Parallel Graph Processing on Graphics Processors Made Easy

Jianlong Zhong
Nanyang Technological University
jzhong2@ntu.edu.sg

Bingsheng He
Nanyang Technological University
bshe@ntu.edu.sg

ABSTRACT

This paper demonstrates Medusa, a programming framework for parallel graph processing on graphics processors (GPUs). Medusa enables developers to leverage the massive parallelism and other hardware features of GPUs by writing sequential C/C++ code for a small set of APIs. This simplifies the implementation of parallel graph processing on the GPU. The runtime system of Medusa automatically executes the user-defined APIs in parallel on the GPU, with a series of graph-centric optimizations based on the architecture features of GPUs. We will demonstrate the steps of developing GPU-based graph processing algorithms with Medusa, and the superior performance of Medusa with both real-world and synthetic datasets.

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. 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, shortest path and even their customized variants (for example, developers 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. 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 (e.g., [7, 5, 8]). While those GPU-based solutions have demonstrated significant performance improvement over the 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. Since graph applications usually involve irregular accesses to the graph data, careful designs on data layouts 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 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 [4] and its variant on the GPU [6], Medusa provides a set of APIs for developers to implement their applications by writing sequential (C/C++) code. 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. Moreover, Medusa embraces a series of graph-centric optimizations based on the architecture features of GPUs.

We implement Medusa with NVIDIA CUDA 5.0. In this

Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... $ 10.00.

demo, we will demonstrate the ease-of-programming feature and the superior performance of Medusa with a series of common graph processing operations.

2. RELATED WORK

Parallel graph processing. Parallel algorithms have been a classical way to improve the performance of graph processing. On multi-core CPUs, parallel libraries like MTGL [3] have been developed for parallel graph algorithms.

Previous studies [9, 10] 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 the distributed graph processing frameworks like Pregel [11]. For example, Pregel applies a user-defined function *Compute()* on each vertex in parallel in each iteration of the GBSP execution. 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.

GPGPU. We follow NVIDIA's terminology. 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 access*. 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.

3. SYSTEM OVERVIEW

In this section, we give an overview on how users implement their graph processing algorithms based on Medusa, and then outline the key modules of Medusa. The reader may refer to the project website (`http://code.google.com/p/medusa-gpu/`) and a paper [16] for details.

3.1 Programming with Medusa

Medusa adopts an EMV model which enhances the current single vertex-based API design to support efficient and fine-grained graph processing on the GPU. In particular, Medusa hides the GPU programming details from users by offering two kinds of APIs, user-defined APIs and system-provided APIs. Through those APIs, Medusa enables programmability and efficiency for parallel graph processing on the GPU.

```
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;
}
```

Figure 1: User-defined functions in PageRank implemented with Medusa.

First, Medusa provides six device code APIs for developers to write GPU graph processing algorithms. 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 these APIs is mainly for efficiency. 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, a *Combiner* API is provided to aggregate results of *EDGE* and *MESSAGE* using an associative operator.

Second, Medusa hides the GPU-specific programming details with a small set of system provided APIs. Particularly, Medusa provides $EMV < type >:: Run()$ to invoke the device code API, which automatically sets up the thread block configurations and calls the corresponding user-defined function. Medusa allows developers to define an *iteration* which executes a sequence of $EMV < type >:: Run()$ calls 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.

To demonstrate the usage of Medusa, we show an example of the PageRank implementation with Medusa, as shown in Figure 1. Data structures (e.g., *vertex*) are defined. The function *PageRank()* consists 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

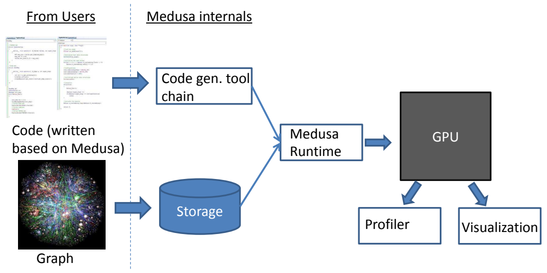

Figure 2: The key modules in Medusa.

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.

3.2 System Internals

Figure 2 shows the system architecture of Medusa. It consists of the following key modules.

Graph storage. In the storage module, Medusa allows developers to initialize the graph structure through adding vertices and edges with two system provided APIs namely *AddEdge* and *AddVertex*. The storage component of Medusa stores the graph with optimized graph layout and transfer the graph to GPU automatically. The optimized graph layout exploits the coalesced memory access feature of the GPU, whereas the classic adjacency list cannot.

Medusa code generation tool chain. This module generates the CUDA code for graph computation based on the user-defined EMV APIs. Particularly, users define the data structures (e.g., vertex, edge and messages) and implement the EMV APIs and control program according to their graph computation logic. The example code of PageRank has been given in Figure 1. The code generation tool chain performs the following two steps on the above-mentioned user code. First, a source-to-source transformation tool generates code for memory allocations and transforms definitions from array of structure (AOS) to SOA. This transformation allows Medusa to leverage the coalesced memory access feature of the GPU. Programming with SOA diverts developers from the natural way of thinking about data [13]. To simplify the programming interface, Medusa allows customized data structures such as vertex and edge to be defined using C/C++ *struct*.

Second, Medusa inserts segmented-scan code [12] for the *EDGE* and *MESSAGE* APIs which are declared to be executed by *Combiner*. Many collective APIs (including *ELIST* and *MLIST*) 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. Being implemented as a segmented scan operation, the *Combiner* API eliminates the load imbalance problem that each instance of the collective API processes different numbers of edges or messages.

Finally, the system generates the CUDA code for the entire graph computation by adding the code of Medusa runtime. Given the code, we allow users to investigate the efficiency of the Medusa-based program, and users can further apply specific optimizations if they want to improve the performance. Then, the program is compiled and linked with the Medusa libraries.

Medusa runtime. The runtime module is responsible for kernel execution and scheduling and memory management on the GPU. It supports multiple concurrent programs from users. The runtime system maintains a number of data structures to monitor the current state of the GPU. The data structures include 1) a command queue which consists of different kinds of commands. A command can be executing a kernel, memory allocation/deallocation, and memory transfer between the main memory and the GPU memory. 2) a queue of active memory objects (e.g., arrays and global variables) in the GPU memory. 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.

First, the runtime prepares the kernel execution and submits the kernel for execution. Particularly, the runtime system first enqueues memory deallocation commands if there is no sufficient GPU memory available, and then enqueues memory allocation and kernel execution commands. The memory deallocation is according to the memory queue maintained by the runtime. The victims for memory deallocation are chosen according to whether they will be referenced by the kernels in the command queue and their access patterns. Thus, the memory queue is implemented as an LRU queue with looking ahead.

Second, the scheduler schedules the commands in the command queue for execution. On the machine with N GPUs, Medusa automatically runs the EMV APIs on individual graph partitions stored in the N GPUs. There are two considerations on the hardware features of the latest GPU. First, latest GPUs can overlap kernel execution with PCI-e data transfer. We take advantage of this capability to reduce the overhead of PCI-e data transfer. Second, latest GPUs support concurrent kernel executions. We can schedule more kernels smartly for high utilization on the GPU resources. The details of the scheduling can be found in our technical report [15].

Profiler. GPU vendors have offered hardware counters on the GPU to understand the detailed performance of a GPGPU program. We develop a GUI to hide the details from vendor-specific profiler tools. As a start, we integrate NVIDIA command line profiler into our GUI and users can investigate the performance metrics on memory and execution. The main performance metrics include accesses to the global memory and the shared memory and branch divergence. These performance metrics can be shown on per kernel or per program basis. The default setting is per program. With the profiler, we are able to determine the hot region of a graph computation, and compare the profiling results for different implementations if available.

Visualization. This module visualizes a graph in a manner such that the graph structure is well laid out. Our previous study has implemented GViewer [14] to assist graph visualization and mining with GPUs. We re-implement GViewer based on Medusa, using the CUDA-OpenGL interoperability support for collaboration between computation and visualization.

4. DEMO PLAN

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. We plan to conduct the demonstration with remote access to the workstation. Our experimental

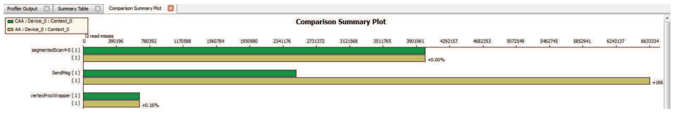

Figure 3: Profiling results (L2 read misses) of a single iteration of PageRank.

datasets include two categories of sparse graphs: real-world and synthetic graphs. The real-world graphs include DBLP and other publicly available ones [2].

Ease of programming. We demonstrate the ease-of-programming feature in two aspects.

First, we invite interested audience to program with Medusa. In the demonstration, we will give a short tutorial to the audience with the examples, and the audience can follow the examples to understand Medusa and to use Medusa to implement their graph applications. The examples currently include a set of common graph processing operations including PageRank, breadth first search (BFS), maximal bipartite matching (MBM), and single source shortest path (SSSP). The fine-grained and flexible API design of Medusa allows the users to implement graph algorithms easily.

Second, we demonstrate the code written by users with Medusa, in comparison with some manual implementations. We will see that, Medusa simplifies GPU programming for graph processing, by significantly reducing the number of GPU-related source code lines written by developers. For example, developers only need to write 7 and 11 lines of source code for defining the APIs in BFS and SSSP, respectively, whereas the manual implementation in the previous work [5] has 56 and 59 lines of GPU-related code. This is because Medusa hides the GPU programming complexity by offering a small set of user-defined APIs. Moreover, compared with the manual implementations, Medusa requires no parallel or GPU specific programming.

Superior performance of Medusa. We shall demonstrate the superior performance of Medusa in two aspects.

First, we shall demonstrate the performance speedup of Medusa over its optimized CPU-based counterparts on multi-core CPUs. We implement the graph processing operations with MTGL [3], as the baseline for graph processing on multi-core CPUs. In our experiments, 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 on the test platform.

Second, we shall demonstrate that the proposed optimizations significantly improve the performance of GPU-based graph processing. Particularly, we show the profiling results by disabling/enabling certain optimizations (e.g., different graph storage layouts and message passing mechanisms). Figure 3 shows the screenshot of comparing the number of L2 read misses for PageRank on RMAT graph (1M vertices and 16M edges) generated from an existing tool [1]. We compare two storage layouts – the adjacency array (AA) and the optimized layout in Medusa (denoted as CAA). We show the three major kernels in a single iteration of PageRank execution. The optimized layout has a much smaller number of read misses on the L2 data cache in the SendMsg kernel, which executes an ELIST API.

Graph Visualization and Mining. We will use DBLP as dataset to demonstrate the benefits of GPU-accelerated graph visualization and mining. Medusa simplifies the implementation of the graph processing algorithm in graph visualization and mining, and also enables better interactive experience to users than the CPU-based counterpart.

5. CONCLUSIONS

Medusa shows that parallel graph computation can efficiently and elegantly be supported on the GPU with a small set of user-defined APIs. The fine-grained API design and graph-centric optimizations significantly improve the performance of graph computation on the GPU. In this paper, we present a demonstration of Medusa to show the steps of building a graph application with Medusa and to demonstrate the performance impact of the optimizations in Medusa and its comparison with CPU-based counterparts. An interactive visualization tool is reinvented based on Medusa to demonstrate the efficiency on assisting graph visualization and mining. The source code of Medusa is available at `http://code.google.com/p/medusa-gpu/`.

6. ACKNOWLEDGEMENT

The authors would like to thank anonymous reviewers for their valuable comments. This work is supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

7. REFERENCES

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

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

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

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

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

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

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

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

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

[10] Y. Low and et al. GraphLab: A new parallel framework for machine learning. In *UAI*, July 2010.

[11] G. Malewicz and et al. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

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

[13] M. C. Shebanow. Pervasive massively multithreaded GPU processors. In *CF*, 2009.

[14] J. Zhong and B. He. GViewer: GPU-accelerated graph visualization and mining. In *SocInfo*, pages 304–307, 2011.

[15] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling, 2013.

[16] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. *IEEE TPDS*, 99(PrePrints):1, 2013.