Large-Scale Graph Processing Algorithms on the GPU

Yangzihao Wang, Computer Science, UC Davis
John Owens, Electrical and Computer Engineering, UC Davis

1 Overview

The past decade has seen a growing research interest in using large-scale graphs to analyze complex data sets from social networks, simulations, bioinformatics, and other applications. As the size of these data sets increases as we move into the petascale and beyond, we see a need for a more efficient method for large-scale graph analysis. Modern graphics processors (GPUs) are high-performance, highly parallel, fully programmable architectures and could be a good fit for this task. Initial research efforts in this area are promising, although widespread use has not yet arrived. However, there are several challenges in graph processing, including dependencies between vertices in the graph, irregular memory accesses during graph processing, and scalability to larger data sets and clusters.

2 Definition of Large-Scale Graph

Sources of real-world large graphs include:

- Social graphs (Facebook, Twitter, Google+, LinkedIn, etc.)
- Endorsement graphs (web link graph, paper citation graph, etc.)
- Location graphs (map, power grid, telephone network, etc.)
- Co-occurrence graphs (term-document bipartite, click-through bipartite, etc.)

These graphs have common characteristics. The first is their large scale. For example, by January 2011, Facebook had about 600 million nodes; major search engines have indexed tens of billions of webpages over a trillion nodes. Second, these graphs are sparse, meaning the number of edges at one vertex is far less than the total number of vertices. The third characteristic of the graphs is rich information on nodes and edges, and we expect the graphs of interest in this project will have substantial information associated with vertices and/or edges. For example, each node in Facebook can have attributes such as age, gender, interests, etc. and each edge in Facebook can have attributes such as creation time, type of relation, communication frequency, etc.

Tasks of large-scale graph analysis range from simple to advanced. Some of the simpler fundamental large-scale graph analyses include 1) efficient search in the graph (e.g. graph traversal and search algorithms); 2) finding patterns in the graph (e.g. shortest path algorithms, matching algorithms, centrality computing algorithms, and list ranking algorithms); and 3) partitioning large graphs into sub-graphs (e.g. connected component algorithms, graph-cut algorithms). Advanced large-scale graph analyses include 1) Graph indexing and ranking (e.g. pagerank); 2) data mining (clustering and classification algorithms); 3) structured graph query (e.g. RDF query languages such as SPARQL) 4) recommendations.

3 Graph Algorithms on GPUs

We begin with preliminaries: what are the keys to high performance on the GPU and what should we be measuring as we evaluate graph implementations? Then we look at how to represent graphs on GPUs—a crucial topic since the graph representation is critical for both parallel efficiency and memory performance—and then proceed to survey the existing work in the field.

3.1 Keys to High Performance on the GPU

The GPUs we propose to use in this project are discrete GPUs connected to the host CPUs by the PCI Express bus. This bus has limited bandwidth (16 GB/s in the current generation) and without careful application design, can become a bottleneck. Consequently successful GPU applications typically *must perform a substantial amount of computation per data element* to amortize the cost of the transfer. Fortunately, for complex graph computations, this would likely not be an issue.

Once the data is on the GPU, the challenges include *sustaining high data bandwidth to the GPU's memory* and *profitably leveraging the parallelism in the GPU compute units.* For both, minimizing divergence is key: accessing memory in large contiguous chunks ("coalesced access") and maximizing the ability of threads within warps to maintain the same control path through the code are both crucial techniques. The choice of graph data structure and graph algorithm strongly influences the performance here.

Finally, *choosing algorithms with better algorithmic complexity* has the potential to make an enormous difference in the runtime of the implementation (note the progression of breadth-first-search implementations below).

3.2 Evaluating Graph Implementations

Much of the early research in this area demonstrates that particular algorithm implementations are possible but not optimal. Evaluations should include the following:

Transfer costs Is the cost of transfer to and from the CPU counted in the comparison? It is legitimate if not, since most individual graph algorithms will not run in isolation but instead will run in series with many other algorithms; but this needs to be mentioned.

Comparison against CPU codes In performance comparisons, does the GPU implementation measure against top-tier CPU codes? Are those CPU codes optimized and parallelized?

Datasets Are GPU implementations run on a wide range of datasets with different depths and distribution of connections? Are these datasets static or dynamic? Are they rich graphs (graphs contain typed links and vertex attributes in addition to link weights)?

Out of core performance How does the implementation perform as the size of the graph exceeds the size of the GPU memory?

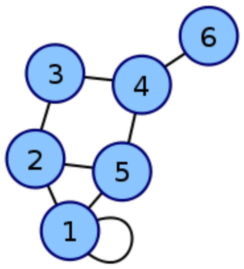
Scalability beyond one GPU How does the implementation scale both with the addition of GPUs on a single node and with additional nodes?

The last two points are largely unaddressed in the current literature.

3.3 Graph Data Representation

Among several graph data representations, the *adjacency matrix* and a collection of *adjacency lists* are the two main representationss used in most existing parallel graph processing works.[4,5,10,11,13]

- The *adjacency matrix* is an $n \times n$ matrix where the non-diagonal entry a_{ij} is the weight value from vertex i to vertex j, and the diagonal entry a_{ii} can be used to count loops on single vertices. For most large-scale graphs, the resulting matrix is typically sparse; representing it directly as a sparse matrix allows the use of existing libraries for sparse matrix operations such as cuSparse. Katz et al.[10] represent the adjacency matrix as a 2D texture in GPU memory. The downside of such a representation is its waste of space.



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 1: *At left, an unweighted undirected graph; at right, its adjacency matrix.*

- Instead, *adjacency lists* provide more compact storage for more widespread sparse graphs. A basic adjacency list stores all edges in a graph. One typical way of implementing it is using one array to store a list of neighbor nodes and another array to store the offset of the neighbor list for each node. Merrill et al.[11] use the well-known compressed sparse row (CSR) sparse matrix format to represent graphs in their BFS implementation. The column-indices array C and row-offsets array R are equivalent to the neighbor nodes list and the offset list in the basic adjacency list definition. This representation enables them 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. In the context of parallel BFS, parallel threads use prefix sum when assembling global edge frontiers from expanded neighbors and when outputting unique unvisited vertices into global vertex frontiers. Jia et al.[9] represent the graph as an edge list to better assign edges to threads in their centrality algorithm. Instead of using two unequal length arrays—an offset array and a neighbor list array—they use two even length arrays to store the vertex pair for each edge. Their edge-centric representation requires more GPU memory though, which could limit their application to larger graphs.

- Blelloch[3] proposes the v-graph (vector graph) for graph data representation. This representation uses segmented vectors to store graph topology. For undirected graphs, it uses a single segmented vector to store edge

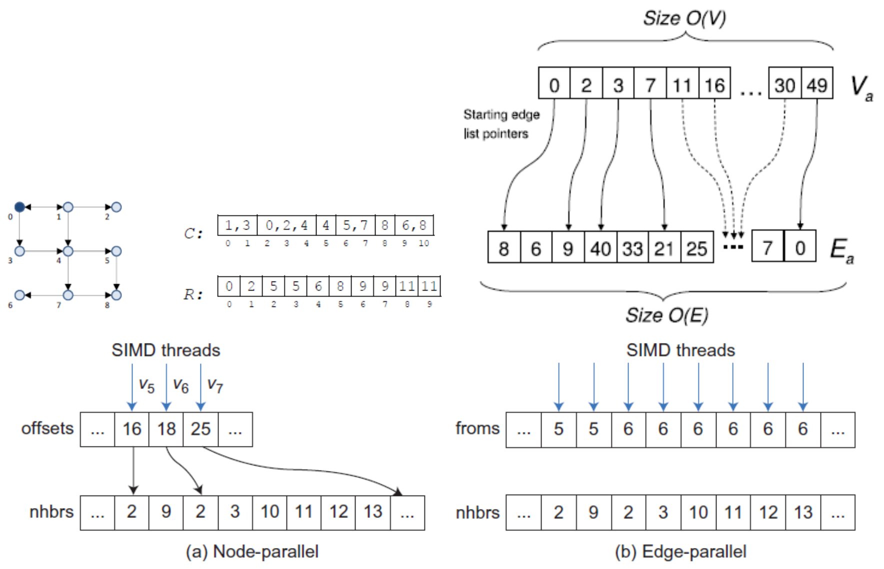
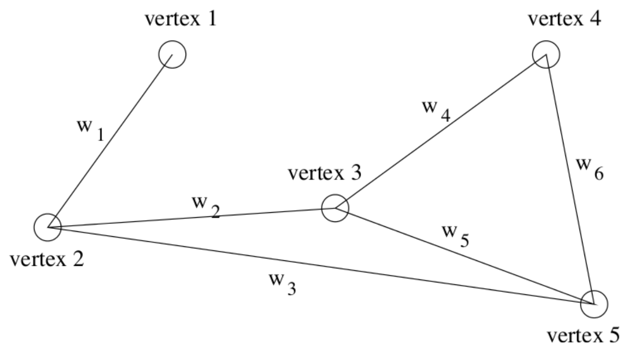


Figure 2: *Upper left: Illustration of the adjacency list used by Merrill et al.*[11] *Upper right: Graph representation with vertex list pointing to a packed edge list. (From Harish et al.*[5]*) Lower: Illustration of a node list and an edge list used by Jia et al.*[9]

information. Each segment corresponds to a vertex and each element within a segment corresponds to one of the edges of that vertex. For directed graphs, it uses two segmented vectors: one for the incoming edges, and one for the outgoing edges. Each element in the outgoing edges vector is a pointer to a position in the incoming edges vector. For both directed and undirected graphs, the representation uses additional vectors to include weights and other information. A graph can be converted from most other representations into the v-graph representation by creating two elements per edge (one for each end) and sorting the edges according to their vertex number. Parallel sorting algorithms such as radix sort can be used to place all edges that belong to the same vertex in a contiguous segment. The cross-pointer array in this representation enables an $O(1)$ step complexity of dynamic graph manipulations such as adding or deleting an edge or a vertex, which is not easily accomplished with the other representations.



Index	=	[0	1	2	3	4	5	6	7	8	9	10	11]
vertex	=	[1	2			3			4		5]		
segment-descriptor	=	[1	3			3			2		3]		
cross-pointers	=	[1	0	4	$\boxed{9}$	2	7	10	5	11	$\boxed{3}$	6	8]
weights	=	[w_1	w_1	w_2	w_3	w_2	w_4	w_5	w_4	w_6	w_3	w_5	w_6]

Figure 3: *An example of the undirected v-graph representation. Each pointer points to the other end of the edge. So, for example, edge w_3 in vertex 2 contains a pointer (in this case 9) to its other end in vertex 5. The segment-descriptor specifies the number of edges incident on each vertex.*

3.4 Current Graph Algorithms on GPU

3.4.1 Breadth-First Search (BFS)

As a core primitive for graph traversal, parallel BFS algorithms on GPUs are representative of data-dependent parallel computation with irregular memory accesses. Merrill et al.[11] notes that most work in this area has quadratic complexity,[5,7,8] using quadratic methods to inspect every edge/vertex during every iteration. The work complexity is $O(V^2 + E)$ as the worst case requires V BFS iterations. The work of Harish et al.[5] is the first to implement BFS using level synchronous operations on the GPU. The algorithm processes all the vertices at a particular level in parallel. Concurrent computation takes place at the vertices of current level and all threads wait for other threads at that level to finish, treating the GPU as a bulk synchronous parallel machine. However, the algorithm shows lower performance on low degree graphs. In such case, expansions of the frontier is very slow at every level, which limits the amount of achieved parallelism. Also, frontier expansion will cause uneven load balancing between different levels. However, running on a single GTX8800 GPU with a randomly generated graph, it still gives a 20–50x speedup vs. a CPU implementation.

The work of Hong et al.[7] is the first to address the irregularity of a

BFS workload. They introduce a novel virtual warp-centric programming method. Rather than threads, entire warps are mapped to vertices. During neighbor expansion, the SIMD lanes of an entire warp are used to stripmine the corresponding adjacency list. They later present a hybrid method[8] that dynamically chooses the best execution method for each BFS-level iteration from three alternatives: sequential execution, multi-core CPU execution, and GPU execution. This hybrid method can prevent poor worst-case performance, but does not improve performance of the BFS algorithm on GPU.

The work of Merrill et al.[11] is the first linear parallelization BFS algorithm and the first on multiple GPUs. It is clearly the most complete and advanced work on BFS traversal on the GPU. Noting that methods that require atomics have limited scalability, they focus on fine-grained task management built upon an efficient prefix sum. With both a memory-access-efficient data representation of graphs and a load-balancing warp-centric algorithm, their work complexity reaches an asymptotically optimal $O(|V| + |E|)$. The algorithm works on diverse graphs and gives a 3.3 billion edges per second performance on single GPU and a 8.3 billion edges per second performance using four GPUs on a single node for both the uniform-random and RMAT datasets. The result on four GPUs is a roughly 6.4x speedup compared to the state-of-the-art CPU BFS algorithm using four 8-core Intel Nehalem-based XEON CPUs.[1] For expanding the algorithm to multiple GPUs, the paper implements a simple partitioning of the graph into equally-sized, disjoint subsets of V. However, this method of partitioning progressively loses any inherent locality as the number of GPUs increases. Better partitioning algorithms, graph data representations and, hardware support of irregular memory accesses will address this issue. Also, the cost of global synchronization between BFS iterations is much higher across multiple GPUs. This would dominate for BFS problem in general. One issue the paper does not address is how to expand the algorithm to multiple GPUs on multiple nodes. Though their algorithm has minimized the required times of global synchronization, the communication cost will still increase when expanded to multiple GPUs on multiple nodes because of the relatively lower network bandwidth compared to GPU-to-GPU (GPUDirect) or GPU-to-CPU (PCIe) communication.

3.4.2 Single Source Shortest Path (SSSP) and All-Pair Shortest Path (APSP)

Given a directed graph $G(V, E)$ with positive weights, the single-source-shortest-path (SSSP) problem requires finding the smallest combined weight of edges that is required to reach every vertex V from the source vertex S. Harish and Narayanan[5] propose the first GPU implementation of the traditional SSSP serial method, Dijkstra's algorithm, using CUDA. They also give a Floyd-Warshall (FW) algorithm for solving the All-Pair Shortest Path (APSP) problem, which is finding the least weighted path from every vertex to every other vertex in the graph $G(V, E)$. However, due to FW's high time complexity ($O(V^3)$) and space complexity ($O(V^2)$), their GPU implementation can only be used on very small graphs. They propose another APSP algorithm that uses their GPU SSSP formulation in order to process larger-scale graphs. Their GPU SSSP, however, suffers from the inefficiency of atomic operations. Their SSSP and APSP algorithms using an NVIDIA GTX8800 GPU show 70 and 17 times speedup over the serial implementation on an Intel Core 2 Duo processor. However, the performance for real-life graphs with several million vertices does not show the same improvement because of the low average degree of these graphs. A degree of 2–3 makes these graphs almost linear. In the case of linear graphs, parallel algorithms have poorer performance, as it becomes necessary to process every vertex in each iteration. Katz and Kider presented a tiled FW algorithm in 2008.[10] It revises the original straightforward FW algorithm into a hierarchically parallel method that can be distributed, in parallel, across multiple GPUs. The algorithm uses an adjacency matrix to represent graph and divides the matrix into tiled submatrices of equal size. During each phase, only submatrices which are dependent on each other are computed. This strategy enables out-of-core graph processing, reduces the overall space complexity, and makes the algorithm scalable to multi-GPUs. Compared to Harish and Narayanan's work, this method has a 5.0–6.5x increase in performance due to a better shared-memory cache-efficient strategy.

Harish et al.[6] implement a Gaussian elimination based APSP algorithm on GPU. The algorithm splits each APSP step recursively into 2 subproblems involving graphs of half the size. The base case is when there are 16 or fewer vertices in a subgraph. They achieve a speed up of 2–4 times over Katz and Kider for larger general graphs with more than 30,000 vertices.

The APSP algorithm can be further used in several advanced graph algorithms such as computing betweenness centrality in graphs.

3.4.3 Other Algorithms

Minimum Spanning Tree (MST) A spanning tree of a connected, undirected graph is a subgraph that connects all the vertices together. An MST is a spanning tree with weights less than or equal to the weight of every other spanning tree. Harish et al.[6] implement a modified parallel Borůvka algorithm on CUDA. They create partial spanning trees called supervertices from all vertices; supervertices can grow individually and they merge when come in contact. This algorithm performs well bacause each supervertex can grow independently. However, the merging step is an irregular operation as an uneven number of vertices might assigned to a single supervertex. Instead of using irregular atomic operations, Vineet et al.[15] exploit parallel data mapping primitives such as scan, split and compaction on the GPU for marking MST edges and merging supervertices. They gain a speed up of 8 to 10 times over their previous implementation. Running on a NVIDIA Tesla S1070, their implementation achieves a speed up of nearly 30 to 50 times over the serial implementation using the Boost C++ graph library on an Intel Core 2 Quad, Q6600, 2.4 GHz processor.

Rostrup et al.[12] introduce a data-parallel adaptation of Kruskal's MST algorithm that uses Borůvka's algorithm to solve subproblems in parallel on the GPU. Tests on random and real-world graphs with up to 25 million vertices and 240 million edges on an NVIDIA Tesla T10 GPU show that their method can process graphs 4X larger and up to 10X faster than was possible with Vineet et al.'s Borůvka's MST algorithm for the GPU. Their use of sort and split primitives also show a possible way to partition the graph into several subgraphs.

Graph Matching A matching or independent edge set in a graph is a set of edges without common vertices. Finding a maximal matching of a graph has applications in bioinformatics and can be used to solve other graph problems such as vertex coloring. Auer and Bisseling[4] give a fine-grained shared-memory parallel algorithm on GPU for greedy undirected graph matching. Running on an NVIDIA Tesla C2050 GPU, on large-scale graphs with millions of vertices, the algorithm achieves a speedup factor of 6.8 over the serial implementation on Intel Xeon E5620 processors with hyperthreading.

Graph Partition A graph cut performs a partition of the vertices of a graph into two disjoint subsets. It is an algorithm which has found several applications in computer vision. For distributed large-scale graph processing, it is also an important graph partition algorithm. Vineet and Narayanan[16] propose a GPU push-relabel maxflow/mincut algorithm. Their 90 graph cuts per second on 640×480 images is 10–12 times faster than the best sequential algorithm reported in 2008. However, their implementation is bounded by the global BFS height value relabel step during each iteration of the algorithm. The global BFS step makes it difficult to expand the algorithm to large-scale graphs while maintaining the same performance. Also, current GPU graph partition algorithms are all 2-way cuts algorithms. They do not address the problem of partitioning the graph into multiple subgraphs, which is known as the minimum k-cut problem. The goal is to find a set of edges whose removal would partition the graph into k connected components. The minimum k-cut problem is NP-complete when k is part of the input. With a fixed k, the problem still has a complexity of $O(|V|^{k^2})$. Since the purpose of graph partitioning is load balancing and task scheduling, for static graphs that only need to run once, the algorithm can run on either the CPU or GPU. But an efficient parallel implementation of such an algorithm on a GPU would improve the performance if the graph is dynamic and the graph structure changes frequently in real time.

Note that there are several advanced topics related to large-scale graph processing on GPUs such as mining, clustering, querying and pagerank computing, etc. Since our goal here is to build a GPU large-scale graph processing library containing basic primitives required by more advanced applications, we will not talk about these advanced topics for now and will continue to talk about the issues of current GPU graph algorithms.

4 Issues and Challenges

4.1 Issues

Although there have been several large-scale graph processing algorithms on GPU, we see two major issues which all current GPU graph algorithms fail to consider. First, all current GPU graph algorithms view vertices and edges in the graph as simple nodes, ignoring the possible rich information associated with them in real-world graphs. As we expect our use cases of interest will have complex information on edges and vertices, we need to address efficient storage, access, and updates of this information during graph processing. Second, all current GPU graph algorithms assume graphs to be static. However, several real-world large-scale graph use cases require dynamic graphs, where The topology of the graphs themselves and information on their vertices and edges can change in real-time. Robust graph algorithms should be aware of these real-time modifications and be able to respond to them and adjust the result accordingly.

4.2 Challenges

As the size of graph we want to process increases to petascale and beyond, there is a growing need to expand current parallel graph algorithms running on single GPU to GPU clusters on multi-node. This poses several challenges. First, multi-GPU communication is difficult as currently GPUs still cannot source or sink network I/O; thus supporting dynamic and efficient communication across many GPUs is hard. Second, GPUs do not have inherent out-of-core support and virtual memory. Finally, most graph algorithms have global dependencies between vertices, thus finding graph partitioning strategy for implementating algorithms on GPU cluster is difficult.

Stuart and Owens point out that for GPU cluster applications, the keys to parallel efficiency are to reduce communication times as much as possible and to overlap communication with computation,[14] because in such a scenario communication is almost always the bottleneck and GPU computation is relatively cheap. In their GPU-MapReduce library (GPMR) for large datasets running on GPU cluster, they partition key-value pairs into chunks and bin them to different GPUs. While one chunk is being processed, another chunk can be simultaneously streamed to or from the GPU. They also have a single module tracking the per-GPU work in a dynamic queue. If one GPU finishes its work in its local queue and other GPUs have much more work to do, they shift chunks between the local queues.

This method provides us a good example of assigning workloads and data to machines within a GPU cluster and keeping the load on every node balanced. However, for large-scale graph processing, finding a way to partition the graph data into parts with small dependency on each other is still a challenge, though we expect that the same methods used for CPU partitioning will be immediately applicable. Even on one node, when the size of graph data exceeds the limit of GPU memory, arranging data transfers to maintain GPU utilization and high throughput is also a challenging problem.

The other major challenge with out-of-core graph processing compared to MapReduce is the predictability of MapReduce accesses vs. the non-predictable access patterns of a graph, where the next chunk to be accessed is dynamically determined. It appears to be likely that an aggressively prefetching memory manager will be necessary to keep the GPU full of work. This is a significant challenge as to the best of our knowledge, such a system has not been built in the context of GPUs.

Other challenges include building data structures for efficient storage, access, and updates of information on vertices and edges, and designing robust algorithms for both static and dynamic graphs.

5 Conclusion

Recent work on GPU graph algorithms shows that GPUs are well-suited for doing graph processing and can achieve high levels of performance on a broad range of graphs. However, most current GPU graph algorithms lack both multi-node scalability and out-of-core support, and run on simple data rather than needing to access and update rich information on vertices and edges and dynamically updating graphs.

Recently, there are several new features of NVIDIA's "Kepler" family of GPUs that could benefit this project. One feature is dynamic parallelism, which enables a CUDA kernel to create and synchronize new nested work. This will improve the load-balancing of several graph algorithms. Another feature is GPUDirect, which enables GPUs within a single computer, or GPUs in different servers located across a network, to directly exchange data without needing to go to CPU/system memory. This will improve the cross-node communication efficiency, which often serves as the bottleneck of multi-node applications. Recently, several major chip manufacturers have been designing hybrid single-chip CPU/GPU architectures. The advent of Intel's Many Integrated Core (MIC), AMD's Accelerated Processing Units (APUs), and NVIDIA's Denver Architecture will reduce the cost of data sharing between CPU and GPU and allow more efficient heterogeneous algorithms.

Trends and looking forward The last five years of the GPU has seen the advent of a full programmable interface and a rapid growth in both the software capabilities and hardware features available to the programmers. The GPU now has full support for double-precision computation, ECC memory, and hardware caches.

The most significant successes on GPUs have led to libraries (such as NVIDIA's CUBLAS) that are optimized and offer significant performance gains. More recently, more irregular computation patterns have led to successes as well; the timeframe of NVIDIA's CUSPARSE library is instructive, with its core data structure introduced at Supercomputing 2009[2] and CUSPARSE's release roughly a year later (August 2010). Generally advances in data structures have been software ones, with more complex data structures developed each year on GPUs.

Graph processing on GPUs offers several challenges that make it more complex than any existing widely-used libraries. Compared to the implementations in these libraries, graph algorithms are substantially more complex and wide-ranging and have more irregular and complex access patterns. Existing libraries handle neither out-of-core nor multi-node computation.

From a hardware perspective, we see several significant challenges. The GPU is far from the CPU in terms of bandwidth and latency, and as with all GPU algorithms, it is necessary to do ample work on the GPU to mitigate the cost of transfer. The size of the largest available GPU memory system (6 GB) is much smaller than a high-end CPU server's, which greatly increases the need for locality in a large graph implementation. The GPU also lacks local control for tasks such as initiating data transfer to or from the CPU and launching its own work, though this last point is the subject of NVIDIA's new "dynamic parallelism" feature; nonetheless it is expected that the initial implementation of this feature will offer no performance advantages but instead is merely a proof of concept. Nonetheless we expect over the lifetime of this grant, newer GPUs (later Keplers and eventually Maxwell) will allow dynamic parallelism to be a first-class component in the GPU toolbox.

The other emerging hardware trend is single-chip heterogeneous devices, notably Intel's Ivy Bridge with a powerful CPU and a fairly strong GPU, though one only programmable with OpenCL. Perhaps more interesting for us is NVIDIA's Project Denver, likely a modest ARM CPU and a beefy GPU, but this chip has not yet been announced much less shipped. One of the important outcomes of this project will be to understand the cost of a high-latency, low-bandwidth connection between CPUs and GPUs, and if this cost is too high, then a Denver machine may be the best fit for this applications domain.

On the software side, the key concepts are locality, divergence, and efficient algorithms. For locality, we must take advantage of GPU caches and programmer-managed shared memories as much as possible, and must partition graphs efficiently both across nodes as well as across the CPU-GPU boundary to gain maximum efficiency. The techniques will not be radically different than what we'd see with multi-node CPU implementations, but in general we will be more willing to trade extra computation for more locality. For divergence, arranging both memory accesses as well as SIMD execution to minimize divergence is crucial for efficiency, as graph algorithms can easily suffer from divergence in either case. And efficient algorithms are certainly critical for our overall performance. Merrill's recent work showed the benefits of both data layouts that minimize divergence as well as superior algorithms that attained better asymptotic complexity. He also eschewed atomic accesses in favor of scan-based primitives and his philosophy there will certainly influence ours as well.

Depending on DARPA's applications of interest, it may be desirable and possible to pursue a ranking on the Graph500 list. GPU machines currently do not appear on this list simply because the efficient primitives have not been developed or have not been brought to Graph500 applications. (Note machine #10 on the list is from the excellent GPU team at Tokyo Tech with its high-performance and influential Tsubame machine; yet the entry on the list only uses Tsubame's CPUs.)

Bibliography

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 46:1–46:11, Washington, DC, USA, November 2010. IEEE Computer Society.

[2] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, pages 18:1–18:11, November 2009.

[3] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, August 1990.

[4] B. O. Fagginger Auer and R. H. Bisseling. A GPU algorithm for greedy graph matching. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge II*, pages 108–119. Springer-Verlag, Berlin, Heidelberg, May 2012.

[5] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, December 2007. Springer-Verlag.

[6] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, INDIA, 2009.

[7] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, February 2011. ACM.

[8] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, Washington, DC, USA, October 2011. IEEE Computer Society.

[9] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C. Hart. Edge v. node parallelism for graph centrality metrics. In Wen-mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 2, pages 15–28. Morgan Kaufmann, October 2011.

[10] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, June 2008. Eurographics Association.

[11] Duane Merrill, Michael Garland, and Andrew 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, February 2012. ACM.

[12] Scott Rostrup, Shweta Srivastava, and Kishore Singhal. Fast and memory-efficient minimum spanning tree on the GPU. October 2011.

[13] Jyothish Soman, Kothapalli Kishore, and P J Narayanan. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010*, pages 1–8, April 2010.

[14] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 1068–1079, May 2011.

[15] Vibhav Vineet, Pawan Harish, Suryakant 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*, HPG '09, pages 167–171, New York, NY, USA, June 2009. ACM.

[16] Vibhav Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08.*, pages 1–8, June 2008.