

# Medusa: Simplified Graph Processing on GPUs

Jianlong Zhong, Bingsheng He

**Abstract**— Graphs are the de facto data structures for many applications, and efficient graph processing is a must for the application performance. Recently, the graphics processing unit (GPU) has been adopted to accelerate various graph processing algorithms such as BFS and shortest path. 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 GPU 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 much 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 the manually tuned GPU graph operations.

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

## 1 INTRODUCTION

GRAPHS are de facto 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. Users 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 [26], shortest path and even their customized variants (for example, users may apply different application logics on 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 [25]. The GPU, originally designed as a co-processor for graphics rendering, has recently evolved into a powerful many-core processor for general-purpose computation. New-generation GPUs can have over an order of magnitude higher memory bandwidth and

higher computation power (in terms of GFLOPS) than CPUs. Thus, the GPU has been used as an accelerator for various graph processing applications [9], [11], [15], [33]. While those GPU-based solutions have demonstrated significant performance improvement over the CPU-based implementations, they are limited to specific graph operations. Developer 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, we 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. Since graph applications usually involve irregular accesses to the graph data, careful designs on the data layout and memory accesses are the 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 those 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. Like existing programming frameworks such as MapReduce [3] and its variant on the GPU [10], Medusa provides

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

a set of APIs for developers to implement their applications. The APIs are oriented at the graph edges and vertices 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 [9], [11], [15], [33]. 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 PCI-e 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 the Medusa-based implementation with the CPU-based and GPU-based manual implementations. The manual GPU implementations are either from existing code [9], or our homegrown implementation adopting recent graph processing techniques [13].

Our experimental results show that: (1) Medusa simplifies programming GPU graph processing algorithms in terms of significant reduction on 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 3.8 and 1.3 times faster than on a single GPU for PageRank and BFS, respectively.

**Organization.** The remainder of this paper is organized as follows. Section 2 reviews the related work on graph processing as well as GPGPU. 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

There has been a lot of research work on graph processing and GPGPU. This section reviews those most relevant on parallel and distributed graph processing and GPGPU.

### 2.1 Graph Processing

Graph computations inherently have fine-grained data parallelism. They tend to perform batch and often iterative operations where a large number of vertices and/or edges are accessed and the same operations are carried out repeatedly. For example, in an iteration of PageRank [26], each vertex propagates its rank to its neighbors along outgoing edges, and collect ranks from incoming edges to calculate its new rank.

Parallel algorithms have been a classical way to improve the performance of graph processing. Software libraries like Parallel BGL [7] and MTGL [23] are provided to construct parallel graph algorithms. Recently, Pingali and Kulkarni et al. [17], [27] studied the generalized data-parallelism namely *amorphous data-parallelism* to deal with irregular data structures.

Previous studies [4], [18], [20] have observed that many common graph algorithms can be formulated using a form of the bulk synchronous parallel (BSP) model specific for graph processing (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 was adopted by the distributed graph processing framework Pregel [20] in Google. In each iteration of the GBSP execution, Pregel applies a user-defined function *Compute()* on each vertex in parallel. 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 PIC-E data transfer, 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.

### 2.2 GPGPU

In this work, we adopt the Nvidia CUDA [24] 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 amount of threads are grouped as a warp and execute synchronously

on one SM. Divergence inside a warp is supported but may introduce severe performance penalty since different paths are executed serially. An important memory feature exposed by CUDA is called *coalesced accesses* [24]. 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. [9] investigated the design and implementation of several most commonly used graph algorithm on GPUs including BFS, single source shortest path (SSSP) and all-pair shortest path (APSP). Hong et al. proposed a virtual warp-centric [13] method and further optimization techniques such as deferring outlier 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. [19] and Merrill et al. [21] implemented BFS with queue structures to store the frontier vertices or edges in order to reduce the excessive accesses. Most existing GPU graph processing studies focus on specific algorithms. The high performance improvement is achieved at the cost of programming complexity, which requires deep understanding on the GPU architecture. Writing a new graph computation program usually needs to revisit or invent those intensive optimizations again for efficiency.

There are a few research studies on utilizing multiple GPUs within the same machine. Spafford et al. [31] studied the non-uniform memory access and contention effects in multi-GPU systems. Schaa and Kaeli [28] provided a method to allow programmers to predict execution time of GPU applications under different multi-GPU configurations. Kim et al. [16] proposed a multi-GPU abstraction which offers a single virtual GPU image. However, their method shows poor performance for algorithms with irregular memory accesses. Medusa exploits graph partitioning and utilizes graph-centric optimizations for reducing the data transfer overhead among multiple GPUs.

### 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 a reasonable performance.

- **Programmability.** Leveraging GPUs for computation requires long learning curve and sophisticated parallel programming skills. Medusa

TABLE 1  
User-defined APIs in the EMV model

API Type	Parameters	Variant	Description
<i>ELIST</i>	Vertex $v$ , Edge-list $el$	Collective	Apply to edge-list $el$ of each vertex $v$
<i>EDGE</i>	Edge $e$	Individual	Apply to each edge $e$
<i>MLIST</i>	Vertex $v$ , Message-list $ml$	Collective	Apply to message-list $ml$ of each vertex $v$
<i>MESSAGE</i>	Message $m$	Individual	Apply to each message $m$
<i>VERTEX</i>	Vertex $v$	Individual	Apply to each vertex $v$
<i>Combiner</i>	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
<i>AddEdge</i> (void* $e$ ), <i>AddVertex</i> (void* $v$ )	Add an edge or a vertex into the graph
<i>InitMessageBuffer</i> (void* $m$ )	Initiate the message buffer
<i>maxIteration</i>	The maximum iterations that Medusa executes ( $2^{31} - 1$ by default)
<i>halt</i>	A flag indicating whether Medusa stops the iteration
<i>Medusa :: Run</i> (Func $f$ )	Execute $f$ iteratively according to the iteration control
<i>EMV &lt;type&gt; :: Run</i> (Func $f'$ )	Execute EMV API $f'$ with $type$ on the GPU

should hide the complexity of GPU programming from developers, and enable developers to write sequential code to implement their graph applications.

- **Efficiency.** The GPU will have more cores over time, and memory latency continues to be an important performance factor for GPU computation. Medusa should exploit the massive thread parallelism of the GPU, with memory optimizations on improving the overall 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

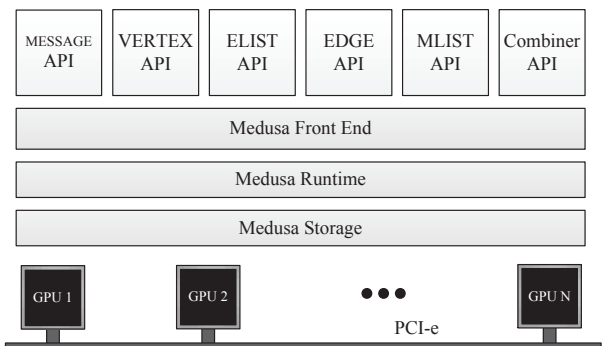


Fig. 1. An overview of Medusa.

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

### 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 users' perspective on how they use Medusa. The detailed designs are described in Section 4.

We focus on the sparse graph applications like web and social networks. We also observe two common access patterns in various graph applications. First, the update of vertices and edges are localized within neighboring vertices. Second, many graph applications have multiple iterations where many edges and vertices are accessed and updated within an iteration. Based on those two observations, Medusa offers the following two mechanisms to ease programmability.

First, Medusa provides six device code APIs for developers to write graph processing algorithms for the GPU, 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.

Second, for iteration control and other functionalities, Medusa provides a set of configuration parameters and utility functions as library calls (Table 2).

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 codes. 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 through 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.

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  $f$

#### 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 definition:

```
struct vertex
{
  float pg_value;
  int vertex_id;
}
struct edge
{
  int head_vertex_id;
  int tail_vertex_id;
}
struct message
{
  float pg_value;
}
```

#### Iteration Definition:

```
void PageRank() {
  /* Initiate message bufer 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 graph using
    AddVertex() and AddEdge()
  */
  conf.combinerOpType = MEDUSA_SUM;
  conf.combinerDataType = MEDUSA_FLOAT;
  conf.gpuCount = 1;
  conf.maxIteration = 30;
  /* Initialize device data structure. */
  Init_Device_DS(my_graph);
  Medusa::Run(PageRank);
  /* Retrive results to my_graph. */
  Dump_Result(my_graph);
  ... ..
  return 0;
}
```

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

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  $type$  ( $type \in \{ELIST, EDGE, MLIST, MESSAGE, MLIST\}$ ).

### 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 *struct*. Second, the developer implements EMV APIs according to his/her application logics. 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 till 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 need

to vote false to continue the iteration. This iteration control mechanism is also used in Pregel [20].

To demonstrate the usage of Medusa, we show an example of the PageRank implementation with Medusa, as shown in Figure 2. Data structures like *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. 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 be integrated into a framework to support general graph processing operations.

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.

### 4.1 Fine-Grained Graph APIs

Previous studies [4], [18], [20] have showed that GBSP model greatly simplifies the composition of graph algorithms by offering a sequential programming interface centered on each individual vertex. 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-API design have achieved good performance and programmability on distributed systems or multi-core CPU environments, it is inefficient for GPUs due to divergence and memory efficiency. The single vertex API exhibits severe divergence which makes it unsuitable for GPU execution. First, different vertices may have different number 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 “Edge-Vertex-Message (EMV)” model as an extension of GBSP. EMV is a finer-grained model, specifically tailored for the GPU architecture. 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 the GPU utilization [24]. The fine-grained data parallelism exposed by the EMV model can better exploit the massive parallelism of the GPU.

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 the an *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 operation, for example, PageRank sum 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 load balanced implementation on GPUs [29].

### 4.2 GPU Transparent Programming Interface

To leverage coalesced memory access feature, Medusa uses structure of array (SOA) data arrangement instead of array of structure (AOS) extensively. Programming with SOA diverts users from the natural way of thinking about data [30]. To simplify the programming interface, Medusa allows customized data structures such as vertex and edge to be defined using C/C++ *struct*. Medusa adopts a source-to-source transformation tool to generate SOA code from the *struct* definition and generates related memory allocation and data transfer code.

We use the same PageRank example to illustrate the code generation process. The top of Figure 3 gives the



TABLE 3  
Summary of techniques used in Medusa and their advantages

Problem	Solution	Advantage
Massive parallelism	EMV API	Fine grained parallelism for massive parallelism.
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 grouping overheads
Multi-GPU execution	Replication, memory transfer/computation overlapping	Alleviate PCI-e overheads

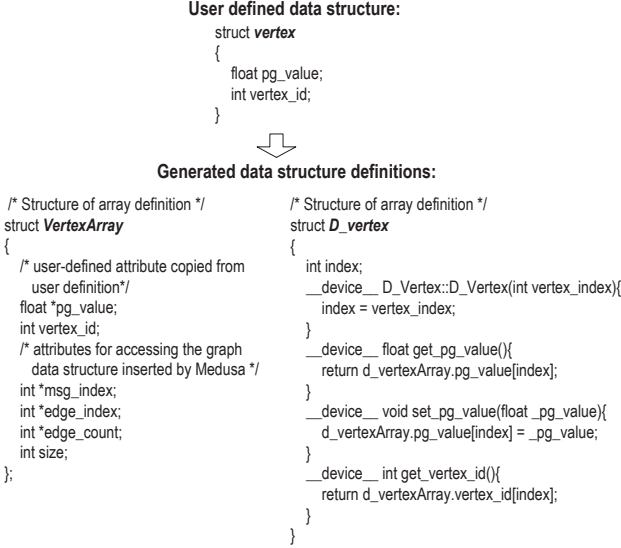


Fig. 3. Device data structure definitions and CUDA kernels generated from front end for PageRank.

definition for vertex. A naive way of defining vertex array is to use AOS, which suffers a lot of uncoalesced accesses. The bottom of Figure 3 shows the generated device data structure definitions from the user defined *struct*. Medusa copies the attributes from user code and generates pointer for the SOA definition. Also, Medusa generates a series of *get* and *set* functions for accessing the generated SOA.

Besides AOS transformation, the Medusa frontend automatically generates GPU related codes including device memory management and kernel invocation etc. Providing simplified sequential interfaces and automatic code generation reduces learning curves from developers and improves the programming productivity. Although the GPGPU programming environment has been dramatically improved over the years, it still requires long learning curves on programming and optimizations. Medusa is designed to ease this pain and increase productivity. One example is that our colleague has successfully developed a system for simulating information propagation with Medusa [14]. It took him about two weeks only, given that he has no GPU programming experience before.

### 4.3 Storage Layout Optimizations

We start with two basic storage layouts: vertex- and edge-oriented storage layouts. The vertex-oriented storage layout is the classic adjacency array (AA). Each vertex is represented as  $\langle id, d, edge-list \rangle$ , where  $id$  is the vertex ID, and  $d$  is the out-degree of the vertex. Each edge is represented as the ID of its tail vertex, and other associating data. As shown in Figure 4 (a), AA consists of two arrays, *Edge* for storing all the edge-lists, and *Start* for the starting position of each edge-list in *Edge*. The problem of this layout is that concurrent accesses to the adjacency array are scattered in different memory segments, which induces excessive uncoalesced memory accesses.

The edge-oriented storage layout (ESL) attempts to arrange the edge storage to exploit the coalesced memory access feature. The edge-oriented storage layout stores all the edges with the same local ID consecutively into an array. Since we require fast accesses on getting all the edges for a vertex in the collective *ELIST*, we store the edges according to the ascending order of their head vertex IDs. The edge-oriented layout virtually forms a 2-D matrix,  $e_{k,i}$  ( $0 \leq k < d_{max}$ ,  $0 \leq i < V$ ).  $e_{k,i}$  stores the edge with a local ID  $k$  for the head vertex with ID  $i$ . With edge-oriented layout, memory accesses to the edges with the same local ID are coalesced accesses. However, if the out-degrees of vertex have a large variance (e.g., in the power-law graphs), lots of space in the edge-oriented storage layout is wasted and memory transactions are not fully utilized.

Capturing the advantage of both basic layouts results in a hybrid layout (HY): we store the edges with local ID from 0 to  $t$  in ESL, and the remainder of the graph using the AA.  $t$  is a tuning parameter according to the distributions of out-degrees on the vertices of the graph. For power-law graphs, we set the suitable  $t$  such that most vertices with a small out-degree are accessed according to ESL; the vertices with large out-degrees are accessed according to AA.

The problem of the hybrid layout is that it requires tuning threshold value on AA and ESL and it still has the uncoalesced memory access to the part stored in AA. We further develop a column-major adjacency

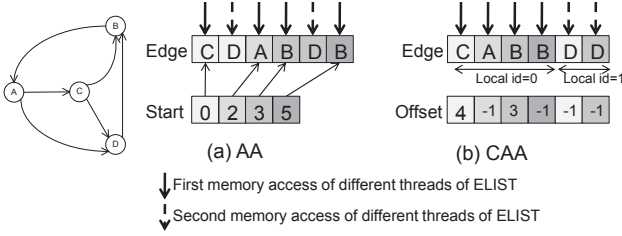


Fig. 4. Graph layouts: AA and CAA.

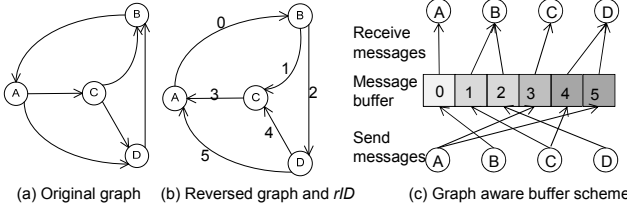


Fig. 5. Graph aware buffer scheme.

array (CAA) to maximize the coalesced memory accesses. The basic idea is to store the edges in the adjacency array such that the memory accesses to those edges are coalesced. The CAA storage has two arrays, *Edge* for storing edges, and *Offset* for storing the offset information. In the *Edge* array, we store the edges with the same local ID consecutively in the ascending order of the head vertex IDs. The *Offset* array has the same number of entries as the *Edge* array. Suppose the local ID of edge  $Edge[i]$  is  $l$  ( $0 \leq i < E$ ,  $E$  is the number of edges in the graph),  $Offset[i]$  stores the relative offset of the edge with local ID  $(l+1)$ . If there is no edge with a local ID  $(l+1)$  with the head vertex, we set  $Offset[i]$  to be  $-1$ . Otherwise, the edge is located with  $(i + Offset[i])$  in the *Edge* array.

Figure 4 illustrates an example of CAA layout in comparison with basic adjacency array storage. The edges in CAA are stored in the order of their head vertices A, B, C and D. For example, the edges of vertex A are stored in the first and the fifth entry of the *Edge* array, in Figure 4 (b). We also illustrate the memory access pattern for executing the collective *ELIST* when the number of threads is four. The memory accesses for AA are not consecutive among the threads, and those for CAA are consecutive and exploit the coalesced memory access feature of the GPU.

To reduce load imbalance among threads within one thread block, the vertices can be reordered such that vertices processed by the same thread block have more balanced number of edges. By launching sufficiently large number of thread blocks, SMs tend to have more balanced loads.

#### 4.4 Graph-Aware Buffer Scheme

Messages are temporarily stored in the buffer, initiated by calling 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 [12] to store messages. This method eliminates the pre-computation on the message size 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 hardly 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”. We assign an *rID* (reverse ID) for each edge in the reverse of the graph. The *rID* is defined to be the access rank of a BFS on the *reversed* graph. The BFS starts with the vertex with ID zero, and the rank starts from zero. The *rID* value of each edge is calculated during the graph initialization. Figure 5 (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 targeting to 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 5 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

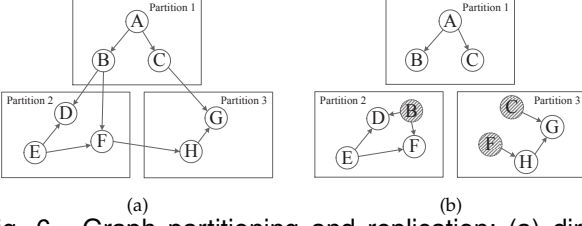


Fig. 6. Graph partitioning and replication: (a) direct partitioning; (b) replication for EMV executions (shaded circles represent the replicas).

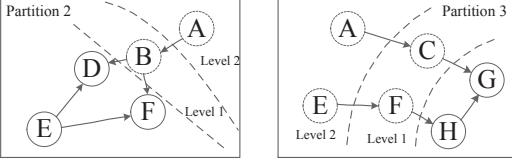


Fig. 7. Graph partitioning with multi-hop replication.

is  $(k \times m)$  in the message buffer. Figure 5 (c) shows an example of the graph aware buffer scheme for PageRank ( $m = 1$ ).

When sending messages, the  $rID$  values give the write location 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.

#### 4.5 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 [2], [5]. Differently, we target at partitioned graphs in multi-GPU environments. At last, we take advantage of the computation and data transfer overlapping capability of the latest GPUs to further reduce the cost of maintaining replicas.

**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 message passing between different GPUs. It is our future work to investigate other graph partitioning algorithms.

Figure 6 (a) shows an example with three GPUs. A directed graph is partitioned into three parts and each part is stored on one GPU. As described in section 4.1, 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 6 (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 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 [5]. 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 7 shows an example of the same graph as in Figure 6. 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 not 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 can lead to explosive growth of replica



vertices. However, since Medusa deals with sparse graphs, multi-hop replication can be beneficial. For a give 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.

The total time cost of  $n$ -hop replication scheme is the sum of computation time and the replica update time. For each round of  $n$  stages, the  $n$  hops of replicas need to be updated only once. For an input graph, we can obtain the number of replicas and then we calculate the time cost of updating the replicas based on the PCI-e bus bandwidth. Comparing this with the time cost of updating one-hop replicas, we can get the average amount of saved time on communication  $TE_n$  for one stage.

Similarly, we can find out the number of extra calls on the EMV APIs. Next we calibrate the unit cost of running the APIs on a small graph to approximate the cost in real execution and obtain the average extra computation time  $TC_n$  for one stage. We determine  $n$  as the optimal number of hops of replication such that  $(TE_n - TC_n)$  is minimized. Also note that  $n$  should be bounded by the GPU memory size, since the required amount memory to stored graph partitions increases with  $n$ .

**Overlapping communication with computation** Latest GPUs can overlap kernel execution with PCI-e data transfer. We take advantage of this capability to reduce the overhead of inter-GPU communication. Since non-replicated vertices are not affected by replication, EMV APIs can be applied to them before the replicas are updated. This allows us to overlap computation and replica update. After the replicas are updated, the same set of EMV APIs can be applied to the replicas. If the computation time on non-replicated vertices is larger than the overhead of replica updates, the overhead of replica updates can be fully masked.

## 5 EVALUATION

To evaluate the programmability and efficiency of Medusa, we have developed a set of common graph processing operations for manipulating and visualizing a graph on top of Medusa. We use those operations as workloads to compare the Medusa-based implementation with other existing or our homegrown implementations as well as to evaluate the effectiveness of our optimizations.

### 5.1 Graph Primitives

We use Medusa to implement four common graph computation tasks: PageRank, breadth first search (BFS), maximal bipartite matching (MBM), and single source shortest path (SSSP). Those operations exhibit different graph operation patterns, which can also be used as building blocks for higher level applications such as graph-based analysis and ranking, community

discovery and finding the influential nodes. We also implemented a set of visualization algorithms, which can be used to build interactive visualization assisted graph mining [32].

**PageRank.** As we can see from the source code of Medusa-based PageRank implementation in Figure 2, Medusa first invokes an *ELIST* API on each vertex for sending messages to all neighbors in each iteration. Next, Medusa invokes the *Combiner* interface to calculate the sum of messages for each vertex. Finally, Medusa updates the rank of each vertex with an *VERTEX* API. Note that, we use an *ELIST* API in the first step to enable more generic PageRank implementation. Users actually could use an *EDGE* API (if appropriate) to leverage its finer-grained parallelism and more balanced execution.

**BFS.** BFS is a widely used graph search algorithm. The searching process starts from a predefined root vertex and iteratively expands to all the reachable vertices.

In BFS, vertices are visited in levels, which correspond to the iterations in Medusa. An attribute *level* is added to the vertex structure to indicate its level. The root vertex is at the first level, and is accessed in the first iteration. In the iteration  $i$ , we invoke an *EDGE* API to access each edge and check whether the *level* attribute of its head vertex equals to  $i$ . For each edge satisfying this condition, we set the *level* of the tail vertex to be  $(i + 1)$ , if the vertex has not been visited.

**MBM.** We implement the same randomized maximal matching algorithm as implementation. Two attributes are added to the vertex structure: *flag* denoting whether the vertex is in  $L$  set or in  $R$  set, and *mID* representing its matching vertex. *mID* is initialized to be *null*, meaning the vertex is unmatched.

The randomized maximal matching algorithm executes iteratively and each iteration has four phases: 1) Each unmatched  $L$  vertex sends messages to all its neighbors (which are all  $R$  vertices) and requests a match by invoking an *EDGE* API. 2) Each unmatched  $R$  vertex chooses one of its received messages randomly and sends a grant message to the sender of the chosen received message. This phase can be implemented using a *VERTEX* API. 3) If an unmatched  $L$  vertex receives any grant message, it sets its *mID* to the sender of the grant message. 4) Each  $L$  vertex, which sets its *mID* in the previous phase, notifies its matching  $R$  vertex about the newly established matching. For efficiency, we have implemented the last two phases using a single invocation of the *VERTEX* API.

**SSSP.** The SSSP problem is to find the shortest paths from a specified source vertex to all the other vertices. Same as the MTGL counterpart, we adopt the Bellman-Ford algorithm in our Medusa implementation. In particular, we use the individual API message *Combiner* to promote a more load-

balanced implementation. The vertex structure has an attribute *distance* to denote its distance to the source vertex, and the edge structure has a *distance* attribute to indicate the distance from the head vertex to the tail vertex.

During initialization, the distance of the source vertex is initialized with zero and the distance of other vertices with  $\infty$ . In each iteration, all the vertices which are updated in the previous iteration send the new distances to neighbors via outgoing edges plus the corresponding edge length. This is done by using an *EDGE* API. By applying a minimal operator on the message buffer using the *Combiner* interface, we can get the minimum message received by each vertex. Finally, a *VERTEX* API is used to update the distance of each vertex if the minimum message received is smaller than the current distance.

**Visualization.** We developed a set of visualization operations such as graph layout and drilling up/down. We implement the force direct layout algorithm [6]. The drilling up/down operation is implemented using BFS. We use OpenGL for graphics rendering and the CUDA-OpenGL inter-operability support for collaboration between computation and visualization.

## 5.2 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 experiment dataset includes two categories of sparse graphs: real-world and synthetic graphs. Table 4 shows their basic characteristics. We use the GTgraph graph generator [8] to generate the RMAT and Random graph. The RMAT graph exhibits power-law degree distribution. 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. To demonstrate the effectiveness multi-GPU extension of Medusa, we use a synthetic graph RHOP with 24 million vertices and 24 million edges. We generate RHOP such that the number of replicas of each hop is controllable to test the efficiency of replication. All the experiments are executed for ten runs and the average execution time are reported. The difference among runs for the same experiment is smaller than 2%.

## 5.3 Comparison with Manual Implementation

We study the programmability of Medusa with two examples (BFS and SSSP) in comparison with two manual implementations on GPU graph processing: Harish’s work [9] and Hong’s work [13].

Harish’s work provides open source implementation of BFS and SSSP using CUDA and we tune the

TABLE 4  
Characteristics of graphs used in the experiments

Graph	Vertices ( $10^6$ )	Edges ( $10^6$ )	Max $d$	Avg $d$	$\sigma$
RMAT	1.0	16.0	1742	16	32.9
Random	1.0	16.0	38	16	4.0
WikiTalk	2.4	5.0	100022	2.1	99.9
RoadNet-CA	2.0	5.5	12	2.8	1.0
BIP	4.0	16.0	40	4	5.1
RHOP	24.0	24.0	10	1	1.0

thread configuration and shared memory optimizations according to the C2050 Fermi architecture. We use it as the basic implementation for GPU graph processing. We implement the virtual warp-centric BFS proposed in Hong’s work [13]. The underlying difference between Medusa implementation and the warp-centric method is that Medusa apply  $L$  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.

Figure 8 shows the performance comparison between Medusa and basic implementation of SSSP. Table 5 shows the traversed edge per second (TE/s) comparison between the three implementations. For both algorithms, Medusa outperforms the basic implementation on the four graphs (except RoadNet-CA). Medusa performs much better on Wikitalk, which indicates Medusa implementation of BFS and SSSP is insensitive to graph irregularity and outliers. However, performance of Medusa on RoadNet-CA is worse than the basic implementation. This is because its large diameter (over 700), and BFS and SSSP require more iterations to terminate. Since EMV APIs are applied to all operands, Medusa incurs a fixed overhead for each iteration even if the actual amount of workload targets at a small amount of input. Large number of iterations enlarges this overhead. Although Medusa incurs more memory access and runtime overhead than the highly optimized warp-centric method, Table 5 shows that the overhead is small and Medusa performance is comparable with warp-centric. Note that the reported results of the warp-centric approach is better than those in the original paper [13], mainly because the GPU in our experiment is more powerful.

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 9 and 13 lines of source code

TABLE 5

Traversed edge per second ( $10^6$  TE/s) comparison with manual implementations [9], [13].

	Basic	Warp-centric	Medusa
Wikitalk	61.37	152.86	1091.14
RoadNet-CA	26.23	45.66	16.93
RMAT	593.21	971.1	895.76
Random	648.61	844.95	765.84

TABLE 6

Coding complexity of Medusa implementation and manual implementations.

	Baseline	Warp-centric	Medusa
GPU code lines (BFS)	56	76	9
GPU code lines (SSSP)	59	N.A.	13
GPU memory management	Yes	Yes	No
kernel configuration	Yes	Yes	No
parallel programming	Thread	Thread+Warp	No

for defining the APIs in BFS and SSSP, respectively, whereas the basic implementation [9] has 56 and 59 lines of GPU-related code. Moreover, compared with the manual implementations, Medusa requires no parallel or GPU specific programming.

We note that comparing Medusa with manual implementations is actually unfair. The major issue is that they have different optimization goals: Medusa is to offer a good programmability with a 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 the programmability. Nevertheless, we conduct the comparison with manual implementations to demonstrate that the optimized Medusa delivers a reasonable performance.

#### 5.4 Experiments on Efficiency

**Overall comparisons.** We implement the graph processing operations with MTGL [23], as the baseline for graph processing on multi-core CPUs. Similar to Medusa, MTGL offers a set of data structures and APIs for building graph algorithms. The MTGL API is

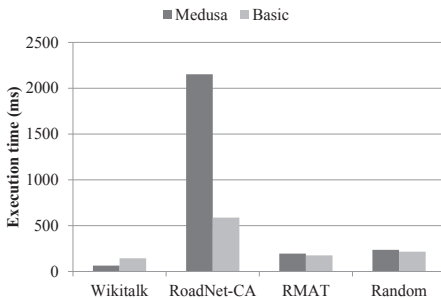


Fig. 8. Performance comparison between Medusa and existing GPU implementation of SSSP [9].

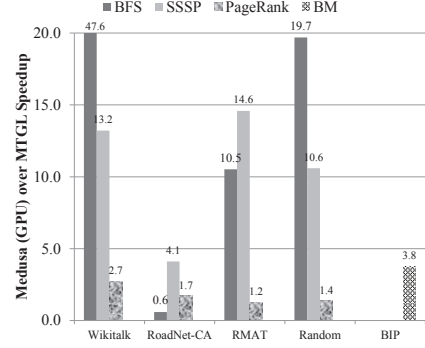


Fig. 9. Performance speedup of Medusa running on the GPU over MTGL [23].

modeled after the Boost Graph Library and optimized to leverage shared memory multithreaded machines.

The BFS and PageRank implementation is offered by MTGL and we implement the Bellman-Ford single source shortest path algorithm and a randomized maximal matching algorithm [1] 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. Targeting at general graph processing operations, MTGL demonstrates good scalability for some operations such as PageRank. However, the runtime overhead of MTGL is significant for BFS. The MTGL-based BFS on twelve CPU cores is only 1.9 times faster than that on a single CPU core.

Figure 9 shows the speedup of Medusa over MTGL. The PageRank is executed with 100 iterations. The *speedup* is defined to the ratio between the elapsed time of the CPU-based execution and that of Medusa-based execution. Medusa is significantly faster than MTGL on all the input graphs (except BFS on RoadNet-CA). The performance speedup is 0.6–47.6, with an average of 10.1. Medusa-based BFS is slower than MTGL-based BFS on RoadNet-CA because it has a large diameter (due to the same reason stated in Section 5.3). Despite this, our GPU-specific optimizations on unleashing the GPU power significantly contribute to the performance improvement. For example, CAA has significantly improved the bandwidth utilization, as demonstrated in the later experiments.

**Impact of individual optimizations.** In order to study the impact of a certain optimization, we conduct the experiment by disabling/enabling the optimization while other optimizations are enabled. In particularly, we first evaluate the impact of different graph layouts, followed by our graph-aware buffer scheme.

Since we observed similar results on different operations, we present the results for PageRank.

Figure 10 (a) compares the elapsed time for PageRank with Medusa using different graph storage lay-

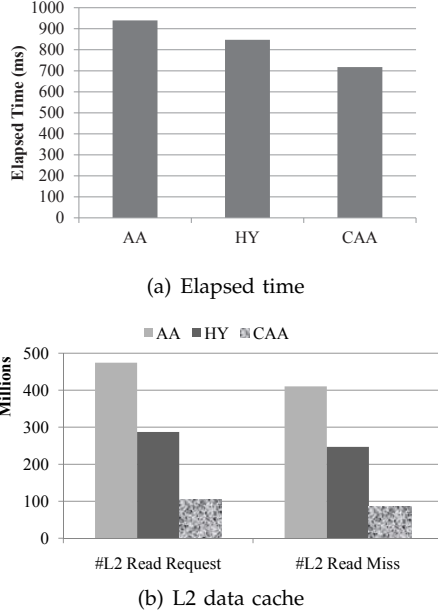


Fig. 10. Performance impact of different graph layouts on PageRank using RMAT as input.

outs on RMAT. The edge-oriented storage layout results in “out-of-memory” problems, and thus its results are not included. The hybrid layout chooses the threshold value to be the average out-degree. Among different graph layouts, CAA is the best, the hybrid layout comes the second, and AA is the last. PageRank with CAA is 34% faster than that with AA. This is correspondent with the number of memory requests to the device memory. We have performed profiling on the L2 cache of one multiprocessor with CUDA visual profiler. As shown in Figure 10 (b), CAA has the smallest number of read misses on the L2 data cache, and the smallest number of read requests to the L2 data cache. This indicates that CAA is optimized with the coalesced memory accesses. The bandwidth utilization is improved, reaching over 50%.

We study the impact of different buffer schemes in Medusa with other optimizations enabled. It was shown that list-based buffer scheme is superior to the array-based scheme [12], and thus we adopt only the list-based buffer scheme in Medusa. The awareness of the graph structure significantly improves the efficiency of Medusa. Compared with the list-based buffer scheme, our buffer scheme improves the memory performance as well as the overall performance, with an average improvement of 77% on PageRank.

**Efficiency of visualization operations.** The visualization experiment is conducted on a graphics workstation with one NVIDIA Quadro 5000 GPU and one Intel Xeon W3565 processor with 4 GB memory. We use the DBLP graph to evaluate visualization assisted graph mining. The DBLP graph is extracted from DBLP (<http://dblp.uni-trier.de/xml/>): each author as a vertex, and an edge meaning co-authorship between

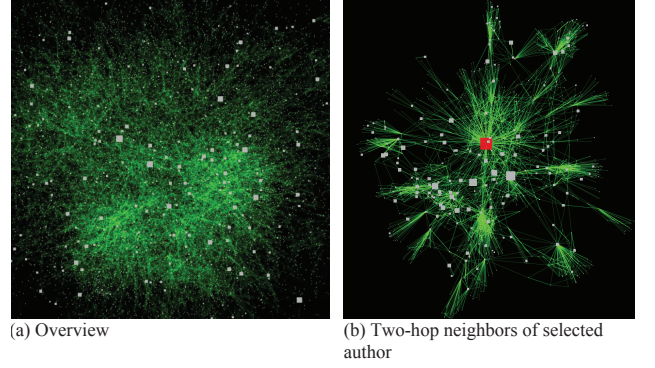


Fig. 11. Visualization results on DBLP co-authorship graph.

the two corresponding authors.

With Medusa, users can compose a variety of visualized graph discovery tasks. Additionally, combining computation and visualization into a single GPU eliminates the PCI-e data transfer for each invocation of graphics rendering.

Figure 11 (a) shows the visualization result of the DBLP graph. On the Quadro 5000 GPU, Medusa takes only 120 seconds to get the overview, while the multi-threaded CPU implementation on the Intel Quad-core CPU takes over 1000 seconds. Figure 11 (b) shows the two-hop neighbors of a selected author Jiawei Han obtained by a drill-down operation. Medusa greatly improves the responsiveness of graph visualization tasks.

## 5.5 Results on Multi-GPU Extension

We have evaluated the common graph processing algorithms, and present the results on BFS and PageRank. BFS exhibits a wave-front computing pattern and involves little computation, while PageRank has relatively heavy computation. The number of first-hop replicas is around 5% of the total number of vertices.

**One-hop Replication with Execution/Memory Overlap.** Table 7 shows the performance of BFS and PageRank with the number of GPUs varied. The execution/memory overlapping slightly improves the overall performance for both BFS and PageRank. Compared with single-GPU implementations, the speedup of PageRank is 2.0 and 3.8 when the number of GPUs is two and four, respectively. BFS slightly performs better as the number of GPUs increases. This is because the computation of BFS is lightweight and the inter-GPU communication time dominates the total execution time.

**Multi-hop Replication.** The number of replicas on different hops in RHOP follows the rule beneath:

$$v_n = \alpha v_{n-1} (n = 1, \dots, 4), v_1 = 0.05 v_0$$

Figure 12 shows the measured performance of BFS and PageRank with the number of hops varied.

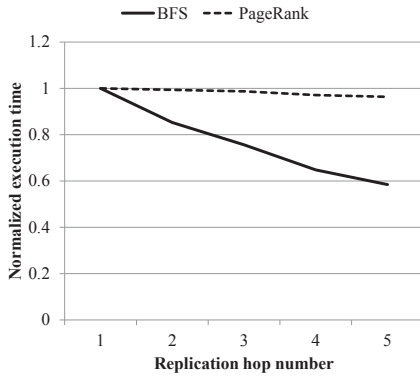


Fig. 12. Normalized execution time with increasing hops of replication.

TABLE 7

Performance comparison of varying the number of GPUs with/without overlapping.

	#GPUs	PageRank (ms)	BFS (ms)
Basic	1	5962.6	20.5
	2	3020.6	18.1
	4	1627.8	17.4
Overlap	2	2961.4	16.6
	4	1585.2	15.4

Multi-hop replication effectively increases the performance of BFS. However, there is little benefit using this approach for PageRank. Since BFS carries a very small amount of computation and the communication overhead is the performance bottleneck, reducing the amount of data transfer can greatly reduce the overall execution time of BFS. In contrast, PageRank is computation bounded and communication cost takes a much smaller share of the total execution time.

We also evaluate the effectiveness of optimal hop number estimation. Our estimation of  $TC_n$  and  $TE_n$  when the graph generation parameter ( $\alpha$ ) equals 0.8 is plotted in Figure 13, together with the measured decreased time for one stage. The prediction for BFS is quite close to the actual saved time, with prediction error of smaller than 10% for different  $\alpha$  values.

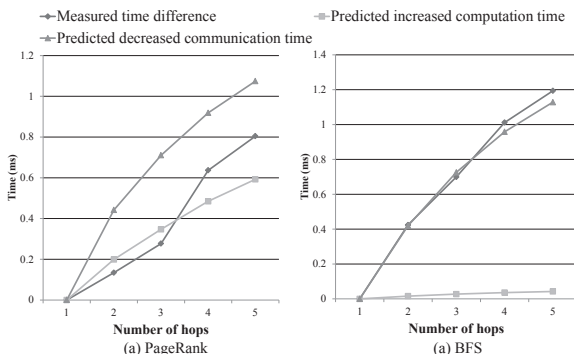


Fig. 13. Comparison between the predicted  $TC_n$ ,  $TE_n$  and actual measured average decreased time for one stage.

The prediction for PageRank is less accurate and the prediction error is 46% on average. This is because PageRank includes more APIs and the execution time of the *Combiner* interface does not conform to our linear cost assumption. Nevertheless, both the prediction and actual measurement show positive effects on increasing the number of replication hops for BFS and PageRank.

## 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 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 is 9.1 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 plan to support dynamic graph operations (with structure mutation during processing) and huge graphs larger than the GPU memory. Medusa is open-source, and the source code can be accessed at <http://code.google.com/p/medusa-gpu/>.

## 7 ACKNOWLEDGEMENT

The authors would like to thank Pawan Harish for providing their source code on CUDA-based BFS and shortest path.

## REFERENCES

- [1] 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, November 1993.
- [2] 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.
- [3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
- [4] 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.
- [5] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, 2005.
- [6] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21, 1991.
- [7] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *POOSC*, 2005.
- [8] GTGraph Generator. <http://www.cc.gatech.edu/kamesh/GTgraph/>.
- [9] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, 2007.



- [10] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *PACT*, 2008.
- [11] G. He, H. Feng, C. Li, and H. Chen. Parallel SimRank computation on large graphs with iterative aggregation. In *SIGKDD*, 2010.
- [12] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: writing parallel program portable between CPU and GPU. In *PACT*. ACM, 2010.
- [13] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, 2011.
- [14] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. HPC simulations of information propagation over social networks. *Procedia Computer Science*, 9:292 – 301, 2012.
- [15] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Graphics hardware*, 2008.
- [16] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple gpus. In *PPoPP*, 2011.
- [17] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.
- [18] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *MLG*, 2010.
- [19] L. Luo, M. Wong, and W.-m. Hwu. An effective GPU implementation of breadth-first search. In *DAC*, 2010.
- [20] 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.
- [21] D. Merrill, M. Garland, and A. Grimshaw. High performance and scalable gpu graph traversal. Technical Report UVA CS-2011-05, University of Virginia, 2011.
- [22] Metis. <http://glaros.dtc.umn.edu/gkhome/views/metis/>.
- [23] MultiThreaded Graph Library (MTGL). <https://software.sandia.gov/trac/mtgl>.
- [24] NVIDIA. *NVIDIA CUDA C Programming Guide 4.0*, 2011.
- [25] 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.
- [26] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Stanford InfoLab. Technical report, 1999.
- [27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Proutzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.
- [28] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In *IPDPS*, 2009.
- [29] S. Sengupta, M. Harris, and M. Garland. M.: Efficient parallel scan algorithms for GPUs. NVIDIA. Technical report, 2008.
- [30] M. C. Shebanow. Pervasive massively multithreaded gpu processors. In *CF*, 2009.
- [31] K. Spafford, J. S. Meredith, and J. S. Vetter. Quantifying NUMA and contention effects in multi-gpu systems. In *GPGPU*, 2011.
- [32] F. van Ham and J. J. van Wijk. Interactive visualization of small world graphs. In *Proceedings of the IEEE Symposium on Information Visualization, INFOVIS '04*, 2004.
- [33] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *CVPR Workshops*, 2008.