Synchronization Trade-offs in GPU implementations of Graph Algorithms

Rashid Kaleem
rashid@cs.utexas.edu
University of Texas
Austin, Texas

Anand Venkat
anandv@cs.utah.edu
University of Utah
Salt Lake City, Utah

Sreepathi Pai
sreepai@ices.utexas.edu
University of Texas
Austin, Texas

Mary Hall
mhall@cs.utah.edu
University of Utah
Salt Lake City, Utah

Keshav Pingali
pingali@cs.utexas.edu
University of Texas
Austin, Texas

***Abstract*—Although there is an extensive literature on GPU implementations of graph algorithms, we do not yet have a clear understanding of how implementation choices impact performance. As a step towards this goal, we studied how the choice of synchronization mechanism affects the end-to-end performance of complex graph algorithms, using stochastic gradient descent (SGD) as an exemplar. We implemented seven synchronization strategies for this application and evaluated them on two GPU platforms, using both road networks and social network graphs as inputs. Our experiments showed that although none of the seven strategies dominates the rest, it is possible to use properties of the platform and input graph to predict the best strategy.**

I. Introduction

Over the past decade, many irregular graph algorithms have been implemented on GPUs. These include single-source shortest path (SSSP) [1] and all-pairs shortest paths algorithms [2], breadth-first search (BFS) [3], minimum spanning tree computation [4], [5], the MPM algorithm for max-flow computation [6], 0-CFA analysis [7] and Andersen-style points-to analysis [8].

When writing code for such algorithms, GPU programmers are faced with many implementation choices, but the performance implications of these choices are usually not obvious. Some implementation choices can be easily explored by changing compiler flags or modifying a few lines of code, but others lead to entirely different programs so exploring the space of possibilities may involve substantial programmer effort. The choice of synchronization strategy is an example. At a high level, programmers have a choice between coarse-grain, barrier-style synchronization or fine-grain synchronization using constructs like atomics or locks. To use barrier-style synchronization, the program must be executed in rounds and the tasks in each round must be independent; with fine-grain synchronization, there may not be a notion of rounds, and concurrently executing tasks may read and write the same locations provided these memory accesses are properly synchronized. Not only are the resulting programs very different but each synchronization style can itself be implemented in many ways, as we show in this paper. Furthermore, the performance of irregular graph programs can be very dependent on the structure of the input graph: a program that performs well for power-law graphs may perform poorly for high-diameter graphs like road networks. Given all these complications, programmers would obviously benefit from guidelines that would help them make the right implementation choices.

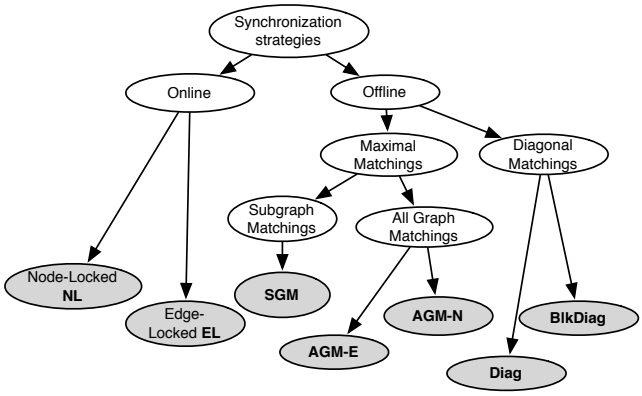


Figure 1: Taxonomy of scheduling strategies.

As a step towards this goal, we consider the problem of implementing synchronization for graph algorithms, using non-negative matrix factorization (NMF) [9] as an exemplar. NMF is used to solve problems such as product recommendation and object recognition [10]. In Section II, we describe a particular approach for solving NMF called *stochastic gradient descent* (SGD), which is an important general optimization method in machine learning. Figure 1 shows a taxonomy of scheduling strategies for implementing SGD on GPUs.

In Section III, we describe *offline* techniques, which preprocess the input to find independent tasks before executing the program, and generate code in which all synchronization is barrier synchronization. Some of these techniques compute *maximal matchings* in the graph to minimize the number of barrier synchronizations. We also explore a class of preprocessing techniques called *diagonal matchings*, which have lower preprocessing time but may require more barrier synchronization.

In Section IV, we describe two *online* schedules that we call *Edge-locked* (EL) and *Node-locked* (NL) implementa-

tions, which use fine-grain synchronization to coordinate the parallel tasks.

In Section V, we evaluate the performance of these implementations of SGD on two platforms, an NVIDIA Tesla K40C and an AMD Hawaii Radeon R9-290X for both power-law graphs and road networks. Key insights from this study include the following.

- Conventional wisdom is that fine-grain synchronization is expensive on GPUs. Therefore, we expected that ignoring preprocessing time, the offline implementations, which use barrier synchronization, would perform much better than the online ones. To our surprise, we found that this was true only on the AMD GPU; on the Tesla, online schedules perform better for all types of graphs. We provide a simple performance model to explain these results.
- On the Tesla, neither online implementation dominates the other one. For power-law graphs, EL performs better but for road networks, NL performs better. We provide an explanation for this result. This result motivated us to design a hybrid online scheme that performs better than both EL and NL.
- On the AMD GPU, the choice of offline strategy depends on the weight given to preprocessing time. If preprocessing time can be ignored, the implementation based on maximal matching gives better performance. If not, the diagonal matchings implementation gives better end-to-end performance.

Related work is described in Section VI.

II. Background

Recommendation systems [11] solve problems like the Netflix challenge problem which can be described abstractly as follows: given a set of users U, a set of movies M, and an incomplete database of movie ratings by users, predict how users will rate movies they have not yet rated.

One way to solve this problem is through non-negative matrix factorization, which is a kind of low-rank approximation. The database of ratings is represented as a sparse matrix R in which the rows represent users and the columns represent movies. Low-rank approximation finds two low-rank dense matrices W and H such that $R \approx W * H$ as shown in Figure 2. That is, each non-zero entry in R must be roughly equal to the corresponding entry in $W * H$; the remaining entries in $W * H$ are the predictions for the missing ratings.

Low-rank approximation can be formulated as a graph problem. The database of ratings R is represented as a bipartite graph between users and items; if user u assigned a rating r to a movie m, there is an edge (u, m) in the graph with weight r. The matrices W and H are represented by unknown vectors of length t associated with the nodes representing users and movies respectively, as shown in Figure 3 (these are known as *feature vectors*). The problem is to find values for these vectors such that for every edge (u, m) with weight r, the inner-product of the vectors on nodes u and m is roughly equal to r.

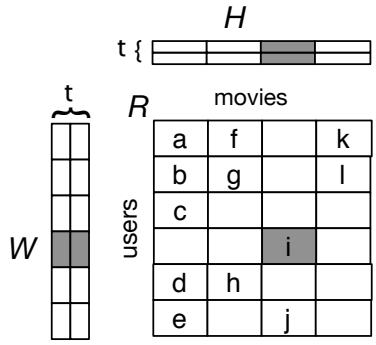


Figure 2: Low-rank approximation of a sparse matrix R by low rank matrices $W : |U| \times t$ and $H : t \times |M|$. Usually $t \approx 16$.

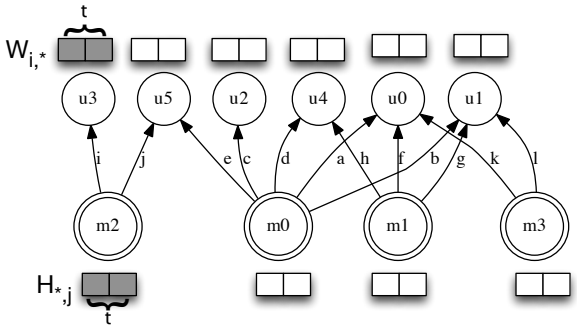


Figure 3: Sample bipartite graph between 6 users and 4 movies. Edge labels indicate ratings.

SGD is an iterative algorithm that computes feature vectors by making a number of sweeps over the bipartite graph. The vectors are initialized to some arbitrary values. In each sweep, all edges (u, m) are visited. If the inner-product of the vectors on nodes u and m is not equal to the weight on edge (u, m), the difference is used to update the two feature vectors. Sweeps are terminated when some heuristic measure of convergence is reached.

Parallelism can be exploited in each sweep by processing edges in parallel. Two edges can be processed in parallel provided they do not share a node; otherwise, they are said to *conflict* and must be processed serially. In our example, edges a and b conflict because they share the same movie m_0; similarly, edges a and f conflict because they share the same user u_0. Thus, the programmer needs to synchronize accesses to edges to avoid processing conflicting edges concurrently. The rest of this paper explores the performance implications of different ways of implementing this synchronization.

III. Offline Schedules

In a given graph, a set of edges is said to constitute a *matching* if no two edges in that set have a node in common [12]. Matchings are useful for parallel SGD computation because the edges in a matching can be processed in parallel without the need for synchronization. A *maximal matching* is a matching m such that every edge not in m conflicts with some edge in m.

Offline schedules preprocess the graph by partitioning its edges into a set of matchings. The SGD computation is then implemented as a series of supersteps separated by barriers; in each superstep, the edges in one matching are processed in parallel without synchronization. In Section III-A, we describe *maximal-matching schedules* which partition the edges of the graph into a sequence of maximal matchings.

The second approach relies on the structural properties of the bipartite graph. If the graph is viewed as an adjacency matrix, entries along the diagonals of the matrix can be processed concurrently as they do not share any end-points. This observation allows us to utilize sparse linear algebra frameworks such as CUDA-CHiLL [13] to synthesize scheduling routines for graph applications such as SGD. These *diagonal-matching schedules* are described in Section III-B.

To illustrate the schedules, we use the graph of Figure 3 and a hypothetical GPU with two threads. Our actual implementations run on an NVIDIA Tesla K40 and AMD R9-290X, as explained in Section V, so the two-thread hypothetical GPU is used only for illustration.

A. Maximal matchings schedules

The first category of schedules rely on maximal matchings. Algorithm 1 shows the algorithm to construct a maximal-matching schedule. To build a conflict free schedule, *i)* a maximal matching m is constructed from the graph; *ii)* the edges belonging to the maximal matching are removed from the graph; and, *iii)* the process repeated until there are no edges left in the graph. We refer to this set of maximal matching as a *matchings-set* M. The number of matchings in M is greater than or equal to the max-degree of the graph D_{max} since all edges of that node must be processed in separate matchings. Figure 4(a) shows the matchings-set M for the sample graph.

Algorithm 1 Algorithm for constructing a maximal-matching schedule. Given a graph g, returns the set of matchings M.

```
M = φ
while edges(g) > 0 do
    m = maximal_matching(g)
    g = g \ m
    M = M ∪ {m}
end while
return M
```

Given a matchings-set M, we describe three different strategies for scheduling edges within a set $m \in M$.

1) All-Graph Matching-Edge schedule (AGM-E): In an AGM-E schedule, matchings are processed one at a time. Each thread grabs an edge, load the labels at the end-points of that edge, performs the SGD computations, and updates those labels. This process is repeated until there are no more edges left to be processed in that matching. Note that AGM-E *makes no attempt to schedule edges connected to the same node on the same thread.*

For our sample graph, an edge schedule of this sort is shown in Figure 4(b). In our model GPU, we can execute only two edges per step so the processing of the first matching takes two steps, and a sweep over all edges takes eight steps.

2) All-Graph Matching-Node schedule (AGM-N): Unlike the AGM-E schedules, these schedules attempt to exploit locality in processing edges and utilize the local shared memory of the GPU to store the data associated with the nodes.

In our implementation, edges connected to a given movie node are all processed by the same thread. This is accomplished by processing movie nodes in blocks of T nodes, where T is the number of threads (the last block may have fewer nodes). Consider Figure 4(c), which shows a matrix in which the rows are the matchings and the columns are the movie nodes. Conceptually, we divide the columns of this matrix into blocks of T nodes, and process these blocks sequentially. Since we have two threads in our example, m_0 and m_1 are in the first block, and m_2 and m_3 are in the second block. When processing a given block of nodes, we iterate over all matchings in sequence, processing the appropriate edges as shown in Figure 4(d).

Each block column is processed by making a kernel call. Before a block column of movie nodes is processed, the associated movie node data is read into shared-memory. Global inter-thread block synchronization is used to separate the processing of edges from different matchings. After the processing is complete, the movie node data is written back into memory.

3) Sub-Graph Matching (SGM): This strategy can be viewed as a refinement of AGM-N. For large graphs, the number of nodes will be more than the number of threads T. In that case, computing a matchings-set for the entire graph and then repackaging it for the AGM-N schedule can be inefficient. In Figure 4(d), it takes four steps to execute the second block of nodes consisting of $\{m_2, m_3\}$ even though edges i and j can be processed in parallel with edges k and l respectively. Intuitively, if the number of threads is smaller than the number of nodes, the nodes will be processed in blocks, so matchings should be computed only for nodes in the same block.

This is accomplished by the SGM scheduling strategy. SGM first sorts the nodes in decreasing order of node degree.

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 4: Maximal matchings 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.

Then it partitions the nodes into blocks of size T. For each block, the matchings-set is computed, and edges are scheduled for that matchings-set as in AGM-N. The sub-graph matchings for the sample graph is shown in Figure 4(e), and the SGM schedule is shown in Figure 4(f).

The preprocessing time for SGM is different from the preprocessing time for the all-graph matching variants. When building all-graph matching, a single matchings-set is built for the entire graph. However, for SGM, we build the matchings-set for each block of nodes separately.

B. Diagonal matchings schedules

The schedules discussed in Section III-A are based on maximal matchings. To reduce the preprocessing overhead, schedules can be constructed using matchings that are not necessarily maximal.

One way to construct matchings cheaply is to exploit the matrix representation of the graph [14]. In the matrix representation, edges along a diagonal do not share any nodes and can be processed concurrently. Different diagonals must be serialized, however.

Diagonal matchings schedules can be advantageous as they facilitate temporal reuse of the nodes, but the benefits must outweigh the overhead of the barrier synchronization between diagonals. We increase the granularity of work within a diagonal, and therefore reduce the frequency of barrier synchronization, using two diagonal variants: (1) Diag (Section III-B1) and, (2) BlkDiag (Section III-B2).

1) Diagonal (Diag) schedule: Diag also exploits the parallelism within a single edge by processing the update to the feature vector in parallel. This ordering also achieves global memory coalescing for accesses to the feature vector across threads. We launch a 2-D grid of threads of dimension F by E, where F is the size of feature vector (e.g., 16 floats), and E is the number of edges to be processed in a kernel call.

D	Diagonal
d_0	(k)
d_1	$(-, l)$
d_2	$(f, -, -)$
d_3	$(a, g, -, -)$
d_4	$(b, -, i, -)$
d_5	$(c, -, -, -)$
d_6	$(-, h, j)$
d_7	$(d, -)$
d_8	(e)

(a) Diagonals.

	m0	m1	m2	m3
u0	a	f		k
u1	b	g		l
u2	c			
u3			i	
u4	d	h		
u5	e		j	

(b) 2×2 block diagonal schedule.

Figure 5: Diagonal matchings schedules for the sample input.

A thread (i, j) processes the i^{th} component of the feature vectors of the end points for edge j.

The maximum number of diagonals is $|M| + |U| - 1$ and the maximum width of a diagonal will be the number of columns. In our example, 4 movies (columns) and 6 users (rows) result in $4 + 6 - 1 = 9$ diagonals with the longest diagonal containing 4 entries. The complete list of diagonals, starting from the top is given in Figure 5(a).

2) Block-Diagonal (BlkDiag) schedule: The BlkDiag schedule reduces the size of the matrix by blocking along both dimensions. This reduced matrix has a reduced number of diagonals – if the movies are blocked by a factor R, and the users by a factor C, then the total number of diagonals in the BlkDiag schedule is $|M|/R + |U|/C - 1$.

A diagonal schedule obtained after 2×2 blocking is shown in Figure 5(b). There are now only 4 diagonals with each block consisting of at most 4 edges. Our implementation assigns each block to a thread. Within a block, the same set of movies and users are used repeatedly and the feature vectors corresponding to those rows and columns are cached

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.

in registers or GPU shared memory.

Comparison to Matchings: The diagonal matchings schedules are relatively easy to compute; the diagonal is determined by the difference in the row and column indices for an entry. It is, however, conservative because it will schedule entries concurrently *only* if they are along the same diagonal. In contrast, the maximal matchings schedules are more liberal but also costly to compute. Since the maximal matching does not constrain itself to any diagonal, it can discover more entries to schedule concurrently. For instance, in our example d_3 contains (a, g). We can also schedule either of i or j with these entries since they do not have any end point in common. The diagonal matchings schedule will not schedule i or j with d_3 since neither is along the diagonal d_3.

IV. Online Schedules

Online schedules assign edges to threads without attempting to avoid conflicts. Therefore, synchronization primitives such as atomics must be used to ensure mutual exclusion.

We describe two strategies. The *Edge-locked* strategy EL, described in Section IV-A, assigns edges to threads. The *Node-locked* strategy NL, described in Section IV-B, assigns nodes to threads.

A. Edge-locked (EL)

In each SGD sweep, threads make a number of passes over the set of edges until all edges have been processed. To process an edge, the thread attempts to acquire locks on its two nodes, and updates node labels if lock acquisition succeeds. Otherwise, the edge is deferred and retried in the next pass.

One possible schedule for the edges of Figure 3 is shown in Figure 6(a). We assume that the edges in the graph are stored in alphabetical order, which is similar to a CSR representation of Figure 3. Since our hypothetical GPU can execute two tasks at once, it will pick chunks of two edges from the work-list and try to process them. The first two edges are $\{a, b\}$. Since they share the same source m_0, only one thread succeeds in acquiring the lock, and edge b is delayed to the next pass. The next chunk to be executed is $\{c, d\}$, and only one edge gets processed while the other is moved to the next pass, and so on. The second pass, which starts at step 6, processes edges $\{b, d, h, j, l\}$ which could not be processed in the first pass.

The main problem with this strategy is that if edges connected to the same movie are tried concurrently, only one of the threads will make progress. The original ordering of the edges was derived from the CSR layout of the graph, which stores the edges of a given node in adjacent memory locations. This introduces a large number of conflicts, particularly for high-degree nodes.

To ameliorate this problem, we can shuffle edges randomly[1] before assigning them to threads. This lowers the likelihood that edges sharing the same movie are scheduled concurrently. 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 Figure 6(b). By mixing the edges of m_0 with edges from other nodes, we reduce the likelihood of conflicts. Experiments on actual input graphs confirm that shuffling can improve performance significantly.

For EL, preprocessing time involves the shuffling of edges to reduce the conflicts as described above. The execution time includes the time to perform kernel calls and the determination of whether all edges have been processed.

B. Node-locked (NL)

The Node-locked (NL) scheduling strategy assigns movie *nodes* to threads. This has two benefits. First, there is no need to acquire locks on the source node (i.e. movie) since a node is assigned to a single thread. 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.

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

[1]Our implementation uses the *std::random_shuffle* call to shuffle the edges.

Table I: Specifications of the platforms used for evaluation.

	Host	Device
K40	Scientific Linux 6.6, Kernel 2.6.32 on Intel Xeon E5-2609 with 32G RAM	Tesla K40c with 12GB VRAM
R9-290X	Ubuntu 14.04, Kernel 3.16.0 on Intel i7-3770K with 8GB RAM	AMD R9-290X with 8GB VRAM

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

There is no preprocessing required for NL since the graph representation allows threads to traverse neighbors of each movie/user directly. The execution time includes the time to invoke the kernels as well as polling the number of edges processed to check for completion.

V. Evaluation

Our evaluation examines the performance of the different synchronization schemes on two hardware platforms described in Table I. The online and maximal matching schedules are implemented[2] in OpenCL 1.2, the latest supported by NVIDIA The diagonal matchings schedules are generated via CUDA-CHiLL [13].

We use twelve input graphs in our experiments (Table II). Eight are scale-free networks which have a power-law degree distribution with the max-degree D_{max} shown in the table. These resemble real-life inputs to recommendation systems. To study the effect of input graph structure on performance, we also evaluate four road networks with relatively uniform degree distribution. The column labeled $EL(s)$ in Table II shows the running times of the EL version of SGD for each combination of input and platform. In the rest of this section, the running times of all other versions of SGD are normalized with respect to the running time of the EL version for that combination of input and platform.

A. Overall performance

Since fine-grain synchronization on GPUs is believed to be expensive compared to barrier synchronization, we expected the offline implementations to perform better on both platforms (ignoring preprocessing costs).

[2]Source code is available from http://iss.ices.utexas.edu.

Table II: Characteristics of the scale-free and uniform inputs. $|V|$ is the total number of vertices in the graph, $|E|$ is the number of edges in the graph, and D_{max} represents the maximum degree of any node in the graph. $EL(s)$ is the running time of the EL versions in seconds.

	$\|V\|$	$\|E\|$	D_{max}	$EL(s)$	
				K40	R9-290X
Scale-free					
STACK	$0.6M$	$0.1M$	6119	0.04	0.18
IMDB	$1.3M$	$3.7M$	1590	0.07	0.38
WIKI	$0.1M$	$5.0M$	100022	0.39	1.04
BGG	$0.1M$	$6.0M$	43331	0.22	0.53
CITP	$7.5M$	$16.5M$	779	0.32	1.69
POKEC	$3.2M$	$22.3M$	14734	0.42	2.3
LIVEJ	$9.6M$	$68.9M$	20293	1.5	7.2
NFLIX	$0.4M$	$99.0M$	227715	2.13	5.28
Road					
CAL	$3.7M$	$4.6M$	7	0.08	0.49
E	$7.1M$	$8.7M$	9	0.19	0.91
W	$12.5M$	$15.1M$	9	0.37	1.58
CTR	$28.1M$	$33.8M$	8	0.9	3.54

Figure 7 presents the running times of the different SGD implementations (ignoring preprocessing costs), normalized to the running time of the EL version.

The first important point to note is that on the K40, the online implementations are best for both scale-free and road networks, even if we ignore the preprocessing cost for the offline implementations.

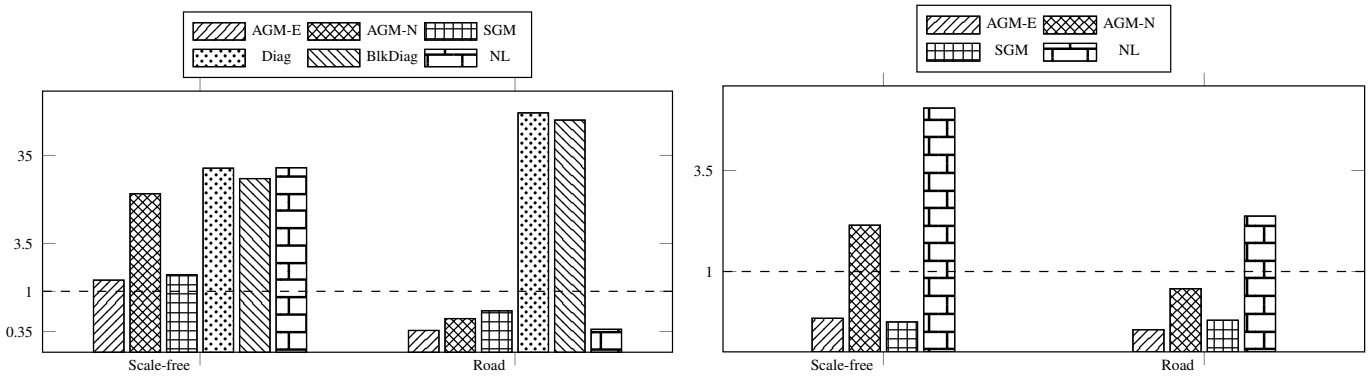
Figure 7 shows that on the K40, the best online implementation is $1.3\times$ and $2\times$ faster for power-law graphs and road networks respectively than the best offline implementation. In contrast, on the R9-290X, the maximal matching schedule is nearly twice as fast as the best online schedule, a result that is more in tune with conventional wisdom.

To investigate this further, we measured the throughput of atomic operations on both GPUs [15]. Figure 9 shows that for atomic writes to the same location (*i.e.*, atomics with the slowest throughput), the NVIDIA K40 achieves a throughput of roughly 600M atomics/s (nearly 1 atomic a clock) whereas the AMD R9-290X languishes far behind at 45M atomics/s.

This explains why the online implementations perform poorly on the AMD GPU: the NL and EL versions have to do at least one and two atomics per edge respectively, and atomics are relatively slow on this GPU. In contrast, the offline implementations execute a variable, but considerably fewer, number of atomics to implement the device-side barrier synchronization.

Nevertheless, the fact that atomics are relatively fast on the K40 does not explain why the EL version performs so much better than the offline ones for power-law graphs *even though it performs much more fine-grain synchronization*.

The explanation for this counterintuitive behavior is the



(a) NVIDIA K40 (b) AMD R9-290X

Figure 7: Geomean normalized runtime of scheduling schemes over the two input classes evaluated over two platforms. The runtimes are normalized to EL's runtime. Lower is better.

following. Since offline schedules are based on matchings, they can process at most one edge connected to a given node between successive barriers. Let d_m be the number of edges connected to the highest degree node N in the graph. An offline schedule must have at least d_m matchings, so if p is the average time for processing an edge, the execution time of the program is at least $d_m(p+b)$ where b is the cost of a barrier.

In an online schedule on the other hand, it is possible for several edges connected to node N to be processed between successive barriers due to optimistic concurrency. If on the average, a fraction f of edges connected to N are processed in each step and the cost of fine-grain synchronization to process one edge is l, the time to process all the edges connected to N is at most $(d_m f(p+l)+b)/f$ since it will take $1/f$ steps to process all the edges connected to N. This can be simplified to $d_m(p+l)+(b/f)$. The first term is the cost to process the edges, and the second term is the cost of barrier synchronization.

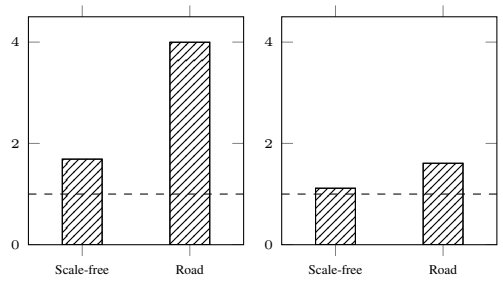
Therefore the relative costs are:

$$d_m p + b(d_m) \quad \text{vs.} \quad d_m(p+l) + b(1/f)$$

If l, the cost of fine-grain synchronization, is very high, the reduction in barrier synchronization may not pay off, as on the R9-290X. However, if fine-grain synchronization is not very expensive and the online schedule can process multiple edges from high-degree nodes in each step, the total cost of barrier synchronization is lowered substantially, and the online schedule wins like on the K40.

Offline implementations: Ignoring preprocessing time, the diagonal-based schemes are slower than the maximal matching schemes on the K40[3]. This is expected because the diagonal schedules process fewer edges between successive barriers, so they also execute more barrier operations.

[3]As our compiler produces CUDA code, we were unable to run these on the AMD GPU.



(a) K40 (b) R9-290X

Figure 8: Speedup of hybrid schedule over EL on the different GPUs across the two input classes. Higher is better.

However, if preprocessing time is taken into account, the diagonal schedules are faster than the matching-based versions for scale-free graphs. Thus, when processing scale-free graphs once (or if the graph structure changes), diagonal schedules should be preferred over the offline matching-based versions.

B. Hybrid schedules

Figure 7 also reveals that on the K40, while the online schemes outperform the maximal matchings schemes, the best-performing schedule varies by input class. EL performs better on scale-free inputs while NL suffers from load imbalance as it processes edges of a high-degree node serially which outweighs the benefit from shared memory reuse. However, NL performs better on road networks as it is able to better utilize the locality by scheduling nodes to threads. Since the degree of nodes in a road network is uniform and small, the overhead of scanning the edge-list of each node on each pass is small. We could choose between EL and NL using input characteristics by using a framework such as Nitro [16]. Alternatively, a hybrid schedule could be used.

We investigated such a hybrid online schedule which

runs NL as the first pass and processes all the remaining edges with EL schedule. NL processes most of the edges while EL processes the remaining edges. NL also exploits shared memory. This combination of schedules produces better performance on both the scale-free networks as well as road networks compared to a single online schedule as shown in Figure 8.

We also investigated if combining an online scheme with an offline scheme could improve overall performance. Essentially, we observed that the performance of maximal matchings schedule is limited by the highest degree nodes – the edges of these nodes must occur in different matchings and hence the highest degree degree node determines the length of the critical path. Therefore, we built a hybrid schedule which processes a set of high-degree nodes using an EL schedule and the remaining nodes using SGM. Unfortunately, while this improves the performance of SGM, the performance of EL is severely affected, since the high-degree nodes exhibit a large number of conflicts amongst themselves.

C. Offline schedules

On scale-free inputs, AGM-E performs best amongst the maximal matching schemes as it mimics EL without the overhead of locks. AGM-N suffers the most from the high degree nodes in a scale-free graph as all the threads have to go through at least $|M| \geq d_m$ time steps. This overhead is avoided by SGM, which produces better matching based on the number of hardware threads. Road network graphs which have uniformly low degree nodes allow AGM-E, AGM-N and SGM to outperform EL.

The sparsity of the inputs affects the performance of the diagonal matchings schemes. The matching schedules greedily pack as many edges into a matching set producing a smaller number of matchings compared to a diagonal schedule which produces a matching for every non-empty diagonal.

VI. Related work

We build upon earlier work that described efficient SGD implementations on NVIDIA GPUs [17]. Based on that preliminary study, we expanded our investigation to the performance of synchronization constructs available on multiple GPU hardware platforms to derive general lessons that are applicable to other graph-based algorithms which use fine-grained synchronization.

Recent work [18] has investigated performance of applications that use GPU atomic constructs which write to one location. Our work studies applications that use locking constructs whose atomics write to different locations and hence the techniques they recommend to improve performance are not applicable.

Application specific scheduling strategies on the GPU have been explored extensively and led to highly efficient

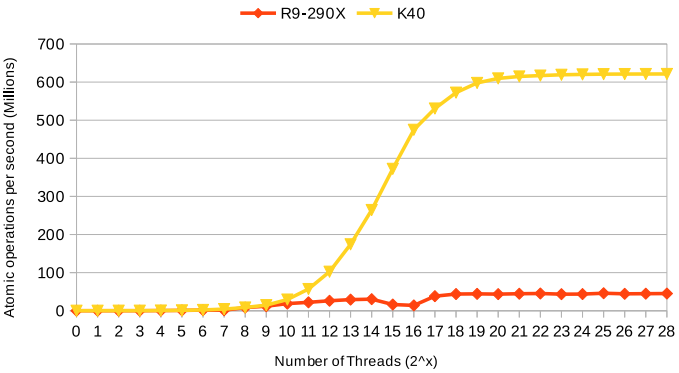


Figure 9: Atomic throughput of the NVIDIA K40 and AMD R9-290X. X-axis shows the number of threads launched, and Y-axis shows the throughput of atomics operations *achieved*. The K40 peaks at about 600M atomic operations per second, while the R9-290X saturates at about 45M.

implementation of irregular applications[19], [1], [20]. This paper aims to explore the different scheduling strategies for a broader class of problems, topology driven algorithms, of which SGD is one instance.

GPUs have accelerated machine learning in many applications [21], [22], [23], however SGD is not commonly parallelized. GPU A-SGD [24] exploits both *model* and *data* parallelism to speed up neural network training for computer vision. Dean *et al.* [25] compare the performance of a GPU implementation for speech model training and observed a large overhead compared to a CPU cluster. Collaborative filtering and support vector machines have also been implemented on the GPU [26] and shown to perform comparable to CPU implementations.

The diagonal scheduling schemes Diag and BlkDiag are based on the DIA format used commonly in standard discretization of partial differential equations or the application of stencil operators to grids. The DIA format has been parallelized on a GPU [27] in the context of the SpMV kernel. However, in SpMV, the diagonals can be processed concurrently while SGD must process diagonals serially. The block diagonal format has also been used in stencil based solvers for partial differential equations [28]. The derivation of the blocks containing non-zero entries in BlkDiag is similar to the Block-CSR format in OSKI [29], where the column and row indices of each entry of the matrix are traversed to identify the corresponding block position of the non-zero entry.

VII. Conclusion

In this paper, we studied the impact of the choice of synchronization strategy on the performance of SGD, a widely used kernel in machine learning. It is a step towards the ultimate goal of providing guidelines to GPU programmers for making implementation choices when coding irregular graph programs. We implemented seven synchronization

strategies for this application and evaluated them on two GPU platforms using both road networks and social network graphs as input. The synchronization strategies can be classified as offline strategies and online strategies. Offline strategies preprocess the graph to find independent tasks that can be run in parallel with barrier synchronization. Online strategies do not require preprocessing and use fine-grain synchronization to ensure that tasks execute atomically.

Although conventional wisdom tells us that online strategies are not competitive because of the cost of fine-grain synchronization on GPUs, we found that this was true only on one of the GPUs in our study. On the other GPU, the cost of synchronization was small enough that online schedules could be competitive, and in fact they outperformed offline schedules, particularly for power-law graphs. Furthermore, our results showed that power-law and road networks required different online schedules because of an interaction between load-balancing and locality. This motivated us to invent a hybrid online schedule that dominated the other schedules.

Even on devices with slow atomics, the exact choice of offline schedule is not clearcut. For computations that involve scale-free graphs, customizing the lock-free schedule to the device, as we do with the SGM strategy, to better utilize the hardware, can improve performance significantly.

Performance programming of GPUs is currently an art. We believe that studies like the one in this paper are needed to make it into a science.

VIII. Acknowledgement

The work presented in this paper has been supported by the National Science Foundation grants CNS 1111766, CNS 1302663, CCF 1218568, XPS 1337281, CNS 1406355, DARPA BRASS contract FA8750-16-2-0004, and DARPA grant FA8650-15-C-7563. We gratefully acknowledge the support of NVIDIA Corporation for donations of equipment used in this research.

References

[1] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 349–359.

[2] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *HiPC'07: Proceedings of the 14th international conference on High performance computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208.

[3] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128.

[4] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the gpu," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 167–171.

[5] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2012, pp. 205–214.

[6] S. Solomon, P. Thulasiraman, and R. K. Thulasiram, "Exploiting parallelism in iterative irregular maxflow computations on gpu accelerators," in *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, ser. HPCC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 297–304.

[7] T. Prabhu, S. Ramalingam, M. Might, and M. Hall, "Eigencfa: accelerating flow analysis with gpus," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 511–522.

[8] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU implementation of inclusion-based points-to analysis," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2012, pp. 107–116.

[9] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, 2001, pp. 556–562.

[10] ——, "Learning the parts of objects by non-negative matrix factorization," vol. 401, no. 6755. Nature Publishing Group, 1999, pp. 788–791.

[11] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE Trans. on Knowl. and Data Eng.*, vol. 17, no. 6, pp. 734–749, Jun. 2005.

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

[13] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 136–150.

[14] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 521–532.

[15] M. Elteir, H. Lin, and W.-C. Feng, "Performance characterization and optimization of atomic operations on amd gpus," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, ser. CLUSTER '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 234–243.

[16] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 501–512.

[17] R. Kaleem, S. Pai, and K. Pingali, "Stochastic gradient descent on gpus," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-8. New York, NY, USA: ACM, 2015, pp. 81–89.

[18] I. J. Egielski, J. Huang, and E. Z. Zhang, "Massive atomics for massive parallelism on gpus," in *Proceedings of the 2014 International Symposium on Memory Management*, ser. ISMM '14. New York, NY, USA: ACM, 2014, pp. 93–103.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128.

[20] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A gpu implementation of inclusion-based points-to analysis," *SIGPLAN Not.*, vol. 47, no. 8, pp. 107–116, Feb. 2012.

[21] 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*, ser. ICML '09. ACM, 2009.

[22] 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*. ACM, 2008.

[23] 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*. IEEE Computer Society, 2005.

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

[25] 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*, 2012.

[26] D. Zastrau and S. Edelkamp, "Stochastic gradient descent with gpgpu," in *Proceedings of the 35th Annual German Conference on Advances in Artificial Intelligence*, ser. KI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–204.

[27] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of SC '09*, Nov 2009.

[28] D. Lowell, J. Godwin, J. Holewinski, D. Karthik, C. Choudary, A. Mametjanov, B. Norris, G. Sabin, P. Sadayappan, and J. Sarich, "Stencil-aware GPU optimization of iterative solvers," *SIAM J. Scientific Computing*, 2013.

[29] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, pp. 521–530, 2005.