Evaluating High Performance Graph Frameworks for GPUs

Euna Kim[1], Zahra Ronaghi[2], Ramakrishnan Kannan[3], Alex Fender[2], Bradley Rees[2], Joe Eaton[2],
Muhammad Osama[4], Yuechao Pan[4], John Owens[4], David Bader[1] and Oded Green[2,1]
[1]Georgia Institute of Technology {euna.kim,bader}@gatech.edu
[2]NVIDIA corp. {zronaghi,afender,brees,eaton,ogreen}@nvidia.com
[3]Oak Ridge National Laboratory kannanr@ornl.gov
[4]UC Davis {mosama,ychpan,jowens}@ece.ucdavis.edu

***Abstract*—Graph frameworks are a great way to abstract both the graph data structure and the algorithmic programming model to enable productivity and high performance on a wide range of systems, including shared-memory, heterogeneous systems, distributed systems, or accelerators. Over the last two decades, a plethora of such systems have been created with the introduction of new hardware. The introduction of CUDA as a parallel programming language for the NVIDIA GPU roughly a decade ago also created a wider interest in parallel graph algorithms and frameworks. In the last five years, a wide range of algorithms have been designed for the GPU, though many of these are standalone codes that can only deal with one problem or problem type. However, a few frameworks have also been put together to enable portability, productivity, scalability, and performance. These frameworks greatly vary in the exact tools and knowledge needed to develop additional algorithms.**

In this paper we explore several of the best performing GPU-based graph frameworks on six NVIDIA GPU systems. While we do compare the performance of the frameworks across several algorithms and input graphs, our goal is not to find and declare one framework to be the winner over the others as we understand that each framework was designed with different objectives and across a different time-span. Rather, our goal is to share our practical experiences with developing both the frameworks and graph algorithms for these frameworks, as well as performance differences in different system configurations, and to offer readers a set of good practices when trying to achieve high performance on modern GPU frameworks.

***Index Terms*—GPU, Graph frameworks, Benchmark**

I. Introduction

Graphs are ubiquitous and can be found in a wide range of real world applications. Many of these problems are sparse and unstructured in nature. Yet, using graphs brings certain structure to these problems and allows for using well-researched graph utilities and tools. Using graphs it is easy and intuitive to create relationships between entities followed by advanced queries. A few domains that use graphs are molecular structures in chemistry, lattice structures in physics, functional connections of brains, web graphs, social networks, and natural language processing.

The ability to analyze larger graphs (also referred to as networks) at significantly higher rates has, in return, created more interest and need for faster and more scalable graph solutions. For some very well-defined problems, distributed solutions can be implemented using Hadoop [30] and GraphX [13] (which is an extension of SPARK [31]). These frameworks are extremely limited to solving problems that fit the bulk synchronous model. Therefore, they cannot be applied to computationally intensive problems as these are not suitable for the bulk synchronous model. In sharp contrast, NetworkX [16] and iGraph [17] have larger user bases because of their functionality, despite the limited scalability and sequential nature. These frameworks sacrifice efficiency and data scalability for usability.

The HPC community has also spent a lot of time and effort developing various graph frameworks. These typically split between distributed (shared-nothing) systems and shared-memory systems. HPC systems such as the Cray Urika and massive supercomputers such as Titan and Summit have lots of compute, memory, and communication resources available. The irregularity of many of these graph problems and the lack of structure found in most graphs make them especially challenging. This makes designing and implementing good tools for productivity even more difficult.

In the last decade a plethora of shared-memory graph frameworks have been created focusing on productivity and enabling relatively simple parallelization. Some of these frameworks designed for the CPU include the following: [2], [5], [23], [26], [27]. Parallel programming on these shared memory systems is convenient and the capability of these systems to have as much as 8TB of DRAM makes them attractive. Yet, they lack the computational power to efficiently process large graphs with respect to time.

For this reason, accelerators such as NVIDIA's GPU have become an integral part of shared memory systems. Specifically, GPUs have significantly more computational power than most servers with CPUs, which have higher bandwidth memory. The use of GPUs comes at the cost of having a small amount of shared memory and slow communication bus between the CPU and GPU. Effective use of the NVIDIA GPUs, using the Volta architecture, requires parallelizing with approximately 40,000 threads. Currently, it is possible to obtain a wide range of HPC servers with multiple GPUs: IBM's Power 8 and Power 9 servers can have up to 4 and 6 GPUs, respectively. NVIDIA's DGX-1 and DGX-2 can have up to 8 and 16 GPUs in a single server. These also have a fast communication bus among the GPUs in the form of NVLink and NVSwitch [10]. To design a effective graph framework for these multi-GPU servers, we first need to understand the characteristics and capabilities for a single GPU system.

The focus of this paper is to review several recent graph frameworks and their programming models. This includes considering resource management and constraints such as memory, bandwidth, data layout, memory access pattern, workload mapping, branch divergence, and the scope of their capabilities. For this work, we choose to consider Hornet [8], Gunrock [28], and cuGraph (formerly nvGraph [22]).

In this paper we will explore the difference in performance of these frameworks with an emphasis on understanding when and why one framework performed better than the other frameworks. Is the performance result of better algorithm design, implementation, load-balancing, problem formulation, or an appropriate programming model (which can vary significantly between GAS, vertex-centric, or BLAS)? We will explore performance only using GPU-based frameworks as there are tuning parameters available to take into consideration so that we avoid comparing different programming models on vastly different architectures, such as CPUs.

We believe this study will help (a) the end-users to determine a framework for solving their problems at hand and (b) the researchers to share the best practices and to evolve towards a common GPU framework that allows the community to design scalable graph analysis algorithms.

The remainder of the paper is organized as follows: Section III provides an overview of the GPU graph framework; Section IV describes the evaluation methodology for a graph benchmark; Section V describes our experimental setup; and Section VI describes and discusses our experimental results.

II. Background and Related Work

Achieving good run-time performance for a graph algorithms is challenging. The execution time is dependent on a wide range of parameters: type of system (shared-memory vs. distributed), number of threads (from tens of threads on CPU systems to tens of thousands on current GPU processors), algorithm and problem statement (BLAS, vertex centric, GAS, BSP), load-balancing, and the input graph itself. Given these complexities, programming frameworks which can encompass and abstract away many of these programming challenges will have the benefit of being accessible to a wider range of programmers and users. Additional benefits of frameworks include improved user productivity, increased code portability, and additional functionality (that can also include basic ETL functionality such as loading and storing a graph).

Over the years, various frameworks have been developed across a broad range of languages and parallel frameworks, including: OpenMP MPI, Cilk, C++ AMP (Accelerated Massive Parallelism) or CUDA. Other approaches include Intel's TBB and Kokkos [9] which give the user a set of basic primitives to implement their algorithms. However, these frameworks tend to require a good amount of expertise and do not offer many implementations.

1) Shared Memory vs. Distributed Memory Systems: Over the last two decades there has been in an increase in the number of scalable graph frameworks. These have primarily split into two types of frameworks, the distributed memory frameworks and the shared memory framework. While there are several frameworks that can work on both systems (such as Kokkos [9]), they are typically designed for one type of system. Furthermore, these frameworks are designed to solve different types of problems. For example, PowerGraph [12], Pregel [20], and GraphX [13] which are all widely used distributed frameworks currently support a small subset of graph problems, which fall under the category of algorithms that can be implemented with bulk synchronous operation. This makes the algorithms very communication bound and limits these graph frameworks ability to support computationally intensive and IO intensive graph algorithms due to their random (sometimes referred to as irregular) memory access pattern.

In contrast, shared memory graph frameworks such as LIGRA [26] and GAP [5] can support a wider range of algorithms however are limited to the computational resources and memory of a single compute node. GAP, unlike LIGRA, requires lower programming and does not abstract away the HPC details of its framework. GPU graph frameworks fall into this category as well and offer significantly higher computational resources at the cost of reduced memory subsystem. These frameworks include (to name a few): Gunrock [28], Hornet [8] (and cuSTINGER [15], cuGraph (formerly nvGraph [22]), and cuSHA [18]. Gunrock and Hornet use a vertex centric programming model. nvGraph's programming model is based on GraphBLAS (an extension of BLAS designed for graphs). cuSHA uses a (Gather-Apply-Scatter) GAS programming model.

A. Graph Frameworks Functionality

Given the challenges of implementing high performance and scalable graph algorithms, there is clear need for higher level functionality that abstracts away the key functionality of graph algorithms from the implementation on actual compute systems. This topic has received a good amount of research over the last decade and several paradigms have been designed. While these frameworks have a good amount of similarity, they are also quite different. One thing that they share in common is that they require the programmer to "think" in their programming paradigm.

LIGRA [26] and Gunrock [28] use frontier based approach that suggests that parallelism be visualized as "waves" of vertices or edges that can be traversed concurrently. In practice, these frontiers are created explicitly by the user using very simple and scalable filtering functionality . Then, all vertices and edges in the frontier can be traversed concurrently. Thus, these frontier operations consist of two phases. Hornet [8] takes a slightly different approach than LIGRA and Gunrock and offers a set of primitives (sometimes referred to as $parallel - for$ loops) that operates on sets of vertices and edges. Specifically, in Hornet the edge lists are created explicitly by the framework and implicitly by the user. Thus, the frontier-like operations occur in one phase rather than two phases.

Galois [23] uses an operator based formulation that separates the compute operation from the actual parallel runtime. Thus, Galois is responsible for the runtime and dispatches the various work units to the threads.

GraphMat [27], CombBLAS [7], and nvGraph [22] use BLAS operations to implement graph algorithms. Specifically, graph algorithms are implemented using a small subset of linear algebra operations. The benefit of this approach is that improvements made to SpGEMM or SpMV immediately improves the performance of the graph algorithms. This also allows for improving scalability of BLAS based solutions; specifically as these operations also receive significant attention for distributed systems and these solutions can they be applied to the graph problem. The key downside of this approach is that not all problems can be formulated as BLAS operations.

Gather-Apply-Scatter (GAS) is also very popular due to its simple three phase bulk synchronous programming model. Each phase in the algorithm consists of three sub-phases: 1) each vertex receives a set of messages from its neighbors (Gather), 2) each vertex processes the messages (Apply) and 3) each vertex sends a message to its neighbor after process the previous rounds of messages (Scatter). The simplicity of this model makes it very attractive for distributed environments (PowerGraph [12], Pregel [20], and GraphX [13]). The GAS programming model introduces a large number of synchronization as well a good amount of communication (to send and receive messages) that limits it applicability to a wider range of applications.

III. GPU Graph Frameworks

A. Hornet

Hornet [1], [8] is a dynamic graph data structure designed for sparse data. Hornet shares many properties with CSR and can be considered a dynamic version of CSR. Specifically, for both CSR and Hornet the adjacency array for each vertex is a single compact array that allows accessing the neighbors of a vertex using sequential accesses. Further, CSR and Hornet store these adjacency arrays in compact format.

Hornet's algorithm library is HornetsNest. One of the key objectives in the design of Hornet and HornetsNest was programmer productivity, specifically the goal was to create a high performance graph frameworks for both static and dynamic graph problems where algorithms could be implemented in a short amount of time while achieving excellent performance.

These graph primitives are essentially $parallelfor$ loops that enable simple traversal of the vertices and edges. The edge traversal operations are also supported by various load-balancing mechanisms that ensure that the execution of an algorithm can be easily extended to execute on the tens of thousands of threads available in modern GPU systems. From a productivity perspective, users do not need to understand the internal details of Hornet to achieve a high level of performance. Lastly, HornetsNest includes several graph primitives that are currently unique and cannot be found in most frameworks—namely, dynamic graph primitives such as $edgeInsertion$ and $edgeDeletion$ (both of which include batched versions that allow doing their respective operation concurrently on a large number of edges).

B. [TBD]Gunrock - gunrock team

C. [TBD]cuGraph(nvGraph) - Joe, Alex

IV. Benchmarks

To benchmark the GPU graph frameworks, we have selected five basic graph kernels which are used in a wide range of applications. Specifically, we focus on Triangle Counting (TC, which is used in a wide range of common neighbor explorations) Connected Components (CC), PageRank (PR), and two widely used graph traversal operations: Breadth First Search (BFS) and Betweenness Centrality (BC). This problem set was selected as together they encompass a much larger number of problems. Specifically, triangle counting was selected as it captures the capability to execute a large number of small and imbalanced set intersections to find common neighbors. PageRank and connected components are two analytics that can show the cost of data propagation across a network. PageRank, unlike connected components, typically needs a large number of iterations to converge due to accuracy requirements. BFS and BC are traversal based analytics for finding the distance between a root (or a selected vertex) and the remaining vertices in the graph. Together these algorithms represent a much larger set of application use cases (to name just a few): all-pair shortest-paths, s-t connectivity, single-source shortest path, and closeness centrality. While BC requires doing a BFS as part of its computation, these two problems are very different as we will summarize below.

A. Triangle Counting

Counting triangles in a graph is the operation in which common neighbors of two vertices are found. Specifically, for each edge, (u, v), in the graph we find the number of common neighbors that they have. Further, as the edge between u and v already exists, by finding common neighbors in the format of edges (u, w) and (v, w) we are in fact finding triangles. Counting triangles or finding common neighbors represents a much large set of problems: clustering coefficients, Jaccard indices, K-Trusses, clique-finding and much more.

Finding triangles in a graph can be accomplished in numerous manners: enumerating over all node triplets, matrix multiplications, and set intersections (both sorted and unsorted approaches exist). The upper-bound time complexity of each of these approaches vary quite a bit, yet the actual time it takes to find triangles is very data dependent. Numerous optimization's have been applied to the problem of triangle counting over the last two decades, including: avoiding counting the same triangle multiple times, load-balancing at different thread granularities, different approaches for conducting the set intersection.

B. Connected Component

The Connected Component(CC) algorithm (sometimes referred to as "Union Find") finds the maximal sets of connected vertices in undirected graphs such that each vertex is reachable from any vertex in that set. A serial form of the CC algorithm can be computed using a depth-first search(DFS) or Breadth-first search(BFS). The low computational requirements of this problem makes parallelizing it computationally challenging. Parallel algorithms are typically based on either a slightly modified parallel BFS or through "label propagation". Additional details on parallel BFS can be found in Section IV-D, though the modifications made to BFS are minimal. Instead of storing the distance to the root, we store the root itself.

In contrast, for the label propagation approach, initially all vertices in the graph are assigned their own label. In each iteration of the label propagation, each vertex can update its own component id by that of the component id of its neighbors. The iterations work in a bulk-synchronous like manner. The connected components of the graph are considered to have been found once no changes are made by all the vertices in the graph. Typically, the number of iterations is limited by the diameter of the graph, dia, though in a parallel execution there are numerous ways to cut this time complexity down to $log(dia)$.

C. PageRank

PageRank is a link analysis method used to assess the relative importance vertices in a graph based on the number of relative number of random walks that go through the vertices. By default, the PageRank algorithm is the stationary distribution of a random walk which has a damping factor d probability to jump to a random node in a graph, and $(1-d)$ probability to choose one of connected edges(outgoing edges in directed graphs) from the current node. Unless the graph is regular and cyclic, then the PageRank value typically converges to a final value. PageRank is computed in an iterative manner such that the next PageRank value of a vertex is dependent on the PageRank values of the past iteration. Thus, PageRank also works in a bulk synchronous manner similar to connected components. Another similarity with connected components is the fact that the values slowly propagate in the graph. However, unlike connected components where the propagation is in both directions, in PageRank the propagation is weighted and is in one direction (incoming edges).

In the iterative method, PageRank is computed as follows (where t represents the iteration):

$$PR^{t+1}(v) = \frac{(1-d)}{|V|} + d \sum_{u \in adj(v)} \frac{PR^t(u)}{out\_degree(u)} \quad (1)$$

d is the damping factor which is typically set around 0.85 and it is a compromise between accuracy and convergence rate. Furthermore, convergence of PageRank values can be determined in different manners and are application dependent.

The linear algebra expression for PageRank is as below [24]:

$$(I - \alpha A^T D^{-1})x = (1-\alpha)v \quad (2)$$

A is an adjacent matrix, D is a diagonal matrix of out-degree edges, x is the PageRank vector, v is a vector having $1/|V|$ as elements and α is teleportation constant (a.k.a d, damping factor)

Due to the intensive computation required for PageRank computation, the performance is highly affected by the hardware support and how the hardware is used. FPGA is used for the accelerating the PageRank processing [32], and optimization considering cache sizes and data blocks for PageRank computation to increase the locality is examined in [3].

D. Breadth-first Search

Breadth-first search (BFS) is one of the most basic algorithms in graph analysis. A BFS traversal starts from a root (sometimes referred to as a source) vertex and finds all its neighbors while marking their distance (to ensure that they are not traversed again). This operation is repeated until all the vertices reachable from the root have been found. This is commonly referred to as the top-down approach. BFS can also be implemented using the bottom-up and direction optimizing approach suggested by Beamer *et al.* [4]. This bottom-up approach reduces the number of random memory operations by avoiding unnecessary traversals.

BFS is very sensitive to workload imbalance and memory operations. Specifically, in massively multi-threaded systems it is very important to ensure that work is available to all threads. The randomness of the traversals also makes the memory accesses harder than traditional sequential memory operations. Given the low computational effort required by each traversal, BFS is a great benchmark for testing the IO capabilities of a system. It is for this reason that the Graph500 [21] benchmark was created. Graph500 benchmark does not state how the traversal should be implemented, rather it focuses on the number of traversals per second that can be executed. Most scalable BFS implementations use either a BLAS or vertex-centric formulation. While GAS based formulations are possible, their performance is typically below desirable.

E. Betweenness Centrality

The Betweenness Centrality, BC, of a vertex is the fraction of shortest paths in the graph that go through this vertex over the total number of shortest paths in the graph [11]. Formally, this is defined as follows: σ_{st} is the number of shortest paths from a vertex s to a vertex t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t through a vertex v. For a vertex v, the BC $C_B(v)$ is

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (3)$$

Brandes [6] showed an efficient algorithm, with the best known complexity bounds, for computing the BC values in

$O(V \cdot (V + E))$ time. Brandes's algorithm consists of two phases (executed for each vertex in the graph), a BFS traversal executed from each vertex (root) followed by a second phase called the dependency accumulation phase. Readers are referred to [6] for additional detail. The BFS phase is not able to benefit from the Direction-Optimized BFS [4] as it requires traversing "all" shortest paths and not "a" single path.

BC has received significant attention for its much larger computational elements than that found in BFS. Specifically, BC has both the BFS and dependency accumulation phases where the latter requires floating point operations (multiplication and additions), as well it requiring synchronization through the use of atomic instructions [14], [19].

V. Experiment Setup

A. Systems

Our experiments are conducted on two NVIDIA GPUs: a V100 GPU and a P100 GPU in six systems of different memory size and interconnection type. Specific details of these GPUs can be found in Table I. The P100 is a Pascal based GPU and the V100 is a Volta based GPU. Both the V100 and P100 have two form factors: PCI-E and SXM. PCI-E is the defacto form factor that most consumer GPUs are manufactured. The SXM form factor is equivalent to placing a GPU on a board. The SXM form factor also has multiple NVLink channels allowing GPUs to communicate with other multiple GPUs concurrently. For both the V100 and P100, the SXM form factor GPU is known for outperforming its PCI-E counterpart due to increased frequency and power consumption.

The effect of CPUs used in our experiments are not significant; specifically, we do not time the transfer to and from the GPU. To ensure that execution times capture the time that it takes to run an analytic, we time only sections that are entirely on the GPU side and are part of the analytic procedure.

B. Graphs

Details of the graphs used in our experiments can be found in the upper part of Table II. This table covers the number of vertices, edges, and average degree of each graph. For each network, the table also denotes if that network is stored as a directed or undirected graph. Undirected graphs store both directions of each edge. The experiments are executed for both directed and undirected versions of each graph. Thus, if a graph is given as a directed graph, for its undirected version the inverse direction of each edge is added. If that edge already exists, then that duplicate edge is not included. Different frameworks load directed and undirected graphs using different methods; Most of graph frameworks support graph loading features with user parameters. Loading graphs as the original structure(undirected graphs as undirected, directed graph as directed) is not complicated and explained above. We found that Hornet generates a directed graph from an undirected graph by taking the first read edge and ignoring the second edge in the opposite direction while Gunrock ignores ”–directed” for loading undirected graphs.

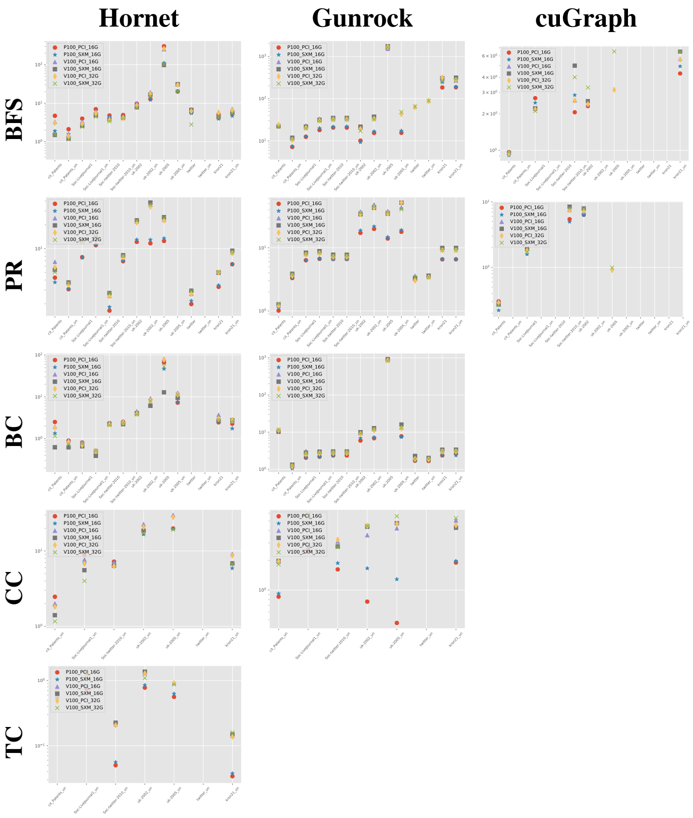

Fig. 1. Logarithmic GTEPS of each benchmark with real world and synthetic graph(s) in Hornet, Gunrock, and cuGraph

VI. Empirical Performance

We configured the frameworks with appropriate parameters for the functionality we evaluated. For instance, for Gunrock, we obtained results using both tuned and default parallelization options.

For each case the operating systems, device driver types, and the versions of each are identical. As mentioned in V, performance effects as a result of CPU differences are minimal since the measurement of computation time is only conducted on the GPU side. However, any processor-specific optimizations in the driver might affect performance. Note that the CPU types are different with respect to form factor and cache size; all of these differences are encapsulated in the general interconnect types, *i.e.*, PCI-E and SXM, in the rest of this section.

A. Breadth First Search

For most cases we found that cuGraph, the only BLAS based implementation, was slower than the vertex-centric implementation. In some cases it was up to 4× slower than Hornet and 20× slower than Gunrock's faster implementation with parallelization options. Gunrock's BFS utilizes the popular bottom-up BFS scheme which is an algorithmic optimization available for vertex-centric algorithms that is not available for BLAS based implementations.

Hornet's vertex-centric BFS algorithm has comparable performance to Gunrock's default BFS. For Gunrock we also

TABLE I
[TBD] DESCRIPTION OF SIX GPU SYSTEMS USED FOR THE EXPERIMENTS

System[a]	CPU	Form factor	Processor	M.Arch.	SMs	SP/SM	Total SPs	DRAM Size	DRAM Type	Power
P100_16GB_PCI	2.30GHz[b]	PCI-E	P100	Pascal	56	64	3854	16GB	HBM2	250W
P100_16GB_SXM	(TBD)	SXM-2	P100	Pascal	56	64	3854	16GB	HBM2	300W
V100_16GB_PCI	2.30GHz[b]	PCI-E	V100	Volta	80	64	5120	32GB	HBM2	250W
V100_16GB_SXM	2.20GHz[c]	SXM-2	V100	Volta	80	64	5120	32GB	HBM2	300W
V100_32GB_PCI	2.30GHz[b]	PCI-E	V100	Volta	80	64	5120	32GB	HBM2	250W
V100_32GB_SXM	2.20GHz[c]	SXM-2	V100	Volta	80	64	5120	32GB	HBM2	300W

[a] Device driver version 418.67 and operating system CentOS Linux 7 for all systems, [b] Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz with cache 40960KB, [c] Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz with cache 51200KB

TABLE II
GRAPH DATA USED FOR EXPERIMENTS

Name	$\|V\|$	$\|E\|$ [a]	Size	Structure	Deg [b]
cit_Patents	4E+06	17E+06	249 MB	Directed	24
Soc-LiveJournal1	5E+06	69E+06	964 MB	Directed	14
Soc-twitter-2010	19E+06	298E+06	4,809 MB	Undirected	16
uk-2002	21E+06	530E+06	4,977 MB	Directed	25
uk-2005	40E+06	936E+06	15,689 MB	Directed	24
twitter	42E+06	2,405E+06	22,885 MB	Directed	35
kron21 [c]	2E+06	182E+06	1,471 MB	Undirected	87

[a] The average degree of each graph, [b] Sorted by the number of edges, [c] Synthetic graph

report an optimized execution times where the BFS runtime parameters, including the parameters that determine when to swap between the classic top-down and the new bottom-up algorithm, are set to get the best execution for a specific graph. In many cases, these parameters are not known *a priori*, though they do offer an insight on performance gains if these parameters selected in advance. In order to fully exploit the Gunrock's BFS performance enhancement, we used optimization parameter for Gunrock as "–idempotence –queue-sizing=6.5 –in-sizing=4 –iteration-num=10 –direction-optimized –device=0 –traversal-mode=LB_CULL –do_a=0.200 –do_b=0.1". The details of optimization strategies for graph traversal, as well as their effectiveness and trade-offs are enumerated in [29].

The results of our experimentation show that in the BFS case, with respect to SXM versus PCI-E, cuGraph is better when SXM is used, Hornet is better when using a PCI-E form factor, and Gunrock is slightly faster in the PCI-E form factor. Unlike in the Hornet case, the performance difference when using SXM versus PCI-E is not as pronounced with Gunrock.

With respect to the P100 versus the V100, Hornet's BFS performance is better with the P100; cuGraph's performance is better with the V100; and Gunrock runs faster with the V100 as well. It should also be noted that cuGraph's V100 performance is significantly improved with larger graphs.

From the memory access latency perspective, the best performer was V100 with 16GB among the three systems tested, but some very large graphs, such as the uk-2002 and twitter graphs, could not complete the computation within 16GB. A traversal algorithm like BFS shows different behaviors than other types of algorithms. Other algorithms will be discussed below.

B. PageRank

Recall, PageRank algorithms can terminate their execution depending on the convergence requirements, as such we report the average time for each iteration. For efficiency, the termination conditions for PageRank are typically set as a maximum number of iterations or a convergence criteria that ensures the gap of PageRank scores between the current iteration and that of previous iteration meet some lower bound. We set the maximum iteration number as 50 for our experiments for all the frameworks. Each PageRank implementation in each framework represents this score gap with different names ("threshold" in Hornet, "error" in Gunrock and "tolerance" in cuGraph) and have a distinct function of this user algorithmic parameter. The *threshold* for Hornet is the sum of PageRank score gaps in all vertices in the graph while the *error* in Gunrock is used to check each score gap in each vertex. The *tolerance* value in cuGraph is compared to the Euclidean distance of all vertices in the graph. Furthermore, those values for the convergence criteria have different behaviors and are set differently in each framework. We choose these score gap values as carefully as possible so that they result in generating a similar number of PageRank iterations across all of the frameworks. Note that we are not aiming for the number of iterations to be equal, just close. Because of this, we provide the runtime per each iteration rather than that of the total PageRank computation.

From a performance perspective, the execution times of Hornet and Gunrock are comparable for most graphs, though there are some cases where one of the frameworks is up to 2× faster than the other. This occurs for both frameworks. cuGraph did not finish for a large number of graphs.

Similar to the BFS results, using the V100 with 16GB produces the best performance in PageRank algorithm. The difference is that the V100 is always faster than the P100, including Hornet. Interestingly, Gunrock optimized parallelization options did not produce a significant difference in runtime unlike in the BFS case. Another interesting finding is that SXM works well with algorithms requiring iterative and intensive computation like PageRank. We found SXM outperformed PCI-E in most frameworks, especially on large graphs. When graphs are big, the amortized cost produces a better performance in SXM.

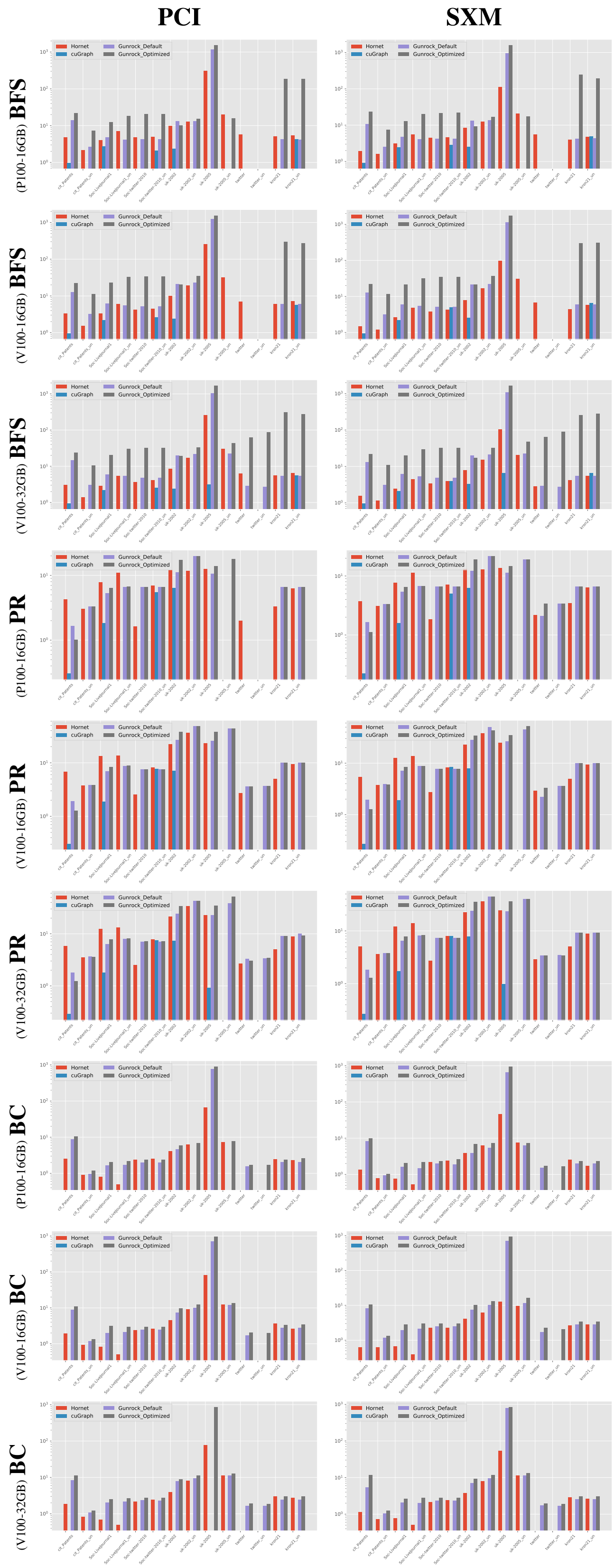

Fig. 2. GTEPS of benchmarks BFS, PR, BC, CC, and TC in six different GPU systems. Traveral algorithms such as BFS are executed with the largest degree node as a root. Some benchmarks are available in all frameworks, such as CC in cuGraph. A few computations are not completed and not shown to OOM(Out-of-memory)

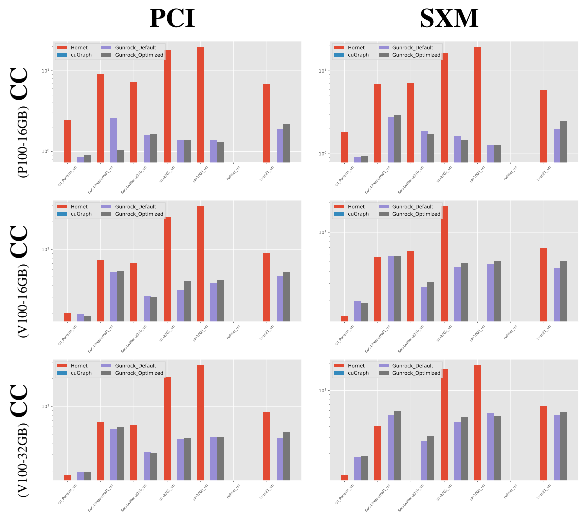

Fig. 3. [TBD] TC cuGraph results will be added

C. Betweenness Centrality

For betweenness centrality (BC), we used the root with the largest degree to compare the performance of the frameworks. Gunrock's faster algorithm was in most cases $3\times - 4\times$ faster than Hornet. There are a few instances where Hornet is 2× faster than Gunrock. The BFS traversal used in BC is a top-down BFS for both Hornet and Gunrock. We are unable to report BC times for cuGraph as it is currently unsupported.

BC is a traversal algorithm that requires a significant amount of computation. V100 produced the best performance in all three frameworks and V100 with 32GB is the winner in BC unlike PR or BFS, as the memory intensive computations deteriorates the performance of the V100 with 16GB of memory. Similar to the previous results in BFS and PR, SXM can be a good consideration if the size of the graph of interest is bigger than the uk-2005 directed graph. Note that Gunrock's parallelization options are not effective when computing BC.

D. Connected Components

Hornet and Gunrock have two very distinct implementations of connected components. Gunrock uses a label propagation approach that is an extension of the popular [25] algorithm. In contrast, Hornet uses a BFS based implementation that finds all the vertices that are connected to a given root. Hornet then iterates over the vertices to find vertices that are not part of a connect component. We are unable to report times for cuGraph as it is currently does not support connected components.

Since Hornet and Gunrock run distinct connected component algorithms, as a result, they return sets of different CCs. Thus, direct comparison of their runtimes do not show the capabilities of the frameworks. Similar to the BFS results, Hornet performs better in systems with a PCI-E form factor while Gunrock produces the better performance in SXM systems. When the graph size is below uk-2005 in the P100 case, V100 is always a better performer with SXM, including with uk-2005. Computation of the twitter case failed in both

Gunrock and Hornet. In the V100 32GB case, Gunrock optimization occasionally appears slower than the default Gunrock configuration with the uk-2005 and twitter graphs.

E. Triangle Counting

[TBD] Triangle Counting (TC) is currently only available in the Hornet framework. The SXM form factor shows better performance than PCI-E; the difference is greater in P100 than V100. The V100 produces almost 2× speed up in TC than P100.

a) Each Framework: There are occasionally outlying cases, but in general Gunrock with parallelization options produces the best performance among the three frameworks. Gunrock default shows similar results with Hornet in many cases, while cuGraphs shows the slowest runtime and largest memory usage in most of benchmarks.

b) PCI-E Vs. SXM Form Factor: As part of our extensive benchmarking, the algorithms were executed on a wide range of GPUs, including identical GPUs that came in different form factors. For Gunrock, we typically see the performance increase when moving from the PCI-E version of the GPU to its respective SXM form factor (which has better performance and a higher frequency). In contrast this was not the case for traversal algorithms in Hornet, especially for smaller graphs. For larger graphs, Hornet obtained better performance with the SXM form factor with computationally intensive algorithms.

Specifically, Hornet executes a large number of simple GPU kernels as part of its load-balancing whereas, Gunrock executes fewer and more elaborate kernels. While the performance of SXM cards is typically higher than its PCI-E counterpart, it seems that GPU kernel launch is higher for SXM GPUs. Therefore, on the SXM cards, running a larger number of simple kernels introduces a greater overhead. Thus, it seems that it might be desirable to design a load-balancing that requires fewer GPU kernel launches. This is why SXM produces better performance in general in heavy computations on larger graphs due to its amortized launch cost.

VII. Conclusion

References

[1] "Hornet Data Structure repository," https://github.com/hornet-gt/hornet, 2017.

[2] D. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," Georgia Institute of Technology, Tech. Rep., 2009.

[3] S. Beamer, K. Asanovi, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 820–831.

[4] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012, pp. 1–10.

[5] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, 2015.

[6] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, pp. 163–177, 2001.

[7] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516

[8] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[9] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[10] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[11] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. pp. 35–41, 1977.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, vol. 12, 2012.

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 599–613.

[14] O. Green and D. Bader, "Faster Betweenness Centrality Based on Data Structure Experimentation," in *International Conference on Computational Science (ICCS)*. Elsevier, 2013.

[15] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016.*

[16] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[17] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, "igraph: A framework for comparisons of disk-based graph indexing techniques," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 449–459, Sep. 2010.

[18] F. Khorasani, K. Vora, R. Gupta, and L. Bhuyan, "CuSha: Vertex-Centric Graph Processing on GPUs," in *23rd ACM Int'l Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, 2014, pp. 239–252.

[19] K. Madduri, D. Ediger., K. Jiang, D. 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 and Distributed Processing (IPDPS)*, 2009.

[20] 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 *2010 ACM SIGMOD Int'l Conf. on Management of data*, 2010, pp. 135–146.

[21] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray User?s Group (CUG)*, 2010.

[22] NVIDIA, "nvGraph," 2016.

[23] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The Tao of Parallelism in Algorithms," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 12–25, 2011.

[24] J. Riedy, "Updating pagerank for streaming graphs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 877–884.

[25] Y. Shiloach and U. Vishkin, "An O(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.

[26] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *18th ACM SIGPLAN symposium on Principles and practice of Parallel Programming*, 2013, pp. 135–146.

[27] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*

[28] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 265–266.

[29] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: GPU Graph Analytics," *arXiv preprint arXiv:1701.01170*, 2017.

[30] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[32] S. Zhou, K. Lakhotia, S. G. Singapura, H. Zeng, R. Kannan, V. K. Prasanna, J. Fox, E. Kim, O. Green, and D. A. Bader, “Design and Implementation of Parallel PageRank on Multicore Platforms,” in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2017.