UC Davis
UC Davis Previously Published Works

Title
WTF, GPU! Computing Twitter's Who-To-Follow on the GPU

Permalink
https://escholarship.org/uc/item/5xq3q8k0

Authors
Geil, Afton
Wang, Yangzihao
Owens, John D

Publication Date
2014-10-01

DOI
10.1145/2660460.2660481

Supplemental Material
https://escholarship.org/uc/item/5xq3q8k0#supplemental

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

WTF, GPU! Computing Twitter's Who-To-Follow on the GPU

Afton Geil, Yangzihao Wang, and John D. Owens
University of California, Davis

Note: This is a revision to the ACM Digital Library version of the paper (which can be found under Supporting Material). This version includes a performance comparison against a Cassovary CPU library implementation of WTF not included in the original paper.

ABSTRACT

In this paper, we investigate the potential of GPUs for performing link structure analysis of social graphs. Specifically, we implement Twitter's WTF ("Who to Follow") recommendation system on a single GPU. Our implementation shows promising results on moderate-sized social graphs. It can return the top-K relevant users for a single user in 172 ms when running on a subset of the 2009 Twitter follow graph with 16 million users and 85 million social relations. For our largest dataset, which contains 75% of the users (30 million) and 50% of the social relations (680 million) of the complete follow graph, this calculation takes 1.0 s. We also propose possible solutions to apply our system to follow graphs of larger sizes that do not fit into the on-board memory of a single GPU.

Categories and Subject Descriptors

H.4 [**Information Systems**]: World Wide Web—*social networks*; E.1 [**Data**]: Data Structures—*graphs and networks*; D.1.3 [**Software**]: Programming Techniques: Concurrent Programming—*parallel programming.*

Keywords

Online Social Networks; GPU Computing; Recommendation Systems; Graph Processing; Twitter

1. INTRODUCTION

Many web service providers use recommendation systems to sell products or to increase the value of their service. Shopping services like Amazon suggest products users may be interested in buying, news sites recommend articles based on a user's reading history, and streaming services like Netflix recommend movies and television shows to watch. Social networking services recommend people that the current user may want to connect with. In a social network, where the content is entirely user-generated, a good recommendation system is key in retaining and engaging users.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
COSN'14, October 1–2, 2014, Dublin, Ireland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3198-2/14/10 ...$15.00.
http://dx.doi.org/10.1145/2660460.2660481.

Twitter is a social media service that allows users to broadcast short, 140-character *tweets* to all users who choose to *follow* them. Twitter's success depends on acquiring and maintaining an active user base. A user's satisfaction with the service depends almost entirely on the content generated by the other users in their social network. They need to subscribe to the tweets of users they are interested in reading updates from. Finding users to follow can be difficult, especially for new users, so Twitter provides them with recommendations of accounts they may be interested in subscribing to. This service is called "Who to Follow", or WTF. Twitter calculates these recommendations by analyzing the link structure of the follow graph [4]. Recommendations should be provided and updated in real time to keep up with changes in the follower graph. For a graph with hundreds of millions of nodes and billions of edges, this is no small problem. Graphics processing units (GPUs) have a highly parallel architecture that is potentially well-suited for this kind of large-scale graph analysis.

GPUs are throughput-oriented, highly parallel architectures that have proven to be very efficient for solving many large-scale problems with plenty of available parallelism. The modern GPU has become the most cost-effective way to perform massive amounts of computation. A single GPU can often outperform a cluster of CPUs; however, they have not yet been able to gain a foothold in data centers. This is due to various issues, such as programmability, limited memory size, and communication costs. GPUs have a very different architecture from CPUs. This makes GPUs efficient at solving large problems, but also creates unique programming challenges.

Solving graph problems on a GPU is particularly difficult because although these problems usually have a large amount of parallelism, that parallelism is irregular: in social graphs, nodes typically have widely varying connectivity and thus wildly varying work per node. As well, graph algorithms usually involve traversing the graph, including paths that split and join along the way; this complicates the control flow, because paths could intersect at any time. Nonetheless, some work has been done on implementing PageRank, a popular graph analysis algorithm, on GPUs [12,14]; however, this work does not confront the irregular challenges of graph traversal because they use the linear algebra formulation of the PageRank algorithm. GPUs can work well on native graph representations: Merrill et al. used GPUs to achieve 4x speedups over multicore CPUs with breadth-first search [9]. Our work leverages some of Merrill et al.'s traversal strategies, but maps a more complex graph algorithm to GPUs. Very

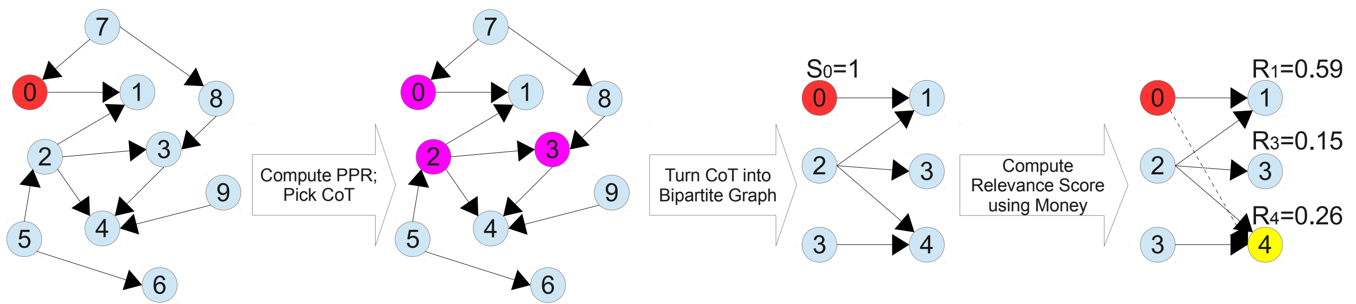

Figure 1: Overview of Twitter's WTF algorithm. *Frame 1:* The initial graph (red [dark] node is the user for whom recommendations are being computed). *Frame 2:* The Circle of Trust (nodes in pink [dark]) is found using Personalized PageRank. *Frame 3:* The graph is pruned to include only the CoT and the users they follow. *Frame 4:* The relevance scores of all users on the right side are computed with Twitter's Money algorithm. Node 4 will be suggested for Node 0 to follow because it has the highest value of all nodes the user is not already following.

little work has been done using GPUs for recommendation systems. Srinivasa et al. implemented the friends-of-friends algorithm on a GPU [13]. Friends-of-friends is a basic recommendation algorithm that takes a user and nodes adjacent to the user and returns the set of second level nodes that are adjacent to these nodes. Srinivasa et al. only tested their algorithm on small graphs (fewer than 40,000 nodes), but they did achieve a speedup of about 2x over a basic CPU implementation.

2. TWITTER'S WTF ALGORITHM

Two main stages comprise the WTF recommender. In the first stage, Twitter calculates a Personalized PageRank (PPR) for the user. PPR assigns a ranking to all nodes in the network based on how closely connected they are to the main node (the user interested in recommendations). This ranking is used to find the top 1000 ranking nodes. These nodes form the user's "Circle of Trust" (CoT), which consists of the 1000 nodes closest to the user. Pruning the graph in this way increases the personalization of the recommendation and reduces spam. Next, they create a bipartite graph with the CoT on one side, and the users followed by the CoT on the other. All other nodes are pruned from the graph. The final step is Twitter's "Money" algorithm, a graph analysis algorithm that determines which accounts the user is most likely to be interested in following. Figure 1 shows a schematic of the entire WTF algorithm.

2.1 Personalized PageRank

PageRank is an algorithm for ranking nodes in a graph based on the structure of the graph [11]. It was originally used on the web graph to rank webpages for search engines. The PageRank of a node can be thought of as the probability that a random walker traversing the graph along the edges will land on that node. It is a measure of how well-connected the node is, and therefore, how important it is to the graph. A Personalized PageRank (PPR) calculation relative to node A is identical to the normal PageRank calculation, except all random walks begin at node A, rather than a random node. Overall, a Personalized PageRank calculation for A shows which nodes are most closely related to A.

PageRank can be calculated using Monte Carlo methods or power iteration. A Monte Carlo method for Personalized PageRank would be to actually perform many random walks on the graph and maintain a count of the number of times each node is visited, then use these counts to estimate the stationary distribution. Power iteration methods formulate the problem as a system of linear equations and use linear algebra techniques to solve for the ranking values. Twitter chose a Monte Carlo method for their PPR, while we chose a power iteration method for our implementation. We discuss the reasoning for this decision in Section 6.

2.2 Money

Twitter's Money algorithm [3] is similar to a combination of Kleinberg's HITS algorithm [5] and Lempel and Moran's SALSA [7]. First, the graph is transformed into a bipartite graph, with the CoT on the left side, and the users the CoT follows on the right side, as shown in the second and third stages of Figure 1. If a user is both in the CoT and followed by someone in the CoT, this node will appear on both sides of the bipartite graph. The CoT nodes are assigned a *similarity* value, and the users they follow are assigned a *relevance* value. Initially, the user we are trying to get recommendations for, C, has their similarity score set to 1, and all others have their similarity or relevance scores set to 0. The Money algorithm distributes each CoT member's similarity score to the relevance scores of all the users they follow. The followers then distribute their relevance scores back across the graph to all of their followers. As in PPR, the Money algorithm can also be written as a system of linear equations. From the solution of this system of equations, Twitter finds the nodes with the highest relevance scores. These are the accounts they recommend to the user in the Who to Follow feature. The similarity scores are used for other features, such as the "similar to you" feature and targeted advertising.

3. PARALLELIZING GRAPH ALGORITHMS ON THE GPU

Both PPR and Money are link prediction algorithms that traverse the graph and assign rank values for a subset of nodes. Like many other graph algorithms, they can be viewed as iterative convergent processes. There is a data dependency between iterative steps, but within each step, a large number of independent edge-centric or vertex-centric operations can be parallelized. However, to fully exploit the

compute capabilities of GPUs, we need special strategies to handle irregular memory access and work distribution.

For compact and efficient memory access, we use the compressed sparse row (CSR) format to represent the follow graph on the GPU. It uses a column-indices array, C, to store a list of neighbor vertices and a row-offsets array, R, to store the offset of the neighbor list for each vertex. This representation enables us to use parallel primitives such as prefix sum to reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph processing.

3.1 Graph Primitives

To solve the issue of irregular work distribution, we design two graph primitives: *graph traversal* and *filtering*. Graph traversal starts with a queue of vertices we call the *frontier*. Traversal then takes several iterations to advance toward other vertices by visiting the neighbor list of all vertices in the current frontier in parallel and adding the neighbor vertices to the new frontier. Filtering starts with a frontier that contains either edges or vertices, does a user-defined validation test for every item in the frontier in parallel, then generates a new frontier containing only the items that have passed the validation test.

In PPR, graph traversal and filtering alternate until all the rank values converge and there are no vertices in the frontier for the graph traversal primitive. In Money, we use the graph traversal primitive to bounce back and forth between two disjoint sets in a bipartite graph: the CoT and the union of all CoT vertices' neighbor lists. Since in graph traversal primitives, the neighbor lists can vary greatly in size, traversing these neighbor lists in parallel efficiently and in a load-balanced way is critical for the performance of our system. Merrill et al. [9] uses specialized strategies according to neighbor list size in the context of a parallel breadth-first search (BFS) algorithm. The algorithm efficiently reduces the amount of overhead within each kernel and better utilizes the GPU. Our implementation extends this strategy with two important improvements. First, we add inline device functions to perform user-specific computations and to reuse graph traversal primitives for both PPR and Money by just replacing the function we load when visiting edges and vertices. Second, BFS's operations on vertices are idempotent, but ours are not, so we guarantee the correctness of both algorithms by adding atomics to resolve race conditions between edges converging on a vertex.

3.2 Application to Other Graph Algorithms

The graph traversal primitive distributes the parallel workload per edge or per node in a graph to tens of thousands of threads on the GPU to process in parallel, and the filtering primitive reorganizes the elements we want to process in parallel for the next iteration. By reusing these two primitives, we have implemented several graph-traversal-based ranking algorithms such as PageRank, PPR, Money, HITS, and SALSA with minimal programming cost. Several of these algorithms run on bipartite graphs, which are key abstractions in several ranking and link prediction algorithms. As far as we know, we are the first to target bipartite graph algorithms on the GPU.

We implement bipartite graphs in the following way. A directed bipartite graph is similar to a normal undirected graph, except in a directed bipartite graph, we need to consider the outgoing edges and the incoming edges of a vertex separately. We achieve this by reversing the source and destination vertex ID for each edge while constructing the CSR data structure to record the incoming edges and using an additional prefix sum pass to compute the incoming degree for each vertex. In our graph primitives, we also added a feature to switch between visiting the outgoing edges of a vertex and visiting the incoming edges of a vertex. We use this method in our SALSA implementations. In our Money and HITS implementation, we do not need to calculate the in-degree of every node in the graph, but just the ones connected to the CoT. We take advantage of this by finding the incoming degree values for neighbors of the CoT on the fly with an additional pass of the graph traversal primitive. Because of the small size of the CoT, this extra pass takes negligible time but saves us gigabytes of memory for large datasets.

4. WTF IMPLEMENTATION

In our implementation, we start by putting all the graph topology information on the GPU. First, we compute the PPR value for each vertex in the follow graph with a user-defined seed vertex. Then we radix-sort the PPR values and take the vertices with the top k PPR values (in our implementation, $k = 1000$) as the CoT and put them in a frontier. We then run the Money algorithm, sort the vertices that the CoT follows, and finally extract the vertices with the top relevance scores to use for our recommendation.

4.1 Personalized PageRank

Algorithm 1 shows our implementation of PPR using the graph primitives and functors we design. We update PPR using the following equation:

$$PPR(v_i) = \begin{cases} (1-\delta) + \delta \cdot \sum\limits_{v_j \in pred(v_i)} \frac{PPR(v_j)}{d_{\text{OUT}}(v_j)} & v_i \text{ is seed} \\ \delta \cdot \sum\limits_{v_j \in pred(v_i)} \frac{PPR(v_j)}{d_{\text{OUT}}(v_j)} & \text{otherwise} \end{cases}$$

where δ is a constant damping factor (typically set to 0.85), $d_{\text{OUT}}(v_j)$ is the number of outbound edges on vertex v_j, and N is the total number of vertices in the graph.

The PPR algorithm starts by initializing problem data (line 1). It assigns the initial rank value for each vertex as $\frac{1}{N}$, and puts all vertices in the initial frontier. The main loop of the algorithm contains two steps. For each iteration, we first use GraphTraversal to visit all the edges for each vertex in the frontier and distribute each vertex's PPR value to its neighbors (line 7). Then we use Filtering to update the PPR value for the vertices that still have unconverged PPR values. We give the seed vertex an extra value (line 10) as in the equation. Then we remove the vertices whose PPR values have converged from the frontier (line 12). The algorithm ends when all the PPR values converge and the frontier is empty.

4.2 Twitter's Money Algorithm

Algorithm 2 shows our implementation of Twitter's Money algorithm. We treat people in the CoT and people they follow as two disjoint sets X and Y in the bipartite graph $G = (X \cup Y, E)$ they form. For each node in X (the CoT), the Money algorithm computes its similarity score; for each node in Y, the Money algorithm computes its relevance score.

Algorithm 1 Personalized PageRank

```
1: procedure SET_PROBLEM_DATA(G, P, seed, delta)
2:     P.ranks_curr[1..G.verts] ← 1/N
3:     P.src ← seed, P.delta ← delta
4:     P.frontier.Insert(G.nodes)
5: end procedure
6: procedure DISTRIBUTEPPRVALUE(s_id, d_id, P)
7:     atomicAdd(P.rank_next[d_id], P.rank_curr[s_id] / P.out_degree[s_id])
8: end procedure
9: procedure UPDATEPPRVALUE(node_id, P)
10:    P.rank_next[node_id] ← (P.delta · P.rank_next[node_id]) + (P.src == node_id)?(1.0 − P.delta) : 0
11:    diff ← fabs(P.rank_next[node_id] − P.rank_curr[node_id])
12:    return diff > P.threshold
13: end procedure
14: procedure COMPUTE_PPR(G, P, seed, delta, max_iter)
15:    SET_PROBLEM_DATA(G, P, seed, delta)
16:    while P.frontier.Size() > 0 do
17:        GRAPHTRAVERSAL(G, P, DistributePPRValue)
18:        FILTERING(G, P, UpdatePPRValue)
19:        swap(P.rank_next, P.rank_curr)
20:    end while
21: end procedure
```

Algorithm 2 Twitter's Money Algorithm

```
1: procedure SET_PROBLEM_DATA(G, P, seed, alpha)
2:     P.relevance_curr[1..G.verts] ← 0
3:     P.sim_curr[1..G.verts] ← 0
4:     P.src ← seed, P.alpha ← alpha
5:     P.sim_curr[seed] ← 1
6:     P.frontier.Insert(G.cot_queue)
7: end procedure
8: procedure DISTRIBUTERELEVANCE(s_id, d_id, P)
9:     atomicAdd(P.relevance_next[d_id], P.sim_curr[s_id] / P.out_degree[s_id])
10: end procedure
11: procedure DISTRIBUTESIM(s_id, d_id, P)
12:    val ← (1 − P.alpha) · P.relevance_curr[d_id] / P.in_degree[d_id] + (P.src == s_id)?(P.alpha / P.out_degree[s_id]) : 0
13:    atomicAdd(P.sim_next[d_id], val)
14: end procedure
15: procedure COMPUTE_MONEY(G, P, seed, alpha)
16:    SET_PROBLEM_DATA(G, P, seed, alpha)
17:    for iter++ < 1/alpha do
18:        GRAPHTRAVERSAL(G, P, DistributeRelevance)
19:        swap(P.relevance_next, P.relevance_curr)
20:        GRAPHTRAVERSAL(G, P, DistributeSim)
21:        swap(P.sim_next, P.sim_curr)
22:    end for
23: end procedure
```

We use the following equations in our implementation:

$$sim(x) = \begin{cases} \alpha + (1-\alpha)\cdot \sum_{(x,y)\in E} \frac{relevance(y)}{d_{\mathrm{IN}}(y)} & x \text{ is seed} \\ (1-\alpha)\cdot \sum_{(x,y)\in E} \frac{relevance(y)}{d_{\mathrm{IN}}(y)} & \text{otherwise} \end{cases}$$

$$relevance(y) = \sum_{(x,y)\in E} \frac{sim(x)}{d_{\mathrm{OUT}}(x)}$$

In the Money algorithm, we also initialize problem data first (line 1). We set the similarity score for the seed vertex to 1, and set the similarity and relevance scores for all other vertices to 0. We then put all vertices in the CoT in the initial frontier. The main loop of the algorithm contains two steps of GraphTraversal, for each iteration we use the first GraphTraversal to visit all the edges for each vertex in the CoT and distribute each vertex's similarity score to its neighbors' relevance scores (line 9). After this step, we update the relevance scores so that they may be used in the computation of similarity scores (line 19). The second GraphTraversal will visit all the edges for each vertex in the CoT again, but this time it will distribute the neighbors' relevance scores back to each CoT vertex's similarity score (line 12). Again, we treat the seed vertex differently so that it can get more similarity score during each iteration, which in turn will affect its neighbors' relevance scores and other CoT vertices' similarity scores. After the second GraphTraversal step, we update the similarity scores (line 21). As Twitter does in their Money algorithm [3], we end our main loop after $1/\alpha$ iterations.

5. EXPERIMENTS

We ran all experiments in this paper on a Linux workstation with 2 × 2.53 GHz Intel 4-core E5630 Xeon CPUs, 12 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The parallel programs were compiled with NVIDIA's nvcc compiler (version 6.0.1) with the -O3 flag. The sequential programs were compiled using gcc 4.6.3 with the -O3 flag. The datasets used in our experiments are shown in Table 1, and Table 2 shows the runtimes of our GPU recommendation system on these datasets. Runtimes are for GPU computation only and do not include CPU-GPU transfer time; we discuss why in Section 6. The wiki-Vote dataset is a real social graph dataset that contains voting data for Wikipedia administrators; all other datasets are follow graphs from Twitter and Google Plus [6,8]. Twitter09 contains the complete Twitter follow graph as of 2009; we extract 75% of its user size and 50% of its social relation edge size to form a partial graph that can fit into the GPU's memory.

Table 1: Experimental Datasets

Dataset	Vertices	Edges
wiki-Vote	7.1k	103.7k
twitter-SNAP	81.3k	2.4M
gplus-SNAP	107.6k	30.5M
twitter09	30.7M	680M

5.1 Scalability

In order to test the scalability of our WTF-on-GPU recommendation system, we ran WTF on six differently-sized subsets of the twitter09 dataset. The results are shown in Figure 2. We see that the implementation scales sublinearly with increasing graph size. As we double the graph size, the total runtime increases by an average of 1.684x, and the runtime for Money increases by an average of 1.454x. The reason lies in our work-efficient parallel implementation. By doing per-vertex computation exactly once and visiting each

Table 2: Runtimes for Different Graph Sizes

Time (ms)	wiki-Vote	twitter	gplus	twitter09
PPR	0.45	0.84	4.74	832.69
CoT	0.54	1.28	2.11	51.61
Money	2.70	5.16	18.56	158.37
Total	4.37	8.36	26.57	1044.99

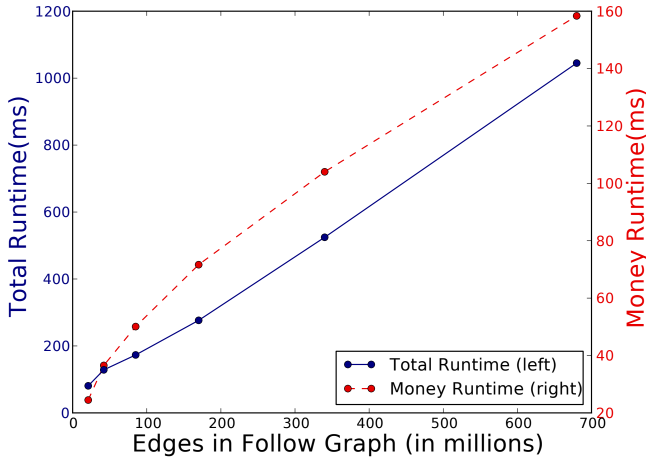

Figure 2: Scalability of runtime versus edge count for our GPU recommendation system.

edge exactly once, our parallel algorithm performs linear $O(m+n)$ work. The reason that we have better scalability for the Money algorithm is that although we are doubling the graph size each time, the CoT size is fixed at 1000. We address scalability beyond a single GPU in Section 6.

5.2 Comparison to Cassovary

We chose to use the Cassovary graph library for our CPU performance comparison. The results of this comparison are shown in Table 3. Cassovary is a graph library developed at Twitter. It was the library Twitter used in their first WTF implementation [4]. The parameters and Cassovary function calls for this implementation can be found in the Appendix.

We achieve speedups of up to 1000x over Cassovary for the Google Plus graph, and a speedup of 14x for the 2009 Twitter graph, which is the most representative dataset for the WTF application. One difference between the GPU algorithm and the Cassovary algorithm is that we used the SALSA function that comes with the Cassovary library, instead of using Twitter's Money algorithm for the final step of the algorithm. Both are ranking algorithms based on link analysis of bipartite graphs, and in the original Who-To-Follow paper [4], Gupta et al. use a form of SALSA for this step, so this is reasonable for a comparison.

6. DISCUSSION

On a graph with more than 175 million nodes and 20 billion edges, the WTF algorithm currently takes around 500 ms in Twitter's data center [10]. In contrast, our implementation can process a graph with 25 million nodes and 340 million edges in a similar amount of time (524 ms), and it takes 1 second to process our largest dataset, which is still significantly smaller than the complete Twitter graph.

The Personalized PageRank calculation takes up the vast majority of the runtime for larger graphs (Table 2). This is because PPR runs on the entire graph, but Money only runs on the pruned CoT graph, which does not grow as quickly. One possible way to reduce the runtime would be to precompute the CoT, and only run Money to update WTF. In this scheme, PPR would only be run periodically, and it could be an offline process. Alternatively, an incremental PPR calculation, as in Bahmani et al. [2], could provide an estimate of the new CoT without needing to iterate through the entire graph. With the precomputed CoT, we would only need to run Twitter's Money algorithm to get the result. According to our sublinear scalability model of Twitter's Money algorithm runtime, we could compute the result in 300 ms on Twitter's circa-2009 follow graph.

We also note that our PPR calculation uses a power iteration method, while Twitter's is a Monte Carlo method. Both methods are quite accurate, but there are a few trade-offs [1]. The power iteration method is deterministic. Every time the algorithm runs on the same set of data, we will get the same results. Monte Carlo methods involve actually performing random walks to compute the probability distribution, so the outcome will not be exactly the same every time. In terms of performance, Monte Carlo methods give a reasonable approximation after only the first iteration, but the error decreases very slowly with more iterations. Power iteration, on the other hand, starts with a much more inaccurate estimation, but the error decreases and converges quicker than for Monte Carlo. We chose power iteration because it is both efficient and easy to implement on a GPU; however, it is possible that a Monte Carlo method would perform better on a GPU. Because Monte Carlo methods involve many independent walks, they are potentially well-suited to the GPU's massively parallel architecture.

The difference between Monte Carlo and iterative methods can be seen in the camparison with the Cassovary runtimes in Table 3. While our GPU version is PPR-limited, the Cassovary implementation tends to be SALSA-limited. This is because of the difference in the way we compute Personalized PageRank. The Cassovary library PPR function uses the Monte Carlo random walks method, walking the graph and maintaining a visit count at each node, then ranking nodes by the number of visits. The number of steps is fixed, so the Cassovary PPR scales well. Our method iterates until convergence and computes the PPR for every node, not just ones that are in the vicinity of the start node.

One major limitation of our current implementation is the size of GPU memory. Today's GPUs have at most 12 GB of memory, whereas a CPU system can easily have ten times as much memory. This means that the entire current Twitter follow graph cannot fit in GPU memory. The twitter09 dataset is close to the maximum size that can fit on a single GPU. To compute WTF on the full graph, we would need to partition the graph and/or distribute it across multiple nodes. Scaling any large-scale parallel data analysis system, especially online social network system, beyond a single GPU remains a major challenge today. The unstructured and highly irregular connectivity of power-law graphs like social graphs makes it difficult to design partitioning and synchronization strategies for such graphs.

Table 3: Runtime comparison to Cassovary.

	wiki-Vote		twitter		gplus		twitter09	
Step (runtime)	Cassovary	GPU	Cassovary	GPU	Cassovary	GPU	Cassovary	GPU
PPR (ms)	418	0.45	480	0.84	463	4.74	884	832.69
CoT (ms)	262	0.54	2173	1.28	25616	2.11	2192	51.61
Money/SALSA (ms)	357	2.70	543	5.16	2023	18.56	11216	158.37
Total (ms)	1037	4.37	3196	8.36	28102	26.57	14292	1044.99
Speedup		235.7		380.5		1056.5		13.7

Another limitation is the bandwidth of the connection between main memory and GPU memory. Results in Section 5 do not include the data transfer time from CPU memory to the GPU. For our largest graph, the transfer time is 852.24 ms—about 85% of the compute time. In our experiments, we assume that graph data will be resident on the GPU, because it will be used to run a variety of algorithms on the follow graph, so transfer time will not be significant overall; however, for a different use case or very frequent updates to the graph, data transfer time could seriously limit performance.

Fortunately, future GPU systems have potential solutions for the modest size of today's GPU memories. The next node on NVIDIA's GPU roadmap is "Pascal" (2016), which can be connected to the CPU via a high-speed "NVLink" connection that allows access to the CPU's main memory at CPU-main-memory speeds (as well as supporting unified virtual memory across CPU and GPU). While such systems will still require careful memory management, they eliminate the current performance disadvantage in the case data fits in CPU memory but not GPU memory.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have shown that it is possible to use a GPU for recommendation algorithms on social graphs, but there are still many ways in which the performance could be improved. Software platforms for large-scale online social network analysis on hybrid CPU-GPU architectures could potentially offer better throughput and performance on systems that are more cost-effective than today's CPU-based cluster architectures. However, moving workloads to GPUs is challenging for the following reasons:

- Designing parallel algorithms for such systems is both difficult and time consuming.
- Today, limited PCIe bandwidth is a constraint for using discrete GPUs for communication-bounded tasks.
- The limited memory size on current GPUs makes it difficult to run algorithms on datasets that cannot fit in the GPU memory.

With the appearance of more graph processing libraries on the GPU, the design and implementation of online social network analysis software on the GPU is becoming easier and more efficient. Today, most online social network analysis algorithms are still compute-bound when running on large datasets. This makes a multi-CPU/multi-GPU architecture running across multiple nodes a promising solution.

We propose a two-layer framework running on a three-layer memory hierarchy where GPU memory serves as the fast cache, and CPU main memory serves as the second level cache, sitting atop data stored on hard disks/SSDs. In our case of building a recommendation system, the entire follow graph will be stored in CPU memory with disk as backing store. Multiple nodes, each containing one or more GPUs, will store the CoT for each user in the graph. Currently Twitter has 250 million users. If we keep the size of CoT at 1000 and use unsigned integers for vertex IDs, then we will need 2 TB to store CoTs for all users. That can be easily partitioned by user ID to fit on 4 or more machines with 512 GB or less of main memory. In this case, we can run PPR as an offline algorithm that runs only once per day or after a certain number of graph updates. The recommendation system can then work in real time with around 100 ms running time. Because the number of vertices in CoT is a constant 1000, the pruned graph that contains only vertices in CoT and vertices they connect to will be much smaller than the original follow graph. This set-up would reduce both the computational workload and GPU memory requirements.

Acknowledgments

Thanks to Brian Larson, Aneesh Sharma, Ashish Goel, and Pankaj Gupta (Twitter) for helpful comments and guidance on this work. We appreciate the financial support of UC Lab Fees Research Program Award 12-LR-238449, the DARPA XDATA program under AFRL Contract FA8750-13-C-0002, NSF awards CCF-1017399 and OCI-1032859, and a National Science Foundation Graduate Research Fellowship.

8. REFERENCES

[1] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte Carlo methods in PageRank computation: When one iteration is sufficient. *SIAM Journal of Numerical Analysis*, 45(2):890–904, Feb. 2007.

[2] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. *Proceedings of the VLDB Endowment*, 4(3):173–184, Dec. 2010.

[3] A. Goel. The "who-to-follow" system at Twitter: Algorithms, impact, and further research. WWW 2014 industry track, 2014.

[4] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: The who to follow service at Twitter. In *Proceedings of the International Conference on the World Wide Web*, pages 505–514, May 2013.

[5] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, Sept. 1999.

[6] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the International Conference on the World Wide Web*, pages 591–600, Apr. 2010.

[7] R. Lempel and S. Moran. SALSA: The stochastic approach for link-structure analysis. *ACM Transactions on Information Systems*, 19(2):131–160, Apr. 2001.

[8] J. Leskovec. SNAP: Stanford large network dataset collection. `http://snap.stanford.edu/data/`. Accessed: 2014-05-18.

[9] 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*, PPoPP '12, pages 117–128, Feb. 2012.

[10] S. A. Myers, A. Sharma, P. Gupta, and J. Lin. Information network or social network?: The structure of the Twitter follow graph. In *Proceedings of the Companion Publication of the International Conference on the World Wide Web*, WWW Companion '14, pages 493–498, Apr. 2014.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, Nov. 1999.

[12] A. Rungsawang and B. Manaskasemsak. Fast PageRank computation on a GPU cluster. In *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 450–456, Feb. 2012.

[13] K. G. Srinivasa, K. Mishra, C. S. Prajeeth, and A. M. Talha. GPU implementation of friend recommendation system using CUDA for social networking services. In *Proceedings of the International Conference on Emerging Research in Computing, Information, Communication, and Applications*, pages 890–895, Aug. 2013.

[14] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient PageRank and SpMV computation on AMD GPUs. In *Proceedings of the 39th International Conference on Parallel Processing*, pages 81–89, Sept. 2010.

APPENDIX

Here are the parameters and function calls used in our comparison using the Cassovary graph library. Our inputs for the personalized PageRank computation are as follows:

```
val numSteps = 100L * 1000L
val resetProb = 0.15
val maxSteps = None
val pathsSaved = Some(2)
val walkParams = RandomWalkParams(numSteps, resetProb, maxSteps, pathsSaved)
val graphUtils = new GraphUtils(graph)
val (topNeighbors, paths) = graphUtils.calculatePersonalizedReputation(startNode, walkParams)
```

Parameters for the SALSA calculation:

```
val leftResetProb = 0.2
val rightResetProb = 0
val numTopContributors = 5
val SALSA = new IterativeLinkAnalyzer(bipartiteGraphUtils, leftResetProb, rightResetProb, numTopContributors)
val numIterations = 5
val (topSimilarities, topRelevances) =
  SALSA.analyze(leftNodeInfo, numIterations, {LHSNodes => LHSNodes.neighborIds(OutDir)})
```