Demystifying the NVIDIA GPU Thread Block Scheduler

Guin Gilman
Samuel S. Ogden
Tian Guo
Robert J. Walls

WPI Cake Lab website
Concurrent Workloads and the Scheduler

• Concurrent kernel execution for higher GPU resource utilization
• Thread block scheduler partitions computational resources
  • Shared memory
  • Threads
  • Registers
• Difficult to measure performance
  • NVIDIA devices are black-box
  • We rely on empirical observations of the scheduler
Key Findings

• The scheduler uses a **most-room policy** for placing blocks on SMs
• Counter-intuitive **performance degradation** for concurrent workloads
• **Predictability is challenging** due to external factors
  • Block placement
  • Resource contention
  • Launch order
The CUDA Programming Model
Policies of the Thread Block Scheduler

• **When** are blocks scheduled?
• **Which** block does the scheduler choose?
• **Where** will that block be placed?
The Leftover Policy

• The leftover policy: determining when and which block
  • As soon as space is available on some SM
  • Only thread blocks from the earliest launched kernel
The Leftover Policy

• The *leftover* policy: determining *when* and *which* block
  • As soon as space is available on some SM
  • Only thread blocks from the *earliest launched* kernel
The Leftover Policy

- The leftover policy: determining when and which block
  - As soon as space is available on some SM
  - Only thread blocks from the earliest launched kernel
The Leftover Policy

- The **leftover** policy: determining **when** and **which** block
  - As soon as space is available on some SM
  - Only thread blocks from the **earliest launched** kernel

![Execution Queue Diagram]

- SM0
- SM1
- SM2
- SM3
- SM4
The Leftover Policy

- The *leftover* policy: determining *when* and *which* block
  - As soon as space is available on some SM
  - Only thread blocks from the *earliest launched* kernel
The Leftover Policy

- The leftover policy: determining *when* and *which* block
  - As soon as space is available on some SM
  - Only thread blocks from the *earliest launched* kernel
The Leftover Policy

• The leftover policy: determining when and which block
  • As soon as space is available on some SM
  • Only thread blocks from the earliest launched kernel
The Leftover Policy

- The leftover policy: determining **when** and **which** block
  - As soon as space is available on some SM
  - Only thread blocks from the **earliest launched** kernel
The Leftover Policy

• The leftover policy: determining when and which block
  • As soon as space is available on some SM
  • Only thread blocks from the earliest launched kernel
The Leftover Policy

• The **leftover** policy: determining **when** and **which** block
  • As soon as space is available on some SM
  • Only thread blocks from the **earliest launched** kernel
The Leftover Policy

- The leftover policy: determining **when** and **which** block
  - As soon as space is available on some SM
  - Only thread blocks from the **earliest launched** kernel
The Most-Room Policy

- The **most-room** policy: determining **where**
  - The selected block will be placed on the SM with the most room for blocks of the current kernel
  - Based on each SM's current resource availability
- The first resource to run out becomes the **limiting resource**
  - Computational resources, i.e. shared memory, threads, registers
  - Hardware limits, i.e. max blocks per SM, max warps per SM
- Ties are broken using a set tie-breaking ordering
The Most-Room Policy

• The **most-room** policy: determining where
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability

![Diagram showing SM1 and SM0 over time]
The Most-Room Policy

• The **most-room** policy: determining **where**
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The *most-room* policy: determining *where*
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The **most-room** policy: determining **where**
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The **most-room** policy: determining *where*
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The **most-room** policy: determining *where*
  
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The **most-room** policy: determining *where*
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The **most-room** policy: determining **where**
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

• The **most-room** policy: determining *where*
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability

![Diagram showing SM0 and SM1 with blocks A0, A1, and B0]
The Most-Room Policy

- The **most-room** policy: determining *where*
  - The selected block will be placed on the SM with the most room for blocks of the current kernel
  - Based on each SM's current resource availability

![Diagram showing SM0 and SM1 with blocks A0, A1, B0, B1 placement over time A and B]
The Most-Room Policy

• The **most-room** policy: determining *where*
  • The selected block will be placed on the SM with the most room for blocks of the current kernel
  • Based on each SM's current resource availability
The Most-Room Policy

- The **most-room** policy: determining **where**
  - The selected block will be placed on the SM with the most room for blocks of the current kernel
  - Based on each SM's current resource availability
Performance Implications
Kernel Classes

• **L1-cache-dependent**
  • Performance depends primarily on the amount of L1 cache contention

• **Compute-intensive**
  • Performance is bounded by the number of computations that can be performed per unit of time

• **Memory-intensive**
  • Performance is dependent on global memory throughput

• **PCIe-transfer-dependent**
  • On discrete GPUs, performance depends on the speed at which page faults can be handled by the GPU
## Results (Turing GPU)

<table>
<thead>
<tr>
<th></th>
<th>Serial (ms)</th>
<th>Concurrent-Isolated (ms)</th>
<th>Concurrent-Colocated (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel X</td>
<td>Kernel Y</td>
<td>Kernel X</td>
</tr>
<tr>
<td>L1-Cache-Dependent</td>
<td>85</td>
<td>79</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>164</td>
<td>85</td>
<td>105 (1.24X)</td>
</tr>
<tr>
<td>Compute-Intensive</td>
<td>523</td>
<td>365</td>
<td>527</td>
</tr>
<tr>
<td></td>
<td>888</td>
<td>529 (1.45X)</td>
<td>530</td>
</tr>
<tr>
<td>Memory-Intensive</td>
<td>949</td>
<td>10</td>
<td>951</td>
</tr>
<tr>
<td></td>
<td>959</td>
<td>224 (22.4X)</td>
<td>955</td>
</tr>
<tr>
<td>PCIe-Bandwidth-Dep.</td>
<td>369</td>
<td>130</td>
<td>385 (1.04X)</td>
</tr>
<tr>
<td></td>
<td>499</td>
<td>355 (2.73X)</td>
<td>466 (3.58X)</td>
</tr>
</tbody>
</table>
## Results (Turing GPU)

<table>
<thead>
<tr>
<th></th>
<th>Serial (ms)</th>
<th>Concurrent-Isolated (ms)</th>
<th>Concurrent-Colocated (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel X</td>
<td>Kernel Y</td>
<td>Total</td>
</tr>
<tr>
<td>L1-Cache-Dependent</td>
<td>85</td>
<td>79</td>
<td>164</td>
</tr>
<tr>
<td>Compute-Intensive</td>
<td>523</td>
<td>365</td>
<td>888</td>
</tr>
<tr>
<td>Memory-Intensive</td>
<td>949</td>
<td>10</td>
<td>959</td>
</tr>
<tr>
<td>PCIe-Bandwidth-Dep.</td>
<td>369</td>
<td>130</td>
<td>499</td>
</tr>
</tbody>
</table>
## Results (Turing GPU)

<table>
<thead>
<tr>
<th></th>
<th>Serial (ms)</th>
<th>Concurrent-Isolated (ms)</th>
<th>Concurrent-Colocated (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel X</td>
<td>Kernel Y</td>
<td>Total</td>
</tr>
<tr>
<td>L1-Cache-Dependent</td>
<td>85</td>
<td>79</td>
<td>164</td>
</tr>
<tr>
<td>Compute-Intensive</td>
<td>523</td>
<td>365</td>
<td>888</td>
</tr>
<tr>
<td>Memory-Intensive</td>
<td>949</td>
<td>10</td>
<td>959</td>
</tr>
<tr>
<td>PCIe-Bandwidth-Dep.</td>
<td>369</td>
<td>130</td>
<td>499</td>
</tr>
</tbody>
</table>
## Results (Turing GPU)

<table>
<thead>
<tr>
<th></th>
<th>Serial (ms)</th>
<th>Concurrent-Isolated (ms)</th>
<th>Concurrent-Colocated (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel X</td>
<td>Kernel Y</td>
<td>Total</td>
</tr>
<tr>
<td>L1-Cache-Dependent</td>
<td>85</td>
<td>79</td>
<td><strong>164</strong></td>
</tr>
<tr>
<td>Compute-Intensive</td>
<td>523</td>
<td>365</td>
<td><strong>888</strong></td>
</tr>
<tr>
<td>Memory-Intensive</td>
<td>949</td>
<td>10</td>
<td><strong>959</strong></td>
</tr>
<tr>
<td>PCIe-Bandwidth-Dep.</td>
<td>369</td>
<td>130</td>
<td><strong>499</strong></td>
</tr>
</tbody>
</table>
## Results (Turing GPU)

<table>
<thead>
<tr>
<th></th>
<th>Serial (ms)</th>
<th>Concurrent-Isolated (ms)</th>
<th>Concurrent-Colocated (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel X</td>
<td>Kernel Y</td>
<td>Kernel X</td>
</tr>
<tr>
<td><strong>L1-Cache-Dependent</strong></td>
<td>85</td>
<td>79</td>
<td>164</td>
</tr>
<tr>
<td><strong>Compute-Intensive</strong></td>
<td>523</td>
<td>365</td>
<td>888</td>
</tr>
<tr>
<td><strong>Memory-Intensive</strong></td>
<td>949</td>
<td>10</td>
<td>959</td>
</tr>
<tr>
<td><strong>PCIe-Bandwidth-Dep.</strong></td>
<td>369</td>
<td>130</td>
<td>499</td>
</tr>
</tbody>
</table>
### Results (Turing GPU)

<table>
<thead>
<tr>
<th></th>
<th>Serial (ms)</th>
<th>Concurrent-Isolated (ms)</th>
<th>Concurrent-Colocated (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel X</td>
<td>Kernel Y</td>
<td>Total</td>
</tr>
<tr>
<td>L1-Cache-Dependent</td>
<td>85</td>
<td>79</td>
<td>164</td>
</tr>
<tr>
<td>Compute-Intensive</td>
<td>523</td>
<td>365</td>
<td>888</td>
</tr>
<tr>
<td>Memory-Intensive</td>
<td>949</td>
<td>10</td>
<td>959</td>
</tr>
<tr>
<td>PCIe-Bandwidth-Dep.</td>
<td>369</td>
<td>130</td>
<td>499</td>
</tr>
</tbody>
</table>
Conclusions

• The scheduler uses a **most-room policy** for placing blocks on SMs
  • For choosing which SM to schedule the next thread block to
  • Chooses the SM which can fit the highest number of blocks from the kernel
  • Not a round-robin policy as previously believed

• Counter-intuitive **performance degradation** for concurrent workloads
  • Depends on the kernel type
  • Is influenced by resource contention & SM placement

• **Predictability is challenging** due to external factors
  • Block placement
  • Resource contention
  • Launch order