An Energy-Efficient Abstraction for Simultaneous Breadth-First Searches

Adam McLaughlin Jason Riedy David A. Bader
Georgia Institute of Technology
Atlanta, GA, USA

***Abstract*—Optimized GPU kernels are sufficiently complicated to write that they often are specialized to specific input data, target architectures, or applications. This paper presents a multi-search abstraction for computing multiple breadth-first searches in parallel and demonstrates a high-performance, general implementation. Our abstraction removes the burden of orchestrating graph traversal from the user while providing high performance and low energy usage, an often overlooked component of algorithm design. Energy consumption has become a first-class hardware design constraint for both massive and embedded computing platforms.**

Our abstraction can be applied to such problems as the all-pairs shortest-path problem, community detection, reachability querying, and others. To map graph traversal efficiently to NVIDIA GPUs, our hybrid implementation chooses between processing active vertices with a single thread or an entire warp based on vertex outdegree. For a set of twelve varied graphs, the implementation of our abstraction saves 42% time and 62% energy on average compared to representative implementations of specific applications from existing literature.

I. Introduction

Graphs can represent diverse sets of data from social networks [1] to the structure of computer programs [14]. Problems in areas such as urban planning [25] and epidemiology [17] are well-expressed by graphs and solved by different graph traversal algorithms. The applications often use large data sets relative to the platform and can leverage massive parallelism and memory bandwidth in GPUs for efficient solution.

The difficulty of programming GPU kernels and the immature state of GPU software has made it difficult for end-users to leverage the contributions of the domain experts who spend months of time optimizing GPU applications. GPU kernels are typically written and manually optimized for peak performance on a particular architecture, input data set, or application. Furthermore, relatively little work has been done to study the energy consumption of GPU algorithms. A lack of Dennard scaling and the era of dark silicon [7] imply that knowledge regarding the energy consumption of algorithmic choices is more important than ever.

This paper addresses issues in abstraction and efficiency with an abstraction for solving simultaneous graph traversals on the GPU that allows energy- and time-efficient general implementations. The abstraction itself is simple. Users only write functions for parts of the graph traversal relevant to their target application. Error prone and performance-sensitive details of executing parallel graph traversals on the GPU is buried in the general implementation, allowing users to focus on application-specific functionality. We evaluate our implementation against a set of diverse graphs, ensuring that the performance of our abstraction is not specialized to certain classes of graphs. Finally, we show that performance efficiency translates to energy efficiency.

In summary, we present the following contributions:

- We present an abstraction for executing simultaneous graph traversals on the GPU that permits a hybrid implementation. Vertices with sufficiently high outdegree are cooperatively processed by an entire warp whereas vertices with fewer neighbors are handled by a single thread.
- On NVIDIA GPUs, our implementation maximizes warp utilization and uses dynamic scheduling of warps to tasks to load-balance warps. We show that the additional performance efficiency reduces energy requirements.
- From web-crawls to road networks, we show that our abstraction achieves better performance and energy-efficiency than existing methods in addition to being more general. Our implementation saves 42% time and 62% energy on average over an oracle that chooses the best existing method for the graphs we studied.

II. Background

The computational throughput and memory bandwidth of GPUs provides a significant advantage over conventional CPU architectures in terms of performance and energy-efficiency [20]. Unfortunately, major efforts spent in the development of GPU algorithms that achieve high processor utilization see little to no reuse. Programming abstractions not only allow for more modular code but also make it easier for users to reason about the problems they are trying to solve rather than the details of parallel algorithm design or hardware. The remainder of this section presents the *multi-search* abstraction, a simple abstraction we employ for the execution of simultaneous breadth-first searches on sparse data sets.

A. The Multi-Search Abstraction

A number of existing libraries provide abstractions in order to simplify the development of parallel graph algorithms without sacrificing performance [1], [24], [26], [27]. These libraries typically use traversal-based abstractions that handle the performance-sensitive steps of a breath-first search, such as gathering neighbors and appropriately partitioning work to threads. Users of these abstractions are only required to implement a small number of functions that are specific to the problem that they are trying to solve. These functions typically handle what data structures need to be updated when vertices

are visited as well as the initialization and termination of the algorithm.

Inspired by these techniques and findings from previous work of our own, we generalize these traversal-based abstractions in the event that many such breadth-first searches are required by the user. The *multi-search* abstraction fits any problem that can execute independent breadth-first searches. The semantics of each search are the same with the exception that the searches start from different sources and write their own output. Examples of classical graph algorithms that fit this abstraction well are the All-Pairs Shortest Path (APSP) Problem, Reachability Querying, Betweenness Centrality, and Diameter Computation. Although traversal-based approaches can be applied to all of these problems, existing frameworks neglect the available coarse-grained parallelism that these problems provide and thus miss out on opportunities for performance improvements, particularly for high-diameter graphs.

Finally, there have been a number of studies on the power consumption and energy-efficiency of GPU applications. McLaughlin *et al.* present a study on GPU optimizations for static and dynamic betweenness centrality that showed an 83% average reduction in energy-to-solution over prior techniques [20]. Nagasaka *et al.* use a statistical approach to model the power consumption of GPU kernels using hardware performance counters [23]. Hong and Kim develop an integrated power and performance model for GPU kernels and show potential energy savings for memory-bound applications [11].

B. Related Work

Ligra [26], Galois [24], GraphLab [8], the Parallel Boost Graph Library (BGL) [9], and Green Marl [10] are frameworks that all provide CPU-based abstractions for graph analysis. GraphLab takes a disk-based approach, improving upon distributed frameworks such as Pregel [18]. Green-Marl takes a domain-specific language approach to graph processing by applying compiler optimizations that could not be applied to more general purpose programs [10]. Galois and Ligra are shared memory CPU approaches to processing large graphs in memory. Galois uses internal parallel data structures to asynchronously process worklists while Ligra uses a traversal-based abstraction that internally uses a hybrid method of graph traversal based on the density of the graph.

The GraphBLAS provides a set of primitives for graph processing in the context of linear algebra [13]. Users of the GraphBLAS define a semiring on which to perform sparse matrix products. For instance, the APSP problem can be solved on the tropical (min,+) semiring. In contrast, users of our system instead define callback functions that are invoked when vertices are visited.

GPU efforts in the realm of graph analysis have mostly focused on manual, monolithic implementations of specific algorithms [19], [21] although a number of frameworks have been proposed in recent literature [4], [27], [28]. Medusa was the first such approach, providing APIs that can act on edges and vertices [28]. The Gunrock library from Wang *et al.* improves upon this work with load-balancing techniques that significantly improve performance [27]. GasCL is an OpenCL graph framework that uses GraphLab's Gather-Apply-Scatter (GAS) abstraction [4]. Finally, the CUSP library focuses linear algebraic implementations of algorithms that operate on sparse data sets [5].

III. Methodology

Breadth-First Searches (BFSs) consist of a number of search iterations, beginning with the source vertex of the search. Each iteration explores the unvisited neighbors discovered by the previous iteration. We define a *vertex frontier* as the set of vertices to be explored during a specified iteration of a breadth-first search. Users of our abstraction define the set of vertices in which traversals are to be enacted from as a small number of functions:

- `init()`: Initialize data structures at the beginning of program execution.
- `prior()`: Handle any computation that may occur just prior to a search iteration.
- `visitVertex()`: When an edge (u, v) is traversed from source i, update the appropriate data structures in terms of u, v, and i.
- `post()`: Handle any computation that may occur at the end of a search iteration.
- `finalize()`: Handle any computation that may occur after all the searches have completed.

These functions are typically short and performance-insensitive. When writing these functions, users will have to be aware that synchronization will sometimes be necessary to avoid race conditions, a consequence that has been observed in existing frameworks [26], [27].

A. Implementation

Prior literature has presented a number of ways to solve the APSP problem (or other algorithms requiring its solution as a subroutine) on the GPU [12], [19]. A number of these implementations implicitly implement the multi-search abstraction to handle graph traversals for their particular use case. Our approach, in addition to being more general, improves upon the performance provided by these techniques through the use of warp-synchronous programming and a hierarchical queueing scheme. GPU computing involves distributing work to Cooperative Thread Arrays (CTAs) as well as to the threads within each CTA. Warp-synchronous programming leverages additional knowledge about how CTAs are mapped onto GPU hardware, namely the fact that each streaming multiprocessor of the GPU executes instructions in lockstep in groups of 32 threads (on current NVIDIA platforms) referred to as warps (using CUDA terminology). This execution model allows programmers to have warps cooperatively and asynchronously process sets of data from other warps, minimizing intrablock barriers and allowing dynamic scheduling of tasks. For instance, we assign each warp to a vertex in the active frontier of the BFS being processed by the SM to which the warp belongs. Rather than statically assigning warps $\{0 \dots k-1\}$ to elements $\{0, k, \dots\} \dots \{k-1, 2k-1, \dots\}$ of the frontier we use a dynamic scheduling policy that has each warp grab the next unprocessed queue element (using atomic operations to prevent race conditions). Although atomic operations have been shown to have significant performance impacts [22], this particular usage of them has a negligible effect on performance: the memory location under contention resides in shared memory

and a maximum of 32 threads (one thread per warp and a maximum of 32 warps per thread block) will ever try to increase the counter that points to the next queue element to be accessed. The use of dynamic scheduling is significant for scale-free graphs since idle warps can effectively steal work from the critical warp, providing better load-balancing between the warps of each SM. NVIDIA's Kepler (and newer) architectures provide the `__shfl()` intrinsic that allows for exchanging data between the threads of a warp without explicit synchronization. We also leverage the `__ldg()` intrinsic to leverage the GPU's read only data cache for certain loads from global memory. Of course, this is just one implementation for a set of hardware problems. Others are possible, and users of the abstraction won't need to change their code when implementations of the abstraction are improved.

B. Thresholding

Initial experiments with the above approach performed poorly on graphs containing many vertices of low outdegree. When an entire warp is assigned to a vertex with outdegree smaller than the architectural warp size, some threads within the warp will be idle. Hence, for vertices with sufficiently small outdegree, we assign a single thread per vertex to gather its neighbors. We use two distinct queues, one that consists of vertices with sufficiently small outdegree to be processed by a single thread (Q_{small}) and one that consists of vertices with a larger outdegree to be processed by an entire warp (Q). During each iteration of the search, the vertices in Q are processed by the warps in each SM followed by the vertices in Q_{small} by individual threads. In order to determine how small the outdegree of a vertex must be to be enqueued into Q_{small}, we use a threshold T. If the outdegree of a vertex is strictly less than T it will be enqueued into Q_{small}, else it will be enqueued into Q. The MultiThreaded Graph Library [3] uses a similar partitioning of vertices based on outdegree to avoid load imbalance on CPUs.

Algorithm 1 shows a pseudocode implementation our of multi-search abstraction. On the GPU we use a pair of arrays, Q_{curr} and Q_{next}, to represent a single queue. Q_{curr} contains the vertices in the active vertex frontier whose neighbors will be explored during the current iteration of the traversal. Q_{next} contains the unexplored neighbors of vertices in Q_{curr} that will be explored during the next iteration of a traversal. We use two such queues to implement our hybrid approach: Q and Q_{small}. The `for` loops on Lines 11 and 12 process vertices with large adjacency lists sequentially by assigning an entire warp of threads to gather the neighbors of each active vertex. In contract, the `for` loops on Lines 14 and 15 process vertices with small adjacency lists by assigning a single thread to sequentially gather the neighbors of each active vertex. The functions `init()`, `prior()`, `visitVertex()`, `post()`, and `finalize()` are user-defined functions to fit the higher-level application that they wish to target. These functions are typically concise and do not have significant impacts on performance. Depending on the user's application, some of these functions may even be left empty. For instance, to solve the APSP problem, only `init()` and `visitVertex()` need to be defined. Our implementation uses C++ templates to allow users to define these functions as functor objects, function pointers, or lambda expressions.

Algorithm 1: Pseudocode for the Warp-Thread Hybrid Multi-Search Abstraction Kernel

```
   // Loop across SMs
1  for i ∈ S do in parallel
2      if outdegree(i) < T then
3          Q_small_curr.enqueue(i)
4      else
5          Q_curr.enqueue(i)
6      init(i)
7      barrier()
8      while ¬Q_curr.empty() ∧ ¬Q_small_curr.empty() do
9          prior()
10         barrier()
11         for v ∈ Q_curr do
               // Loop across threads
12             for w ∈ neighbors(v) do in parallel
13                 visitVertex(i, v, w, Q_next, Q_small_next)
           // Loop across threads
14         for v ∈ Q_small_curr do in parallel
15             for w ∈ neighbors(v) do
16                 visitVertex(i, v, w, Q_next, Q_small_next)
17         move(Q_curr, Q_next)
18         move(Q_small_curr, Q_small_next)
19         barrier()
20         post()
21         barrier()
22     finalize(i)
```

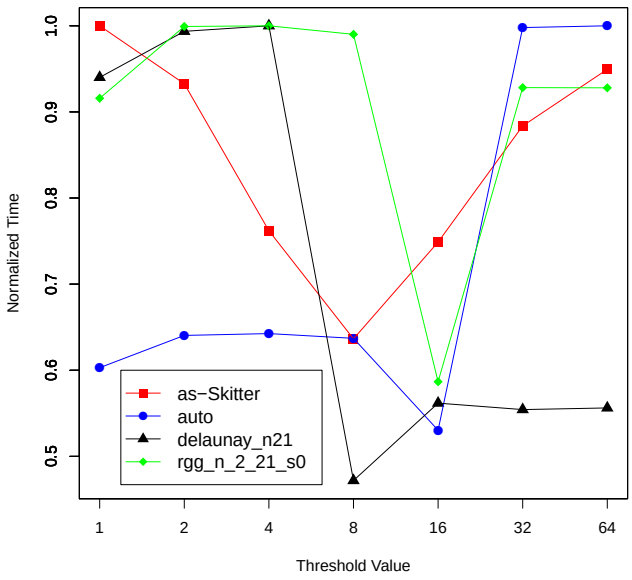

Fig. 1. Relative effect of the threshold parameter T on performance.

TABLE I. GRAPH DATASETS USED FOR THIS STUDY.

Graph	Nodes	Edges	Significance
333SP	3,712,815	22,217,266	Ferrari
adapative	6,815,744	27,248,640	Urban Sim.
as-Skitter	1,696,415	22,190,596	Internet
auto	448,695	6,629,222	Partitioning
delaunay_n21	2,097,152	12,582,816	Triangulation
ecology1	1,000,000	3,996,000	Gene Flow
hollywood-2009	1,139,905	115,031,232	Movie Actors
kron_g500-logn19	524,288	43,561,574	Kronecker
ldoor	952,203	45,570,272	Large Door
rgg_n_2_21_s0	2,097,152	28,975,990	Geometric
roadNet-CA	1,971,281	5,533,214	Intersections
thermal2	1,227,087	7,352,268	Diffusion

Figure 1 shows the relative improvement in performance for varying values of T for a few graphs. Note that when $T = 0$, all vertices are placed into Q_{large} and are thus processed by an entire warp. At the other extreme, when $T = \infty$, all vertices are placed into Q_{small} and are thus processed by a single thread. From Figure 1 we can see that $T = 8$ and $T = 16$ lead to the largest improvements in performance, depending on the particular input. When the threshold is too low, vertices with low outdegree are processed by an entire warp, leading to many idle threads within the warp. Conversely, when the threshold is too high, vertices with high outdegree are processed by a single thread when they instead supply enough parallelism for an entire warp of threads, leading to load imbalances from critical threads that have to traverse large adjacency lists. We consider using a histogram of vertex outdegree for the entire graph as a method of dynamically determining an appropriate value of T to be an interesting idea for future work. For our complete set of graphs using the value $T = 16$ worked best overall, and this value will be used to report results in the next section.

IV. EVALUATION

A. Experimental Setup

Table I shows the input graphs used to evaluate our techniques. These graphs are publicly available data sets from the DIMACS Challenge archives [2], the University of Florida Sparse Matrix Collection [6], and the Stanford Network Analysis Project (SNAP) [16]. We use both real world graphs, such as the *as-Skitter* Internet topology graph, and randomly generated graphs such as the *rgg_n_2_21_s0* geometric graph. These graphs have a broad range of diameters as well, which is important as graph diameter has previously been shown to significantly impact performance [20]. Graphs such as *delaunay_n21* have a high diameter, which leads to small vertex frontiers and many search iterations to completely traverse the graph. In contrast, graphs such as *hollywood-2008* have a low diameter, which leads to the majority of vertices being explored in a single search iteration and typically fewer than 10 search iterations to completely traverse the graph.

Code was written in CUDA C++ using the CUDA 7.0 toolkit. For timing experiments we use an NVIDIA GTX GeForce Titan GPU; since energy measurement via the NVIDIA Management Library (NVML) can only be used on Tesla GPUs, we use an NVIDIA Tesla K40c for experiments involving measurements of power and energy. Both the Titan and K40 GPUs are based on the "Kepler" architecture, have compute capability 3.5, and a peak theoretical memory bandwidth of 288.4 GB/s. The Titan has 14 SMs, 6 GB of global memory, and a base clock frequency of 837 MHz. The K40 has 15 SMs, 12 GB of global memory, a base clock frequency of 745 MHz, and a TDP of 245 W.

Using NVML and C++11 futures, we spawn off a CPU thread to measure power asynchronously as GPU kernels of interest are launched[1]. We sample the power of the GPU once every ten milliseconds with a call to `nvmlDeviceGetPowerUsage()`, which will only work for Tesla class GPUs (hence our use of the K40). Using these samples we record the numerical integration of the sampled power consumption for the lifespan on a kernel, where one kernel launch is all that is necessary for all 1024 graph traversals for a given data set.

We evaluate our approach by comparing it to other GPU methods used to solve the APSP problem (or problems that it builds upon). We choose a value of $k = 1024$ source vertices to perform graph traversals from in order to keep execution times reasonable. For graphs that compromise of one large connected component that contains greater than 90% of all vertices in the graph (such as many real world graphs [15]), the time to execute a graph traversal varies minimally from one source vertex to another. Hence, the conclusions we draw from our use of a subset of source vertices can be confidently applied to the computation of the entire APSP problem. The methods we compare to are as follows:

- `Edge-parallel`: Assigns a thread to every edge of the graph for every search iteration, regardless of whether or not the endpoints of that edge are vertices in the active frontier. This approach is most effective when many vertices belong to the active frontier [12].
- `Work-efficient`: Assigns a thread to every vertex in the active frontier [19]. When $T = \infty$, our hybrid approach simplifies to this approach.
- `Oracle`: A pseudo-hybrid approach that chooses between `Edge-parallel` and `Work-efficient`, depending on whichever method is better for the particular input graph being analyzed. We present this result as a proxy for the hybrid method used to compute Betweenness Centrality in [19].
- `Warp-based`: Our method that assigns a warp to every vertex in the active frontier. Threads within the warp process consecutive outgoing edges from the active vertex. When $T = 0$, our hybrid approach simplifies to this approach.
- `Warp-thread Hybrid`: Our method that uses two queues, one containing elements to be processed by a single thread and the other containing elements to be processed by an entire warp. When the outdegree of a vertex is less than T, we use a single thread to collect its neighbors. We use a static value of $T = 16$ for all experiments.

B. Experimental Results

Table II shows the time required to execute all 1024 graph traversals (in seconds) on the GeForce GTX Titan using

[1]Source can be found at https://github.com/Adam27X/graph-utils/

TABLE II. TIMINGS FOR VARIOUS METHODS OF GRAPH TRAVERSAL IN SECONDS.

Graph	Edge-parallel	Work-efficient	Oracle	Warp-based	Warp-thread Hybrid	Savings of Hybrid over Oracle
333SP	1279.5	47.8	47.8	68.0	**32.1**	33%
adapative	7704.7	54.8	54.8	183.6	**42.8**	22%
as-Skitter	30.7	27.8	27.8	12.9	**9.66**	65%
auto	21.6	15.6	15.6	5.48	**4.82**	69%
delaunay_n21	436.8	23.3	23.3	25.1	**15.0**	36%
ecology1	426.9	8.86	8.86	29.1	**6.51**	27%
hollywood-2009	81.6	145.8	81.6	21.3	**20.4**	75%
kron_g500-logn19	35.4	55.1	35.4	**16.4**	17.1	52%
ldoor	434.0	**35.4**	**35.4**	35.7	36.5	-3%
rgg_n_2_21_s0	1824.9	67.0	67.0	37.0	**23.7**	65%
roadNet-CA	183.8	12.3	12.3	15.0	**9.15**	26%
thermal2	650.4	11.7	11.7	19.4	**7.71**	34%

the methods explained in the previous section. Although our `Warp-thread Hybrid` approach is best for 10 of the 12 graphs tested, the magnitude by which it is best is dependent on the threshold parameter T. For instance, our non-threshold based `Warp-based` approach does better than our hybrid approach for *kron_g500-logn19* and *ldoor*; however, simply setting $T = 0$ for such graphs would solve this issue. Hence, for future work we will consider an approach that determines the appropriate value of T for a given graph based on the distribution of the outdegree of its vertices.

Interestingly, for *ldoor* the `Work-efficient` approach is slightly better than both our `Warp-based` and `Warp-thread Hybrid` approaches. Again, setting the threshold dynamically (to $T = \infty$ in this case) could solve this problem. The maximum outdegree for any vertex of *ldoor* is 76 and 99.8% of vertices in *ldoor* have an outdegree of 63 or less. Since the current warp size of NVIDIA GPUs is 32 threads, this means any vertex assigned to a warp will process at most three (and very often only two) edges, which doesn't provide sufficient instruction level parallelism to each thread. Hence, assigning active vertices to threads for this graph results in marginally better performance. Overall, our `Warp-thread Hybrid` approach improves upon that of the `Oracle` by 42%. In practice, the implementation of such an oracle would have some overhead associated with choosing between the `Edge-parallel` and `Work-efficient` methods, making this result a lower bound for the improvement of our approach in practice.

In addition to being faster than existing approaches, our `Warp-based` and `Warp-thread Hybrid` approaches tend to consume less instantaneous power. The `Edge-parallel` approach is energy-inefficient because threads are assigned to edges that don't necessarily belong to the active frontier and the `Work-efficient` approach is energy-inefficient because threads have an imbalanced amount of neighbors to gather and hence cause other threads within the same warp to stall and wait for whichever thread belongs to the warp's critical path. Table III shows the energy required to execute all 1024 graph traversals (in Joules) on the Telsa K40c using these approaches. For almost every graph we tested, the energy savings of our techniques are greater than the savings in time shown in Table II, confirming the above analysis regarding the energy-efficiencies of prior work. Even though our performance results were slightly slower than the `Oracle` for *ldoor*, our energy usage is much better due to our efficient warp utilization. For *as-Skitter*, *hollywood-2009*, and *roadNet-CA*, we can see that there are interesting trade-offs between performance and energy consumption; although our hybrid approach provides the best performance for each of these graphs, our warp-based approach provides better energy-efficiency. The choice of the threshold parameter T again plays a considerable role for these trade-offs. Overall, our `Warp-thread Hybrid` approach saves 62% energy on average compared to the `Oracle` approach and we again note that this figure neglects the energy cost of choosing a preferential distribution of threads to work that would be required by the implementation of such an oracle.

V. CONCLUSION

This paper explored the performance and energy characteristics for multi-search, a simple GPU abstraction to execute simultaneous breath-first searches. Our initial approach of assigning warps to cooperatively gather neighbors from vertices in the active vertex frontier worked well for low-diameter graphs, but suffered from warp occupancy and utilization for high-diameter graphs. To account for this deficiency, we presented a hybrid approach that assigns a single thread to gather neighbors of vertices with sufficiently small outdegree. Across a varied set of real-world and synthetic graphs, our hybrid approach saves 42% time and 62% energy on average over an oracle that is an idealized representation of previous literature.

In addition to implementing a dynamic version of our hybrid approach we plan to consider performance and programmability tradeoffs to obtain a desirable level of abstraction for future work. The automation of GPU kernel optimization is another area of work that we consider to be important. Such automation can be achieved through compiler optimizations, runtime libraries, and even domain-specific languages. Finally, the management of power for accelerators, heterogeneous processors, and distributed systems is a growing area of promising research.

ACKNOWLEDGMENT

The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no

TABLE III. ENERGY CONSUMPTION FOR VARIOUS METHODS OF GRAPH TRAVERSAL IN JOULES.

Graph	Edge-parallel	Work-efficient	Oracle	Warp-based	Warp-thread Hybrid	Savings of Hybrid over Oracle
333SP	36,676	2,220	2,220	2,880	**1,083**	51%
adapative	316,299	8,551	8,551	6,790	**1,433**	83%
as-Skitter	5,004	4,688	4,688	**1,053**	1,606	66%
auto	635	1,164	635	539	**159**	75%
delaunay_n21	12,403	1,425	1,425	1,495	**508**	64%
ecology1	12,309	856	856	1,642	**219**	74%
hollywood-2009	11,546	25,577	11,546	**1355**	3,342	71%
kron_g500-logn19	1,180	2,492	1,180	1,195	570	52%
ldoor	12,527	1,829	1,829	1,855	**1,210**	34%
rgg_n_2_21_s0	50,525	2,851	2,851	1,908	**759**	73%
roadNet-CA	26,950	2,050	2,050	**1,144**	1,539	25%
thermal2	17,566	996	996	1,311	**259**	74%

official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." This work was also partially sponsored by NSF Grant ACI-1339745 (XScala). Finally, we would like to thank NVIDIA Corporation for their donation of GeForce GTX Titan and Telsa K40 GPUs.

REFERENCES

[1] D. A. Bader and K. Madduri, "SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[2] D. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, "Benchmarking for Graph Clustering and Partitioning," in *Encyclopedia of Social Network Analysis and Mining*, R. Alhajj and J. Rokne, Eds. Springer New York, 2014, pp. 73–82.

[3] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1–14.

[4] S. Che, "Gascl: A vertex-centric graph model for gpus," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[5] S. Dalton, N. Bell, and L. Olson, "Optimizing sparse matrix-matrix multiplication for the gpu," 2013.

[6] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[7] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." *OSDI*, vol. 12, no. 1, p. 2, 2012.

[9] D. Gregor and A. Lumsdaine, "The Parallel BGL: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[10] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 349–362.

[11] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 280–289.

[12] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, "Edge v. Node Parallelism for Graph Centrality Metrics," *GPU Computing Gems*, vol. 2, pp. 15–30, 2011.

[13] J. Kepner, D. A. Bader, A. Buluc, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, Matrices, and the GraphBLAS: Seven Good Reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[15] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 2, 2007.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[17] F. Liljeros, C. R. Edling, L. A. Amaral, H. E. Stanley, and Y. Aberg, "The Web of Human Sexual Contacts," *Nature*, vol. 411, no. 6840, pp. 907–908, 2001.

[18] 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*. ACM, 2010, pp. 135–146.

[19] A. McLaughlin and D. A. Bader, "Scalable and High Performance Betweenness Centrality on the GPU," in *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.

[20] A. McLaughlin, J. Riedy, and D. A. Bader, "Optimizing Energy Consumption and Parallel Performance for Static and Dynamic Betweenness Centrality using GPUs," in *Eighteenth IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

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

[22] D. G. Merrill, III, "Allocation-oriented algorithm design with application to gpu computing," Ph.D. dissertation, Charlottesville, VA, USA, 2011.

[23] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of gpu kernels using performance counters," in *Green Computing Conference, 2010 International*. IEEE, 2010, pp. 115–122.

[24] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 456–471.

[25] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and R. Messora, "Street Centrality and Densities of Retail and Services in Bologna, Italy," *Environment and Planning B: Planning and design*, vol. 36, no. 3, pp. 450–465, 2009.

[26] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146.

[27] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *CoRR*, vol. abs/1501.05387, no. 1501.05387v1, Jan. 2015.

[28] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," 2013.