Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication

Lingxiao Ma, Zhi Yang, and Han Chen, *Computer Science Department, Peking University, Beijing, China;* **Jilong Xue,** *Microsoft Research, Beijing, China;* **Yafei Dai,** *Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications, School of Electronics and Computer Engineering (SECE), Peking University, Shenzhen, China*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/ma

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

Garaph: Efficient *G*PU-*a*ccelerated G*raph* Processing on a Single Machine with Balanced Replication

Lingxiao Ma[§#], Zhi Yang[§#*], Han Chen[§], Jilong Xue[†] and Yafei Dai[‡]
[§]*Computer Science Department, Peking University, Beijing, China*
[†]*Microsoft Research, Beijing, China*
[‡]*Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications, Peking University, China*

Abstract

Recent advances in storage (e.g., DDR4, SSD, NVM) and accelerators (e.g., GPU, Xeon-Phi, FPGA) provide the opportunity to efficiently process large-scale graphs on a single machine. In this paper, we present Garaph, a GPU-accelerated graph processing system on a single machine with secondary storage as memory extension. Garaph is novel in three ways. First, Garaph proposes a vertex replication degree customization scheme that maximizes the GPU utilization given vertices' degrees and space constraints. Second, Garaph adopts a balanced edge-based partition ensuring work balance over CPU threads, and also a hybrid of notify-pull and pull computation models optimized for fast graph processing on the CPU. Third, Garaph uses a dynamic workload assignment scheme which takes into account both characteristics of processing elements and graph algorithms. Our evaluation with six widely used graph applications on seven real-world graphs shows that Garaph significantly outperforms existing state-of-art CPU-based and GPU-based graph processing systems, getting up to 5.36x speedup over the fastest among them.

1 Introduction

Triggered by the availability of graph-structured data in domains ranging from social networks to genomics and business, the need for efficient large scale graph processing has grown. The resulting demand has driven the development of many distributed systems, including Pregel [22], Giraph [2], GraphX [11], GraphLab [21], PowerGraph [10], PowerLyra [7] and Gemini [36]. These systems attempt to scale to graphs of billions of edges by distributing the computation over multiple cluster nodes. However, the performance of existing graph frameworks relies on effective partitioning to minimize communication, which is very difficult for natural graphs [1, 20, 10]. Therefore, network performance required for communication between graph partitions emerges as the bottleneck, and thus distributed graph systems require very fast networks to realize good performance.

As an alternative, several non-distributed graph processing systems have been proposed. Galois [27] and Ligra [32] are specific for shared-memory/multi-core machines, whereas GraphChi [17], X-Stream [29] and GridGraph [37] are designed for processing large graphs on a single machine, by relying on secondary storage. Such solutions no longer require the resources of very large clusters, and users need not to be skilled at managing and tuning a distributed system in a cluster.

But the large amount of data to be processed in a single machine put pressure on two scarce resources: memory and computing power. We observe, however, that today more efficient non-distributed solutions are affordable. On the one hand, current commodity single unit servers can easily aggregate hundreds of GBs to TBs of RAM [32]. Further, with recent advances of secondary storage such as SATA/PCIe-based solid-state drive (SSD) and non-volatile memory (NVM), it is feasible to aggregate multiple secondary storages to achieve a high access bandwidth close to memory. On the other hand, current GPUs have much higher massive parallelism and memory access bandwidth than traditional CPUs, which has the potential to offer high-performance graph processing.

Given these recent advances, GPU-accelerated, secondary-storage based graph processing has the potential to offer a viable solution. However, while several attempts [8, 34] have been made recently, efficient large-scale graph computation on CPU/GPU hybrid platforms still remains a challenge due to the highly skewed degree distribution of the natural graphs and heterogeneous parallelism of CPU and GPU. In particular, the skewed degree distribution implies that a small fraction of the vertices are adjacent to a large fraction of the edges. This

*Corresponding author (yangzhi@pku.edu.cn)
#These authors contributed equally to this work.

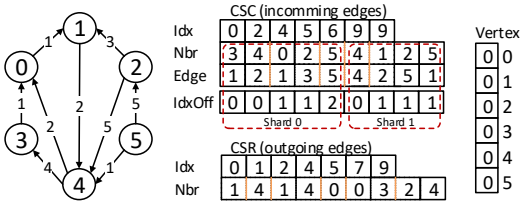


Figure 1: Graph Representation in Garaph

concentration of edges results in heavy write contention among GPU threads due to atomic updates of the same vertices. Colliding threads will be serialized, seriously harming performance on GPUs. Further, the power-law degree distribution can lead to substantial work imbalance across CPU threads in existing non-distributed graph systems that treat vertices symmetrically. Thus, its impact on the parallelism of both GPU and CPU should be alleviated by effective optimization techniques. Another challenge is how to efficiently integrate heterogeneous parallelism of CPU and GPU for graph processing under an unified abstraction, as CPUs are fast for the sequential processing whereas GPUs are suitable for the bulk parallel processing.

In this context, we present Garaph, a non-distributed system that supports *GPU-accelerated* graph processing with secondary storage as memory extension. Garaph enables using all CPU and GPU cores on a given node for graph processing, and with Garaph's abstractions, users only need to write one program that can be executed by both CPU and GPU. Besides, Garaph uses an array of SSDs to achieve high throughput and low latency storage, which enables the system to process large-scale graphs exceeding the computer's memory capacity.

Garaph is novel in the following aspects: To cope with the heterogeneity of vertex degree, Garaph proposes a vertex replication degree customization scheme on the GPU side that maximizes the GPU utilization given vertex degree and space constraints. On the CPU side, Garaph adopts a balanced edge-based partition to ensure work balance over CPU threads. For the heterogeneity of computation units, Garaph first uses a pull computation model matching the SIMD processing model of GPU, and a hybrid of notify-pull and pull computation models optimized for fast sequential processing on the CPU. Further, Garaph uses a dynamic workload assignment scheme which takes into account both the characteristics of processing elements and the properties of graph applications. These new schemes together make for an efficient implementation, achieving full parallelism on both CPU and GPU sides in a single machine.

We evaluate our Garaph prototype with extensive experiments and compared it with four state-of-the-art systems. Experiments with six applications on seven real-world graphs demonstrate that Garaph significantly outperforms existing state-of-art CPU-based and GPU-based systems, getting a speedup of 2.56x on average (up to 5.36x). Through solving conflicts in computation, customized replication scheme can improve GPU's performance by 4.84x speedup on average (up to 32.15x).

2 System Overview

In this section, we give a brief overview on the graph representation, the architecture and the computation abstraction of Garaph.

2.1 Graph Representation

Garaph adopts both *Compressed Sparse Column (CSC)* and *Compressed Sparse Row (CSR)* for organizing incoming and outgoing edges, respectively. The index array `Idx` records each vertex's edge distribution: for vertex i, `Idx[i]` and `Idx[i+1]` indicate the beginning and ending offsets of its incoming/outgoing edges. The array `Nbr` records sources of incoming edges or destinations of outgoing ones. The arrays `Vertex` and `Edge` record values of vertices and edges, respectively. For example, as shown in CSC of Figure 1, `Idx[0]` and `Idx[1]` indicates that vertex 0 has two incomming edges with sources $3,4$ and values $1,2$, respectively.

As stated in CuSha [16], the CSR representation is not friendly for processing graphs in GPUs, which could incur high frequency of non-coalesced memory accesses and warp divergence. To overcome this problem, we also adopt the concept of **shard**. In particular, we split the vertices V of graph $G = (V,E)$ into disjoint sets of vertices and each set is represented by a shard that stores all the incoming edges whose destination is in that set. Edges in a shard are listed based on increasing order of their indexes of destination vertices. Given sorted edges, we index the destination of each edge by the offset to the first destination vertex in this shard, represented by a array of `IdxOff`, for example, as illustrated in Figure 1, edges with destination $3,4,5$ are in shard 1 and destinations' `IdxOff` are $0,1,2$, respectively.

To improve GPU utilization, we allow each shard to be fit into the shared memory for high bandwidth. Specifically, the number of vertices in each shard is determined by $C_{shm}/(N_{Block} \cdot S_{vertex})$, where C_{shm} is the size of the shared memory, N_{Block} is the number of threads blocks in the SM and S_{vertex} is the size of one vertex. However, if a chunk adopts vertex replication with a factor of R (described later), the shard size should reduce by R times. In practice, one block of GPU threads can use up to 48KB shared memory. Let $S_{vertex} \geq 32$ bits for billion-scale graphs, so each shard contains at most 12K vertices. This also implies that maximum offset in `IdxOff` is 12K, so we can use 16-bit integer to represent the index of destination vertices. This compression could not only save GPU memory, but also reduce the traffic of copying

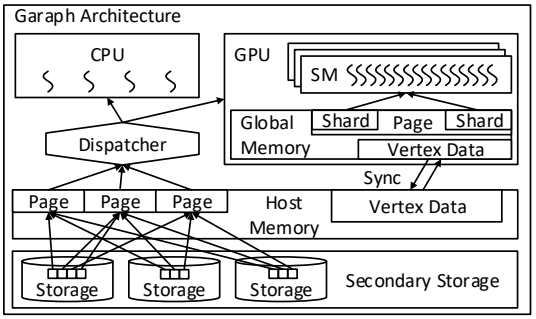


Figure 2: Garaph Architecture

a shard from the host memory to the GPU memory.

To improve the efficiency of CPU-GPU memory copy, we transfer the shards from host memory to GPU memory in batch. In the following, we call the set of shards transferred in a batch as a **page**. Each page contains the maximum number of consecutive shards that can be stored completely in the GPU memory. As we shall describe later, the system also leverages the multi-stream feature of GPUs for the overlap of memory copy and kernel execution. Let N_s be the number of streams. In this case, the page size is chosen as the maximum number of shards that can be stored in $1/N_s$ of the GPU memory.

With above graph representations, our system adopts the following two vertex-centric computation models. The first is *pull* model where every vertex updates its state by pulling the new states of neighboring vertices through incoming edges. The other is the *notify-pull* model where only the active vertices notify their outgoing neighbors to update, who in turn perform local computation by pulling states of their incoming neighbors. Clearly, this model is more effective in case of few active vertices. Note that the CSR is only used for notification. To save memory usage, Garaph does not store the values of outgoing edges in the CSR (see Figure 1).

2.2 System Architecture.

Figure 2 shows the architecture of Garaph, which consists of three main components: dispatcher, CPU and GPU computation kernels.

Dispatcher. This functional module is responsible for loading graph from secondary storages, distributing the computation over CPU and GPU and making adjustment if necessary. To exploit I/O parallelism, Garaph partitions the each graph page of into equal-size data blocks, which are uniformly distributed over multiple secondary storages (e.g., SSDs) with a hash function.

To process the graph, data blocks are loaded from the secondary storages to the host memory to construct pages by the dispatcher. After one page has been constructed, it will be dispatched to either the CPU or the GPU.

GPU/CPU computation kernel. These two kernels

```
interface GASVertexProgram(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  activate(D_u^new, D_u) → A[u]
```

Figure 3: Garaph API

are in charge of graph processing. After receiving a page from the dispatcher, the GPU kernel processes the shards of page in a parallel manner, where each shard is processed by a block in the GPU. For efficient graph processing, only the pull model is enabled on the GPU side. This is because the notify-pull model can lead to high frequency of non-coalesced memory accesses because of poor locality and warp divergence caused by distinguishing active/inactive vertices, significantly limit its performance while processing graphs in the GPU.

The CPU kernel enables both *pull* and *notify-pull* computation models. To balance the computation across multiple threads, the kernel divides edges of a page into sets of equal size, with each thread processing one edge set. When either of two kernels has processed one page, there will be a synchronization between the CPU and the GPU. We shall describe these two kernels in Section 3 and 4.

Garaph enables programs to be executed both synchronously and asynchronously. As the system processes the graph page-by-page, we define an *iteration* as a complete process over all the pages for one time, irrespective of synchronous or asynchronous execution. The synchronous execution model ensures a deterministic execution regardless of the number of machines and closely resembles Pregel [22]. Changes made to the vertex data are committed at the end of each iteration and are visible in the subsequent iteration. When run asynchronously, changes made to the vertex data are immediately committed and visible to current and subsequent iterations. The system terminates the processing if the graph state converges or a given number of iterations are completed.

Fault Tolerance. Garaph enables fault tolerance by writing the vertex data to secondary storages periodically. When Garaph runs synchroniously, Garaph will write the vertex data into stable storages after one or several iterations (user-defined). When Garaph runs asynchroniously, Garaph will write the updated vertex data from the main memory into stable storages after one page has been processed or write the whole vertex data into stable storages after one or several iterations (user-defined). When a fault occurs, it loads the vertex data from the secondary storage and continues to run the application.

2.3 Programming APIs

Garaph implements a modified Gather-Apply-Scatter (GAS) abstraction used in PowerGraph [10]. For a vertex

```
template<typename T>
__host__ __device__ void unifiedAdd(T *addr, T val) {
#ifdef __CUDA_ARCH__ // For GPU, atomic operation
    atomicAdd(addr, val);
#else // For CPU, non-atomic operation
    *addr += val;
#endif
}
```

Figure 4: Garaph unifiedAdd Operation

u, the *gather* function is passed the data on the adjacent vertex D_v and edge $D_{(u,v)}$ and returns a temporary accumulator *Accum*, which is combined using the commutative and associative *sum* operation. When algorithms are commutative and associative, this abstraction is equivalent to original GAS abstraction [10]. After the gather phase has completed, the *apply* function takes the final accumulator and computes a new vertex value D_u^{new}.

In the GAS abstraction, the scatter function is invoked to produce new edge value $D_{(u,v)}^{new}$ which are written back to different machines in a distributed environment. However, Garaph is specific for a non-distributed platform, where each vertex can access its neighbors' updated values in memory so that pushing new data over edges is unnecessary. Thus, we modify the *scatter* function to *activate* function which sets $A[u] = 1$ if vertex u satisfies the active condition defined in the function (e.g., there is a significant change in the vertex value), otherwise, the function sets $A[u] = 0$ to indicate u is inactive. Figure 3 shows the functions Garaph supports.

Garaph packages a set of operations with atomic and non-atomic as unified operations (e.g. unifiedAdd in figure 4) which covers all the atomic operations of CUDA [25]. In Garaph, users only need to write one kernel code which could be executed by both CPU and GPU. Since multiple GPU threads might simultaneously modify the same memory address, the user-provided sum function must be atomic on the GPU side. Garaph packages the non-atomic operations for CPU and the atomic operations for GPU into one set of operations, so that users could directly invoke irrespective of implementing the program on GPU or CPU.

3 GPU-Based Graph Processing

In this section, we first describe how Garaph executes iterative parallel graph algorithms on the GPU side. We then propose an vertex replication degree customization scheme that maximizes the expected GPU performance given properties of the graph and the GPU.

3.1 Graph Processing Engine

To facilitate efficient processing of graphs on GPU using shards, the GPU kernel maintains an array named

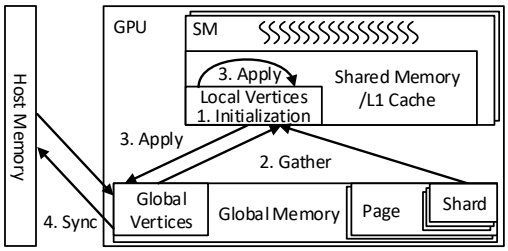


Figure 5: GPU-Based Graph Processing Engine

GlobalVertices in the global memory, which allows quick access to the values of vertices. Current GPUs can support up to 24GB global memory, whereas the size of vertices is usually 4 bytes (FP32 or INT) or 2 bytes (FP16). So GPUs can store up to 6 billion (or 12 billion) vertices in global memory, which is sufficient for most datasets, for example, the largest open source dataset (HyperLink12 [18]) has 3.5 billion vertices.

Multiple shards of a page are processed by threads blocks running on many streaming-multiprocessors (SM) in a parallel manner, where each shard is processed by a thread block. As illustrated in Figure 5, each shard in a page is processed by one GPU block in three phases: *initialization*, *gather* and *apply*. When the page has been processed, the new vertex values are *synchronized* between GPU global memory and host memory. Let S_i represent the destination vertices in one shard.

Initialization. At the beginning, the GPU allocates an array *LocalVertices* in the shared memory of this SM to store the accumulate value of each vertex in a shard. Then, consecutive threads of a block initialize this array with default vertex values defined by users, e.g., 0 for the PageRank application.

Gather. Threads of one GPU block process edges of an individual shard. For each edge (u,v), one thread fetches vertex and edge data from the global memory and increases the accumulate value: $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v)), \forall v \in \mathbf{Nbr}[u]$. To have coalesced global memory accesses, consecutive threads of the block read consecutive edges' data in global memory.

Apply. Each thread of a block updates the vertex value in the shared memory: $D_u^{new} \leftarrow \text{apply}(D_u, a_u), \forall u \in S_i$. Consecutive threads of this block process consecutive vertices. When executing programs asynchronously, the system commits new vertex data to the *GlobalVertices* array, which are immediately visible to the subsequent computation. Otherwise, these values are written to a temporary array in the global memory, which would be visible in the next iteration.

Synchronization from GPU to CPU. Once the whole graph page has been processed, the GPU global memory will be synchronized with the host memory. For programs executed asynchronously, the system transmits the

updated values of the *GlobalVertices* in the GPU global memory to the array storing the most updated values of vertices in the host memory. For those executed synchronously, updated values stored in the temporary space of the GPU global memory are transmitted to a temporary array in the host memory, which will be committed after this iteration ends. As the PCIe bus is full-duplex and most GPUs have two copy engines, the synchronization can be overlapped with processing pages in GPU.

3.2 Replication-Based Gather

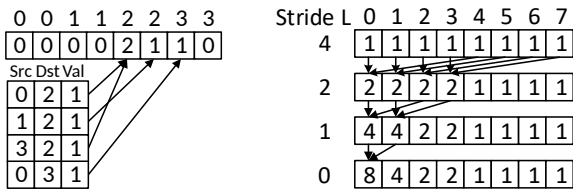
Our above GPU-accelerated framework provides a convenient environment to write graph processing applications. However, the current design still suffers the write contention problem in the gather phase, since multiple threads might collide while simultaneously modifying the same shared memory address (e.g., processing edges with the same destination). Such a collision is called as *position conflict*. The position conflict typically entails a need to serialize memory updates that is resolved by using atomic operations. These consist of a memory read, an arithmetic operation, and a memory write, entailing a latency penalty that is proportional to the number of colliding threads n: a n-way position conflict incurs a penalty of $(n-1)\times t_{position}$, where $t_{position}$ is the processing time of one atomic operation [9].

Notice that natural graphs in the real-world have highly skewed power-law degree distributions, which implies that position conflicts will be very frequent, especially for those vertices of high degree. The heavy write contention leads to an immense performance bottleneck on the GPU side, so its impact on the gather phase should be alleviated by effective optimization techniques.

A general strategy for reducing the conflicts is *replication*, which consists of placing R adjoining copies of the partial accumulated value a'_u in the shared memory to spread these accesses over more shared memory addresses. Then these R partial accumulated values are aggregated to calculate the final accumulated value a_u for a vertex u. Here, R is called as *replication factor*.

Mapping and Aggregation To implement the replication, a mapping function is needed to assign to each thread a replicated copy of the vertex, where the thread will perform the atomic operations. For efficient mapping, we require the vertices of a shard S_i have the same replica factor of R_i. So for any vertex u_i in this shard, the mapping function used in our system is given by: $addr[u'_i] = (i - r_s) * R_i + tid\%R_i, \forall u_i \in S_i$, where $addr[u'_i]$ represents the address of the replica u'_i assigned to the thread tid, and the r_s is the beginning index of the shard.

The mapping makes consecutive threads access consecutive copies. With the mapping, threads perform gather on multiple replicas in a parallel manner: $a'_u \leftarrow \text{sum}(a'_u, \text{gather}(D_u, D_{(u,v)}, D_v)), \forall v \in \mathbf{Nbr}[u']$, where u' is a replica of vertex u.



(a) Mapping ($R = 2$) (b) Aggregation ($R = 8$)

Figure 6: Mapping and Aggregation

After all the edges in the shard have been processed, the system needs a additional phase to aggregate values of different replicas in the shared memory, i.e., $a_u \leftarrow \text{sum}(a_u, a'_u)$. For fast aggregation, we set $R_i = 2^n (n \geq 0)$ to implement a two-way merge illustrated in Figure 6(b). In each iteration, the thread tid executes the user-defined *sum* function with a stride length of L. Initially, $L = R_i/2$, after all replicas are processed, $L \leftarrow L/2$ and the next iteration begins. This procedure stops until $L < 1$.

Replication Factor Customization. Although replication can reduce write conflicts, excessive replication could still lead to GPU underutilization in that fewer vertices can be fit in the shared memory. To achieve a balanced replication, we propose a replication factor customization scheme that maximizes the expected performance under given conflict degree and space constraints.

To do so, we model the execution time of the replication-based gather phase to examine impact of R_i. Let V_i, E_i be the number of destination vertices and edges in a shard S_i, and let t_l and t_a be the time of accessing the global memory and executing an atomic operation in the shared memory, respectively.

In the gather phase, to process a shard S_i with replication factor R_i, threads read $R_i \times |V_i|$ vertices' data and $|E_i|$ edges' data from the global memory into the shared memory, thus taking $R_i \times |V_i| \times t_l$ and $|E_i| \times t_l$, respectively. In the shared memory, threads execute $|E_i|$ atomic operations with the average conflict degree of $\frac{|E_i|}{|V_i| \times R_i}$, which takes $\frac{|E_i|^2}{|V_i| \times R_i} \times t_a$. The final step of aggregation takes $|V_i| \times t_a \times \log R_i$. Thus, the total time $T_G(R_i)$ of processing the shard S_i is given by:

$$T_G(R_i) = R_i|V_i|t_l + |E_i|t_l + \frac{|E_i|^2}{|V_i|R_i}t_a + |V_i|t_a \log R_i. \quad (1)$$

Our goal is to find the R_i minimizing $T_G(R_i)$. We simplify the above equation with $\log R_i \ll R_i$, and $T(R_i)$ is minimized when $R_i \times |V_i| \times t_l = \frac{|E_i|^2}{|V_i| \times R_i} \times t_a$.

Solving the above equation, we get the best replication factor for a shard S_i as: $R_i = \frac{|E_i|}{|V_i|} \cdot \frac{t_a}{t_l}$. In practice, $t_a \approx t_l$, and we get $R_i = \frac{|E_i|}{|V_i|}$. Notice that position conflicts between two consecutive threads will be complete-

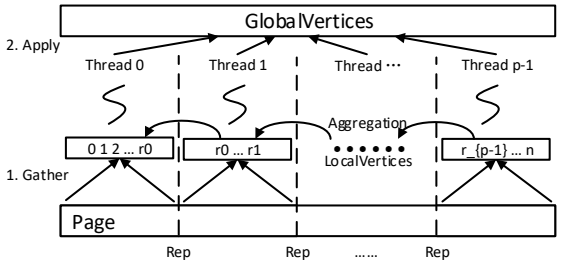


Figure 7: Processing with Edge Partitions on the CPU

ly removed when the replication factor is up to 32, i.e., one copy for each thread of one warp [13]. Also recall $R_i = 2^n (n \geq 0)$ for efficient aggregation, so we choose the value of 2^n form that is closest to average degree of a shard as its final replication factor:

$$R_i = 2^{\min\left\{\left\lceil \log \frac{|E_i|}{|V_i|} - 0.5 \right\rfloor, 5\right\}}. \quad (2)$$

Our replication policy suggests that the number of replicas for each vertex should be customized to the average degree of vertices in a given shard. Specifically, shards containing more high-degree vertices tend to have higher replication degrees for a larger probability of reduction in conflicts.

4 CPU-Based Graph Processing

In this section, we first describe the graph processing of Garaph on the CPU side, which adopts a balanced edge-based partition to exploit full parallelism. We then describe the dual-mode processing model of Garaph, which adaptively switches between the pull/notify-pull modes according to the density of active vertices in the page.

4.1 Processing with Edge Partitions

Existing single-node graph systems treat vertices symmetrically and parallelize the graph processing by assigning each thread a subset of vertices to process. However, this method leads to substantial computation imbalance due to the power-law degree distribution. Further, it also increases the random memory access of edge data if adjacent vertices are assigned to different threads. These issues degrade the overall system performance.

Different from common systems, Garaph adopts edge-centric partition. As illustrated in Figure 7, the edges of a page are equally partitioned across threads, where multiple CPU threads process independent edge sets in a parallel manner. Vertices cut at the partition boundaries would be replicated, and the system would aggregate the replicas' values to obtain the vertex value. This partition enhances the sequential access of edge data and improves work balance over threads.

The CPU engine also maintains a *GlobalVertices* array in the host memory for quick access to values of vertices. Each page is processed in three stage: *initialization*, *gather*, *apply*. If a page has been processed on the GPU side, the system also synchronizes new vertex values between the GPU memory and the host memory. For a common graph application, the processing is done by pulling new vertex states along outgoing edges, until the graph state converges (e.g., no active vertices) or a given number of iterations are completed. Vertices with significant state change are called active vertices (determined by *activate*() function). We use a bitmap A to indicates the inactive/active state of each vertex.

Initialization. Let n_t be the number of CPU threads. The edges of the page is divided to n_t partitions of the same size, and thread tid processes the tid_{th} partition. The fist and last vertices of partition will create a replica respectively if they are cut at the boundaries. So the number of replicas is at most $n_t - 1$. Each CPU thread maintains a *LocalVertices* array to store the accumulate values of destination vertices in the corresponding partition. Like the GPU, this array is initialized with the vertices' default value defined by users.

Gather. Each partition of the page is processed by one CPU thread. For each edge, the CPU thread performs gather and updates the accumulate value a_u in *LocalVertices* with sum function. Edges are processed in a sequential order whereas the source vertices' values are accessed randomly by each thread. After each thread has processed its partition, an *aggregation* phase aggregates values of vertices replicated at the partition boundaries. Recall that the number of replicas is at most $n_t - 1$, which is small enough for one CPU thread to process. In Garaph, thread 0 scans the whole partitions in the reverse order and aggregates values of replicated vertices, as illustrated in Figure 7.

Apply. After the gather phase of each page is finished, every thread updates vertices' values in their own *LocalVertices* array. For each partition, if the first vertex is replicated, it will be ignored because its value has already been aggregated to the last vertex of the previous adjacent partition. For each vertex in a partition, the corresponding thread calls *activate*() function to examine if the vertex is active or not and updates the bitmap A.

Synchronization from CPU to GPU. As described in Section 3.1, after the GPU has processed a page, it sends the corresponding vertex values to the host memory. When receiving the new values, the system first calls *Activate*() function to update the bitmap A of these updated vertices. When runs asynchronously, the system enables the updates received from the GPU immediately visible through writing them into the *GlobalVertices* array of host memory. After that, the system sends back new ver-

tices updated on CPU side to the GPU and overwrites the corresponding part of the *GlobalVertices* array in the GPU global memory. When run synchronously, the system stores updated values in a temporary array, and commits these new values at the end of each iteration. In this case, the CPU transmits the new *GlobalVertices* array to that in the GPU memory at the end of each iteration.

4.2 Dual-Mode Processing Engine

To this end, our CPU-based processing engine adopts a *pull* mode where every vertex performs local computation by pulling the states of neighboring vertices through incoming edges. However, a vertex needs to be updated only when one of its source vertices is active in the previous iteration. Thus, another way to process vertices is *notify-pull* mode where only the vertices activated in the last iteration notify their outgoing neighbors, who in turn perform local computation by pulling states of their incoming neighbors. Intuitively, the notify-pull mode is more efficient when few vertices are active in the last iteration (sparse active vertex set), as the system only traverses the outgoing edges of active vertices where new updates to be made. In contrast, the pull mode is more beneficial when most vertices are activated (dense active vertex set), which avoids the extra cost of notifications.

At a given time during graph processing, the active vertex set may be dense or sparse. For example, SSSP or the BFS starts from a very sparse set, becoming denser as more vertices being activated by their in-neighbors, and sparse again when the algorithm approaches convergence. To incorporate the benefits of both modes, we extend our CPU processing to a dual engine design determined by the size of the vertex set $V_A = \{v|(u,v) \in E, A[u] = true\}$ given graph $G(V,E)$, i.e., the outgoing neighbors of vertices activated in the last iteration.

We first consider the case where the graph representations can be completely loaded into the host memory. Let T_{pull} be the time of graph processing in the pull mode. Clearly, T_{pull} is independent of $|V_A|$. In contrast, the notify-pull mode only notifies a fraction of $f = |V_A|/|V|$, who are updated in turn. Hence, we estimate the average processing time in this mode as $T_{notify-pull} = 2fT_{pull}$, as the time to notify is at most equal to pulling state along edges. So the system would adopt *notify-pull* mode if the $f \leq 1/2$ (i.e., $T_{notify-pull} < T_{pull}$), otherwise, the *pull* model would be adopted.

However, for the scenario where only part of graph can be loaded into the host memory, the system entails I/O cost due to sequential and random accesses of outgoing/incoming edges on secondary storage for pull and notify-pull modes respectively. Let k $(k > 1)$ be the rate of speeds between sequential read and random read (e.g., $k \approx 10$ in SSD), the $T_{notify-pull} = 2kf \cdot T_{pull}$. In this case, the system would adopt *notify-pull* mode if $f \leq \frac{1}{2k}$.

Let $\Gamma = \{u|A[u] = 1\}$ be the set of active vertices in the last iteration, the system can estimate $f \approx \frac{\sum_{u \in \Gamma} d_u}{|E|}$ where d_u is the out-degree of an active vertex u and E is the set of edges. Garaph estimates f in the beginning of each iteration and choose which mode to use based on f.

5 Dispatcher

We have discussed how to design and optimize graph processing kernel for efficient execution on both GPU and CPU sides. To further improve the performance, we propose an adaptive scheduling mechanism to exploit the overlap of two engines. Besides, we also perform multi-stream scheduling for data transfer and GPU kernel execution overlap. Below we shall detail each scheduling strategy respectively.

5.1 CPU-GPU Scheduling

We first determine when it is beneficial to adopt GPU acceleration. From Section 4.2 we know that the processing time with only CPU kernel is $T_{CPU} = \min\{2f\rho, 1\} \cdot T_{pull}$, where f is the fraction of vertices to be updated and T_{pull} is processing time in the pull mode. Here, $\rho = 1$ if the graph representations can be fit into the host memory, otherwise, $\rho = k$, which is the rate of speeds between sequential read and random read of secondary storage. Notice that GPU-based kernel needs to process all the edges of a given page, so the GPU's processing time T_{GPU} is independent of f. Therefore, if $T_{CPU} < T_{GPU}$, the system adopts CPU kernel only due to sparse active vertex set. Otherwise, Garaph adopts both GPU and CPU kernels to reduce the overall time of the processing.

Based on the above insight, our scheduler works as follows: At beginning of every iteration, the scheduler calculates the following ratio of T_{CPU} to T_{GPU}:

$$\alpha = \min\{2f\rho, 1\} \cdot \frac{T_{pull}}{T_{GPU}}, \tag{3}$$

where the fraction T_{pull}/T_{GPU} is initialized by the speed ratio of CPU/GPU hardwares, and is updated once both kernels have begun to process pages. Specifically, let t^p_{cpu} and t^p_{gpu} be the measured time to process a page via CPU and GPU kernels, respectively, we can estimate $T_{pull}/T_{GPU} = t^p_{cpu}/(f \cdot t^p_{gpu})$.

In the case of $\alpha < 1$, only CPU kernel is used for graph processing in this iteration as most vertices are inactive (e.g., a very small f). Otherwise, the system processes graph pages in parallel on both CPU and GPU kernels. In the hybrid mode, the system reactively assigns a page to a (GPU or CPU) kernel once the kernel becomes free. The processing is finished if the graph state converges (i.e., $f = 0$) or a given number of iterations are completed.

Graph	$\|V\|$	$\|E\|$	Max in-deg	Avg deg	Size edgelist
uk-2007@1M	1M	41M	0.4M	41	0.6GB
uk-2014-host	4.8M	51M	0.7M	11	0.8GB
enwiki-2013	4.2M	0.1B	0.4M	24	1.7GB
gsh-2015-tpd	31M	0.6B	2.2M	20	10GB
twitter-2010	42M	1.5B	0.8M	35	27GB
sk-2005	51M	1.9B	8.6M	39	35GB
renren-2010	58M	2.8B	0.3M	48	44GB
uk-union	134M	5.5B	6.4M	41	0.1TB
gsh-2015	988M	34B	59M	34	0.7TB

Table 1: Graph datasets [19, 5, 4, 23] used in evaluation.

5.2 GPU Multi-Stream Scheduling

To trigger the graph processing on the GPU side, there are two threads running on the host: the transmission thread and the computation thread. The former thread continuously transmits each page from the host memory to GPU's global memory. The later thread launches a new GPU kernel to process the page that has already been transmitted.

Using NVIDIA's Hyper-Q feature [24], we perform multi-stream scheduling for the pipelining of CPU-GPU memory copy and kernel execution, so that the processing tasks of pages can be dispatched onto multiple streams and handled concurrently. In particular, we schedule tasks of processing pages onto N_{stream} streams, the transmission thread periodically examines which stream is *idle*, and dispatches the transmission task of one page to the idle stream, where pages are asynchronously transferred from the host memory to the GPU global memory. The computation thread periodically examines which page has been completely transferred, and triggers the computation of that page by dispatching the computation task to the corresponding stream. This multi-stream scheduling enables a high overlapping between CPU-GPU memory copy and kernel execution.

6 Evaluation

In this section, we describe and evaluate our implementation of the Garaph system. Garaph is implemented in more than 8,000 lines of C++ and CUDA code, compiled by GCC 4.8 and CUDA 8.0 on Ubuntu 14.04 with O3 code optimization. In the CPU processing kernel, the number of processing threads is equal to the number of CPU cores by default. In the dispatcher module, each I/O thread reads/writes one secondary storage device so that threads process I/O operations in a parallel manner.

The experiments are performed on a system with Nvidia GeForce GTX 1070 which has 15 SMs (1920 cores) and 8GB global memory. On the host side, there is an Intel Haswell-EP Xeon E5-2650 v3 CPU with 10 cores (hyper-threading enabled) operating at 2.3 GHz clock frequency, and 64GB dual-channel memory. PCI Express 3.0 lanes operating at 16x speed transfer data between the host DDR4 RAM (CPU side) and the device RAM (GPU side).

We use the real-world graphs in Table 1 for evaluation. The largest graph is the gsh-2015 graph with about 1 billion vertices and 34 billion edges. We use six representative graph analytics applications: single source shortest paths (SSSP), connected components (CC), PageRank (PR) [26], neural network (NN) [3], heat simulation (HS) [16], circuit simulation (CS) [16]. We run PR, NN, HS, CS for 10 iterations and CC, SSSP till convergence. To get stable performance, the reported runtime is calculated as the average time of 5 runs.

6.1 Comparison with Other Systems

We compare Garaph of hybrid CPU/GPU kernels (marked as Garaph-H in Table 2) with four state-of-the-art systems: shared-memory systems including CuSha [16], Ligra [32] and Gemini [36], and one secondary-storage-based system GraphChi [17]. Here, CuSha is a GPU framework for processing graphs that can be fit in the GPU memory. To show the performance of each kernel, we also give the performance of Garaph with CPU-kernel only and GPU-kernel only (marked as Garaph-C and Garaph-G in Table 2, respectively).

For datasets that can be placed in host memory, Table 2 presents the performance of evaluated systems. Benefit from customized replication for reducing position conflicts, Garaph-G significantly outperforms CuSha in all cases, 2.34x on average and up to 3.38x for PR on the uk-2014-host dataset. Garaph-G also outperforms other CPU-based systems in compute-intensive applications such as PR, NN, HS and CS. But for SSSP and CC, both Garaph-G and CuSha take longer time to get convergence, as they have to process the whole graph despite of a few active vertices to be processed.

With balanced replication and the optimization for sequential memory access, Garaph-C also outperforms existing systems in many cases: e.g., for PR excluding enwiki-2013 and sk-2005 datasets, for NN and SSSP excluding renren-2010 dataset and for CC in all datasets. Adaptive dual-mode processing engine enables Garaph-C to significantly outperform GPU-based systems in the cases of SSSP and CC.

The above results reveal that Garaph-G is suitable for compute-intensive applications whereas and Garaph-C performs well in applications like SSSP and CC. With the CPU-GPU scheduling, Garaph-H combines the advantages of both Garaph-C and Garaph-G. As a result, Garaph-H significantly outperforms other systems in all cases, e.g., 2.56x on average and up to 5.36x for CC on the twitter-2010 dataset.

	PR						NN					
Graph	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H
uk-2007@1M	0.48	0.77	0.43	0.43	0.20	**0.16**	0.70	0.78	0.44	0.33	0.33	**0.20**
uk-2014-host	0.83	1.06	0.74	0.60	0.25	**0.22**	0.82	0.98	0.88	0.46	0.41	**0.26**
enwiki-2013	1.39	1.80	0.96	1.29	0.49	**0.39**	1.63	1.27	1.18	0.96	0.81	**0.52**
gsh-2015-tpd	-	11.80	8.08	7.80	2.91	**2.42**	-	11.00	7.78	5.91	4.83	**3.36**
twitter-2010	-	-	22.82	22.40	7.50	**6.13**	-	-	20.67	18.53	11.80	**8.61**
sk-2005	-	-	15.85	17.18	9.38	**6.83**	-	-	18.42	12.46	15.67	**9.73**
renren-2010	-	-	84.63	79.77	22.54	**20.89**	-	-	67.60	73.50	22.47	**20.63**
	SSSP						CC					
Graph	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H
uk-2007@1M	6.57	0.57	1.42	0.56	3.95	**0.48**	0.50	0.45	0.38	0.20	0.20	**0.14**
uk-2014-host	13.93	0.73	2.19	0.72	5.88	**0.57**	1.20	0.62	0.72	0.24	0.54	**0.17**
enwiki-2013	11.56	0.99	1.73	0.92	5.54	**0.70**	0.97	0.80	0.77	0.45	0.44	**0.28**
gsh-2015-tpd	-	8.36	9.54	6.70	14.51	**4.32**	-	7.47	5.14	2.58	2.62	**1.21**
twitter-2010	-	-	26.97	23.24	42.49	**12.75**	-	-	17.78	8.78	12.04	**3.32**
sk-2005	-	-	61.04	26.82	1335.29	**18.13**	-	-	13.33	6.49	22.53	**4.74**
renren-2010	-	-	54.60	73.00	615.82	**26.79**	-	-	41.80	35.79	205.03	**8.55**
	HS						CS					
Graph	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H	CuSha	Ligra	Gemini	Garaph-C	Garaph-G	Garaph-H
uk-2007@1M	0.80	0.87	0.30	0.39	0.33	**0.23**	0.80	0.87	0.32	0.43	0.33	**0.23**
uk-2014-host	0.98	1.06	0.71	0.55	0.41	**0.29**	0.99	1.06	0.83	0.59	0.41	**0.29**
enwiki-2013	1.95	1.69	0.98	1.14	0.81	**0.55**	1.95	1.44	0.91	1.19	0.81	**0.55**
gsh-2015-tpd	-	13.80	6.15	6.67	4.84	**3.40**	-	11.10	6.30	4.84	7.20	**3.45**
twitter-2010	-	-	18.83	21.17	11.81	**9.11**	-	-	16.10	21.74	11.80	**8.99**
sk-2005	-	-	13.52	14.34	15.66	**9.95**	-	-	13.24	15.33	15.66	**10.62**
renren-2010	-	-	78.10	82.97	22.47	**21.53**	-	-	59.51	83.84	22.47	**21.29**

Table 2: Runtime (in seconds) of six applications in memory. '-' indicates incompletion due to running out of memory.

	PR		CC	
Graph	GraphChi	Garaph	GraphChi	Garaph
In-memory (10 iters for PR, convergence for CC.)				
uk-2007@1M	2.58	0.16	12.47	0.20
uk-2014-host	4.55	0.22	21.17	0.24
enwiki-2013	7.39	0.39	27.46	0.46
gsh-2015-tpd	39.69	2.42	179.27	1.21
twitter-2010	253.45	6.13	618.58	3.32
sk-2005	-	6.83	-	4.74
renren-2010	-	20.89	-	8.55
Secondary Storage (5 iters for PR and CC.)				
uk-union	899	161	2558	157
gsh-2015	11595	2269	13897	1383

Table 3: Runtime (in seconds) of PR and CC in memory and secondary storages (three SATA SSDs). '-' indicates incompletion due to running out of memory.

As uk-union and gsh-2015 datasets can be only placed in secondary storage, CuSha, Ligra and Gemini cannot run any applications due to the limit of memory capacity.

We compare Garaph with GraphChi in two ways: (1) For datasets that can be fit in memory, we redirect GraphChi's I/O operations from secondary storages to memory by modifing its open-source code. We run PR for 10 iterations and CC till convergence. (2) For datasets that need the extension of secondary storage, we run GraphChi on a Raid-0 provided by three SATA SSDs. In this case, Garaph also uses the same SSDs managed by the dispatcher. We only run 5 iterations for PR and CC on large-scale graphs such as uk-union and gsh-2015 that are very time-consuming to get convergence.

Table 3 shows that Garaph outperforms GraphChi in all cases. The experiments of in-memory graphs demonstrate that Garaph's computation engine is more efficient than GraphChi. There are three reasons why Garaph outperforms GraphChi for SSD-based computation: First, benefiting from the the compressed graph representation, Garaph can use less space to store graph data. Second, GraphChi needs both read data from SSDs and write data to SSDs, which may also cause I/O conflicts. Garaph only reads data from SSDs. Futher, in SSDs, the sequential read speed is much faster than the sequential write speed. Finally, according to our tests, Garaph's disk manager is more efficient than the RAID-0 supported by the raid card which is not linear scalable. Note this is a just preliminary result of adopting the secondary storages. We shall try PCIe SSDs or NVMs in the future work.

6.2 Customized Replication

In this section, we evaluate the performance of the customized replication on the GPU side. We partition the sk-2005 dataset into 33 subgraphs (pages) of similar sizes but different topological structures. We run PR on these pages to evaluate the runtime (computation time only) of each page by using the CUDA toolkit profiler.

Figure 8 shows the runtime of each page in one iteration with/without replication. Without replication, the processing time of pages varies significantly, where the slowest one is about 45.17x slower than the fastest one. We also find that the correlation between the runtime and the maximum degree of individual pages is 0.9853, which implies that the time of processing a page is main-

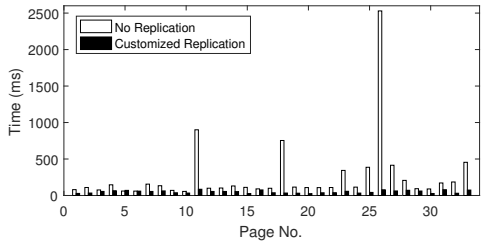


Figure 8: Per-page runtime (computation time only) of PR on the sk-2005 dataset

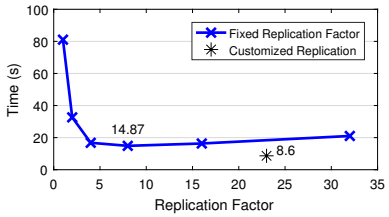
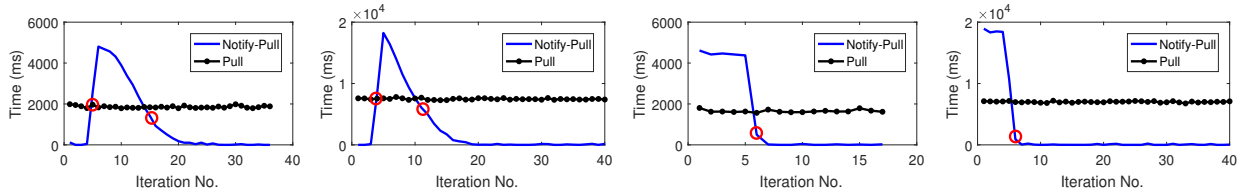


Figure 9: Runtime with different replication factors, the X-axis of customized replication is the average of R_i.



(a) SSSP: twitter-2010 (b) SSSP: renren-2010 (c) CC: twitter-2010 (d) CC: renren-2010

Figure 10: Runtime under notify-pull and pull modes, "o" indicates the iteration where Garaph switches modes.

ly impacted by the vertices of high degree. In contrast, with the customized replication, the processing time of pages is much more balanced and efficient, getting a 4.84x speedup on average (up to 32.15x), significantly reducing the overall time.

We next show customized replication can gain a better performance than a fixed replication factor. To do so, we run 10 iterations PR on the sk-2005 dataset with the fixed factor $R \in \{1,2,4,8,16,32\}$ for the whole graph. Figure 9 shows that the runtime (computation time only) decreases at the beginning and increases with the growing of R. Customized R_i according to equation (2) gets the best performance of 8.6s, getting 1.73x speedup than the best one (14.87s) among all fixed factors.

6.3 Dual Modes of the CPU Kernel

Adaptive switching between pull and notify-pull modes according to the density of active vertices improves the performance of Garaph-C significantly. We propose an experiment by forcing Garaph-C to run under the two modes for each iteration respectively to illustrate the necessities of the dual-mode abstraction.

Figure 10 shows that the performance gap between notify-pull and pull modes is quite significant. For SSSP, the notify-pull mode outperforms the pull mode in most iterations, except several iterations where most vertices are updated. For CC, the pull mode only outperforms the notify-pull mode at the first few iterations when most of the vertices remain active. However, with switching model proposed in Section 4.2, Garaph-C is able to adopt the better mode for each iteration. We see that the switch of Garaph-C occurs at the next iteration around the intersection of the two modes' performance curves.

	CuSha	Ligra	GraphChi	Gemini	Garaph
enwiki-2013	47.57	70.45	41.1	17.96	26.2
gsh-2015-tpd	-	442	249	107.21	137.6
twitter-2010	-	-	654	266.18	353.8

Table 4: Preprocessing Cost (in seconds) of PR

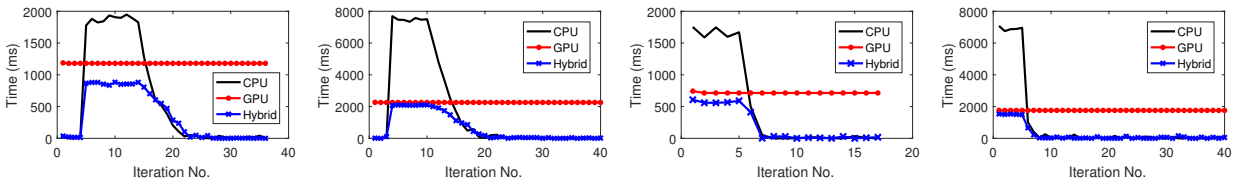
6.4 Scheduling Performance

To demonstrate the speedup of processing graphs on a hybrid platform (compared to processing it on the host only or the GPU only), we run SSSP and CC on twitter-2010 and renren-2010 datasets under CPU-only, GPU-only and hybrid for each iteration, respectively.

As Figure 11 shows, Garaph-H gains much better performance by combing CPU and GPU kernels. For both SSSP and CC, when a few vertices are active, Garaph-H chooses to only use the CPU to process graphs with notify-pull model. However, when most of vertices are active, Garaph-H switches to pull mode in the CPU kernel, and the GPU also joins in computation and accelerates the processing significantly. In contrast, The runtime of the Garaph-C incurs long processing time when most of vertices are active, whereas Garaph-G remains constant because the amount of computation does not change in all iterations of SSSP and CC.

6.5 Preprocessing Cost

Finally, we evaluate the preprocessing cost of Garaph compared to CuSha, Ligra, Gemini and GraphChi on a RAID-0 provided by three SATA SSDs. Garaph and GraphChi will write preprocessed data into secondary storages. Garaph's preprocessing is light-weight, which only needs to scan the input data twice to build the CSC and the CSR data. Table 4 shows the preprocessing cost



(a) SSSP: twitter-2010 (b) SSSP: renren-2010 (c) CC: twitter-2010 (d) CC: renren-2010

Figure 11: Runtime under CPU-only, GPU-only and hybrid modes in two datasets.

of PR on three graphs. It is clear that Garaph's preprocessing is faster than CuSha, Ligra and GraphChi. As Garaph needs to write preprocessed data to secondary storages, Garaph's preprocessing is slower than Gemini.

7 Related Works

In recent years, a large number of graph processing systems have been proposed [36, 28, 37, 7, 16, 11, 32, 29, 21, 17, 10, 2, 27, 22, 35, 33]. We mention here only those most closely related to our work.

GPUs provide a massive amount of parallelism with the potential to outperform CPUs. Numerous graph processing systems [33, 16, 35, 31] have been proposed to use GPUs for high-performance graph processing. Medusa [35] is a generalized GPU-based graph processing framework that focuses on abstractions for easy programming and scaling to multiple GPUs. CuSha [16] primarily focuses on exploring new graph representations to allow faster graph processing. It uses two graph representations G-Shards and Concatenated Windows to improve coalesced memory access and GPU utilization. In CuSha, vertices' values are stored in shards and CuSha needs a phase to write updated values to shards. This design would incur significant data transfer cost between GPU and CPU if it enxtened data to host memory. In Garaph, updated vertices' values are written to global memory instead of shards. Further, CuSha incurs heavy conflicts without any data replicaiton. However, Garaph adopts the replication-based gather to reduce conflicts. Futher, both Medusa and CuSha cannot process graphs exceeding the GPU memory capacity.

To scale out GPU-accelerated graph processing, TOTEM [8] is a processing engine that provides a convenient environment to implement graph algorithms on hybrid CPU and GPU platforms. TOTEM can process graphs whose size is smaller than the host memory capacity. gGraph [34] is another hybrid CPU-GPU system which uses hard-disk drives (HDDs) as secondary storages. For load balancing, both systems initially partition graph into subgraphs that are proportional to the processing power of CPUs and GPUs.

However, existing GPU-accelerated systems cannot fully utilize the GPU for processing large-scale graphs due to ignoring heavy write contention caused by skewed power-law degree distributions and properties of graph algorithms. Garaph further exploits the replication and dynamical scheduling to achieve the best performance on the CPU/GPU hybrid platform.

Shared-memory graph processing systems provide either a push-based [22, 30, 2, 15] or a pull-based model [21, 10, 11, 6, 7], or a switchable model [32, 12, 36]. Garaph modifies push/pull models of Ligra [32] to notify-pull/pull models to achieve lock-free processing. In particular, Ligra's push operations are atomic, whereas our notify-pull/pull model is lock-free with edge-based partitioning. Also, Ligra's critical switching parameter is set by experience, which may not be the best parameter for different applications and datasets. However, notify-pull/pull model is switched by the data-driven model, and thus can achieve a better performance.

8 Conclusion

In this work, we designed a general graph processing platform called Garaph which can efficiently process large-scale graphs using both CPUs and GPUs on a single machine. We design critical system components such as replication-based GPU kernel, optimized CPU kernel with edge-based partition and dual computation modes, and dispatcher with dynamic CPU-GPU scheduling. Our deployment and evaluation reveal demonstrate that Garaph can fully explore both CPU and GPU parallelism for graph processing. Although Garaph is designed for a single machine, the proposed techniques could also be easily applied to distributed, CPU/GPU hybrid systems. Garaph focuses on systems with fast storage (e.g., RAM, or NVM/PCIe-SSD arrary). However, for enviorment with slow secondary storages (e.g., HDD-based system), other optimizations on I/O of secondary storages should be introduced to allivate the bottleneck.

Acknowledgements

Authors would like to thank Christopher J. Rossbach, our shepherd, and the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation under Grant No. 61472009 and Shenzhen Key Fundamental Research Projects under Grant No. JCYJ20151014093505032.

References

[1] ABOU-RJEILI, A., AND KARYPIS, G. Multilevel algorithms for partitioning power-law graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), IEEE, pp. 10–pp.

[2] AVERY, C. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara 11* (2011).

[3] BAKHODA, A., YUAN, G. L., FUNG, W. W., WONG, H., AND AAMODT, T. M. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 163–174.

[4] BOLDI, P., CODENOTTI, B., SANTINI, M., AND VIGNA, S. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience 34*, 8 (2004), 711–726.

[5] BOLDI, P., MARINO, A., SANTINI, M., AND VIGNA, S. BUbiNG: Massive crawling for the masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web* (2014), International World Wide Web Conferences Steering Committee, pp. 227–228.

[6] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 215–226.

[7] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[8] GHARAIBEH, A., REZA, T., SANTOS-NETO, E., COSTA, L. B., SALLINEN, S., AND RIPEANU, M. Efficient large-scale graph processing on hybrid cpu and gpu systems. *arXiv preprint arXiv:1312.3018* (2013).

[9] GOMEZ-LUNA, J., GONZALEZ-LINARES, J. M., BENITEZ, J. I. B., AND MATA, N. G. Performance modeling of atomic additions on gpu scratchpad memory. *IEEE Transactions on Parallel and Distributed Systems 24*, 11 (2013), 2273–2282.

[10] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[11] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *OSDI* (2014), vol. 14, pp. 599–613.

[12] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 1.

[13] HARRIS, M., ET AL. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology 2*, 4 (2007).

[14] INTEL. Intel ssd dc p3608 series product brief. Tech. rep., 2015.

[15] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[16] KHORASANI, F., VORA, K., GUPTA, R., AND BHUYAN, L. N. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 239–252.

[17] KYROLA, A., BLELLOCH, G. E., GUESTRIN, C., ET AL. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[18] LEHMBERG, O., MEUSEL, R., AND BIZER, C. Graph structure in the web: Aggregated by pay-level domain. In *Proceedings of the 2014 ACM conference on Web science* (2014), ACM, pp. 119–128.

[19] LESKOVEC, J., AND KREVL, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[20] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1 (2009), 29–123.

[21] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[22] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[23] MISLOVE, A., MARCON, M., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)* (San Diego, CA, October 2007).

[24] NVIDIA. Kepler gk110 architecture whitepaper, v1.0. Tech. rep., 2012.

[25] NVIDIA. *CUDA C Programming Guide*. nvidia, 2017.

[26] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab, 1999.

[27] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., ET AL. The tao of parallelism in algorithms. In *ACM Sigplan Notices* (2011), vol. 46, ACM, pp. 12–25.

[28] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 410–424.

[29] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[30] SALIHOGLU, S., AND WIDOM, J. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013), ACM, p. 22.

[31] SEO, H., KIM, J., AND KIM, M.-S. Gstream: A graph streaming processing method for large-scale graphs on gpus. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 253–254.

[32] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices* (2013), vol. 48, ACM, pp. 135–146.

[33] WANG, Y., DAVIDSON, A., PAN, Y., WU, Y., RIFFEL, A., AND OWENS, J. D. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2016), ACM, p. 11.

[34] ZHANG, T., ZHANG, J., SHU, W., WU, M.-Y., AND LIANG, X. Efficient graph computation on hybrid cpu and gpu systems. *The Journal of Supercomputing 71*, 4 (2015), 1563–1586.

[35] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (2014), 1543–1552.

[36] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA* (2016).

[37] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference* (2015), pp. 375–386.