Linear Algebra for Graph Processing on GPUs: Opportunities and Limitations

MARZIEH BARKHORDAR, Sharif University of Technology, Iran
MORTEZA RASHIDINIA, Sharif University of Technology, Iran
SHABNAM SHEIKHHA, Sharif University of Technology, Iran
NEGIN MAHANI, Sharif University of Technology, Shahid Bahonar University, Iran
SAMIRA HOSSIEN GHORBAN, Institute for Research in Fundamental Sciences (IPM), Iran
MOHAMMAD SADROSADATI, Institute for Research in Fundamental Sciences (IPM), Iran
HAMID SARBAZI-AZAD, Sharif University of Technology, IPM, Iran

Graphics Processing Units (GPUs) show great potential in exploiting the inherent parallelism of graph applications. However, issues such as high global synchronization demand across running threads, irregular memory access patterns, and load imbalance, make graph processing a challenge on GPUs. Several pieces of related work propose various GPU-based frameworks relying on different parallel programming models to effectively implement graph algorithms in GPUs. However, each of these frameworks targets a few graph processing issues, and unfortunately, none of them can completely address all graph processing issues. Linear algebra is a powerful paradigm that can potentially address graph processing challenges by employing a sequence of primitive matrix operations due to more regular operations in matrix operations. Both industry and academia develop several linear algebraic libraries (e.g., nvGRAPH, GraphBLAST, GBTL) for implementing graph applications in GPUs. However, to the best of our knowledge, there is no prior work that comprehensively studies the opportunities and limitations of linear algebra in graph processing from a GPU architectural view.

In this paper, we aim to (1) characterize the performance of linear algebraic graph processing in GPUs, (2) comprehensively study the reasons behind its performance improvement and degradation with respect to non-linear-algebraic implementations, and (3) make several key insights and solutions to mitigate the performance limiters of linear algebraic graph processing. To this end, we characterize six well-known graph algorithms using 160 real-word datasets on a real machine and a GPU simulator (i.e., Accelsim). Based on our findings, we discuss potential hardware/software research directions for performance improvement in graph analysis. As a case study, we devise two software-based optimization techniques that reduce the number of executed instructions through incorporating the algorithm semantics into the matrix operations. Our experimental results show up to 39.2× (8.9× on average) speed up using the proposed optimization techniques.

Additional Key Words and Phrases: GPUs, Graph Processing, Linear Algebra

1 INTRODUCTION

Many real-life applications, e.g., social networks [8, 108], molecule structures in chemistry [64], and computational linguistics [52, 87, 92], use graphs as a form of data representation. The past few years have witnessed rapid growth in the scale of graph problems, with the number of nodes and edges reaching an order of billions. Large-scale graph processing outgrows the computational capacity and memory bandwidth provided by single-core processors. Hence, modern approaches are increasingly turning towards parallel processors.

Graphics Processing Units (GPUs) are a form of massively parallel processors that employ many cores to process large amounts of data in parallel [95, 102]. Due to their throughput-oriented architecture and high memory access bandwidth,

Authors' addresses: Marzieh Barkhordar, Sharif University of Technology, Tehran, Iran, marzieh.barkhordar@gmail.com; Morteza Rashidinia, Sharif University of Technology, Tehran, Iran, mortezarashidi.mr@gmail.com; Shabnam Sheikhha, Sharif University of Technology, Tehran, Iran, shablandelina@gmail.com; Negin Mahani, Sharif University of Technology, Shahid Bahonar University, Tehran, Iran, negin.mahani@gmail.com; Samira Hossien Ghorban, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, s.hosseinghorban@ipm.ir; Mohammad Sadrosadati, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, m.sadr89@gmail.com; Hamid Sarbazi-Azad, Sharif University of Technology, IPM, Tehran, Iran, azad@sharif.edu.

these platforms are highly effective for applications with properties such as fine-grained parallelism, load-balanced work distribution, regular memory access patterns, etc [22, 83, 94]. In contrast, graph applications rarely exhibit these properties in real life; therefore, designing efficient implementations that fully exploit the high level of parallelism in GPUs is not a trivial task for these applications [121, 136]. This makes efficient graph processing on GPUs a major challenge that entails many optimization opportunities.

Several pieces of related work propose various GPU-based frameworks relying on different parallel programming models, such as edge-/vertex-based [37], data-centric [130], message-passing [148], and Gather-Apply-Scatter [40, 73], to effectively implement graph algorithms on GPUs. We refer to these frameworks as *non-linear-algebraic* implementations hereinafter. However, prior work shows that each of these frameworks targets a few graph processing challenges, and, unfortunately, none of them can completely address all graph processing challenges [40, 43, 50, 73, 84, 117, 118, 120, 122, 130, 146, 148].

A powerful paradigm for addressing the challenges of graph processing on GPUs is linear algebra. Linear algebraic approaches map graph applications to GPUs by employing a sequence of primitive matrix operations [69]. These primitive operations are relatively load balanced and can utilize GPU memory hierarchy more efficiently. Several pieces of prior work propose linear-algebraic implementations for specific graph algorithms (e.g., BFS) [17] and different frameworks to support a broader range of graph algorithms, such as GraphBLAST [140], GBTL [145], and nvGRAPH [25]. However, to the best of our knowledge, no prior work comprehensively studies the opportunities and limitations of linear algebra in graph processing from a GPU architectural view In this paper, we aim to (1) characterize the performance of linear algebraic graph processing in GPUs, (2) comprehensively study the reasons behind its performance improvement and degradation with respect to non-linear-algebraic implementations, and (3) make several key insights and solutions to mitigate the performance limiters of linear algebraic graph processing.

To this end, we design an experimental characterization in four steps. First, we select six well-known graph algorithms from the cuGRAPH/nvGraph library for linear algebraic implementations, and the Tigr and Gunrock libraries as state-of-the-art libraries for non-linear-algebraic implementations. Second, we select 160 real-word datasets across a wide variety of application domains. In addition to these real-world datasets, we perform a few experiments on synthetic datasets based on two well-known random graph generation models, Watts-Strogatz [132] and R-MAT [21], to come to a deeper understanding of our observations. Third, we perform extensive execution profiling on a real machine and the state-of-the-art GPU simulator (Accelsim [71]) to extract important performance counters. Fourth, we analyze these metrics to determine the behavioral patterns of linear algebraic kernels based on dataset and algorithm characteristics.

We observe that whether linear algebra can outperform non-linear-algebraic implementations in terms of execution time depends on the application and dataset characteristics. To come to a deeper understanding of this behavior, we break the execution time into two main factors, Instructions Per Cycle (IPC) and number of executed instructions. We make three key observations. First, we observe that linear algebraic implementations usually experience higher IPC results as they execute more ALU instructions and better utilize the memory hierarchy. Second, although graph algorithms are well-known to be memory-bound, linear algebraic implementations are usually compute-bound (as determined by the Roofline charts [34]), mainly due to the better temporal and spatial locality in memory accesses. Third, despite achieving a higher IPC, linear algebraic kernels suffer from a significant increase in the number of instructions, which, in some cases, offsets the benefits of the higher IPC. This is because in non-linear-algebraic implementations, the instruction count in each iteration is attributed to the active nodes and their incoming edges. On the opposite, linear-algebraic implementations multiply the adjacency matrix by a vector/matrix over a semiring in each iteration. As a result, the instruction count in each iteration is attributed to the total node and edge counts of the graph.

Our study enables and empowers several hardware/software research directions and visions to improve the performance of linear algebraic graph processing in GPUs, including reducing the number of instructions, improving instruction throughput, and scalable implementations for extremely large (i.e., giant) graphs. We discuss these research directions in the paper. As a case study, we devise some software-level optimization techniques to reduce the number of executed instructions in the Single-Source Shortest Path (SSSP) algorithm. Each step in SSSP algorithm is dual with a matrix-vector multiplication with a semiring of extended real numbers with element-wise addition and reduction minimum operations, replacing the element-wise multiplication and reduction sum, respectively (See § 2 for more detail on linear algebraic implementation and semirings). We devise two techniques to reduce the instruction count through incorporating the algorithm semantics into the matrix-vector multiplication. First, we focus on reducing the number of element-wise operations. In SSSP algorithm, multiplying each row of the adjacency matrix by the input vector is equivalent to relaxing the incoming edges of the vertex corresponding to that row [69].[1] If a neighbor of a vertex has not been updated in the previous iteration, we can ignore relaxing the edge connected to that neighbor in the current iteration, and hence, removing redundant element-wise operations from the matrix-vector multiplication. This approach not only reduces the number of element-wise operations but also can shorten the depth of the reduction tree. Second, we incorporate a *worklist* idea from Tigr implementations [99] (collecting a set of active nodes in each iteration, and processing only the active ones), to the matrix-vector multiplication. We remove the computation of rows corresponding to the non-active nodes from the matrix-vector multiplication in each iteration. These two optimizations for reducing the instruction count are not free and can negatively affect the instruction throughput mainly due to reducing the regularity of operations. We observe that these two optimization techniques can reduce the number of instructions by about 250×, on average, at the price of 66% IPC overhead compared to the baseline linear-algebraic implementation. Overall, we observe up to 39.2× (8.9× on average) speed up using the proposed optimization techniques. The trade off between the instruction counts and instruction throughput should be explored deeply to optimize the performance of the linear algebraic implementations. This paper makes the following contributions:

- We characterize the performance of linear algebraic graph implementations, compared to non-linear-algebraic ones, and find out graph and algorithm features that make one formulation preferred over the other one.
- We perform an extensive execution profiling to understand the reasons behind performance improvement or degradation. We make a key observation that although linear algebraic implementations usually exhibit a higher IPC, the number of executed instructions can offset the gained benefits since the overall execution time depends on both metrics.
- We discuss thoroughly our experimental results and observations, and present several potential research directions, such as reducing the number of executed instructions, improving instruction throughput, and scalable implementations for giant graphs.
- As a case study, we devise some software-level optimization techniques to reduce the number of executed instructions in linear-algebraic graph implementations, and observe up to 39.2× (8.9× on average) speed up.

2 GRAPH ALGORITHMS USING LINEAR ALGEBRA

Many challenges arise when processing graph applications on GPUs. A few of the most important are as follows: First, many graph applications incorporate a Bulk Synchronous Parallel Model [15, 62, 130, 136], in which each iteration is dependent on the results of the previous one, therefore, a global synchronization is needed at the end of each iteration to

[1] relaxing process of an edge (u,v) refers to updating the distance of vertex v from the source node, if the distance becomes shorter by including the edge (u,v) in the path of the source node to the vertex v.

guarantee its completion. As GPUs do not support global synchronization among SMs, graph applications rely on kernel invocations and returning control to the CPU, which results in a high CPU intervention rate. Second, the irregular memory access patterns lead to a high cache miss rate. This ineffective cache usage leads to noticeable memory latency and bandwidth usage [41, 42, 76]. Third, graph applications are unable to effectively leverage shared memory due to the insufficient amount of data reuse [136]. Fourth, the amount of processing required for each graph vertex is heavily dependent on its degree, which leads to a considerable load imbalance [58, 65, 73, 76, 118, 136, 150].

One potential solution is using linear algebra. Linear algebraic approaches map graph applications to GPUs by performing a sequence of primitive matrix operations. These operations can better exploit shared memory and execute in load-balanced manner [19, 32, 33, 67–69, 88]. As an example, each step in Breadth-First Search (BFS), a graph traversal algorithm, is dual with a matrix-vector multiplication. To perform the traversal starting from the s-th vertex, a vector v_s is initialized such that $v_s = 1$. By performing the operation $v = A^T.v$ repeatedly, the non-zero elements in vector v represent the visited vertices in each step. However, to implement a broader range of graph algorithms, the traditional matrix-vector/matrix multiplication is not sufficient. We can overcome this limitation by using a broader definition of matrix and vector/matrix operations, called semiring [6, 23, 35, 36, 45, 46, 69]. A semiring is a set of elements with two operations, mainly called addition and multiplication, denoted by $\oplus$ and $\otimes$, respectively. The main properties of a semiring are: (1) $\oplus$ and $\otimes$ are associative, (2) $\oplus$ is commutative, and (3) $\otimes$ is distributive over $\oplus$. As an example, the Single Source Shortest Path (SSSP) algorithm is dual with a semiring of extended real numbers with operations minimum and addition, replacing the traditional operations of addition and multiplication, respectively.

Several pieces of prior work propose linear-algebraic implementations for specific graph algorithms (e.g., BFS) [17] and different frameworks to support a broader range of graph algorithms, such as GraphBLAST [140], GBTL [145], and nvGRAPH [25]. None of these works investigates the opportunities and limitations of linear algebra in graph processing from a GPU architectural point of view. We aim to (1) comprehensively study the advantages and disadvantages of linear algebra-based graph processing on GPU by performing extensive execution profiling on a real machine and a GPU simulator, and (2) make several key insights and solutions to improve the performance of graph processing in GPUs.

3 METHODOLOGY

3.1 Graph Applications

We choose six well-known graph algorithms, including Breadth-First Search (BFS), Single Source Shortest Path (SSSP), PageRank (PR), Betweenness Centrality (BC), Triangle Counting (TC), and Graph Clustering (GC). These algorithms are widely-used, with many applications in real life [24, 74, 75, 128]. A few examples are: BFS, a graph traversal algorithm, is useful for finding neighboring nodes in a peer-to-peer network [66, 138, 139]; SSSP is useful in road networking, namely, automatic detection of best directions between physical locations, as offered by mapping services [2, 3, 126]; PageRank is one of the best-known algorithms for ranking web pages [104]; Betweenness Centrality [39] is widely used in network theory, in particular social networks, as a measure of the control a node has over information flow in a network [10, 97]; TC and GC are mainly used in network analysis for evaluating the community structure of social networks [96]. Of these six algorithms, we choose three, which exemplify the trends we observe in the others, to study thoroughly. Table 1 summarizes the operations used in each algorithm along with its semiring.

For each algorithm, we need to choose a **non-linear-algebraic** and a **linear algebraic** parallel implementation. For **non-linear-algebraic** implementations, there are various GPU-based frameworks which employ different parallel

Table 1. Algorithms we use along with their semirings.

Algorithm	Semiring	Operations
BFS	0, 1	| &
SSSP	$R \cup \infty$	min +
PR	R	+ ×
BC	R	+ ×
TC	R	+ ×
GC	R	+ ×

Table 2. GP100 micro-architectural specs.

Streaming Multiprocessors	56
CUDA Cores (single precision) per SM	64
Base Clock (MHz)	1328 MH
Register File Size	256 KB per SM
Shared Memory Size	64 KB per SM
L2 Cache Size	4096 KB
Memory	16GB HMB2
Memory Bandwidth	732 GB/s
Fabrication Process	16 nm
Thermal Design Power (TDP)	300 Watts

programming models. A few examples are: (1) Medusa [148] (message-passing parallel model), (2) Totem [43] (GPU-CPU hybrid framework), (3) Frog [120] (lock-free semi-asynchronous parallel model), (4) VertexAPI2 [37] (vertex-centric, Gather-Apply-Scatter parallel model), (5) CuSha [73] (Gather-Apply-Scatter parallel model), (6) MapGraph [40] (modified Gather-Apply-Scatter), (7) Ligra [122] (CPU-GPU hybrid graph traversal parallel model), (8) Gunrock [131] (bulk-synchronous, data-centric parallel model), and (9) Tigr [99] (a graph transformation framework). We use Tigr as much as possible because (1) it effectively reduces the irregularity of graphs by using a lightweight virtual transformation scheme; and (2) it employs different optimization strategies such as *worklist*, *edge-array coalescing*, and *synchroniztion relaxation* for memory efficiency and decreasing instruction count; therefore, it achieves significant and consistent speedup over the state-of-the-art GPU graph processing frameworks. For TC and PR algorithms, we use Gunrock, since Tigr either not implements or offers lower performance. For **linear algebraic** implementations, we use cuGraph [109], available as part of the RAPIDS [116], due to its better performance results compared to GraphBLAST [140] and GBTL [145]. Note that the cuGraph includes algorithms from nvGRAPH[25].

From here on, for simplicity, we refer to linear algebraic and non-linear-algebraic implementations as *LinAlg* and *NonLinAlg*, respectively. As for the individual algorithms, we respectively refer to the linear algebraic and non-linear-algebraic implementation of BC as *LinAlg-BC* and *NonLinAlg-BC*, and so on.

3.2 Tools

Real Machine. To study graph applications on real GPUs, we choose NVIDIA GP100 GPU [100]. Table 2 reports the micro-architectural details of our GPU. We install the GPU on a server with an Intel Xeon E3-1270 3.40-3.80 GHz processor and 32GB of DDR4 main memory. The system uses Ubuntu 18.04 with version 5.4.0 of the Linux kernel stored in a 500 GB Western Digital HDD. The implementations are in CUDA 10.0 we use nvprof tool [27] for profiling .

Table 3. The characteristics of the example 10 datasets

Datasets	Abbreviation	Network Domain	#Nodes	#Edges	Density	BFS Tree Depth
bio-mouse-gene	bio-mg	Biological	45101	14506196	321.6	8
ca-coauthors-dblp	ca-dblp	Collaboration	540486	15245729	28.2	14
DIM-MANN-a81	DIM-M81	DIMACS	3321	5506380	1658.04	2
frb100-40	frb100-40	BHOSLIB	4000	7425226	1856.3	2
ia-wiki-Talk-dir	ia-wTalk	Interaction	2394385	4659565	1.9	7
labled-CL-10M-1d8-L5	lab-CL	labeled	10000000	43985360	4.3	11
road-roadNet-CA	road-CA	Road	1957027	2760388	1.4	554
sc-ldoor	sc-ldoor	Scientific Computing	952203	20770807	21.8	177
soc-twitter-follows	soc-twitter	Social	404719	713319	1.7	7
web-BerkStan	web-BS	Web	685230	6649470	9.7	123

Simulator. To study how some architectural changes in GPUs can affect the performance of graph algorithms, we use a state-of-the-art GPU simulator, Accelsim [71].

3.3 Datasets

Graphs have many applications in real world systems, such as social networks, road networks, and the world wide web. We pick 160 datasets over a wide range of real-world domains [110]. Table 3 lists the information of example 10 datasets, including their name, application domain, an abbreviation for the name, edge count, node count, density (ratio of edge count to node count), and BFS tree depth (level count of BFS tree from source node to leaf nodes). To perform a deeper analysis of a few observations, we generate synthetic datasets with modifiable properties using two well-known random graph models, Watts-strogatz [132] and R-MAT [21]. (see § A in Appendix for detail of these synthetic datasets).

3.4 Metrics

Two key metrics contribute to the execution time of an application are instruction count and IPC. To determine these components, we use nvprof [27] to obtain a set of relevant architectural metrics. Instruction count is independent from the underlying architecture, therefore, we only need to examine its corresponding metric (`inst_executed`). However, the same cannot be said about IPC. Different architectural metrics affect IPC depending on whether the application is restricted by memory throughput or instruction throughput. We use the Roofline model [34] to figure out whether the execution is *compute-bound* or *memory-bound*. Table 4 provides an explanation for the first-level metrics affecting compute-/memory-bound kernels.

Roofline Model. The Roofline model is a model that visualizes the performance of a kernel. The model's purpose is to identify the performance bottlenecks of a kernel executing on multi-/many-core platforms. In this study, we use the instruction Roofline model for GPUs [34]. This model draws a plot consisting of two ceilings derived from architectural properties. The first ceiling is derived from the processor's peak performance and the second ceiling is derived from the memory bandwidth. The processor's peak performance is the theoretical maximum (warp-based) instructions per second, which we estimate as $56(SM) \times 2(warp\ scheduler) \times 1(instruction/cycle) \times 1.328(GHz) = 148.7\ GIPS$. To estimate the peak memory bandwidth we divide the total bandwidth (in GTXN/s) by the transaction size [34, 141]. The bandwidth and transaction size for HBM2 are 732 GB/s and 32 bytes, respectively; therefore, the peak memory bandwidth is 22.8 GTXN/s.

Table 4. Metrics of compute-/memory-bound analysis

Type	Metrics	Description
Compute-bound	inst_fp_32	Number of single-precision floating-point instructions executed by non-predicated threads
	inst_integer	Number of integer instructions executed by non-predicated threads
	inst_bit_convert	Number of bit-conversion instructions executed by non-predicated threads
	inst_fp_64	Number of double-precision floating-point instructions executed by non-predicated threads
	inst_compute_ld_st	Number of compute load/store instructions executed by non-predicated threads
	stall_exec_dependency	Percentage of stalls occurring because an input required by the instruction is not yet available
	stall_sync	Percentage of stalls occurring because the warp is blocked at a __syncthreads() call
	shared_ld_bank_conflict	Number of shared load bank conflict generated when the addresses for two or more shared memory load requests fall in the same memory bank
	shared_st_bank_conflict	Number of shared store bank conflict generated when the addresses for two or more shared memory store requests fall in the same memory bank
	branch	Number of branch instructions executed per warp on a multiprocessor
	divergent_branch	Number of divergent branches within a warp
	hared_load_transactions	Number of shared memory load transactions
	shared_store_transactions	Number of shared memory store transactions
	gld_transactions	Number of global memory load transactions
	gst_transactions	Number of global memory store transactions
	atomic_transactions	Global memory atomic and reduction transactions
Memory-bound	gld_throughput	Global memory load throughput
	gst_throughput	Global memory store throughput
	gld_efficiency	Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage
	gst_efficiency	Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage
	shared_efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage

Figure 1 plots the logarithmic Roofline chart for our GPU (see § 3.2 for the details of our GPU). The x-axis and y-axis represent warp instructions per transaction (Instruction Intensity) and Giga instructions per seconds (Performance), respectively. To determine whether a kernel is compute-/memory-bound we calculate its performance and instruction intensity and see where in falls in the plot. If it falls under the peak performance ceiling (horizontal), it is compute-bound; if it falls under the peak memory bandwidth ceiling (diagonal) ceiling, it is memory-bound.

To draw the kernel, the Roofline model defines a kernel's performance (GIPS) as a function of peak machine bandwidth (GTXN/s), Instruction Intensity, and theoretical maximum (warp-based) instructions per second. Equation 1 defines this relation.

$$GIPS \leq min \begin{cases} Peak\ GIPS \\ Peak\ GTXN/s \times Instruction\ Intensity \end{cases} \tag{1}$$

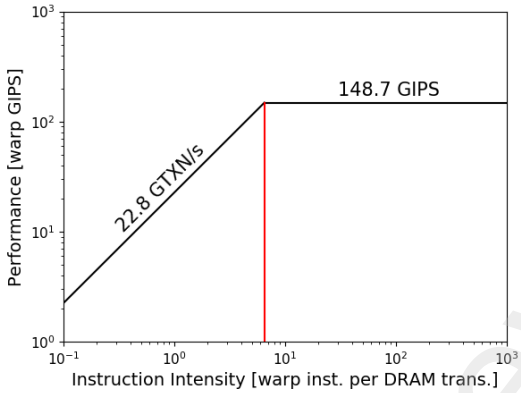


Fig. 1. Instruction Roofline model

The performance and Instruction Intensity of a given kernel are respectively described in Equations 2 and 3. In these equations `inst_executed_thread`/32 is the number of warp instructions executed by each kernel and DRAM transactions is the sum of `dram_read_transactions` and `dram_write_transactions`.

$$Performance = \frac{inst\_executed\_thread/32}{1e^9 \times run\_time} \tag{2}$$

$$Instruction\ Intensity = \frac{inst\_executed\_thread/32}{DRAM\ Transactions} \tag{3}$$

Compute-bound Metrics. In general, compute-bound kernels are limited by ALU instruction mix and instruction serialization. Different ALU instructions (e.g., integer and floating-point operations) have different throughputs. Minimizing the use of low-throughput arithmetic instructions in a kernel leads to maximizing its compute throughput. Five factors cause instruction serialization: (1) Warp divergence causes the threads in a warp to issue the same instruction sequentially. We estimate this overhead by dividing divergent branch count (`divergent_branch`) by total branch count (`branch`), expressed as percentage. (2) Bank conflict occurs when two or more threads in a warp access addresses in the same shared memory bank. We estimate this overhead by dividing the load/store bank conflict count (`shared_ld/st_bank_conflict`) by total shared memory load/store transaction count (`shared_load/store_transactions`), expressed as percentage. (3) Atomic operations stall the other threads until completion. We estimate this overhead by dividing the atomic transaction count (`atomic_transactions`) by total global transaction count (`gld/gst_transactions`), expressed as percentage. (4) Execution dependency occurs when instruction operands are dependent on each other. We estimate this overhead with the `stall_exec_dependency` metric. (5) Synchronization instructions are a necessary barrier that prevent the progress of warps. We estimate this overhead with the `stall_sync` metric.

Memory-bound Metrics. In general, memory-bound kernels are limited by number of concurrent accesses in flight and memory access patterns. In the case of memory access concurrency, high independent accesses per thread increase the chance of achieving sufficient accesses in flight. Therefore, by utilizing the memory bandwidth, we expect the memory throughput to be closer to the theoretical throughput. The optimal memory access pattern is accessing data in a coalesced way. We can divide sub-optimal patterns into four categories: (1) offset warp addresses that are not aligned on cache line boundary; (2) intra-warp addresses at large regular distances (i.e., greater than a word); (3) accessing large

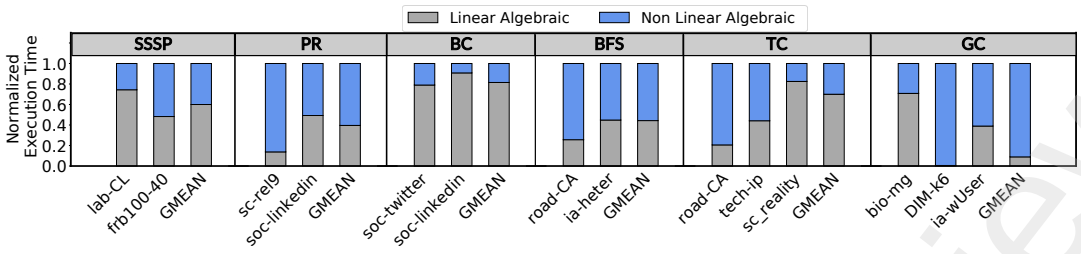


Fig. 2. Normalized execution time

contiguous regions of memory (several words in size) by each thread; and (4) intra-warp accesses to many cache lines in irregular access patterns. In all cases, the number of transferred bytes is higher than application requirements, resulting in wasted bandwidth. We can determine access patterns through memory efficiency. Memory efficiency is the ratio of requested memory throughput by the application (bytes requested by the app) to actual memory throughput achieved by the hardware (bytes transferred by the hardware). These two values can differ for two reasons. First, application does not use all the bytes transferred due to a sub-optimal access pattern. Second, when all threads within a warp access the same word in a bank, an operation called broadcast is performed. In this case, the application has only made one request, however, the total bytes transferred is multiplied by the number of threads. To estimate memory efficiency, we use three metrics `gld_efficiency`, `gst_efficiency`, and `shared_efficiency`.

4 HIGH-LEVEL PERFORMANCE EVALUATION

To get a high-level grasp of the effects of linear algebra in graph processing in terms of performance, in this section, we perform a comparison between *NonLinAlg* and *LinAlg* on numerous real-world datasets in various domains (see § 3.4 for details of our methodology).

Figure 2 compares the *NonLinAlg* and *LinAlg* of six graph algorithms: BFS, SSSP, BC, TC, PR, and GC, in terms of application execution time. The x-axis, y-axis, and legend represent the selected datasets, the normalized execution time (normalized with respect to the sum of *LinAlg* and *NonLinAlg* execution time for each dataset), and implementation type (*LinAlg* and *NonLinAlg*), respectively. In this chart, for each algorithm, we only represent the results of few representative datasets for a set of datasets with similar trends and the normalized geometric mean of two types of implementations between all selected datasets. We make two key observations. *First*, *LinAlg* does not always improve the average performance of an algorithm. As an example, we can see that compared to the *NonLinAlg*, *LinAlg* behaves poorly for most of the datasets profiled in BC. *Second*, the level of performance improvement or degradation varies depending on algorithm and dataset characteristics. For example, *LinAlg* greatly improves the performance for `sc-rel9` in PR, whereas we observe only a slight improvement for `ia-wUser` in GC. *LinAlg* also severely degrades the performance for `sc_reality` in TC. Moreover, for a number of datasets, such as `ia-heter` in BFS, the performance is relatively the same for both implementations.

To summarize, we have observed that *LinAlg* is not always superior to *NonLinAlg*. Not only this but also, in some cases, it severely downgrades the execution time of the application. Therefore, a thorough study for examining the limits and potentials of linear algebra in graph processing on GPUs is missing. We have three key goals. First, we provide a reasoning for the performance observations by performing a detailed, in-depth profiling of these applications, examining the dataset characteristics, and finding patterns that relate them. Second, we aim to come to an understanding

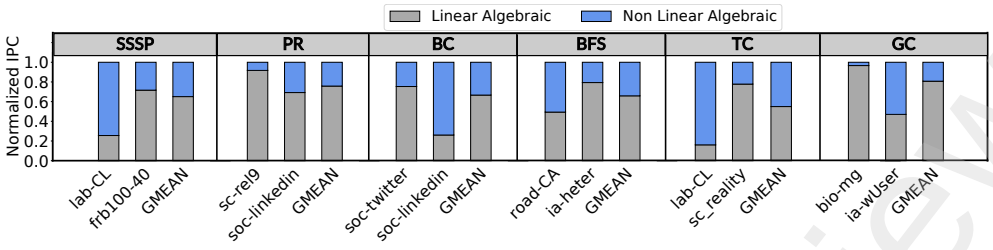


Fig. 3. Normalized IPC results

of limits and bottlenecks in application execution. In the end, we make a few key insights for potential optimizations regarding the programming and/or underlying architecture.

5 DETAILED PERFORMANCE ANALYSIS

We analyze the performance results via detailed profiling metrics for six graph algorithms, including BC, PR, SSSP, BFS, TC, and GC. To be able to profile the execution behavior in a reasonable amount of time, we choose a minimal set of kernels that collectively take over most (e.g., 80%) of the computation time, called as representative kernels. A kernel's execution time depends on IPC and the total number of executed instructions. We incorporate the number of kernel invocations in its instruction count. The rest of this section focuses on analyzing these two factors.

5.1 IPC Analysis

Figure 3 plots the IPC of *LinAlg* and *NonLinAlg* for six algorithms. For each algorithm, we report the average result and the results of some representative datasets. We make four key observations. *First*, in most cases, *LinAlg* achieves a higher IPC compared to *NonLinAlg*. *Second*, this improvement varies depending on algorithm and dataset. *Third*, in some cases, the IPC for *LinAlg* is almost equal to *NonLinAlg*. `sco-linkedin` in BC algorithm is a good example of such behavior. *Fourth*, in a few cases, *NonLinAlg* achieves a higher IPC than *LinAlg*. `road-CA` in both BFS and TC is a good example that *NonLinAlg* outperforms *LinAlg* in terms of IPC. We aim to justify these observations through detailed profiling. To this end, we draw the Roofline model for all applications to determine whether they are limited by memory or compute throughput. The Roofline plot consists of two ceilings, one derived from the processor's peak performance, and the other derived from the memory bandwidth. Based on where the kernels fall (i.e., under which ceiling), we can determine if they are compute-/memory-bound [34].

Table 5 (a)-(b) report the average instruction intensity of compute-bound and memory-bound datasets for two implementations of each algorithm. This table also reports the percentage of compute-/memory-bound datasets to the total 160 datasets. We make three key observations. *First*, although graph applications are well-known to be memory-bound [4, 5, 44], we observe that *LinAlg* is usually compute-bound. *Second*, depending on the dataset characteristic, *LinAlg* and *NonLinAlg* can be compute-/memory-bound. For further analysis, we generate synthetic datasets using the Watts-Strogatz [132] and R-MAT [21] graph generation model. (see Appendix B.1 for details of our experiments and analysis). We make three key observations: (1) Graph density, i.e., edge count to node count ratio, plays a significant role in making an application compute-/memory-bound. In general, increasing the density shifts the application from memory-bound regions to compute-bound regions. (2) *LinAlg* has significantly lower density threshold values in which it places into compute-bound region, marking the key point that *LinAlg* is more likely to be compute-bound under

Table 5. Average instruction intensity of memory/compute-bound datasets for six algorithm: (a) SSSP, PR, and BC; (b) BFS, TC, and GC

(a)

	LinAlg-SSSP	*NonLinAlg-SSSP*	*LinAlg-PR*	*NonLinAlg-PR*	*LinAlg-BC*	*NonLinAlg-BC*
Mem (%)	29.6	100	22.3	100	29.6	100
Comp (%)	70.4	-	77.8	-	70.4	-
Intensity (Mem)	3.7	2.0	3.6	3.4	3.7	2.3
Intensity (Comp)	7.5	-	8.0	-	7.3	-

(b)

	LinAlg-BFS	*NonLinAlg-BFS*	*LinAlg-TC*	*NonLinAlg-TC*	*LinAlg-GC*	*NonLinAlg-GC*
Mem (%)	-	100	46.2	69.3	71.4	100
Comp (%)	100	-	53.8	30.7	28.6	-
Intensity (Mem)	-	2.4	2.6	1.9	3.3	1.1
Intensity (Comp)	43.5	-	575.3	177.8	32.8	-

different datasets. (3) Graph density is not the only contributing factor in compute-/memory-bound. In some cases, applications with low-density datasets are also compute-bound, especially for *LinAlg*. This is due to the distribution of non-zero elements in the adjacency matrix. In particular, if the non-zero elements of a low-density graph are distributed in a concentrated manner, the application is more likely to be compute-bound, especially for *LinAlg*. This is mainly because that spatial locality exploits the memory hierarchy more efficiently, significantly decreasing DRAM transactions. In contrast, if non-zero elements of a low-density graph are distributed with high randomness, the execution is more likely to be memory-bound. *Third*, memory-bound datasets in *LinAlg* are usually close to the boundary between compute-/memory-bound regions. As a result, we can shift memory-bound *LinAlg* applications to the compute-bound region by making some minor hardware/software improvements (see § 6 for more discussion).

The rest of this section focuses on analyzing compute-/memory-bound applications through detailed profiling metrics (see § 3.4 for more details about our selected metrics).

5.1.1 Memory-bound Analysis. Table 6 (a)-(b) report IPC averaged across memory-bound datasets for six algorithms. We observe that none of the two implementations is always superior to another for memory-bound datasets in terms of IPC. For example, *LinAlg-GC* gains 3× improvement compared to *NonLinAlg-GC*, while *NonLinAlg-SSSP* improves IPC by 1.10× compared to *LinAlg-SSSP*. Two main factors limit the IPC for memory-bound kernels are (1) memory access concurrency and (2) memory access patterns. Note that we do not compare *LinAlg-BFS* to *NonLinAlg-BFS* in this section since *LinAlg-BFS* is always compute-bound.

Table 6. Average IPC of compute-/memory-bound datasets for six algorithm: (a) SSSP, PR, and BC; (b) BFS, TC, and GC

(a)

IPC	*LinAlg-SSSP*	*NonLinAlg-SSSP*	*LinAlg-PR*	*NonLinAlg-PR*	*LinAlg-BC*	*NonLinAlg-BC*
Compute-bound	1.21	-	0.91	-	1.25	-
Memory-bound	0.40	0.46	0.44	0.26	0.38	0.42

(b)

IPC	*LinAlg-BFS*	*NonLinAlg-BFS*	*LinAlg-TC*	*NonLinAlg-TC*	*LinAlg-GC*	*NonLinAlg-GC*
Compute-bound	0.98	-	1.26	1.30	1.43	-
Memory-bound	-	0.45	0.57	0.47	0.30	0.10

Table 7. Average global throughput and global transactions of memory-bound datasets for six algorithm: (a) SSSP, PR, and BC; (b) BFS, TC, and GC

(a)

Metrics	*LinAlg-SSSP*	*NonLinAlg-SSSP*	*LinAlg-PR*	*NonLinAlg-PR*	*LinAlg-BC*	*NonLinAlg-BC*
Global Transactions (GB)	2.3	0.44	1.03	0.89	2.2	0.4
Global Throughput (GB/s)	269.4	435.5	495.4	132.1	269.3	284.2

(b)

Metrics	*LinAlg-BFS*	*NonLinAlg-BFS*	*LinAlg-TC*	*NonLinAlg-TC*	*LinAlg-GC*	*NonLinAlg-GC*
Global Transactions (GB)	-	0.29	4.5	4.1	1.0	7.0
Global Throughput (GB/s)	-	356.6	105.7	232.7	94.5	64.9

Table 8. Average memory efficiency of memory-bound datasets for six algorithm: (a) SSSP, PR, and BC; (b) BFS, TC, and GC

(a)

Memory Efficiency	*LinAlg-SSSP*	*NonLinAlg-SSSP*	*LinAlg-PR*	*NonLinAlg-PR*	*LinAlg-BC*	*NonLinAlg-BC*
Global Load	34.8	38.4	24.3	59.2	33.8	49.2
Global Store	70.9	4.9	26.8	14.8	70.9	12.9
Shared Memory	36.1	0	16.3	6.4	35.3	0

(b)

Memory Efficiency	*LinAlg-BFS*	*NonLinAlg-BFS*	*LinAlg-TC*	*NonLinAlg-TC*	*LinAlg-GC*	*NonLinAlg-GC*
Global Load	-	47.4	14.4	82.62	71.5	21.8
Global Store	-	17.4	61.7	42.87	46.8	25.1
Shared Memory	-	0	8.3	0	32.6	0

To study memory access concurrency, we report global memory transactions and throughput in Table 7 (a)-(b) for six algorithms. In general, we expect higher global memory throughput for *LinAlg*. As an example, we observe that *LinAlg-PR* achieves roughly 3× higher throughput for servicing its 1.4× more global transactions compared to *NonLinAlg-PR*. This is because (1) matrix operations involve more regular accesses; therefore, *LinAlg* can utilize the cache better; and (2) a few of the global memory transactions in *NonLinAlg* are global atomic transactions, which significantly reduce the concurrency among transactions. However, *LinAlg* does not always provide higher global memory throughput compared to *NonLinAlg*. As an example, *NonLinAlg-SSSP* achieves higher throughput with lower number of global transactions compared to *LinAlg-SSSP*. Looking into each dataset results in addition to the average results shows that while most memory-bound datasets in *LinAlg-SSSP* experience higher global memory throughput than *NonLinAlg-SSSP*, there are a few memory-bound datasets, such as the `lab-CL`, that *LinAlg-SSSP* cannot provide high global memory throughput. In such cases, *LinAlg* severally has lower IPC compared to *NonLinAlg*. In the case of `lab-CL`, IPC of *LinAlg* and *NonLinAlg* are 0.1 and 0.3, respectively. This is mainly due to the fact that these datasets are huge with extremely random distribution of non-zero elements in the adjacency matrix. Therefore, the spatial locality becomes low and *LinAlg-SSSP* cannot utilize the caches efficiently. Note that we can make similar observations for TC and BC.

To study memory access pattern such as scattered or misaligned patterns, we examine memory efficiency. We estimate memory efficiency as the ratio of requested memory throughput (i.e., bytes requested by the application) to the achieved memory throughput (i.e., bytes transferred by the hardware). Table 8 (a)-(b) report the average global load/store and shared memory efficiency results. We make three key observations. To provide explanations, we design several experiments for each observation (see Appendix B.2 for details) and briefly describe our findings here.

First, *NonLinAlg* does not use shared memory in all algorithms except PR, while *LinAlg* use shared memory for all intermediate matrix operations (e.g., element-wise operations, reduction) and only the final results are written to global memory. Since the shared memory is much faster than global memory, using this memory in implementation improves IPC. We also observe that although *LinAlg* makes extensive use of shared memory, its shared efficiency is low (less than 40%). Shared efficiency is limited by (1) bank conflict and (2) broadcasting. Bank conflict occurs when multiple threads in a warp access addresses that fall in the same memory bank. In this case, conflicting memory requests are split into as many separate conflict-free requests as necessary. Broadcasting occurs when all threads within a warp access the same word in a shared memory bank. In this case, from the application's point of view, only one word is requested. However, to service all threads in a warp, several extra words (e.g., 32 words for a utilized warp) are transferred. We perform some experiments using synthetic datasets to figure out which of these two factors affect the shared memory efficiency more. Our experimental results show that (1) increasing the density from low to high shifts the application from 2-way bank conflict to no bank conflict, and (2) in high-density datasets with no bank conflict, the shared efficiency is about 60%. This indicates that low shared memory efficiency in *LinAlg* is mostly due to the broadcasting. To estimate how much broadcasting negatively impacts IPC, we compare the IPC results of two extreme cases: maximum broadcasting (i.e., all threads of warp access the same word) and no broadcasting (i.e., all threads access different shared memory banks with no bank conflict). We observe about 28% higher IPC in the second case, underscoring that modern GPUs do not implement broadcasting efficiently. We conclude that although the use of shared memory in *LinAlg* has a positive effect on improving IPC, *LinAlg* can not achieve maximum potential IPC due to two overheads, bank conflict and broadcasting. This indicates room for achieving better performance by some software/hardware improvements in shared memory.

Second, in most cases, both *LinAlg* and *NonLinAlg* exhibit low global load efficiencies. This is expected for *NonLinAlg* since it contains many random accesses. However, it is not expected for *LinAlg*, since it involves matrix operations which usually have more regular memory accesses compared to *NonLinAlg*. We perform several experiments using many synthetic datasets. *We make the key observation that global load efficiency in* LinAlg *is only affected by the distribution of non-zero elements in adjacency matrix*. In fact, datasets with random distribution of non-zero elements experience lower global load efficiency compared to datasets with concentrated non-zero entries. *We also observe in high density datasets, close to a complete graph, effect of randomness on global load efficiency is lower*. Since memory-bound datasets in *LinAlg* are sparse with random distribution of non-zero elements, we observe a scatter pattern in accessing the input vector for sparse matrix-vector multiplications, decreasing the global load efficiency. In other words, due to the low spatial locality, only a few of the loaded words in a cache block are utilized in each Multiply-and-Add operation. The unused loaded words are unlikely to be used in future operations due to the high cache eviction rate in GPUs. Sector cache designs [70, 71, 79, 119] decide whether or not it is beneficial to load a subblock (or a word) in a cache block. We believe employing such designs can be useful in sparse matrix operations.

Third, *LinAlg* has higher global store efficiency compared to *NonLinAlg*. Our experiments using synthetic datasets show that global store efficiency has an inverse relationship with dataset density in *LinAlg*. We explain this relationship based on two observations: (1) according to profiling results, we find out all intermediate computations (element-wise and reduction operations) are performed in shared memory and only the final results are written in the global memory; (2) in high-density datasets, each warp is responsible for small number of rows in the adjacency matrix, while in sparse datasets, each warp covers a larger number of matrix rows. According to these two observations, we conclude that for high-density datasets, each warp writes to few elements in the output vector. In this case, only a small portion of the bytes transferred by the hardware is used by the warp, decreasing global store efficiency. The reverse of this

argument can be made for the case of low-density datasets. As a result, since memory-bound datasets in *LinAlg* are sparse, *LinAlg* experiences high store efficiency.

5.1.2 Compute-bound Analysis. Table 6 reports IPC averaged across compute-bound datasets for each algorithm We make a key observation that in compute-bound datasets of *LinAlg*, the IPC is in the range of 0.9 to 1.4 and is far from the nominal maximum IPC reported for NVIDIA GP100, i.e., 3. Therefore, it is crucial to improve compute throughput of these datasets to improve *LinAlg* performance. We further discuss the potential solution for this observation in §6.1.2.

We only compare *LinAlg-TC* to *NonLinAlg-TC* here since *NonLinAlg-SSSP*, *NonLinAlg-BC*, *NonLinAlg-PR*, *NonLinAlg-BFS*, and *NonLinAlg-GC* are always memory-bound. For compute-bound datasets, *NonLinAlg-TC* and *LinAlg-TC* experience almost the same IPC results, on average. Two main factors limit the IPC for compute-bound kernels: ALU instruction mix and instruction serialization In the case of ALU instruction mix, we observe that both *LinAlg-TC* and *NonLinAlg-TC* only use 32-bit integer instructions, thus IPC is equally affected by ALU instruction mix in both implementations. Regarding the instruction serialization overhead, including atomic instructions, bank conflicts, synchronization, execution dependencies, and branch divergence, we observe that *LinAlg-TC* mostly suffers from synchronization and bank conflict overheads. This is mainly because *LinAlg* uses shared memory for all intermediate matrix operations (e.g., element-wise operations, reduction) in matrix-matrix/vector multiplication. In contrast, *NonLinAlg-TC* suffers from atomic instructions. This is because *NonLinAlg* usually use atomic instructions instead of shared memory for synchronization purposes. Both *LinAlg-TC* and *NonLinAlg-TC* are almost equally affected by execution dependencies and branch divergence. We conclude that, for compute-bound datasets, the IPC overhead of atomic instructions in *NonLinAlg-TC* is almost equal to the overhead of bank conflict and synchronization in *LinAlg-TC*.

5.2 Executed Instruction Analysis

Figure 4 plots the instruction count for the average case and some representative datasets. We make two key observations. *First*, *LinAlg* mostly executes more instructions compared to *NonLinAlg*. We observe a roughly equal number of kernel invocations in *LinAlg* and *NonLinAlg*. However, we observe that each kernel invocation in *LinAlg* has a significantly higher instruction count compared to *NonLinAlg*, leading to a higher instruction count in total. This is because in *NonLinAlg*, especially traversal algorithms such as *NonLinAlg-SSSP* and *NonLinAlg-BC*, we handle only a subset of nodes and edges in most iterations of the algorithm. Therefore, the instruction count in each iteration of *NonLinAlg* is attributed to the active nodes and their incoming edges. On the opposite, *LinAlg* multiply the adjacency matrix by a vector/matrix over a semiring in each iteration. As a result, the instruction count in each iteration is attributed to the total node and edge counts of the graph. Therefore, in each iteration, *LinAlg* executes a higher number of instructions compared to *NonLinAlg*. *Second*, the difference between *LinAlg* and *NonLinAlg* in instruction count is highly dependent on dataset characteristics and algorithm. As an example, for the `sc-ldoor` dataset, *LinAlg-SSSP* executes 16× more instructions, while for the `frb100-40` dataset, *LinAlg-SSSP* executes 4× more instructions. In this case, we observe that the length of the longest shortest path (minimum sum of weights for SSSP) greatly affects the instruction count in *NonLinAlg*. To elaborate more, we rearrange the edges of a given graph to increase the length of the longest shortest path, while keeping the node and edge count unchanged. We observe that the instruction count in each iteration decreases in *NonLinAlg*, while it remains nearly unchanged for *LinAlg*. *NonLinAlg-PR* and *LinAlg-PR* have more closer number of instructions. For some datasets, such as `soc-wConf`, we even see almost equal number of executed instructions. This is because, each iteration updates the PageRank values for all nodes. Since this computation is similar

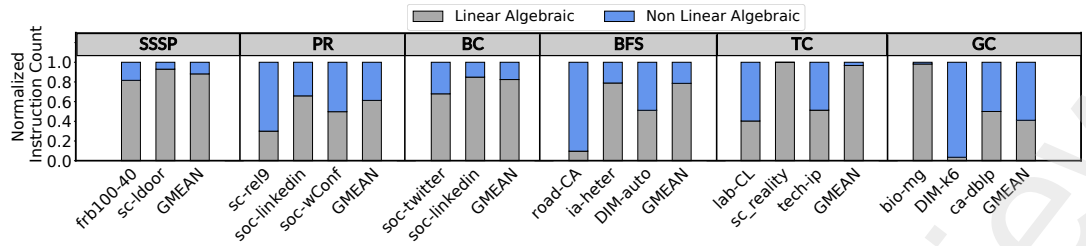


Fig. 4. Normalized instruction count

Table 9. Sppedup, normalized IPC, and normalized instruction count of different versions of *LinAlg-SSSP* and *NonLinAlg-SSSP*

	Speed up					Normalized IPC					Normalized Instruction Count				
Datasets	***V1-Atomic***	***V1-Shared***	***V2***	***V3***	***NonLinAlg-SSSP***	***V1-Atomic***	***V1-Shared***	***V2***	***V3***	***NonLinAlg-SSSP***	***V1-Atomic***	***V1-Shared***	***V2***	***V3***	***NonLinAlg-SSSP***
bio-mg	0.67	1.24	1.87	2.10	1.18	0.79	1.07	0.59	0.72	0.45	0.77	0.77	0.46	0.32	0.10
ca-dblp	1.14	1.35	1.90	4.42	1.29	0.72	0.79	0.54	0.22	0.37	0.45	0.56	0.43	0.06	0.11
DIM-auto	1.30	1.23	3.77	3.24	2.17	0.58	0.70	0.56	0.12	0.44	0.44	0.57	0.21	0.05	0.15
DIM-M81	0.39	1.26	0.99	0.39	0.94	0.71	1.16	0.98	0.71	0.41	0.86	0.90	0.99	0.89	0.23
frb100-40	0.45	1.14	1.0	0.38	0.93	0.66	1.03	0.98	0.62	0.38	0.86	0.89	0.98	0.88	0.22
lab-ER	1.64	1.33	1.66	4.17	1.95	0.88	0.97	0.58	0.37	1.26	0.51	0.66	0.54	0.16	0.20
road-CA	3.54	1.18	39.20	37.14	3.99	2.17	0.97	0.01	0.01	0.31	0.61	0.82	0.0004	0.0004	0.06
sc-ldoor	2.33	1.51	29.14	28.99	2.75	1.00	0.81	0.59	0.13	0.34	0.43	0.54	0.03	0.006	0.08
soc-twitter	0.91	0.94	1.88	1.13	1.06	0.49	0.68	0.45	0.27	0.36	0.51	0.70	0.34	0.34	0.23
web-ND	1.14	1.19	1.84	1.25	1.34	0.56	0.76	0.23	0.16	0.34	0.45	0.61	0.20	0.17	0.11

to sparse matrix-vector multiplication [57], we expect the instruction count to be approximately equal for *LinAlg-PR* and *NonLinAlg-PR*.

6 KEY INSIGHTS AND SOLUTIONS

In this section, we discuss some research directions to improve the performance of graph processing on GPUs.

6.1 Performance Improvement

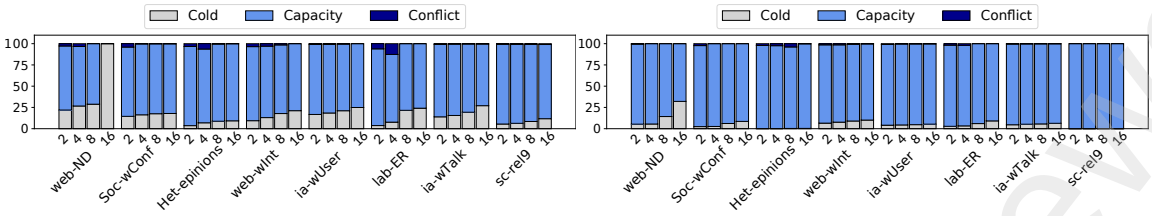
In this section, we first devise some software-level optimization techniques to reduce instruction count (§ 6.1.1). We then make the case for some GPU architectural changes to improve both memory and compute throughput (§ 6.1.2).

6.1.1 Software-level Optimization. As we observed in § 5.2, *LinAlg* mostly executes more instructions compared to *NonLinAlg*. In this section, we aim to reduce the number of executed instructions via some software-level optimizations. As a case study, we focus on SSSP algorithm that is dual with a matrix-vector multiplication with a semiring of extended real numbers with operations bit-wise addition and reduction minimum, replacing the operations of bit-wise multiplication and reduction sum, respectively. We devise two techniques to reduce the number of executed instructions in *LinAlg-SSSP*through incorporating the algorithm semantics into the matrix-vector multiplication. First, in *LinAlg-SSSP*, multiplying each row of the matrix by the input vector is equivalent to relaxing the incoming edges of the vertex corresponding to that row [69]. If a neighbor of a vertex has not been updated in the previous iteration, we can ignore relaxing the edge connected to that neighbor in the current iteration. Therefore, several element-wise operations are removed from the matrix-vector multiplication and the depth of the reduction tree is reduced. We implement this idea using atomic instructions (*V1-Atomic*) and shared memory (*V1-Shared*). Second, we implement a worklist to collect a set of active nodes and process only the active ones in each iteration, similar to Tigr implementations [99]. To this end, we remove the computation of rows corresponding to the non-active nodes from the matrix-vector multiplication in each iteration. We call our worklist-aware implementation *V2*. We also implement a version of SSSP that benefits

from two aforementioned optimizations, called *V3*. Table 9 compares the speed up, normalized IPC, and normalized instruction count of different versions of *LinAlg-SSSP*, and *NonLinAlg-SSSP* with respect to the baseline *LinAlg-SSSP* for 10 representative datasests.

We make nine observations. *First*, in most datasets, optimization techniques greatly reduce the instruction count at the price of some IPC overhead due to reducing the regularity of matrix operations. The level of IPC degradation in each implementation varies depending on dataset characteristics. Overall, we observe less execution time for most of the datasets with respect to the baseline *LinAlg-SSSP*. *Second*, *V1-Atomic* outperforms *V1-Shared* for highly sparse datasets with regular or normal degree distribution (see Table 9) `road-CA` dataset with a density of 1.41 and regular degree distribution is a good example for this observation in which, *V1-Atomic* outperforms *V1-Shared* by 3×. In the adjacency matrix of such datasets, the number of non-zero elements per row is significantly low and all rows have almost the same amount of non-zero elements (due to regular/normal degree distribution). Hence, the depth of reduction tree per row is super small. We observe that *V1-Atomic* works better that *V1-Shared* implementation for low-depth reduction tree. *Third*, denser datasets with deep reduction tree experience performance improvement from *V1-Shared*, while *V1-Atomic* usually degrades the performance of these datasets. As an example, in `DIM-M81` dataset with a density of 1658, *V1-Shared* improves performance by 20%, while *V1-Atomic* degrades performance by 155% compared to the baseline *LinAlg-SSSP*. *Forth*, both *V1-Atomic* and *V1-Shared* either degrade performance or provide low performance improvement for sparse datasets with power-law degree distribution. For example, in `soc-twitter` dataset with a density of 1.7 and power-law degree distribution, *V1-Atomic* and *V1-Shared* degrade performance by 9% and 6%, respectively. On one hand, in the adjacency matrix of these datasets, there are several rows with a small number of non-zero elements that cause performance overhead using *V1-Shared*. On the other hand, a few rows of the matrix have a large number of non-zero elements that result in performance overhead by using *V1-Atomic*. Therefore, there is no proper choice between *V1-Atomic* and *V1-Shared* for these datasets. *Fifth*, in some datasets, *V1-Atomic* or *V1-Shared* provide even higher IPC results compared to the baseline *LinAlg-SSSP*. Examples are `road-CA` and `DIM-M81` datasets. As we mentioned before, in the technique of decreasing depth of reduction tree, several element-wise operations are removed from the matrix-vector multiplication. This technique reduces the number of accesses to the global memory, which in some cases can lead to more coalesced reads from global memory. In such cases, the global memory efficiency increases that causes better IPC results. *Sixth*, we observe that the length of the longest shortest path (minimum sum of weights) greatly affects the instruction count in *V2*. In other words, in datasets with longer shortest path, a small number of active nodes are processed in each iteration, which results in a significant decrease of instruction count in total. As an example, in `sc-ldoor` dataset with a long shortest path, we observe 29.1× speed up using *V2*. *Seventh*, some datasets benefit from *V3* implementation. As an example, in `ca-dblp` dataset, we observe 4.4× speed up using *V3*. *Eighth*, datasets that are close to the complete graph do not achieve much improvement from any of these optimization techniques. `frb100-40` dataset is a good example that see only a small performance improvement using *V1-Shared*. In this dataset, it is necessary to multiply almost the entire adjacency matrix by the input vector, and none of the optimizations can effectively reduce the instruction count. *Ninth*, in each dataset, there is at least one optimized version of *LinAlg-SSSP* that outperforms compared to *NonLinAlg-SSSP*.

6.1.2 Hardware-level Optimization. In § 5, we observed *LinAlg* is compute-bound for most datasets and memory-bound for others. Therefore, it is crucial to improve both memory and compute throughput to improve *LinAlg* performance. In the remaining sections, first, we discuss how to improve memory throughput to shift memory-bound datasets into the compute-bound region. Second, we discuss how to improve compute throughput for better performance in the



((a)) *LinAlg-BC* ((b)) *NonLinAlg-BC*

Fig. 5. Distribution of L2 cache misses

compute-bound cases.

Memory Throughput Enhancement. Different components in the memory hierarchy, such as L1 and L2 caches, network-on-chip, and device memory bandwidth, can affect the throughput of the memory accesses. As a case study, we focus on L2 cache hit rate. Figure 5 shows the distribution of *cold*, *capacity*, and *conflict* misses for various L2 cache sizes (2MB to 16MB) using an example BC algorithm (both *LinAlg-BC* and *NonLinAlg-BC*) under eight representative datasets. We make the key observation that due to better cache locality in *LinAlg*, the capacity miss is reduced at higher rates for *LinAlg-BC* compared to *NonLinAlg-BC* by increasing the L2 cache size. Therefore, *LinAlg* can highly benefit from larger L2 caches. To elaborate more, Figure 6 plots the IPC for BC, PR, and SSSP algorithms for both *LinAlg* and *NonLinAlg*, using different L2 cache sizes (normalized to *LinAlg* with the baseline L2 cache size, i.e., 4MB). The labels on each point show the percentage of memory-bound datasets in each comparison point. We make two key observations. First, *LinAlg* experiences higher performance improvement by enlarging the L2 cache size. We observe that increasing the L2 cache size from 4MB to 8MB (16MB), respectively increases the average IPC by 31%/12%/21% (41%/28%/37%) and 10%/7%/7% (26%/17%/16%) for *LinAlg-BC*/*LinAlg-PR*/*LinAlg-SSSP* and *NonLinAlg-BC*/*NonLinAlg-PR*/*NonLinAlg-SSSP*, underscoring that *LinAlg* benefits more from larger L2 caches. Second, doubling the baseline L2 cache shifts almost all memory-bound datasets in *LinAlg* to the compute-bound region. We conclude that enlarging L2 cache is worthy for *LinAlg*, and even using 2× larger L2 cache greatly improves the memory throughput.

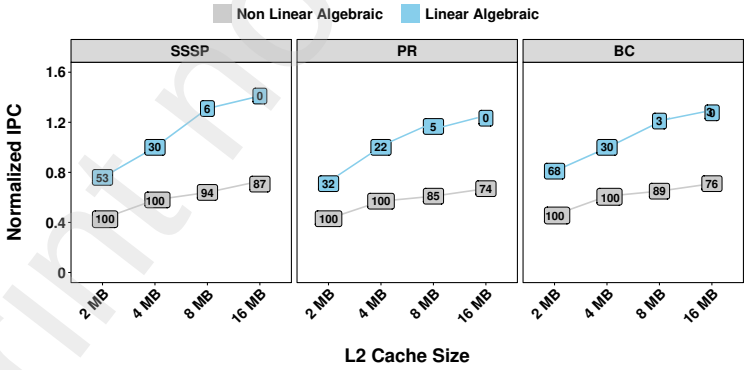


Fig. 6. IPC results of both implementations using various L2 cache sizes normalized to the IPC result of a 4MB cache.

Compute Throughput Enhancement. One potential solution to improve the instruction throughput for *LinAlg* implementations is to exploit Tensor Core Units (TCUs). TCUs are special units in modern GPUs to accelerate matrix-matrix

multiplication [101]. TCUs can potentially be used to accelerate matrix-matrix/vector multiplication in *LinAlg* implementations. However, There are two major challenges. First, TCUs only have support for dense matrix-matrix multiplication, leading to inefficient execution of sparse matrices. Second, TCUs do not execute efficiently *sparse matrix-vector multiplication* which is an essential operation for linear algebraic implementations. Prior work [143] has addressed the first issue by proposing tSparse mechanism in which they tile the input data and operate only on tiles which contain at least one element, however, none prior works provide an efficient execution of *sparse matrix vector multiplication* via TCUs. We believe TCUs are a potential platform for solving many linear algebraic operations and this research area merits further exploration.

6.2 Giant Graph Analytic

Real-world graphs, such as social networks and web pages, are becoming increasingly larger, nearing orders of tens to hundreds of gigabytes. Most previous frameworks are limited to graphs that fit in the GPU memory. For instance, Gunrock and CuSha, do not support giant graphs. To address this issue, programmers must tile the datasets, divide the execution to multiple steps, and handle copying the data from CPU to GPU iteratively. This adds extra complexity to programming.

For ease of programming, today's GPUs are equipped with Unified Virtual Memory (UVM) through which GPUs can oversubscribe CPU memory. UVM allows us to run applications with large datasets that do not fit in the GPU memory. However, using UVM naively for graph processing can reduce performance. This is because graph processing platforms usually have irregular memory access patterns which cause the number of page faults to rise. In addition, designing data-prefetching policies to lower the penalty for page faults is not an easy task. On the other hand, *LinAlg* exhibits a high locality with which we can combine data-prefetching policies to minimize page faults penalty. As a result, leveraging UVM for *LinAlg* while using an appropriate data-prefetching policy is a potential solution that merits further exploration.

7 RELATED WORK

To the best of our knowledge, this is the first work to conduct a holistic study on *LinAlg* graph processing on GPUs. Our work (1) highlights a set of bottlenecks and opportunities in *LinAlg* and (2) makes several key insights and solutions to improve the performance of *LinAlg*. We briefly discuss prior work in these categories: (1) Graph processing frameworks, (2) Graph processing accelerators, (3) Sparse matrix computation in GPUs, and (4) Graph processing analysis in GPUs.

Graph processing frameworks. We place these frameworks in two categories: (1) CPU-based frameworks, which support sequential or coarse-grained parallel programming models. To overcome scalability issues, distributed CPU-based frameworks have also been proposed, which address challenges such as synchronization overheads and load imbalance [47, 48, 59, 77, 86, 98, 114, 127]; (2) GPU-based frameworks which implement graph primitives, with specialized parallel graph algorithms [9, 11, 18, 20, 31, 49, 51, 56, 60, 80, 82, 89–91, 106, 107, 111, 115, 123] not generalizing well of different kinds of graph applications and showcasing substantial increase in performance, while generalized frameworks [1, 15, 40, 42, 43, 50, 55, 58, 61, 65, 72, 73, 76, 81, 84, 99, 105, 113, 118, 120, 122, 129, 131, 145, 146, 148, 148, 149] provide programming flexibility.

Graph processing accelerators. Several accelerators have been proposed for graph processing. FPGA-based accelerators utilize their re-configurable features to achieve flexible architectures suitable to some graph algorithms [28, 29, 85, 135]. ASIC-based accelerators [53, 103, 124] offer efficient hardware organizations (e.g., dedicated and accurate placement of resources, higher frequencies compared to FPGAs). PIM-based accelerators overcome graph

processing bottlenecks in terms of memory bandwidth [5, 30, 93, 144]. Several prior works employ different types of memory in the accelerators, such as integrating a specialized graph accelerator in the HMC logic layer [5] and leveraging ReRAMs in crossbars [54].

Sparse matrix computation in GPUs. Many studies focus on optimizing SpMV computation in GPUs [7, 12, 13, 17, 38, 78, 133, 137, 142, 147]. Sparse matrices are often represented by different compact formats (e.g., CSR, CSC, BSR, COO, LIL, and BiELL). The format choice heavily impacts performance, therefore it is crucial to pick a suitable format for each application and/or platform. A machine learning approach has been employed in [16] to select a format best suitable to GPUs for an input matrix. The ClSpMV framework [125] aims to recommend the best representation on different platforms for many types of sparse matrices at runtime. In addition, many commercial sparse libraries (e.g., MKL [63], CUSP [14], ViennaCL [112], and NVIDIA's cuSPARSE [26]) are widely used in scientific computation. A few works have used GPU hardware components, such as tensor cores, to perform sparse matrix multiplication [151].

Graph processing analysis in GPUs. With the prevalence of graph datasets and algorithms, extensive studies have been conducted to analyze graph processing on GPUs. A few studies [121, 136] focus on examining the bottlenecks observed in *NonLinAlg*and suggest relevant research opportunities in future. Other studies [134] analyze specific high-level GPU graph analytics frameworks (e.g., Gunrock [131], MapGraph [40], and VertexAPI2 [37]). These studies evaluate the impact of critical factors (e.g., efficient building-block operators, synchronization and data movement, workload distribution and load balancing, and memory access patterns) on real-world and synthetic graphs in a multitude of application domains.

8 CONCLUSION

Graph processing in GPUs has various challenges. A few of which are load imbalance and irregular memory access patterns. While prior graph processing frameworks address individual challenges, none have been able to address all simultaneously. Linear algebraic implementations, which involve primitive matrix operations, seem a powerful solution to cope with graph processing challenges in GPUs. Our work comprehensively studies the advantages and disadvantages of linear algebraic approaches in graph processing from a GPU architectural point of view. To this end, we characterize six well-known graph algorithms using 160 real-word datasets on a real machine and a GPU simulator (i.e., Accelsim). We make a key observation that although linear algebraic implementations usually experience higher instruction throughput (IPC), they suffer from a significant increase in the number of instructions, which, in some cases, offsets the benefits of the higher IPC. Based on our findings, we discuss potential hardware/software research directions for performance improvement in graph analysis. As a case study, we devise two software-based optimization techniques that reduce the number of executed instructions through incorporating the algorithm semantics into the matrix operations. Our experimental results show up to 39.2× (8.9× on average) speed up using the proposed optimization techniques.

REFERENCES

[1] Christopher R Aberger, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2015. EmptyHeaded: boolean algebra based graph processing. *ArXiv e-prints* (2015).

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*. Springer, 230–241.

[3] Ittai Abraham, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 782–793.

[4] Ilya Afanasyev and Vladimir Voevodin. 2017. The comparison of large-scale graph processing algorithms implementation methods for Intel KNL and NVIDIA GPU. In *Russian Supercomputing Days*. Springer, 80–94.

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

[7] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 781–792.

[8] Ariful Azad, Aydin Buluç, and Alex Pothen. 2015. A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1075–1084.

[9] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceška. 2011. Computing strongly connected components in parallel on CUDA. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 544–555.

[10] Marc Barthelemy. 2004. Betweenness centrality in large complex networks. *The European physical journal B* 38, 2 (2004), 163–168.

[11] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[12] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

[13] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[14] Nathan Bell and Michael Garland. 2012. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *Version 0.3. 0* 35 (2012).

[15] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.

[16] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 496–505.

[17] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. Slimsell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 32–41.

[18] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[19] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 643–652.

[20] Federico Busato and Nicola Bombieri. 2014. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2014), 1826–1838.

[21] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[22] Aleksandar Colic, Hari Kalva, and Borko Furht. 2010. Exploring nvidia-cuda for video coding. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. 13–22.

[23] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. 2001. *Introduction to algorithms*. Vol. 5. MIT press Cambridge.

[24] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. 2001. *Introduction to algorithms*. Vol. 5. MIT press Cambridge.

[25] NVIDIA Corporation. 2019. nvGRAPH Library. https://developer.nvidia.com/nvgraph.

[26] NVIDIA Corporation. 2020. NVIDIA's cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html.

[27] NVIDIA Corporation. 2020. Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[28] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 105–110.

[29] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 217–226.

[30] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[31] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 349–359.

[32] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[33] Timothy A Davis. 2020. User Guide for SuiteSparse: GraphBLAS.

[34] Nan Ding and Samuel Williams. 2019. An instruction roofline model for gpus. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 7–18.

[35] Stephen Dolan. 2013. Fun with semirings: a functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 101–110.

[36] David Dolžan and Polona Oblak. 2011. Commuting graphs of matrices over semirings. *Linear algebra and its applications* 435, 7 (2011), 1657–1665.

[37] Erich Elsen and Vishal Vaidyanathan. 2013. A vertex-centric CUDA/C++ API for large graph analytics on GPUs using the gather-apply-scatter abstraction.

[38] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)* 43, 4 (2017), 1–49.

[39] Linton C Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[40] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*. 1–6.

[41] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 407–420.

[42] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 345–354.

[43] Abdullah Gharaibeh, Tahsin Reza, Elizeu Santos-Neto, Lauro Beltrao Costa, Scott Sallinen, and Matei Ripeanu. 2013. Efficient large-scale graph processing on hybrid CPU and GPU systems. *arXiv preprint arXiv:1312.3018* (2013).

[44] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. 2013. The energy case for graph processing on hybrid cpu and gpu systems. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[45] Michel Gondran and Michel Minoux. 2007. Dioïds and semirings: Links to fuzzy sets and other applications. *Fuzzy Sets and Systems* 158, 12 (2007), 1273–1294.

[46] Michel Gondran and Michel Minoux. 2008. *Graphs, dioids and semirings: new models and algorithms*. Vol. 41. Springer Science & Business Media.

[47] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

[48] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[49] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[50] Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)* 2 (2005), 1–18.

[51] John Greiner. 1994. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. 16–25.

[52] Udo Hahn and Ulrich Reimer. 1984. Computing text constituency: An algorithmic approach to the generation of text graphs. In *Proceedings of the 7th annual international ACM SIGIR conference on Research and development in information retrieval*. 343–368.

[53] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[54] Lei Han, Zhaoyan Shen, Zili Shao, H Howie Huang, and Tao Li. 2017. A novel ReRAM-based processing-in-memory architecture for graph computing. In *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[55] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 233–245.

[56] Pawan Harish and Petter J Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*. Springer, 197–208.

[57] Taher Haveliwala and Sepandar Kamvar. 2003. *The second eigenvalue of the Google matrix*. Technical Report. Stanford.

[58] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. Multigraph: Efficient graph processing on gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 27–40.

[59] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 349–362.

[60] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. *Acm Sigplan Notices* 46, 8 (2011), 267–276.

[61] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 78–88.

[62] Qiming Hou, Kun Zhou, and Baining Guo. 2008. BSGP: bulk-synchronous GPU programming. *ACM Transactions on Graphics (TOG)* 27, 3 (2008), 1–12.

[63] Intel. 2020. MKL sparse-BLAS library. http://software.intel.com/en-us/intel-mkl.

[64] O Ivanciuc and AT Balaban. 1999. The graph description of chemical structures. *Topological indices and related descriptors in QSAR and QSPR* (1999), 59–167.

[65] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[66] Vana Kalogeraki, Dimitrios Gunopulos, and Demetrios Zeinalipour-Yazti. 2002. A local search mechanism for peer-to-peer networks. In *Proceedings of the eleventh international conference on Information and knowledge management*. 300–307.

[67] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[68] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science* 51 (2015), 2453–2462.

[69] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[70] Mahmoud Khairy, Jain Akshay, Tor Aamodt, and Timothy G Rogers. 2018. Exploring modern GPU memory system design challenges through accurate modeling. *arXiv preprint arXiv:1810.07269* (2018).

[71] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[72] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 39–50.

[73] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[74] Samir Khuller and Balaji Raghavachari. 1996. Graph and network algorithms. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 43–45.

[75] Samir Khuller and Balaji Raghavachari. 2010. Basic graph algorithms. In *Algorithms and theory of computation handbook: general concepts and techniques*. 7–7.

[76] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*. 447–461.

[77] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 31–46.

[78] Kenli Li, Wangdong Yang, and Keqin Li. 2014. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (2014), 196–205.

[79] John S. Liptay. 1968. Structural aspects of the System/360 Model 85, II: The cache. *IBM Systems Journal* 7, 1 (1968), 15–21.

[80] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[81] Hang Liu and H Howie Huang. 2019. Simd-x: Programming and processing of graph algorithms on gpus. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*. 411–428.

[82] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. 403–416.

[83] Peter J Lu, Hidekazu Oki, Catherine A Frey, Gregory E Chamitoff, Leroy Chiao, Edward M Fincke, C Michael Foale, Sandra H Magnus, William S McArthur, Daniel M Tani, et al. 2010. Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station. *Journal of Real-Time Image Processing* 5, 3 (2010), 179–193.

[84] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*. 195–207.

[85] Xiaoyu Ma, Dan Zhang, and Derek Chiou. 2017. FPGA-accelerated transactional execution of graph workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 227–236.

[86] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[87] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[88] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. 2019. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 276–284.

[89] Adam McLaughlin and David A Bader. 2014. Scalable and high performance betweenness centrality on the GPU. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 572–583.

[90] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[91] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[92] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[93] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 457–468.

[94] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, Mario Paulo Drumond, Hamid Sarbazi-Azad, and Babak Falsafi. 2020. Efficient nearest-neighbor data sharing in GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2020), 1–26.

[95] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad. 2018. Neda: Supporting direct inter-core neighbor data exchange in GPUs. *IEEE Computer Architecture Letters* 17, 2 (2018), 225–229.

[96] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.

[97] Mark EJ Newman. 2005. A measure of betweenness centrality based on random walks. *Social networks* 27, 1 (2005), 39–54.

[98] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 456–471.

[99] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[100] NVIDIA. 2016. NVIDIA® Tesla® P100–The Most Advanced Data Center Accelerator Ever Built.

[101] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture.

[102] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. 2007. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, Vol. 26. Wiley Online Library, 80–113.

[103] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 166–177.

[104] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[105] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–19.

[106] P Pande and David A Bader. 2011. Computing betweenness centrality for small world networks on a GPU. In *15th Annual High performance embedded computing workshop (HPEC)*.

[107] Adam Polak. 2016. Counting triangles in large graphs on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 740–746.

[108] Greta L Polites and Richard T Watson. 2008. The centrality and prestige of CACM. *Commun. ACM* 51, 1 (2008), 95–100.

[109] RAPIDS. 2021. cuGraph Library. https://github.com/rapidsai/nvgraph.

[110] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[111] Arnon Rungsawang and Bundit Manaskasemsak. 2012. Fast pagerank computation on a gpu cluster. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 450–456.

[112] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jungel, and Siegfried Selberherr. 2016. ViennaCL—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing* 38, 5 (2016), S412–S439.

[113] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[114] Semih Salihoglu and Jennifer Widom. 2014. HelP: High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*. 1–6.

[115] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 76–85.

[116] Data science framework. 2021. RAPIDS. https://rapids.ai/start.html.

[117] Dipanjan Sengupta, Kapil Agarwal, Shuaiwen Leon Song, and Karsten Schwan. 2015. Graphreduce: Large-scale graph analytics on accelerator-based hpc systems. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 604–609.

[118] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. 2015. Gstream: A graph streaming processing method for large-scale graphs on gpus. *ACM SIGPLAN Notices* 50, 8 (2015), 253–254.

[119] André Seznec. 1994. Decoupled sectored caches: conciliating low tag implementation cost. In *Proceedings of the 21st annual international symposium on Computer architecture*. 384–393.

[120] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. 2015. Optimization of asynchronous graph processing on GPU with hybrid coloring model. *ACM SIGPLAN Notices* 50, 8 (2015), 271–272.

[121] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 1–35.

[122] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[123] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[124] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[125] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. 353–364.

[126] Google Tech Talk. 2009. Fast Route Planning.

[127] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

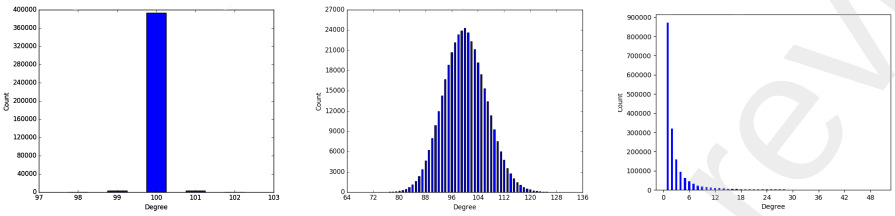
[128] Jan van Leeuwen. 1990. Graph algorithms. In *Algorithms and complexity*. Elsevier, 525–631.

[129] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 38–52.
[130] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.
[131] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 1–49.
[132] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.
[133] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. 2010. Efficient pagerank and spmv computation on amd gpus. In *2010 39th International Conference on Parallel Processing*. IEEE, 81–89.
[134] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D Owens. 2015. Performance characterization of high-level programming models for GPU graph analytics. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 66–75.
[135] Chongchong Xu, Chao Wang, Yiwei Zhang, Lei Gong, Xi Li, and Xuehai Zhou. 2018. Domino: An Asynchronous and Energy-efficient Accelerator for Graph Processing. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 289–289.
[136] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks?. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 140–149.
[137] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: yet another SpMV framework on GPUs. *Acm Sigplan Notices* 49, 8 (2014), 107–118.
[138] Beverly Yang and Hector Garcia-Molina. 2002. Efficient search in peer-to-peer networks. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*.
[139] Beverly Yang and Hector Garcia-Molina. 2002. Improving search in peer-to-peer networks. In *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE, 5–14.
[140] Carl Yang, Aydin Buluc, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.
[141] Charlene Yang, Thorsten Kurth, and Samuel Williams. 2020. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience* 32, 20 (2020), e5547.
[142] Carl Yang, Yangzihao Wang, and John D Owens. 2015. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 841–847.
[143] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers & Electrical Engineering* 88 (2020), 106848.
[144] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.
[145] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. 2016. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 912–920.
[146] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 601–614.
[147] Cong Zheng, Shuo Gu, Tong-Xiang Gu, Bing Yang, and Xing-Ping Liu. 2014. BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs. *J. Parallel and Distrib. Comput.* 74, 7 (2014), 2639–2647.
[148] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.
[149] Jianlong Zhong and Bingsheng He. 2013. Towards GPU-accelerated large-scale graph processing in the cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 1. IEEE, 9–16.
[150] Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, Zhiwen Chen, Chang Cheng, and Xuanhua Shi. 2016. Optimizing graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2016), 1149–1162.
[151] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 359–371.

A SYNTHETIC DATASETS

To perform a deeper analysis of a few observations, we generate synthetic datasets with modifiable properties. Our synthetic datasets are supersets of real-world datasets, i.e., they cover our own set of properties, as well as the properties viewed in real-world datasets.

By observing the degree distribution of real datasets, we observe that the datasets can be roughly placed in three groups. The first group consists of graphs where most of the nodes have degrees close to each other. We call these graphs semi-regular graphs. The second group consists of graphs following a normal degree distribution, i.e. graphs whose degree distribution resembles the shape of the normal bell curve. The third group are graphs whose degree distribution follows a power-law. A power-law states that the fraction of nodes with degree k goes to k^{-y} for large k, where the y parameter is a real number typically in the range $2 < y < 3$. Figures 7 (a), (b), and (c) respectively show regular, normal, and power-law graphs, as well as charts depicting their degree distribution.



(a) Regular (b) Normal (c) Power-law

Fig. 7. Graphs with three different degree distribution

To generate graphs following a normal or semi-regular distribution we choose the Watts–Strogatz model [132]. Watts–Strogatz interpolates between a regular and random structure by following a two-step procedure. In the first step, it starts with order, forming a K-regular ring lattice with n vertices connecting to their first $\frac{K}{2}$ neighbors on each side. In the second step, it randomizes the graph by rewiring each edge with probability p, excluding self-connections and duplicate edges. Changing p in the range of $0 < p < 1$, enables us to generate graphs from complete order to complete disorder. For weighted graphs, we modify the algorithm by assigning a random number to each edge in the graph. However, the Watts-Strogatz model cannot generate graphs following a power-law degree distribution. This is because the Watts-Strogatz model generates graphs with a degree distribution peaking at K and as we get further away from K, on both sides, it decays exponentially. Graphs following a power-law degree distribution do not fall in this spectrum, as there is a substantial probability of viewing nodes with high degrees, which also results in many nodes having a small degree. To generate synthetic graph with power-law degree distribution, we use R-MAT, short for Recursive Matrix [21]. R-MAT operates by dividing the adjacency matrix of a graph into equally-sized quadrants and placing an edge in each quadrant with unequal probability. It begins with an empty adjacency matrix; in each step, one of the quadrants is chosen with non-negative probability a, b, c, or d, such that $a + b + c + d = 1$. The algorithm is repeated for the chosen quadrant until we reach a single cell, in which we finally place an edge. As an example, Figure 8 demonstrates how we place a single edge.

We repeat the procedure until we have placed M edges in the adjacency matrix.

By tuning the models parameters, we can create datasets with different properties. Therefore, we can use Watts-Strogatz and R-MAT models for creating our synthetic datasets we use in experiments.

B EXPERIMENTS

Our goal is to explain the observations mentioned in the 5.1. To do so, we design a set of experiments based on synthetic datasets to detect the effecting factors on a specific metric. We use two well-known random graph generation models,

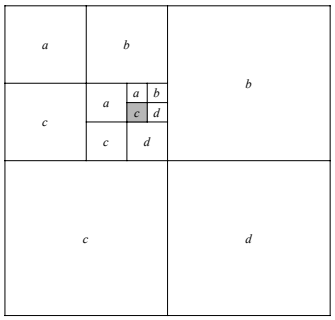


Fig. 8. Placing a single edge in adjacency matrix in R-MAT model

R-MAT [21] and Watts–Strogatz [132], to create these synthetic datasets. We tune the model parameters (refer to § A for more details) to create datasets within a range of properties.

Our primary focus in this study is an in-depth analysis of *LinAlg*. *LinAlg* only involves matrix operations; therefore, we generate different synthetic datasets regarding the matrix properties. We choose two main matrix properties to modify: (1) density, i.e., edge to node count ratio, and (2) non-zero distribution, i.e., the distribution of non-zero elements in the adjacency matrix.

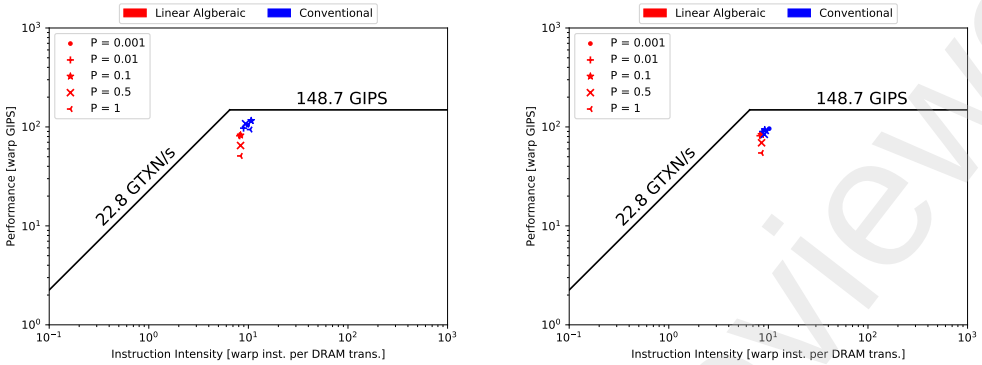
B.1 Compute/Memory-bound Analysis

We examine the effect of density and non-zero distribution on whether a dataset is compute-bound or memory-bound. To modify the density, we can modify the edge count, the node count, or both. We decide to modify the node count and keep the edge count fixed. The reason is that an increase in edge count increases computation, which, in turn, increases memory accesses and transactions; therefore, we cannot attribute the change from compute-bound to memory-bound and vice versa to a shift in density alone. In this experiment, we keep the edge count 8,000,000 and double the density, step by step, from 2 to 64. To modify the non-zero distribution, we modify the probability parameters P and a in Watts-Strogatz and R-MAT models, respectively. For the Watts-Strogatz model, we increase probability P from 0.001 to 1, and for the R-MAT model, we increase probability a from 0.28 to 0.48.

Figures 9, 10, 11, and 12 illustrate the roofline charts for the abovementioned graphs in *LinAlg* and *NonLinAlg* for PR and SSSP (BC behaves similarly to SSSP). We make two key observations. First, density plays an important role in determining whether a dataset is compute-bound or memory-bound. For example, in both implementations of SSSP, all Watts-Strogatz graphs with densities higher than 16 are compute-bound, regardless of the value of P. Graph density threshold determines the border between memory-bound to compute-bound. This threshold is different depending on graph algorithms and implementation types. We observe that *LinAlg* has significantly lower density threshold values, making the key point that *LinAlg* is more likely to be compute-bound under different datasets. Second, density is not the only contributing factor that causes an algorithm being compute-bound or memory-bound. There are some cases in low-density datasets that graph algorithms experience compute-bound execution, expecially for *LinAlg*. For example, in *LinAlg-SSSP*, Watts-Strogatz graphs with a density of 2 and P equal to 0.001, 0.01, and 0.1 are all compute-bound. This is because the non-zero elements are distributed in a concentrated manner, increasing spatial locality. Spatial locality causes the application to use the memory hierarchy more efficiently and create fewer transactions for DRAM. From

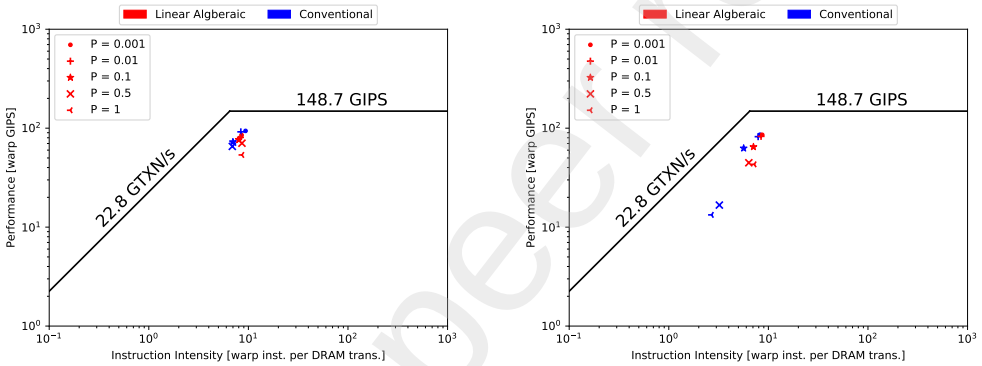
these two observations we conclude that in compared to *NonLinAlg*, *LinAlg* is more likely to be compute-bound under different datasets.

We use heatmap charts to illustrate non-zero distribution. To do so, we divide the adjacency matrix of a dataset into 128x128 tiles. Each pixel in the heatmap represents the number of non-zero elements of a single tile. A white pixel represents a tile with no non-zero elements, and a black pixel represents a tile with all non-zero elements. Figure 13 depicts the heatmap charts of Watts-Strogatz graphs for different densities with P set to 1 and 0.001. The figure shows that for p = 1, the non-zero elements are distributed evenly and completely at random for all densities. In contrast, for p = 0.001, the non-zero elements are mostly concentrated around the matrix diagonal. Figure 14 depicts the heatmap charts of R-MATS graphs for different densities with probability a set to 0.28 and 0.48. When the parameters a, b, c, and d are relatively close in the R-MAT model, the non-zero distribution is close to uniform. Consequently, or a = 0.28, the non-zero distribution is uniform, while for a = 0.48, the elements are mostly concentrated in the upper left region.



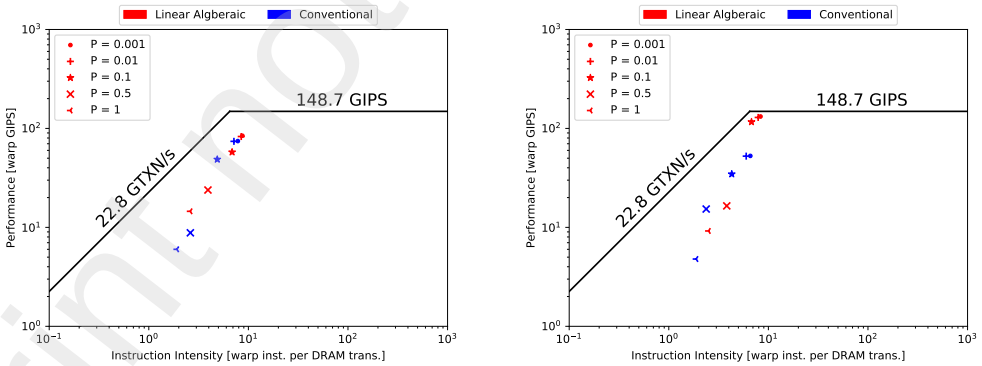
(a) Density = 64

(b) Density = 32



(c) Density = 16

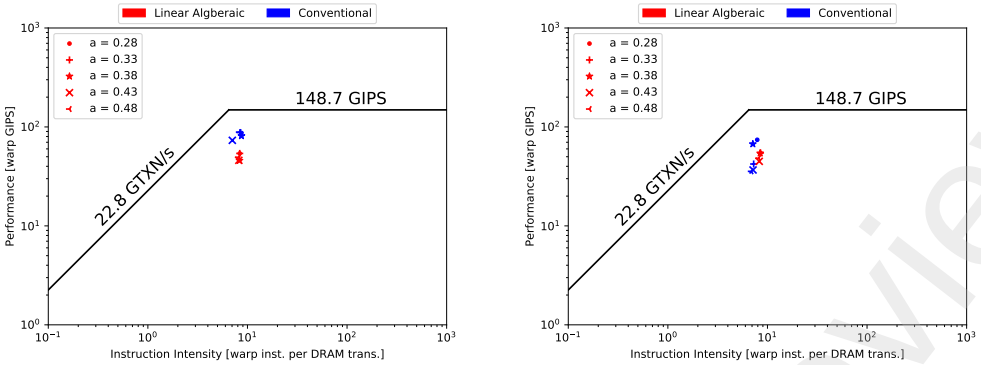
(d) Density = 8



(e) Density = 4

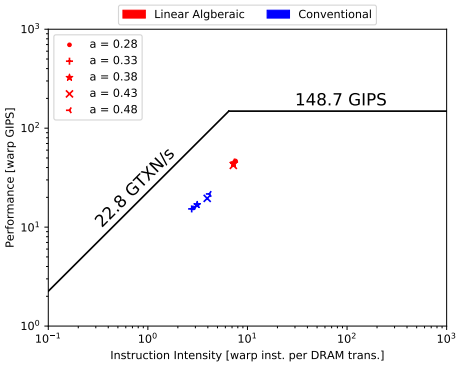
(f) Density = 2

Fig. 9. Roofline charts of SSSP algorithm for Watts-Strogatz datasets

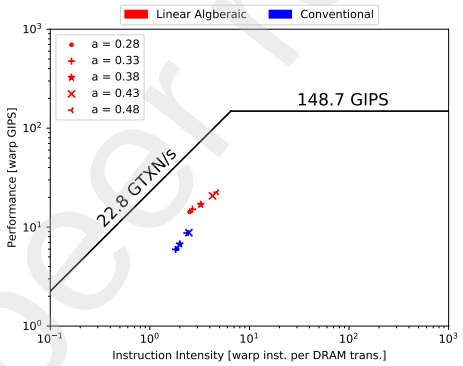


(a) Density = 64

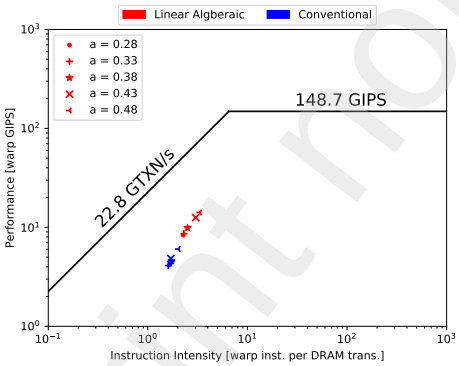
(b) Density = 32



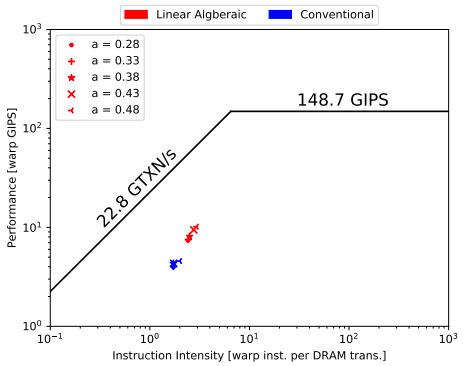
(c) Density = 16



(d) Density = 8

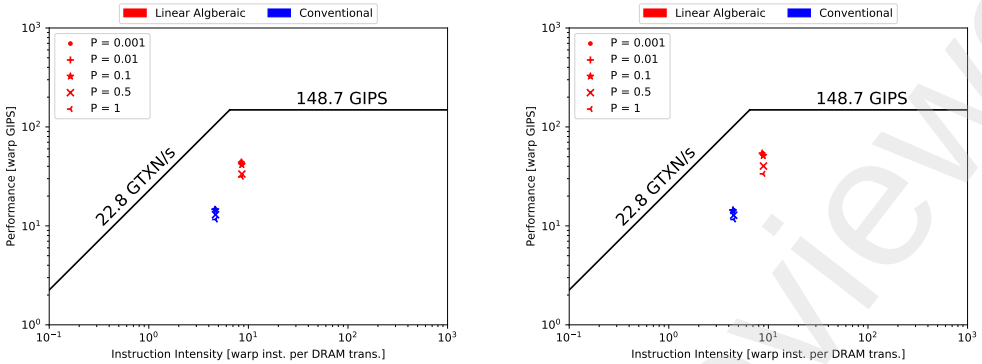


(e) Density = 4



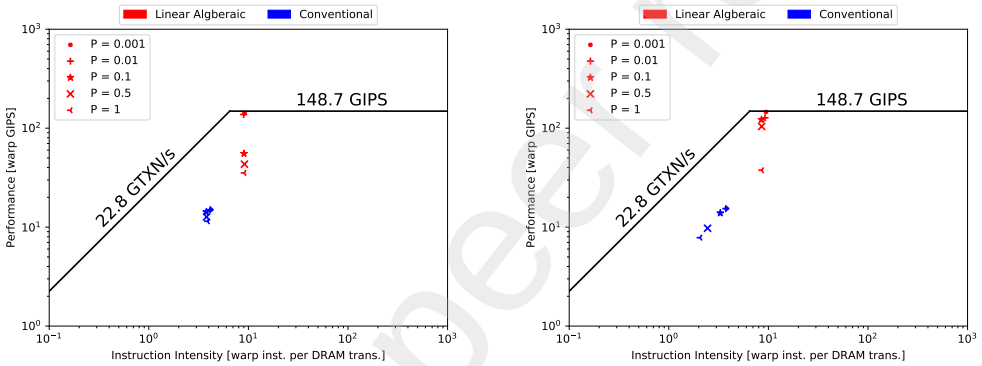
(f) Density = 2

Fig. 10. Roofline charts of SSSP algorithm for R-MAT datasets



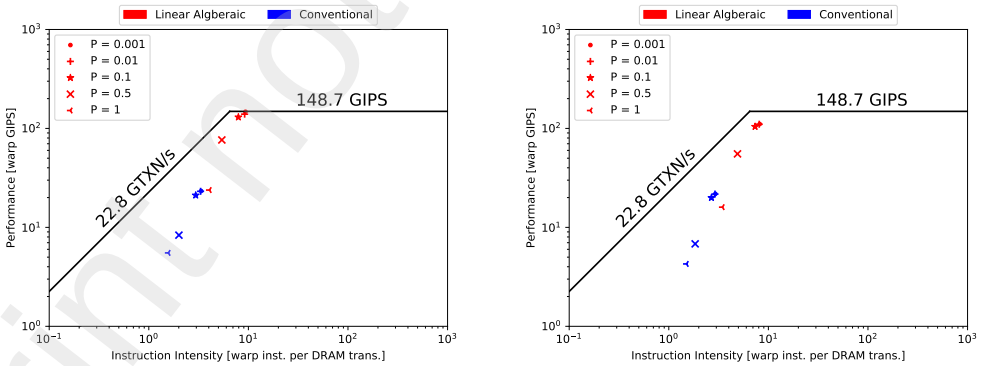
(a) Density = 64

(b) Density = 32



(c) Density = 16

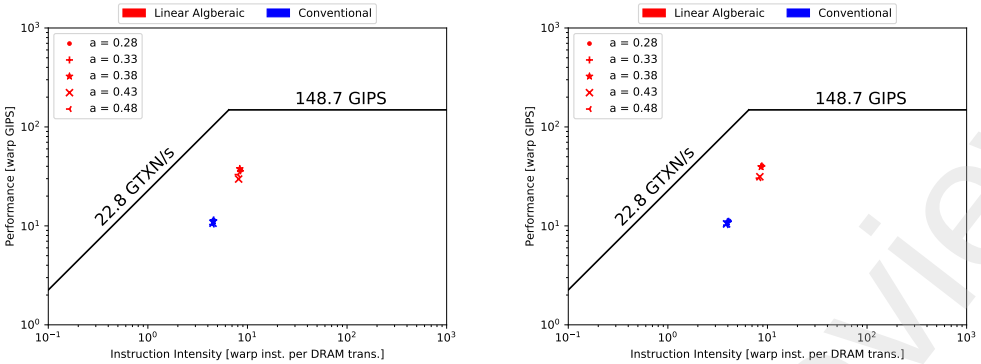
(d) Density = 8



(e) Density = 4

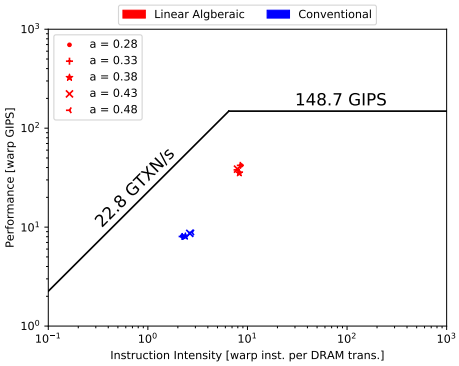
(f) Density = 2

Fig. 11. Roofline charts of PR algorithm for Watts-Strogatz datasets

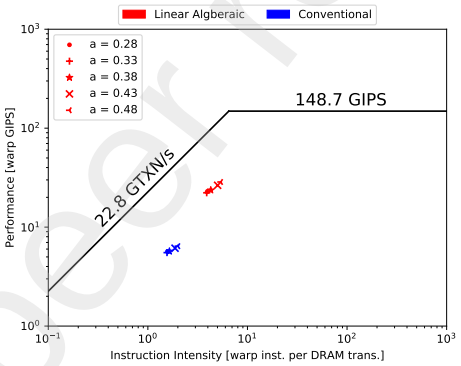


(a) Density = 64

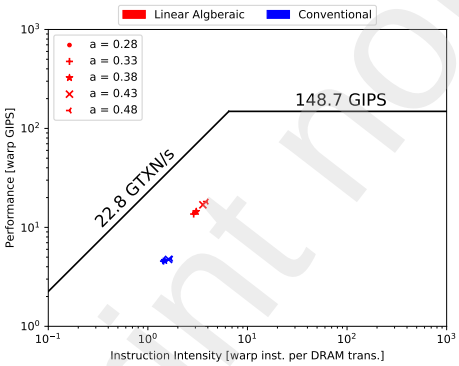
(b) Density = 32



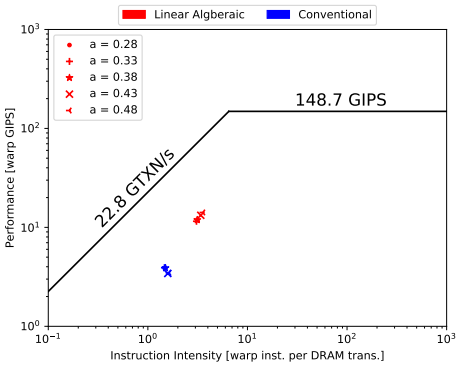
(c) Density = 16



(d) Density = 8

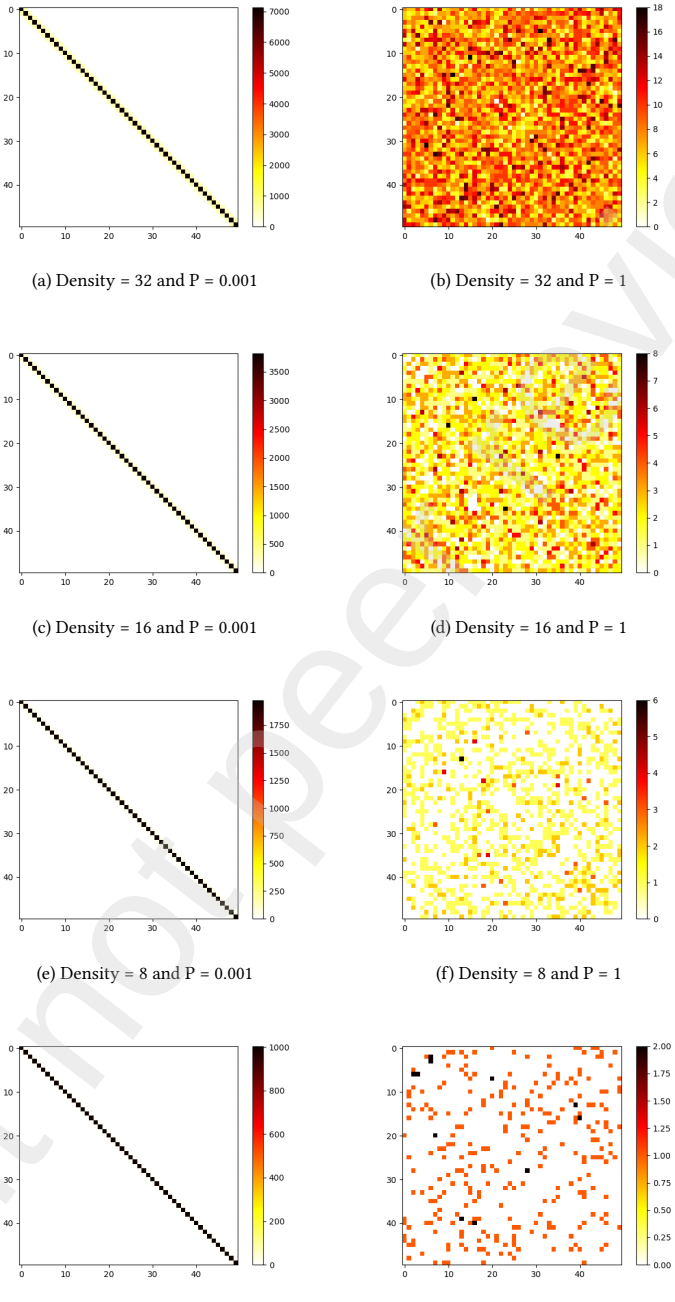


(e) Density = 4



(f) Density = 2

Fig. 12. Roofline charts of PR algorithm for R-MAT datasets



(a) Density = 32 and P = 0.001

(b) Density = 32 and P = 1

(c) Density = 16 and P = 0.001

(d) Density = 16 and P = 1

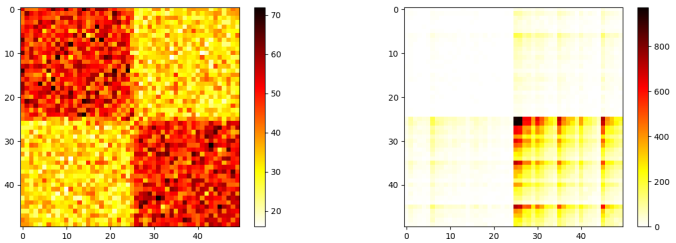
(e) Density = 8 and P = 0.001

(f) Density = 8 and P = 1

(g) Density = 4 and P = 0.001

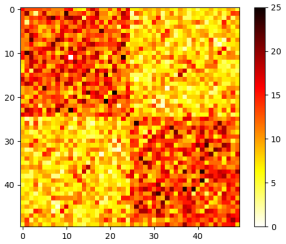
(h) Density = 4 and P = 1

Fig. 13. Heatmap charts of Watts-Strogatz graphs

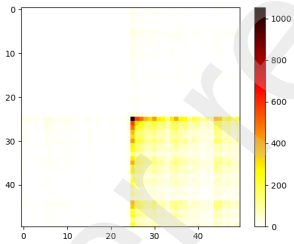


(a) Density = 32 and a = 0.28

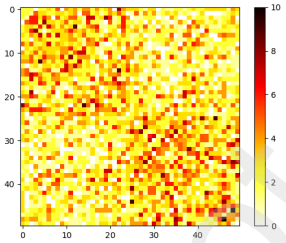
(b) Density = 32 and a = 0.48



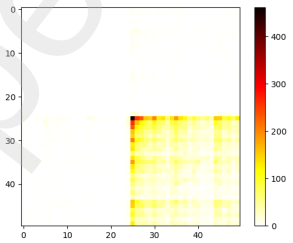
(c) Density = 16 and a = 0.28



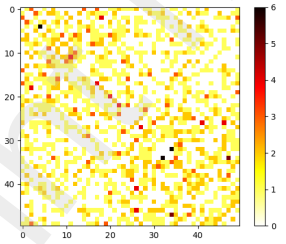
(d) Density = 16 and a = 0.48



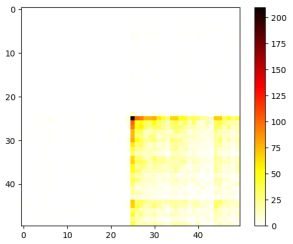
(e) Density = 8 and a = 0.28



(f) Density = 8 and a = 0.48



(g) Density = 4 and a = 0.28



(h) Density = 4 and a = 0.48

Fig. 14. Heatmap charts of R-MAT graphs

B.2 Memory Efficiency Analysis

In this section, we examine the effect of density and non-zero distribution on global load/store and shared memory efficiency (the ratio of the throughput requested by the application to the throughput achieved by the hardware). We only use the Watts-Strogatz model because it provides all the properties we need to tune. We analyze density by modifying edge count because memory efficiency is directly affected by increasing number of edges and number of memory transactions. We generate a wide range of datasets, setting the node count to 20000 and increasing the density from 20 to 10000. For each density, we generate graphs with values of p from 0.001 to 1. Table 10 summarizes the results of both global and shared memory efficiency for all the generated graphs.

Table 10. Memory efficiency metrics in Watts-Strogatz graph datasets

Datasets		shared_efficiency	gld_efficiency	gst_efficiency
Density = 10	P = 0.001	27.09	76.73	55.59
	P = 0.01	27.43	76.24	55.03
	P = 0.1	28.43	68.92	55.71
	P = 0.5	28.01	51.09	55.32
	P = 1	28.13	40.91	55.01
Density = 50	P = 0.001	30.24	75.26	48.35
	P = 0.01	30.63	73.99	48.37
	P = 0.1	31.17	63.41	48.36
	P = 0.5	31.21	38.96	48.42
	P = 1	31.24	28.85	48.33
Density = 100	P = 0.001	38.72	76.98	34.24
	P = 0.01	38.81	75.72	34.28
	P = 0.1	39.04	64.51	34.23
	P = 0.5	39.01	39.21	34.22
	P = 1	38.98	28.93	34.24
Density = 1000	P = 0.001	53.91	81.22	12.56
	P = 0.01	53.92	79.75	12.56
	P = 0.1	53.89	67.57	12.56
	P = 0.5	53.89	41.99	12.56
	P = 1	53.89	32.57	12.56
Density = 10000	P = 0.001	60.18	81.54	12.55
	P = 0.01	60.18	80.08	12.55
	P = 0.1	60.19	70.88	12.55
	P = 0.5	60.19	62.57	12.55
	P = 1	60.19	62.37	12.55

B.2.1 Global Efficiency. Table 10 shows that global load efficiency is only affected by the non-zero distribution. In fact, an increase in p decreases global load efficiency. This is because high values of p imply an increase in randomness and, therefore, an increase in random access patterns. Random access patterns cause a decrease in the spatial locality, which means that the threads within a warp access many cache lines. This decreases the L1 cache hit rate, which, by definition, causes an increase in L2 cache accesses. Therefore, the number of transferred bytes is more than the number of requested bytes (the application does not use all the transferred bytes), and the bandwidth is wasted. The results for metric l2_global_load_bytes in Table 11 confirm our reasoning. This metric represents the number of bytes read

from the L2 cache due to misses in the L1 cache for all global loads. We observe that an increase in p increases this metric. It is worth mentioning that for high density values, the graph is close to a complete graph, reducing the effect of randomness on global load efficiency.

We also observe that global store efficiency has an inverse relationship with dataset density. Table 10 shows that an increase in density results in an increase in the number of launched warps. We conclude that *LinAlg* uses more warps for computation on higher edge counts. In addition, through examining other metrics, we conclude that all intermediate matrix operations (e.g., element-wise operations, reduction, etc.) occur in shared memory, which means only the final results are written to global memory. Therefore, when we keep the nodes fixed and increase the edge counts, the number of launched warps increases and each warp operates on fewer rows. In this case, each warp writes to fewer elements in the output vector. In contrast, for low density datasets, each warp is responsible for more rows in the matrix and, consequently, more writes for elements in the output vector. In the first case, only a small portion of the bytes transferred by the hardware is used by the warp. However, in the second case, there are more requests from each warp and a noticeable portion of the transferred bytes are used.

B.2.2 Shared Efficiency. Table 10 shows that density is the only property that limits shared efficiency.

There are two important factors when considering shared efficiency: **bank conflict** and **broadcasting**. **Bank conflict** occurs when multiple threads in a warp access addresses that fall in the same memory bank. In this case, conflicting memory requests are split into as many separate conflict-free requests as necessary. For example, 2-way bank conflict occurs when there are at least two threads within a warp that access the same shared memory bank. In the case, two separate conflict-free shared memory transactions are generated for a single memory request. In other words, there is a difference between the number of bytes transferred by the hardware and the number of bytes requested by the application. **Broadcasting** occurs when all threads within a warp access the same word in a bank. In this case, there are only 4 bytes (the size of a word) requested by an application, however, to service all the warps, 128 bytes are transferred. This causes a reduction in shared efficiency.

To examine bank conflict, we consider the metric `shared_transactions_per_request`. This metric reports the average number of conflict-free transactions required for a conflicting memory request for all warps. For example, if all warps exhibit 2-way bank conflict, then this metric will equal two. If there are no bank conflicts, this metric will equal one. We report this metric in Table 11. We make two key observations. Frist, 2-way bank conflict occurs in datasets where density = 10 and they only achieve about 28% shared efficiency. Second, there are no bank conflicts in datasets where density = 10000 and shared efficiency is about 60%. We conclude that, for low-density datasets, both bank conflict and broadcasting affect shared efficiency, while in high-density datasets, the only limiting factor is broadcasting.

To determine whether broadcasting reduces performance or not, we examine IPC in two cases, one where broadcasting occurs, and the other where all threads access different warps. We observe that IPC is lower in the first case compared to the second case. Therefore, we can make the conclusion that the broadcasting mechanism used in modern GPUs in not fast enough and results in a reduction in performance.

Table 11. Metrics required for analysis of memory efficiency in Watts-Strogatz graph datasets

Datasets		l2_global_load_bytes	shared_transactions_per_request	unique_warps_launched
Density = 10	P = 0.001	3771913	1.72	2558
	P = 0.01	4032407	1.71	2558
	P = 0.1	4880452	1.71	2558
	P = 0.5	8812963	1.71	2558
	P = 1	13115697	1.71	2558
Density = 50	P = 0.001	16281504	1.66	4556
	P = 0.01	16560224	1.65	4556
	P = 0.1	19324704	1.63	4556
	P = 0.5	31449184	1.63	4556
	P = 1	42476544	1.63	4556
Density = 100	P = 0.001	31613408	1.26	9020
	P = 0.01	32148480	1.26	9020
	P = 0.1	37727616	1.25	9020
	P = 0.5	62072224	1.26	9020
	P = 1	84100864	1.26	9020
Density = 1000	P = 0.001	296219904	1.03	89376
	P = 0.01	301679136	1.03	89376
	P = 0.1	356068640	1.03	89376
	P = 0.5	572957056	1.03	89376
	P = 1	738731168	1.03	89376
Density = 10000	P = 0.001	2653966912	1.01	803664
	P = 0.01	2702338240	1.01	803664
	P = 0.1	3053126528	1.01	803664
	P = 0.5	3458770464	1.01	803664
	P = 1	3469896352	1.01	803664