SIMD-X: Programming and Processing of Graph Algorithms on GPUs

Hang Liu
University of Massachusetts Lowell

H. Howie Huang
The George Washington University

Abstract

With high computation power and memory bandwidth, graphics processing units (GPUs) lend themselves to accelerate data-intensive analytics, especially when such applications fit the single instruction multiple data (SIMD) model. However, graph algorithms such as breadth-first search and k-core, often fail to take full advantage of GPUs, due to irregularity in memory access and control flow. To address this challenge, we have developed SIMD-X, for programming and processing of single instruction multiple, complex, data on GPUs. Specifically, the new Active-Compute-Combine (ACC) model not only provides ease of programming to programmers, but more importantly creates opportunities for system-level optimizations. To this end, SIMD-X utilizes just-in-time task management which filters out inactive vertices at runtime and intelligently maps various tasks to different amount of GPU cores in pursuit of workload balancing. In addition, SIMD-X leverages push-pull based kernel fusion that, with the help of a new deadlock-free global barrier, reduces a large number of computation kernels to very few. Using SIMD-X, a user can program a graph algorithm in tens of lines of code, while achieving 3×, 6×, 24×, 3× speedup over Gunrock, Galois, CuSha, and Ligra, respectively.

1 Introduction

The advent of big data [40, 27, 35, 36, 37, 5, 25, 26, 28, 14, 83] exacerbates the need of extracting useful knowledge within an acceptable time envelope. For performance acceleration, many applications utilize graphics processing units (GPUs) whose huge success comes from exploiting the data-level parallelism in these applications. Implicitly, the traditional single instruction multiple data (SIMD) model of GPUs assumes regular programming and processing, that is, not only the same instruction is executed but also the same amount of work is expected to perform on each piece of data. Unfortunately, neither assumption holds true for many emerging irregular applications, especially graph analytics which is the focus of this work. That is, such applications do not conform to the SIMD model, where different amount of work, or worse, completely different work, need to be performed on the data in parallel.

To enable graph computation on GPUs, this work advocates a new parallel framework, SIMD-X, for the programming and processing of *single instruction multiple, complex, data* on GPUs. At the heart of SIMD-X is the decoupling of programming and processing, that is, SIMD-X utilizes the *data-parallel* model for ease of expressing of graph applications, while enabling system-level optimizations at run time to deal with the *task-parallel* complexity on GPUs. With SIMD-X, a programmer simply needs to define what to do on which data, without worrying about the issues arisen from irregular memory access and control flow, both of which prevent GPUs from achieve massive parallelism.

SIMD-X consists of three major components: First, SIMD-X utilizes a new Active-Compute-Combine (ACC) programming model that asks a program to define three data-parallel functions: the condition for determining an *active* vertex, *computation* to be performed on an associated edge, and *combining* the updates from edge compute to vertex state. As we will show later, ACC is able to support a large variety of graph algorithms from breadth-first search, k-core, to belief propagation. While ACC adopts the Bulk Synchronous Parallel (BSP) model [49], it differs from traditional CPU-based graph abstractions such as edge- or vertex-centric models in that ACC avoids atomic operation, enables collaborative early termination (for BFS) and fine-grained task management on GPUs.

Second, SIMD-X relies on just-in-time (JIT) task management to balance parallel workloads across different GPU cores with minimal overhead. A good task list can increase not only parallelism, but also sequential memory access for the computation of next iteration, both

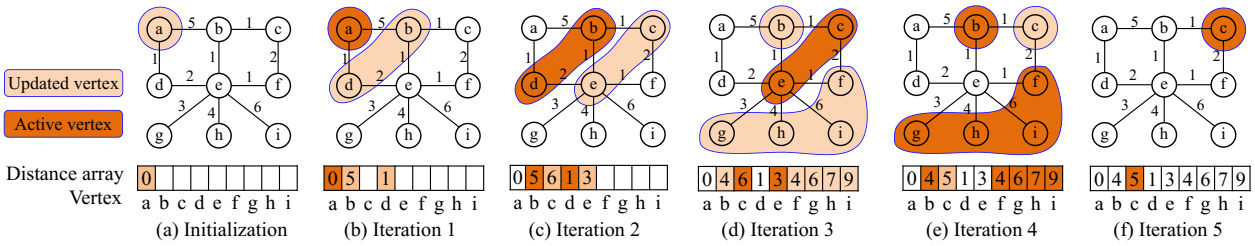


Figure 1: SSSP on a graph, with nine vertices {a, b, c, d, e, f, g, h, i} and ten undirected edges (with weights). SSSP iteratively computes on the graph and generates the distance array. Particularly, heavy and light shadows represent active and most recently updated vertices, respectively.

of which are crucial for high-performance computing on GPUs. To this end, we have designed a set of new task management mechanisms, that is, online and ballot filters, each of which excels at the complementary scenarios, i.e., the former favors a small amount of tasks while the latter larger tasks. At runtime, SIMD-X judiciously selects the more suitable filter to assemble the active work list for the next iteration. Our JIT task management can largely reduce the memory consumption, thereby accommodate the graphs much larger than prior work [50, 77]. Moreover, SIMD-X delivers 16×, on average, speedup across various algorithms and graphs.

Third, SIMD-X designs a new technique of push-pull based kernel fusion which aims to further accelerate graph computing by reducing kernel invocation overhead and global memory traffic. SIMD-X addresses the deadlock issue which occurs in existing software global barrier [79] that is adopted by Gunrock [77]. Besides, instead of aggressively fusing the algorithm into one giant kernel, SIMD-X fuses the kernels around the pull and push stages within each computation to minimize both register consumption and kernel relaunching. The evaluation shows that the new fusion technique can reduce the register consumption by half and thus double the configurable thread count, leading to 42% and 25% performance improvement over non-fused and aggressive fusion, respectively.

SIMD-X is different from prior work in several aspects. First, despite an array of graph frameworks has surged, majority of them are for CPU systems while SIMD-X is for GPU accelerators that come with mounting programming challenges. In order to use GPUs efficiently, a programmer needs to possess an in-depth knowledge of GPU architecture [16, 1], e.g., Gunrock requires explicit management of GPU threads and memory [76], and B40C [50] and Enterprise [41] need thousands of lines of CUDA code for BFS specific optimizations. One of the goals of this work is to provide a simple programming model and delegate the responsibility of task management to SIMD-X. Second, current systems either ignore workload imbalance as in [33, 91], or resolve it reactively as in [76, 72], both of which result in undesired system performance. Lastly, because GPUs lack support for global synchronization, existing systems [73, 76, 41, 43, 69] either rely on the multi-kernel design or runtime tunning, both of which come with considerable overhead, especially for graph algorithms with high iteration count. SIMD-X addresses these challenges with new filters, and a deadlock-free software barrier.

2 SIMD-X Challenges and Architecture

2.1 Graph Computing on GPUs

Generally speaking, regular applications present uniform workload distribution across the data set. As a result, such applications lend themselves to the data-parallel GPU architecture. For development and evaluation, this work mainly uses NVIDIA GPUs, which have tens of streaming processors and in total thousands of Compute Unified Device Architecture (CUDA) cores [1, 56]. Typically, a *warp* of 32 threads execute the same instruction in parallel on consecutive data.

On the other hand, task management for irregular applications is challenging on GPUs. In this work, we focus on a number of graph algorithms such as breadth-first search, k-core, and belief propagation. Here we use one algorithm – Single Source Shortest Path (SSSP) – to illustrates the challenges. Simply put, a graph algorithm computes on a graph $G = (V, E, w)$, where V, E and w are the sets of vertices, edges, and edge weights. The computation updates the algorithmic metadata which are the states of vertices or edges in an iterative manner. A typical workflow of SSSP is shown in Figure 1. Initially, SSSP assigns the infinite distance to each vertex in the *distance array*, which is represented as blank in the figure. Assuming the source vertex is a, the algorithm assigns 0 as its initial distance, and now vertex a becomes active. Next, SSSP computes on this vertex, that is, calculating the updates for all the neighbors of vertex a. In this case, vertices $\{b, d\}$ have their distances updated to 5 and 1 in the distance array. At the next iteration, the vertices with newly updated distances become active and perform the same computation again. This process continues until no vertex gets updated. Different from breadth-first search, SSSP may update the distances of some vertices across multiple iterations, e.g., vertex b is updated in iteration 1 and 3.

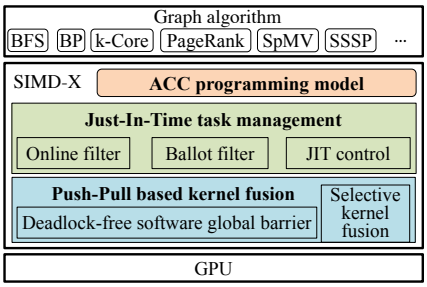


Figure 2: SIMD-X architecture

In this example, not every vertex is active at all time, and vertices with different degrees (number of edges) yield varying amounts of workloads. For instance, at iteration 3 of Figure 1(d), one thread working on vertex *c* computes two neighbors, while another thread on vertex *e* four neighbors.

2.2 Architecture

SIMD-X is motivated to achieve two goals simultaneously: providing ease of programming for a large variety of graph algorithms, whereas enabling fine-grained optimization of GPU resources at the runtime. Figure 2 presents an overview of SIMD-X architecture. To achieve the first goal, SIMD-X utilizes a simple yet powerful Active-Compute-Combine (ACC) model. This data-parallel API allows a programmer to implement graph algorithms with tens of lines of code (LOC). Prior work requires significant programming effort [50, 41, 76], or runs the risk of poor performance [33].

In SIMD-X, high-performance graph processing on GPUs is achieved through the development of two components: (1) JIT task management, which is responsible for translating data-parallel code to parallel tasks on GPUs. Essentially, SIMD-X "filters" the inactive tasks and groups similar ones to run on the underlying SIMD architecture. In particular, SIMD-X develops online and ballot filters for handling different types of tasks, and dynamically selects the better filter during the execution of the algorithm. And (2) Pull-push based kernel fusion. Graph applications are iterative in nature and thus require synchronizations. Fusing kernels across iterations would yield indispensable benefits, because kernel launching at each iteration incurs non-trivial overhead. In SIMD-X, we observe that with aggressive kernel fusion, register consumption would increase dramatically, lowering the occupancy and thus performance. To this end, SIMD-X deploys kernel fusion around pull and push stages of each graph computation, seeking a sweet spot that not only maximizes the range of each kernel fusion but also minimizes the register consumption. It is worthy noting that we also address the deadlock issue faced by software global barrier in SIMD-X.

3 ACC Programming Model

The novelty of SIMD-X lies at achieving both ease of programming and efficient workload scheduling, which is *especially hard on GPUs*. When it comes to graph computing, there are two main programming models: vertex-centric vs. edge-centric. Vertex-centric model, also referred to as "Think like a vertex" [49, 90] focuses on active vertices in a graph, whereas the latter one [61, 60] iterates on edges and simplifies programming.

3.1 Motivation

Graph programming converges to either *vertex-centric* or *edge-centric* models. In particular, the vertex-centric model contains two functions: `vertex_scatter` defines what operations should be done on this vertex, and `vertex_gather` applies the updates on the vertex. This model has been adopted by a number of existing projects, e.g., Pregel [49], GraphLab [45], PowerGraph [18], GraphChi [39], FlashGraph [90], Mosaic [47], and GridGraph [92], as well as GPU-based implementation such as CuSha [33] and Gunrock [76]. On the other hand, the edge-centric model is initially introduced by the external-memory graph engine X-stream [61] to improve IO performance. It requires a programmer to define two functions needed on each edge, `edge_scatter` and `edge_gather`. As such, this model schedules threads by the edge count. Particularly, one thread needs to send the information of the source vertex and the outbound edge to the destination vertex (`edge_scatter`), which atomically applies the new updates in `edge_gather`.

In this work, we believe the many-threaded nature of GPU architecture demands a new abstraction. We intend to exploit various thread scheduling options to better tackle workload imbalance [41, 77], while minimizing the overhead with regards to atomic operations on GPUs [46]. Table 1 summarizes the designs of recent GPU-based graph analytics systems. To avoid wasting the threads to compute on inactive vertices, *task filtering* is essential in generating a list of active vertices. Once task lists are ready, *workload imbalance* caused by skewed degree distribution in many graphs becomes the next concern. Since handling this issue in a vertex centric model involves nontrivial programming efforts [41], edge-based computing presents a desirable alternative. However, traditional edge-centric approach would result in atomic updates at the destination vertex, thus a proper schedule before applying the update is essential to *avoid atomic operation*. It is also important to note that *compressed sparse row* (CSR) is a preferable graph format which can save around 50% of the space over edge list format, as contemporary GPUs only feature tens of GB memory [1]. The proposed ACC framework is designed to address these three challenges.

Table 1: Comparison between ACC and relevant GPU-based programming models. ■ denotes desirable feature.

Abstraction	Related Work	Stages			Graph format
		Task filtering	Workload balancing	Avoid atomic operation	
ICU	CuSha [33], Lux [30]		Init/Compute (Edge)	Update	Edge list
ICRU	WS [32]		Init/Compute (Edge)	Reduce/IsUpdate	CSR
AFC	Gunrock [77]	Advance/Filter	Compute (Vertex, with atomic update)		CSR
GAS	GTS [34], GraphReduce [62]		Gather (Edge)	Apply/Scatter	Edge list
ACC	SIMD-X	Active	Compute (Edge)	Combine	CSR

3.2 ACC Model

The new ACC model contains three functions: **A**ctive, **C**ompute, and **C**ombine. ACC supports a wide range of graph algorithms and requires much fewer lines of code compared to prior work. In this following, we will discuss the three functions.

Active allows a programmer to specify the condition whether a vertex is active. Formally it can be defined:

$$\exists_v \leftarrow active(M_v, v)$$

where v is the vertex ID and M_v represents its metadata. Depending on the algorithm, the Active function may vary. Belief propagation (BP) is simple which treats all vertices as active. In comparison, SSSP, as shown in Figure 3(a), considers the vertices active when their current metadata differs from the prior iteration.

Simply put, SIMD-X distinguishes active vertices from inactive ones, and focuses on the calculation needed for each vertex. This is different from the vertex-centric model which deals with not only the active vertex but also its neighbors. Because two vertices may have different numbers of neighbors, existing systems [49, 18] likely suffer from workload imbalance. To this end, SIMD-X leverages a classification technique, similar to Enterprise [41], to group the active vertices depending on the expected workload.

Compute defines the computation that happens on each edge. In particular, it specifies the operations on the metadata of edge $(v,\ u)$ and two vertices v and u, which can be written as follows:

$$update_{v\rightarrow u} \leftarrow compute(M_v, M_{(v,u)}, M_u)$$

where the return value of $update_{v\rightarrow u}$ will be used by the Combine function. For example, SSSP can be defined as shown in Figure 3(a).

Combine merges all the updates, once the computations are completed. It can be represented:

$$update_u \leftarrow \bigoplus_{v\in Nbr[u]} update_{v\rightarrow u}$$

where $\oplus$ must be commutative and associative, e.g., sum and minimum, and is being applied to all the neighbors of vertex u. Figure 3(a) presents the Combine examples of SSSP. Particularly, BP summarizes all updates, where SSSP combines all updates from compute by selecting the minimum.

SIMD-X optimizes two types of combine operations, i.e., aggregation and voting. Particularly, aggregation cannot tolerate overwrites, that is, all updates are needed to arrive at the correct results. PageRank, SSSP and k-Core are representative examples of such operation. In contrast, voting relaxes this condition, that is, the algorithm is correct as long as one update is received because all updates are identical. For instance, BFS is valid once one parent vertex successfully visited the child vertex. Other algorithms, such as, weakly connected component and strongly connected component algorithms [67] also fall into this category.

3.3 Processing with ACC

This section uses SSSP an example to illustrate how the SIMD-X framework works. SSSP computes the shortest paths between the source vertex and the remaining vertices of the graph. Although similar to Breadth-First Search (BFS), SSSP is more challenging as only one vertex with the shortest distance should be computed at one time. To improve the parallelism, we adopt the delta-step [51] algorithm which permits us to simultaneously compute a collection of the vertices whose distances are relatively shorter. We assume positive edge weights.

As shown in line 12 - 21 of Figure 3(b), SIMD-X structures graph computation as a loop. Similar to popular GPU-based frameworks [77, 33, 32], ACC follows BSP model, that is, synchronization is required at the end of each iteration. As we will discuss in the next section, SIMD-X employ three kernels to balance the workload, Thread, Warp and CTA kernels working on small_list, med_list and large_list, respectively. During computing, the online_filter (Section 4) attempts to track the active vertices with the thread bins (i.e., small_bin, med_bin and large_bin). Note that each active vertex is stored in one of these three bins based upon its degree. After a deadlock free software global barrier (Section 5), SIMD-X checks whether an overflow happens in any of the thread bins, which leads to either a ballot filter-based active lists generation or a simple prefix-scan based concatenation of all thread bins to produce the active lists (line 17-21).

In Figure 3(b), Line 1 - 8 exemplifies the interactions between ACC and SIMD-X. Firstly, SIMD-X will schedule a warp of threads to work on the neighbors of one active vertex from med_list. Similarly, Thread and CTA will schedule a thread and CTA to work on each active vertex from small_list and large_list, respectively. During

```
1: Init (src){
2:     dist_curr [src] = 0;
3:     large_list.insert (src);
   }
4: Active (v){
5:     return dist_curr [v] != dist_prev [v];
   }
6: Compute (edge, weight){
7:     old_dist = dist_curr [edge.dest];
8:     new_dist = dist_curr [edge.src] + w;
9:     return old_dist > new_dist ? new_dist: old_dist;
   }
10: Combine (dist[]){
11:     return min (dist[]);
   }
```

(a) SSSP in ACC

```
1: Warp (med_list, Compute, Combine, Active, overflow)
2:     for each active vertex v in med_list: //warp in parallel
       //Intra-warp parallel reduction.
       //Splitting compute and combine to avoid atomic operation.
3:     for each neighboring edge set edge[32] to vertex v:
4:         res [lane_id] = Compute ( edge[lane_id] );
5:     final = Combine (res[0 - 31]);
6:     if lane_id == 0:
7:         metadata_curr[v] = final;
8:         small_bin, med_bin, large_bin =
                    online_filter (Active, v, overflow);

   //Similar to Warp
   Thread(){//One thread working on one active vertex}
   CTA(){//One CTA working on one active vertex}

10: SSSP_main{
11:     Init (src);
12:     while conditions:
        //Thread assignment (Workload balancing step II)
13:     Thread (small_list, Compute, Combine, Active, overflow);
14:     Warp (med_list, Compute, Combine, Active, overflow);
15:     CTA (large_list, Compute, Combine, Active, overflow);
16:     __software_global_barrier ();

        //Task management (Workload balancing step I)
17:     if (overflow):
18:       ballot_filer (small_list, med_list, large_list, Active);
19:     else:
20:       small_list, med_list, large_list = concatenate
                              (small_bin, med_bin, large_bin);
21:     __software_global_barrier ();
   }
```

(b) ACC in SIMD-X

Figure 3: (a) SSSP in ACC model and (b) ACC abstraction and task management within SIMD-X framework.

computation, each thread will conduct a local Compute and Combine at line 4. Once finished, a cross Warp Combine happens at line 5. Eventually, the first thread from the Warp applies the final updates (without atomic operation) and store this vertex (if active) into corresponding thread bins.

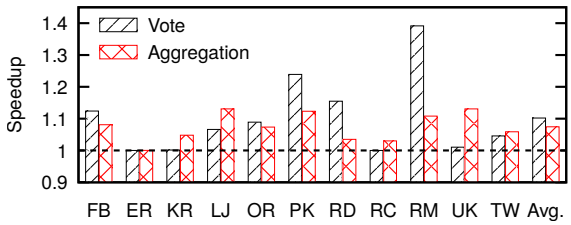


Figure 4: Speedup of our ACC model over Gunrock. Note vote and aggregation operations are materialized by BFS and SSSP algorithms, respectively, and x-axis contains the graph datasets which are defined in Table 3.

Comparison Figure 4 studies the performance impact of ACC vs. Gunrock. The new ACC model follows a computation then combine approach which pays the extra overhead (i.e., assembling all updates residing in shared memory from participating threads) in order to achieve the benefits of atomic-free updates. Gunrock, in contrast, directly applies the update to vertex status with atomic operations, thereby avoids inter-thread communication but experiences heavier overhead from atomic operation. One can see that ACC is, on average, 12% and 9% faster on vote and aggregation operations, respectively. For vote, the speedup comes from that ACC can schedule all threads to collaboratively determine early termination, which is not possible in Gunrock. Aggregation gains the performance from the elimination of atomic updates.

4 Just-In-Time Task Management

Workload balancing is essential for graph applications. The key is to ensure each GPU core, regardless of from which streaming processor, accounts for a similar amount of workload, which is often achieved with the following twin steps. Particularly, in **step I: task management**, the tasks are classified into various lists, namely small_list, med_list and large_list. In **step II: thread assignment**, various granularity of GPU threads are scheduled to work on different worklists. That is, a single thread per small task, a warp per medium task and a CTA per large task. Note, Figure 3(b) presents the pseudo code of step II and the bottom part of Figure 6 paints the corresponding workflow. We refer the readers to Enterprise [41] for more details regarding the landscape of this attempt.

Unlike prior work [41, 77, 50] which places particular efforts at step II, SIMD-X focuses on step I as we find it to be the major culprit that offsets the benefits of workload balancing. In the following, we will first analyze the drawback of existing batch filter method, then describe two new filters, and JIT selection mechanism.

Drawback of batch filter. This approach [76, 50, 11] first loads all the edges of the active vertices to construct an active edge list. Still using the example of SSSP in

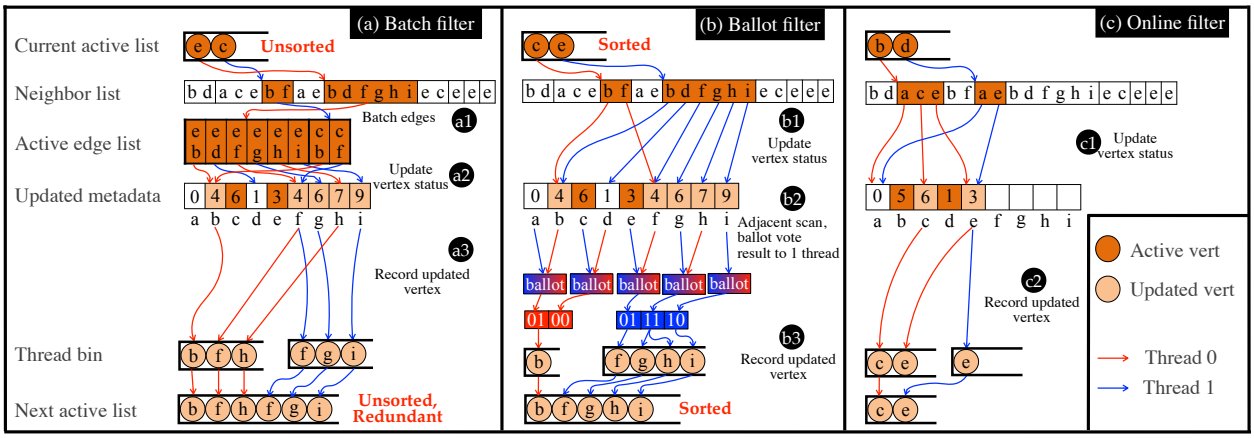


Figure 5: Three task management methods. Particularly, batch filter and ballot filter work on Figure 1(d) to produce a task list for next iteration. Online filter does that for Figure 1(c). Note, we assume the arrow flows of red and blue indicate the execution paths of red and blue threads.

Figure 1(c), this step loads neighbors of vertex {e, c} and constructs the active edge list in a1 of Figure 5 (a). Next, batch filter checks these edges and updates vertex metadata a2, followed by recording the updated vertices in thread bin at step a3. Eventually, batch filter will concatenate these thread bins to arrive at a potentially *unsorted and redundant* next active list – $\{b, f, h, f, g, i\}$. Note, thread private local storage – thread bin – is used to avoid the expensive atomic operations, because multiple threads would need atomic operation to put active vertices directly into next active list.

We observe several drawbacks when using the batch filter for various graph algorithms. First, the active list can consume up to $2{\cdot}|E|$ memory space because majority of the vertices in a graph can become active at one iteration [4, 41], which is especially true for popular social and web graphs. Considering GPU has very limited onboard memory (e.g., 16 GB), this restriction makes large-scale GPU-based graph computing intractable. Second, batch filter produces a worklist with unsorted and redundant active vertices, e.g., next active list – $\{b, f, h, f, g, i\}$ of Figure 5(a), which will lead to poor memory performance for next iteration computation.

Ballot filter is designed to overcome all these shortcomings. It first loads the neighbors of active vertices and immediately updates vertex metadata. As shown at step b1 in Figure 5(b), the neighbors of {e, c} get updated immediately. Afterwards, thread 0 and 1 (red and blue lines) will exploit ballot scan to inspect the updated metadata and record those updated vertices in local thread bin at step b3. The eventual step is similar to batch filter – we concatenate these two thread bins to get the next active list, whereas, with *sorted nonredundant* active vertices.

Ballot scan is the key to comprehend why we arrive at a better next active list. In steps b2 and b3 of Figure 5(b), threads 0 and 1 perform coalesced scan of vertex metadata, and with the CUDA $\_\_ballot()$ primitive, return a bit variable '01' to the first thread. Here 1 means active and 0 otherwise, in this case, vertex a is not active while b is. Through collaboratively working on the entire metadata array, the first thread eventually gets the bit value '0100' for the first four vertices, while the second thread '011110' for the remaining six vertices. Consequently, this approach produces a sorted active list, that is, $\{b, f, g, h, i\}$ in b3.

We intentionally schedule thread 0 and 1 to collaboratively scan the metadata in order to achieve coalesced memory access during scan, as well as, making thread 0 and 1 account for a continuous range of vertices, that is, vertices a - d to thread 0 and e - i to thread 1. This achieves the dual benefits: coalesced scan and sorted active vertices in next active list. Last but not the least, this scheduling lends ballot filter to be many-thread safe.

We also notice an unpublished parallel efforts from Khorasani's dissertation [31] which is closely related to ballot filter. However, his design relies on atomic operation to compute the offsets of active vertices from each Warp in the next active list and subsequently assigns merely a single thread from the Warp to enqueue all these active vertices. This design implies twin disadvantages comparing to ours. First, atomic operation-based offset computation cannot yield sorted active lists. Second, single thread-based active vertices recording tends to be slower than Warp-based one which is our design.

Ballot filter is not without its own issue, especially when the amount of active vertices is low. In that case, scanning the metadata array would account for the majority of the runtime. For instance, in ER and RC graphs, 99.23% and 96.67% of the time is spent on scanning metadata in ballot filter alone solution, respectively.

Online filter is designed to accommodate the issue faced by ballot filter. In the first step, this method loads the ac-

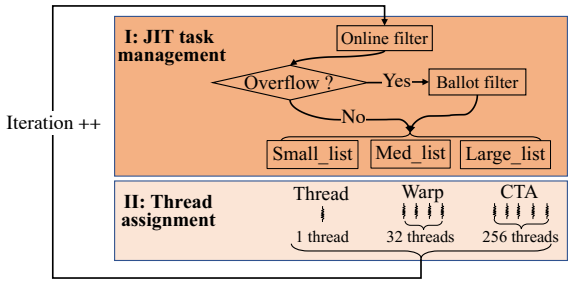


Figure 6: Workload balancing with the essential two steps: the novel JIT task management from SIMD-X and the thread assignment.

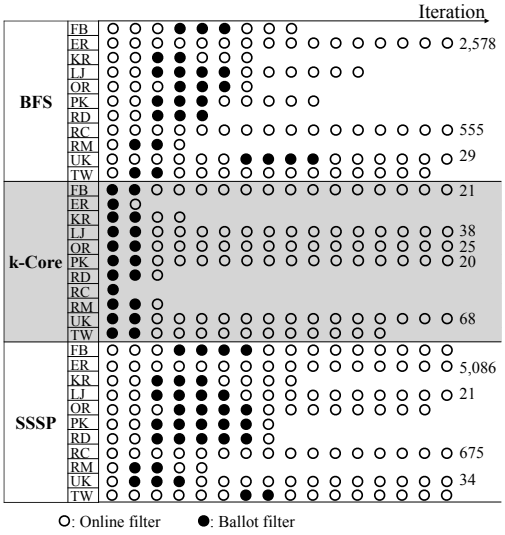
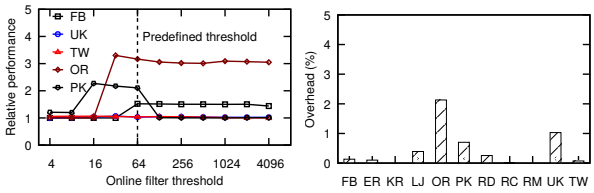


Figure 7: Ballot filter activation patterns.

tive neighbors, updates the destination vertex, and simultaneously records the active vertices in the thread bin. In the last step, it assembles all thread bins together as the next active list. When the number of active vertices is small, this approach turns out to be extremely fast. Here we use the early stage of SSSP as an example to explain its working process. As shown in Figure 5(c), $\{b, d\}$ are active vertices, this approach loads their neighbors for computation (c1), and immediately records the destination vertices. Eventually, it generates $\{e, c\}$ as the active list for the next iteration as shown in c2. It is also important to note that for online filter, the vertices in the active list may become redundant, and out of order.

In graph computing, it is possible that one GPU thread may encounter exceeding amount of active vertices, e.g., our tests on Twitter graph shows one GPU thread can reap more than 4,096 active vertices. Clearly, one cannot afford such a large thread bin for all threads, thus online filter will inevitably suffer from an overflow problem. Fortunately, ballot filter largely avoids this issue because it first updates the metadata of active vertices b2, which, to some extent, averages out the active vertices across threads in step b3.



(a) Relative performance. (b) JIT Overhead

Figure 8: The (a) relative performance of JIT management with respective to various online filter overflow thresholds on BFS and (b) the overhead of JIT on SSSP.

Just-In-Time control adaptively exploits ballot and online filters to retain the best performance. As shown in Figure 6, SIMD-X always activates the online filter first. Once a thread bin overflows, SIMD-X will switch on ballot filter to generate the correct task list for the next iteration. It is also worthy of mentioning that after JIT task management, we assign various granularity of threads to different lists in order to balance workload.

Interestingly, we find out that various algorithms and graph datasets present different selection patterns which tie closely to the amount of workload, that is, the higher volume of workload often results in the activation of ballot filter. As shown in Figure 7, BFS and SSSP typically use the ballot filter in the middle of the computation and online filter at the beginning and end. For high-diameter graphs, BFS and SSSP avoid the use of ballot filter. For instance, ER and RC always use the online filter along 2,578, 555, 5,086 and 675 iterations. k-Core activates the ballot filter at the initial iterations, i.e., typically the first two iterations except RC which only experiences one iteration because all its vertices have < 16 neighbors. At the extreme, BP and PageRank need the ballot filter at exactly the first iteration of computation.

Overflow thresholds for online filter. Clearly, this parameter directly determines when to switch on ballot filter, thereby affects the overall performance. Figure 8(a) presents the normalized performance with respect to various thresholds. As expected, a too low or too high threshold limits the performance because in either case, SIMD-X is forced to switch to ballot filter either too early or too late, leading to performance penalty. As such, in this work we select 64 as the predefined overflow threshold for all algorithms.

Overhead of online filter. After switching to ballot filter, JIT task management also executes the online filter in case it needs to switch back. Figure 8(b) studies the overhead of this design. On average, there is 0.02% slowdown, with the maximum of 2.1% observed for the OR graph. The reason for the small overhead is because online filter only tracks upto 64 (predefined threshold) active vertices for the next iteration and this operation is not on the critical path of the execution.

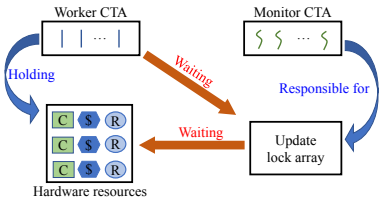


Figure 9: Deadlock in software global barrier, where 'C', '$', and 'R' represent core, L1 cache and register, respectively.

5 Push-Pull Based Kernel Fusion

Kernel fusion [73], a common optimization for a collection of iterative GPU applications, such as graph computing and deep learning [2, 58, 29, 10, 8], reduces expensive overhead of kernel invocation, as well as minimizes the global memory traffic as the life time of registers and shared memory is limited in each kernel. However, traditional efforts, such as Gunrock [77] and Xiao et al [79], fail to achieve cross the global barrier kernel fusion. This section starts with our observation and analysis of potential deadlock in the mainstream global barrier design [79, 82] and subsequently introduces a light-weighted deadlock free solution which enables the global thread synchronization within the fused kernel. However, aggressive kernel fusion requires a large amount of the registers and thus supports fewer parallel warps which could hurt the overall performance. To this end, we introduce a push-pull based kernel fusion to minimize the kernel invocation times and register consumption.

Software global barrier is needed to enable the balanced kernel fusion. Generally speaking, this approach uses an array – *lock* – to synchronize all GPU threads upon arrival and departure. During the processing, it assumes the thread CTA as the monitor while the remaining threads as workers. At arrival, each worker CTA updates its own *status* in *lock*. Once all worker CTAs have arrived, the monitor changes the *statuses* of all CTAs to *departure*, allowing all threads to proceed forward.

This approach, unfortunately, suffers from potential deadlock [79], as illustrated in Figure 9. Specifically, the worker thread CTAs may hold all GPU hardware resources, such as streaming processors, registers and shared memory, while waiting for the monitor to update the *lock* array. In the meantime, the monitor cannot update the *lock* array, due to lack of hardware resources (e.g., thread over subscription).

Compiler-based deadlock free barrier. SIMD-X utilizes the barrier in a way to ensure that every CTA, regardless of a work or the monitor, can obtain hardware resources when needed. This is achieved through comparing the resources needed by the kernels, against the total available resources. Based on the GPU architecture, we can obtain the total amount of registers (*#registerPerSMX*) that can be provided by each streaming processor, e.g., 65,536 registers of NVIDIA K40 GPUs and 32,768 from K20 GPUs. On the other hand, we can collect the register consumption (*#registerPerThread*) of each kernel at the compilation stage. Putting together, SIMD-X is able to calculate the appropriate thread configuration for kernels.

The number of CTA can be computed as follows:

$$\#CTA = floor(\frac{\#registersPerSMX}{\#registersPerThread \cdot \#threadsPerCTA}) \cdot \#SMX \quad (1)$$

where *#threadsPerCTA* is configured by a user, i.e., 128 by default. For example, when deploying a kernel, each thread consumes 110 registers, and on K40 that contains 15 SMXs, each of which contains 65,536 registers. If *#threadsPerCTA* is set to 128, one gets $\#CTA = ceil(\frac{65536}{110\times128}) \times 15 = 60$. As a result, we can configure this kernel as CTA and thread count per CTA as 60 and 128, respectively.

Notably, portable Inter-Block Barrier [69] is closely relevant to our effort. However, this method proposes extremely complicated thread block management mechanism that requires to distinguish whether one thread block will execute useful workloads or not during runtime. This requires nontrivial programmer efforts and scheduling overhead. In comparison, our method achieves this deadlock-free configuration before runtime and is completely transparent to the end users.

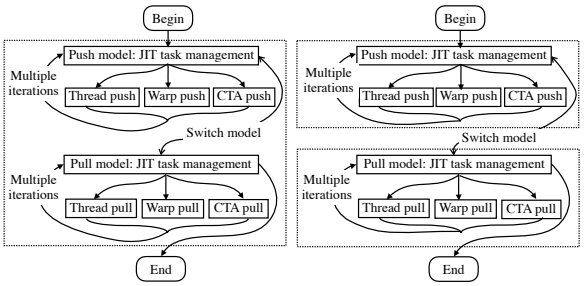
Push-Pull based kernel fusion. As shown in Table 2, the register consumption (using the compilation flag *-Xptxas -v*) increases from average 25 to 110, that is a 4.4× difference. Note, consuming too many registers will curb the number of active threads (according to equation 1). Unfortunately, majority of the graph algorithms are data intensive, thus prefer a higher volume of active threads because more active threads can better hide the frequent memory access stalls caused by data intensive applications. Consequently, we need a balanced fusion strategy that keeps both register consumption and kernel invocation low.

To this end, SIMD-X leverages the push-pull model used in the graph algorithms. That is, such algorithms often use push or pull based computing in several consecutive iterations. Lines 12 - 21 from Figure 3(b), for example, discuss the pull model and we can fuse these lines into a single GPU kernel. Similarly, push model can also be fused into a single kernel. Section 6 details how pull/push iterations occur in various graph algorithms.

SIMD-X adopts the pull-push model as in [66, 4, 41], by controlling where (in/out edge) Compute happens and how to Combine the results and apply (in atomic or atomic free manner). Particularly, in the push model, SIMD-X conducts Compute on the out neighbors of each active vertex, and relies on atomic operations to apply the

Table 2: Register consumption for various kernels.

Kernel	Push (no fusion)				Pull (no fusion)				Selective fusion		All fusion
	Thread	Warp	CTA	Task mgt	Thread	Warp	CTA	Task mgt	push	pull	
Register consumption	26	27	28	24	24	24	22	30	48	50	110
Kernel launching count	up to 40,688								3		1



(a) All fusion (b) Selective fusion

Figure 10: Consecutive iterations from graph algorithms often cluster to push and model computation separately: (a) all fusion, (b) selective fusion.

updates to the destination vertices. In contrast, the pull model schedules Compute on the in neighbors of active vertices, and uses atomic-free strategy to Combine all updates and apply to the destination vertices. As different iterations favor one model over the other, we follow a similar rule as in Ligra [66] to alternate between the push and pull models. That is, when the workload on the push model works on more than 30% of the edges, SIMD-X will switch to pull model.

The idea of push-pull based kernel fusion is to fuse kernels around the pull and push computing. In other words, for the push-based iterations, SIMD-X fuses different compute kernels (for thread, warp, CTA), as well as task management kernel, into one push kernel. The kernel only terminates when the computation finishes or it needs to switch to pull computing according to the criterion discussed in Section 3.3. Similar optimizations are done for the pull-based iterations.

Using the new push-pull based fusion, the register consumption decreases to 48 and 55 thus increases the configurable thread count by 50%. Table 2 presents the register consumption and kernel invocation of different kernel fusion techniques. By using the push-pull based kernel fusion, the kernel relaunch is merely three while its register consumption is cut by half.

6 Graph Algorithms and Datasets

In addition to SSSP that is discussed in Section 3.3, this section further presents a variety of algorithms which are implemented on SIMD-X to examine the expressiveness of ACC programming model, and performance impacts of task management and kernel fusion techniques.

BFS [41] traverses a graph level by level. At each level, it loads all neighbors that are connected to vertices visited in the preceding level, inspects their statuses (metadata), and subsequently marks those unvisited neighbors as active for the next iteration. Notably, BFS conducts synchronizations at the end of each level, relies on vote to combine the updates. During the entire process of traversal, BFS typically experiences light workload at the beginning and end of the computation while heavy workload in the middle.

Table 3: Graph Dataset.

Graph Name	Abbrev.	Vertex Count	Edge Count
Facebook	FB	16,777,215	775,824,943
Europe-osm	ER	50,912,018	108,109,319
Kron24	KR	16,777,216	536,870,911
LiveJournal	LJ	4,847,571	136,950,781
Orkut	OR	3,072,626	234,370,165
Pokec	PK	1,632,803	61,245,127
Random	RD	4,000,000	511,999,999
RoadCA-net	RC	1,971,281	5,533,213
R-MAT	RM	3,999,983	511,999,999
UK-2002	UK	18,520,343	596,227,523
Twitter	TW	25,165,811	787,169,139

Belief propagation (BP), also known as sum-product message passing algorithm, infers the posterior probability of each event based on the likelihoods and prior probabilities of all related events. Once modeled as a graph (Bayesian network or Markov random fields), each event becomes a vertex with all incoming vertices and edges as related events and corresponding likelihoods. In BP, vertex possibility is the metadata.

***k*-Core** (KC), which is widely used in graph visualization application [42, 53], iteratively deletes the vertices whose degree is less than k until all remaining vertices in this graph possess more than k neighbors. k-Core experiences large volume of workloads at initial iterations and follows with light workloads. This work uses a default value of $k = 16$.

PageRank (PR) [57] updates the rank value of one vertex based on the contribution of all in-neighbors iteratively till all vertices have stable rank values. Because the contributions of in neighbors are summarized to the destination vertex, we start PageRank with the pull model and *agg_sum* as the merge operation. At the end of PageRank, we switch to the push model because the majority of the vertices are stable [87]. The switch is decided by a decision tree.

Graph Benchmarks. We evaluate on a wide range of graphs as shown in Table 3, which falls into four types, i.e., social networks, road maps, hyperlink web and synthetic graphs. Particularly, Facebook [17], LiveJournal [68], Orkut [68], Pokec [68], and Twitter [38] are common social networks. Europe-osm [12] and

RoadCA-net [70] are two large roadmap graphs, and UK-2002 [70] is a web graph. Furthermore, we use Graph500 generator to generate Kron24 [6], and GTgraph [19] for R-MAT and random graphs. Europe-osm and RoadCA-net are high diameter graphs, with 2570 and 555 as their diameters, respectively. LiveJournal, Pokec, Twitter and UK-2002 are medium diameter graphs, i.e., 10 - 30 as their diameters. The diameters of the remaining graphs are all smaller than 10. For graphs without edge weight, we use a random generator to generate one weight for each edge similar to Gunrock [76]. These graphs are stored in *compressed sparse row* (CSR) format.

7 Experiments

We implement SIMD-X[1] with 5,660 lines of CUDA and C++ code. All the algorithms presented in Section 6 are implemented with around 100 lines of C++ code. The source code is compiled by GCC 4.8.5 and NVIDIA nvcc 7.5 with the optimization flag as O3. In this work, we evaluate SIMD-X on a Linux workstation with two Intel Xeon E5-2683 CPUs (14 physical cores with 28 hyperthreads), and 512GB main memory. Throughout the evaluation, we use uint32 as the vertex ID and uint64 as index and evaluate our system on NVIDIA K40 GPUs unless otherwise is specified. We also test SIMD-X on earlier K20 and latest P100 GPUs. The timing is started once the graph data is loaded in GPU global memory. Each result is reported with an average of 64 runs.

7.1 Comparison with State-of-the-art

Table 4 summarizes the runtime of SIMD-X against Galois and Gunrock which are state-of-the-art CPU and GPU graph processing systems, respectively, as well as CuSha (GPU) and Ligra (CPU), two popular graph frameworks. The take aways of this table are two folds.

First, SIMD-X is both space efficient and robust. As one can see, since CuSha requires edge list as the input for computation, it cannot accommodate large graphs (e.g., FB and TW) across all algorithms. Besides, since Gunrock requires large amount of space for batch filter, it suffers out of memory (OOM) error for all larger graphs in SSSP. Even CPU systems (Galois and Ligra) enjoys affluent memory space (512 GB) from CPU, they cannot converge to a result for high diameter graphs. That is, Galois cannot converge for SSSP on ER while Ligra fails to obtain result for BFS on UK graph.

Second, SIMD-X outperforms all graph processing frameworks. In general, SIMD-X is 24×, 2.9×, 6.5× and 3.3× faster than CuSha, Gunrock, Galois and Ligra, respectively. In BFS, SIMD-X bests CuSha, Gunrock, Galois and Ligra by 9.6×, 4.8×, 2.1× and 2.4×, respectively. We also notice that SIMD-X is slower than Galois on the RD graph because workload balancing brings negligible benefits to uniform-degree graph (RD). Also, SIMD-X is slightly worse than Ligra on RM graph since this graph only has a diameter of four thus both JIT task management and kernel fusion brings trivial benefits to GPU based graph systems, as evident by much lower performance on CuSha and Gunrock.

In PageRank, SIMD-X achieves 1.2×, 2.1×, 2.3× and 4× speedups over CuSha, Gunrock, Galois and Ligra, respectively. Note, even CuSha cannot support all large graphs due to large memory space consumption, it performs roughly similar to SIMD-X with even outperforming SIMD-X on LJ and OR. This is generally because PageRank tends to be more computation intensive than other graph algorithms and needs to compute all edges, curbing the benefits of task management and kernel fusion. However, edge list format (of CuSha) doubles memory consumption, facing OOM for large graphs.

For SSSP, SIMD-X wins 21×, on average, over all four projects. We project SIMD-X to better outperform all systems than observed for BFS algorithm because SSSP experiences more iterations with larger volume of active tasks, placing more favor towards ACC model, JIT task management and push-pull based kernel fusion. However, because Gunrock fails to accommodate all large graphs, our benefits cannot surface – ending with merely 1.8× speedup. Second, CuSha spends 519,674 ms on the high diameter ER graph which is 480× slower than SIMD-X because task management is absent from CuSha. We also notice Galois performs better than SIMD-X in RD, again, due to the uniform degree distribution.

For k-Core, where $k = 32$, SIMD-X wins Ligra by 20×. Such a striking advantage comes from three parts. First, as reflected by Figure 11(b), k-Core generates extensive amount of workload variations thus benefits tremendously from JIT task management. Second, k-Core's iterative nature also enjoys the benefits from push-pull based kernel fusion, as shown in Figure 12(c). Lastly, the flexibility of ACC allows innovative k-Core algorithm designs – we will stop further subtracting the degree of destination vertex once the destination vertex's degree goes below k – this reduces tremendous unnecessary updates. Note comparisons of Belief Propagation, as well as other systems for k-Core are not included because those systems fail to support such algorithms.

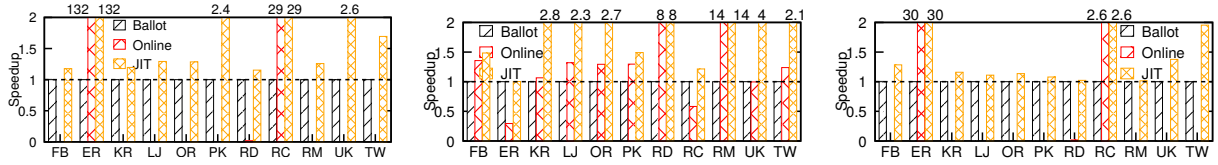
7.2 Benefits of Various Techniques

This section studies the performance impacts brought by JIT task management and push-pull based kernel fusion. As we have presented in Section 4, JIT task management only works for applications that experience work-

[1] SIMD-X source `https://github.com/asherliu/simd-x`.

Table 4: Runtime (ms) of SIMD-X and Gunrock, and Galois. A K40 GPU is used to test SIMD-X and Gunrock, and a CPU with 28 threads for Galois. The **blank space** indicates the test cannot complete for the given algorithm and graph.

Alg	System	FB	ER	KR	LJ	OR	PK	RD	RC	RM	UK	TW	Avg. speedup
BFS	SIMD-X	198	400	130	59	40	20	82	15	47	308	210	-
	CuSha			988	224	341	72	435	297	674	4298		9.6
	Gunrock	685	849	677	71	225	44	647	146	506	312	697	4.8
	Galois	482	1068	140	139	42	34	48	53	65	229	322	2.1
	Ligra	1086	1426	176	89	51	31	88	48	40		496	2.4
PR	SIMD-X	1553	346	1141	236	435	118	1105	13	800	637	1525	-
	CuSha			1704	182	323	180	1402	15	886			1.2
	Gunrock	3004	884	3129	275	927	166	2963	43	2208	784	3180	2.1
	Galois	4552	603	3069	424	1061	218	3576	20	2067	842	4178	2.3
	Ligra	16780	1368	2000	1324	1786	310	809	35	1703		9360	4
SSSP	SIMD-X	1816	1080	998	284	604	143	1505	223	478	703	1344	-
	CuSha		519674	1663	692	1120	260	1610	438	1236			62
	Gunrock		1206	1220	431	1259	336	5059	229				1.8
	Galois	161596		8485	1785	1166	356	747	3440	5877	9081	1818	15
	Ligra	14067	3043	2893	1627	1567	605	3353	301	2783	1300	5217	3.7
k-Core	SIMD-X	366	78	131	60	63	33	10	4	19	151	277	-
	Ligra	6337	1167	2813	1707	1700	654	27	36	235	6627	5783	20



(a) BFS (b) k-Core (c) SSSP

Figure 11: Benefit of just-in-time task management, normalized to the performance of the ballot filter.

load variations, that is, BFS, k-Core and SSSP. On the other hand, push-pull based kernel fusion is applicable for all five algorithms

On average, JIT task management presented in Figure 11, is 16×, 26× and 4.5× faster than the ballot filter for BFS, k-Core and SSSP. As expected, online filter alone cannot work for many graphs, particularly large ones, e.g., Facebook, Twitter and UK2002 graphs in BFS and SSSP. Without considering overflow problem (ER and RC graphs), JIT task management adds a small 1-2% overhead on top of the online filter on BFS and SSSP.

On k-Core, JIT task management is, on average, 28.5× and 5% faster than ballot and online filter, respectively. We also observe that the ballot filter outperforms the online filter on ER and RC graphs by 3.4× and 1.7×, because k-Core removes a large volume of vertices which favors the former to produce a non-redundant and sorted work list.

Push-pull based kernel fusion brings, on average, 43% and 25% improvement over non-fusion and all-fusion across all algorithms and graphs. In particular, push-pull based kernel fusion tops non-fusion by 74%, 11%, 85%, 10% and 66% on BFS, BP, k-Core, PageRank and SSSP. BFS, k-Core and SSSP achieves more performance gains because they are not computation intensive and tend to run a higher number of iterations. For all fusion, our new kernel fusion is 55%, 6%, 62%, 25% and 11% faster on BFS, BP, k-Core, PageRank and SSSP. It is important to note that all fusion is not always beneficial, i.e., all fuse option of PageRank is average 13% slower than no fusion because all fusion limits the amount of configurable threads. However, for memory intensive applications, like BFS and SSSP on ER and RC, all fusion is on average 2× better.

7.3 Performance on Different GPUs

We also evaluate SIMD-X, Gunrock and CuSha on various GPU models, such as K20 and P100 GPUs. It is not surprising to see tht SIMD-X presents the biggest performance gain on the latest GPUs. In detail, SIMD-X on K40 and P100 performs 1.7× and 5.1× better than K20 GPU. In contrast, Gunrock merely gets 1.1× and 1.7× performance improvement when moving from K20 to K40 and P100, respectively. Similarly for CuSha, its performance on K40 and P100 are 1.2× and 3.5× better than K20, respectively. The root cause of this disparity is that SIMD-X's kernel fusion technique can dynamically configure its GPU kernels to the fitting thread count on the corresponding hardware so as to achieve the peak performance. For instance, the thread count increases by 1.2× and 5.1× on K40 and P100 than on K20 for BFS.

8 Related Work

Recent advance in graph computing falls in algorithm innovation [51, 87, 15], framework developments [49, 18, 45, 39, 42, 90, 92, 22, 66, 63, 61, 23, 54, 60, 74, 7, 80, 84, 65, 88, 75, 55, 89, 86, 85, 3, 78, 52, 21, 9, 81] and accel-

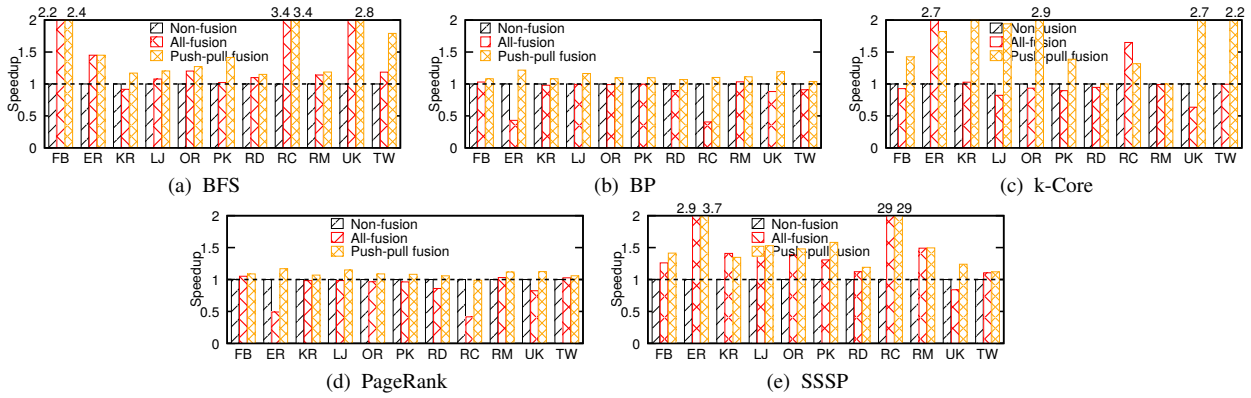


Figure 12: Benefit of push-pull based kernel fusion, normalized to the performance of no fusion.

erator optimizations [76, 41, 50, 33, 43, 59, 64, 13]. This section covers relevant work from three aspects: programming model, task management and kernel fusion.

Recently, we witness an array of graph analytical models. For instance, "think like a graph" [71] requires each vertex to obtain the view of the entire partition on one machine in order to minimize the communication cost. Furthermore, domain specific programming language systems, such as Galois [54], Green-Marl [23] and Trinity [63], allow programmers to write single-threaded source code while enjoying multi-threaded processing. In comparison, SIMD-X decouples the goal of programming simplicity and performance: with ACC, SIMD-X ultimately designs a data-parallel abstraction for deploying irregular graph applications on GPU. With JIT task management and push-pull based kernel fusion, SIMD-X is an order of magnitude faster than state-of-the-art CPU and GPU frameworks.

Task management is an important optimization for GPU-based graph computing. Besides batch filter [76, 50], there also exist other task management approaches – strided filter [41, 43] and atomic filter [46]. Particularly, strided filter resembles ballot filter but the former one experiences strided memory access when scanning the metadata thus performs up to 16× worse than ballot filter. Atomic filter relies is similar to online filter but it relies on atomic operation to put active vertices into global active list which suffers from orders of magnitude slow down than online filter. Besides ballot and online filter bests batch, stride and atomic filter, SIMD-X goes further via introducing a JIT controller to adaptively use online filter and ballot filter to further improve the performance. We also find that JIT task management can be exploited to help manage active lists for other applications such as warp segmentation [32] and CSR5 [44].

Kernel fusion affects applications far beyond graph computations. SIMD-X is closely related to global software barrier [79, 82]. However, previous work fails to identify the deadlock issue in this global software barrier problem, thus no solution towards this issue. In contrast, SIMD-X unveils, systematically analyzes, and resolves this problem. To avoid high register consumption, SIMD-X further selectively fuse kernels via exploiting the special kernel launching patterns of graph algorithms. It is also important to mention existing work [73] that only fuse kernels to barrier boundary. In comparison, SIMD-X fuses kernels across barriers. Our design can also benefit the popular Persistent Kernel [20] designs which have been found suffer from deadlock issues when the occupancy exceed an unknown bound [48, 24].

9 Conclusion

In this work, we propose SIMD-X, a parallel graph computing framework that supports programming and processing of *single instruction multiple, complex, data* on GPUs. Specifically, the Active-Compute-Combine (ACC) model provides ease of programming to programmers, while just-in-time task management and push-pull based kernel fusion leverage the opportunities for system-level optimization. Using SIMD-X, a user can program a graph algorithm in tens of lines of code, while achieving significant speedup over the state-of-the-art.

Acknowledgment

The authors would like to thank the anonymous reviewers and Shepherd Chris Rossbach for their feedback and suggestions. Hang Liu did part of this work at the George Washington University. This work was partially supported by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774 at George Washington University and CRII Award No. 1850274 at University of Massachusetts Lowell. We also would like to gracefully acknowledge the support from XSEDE supercomputers and Amazon AWS, as well as the NVIDIA Corporation for the donation of the Titan Xp and Quadro P6000 GPUs to the University of Massachusetts Lowell.

References

[1] Nvidia cuda c programming guide. *NVIDIA Corporation*, 2011.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[3] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, 2017.

[4] S Beamer, K Asanovic, and D Patterson. Direction-optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10. IEEE, 2012.

[5] Bibek Bhattarai, Hang Liu, and H Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, volume 19, 2019.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, 2004.

[7] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 215–226. ACM, 2014.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[9] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

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

[12] European Open Stream Map. `http://download.geofabrik.de/europe-latest.osm.bz2`,.

[13] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. Dr. BFS: Data Centric Breadth-First Search on FPGAs. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 208. ACM, 2019.

[14] Anil Gaihre, Yan Luo, and Hang Liu. Do Bitcoin Users Really Care About Anonymity? An Analysis of the Bitcoin Transaction Graph. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1198–1207. IEEE, 2018.

[15] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the international symposium on High-performance parallel and distributed computing (HPDC)*. ACM, 2019.

[16] Benedict R Gaster and Lee Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 2012.

[17] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. Practical Recommendations on Crawling Online Social Networks. *IEEE Journal on Selected Areas in Communications*, 2011.

[18] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.

[19] GTgraph: A suite of synthetic random graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/`.

[20] Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[21] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[22] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of international conference on Knowledge discovery and data mining (SIGKDD)*, pages 77–85, 2013.

[23] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 40, pages 349–362, 2012.

[24] Derek R Hower, Blake A Hechtman, Bradford M Beckmann, Benedict R Gaster, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous-race-free memory models. *ACM SIGARCH Computer Architecture News*, 42(1):427–440, 2014.

[25] Yang Hu, Hang Liu, and H Howie Huang. High-Performance Triangle Counting on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2018.

[26] Yang Hu, Hang Liu, and H Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[27] H Howie Huang and Hang Liu. Big data machine learning and graph analytics: Current state and future challenges. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 16–17. IEEE, 2014.

[28] Yuede Ji, Hang Liu, and H Howie Huang. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 731–742. IEEE, 2018.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[30] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A Distributed Multi-GPU System for Fast Graph Processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[31] Farzad Khorasani. *High Performance Vertex-Centric Graph Analytics on GPUs*. PhD Dissertation: University of California, Riverside, 2016.

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

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

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

[35] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 71. IEEE Press, 2016.

[36] Pradeep Kumar and H Howie Huang. Falcon: scaling IO performance in multi-SSD volumes. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, pages 41–53. USENIX Association, 2017.

[37] Pradeep Kumar and H Howie Huang. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, 2019.

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[39] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 31–46. USENIX Association, 2012.

[40] Hang Liu and H Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300. USENIX Association.

[41] Hang Liu and H. Howie Huang. Enterprise: Breadth-First Graph Traversal on GPU Servers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[42] Hang Liu and H. Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017.

[43] Hang Liu, H Howie Huang, and Yang Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[44] Weifeng Liu and Brian Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[45] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. 2010.

[46] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[47] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[48] Sepideh Maleki, Annie Yang, and Martin Burtscher. *Higher-order and tuple-based massively-parallel prefix sums*, volume 51. ACM, 2016.

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

[50] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *PPoPP*, 2012.

[51] Ulrich Meyer and Peter Sanders. Δ-Stepping: A Parallel Single Source Shortest Path Algorithm. *Algorithms—ESA'98*, 1998.

[52] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 2015.

[53] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

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

[55] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–636. ACM, 2018.

[56] Nvidia. NVIDIA Kepler GK110 Architecture Whitepaper. 2013.

[57] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[59] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of USENIX conference on Annual Technical Conference*. USENIX Association, 2012.

[60] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424. ACM, 2015.

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[62] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: processing large-scale graphs on accelerator-based systems. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[63] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 505–516, 2013.

[64] Zachary Sherer, Eric Finnerty, Yan Luo, and Hang Liu. Software Hardware Co-Optimized BFS on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 190–190. ACM, 2019.

[65] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 16)*, pages 317–332.

[66] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[67] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[68] SNAP: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/`.

[69] Tyler Sorensen, Alastair F Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Portable inter-workgroup barrier synchronisation for GPUs. In *ACM SIGPLAN Notices*, volume 51, pages 39–58. ACM, 2016.

[70] The University of Florida: Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[71] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From Think Like a Vertex to Think Like a Graph. *Proceedings of the VLDB Endowment*, 2013.

[72] Stanley Tzeng, Anjul Patney, and John D Owens. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010.

[73] Mohamed Wahib and Naoya Maruyama. Scalable Kernel Fusion for Memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014.

[74] Kai Wang and Zhendong Su. GraphQ: Graph Query Processing with Abstraction Refinement-Scalable and Programmable Analytics over Very Large Graphs on a Single PC.

[75] Siyuan Wang, Chang Lou Lou, Rong Chen, and Haibo Chen. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[76] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 265–266. ACM, 2015.

[77] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: GPU Graph Analytics. *arXiv preprint arXiv:1701.01170*, 2017.

[78] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. G ra M: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.

[79] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010.

[80] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *ACM SIGPLAN Notices (PPoPP)*, volume 50, pages 194–204. ACM, 2015.

[81] Da Yan and Hang Liu. Parallel graph processing. *Encyclopedia of Big Data Technologies*, pages 1–8, 2018.

[82] Shengen Yan, Guoping Long, and Yunquan Zhang. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *PPoPP*, 2013.

[83] Jialing Zhang, Xiaoyan Zhuo, Aekyeung Moon, Hang Liu, and Seung Woo Son. Efficient Encoding and Reconstruction of HPC Datasets for Checkpoint/Restart. In *IEEE Symposium on Mass Storage Systems and Technologies*, 2019.

[84] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. *ACM SIGPLAN Notices (PPoPP)*, 50(8):183–193, 2015.

[85] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the Hidden Dimension in Graph Processing. In *OSDI*, pages 285–300, 2016.

[86] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 608–621. ACM, 2018.

[87] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[88] Yunhao Zhang, Rong Chen, and Haibo Chen. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 614–630. ACM, 2017.

[89] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data),*, pages 293–302. IEEE, 2017.

[90] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 45–58. USENIX Association, 2015.

[91] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1543–1552, 2014.

[92] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386. USENIX Association, 2015.