Medusa Implementation

By:

Sepand Haghighi

Vahid Amini

Amin Fotovat

Overview

- Previous talk
- Implementation
- Results

Let's Flashback!

- Why graphs?
  - Many real-world applications:
    - Social Networks
    - Web link analysis
- Challenging CUDA code?
  - Solution: sequential programming framework

GPU As An Accelerator

- GPU-based solutions vs CPU-based
  - Improved performance
  - Limited to specific graph operations
  - Optimization for each graph processing task

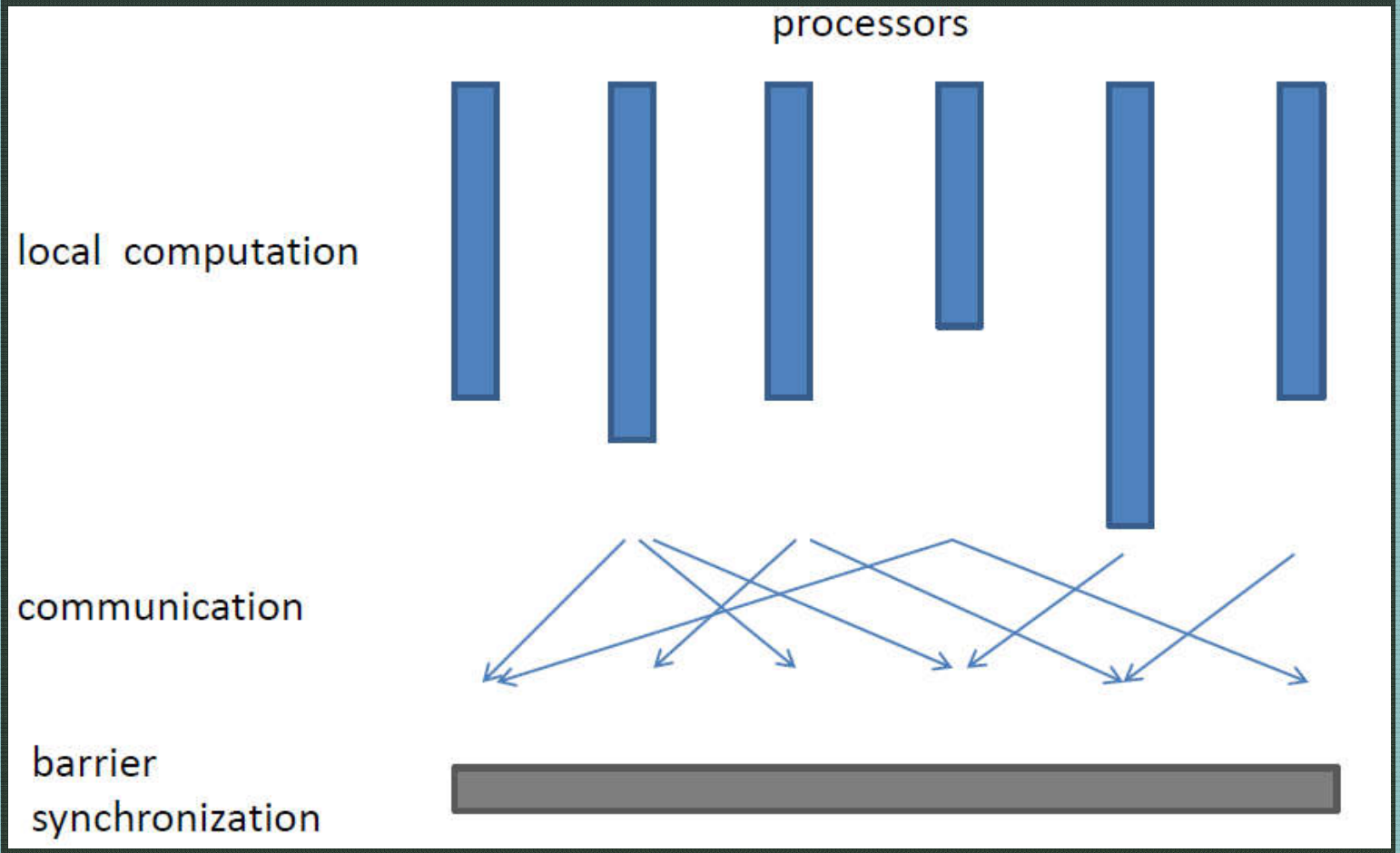
What Medusa Offers

- A Soft Framework
  - Simplify Programming
- Novel Graph Programming Model
  - Edge-Message-Vertex
    - Suitable for parallel graph processing

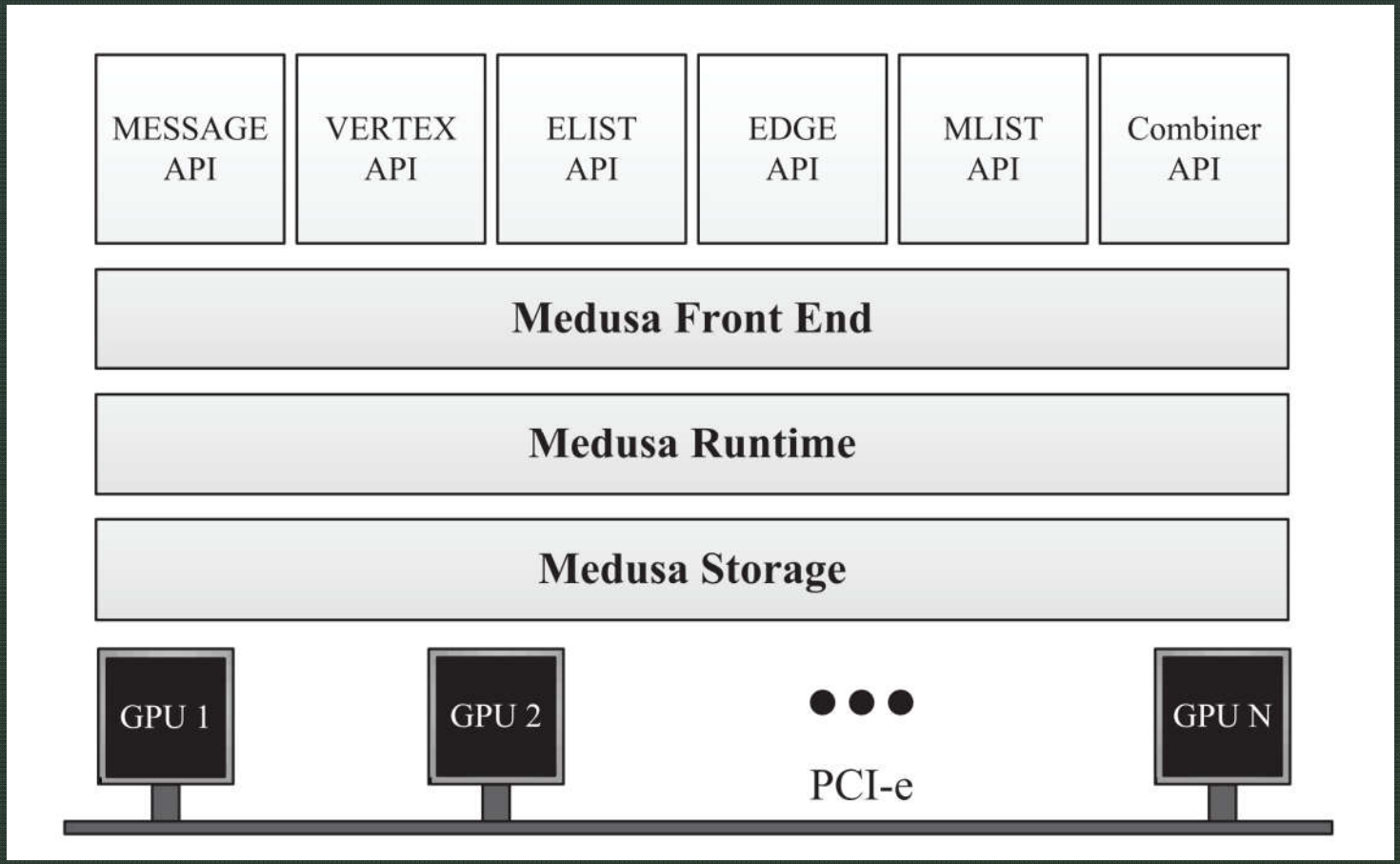
Bulk Synchronous Parallel

- A computation contains serial global steps
- Each processor
  - Fast local memory
  - Communication network
- Google adopt this for graph processing
  - Pregel
  - Mapreduce

Bulk Synchronous Parallel



Overview of Medusa



Medusa Workflow

- Define basic structures
  - Edge, Message and Vertex
- Composes the main program
  - Initializing the graph structure
  - Configuring the framework parameters
  - Invoking the customized emv apis with the system provided apis

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

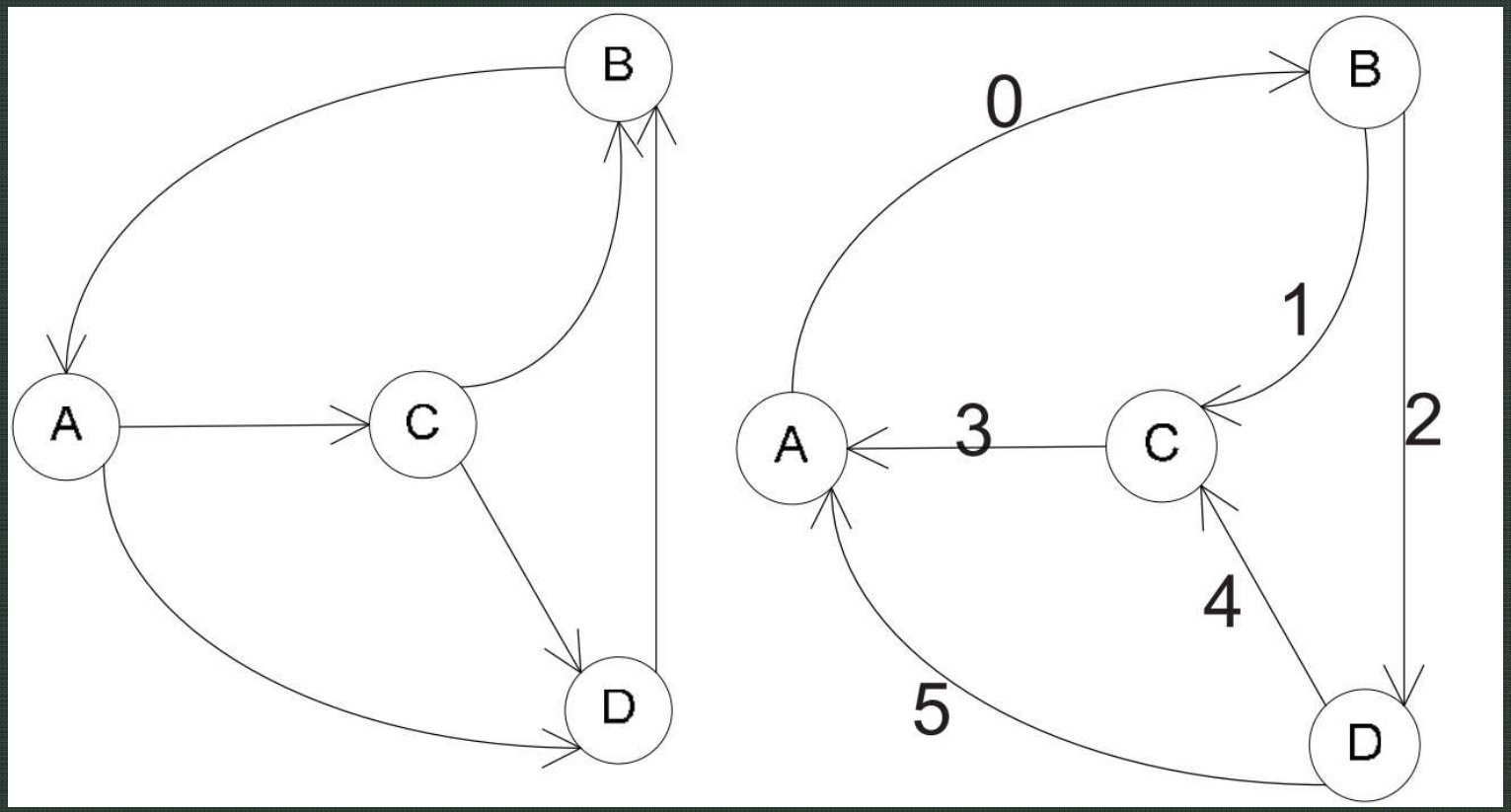
Work Efficiency

- Medusa's Default
  - Applies the user-defined API on all vertices/edges
- SetActive API
  - Dynamic queue
  - Apply the EMV APIs to the active vertices and edges only

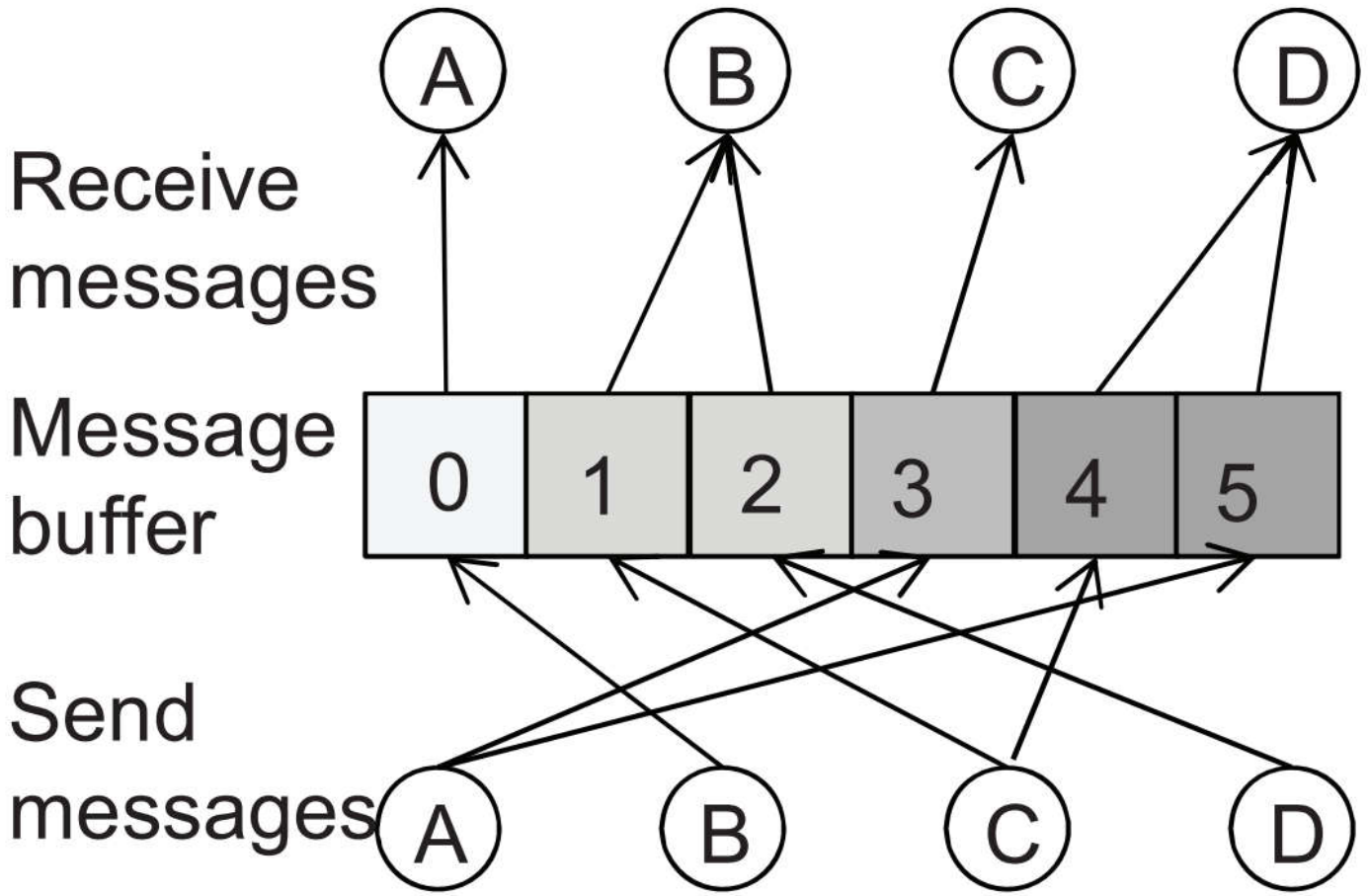
Message Passing Efficiency

- Fixed-size array
  - Information of the buffer size
  - Output position
  - Need grouping operation
- Dynamic list
  - No pre-computation
  - No grouping operation
  - Need atomic operation

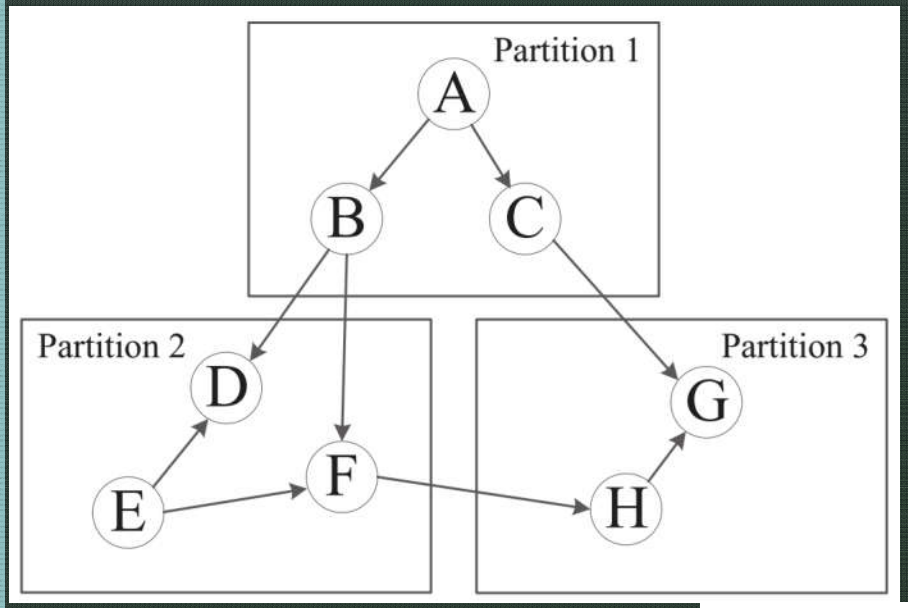
Reversed Edge Indexed

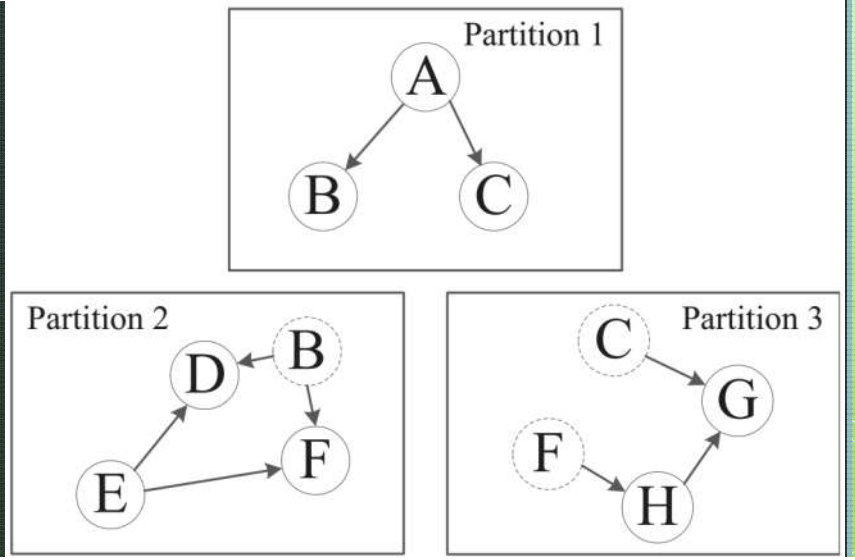


Reversed Edge Indexed

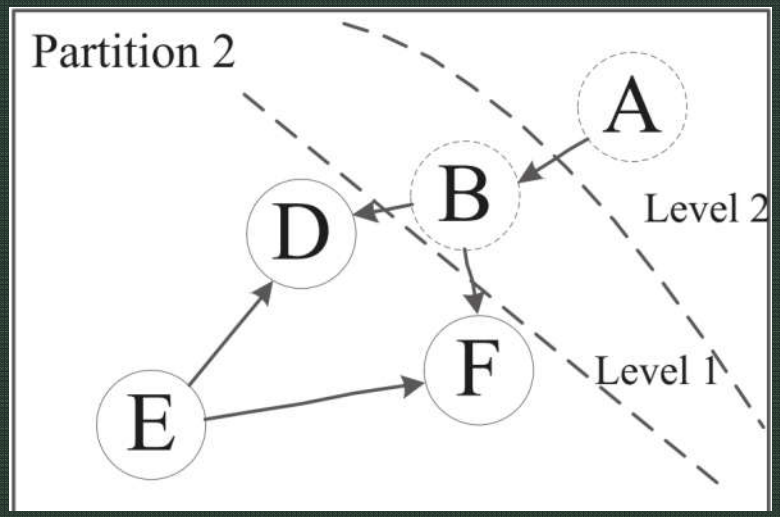


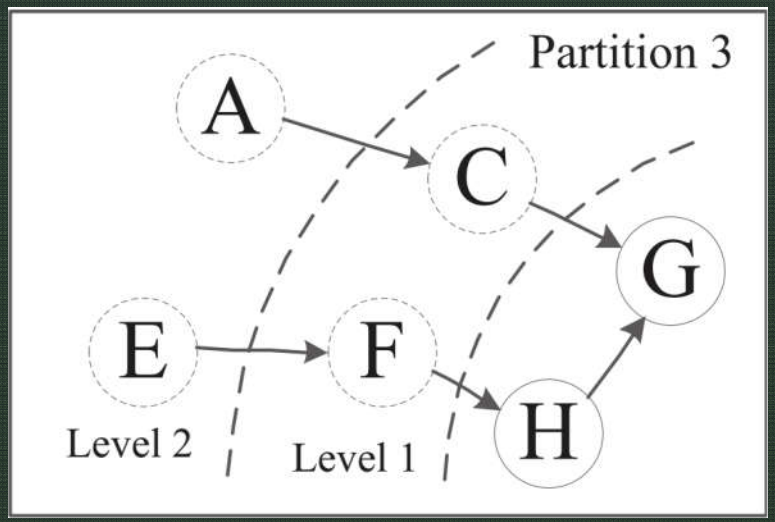
Multi-GPU Execution (Replication)



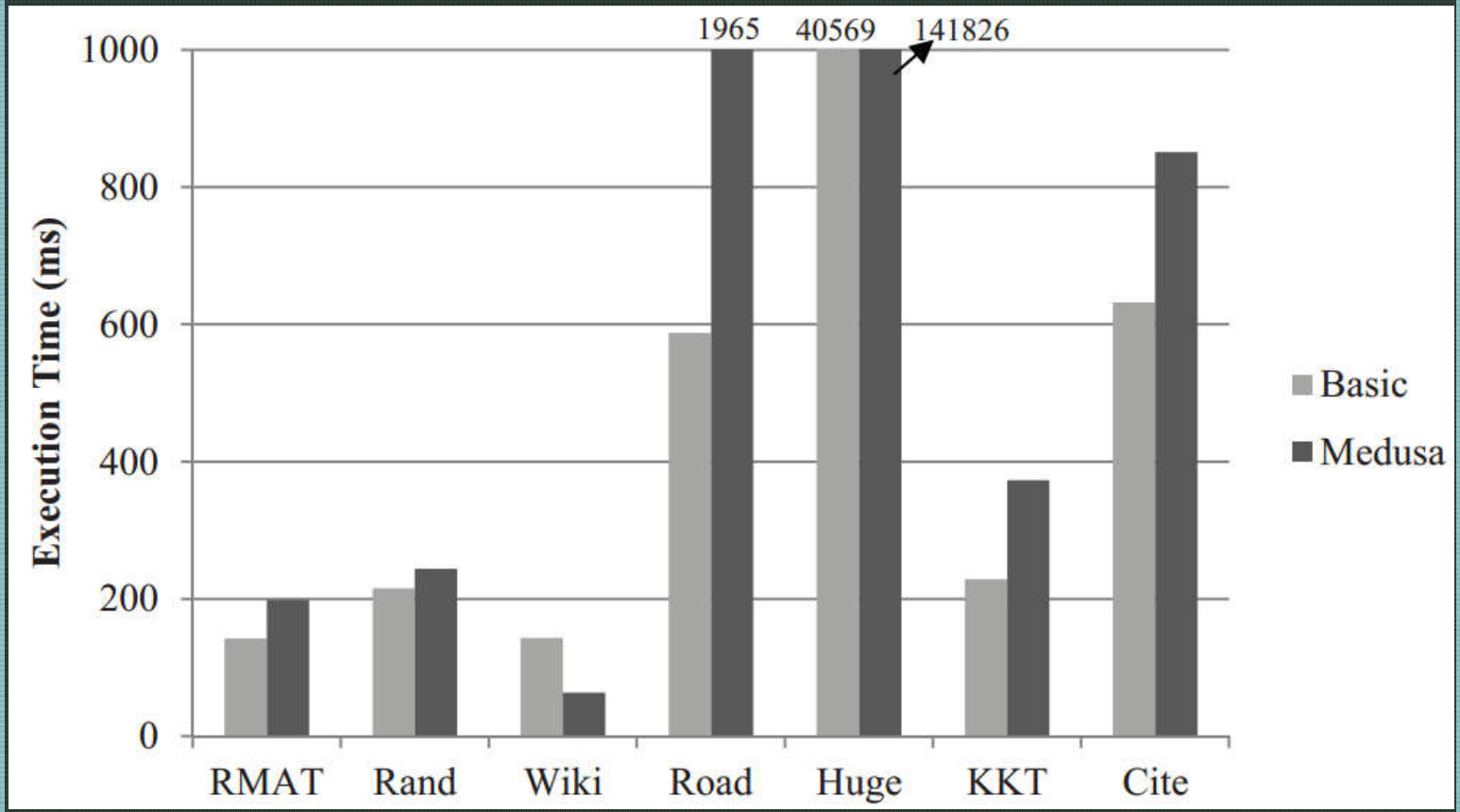


Multi-GPU Execution (Multi-hop)





Performance



TEPS Comparison

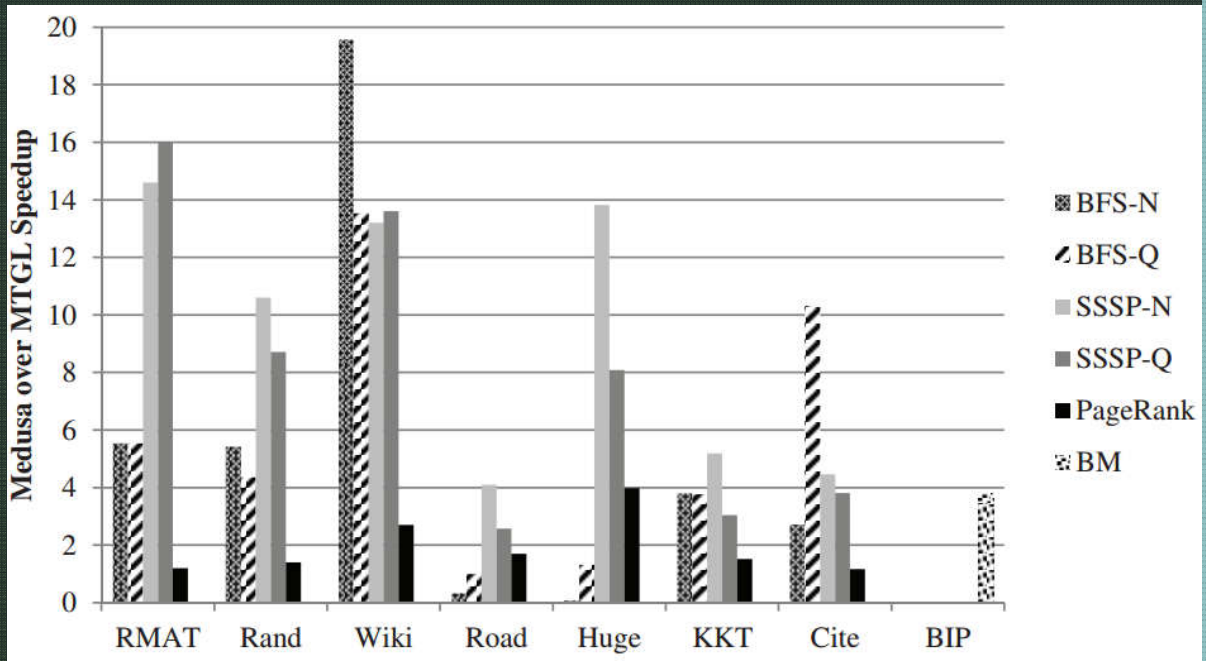
	Basic	Warp-centric	Medusa
Wiki	61.4	152.9	1091.1
Road	26.2	45.7	63.5
RMAT	593.2	971.1	895.8
Rand	648.6	844.95	765.8
Huge	5.7	1.3	68.1
KKT	480.7	175.7	351.5
Cite	1460.4	1503.1	2686.7

Programmability

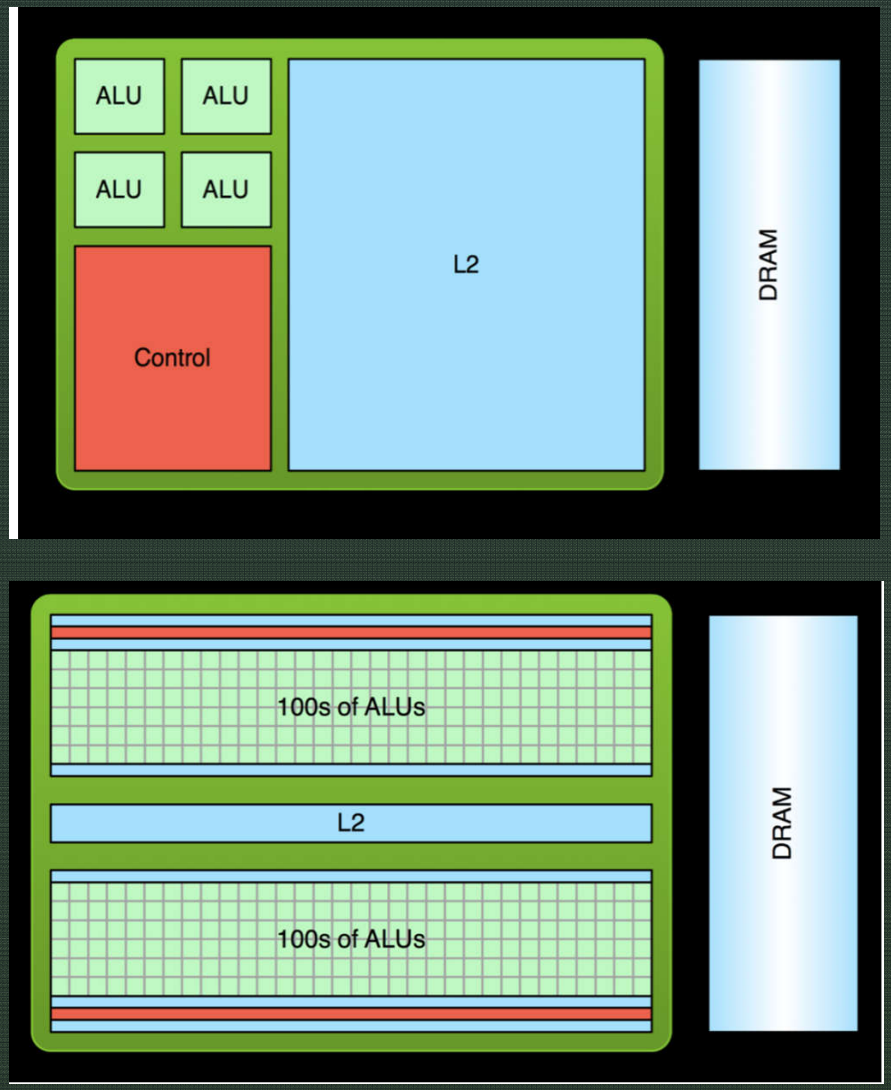
	Baseline	Warp-centric	Medusa (N/Q)
GPU code lines (BFS)	56	76	9/7
GPU code lines (SSSP)	59	N.A.	13/11
GPU memory management	Yes	Yes	No
Kernel configuration	Yes	Yes	No
Parallel programming	Thread	Thread+Warp	No

Efficiency

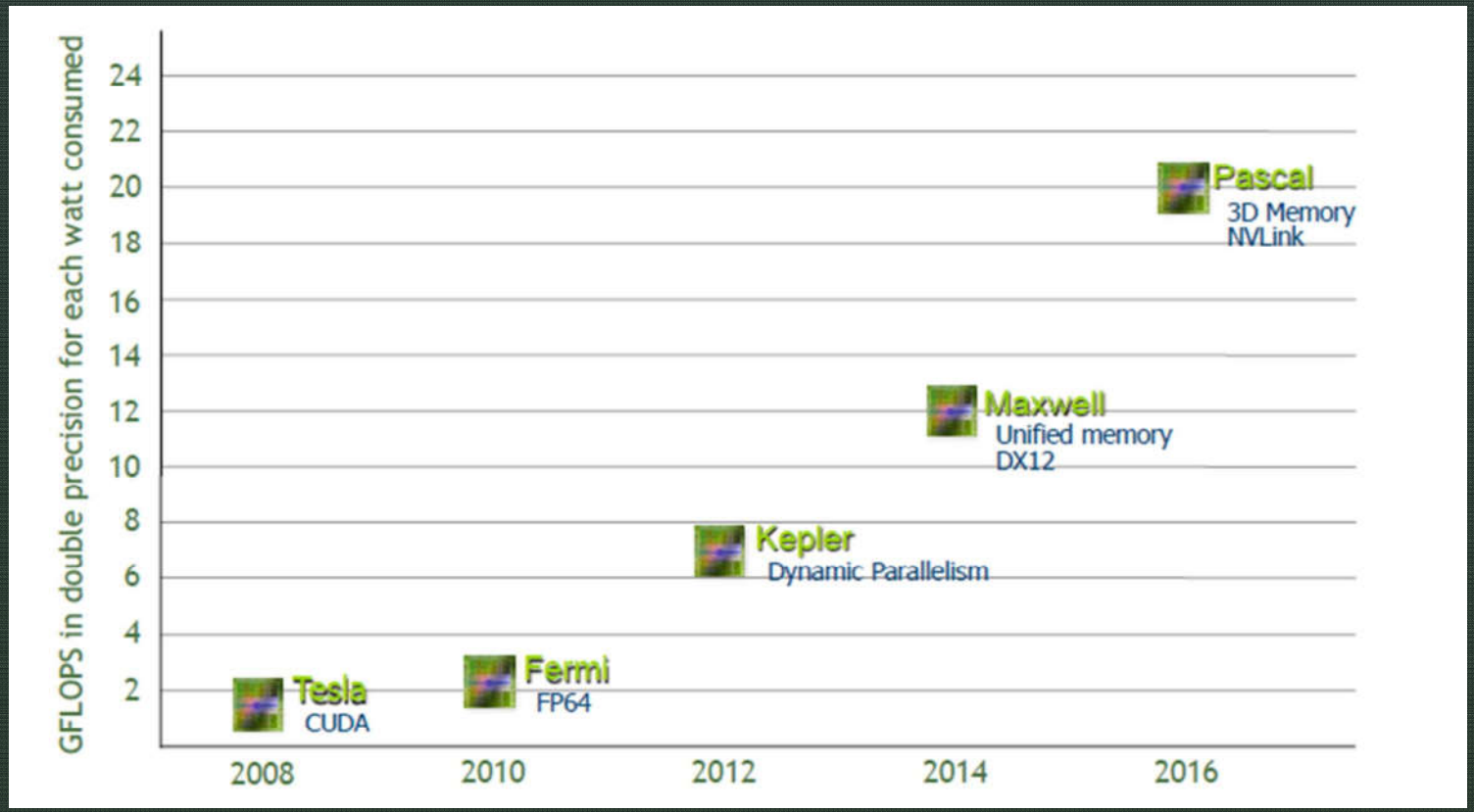
- Compare with MTGL
- Speedup
  - MTGL / Medusa



GPU vs CPU


ALU
ALU
ALU
ALU
Control
L2
DRAM
100s of ALUs
L2
100s of ALUs
DRAM

Hardware Generation



Nvidia Fermi GPU (2010)

- 3.0B transistors
- 15SMs
- 32 CUDA cores per SM

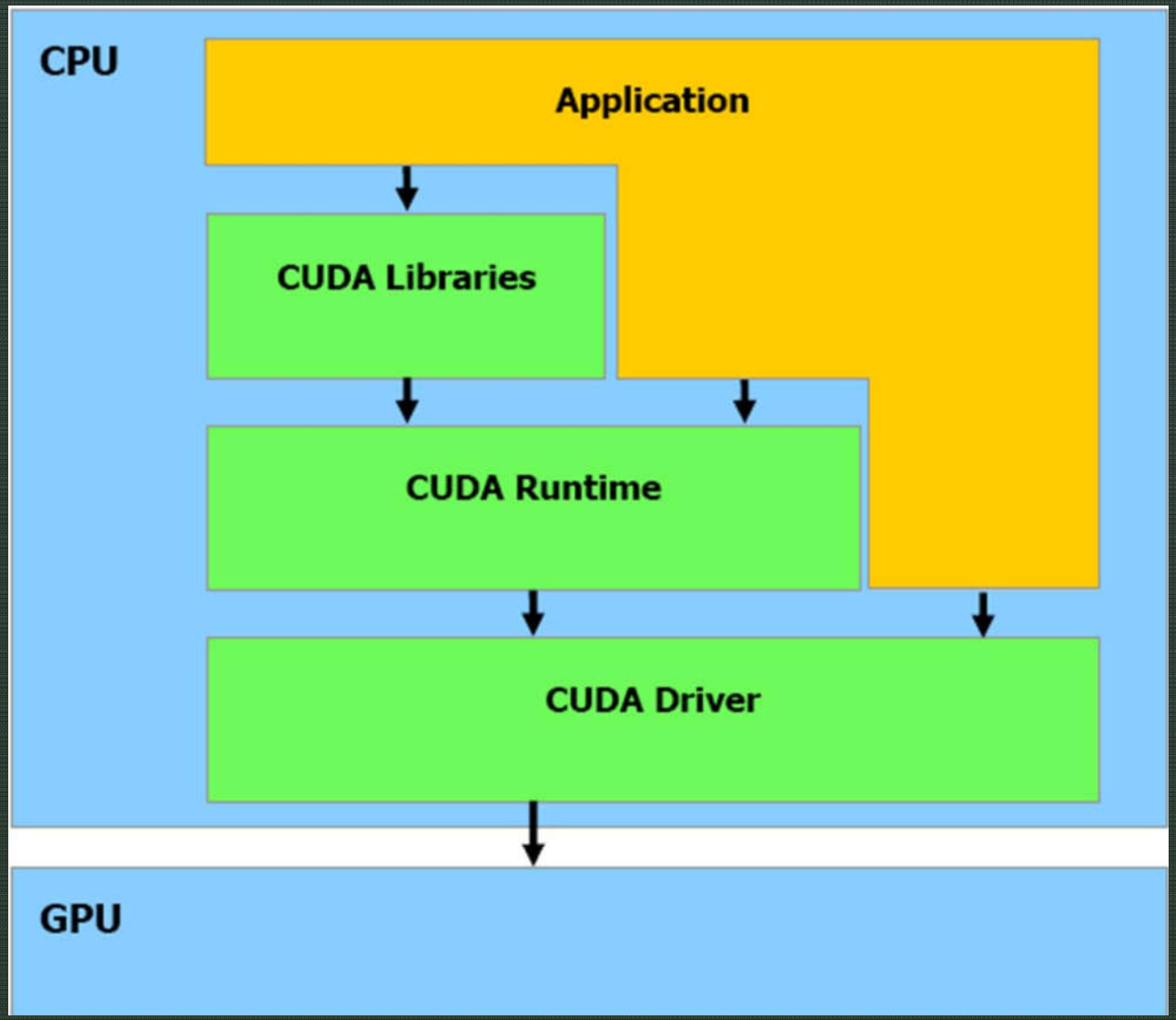
Nvidia Kepler (2012)

- 7.1B transistors
- 15 SM
- 192 CUDA cores per SM
- 6 64-bit memory controller

Nvidia MAXWELL (2015)

- 5.2B transistors
- 16 SM
- 2048 CUDA Cores

CUDA Software Stack



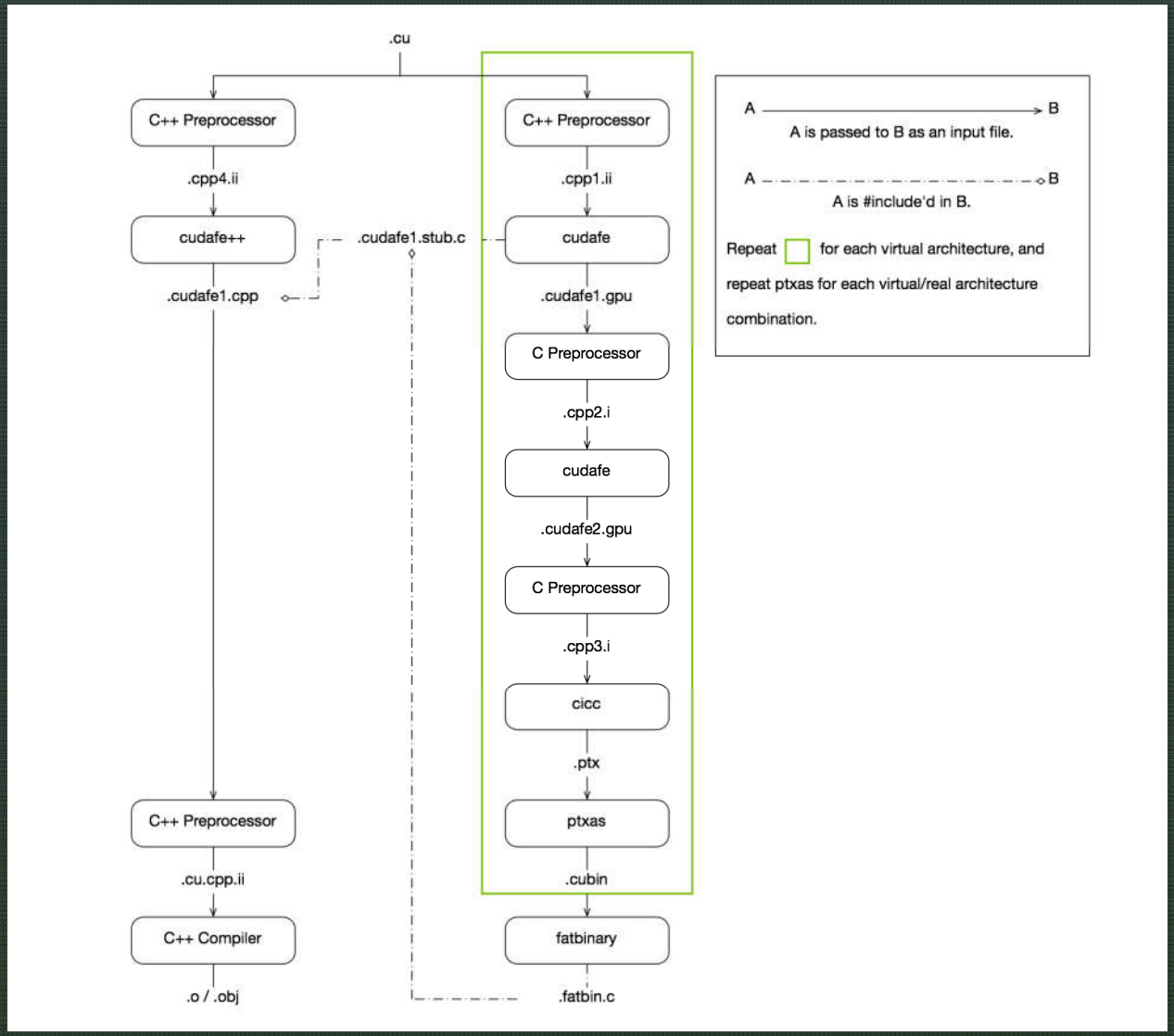
CUDA BFS Sample

```
Node* d_graph_nodes;
    cudaMalloc( (void**) &d_graph_nodes,
sizeof(Node)*no_of_nodes) ;
    cudaMemcpy( d_graph_nodes, h_graph_nodes,
sizeof(Node)*no_of_nodes, cudaMemcpyHostToDevice);

    int* d_graph_edges;
    cudaMalloc( (void**) &d_graph_edges,
sizeof(int)*edge_list_size) ;
    cudaMemcpy( d_graph_edges, h_graph_edges,
sizeof(int)*edge_list_size, cudaMemcpyHostToDevice);

    //Copy the Mask to device memory
    bool* d_graph_mask;
    cudaMalloc( (void**) &d_graph_mask,
sizeof(bool)*no_of_nodes) ;
    cudaMemcpy( d_graph_mask, h_graph_mask,
sizeof(bool)*no_of_nodes, cudaMemcpyHostToDevice);
```

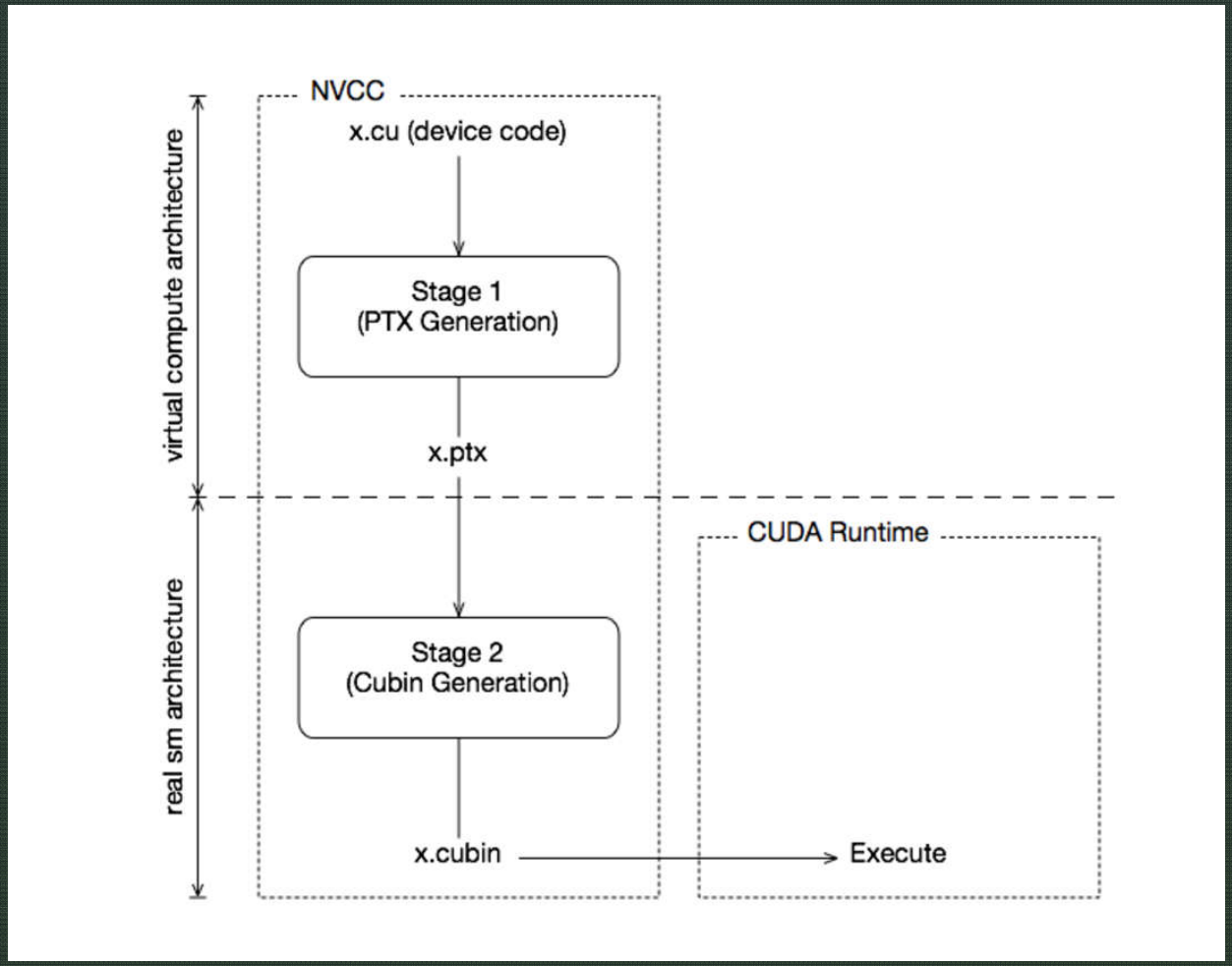
CUDA Kernel

```
#ifndef _KERNEL_H_
#define _KERNEL_H_

__global__ void
Kernel( Node* g_graph_nodes, int* g_graph_edges,

 …
{
    int tid = blockIdx.x*MAX_THREADS_PER_BLOCK +
threadIdx.x;
    if( tid<no_of_nodes && g_graph_mask[tid])
    {
        g_graph_mask[tid]=false;
        for(int i=g_graph_nodes[tid].starting;
i<(g_graph_nodes[tid].no_of_edges +
g_graph_nodes[tid].starting); i++)
            {
            int id = g_graph_edges[i];

…
```

CUDA Compilation



CUDA Compilation (cont'd)



IMPLEMENTATION

Medusa Requirements

- Programming
  - CUDA 4.2
  - C++ 03
- Build
  - gcc
  - CUDA nvcc

How to build Medusa

```
CUDA_SDK_PATH :=/home/jianlong/cuda4.2/NVIDIA_GPU_Computing_SDK
CUDPP_PATH := /home/jianlong/Projects/cudpp2.0
OPT=-O3
CFLAGS=$(OPT) -c -w -fopenmp -lgomp
INCLUDES :=  -I$(CUDA_SDK_PATH)/C/common/inc -I/home/jianlong/Projects/cudpp2.0/include
CULIBS := -L$(CUDA_SDK_PATH)/C/lib -lcutil_x86_64 -L/lib -lcudpp
NVFLAGS=$(OPT) -arch=sm_20 $(INCLUDES)
LDFLAGS=$(OPT) $(CULIBS)
CC=g++
NVCC=/home/jianlong/cuda4.2/toolkit/bin/nvcc -Xcompiler -fopenmp
LD=/home/jianlong/cuda4.2/toolkit/bin/nvcc  -lgomp -lpthread
```

Difficulties

- Graph Generator Problem
- Bad Documentation
- About 4000 lines of code without comments
- No details about compiler version
- Cuda backward compatibility problem
- Old Nvidia Architecture Kernels

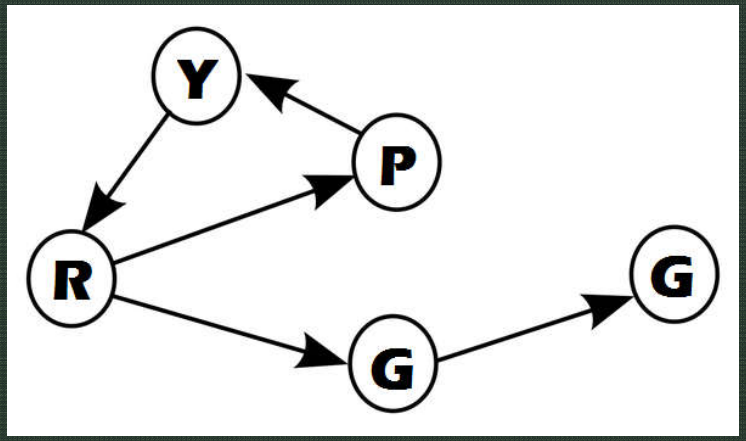
Maxwell Compatibility Problem

- Needs PTX if application built using CUDA v. 2.1~ 5.5
- Different major version numbers aren't compatible with each other

http://docs.nvidia.com/cuda/maxwell-compatibility-guide/#axzz4iGD7BuLS

Python Random Graph Generator

- Based on DIMACS
- Written in Python
- Available on our repo (Github)



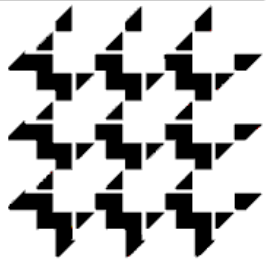
http://github.com/sepandhaghighi/pyrgg

DIMACS Standard

- A graph contains *n* nodes and *m* arcs
- Nodes are identified by integers 1...*n*
- Graphs can be interpreted as directed or undirected
- Graphs can have parallel arcs and self-loops
- Arc weights are signed integers

DIMACS

Center for Discrete Mathematics & Theoretical Computer Science
Founded as a National Science Foundation Science and Technology Center



Pyrgg (branch_gen)

```
def branch_gen(random_edge,vertices_number,min_range,max_range):
    '''
    This function generate branch and weight vector of each vertex
    :param random_edge: number of vertex edges
    :type random_edge:int
    :param vertices_number: number of vertices
    :type vertices_number:int
    :param min_range: weight min range
    :type min_range:int
    :param max_range: weight max range
    :type max_range:int
    :return: branch and weight list
    '''
    index = 0
    branch_list = []
    weight_list=[]
    while (index < random_edge):
        random_tail = random.randint(1, vertices_number + 1)
        random_weight=random.randint(min_range,max_range)
        if random_tail not in branch_list:
            branch_list.append(random_tail)
            weight_list.append(random_weight)
            index += 1
    return [branch_list,weight_list]
```

Pyrgg (edge_gen)

```
def edge_gen(vertices_number,min_range,max_range,min_edge,max_edge):
    '''
    This function generate each vertex connection number
    :param vertices_number: number of vertices
    :type vertices_number:int
    :param min_range: weight min_range
    :type min_range:int
    :param max_range: weight max_range
    :type max_range:int
    :return: list of 2 dictionary
    '''
    temp=0
    vertices_id=list(range(1,vertices_number+1))
    vertices_edge=[]
    weight_list=[]
    for i in vertices_id:
        random_edge=random.randint(min_edge,max_edge)
        temp_list=branch_gen(random_edge,vertices_number,min_range,max_range)
        vertices_edge.append(temp_list[0])
        weight_list.append(temp_list[1])
        temp=temp+random_edge
    return [dict(zip(vertices_id,vertices_edge)),dict(zip(vertices_id,weight_list)),temp]
```

Output

```
c FILE                          :100.gr
c No. of vertices               :100
c No. of directed edges         :4538
c Max. weight                   :200
c Min. weight                   :2
p sp 100 4538
c 1 53 2
c 1 52 29
c 1 73 145
c 1 50 111
c 1 2 198
c 1 33 50
c 1 85 106
c 1 24 108
c 1 18 83
```


Q&A