Optimizing Energy Consumption and Parallel Performance for Static and Dynamic Betweenness Centrality using GPUs

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

***Abstract*—Applications of high-performance graph analysis range from computational biology to network security and even transportation. These applications often consider graphs under rapid change and are moving beyond HPC platforms into energy-constrained embedded systems. This paper optimizes one successful and demanding analysis kernel, betweenness centrality, for NVIDIA GPU accelerators in both environments. Our algorithm for static analysis is capable of exceeding 2 million traversed edges per second per watt (MTEPS/W). Optimizing the parallel algorithm and treating the dynamic problem directly achieves a 6.9× average speed-up and 83% average reduction in energy consumption.**

I. Introduction

Graphs are used to model the structure of the internet [11], interactions in social communties [19], and dynamic simulations of physical phenomena [21]. Many common graph problems have efficient sequential solutions but resist attempts at parallel efficiency. Increasingly parallel architectures and accelerators require new algorithms for both performance and power efficiency. The high memory bandwidth and power efficiency of Graphics Processing Units (GPUs) make them attractive to bandwidth-hungry graph algorithms, but mapping the analytics to GPU hardware is challenging.

Graph analysis algorithms often require fine-grained synchronization that limits parallelization. Some algorithms, like lexicographic depth-first search, are known to be P-complete and are inherently sequential [25]. Limited spatial locality and widely varying computational load also present challenges beyond those common in scientific computing or map-reduce-style data analysis. Maintaining analytics as new data streams into the graph without entirely recomputing results is another new challenge.

This paper tackles these challenges with the following contributions:

- We propose various parallel methods for calculating betweenness centrality (BC), a successful analytic that tracks the influence of vertices in a network. We consider both coarse-grained and fine-grained methods of parallelism.
- We compare static methods for re-computing BC scores to a natively dynamic method that updates BC scores. We show that most edge changes affect a surprisingly small portion of the graph and that asymptotically efficient algorithms are crucial to analyzing time-varying graphs.
- We present results comparing our methods to the state-of-the-art on embedded and HPC platforms considering both time and energy to solution. Our static implementation of the algorithm is capable of exceeding 2 MTEPS/W. Our dynamic implementation of the algorithm achieves greater than a 25× speedup over existing sequential methods on the CPU. On the GPU, our implementation achieves on average a 6.9× speedup and 83% reduction in energy consumption compared to a static recomputation.

II. Background

A. GPU Computing

Although GPUs are typically known for rendering computer graphics, the introduction of programming models such as CUDA and OpenCL have opened the computational power of the GPU to domains such as databases, electronic design automation, and biology [8], [18], [27]. GPUs have been successful in accelerating compute-bound applications that have regular structure and lots of floating point arithmetic [20]. Recent research also has shown successful acceleration of irregular and memory-bound applications that have randomized memory access patterns [7], [24].

GPUs are designed for highly parallel operation and dedicate transistors to arithmetic units rather than branch predictors or large caches. They leverage a single-instruction, multiple-thread (SIMT) programming model where consecutive threads execute the same instruction on different elements of data. A GPU consists of a number of streaming multiprocessors (SMs) that each execute threads in groups, known as warps on NVIDIA's GPUs. In the case of a branch instruction, the resulting paths of the branch are executed sequentially by predicated execution.

Programmers using NVIDIA's CUDA specify a number of *grid* and *block* dimensions for each kernel. These dimensions specify how many groups of threads are assigned to each SM and how many threads coexist within those groups. Programmers also manage *shared memory*, which is scratchpad storage assigned to each SM. Shared memory has much higher bandwidth than global memory but is smaller and hence harder to use in applications that require data scalability.

Compared to conventional CPUs, GPUs tend to consume more instantaneous power but provide significantly higher

978-1-4799-6233-4/14/$31.00

throughput which results in better overall energy efficency in terms of performance per Watt. For instance, all of the top 10 computers on the November 2013 Green500 list utilize GPU accelerators [14].

B. Betweenness Centrality

Centrality metrics are an important class of graph algorithms used in applications such as graph visualization [16], urban planning [6], and community detection [4]. Betweenness Centrality (BC) was a metric developed in the social sciences for tracking the control of information in communication networks [12]. Recently it has been used to determine influential members of social networks [10]. BC scores are obtained by calculating the ratio of the number of times a vertex is on a shortest path between pairs of other vertices to the total number of shortest paths between those vertices.

Let σ_{st} be the number of shortest paths between vertices s and t and let $\sigma_{st}(v)$ be the number of these paths that pass through a particular vertex v. The betweenness centrality of v can be defined in terms of these numbers as follows:

$$BC(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

The fastest known sequential algorithm for computing BC scores was developed by Brandes [5]. Rather than using the $O(n^3)$ Floyd-Warshall algorithm to solve the all-pairs shortest path (APSP) problem, Brandes derived a recursive relationship between vertices and their successors. The algorithm performs a breadth-first search traversal to solve the APSP problem and uses these results in a backward traversal on the graph referred to as the dependency accumulation to recursively obtain the centrality scores. Even with these improvements, the algorithm is computationally demanding as it requires $O(mn)$ time for unweighted graphs, where n is the number of vertices and m is the number of edges in the graph.

Several high-level strategies have been used to accelerate the computation of betweenness centrailty, such as approximation techniques [1], parallelism [22], and streaming [23]. The simplest method of approximating BC scores is to use a subset of the source vertices for the calculation. This step reduces the time complexity of the algorithm from $O(mn)$ to $O(mk)$ where k is the number of approximated source vertices. Essentially, the number of shortest paths from the k vertices to all vertices are found instead of the number of shortest paths between all pairs of vertices. Betweenness centrality lends itself well to parallelism since both coarse and fine-grained opportunities for parallelizing Brandes's algorithm exist. Coarse-grained parallelism involves assigning different source vertices to different threads or compute units. This assignment of work is embarassingly parallel since all source vertices can be handled independently. Fine-grained parallelism of BC assigns threads to cooperatively execute stages of the graph traversal needed for the shortest path calculation and dependency accumulation stages. Finally, several methods for incrementally updating centrality scores rather than recomputing them have been proposed in the literature [13]. Streaming methods are becoming increasingly important to analyze *dynamic* graphs that change over time. Typical network updates only affect a local region of the graph, making global recomputations wasteful in terms of both time and energy. Experimental results for both our static and dynamic implementations of Betweenness Centrality on the GPU can be found in Section V.

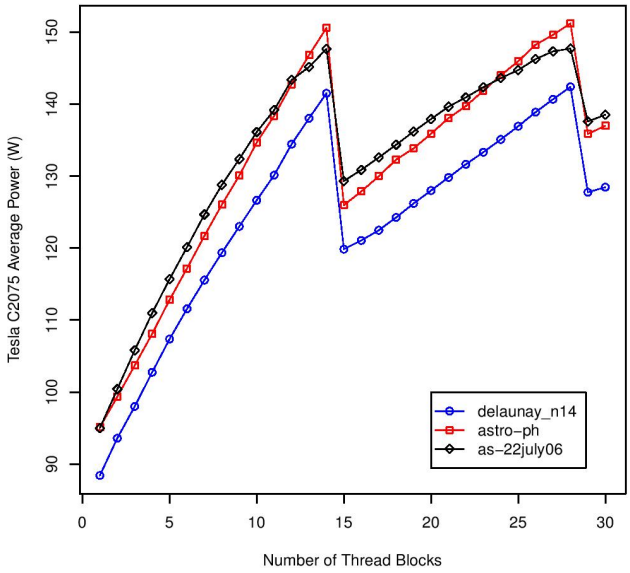

Fig. 1. Power consumed as a function of the number of thread blocks launched

III. Methodology

A. Coarse-grained Parallelism

The most important consideration for both our static and dynamic implementations of betweenness centrality is the decomposition of threads to units of work. Previous work investigating absolute performance showed that the number of thread blocks should be equivalent to the number of SMs for calculating betweenness centrality [17]. This also proves true for energy efficiency. Figure 1 shows how the average instantaneous power consumption of a Tesla C2075 GPU varies with thread blocks. Since the C2075 has 14 SMs, we can see that the most power is consumed when the number of thread blocks issued is a multiple of the number of SMs. Interestingly, it appears that when 15 blocks are issued, rather than scheduling one block to each SM with one block leftover, the hardware opts to issue two blocks to 7 of the SMs and one block to an 8th SM in an attempt to conserve power by idling the remaining 6 SMs. Noting the scale of the y-axis, assigning thread blocks to all of the SMs on the GPU requires less than twice as much power than using just one thread block. Since the performance of the algorithm scales linearly with the number of active SMs (because each SM can execute independently in parallel), assigning one thread block to each SM is clearly the most energy-efficient method of operation.

B. Fine-grained Parallelism

Each cooperative thread array (CTA), or thread block, of the GPU is assigned a root vertex to traverse from and perform shortest path and dependency calculations. This results in attributing that root vertex's impact on the BC scores. Once this process has been completed for all of the roots in the graph (or all of the roots to be approximated), the algorithm terminates. The threads within each CTA work together to traverse the graph and calculate shortest paths and dependencies in parallel.

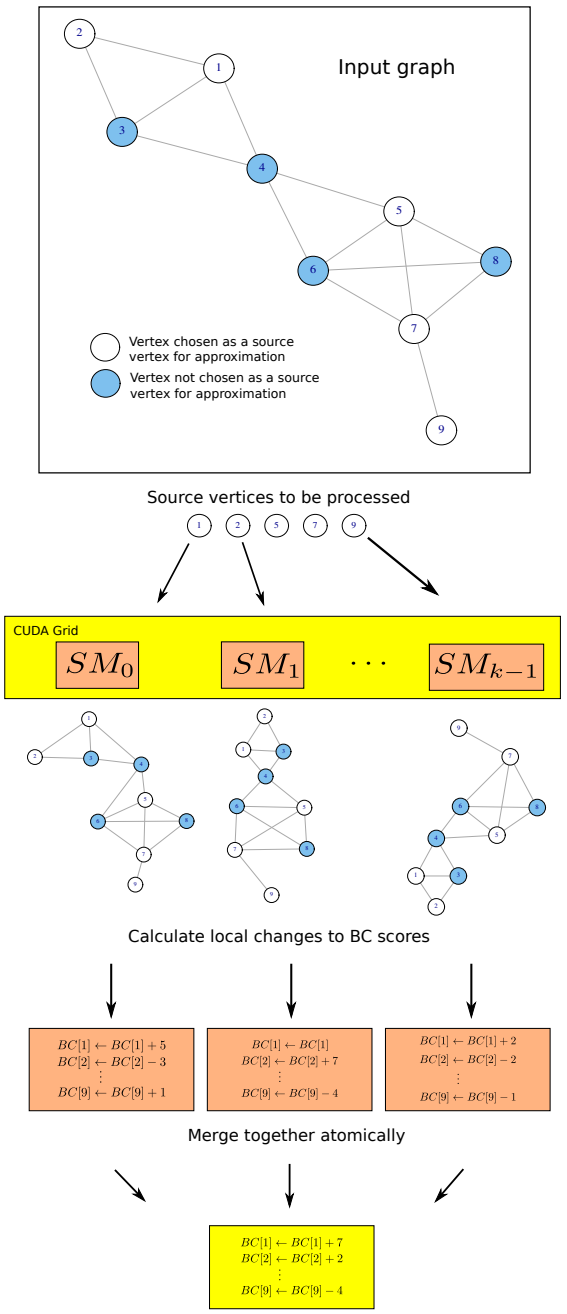

Fig. 2. Decomposition of work to parallel compute units

Figure 2 illustrates this process. Each root, or source vertex, to be processed is assigned to a CTA that is scheduled to one of the SMs on the GPU. The threads within this SM traverse the graph from the root, calculating local changes to the BC scores. Finally, each SM adds its changes to the global BC scores atomically. Since CTAs are executing independently they will not finish calculating their local scores simultaneously. Hence, the contention of resources for the atomic updates to the global BC scores is low.

One of the most significant factors in how fast the algorithm executes is the choice of graph traversal method. For a graph traversal at the level of a CTA for betweenness centrality, it has been shown that assigning threads to each edge rather than each vertex of the graph achieves greater memory throughput on the GPU [17]. Alternatively, the use of an explicit queue can obtain even better performance for especially sparse graphs, such as road networks, or for dynamic updates to the graph that touch only a local subset of vertices [23]. Section V explores how these methods of parallelism impact the power consumption of the GPU.

TABLE I. GPUs USED FOR THIS STUDY

GPU	*Tesla C2075*	*Tesla K40c*	*GT 640*
SMs	14	15	2
Memory (GB)	6	12	1
Frequency (GHz)	1.15	0.745	0.95
Compute Capability	2.0	3.5	3.5
TDP (W)	225	245	75

IV. Experimental Setup

Table I shows the various GPUs used for these experiments. The Tesla C2075 GPU is based on NVIDIA's "Fermi" architecture and cannot leverage the latest features of the CUDA programming model, such as Dynamic Parallelism; however, our implementations do not rely on such features. The Tesla K40c is NVIDIA's latest GPU designed specifically for HPC applications and is based on NVIDIA's "Kepler" architecture. These GPUs were designed with scientific computing in mind and have more memory than typical desktop GPUs. The GT 640 is a commodity GPU that is a part of the NVIDIA Kayla platform, an embedded system consisting of an NVIDIA Tegra 3 ARM Cortex A9 Quad-Core processor and the GT 640 GPU.

Algorithms were implemented in CUDA C++ using the CUDA 5.5 toolkit. Static computations were implemented to compute exact centrality scores whereas dynamic computations were implemented to also compute approximations to centrality scores using $k = 256$ randomly chosen roots as suggested by the DARPA SSCA benchmark suite [2]. We simulate dynamic graphs by randomly choosing 100 edges, removing them from the graph, and reinserting them sequentially, updating the BC scores after each insertion. This is the limit for low-latency applications that must respond to changes rapidly.

On the Kayla platform, power was measured using a Watts Up wall-plug meter, which measures system power. Since the entire computation is executed on the GPU, the CPU is idle and its power is constant and small enough to be neglected. Power was sampled at one second intervals and averaged over the lifespan of a kernel. Typical edge updates take a small number of seconds and the instantaneous power does not significantly change throughout a kernel execution. Power on the Tesla GPUs is measured directly using the NVIDIA Management Library (NVML). This library provides a C-based API for measuring power and temperature of Tesla GPUs. We sampled power at 10 ms intervals and report the average of the lifespan of a kernel.

Finally, Table II shows the graph datasets used for this study. These graphs were obtained from the DIMACS Challenge archives [3] and represent a diverse set of networks ranging from planar road maps (*luxembourg.osm*) to power-law graphs representing the structure of web domains (*eu-2005*).

TABLE II. GRAPH DATASETS USED FOR THIS STUDY

Graph	Nodes	Edges
as-22july06	22,963	48,436
astro-ph	16,706	121,251
caidaRouterLevel	192,244	609,066
coPapersCiteseer	434,102	16,036,720
delaunay_n12	4,096	12,264
delaunay_n14	16,384	49,122
delaunay_n20	1,048,576	3,145,686
eu-2005	862,664	16,138,468
kron_g500-logn16	55,321	2,456,071
kron_g500-logn19	524,288	21,780,787
luxembourg.osm	114,599	119,666
preferentialAttachment	100,000	499,985
smallworld	100,000	499,998

TABLE III. ENERGY-EFFICIENCY OF STATIC BC COMPUTATIONS ON THE GPU FOR VARIOUS CLASSES OF NETWORKS

Graph	Avg Power (W)	MTEPS/W
delaunay_n20	129.38	0.85
luxembourg.osm	95.41	0.35
preferentialAttachment	127.18	1.33
smallworld	127.10	2.54

V. EXPERIMENTAL RESULTS

A. Static Experiments

Since graph algorithms are memory bound the faster that they can traverse edges the faster they tend to execute. Analgous to FLOPS for compute bound applications is the notion of Traversed Edges per Second, or TEPS. For an instance of betweenness centrality, the number of TEPS is defined as follows [26]:

$$TEPS_{BC}(G,t) = \frac{mn}{t} \tag{2}$$

where n is the number of graph vertices, m is the number of graph edges, and t is the time in seconds. Defining a single work amount, here mn, regardless of the implementation is equivalent to defining the FLOPS for LU factorization as $2/3n^3$ regardless of the matrix arithmetic operations [9].

For the approximation of BC, n is replaced with k in defining TEPS. Table III shows the average power consumption and million of TEPS per W (MTEPS/W) for four different classes of graphs: meshes (*delaunay_n20*), road networks (*luxembourg.osm*), scale-free networks (*preferentialAttachment*), and networks with a diameter that is logarithmic in the number of vertices (*smallworld*). The TEPS/W metric is used to rank the energy-efficiency of graph processing systems for the Green Graph 500 [15]. These results were recorded using NVML and a Tesla K40c GPU. We can see that the *luxembourg.osm* road network consumed significantly less power on average than the other classes of graphs. Road networks tend to be extremely sparse and have very consistent degree distributions. In fact, no vertex (i.e. intersection) in this particular road network has more than 6 neighbors (i.e. incoming roads). As a result, each iteration of a breadth-first search over this graph results in a small amount of new vertices to be explored and consequently, few warps of execution per CTA and lower power consumption. Note that for all classes of graphs there isn't enough computation for the average power consumed to be anywhere near the TDP of the device.

TABLE IV. COMPARISON OF DYNAMIC BC COMPUTATIONS ON THE CPU AND GPU OF THE KAYLA PLATFORM

Graph	*delaunay_n12*	*kron_g500-logn16*
Solution Quality	Exact	Approx. ($k = 256$)
CPU Time (s)	35.44	33.79
GPU Time (s)	1.32	1.33
Speedup	***26.92***×	***25.39***×
Average CPU Energy (J)	914.35	875.08
Average GPU Energy (J)	42.64	43.79
Energy Savings	***95.3%***	***95.0%***
CPU MTEPS/W	0.05	0.72
GPU MTEPS/W	1.18	14.37

TABLE V. COMPARISON OF STATIC AND DYNAMIC BC COMPUTATIONS ON THE GPU OF THE KAYLA PLATFORM

Graph	*delaunay_n12*	*kron_g500-logn16*
Solution Quality	Exact	Approx. ($k = 256$)
Static Time (s)	12.63	5.63
Dynamic Time (s)	1.32	1.33
Speedup	***9.6***×	***4.2***×
Static Energy (J)	424	188
Dynamic Energy (J)	42.6	43.8
Energy Savings	***90.0%***	***76.7%***
Static MTEPS/W	0.12	3.34
Dynamic MTEPS/W	1.18	14.37

Table III shows that our algorithm is more power-efficient on scale-free and small-world graphs. For these graphs we use an edge-based graph traversal to maximize the memory throughput of the GPU rather than using an asymptotically optimal traversal algorithm. These graphs tend to have a smaller number of traversal iterations that each contain tens of thousands of edges to traverse in parallel. In contrast, networks with larger diameters tend to have hundreds of edges to traverse per iteration, making it more challenging to fully utilize the GPU.

B. Dynamic Experiments

For dynamic calculations we compare against two baselines. First we can compare CPU and GPU implementations of the dynamic BC algorithm to see the benefit of using a massively parallel architecture. Second we can compare the dynamic GPU approach to a static GPU approach to see the benefit of updating analytics rather than recomputing them.

Table IV compares using the CPU and GPU for computing BC scores dynamically. Note that the CPU algorithm is sequential. The times recorded represent the average time to update the BC scores for 100 edge insertions (one update occurs per edge insertion). Although the GPU requires slightly more instantaneous power than the CPU, we can see that the throughput provided by the GPU more than makes up for this additional power cost. The GPU implementation uses 19.69× less energy on average than the CPU for the two graphs above. Since these results were obtained on the Kayla platform, we had to restrict our analysis to significantly smaller data sets (and hence used approximation for the Kronecker *kron_g500-logn16* graph).

Using the same graphs, we compare static and dynamic methods for betweenness centrality in Table V. The static implementation used as a reference here is from Jia *et al.* [17]

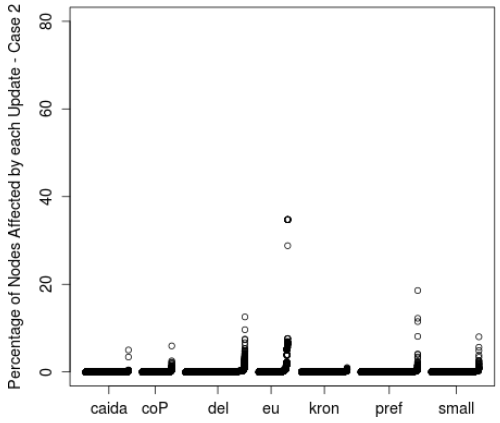

Fig. 3. Percentage of vertices touched by Case 2 scenarios

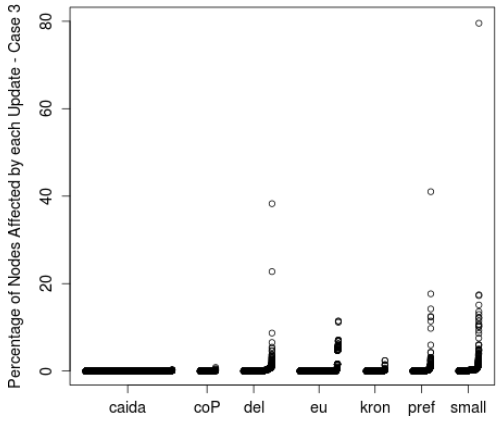

Fig. 4. Percentage of vertices touched by Case 3 scenarios

(previous state-of-the-art) and the dynamic implementation is our own. Note that this static implementation differs from the one used in Table III, which was our own implementation that improves upon the results from [17]. The times presented are again averaged over all 100 edge insertions. Although the time required for each update is highly dependent on the amount of work required by that update, even the slowest updates are faster than static recomputation. In addition to being faster than the static approach, the dynamic approach also consumes less power. The intuition behind this result is that a static computation of BC scores for the updated graph is an upper bound for the amount of work required by a dynamic update. Since the dynamic update only traverses edges that are affected by the update it avoids unwarranted accesses to memory. Overall, our dynamic method sees a 6.9× average speedup compared to a static recomputation for these two graphs. Dynamic updating consumes an average 83% less energy than static recomputation.

The insertion of an edge into the graph presents one of three possible scenarios from each root. The inserted edge can either connect vertices that are the same distance (Case 1), adjacent distances (Case 2), or non-adjacent distances (Case 3) from a given root. Case 1 insertion scenarios do not change BC scores whereas Case 2 and Case 3 insertion scenarios require additional computation to account for the newly inserted edge [13]. To quantify how much work is required by the dynamic algorithm for a typical edge insertion, we record the

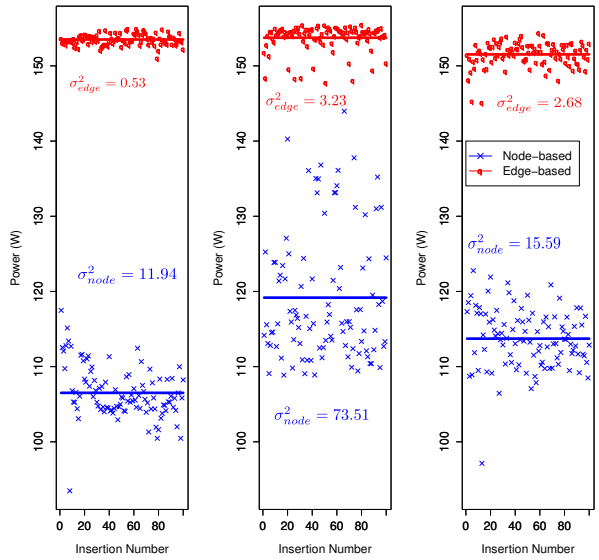

Fig. 5. Left: *preferentialAttachment* Middle: *kron_g500-logn19* Right: *smallworld*

percentage of vertices that are touched by the shortest path recalculations and dependency accumulations for each edge insertion. Figures 3 and 4 sort and display these percentages as a scatterplot for Case 2 and Case 3 insertion scenarios, respectively. Amazingly, a vast majority of edge insertions impact less than 1% of vertices in the graph. Out of the 62,844 Case 2 scenarios encountered, no more than approximately 35% of vertices were touched by any of them. Similarly, for Case 3, which tends to have more work as it pulls up vertices from further away from the root than Case 2 does, only three scenarios touched more than 30% of vertices in their respective graphs. This result implies that the use of asymptotically efficient algorithms is crucial to obtaining high performance for dynamic graph analytics.

To illustrate the effect of using a dynamic approach in terms of power, Figure 5 shows a scatter plot of the average power consumption during each edge update for two methods of parallelism for three graphs. The edge-based parallel method was introduced by Jia et al. [17] and assigns a thread to each edge of the graph to be inspected during each iteration of the graph traversal. The node-based parallel method instead uses an explicit queue to only traverse edges coming from vertices that are at the current depth of the graph traversal. The solid lines in the figure represent the average power consumption across all edge insertions. Since the edge-based parallel method checks every edge at every iteration of the search, it causes unecessary branching overhead and fetches to global memory. Since the edge approach does this unnecessary work regardless of where the edge is inserted into the graph the variance in power for the edge-based approach is small as the GPU consistently draws significant power. While this may normally be a sign that the processor is utilized in this case the processor is being fed superfluous instructions.

In contrast, the average power consumption for the node-based parallel method varies greatly with insertion. Intuitively, each edge insertion has some variable cost in terms of the portion of the graph that is affected by each update. Since the work done by the node-based method depends entirely on this

cost, the power consumed by the node-based method is also variable. Note that in all cases tested the edge-parallel method consumes more power than the node-parallel method.

Finally, the above results are consistent regardless of the graph tested. The left portion of Figure 5 shows results for a scale-free graph, the middle portion shows results for a Kronecker graph, and the right portion shows results for a small-world graph. In each case the power consumption for the node-based method is significantly smaller and more volatile than for of the edge-based method.

VI. Conclusions and Future Work

This paper presents performance and energy efficiency optimizations for static and dynamic betweenness centrality. Our static implementation of the algorithm is capable of exceeding 2 MTEPS/W. Our dynamic implementation of the algorithm achieves greater than a 25× speedup over existing sequential methods on the CPU and a 6.9× average speedup along with an 83% average reduction energy-to-solution compared to a static recomputation of the analytic on the GPU. Our methods have been shown to work well on both embedded systems such as the Kayla platform and HPC systems such as Tesla GPUs. Furthermore, our methods are easily scalable to multiple GPU nodes for even faster processing.

Both parallel optimization as well as dynamic updating prove important to reducing total energy consumption. Applying these techniques to other algorithms may drastically increase the range of applications for graph analysis. More in-depth models of concurrency and energy consumption can guide analytic development. With sufficient hardware flexibility and programming models to match, advanced analysis of dynamic graphs will move from machine rooms to embedded and hand-held devices.

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 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, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Algorithms and models for the web-graph*. Springer, 2007, pp. 124–137.

[2] D. A. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *Cyberinfrastructure Technology Watch*, vol. 2, pp. 1–10, 2006.

[3] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge*, ser. Contemporary Mathematics, vol. 588, 2013.

[4] J. P. Bagrow and E. M. Bollt, "Local method for detecting communities," *Phys. Rev. E*, vol. 72, p. 046108, Oct 2005.

[5] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[6] P. Crucitti, V. Latora, and S. Porta, "Centrality in networks of urban streets," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 16, no. 1, 2006.

[7] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *International Parallel and Distributed Processing Symposium*, vol. 28, 2014.

[8] L. Dematté and D. Prandi, "GPU computing for systems biology," *Briefings in bioinformatics*, vol. 11, no. 3, pp. 323–333, 2010.

[9] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future. concurrency and computation: Practice and experience," *Concurrency and Computation: Practice and Experience*, vol. 15, p. 2003, 2003.

[10] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive social network analysis: Mining Twitter for social good," in *39th International Conference on Parallel Processing (ICPP)*. IEEE, 2010, pp. 583–593.

[11] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[12] L. C. Freeman, "A set of measures of centrality based upon betweenness," *Sociometry*, vol. 40, pp. 35–41, 1977.

[13] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," in *International Confernece on Social Computing (SocialCom)*. IEEE, 2012, pp. 11–20.

[14] "The Green500 list," http://www.green500.org/lists/green201311, Nov. 2013, (accessed on 19 May 2014).

[15] "The Third Green Graph 500 list," http://green.graph500.org/lists.php, Jun. 2014, (accessed on 12 July 2014).

[16] Y. Jia, J. Hoberock, M. Garland, and J. C. Hart, "On the visualization of social and other scale-free networks," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 14, no. 6, pp. 1285–1292, 2008.

[17] 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.

[18] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong, "A novel application of parallel betweenness centrality to power grid contingency analysis," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–7.

[19] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the International Conference on Knowledge Discovery and Data mining (KDD)*, 2003.

[20] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[21] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[22] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[23] A. McLaughlin and D. A. Bader, "Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics," in *Eighth Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2014.

[24] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, pp. 117–128.

[25] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[26] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. Çatalyürek, "Regularizing Graph Centrality Computations," *Journal of Parallel and Distributed Computing (JPDC)*, 2014 (to appear).

[27] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili, "Red Fox: An execution environment for relational query processing on GPUs," in *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.