Stochastic Gradient Descent on GPUs*

Rashid Kaleem
rashid@cs.utexas.edu

Sreepathi Pai
sreepai@ices.utexas.edu

Keshav Pingali
pingali@cs.utexas.edu

The University of Texas at Austin
Austin, Texas, USA

ABSTRACT

Irregular algorithms such as Stochastic Gradient Descent (SGD) can benefit from the massive parallelism available on GPUs. However, unlike in data-parallel algorithms, synchronization patterns in SGD are quite complex. Furthermore, scheduling for scale-free graphs is challenging. This work examines several synchronization strategies for SGD, ranging from simple locking to conflict-free scheduling. We observe that static schedules do not yield better performance despite eliminating the need to perform conflict detection and resolution at runtime. We identify the source of the performance degradation to be the structure of certain parts of the graph (dense vs sparse). This classification can be used to devise hybrid scheduling strategies which exploit different schedules for different regions of the graph to obtain better performance. We found that the best schedule for some problems can be up to two orders of magnitude faster than the worst one. To evaluate the performance of our GPU implementation, we also compare against a CPU implementation of SGD. Dynamic schedules perform comparably to a 14-thread CPU implementation, while a static schedule performs comparably to a 6-thread CPU implementation.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; E.1 [**Data Structures**]: Graphs and Networks

General Terms

Algorithms, Design, Performance

Keywords

GPGPU, Stochastic Gradient Descent, Edge-coloring

*The work presented in this paper has been supported by the National Science Foundation grants CNS 1111766, CCF 1218568, XPS 1337281, and CNS 1406355.

GPGPU-8, February 7, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3407-5/15/02...$15.00
http://dx.doi.org/10.1145/2716282.2716289

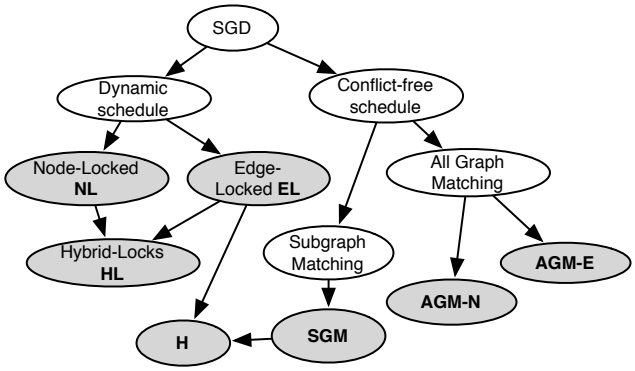


Figure 1: Taxonomy of different schedules, gray boxes represent actual schedules.

1. INTRODUCTION

Stochastic gradient descent (SGD) is a gradient descent optimization strategy used to minimize certain kinds of objective functions that arise in big data machine learning problems like building recommender systems. One example of such a problem is the Netflix challenge: *given a database of movie ratings from a set of users, predict the rating a given user might give to a movie he or she has not seen.* SGD can be used to solve this problem, as described in Section 2. The database of ratings is modeled by a bipartite graph in which one set of nodes represents users and the other set of nodes represents movies; if a user a gives a rating of l to a movie b, there is an edge (a, b) with weight l. An example is shown in Fig. 4(a). The SGD algorithm operates in rounds; in each round, it traverses all edges in some order and updates labels at the end-points of each edge using a computation described in more detail in Section 2. Parallelism can be exploited by processing edges in parallel. However, threads have to update node data exclusively, so we need some form of synchronization.

A previous study by Dean *et al.* [5] concluded that GPUs are not suitable for SGD. One key issue is the need for fine-grain synchronization, which can be expensive on GPUs. Moreover, SIMD execution inflates the cost of thread divergence from synchronization conflicts (failure to acquire locks). Finally, in scale-free graphs [1], some nodes have a large number of neighbors while most nodes have a small number of neighbors. In the movie-user example, nodes corresponding

$$U.M = A$$

$$\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} . \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \end{bmatrix} = \begin{array}{|c|c|c|c|} \hline a & f & & k \\ \hline b & g & & l \\ \hline c & & & \\ \hline & & i & \\ \hline d & h & & \\ \hline e & & j & \\ \hline \end{array}$$

Figure 2: Matrix completion for matrices U, M and the desired result A.

to popular movies would be high-degree nodes. These high-degree nodes pose a challenge for efficient scheduling because they conflict with many other nodes.

In this paper, we explore the performance trade-offs of different scheduling strategies for implementing SGD on GPUs, with the goal of extracting general lessons for implementing irregular algorithms on GPU. The contributions of this paper are as follows.

1. We explore the performance trade-offs of different synchronization strategies for implementing SGD on GPUs with the goal of extracting general lessons for implementing irregular algorithms on GPUs. We believe this is the first study to evaluate different schedules for SGD on the GPU.

2. We develop and investigate three different static schedules with varying levels of performance. We show that simple static schedules can be too conservative and show how to relax those constraints while maximally utilizing the underlying hardware.

3. We present hybrid schedules which exploit the performance benefits of each schedule on different regions of the graph.

4. We evaluate the different schedules on different scale-free real world inputs and show how good performance can be obtained on such inputs.

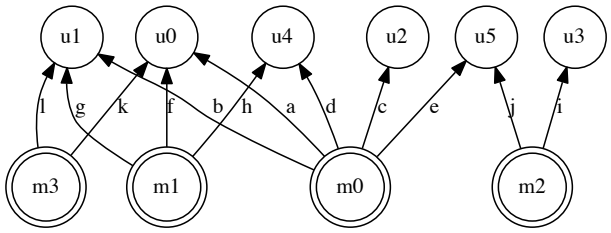
The rest of the paper is organized as follows. Sec. 2 describes the SGD problem in more detail and lays the foundation for the rest of the paper. Sec. 3.1 describes dynamic schedules in which conflicts are resolved at runtime and Sec. 3.2 describes the static schedules, which are generated off-line and do not require run-time conflict detection and resolution. This is followed by an evaluation of the different schedules in Sec. 4. We discuss related work in Sec. 5, followed by directions for future work in Sec. 6 and conclude in Sec. 7.

2. PROBLEM STATEMENT

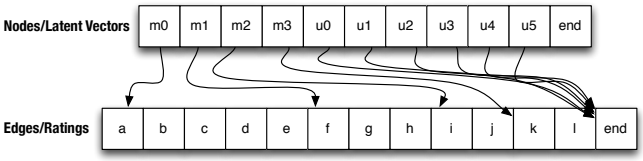
Consider a set of movies and users. Each user has assigned ratings to some movies. We can represent these ratings as a matrix in which rows represent users and columns represent movies, as shown in Fig. 2. Each non-zero entry, shown as a letter, represents the numerical rating assigned to the movie by the corresponding user. We want to predict the ratings for the empty cells in the matrix to decide which movies should be recommended to that user.

```
sgd_update(Movie m, User u, Rating r){
   delta = abs(f(m.lv, u.lv)-r);
   m.lv = update_lv(m.lv, delta);
   u.lv = update_lv(u.lv, delta);
}//End sgd_update
```

Figure 3: Pseudo-code for SGD update.



(a) Sample graph



(b) CSR representation

Figure 4: (a) Graph representation, (b)Compressed sparse row (CSR) representation.

One way to solve this problem is to use *matrix completion*. The goal is to find two dense low-rank matrix U and M such that the product UM approximates A; that is, each non-zero entry in A must be close in value to the corresponding entry in the product UM. Note that the product UM produces values even for entries in A where the value is zero; these are considered to be the predicted ratings for each user and movie combination.

As mentioned before, it is convenient to think of this computation in terms of graphs. The matrix A is considered to be a bipartite graph that has an edge for each non-zero (user,movie) entry in A. The matrices U and M can be considered to be the composition of *latent* or *feature* vectors, where each user and movie is assigned one vector. For our example, the latent-vector product for u_3 and m_2 should produce a value close to i. If this product is not equal to i, SGD updates the values of u_3 and m_2 to obtain a better approximation, as shown in Fig. 3. The overall computation consists of a series of supersteps; in each superstep, all edges are visited in some order and the latent vector update computation is performed at each edge. At the end each superstep, the RMS error is computed and if this is below some threshold, the computation is terminated.

Since the edges of the bipartite graph can be processed in any order in each superstep, SGD is an example of an *unordered* algorithm in the TAO classification of algorithms [8]. Moreover, a subset of edges can be processed in parallel provided they form a *matching*; that is, no two edges in the graph share a node. The implementations discussed in this paper are concerned with finding such matchings either explicitly, in a preprocessing step, or implicitly, during SGD computation, using speculative execution.

```
void edge_operator(Graph, graph){
   parallel_for(edge e : graph){
      if(e.unmarked){
         if(lock(e.movie and lock(e.user)){
            sgd_update(e.movie, e.user, e.rating);
            e.unmarked=false;
          } //end lock
      }//end if unmarked
   }//end parallel-for
}//end edge_operator
void edge_locked(Graph graph){
   while(any unmarked edge in graph){
      edge_operator(graph);
   }
}
```

Figure 5: Pseudo-code for edge-locked version

T	t_0	t_1
0	a	
1	c	
2	e	f
3	g	
4	i	
5	k	
6		b
7		d
8		h
9		j
10		l

(a) Edge-lock schedule without shuffle.

T	t_0	t_1
0	k	e
1	b	
2	j	c
3	h	
4	f	
5	i	l
6		d
7		g
8		a

(b) With shuffle.

T	t_0	t_1
0	a	
1	b	
2	c	h
3	d	
4	e	
5		f
6		g
7	i	k
8	j	l

(c) Node schedule.

Figure 6: Schedules observed for sample input under EL(without and with shuffle) and NL on the hypothetical GPU. Each row indicates edges scheduled at that time slot, and each column indicates the item processed, if any, by each thread.

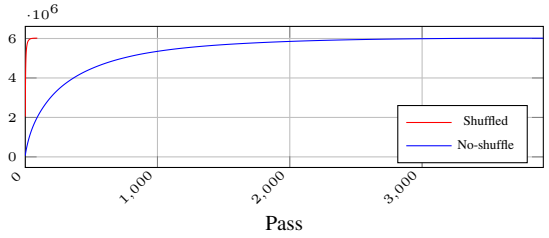


Figure 7: Edges processed by edge-locked version. Horizontal axis shows iterations and vertical axis shows number of edges processed. Shuffled version (92) terminates early due to less conflicts, whereas un-shuffled (3,916) version takes significantly more iterations due to intra-block conflicts for high-degree nodes (max degree is 43,331) in the graph.

3. SGD IMPLEMENTATION

A parallel algorithm can be viewed as the composition of an *operator* and a *schedule* [8]. For SGD, the operator is `sgd_update` (Fig. 3). The schedule governs the order in which the operator is applied to the graph. We explore seven scheduling schemes classified broadly into *dynamic schedules* and *static schedules* (Fig. 1)

For the purpose of exposition, we consider a hypothetical GPU with a single execution unit (streaming multiprocessor) capable of running two threads (2-SIMT) to illustrate the execution of different schedules. Due to lack of synchronization facilities on GPUs, some schedules may need to make multiple passes to apply the SGD update to every edge.

3.1 Dynamic Schedules

The dynamic schedules use locking primitives to ensure mutual exclusion and correctness when applying `sgd_update`. We propose three dynamic schedules, based on the order of processing.

3.1.1 Edge-Locked (EL)

The Edge-Locked (EL) schedule assigns individual *edges* of the input graph to individual threads. The edges are maintained via Coordinate Format (COO) [3] representation making it easy to access the source, destination and rating for any particular index. Each thread first locks the source (i.e. movie) and destination (i.e. user) node of an edge assigned to it. If the locks are successfully acquired, `sgd_update` is applied to the edge. If lock acquisition failed, the edge is deferred to be processed in the next pass. The algorithm in Fig. 5 shows the pseudo-code for the EL schedule. The algorithm repeats until every edge is processed.

One possible schedule is shown in Fig. 6(a). which takes 11 steps. The schedule in Fig. 6(a) can be explained as follows. The initial ordering of the edges is the same as in Fig. 4(b), and since our hypothetical GPU can execute two tasks at once, it will pick chunks of two items from the work-list and execute them. The first two items are $\{a, b\}$. Note that these two share the same source m_0, and hence only one of them succeeds in acquiring the lock, the other thread sits idle. The next chunk to be executed are $\{c, d\}$, again, which share the source, so only one of them gets processed while the other is moved to the next *pass*. This completes the first pass over all the edges, indicated by a double line. The second pass processes all the remaining edges $\{b, d, h, j, l\}$. No further passes are required. So, the execution required two passes and eleven steps.

One problem that becomes immediately apparent is that if edges from the same movie are processed concurrently, only one of the threads will make progress. For instance edges $\{a, b\}$ share the same movie node m_0 and hence cannot be processed in parallel. Since the COO format was derived from the CSR format which lays out the edges of the same node in adjacent memory locations, this introduces a large number of conflicts which persist through passes.

Therefore, we shuffle the edges in the initial worklist. The goal is to ensure that edges for the same movie are no longer adjacent in the shuffled worklist. This lowers the likelihood that those edges, sharing the same movie, are scheduled concurrently and hence conflict on the source movie. For our sample graph, we shuffle the edges (for instance to $\{k, e, b, d, j, c, h, g, f, a, i, l\}$) and obtain a schedule as shown in Fig. 6(b). Notice how the first four steps now process 6 edges even though there are three conflicts; between the edge pairs (b, d), (h, g), and (f, a). By mixing the edges of m_0 with other nodes, we perform more useful work. Experiments on actual input graphs confirm that shuffling can improve performance significantly. For the BGG input (Section 4),

```
void node_operator(Graph graph){
   parallel_for(movie m : graph){
      //Copy m.lv to shared memory
      for(user u : m.neighbors){
         if(edge(m, u) is unmarked){
            if(lock(u)){
               sgd_update(m, u, edge(m,u));
               unlock(u);
               mark(u);
            }//if-locked
         }//if-unmarked
      }//end for-neighbors
      //write m.lv to global memory
   }//end parallel-for
}//end node_operator

void node_locked(Graph graph){
   while(any unmarked edges in graph){
      node_operator(graph);
   }
}
```

Figure 8: Pseudo-code for node-locked version

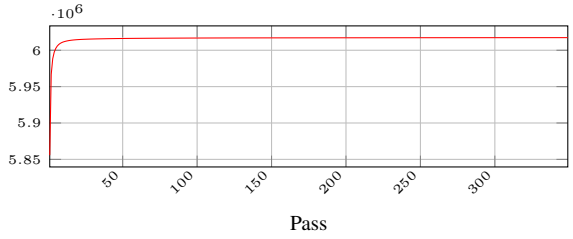


Figure 9: Edges processed as the node-locked version proceeds. Horizontal axis represents iteration number, and vertical axis represents number of edges processed up to that point. Termination occurs (350 iterations) when all the edges ($6,017,340$ for BGG) have been processed.

processing the shuffled worklist takes 92 passes whereas the unshuffled worklist takes 3916 passes (Fig. 7).

3.1.2 *Node-Locked (*NL*)*

The Node-locked (NL) schedule assigns individual *nodes* to individual threads. This has two benefits. First, there is no need to acquire locks on the source node (i.e. movie) anymore since each thread is assigned a distinct source node. Locks will still need to be acquired on the destination nodes (i.e. user). Second, unlike the EL schedule whose access patterns make it hard to exploit locality, the NL schedule can exploit reuse of the source node data.

Like the EL schedule, the NL schedule uses multiple passes to process all the edges of a graph. The pseudo-code for NL is shown in Fig. 8.

Fig. 6(c) presents a possible NL-schedule We schedule nodes m_0 and m_1 first on the GPU, and the edges are processed in order. In the first step, all the edges for m_0 are processed while f and g are deferred to the next pass. In the next pass, f and g, from the previous pass, are processed. Next, we schedule the remaining nodes m_2 and m_3, which concludes without any conflict.

NL behaves similar to EL for the initial few passes as it can find work easily. But after the initial few passes, there

```
//worklist of edges edge_wl;
void node_operator(Graph graph, WL edge_wl){
   parallel_for(movie m : graph){
     //Copy m.lv to shared memory
     for(user u : m.neighbors){
           if(lock(u)){
              sgd_update(m, u, edge(m,u));
              unlock(u);
           } else{
              edge_wl.push_back(edge(m,u));
           }
        }
     //write m.lv to global memory
   }//end parallel-for
} //end node_operator
void edge_operator(Graph graph, WL edge_wl){
   parallel_for(edge e: edge_wl){
      if(lock(e.movie)) {
         if(lock(e.user) ){
            sgd_update(e.movie,e.user,e.rating);
            unlock(e.user);
         }
         unlock(e.movie);
      }
   }//End parallel-for
}//End edge_operator
void hybrid_locked(Graph graph){
   WL edge_wl;
   node_operator(graph, edge_wl);
   while(!edge_wl.emptt()){
      edge_operator(graph, edge_wl);
   }
}
```

Figure 10: Pseudo-code for hybrid-locked version

is a large overhead of finding new work as each thread has to scan a node's entire edge-list. This can be prohibitive for high-degree nodes as the repeated scans become expensive. The behavior of NL can be viewed in Fig. 9, where we show the number of edges processed (vertical axis) across passes (horizontal axis). We observe that the number of edges processed in the first few passes is very high but overall termination takes very long.

The use of shared memory for storing the movie node's latent vector reduces the residency of the kernel, which means the number of edges that can be concurrently processed on the GPU is reduced. Further, since only one thread processes all edges of a node, nodes with high degrees can lead to serialization and load imbalance. The use of marks implies that all edges must be scanned in every pass to determine if they must be processed. As we shall see in the evaluation, these factors play a major role in the performance of NL.

3.1.3 *Hybrid-Locked (*HL*)*

The Hybrid-Locked (HL) schedule was motivated by experimental evaluation of the EL and NL schemes. We observed that the NL scheme processes a significant fraction of edges in the first pass. However, since its succeeding passes must scan the entire graph in order to discover unmarked edges, they exhibit high overhead.

Therefore, HL combines the EL and NL schemes. The first pass over the input graph uses a NL-like scheme which moves conflicting edges to a worklist instead of using marks. All later passes use the EL scheme to process these conflicting edges.

```
void build_schedule(Graph g){
   MatchingSet m;
   while(edges remaining in g){
      Matching m_i = matching(g);
      //remove edges in m_i from g;
      g = g - m_i;
      m[i] = m_i;
      i+=1;
   }
}
```

Figure 11: Build the matching for the graph.

```
   agm_e_operator(Graph graph, MatchingSet m){
      for (Matching curr_set : m){
         parallel_for(Edge e : curr_set){
            sgd_update(e.movie,
                       e.user, e.rating);
         }//End parallel for
      }//End for matching
   }//End operator
```

Figure 12: Pseudo-code for AGM-E.

3.2 Static Scheduling

EL, NL, and HL use locks at runtime to ensure mutual exclusion. Alternatively, conflict-free subsets of edges in the input graph can be identified in a preprocessing step and processed without locks during the SGD computation. Such conflict-free sets correspond to the *matchings* [6] of the bipartite input graph. To process an input graph, we need to obtain a set of matchings that cover the entire input graph. We do this by *(i)* extracting a matching from a graph and, *(ii)* removing its edges from the graph, *(iii)* repeating the process on the resulting graph. We explore three different implementations of off-line scheduling, two of which compute matchings over the entire graph and one that only computes matchings for a portion of the graph.

3.2.1 All-Graph Matching (AGM)

In the *All-graph matching scheme*, we build a set of matchings for the entire graph. Each edge is associated with a matching which can be numbered from 0 to at least *max_degree*, where *max_degree* is the maximum degree of any node. The maximum degree in the graph is thus the minimum length of the critical path. All the edges belonging to a particular matching can be safely processed in parallel without any locks since they do not share any end-points. However, matchings must be processed serially. The pseudo-code for constructing the matchings is given in Fig. 11. We repeatedly find matchings m_i in the graph via matching, which represent an independent set of edges. These are stored in a Matching object and removed from the graph. The process is repeated until all the edges have been assigned to a matching set.

The matchings for the sample graph are given in Fig. 13(a), where each row lists a different matching. The sample graph contains five matchings. In our implementation, we actually use two different schemes to do this, which we describe below.

Edge Schedule (AGM-E)

The edge-schedule (Sec. 3.1.1) is applied to each edge in a matching, except locking is not needed. Further, multiple passes over a matching are not needed, since none of the edges in a matching conflict. This also obviates the need for shuffling. However, all matchings must be processed serially.

The pseudo-code for the lock-free edge version is given in Fig. 12. Given a set of matchings m, we process all the matching serially, processing all the edges of each matching concurrently on the GPU.

For our sample graph, a lock-free edge schedule is given in Fig. 13(b). In our example, we can only execute two items per step, so we have to complete execution of each matching (2 items from a matching at a time) and move to next matching. So, the first two steps are used to process edges belonging to the first matching $\{a, g, i\}$. Once the first matching is completed, execution of the second matching proceeds, and this process repeats until all matching sets have been processed. Our example requires eight steps to complete the execution.

```
agm_n_operator(NodeSet ns){
   parallel_for(movie n : ns){
      for(Edge e: n){
         sgd_update(n, e.dst, e.rating);
         barrier();
      }
   }
}
void agm_n(Graph g, MatchingSet m){
   //NodeSet is set of nodes that
   // can concurrently run on GPU.
   for(NodeSet ns : g){
      for(Node n : ns){
         //sort neighbors in matching order
         sort(n.edges, m);
      } //End for Node
   } //End for NodeSet
   for(NodeSet ns: g){
      agm_n_operator(ns)
   }//end for-NodeSet
}
```

Figure 14: Pseudo-code for AGM-N.

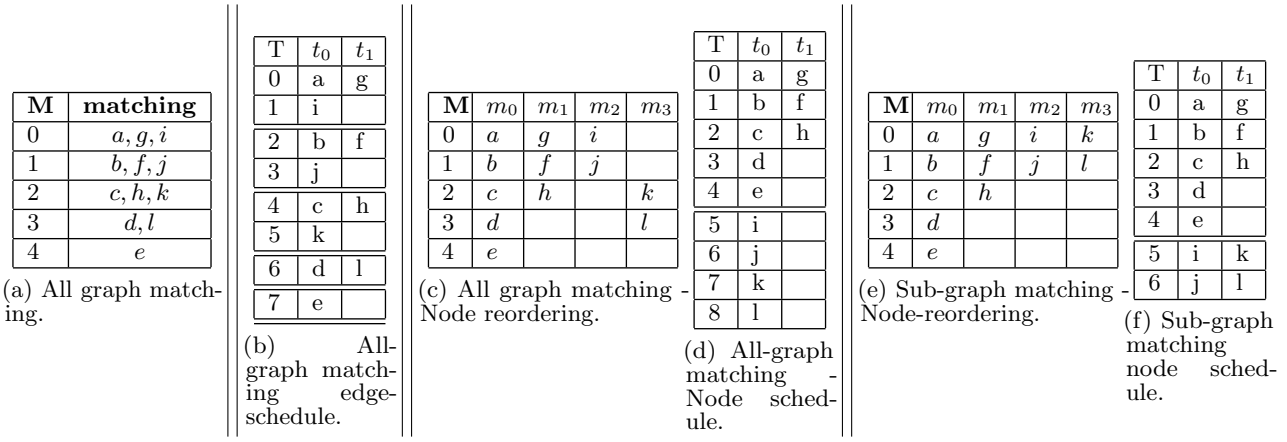
Node Schedule (AGM-N)

The AGM-N schedule assigns individual nodes to individual threads as in the dynamic NL schedule, but uses the information from the matching to avoid locking. The basic idea is to associate with each edge of a node a *time-stamp* at which it is safe to process that edge. All edges with the same time-stamp can be processed in parallel without conflicts. The time-stamp value is obtained from the computed matchings.

Like NL, AGM-N exhibits better locality and reuse. To achieve synchronous execution, we invoke the kernel once and use global inter-thread block synchronization to ensures all threads process the same time-stamp edges in parallel.

The pseudo-code for AGM-N is shown in Fig. 14. It uses the same matching set constructed via Fig. 11. However, in order to ensure that each thread processes edges belonging to the same matching synchronously, we need to sort the edges for each node according to the matching-set they blong to. Once we have sorted the edges for each node we can process the nodes. This is achieved by device-side barriers, which limit the number of threads that can simultaneously run on the GPU.

The node-schedule computed via the all-graph matching



M	matching
0	a, g, i
1	b, f, j
2	c, h, k
3	d, l
4	e

(a) All graph matching.

T	t_0	t_1
0	a	g
1	i	
2	b	f
3	j	
4	c	h
5	k	
6	d	l
7	e	

(b) All-graph matching edge-schedule.

M	m_0	m_1	m_2	m_3
0	a	g	i	
1	b	f	j	
2	c	h		k
3	d			l
4	e			

(c) All graph matching - Node reordering.

T	t_0	t_1
0	a	g
1	b	f
2	c	h
3	d	
4	e	
5	i	
6	j	
7	k	
8	l	

(d) All-graph matching - Node schedule.

M	m_0	m_1	m_2	m_3
0	a	g	i	k
1	b	f	j	l
2	c	h		
3	d			
4	e			

(e) Sub-graph matching - Node-reordering.

T	t_0	t_1
0	a	g
1	b	f
2	c	h
3	d	
4	e	
5	i	k
6	j	l

(f) Sub-graph matching node schedule.

Figure 13: Static-schedules executed by different strategies for sample input. Tables with M in top-left cell indicate matching sets, whereas tables with T in top-left cell indicate schedules where each row indicates a time-step and each column lists the edges processed by a thread.

is displayed in Fig. 13(d). We see that it takes four steps to execute the second block of nodes consisting of $\{m_2, m_3\}$. However, upon observing the original graph, we see that edges for m_2 and m_3 can be processed in parallel since they are independent; i and j can be run in parallel with k and l since they do not share any user node. This leads us to conclude that the matchings computed over all the edges of a graph are too conservative for a node-schedule if the number of threads (movies) to be executed simultaneously is smaller than the total number of movies. In order to build an offline schedule per-node, we need to limit the edges used for the constructing the matching only to movies that will be executed simultaneously on the GPU.

3.2.2 Sub-Graph Matching (SGM)

The SGM schedule improves on the AGM-N version by computing matchings over a subset of the graph. Each subset groups movies into blocks. Only movies in the same block will be processed simultaneously. Thus, the matching is built only for those edges that are expected to be processed concurrently.

The sub-graph node-schedule for the sample graph is shown in Fig. 13(b). Compared to the all-graph matching schedules, we see that edges of m_2 and m_3 can be processed in parallel since they do not share endpoints.

We observe that the number of steps taken to process a node reduces in SGM. Consider the number of steps for each node-set in AGM-N as shown in Fig. 13(c). For the subset (m_0, m_1), there are five steps since each there are edges for at least one of the nodes in the five matching sets. Similarly for the subset (m_2, m_3), the number of steps is four, as each node has an edge in at least one of the four matching sets. Contrast this with SGM as shown in Fig. 13(e), where the subset (m_2, m_3) only has two steps to process all the edges. SGM scheduling, by not considering conflicts across partitions of nodes, produces schedules which are less conservative.

3.3 Hybrid (H)

We observe that the SGM schedule performs well, but generating the schedule takes a long time. In particular, the nodes with the highest degree take the longest time since they participate in the largest number of matchings, proportional

```
sgm_n_operator(NodeSet ns, Matching m){
   parallel_for(movie n : ns){
      for(Edge e: n){
         sgd_update(n, e.dst, e.rating);
         barrier();
      }//end for edge
   }//end parallel-for
}//end sgm_n_operator

MatchingMap build_sub_matching(Graph g){
   MatchingMap map;
   //NodeSet is set of nodes that
   // can concurrently run on GPU.
   for(NodeSet ns : g){
      m = build_schedule(g,ns);
      map[ns] = m;
      for(Node n: ns){
        //sort edges of n w.r.t m
      }//end for node
   }//end for node-set
}//end build_sub_matching

void sgm_n(Graph g){
   MatchingMap map = build_sub_matching(g);
   for(NodeSet ns: g){
      agm_n_operator(ns, map[ns]);
   }//end for-NodeSet
}//end sgm_n
```

Figure 15: Pseudo-code for SGM-N.

to their degrees. We can avoid the overhead of building schedules for SGM by processing the edges for the highest degree nodes with EL, and processing the remaining nodes via SGM. This has two benefits: *(i)* the conflicts in the high degree nodes are handled by the shuffled EL schedule and *(ii)* the SGM schedule for the lower degree nodes does not have to handle conflicts with the higher degree nodes, which otherwise lead to longer schedules.

The Hybrid schedule H partitions nodes into two subsets based on the degree of a node. We first order the nodes by their out-degree, and partition them into subsets to build the matching (for instance (m_0, m_1) and (m_2, m_3)). The first partition (with highest degree nodes) will be processed via

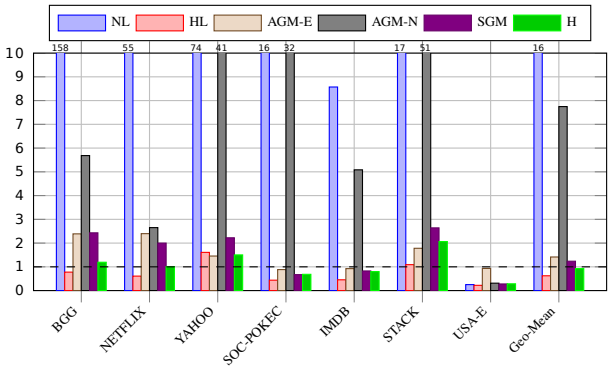


Figure 16: Normalized execution time for all schedules with respect to the EL schedule. Lower is better.

Table 1: Nodes and Edges for the input graphs. EL is the absolute runtime for EL schedule and is used as the baseline for the results. Runtimes were averaged over 5 runs.

Input	V	E	MaxDegree	EL
BGG	155,998	6,017,340	43,331	0.21s
NETFLIX	497,959	99,072,112	227,715	1.94s
YAHOO	1,625,951	252,800,275	463,820	5.55s
SOC-POKEC	1,632,803	22,301,964	14,854	0.41s
IMDB	1,324,745	3,782,462	1,590	0.063s
STACK	641,876	1,302,439	6,121	0.038s
USA-E	7,197,246	8,778,114	9	0.19s

an EL schedule. The remaining nodes use a SGM schedule.

4. EVALUATION

We present an evaluation of the proposed scheduling schemes. We discuss the overall performance of each schedule, compare the dynamic and static schedules and also provide a comparison with state-of-the-art CPU implementation.

4.1 Methodology

We implement SGD in OpenCL 1.1 (the latest supported by NVIDIA) and execute the different implementations on a NVIDIA Tesla K40C. The GPU has 12GB of memory but we can only use 4GB due to NVIDIA's OpenCL implementation. The host machine is a Quad core Intel Xeon E5-2609 with 32GB system memory running Scientific Linux 6.6 with Linux kernel 2.6.32.

4.2 Inputs

Table 1 describes the inputs used in our study. All the inputs, except USA-E, are power-law graphs. USA-E is derived from the road network graph and thus exhibits a uniform degree distribution – we use this graph to isolate issues peculiar to power-law graphs. Since USA-E is not a bipartite graph, we direct the edges and duplicate the nodes such that one copy has only outgoing edges (hence a movie) and the other copy only has incoming edges (hence a user). The size of the latent vectors is 16 floats.

4.3 Optimizations

We briefly describe the optimizations made to each implementation to assist reproduction of results. All of the implementations use the float16 vector data type supported by OpenCL.

1. EL – We use a COO (Coordinate format) to represent the edges. The edges are shuffled prior to building the COO representation to ensure less intra-warp conflicts due to edges from the same source.

2. NL – The implementations uses shared memory to store the source feature vectors and goes over the neighbors of each movie to process edges. Instead of maintaining two work-lists, which would require twice the space, we mark edges as processed and skip them in subsequent passes.

3. AGM-E – The implementation uses a device-side barriers between each matching set. Each thread is assigned one or more edges from a matching to process, which it can do concurrently, followed by a device-wide barrier. Once all the threads reach the barrier, the next matching set can be processed.

4. AGM-N – The implementation uses shared memory to store the source node and goes over all the neighbors. It also employs device-side barriers, and hence not all the sources will be processed concurrently. The updated feature vectors for the source node are read once at the start, and written to global memory at the end of the pass.

5. SGM – The implementation is similar to AGM-N, with a less conservative matching to produce better schedules.

4.4 Overall Performance

Fig. 16 presents runtimes of the different schedules normalized to the runtime of EL schedule.

Overall, we observe that HL is the best performing schedule. Recall that HL works by first executing NL over the input graph *once*, followed by repeated invocations of EL on any unprocessed edges. In this case, NL processes 78% to 98% of the edges leaving the rest to EL. Repeated invocations of NL alone are counter-productive because NL must always scan the graph's edges to identify unprocessed edges, unlike EL which maintains a work-list of unprocessed edges. The pure NL schedule thus executes slowly. HL also outperforms EL overall because the majority of edges are processed by NL which has a significantly lower number of global memory transactions due to use of shared memory and which also uses fewer locks. HL does not perform well on YAHOO, due to the poor throughput for NL. This is due to multiple thread-blocks containing high-degree movies causing many thread-blocks to process a large number of edges. If we sort the movies by degree before we schedule them to the GPU (to reduce divergence), the time to process the graph goes down (first pass takes 5.9s compared to 8s for the unsorted movies), but the percentage of edges processed goes down as well (from 98% to 84%) due to increased conflicts between the high-degree movies which are now more likely to be scheduled concurrently.

4.5 Static vs Dynamic Schedules

Perhaps surprisingly, the static schedules AGM-E, AGM-N and SGM are slower overall compared to the dynamic schedules even when the time to construct the schedules is not taken into account as in Fig. 16. The only exceptions

Table 2: Input features and relative running times for different schedules. Running times are normalized to EL schedule, which has absolute times listed. Absolute running times for CPU are also given; one for 40 threads, and another closest to the EL with the number of threads in parenthesis. All running times are averaged over 5 runs and use a 16-float vector.

	Bgg	Netflix	Yahoo
40T-CPU(s)	$0.048s$	$0.294s$	$2.561s$
CPU(s)	$0.221s(4T)$	$1.895(6T)s$	$5.356s(14T)$

are SOC-POKEC and IMDB, which have a relatively low max-degree allowing for better static schedules. The best-performing static schedule is the SGM schedule which is $2\times$ slower than HL overall, and is never faster than HL for any input.

Firstly, SGM outperforms the other static schedules because it is less conservative than the AGM-E and AGM-N. Recall that SGM partitions the nodes of the graph and prepares a matching for each partition separately. Since SGM guarantees never to execute two partitions concurrently, the resulting schedule is correct. The partitioning captures the fact that the number of nodes in the input graph is much larger than the number of threads executing concurrently on the GPU. Thus, not all conflicts in the graph will necessarily occur at runtime. In contrast, AGM-E and AGM-N assume that *all* nodes may be processed concurrently and hence they produce very conservative schedules that underutilise the GPU hardware. In particular, since each edge of a particular node must occur in a different matching, the length of the AGM-E and AGM-N schedules is at least the maximum node degree in the graph Table 1.

Secondly, SGM is unable to outperform the dynamic schedules because its schedules are still too conservative. The dynamic schedules process edges optimistically, using locking to ensure mutual exclusion and correctness. For the majority of inputs, this produces schedules that execute in fewer steps than any statically determined schedule. Significantly, although static schedules save on conflict detection and re-execution, any gains obtained are not enough to match the performance and flexibility of dynamic schedules. Thus, unless criteria such as ensuring deterministic execution are important, dynamic schedules should be preferred over static schedules.

4.6 CPU Comparison

We also evaluate the performance of our GPU implementation against a state of the art CPU multi-threaded implementation [10] running on a 40-core Xeon E7-4860. We present both the 40-thread runtime as well as for thread counts with the closest runtime to the EL schedule. We can see that the EL-based GPU implementation is competitive with a CPU implementation, and for larger inputs, is equivalent in performance to a 14-threaded CPU implementation. This combination can be very useful in modern systems, which can be equipped with multiple GPUs, as a heterogeneous implementation of SGD can utilize both the CPU and the GPU to compute more efficiently. Furthermore, the partitioning between the two devices can yield further benefits; GPU can be allocated more denser regions whereas the CPU can process the sparser regions.

5. RELATED WORK

GPUs have been used to accelerate machine learning applications in many systems [9, 4, 12], however SGD is not a common candidate for parallelization. GPU A-SGD [7] exploits both *model* and *data* parallelism to speed up neural network training for computer vision. They evaluate GPU A-SGD on the ILSVRC 2012 input ($1.2M$ training images and $150K$ validation images labelled with $1K$ categories) and report 10.7 days on a single GPU, scaling to 3.3 days on 8 GPUs. An alternate approach is to use SPMV routines to compute the gradient, and port the computation to GPUs. A study [2] found that despite the speedup, the programming overhead was too large. Siede *et. al* [11] investigate the theoretical efficiency of model and data parallel SGD, and conclude that fundamental change in the training algorithm is necessary to obtain any speedup. Dean *et al.* [5] compare the performance of a GPU implementation to for speech model training (about 1.1 billion labelled inputs) and observed a large overhead compared to a CPU cluster. However, none of the aforementioned studies investigate different scheduling strategies for the GPU. As our experiments show, a poor scheduling strategy can be up to $158\times$ slower than the fastest schedule. We believe that scheduling is essential for better performance on the GPU.

6. FUTURE WORK

The performance of SGD can be further improved by distributing the graph across multiple GPUs. Here, each GPU can utilize the best scheduling strategy based on the characteristics of the partition being processed. A two-dimensional tiling can be used to partition the rating matrix along the movie and user dimension. Under this scheme, partitions along a diagonal can be scheduled to execute independently on different devices. However, now the scheduling decision also needs to consider the cost of data movement for the latent vectors across different devices.

The performance on a single device can be further improved by optimizing for memory access to the global memory. In particular, the matching schedules do not take into account the memory access patterns for each thread. Multiple memory transactions per-instruction can incur large performance overheads. However, the length of latent vectors (typically larger than a hundred bytes) prohibits coalesced memory accesses for different threads, even if they are operating on adjacent movies or users. Better data-layouts can be explored to increase coalesced accesses for the latent vectors.

7. CONCLUSION

To the best of our knowledge, this is the first study comparing the alternative *scheduling* for SGD. We show that SGD can execute efficiently on the GPU, and that a dynamically scheduled implementation is comparable to a 14-thread CPU implementation. Dynamic schedules like EL rely on the randomness of edge selection and in the underlying hardware to produce efficient schedules. Further performance improvements can be obtained by the hybrid HL schedule, which utilizes the on-chip shared memory while avoiding the overhead associated with a NL scheme. Static schedules, on the other hand, sacrifice performance for determinism. The static schedules, AGM-E, AGM-N, and SGM use matchings to construct schedules which can be executed without any conflict-detection. The fastest static schedule, SGM, performs

comparably to a 6-thread CPU implementation.

The scale-free nature of real-world inputs poses a challenge to data-parallel implementations for applications. In particular, high-degree nodes pose a performance bottleneck. To efficiently process such inputs, we need to address these high-degree nodes. Given the large variance in graph structure (degrees of nodes), it is not surprising to see hybrid approaches, HL and H, be the best candidates in terms of performance. By exploiting different implementation strategies for different parts of the graph, we can achieve better performance. The HL schedule utilizes the benefits of a single pass of NL with the low overhead of EL for the remaining passes whereas the H schedule relies upon EL to efficiently execute nodes with very high degrees and SGM to process the remaining edges without the need for synchronization.

We believe these scheduling insights will be useful when implementing other algorithms on scale-free graphs on the GPU.

8. ACKNOWLEDGEMENT

We gratefully acknowledge the support of NVIDIA Corporation for donations of equipment used in this research. We would also like to thank Andrew Lenharth for insightful discussions and suggestions.

9. REFERENCES

[1] B. A.-L. Barabási and E. Bonabeau. Scale-free networks. *Scientific American*, 2003.

[2] Y. Bengio. Speeding up stochastic gradient descent. In *NIPS workshop on Efficient Machine Learning*, 2007.

[3] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra. Matrix market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137, 1996.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111. ACM, 2008.

[5] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems 2012. December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1232–1240, 2012.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[7] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang. GPU asynchronous stochastic gradient descent to speed up neural network training. *CoRR*, abs/1312.6186, 2013.

[8] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[9] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM.

[10] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[11] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech DNNs. In *Proc. ICASSP*, 2014.

[12] D. Steinkrau, P. Y. Simard, and I. Buck. Using gpus for machine learning algorithms. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition*, ICDAR '05, pages 1115–1119, Washington, DC, USA, 2005. IEEE Computer Society.