Scalable SIMD-Efficient Graph Processing on GPUs

Farzad Khorasani Rajiv Gupta Laxmi N. Bhuyan
Computer Science and Engineering Department
University of California Riverside, CA, USA
{fkhor001, gupta, bhuyan}@cs.ucr.edu

***Abstract*—The vast computing power of GPUs makes them an attractive platform for accelerating large scale data parallel computations such as popular graph processing applications. However, the inherent irregularity and large sizes of real-world power law graphs makes effective use of GPUs a major challenge. In this paper we develop techniques that greatly enhance the performance and scalability of vertex-centric graph processing on GPUs. First, we present *Warp Segmentation*, a novel method that greatly enhances GPU device utilization by dynamically assigning appropriate number of SIMD threads to process a vertex with irregular-sized neighbors while employing compact CSR representation to maximize the graph size that can be kept inside the GPU global memory. Prior works can either maximize graph sizes (VWC [11] uses the CSR representation) or device utilization (e.g., CuSha [13] uses the CW representation; however, CW is roughly 2.5x the size of CSR). Second, we further scale graph processing to make use of multiple GPUs while proposing *Vertex Refinement* to address the challenge of judiciously using the limited bandwidth available for transferring data between GPUs via the PCIe bus. Vertex refinement employs parallel binary prefix sum to dynamically collect only the updated boundary vertices inside GPUs' outbox buffers for dramatically reducing inter-GPU data transfer volume. Whereas existing multi-GPU techniques (Medusa [31], TOTEM [7]) perform high degree of wasteful vertex transfers. On a single GPU, our framework delivers average speedups of 1.29x to 2.80x over VWC. When scaled to multiple GPUs, our framework achieves up to 2.71x performance improvement compared to inter-GPU vertex communication schemes used by other multi-GPU techniques (i.e., Medusa, TOTEM).**

***Keywords*-Graphs; Power Law Graphs; GPU; Irregular Computations; Scalability; Multi-GPU;**

I. Introduction

Due to their ability to represent relationships between entities, graphs have become the building blocks of many high performance data analysis algorithms. A wide variety of graph algorithms can be expressed in an iterative form – during each iteration vertices update their state based upon states of neighbors connected by edges using a computation procedure until the graph state becomes stable. The inherent data parallelism in an iterative graph algorithm makes many-core processors with underlying SIMD hardware such as GPUs an attractive platform for accelerating the algorithms. However, efficient mapping of real-world power law graphs with irregularities to symmetric GPU architecture is a challenging task [19].

In this paper we present techniques that maximize the scalability and performance of vertex-centric graph processing on multi-GPU systems by fully exploiting the available resources as follows:

SIMD hardware – The irregular nature of power law graphs makes it difficult to balance load across threads leading to underutilization of SIMD resources. We address the device underutilization problem of a GPU by developing *Warp Segmentation* that dynamically assigns appropriate number of SIMD threads to process a vertex with irregular-sized neighbors. Our experiments show that the warp execution efficiency of warp segmentation exceeds 70% while for the well known VWC [11] technique it is around 40%.

GPU global memory – For scaling performance to large graphs, they must be held in the global memory of the GPUs. To maximize the graph sizes that can be held in global memories, a compact graph representation must be used. Therefore *Warp Segmentation* makes use of the compact CSR representation. To tolerate the long latency of non-coalesced memory accesses that arise while accessing the neighbors of a vertex in CSR, warp segmentation keeps the GPU cores busy by scheduling other useful operations that compute the segment size and lane's intra-segment index.

Inter-GPU communication bandwidth – Since large graphs must be distributed across the global memories of multiple GPUs, processing at each GPU requires values of neighboring vertices that reside on other GPUs. Here we must make judicious use of the limited bandwidth available for transferring data between GPUs via the PCIe bus. We introduce an approach based upon parallel binary prefix sum that dynamically limits the inter-GPU transfers to only include updated vertices. Existing multi-GPU techniques perform high degree of wasteful vertex value transfers.

Our solution maximizes the graph sizes for which high performance can be achieved by fully utilizing GPU resources of SIMD hardware, memory, and bandwidth.

Let us briefly consider the related works and see how our approach overcomes their drawbacks. First, we consider the prominent single GPU techniques for vertex-centric graph processing, namely VWC [11] and CuSha [13]. Virtual-Warp Centric (VWC) [11] is the state-of-the-art method that uses the compact CSR representation and is inevitably prone to load imbalance when processing real-world graphs due to high variation in degrees of vertices. When the size of the virtual warp is less than the number of neighbors for a vertex, the virtual warp needs to iterate over the neighbors forcing other virtual warps within the warp that are assigned to vertices with fewer neighbors to stay inactive. When the size of the virtual warp is greater than the the size of the neighbors for a vertex, a great portion of the virtual warp is disabled. Both cases lead to underutilization of SIMD resources and poor warp execution efficiency. In addition, discovering the best virtual warp size for every graph and every expressed graph algorithm requires multiple tries. CuSha [13] addresses the drawbacks of VWC, namely warp execution inefficiencies and non-coalesced accesses, but at the cost of using G-Shards and CW graph representations which are 2x-2.5x larger than the CSR representation due to vertex replication. *In contrast, Warp Segmentation uses the compact CSR representation while delivering high SIMD hardware utilization.* In warp segmentation the neighbors of warp-assigned vertices are grouped into segments. Warp lanes then get assigned to these neighbors and recognize their position inside the segment and the segment size by first performing a quick binary search on the fast shared memory content and then comparing their edge index with corresponding neighbor indices. When the segment size and the position in the segment are known for the lanes, user-defined reduction can be efficiently performed between neighbors of a vertex without introducing any intra-warp load imbalance. When processing on a single device, our framework employs asynchronous parallelism paradigm that allows simultaneous program execution and data transfer in an iteration. It also permits visibility of the updated neighbor vertex content, enabling a faster convergence.

Next let us consider the related works on multi-GPU graph processing [31] [7]. Given a partitioning of a graph across multiple GPUs, these techniques underestimate the importance of efficient inter-device communication and do not effectively utilize the PCIe bandwidth. This is a significant problem because the PCIe bus, as the path to communicate data from one GPU to other GPUs, is tens of times slower than GPU global memory. Previous multi-GPU techniques either copy the whole vertex set belonging to one GPU to other GPUs at every iteration [31], or they identify boundary vertices in a pre-processing stage and make GPUs exchange these subsets of vertices in every iteration [7] [8]. In both approaches, a great number of vertices that are exchanged between devices is redundant. *In contrast, we propose Vertex Refinement, a new strategy that enables our framework to efficiently scale to multiple GPUs.* Vertex Refinement refines and transfers only those vertices that are updated in the previous round and are needed by other devices. It consists of two stages: *online* and *offline*. In the offline stage, boundary vertices are recognized and marked during pre-processing. In the online stage, we exploit parallel binary prefix sum to refine updated vertices from not-updated ones on-the-fly. A vertex is transferred to another device only if it is marked and refined by the online stage. Thus, Vertex Refinement eliminates the communication overhead and provides higher multi-GPU performance.

The key contributions of this work are:

- We introduce *Warp Segmentation* (WS), a novel technique for compact graph representations that overcomes SIMD underutilization during graph processing. On average, WS outperforms VWC by 1.29x-2.80x.
- We introduce *Vertex Refinement* that enables effective scaling of the graph processing procedure to multiple GPUs. It efficiently filters updated vertices of a GPU on-the-fly via parallel binary prefix sum and provides exclusive speedup of up to 2.71x over other multi-GPU vertex communication schemes.
- We implemented a framework that embeds above items and enables users to easily express desired iterative graph algorithm and execute it on one or multiple CUDA-enabled GPUs.

The remainder of the paper is organized as follows. In Section II, we present *Warp Segmentation* and in Section III we introduce the framework interface. Then in Section IV, we describe efficient scaling of our framework to multiple GPUs via Vertex Refinement. In Section V, we present the experimental evaluation. Sections VI and VII give related work and conclusion respectively.

II. Warp Segmentation for SIMD-Efficiency

Here we present *Warp Segmentation* (WS) that eliminates intra-warp load imbalance and enhances execution efficiency for processing a graph in CSR form. CSR is a compact form suitable for representing large and sparse graphs in a minimum space. Due to its space-efficiency, CSR is a good choice to hold large graphs inside the limited GPU memory.

As Figure 1 shows, CSR consists of 4 arrays:

- *VertexValues* holds the content of the i^{th} vertex in its i^{th} element.
- *NbrVertexIndices* holds the indices for a vertex's neighbors in a contiguous fashion.
- *NbrIndices* holds a prefix sum of the number of neighbors for vertices. The i^{th} vertex's neighbors inside *NbrVertexIndices* start at $NbrIndices[i]$ and end before $NbrIndices[i+1]$.
- *EdgeValues* holds the edge values corresponding to the neighbors inside *NbrVertexIndices*.

To motivate the need for WS, we first describe the drawbacks of the Virtual-Warp Centric (VWC) [11] method that also uses the CSR representation.

Drawbacks of VWC: VWC divides the SIMD group (warp, in CUDA terms) with the physical length of 32 into smaller virtual warps with fixed lengths (2, 4, 8, 16, or 32). Virtual warp size is kept the same throughout the graph processing. Each virtual warp is assigned to process one vertex and its incoming edges. As an enhancement

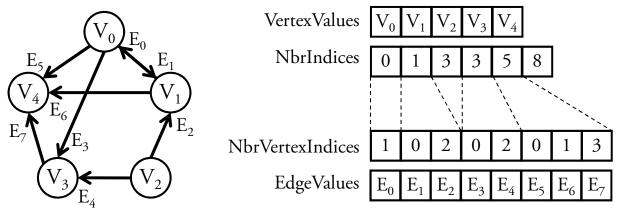


Figure 1. A graph with 5 vertices and 8 edges and its CSR representation.

of the original work [11], Khorasani et al. proposed a generalized form of VWC in [13] in which threads of the virtual warp are involved in reduction over the computed values. However, real-world graphs often exhibit power-law degree distribution, i.e. the number of neighbors a vertex owns vary greatly from one vertex to another. Thus, due to fixed number of virtual lanes involved in a reduction, VWC unavoidably suffers from underutilization:

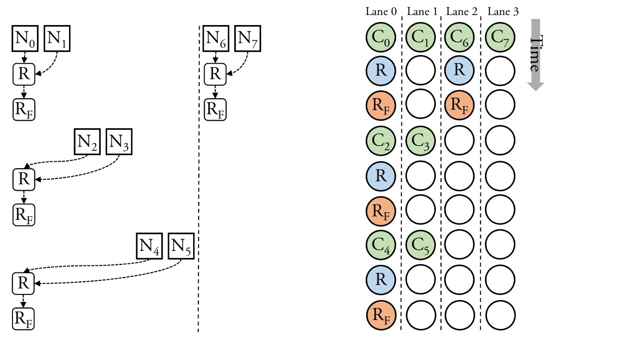
– If the virtual warp is *smaller* than the vertex's number of neighbors, it will have to iterate over the vertex's connected edges hence dragging with it other virtual warps that have already finished their jobs (see the example in Figure 2(a)); and

– If the virtual warp has a size that is *larger* than the number of neighbors for a vertex, a portion of virtual warp's lanes stays *idle* during the reduction leading to underutilization (see the example in Figure 2(b)).

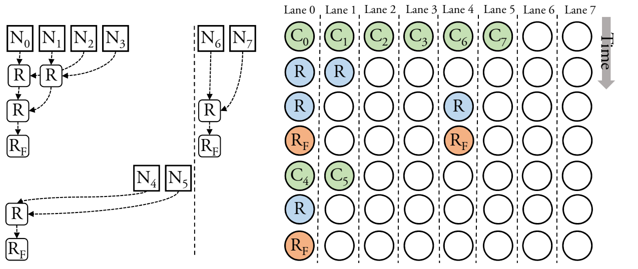
This motivates the need for a technique that, independent of inner graph structure, takes minimum number of reduction steps in a SIMD environment, i.e. Warp Segmentation. Note that VWC suffers from the SIMD load imbalance in the same way PRAM-style thread assignment [9] does. In both PRAM-style and VWC, assigning fixed number of SIMD threads to process one vertex and its edges leads to thread-idling due to highly irregular vertex degree distribution. This fixed number in the former is exactly one while in the latter it can be a power of 2.

Boosting SIMD Utilization via Warp Segmentation: To remedy the drawbacks of fixed-sized virtual warps, we propose *Warp Segmentation* (WS) technique. In WS, a warp is assigned to a group of 32 consecutive vertices and their connected edges. When warp lanes process edges iteratively, those that process edges belonging to one vertex—i.e. having the same destination index—form a *segment*. By knowing the segment size and the index inside the segment, lanes can participate in the appropriate reduction of segment, minimizing the total number of reduction steps.

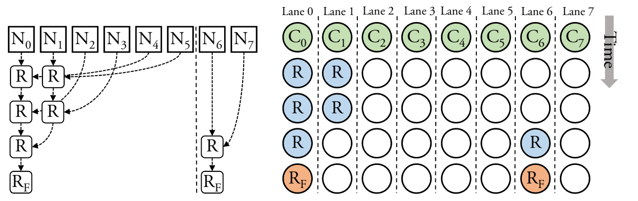
Figure 2(c) shows the reduction in WS in an example scenario. In this example, first six lanes belong to one segment and two last lanes belong to another. The minimum number of reduction steps in this case is $\lceil \log_2 6 \rceil = 3$ which is also the case in WS. As Figure 2 shows, on-the-fly efficient reduction procedure in WS leads to better utilization of SIMD resources compared to VWC. In addition, WS does not need any pre-processing or trial-and-error for the best configuration determination.



(a) VWC with Virtual Warp size 2.



(b) VWC with Virtual Warp size 4.

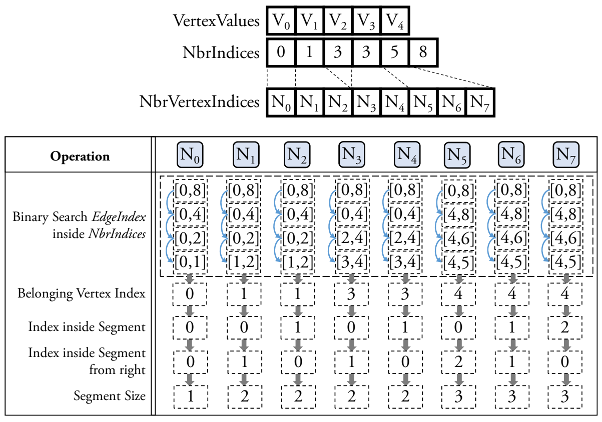


(c) Warp Segmentation.

Figure 2. Reduction in WS and VWC with assumed warp size of 8 and first 6 neighbors belonging to one vertex and last 2 belonging to another. R denotes reduction between vertex's connected neighbors and R_F refers to reduction with the vertex value in shared memory.

The key feature of WS is its fast determination of the segment a lane belongs to and the index of the lane within the segment. The step-by-step approach shown in Figure 3 illustrates this. Warp lanes perform a binary search over *NbrIndices* elements for their assigned edge index. Since *NbrIndices* elements are already fetched to the fast shared memory of the GPU, the binary search is performed quickly. After $\log_2(warpSize)$ steps, the starting position of the resulting search boundary shows the vertex index to which the edge belongs. Knowing the vertex index, the lane's index inside the segment and the segment size is retrieved using *NbrIndices* array. The distance of the holding edge index from the vertex's corresponding *NbrIndices* element reveals the position of the vertex in the segment. The difference between the holding edge index and the next vertex's corresponding *NbrIndices* element, minus one, yields the distance of the lane from the end of the segment. Addition of these two distances plus one represents the segment size.

WS is based upon the vertex-centric paradigm where in every iteration the shared memory serves as a scratchpad for vertices. The shared memory regions corresponding



Operation	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
Binary Search *EdgeIndex* inside *NbrIndices*	[0,8] [0,4] [0,2] [0,1]	[0,8] [0,4] [0,2] [1,2]	[0,8] [0,4] [0,2] [1,2]	[0,8] [0,4] [2,4] [3,4]	[0,8] [0,4] [2,4] [3,4]	[0,8] [4,8] [4,6] [4,5]	[0,8] [4,8] [4,6] [4,5]	[0,8] [4,8] [4,6] [4,5]
Belonging Vertex Index	0	1	1	3	3	4	4	4
Index inside Segment	0	0	1	0	1	0	1	2
Index inside Segment from right	0	1	0	1	0	2	1	0
Segment Size	1	2	2	2	2	3	3	3

Figure 3. Discovering segment size and the index within segment by warp lanes for the graph in Figure 1. Warp size is assumed 8.

to vertices are: initialized by the vertex content within the global memory, modified depending upon the edges connected to the vertex using appropriate reductions, and at the end of the iteration, the updated values are pushed back to the global memory. Two alternatives for the intra-warp reduction in WS are possible. The first one is to use atomics to survive the concurrent modifications of the vertices as in [13]. However, this alternative imposes heavy use of atomics on shared memory locations on top of CSR's inherent non-coalesced neighbor accesses. The second alternative is processing a group of vertices by one thread block instead of one warp. However, this approach necessitates multiple synchronization primitive across the thread block that degrade the performance. WS assigns a set of vertices to GPU's architectural SIMD grouping (warp) and performs efficient reductions hence *it avoids shared memory atomic operations alongside any explicit synchronizations throughout the kernel.*

The reduction in WS can be viewed as a form of intra-warp segmented reduction but without a *head flags* array, consisting of two main steps. First, warp lanes identify the vertex index via a fast binary search. Second, they discover the intra-segment index and the segment size. Also, note that these two sets of operations are independent from the neighbor vertex value hence can be used to cover the latency of the non-coalesced access. The thread exploits instruction level parallelism by simultaneously executing non-dependent instructions. Thus, GPU cores are kept busy performing operations while neighbor's vertex value is on its way.

III. Graph Processing Framework for WS

Next we describe the framework that uses the graph processing procedure based on WS. Then, we present the interface functions that allow easy expression of graph algorithms by non-expert users.

A. Core Processing Procedure

Figure 4 shows the graph processing procedure. The convergence of iterative graph processing is controlled via a variable passed between the host and the device. If no thread updates this variable, it means the algorithm has converged and no more iterations are needed. In the outer-most `for` loop, according to the WS paradigm, each warp is assigned to process a contiguous set of vertices with the size equivalent to the warp size (32 for current CUDA devices). A warp task during one iteration is to process its assigned vertices. This task consists of three major steps.

```
0.  converged = false;
1.  while( !converged ) {
2.    converged = true;
3.    parallel-for warp w {
5.      __shared__ Vertex V[ blockDim ];
6.      __shared__ Vertex tLocal_V[ blockDim ];
7.      __shared__ uint NIdx[ blockDim ];
8.      w_V = V + warpOffsetInCTA;
9.      w_tLocal_V = tLocal_V + warpOffsetInCTA;
10.     w_NIdx = NIdx + warpOffsetInCTA;
        /* 1st major step */
11.     initVertex( w_V + laneID,
         VertexValues + globalTID );
12.     w_NIdx[laneID] = NbrIndices[ globalTID ];
13.     startEIdx = w_NIdx[ 0 ];
14.     endEIdx = NbrIndices[warpGlobalOffset+32];
        /* 2nd major step */
15.     for( currEIdx = startEIdx + laneID;
         currEIdx < endEIdx;
         currEIdx += 32 ) {
16.       nbrIdx = NbrVertexIndices[ currEIdx ];
17.       srcV = VertexValue[ nbrIdx ];
18.       belongingVIdx =
           binarySearch( currEIdx, w_NIdx );
19.       inSegID = min( laneID,
           currEIdx - w_NIdx[ belongingVIdx ] );
20.       SegSize= inSegID + 1 + min( 31 - laneID,
           ( ( belongingVIdx == 31 ) ?
           endEIdx : w_NIdx[ belongingVIdx + 1 ] )
           - currEIdx - 1 );
21.       ComputeNbr( srcV, EdgeValues + currEIdx,
           w_tLocal_V + laneID );
22.       reduceInsideSegment( w_tLocal_V + laneID,
           inSegID, SegSize );
23.       if( inSegID == 0 )
24.         ReduceVertices( w_V + belongingVIdx,
             w_tLocal_V + laneID );
25.     }
        /* 3rd major step */
26.     if( IsUpdated( w_tLocal_V + laneID,
         VertexValues + globalTID ) ) {
27.       atomicExch( VertexValues + globalTID,
           w_tLocal_V[ laneID ] );
28.       converged = false;
29.     }
30.   } sync_device_with_host(); }
```

Figure 4. Framework's graph processing procedure pseudo-algorithm. Assumed warp size is 32. Shared memory pointers in the program code are declared with `volatile` qualifier.

First step: In this step (lines 11 to 14 in Figure 4) threads of a warp fetch 32 elements of *VertexValues* and initialize the designated shared memory region for vertex values using user-provided initialization function. The threads also put $32+1$ corresponding elements of *NbrIndices* into another shared memory buffer. Using the *NbrIndices* starting and ending element, warp lanes can recognize the region within *EdgeValues* and *NbrVertexIndices* arrays that belongs to the assigned group of vertices.

Second step: This step involves iteration of warp lanes over the elements of the *EdgeValues* and *NbrVertexIndices* arrays region (lines 15 to 25 in Figure 4). Warp lanes perform a user-provided compute function with the fetched neighbor vertex value and the connected edge value and save the outcome in a local shared memory buffer (line 21). Besides, every warp lane must discover which of 32 vertices that are assigned to the warp owns the processed edge and neighbor. This involves a $\log 32 = 5$ stepped binary search on fetched *edgeIndices* in the shared memory (line 18). Using the resulting vertex index, warp lanes can be grouped into segments, each segment corresponding to one vertex. Each lane identifies its position within the segment and the size of the segment it belongs to (lines 19 and 20). Therefore warp lanes can execute user-provided reduction function in parallel (line 22). Finally, the first lane in each segment performs the reduction function over the outcome and associated element in the shared memory region for vertex values (lines 23 and 24). Warp lanes perform these steps iteratively until all the edges for the set of vertices are processed.

Third step: In this step, the warp lanes compare the content of designated shared memory region for vertex values with the corresponding *VertexValues* elements using the user-provided function (line 26). If the function returns true, the vertex content inside the global memory will be updated.

Once all the vertices are processed, the framework executes another iteration of the algorithm on all the graph vertices if any vertex in the current iteration is updated. Graph processing with WS method dynamically determines the proper size for reduction based on the segment size and it is guaranteed that the number of steps for parallel reduction will never exceed five ($\log warpSize$).

Note that the memory transactions in all the steps are *coalesced* except for accessing the neighbor vertex value (line 17), which is inherent in the compact graph representation. However by moving "binary search" and "segment realization" functions (lines 18 to 20) before the neighbor computation function, we exploit instruction level parallelism to hide the latency associated with the non-coalesced memory access.

B. Framework Interface

In addition to trivial input/output handling functions, type definition for the vertex, and the structure definition for the edge, our framework accepts the following user specified functions:

```
struct Edge{ uint BW; };
typedef unsigned int Vertex;
inline __device__ void initVertex(
 volatile Vertex* initV, Veretx* V ){
  *initV = *V;
}
inline __device__ void ComputeNbr( Vertex SrcV,
 Edge* E, volatile Vertex* localV ) {
  *localV = min( SrcV, E->BW );
}
inline __device__ void ReduceVertices(
 volatile Vertex* firstV, Veretx* secondV ){
  *firstV = max( *firstV, *secondV );
}
inline __device__ bool IsUpdated(
 volatile Vertex* computedV, Veretx* V ){
  return ( *computedV > *V );
}
```

Figure 5. User-specified structures and functions for SSWP.

- *InitVertex* initializes the vertex at the beginning of an iteration.
- *ComputeNbr* is performed for every neighbor vertex.
- *ReduceVertices* acts as the reduction function between the results of *ComputeNbr* for two neighbors of a vertex.
- *IsUpdated* verifies if a vertex has updated during the current iteration.

Figure 5 illustrates use of the framework by showing the functions for Single Source Widest Path (SSWP) algorithm as an example. SSWP requires a variable for expressing the edge bandwidth and another variable for specifying maximum visible bandwidth by the vertex from the source. In SSWP, during multiple rounds, the content of a vertex is updated by the maximum bandwidth it observes picked from the minimums between incoming edges and corresponding neighbors. As Figure 5 shows, this algorithm can be easily expressed in our framework via the above processing functions. First, the vertex content inside the shared memory is initialized by the most updated content of the vertex. Second, for each neighbor a local value is computed, which in this case is the minimum between every connecting edge bandwidth and its corresponding source vertex visible bandwidth. Third, these values are reduced two-by-two using the reduction function and the result is saved to the first argument content. For SSWP, reduction function selects the maximum of visible values through neighbors. Also, at the end of the third step of the processing procedure, the reduction function is executed for the initialized vertex and the final reduction result. Finally, in the fourth step, the framework verifies if the vertex should be updated. If the *IsUpdated* function returns true—which in case of SSWP is observing a greater bandwidth to the source—the content of the vertex inside global memory is replaced with the reduced vertex content in the current iteration. If any vertex is updated, the host executes another iteration.

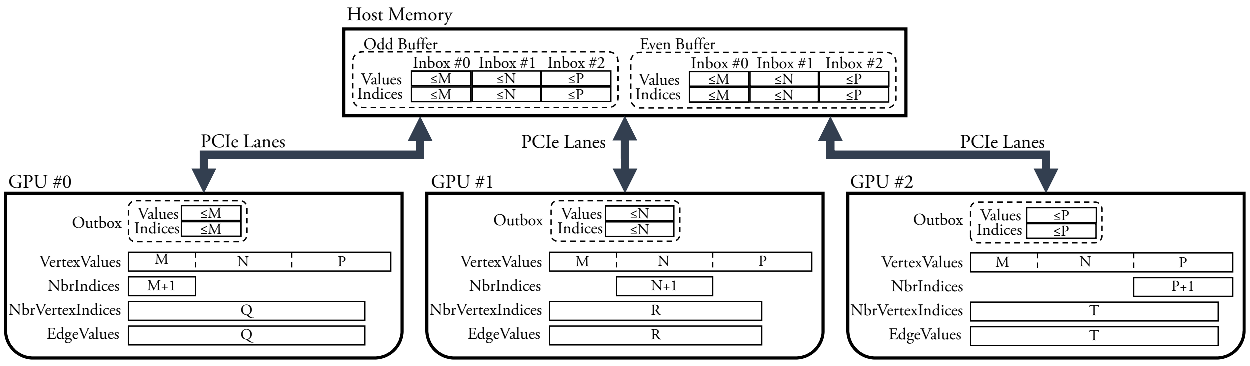


Figure 6. Organization of data structures in multi-GPU processing required for Vertex Refinement. The example graph represented in above configuration has the total number of $M+N+P$ vertices and $Q+R+T$ edges. The letters inside the boxes stand for the number of elements in the buffer. The size of inbox and outbox buffers are determined during Offline Vertex Refinement.

IV. Scaling via Vertex Refinement

To handle larger graphs we must scale our method to use multiple GPUs that provide more memory and processing resources. Although graph partitioning strategies between GPUs have been explored, inter-GPU data transfer efficiency has not received adequate attention. Given a partitioning, for scaling of graph processing to be effective, we must make good use of limited PCIe bandwidth. We show the inefficiency of existing techniques and present Vertex Refinement that avoids redundant data exchange between GPUs.

A. *Inefficiency of Existing Inter-GPU Communication*

Existing multi-GPU generic graph processing schemes divide the graph in two or more partitions and assign each partition to one GPU. Graph vertices completely fall into partitions while there can be edges that pass the partition boundaries. Due to these boundary edges, a GPU needs to be informed of the vertex updates happening in other GPUs. To keep the content of its assigned vertices held inside other GPUs updated, the GPU needs to transfer vertices belonging to its own partition over the PCIe bus. PCIe data transfer rate happens to be tens of times lower than GPU global memory's thus extra care must be taken to transfer only necessary data so as not to waste PCIe precious bandwidth.

Nonetheless, since implementing a mechanism to efficiently manage queues in GPU's massively multithreaded environment is challenging, previous works choose simple but inefficient approaches. Medusa [31] copies all the vertices belonging to one device to other devices at every iteration. We refer to this solution as the ALL method. TOTEM [7] [8] pre-selects the boundary vertices in a preprocessing stage but similar to Medusa copies the boundary vertices after every iteration. We refer to this solution as Maximal Subset (MS) method. Both of these methods suffer from wastage of PCIe bandwidth because usually only a small portion of the vertices are updated during each iteration. Table I shows the ratio of useful transferred vertices—vertices that are updated in the last iteration—to all the vertices that are transferred in such schemes. Such low percentages motivate the need for a new solution to utilize limited PCIe bandwidth economically.

B. *Vertex Refinement: Efficient inter-GPU Communication*

To eliminate the overhead of transferring unnecessary vertices between devices, our framework performs Vertex Refinement in two steps: *offline* and *online*. We first describe the required data structures and then present the two-staged refinement procedure.

Data structures for Vertex Refinement: To process a graph with multiple GPUs, our framework divides the vertices and their associated edges into partitions and assigns each partition to one GPU, so that each GPU processes a continuous set of vertices. Since the processing time is mostly affected by the memory accesses associated with gathering the values of neighbor vertices, determining the boundaries of vertex partitions depends upon the total number of edges that vertices of each subset hold. In our scheme, vertices of each partition will have roughly the same number of edges in order to provide a balanced load between GPUs. Each GPU will hold relevant subset of *NbrIndices*, *NbrVeretxIndices*, and *EdgeValues* but will contain a full version of *VertexValues* array. This organization allows each device to process vertices belonging to its own partition as long as vertices inside *VertexValues* that belong to other GPUs are updated during an iteration.

In addition to CSR representation buffers, each GPU will hold one *Outbox* buffer that is filled with updated vertex indices and vertex values of the GPU-specific division. As shown in Figure 6, we keep the inboxes inside host pinned buffers. In other words, the set of host buffers is similar to a hub that are filled by devices. At the start of an iteration, a device accesses inboxes corresponding to other devices and updates its own *VertexValues* array. Also at the end of an iteration, the device transfers its own outbox content to device's corresponding inbox. Moreover, we apply double buffering technique by alternating read buffers and write buffers. In an odd (even) iteration, devices read from the

Graph Algorithm	ALL %	MS %
Breadth-First Search (BFS)	10.43	12.21
Connected Components (CC)	9.55	11.19
Circuit Simulation (CS)	2.34	2.39
Heat Simulation (HS)	31.29	36.66
Neural Network (NN) [2]	15.66	18.34
PageRank (PR) [25]	10.38	13.65
Single Source Shortest Path (SSSP)	13.65	15.99
Single Source Widest Path (SSWP)	3.14	3.68

Table I
THE PERCENTAGE OF USEFUL VERTEX DATA AMONG ALL THE TRANSFERRED DATA WHEN ALL THE VERTICES (ALL) OR THE MAXIMAL SUBSET OF THEM (MS) ARE COPIED FROM ONE GPU TO ANOTHER. IN THIS TWO-GPU CONFIGURATION, THE GRAPH UNDER THE EXAMINATION IS AN RMAT GRAPH WITH APPROXIMATELY 40 MILLION VERTICES AND 470 MILLION EDGES.

odd (even) inbox buffers and copy their outbox to their designated even (odd) inbox buffer. In summary:

– *Inbox and outbox buffers* are vital for a fast data transfer between GPUs. Direct peer-device memory access as an alternative will introduce significant performance penalty due to non-coalesced transactions over PCIe bus [28]. In contrast, inbox and outbox buffers allow the collection of necessary data together and hence accelerate the inter-device communication.

– *Using Host memory as the hub* not only reduces memory constraint pressure for GPUs, but is also beneficial when more than two GPUs are processing the graph. A device copies its own outbox to a host buffer only once. In contrast, if there is no intermediate host buffer, the device has to copy the outbox to each of the other GPUs' inboxes causing unnecessary traffic over connected PCIe lanes since the same data are passed over more than once. Our experiments show that using host as the hub is always beneficial in reducing the communication traffic and overall multi-GPU processing time in comparison to using inbox and outbox buffers residing inside the GPUs.

– *Double buffering* eliminates the need for additional costly inter-device synchronization barriers between data transfers and kernel executions. For instance, when device A grabs inbox buffer content of the device B during an iteration, since device B is going to fill another inbox buffer in the current iteration, needless of synchronizing with device B we will be sure that device A does not receive corrupted data.

If there are two GPUs processing the graph, during the runtime our framework queries the available global memory on the GPUs. If there is enough memory to hold the pertained part of the graph plus both the odd and even inboxes belonging to the other device, the framework puts the inboxes inside the GPUs global memory. Otherwise, it chooses host pinned buffers for this purpose.

Offline Vertex Refinement: In this pre-processing stage, the framework scans *NbrVertexIndices* elements and identifies the boundary vertices: those that are being accessed by edges of one division while belonging to another division.

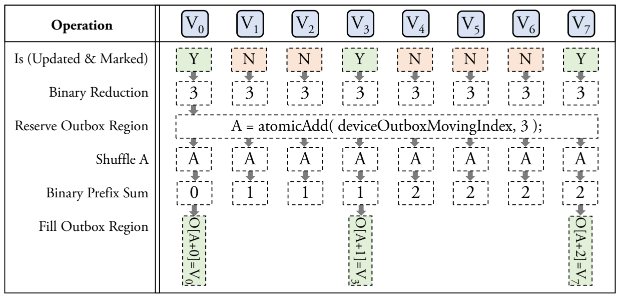


Figure 7. An example of online vertex refinement stages via intra-warp inclusive binary prefix sum – warp size in the figure is 8.

Inspired by TOTEM [7], for such a vertex we set the most significant bit of its corresponding element inside *NbrIndices* buffer. During the online refinement, if a vertex is not a boundary vertex, it will be filtered out. Note that this bit will be ignored during other computations that involve *NbrIndices* buffer. Also during this stage, the framework can determine the maximum size to allocate for inbox and outbox buffers.

Online Vertex Refinement via parallel binary prefix sum: As opposed to Offline Vertex Refinement, Online Vertex Refinement happens on-the-fly inside the GPU kernel. At the last level of graph processing, lanes of a warp examine warp-assigned vertices for updates, each producing a binary predicate associated with one vertex. If this predicate is true and at the same time the vertex is marked during the offline stage, the vertex is required to be transferred to other devices.

By means of `__any()` intrinsic, we first verify if any of the warp lanes has an eligible vertex to transfer. If yes, warp lanes quickly count the total number of updated vertices inside the warp via intra-warp binary reduction and realize the number of updated vertices in lower lanes via intra-warp binary prefix sum. For a fast computation of binary reduction and inclusive binary prefix sum, our framework utilizes Harris et al. approach [10] in which `__popc()` and `__ballot()` CUDA intrinsic functions are exploited. Having total number of updated vertices, one lane in the warp atomically adds it to a moving index inside the global memory, which its returned value specifies the starting position in the designated outbox buffer to write the warp's updated vertex indices and values. In other words, a lane reserves a region inside the Outbox for eligible warp lanes. The starting position of the region is shuffled to other lanes in the warp via `__shfl()` intrinsic, and lanes with updated vertex fill up the buffer using this position plus their intra-warp prefix sum. Figure 7 presents an example showing online vertex refinement procedure.

An alternative to above approach is extending the binary reduction and the binary scan to the CTA; however, we did not find this alternative faster since it required two synchronizations across the thread-block. Whereas in our approach the atomic addition is performed by only one lane in the warp which avoids contention for the atomic variable.

When processing in an iteration is done, the moving index determines how much of the device outbox buffer has been filled. We significantly reduce the communication time by transferring the content of this buffer to the corresponding inbox buffer *only with the length specified by the moving index*. At the beginning of the next iteration, in order to have newly updated vertex values from other devices, each device distributes the content of other devices' inboxes *only with the length specified by their associated moving indexes* over its own *VertexValues* array.

In summary, Offline Vertex Refinement identifies boundary vertices and Online Vertex Refinement recognizes the vertices updated in the previous iteration. The combination of two yields the set of updated boundary vertices and maximizes the inter-device communication efficiency.

V. Experimental Evaluation

The system we performed experiments on has 3 NVIDIA GeForce GTX780 GPUs each having 12 Kepler Streaming Multiprocessors and approximately 3 GBytes of global memory. The first GPU is connected to the system with PCIe 3.0 16x while the rest are operating at 4x speed. The single-GPU experiments are reported from the GPU with the highest PCIe bandwidth. We compiled and ran all programs for Compute Capability 3.5 on Ubuntu 14.04 64-bit with CUDA 6.5 and applied the highest optimization level flag.

A. Warp Segmentation Performance Analysis

In this section, we analyze the performance of Warp Segmentation on a single GPU. We use the graphs shown in Table II for experiments in this section. In the table, graphs with prefix *RM* refer to Rmat [3] graphs created using PaRMAT [14] with parameters $a = 0.45$, $b = 0.25$, and $c = 0.15$. Rmat graphs are known to imitate the characteristics of real-world graphs such as power-law degree distribution. The graph with prefix *ER* is a uniformly random (Erdős-Rényi) graph. Other graphs are extracted from real-world origins and are publicly available at SNAP dataset [16]. Graphs in Table II cover a wide range of sizes with different densities and characteristics.

Warp Segmentation vs VWC – Performance Comparison: First we compare the performance of WS method against VWC with graphs shown in Table II for the benchmarks in Table I. Table III presents the raw processing time for the completion of all the benchmarks over all the graphs for both methods. We experimented on VWC with all the possible virtual warp sizes (2, 4, 8, 16, and 32) hence its processing times are specified in ranges. Table IV shows the average speedup of WS compared to VWC over input graphs and benchmarks. In comparison with VWC, WS shows better performance across all the graphs and all the benchmarks. WS speedup over VWC averaged across all the input graphs and benchmarks ranges from 1.29x to 2.80x.

Input Graph	N. Vertices	N. Edges	CSR size	CW size
RM33V335E	33 554 432	335 544 320	1611-3087	**5503-8321**
ComOrkut [30]	3 072 441	234 370 166	962-1912	**3762-5649**
ER25V201E	25 165 824	201 326 592	1007-1913	**3322-5033**
RM25V201E	25 165 824	201 326 592	1007-1913	**3322-5033**
RM16V201E	16 777 216	201 326 592	940-1812	**3288-4966**
RM16V134E	16 777 216	134 217 728	671-1275	2215-**3355**
LiveJournal [1]	4 847 571	68 993 773	315-610	1123-1695
SocPokec [27]	1 632 803	30 622 564	136-265	496-748
HiggsTwitter [6]	456 631	14 855 875	63-124	240-360
RoadNetCA [17]	1 965 214	5 533 214	38-68	96-149
WebGoogle [17]	875 713	5 105 039	27-51	85-130
Amazon0312 [18]	400 727	3 200 440	16-30	53-80

Table II
Graphs used in single-GPU experiments – across benchmarks the size ranges in MBytes for CSR and CW representations. Sizes exceeding GPU's global memory capacity are bolded.

Averages Across Input Graphs		Averages Across Benchmarks	
BFS	1.27x−2.60x	RM33V335E	1.23x−1.56x
CC	1.33x−2.90x	ComOrkut	1.15x−1.99x
CS	1.43x−3.34x	ER25V201E	1.09x−1.69x
HS	1.27x−2.66x	RM25V201E	1.15x−1.57x
NN	1.21x−2.70x	RM16V201E	1.16x−1.41x
PR	1.22x−2.68x	RM16V134E	1.22x−1.69x
SSSP	1.31x−2.76x	LiveJournal	1.29x−1.99x
SSWP	1.28x−2.80x	SocPokec	1.27x−1.77x
		HiggsTwitter	1.34x−4.78x
		RoadNetCA	1.24x−9.90x
		WebGoogle	1.79x−2.69x
		Amazon0312	1.53x−2.68x

Table IV
Speedup ranges of Warp Segmentation over VWC excluding data transfer times. Since both methods use CSR representation, their data transfer times are equal.

To further examine the effectiveness of WS against VWC, as the state-of-the-art CSR based generic graph processing method, we profiled both our framework and VWC over different graphs for warp execution efficiency. Figure 8 shows the average warp execution efficiency (predicated and non-predicated combined) over all the iterations of graph processing with *SSSP* benchmark. It is evident from the figure that for different graphs, best warp execution efficiency for VWC happens in different virtual warp sizes. For example with *RoadNetCA*, a 2D mesh of intersections and roads, virtual warp size 2 yields the best results due to special structure of the graph; while it leads to the poorest performance for other graphs. On the other hand, WS exhibits a steady warp execution efficiency (71.8% on average) regardless of the graph. WS warp execution efficiency is 1.75x-3.27x better than VWC when averaged across all graphs. This confirms the SIMD efficiency of WS over fixed-width intra-SIMD thread assignment in VWC.

Warp Segmentation Performance against CW: We present the speedup of WS over CW having large graphs in Table V and having small graphs in Table VI. For the large graphs, CW representation cannot fit the whole graph inside GPU global memory. For these combinations, CuSha fails;

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM33V335E	WS	1257	1118	1629	2812	1416	6056	2882	5505
	VWC	1428-1811	1270-1680	2012-2562	3501-4412	2030-2506	6563-8275	3237-3959	6740-8268
ComOrkut	WS	403	351	4162	681	904	4290	1398	931
	VWC	455-664	382-572	5566-8847	692-1056	989-1634	6296-13334	1515-2519	1029-1626
ER25V201E	WS	837	644	704	8330	773	5004	2181	2462
	VWC	976-1385	710-1045	748-1313	9499-16047	805-1160	5287-6095	2386-3505	2574-3756
RM25V201E	WS	845	835	1052	4782	1023	3856	1802	4216
	VWC	933-1231	935-1233	1287-1709	5619-8716	1190-1529	4183-5491	2080-2653	4787-5991
RM16V201E	WS	667	663	959	1762	840	3762	1625	2998
	VWC	750-907	746-908	1187-1438	2058-2337	984-1159	4043-4526	1800-2230	3403-4284
RM16V134E	WS	512	514	660	1244	572	4068	1159	2028
	VWC	591-820	592-822	850-1218	1539-2133	691-913	4448-5656	1402-1832	2427-3267
LiveJournal	WS	172	154	535	346	2061	2326	446	772
	VWC	215-296	201-273	807-1084	378-536	2297-4746	2498-4043	619-814	1059-1345
SocPokec	WS	75	66	121	226	464	1145	194	194
	VWC	90-107	80-106	175-203	264-329	614-761	1302-2817	237-327	236-314
HiggsTwitter	WS	48	37	117	75	159	483	100	77
	VWC	54-170	49-178	157-495	95-294	192-812	927-2433	113-432	98-355
RoadNetCA	WS	386	330	1694	41	193	55	465	1077
	VWC	480-3400	493-3437	2392-23659	45-301	191-1668	62-448	619-4402	118-5619
WebGoogle	WS	41	36	61	15	84	109	63	108
	VWC	81-109	75-99	124-186	23-35	124-167	145-248	113-172	186-288
Amazon0312	WS	17	17	263	81	41	44	33	38
	VWC	25-46	26-46	419-797	142-237	42-78	63-110	63-90	57-92

Table III
RAW RUNNING TIMES (MS) OF WARP SEGMENTATION (WS) AND VWC INCLUDING KERNEL EXECUTIONS AND HOST-DEVICE DATA TRANSFERS FOR DIFFERENT ALGORITHMS AND DIFFERENT GRAPHS.

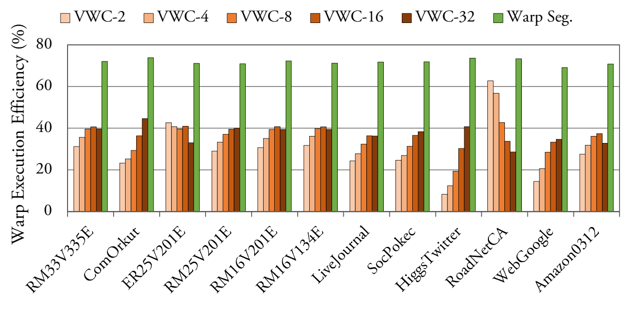


Figure 8. Profiled average warp execution efficiency of Warp Segmentation compared to VWC's. *SSSP* is the benchmark.

therefore, as a straightforward workaround, we kept vertex value and small auxiliary buffers inside the GPU global memory and put shards at mapped pinned buffers inside the host. For large graphs, CW processing time is significantly higher than our method's due to involvement of PCIe bus, limiting the scalability of CW representation. Also for the small graphs, although CW provides fully regular access patterns, it incurs larger memory footprints. In addition, our framework covers the latency of CSR-inherent irregular accesses, therefore we observe near par performance, as shown by averages in Table VI.

B. Vertex Refinement Performance Analysis

Next we analyze the performance of our framework when it is scaled to multiple GPUs. First we present the speedup provided by Vertex Refinement compared to existing methods over very large input graphs, and analyze its cost and benefits. For the experiments in this section, we created 12 Rmat and Erdős-Rényi graphs with different sizes and densities, shown in Table VII. Six of these graphs can be fit inside two of our GPUs and Six require three GPUs. Finally, we analyze the performance when smaller graphs are processed on multiple GPUs.

Input Graph	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM33V335E	3.41	3.21	8.44	14.14	4.02	5.38	4.36	4.66
ComOrkut	5.11	5.91	1.72	10.76	5.23	6.85	7.92	5.72
ER25V201E	3.47	3.36	6.20	10.43	3.72	2.59	4.46	4.34
RM25V201E	3.07	2.76	7.71	9.65	3.55	3.54	3.99	4.14
RM16V201E	3.45	3.06	6.53	8.42	3.87	4.50	4.63	4.41
RM16V134E	x	x	3.19	4.97	x	3.93	x	x
Average	3.70	3.66	5.63	9.73	4.08	4.47	5.07	4.65

Table V
THE SPEEDUP OF WARP SEGMENTATION OVER CUSHA'S [13] CW FOR *large* GRAPHS. THE SHARDS RESIDE INSIDE THE HOST PINNED BUFFERS (X MEANS GRAPH IS SMALL - FITS IN GPU MEMORY).

Input Graph	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM16V134E	0.74	0.80	x	x	0.88	x	0.67	0.56
LiveJournal	1.06	1.21	0.74	1.10	1.03	0.60	0.86	0.82
SocPokec	0.92	1.02	1.04	0.81	0.41	0.34	0.73	0.67
HiggsTwitter	1.48	2.30	1.48	1.64	2.20	1.19	1.65	2.03
RoadNetCA	0.67	1.13	0.98	0.92	1.02	1.20	0.76	0.91
WebGoogle	0.58	0.82	0.78	1.74	1.69	0.59	0.61	0.74
Amazon0312	1.05	1.47	0.39	0.91	0.91	0.97	1.21	1.20
Average	0.93	1.25	0.90	1.19	1.16	0.82	0.93	0.99

Table VI
THE SPEEDUP OF WARP SEGMENTATION OVER CUSHA'S [13] CW FOR *small* GRAPHS. THE SHARDS RESIDE INSIDE THE GPU'S GLOBAL MEMORY (X MEANS GRAPH IS LARGE - REQUIRES HOST MEMORY).

Input Graph	N. Vertices	N. Edges
RM54V704E	54 525 952	704 643 072
ER50V671E	50 331 648	671 088 640
RM50V671E	50 331 648	671 088 640
RM46V671E	46 137 344	671 088 640
RM46V603E	46 137 344	603 979 776
RM41V536E	41 943 040	536 870 912
RM41V503E	41 943 040	503 316 480
ER39V469E	39 845 888	469 762 048
RM39V469E	39 845 888	469 762 048
RM37V469E	37 748 736	469 762 048
RM37V436E	37 748 736	436 207 616
RM35V402E	35 651 584	402 653 184

Table VII
GRAPHS FOR MULTI-GPU EXPERIMENTS: TOP 6 GRAPHS USED IN EXPERIMENTS WITH 3 GPUS; REST USED WITH 2 GPUS.

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM54V704E	over ALL	1.85	1.81	2.53	1.64	1.66	1.48	1.75	2.03
	over MS	1.82	1.78	2.46	1.59	1.63	1.47	1.71	1.98
ER50V671E	over ALL	1.64	1.36	2.19	1.55	1.43	1.22	1.72	2.02
	over MS	1.67	1.4	2.24	1.49	1.48	1.21	1.76	2.05
RM50V671E	over ALL	1.83	1.76	2.51	1.68	1.63	1.49	1.72	1.98
	over MS	1.78	1.74	2.47	1.6	1.6	1.36	1.68	1.93
RM46V671E	over ALL	1.78	1.77	2.48	1.7	1.62	1.43	1.72	1.98
	over MS	1.75	1.74	2.42	1.64	1.6	1.41	1.69	1.93
RM46V603E	over ALL	1.84	1.82	2.58	1.67	1.67	1.43	1.79	2.07
	over MS	1.81	1.8	2.51	1.59	1.64	1.37	1.75	2.01
RM41V536E	over ALL	1.89	1.84	2.71	1.62	1.69	1.44	1.8	2.1
	over MS	1.82	1.81	2.63	1.58	1.66	1.39	1.75	2.04
RM41V503E	over ALL	1.29	1.29	1.61	1.23	1.21	1.18	1.24	1.35
	over MS	1.27	1.28	1.57	1.21	1.2	1.15	1.21	1.32
ER39V469E	over ALL	1.21	1.06	1.49	1.18	1.14	1.19	1.23	1.21
	over MS	1.23	1.09	1.53	1.16	1.17	1.15	1.25	1.24
RM39V469E	over ALL	1.29	1.3	1.64	1.28	1.21	1.39	1.26	1.38
	over MS	1.28	1.28	1.61	1.24	1.2	1.29	1.23	1.35
RM37V469E	over ALL	1.26	1.26	1.6	1.24	1.2	1.23	1.22	1.36
	over MS	1.25	1.26	1.57	1.21	1.19	1.18	1.2	1.33
RM37V436E	over ALL	1.33	1.32	1.66	1.27	1.22	1.25	1.28	1.41
	over MS	1.31	1.29	1.63	1.23	1.22	1.24	1.26	1.39
RM35V402E	over ALL	1.32	1.31	1.72	1.28	1.23	1.21	1.25	1.41
	over MS	1.3	1.29	1.66	1.23	1.22	1.2	1.22	1.38

Table VIII
THE SPEED-UP OF VR OVER ALL AND MS FOR THREE-GPU AND TWO-GPU CONFIGURATIONS.

Vertex Refinement Performance Comparison: To better realize the importance of data communication strategy and the efficiency of Vertex Refinement, we have implemented two other inter-device communication methods (mentioned in Section IV) in our framework. The first method is the straightforward solution that copies all the vertices belonging to one device to other devices at every iteration. We refer to this solution as *ALL*. The second one is the maximal subset method where vertices that belong to one device and can be accessed by another device are identified in a pre-processing stage. During the iterative execution, only these vertices are communicated to other devices. We refer to this method as *MS*. We compare these methods with Vertex Refinement - *VR*. Note that to better realize the benefits of VR, for all the inter-device communication methods, we keep intra-device processing style intact. In other words, underlying graph processing method is WS for all experiments in this section.

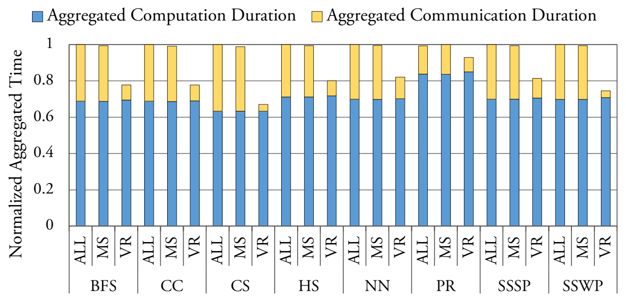
Table VIII shows the speedup of our framework when VR is employed over ALL and MS, for all the graphs and benchmarks. In all cases, our solution performs better than other methods. When averaged over all the graphs and benchmarks, our approach provides 1.81x and 1.30x speedups over ALL and 1.77x and 1.28x speedups over MS for three-GPU and two-GPU configurations respectively.

In Figure 9, we analyzed the cost of VR queue management versus the savings it provides (in terms of eliminating redundant inter-device vertex communication) by breaking down the processing time into computation time and communication time. To create this plot, we measured the time for each and every kernel execution, memory copy, and outbox-loading/inbox-unloading. Aggregated computation duration refers to the total duration of GPU kernel executions, whereas aggregated communication time refers to the total duration of copies and/or box handling kernels.

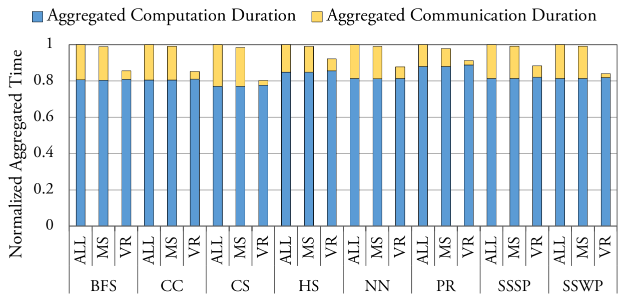
First, it is evident from both plots in Figure 9 that MS is not an effective solution for reducing communication overhead. In fact, in one case (*PR* in Figure 9(a)) the overhead of outbox handling overcomes the benefits of preselection. Second, unlike MS, VR significantly reduces the total communication duration by refining vertices on-the-fly while adding negligible overhead to the computation duration. Note that even though in VR the vertex information is communicated accompanying its index, the communication duration is still much less compared to ALL and MS for all the cases. Third, by comparing Figure 9(a) and Figure 9(b), we notice that more time is spent on the communication by employing more GPUs. By adding another GPU, each device needs to send and receive more vertex information to and from more devices, signifying VR's supremacy even further. Especially in the 3-GPU configuration, using host as the hub supports reducing inter-device traffic by passing the data over PCIe only once.

Scaling to multiple GPUs for smaller graphs: To observe the effect of scaling graph processing procedure from one or two GPUs to three GPUs, we experimented our framework with smaller graphs and more GPUs and reported the speedups in Table IX. As this table shows, the performance does not scale linearly as we add more GPUs. This is due to comparatively slow PCIe paths and also imperfect load division between different GPUs. Also, the speedup of adding more GPUs greatly depends on the graph algorithm. For example, in PageRank (PR) the chances that a vertex is updated during an iteration is relatively high (especially in earlier iterations) thus more vertices have to be transferred from one GPU to another. As a result, we observe lower speedups in PageRank compared to other algorithms when adding more GPUs.

We also present the effect of the graph characteristics (graph size and density) on the scalability of our framework in Figure 10. By comparing large graphs and small graphs



(a) RM54V704E graph with 3 GPUs.



(b) RM41V503E graph with 2 GPUs.

Figure 9. Processing-time break down into computation time and communication time for the Vertex Refinement (VR) compared to ALL and MS. Computation time is the total duration of kernel execution, and communication time is the total duration of inbox/outbox management plus inter-device memory copies. For each benchmark, the times are normalized with respect to the longest time. Note that this times cannot be used to infer the overall speedup due to asynchronicity of devices.

Input Graph	GPUs	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM41V503E	3 vs. 2	1.39	1.38	1.32	1.23	1.21	1.12	1.32	1.35
ER39V469E	3 vs. 2	1.36	1.11	1.44	1.19	1.33	1.09	1.28	1.26
RM39V469E	3 vs. 2	1.3	1.37	1.42	1.22	1.28	1.13	1.42	1.29
RM37V469E	3 vs. 2	1.21	1.27	1.32	1.24	1.38	1.19	1.34	1.43
RM37V436E	3 vs. 2	1.22	1.18	1.32	1.17	1.21	1.18	1.28	1.34
RM35V402E	3 vs. 2	1.5	1.37	1.42	1.23	1.29	1.14	1.33	1.39
RM33V335E	3 vs. 1	1.75	1.56	1.95	1.33	1.52	1.1	1.55	1.59
	2 vs. 1	1.27	1.24	1.4	1.07	1.12	1.06	1.21	1.2
ComOrkut	3 vs. 1	1.65	1.81	1.95	1.28	1.97	1.43	1.96	1.85
	2 vs. 1	1.19	1.31	1.65	1.15	1.36	1.32	1.4	1.39
ER25V201E	3 vs. 1	1.5	1.55	1.44	1.19	1.38	1.18	1.48	1.58
	2 vs. 1	1.14	1.33	1.16	1.07	1.13	1.15	1.11	1.19
RM25V201E	3 vs. 1	1.47	0.96	1.56	1.29	1.38	0.93	1.29	1.17
	2 vs. 1	1.08	0.97	1.26	1.1	1.08	1.01	1.07	0.94
RM16V201E	3 vs. 1	1.45	1.64	1.74	1.3	1.56	1.02	1.42	1.6
	2 vs. 1	1.26	1.36	1.44	1.12	1.21	1.06	1.17	1.34
RM16V134E	3 vs. 1	1.36	1.58	1.86	1.36	1.47	1.19	1.46	1.66
	2 vs. 1	1.21	1.21	1.44	1.14	1.11	1.12	1.12	1.31

Table IX
THE SPEEDUP OF OUR FRAMEWORK WHEN SCALING TO MORE GPUS: FROM 2 TO 3 GPUS FOR THE TOP 6 GRAPHS; AND FROM 2 TO 3 AND FROM 1 TO 2 GPUS FOR THE REST OF THE GRAPHS.

in Figure 10, we observe that as the graphs get larger with greater number of edges, adding more GPUs produces greater reductions in graph processing time. In addition, higher density in larger graphs signifies the reduction in the processing time when scaling to multiple GPUs by downsizing inter-device vertex transfer volumes.

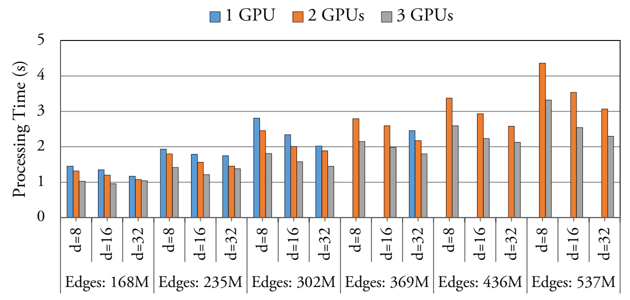


Figure 10. The scalability of our framework over graphs with different number of edges and densities for *SSSP* benchmark. All the graphs are Rmat created with parameters $a = 0.45$, $b = 0.25$, and $c = 0.15$. y axis is the processing time (lower is better).

VI. Related Work

Harish and Narayanan pioneered GPU graph processing in [9] by privatizing the processing of a vertex to one GPU thread. This solution was is prone to load imbalance (i.e., warp execution inefficiency) and suffers heavily from non-coalesced accesses to edge and vertex indices. Hong et al. improved upon this solution with Virtual-Warp Centric manner of graph processing [11] [12] nonetheless, as explained in the context, this method does not efficiently utilize available SIMD resources. CuSha [13] is a generic CUDA graph processing framework that uses G-Shards and CW representation to avoid warp execution inefficiencies and non-coalesced memory accesses. Although effective, such representations consume 2 to 2.5 times more space than CSR which can hinder the framework from processing very large graphs. In addition, CuSha relies on atomic operation in the computation function which can be limiting general applicability of the framework. [29] proposes an static load-balancing scheme that puts vertices into multiple bins based on the number of neighbors and assigns appropriate number of threads to each bin accordingly. [4] and [26] aim to provide regular GPU-friendly data patterns in order to balance the load however their usage is confined to predictable data structures. Moreover, [22] and [24] propose solutions for algorithms that change the structure of the graph. Our framework, in contrast, focuses on the graphs in which the connectivity of vertices via edges do not change at any time.

Merril et al. recognized the potential of parallel scan on GPUs for graph traversal in order to efficiently construct vertex frontiers and edge frontiers [23] however their solution is limited to BFS. Similarly, [5] and [21] suggest work-efficient solutions respectively for SSSP and betweenness centrality. Our focus in this work is rather on a scalable and generic framework that allows the expression of numerous iterative vertex-centric algorithms and proposing generally-applicable techniques. [20] is also for BFS graph traversal that due to its excessive usage of atomic operations to control the queue does not exhibit acceptable performance. We avoid the contention over the atomic variable by mainly relying on binary prefix sum for vertex refinement and involving only one warp lane in the outbox region reservation process.

In order to involve more GPUs to process the graph, Medusa [31] employs METIS [15] an off-the-shelf graph partitioner in order to distribute vertices between devices and reduce the number of edges that pass the boundaries. TOTEM [7] is a CPU-GPU hybrid framework that pre-processes the graph and uses the highest order bits of neighbor vertex indices to flag the boundary vertices so the read access is redirected to the device inbox. However, the whole content of the outbox in TOTEM or all the partition vertices in Medusa have to be copied over to the remote device, incurring massive unnecessary traffic over PCIe lanes. [12] is another CPU-GPU hybrid solution for BFS that after a few iteration, transfers the whole graph from the CPU side to the GPU side. Our work is the first generic multi-GPU framework that reduces inter-device communication by filtering out not-updated and non-boundary vertices.

VII. Conclusion

We introduced a CUDA-based framework for efficient scaling of iterative graph algorithms to larger graphs and multiple GPUs. The graphs are stored in the space-saving CSR form that allows processing large graphs. To overcome the SIMD execution inefficiency we employ Warp Segmentation leading to 1.29x−2.80x speedup over state-of-the-art VWC method. To scale the graph processing over multiple GPUs in our framework, we introduced Vertex Refinement that collects and transfers only those vertices that are boundary and recently updated. Vertex Refinement maximizes inter-device bandwidth utilization leading to 2.71x speedup over existing multi-GPU communication schemes.

Acknowledgment

This work is supported by NSF Grants CCF-0905509, CNS-1157377, CCF-1318103, and CCF-1524852.

References

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *KDD*, 2006, pp. 44–54.

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

[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, ch. 43, pp. 442–446.

[4] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *SC*, 2011, pp. 13:1–13:11.

[5] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single source shortest paths," in *IPDPS*, May 2014, pp. 349–359.

[6] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi, "The anatomy of a scientific rumor," *Sci. Rep.*, vol. 3, 2013.

[7] 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*, 2012, pp. 345–354.

[8] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu, "Efficient large-scale graph processing on hybrid CPU and GPU systems," *CoRR*, vol. abs/1312.3018, 2013. [Online]. Available: http://arxiv.org/abs/1312.3018

[9] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *HiPC*, 2007.

[10] M. Harris and M. Garland, "Optimizing parallel prefix operations for the fermi architecture. in *gpu computing gems, jade edition*," *Morgan Kaufmann*, pp. 29–38, 2011.

[11] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *PPoPP*, 2011, pp. 267–276.

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

[13] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *HPDC*, 2014.

[14] F. Khorasani, K. Vora, and R. Gupta, "Parmat: A parallel generator for large r-mat graphs," 2015. [Online]. Available: https://github.com/farkhor/PaRMAT

[15] D. Lasalle and G. Karypis, "Multi-threaded graph partitioning," in *IPDPS*, 2013, pp. 225–236.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[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," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

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

[19] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *PPL*, 17(1), 2007.

[20] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *DAC*, 2010, pp. 52–55.

[21] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *SC*, 2014, pp. 572–583.

[22] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A gpu implementation of inclusion-based points-to analysis," in *PPoPP*, 2012, pp. 107–116.

[23] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *PPoPP*, 2012, pp. 117–128.

[24] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *PPoPP*, 2013, pp. 147–156.

[25] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

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

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

[28] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013, pp. 127–128.

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

[30] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *ICDM*, 2012.

[31] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE TPDS*, 25:6, pp. 1543–1552, June 2014.