CuSha: Vertex-Centric Graph Processing on GPUs

Farzad Khorasani Keval Vora Rajiv Gupta Laxmi N. Bhuyan

Computer Science and Engineering Department
University of California Riverside, CA, USA
{fkhor001, kvora001, gupta, bhuyan}@cs.ucr.edu

ABSTRACT

Vertex-centric graph processing is employed by many popular algorithms (e.g., PageRank) due to its simplicity and efficient use of asynchronous parallelism. The high compute power provided by SIMT architecture presents an opportunity for accelerating these algorithms using GPUs. Prior works of graph processing on a GPU employ *Compressed Sparse Row* (CSR) form for its space-efficiency; however, CSR suffers from irregular memory accesses and GPU underutilization that limit its performance. In this paper, we present **CuSha**, a CUDA-based graph processing framework that overcomes the above obstacle via use of two novel graph representations: *G-Shards* and *Concatenated Windows* (CW). *G-Shards* uses a concept recently introduced for non-GPU systems that organizes a graph into autonomous sets of ordered edges called *shards*. CuSha's mapping of GPU hardware resources on to shards allows fully coalesced memory accesses. CW is a novel representation that enhances the use of shards to achieve higher GPU utilization for processing *sparse graphs*. Finally, CuSha fully utilizes the GPU power by processing multiple shards in parallel on GPU's streaming multiprocessors. For ease of programming, CuSha allows the user to define the vertex-centric computation and plug it into its framework for parallel processing of large graphs. Our experiments show that CuSha provides significant speedups over the state-of-the-art CSR-based virtual warp-centric method for processing graphs on GPUs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Performance

Keywords

GPU, Graph Representation, G-Shards, Concatenated Windows, Coalesced Memory Accesses

HPDC'14, June 23–27, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2600212.2600227 .

1. INTRODUCTION

The need for efficient large scale graph processing has grown due to the importance of applications involving graph mining and graph analytics. However, using GPUs for efficient graph processing remains a challenging open problem. Even though GPUs provide a massive amount of parallelism with the potential to outperform CPUs, the SIMD architecture demands repetitive processing patterns on regular data which is contrary to the irregular nature of graphs. This leads to the problems of irregular memory accesses and underutilization of GPUs; thus limiting the performance of graph algorithms on GPUs.

Existing graph processing techniques [10, 12, 21] primarily rely on the Compressed Sparse Row (CSR) representation of graphs because CSR consumes minimal storage space. However, accesses involving a node's neighbors lead to poor locality causing large amounts of random input-dependent memory references, popularly known as *non-coalesced accesses*. Also, these techniques are inherently fraught with GPU underutilization caused by workload imbalance resulting from mapping of irregular graphs to the GPU's symmetric hardware architecture.

In this paper we present **CuSha**[1], a framework for processing graphs on GPUs, that overcomes the drawbacks associated with the CSR representation. We recognize and explore the potential of a recently introduced representation for efficient disk based graph processing, known as *shards* [14]. Shards distribute graph data in a manner that places edges and vertices required by a subset of computation contiguously in memory. *G-Shards* adapts the shard based representation to efficiently process graphs on GPUs. In G-Shards each shard becomes a workload for a GPU thread block and multiple thread blocks are processed in parallel. Within each block, entries of a shard (representing edges) are processed in parallel by the threads. By mapping shards to blocks in this manner, we leverage the parallelism in the computation involving both vertices and edges. Even though G-Shards provides better locality, it is sensitive to the nature of input graphs. For large sparse graphs – large real world graphs are often sparse – the workloads assigned to warps inside a block become too small causing threads within the block to remain idle leading to GPU underutilization. To efficiently process sparse graphs, we propose a modification of G-Shards called *Concatenated Windows* (CW). CW representation concatenates multiple computation windows from shards so that GPU threads are highly utilized.

[1]Available at *http://farkhor.github.io/CuSha.*

We have developed a CUDA based prototype of the *CuSha* graph processing framework that internally makes use of G-Shards and CW representations. CuSha relies on an iterative vertex-centric model where a given compute function is iteratively applied to each vertex in the graph until a convergence condition is met. This allows developers to easily program graph applications using CuSha – programmer provides simple processing functions which deal with a vertex and its neighbors and the framework automatically parallelizes the computation over the entire input graph. As opposed to the Bulk Synchronous Parallel (BSP) model [26], CuSha provides asynchronous execution that lets updated vertex values to be visible during the same iteration; hence enabling faster convergence for iterative graph algorithms.

The key contributions of this work are as follows:

- We recognize the potential of shards and introduce an effective mapping of shards to various GPU subcomponents via the *G-Shards* representation. On average, our approach improves memory load and store efficiency by nearly 26% and 52% respectively.
- We propose *Concatenated Windows*, an extension built upon G-Shards to leverage better locality. On average, for large sparse graphs, CW improves GPU utilization by 57%.
- We implemented *CuSha*, a vertex-centric framework that internally uses G-Shards and Concatenated Windows to represent graphs. CuSha allows non-expert developers to quickly implement graph algorithms on GPUs without worrying about the inner details related to parallelization and synchronization.
- We demonstrate that *CuSha* outperforms state-of-the-art warp-centric algorithm [12] across wide range of benchmarks and large real world input graphs. For *PageRank*, average speedup of 7.21x is observed across the input graphs.

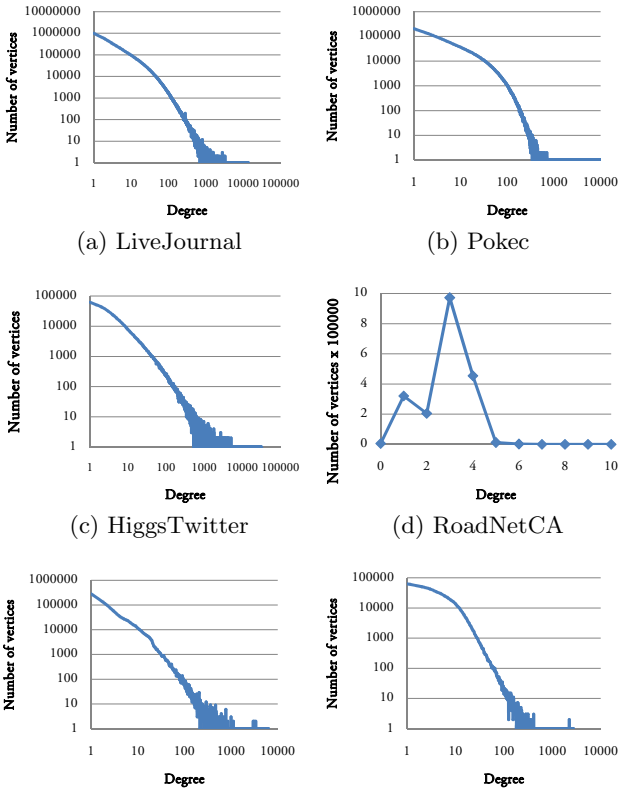
The rest of the paper is organized as follows. Section 2 overviews and evaluates state-of-the-art CSR-based virtual warp-centric method using several graph applications and large real world graphs. Section 3 presents the graph representations we proposed (G-Shards and Concatenated Windows) to overcome drawbacks of CSR. Section 4 presents details of the CuSha framework including the iterative execution model and the easy to use programming interface. Experimental evaluation is presented in Section 5. Sections 6 and 7 present related work and conclusion.

2. MOTIVATION: LIMITATIONS OF CSR

Representing graphs in memory to efficiently process them on GPUs has been a challenging task. Consider examples of some real world graphs shown in Table 1 whose degree distribution is shown in Figure 1. As we can see, these graphs are usually sparse and their sizes are large involving processing over millions of vertices and edges. The latter makes it infeasible to store the graph in the space inefficient *adjacency matrix* representation. Hence, prior graph processing approaches [22, 8, 4] primarily rely on the *Compressed Sparse Row* (CSR) representation because of its compact nature. For any given vertex, the CSR representation allows fast access to its incoming/outgoing edges along with the addresses of source/destination vertices at the other end of these edges. The representation mainly consists of 4 arrays:

Graph	Edges	Vertices
LiveJournal [17]	68 993 773	4 847 571
Pokec [25]	30 622 564	1 632 803
HiggsTwitter [7]	14 855 875	456 631
RoadNetCA [17]	5 533 214	1 971 281
WebGoogle [17]	5 105 039	916 428
Amazon0312 [16]	3 200 440	400 727

Table 1: Real-world graphs used in the experiments.

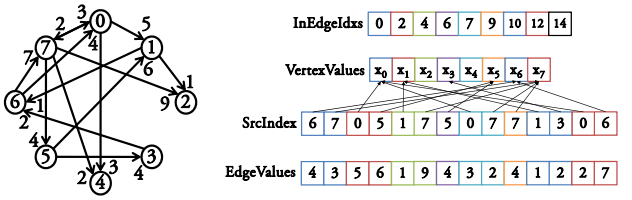


(a) LiveJournal (b) Pokec (c) HiggsTwitter (d) RoadNetCA (e) WebGoogle (f) Amazon0312

Figure 1: Degree distribution for graph vertices.

- *VertexValues*: $VertexValues[i]$ $(0 \leq i < n)$ represents the value of vertex v_i.
- *SrcIndxs*: $SrcIndxs[i]$ $(0 \leq i < m)$ represents for edge e_i, the index of the source vertex in *VertexValues*. The incoming edges for a given vertex are stored in consecutive locations of this array.
- *InEdgeIdxs*: $InEdgeIdxs[n] = m$. $InEdgeIdxs[i]$ $(0 \leq i < n)$ represents the starting index of a sub-array E_i of *SrcIndxs*. The end of this sub-array E_i can be determined by the entry at $i + 1$. E_i combined with *SrcIndxs* represents the incoming edges for node n_i.
- *EdgeValues*: $EdgeValues[i]$ $(0 \leq i < m)$ represents the value of the edge e_i.

The neighborhood of vertex n_i can be determined by looking at locations of *VertexValues* which are represented by the sub-array starting at $SrcIndxs[InEdgeIndxs[i]]$ and ending at $SrcIndxs[InEdgeIndxs[i + 1]]$ and the edge weights can be determined by the sub-array of *EdgeValues* starting at $InEdgeIndxs[i]$ and ending at $InEdgeIndxs[i+1]$. Figure 2



(a) Example graph. (b) CSR representation of the graph.

Figure 2: An example graph and its CSR representation.

shows an example of a graph and its CSR representation using its incoming edges. As we can see, the neighborhood of vertex 2 (shown in green) is represented by *VertexValues*[1] and *VertexValues*[7] and the sub-array *EdgeValues*[4:5].

To process graphs on GPUs, Virtual Warp-Centric technique [12] has been shown to perform better than other techniques like [10]. Here, the physical warp is broken into 2, 4, 8 or 16 smaller virtual warps to control the trade-off between GPU underutilization and path divergence. Processing is done iteratively such that each iteration is performed by a separate GPU kernel call. In each iteration, a virtual warp handles the computation of a set of vertices. For each vertex, threads within the virtual warp process the vertex neighbors in parallel; each of its incoming edges is read and processed by a separate thread in the virtual warp. Then, parallel reduction technique [11] is used to calculate the new vertex value. Even though Virtual Warp-Centric method achieves faster graph processing compared to other techniques, its performance is limited by two major phenomena: high *non-coalesced memory accesses*; and high *GPU underutilization & intra-warp path divergence*. We elaborate upon these drawbacks next.

***Non-Coalesced Accesses*:** As discussed, $SrcIndxs[i]$ and $SrcIndxs[i+1]$ represent non-consecutive indices in *VertexValues* array. Hence, parallel reading of these values by threads in a virtual warp leads to random non-coalesced memory accesses requiring multiple inefficient memory transactions. Note that a major portion of graph processing is reading these vertex structures over and over again, making the problem very significant. Table 2 shows the average efficiency of memory accesses for different applications. Global memory access efficiency essentially tells how well coalesced global accesses are. Such low percentages of efficiency indicates that a great number of accesses were fulfilled using greater than minimal number of transactions due to poor locality of data of interest.

***Underutilization & Intra-warp Divergence*:** Graph processing is highly sensitive to the degrees of vertices in the input graph. Processing of a low-degree vertex causes threads within the virtual warp to remain idle, leading to GPU underutilization. If we select a smaller virtual warp size to decrease underutilization, different amounts of computation load for threads inside the physical warp cause intra-warp path divergence. Figure 1 shows that real world graphs have a mix of low and high degree vertices. Table 2 shows that the warp execution efficiency is quite low for eight graph applications on different input graphs. High intra-warp divergence and GPU underutilization limit warp execution efficiency; thus, degrading the overall performance of the GPU kernel.

Application Name	Global Memory Accesses	Warp Execution
Breadth-First-Search (BFS)	12.8%-15.8%	27.8%-38.5%
Single Source Shortest Path (SSSP)	14.0%-19.6%	29.7%-39.4%
PageRank (PR)[23]	10.4%-14.0%	25.3%-38.0%
Connected Components (CC)	12.7%-20.6%	29.9%-35.5%
Single Source Widest Path (SSWP)	14.5%-20.0%	29.7%-38.4%
Neural Network (NN)[3]	13.5%-17.8%	28.2%-37.4%
Heat Simulation (HS)	14.5%-18.1%	27.6%-36.3%
Circuit Simulation (CS)	12.0%-18.8%	28.4%-35.5%

Table 2: CSR-based Virtual Warp-Centric method [12] for graphs in Table 1: Minimum and maximum efficiency of global memory accesses and warp execution across all iterations between all graphs.

In conclusion, even though CSR representation is a popular choice because of its space-efficiency, high frequency of non-coalesced memory accesses because of poor locality and path divergence because of variable degree distribution of real graphs, significantly limit its performance while processing graphs on GPUs. In addition, the user is always trapped in a trade-off between intra-warp path divergence and GPU underutilization which has a different best configuration for different graphs. This motivates the need to explore novel graph representations that are GPU friendly and allow faster processing.

3. CUSHA GRAPH REPRESENTATIONS

In this section we discuss two graph representations that result in improved coalescence in memory accesses and high GPU utilization and hence achieve higher performance.

3.1 G-Shards

Representing a graph with shards has been shown to improve I/O performance for disk based graph processing on a shared memory system [14]. Since shards allow contiguous placement of the graph data required by a subset of computations, G-Shards uses the shard concept to secure benefits from coalesced accesses.

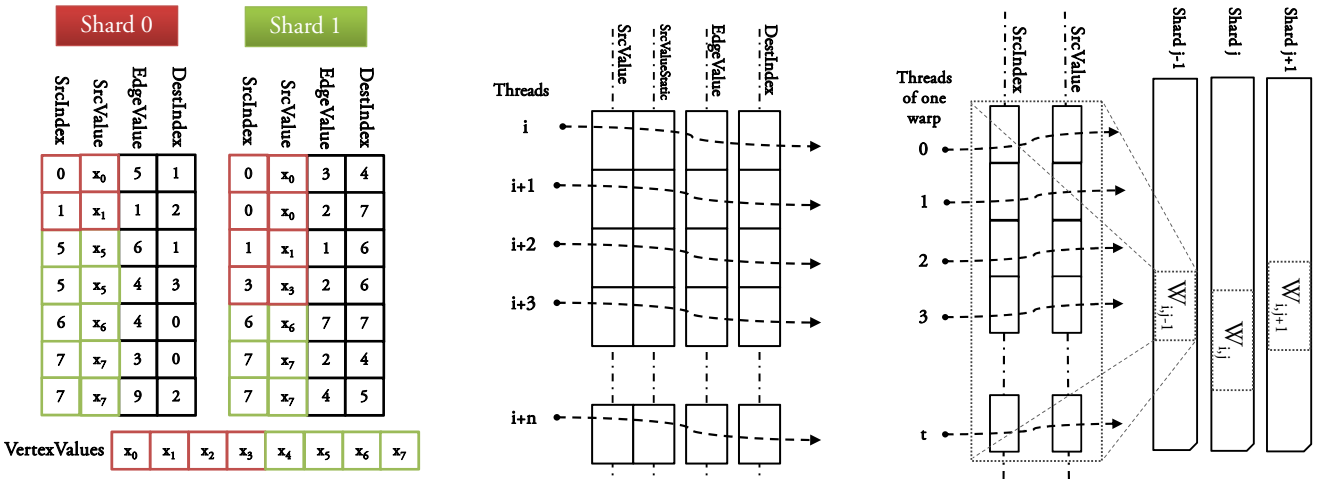
G-Shards presents a graph G as a set of shards where each shard is an ordered list of incoming edges and each edge $e = (u, v)$ in the shard is represented by a 4-tuple:

- *SrcIndex*: Index of the source vertex u
- *SrcValue*: Content of source vertex u
- *EdgeValue*: Content or weight of the edge e
- *DestIndex*: Index of the destination vertex v

The set of shards used to represent a graph G exhibit the following properties:

- *Partitioned*: V is partitioned into disjoint sets of vertices and each set is represented by a shard such that it stores all the edges whose destination is in that set.
- *Ordered*: The edges in a shard are listed based on increasing order of their *SrcIndex*.

Figure 3(a) shows G-Shards representation for the graph shown in Figure 2(a). We divide the vertex-set into two groups so that *Shard-0* has the list of edges whose destination is between 0 to 3 and *Shard-1* has the list of edges whose



(a) *G-Shards* representation of the graph in Figure 2(a).

(b) Global memory accesses inside a shard in step 2.

(c) Global memory accesses in a window in write-back stage (step 4).

Figure 3: G-Shards representation providing coalesced memory accesses.

destination is between 4 to 7. Note that within *Shard-0* (and *Shard-1*), the edges are sorted based on *SrcIndex*.

To facilitate efficient processing of graphs on GPU using shards, G-Shards also maintains a separate array named *VertexValues* which allows quick access to values of vertices. Throughout the computation. *VertexValues*[i] represents the most updated value of vertex v_i.

Each shard is processed by a block in the GPU in 4 steps. In step 1, the threads fetch the updated vertex values from the *VertexValues* array to the shared memory of the block. Consecutive threads of the block read consecutive elements of *VertexValues* array; hence load requests coalesce into minimum number of transactions. In step 2, using the fetched values, block threads process edges inside the shard in parallel. Figure 3(b) shows consecutive threads of the block read consecutive shard entries residing in global memory thus providing coalesced global memory loads. In step 3, the threads write back the newly computed values to the *VertexValues* array. This step is done in a similar manner as step 1 except that reads are replaced by writes. Thus, global memory stores in this step, similar to global loads in step 1, are satisfied by minimum number of write transactions in memory controller. Step 4 (write-back stage) performs the remaining task which is to propagate computed results to other shards *SrcValue* array. To have coalesced global memory accesses in write-back stage as well, we assign each warp in the block to update necessary SrcValue elements in one shard. Because of aforementioned *Ordered* property of shards, elements in one shard that need to be read and written by another shard are arranged contiguously. Figure 3(c) shows consecutive threads inside a warp read consecutive *SrcIndex* elements inside another shard and write to consecutive *SrcValue* elements; therefore memory accesses are coalesced.

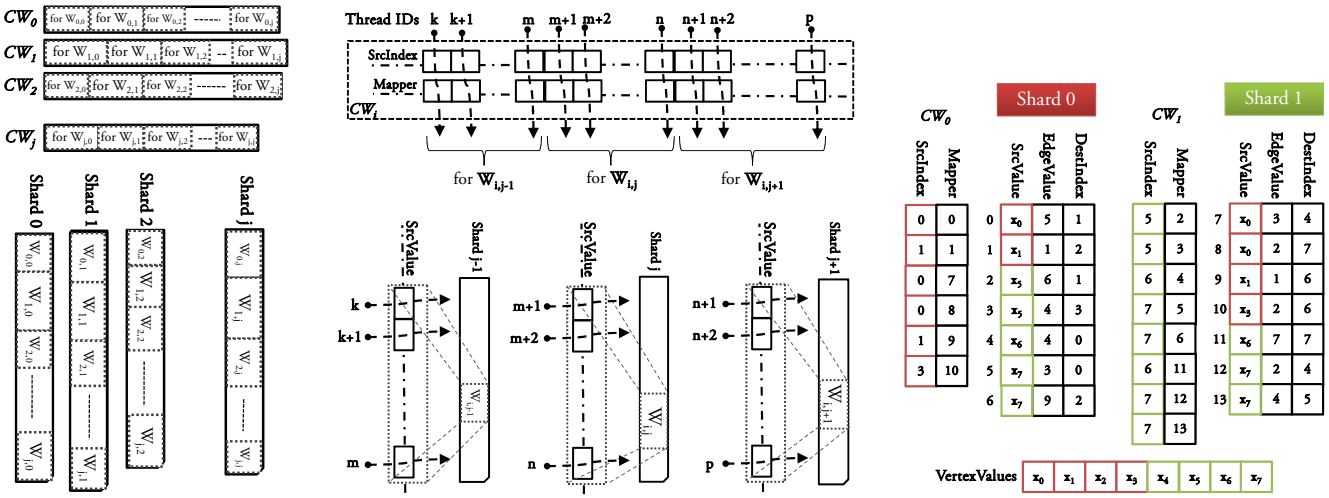
Thus, we observe that shard processing by threads of a block on GPU involves fully coalesced global memory reads and writes during all steps. Accesses to *VertexValues* in steps 1 and 3, reading shard elements in step 2, and updating regions of other shards in step 4 all become coalesced.

Each shard region whose elements need to be accessed and updated together by another shard is called a computation window. A computation window W_{ij}, is the set of entries in shard j that are involved during processing of shard i such that, each edge in W_{ij} has *SrcIndex* in the range of vertex indices associated with shard i. This means that the source vertices of all the edges in W_{ij} belong to the vertex-range a to b if shard i represents edges whose destination vertices belong to the same range a to b. As an example, different colors for entries in Figure 3(a) distinguish different computation windows; the windows W_{0j} are represented in red (first two elements of shard-0 and first four elements of shard-1) and windows W_{1j} are represented in green for $0 \leq j < 2$. Intuitively, for a constant k, if number of shards is p, the collection of edges in windows W_{kj}, $0 \leq j < p$, completely represents the sub-graph induced over the subset of vertices associated with shard k.

3.2 Concatenated Windows (CW)

G-Shards representation on GPU provides coalesced global memory accesses to neighbor's contents which was not achievable by CSR. However, the performance can be limited by various characteristics of input graphs. First, imbalanced shard sizes can cause inter-block divergence. We found that this effect is insignificant because of the abundance of shards to be processed that keeps the Streaming Multiprocessors busy. Second, unbalanced window sizes can cause intra-block/inter-warp divergence; however, due to similar reason, we conclude that its impact is insignificant too. Finally, sparse graphs lead to small window sizes which cause GPU underutilization. This is mainly because most of the threads within the warp are idle when entries for the computation window are being processed by other threads. This makes G-Shards representation on GPUs sensitive to window sizes; in particular, smaller window sizes lead to inefficient write-back of updated values in the windows.

Next we show that the window size is mainly determined by the size of the input graph, its sparsity, and the number of vertices assigned to shards. Let us consider a graph $G = (V, E)$ to be represented by $|S|$ shards. The average shard size (the number of edges in the shard) is $\frac{|E|}{|S|}$. Since each shard has at most $|S|$ windows (one for each shard), the average window size becomes $\frac{|E|}{|S|^2}$. Assuming that a shard



(a) CW_i formed by concatenating all W_{ij} ordered by j.

(b) Global memory accesses in write-back stage using CW representation.

(c) CW representation of graph in Figure 2(a).

Figure 4: Concatenated Windows Representation and its Avoidance of GPU underutilization in presence of small windows.

is assigned at most $|N|$ vertices, $|S| = \lceil \frac{|V|}{|N|} \rceil$. Hence, the average window size is approximately $\frac{|E||N|^2}{|V|^2}$. Thus, the windows become smaller as the graph becomes sparser. Also, the quadratic term in the denominator indicates that the windows rapidly become smaller as the number of vertices in the graph increases.

We develop the *Concatenated Windows* representation to address the above issue. To avoid GPU underutilization induced by large sparse graphs, CW collocates computation windows. For a given shard i, a Concatenated Window CW_i is defined as a list of *SrcIndex* elements of all computation windows W_{ij}, ordered by j, as shown in Figure 4(a). Here, we retain the original representation of shards, but separate out the *SrcIndex* entries to order them differently. *SrcIndex* entries for shard i in CW representation can be created by concatenating *SrcIndex* in all W_{ij} in G-Shards representation. Hence, a directed graph is represented as a set of shards, each of them associated with a separate *SrcIndex* array. Each shard is an ordered list of incoming edges where each edge is now represented by a 3-tuple: *SrcValue*, *EdgeValue*, and *DestIndex*. The set of shards is *Partitioned* and *Ordered* as described in the previous section. Note that by separating out *SrcIndex* array from the rest of the shard, we break the association between *SrcIndex* and *SrcValue* entries which is required to write-back the updated values. Therefore, to facilitate fast access of *SrcValue* entries using *SrcIndex* entries, we use an additional *Mapper* array.

Processing graphs using Concatenated Windows, similar to G-Shards representation, takes 4 steps with a difference in 4th step (write-back stage). As Figure 4(b) shows, a thread is assigned to every entry of *SrcIndex*. Using the entries in *SrcIndex* and *Mapper* arrays, the thread updates corresponding *SrcValue* entries in the shard. By concatenating small windows to form a larger set, consecutive threads within the block continuously process consecutive entries, thus improving GPU utilization for large sparse graphs.

Figure 4(c) shows the Concatenated Windows representation for the graph shown in Figure 2(a). As we can see, the *SrcIndex* columns are separately ordered compared to the rest of the shards. There are six entries in *SrcIndex* (shown in red) associated with *Shard-0* representing values from CW_0. The first two entries come from W_{00} and the rest come from W_{01}. The eight entries in *SrcIndex* (shown in green) are associated with *Shard-1* and represent values from CW_1. The first five come from W_{10} and the rest come from W_{11}.

By retaining the basic representation of G-Shards, we leverage the locality of data required by each computation set. By changing the ordering of the *SrcIndex* column using Concatenated Windows, we benefit from higher utilization of threads in warps, thus achieving best of both worlds.

4. CUSHA FRAMEWORK

In this section we first describe how CuSha executes iterative parallel graph algorithms. Then we show that CuSha makes various applications easy to program.

Iterative parallel graph processing in CuSha. The parallel execution framework of CuSha implements iterative parallel graph processing where each iteration performs three phases: *gather/read*, *update/compute* and *scatter/write*. CuSha's computational model is largely based on the *read-compute-write* iterative processing mechanism where the *compute* phase is further split into two phases.

Figure 5 shows the pseudo-code for iterative processing. The host continuously launches new GPU kernels until the algorithm converges to a stable solution. At the end of each iteration, the host and the device implicitly synchronize using a *cudaMemcpy* that copies *is_converged* back to CPU-side (line 29). After each iteration, the CPU determines whether or not the next iteration should be performed by launching another GPU kernel.

Each shard is completely processed by one GPU block. For each shard, a *values_updated* flag resides in the shared memory and indicates whether or not the values were updated. It is initially set to false (line 5) by one of the threads in the block. Appropriate *Vertex*, *Edge* and *StaticVertex* structures are initialized where the *StaticVertex* structure refers to properties of the vertex that remain constant throughout the execution.

```
0.  is_converged = false;
1.  while (!is_converged) {
2.    is_converged = true;
3.    parallel-for shard s in shards {
4.      shared Vertex local_vertices[N];
5.      shared values_updated = false;
6.      offset = s.ID * N;
7.      Vertices = VertexValues+offset;
        /* 1st stage */
8.      parallel-for vertex index v in s {
9.        init_compute( local_vertices+v,
           Vertices+v );
10.     }
11.     synchronize; //synchronizes block threads
        /* 2nd stage */
12.     parallel-for edge index e in s {
13.       compute( s.SrcValue+e, s.SrcValueStatic+e,
            s.EdgeValue+e,
            local_vertices+s.DestIndex[e]-offset );
14.     }
15.     synchronize; //synchronizes block threads
        /* 3rd stage */
16.     parallel-for vertex index v in s {
17.       if ( update_condition
           ( local_vertices+v, Vertices+v ) ) {
18.         Vertices[v] = local_vertices[v];
19.         values_updated = true;
20.       }
21.     }
22.     synchronize; //synchronizes block threads
        /* 4th stage */
23.     if( values_updated ) {
24.       w = window_set_from_all_shards(s);
          //Windows in all the shards for shard-k
25.       write_back( local_vertices, w );
26.       is_converged = false;
27.     }
28.   }
29.   barrier;
30. }
```

Figure 5: Graph processing procedure in CuSha.

In the *first stage*, consecutive threads initialize *local_vertices* array from consecutive *VertexValues* array elements in the global memory (line 9) thus providing fully coalesced memory accesses.

The *second stage* mainly involves invoking the *compute* method with the appropriate parameters for shards. Global memory access pattern in this stage is depicted in Figure 3(b). Since multiple threads can simultaneously modify the same shared memory location, the user-provided *compute* function must be *atomic* with respect to updating the destination vertex. Note that the atomic operation will be inexpensive mainly because it is a shared memory update and hence, only can affect other threads inside the same streaming multiprocessor, leaving the threads in other streaming multiprocessors unaffected. Furthermore, the lock contention is low because of the size of shards, allowing these operations to be performed almost independently with respect to each other. Also, since the order of these invocations is non-deterministic, the compute function must be both, *commutative* and *associative*.

In the *third stage*, the threads invoke the *update_condition* method (lines 17 - 19). If a true value is returned by this method, the threads update the contents of *VertexValues* and *values_updated* is set. Note that the *update_condition* method can also be used to perform computations unique to each vertex. In this case, the computation logic can be split across the *compute* and *update_condition* methods such that

```
0.  typedef struct Edge { unsigned int Weight; } Edge;
1.  typedef struct Vertex { unsigned int Dist; } Vertex;
2.  __device__ void init_compute(
     Vertex* local_V, Veretx* V ) {
3.    local_V->Dist = V->Dist;
4.  }
5.  __device__ void compute(
     Vertex* SrcV, StaticVertex* SrcV_static,
     Edge* E, Vertex* local_V) {
6.    if (SrcV->Dist != INF)
7.      atomicMin ( &(local_V->Dist),
          SrcV->Dist + E->Weight );
8.  }
9.  __device__ bool update_condition(
     Vertex* local_V, Vertex* V ) {
10.   return ( local_V->Dist < V->Dist );
11. }
```

Figure 6: SSSP implementation in CuSha.

the *compute* method mainly leverages edge-level parallelism and the *update_condition* method leverages vertex-level parallelism.

In the *last stage*, if *values_updated* flag is set, windows in all the shards are updated with newly computed values (line 25) and one of the threads inside the block sets the *is_converged* flag. With G-Shards representation, the threads of a block are grouped into warps which iterate through the corresponding windows in all the shards (Figure 3(c)). As we discussed in Section 3.2, although the accesses are coalesced in this case, the technique is susceptible to GPU underutilization when windows are small. With CW representation, the threads of a block read the *SrcIndex* and *Mapper* array and appropriately update the windows in all shards (Figure 4(b)). Even though the memory accesses are not fully coalesced in this case, it requires the same number of memory transactions as in G-Shards representation with the added benefit of utilizing all threads.

Selecting shard size. As we know, processing graphs using G-Shards and CW representations is sensitive to window sizes which, in turn, is dependent on the size of shards. Hence, during initialization, CuSha determines the number of vertices assigned to shards for each input graph using average window size formula: $\frac{|E||N|^2}{|V|^2}$ derived in Section 3.2. From the architecture standpoint, $|N|$ is limited by the size of shared memory; to achieve maximum theoretical occupancy and to fully utilize the SM resources, $|N|$ is dependent on the number of blocks residing on a single SM. For example, if a SM has 48KB shared memory and we wish to have two blocks residing in it at the same time, each block can be assigned up to 24KB of shared memory. Assuming that vertex value is 4 bytes, $|N|$ can at most be 6K. Similarly, with four blocks on one SM, $|N|$ can at most be 3K. Choosing the largest value for $|N|$ (6K in the above example) and hence, having a block with a lot of threads increases the likelihood of conflicts during atomic operations due to limited number of shared memory lock indices. Hence, for a given input graph, CuSha first calculates $|N|$ by assuming the average window size to be 32 (equal to the warp size). Then, it determines block size to be the nearest value to the calculated $|N|$ that utilizes all available shared memory quota for the block on the SM. This allows CuSha to generate the set of shards that is best suited for each input graph.

Programming applications using CuSha. The above computational model enables users to easily implement a

Benchmark	typedef struct Vertex { }Vertex;	typedef struct StaticVertex { }StaticVertex;	typedef struct Edge { }Edge;	__device__ void init_compute(Vertex* local_V, Veretx* V){}	__device__ void compute(Vertex* SrcV, StaticVertex* SrcV_static, Edge* E,Vertex* local_V){}	__device__ bool update_condition(Vertex* local_V, Vertex* V){}
BFS	unsigned int Level;			local_V->Level =V->Level;	if(SrcV->Level!=INF) atomicMin(&(local_V->Level) ,SrcV->Level+1);	return(local_V->Level < V->Level);
SSSP	unsigned int Dist;		unsigned int Weight;	local_V->Dist =V->Dist;	if(SrcV->Dist!=INF) atomicMin(&(local_V->Dist) ,SrcV->Dist+E->Weight);	return(local_V->Dist < V->Dist);
PR	float Rank;	unsigned int NbrsNum;		local_V->Rank =0;	unsigned int nbrsNum= SrcV_static->NbrsNum; if(nbrsNum!=0)atomicAdd (&(local_V->rank), SrcV->Rank/nbrsNum);	local_V->rank= (1-DAMPING_FACTOR)+local_V ->rank*DAMPING_FACTOR; return(fabs(local_V-> rank-V->rank)>TOLERANCE);
CC	unsigned int Cmpnent;			local_V->Cmpnent = V->Cmpnent;	atomicMin(&(local_V->Cmpnent) ,SrcV->Cmpnent);	return(local_V->Cmpnent < V->Cmpnent);
SSWP	unsigned int BWidth;		unsigned int Width;	local_V->BWidth =V->BWidth;	if(SrcV->BWidth!=0) atomicMax(&(local_V->BWidth) ,min(SrcV->BWidth,E->Width));	return(local_V->BWidth > V->BWidth);
NN	float x;		float Weight;	local_V->x =0;	atomicAdd(&(local_V->x) ,SrcV->x*E->weight);	local_V->x= tanh(local_V->x); return(fabs(local_V->x - V->x)>TOLERANCE);
HS	float Q; float Q_new;		float coeff;	local_V->Q=V->Q; local_V->Q_new =local_V->Q;	atomicAdd(&(local_V->Q_new) ,(SrcV->Q-local_V->Q) *E->coeff);	bool B=fabs(local_V->Q- local_V->Q_new)>TOLERANCE; if(B) local_V->Q= local_V->Q_new; return B;
CS	float V; float GsumOrA;		float G;	local_V->V=0; local_V-> GsumOrA=0;	float G=E->G; atomicAdd(&(local_V->V) ,SrcV->V*G); atomicAdd(&(local_V ->GsumOrA),G);	if(V->GsumOrA){ local_V-> GsumOrA=1; local_V->V=V->V; return false;} else if(local_V->GsumOrA){ local_V->V /= local_V->GsumOrA; local_V-> GsumOrA=0; return(fabs(local_V->V-V->V)>TOLERANCE);} else return false;

Table 3: Implementation of various benchmarks in CuSha.

wide range of graph processing algorithms. As an example, let us consider the implementation of Single Source Shortest Path (SSSP) algorithm. The vertex-centric approach for SSSP is to iteratively compute the value for each vertex based on the minimum sum of its neighbor's value and the corresponding edge weight. Figure 6 presents the structure of a vertex and the functions required to compute SSSP on a graph. Every vertex holds an integer (initially set to a very large number representing ∞) standing for the shortest distance from the source (line 0). Source vertex value is set to 0. At the beginning of each iteration, the *init_compute* method loads the most updated vertex values into the block's shared memory. The *compute* function sets the distance of a vertex by atomically choosing the minimum of the calculated distances. The *update_condition* signals the caller to execute the next iteration if the new distance of the vertex is smaller than its old value. As we can see, the user only has to provide the *init_compute*, *compute*, and *update_condition* methods along with the *required structures*; hence making it easier to code graph processing algorithms using CuSha. Also, the commonalities among various algorithms allow users to quickly implement different algorithms by simply modifying the existing ones. Table 3 presents 8 graph processing algorithms we implemented using CuSha alongside variables for structures and instructions for three functions used by these algorithms. Note that having arrays of structure in older generations of CUDA devices could limit the effective bandwidth due to strided distribution of elements. However, simultaneous accesses to structure elements alongside the introduction of global L2 cache in newer CUDA-enabled GPUs significantly diminishes the impact of strided accesses.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our CuSha framework using the eight graph applications listed in Table 2 and six publicly available [15] real-world graphs listed in Table 1. The graphs cover a broad range of sizes and sparsity and come from different real-world origins. *LiveJournal* and *Pokec* are directed social networks which represent friendship among the users. *HiggsTweet* is a social relationship graph among twitter users involved in tweeting about the discovery of Higgs particle. *RoadNetCA* is the California road network in which the roads are represented by edges and the vertices represent the intersections. *WebGoogle* is a graph released by Google in which vertices represent web pages and the directed edges are hyperlinks

		BFS	SSSP	PR	CC	SSWP	NN	HS	CS
LiveJournal	CuSha-CW	166	346	709	190	531	203	386	855
	CuSha-GS	170	414	885	195	683	197	465	929
	VWC-CSR	280-420	770-1075	2814-3503	264-396	1346-1954	3872-6568	458-647	984-1423
Pokec	CuSha-CW	70	143	255	103	137	3202	246	186
	CuSha-GS	63	138	267	86	134	3278	244	175
	VWC-CSR	125-172	283-357	1539-3246	109-135	310-375	678-827	313-385	190-246
HiggsTwitter	CuSha-CW	59	130	345	72	94	246	150	96
	CuSha-GS	61	127	375	71	89	246	143	89
	VWC-CSR	76-241	175-556	682-2750	67-164	113-325	224-713	112-319	70-171
RoadNetCA	CuSha-CW	286	384	54	435	1026	247	43	2472
	CuSha-GS	432	647	122	897	1905	328	41	2521
	VWC-CSR	655-5727	710-6731	103-521	747-5665	2071-15500	308-1984	76-253	4634-31792
WebGoogle	CuSha-CW	28	41	69	29	74	115	84	98
	CuSha-GS	27	42	73	26	77	125	83	116
	VWC-CSR	100-138	138-208	181-306	63-123	247-373	133-196	148-213	159-197
Amazon0312	CuSha-CW	19	36	44	17	44	40	47	504
	CuSha-GS	24	45	46	18	52	49	55	509
	VWC-CSR	35-53	80-117	87-157	17-55	79-121	48-83	67-117	621-940

Table 4: CuSha-CW, CuSha-GS, and VWC-CSR running times on different algorithms and inputs. Reported times include host-device data transfers and are in milliseconds.

	CuSha-GS over VWC-CSR	CuSha-CW over VWC-CSR
Averages Across Input Graphs		
BFS	1.94x−4.96x	2.09x−6.12x
SSSP	1.91x−4.59x	2.16x−5.96x
PR	2.66x−5.88x	3.08x−7.21x
CC	1.28x−3.32x	1.36x−4.34x
SSWP	1.90x−4.11x	2.19x−5.46x
NN	1.42x−3.07x	1.51x−3.47x
HS	1.42x−3.01x	1.45x−3.02x
CS	1.23x−3.50x	1.27x−3.58x
Averages Across Benchmarks		
LiveJournal	1.66x−2.36x	1.92x−2.72x
Pokec	2.40x−3.63x	2.34x−3.58x
HiggsTwitter	1.14x−3.59x	1.14x−3.61x
RoadNetCA	1.34x−8.64x	1.92x−12.99x
WebGoogle	2.41x−3.71x	2.45x−3.74x
Amazon0312	1.37x−2.40x	1.57x−2.73x

Table 5: Speedup Ranges of CuSha-GS and CuSha-CW over VWC-CSR Configurations.

	CuSha-GS over MTCPU-CSR	CuSha-CW over MTCPU-CSR
Averages Across Input Graphs		
BFS	2.41x−10.41x	2.61x−11.38x
SSSP	2.61x−12.34x	2.99x−14.27x
PR	5.34x−24.45x	6.46x−28.98x
CC	1.66x−7.46x	1.72x−7.74x
SSWP	2.59x−11.74x	3.03x−13.85x
NN	1.82x−19.17x	1.97x−19.59x
HS	1.74x−7.07x	1.80x−7.30x
CS	2.39x−11.06x	2.49x−11.55x
Averages Across Benchmarks		
LiveJournal	4.1x−26.63x	4.74x−29.25x
Pokec	3.26x−15.19x	3.2x−14.89x
HiggsTwitter	1.23x−5.30x	1.23x−5.34x
RoadNetCA	1.95x−9.79x	2.95x−14.29x
WebGoogle	1.95x−9.79x	2.95x−14.29x
Amazon0312	1.65x−6.27x	1.88x−7.20x

Table 6: Speedup Ranges of CuSha-GS and CuSha-CW over MTCPU-CSR Configurations.

connecting those pages. *Amazon0312* is Amazon's product co-purchasing network collected on March 2, 2003. In this graph, vertices are products and an edge between vertices indicates that the two products were frequently co-purchased.

The experiments were performed on a system with Nvidia GeForce GTX780 which has 12 SMX multiprocessors and 3 GB GDDR5 RAM. On the host side, there is an Intel Core i7-3930K Sandy Bridge CPU with 12 cores (hyper-threading enabled) operating at 3.2 GHz clock frequency. PCI Express 3.0 lanes operating at 16x speed transfer data between the host DDR3 RAM (CPU side) and the device RAM (GPU side). The benchmarks were evaluated using CUDA 5.5 on Ubuntu 12.04, Kernel v3.5.0-45. All the programs were compiled with the highest optimization level flag (-O3).

5.1 Performance Analysis

To evaluate the effectiveness of graph processing on CuSha we compare the performance of following techniques:

- **CuSha-GS**: This is our Cusha framework when using G-Shards representation;
- **CuSha-CW**: This is our CuSha framework when using CW representation;
- **VWC-CSR**: This is *virtual warp-centric* [12] technique using CSR representation. We considered virtual warp sizes of 2, 4, 8, 16, and 32; and
- **MTCPU-CSR**: This is the multi-threaded CPU implementation using the CSR representation built using pthreads such that each thread is assigned to a group of vertices that are adjacent to each other in the CSR representation. We considered the runs with 1, 2, 4, 8, 16, 32, 64, and 128 threads on the 12-core host processor with hyper-threading enabled.

Speedups. Table 4 shows raw processing time (including host-device data transfers) of CuSha-CW, CuSha-GS, and VWC-CSR. These times are presented as ranges (min - max) because VWC-CSR is run for several configurations as its performance varies with chosen virtual warp sizes. From this table we can get the speedups of CuSha over VWC-CSR. When averaging speedups across all benchmarks and inputs, CuSha-GS provides speedups in range of 1.72x -

4.05x while CuSha-CW provides speedups in range of 1.89x - 4.89x over VWC-CSR. Table 5 shows the speedup ranges separately for each benchmark when averaged across all inputs and then separately for each input graph when averaged over all benchmarks. We observe that CuSha outperforms VWC-CSR across all benchmarks and all inputs with maximum improvements observed for PageRank (PR) program and RoadNetCA input graph. We also observe that both G-Shards and Concatenated Windows contribute substantially to the resulting speedups. For example, for PageRank, maximum speedup observed using CuSha-GS is 5.88x and this increases to 7.21x when CuSha-CW is used.

Results in Table 6 demonstrate CuSha's substantial performance improvements over MTCPU-CSR. The maximum speedups correspond to the single-threaded CPU implementation while the minimums correspond to use of best number of CPU threads. Best configuration varies from one benchmark and graph combination to another. When averaging speedups across all benchmarks and inputs, CuSha-GS provides speedups in range of 2.57x - 12.96x while CuSha-CW provides speedups in range of 2.88x - 14.33x over MTCPU-CSR. Highest speedups were observed for PageRank program and the largest input graph *LiveJournal*.

Data transfer times between the host and the device have been included in the results reported in Table 5 and Table 6.

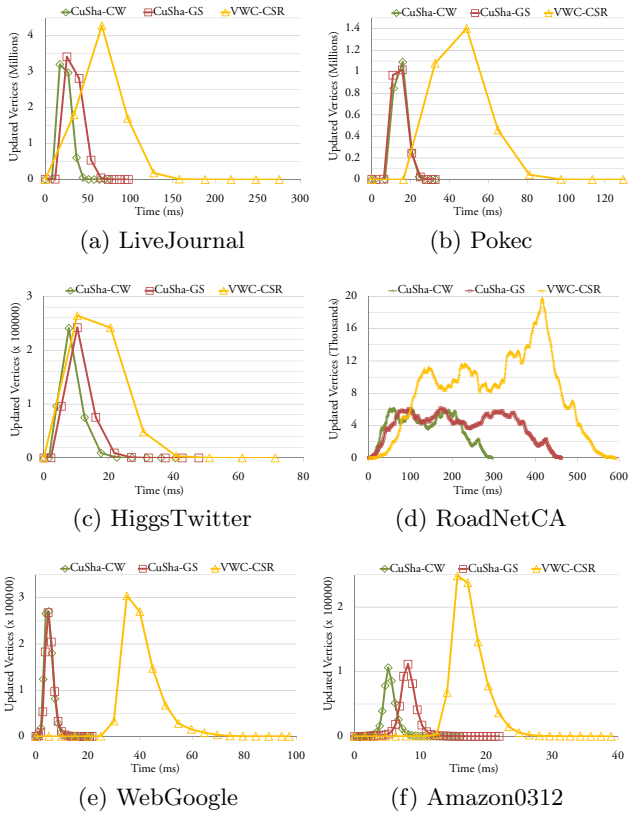
	CuSha-CW	CuSha-GS	Best VWC-CSR
LiveJournal	929.1 M	692.2 M	272.4 M
Pokec	1009.9 M	942.2 M	269.7 M
HiggsTwitter	378.8 M	323.9 M	208.8 M
RoadNetCA	19.9 M	13.0 M	8.5 M
WebGoogle	242.7 M	243.1 M	52.5 M
Amazon0312	208.8 M	149.4 M	89.8 M

Table 7: Traversed Edges Per Second (TEPS) for BFS with CuSha-CW, CuSha-GS, and VWC-CSR using the best configuration for each graph.

TEPS data for BFS traversal. Table 7 shows number of Traversed Edges Per Second (TEPS) in BFS for CuSha-CW, CuSha-GS, and VWC-CSR with the best performance handpicked by running it with different virtual warp sizes. As the table shows, CuSha can be up to 5 times better than the best VWC-CSR. CuSha provides performance gain over VWC-CSR by eliminating non-coalesced accesses and thread divergence, which will be further explored in this section.

Figure 7 shows the number of vertices updated during BFS traversal iteration by iteration over time for CuSha-CW and CuSha-GS, and for VWC-CSR with the warp size exhibiting best performance. Processing with CuSha-CW and CuSha-GS usually includes more iterations than VWC-CSR because G-Shards and CW contain more than one version of vertex values; unlike CSR that only stores one version of it. On the other hand, iterations take much less time with G-Shards and CW because of the GPU-friendly representation. Faster iterations in G-Shards and CW result in much quicker convergence of BFS in all the graphs.

Global memory and warp execution efficiency. The speedups of CuSha over VWC-CSR observed can be explained by studying the improvements in global memory accesses and warp execution efficiencies. As we had shown earlier, these efficiencies are quite low for VWC-CSR. Figure 8 compares the average global memory store efficiency,



(a) LiveJournal (b) Pokec

(c) HiggsTwitter (d) RoadNetCA

(e) WebGoogle (f) Amazon0312

Figure 7: BFS traversal for CuSha-CW and CuSha-GS, and for VWC-CSR with the best handpicked virtual warp size. Each point stands for an iteration.

the average global memory load efficiency, and the average warp execution efficiency of VWC-CSR with best configuration, CuSha-GS, and CuSha-CW while processing *LiveJournal* graph.

The global memory store efficiency is the ratio of the global memory store throughput achieved by the program to the global memory store throughput that is actually needed by the program. It indicates how well the threads within a kernel write to the global memory: a high value shows that more store operations are fulfilled with coalesced writes. The average of this value across all kernel iterations during a run is the average global memory store efficiency. As we can see, VWC-CSR has a very low average global memory store efficiency (1.93% on average) because when the warp needs to update the vertex content, only one thread inside the virtual warp is active and writing to memory. On the other hand, CuSha-GS and CuSha-CW have a much higher average global memory store (27.64% for G-Shards and 25.06% for CW) because updates to vertex contents are done in parallel by multiple threads.

The global memory load efficiency indicates the ratio of achieved global memory load throughput to required load throughput. Compared to CuSha, VWC-CSR achieves lower global memory load efficiency (28.18% on average) mainly because of non-coalesced accesses. CuSha-GS and CuSha-CW achieve 80.15% and 77.59% global memory load efficiency on average, respectively. For both CuSha-GS and CuSha-CW, the average global memory load is higher than store mainly because of the heavy, but coalesced, memory reads of shard entries.

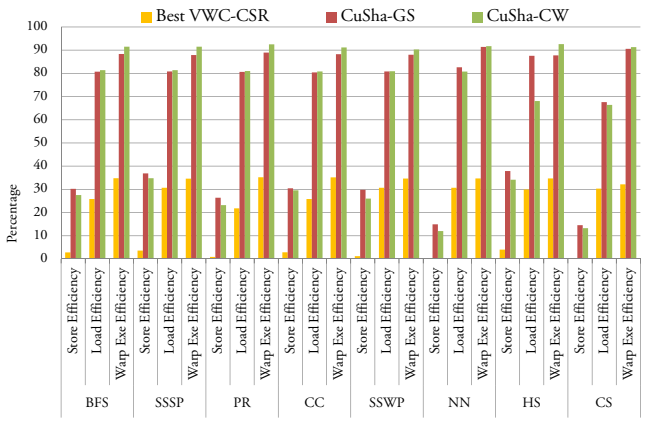


Figure 8: Average profiled efficiencies of CuSha-GS and CuSha-CW vs. best VWC-CSR configuration on *LiveJournal* graph.

Finally, the warp execution efficiency is defined as the ratio of the average active threads in a warp to the maximum possible active threads in a warp per multiprocessor. It indicates how well GPU hardware resources are utilized. Figure 8 shows that the VWC-CSR has a much lower warp execution efficiency (34.48% on average) compared to CuSha (88.90% for G-Shards and 91.57% for CW on average) mainly due to the impact of different number of neighbors in VWC-CSR. Since CuSha organizes graph edges in large shards, this effect is heavily reduced.

Memory occupied by different graph representations. Next we evaluate the cost of using G-Shards and CW representations in terms of increased memory requirement and copying time over CSR.

Figure 9 shows minimum, average, and maximum space consumed by CSR, G-Shards, and CW representations for each input graph, across all benchmarks and normalized with respect to the CSR average for each benchmark. G-Shards and CW take 2.09x and 2.58x more space, on average, than CSR. G-Shards representation adds an overhead of about $(|E|-|V|)\times size\_of(Vertex)+|E|\times size\_of(index)$ bytes over CSR. For CW, this overhead increases by $|E|\times size\_of(index)$ bytes. Even though the overhead is input dependent, technological advancements allow us to leverage reasonably large RAM on the GPU which can easily fit most real world graphs. If graphs do not fit in the GPU RAM, a multi-streamed procedure should be incorporated to overlap computation and data transfer.

Figure 10 breaks the total time, taken by all the benchmarks on *LiveJournal* input, down into the time taken by: 1) H2D copy - time to copy graph from CPU side memory to GPU global memory; 2) GPU Computation - time to process the graph on GPU; and 3) D2H copy - time to copy the results back from GPU global memory to CPU side memory. We can see that CuSha takes more H2D copy time compared to VWC-CSR mainly because of the space overheads involved in using G-Shards and CW. However, computation friendly representations of G-Shards and CW allow faster processing, which in turn significantly improves the overall performance. D2H copy only involves the final vertex values and hence, is negligible. Also, CuSha-CW takes more time to copy compared to CuSha-GS because of the additional mapper array.

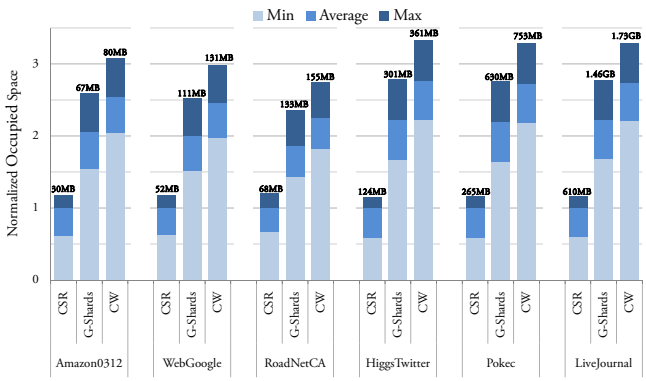


Figure 9: Memory occupied by each graph using CSR, G-Shards, and CW representations over all benchmarks – values are normalized with respect to CSR average. Numbers in the figure are maximums in each case.

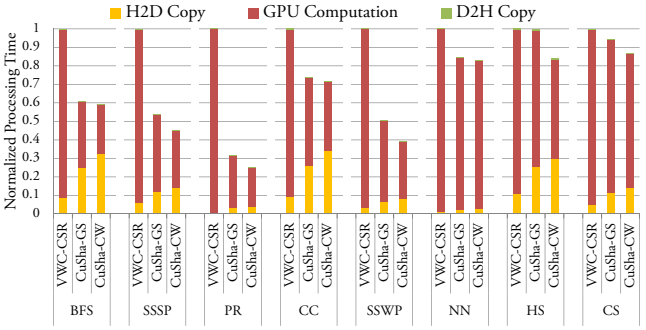


Figure 10: Time breakdown: device-to-host copy time, GPU execution time, and host-to-device copy back time on LiveJournal. VWC-CSR has the best configuration.

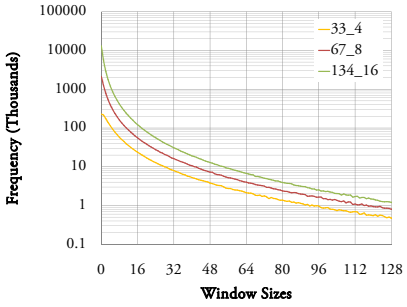
5.2 Sensitivity Analysis of CW

In this section we study the sensitivity of CuSha-CW across different input graph characteristics. To create the graphs used in this study we use the SNAP graph library [2] and the RMAT [5] model that generates scale free graphs which resemble the characteristics of real-world graphs such as power-law graphs.

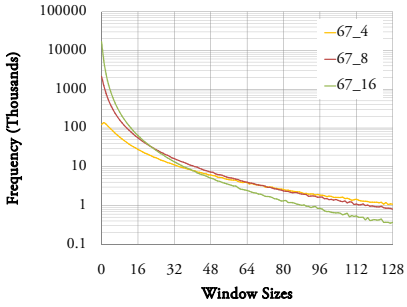
Figure 12 shows the total running time normalized with respect to the shortest time for SSSP on CuSha with nine synthetically-created RMAT graphs across range of different *sizes* and *sparsities*. It confirms the sensitivity of G-Shards representation to the graph size, the graph sparsity, and the number of vertices assigned to a shard ($|N|$). We discuss each of these parameters with the help of Figure 11 which shows the distribution of window sizes in different scenarios.

***Graph size*:** Increasing the number of edges and vertices in the graph causes the frequency of small windows to increase (see Figure 11(a)). This makes the G-Shards representation more vulnerable to graph size compared to CW. This can be seen in Figure 12: processing time with G-Shards more than doubles from 67_8 graph with $|N| = 3k$ to 134_16 graph with $|N| = 3k$ while with CW it increases by 1.6x.

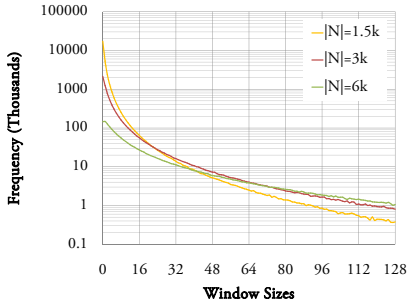
***Sparsity*:** Figure 11(b) shows that by increasing graph sparsity, the number of windows with size close to zero increases.



(a) Observing graph size effect with graph degree equal to 8 and $|N| = 3k$.



(b) Observing graph degree (sparsity) effect having number of edges set to 67 million and $|N| = 3k$.



(c) Observing the effect of changing $|N|$ using 67_8 graph.

Figure 11: Frequency of window sizes (from 0 to 128) having different RMAT input graphs. $|N|$ is the number of vertices assigned to a shard. A i_j graph has around i million edges and j million vertices.

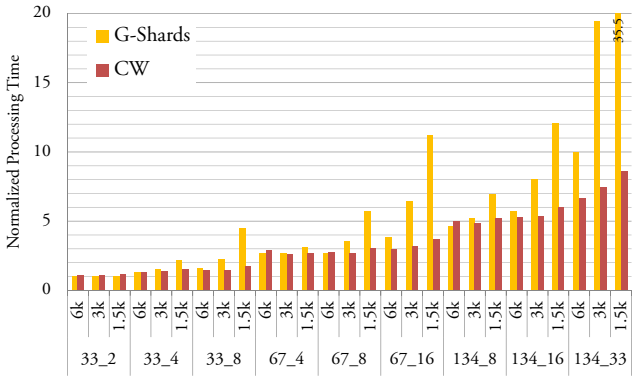


Figure 12: Normalized CuSha running time configured to use G-Shards and CW against RMAT graphs having different shard sizes in SSSP benchmark. In the figure, a i_j graph has around i million edges and j million vertices. Numbers close to x axis are number of vertices assigned to a shard.

G-Shards representation is more sensitive to small sized windows compared to CW, i.e., as graphs become sparser, the performance of G-Shards degrades rapidly compared to CW. In Figure 12, the G-Shards processing time doubles from 67_4 graph with $|N| = 3k$ to 67_16 graph with $|N| = 3k$ while CW processing time increases only slightly.

Vertices assigned to shard **($|N|$):** Larger $|N|$ reduces the number of shards and increases their size. As a result, larger $|N|$ increases the size of windows as shown in Figure 11(c). Hence, while processing very sparse graphs using G-Shards, it is crucial to have a large value for $|N|$. Comparatively, CW is not heavily impacted when $|N|$ is small as shown with the biggest and sparsest graphs in Figure 12.

We further compared CW sensitivity with VWC-CSR's. Figure 13 presents the speedup of CW over VWC-CSR on RMAT graphs. It shows that with increasing size and sparsity of the RMAT graph, CW's superiority over VWC-CSR increases. It also reveals the performance change of VWC-CSR method when different warp sizes are employed. It is evident from the figure that different graphs with different characteristics require different configuration of VWC-CSR for best performance.

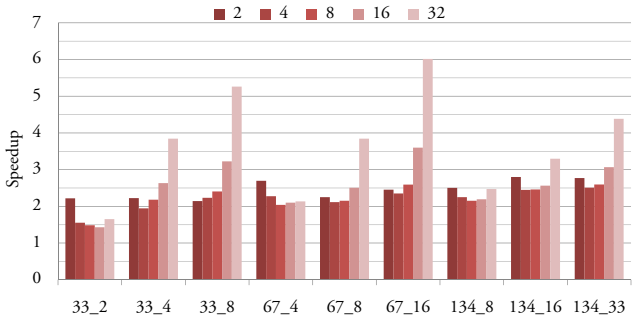


Figure 13: CW speedups over VWC-CSR with virtual warp sizes 2, 4, 8, 16, and 32 against RMAT graphs in SSSP benchmark. CW has $|N| = 3k$. In figure above, a i_j graph has around i million edges and j million vertices.

6. RELATED WORK

Using GPUs for high performance graph processing was first introduced in [10]. Since then CSR has been the most popular representation to store graphs on GPU. Even though efforts have been spent to minimize path divergence as in [12] and minimize load imbalance as in [21], the CSR representation inherently suffers from poor locality [18].

Apart from having virtual warps, [12] offers *deferring outliers* and *dynamic workload distribution* techniques in order to reduce intra-warp divergence and achieve a balanced load for different warps. However, the improvements achieved by these two methods are limited because of the heavyweight atomic operations on global memory. The technique presented in [28] tries to balance the load in graphs represented in CSR by reorganizing the vertices and putting them in three bins. Based on the size of these bins, appropriate number of GPU threads are assigned to process these bins, hence providing a balanced workload distribution.

In [9], TOTEM abstracts away development complexity and reduces communication overhead for processing graphs by message aggregation in a heterogeneous many-core system. Another hybrid CPU-GPU method that improves the efficiency of BFS is presented in [13]. During the initial phases when there are fewer number of vertices to be processed, the CPU performs the computation. Later, when the number of vertices to be processed becomes larger, the computation is moved onto the GPU. Authors also propose

a read-queue hybrid technique that switches the processing scheme based on the size of next level in BFS. [1], [21] and [19] are various multi-core CPU and GPU works that employ queues to handle vertices that should be explored in the next level. The technique presented in [21] efficiently computes the prefix sum for scatter offset in parallel to produce global computation frontier queues. Even though using frontiers is still susceptible to input-dependent non-coalesced memory accesses, the GPU underutilization is eliminated by this technique. CuSha is a framework that supports a broader range of graph algorithms compared to such problem-specific queue-based solutions for BFS.

Medusa [30] is a generalized GPU-based graph processing framework that focuses on abstractions for easy programming and scaling to multiple GPUs. CuSha primarily focuses on exploring new graph representations to allow faster graph processing. Apart from CSR, various other graph representations have been proposed that are typically beneficial for targeted applications. For instance, [20] introduces a novel idea of using sparse bit vectors, a structure similar to linked list. However, this representation is highly space inefficient and is only beneficial for morph algorithms when data access patterns exhibit spatial locality.

Dymaxion [6] is an API to improve memory access patterns on GPUs. It uses two fundamental techniques to leverage high memory coalescing:

- **Data restructuring:** Although this method is effective and quite common [24], its use in Dymaxion is limited to predictable data patterns, such as transformation of two-dimensional matrices from row-major order to column-major order or vice versa.
- **Memory remapping:** Allows efficient accessing of data elements via an intermediate mapping function. It is similar to CuSha's CW method.

In [29], authors present *data reordering* and *job swapping* techniques to remove GPU memory access irregularities. *Data reordering*, similar to *data restructuring*, repositions elements of an array to minimize required global memory transactions. In *job swapping*, threads exchange work in order to achieve more coalesced memory accesses. It is usually done using reference redirection, which is similar to *memory remapping*. Despite their benefits for applications with regular chunkable input data, irregular and unpredictable dependency between real-world graph elements makes it costly to employ these techniques for graph applications.

Recently, Wu et al. classified and analyzed few fundamental methods to minimize non-coalesced memory accesses in [27]. CuSha employs three of these techniques:

- **Duplication:** Vertices and destination indices are duplicated for edges.
- **Reordering:** Edges within the shards are sorted based on their source indices.
- **Sharing:** Irregular memory accesses are confined to fast shared memory in the GPU.

7. CONCLUSION

In this paper, we first recognized the use of shards for efficient graph processing on GPUs through coalesced memory accesses and introduced G-Shards: a graph representation that effectively maps shards to various GPU subcomponents. We also proposed a novel representation named Concatenated Windows to eliminate GPU underutilization for very large and sparse graphs. Finally, we built CuSha, a framework to enable users to easily define vertex-centric algorithms for processing large graphs on GPU. CuSha internally relies on both G-Shards and Concatenated Windows and exposes necessary functions to be provided by the users. CuSha achieves substantial speedups over the fine-tuned state-of-the-art virtual warp-centric method. We believe that increasing amount of shared memory per SM along with performance enhancements of shared memory atomic operations in upcoming CUDA devices will further enhance the superiority of our two newly introduced representations.

8. ACKNOWLEDGMENTS

We thank the reviewers for their detailed feedback for improving the paper.

This work is supported by National Science Foundation grants CCF-1157377 and CCF-0905509 to the University of California Riverside.

9. REFERENCES

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.

[2] D. A. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS*, pages 1–12, 2008.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, pages 163–174, 2009.

[4] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on cuda. In *IPDPS*, pages 544–555, 2011.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[6] S. Che, J. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *SC*, pages 1–11, 2011.

[7] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The anatomy of a scientific rumor. *Scientific Reports*, 2013.

[8] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*, pages 345–354, 2012.

[9] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems. In *CoRR*, 2013.

[10] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[11] M. Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2, 2007.

[12] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *PPoPP*, pages 267–276, 2011.

[13] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *PACT*, 2011.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[15] J. Leskovec. Stanford large network dataset collection. *http://snap.stanford.edu/data/index.html*, 2011.

[16] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Trans. Web*, 1(1), May 2007.

[17] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[18] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[19] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *DAC*, pages 52–55, 2010.

[20] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. In *PPoPP*, pages 107–116, 2012.

[21] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *PPoPP*, pages 117–128, 2012.

[22] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on gpus. In *PPoPP*, pages 147–156, 2013.

[23] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[24] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *PLDI*, pages 13–22, 2012.

[25] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, 2012.

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

[27] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *PPoPP*, pages 57–68, 2013.

[28] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient pagerank and spmv computation on amd gpus. In *ICPP*, pages 81–89, 2010.

[29] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS*, pages 369–380, 2011.

[30] J. Zhong, and B. He. Medusa: Simplified Graph Processing on GPUs. In *IEEE Transactions on Parallel and Distributed Systems*, 2013.

APPENDIX

A. VIRTUAL WARP-CENTRIC METHOD

Figure 14 shows the pseudo-code for Virtual Warp-Centric (VWC) method. Similar to CuSha, it performs each iteration of the vertex-centric algorithm in a single GPU kernel call. During the kernel, the GPU assigns one virtual warp

```
0.  is_converged = false;
1.  while (!is_converged) {
2.    is_converged = true;
3.    parallel-for virtual warp VW{
4.      allVWs = blockDim / VW.size;
5.      shared Vertex old_V[allVWs];
6.      shared Vertex local_V[allVWs];
7.      shared Vertex outcome[blockDim];
8.      shared unsigned int edges_start[allVWs];
9.      shared unsigned int nbrs_size[allVWs];
10.     if ( virtual_lane_ID == 0 ) {
11.       edges_start[VW.ID] = InEdgeIdxs[VW.ID];
12.       nbrs_size[VW.ID] = InEdgeIdxs[VW.ID+1]
            - edges_start[VW.ID];
13.       old_V[offset] = VertexValues[VW.ID];
14.       InitCompute( local_V+offset,
            old_V+offset );
15.     }
16.     parallel-for Nbr in neighbors of vertex
          with index VW.ID{
17.       edge_index = Nbr+edges_start[VW.ID];
18.       Nbr_index = SrcIndex[edge_index];
19.       Compute( VertexValues+Nbr_index,
            VertexValuesStatic+Nbr_index,
            EdgeValues+edge_index, outcome+Nbr );
20.       ParallelReduction(outcome,nbrs_size[VW.ID],
            local_V+offset,Nbr);
21.     }
22.     if ( virtual_lane_ID == 0 &&
          UpdateCondition ( local_V+offset,
          old_V+offset ) ) {
23.       VertexValues[VW.ID]=local_V[offset];
24.       is_converged = false;
25.     }
26.   }
27.   barrier;
28. }
```

Figure 14: Graph processing procedure in virtual warp-centric method using CSR representation.

to process a single vertex (line 3). One virtual lane within the virtual warp retrieves the starting address for the incoming edges array and the number of neighbors and then, calls the *InitCompute* function (lines 10-15). Next, the threads within the virtual warp are assigned to process the neighbors (line 16) using the *Compute* function (line 19). Parallel reduction [11] computes the final value of the vertex assigned to the virtual warp (line 20). Finally, one virtual lane in the virtual warp performs the *UpdateCondition* function and updates the source vertex, if necessary (lines 22-25).

Our implementation of VWC requires only one thread inside the virtual warp to perform the Single Instruction Single Data (SISD) phases as opposed to [12] in which all the threads execute these phases. Therefore, it avoids possible bank conflicts and serialization of write instructions in the shared and global memory respectively.