Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling

Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, Soojung Ryu

HPCA 2014
Outline

• Motivation
• Background
• Lazy CTA Scheduling
• Block CTA Scheduling
• Evaluation
• Conclusion
Motivation

• A collection of threads are grouped to form a warp/wavefront, and the warps are combined to create a CTA (cooperative thread array)/thread block.

• As a result, there are two levels of schedulers within a GPGPU:
  – Thread block/CTA scheduler: assign CTAs to cores
  – Warp/wavefront scheduler: determine which warp is executed

• There has been work on different warp schedulers: cache-conscious wavefront scheduling, two-level warp scheduling, etc., but few work on CTA scheduling.
Motivation

• Performance of different workloads as the maximum number of CTAs allocated to each core is varied

• For some workloads (e.g., EIP, KMN, BFS), the performance actually **degrades** as the number of CTAs assigned to a core continues to increase, which may create resource contention.

• This paper proposes an approach of considering both warp scheduler and block scheduler to improve the efficiency in GPGPU architecture.
Background

• Baseline GPU CTA scheduling
  – CTAs are assigned to each SM (stream multiprocessor)/core in a round-robin manner; assign the maximum number of CTAs to each core
  – The maximum number of CTAs assigned to each core depends on the resource usage of the workload (the amount of registers, shared memory, etc.)
  – Once a particular CTA finishes, the CTA scheduler assigns another CTA to that particular SM, until all CTAs have been assigned to the cores
Workload characteristic

- **Type I**: Increased Performance
- **Type II**: Increased Performance and Saturate
- **Type III**: Decreased Performance
- **Type IV**: Increase then Decrease
Lazy CTA Scheduling for type-II

- **Goal**: reduce the number of thread blocks allocated to each core dynamically and maximize performance
- **RR vs GTO (greedy-then-oldest) warp scheduler**

  - **GTO** prioritizes a single warp until it stalls, and then selects the oldest warp. As a result, GTO ends up prioritizing a single thread block until all warps in the given thread block are stalled – and then, selects a warp from the oldest thread block.
  - First thread block finishes: **RR** – each thread block will likely have issued a similar number of instructions; **GTO** – the number of instructions executed for each thread block will differ.
  - For GTO, 3 CTAs are sufficient for this workload.
  - Type-II: performance saturates with additional thread blocks
Lazy CTA Scheduling for type-III/IV

- RR vs GTO (greedy-then-oldest) warp scheduler

- Assume: **3 MSHR entry/core**: since all CTAs initially generates a memory access, the fourth CTA cannot issue its memory instruction.

- Assume: CTA1 and CTA3 memory access creates an access to the same cache entry and results in a conflict miss

- GTO: three thread blocks are sufficient to keep the core busy; the reduced number of thread blocks avoid the cache eviction that occur between CTA1 and CTA3 and avoid performance degradation
Three phases of Lazy CTA Scheduling

• **Phase 1: Monitor**
  – $T_{\text{max}}$ thread blocks are initially allocated to each core
  – the number of instructions issued (inst) for each thread block $x$ is measured until the first thread block finishes execution

• **Phase 2: Throttle**
  – total num
    \[
    T_{\text{new}} = \left\lfloor \frac{\sum_{x=0}^{T_{\text{max}}} \text{(inst}_x)/\text{max(inst}_x)}{\text{inst}_x} \right\rfloor 
    \]
  – across all the thread blocks in the core / the number of instructions issued from the first thread block that completed
  – e.g. in figure(b): $T_{\text{new}} = \lfloor 10/4 \rfloor = 3$

• **Phase 3: Lazy Execution**
  – Only $T_{\text{new}}$ thread blocks allocated to each core after Phase 2 completes

• The algorithm is repeated for **each kernel** within each workload since the behavior of each kernel can differ.
Block CTA Scheduling

• Many workloads (kernels) in GPGPU workloads are organized as a 2D array of CTAs; common CTA size: $16 \times 16$ (256 threads);

• Inter-CTA locality can exist among sequential CTAs:
  Data accessed by each thread – a single word (4Bytes);
  Each row of data from a CTA – $16 \times 4 = 64$ Bytes;
  The line size of L1 cache – 128Bytes
  
  spatial locality exists between neighboring CTAs

• With RR block scheduling, the inter-CTA spatial locality is lost since sequential CTAs are not assigned to same core
Block CTA Scheduling

- Assigns a block (e.g. 2) of sequential thread blocks or CTAs to the same core; exploit spatial locality across the same cache line within the local L1
- **Delayed scheduling/assignment** of thread blocks: a new thread block is not allocated to a core until pair of sequential thread blocks finish execution
- **Sequential CTA-aware (SCA) warp scheduling** to effectively exploit the inter-CTA locality with BCS
  - Warps are scheduled in a round-robin manner between two warps of neighboring thread blocks or within a block
  - The warp scheduler remains greedy as these set of warps are prioritized, until one of the two warps stall.
  - Then, the next group of warps within the same block is scheduled.
- BCS is not applicable to workloads with one-dimensional CTAs as there is little inter-CTA L1 locality.
Evaluation

- **Lazy CTA Scheduling Results**

  - Baseline: greedy-then-oldest warp scheduler; RR CTA scheduler
  - 7% improvement on average; 23% improvement for Type-III/IV

For some workloads, the number of optimal thread blocks determined by LCS was smaller than OPT value. Modification: $[T_{new}] \rightarrow [T_{new}]$
Evaluation

- **Block CTA Scheduling** Results (for only 2D workloads)

  ![](chart1.png)

  - On average, 3% for BCS + GTO; 15% for BCS + SCA

  ![](chart2.png)

  - On average, L1 miss rate reduced by 8% for BCS + GTO; 24% for BCS + SCA
  - **Delayed thread block assignment** (in order to assign consecutive CTAs to the same core) may negate the benefit from reduced miss rate.
Conclusion

- **LCS (lazy CTA scheduling):** leverage a greedy warp scheduler to determine the optimal number of thread blocks per core
- **BCS (block CTA scheduling):** exploit inter-CTA locality to improve overall performance
  - **alternative warp scheduler:** aware of the consecutive thread blocks allocated to the same core and exploit the inter-CTA locality
Backup
mixed Concurrent Kernel Execution (mCKE)

- Allocate less than the maximum number of thread blocks to each core to underutilized resources within a core opportunity for concurrent execution of different kernels on the same core.

- In the baseline CKE, each core can be stalled at different time point.
- With mCKE, the memory latency can be hidden (or overlapped) with other kernel execution.
Evaluation

• **Mixed Concurrent Kernel Execution Results**

<table>
<thead>
<tr>
<th>Mixed workloads</th>
<th>Type</th>
<th># of CTAs per kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP-LPS</td>
<td>II-IV</td>
<td>3, 5</td>
</tr>
<tr>
<td>MMUL-VADD</td>
<td>I-II</td>
<td>2, 3</td>
</tr>
<tr>
<td>MUM-BLK</td>
<td>III-IV</td>
<td>2, 4</td>
</tr>
<tr>
<td>MMUL-RAY</td>
<td>I-II</td>
<td>2, 2</td>
</tr>
<tr>
<td>CP-MADD</td>
<td>II-II</td>
<td>3, 4</td>
</tr>
<tr>
<td>KMN-SRAD</td>
<td>III-II</td>
<td>1, 5</td>
</tr>
</tbody>
</table>

- Merge different workloads
- The number of CTAs allocated to each core is based on the optimal number of thread blocks from LCS
- For the baseline CKE, the LCS was used such that the optimal number of thread blocks were allocated.
- For a given workload, the benefit of mCKE depends on which workload it is mixed with