MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs

Zhisong Fu[*]
SYSTAP, LLC

Michael Personick
SYSTAP, LLC

Bryan Thompson
SYSTAP, LLC

ABSTRACT

High performance graph analytics are critical for a long list of application domains. In recent years, the rapid advancement of many-core processors, in particular graphical processing units (GPUs), has sparked a broad interest in developing high performance parallel graph programs on these architectures. However, the SIMT architecture used in GPUs places particular constraints on both the design and implementation of the algorithms and data structures, making the development of such programs difficult and time-consuming.

We present *MapGraph*, a high performance parallel graph programming framework that delivers up to 3 billion Traversed Edges Per Second (TEPS) on a GPU. MapGraph provides a high-level abstraction that makes it easy to write graph programs and obtain good parallel speedups on GPUs. To deliver high performance, MapGraph dynamically chooses among different scheduling strategies depending on the size of the frontier and the size of the adjacency lists for the vertices in the frontier. In addition, a Structure Of Arrays (SOA) pattern is used to ensure coalesced memory access. Our experiments show that, for many graph analytics algorithms, an implementation, with our abstraction, is up to two orders of magnitude faster than a parallel CPU implementation and is comparable to state-of-the-art, manually optimized GPU implementations. In addition, with our abstraction, new graph analytics can be developed with relatively little effort.

Keywords

Graph analytics, GPU, high-level API

1. INTRODUCTION

The rapid advancement of many-core processors, in particular graphical processing units (GPUs), has sparked a broad interest in developing high performance graph analytics programs on these architectures, thanks to their low cost, high memory bandwidth and high computing capacity. With appropriate optimization, modern GPUs demonstrate very strong computational performance, comparable to supercomputers of just a few years ago.

[*]Email: fuzhisong@systap.com

DOI:http://dx.doi.org/10.1145/2621934.2621936

However, scalable data-parallel graph processing on many-core hardware is a fundamentally hard problem that goes beyond the current state of the art. The single instruction multiple thread (SIMT) architecture [12] used in GPUs places particular constraints on both the design and implementation of algorithms and data structures, making the development on GPUs difficult, error prone, and inefficient.

Writing a correct and efficient GPU program is challenging. This is especially true for parallel graph algorithms. This is because: (a) these algorithms are data intensive, and hence memory efficiency is very important; (b) graphs are typically stored in irregular data structures; (c) graph operations have irregular data access patterns that are particularly inefficient on GPUs. and (d) different graph algorithms and data sets can impose very different workloads.

We present MapGraph, a programming framework for high performance parallel graph algorithms on the GPU (This an open source project, and the source code can be downloaded at `http://sourceforge.net/projects/mpgraph/`). The proposed framework provides a simple and flexible API that makes it easy to implement a wide range of graph algorithms. This API encapsulates the complexity of the GPU architecture while permitting the MapGraph CUDA kernels to make dynamic runtime decisions among a variety of optimization strategies. In our experiments, MapGraph delivers performance that is comparable to manually tuned GPU graph analytics implementations.

The remainder of this paper is organized as follows. In Section 2 we describe related work from the literature. In Section 3 we introduce the modified Gather-Apply-Scatter (GAS) abstraction [9] used in MapGraph. In Section 4 we discuss implementation details and data structures. In Section 5 we study the proposed platform using both synthetic and real-world graphs to measure performance. In Section 6 we summarize the results and discuss future research directions.

2. RELATED WORK

Several methods and software packages have been introduced in the literature to develop scalable, high performance parallel graph algorithms on many-core architectures. There are three approaches that are typically put forward: (1) low-level approaches that are optimized for specific algorithms, and which often use an algorithm that is intrinsically more efficient; (2) high-level abstractions that make it possible to write many algorithms using the same APIs; and (3) Domain Specific Languages with translation rules that target a specific hardware or software architecture.

Merrill *et al.* [11] developed the first low-level, work efficient implementation of BFS on GPUs, proposed adaptive strategies for assigning Threads, Warps, and CTAs to vertices and edges, and optimized frontier expansion using various heuristics to trade off time and space and obtain high throughput for algorithms with dynamic frontiers. The authors of [5] present a cost-efficient GPU-based PageRank implementation that achieves a 10-20× speedup compared to a quad-core CPU implementation. Other low-level approaches for efficient parallel graph algorithms are proposed in [4, 13]. While these approaches achieve high efficiency, they force users to address hardware issues, the challenges of parallel computations and data representations, and fail to offer a reusable and general purpose framework for writing parallel graph algorithms. The performance gain is achieved at the cost of programming complexity and generality. Further, users are often required to revisit or reinvent low-level optimizations for each new algorithm.

To address these issues, several high-level abstractions are proposed in the literature. Gharaibeh *et al.* [1] present *Totem*, a processing engine that provides a convenient environment to implement graph algorithms on hybrid platforms (CPU and GPU). This processing engine places low-degree vertices on the CPU and places high-degree vertices on the GPU. This improves overall performance if the graphs have a power-law vertex degree distribution. This work also defines a performance model for hybrid CPU/GPU graph processing and tests that model with up to two GPUs using random edge cuts.

Zhong *et al.* introduce *Medusa* [15], a GPU-based programming framework for parallel graph algorithms. Medusa helps developers leverage the capabilities of GPUs by writing sequential C/C++ code. Medusa offers a small set of user-defined APIs, and embraces a runtime system to automatically execute those APIs in parallel on the GPU. The authors develop a series of graph-centric optimizations based on the architecture features of the GPU. While Totem and Medusa present high-level abstractions for graph processing on GPUs, their performance is not comparable to the state of the art low-level GPU implementations.

In [9], the authors introduce *GraphLab*, a machine learning and graph programming framework based on the Gather-Apply-Scatter (GAS) model. GraphLab compactly expresses asynchronous iterative algorithms with sparse computational dependencies, ensures data consistency, and achieves a good performance on CPU architectures when compared with similar CPU graph processing platforms. In [14], the authors examine parallel graph algorithms on the CPU, compare their approach with GraphLab, and note that graph algorithms quickly become memory bound, thus limiting throughput. Our experiments have shown that GraphLab often experiences negative scaling as the number of CPU cores is increased.

Another approach that aims to solve the programability issue is Domain Specific Languages (DSL). Hong *et al.* introduce a DSL for easy and efficient graph analysis in [8] that they call *Green-Marl*. However, Green-Marl is focused on efficiency for coarse-grained parallelism systems (CPUs), and its optimization strategies are not amenable to many-core systems (GPUs) and fail to demonstrate performance on parallel graph algorithms approaching that of low-level GPU implementations. Modern DSLs generate thread assignment decisions through a static analysis of the user's program. However, data-dependent parallelism on GPUs (assigning threads to vertices based on the number of edges) requires deferring the thread assignment until runtime against a specific graph. Therefore, in order to achieve high performance, DSLs may need to target efficient runtimes, such as the one proposed here.

3. GAS ABSTRACTION

The GAS model [7] presents a vertex-centric abstraction similar to *Pregel* [10]. The input to a GAS program is a directed graph $G(V, E)$, where V denotes the vertex set and E denotes the directed edge set. Each vertex is uniquely identified by an integer vertex index, denoted v_i, and $v_i \in V$. A directed edge is associated with a source vertex index v_i and a target vertex index v_j and is denoted e_{ij}. Typically, each vertex and edge is associated with user defined data, that we call the *vertex value* and *edge value* respectively. An edge e_{ij} is called an in-edge of v_j and an out-edge of v_i. We call v_i an in-edge neighbor of v_j and v_j an out-edge neighbor of v_i if there is an edge e_{ij}.

A typical GAS computation consists of three stages: the data preparation stage, the iteration stage, the output stage. The data preparation stage initializes the graph, the vertex and edge values, and the initial frontier for the computation (the frontier is the set of vertices that are active in a given iteration). The iteration stage consists of a sequence of iterations that update the vertex values and edges values until the algorithm terminates. Each iteration updates vertices that are in the current *frontier* and defines the frontier for the next iteration. The output stage extracts the desired data.

Each GAS iteration consists of three conceptual execution phases: *Gather*, *Apply*, and *Scatter*. The Gather phase assembles information from adjacent edges and vertices through a generalized sum over the neighborhood of the central vertex (Figure 1 left). It may read on the in-edges, the out-edges, or both. The Apply phase acts on each vertex in the current frontier and updates the value of this vertex (Figure 1 middle). The Scatter phase distributes messages to the adjacent edges and vertices of the central vertex, and may operate on the in-edges, the out-edges, or both (Figure 1 right).

A large number of interesting algorithms can be implemented with this abstraction. For instance, the PageRank algorithm can be implemented with the GAS model as follows. In the Gather phase, each vertex v_i in the frontier computes $\phi_j = \frac{r_j}{N_j^{out}}$ for all in-edges with source vertex v_j, where r_j denotes the rank of v_j, and N_j^{out} denotes the number of out-edges of v_j. Then, each vertex sums up all ϕ_j: $\phi_i = \sum \phi_j$. In the Apply phase, each vertex updates its value by $r_i = 0.85 + 0.15 * \Phi_i$. Finally, in the Scatter stage, vertex v_i checks if it is changed and if so adds its out-edge neighbors to the new frontier.

4. IMPLEMENTATION

In this section, we provide a detailed description of the MapGraph runtime.

4.1 Data Structure

Graphs are often stored as a sparse matrix. MapGraph currently uses the Compressed Sparse Row (CSR) data structure [3] to store the topology of a graph. CSR consists of two arrays: *row-offsets*, *column-indices*. The *column-indices* array contains the concatenation of the adjacency lists of the graph. The *row-offsets* array contains the indices indicating where each adjacency list starts. (deleted graph for CSR)

4.2 Strategies

Parallel graph algorithms are generally data intensive and hence, memory bound. GPUs enjoy ten times the memory bandwidth of CPU architectures. However, the relatively restrictive architecture of the GPU makes it challenging to fully release their potential. Graphs have an irregular structure and access patterns and the degree of vertices (the number of in-edges or out-edges) in real world graphs can vary over several orders of magnitude. On a GPU, these characteristics cause uncoalesced memory access patterns, create unbalanced workloads for threads, and lead to thread divergence. High performance on GPUs requires low-level optimizations to addresses these problems. For higher productivity, these optimizations should remain hidden from the users. The GAS Apply phase is embarrassingly parallel, therefor our optimization strategies focus on improving the performance of the Gather and Scatter phases. MapGraph uses two strategies to address these architectural challenges: *dynamic scheduling* and *two-phase decomposition*.

4.2.1 Dynamic Scheduling

Merrill *et al.* [11] introduced a work-efficient implementation for BFS on the GPU and used a dynamic scheduling strategy to improve memory performance. This strategy distributes the workload to the threads according to the degree of the vertices. We adopt the same idea and extend it to more algorithms using the GAS abstraction.

In the Scatter phase, each vertex in the frontier accesses its adjacent edges (stored in *column-indices*) and vertices in parallel. If we assign each vertex to a thread, the memory access would be uncoalesced, and workload would be unbalanced due to the variation in vertex degree. Therefore we apply three different strategies depending on the degree of the vertices.

CTA-based scattering distributes the workload for a Cooperative Thread Array (CTA) among threads according to the vertex degrees. It assigns the workload of a vertex in the frontier to a whole CTA, and each thread of the CTA is responsible for only one neighbor of the vertex. This strategy is used when the degree of the vertex is large.

Scan-based scattering uses a prefix sum to compute the starting and ending points in the *column-indices*, according to the vertices assigned to a CTA and the *row-offsets* array. This operation forms a compact scatter vector in shared memory, local to the CTA. Scan-based scattering then enlists the entire CTA to gather the referenced neighbors from the *column-indices* array using this perfectly compact scatter vector.

Warp-based scattering performs a coarse-grained redistribution of scattering workloads. Each thread enlists its entire warp to access information from its adjacent vertices and edges. Each thread first attempts to vie for control of its warp by writing its thread-identifier into a single word shared by all threads of that warp. Only one write will succeed, thus determining which thread is subsequently allowed to command the warp, as a whole, to access its corresponding neighbors. This process repeats until all threads have all had their adjacent neighbors accessed. Warp-based scattering is similar to CTA-based scattering, but each vertex enlists the warp rather than the CTA to access the adjacent vertices and edges.

We combine these three scheduling strategies (CTA-based, scan-based, warp-based) for better load-balance and memory performance. We first perform the CTA-based scattering for vertices with degrees larger than the CTA size. Then, we apply warp-based scattering to vertices with adjacency lists smaller than the CTA size, but greater than the warp width. Finally we perform scan-based scattering to efficiently process the remaining “loose ends”. The same scheduling strategies can be applied to the Gather phase in an analogous manner.

While *dynamic scheduling* yields very high performance for some problems, including BFS and SSSP on many graphs, it has several drawbacks. Since we separate each work-group into three separate stages within our kernel, we lose parallelism among the stages and hence the the instruction level parallelism is decreased. Also, since each thread in the scan portion of the algorithm must communicate its whole adjacency list to the rest of the CTA, other threads stall while waiting for all of these items to be loaded. Finally, the strategy assigns equal numbers of frontier vertices to the CTAs, so the total number of adjacent vertices can vary largely among CTAs. This leads to imbalanced workloads among CTAs.

4.2.2 Two-phase Decomposition

We also explore a two-phase decomposition strategy based on the work of Baxter [2]. This strategy is used in another graph processing work [6]. This strategy attempts to achieve perfect load-balancing for threads within and across CTAs by decomposing the scattering process into two phases: a scheduling phase and a computation phase. Instead of assigning an equal number of vertices to be processed by a CTA, this strategy organizes groups having total numbers of adjacent edges of equal size. The scheduling phase does this by finding the intersection of each CTA’s adjacency list’s starting and ending points within the *column-indices* array using an efficient sorted search. Then, in the computation phase, each thread accesses the same number of adjacent vertices and performs the same operation. The disadvan-

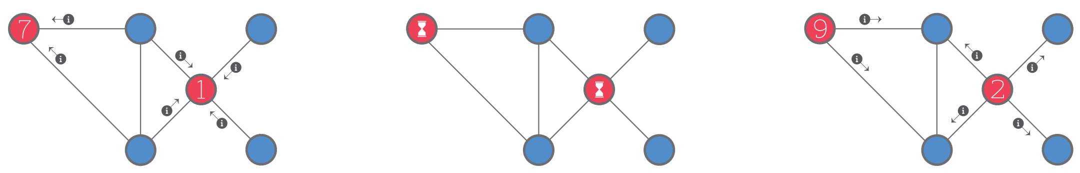


Figure 1: The GAS phases (from left to right: Gather, Apply, Scatter). The red vertices denote the frontier, and the numbers on the vertices are their values.

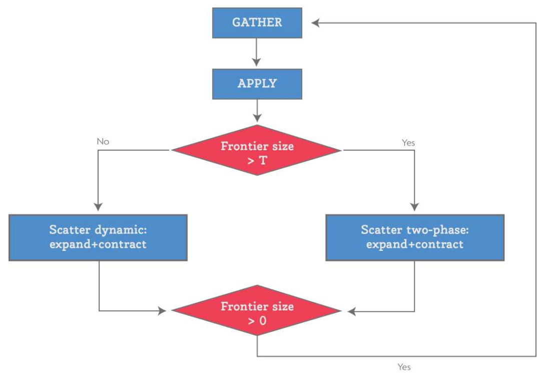
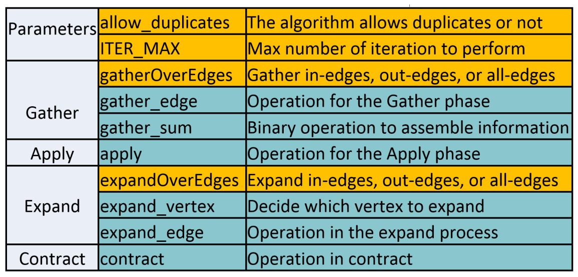

GATHER
APPLY
No
Frontier size > T
Yes
Scatter dynamic: expand+contract
Scatter two-phase: expand+contract
Frontier size > 0
Yes

Figure 2: MapGraph Computation Pipeline.



Parameters	allow_duplicates	The algorithm allows duplicates or not
	ITER_MAX	Max number of iteration to perform
Gather	gatherOverEdges	Gather in-edges, out-edges, or all-edges
	gather_edge	Operation for the Gather phase
	gather_sum	Binary operation to assemble information
Apply	apply	Operation for the Apply phase
Expand	expandOverEdges	Expand in-edges, out-edges, or all-edges
	expand_vertex	Decide which vertex to expand
	expand_edge	Operation in the expand process
Contract	contract	Operation in contract

Figure 3: MapGraph API. The configuration parameters and methods are in orange background, and kernel methods are in blue background.

tage of this strategy is the overhead of the scheduling phase.

4.2.3 *MapGraph Computation Pipeline*

As described above, the dynamic scheduling and the two-phase decomposition have their own advantages and disadvantages. For the Gather phase, we always use the two-phase decomposition strategy, because it is generally faster according to our experiments, and there is some overhead associated with making this decision dynamically. For the Scatter phase, our experiments show that the two-phase decomposition strategy performs better for iterations with a large frontier while for iterations with a small frontier, the dynamic scheduling strategy performs better. In practice, we use one of these strategies to generate the new frontier in each iteration for the Scatter phase. We call the process of generating new frontier the *expand* process. After the expand, we perform another process to reduce (or eliminate) the large numbers of duplicates in the new frontier that can arise due to simultaneous discovery [11]. We call this the *contract* process.

The MapGraph computation pipeline is shown in Figure 2. In each iteration, after the Gather and Apply phases, we check if the frontier size is larger than a pre-defined threshold. If true, we perform the expand and contract with the two-phase decomposition strategy. Otherwise, we perform the expand and contract with the dynamic scheduling strategy.

4.2.4 *Programming Interface*

MapGraph offers three user-defined data types: $VertexType$, $EdgeType$, $FrontierType$. These three types define algorithm-specific values associated with the vertices, edges and frontier, respectively. MapGraph also offers a flexible variant of the GAS API for high programmability. Using user defined methods that are invoked from MapGraph kernels, programmers can define computations on the vertices, edges, and frontier. MapGraph also provides a set of configuration parameters and utility methods as library calls for iteration control and other functionalities (See Figure 3). The user defined methods and the utility methods are all C++ methods. This means users need only write sequential C++ code to use the framework.

4.2.5 *MapGraph Workflow*

There are three steps to implement a graph algorithm with MapGraph. First, the developer defines the basic data structures ($VertexType$, $EdgeType$, $FrontierType$) in C using a Structure Of Arrays (SOA) pattern. Second, the developer implements the methods according to the specific graph algorithm. Third, the developer composes the main program, including initializing the graph structure, configuring the framework parameters and invoking the customized methods.

Many graph computation tasks require multiple iterations till convergence. MapGraph provides two ways to control the number of iterations of the algorithm. Developers can use both of them for a more flexible iteration control. First, the developer can specify the maximum number of iterations, $ITER\_MAX$. MapGraph terminates when the number of iterations reaches the predefined limit. Second, MapGraph continues the iterations until the frontier size is zero.

5. RESULTS

To evaluate the efficiency of MapGraph, we developed a set of common graph algorithms using MapGraph and compared peformance with both CPU and GPU implementations of these algorithms.

5.1 Graph Primitives

We use MapGraph to implement four common graph analytics operations: Breadth First Search (BFS), Single Source Shortest Path (SSSP), Connected Components (CC), and PageRank (PR). These algorithms give rise to different workloads, which makes them useful for understanding the performance of the library. These algorithms can also be used as building blocks for higher level applications such as graph-based analysis and ranking, community discovery and finding influential nodes.

BFS. BFS is a widely used graph search algorithm. The search starts from a predefined root vertex and iteratively expands to find and label all the reachable vertices. In BFS, vertices are discovered in levels. Each level corresponds to one iteration of the algorithm. For BFS, only the Scatter phase of the GAS model is necessary. In this phase, each newly discovered vertex in the current frontier is assigned its level and discovers its out-edge vertices for the next iteration.

SSSP. The SSSP algorithm finds a shortest path from a specified source vertex to all other vertices. The $VertexType$ contains a *distance* attribute array for all the vertices, and the $EdgeType$ contains an *edge_weight* array for all edges. We adopt an improved Bellman-Ford algorithm in our implementation, thanks to our flexible API. The original Bellman-Ford algorithm iteratively updates the distances of the vertices by looking at in-edge vertices. Instead of accessing all in-edge vertices for every vertex in the frontier, we only access the ones whose distances are changed in the last iteration. Specifically, we maintain a *predecessor_distance* array in the $FrontierType$. In the expand process of the Scatter phase, we store the distance of the changed predecessor for every out-edge. Then, in the contract process, we use the *predecessor_distance* array to compute the correct distances for the vertices in the current frontier. In this way, we do not need the Gather phase. We call this method the *push-style* update because essentially each changed vertex pushes its value to its descendant.

CC. The CC algorithm finds how many connected components there are in the graph, assuming that every graph is undirected. The algorithm is simple. The $VertexType$ contains an array of *compindex* for all vertices, and this array is initialized with the vertex indices. The initial frontier contains all vertices. In each iteration, every vertex in the frontier looks at its neighbors' *compindex* values and updates itself with the minimum of these values and its previous value. Similar to SSSP, we can use the $push - style$ update to reduce memory accesses and eliminate the Gather phase.

PageRank. The initial frontier is all vertices in the graph. As described in Section 3, the GAS implementation of PageRank makes use of all three phases: Gather, Apply, and Scatter. PageRank does not allow duplicates in the frontier (it leads to double counting). Hence we used a $flag$ array in $VertexType$ to indicate if a vertex is already in the frontier in the Scatter phase.

5.2 Experimental Setup

We have conducted the evaluations on a workstation equipped with a NVIDIA Tesla K20, two Intel Xeon X5680 CPUs (12x 3.3GHz CPU cores). Our experimental datasets include both real-world and synthetic graphs. Table 1 shows their basic characteristics. Webbase[1] is a directed graph and represents web connectivity. The Delaunay graph[2] is undirected and represents a delaunay triangulation of random points in a 2-D plane. The Bitcoin graph[3] is directed and models bitcoin transaction data. It has a small frontier and a very long tail (over 8000 iterations are required for traversal). The Wiki graph[4] depicts the connections among Wikipedia articles. The Kron graph[5] is an undirected, scale-free random graph with power-law degree distribution.

Table 1: Graphs.

dataset	#Vertices	#Edges	Max degree
Webbase	1,000,005	3,105,536	23
Delaunay	2,097,152	6,291,408	4,700
Bitcoin	6,297,539	28,143,065	4,075,472
Wiki	3,566,907	45,030,389	7,061
Kron	1,048,576	89,239,674	131,505

5.3 Comparison with GPU Implementations

We compare our MapGraph BFS implementation with the state-of-the-art GPU implementation for BFS described by Merrill [11] (B40c) and a GPU-based high-level graph processing framework *Medusa* [15]. The running times in milliseconds are shown in Table 2. The results demonstrate that performance of MapGraph is comparable to the manually optimized, low-level BFS implementation. For three of the graphs, MapGraph is faster and for the other two, MapGraph is slower. The last column gives the speedup of MapGraph versus Medusa. MapGraph achieves higher performance than Medusa for all graphs and is up to 42 times faster.

Table 2: BFS GPU implementation comparison. Speedup is MapGraph against Medusa. B40c and MapGraph have similar performance.

Dataset	Medusa	B40c	MapGraph	Speedup
Webbase	9.1	3.2	1.2	7.6
Delaunay	429.6	33.9	24.5	17.5
Bitcoin	14873.3	411.4	354.3	42.0
Wiki	1167.8	45.2	51.0	22.9
Kron	154.3	31.3	47.7	3.2

5.4 Comparison with CPU-based Framework

In order to compare the performance of MapGraph with a CPU-based high-level graph processing framework, we implemented the same graph algorithms for GraphLab v2.0. The comparison results are shown in Table 3. The PR CPU scaling results in Table 3 are consistent with those obtained

[1]`http://www.cise.ufl.edu/research/sparse/MM/Williams/webbase-1M.tar.gz`
[2]`http://www.cise.ufl.edu/research/sparse/MM/DIMACS10/delaunay_n21.tar.gz`
[3]`https://www.dropbox.com/s/994xui3sgk5pa4c/bitcoin.mtx`
[4]`http://www.cise.ufl.edu/research/sparse/MM/Gleich/wikipedia-20070206.tar.gz`
[5]`http://www.cise.ufl.edu/research/sparse/MM/DIMACS10/kron_g500-logn20.tar.gz`

by the GraphLab authors (private communication) and are self-consistent across the different algorithms. The comparison results are shown in Table 3. The GraphLab results are reported for both single thread (GL-1) and 24 threads (GL-24, 12 cores plus hyperthreading). In many cases, the best performance for GraphLab was obtained with only a single CPU core (Table cells are shaded when GL-1 gives better performance than GL-24). The SU column of the table shows the speedups of MapGraph (MPG) compared to GL-24. All running times are in milliseconds. As shown in the tables, MapGraph achieves up to 119× speedup compared to GraphLab running on a 12-core CPU.

Table 3: Results (running times in milliseconds). Table cells are shaded when GL-1 gives better performance than GL-24. Speedups are against GL-24.

Alg	Dataset	GL-1	GL-24	MPG	SU
BFS	Webbase	80.85	21.0	1.2	17.5
	Delaunay	4,715.8	1,806.8	24.5	73.7
	Bitcoin	104,201.6	34,733.9	354.3	98.0
	Wiki	4,880.7	3,486.2	51.0	68.4
	Kron	686.9	3,434.3	47.7	72.0
SSSP	Webbase	85.2	20.8	5.7	3.6
	Delaunay	4,692.3	1,790.9	98.3	18.2
	Bitcoin	108,288.5	35,044.8	502.5	69.7
	Wiki	4,871.1	6,165.9	118.1	52.2
	Kron	626.7	6,963.2	104.2	66.8
CC	Webbase	863.5	2,272.2	41.6	54.6
	Delaunay	7,988.2	29,586.0	302.2	97.9
	Bitcoin	18,280.9	24,703.9	512.4	48.2
	Wiki	4,836.5	48,364.7	686.3	70.5
	Kron	3,310.0	36,777.8	309.2	118.9
PR	Webbase	731.3	1,924.5	97.0	19.8
	Delaunay	1,555.9	4,576.3	146.6	31.2
	Bitcoin	57,615.4	72,930.9	796.8	91.5
	Wiki	10,706.8	71,378.3	2,610.3	27.3
	Kron	4,931.3	44,830.0	1,671.3	26.8

6. CONCLUSIONS

Parallel graph algorithms have severe scalability problems on CPU architectures due to the limited memory bandwidth. We have shown that it is possible to obtain high performance across a variety of graph algorithms and data sets on a GPU while maintaining a high-level abstraction. Since no single strategy dominates the others, MapGraph dynamically selects the strategy that will best mitigate the architectural limits of the GPU for the actual workload. In future work, we plan to extend the MapGraph platform to parallel graph algorithms on GPU compute clusters.

7. ACKNOWLEDGMENTS

This work was (partially) funded by the DARPA XDATA program under AFRL Contract #FA8750-13-C-0002. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. D14PC00029.

8. REFERENCES

[1] E. S.-N. Abdullah Gharaibeh, Lauro Beltrao Costa and M. Ripeanu. Totem: Accelerating graph processing on hybrid cpu+gpu systems. *GPU Technology Conference*, 2013.

[2] S. Baxter. Modern gpu library. 2013. http://www.moderngpu.com/.

[3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[4] G. Chapuis, H. Djidjev, R. Andonov, S. Thulasidasan, and D. Lavenier. Efficient multi-gpu algorithm for all-pairs shortest paths. In *IPDPS 2014*, May 2014.

[5] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen. Parallel pagerank computation using gpus. In *Proceedings of the Third Symposium on Information and Communication Technology*, SoICT '12, pages 223–230. ACM, 2012.

[6] E. Elsen and V. Vaidyanathan. Vertexapi2 - a vertex-program api for large graph computations on the gpu. 2014. http://www.royal-caliber.com/vertexapi2.pdf.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[8] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[11] 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, New York, NY, USA, 2012. ACM.

[12] NVIDIA. Cuda programming guide. http://www.nvidia.com/object/cuda.html.

[13] J. Soman, K. Kothapalli, and P. J. Narayanan. Some gpu algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(04):325–339, 2010.

[14] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[15] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 99:1, 2013.