DCUDA: Dynamic GPU Scheduling with Live Migration Support

Fan Guo[1], **Yongkun Li**[1], John C.S. Lui[2], Yinlong Xu[1]

[1]University of Science and Technology of China

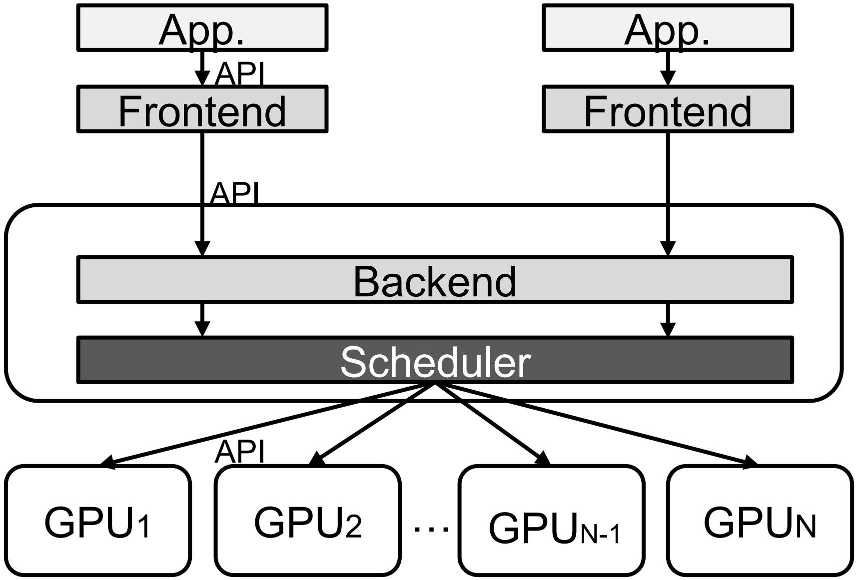
[2]The Chinese University of Hong Kong

Outline

1. Background & Problems
2. DCUDA Design
3. Evaluation
4. Conclusion

GPU Sharing and Scheduling

- GPUs are underloaded without sharing
  - ✓ A server may contain multiple GPUs
  - ✓ Each GPU contains thousands of cores
- GPU sharing allows multiple apps to run concurrently on one GPU

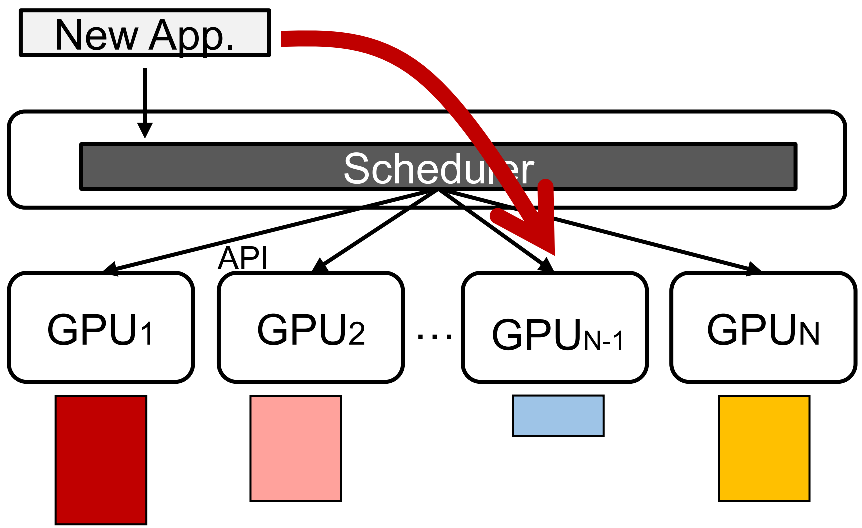


GPU scheduling is necessary

Load balance GPU utilization

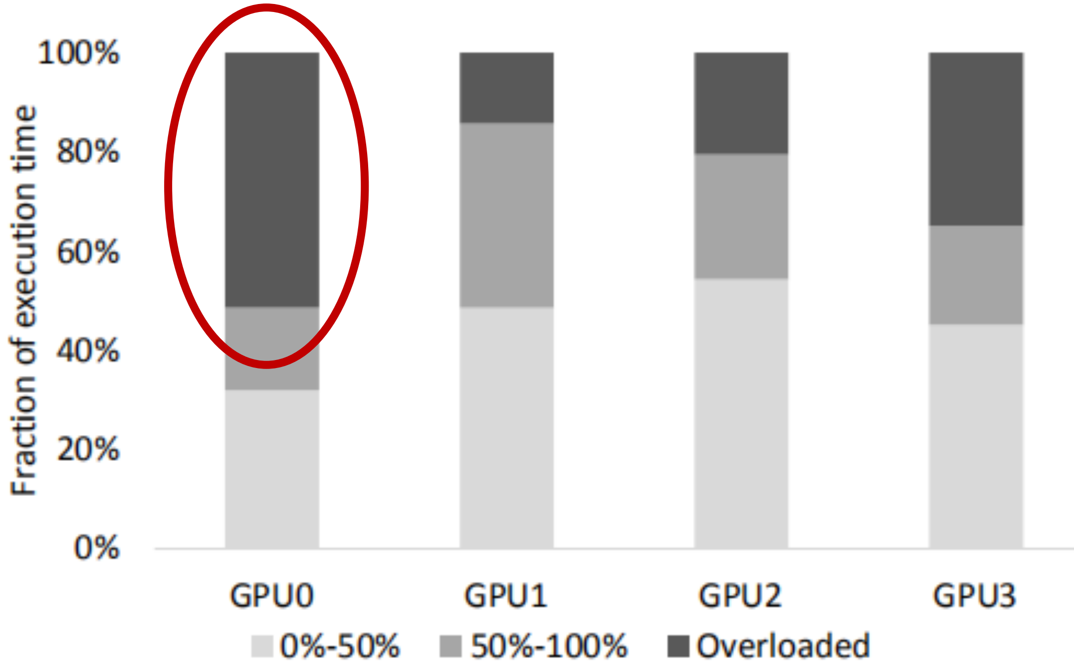
Current Scheduling Schemes

- Current schemes are “static”
  - ✓ Round-robin, prediction-based, least-loaded
  - ✓ They only make the assignment of applications before running them
- State-of-the-art: Least-loaded scheduling
  - ✓ Assign new app to the GPU with the least load



Limitations of Static Scheduling

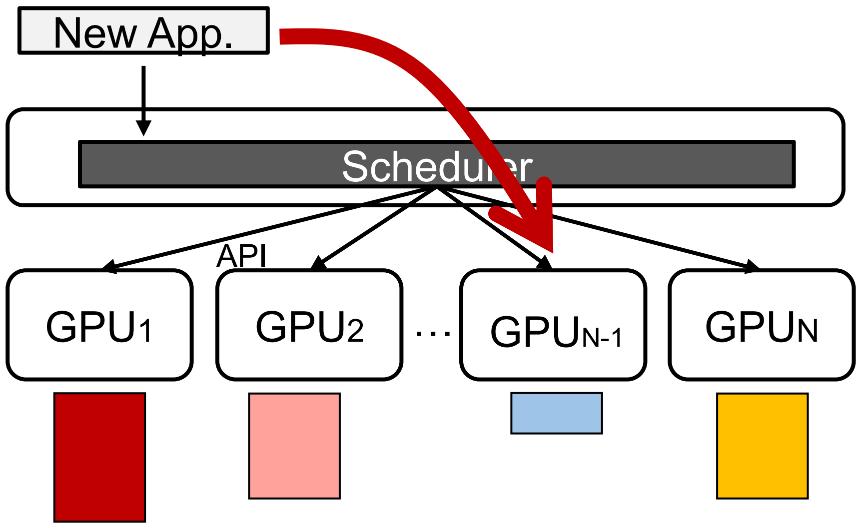
- Load imbalance (least-loaded scheduling)



The fraction of time in which at least one GPU is overloaded and some other GPU is underloaded accounts for up to 41.7% (overloaded: demand > GPU cores)

Limitations of Static Scheduling

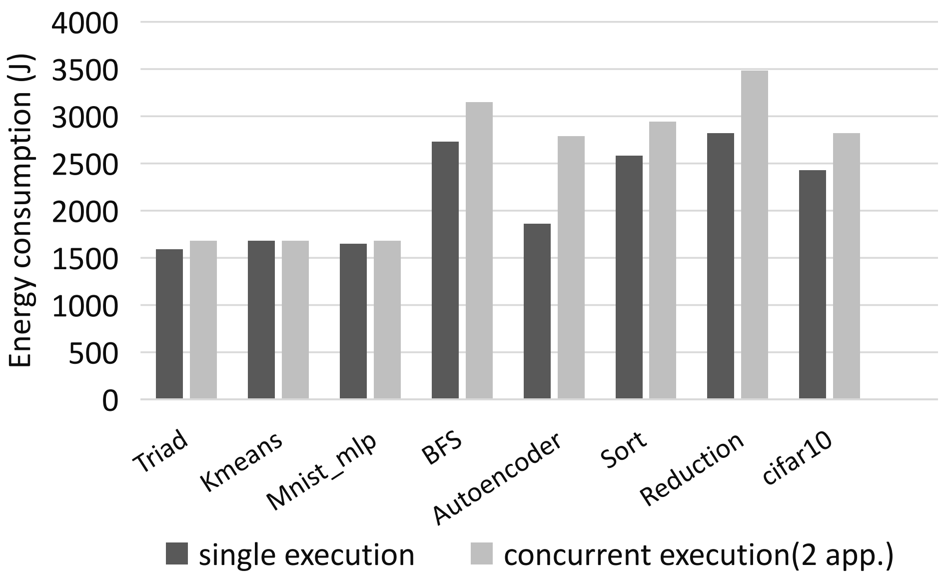
- Why does static scheduling result in load imbalance?



- Assign before running
  - ✓ Hard to get exact resource demand
  - ✓ The assignment is not optimal
- No migration support
  - ✓ No way to adjust online

Limitations of Static Scheduling

- Fairness issue caused by contention
  - ✓ Applications with low resource demand may be blocked by those with high resource demand
  - ✓ May also exists even with load-balancing schemes
- Energy inefficiency



Compacting multiple small jobs on one GPU saves energy

Our Goal

- Our goal is to design a scheduling scheme so as to achieve better
  - ✓ Load balance, energy efficiency, fairness
- Key idea: DCUDA

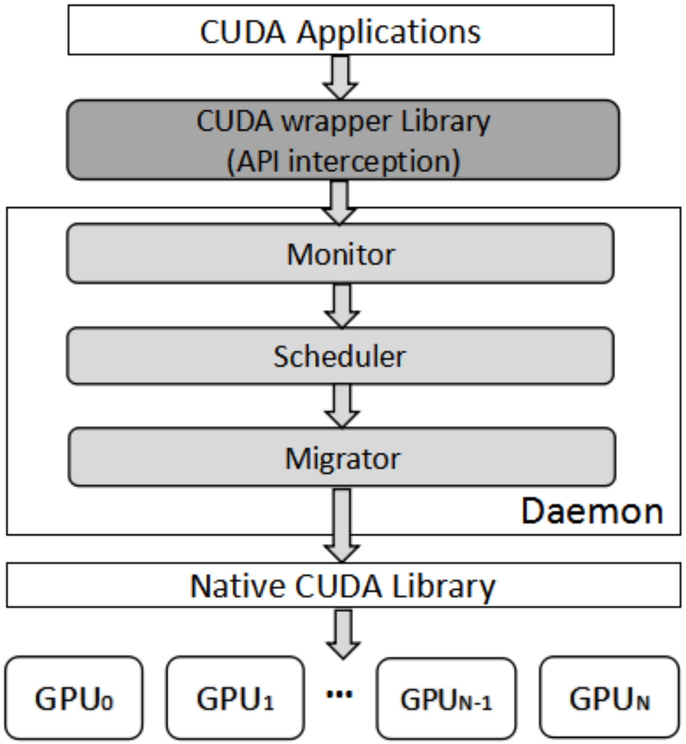
Dynamic scheduling (Schedule after running, fairness and energy awareness)	+	Online migration (running applications, not executing kernels)

Outline

1. Background & Problems
2. DCUDA Design
3. Evaluation
4. Conclusion

Overall Design

- DCUDA is implemented based on the API forwarding framework
- Key three modules at the backend
  - ✓ Monitor
    - GPU utilization
    - App's resource demand
  - ✓ Scheduler
    - Load balance
    - Energy efficiency
    - Fairness
  - ✓ Migrator
    - Migration of running app



The Monitor

- Resource demand of each application
  - ✓ GPU cores and GPU memory
  - ✓ Key challenge: lightweight requirement
- Demand on GPU cores
  - ✓ Existing tool (nvprof): large overhead (replay API calls)

Timer function

(Track info. only from parameters of intercepted API: #blk, #threads)

Optimization

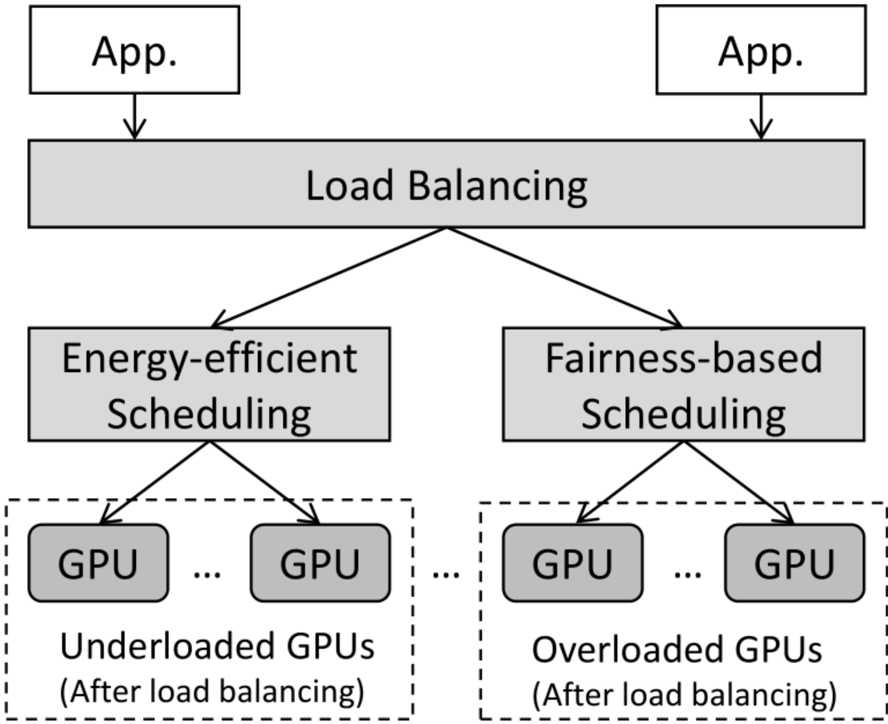
- ✓ Estimate only at the first time when the kernel func is called
- ✓ Use the recorded info. next time
- ✓ Rationale: GPU applications are iteration-based

The Monitor

- Demand on GPU memory
  - ✓ Easy to know allocated mem, but not all mem. are used
- How to detect actual usage?
  - ✓ Pointer check with cuPointerGetAttribute() + sampling
  - ✓ False negative: miss identification of used mem
    - On-demand paging (with unified mem support)
- Estimation of GPU utilization
  - ✓ Periodically scan the resource demand of applications
  - ✓ Aggregate them together

The Scheduler

- A multi-stage and multi-object scheduling policy



First priority:
Load balance

Case 1: (Slightly) overloaded GPU

Must avoid low-demand tasks being blocked

Case 2: Underloaded GPUs: Waste energy

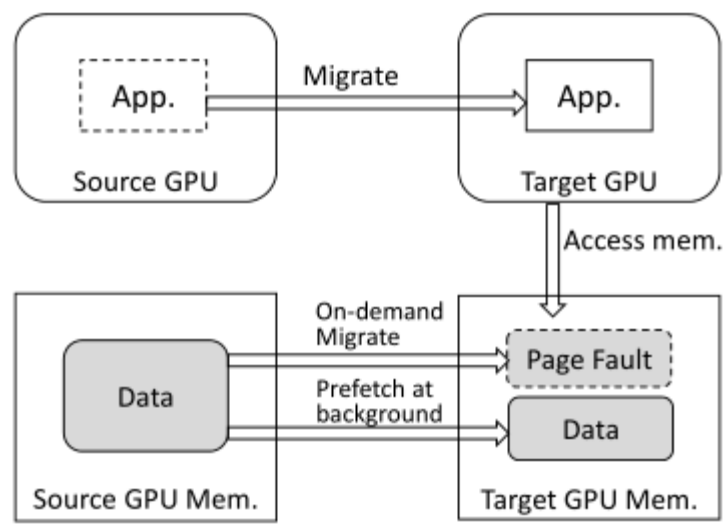
The Scheduler

- Load balance
  - ✓ Which GPUs: check each GPU pair
    - Feasible candidates: An overloaded + an underloaded
  - ✓ Which applications to migrate
    - Minimize migration frequency + avoid ping-pong effect
    - Greedy: Migrate the most heavyweight and feasible applications
- Energy awareness
  - ✓ Compact lightweight apps to fewer GPUs to save energy
- Fairness awareness: Grouping + time slicing

Tradeoff	⇐	Utilization	+	Fairness
Utilization vs fairness		Mixed packing		Priority-based scheme

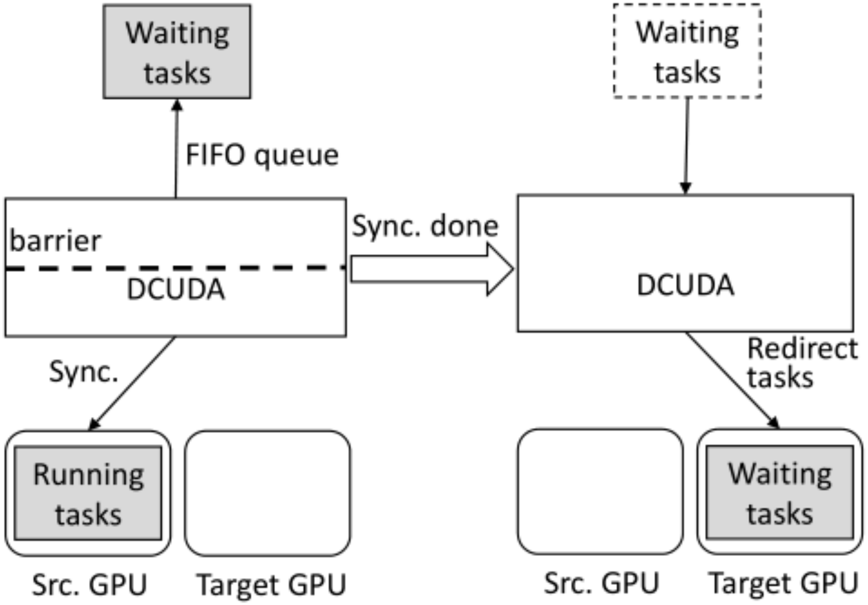
The Migrator

- Clone runtime
  - ✓ Largest overhead: initializing libraries (>80%)
  - ✓ Handle pooling: maintain a pool of libraries' handles for each GPU
- Migrate memory data
  - ✓ Leverage unified memory: Able to immediately run task without migrating data
  - ✓ Transparently support
    - Intercept API and replace
  - ✓ Pipeline
    - Prefetch & on-demand paging



The Migrator

- Resume computing tasks
  - ✓ Two states of tasks: running and waiting
    - Only migrate waiting tasks
  - ✓ Sync to wait for the completion of all running tasks
  - ✓ Redirect waiting tasks to target GPUs
    - Order preserving
    - FIFO queue



Outline

1 Background & Problems

2 DCUDA Design

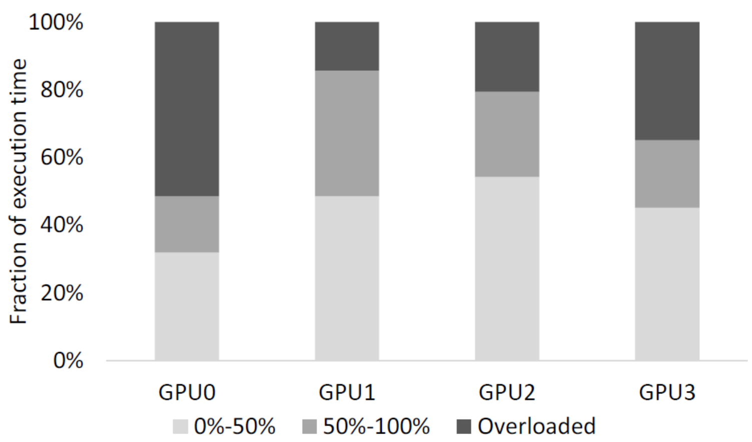
3 Evaluation

4 Conclusion

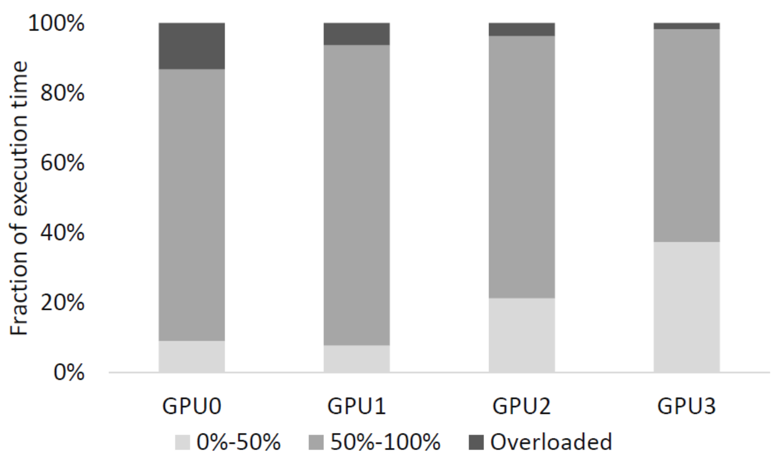
Experiment Setting

- Testbed
  - ✓ Prototype implemented based on CUDA toolkit 8.0
  - ✓ Four NVIDIA 1080Ti GPUs, each has 3584 cores and 12GB memory
- Workload
  - ✓ 20 benchmark programs which represent a majority of GPU applications (HPC, DM, ML, Graph Alg, DL)
  - ✓ Focus on randomly selected 50 sequences, each combines the 20 programs with a fixed interval
- Baseline algorithm
  - ✓ Least-loaded: most efficient static scheduling scheme

Load Balance



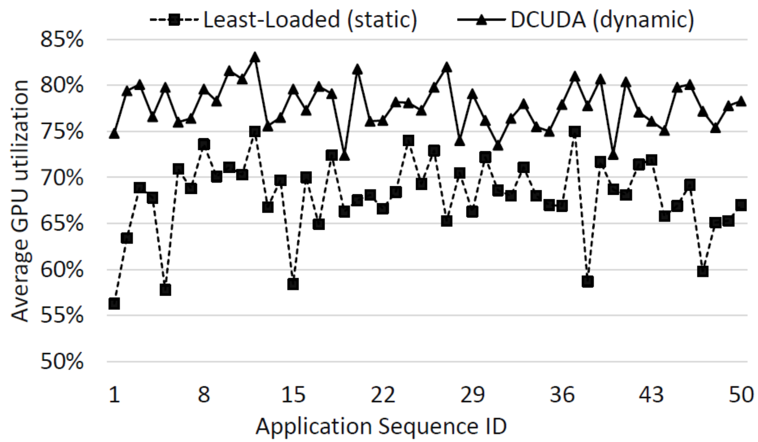
(a) GPU load with Least-Loaded scheduling



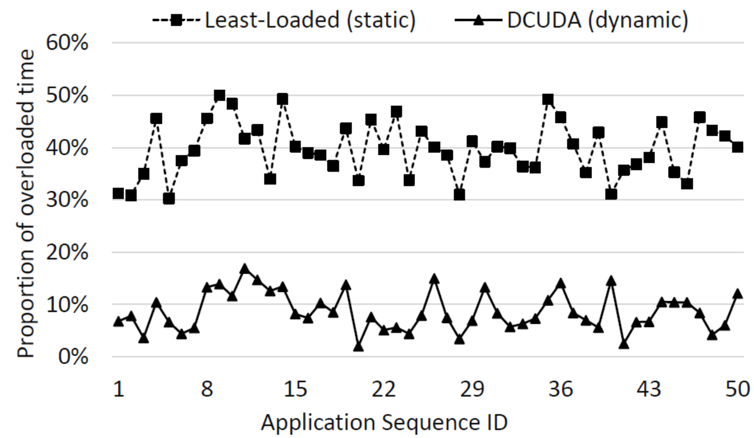
(b) GPU load with DCUDA

- Load states of GPU
  - ✓ 0%-50% utilization, 50%-100% utilization, and overloaded (demand > GPU cores)
- **Overloaded time** of each GPU
  - ✓ Least-loaded: 14.3% - 51.4%
  - ✓ **DCUDA: within 6%**

GPU Utilization



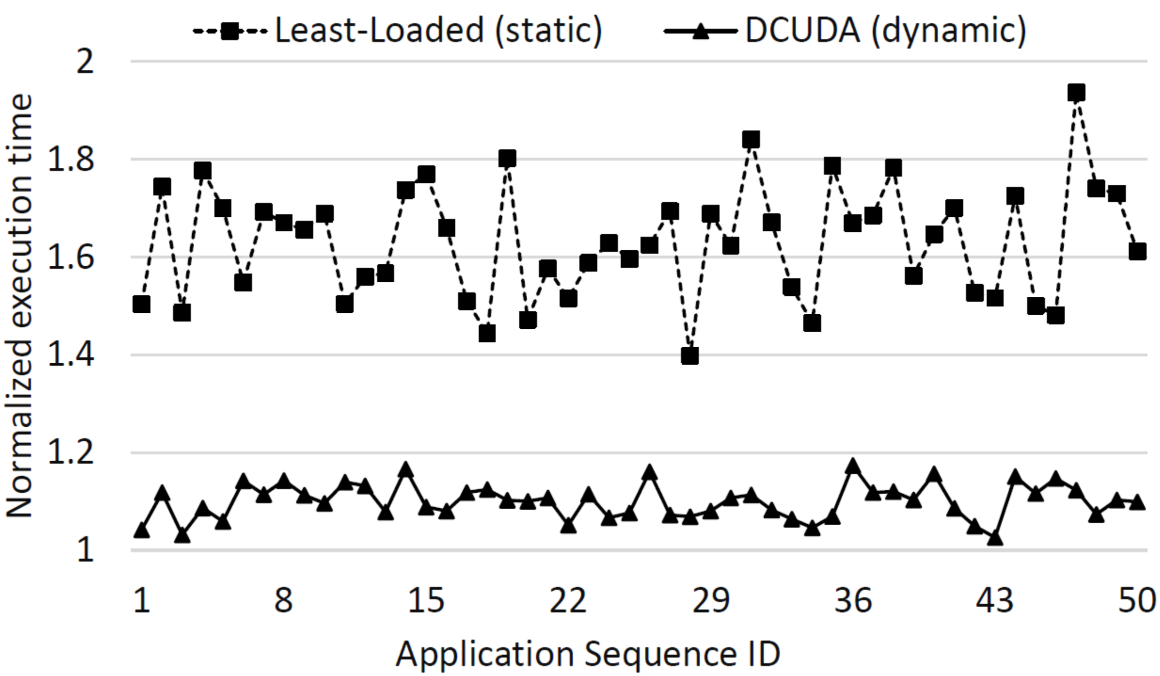
(a) Average GPU utilization



(b) Proportion of overloaded time

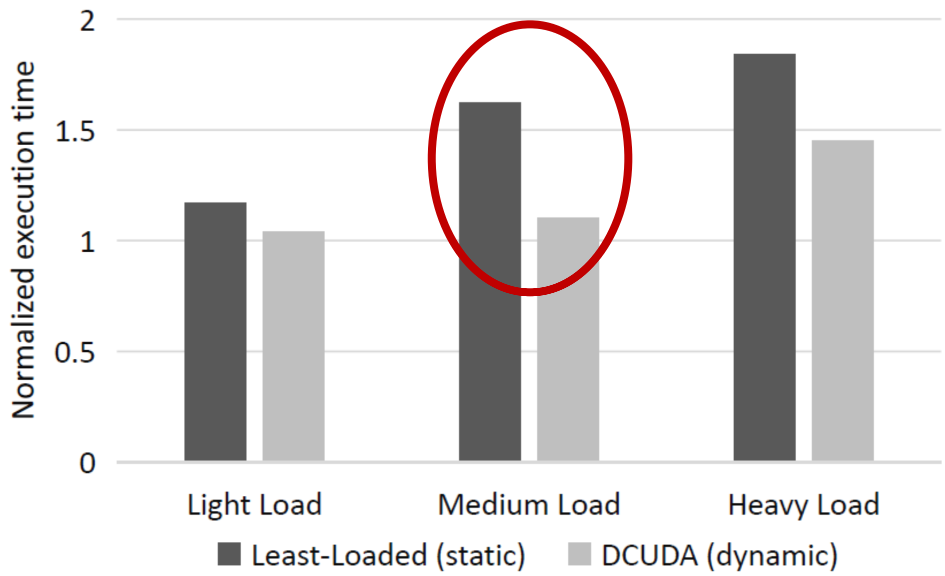
- Improves average GPU utilization by 14.6%
- Reduce the overloaded time by 78.3% on average (over the 50 sequences/workloads)

Application Execution Time



- Normalize the time to single execution
- DCUDA reduces the average execution time by up to 42.1%

Impact of Different Loads



Average Execution Time

- Largest performance improvement in medium load case

	Light Load	Medium Load	Heavy Load
Least-Loaded	81201J	74935J	70611J
DCUDA	70449J	70921J	68771J

Energy Consumption

- Largest energy saving in light load case

Outline

1. Background & Problems
2. DCUDA Design
3. Evaluation
4. Conclusion

Conclusion & Future Work

- Static GPU scheduling algorithm in assigning applications leads to load imbalance
  - ✓ Low GPU utilization & high energy consumption
- We develop DCUDA, a dynamic scheduling alg
  - ✓ Monitors resource demand and util. w/ low overhead
  - ✓ Supports migration of running applications
  - ✓ Transparently supports all CUDA applications
- Limitation: DCUDA only considers scheduling within a server and the resource of GPU cores

Thanks!

Q&A

Yongkun Li

ykli@ustc.edu.cn

http://staff.ustc.edu.cn/~ykli