computational Mathematics and Mathematical Physics*, 1964.

[115] R. Eldan and O. Shamir, “The Power of Depth for Feedforward Neural Networks,” *arXiv*, 2015.

[116] S. Haykin, *Neural Networks: A Comprehensive Foundation*. 1998.

[117] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, 1997.

[118] K. Cho, B. Van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder–decoder for statistical machine translation,” in *EMNLP ’14*, 2014.

[119] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,” *arXiv*, 2014.

[120] G. Hinton, N Srivastava, and K. Swersky, “Lecture 6a overview of mini–batch gradient descent,”