Medusa: Simplified Graph Processing on GPUs

JIANLONG ZHONG, BINGSHENG HE

PRESENTED BY: AMINI, HAGHIGHI, FOTOVAT

Content

- Abstract
- Introduction
- Previous works
- Medusa
- Medusa on Multi GPU
- Evaluation

Novelty

- A software framework (Medusa)
  - Simplify programming graph processing on GPU
  - Provides a set of APIs
  - Automatically execute user defined APIs in parallel
    - Hide complexity of programming from developers
- Novel graph programming model
  - Edge-message-vertex
    - Fine grained processing on vertices and edges
    - Fitted for parallel graph processing on GPU

Evaluation

- Machine with
  - four C2050 GPU
  - Two E5645 CPU
- Cpu-based implementation based on MultiThread Graph Library

Results

- Significant reduction in source code lines
- Improved performance of graph processing due
  - Optimization techniques on
    - graph layout
    - Message buffering
- 1.8 and 2.6 speedup for executing BFS and page rank on four GPU

Goals

- Programmability
- Highly programmable graph processing framework for different apps.

Introduction

- Graphs are common data structure
  - Social network
  - Web link analysis
- Due to high performance of the entire system, graph processing efficiency is a must
- Productivity can greatly improve
  - Programming framework
    - High programmability for various graph processing apps.
    - Providing high efficiency

Introduction (contd.)

- GPU as an accelerator for various graph processing apps.
- GPU-based solutions vs CPU-based
  - Improved performance
  - Limited to specific graph operations
  - Usually need to implement and optimize programs from scratch for different graph processing tasks.

Introduction (contd.)

- GPU programming challenge
  - Correctness
  - Efficiency of program

Introduction (contd.)

- Many-core processor
- Massive thread parallelism
- Need to write parallel program which scale to hundreds of cores
- Lightweight threads
- Fine grained tasks in algorithm
- Different memory hierarchy
- Explicitly memory management

Pre. work

- MTGL
  - Set of APIs for parallel graph processing on Multi Core CPU
- SNAP framework
  - Set of algorithms and building blocks
- Bulk Synchronous Parallel
  - Local computation perform on vertices
  - Vertices able to exchange data

Bulk Synchronous Parallel

- Consist of
  - Components with processing/local memory transaction
  - Network that routes messages between components
  - Hardware allow for Synchronization
- Each processor
  - fast local memory
  - Communication network

Bulk Synchronous Parallel (contd.)

- A computation proceeds in a series of global steps
- Each step has
  - Concurrent computation
  - Communications
  - Barrier synchronization
- Google adopt this for graph processing
  - Pregel
  - Mapreduce

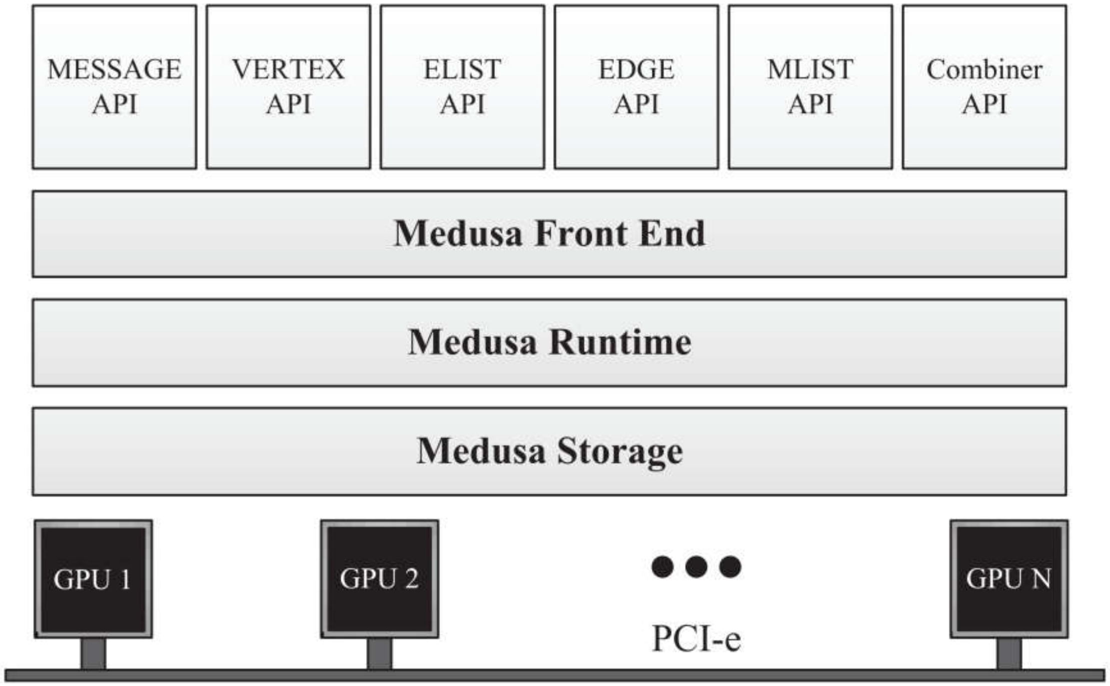
GPU

- Each GPU array of Streaming Multiprocessor (SM)
- Each SM a group of scalar cores
- CUDA enable developers to run hundred of GPU cores with thousand of threads
- CUDA memory hierarchy
    - Shared memory -> low latency
    - Device memory

Assumptions

- Graph is directed
- Each vertex has unique ID ranging [0..V-1]
- Each associated edge with same vertex has ID ranging [0..d-1]
- d is out degree of vertex
- d_{max} is maximum out degree in graph

Overview of Medusa



Medusa Workflow

- Define basic structures → edge, message and vertex
- Application logic → EMV APIs
- Composes the main program
  - initializing the graph structure
  - Configuring the framework parameters
  - invoking the customized EMV APIs with the system provided APIs

Iteration Control

- maxIteration
  - terminates when the number of iterations reaches the predefined limit

- halt flag
  - Global variable
  - modified by the EMV APIs
  - initializing *halt* as false

Sample Code(Part-1)

```
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;
}
```

Sample Code(Part-2)

```
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;
}
```

User-defined APIs in EMV

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

System APIs and Parameters

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

Techniques used in Medusa

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

Massive Parallelism

- Vertex-based API problems
  - different vertices →different numbers of edges→ different workloads on each API instance.
  - different number of received messages →divergence.
- EMV
  - individual vertices, edges or messages
  - Each GPU thread → one instance of the user-defined API
  - Collective & individual

Work Efficiency

- Medusa Default
  - applies the user-defined API on the vertices/edges on the entire graph
- SetActive API
  - Dynamic queue
  - apply the EMV APIs to the active vertices and edges only

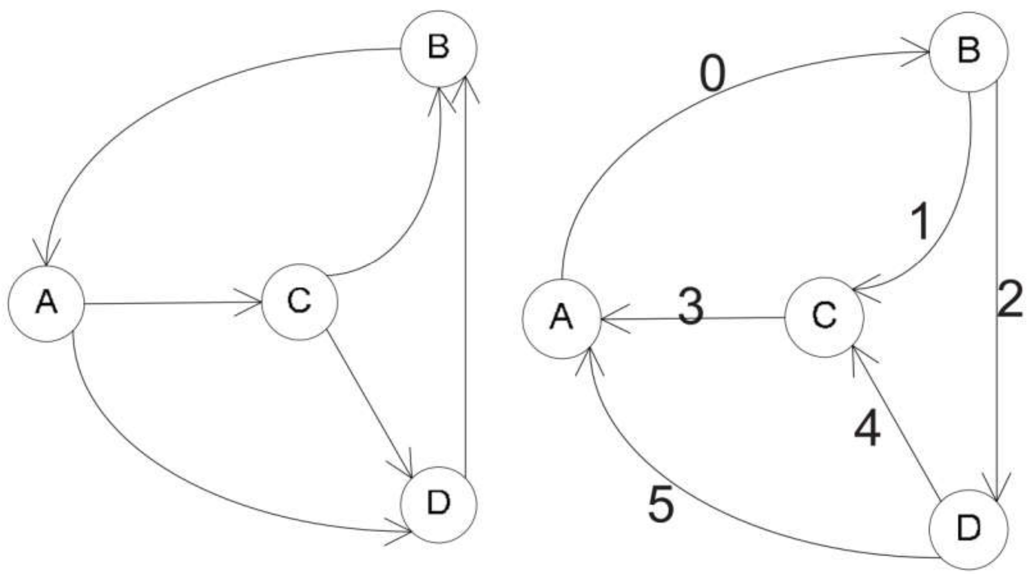
Message Passing Efficiency

- Fixed-Size Array
  - Information of the buffer size
  - Output position
  - Need Grouping Operation
- Dynamic List
  - No pre-computation
  - No grouping operation
  - Need atomic operation

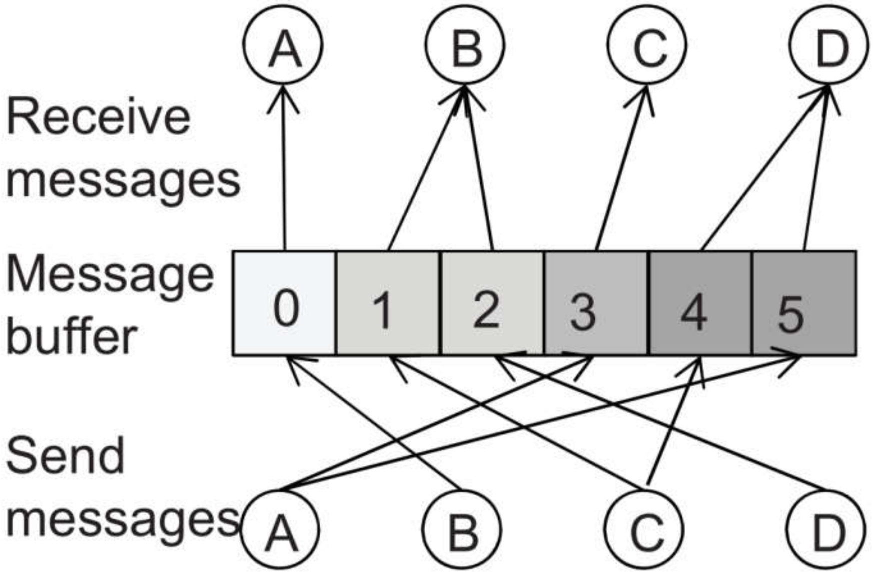
Message Passing Efficiency

- EMV→ sent/received along the edge
- Set maximum number of messages
  - the maximum total number of messages
  - the maximum number of messages that each vertex can receive
- Avoid the grouping operation
  - reversed edge indexed message passing
  - Write positions→ consecutive

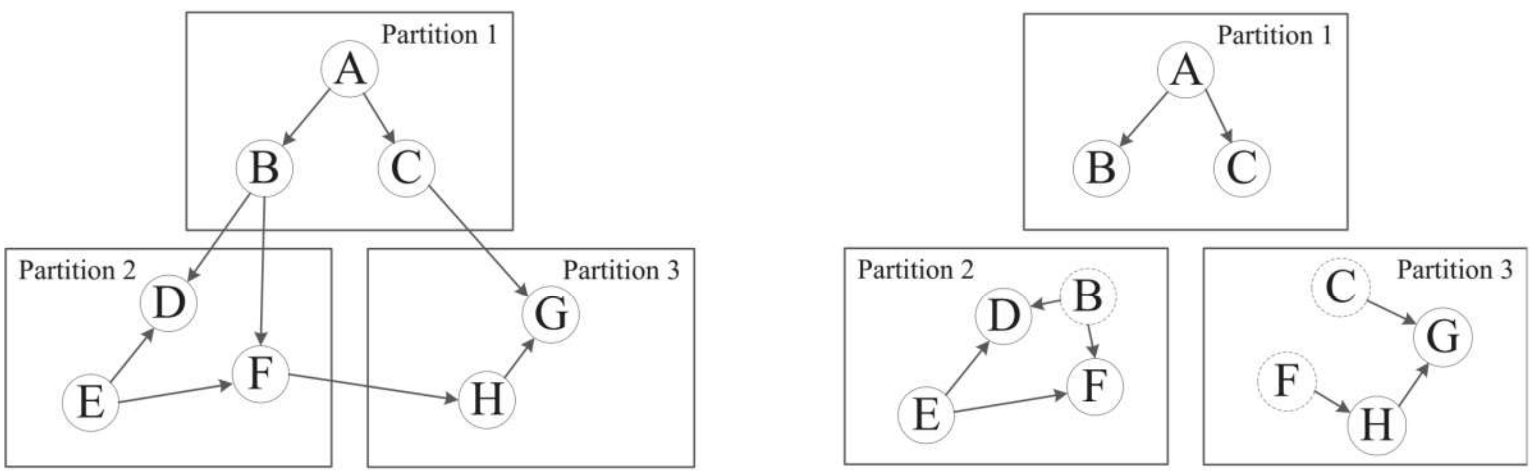
Reversed Edge Indexed



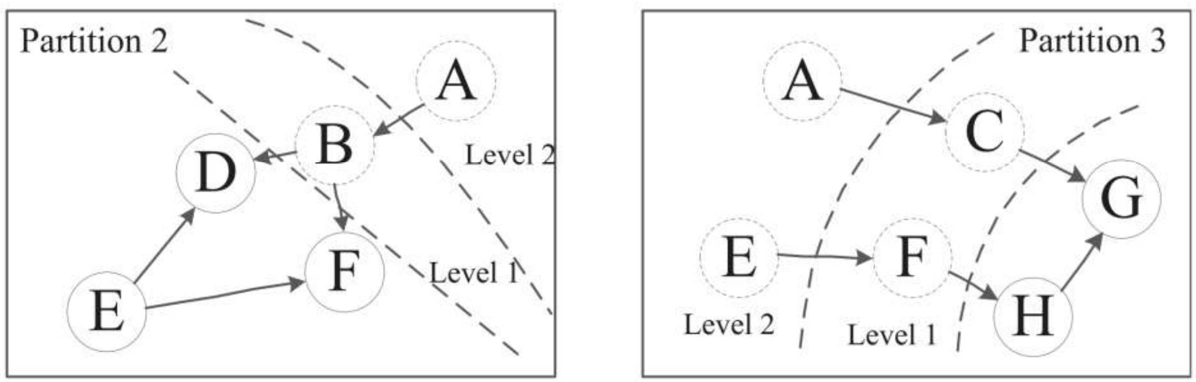
Reversed Edge Indexed



Multi-GPU Execution (Replication)



Multi-GPU Execution (Multi-hop)



Evaluation

- Workstation specs:
  - NVIDIA Tesla C2050
  - 2× Intel Xeon E5645 2.4GHz
  - 24 GB Ram

- Common Graph Processing Operations
  - PageRank
  - Maximum Barpartite Matching (MBM)
  - Breadth First Search (BFS)
  - Single Source Shortest Path (SSSP)

Evaluation (Cont'd)

- Graph Categories
  - Real-World
  - Synthetic

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

Comparison

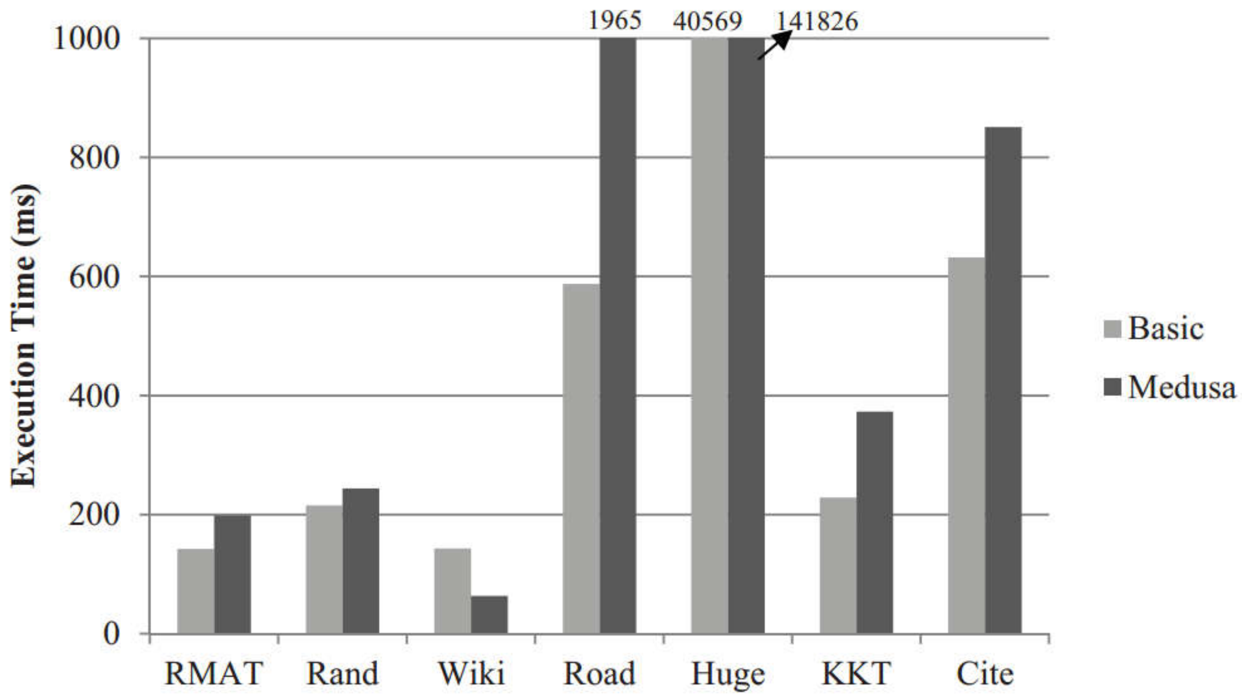
- Medusa Vs Manual Implementation
- Two Approaches
  - Queue-based (-Q)
  - Non Queue-based (-N)

TEPS

- What is TEPS?
  - Traversed Edges Per Second
- As a metric for Super Computers
- TEPS Vs FLOPS
  - Aims to measure communication capabilities

Source:
graph500.org
Wikipedia.org

Performance



TEPS Comparison

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

TEPS Comparison (Cont'd)

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

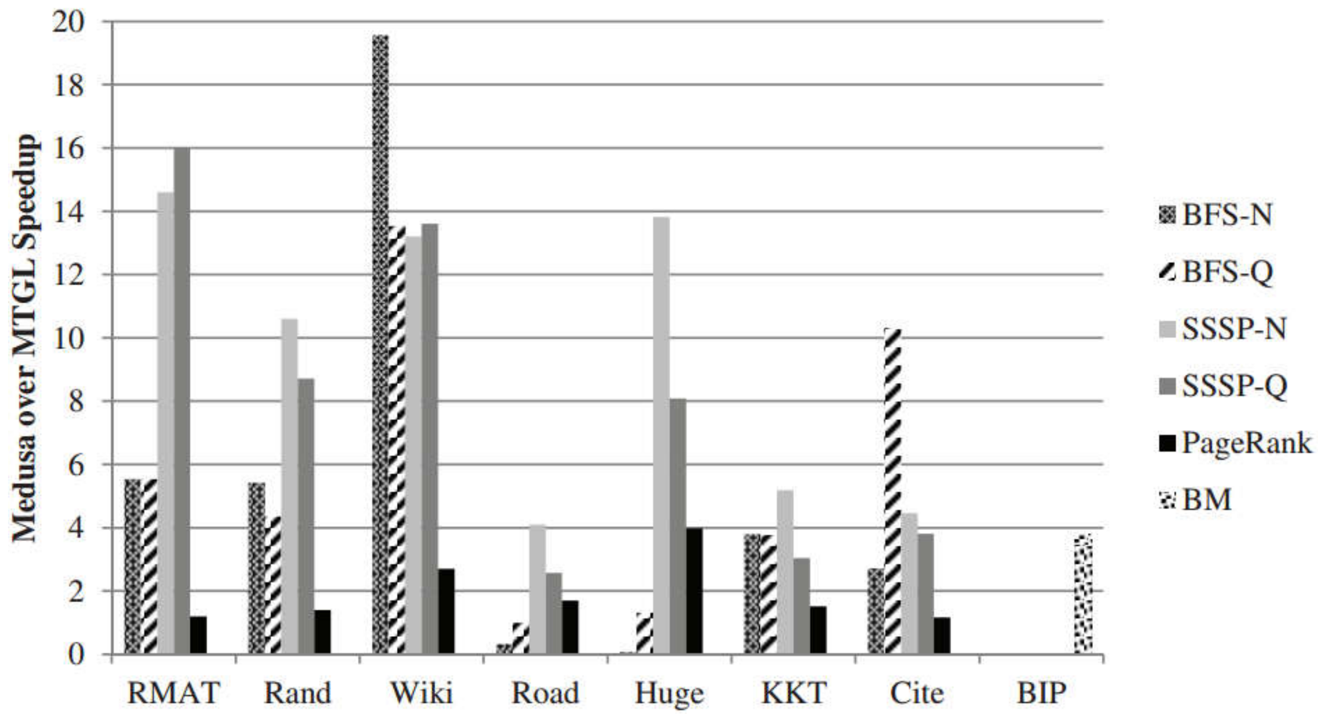
Programmability

- Code Complexity
  - Number of lines!

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

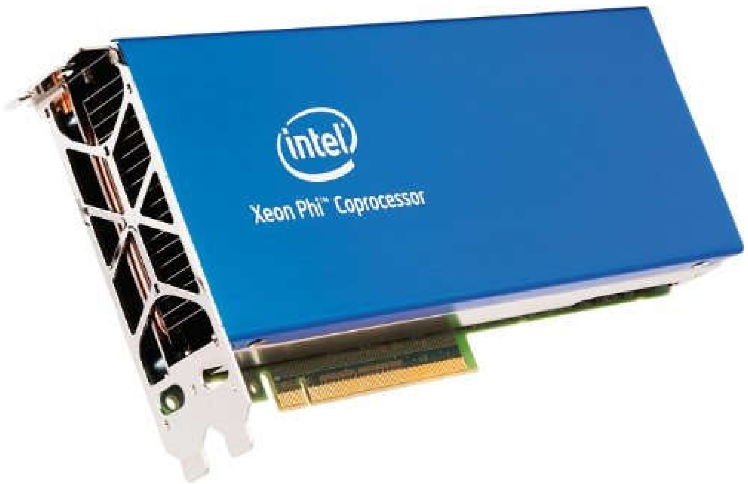
Efficiency

- Compare with MTGL
- Speedup
  - MTGL / Medusa



Future Work

- Medusa on other architectures like:
  - Intel Xeon Phi™ 😍
- Extend Medusa to distributed systems
- Automatic decision on dynamic-queue



CONCLUSIONS

- New programming framework named Medusa
  - efficiency and programmability of GPU-based parallel graph processing
  - hide the programming complexity
  - only need to write sequential programs