Medusa: Simplified Graph Processing on GPUs

Jianlong Zhong, Bingsheng He

Abstract—Graphs are common data structures for many applications, and efficient graph processing is a must for application performance. Recently, the graphics processing unit (GPU) has been adopted to accelerate various graph processing algorithms such as BFS and shortest paths. However, it is difficult to write correct and efficient GPU programs and even more difficult for graph processing due to the irregularities of graph structures. To simplify graph processing on GPUs, we propose a programming framework called Medusa which enables developers to leverage the capabilities of GPUs by writing sequential C/C++ code. Medusa offers a small set of user-defined APIs, and embraces a runtime system to automatically execute those APIs in parallel on the GPU. We develop a series of graph-centric optimizations based on the architecture features of GPUs for efficiency. Additionally, Medusa is extended to execute on multiple GPUs within a machine. Our experiments show that (1) Medusa greatly simplifies implementation of GPGPU programs for graph processing, with many fewer lines of source code written by developers; (2) The optimization techniques significantly improve the performance of the runtime system, making its performance comparable with or better than manually tuned GPU graph operations.

Index Terms—GPGPU, GPU Programming, Graph Processing, Runtime Framework.

1 Introduction

Graphs are common data structures in various applications such as social networks, chemistry and web link analysis. Graph processing algorithms have been the fundamental tools in various fields. Developers usually apply a series of operations on the graph edges and vertices to obtain the final result. The example operations can be breadth first search (BFS), PageRank [32], shortest paths and even their customized variants (for example, developers may apply different application logics on top of BFS). The efficiency of graph processing is a must for high performance of the entire system. On the other hand, writing every graph processing algorithm from scratch is inefficient and involves repetitive work, since different algorithms may share the same operation patterns, optimization techniques and common software components. A programming framework supporting high programmability for various graph processing applications and providing high efficiency as well can greatly improve productivity.

Recent years have witnessed the increasing adoption of GPGPU (General-Purpose computation on Graphics Processing Units) in many applications [31]. The GPU has been used as an accelerator for various graph processing applications [14], [16], [23], [35]. While those GPU-based solutions have demonstrated significant performance improvement over CPU-based implementations, they are limited to specific graph operations. Developers usually need to implement and optimize GPU programs from scratch for different graph processing tasks.

Writing a correct and efficient GPU program is challenging in general, and even more difficult for graph applications. First, the GPU is a many-core processor with massive thread parallelism. To fully exploit the GPU parallelism, developers need to write parallel programs that scale to hundreds of cores. Moreover, compared with CPU threads, the GPU threads are lightweight, and the tasks in the parallel algorithms should be fine grained. Second, the GPU has a memory hierarchy that is different from the CPU's. Since graph applications usually involve irregular accesses to the graph data, careful designs of data layouts and memory accesses are key factors to the efficiency of GPU acceleration. Finally, since the GPU is designed as a co-processor, developers have to explicitly perform memory management on the GPU, and deal with GPU specific programming details such as kernel configuration and invocation. All these factors make the GPU programming a difficult task.

To ease the pain of leveraging the GPU in common graph computation tasks, we propose a software framework named Medusa to simplify programming graph processing algorithms on the GPU. Inspired by the bulk synchronous parallel (BSP) model, we develop a novel graph programming model called "Edge-Message-Vertex" (EMV) for fine-grained processing on vertices and edges. EMV is specifically tailored for parallel graph processing on the GPU. Like existing programming frameworks such as MapReduce [9] and its variant on the GPU [15], Medusa provides a set of APIs for developers to

• *J. Zhong and B. He are with the School of Computer Engineering, Nanyang Technological University, Singapore, 639798. E-mail: jzhong2@ntu.edu.sg, bshe@ntu.edu.sg*

implement their applications. The APIs are oriented at the EMV programming model for fine-grained parallelism. Medusa embraces an efficient message passing based runtime. It automatically executes user-defined APIs in parallel on all the processor cores within the GPU and on multiple GPUs, and hides the complexity of GPU programming from developers. Thus, developers can write the same APIs, which automatically run on multiple GPUs.

Memory efficiency is often an important factor for the overall performance of graph applications [14], [16], [23], [35]. To improve the memory efficiency of Medusa, we have developed a series of memory optimizations. A novel graph layout is developed to exploit the *coalesced* memory feature of the GPU. A graph aware message passing mechanism is specially designed for message passing in Medusa. We also develop two multi-GPU-specific optimization techniques, including the cost model guided replication for reducing data transfer across the GPUs and overlapping between computation and data transfer.

We have evaluated the efficiency and programmability of Medusa on a machine with four NVIDIA C2050 GPUs and two Intel E5645 CPUs. To demonstrate the programmability of Medusa, we develop a set of common graph processing primitives on sparse graphs and compare Medusa-based implementations with manual implementations. The CPU-based manual implementations are based on the MultiThreaded Graph Library (MTGL) [7], and we adopt previous GPU implementations [14], [19], [30] as GPU-based manual implementations.

Our experimental results show that: (1) Medusa simplifies programming GPU graph processing algorithms in terms of a significant reduction in the number of source code lines. Medusa achieves comparable or better performance than the manually tuned GPU graph operations. (2) Our optimization techniques on graph layout and message buffering significantly improve the performance of graph processing operations on the GPU. (3) Medusa executing on four GPUs is up to 1.8 and 2.6 times faster than on a single GPU for BFS and PageRank, respectively.

Organization. The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes the system overview, followed by detailed design in Section 4. We present evaluation results in Section 5, and conclude in Section 6.

2 Related Work

2.1 Graph Processing

Parallel algorithms have been a classical way to improve the performance of graph processing. On multi-core CPUs, parallel libraries such as MTGL [7] have been developed for parallel graph algorithms. Similar to Medusa, MTGL offers a set of data structures and APIs for building graph algorithms. The MTGL API is modeled after the Boost Graph Library [34] and optimized to leverage shared memory multithreaded machines. The SNAP framework [5] provides a set of algorithms and building blocks for graph analysis, especially for small-world graphs. To facilitate developing distributed graph algorithms in the cluster/grid settings, software libraries such as Parallel BGL [13] and Combinatorial BLAS [8] have been developed. Cloud platforms are becoming popular for graph applications [20], [21], [29].

Previous studies [10], [25], [26] have observed that many common graph algorithms can be formulated using a form of the bulk synchronous parallel (BSP) model (we call it *GBSP*). In GBSP, local computations are performed on individual vertices. Vertices are able to exchange data with each other. The same computation and communication procedures are executed iteratively with barrier synchronization at the end of each iteration. This common algorithmic pattern is also adopted by distributed graph processing frameworks such as Pregel [29] and distributed GraphLab [27]. For example, Pregel applies a user-defined function *Compute()* on each vertex in parallel in each iteration of the GBSP execution. The communications between vertices are performed with message passing interfaces. Medusa shares the same design goal as Pregel in providing a programming framework to ease development of graph algorithms, and in hiding the complexity of the underlying runtime from developers.

Medusa differs from Pregel in the following aspects. First, the design, implementation and optimization of Medusa are specific to the hardware features of GPUs. For example, our multi-GPU Medusa adopts graph partitioning to reduce data transfer on the host-device communication link (i.e., PCI-e bus), while Pregel uses random hashing by default. Second, Medusa provides more fine-grained programming interfaces than Pregel, exposing fine-grained data parallelism on edges, vertices and messages. Finally, Medusa does not have the sophisticated design for distributed systems, such as failure handling.

More recently, the GraphLab2 project [12], [24] further decomposes the vertex-program abstraction into small pieces, which also offer fine-grained parallelism like our EMV model. Green-Marl [18] is another recent effort on easing the difficulty of optimizing GPU graph analysis algorithms, which uses domain-specific language (DSL) to provide developers a high level language interface. In comparison with Medusa, Green-Marl processes all vertices with a foreach loop in the order of BFS or DFS, and does not use message passing mechanisms of the GBSP model.

2.2 GPGPU

In this work, we adopt NVIDIA CUDA as our development platform. The GPU consists of an array

of streaming multiprocessors (SM). Inside each SM is a group of scalar cores. CUDA allows developers to write device programs, which are called *kernels*, to run on hundreds of GPU cores with thousands of threads. Each 32 of the massive number of threads are grouped as a warp and execute synchronously on one SM. Divergence inside a warp is supported but may introduce a severe performance penalty since different paths are executed serially. An important memory feature exposed by CUDA is called *coalesced accesses*. If memory requests issued by a warp fall into the same memory segment, they are coalesced into one, thus significantly improving memory bandwidth utilization. Different from common CPUs, the CUDA memory hierarchy includes a scratchpad memory called *shared memory* which has much lower latency than the device memory.

With massive parallelism, GPUs have been adopted to accelerate graph processing. Harish et al. [14] investigated the design and implementation of several most commonly used graph algorithms on GPUs, including BFS, single source shortest paths (SSSP) and all-pair shortest paths (APSP). Hong et al. proposed a virtual warp-centric [19] GPU BFS algorithm with optimization techniques such as deferring outliers to address irregularities of the graph data structure. Compared with Harish's work, the warp-centric method achieved notable speedup when the input graph is highly irregular. Luo et al. [28] and Merrill et al. [30] implemented BFS with queue structures to store the frontier vertices or edges in order to reduce excessive accesses. Most existing GPU graph processing studies focus on specific algorithms.

Both Medusa and our previous work Mars [15] are designed as programming frameworks to simplify parallel GPU programming with sequential interfaces. Medusa is specifically designed for graph processing. We have also addressed some inefficient designs of Mars, e.g., the graph-aware message passing mechanism for Medusa avoids the costly pre-counting result output mechanism in Mars.

3 Overview

The following two design goals guide our design to make a useful programming framework for different graph processing algorithms. Particularly, programmability is our first-class design goal, and our overall goal is to offer a highly programmable graph processing framework for different applications with reasonable performance.

We present our techniques for directed graphs, and the techniques are applicable to undirected graphs. In a directed graph, we define an edge $s \rightarrow t$, where s is the head vertex and t is the tail vertex. We say the edge is *associated* with s. Each vertex in the graph has a unique ID ranging in $[0, V-1]$, where V is the number of vertices in the graph. For the set of edges

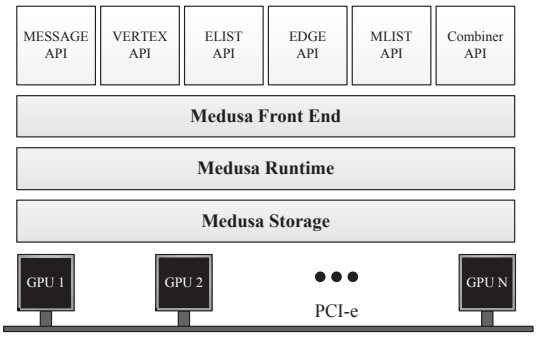

Fig. 1. An overview of Medusa.

associated with the same vertex, we assign a unique local ID for each edge ranging in $[0, d-1]$, where d is the out-degree of the vertex. d_{max} is defined as the maximum value of the out-degrees in the graph.

In the remainder of this section, we present the programming interface and workflow of Medusa, mainly from the developers' perspective.

3.1 Programming Interface

Figure 1 shows the system architecture of Medusa. Medusa is able to run on one or multiple GPUs in the same machine. In this section, we give an overview of the entire system from the developers' perspective on how they use Medusa. The detailed designs are described in Section 4.

Previous studies [10], [25]–[27], [29] have shown that the GBSP model greatly simplifies the composition of graph algorithms by offering a sequential programming interface oriented on individual vertices. This model is derived from the observation of two common access patterns in various graph applications. First, the processing of vertices and edges is often localized within neighboring vertices. Second, many graph applications have multiple iterations where many edges and vertices are accessed and updated within an iteration. Most GBSP-based systems provide a single vertex-based API.

The EMV model of Medusa enhances the current single vertex-based API design to support efficient and fine-grained graph processing on the GPU. In particular, Medusa offers the following two mechanisms for programmability and efficiency.

First, Medusa provides six device code APIs for developers to write GPU graph processing algorithms, as shown in Table 1. Each API is either for processing vertices ($VERTEX$), edges ($ELIST$, $EDGE$) or messages ($MESSAGE$, $MLIST$). Using these APIs, programmers can define their computation on vertices, edges and messages. The vertex and edge APIs can also send messages to neighboring vertices. The idea of providing six APIs is mainly for efficiency (The details are presented in Section 4.1).

Second, Medusa hides the GPU-specific programming details with a small set of system

TABLE 1
User-defined APIs in the EMV model

API Type	*Parameters*	*Variant*	*Description*
$ELIST$	Vertex v, Edge-list el	Collective	Apply to edge-list el of each vertex v
$EDGE$	Edge e	Individual	Apply to each edge e
$MLIST$	Vertex v, Message-list ml	Collective	Apply to message-list ml of each vertex v
$MESSAGE$	Message m	Individual	Apply to each message m
$VERTEX$	Vertex v	Individual	Apply to each vertex v
$Combiner$	Associative operation o	Collective	Apply an associative operation to all edge-lists or message-lists

TABLE 2
System provided APIs and parameters in Medusa

API/Parameter	*Description*
$AddEdge$ (void* e), $AddVertex$(void* v)	Add an edge or a vertex into the graph
$InitMessageBuffer$(void* m)	Initiate the message buffer
$maxIteration$	The maximum iterations that Medusa executes ($2^{31}-1$ by default)
$halt$	A flag indicating whether Medusa stops the iteration
$Medusa :: Run$(Func f)	Execute f iteratively according to the iteration control
$EMV<type>:: Run$(Func f')	Execute EMV API f' with $type$ on the GPU

provided APIs (Table 2). Particularly, Medusa provides $EMV < type >:: Run()$ to invoke the device code API, which automatically sets up the thread block configurations and calls the corresponding EMV user-defined function. Medusa allows developers to define an *iteration* by running multiple $EMV < type >:: Run()$ calls sequentially in one host function (invoked by $Medusa :: Run()$). The iteration is performed iteratively until predefined conditions are satisfied. Medusa offers a set of configuration parameters and utility functions for iteration control.

Given user-defined data structures and definitions of device code APIs, the Medusa front end automatically transforms them into compilable CUDA kernels and related device management code. The design goal of the front end is to hide GPU specific programming details. After the preprocessing using the front end, the program is compiled and linked with the Medusa libraries.

In the storage component, Medusa allows developers to initialize the graph structure by adding vertices and edges with two system provided APIs, namely $AddEdge$ and $AddVertex$. After initialization, the storage component stores the graph with the optimized graph layout on the GPU (Section 4). Note, the memory management on the GPU and data transfer between the GPU memory and the main memory is managed by Medusa, which is transparent to developers.

The Medusa runtime is responsible for executing the user-defined APIs in parallel on the GPU. Medusa offers two system provided APIs for execution, $Medusa :: Run$(Func f) and $EMV<type>:: Run$(Func f'). $Medusa :: Run$(Func f) is the main entry of the Medusa execution, and executes function f according to the iteration control policy, where f usually consists of an execution sequence of the EMV APIs. $EMV<type>:: Run$(Func f') executes an EMV user-defined API on the graph storage according to $type$ ($type \in$ {ELIST, EDGE, MLIST, MESSAGE, MLIST}).

```
Device code APIs:
/* ELIST API */
struct SendRank{
__device__ void operator() (EdgeList el,
Vertex v){
  int edge_count = v.edge_count;
  float msg = v.rank/edge_count;
  for(int i = 0; i < edge_count; i ++)
    el[i].sendMsg(msg);
}
/* VERTEX API */
struct UpdateVertex{
__device__ void operator() (Vertex v, int
super_step){
  float msg_sum = v.combined_msg();
  vertex.rank = 0.15 + msg_sum*0.85;
}
Data structure definitions:
struct vertex{
  float pg_value;
  int vertex_id;
}
struct edge{
  int head_vertex_id, tail_vertex_id;
}
struct message{
  float pg_value;
}
```

```
Iteration definition:
void PageRank() {
  /* Initiate message buffer to 0 */
  InitMessageBuffer(0);
  /* Invoke the ELIST API */
  EMV<ELIST>::Run(SendRank);
  /* Invoke the message combiner */
  Combiner();
  /* Invoke the VERTEX API */
  EMV<VERTEX>::Run(UpdateRank);
}
Configurations and API execution:
int main(int argc, char **argv) {
  ......
  Graph my_graph;
  /* Load the input graph. */
  conf.combinerOpType = MEDUSA_SUM;
  conf.combinerDataType = MEDUSA_FLOAT;
  conf.gpuCount = 1;
  conf.maxIteration = 30;
  /*Setup device data structure.*/
  Init_Device_DS(my_graph);
  Medusa::Run(PageRank);
  /* Retrieve results to my_graph. */
  Dump_Result(my_graph);
  ......
  return 0;
}
```

Fig. 2. User-defined functions in PageRank implemented with Medusa.

3.2 Medusa Workflow

There are three steps to implement a graph algorithm based on Medusa. First, the developer defines the basic data structures such as edge, message and vertex in C/C++ *struct*s. Second, the developer implements EMV APIs according to his/her application logic. Third, the developer composes the main program, including initializing the graph structure, configuring the framework parameters and invoking the customized EMV APIs with the system provided APIs (in Table 2).

Many graph computation tasks require multiple iterations until convergence. To support iterations,

Medusa provides two interfaces for controlling the number of iterations of the execution. Developers can use both of them for a more flexible iteration control. First, the developer can specify the maximum number of iterations, $maxIteration$. Medusa terminates when the number of iterations reaches the predefined limit. Second, Medusa has defined a global variable $halt$, which can be modified by the EMV APIs. By initializing $halt$ as false, the framework continues the iterations until any of the API instance sets $halt$ to be true. This is equivalent to all API instances needing to vote false to continue the iteration. This iteration control mechanism is also used in Pregel [29].

To demonstrate the usage of Medusa, we show an example of the PageRank implementation with Medusa, as shown in Figure 2. Data structures (e.g., *vertex*) are defined. The function *PageRank()* is composed of three user-defined EMV API function calls: an $ELIST$ type API (*SendRank*), a message $Combiner$ and a $VERTEX$ type API (*UpdateRank*). In the main function, we configure the execution parameters such as the $Combiner$ data type and operation type, the number of GPUs to use and the maximum number of iterations. *Init_Device_DS* automatically builds the graph data structures and copies them to the GPU. *Medusa::Run(PageRank)* invokes the *PageRank* function.

4 System Design

This section details the design and implementation of Medusa. The Medusa runtime involves advanced and complicated mechanisms and implementations in order to improve the efficiency with the constraint of preserving high programmability. Most runtime optimizations are entirely transparent to developers. Some implementations may be seemingly trivial for specific applications, but become challenging to integrate into a framework to support general graph processing operations. In particular, Medusa focuses on processing sparse graphs.

Table 3 presents a summary of the list of optimizations in Medusa and their respective advantages. The proposed optimizations enable Medusa to better exploit massive parallelism and memory features of the GPU while preserving the simple programming interface at the same time. For multi-GPU execution, the graph is partitioned using METIS [22]. Due to space limitations, we present the details on the GPU-transparent programming interface and graph layouts in Appendix A of the supplementary file.

4.1 Fine-Grained Graph APIs

Most GBSP model based systems provide a single vertex centered API. Programmers use the single vertex API to access all associated edges and messages (one typical access pattern is iterating edges/messages one by one). While the single vertex-based API design of the GBSP model has achieved good performance and programmability on distributed systems like Pregel [29], such coarse-grained designs are inefficient on GPUs due to execution divergence and irregular memory access. The vertex-based API exhibits severe divergence which makes it unsuitable for GPU execution. First, different vertices may have different numbers of edges, leading to different workloads on each API instance. Second, different number of received messages is another source of divergence. As for memory efficiency, the vertex-centric API makes the memory optimizations on edges and messages a challenging task.

To address those issues, we propose the EMV model as an extension of GBSP. It decouples the single vertex API into separate APIs which target individual vertices, edges or messages. Each GPU thread executes one instance of the user-defined API. The thread configuration such as the number of threads is tuned to maximize GPU utilization. The fine-grained data parallelism exposed by the EMV model can better exploit the massive parallelism of the GPU.

In addition, Medusa supports two variants of APIs for individual and collective operations of edges and messages associated with the same vertex. The collective APIs allow developers to access the elements in each *edge-list* (the set of edges associated with the same head vertex) or a *message-list* (the set of messages sent to the same vertex) sequentially. On the other hand, the individual APIs support operations on individual edges, vertices or messages and expose more parallelism. Medusa also provides a $Combiner$ interface, with which developers can apply an associative operator to all the elements of each edge-list and message-list. All these APIs require no parallel programming, and developers write conventional sequential code to implement those APIs.

The collective APIs forms a superset of the individual APIs in terms of expressibility. Operations which involve dependent computation (e.g., the computation on one edge depends on other edges in the same edge-list) can only be implemented by collective APIs. However, we have observed that many graph algorithms do not need dependent computation on the edge-lists or message-lists. Choosing individual graph elements yields better workload balance and more parallelism. Moreover, many dependent computations are associative operations, for example, PageRank sums the values of received messages of each vertex to update rank values. This enables us to use the $Combiner$ interface. The $Combiner$ interface is implemented as segmented scan, which has the load-balanced implementation on GPUs [33].

By default, Medusa applies the user-defined API on the vertices/edges on the entire graph. This may result in work-suboptimal algorithms for some

TABLE 3
Summary of techniques used in Medusa and their advantages

Problem	*Solution*	*Advantage*
Massive parallelism	EMV API	Fine grained parallelism for massive parallelism
Work efficiency	Queue-based implementation with our *SetActive* API	Allow developing more work-efficient algorithm
GPU specific programming details	Automatic GPU specific code generation	Eliminate the GPGPU learning curve
Graph layout	Novel graph representation	Better memory bandwidth utilization
Message passing efficiency	Graph-aware buffer scheme	Better memory bandwidth utilization and avoid message grouping overheads
Multi-GPU execution	Replication, memory transfer/computation overlapping	Alleviate PCI-e overheads

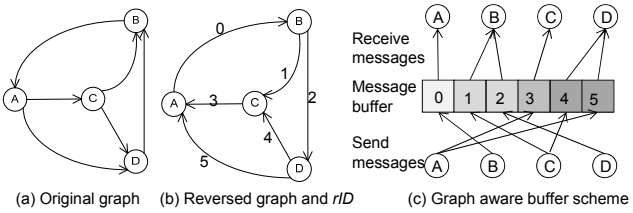

Fig. 3. Graph aware buffer scheme.

applications such as BFS and SSSP. In order to allow developers to implement work-efficient algorithms, we have added an additional device code API called *SetActive*(vertexID/edgeID), and developers are able to indicate whether a vertex or an edge is active in the next EMV API call. The active edges/vertices are maintained in a dynamic queue. We implement the queue structure following the previous study [30] , where we do not have specific order of enqueuing vertices or edges. In subsequent API invocations, developers are able to apply the EMV APIs to the active vertices and edges only and thus implement more work efficient algorithms. With the *SetActive* API, we have implemented work-efficient BFS and SSSP algorithms and experimentally evaluated their performance (described in Section 5).

4.2 Graph-Aware Buffer Scheme

Messages are temporarily stored in buffers, allocated by calling the system provided API $InitMessageBuffer$. We first discuss two basic buffer schemes, array- and list-based buffer schemes with respect to the memory efficiency of sending and receiving messages.

The array-based buffer scheme is to allocate an array for message storage. Implementing the buffer with a fixed-sized array, this buffer scheme requires the information of the buffer size as well as the output positions for each message to avoid conflicts. Even worse, if the messages to the same vertex are not stored consecutively, Medusa needs a grouping operation in order to support message processing in collective user-defined APIs. In contrast, the list-based buffer scheme relies on dynamic memory allocation. We adopt a hash table with dynamic memory allocation [17] to store messages. This method eliminates the pre-computation of message sizes and the grouping operation in the array-based storage scheme. However, the dynamic hash table requires atomic operations and the accesses to the hash table are minimally coalesced.

Neither of the two buffer schemes can achieve good performance on both storing and processing the messages. That motivates us to develop a buffer scheme to capture the best of both worlds. We observe that the messages are usually sent/received along the edge in the EMV model. Given the maximum number of messages that can be sent along each edge, we can compute (1) the maximum total number of messages; (2) the maximum number of messages that each vertex can receive. The awareness of the graph structure helps us to allocate the buffer, and to obtain the write positions of the messages along each edge.

To avoid the grouping operation, we ensure that the write positions of the messages sent to the same vertex are consecutive. This is achieved with the idea of "reversed edge indexed message passing." While loading the graph, Medusa constructs a reverse graph by swapping the head and tail of each edge. The reverse graph is stored in AA format. We assign an rID (*reverse ID*) for each edge in the original graph, whereby the rID value of each edge equals the index of its reverse edge in the adjacency array. Figure 3(b) shows the rID value for each edge in an example graph.

The rID definition has an important property: *the rID values for the edges with the same tail vertex are consecutive integers*. For example, the $rIDs$ of the edges with the same tail vertex D in the original graph in Figure 3 are 4 and 5. We take advantage of this property to ensure that the write positions of the messages sent to the same vertex are consecutive.

The graph aware buffer scheme works as follows. First, a message buffer with $(E \times m)$ entries is allocated, where m is the maximum number of messages that can be sent via each edge. For example, m is equal to one in PageRank. Medusa allows developers to set the m value. Second, when a message is sent along an edge and the rID of that edge is k, the start position for the message generation

is $(k \times m)$ in the message buffer. Figure 3(c) shows an example of the graph aware buffer scheme for PageRank ($m = 1$).

When sending messages, the rID values give the write locations for the message along each edge. When receiving messages, the messages for the same vertex are already stored together. Thus, all the messages are already grouped by the tail vertex. This is because of the property of the rID values. Thus, no additional grouping operation is needed. Moreover, the message buffer uses an array, and thus the memory efficiency of message processing is much higher than that of the list-based buffer scheme, as demonstrated in our experiments.

4.3 Multi-GPU Execution

We first present a basic implementation of the multi-GPU extension, and then our multi-hop replication optimization to reduce the data transfer cost in the PCI-e bus. Our multi-hop replication scheme is inspired by stencil operation optimizations [6], [11]. Differently, we target at partitioned graphs in multi-GPU environments.

Replication. To accommodate multi-GPU graph processing, we divide the graph into equal-sized partitions and store each partition on one GPU. We adopt the widely used graph partitioning tool METIS [22] to partition the input graph. Clearly, the quality of graph partitioning has great effect on the amount of data transfer among different GPUs. It is our future work to investigate other graph partitioning algorithms.

Figure 4(a) shows an example with three GPUs. A directed graph is partitioned into three parts and each part is stored on one GPU. In the design of Medusa, messages are passed along edges. Graph partitioning introduces cross-partition edges, whose head and tail vertices are in different partitions and hence stored on different GPUs.

In order to apply EMV APIs on each graph partition, we maintain replicas of the head vertices of all cross-partition edges in the partitions where the tail vertices reside (we call it the *tail partition*). Each cross partition edge is replicated in its tail partition, as shown in Figure 4(b). Thus, messages are emitted directly from the replicas and every edge can access its head and tail vertices directly. The execution of EMV APIs is performed on each partition independently. After the execution, we update the replicas on each graph partition. The update requires the costly PCI-e data transfer, which can become a bottleneck for some application such as BFS. We therefore propose a multi-hop replication scheme as well as overlapping on the computation and data transfer to alleviate the overhead of PCI-e data transfer.

Multi-hop Replication Scheme. When the inter-GPU communication time is dominant in the total

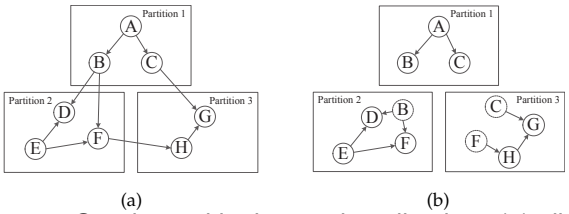

Fig. 4. Graph partitioning and replication: (a) direct partitioning; (b) replication for EMV executions (dashed circles represent the replicas).

execution time, reducing the time cost of communication can significantly improve the application performance. The multi-hop replication scheme presented alleviates the overhead of inter-GPU communication by reducing the number of times of replica update.

Instead of only maintaining head vertices of cross-partition edges as replicas, we introduce the second hop replicas by replicating tail vertices of the first hop replicas. Similarly, more hops of replicas can be added to each partition. We call this approach as *multi-hop replication scheme*. Our multi-hop replication scheme is inspired by stencil operation optimizations [11]. Due to the message propagation nature of the EMV model, replica update only needs to be carried out after every n iterations if there are n hops of replicas. We call n iterations as a round and one *round* has n stages. As the stages are carried out outer hops of replicas are marked as "outdated". That essentially uses the eventual consistency model, and the data are consistent after each round.

Figure 5 shows an example of the same graph as in Figure 4. Now *Partition 2* and *Partition 3* both maintain two-hop replication. The replicas need to be updated every two iterations, reducing the number of replica update by a half. In the first stage of each round, Medusa APIs are applied to all vertices in each partition. After that, the second hop replicas are outdated and are not processed in the second stage. After each round, the replicas are updated and a new round start.

As described above, increasing the number of replica hops can reduce the number of times of updating replicas. However, this scheme is not guaranteed to be beneficial compared with the basic replication scheme since more replicas and edges need to be processed. For example, maintaining multiple hops of replicas for dense graphs or small-world graphs with a small diameter can lead to explosive growth of replica vertices. However, since Medusa mainly deals with sparse graphs, multi-hop replication can be beneficial. For a given graph, we estimate the benefits of all possible hop numbers within the storage constraint and select the best one. Medusa uses a cost model to estimate the benefits of all possible hop numbers. More details can be found in Appendix A.3.

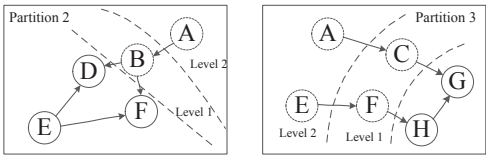

Fig. 5. Graph partitioning with multi-hop replication.

TABLE 4
Characteristics of graphs used in the experiments

Graph	Vertices (10^6)	Edges (10^6)	Max d	Avg d	σ
RMAT	1.0	16.0	1742	16	32.9
Random (Rand)	1.0	16.0	38	16	4.0
BIP	4.0	16.0	40	4	5.1
WikiTalk (Wiki)	2.4	5.0	100022	2.1	99.9
RoadNet-CA (Road)	2.0	5.5	12	2.8	1.0
kkt_power (KKT)	2.1	13.0	95	6.3	7.5
coPapersCiteseer (Cite)	0.4	32.1	1188	73.9	101.3
hugebubbles-00020 (Huge)	21.2	63.6	3	3.0	0.03

5 Evaluation

5.1 Experimental Setup

We have conducted the evaluations on a workstation equipped with four NVIDIA Tesla C2050 GPUs, two Intel Xeon E5645 CPUs (totally 12 CPU cores at 2.4GHz) and 24GB RAM.

Our workloads include a set of common graph processing operations for manipulating and visualizing a graph on top of Medusa. The graph processing operations include PageRank, breadth first search (BFS), maximal bipartite matching (MBM), and single source shortest paths (SSSP). In order to assess the queue-based design in Medusa, we have implemented two versions of BFS: BFS-N and BFS-Q for the implementations without and with the usage of *SetActive* APIs, respectively. Similarly, we have also implemented two versions of SSSP: SSSP-N and SSSP-Q without and with the usage of *SetActive* APIs, respectively. The implementation details are presented in Appendix B of the supplementary file. In the remainder of this section, we use "Medusa" to refer to the better-performing implementation of the two versions on BFS and SSSP, unless we specify "-N" and "-Q" explicitly.

Our experimental dataset includes two categories of sparse graphs: real-world and synthetic graphs. Table 4 shows their basic characteristics. We use the GTgraph graph generator [2] to generate power-law graph RMAT and Random graph. To evaluate MBM, we generate a synthetic bipartite graph (denoted as BIP), where vertex sets of two sides have one half of the vertices and the edges are randomly generated. The real world graphs are publicly available [1], [3].

All the experiments are executed for ten runs and the average execution time is reported. The difference among runs for the same experiment is smaller than 2%. For BFS and SSSP, we randomly choose 100 source vertices and report the average execution time.

5.2 Comparison with Manual Implementations

We first compare the Medusa BFS and SSSP implementations with manual implementations of GPU graph processing: Harish's work [14] and Hong's work [19].

Harish's work provides an open-source implementation of BFS and SSSP using CUDA and we tune the thread configuration and shared memory optimizations according to the C2050 Fermi architecture. We use it as the basic implementation. We implement the virtual warp-centric BFS proposed in Hong's work [19]. The underlying difference between the Medusa implementation and the warp-centric method is that Medusa applies L threads to a vertex if that vertex has L edges, while the warp-centric method applies a virtual warp to a vertex. As a result, our method incurs more memory accesses because we check the head vertex status for every edge.

Table 5 shows the traversed edges per second (TEPS) comparison between the three implementations of BFS. Compared to the basic implementation, Medusa performs better on all graphs except KKT. Although Medusa incurs more memory access and runtime overhead than the highly optimized warp-centric method, Medusa outperforms warp-centric on some graphs and degrades the performance on other graphs. Note that the reported results of the warp-centric approach are better than those in the original paper [19], mainly because the GPU in our experiment is more powerful.

Figure 6 shows the performance comparison between Medusa and basic implementation of SSSP. Medusa provides comparable performance with the basic implementation except on Road and Huge. On large-diameter graphs such as Road and Huge, the performance of Medusa-based SSSP is notably worse than that of the basic implementation. This is because the *Combiner* API invocation in SSSP takes a large part of its execution time and that overhead is almost fixed for every iteration.

Programmability is difficult for a quantitative comparison. As a start, we show the programmability comparisons on some major implementation issues of GPU programs in Table 6. Medusa simplifies GPU programming for graph processing, by significantly reducing the number of GPU-related source code lines written by developers. This is because Medusa hides the GPU programming complexity by offering a small set of user-defined APIs. For example, developers only need to write 7 and 11 lines of source code for defining the APIs in BFS-Q and SSSP-Q, respectively, whereas the basic implementation [14] has 56 and 59 lines of GPU-related code. Moreover, compared to manual implementations, Medusa requires no parallel or GPU specific programming.

Overall, Medusa offers reasonable performance in comparison with manual implementations. With

TABLE 5
Traversed edge per second (10^6 TEPS) comparison with manual implementations [14], [19].

	Basic	Warp-centric	Medusa
Wiki	61.4	152.9	1091.1
Road	26.2	45.7	63.5
RMAT	593.2	971.1	895.8
Rand	648.6	844.95	765.8
Huge	5.7	1.3	68.1
KKT	480.7	175.7	351.5
Cite	1460.4	1503.1	2686.7

TABLE 6
Coding complexity of Medusa implementation and manual implementations.

	Baseline	Warp-centric	Medusa (N/Q)
GPU code lines (BFS)	56	76	9/7
GPU code lines (SSSP)	59	N.A.	13/11
GPU memory management	Yes	Yes	No
Kernel configuration	Yes	Yes	No
Parallel programming	Thread	Thread+Warp	No

different design goals, Medusa is to offer good programmability with reasonable performance, whereas manual implementations usually do not consider programmability. Some techniques that are applicable to manual implementations may not be applicable to Medusa, if they hurt programmability.

We present more experimental results on BFS. Table 7 shows the comparison on BFS between Medusa-based implementation and the Contract-Expand and Hybrid approaches in Merrill et al.'s paper [30]. The Hybrid approach is more optimized than the Contract-Expand approach. For more details of those approaches, we refer the reader to the original paper [30]. The design and implementation of Medusa-based BFS is similar to the Contract-Expand approach, but targets at general graph processing. Overall, Medusa-based implementation can be slower than the Contract-Expand approach on some graphs such as Huge and KKT, and can be faster on other graphs such as Cite. On the other hand, Medusa-based implementation is slower than the Hybrid approach on all the three graphs. Compared with various specific optimizations for BFS, Medusa involves

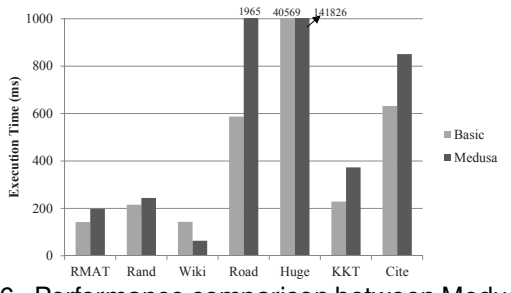

Fig. 6. Performance comparison between Medusa and existing GPU implementation of SSSP [14].

TABLE 7
Traversed edge per second (10^9 TEPS) comparison with Merrill et al.'s paper [30].

	Medusa	Contract-Expand [30]	Hybrid [30]
Huge	0.1	0.4	0.4
KKT	0.4	0.7	1.1
Cite	2.7	1.3	3.0

considerate runtime overhead in supporting general graph processing, for example, message passing based mechanisms.

5.3 Experiments on Efficiency

Overall comparisons. We implement the graph processing operations with MTGL [7], as the baseline for graph processing on multi-core CPUs.

The BFS and PageRank implementations are offered by MTGL and we implement the Bellman-Ford algorithm for single source shortest paths and a randomized maximal matching algorithm [4] using the MTGL APIs. We tuned the number of threads in MTGL and report the best result obtained when the number of threads was 12 on our machine. MTGL running on 12 cores is on average 3.4 times faster than that running on one core. Due to the memory intensive nature of graph algorithms, the scalability of MTGL is limited by the memory bandwidth.

Figure 7 shows the speedup for Medusa over MTGL running on 12 cores. The *speedup* is defined as the ratio between the elapsed time of the CPU-based execution and that of Medusa-based execution. PageRank is executed with 100 iterations. Medusa is significantly faster than MTGL on most comparisons and delivers a performance speedup of 1.0–19.6 with an average of 5.5 (we report the better results of the two implementations of BFS and SSSP, respectively). On some graphs such as Road, BFS-N is notably slower than MTGL-based BFS, because the work-inefficient issue of BFS-N is exaggerated on the graphs with large diameter.

The work-efficient BFS and SSSP algorithms (BFS-Q and SSSP-Q) achieve better performance on the graphs with large diameters, and can degrade the performance in some cases (e.g., Rand, Wiki and KKT) due to the computation and memory overhead in maintaining the queue structure. This is consistent with the previous studies [19]. Currently, we leave the decision on whether to use the *SetActive* API to the users. In the future work, we consider whether this decision can be made automatically in Medusa.

6 Conclusions

In this paper, we address the efficiency and programmability of GPU-based parallel graph processing by developing a programming framework named Medusa. Medusa embraces an optimized runtime

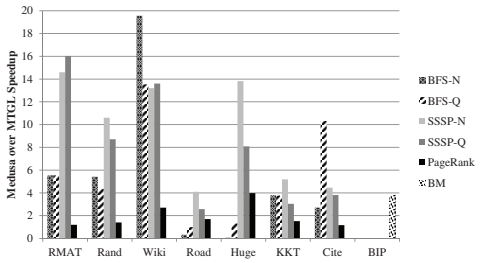

Fig. 7. Performance speedup of Medusa running on the GPU over MTGL [7] running on 12 cores.

system to hide the programming complexity of implementing parallel graph computation tasks for GPUs. Developers only need to write sequential programs to implement a small set of APIs. On an NVIDIA Tesla C2050 GPU, Medusa-based implementations are 5.5 times on average faster than the parallel MTGL based implementations on two Intel six-core CPUs. Moreover, with much less coding complexity, Medusa achieves comparable or even better performance than existing manual implementations. As for future work, we are interested in evaluating Medusa in other architectures such as Intel Xeon Phi and extending Medusa to distributed environments.

The source code of Medusa is available at http://code.google.com/p/medusa-gpu/.

7 Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments, and Pawan Harish for providing the source code for CUDA-based BFS and shortest paths. This work is partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) and an NVIDIA Academic Partnership Award.

References

[1] 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10/index.shtml, accessed on Feb 17th, 2013.

[2] GTGraph generator. http://www.cse.psu.edu/~madduri/software/GTgraph/index.html, accessed on Feb 17th, 2013.

[3] Stanford large network dataset collections. http://snap.stanford.edu/data/index.html, accessed on Feb 17th, 2013.

[4] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11:319–352, November 1993.

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

[6] F. Bassetti, K. Davis, and D. J. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[7] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS*, March 2007.

[8] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4), Nov. 2011.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] M. Delorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, and A. Dehon. GraphStep: A system architecture for sparse-graph algorithms. In *FCCM*, 2006.

[11] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, 2005.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[13] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[14] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, 2007.

[15] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *PACT*, 2008.

[16] G. He, H. Feng, C. Li, and H. Chen. Parallel SimRank computation on large graphs with iterative aggregation. In *SIGKDD*, 2010.

[17] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing parallel program portable between CPU and GPU. In *PACT*, 2010.

[18] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *ASPLOS*, London, England, UK, 2012.

[19] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, 2011.

[20] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop. Technical Report CMU-ML-08-117, Carnegie Mellon University, 2008.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system – implementation and observations. In *ICDM*, 2009.

[22] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[23] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Graphics hardware*, 2008.

[24] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[25] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *MLG*, 2010.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, April 2012.

[28] L. Luo, M. Wong, and W.-m. Hwu. An effective GPU implementation of breadth-first search. In *DAC*, 2010.

[29] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[30] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPoPP*, 2012.

[31] J. D. Owens, D. Luebke, N. K. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics, State of the Art Reports*, 2005.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford InfoLab. Technical report, 1999.

[33] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. *NVIDIA, Tech. Rep. NVR-2008-003*.

[34] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[35] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *CVPR Workshops*, June 2008.

J ianlong Zhong received the bachelor degree in software engineering from Tianjin University (2006-2010), and is now a PhD candidate in School of Computer Engineering of Nanyang Technological University, Singapore. His research interests include GPU computing and parallel graph processing.

B ingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, cloud computing, and database systems.