Title

Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU

Authors

Yang, Carl
Wang, Yangzihao
Owens, John D.

Publication Date

2015

Peer reviewed

Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU

Carl Yang, Yangzihao Wang, and John D. Owens
Dept. of Electrical & Computer Engr.
University of California, Davis
{*ctcyang, yzhwang, jowens*}*@ucdavis.edu*

***Abstract*—We implement a promising algorithm for sparse-matrix sparse-vector multiplication (SpMSpV) on the GPU. An efficient k-way merge lies at the heart of finding a fast parallel SpMSpV algorithm. We examine the scalability of three approaches—no sorting, merge sorting, and radix sorting—in solving this problem. For breadth-first search (BFS), we achieve a 1.26x speedup over state-of-the-art sparse-matrix dense-vector (SpMV) implementations. The algorithm seems generalizeable for single-source shortest path (SSSP) and sparse-matrix sparse-matrix multiplication, and other core graph primitives such as maximal independent set and bipartite matching.**

***Keywords*-parallel, GPU, graph algorithm, sparse matrix multiplication**

I. Introduction

Graphs are a useful abstraction for solving problems in the fields of social networks, bioinformatics, data mining, scientific simulation, and more. With the desire to solve exascale graph problems comes the impetus to design high performance algorithms. In this paper, we study the problem of traversing large graphs. Traversal is the problem of visiting all the vertices in a graph doing per-node or per-edge computations along the way. Breadth-first search (BFS) is one such traversal that is a key building block in several more complicated graph primitives such as finding maximum flow and bipartite graph matching. Its importance is such that it is one of the three benchmarks in the Graph500[1] benchmark suite.

It is natural to try to solve these problems by leveraging the computational, cost, and power efficiency advantages of the modern graphics processing unit (GPU). Graphs present a challenging problem, because the irregular data access and work distribution are not an ideal fit for the architecture and programming model of the GPU.

Linear algebra is a useful way to think about graph problems, because linear algebra implementations on GPUs have regular memory access patterns and balanced work distribution. Sparse-matrix dense-vector multiplication (SpMV) is a key building block in modern linear algebra libraries. It is a key kernel in many areas including scientific and engineering computing, financial and economic modeling, and information retrieval. SpMV is also a key operation that links graph theory and linear algebra by the fact that SpMV can be thought of as one iteration of a BFS traversal on a graph, which we discuss in more detail in Section III.

Despite being implemented quite efficiently on the GPU, existing SpMV algorithms all focus on the optimization of segmented reduction, first introduced by Sengupta et al. [1], while ignoring the possibility that the vector could be sparse. Sparse-matrix sparse-vector multiplication (SpMSpV) has not received much attention on the GPU. Graph applications where sparse vectors arise naturally include finding the maximal independent set, bipartite graph matching, and the aforementioned BFS.

If we use SpMV to perform BFS, each iteration visits every vertex and the runtime should be at least linear with the number of edges and vertices. Intuitively, when the vector is known to be sparse and we don't visit every edge, we should be able to perform the calculation in less time than $O(E)$. This paper makes a case for SpMSpV being a useful primitive when SpMV is too unwieldy and SAXPY (Single-Precision A-X Plus Y) not wieldy enough.

Our contributions are as follows:

1) We implement a promising algorithm for doing fast and efficient SpMSpV on the GPU.
2) We examine the various optimization strategies to solve the k-way merging problem that makes SpMSpV hard to implement on the GPU efficiently.
3) We provide a detailed experimental evaluation of the various strategies by comparing with SpMV and two state-of-the-art GPU implementations of BFS.
4) We demonstrate the potential of applying SpMSpV as the building block for several other graph-based BLAS operations.

II. Related Work

Gilbert, Reinhardt and Shah [2] parallellize SpMSpV by using distributed systems to launch several independent BFS searches as well as using different processors to calculate different rows. Buluç and Madduri [3] implemented the first SpMSpV algorithm for solving BFS on distributed systems. To the best of the authors' knowledge, no equivalent implementation has been made for the GPU.

On GPUs, parallel BFS algorithms are challenging because of irregular memory accesses and data dependency between iterations. Harish and Narayanan [4] gave the first level-synchronous parallel algorithm for the GPU. They

[1] http://www.graph500.org/specifications

were able to obtain a speed-up of 20–50x compared to CPU implementations of BFS. Luo et al. [5] demonstrated an improvement to the Harish-Narayanan method showing their algorithm is up to 10x faster than the Harish-Naranyan method on recent NVIDIA GPUs. Hong et al. was the first to address BFS workload irregularity [6]. They utilized an innovative warp-centric algorithm that uses mapping groups of 32 threads to vertices rather than a single thread to a vertex.

Merrill, Garland and Grimshaw [7] gave the first efficient and load-balanced BFS algorithm for the GPU. Observing that atomic operations do not scale well, they emphasized fine-grained parallelism based on efficient usage of prefix-sum. Another state-of-the-art implementation of BFS is the Gunrock library by Wang et al. [8]. They presented a high-level GPU programming model that performs graph operations at a higher performance than most implementations. We test our BFS implementation by comparing against Merrill's and Gunrock's open-source implementations.

We also compare our implementation with SpMV. Both our SpMSpV implementation and the SpMV implementation use the matrix multiplication method of solving BFS from Kepner [9]. We describe the problem in greater detail in Section III.

By comparing with the SpMV method to solve BFS, our goal is to demonstrate the superior efficiency and performance of SpMSpV in the context of the GPU. After testing several SpMV implementations, we chose the highly-optimized one from the high-performance Modern GPU library [10]. Besides using segmented reduction, Modern GPU also uses clever load-balancing to maximize hardware usage.

III. Background & Preliminaries

A graph is an ordered pair $G = (V, E, w_e, w_v)$ comprised of a set of vertices V together with a set of edges E, where $E \subseteq V \times V$. w_e and w_v are two weight functions that show the weight values attached to edges and vertices in the graph. Let $n = |V|$ and $m = |E|$. A graph is undirected if for all $v, u \in V : (v, u) \in E \iff (u, v) \in E$. Otherwise, it is directed. Among several graph data representations, the adjacency matrix and a collection of adjacency lists are the two main representations used in most existing parallel graph processing works. 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. Instead, the adjacency lists provide more compact storage for more widespread sparse graphs. A basic adjacency list stores all edges in a graph.

SpMV and graph traversal: Suppose we are interested in doing a graph traversal from a vector of vertices represented by x_k. This vector can be either sparse or dense. Supposing it is dense, it would be a length n vector with $x_k[i] = 0$ meaning the the i'th vertex is not in the group and $x_k[i] = 1$ meaning it is. Performing a graph traversal on graph G from the vector of vertices x_k is equivalent to performing the SpMV $x_{k+1} \Leftarrow A^T \times x_k$ with conventional matrix multiplication operations $(+, \times)$ replaced with $(\cup, \cap)$.

The high-level pseudocode of the matrix multiplication algorithm for BFS is given in Algorithm 1. The distinguishing feature between a traditional SpMV formulation and our SpMSpMV implementation is which multiplication kernel is used in the MULTIPLYBFS procedure.

Algorithm 1 Matrix multiplication BFS algorithm.

Input: Sparse matrix G, source vertex s, diameter d.
Output: Dense vector π where $\pi[v]$ is the shortest path from s to v, or -1 if u is unreachable.

```
1:  for each v ∈ V in parallel do
2:      w[v] ⇐ 0
3:      π[v] ⇐ −1
4:  end for
5:  w[s] ⇐ 1
6:  π[s] ⇐ 0
7:  for i ← 1, d do
8:      w ⇐ MULTIPLYBFS(G^T, w)
9:      for each v ∈ V in parallel do
10:         if w[v]=1 and π[v] = −1 then
11:             π[v] ⇐ i
12:         else
13:             w[v] ⇐ 0
14:         end if
15:     end for
16: end for
```

Graph representation: One typical way of representing graph in the GPU memory 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. In this paper, we use the well-known compressed sparse row (CSR) sparse matrix format. 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.

On GPUs: Modern GPUs are throughput-oriented manycore processors that rely on large-scale multithreading to attain high computational throughput and hide memory access time. The latest generation of NVIDIA GPUs have up to 15 multiprocessors, each with hundreds of arithmetic logic units (ALUs). GPU programs are called *kernels*, which run a large number of threads in parallel. Threads run in parallel in single-instruction, multiple-program (SPMD) fashion. For problems that require irregular data access, a successful GPU implementation needs to: 1) ensure coalesced memory access to external memory and efficiently use the memory hierarchy, 2) minimize thread divergence within a warp, 3) expose enough parallelism to keep the entire GPU busy.

IV. Algorithms & Analysis

Algorithm 2 gives the high-level pseudocode of our parallel algorithm that can be used for MULTIPLYBFS in Algorithm 1. The sparse vector x is passed into the MULTIPLYBFS in dense representation. The STREAMCOMPACT consisting of a scan and scatter is used to put the sparse vector into a sparse representation. The natively-supported scatter operation here is a moving of elements from the original array x into a list of new indices given by scan.

This sparse vector representation can be considered an analogue of the CSR format, with the simplification that since there is only one row, so the column-indices array C– which simplifies to the array with two elements $[0, m]$–will be replaced by a single variable, m.

Since the vector is sparse, we use something akin to outer product rather than SpMV's inner product. We do a linear combination on the rows of the matrix G. Even though the product we get is $G^T \times x$, we do not need to do a costly transposition in both memory storage and access since that is exactly the product we need for BFS. This way, we are only performing multiplication when we know for certain the resulting product is nonzero. This is the fundamental reason why SpMSpV is more work-efficient than SpMV.

Algorithm 2 SpMSpV multiplication algorithm for BFS.

Input: Sparse matrix G, sparse vector x (in dense representation)
Output: Sparse vector $w = G^T \times x$

```
1: procedure MULTIPLYBFS(G, x)
2:     STREAMCOMPACT(x)
3:     ind ⇐ GATHER(G, x)
4:     SORTKEYS(ind)
5:     w ⇐ SCATTER(ind)
6: end procedure
```

To get the rows of G, we do a gather operation on the rows we are interested in and concatenate them into one array. The use of this single array is our attempt of solving the multiway merging problem in parallel, which is mentioned in [3]. By concatenating into a single array, we are able to avoid atomic operations, which are known to be costly.

Going into more detail about this gather operation, we use the sparse vector x to get an index into graph G. (1) Then for all $i \in$ *ind* we gather from the graph's column-indices array obtaining two indices $C[i]$ and $C[i+1]$. These two indices give us the beginning and end of row i we are interested in. (2) Next, for all $h \in [C[i], C[i+1])$ we gather elements of row-offsets array $R[h]$ and call this set ind_i. The first two gather operations are shown as a single gather in Line 3 of Algorithm 2 and Algorithm 3.

In the case of Algorithm 3, we perform a third gather. This is to obtain the corresponding value *GVal*$[h]$ of node index $R[h]$ over the same interval $[C[i], C[i+1])$. We are now faced with the problem of doing a k-way merge of different-sized ind_i within *ind*. We tried three different approaches:

1) No sort.
2) Merge sort.
3) Radix sort.

We first try no sorting. Since the array is unsorted, adjacent threads do not write adjacent values; we instead scatter outputs to their memory destinations. The result is uncoalesced writes into GPU memory, with a resulting loss of memory bandwidth. Davidson et al. [11] use a similar strategy when they remove duplicates in parallel in their single-source shortest-path (SSSP) algorithm.

Since we are skipping the sorting, we avoid the logarithmic time factor of merge sort mentioned by Buluç et al. [3]. We scatter 1's into a dense array using the concatenated array value as the index. This approach trades off less work in sorting for lower bandwidth from uncoalesced memory writes.

Algorithm 3 Generalized SpMSpV multiplication algorithm.

Input: Sparse matrix G, sparse vector x (in dense representation), operator $\oplus$, operator $\otimes$.
Output: Sparse vector $w = G^T \times x$.

```
1: procedure MULTIPLY(G, x, ⊕, ⊗)
2:     STREAMCOMPACT(x)
3:     ind ⇐ GATHER(G, x)
4:     GVal ⇐ GATHER(G, ind)
5:     SORTPAIRS(ind, GVal)
6:     for each j ∈ ind in parallel do
7:         flag[j] ⇐ 1
8:         val[j] ⇐ GVal[j] ⊗ x[j]
9:         if ind[j] = ind[j − 1] then
10:            flag[j] ⇐ 0
11:        end if
12:    end for
13:    wVal ⇐ SEGREDUCE(val, flag, ⊕)
14:    w ⇐ SCATTER(wVal, ind)
15: end procedure
```

To increase our achieved memory bandwidth, we could perform the k-way merge by sorting. We first try a merge sort, which does $O(f \log f)$ work, where f is the size of the frontier. Though this asymptotic complexity—which is $O(m \log m)$ in the worst case—sounds bad compared to the $O(m)$ work of SpMV, it is actually much faster in practice due to the nature of BFS on typical graph topologies, which rarely visits a large fraction of the graph's vertices on a single iteration.

We also try radix sort, which has $O(kf)$ work, where k is the length of the largest key in binary. We expect merge sort to be compute-bound; no-sorting to be memory-bound; and radix sort somewhere between the two. We investigate which is more efficient in practice.

Dataset	Runtime (ms)			Dataset Description			
	SpMSpV	Gunrock	b40c	Vertices	Edges	Max Degree	Diameter
ak2010	1.686	0.932	0.104	45K	25K	199	15
belgium_osm	63.937	13.053	1.277	1.4M	1.5M	9	630
coAuthorsDBLP	4.530	2.829	0.452	0.30M	0.98M	260	36
delaunay_13	1.085	0.820	0.117	8.2K	25K	10	142
delaunay_21	11.511	2.207	0.259	2.1M	6.3M	17	230
soc-LiveJournal1	73.722	33.953	21.117	4.8M	68.9M	20333	16
kron_g500-log21	70.935	15.194	23.423	2.1M	90M	131503	6

Table I
DATASET DESCRIPTIONS AND PERFORMANCE COMPARISON OF OUR SPMSPV IMPLEMENTATION AGAINST TWO STATE-OF-THE-ART BFS IMPLEMENTATIONS ON A SINGLE GPU FOR SEVEN DATASETS.

Algorithm 3 is a generalized case of matrix multiplication parameterized by the two operations $(\oplus, \otimes)$. If we set those two operations to $(\cup, \cap)$, we obtain Algorithm 2. For low-diameter, power-law graphs, it is well-known that there are a few iterations when f becomes dense and these are the iterations that dominate the overall running time. For the remainder of BFS iterations, it is wasteful to use a dense vector.

We will investigate whether this crossing point is a fixed number independent of the total number of vertices or edges in the graph or whether it is determined by the percent of descendants f out of the total number of edges. The former would indicate a limit to SpMSpV's scalability since it would only be interesting for a small number of cases, while the latter would demonstrate that SpMSpV could outperform SpMV for BFS calculations on graphs of any scale provided they have a topology similar to those we perform our scalability tests.

V. EXPERIMENTS & RESULTS

We ran all experiments in this paper on a Linux workstation with 2× 3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, 528 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVIDIA's nvcc compiler (version 6.5.12). The C code was compiled using gcc 4.6.4. All results ignore transfer time (from disk-to-memory and CPU-to-GPU). The Gunrock code was executed using the command-line configuration `--src=0 --directed --idempotence --alpha=6`. The merge sort is from the Modern GPU library [10]. The radix sort is from the CUB library [12].

The datasets used in our experiments are shown in Table I. The graph topology of the datasets varies from small-degree large-diameter to scale-free. The soc-LiveJournal1 (soc) and kron_g500-logn21 (kron) datasets are two scale-free graphs with diameter less than 20 and unevenly distributed node degree. The belgium-osm dataset has a large diameter with small and evenly distributed node degree.

Performance summary: Looking at the comparison with two state-of-the-art BFS implementations, SpMSpV is between 2–4x slower. Nevertheless, this shows our implementation is a reasonable implementation, with runtime results in the same ballpark. With some Gunrock optimizations turned off, the results are even closer.

Dataset	Runtime (ms)	
	SpMSpV	SpMV
ak2010	1.686	0.427
belgium_osm	63.937	97.280
coAuthorsDBLP	4.530	6.213
delaunay_13	1.085	0.568
delaunay_21	11.511	22.241
soc-LiveJournal1	73.722	214.357
kron_g500-log21	70.935	230.609

Table II
PERFORMANCE COMPARISON OF OUR SPMSPV WITH SPMV FOR COMPUTING BFS ON A SINGLE GPU FOR SEVEN DATASETS.

One such BFS-specific optimization is direction-optimized traversal. This optimization is known to be effective when the frontier includes a substantial fraction of the total vertices [13]. Another reason may be kernel fusion [7]: b40c is careful to take advantage of producer-consumer locality by merging kernels together whenever possible. This way, costly reads and writes to and from global memory are minimized. Apart from that, both b40c and Gunrock use load-balancing workload mapping strategies during the neighbor list expanding phase of the traversal. Compared to b40c, Gunrock implements the direction-optimized traversal and more graph algorithms than BFS.

Comparison with SpMV: Table 2 compares SpMSpV's performance against SpMV. SpMSpV is 1.26x faster than SpMV at performing BFS on average. The primary reason is simply that SpMV does more work, performing multiplications on zeroes in the dense vector. The speed-up of SpMSpV is most prominent on scale-free graphs "soc" and "kron" where it is 2.9x and 3.3x faster. This is likely because on larger graphs, the work-efficiency of SpMSpV becomes prominent.

Such a conclusion is supported by the road network graph

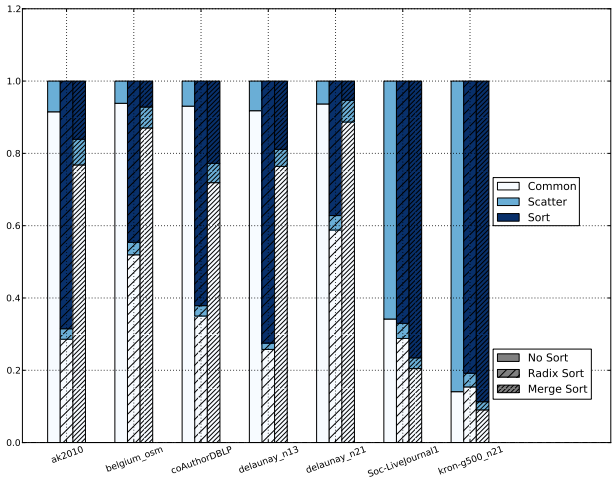

Figure 1. Workload distribution of three SpMSpV implementations. Shown are no sorting, radix sort and merge sort for the datasets listed in Table 1.

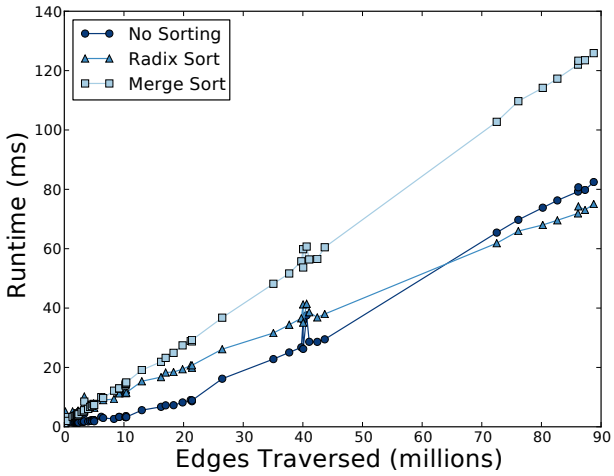

Figure 2. Performance comparison of three SpMSpV implementations on six differently-sized synthetically-generated Kronecker graphs with similar scale-free structure. The raw data used to generate this figure is given in Table IV. Each point represents a BFS kernel launch from a different node. Ten different starting nodes were used in this experiment.

"belgium". It has a large number of edges, but both the average and max degrees are low while the diameter is high. In spite of being a graph of similar size to "delaunay_21", since not many edges need traversal every iteration there is not much difference in work-efficiency between the SpMSpV and SpMV. Perhaps superior load-balancing in the SpMV kernel is the difference maker. In the same vein, it can be seen that on the two smallest graphs "ak2010" and "delaunay_13", SpMV is 3.9x and 1.9x faster.

Figure 1 shows the impact of coalesced memory access on the scatter operation. Without sorting, scatter write takes up a majority of computation time for large datasets, but becomes neglible if prior sorting has been done. The only exception is for the road network graph belgium-osm, which has a high diameter and low node degree. This could be because the neighbor list is small every time and everything in the neighbor list is kept in sorted order, so there is little gained from performing a costly sort operation. The unnormalized data is given in Table III.

Some parts of our SpMSpV implementations are common to all three of our approaches. We see some variance in this common code across our tests. Some of this variance is due to the method by which the execution times were measured, which was using the cudaEventRecord API. The rest of the variance is due to natural run-to-run variance of the GPU. This is why when possible, the runtimes taken were the average of ten iterations.

Figure 2 shows the runtime of BFS on a scale-free network ("kron") plotted against the number of edges traversed. SpMSpV implemented using radix sort and merge sort scale linearly, while SpMV (shown in Table IV and SpMSpV with no sorting scale superlinearly. For a small number of edges, it is faster to do SpMSpV without sorting. Since SpMSpV seems to perform better than SpMV on bigger datasets, it seems that the answer as to whether the crossing point beyond which SpMV becomes more efficient than SpMSpV is governed not by a fixed frontier size, but rather as a function of both frontier size and the total number of edges as well. This indicates that SpMSpV is competitive with SpMV not just on datasets of limited size, but large datasets as well.

To explain the superlinear scaling, we offer a few likely explanations. One is that congestion degrades memory access latency [14]. As Figure 1 shows, the scatter writes are the difference between no sort and sort. One phenomenon that was observed was that if only a few iterations of merge and radix sort were performed, there would be no effect on scatter time and thereby increase the total execution time. Perhaps if a sorting algorithm that divides the array in a manner like quick sort or bucket sort were used, more coalesced memory access could be attained at the cost of additional computation.

Another way to express this idea is that there is an optimal compute to memory access ratio specific for each particular GPU hardware model. It is possible that the no sort implementation reached peak compute to memory access for dataset "kron_g500-logn18", but for larger datasets memory access grew faster than the amount of gather operations, so memory accesses were becoming degraded by congestion. The sorting methods may be closer to the compute-limited side of the compute to memory access peak, so the increased memory accesses are bringing them closer to peak performance.

VI. Conclusions

In this paper we implement a promising algorithm for computing sparse matrix sparse vector multiplication on the GPU. Our results using SpMSpV show considerable

	Runtime (ms)								
	No Sort			Radix Sort			Merge Sort		
Dataset	Common	Scatter	Sort	Common	Scatter	Sort	Common	Scatter	Sort
ak2010	0.5979	0.0556	0	0.5349	0.05433	1.2820	0.5387	0.0499	0.1128
belgium_osm	64.60	4.1906	0	63.03	4.1906	0	62.85	4.1843	5.1752
coAuthorsDBLP	5.4573	0.4067	0	5.3658	0.4403	9.5211	5.3508	0.3984	1.6931
delaunay_13	0.9395	0.0839	0	0.9146	0.0627	2.5720	0.9290	0.0573	0.2295
delaunay_21	11.18	0.7558	0	11.00	0.7479	6.9629	11.02	0.7506	0.6574
soc-LiveJournal1	21.71	41.80	0	21.67	3.1452	50.40	21.67	3.1452	50.40
kron-g500_n21	11.13	67.90	0	11.10	2.7280	58.24	11.12	2.7659	108.93

Table III
WORKLOAD DISTRIBUTION OF THREE SPMSPV IMPLEMENTATIONS SHOWING RUNTIME (MS) ON A SINGLE GPU FOR SEVEN DATASETS. COMMON REFERS TO TIME SPENT RUNNING THE KERNELS COMMON TO ALL THREE IMPLEMENTATIONS.

	Runtime (ms)				Edge rate (MTEPS)			
Dataset	No Sorting	SpMV	Radix Sort	Merge Sort	No Sort	SpMV	Radix Sort	Merge Sort
kron_g500-logn16 ($n = 2^{16}, m = 2.5$M)	1.37	2.21	4.74	3.81	1401.9	868.3	405.07	503.9
kron_g500-logn17 ($n = 2^{17}, m = 5.1$M)	1.71	4.02	5.81	5.35	1923.5	819.6	567.3	615.2
kron_g500-logn18 ($n = 2^{18}, m = 10.6$M)	2.85	7.70	9.79	11.37	2764.8	1022.8	804.7	692.4
kron_g500-logn19 ($n = 2^{19}, m = 21.8$M)	6.79	19.94	16.85	22.31	2372.0	807.9	955.8	721.8
kron_g500-logn20 ($n = 2^{20}, m = 44.6$M)	21.08	75.97	29.25	43.13	1469.0	407.7	1058.7	718.2
kron_g500-logn21 ($n = 2^{21}, m = 91.0$M)	68.09	259.86	64.23	105.92	1087.7	285.0	1153.0	699.2

Table IV
SCALABILITY OF THREE SPMSPV IMPLEMENTATIONS AND ONE SPMV IMPLEMENTATION (RUNTIME AND EDGES TRAVERSED PER SECOND) ON A SINGLE GPU ON SIX DIFFERENTLY-SIZED SYNTHETICALLY-GENERATED KRONECKER GRAPHS WITH SIMILAR SCALE-FREE STRUCTURE. RADIX SORT AND MERGE SORT SCALE LINEARLY; NO SORTING AND SPMV SHOW NON-IDEAL SCALING.

performance improvement for BFS over the traditional SpMV method on power-law graphs. We also show that our implementation of SpMSpV is flexible and can be used as a building block for several other graph primitives.

An open research question now is how to optimize the compute to memory access ratio to maintain linear scaling. We showed merge sort and radix sort are good options, but it is possible a partial quick sort or a hybrid k-way merge algorithm such as the one presented by Leischner [15] can be used to obtain a better compute to memory access ratio, and better performance.

The SpMSpV algorithm used in this paper is generalizable to other graph algorithms through Algorithm 3. This algorithm is still being implemented in CUDA. By setting $(\oplus, \otimes)$ to $(+, \times)$, one performs standard matrix multiplication. A direction may be using SpMSpV as a building block for sparse matrix sparse matrix multiplication. Buluç and Gilbert's work in simulating parallel SpGEMM sequentially using SpMSpV has been promising [16]. Similarly, by setting $(\oplus, \otimes)$ to $(\min, +)$, one performs single-source shortest path (SSSP).

A natural question would be whether SpMSpV brings similar speed-ups to SSSP too. More research will be needed. The analogy to linear algebra is held to bring advances to graph algorithms, so another natural question would be whether advances in graph algorithms can be used to improve sparse matrix multiplication.

VII. ACKNOWLEDGEMENTS

This work was funded by the DARPA XDATA program under AFRL Contract FA8750-13-C-0002 and by NSF awards CCF-1017399 and OCI-1032859.

REFERENCES

[1] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*, Aug. 2007, pp. 97–106.

[2] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *Applied Parallel Computing: State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science. Springer, Mar. 2007, vol. 4699, pp. 260–269.

[3] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2011, pp. 65:1–65:12.

[4] P. 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*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, Dec. 2007, pp. 197–208.

[5] L. Luo, M. Wong, and W. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, Jun. 2010, pp. 52–55.

[6] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, Feb. 2011, pp. 267–276.

[7] 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, Feb. 2012, pp. 117–128.

[8] 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.

[9] J. Kepner and J. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011, vol. 22.

[10] S. Baxter, "Modern GPU library," http://nvlabs.github.io/moderngpu/, 2015, accessed: 2015-02-22.

[11] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single source shortest paths," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, May 2014, pp. 349–359.

[12] D. Merrill, "CUB library," http://nvlabs.github.io/cub, 2015, accessed: 2015-02-22.

[13] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Nov. 2012, pp. 12:1–12:10.

[14] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W. W. Hwu, "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, Feb. 2012, pp. 23–34.

[15] N. Leischner, "GPU algorithms for comparison-based sorting and merging based on multiway selection," Ph.D. dissertation, Karlsruhe Institute of Technology, 2010.

[16] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS 2008, Apr. 2008.