Enabling Fast Preemption via Dual-Kernel Support on GPUs

Li-Wei Shieh[1], **Kun-Chih Chen[2]**, Hsueh-Chun Fu[1], Po-Han Wang[1], and Chia-Lin Yang[1]

[1]National Taiwan University
[2]National Sun Yat-sen University

Email: kcchen@mail.cse.nsysu.edu.tw
Website: https://sites.google.com/site/cereslaben/jimmychen

GPUs in Heterogeneous Computing

- Heterogeneous computing have merged as an efficient way for application acceleration
- GPUs have been widely used in computer systems
  - Data centers, cloud computing, mobile devices, and etc.




QoS Needs in Mobile GPUs

- A fast GPU preemption mechanism is needed to meet QoS for those resource-limited mobile systems

Gaming

3D GUI

Image Editing

Augmented Reality

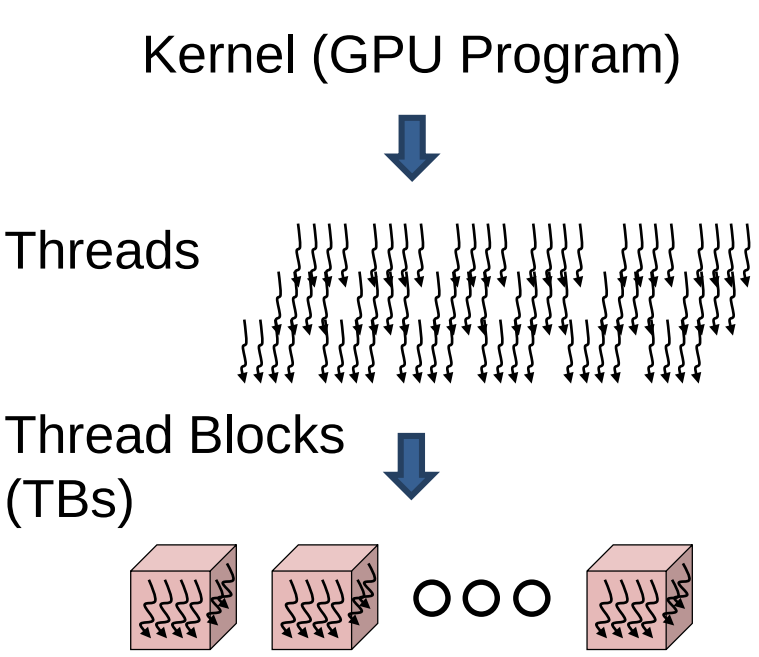
- We can also see these requirements in:

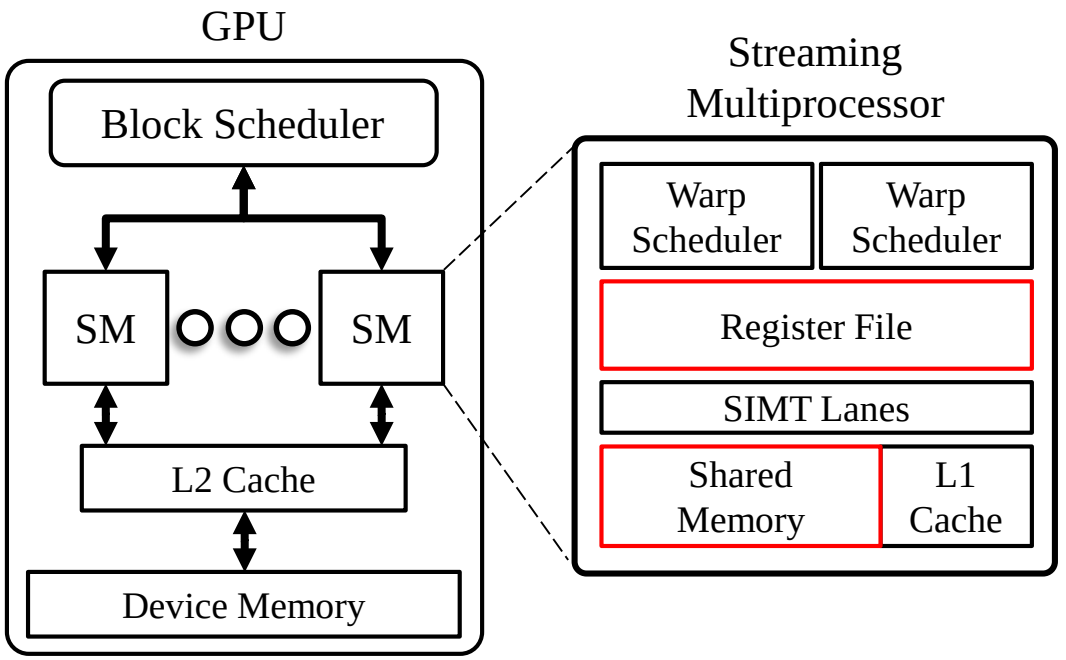
AMD HSA FEATURE ROADMAP

Physical Integration	Optimized Platforms	Architectural Integration	System Integration
Integrate CPU & GPU in silicon	GPU Compute C++ support	Unified Address Space for CPU and GPU	GPU compute context switch
Unified Memory Controller	HSA Memory Management Unit	GPU uses pageable system memory via CPU pointers	GPU graphics pre-emption
Common Manufacturing Technology	Bi-Directional Power Mgmt between CPU and GPU	Fully coherent memory between CPU & GPU	Quality of service

GPU Execution Model

- The resource usage per TB from the same kernel is fixed (registers and shared memory)
  - The allocated resource for a TB must be consecutive



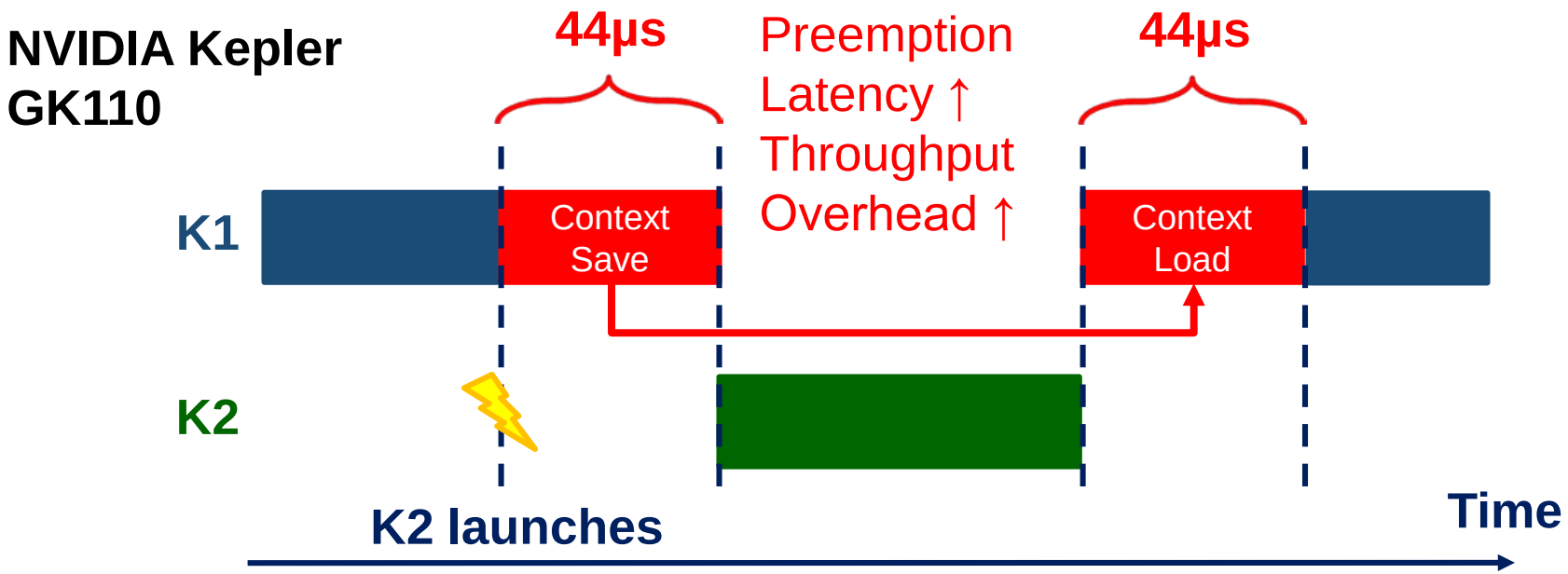


Design Goal of GPU Preemption

- Reduce preemption latency
  - Meet QoS / Satisfy user experience
  - Provide flexible task scheduling
- Reduce throughput overhead
  - Avoid resource utilization degradation during preemption
  - More wasted throughput may cause longer execution time

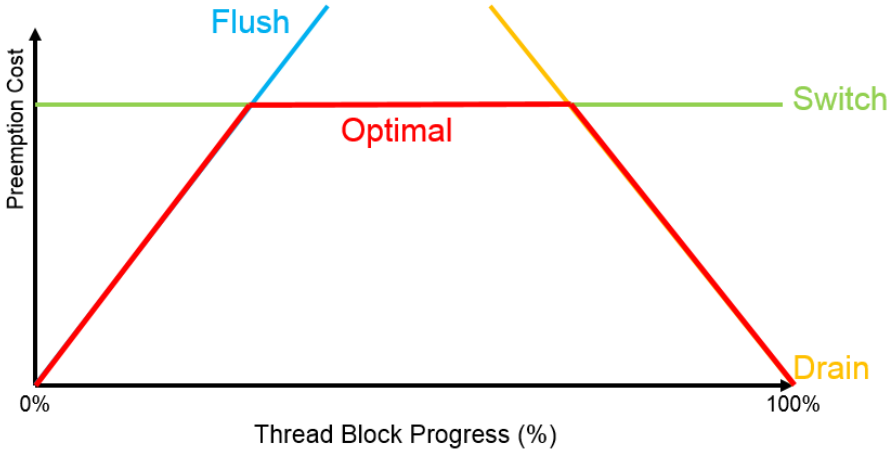
Traditional Context Switching

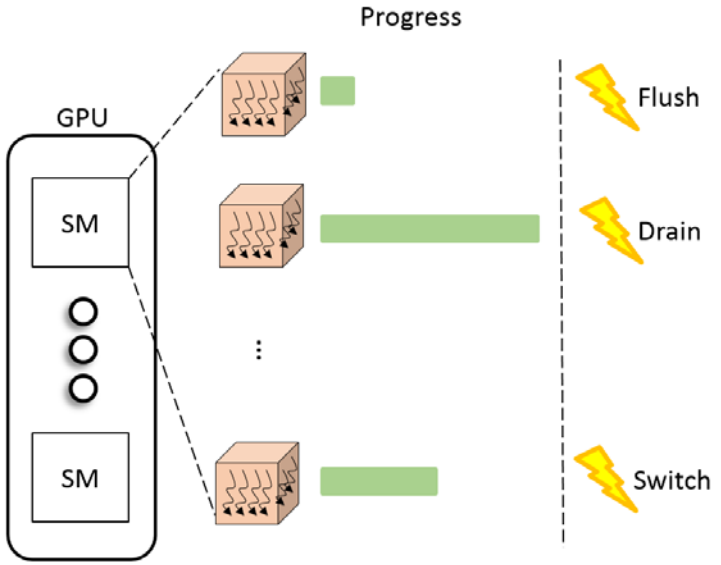
- Modern GPUs can have up to 2048 threads concurrently running on an SM
  - Take up to 44μs preemption latency assuming peak memory bandwidth (< 1μs on modern CPUs)



Related Work: Collaborative Preemption

- Utilize flush and drain collaboratively with context switch to reduce preemption cost [1]
  - Flush: Drop the execution of running TBs
  - Switch: Save/Load the context of running TBs
  - Drain: Wait for running TBs to finish



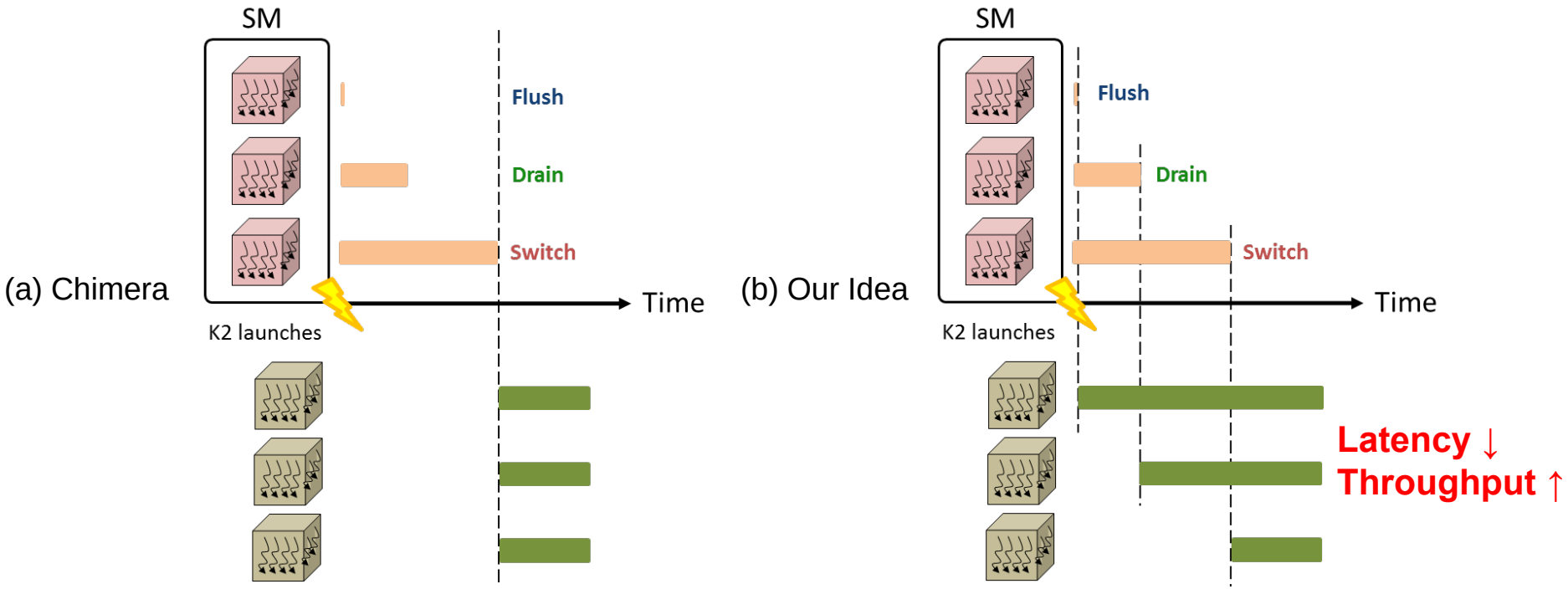


[1] J. J. K. Park *et al.*, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 593-606, Mar. 2015.

Problem Formulation:

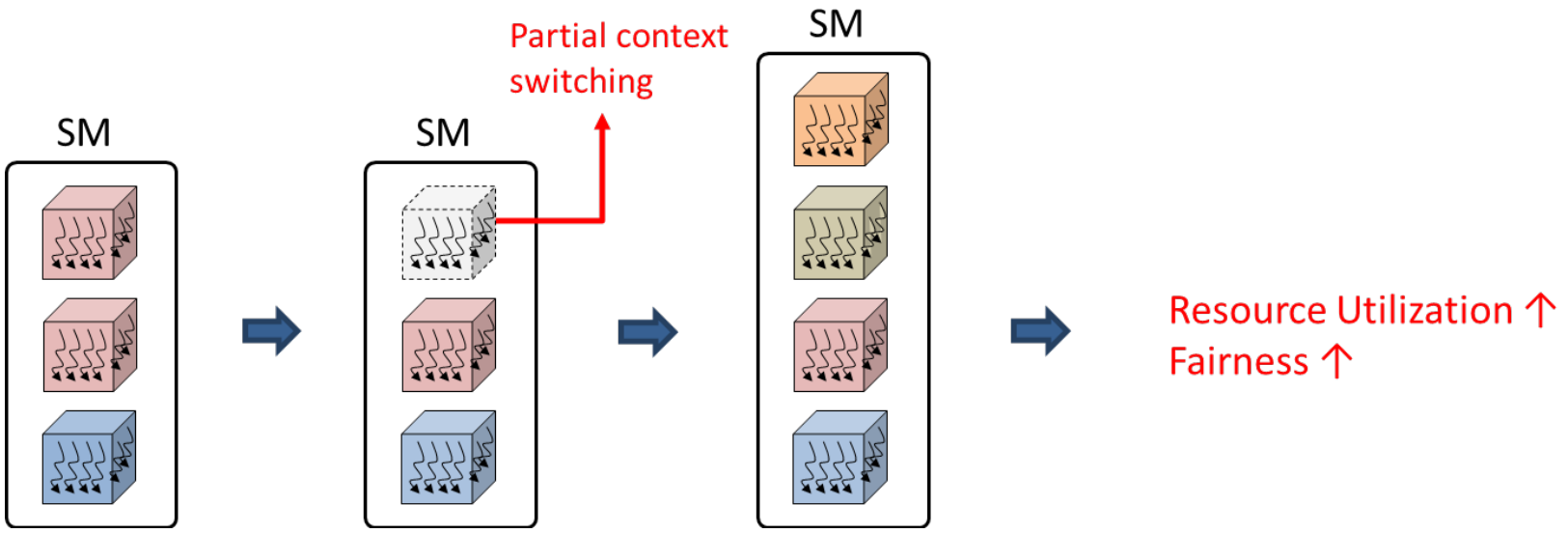
The last leaving TB dominates the preemption latency

- Observation: The preemption granularity of prior works is an entire SM
  - If the preempting kernel can be dispatched once the resources are partially released …



Related Work: Fine-grained Context Switching

- SMK allows multiple kernels to share the resources within an SM [2]
  - Partial context switching is proposed to achieve fairness

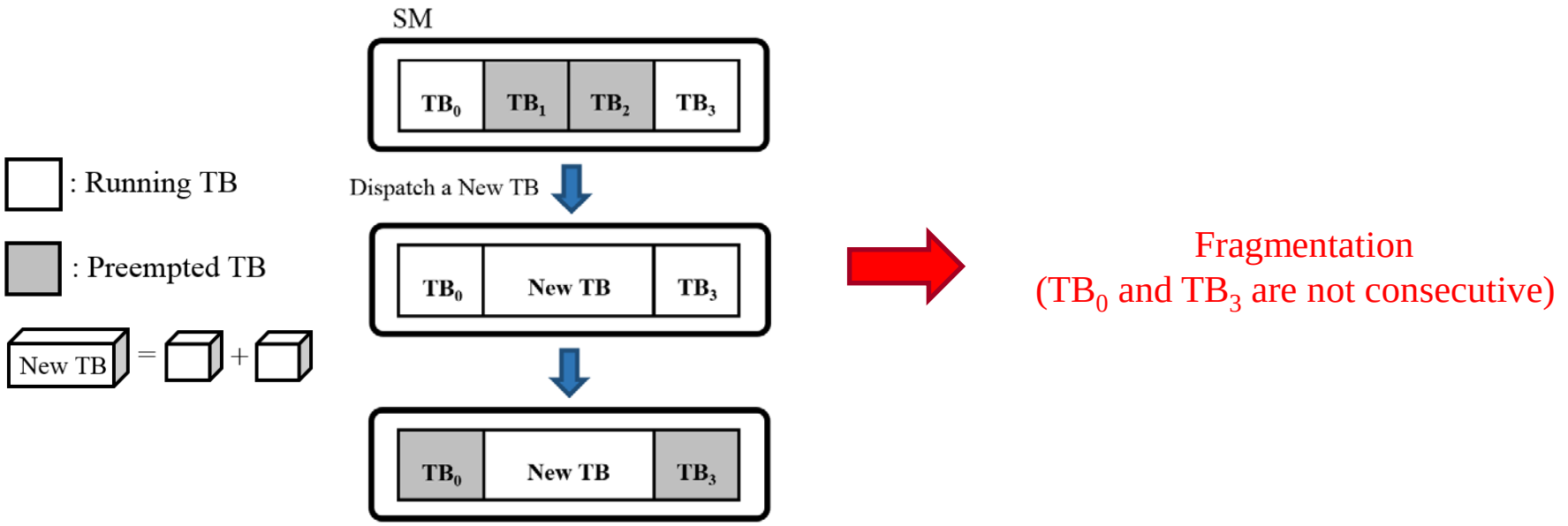


[2] Z. Wang *et al.*, "Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing," IEEE Int. Symp. High-Performance Computer Architecture, Mar. 2016.

Problem Formulation:

The fragmentation problem degrades the resource utilization

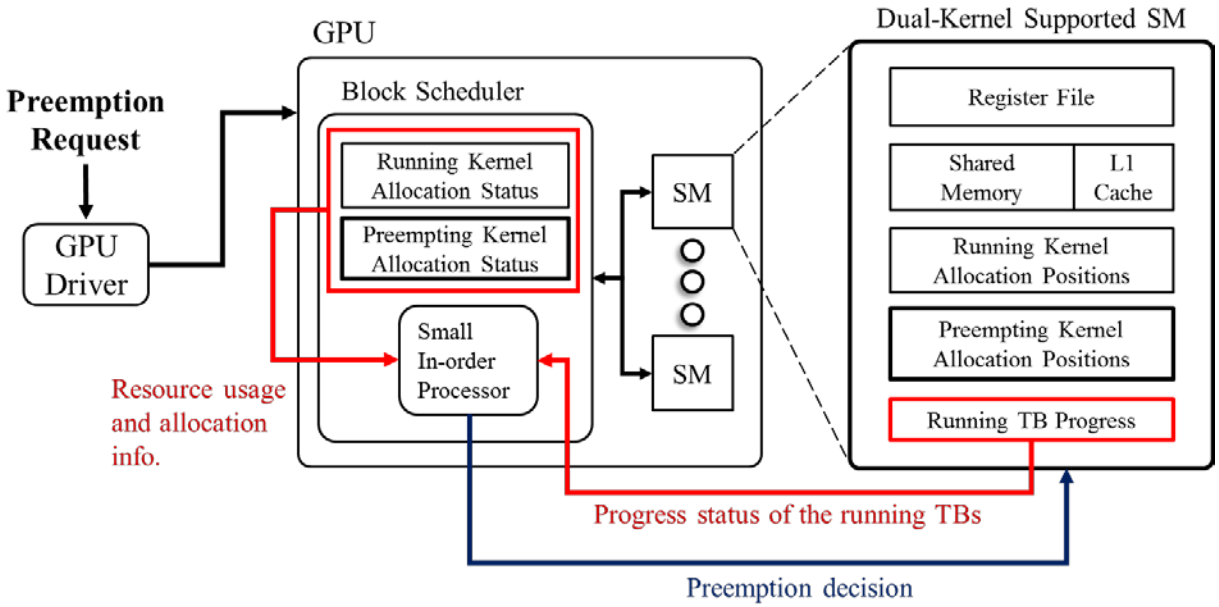
- Naive allocation for the preempting kernel may lead to resource fragmentation within an SM
  - The required resource for a new TB must be consecutive



Fast GPU Preemption Mechanism

Design Goal: We need a lightweight preemption mechanism for the QoS requirement in the mobile system.

- Support Dual-Kernel execution in an SM
  - Normal mode: Execute the task as usual
  - Preemption mode: Make the high-priority task can preempt as soon as possible



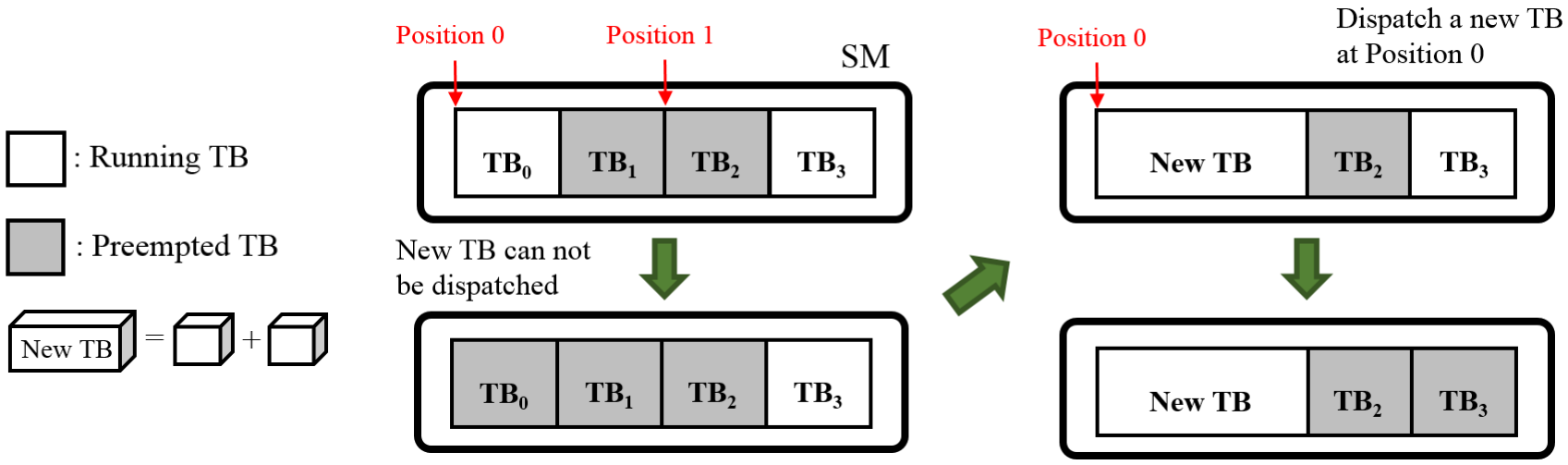
Design Challenge

1. Fragmentation problem avoidance
2. Preempting victim and strategy determination

Fragmentation Problem Avoidance

Resource Allocation Alignment

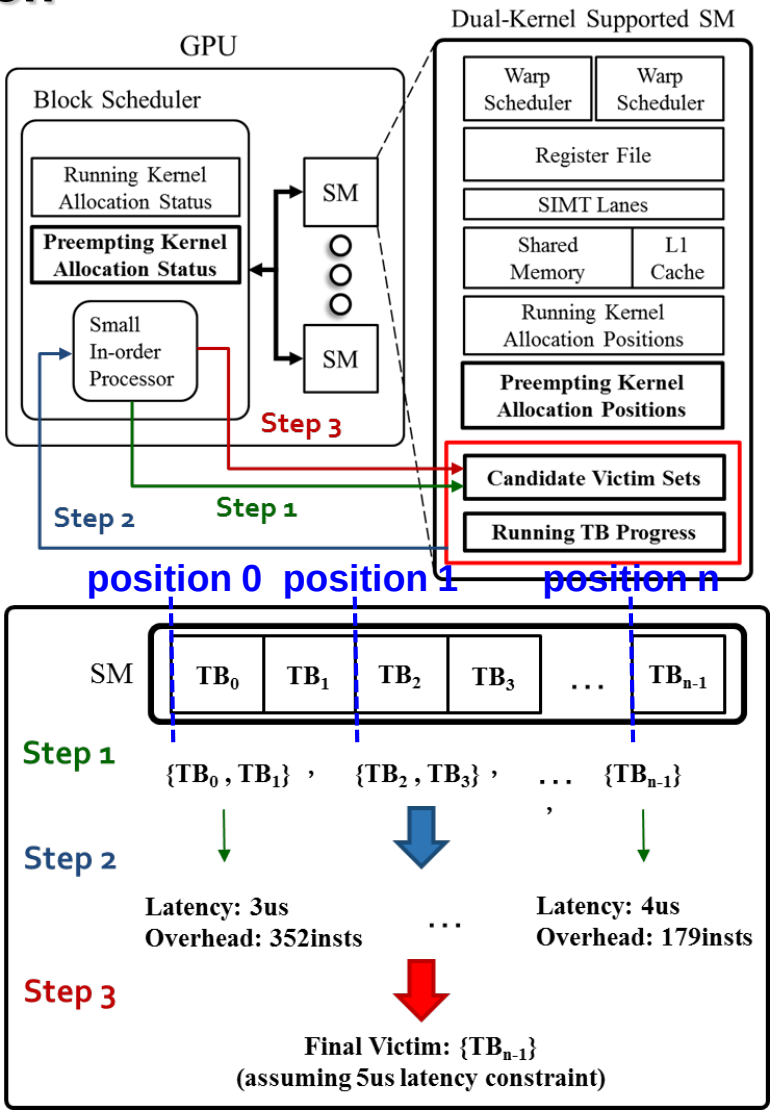
- By restricting the allocation positions, resource within an SM can be allocated consecutively



Preempting Victim/ Strategy Determination

Victim Selection

- Step 1. Candidate Victim Sets Determination
  - A victim set could release the required resource for a new TB while it is preempted
- Step 2. Preemption Cost Estimation
  - Estimate the preemption latency and throughput overhead for each victim set
- Step 3. Identifying the Final Victim
  - Selection Criteria
    - Least throughput overhead while meeting the preemption latency constraint



Preemption Cost Estimation

- Estimate the preemption cost for each candidate with Switch/Drain/Flush
  - The cost of a candidate can be derived from the TBs in the candidate
    - The preemption latency is bounded by the last leaving TB

$$P\_Latency(Candidatek) = \underset{i=0...n-1}{Max}(P\_Latency(TBi))$$

    - The throughput overhead is the summation of the overhead of each TB
      - Throughput overhead is defined as the total wasting instructions during the preemption

$$T\_Overhead(Candidatek) = \sum_{i=0}^{n-1}(T\_Overhead(TBi))$$

Estimation: Preemption Latency

- Switch
  - (Context size) / (Mem. BW / # of SMs)
- Flush
  - Zero preemption latency
- Drain
  - (Remaining instructions) x (Average CPI)

- Draining latency estimation is difficult due to CPI variation across time
  - Intra-block Variation
    - CPIs may vary across time for the same TB because of indirect memory accesses or branch divergences
  - Inter-block Variation
    - CPIs can also be data dependent, as some TBs access the data regions with better locality than the others
- In order to predict the future CPIs more accurately, we choose the way that has lower variation

Estimation: Throughput Overhead

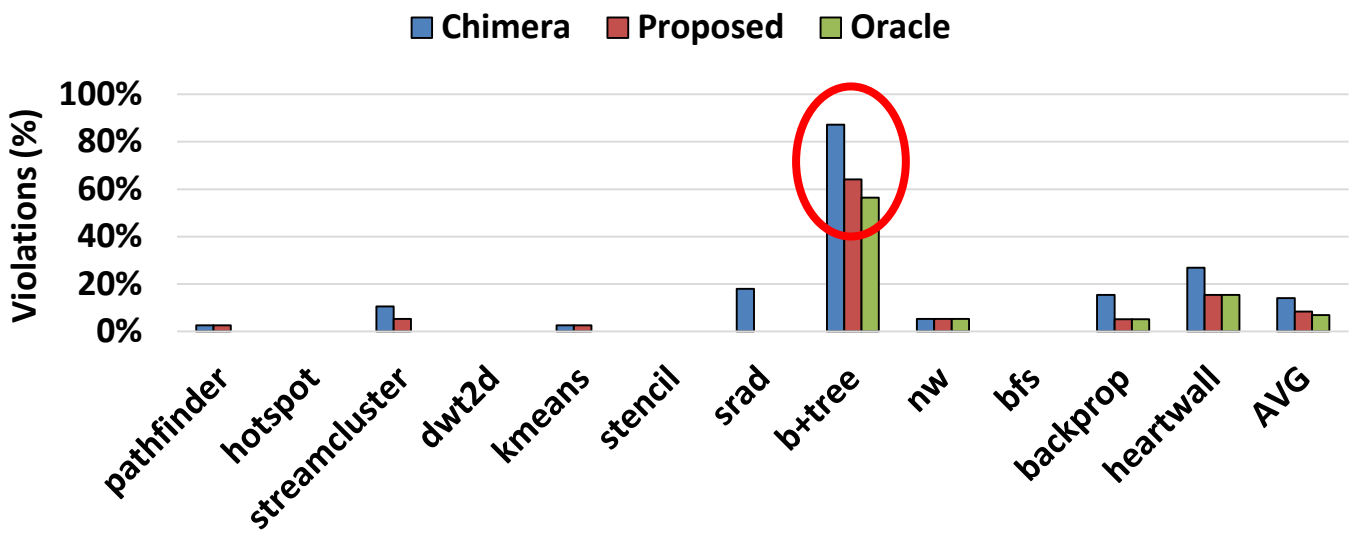
- Switch
  - Average IPC x Preemption latency x 2
  - The switching overhead is doubled due to both saving/loading the context
- Drain
  - Zero throughput overhead due to dual-kernel support
- Flush
  - The number of executed instructions

Experimental Setup

- GPGPU-Sim v3.2.2
  - GPU Model: NVIDIA Fermi GTX 480
  - 128kB registers and 48kB shared memory per SM
- Workloads
  - 12 benchmarks from Rodinia and Parboil with different resource usage and idempotence (restriction of flushing)
  - GPGPU benchmark + Synthetic benchmark
    - Mimics high-priority tasks with deadline
    - Deadline = Preemption latency constraint + Execution time

Results: Deadline Violation

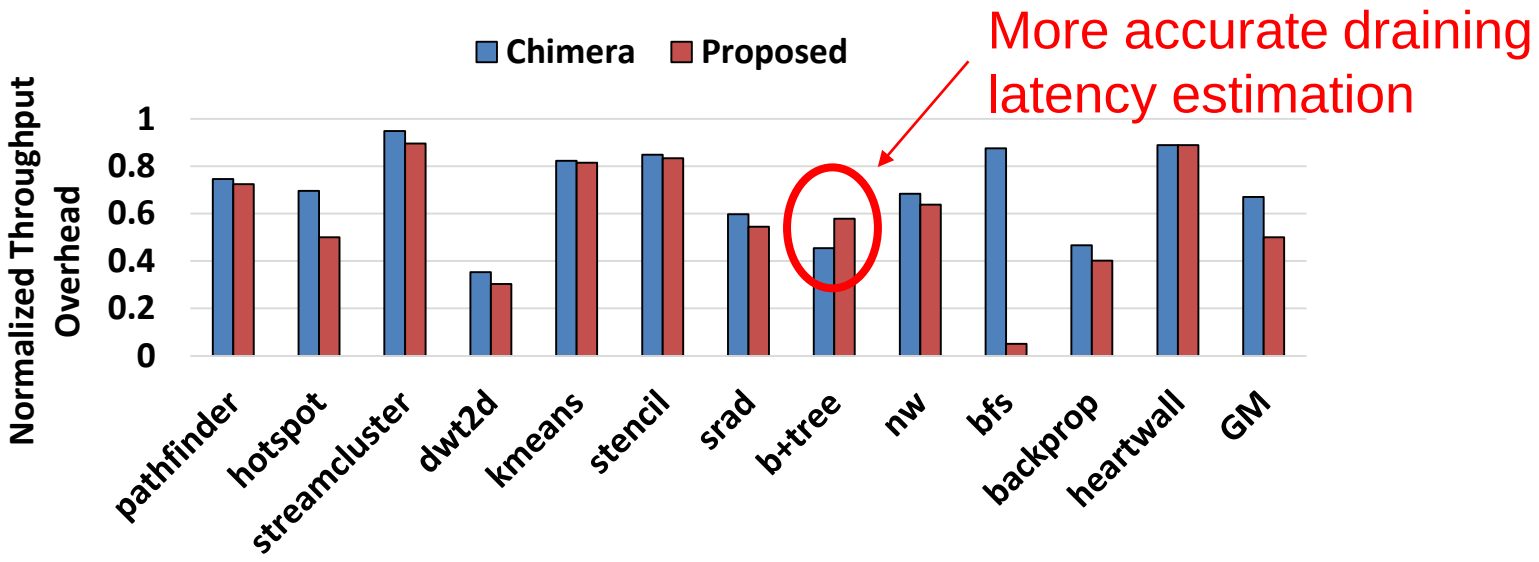
- On average, Chimera misses deadline for 14.0%, while the proposed scheme is 8.4% (Oracle = 6.9%)
  - *b+tree* shows highest violation rate due to the restriction of flushing and many indirect memory accesses



Violations among multiple preemption requests under 2us latency constraint

Results: Throughput Overhead

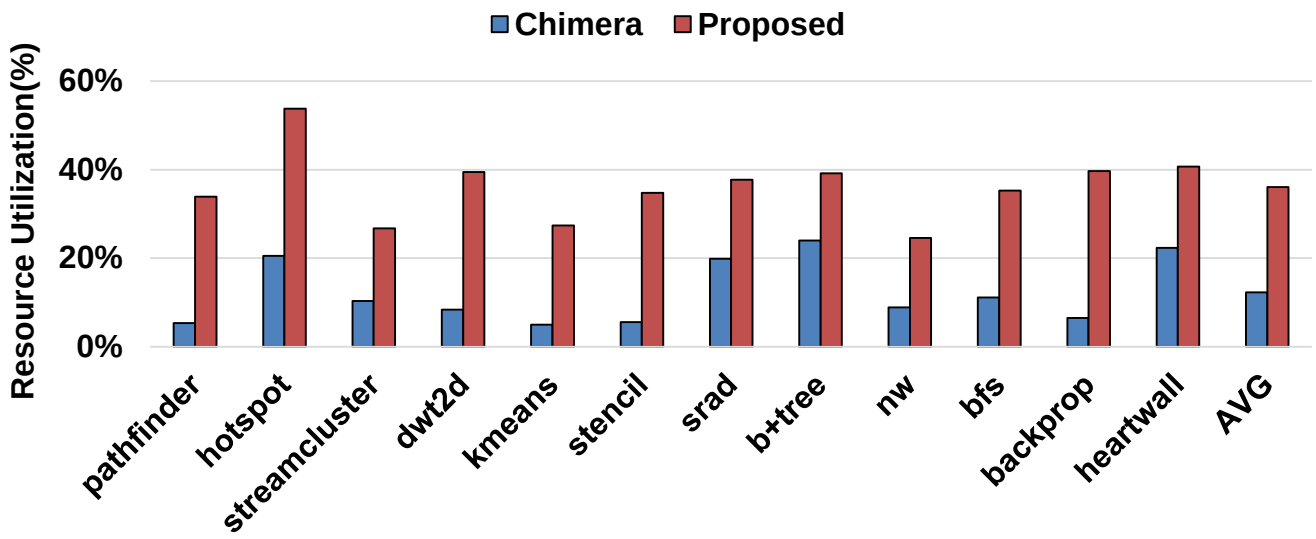
- We define the throughput overhead as the wasted instructions, the results are normalized to Flushing
  - The incurred overhead of our scheme is no more than Chimera under most of the cases



Normalized throughput overhead under 2us latency constraint

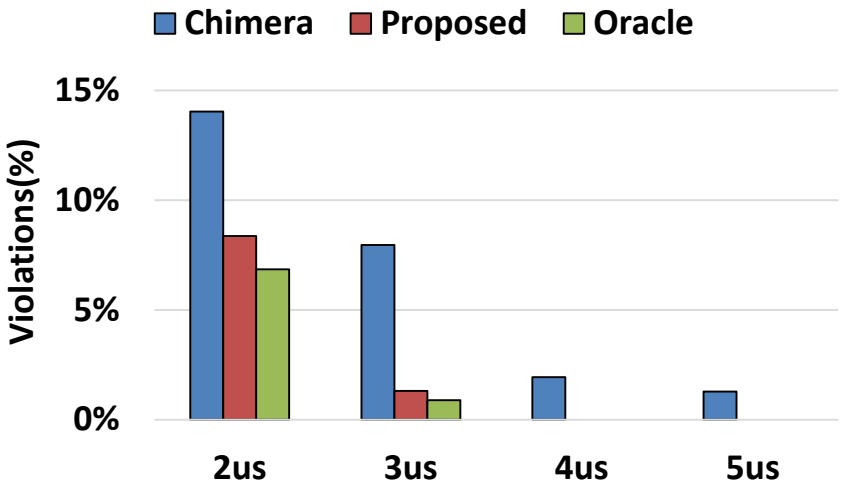
Results: Resource Utilization

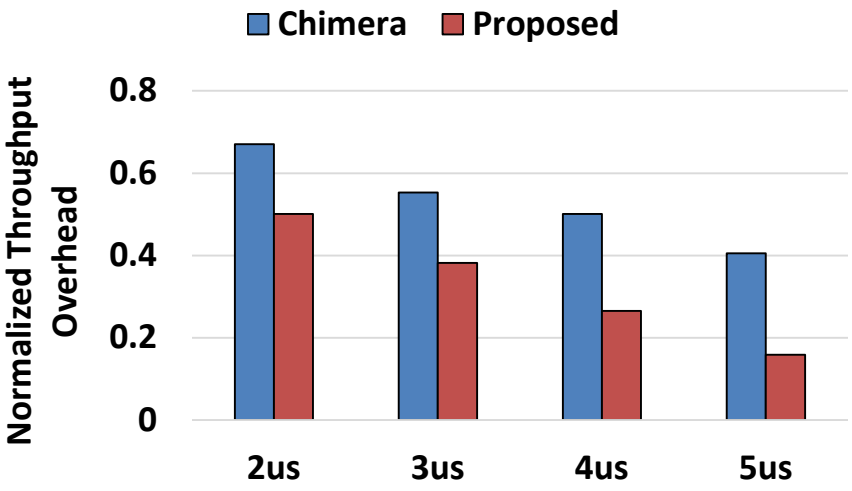
- On average, we improve GPU resource utilization by 2.93x over Chimera during preemption



Impact of Preemption Latency Constraint

- We benefits more when the preemption latency constraint increase due to the increasing slack time
  - Violations: within 2% difference compared with Oracle
  - Throughput Overhead: overall 38.6% improvement





Conclusion

- We propose a simple dual-kernel SM design to support fine-grained preemption and a resource allocation policy to avoid fragmentation
- The proposed victim selection scheme is able to make proper preemption decisions
  - Achieving very low deadline violations while avoiding significant throughput degradation effectively