

Medusa: Simplified Graph Processing on GPUs

(Supplementary File)

Jianlong Zhong, Bingsheng He

APPENDIX A MORE DETAILS ON SYSTEM DESIGN

In this section, we present the details on GPU-transparent programming interface, graph layouts and multi-GPU execution of Medusa.

A.1 GPU-Transparent Programming Interface

To leverage the coalesced memory access feature, Medusa uses structure of array (SOA) data arrangement instead of array of structure (AOS) extensively. Programming with SOA diverts developers from the natural way of thinking about data [8]. To simplify the programming interface, Medusa allows customized data structures such as vertex and edge to be defined using C/C++ *struct*. Developers define data structures in AOS, and Medusa automatically transforms AOS into SOA. We have developed a home-grown source-to-source transformation tool to generate SOA code for memory allocations and transforming definitions from AOS to SOA. Those transformations are straightforward but tedious. Thus, it is desirable to make this transformation automatic for ease of programming.

We use the same PageRank example to illustrate the code generation process. The top of Figure 1 gives the 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 1 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 front end automatically generates GPU related codes including device memory management and kernel invocation etc. Providing simplified sequential interfaces and

```

User defined data structure:
struct vertex{
    float pg_value;
    int vertex_id;
}

Generated data structure definitions:

/* Structure of array definition */
struct VertexArray{
    /* user-defined attribute copied from
    user definition*/
    float *pg_value;
    int vertex_id;
    /* attributes for accessing the graph
    data structure inserted by Medusa */
    int *msg_index;
    int *edge_index;
    int *edge_count;
    int size;
};

/* Structure of array definition */
struct D_vertex{
    int index;
    __device__ D_Vertex::D_Vertex(int
    vertex_index){
        index = vertex_index;
    }
    __device__ float get_pg_value(){
        return d_vertexArray.pg_value[index];
    }
    __device__ void set_pg_value(float pg_value){
        d_vertexArray.pg_value[index] = pg_value;
    }
    __device__ int get_vertex_id(){
        return d_vertexArray.vertex_id[index];
    }
}

```

Fig. 1. Device data structure definitions and CUDA kernels generated from front end for PageRank.

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 a steep 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 [6]. It took him about only two weeks, given that he has no GPU programming experience before.

A.2 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 associated data. As shown in Figure 2(a), AA consists of two arrays, *Edge* for storing all the edge-lists, and *Start* for the starting position of

• 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

each edge-list in $Edge$. The problem with 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 (ELL) 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 to 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 local ID k for the head vertex with ID i . With the edge-oriented layout, memory accesses to the edges with the same local ID are coalesced accesses. However, if the out-degrees of vertices have a large variance (e.g., in 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 ELL, and the remainder of the graph using the AA. t is a tuning parameter according to the out-degree distribution 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 ELL; the vertices with large out-degrees are accessed according to AA.

The problem of the hybrid layout is that it requires tuning the threshold value on AA and ELL and it still has the uncoalesced memory access to the part stored in AA. We further develop a column-major adjacency 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 2 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 2(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

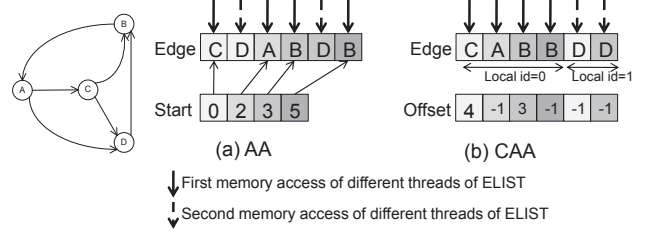


Fig. 2. Graph layouts: AA and CAA.

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 numbers of edges. By launching a sufficiently large number of thread blocks, SMs tend to have more balanced loads.

We note that AA, ELL and HY have their counterparts in sparse matrix representations (CSR, ELL-PACK and hybrid formats respectively in previous studies (e.g., [2]). CAA is similar to JDS format of a sparse matrix developed by Saad [7]. The major difference is that CAA maintains the offset information to facilitate the accesses to all edges associated to the same vertex.

A.3 Multi-GPU Execution

We present more details on multi-GPU execution, including the cost model and overlapping communication with computation.

Cost model. The total time cost of n -hop replication scheme is the sum of computation time on replicas 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 latency and 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 maximized. Also note that n should be bounded by the GPU memory size, since the required amount memory to store 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.

APPENDIX B 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 paths (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 [9].

PageRank. As we can see from the source code of Medusa-based PageRank implementation in Section 3 of the main file, 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 order to evaluate 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. With *SetActive* API, BFS-Q is work-efficient. In comparison, while BFS-N is work-inefficient, it is able to exploit the coalesced memory feature of the GPU.

In BFS-N, 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.

In BFS-Q, we invoke the *EDGE* API on the active edges only, unlike BFS on all the edges in the graph. Initially, we set the edges of the root vertex as active.

In the *EDGE* API, we set edges of newly reached vertices as active. Medusa finishes execution when no edge is marked as active.

MBM. To implement the randomized maximal matching algorithm [1], 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. Similar to BFS, we have also implemented two versions of SSSP: SSSP-N and SSSP-Q without and with the usage of *SetActive* APIs, respectively.

In the initialization of SSSP-N, the distance of the source vertex is initialized with zero and the distance of other vertices with ∞ . 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. This SSSP-N implementation is work-inefficient in two aspects. First, the *EDGE* API works on every edge, even if the head vertex of an edge is not updated in the previous iteration. Second, the *VERTEX* API works on every vertex. However, the vertices which receive no message do not require that processing.

SSSP-Q addresses the work-inefficient issues of SSSP-N with the *SetActive* API. We adopt the

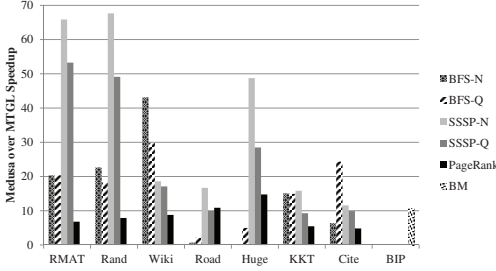


Fig. 3. Performance speedup of Medusa running on the GPU over MTGL [3] running on one core.

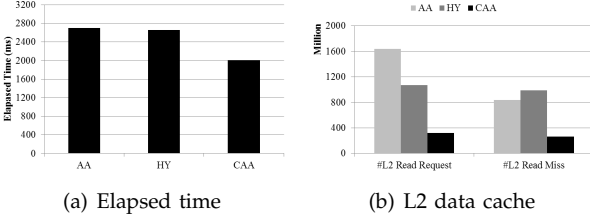


Fig. 4. Performance impact of different graph layouts on PageRank using RMAT as input.

SetActive API so that the *EDGE* API is applied to the edges whose head vertex is updated in the previous iteration, and the *VERTEX* API is applied to the vertices that have received messages.

Visualization. We developed a set of visualization operations such as graph layout and drilling up/down. We implement the force direct layout algorithm [4]. The drilling up/down operation is implemented using BFS. We use OpenGL for graphics rendering and the CUDA-OpenGL interoperability support for collaboration between computation and visualization.

APPENDIX C MORE EXPERIMENTAL RESULTS

C.1 More results on performance speedup

We show the performance speedup of comparing Medusa with MTGL with a single thread (as shown in Figure 3). 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. The performance speedup over MTGL on a CPU core is 2.1–67.7, with an average of 15.2. Medusa-based BFS-N is slower than MTGL-based BFS on the Road graph because the large searching depth of the graph.

C.2 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 particular, 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 4(a) compares the elapsed time for PageRank with Medusa using different graph storage layouts 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. The ELIST API of PageRank with CAA is 29.6% 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 with CUDA command line profiler. As shown in Figure 4(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 [5], 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 to 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.

C.3 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 the two corresponding authors.

With Medusa, developers 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 5(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 5(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.

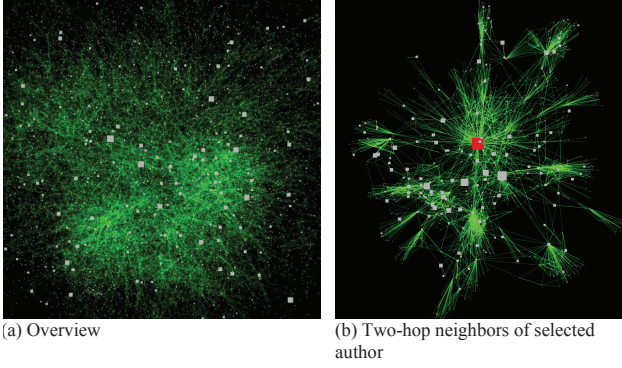


Fig. 5. Visualization results on DBLP co-authorship graph.

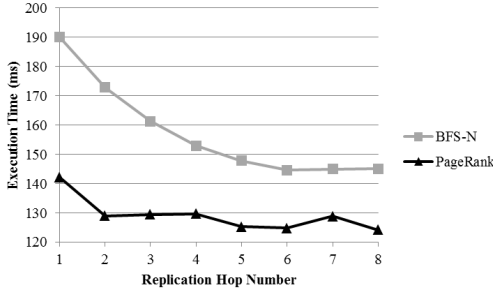


Fig. 6. Execution time with increasing hops of replication.

C.4 Results on Multi-GPU Extension

We have evaluated the common graph processing algorithms, and present the results on BFS-N and PageRank. BFS-N exhibits a wave-front computing pattern and involves little computation, while PageRank has relatively heavy computation.

One-hop Replication with Execution/Memory Overlapping. Table 1 shows the performance of BFS-N and PageRank on the Road graph with the number of GPUs varied. Execution/memory overlapping improves the overall performance for both BFS-N and PageRank. Execution/memory overlapping brings more benefits to BFS-N. This is because the computation of BFS-N is lightweight and the inter-GPU communication time dominates the total execution time. On the other hand, communication takes a smaller portion of PageRank execution time. When overlapping is enabled, the speedup of BFS-N and PageRank compared with single-GPU implementation is 1.3–1.4 and 1.6–2.3, respectively.

Multi-hop Replication. Figure 6 shows the measured performance of BFS-N and PageRank with the number of hops in replication varied on four GPUs. Multi-hop replication effectively increases the performance of BFS-N. However, there is little benefit using this approach for PageRank. Since BFS-N carries a very small amount of computation and the communication overhead is the performance bot-

TABLE 1
Performance comparison of varying the number of GPUs with/without overlapping.

	#GPUs	PageRank (ms)	BFS-N (ms)
Basic	1	319.2	261.6
	2	203.2	347.3
	4	146.1	444.7
Overlap	2	200.2	197.1
	4	140.4	189.2

tleneck, reducing the amount of data transfer can greatly reduce the overall execution time of BFS-N. In contrast, PageRank is computation bounded and communication cost takes a much smaller share of the total execution time. With multi-hop replication, the speedup of BFS-N and PageRank on four GPUs is increased up to 1.8 and 2.6, respectively.

We also evaluate the effectiveness of the suitable hop number estimation. We find that our cost estimation is sufficiently accurate to guide our decision on the suitable number of hops in replications. Take BFS-N and PageRank as an example. The prediction on the decreased time with the increasing number of hops for BFS-N is quite close to the actual decreased time. In comparison, the prediction for PageRank is less accurate. 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-N and PageRank.

Replication for Power Law Graphs. Power-law graphs such as social networks and the web graph usually have very small diameters. Replication may replicate a large portion of the vertices for power-law graphs, and our cost model decides not to perform replication. We study the number of replicated vertices and execution time of Medusa on four GPUs with the number of hops in replication varied on RMAT. We find that one-hop replication leads to a significant portion of replicated vertices (over 96% of the vertices in RMAT) and the execution time is almost the same across different hops of replications (even with slight degradation sometimes).

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.*, 11:319–352, November 1993.
- [2] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*, 2009.
- [3] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS*, March 2007.
- [4] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21, 1991.
- [5] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing parallel program portable between CPU and GPU. In *PACT*, 2010.

- [6] 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.
- [7] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput.*, 10(6):1200–1232, Nov. 1989.
- [8] M. C. Shebanow. Pervasive massively multithreaded GPU processors. In *CF*, 2009.
- [9] F. van Ham and J. J. van Wijk. Interactive visualization of small world graphs. In *INFOVIS*, 2004.