Demystifying the NVIDIA GPU Thread Block Scheduler

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

WPI

POLITECNICO
DI MILANO

Performance 2020
November 2-6

WPI Cake Lab website

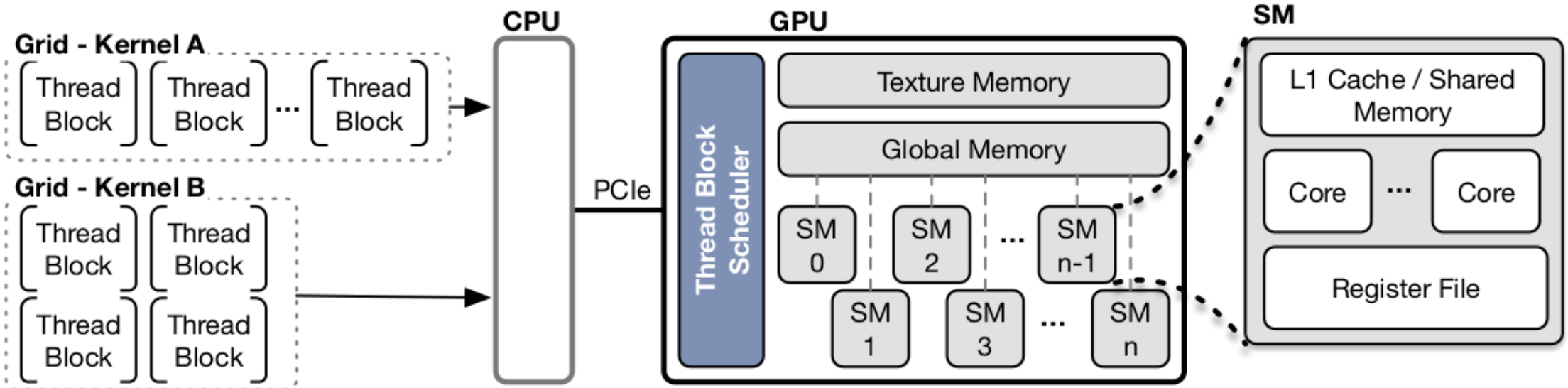
Concurrent Workloads and the Scheduler

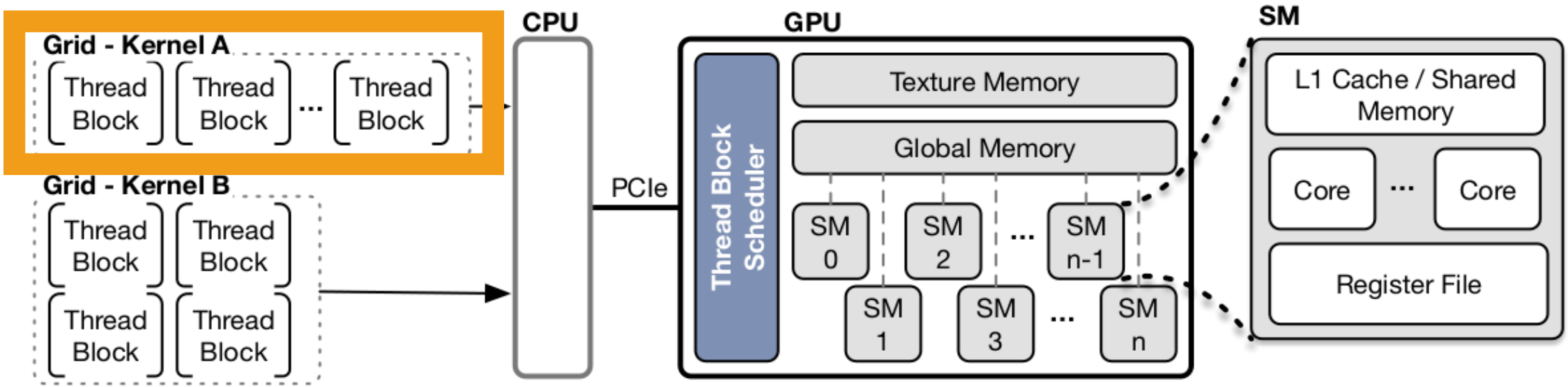
- Concurrent kernel execution for higher GPU **resource utilization**
- Thread block scheduler partitions computational resources
  - Shared memory
  - Threads
  - Registers
- Difficult to measure performance
  - NVIDIA devices are black-box
  - We rely on empirical observations of the scheduler

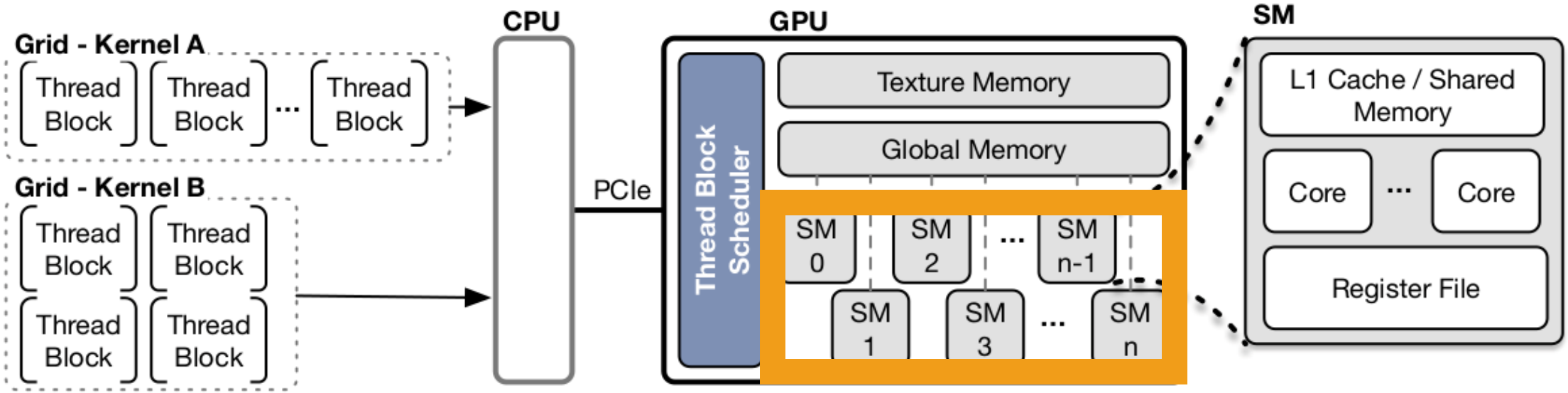
Key Findings

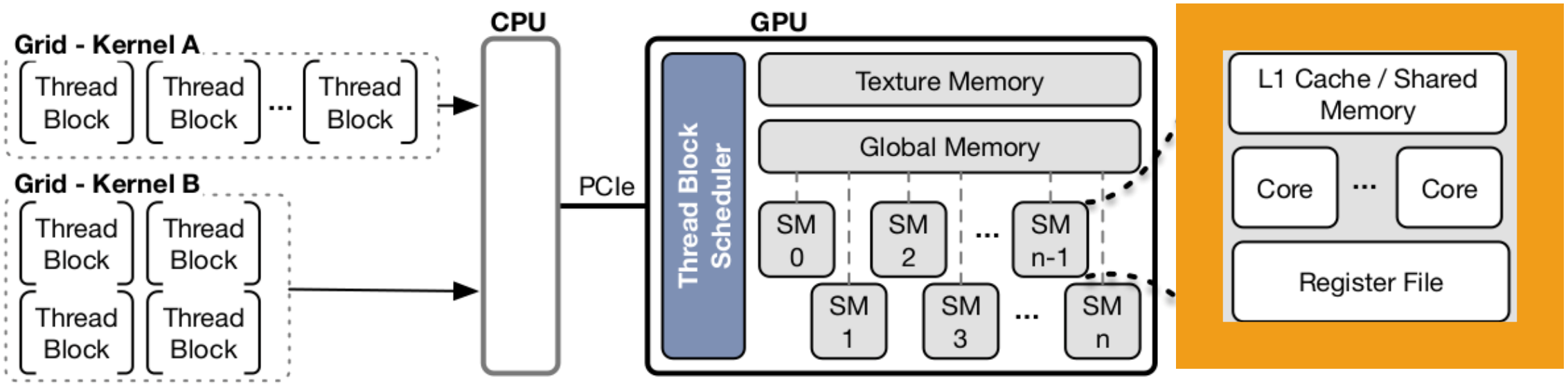
- The scheduler uses a **most-room policy** for placing blocks on SMs
- Counter-intuitive **performance degradation** for concurrent workloads
- **Predictability is challenging** due to external factors
  - Block placement
  - Resource contention
  - Launch order

The CUDA Programming Model


Grid - Kernel A
Thread Block
Thread Block
...
Thread Block
Grid - Kernel B
Thread Block
Thread Block
Thread Block
Thread Block
CPU
PCIe
GPU
Thread Block Scheduler
Texture Memory
Global Memory
SM 0
SM 2
...
SM n-1
SM 1
SM 3
...
SM n
SM
L1 Cache / Shared Memory
Core
...
Core
Register File


Grid - Kernel A
Thread Block
Thread Block
...
Thread Block
Grid - Kernel B
Thread Block
Thread Block
Thread Block
Thread Block
CPU
PCIe
GPU
Thread Block Scheduler
Texture Memory
Global Memory
SM 0
SM 2
...
SM n-1
SM 1
SM 3
...
SM n
SM
L1 Cache / Shared Memory
Core
...
Core
Register File


Grid - Kernel A
Thread Block
Thread Block
...
Thread Block
Grid - Kernel B
Thread Block
Thread Block
Thread Block
Thread Block
CPU
PCIe
GPU
Thread Block Scheduler
Texture Memory
Global Memory
SM 0
SM 2
...
SM n-1
SM 1
SM 3
...
SM n
SM
L1 Cache / Shared Memory
Core
...
Core
Register File


Grid - Kernel A
Thread Block
Thread Block
...
Thread Block
Grid - Kernel B
Thread Block
Thread Block
Thread Block
Thread Block
CPU
PCIe
GPU
Thread Block Scheduler
Texture Memory
Global Memory
SM 0
SM 2
...
SM n-1
SM 1
SM 3
...
SM n
L1 Cache / Shared Memory
Core
...
Core
Register File

Policies of the Thread Block Scheduler

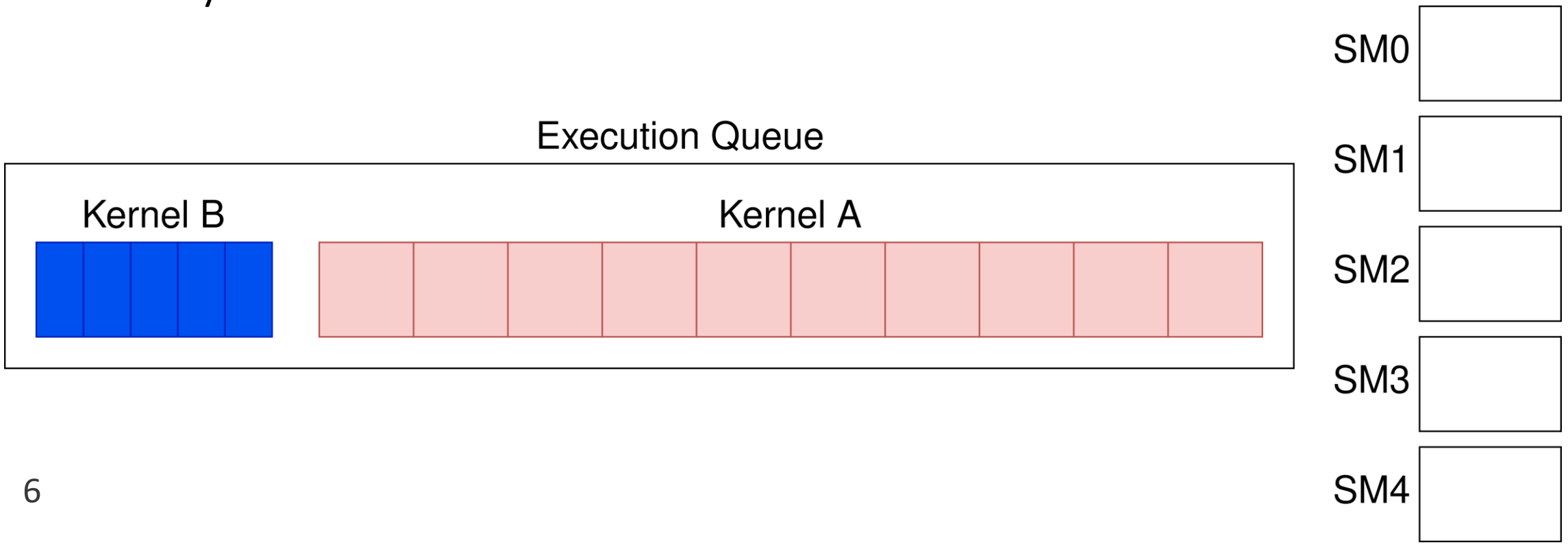
- **When** are blocks scheduled?
- **Which** block does the scheduler choose?
- **Where** will that block be placed?

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

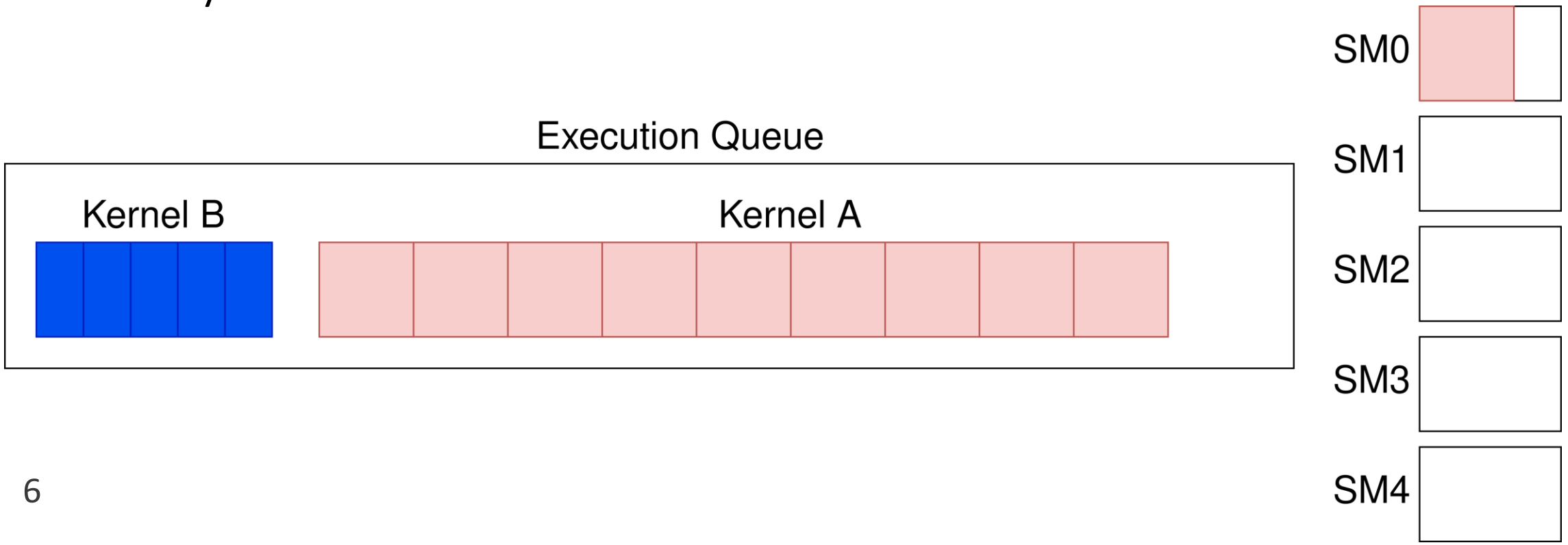
The Leftover Policy

- 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



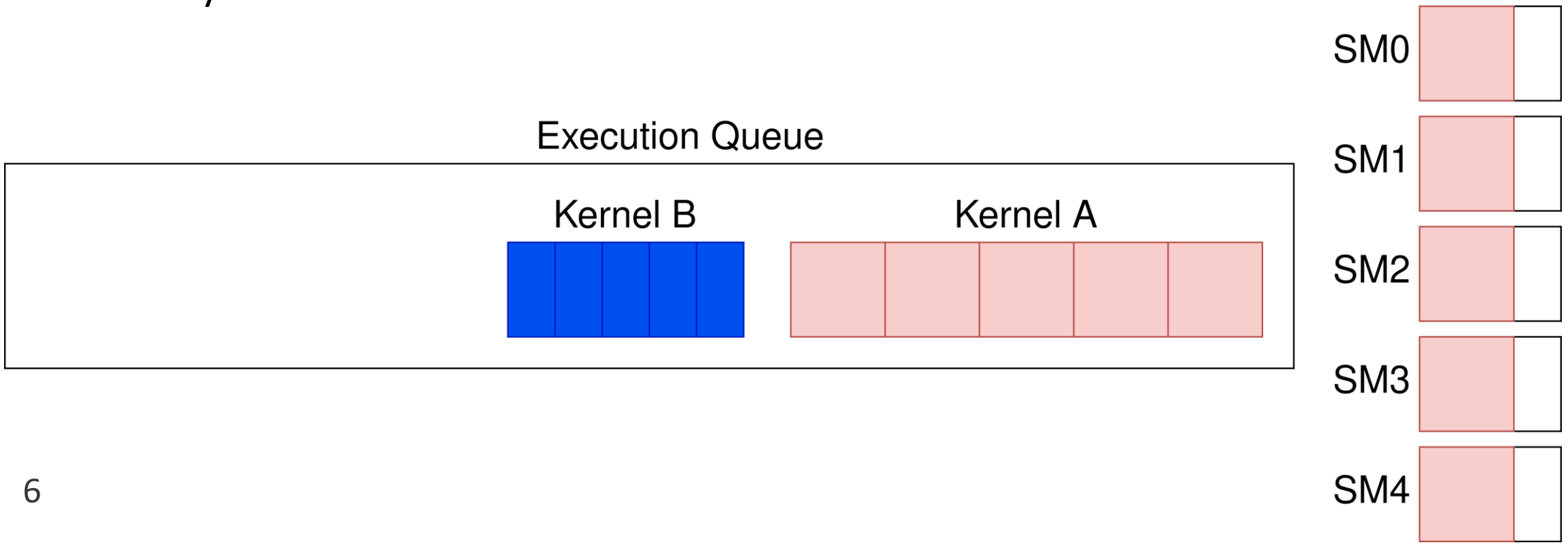
The Leftover Policy

- 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



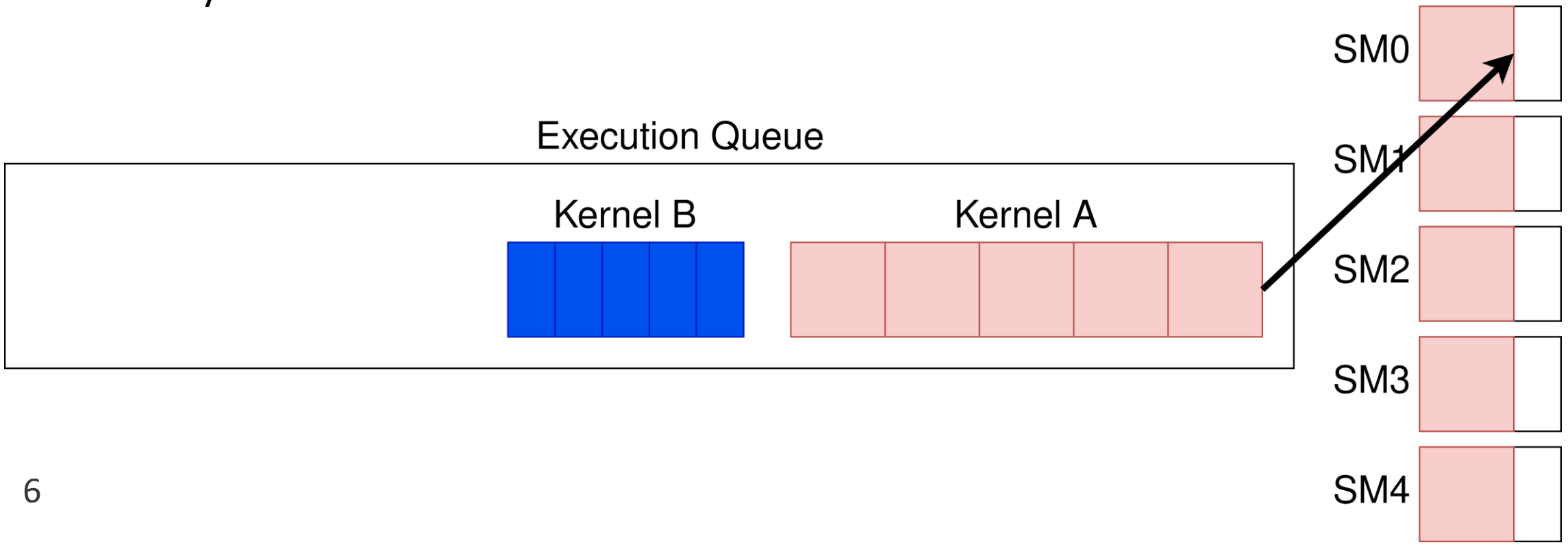
The Leftover Policy

- 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



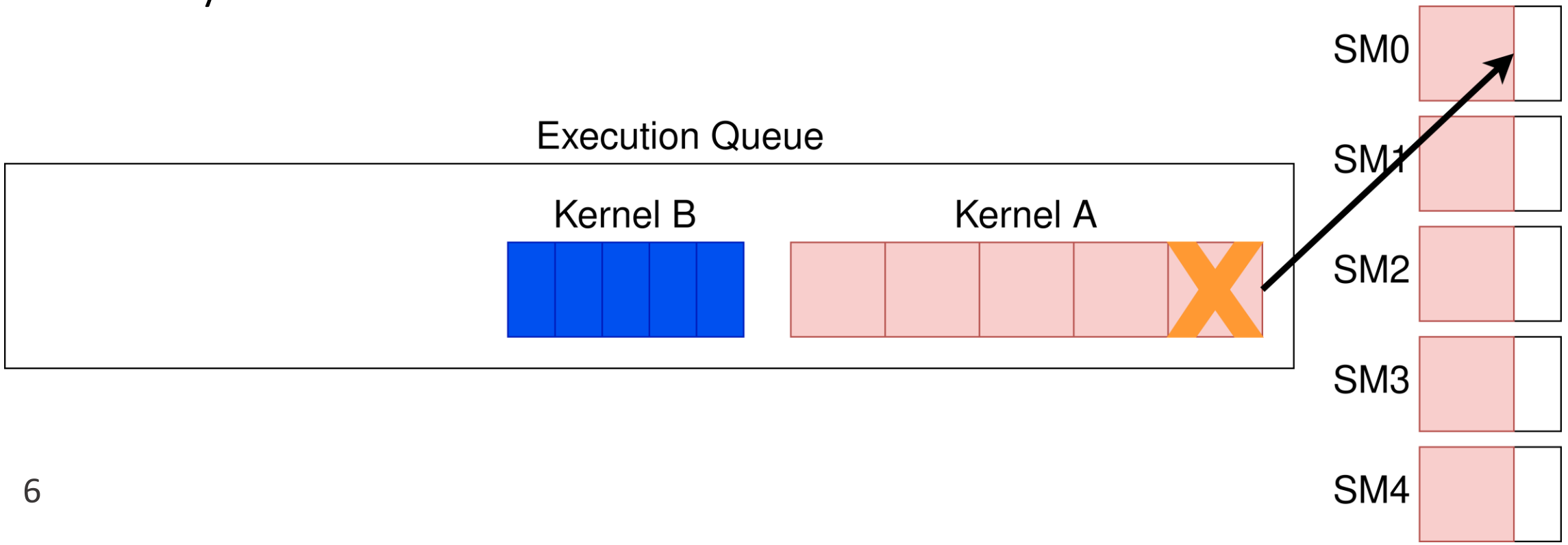
The Leftover Policy

- 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



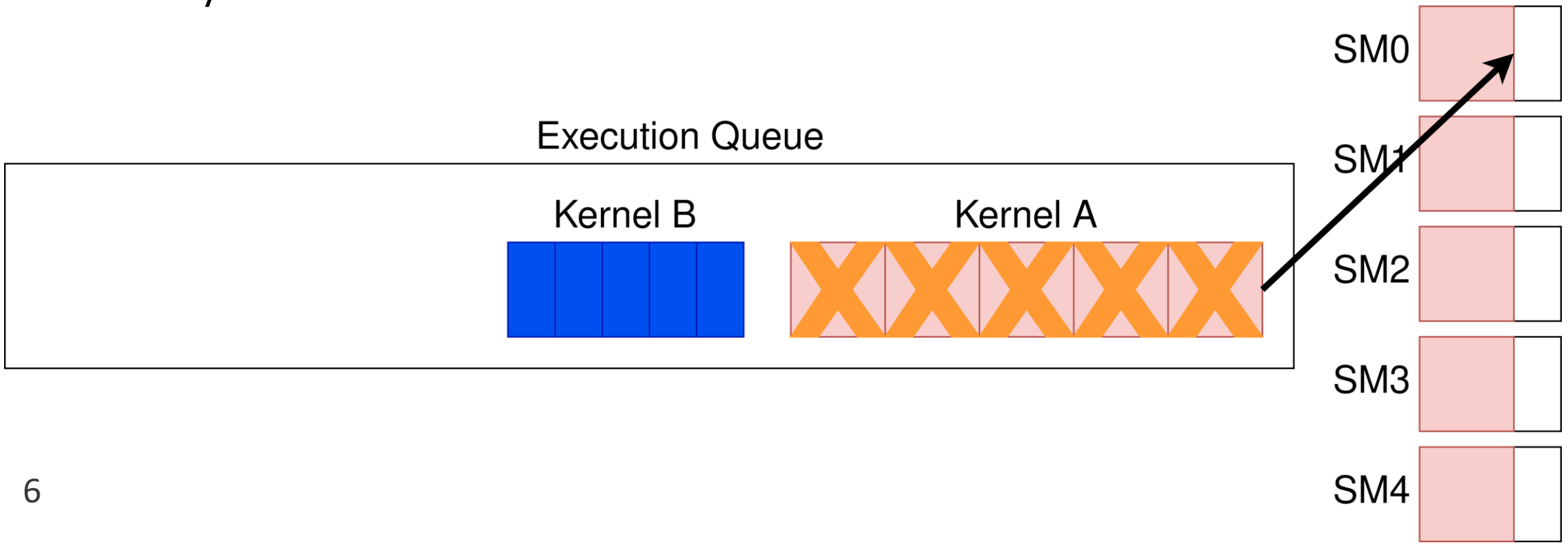
The Leftover Policy

- 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



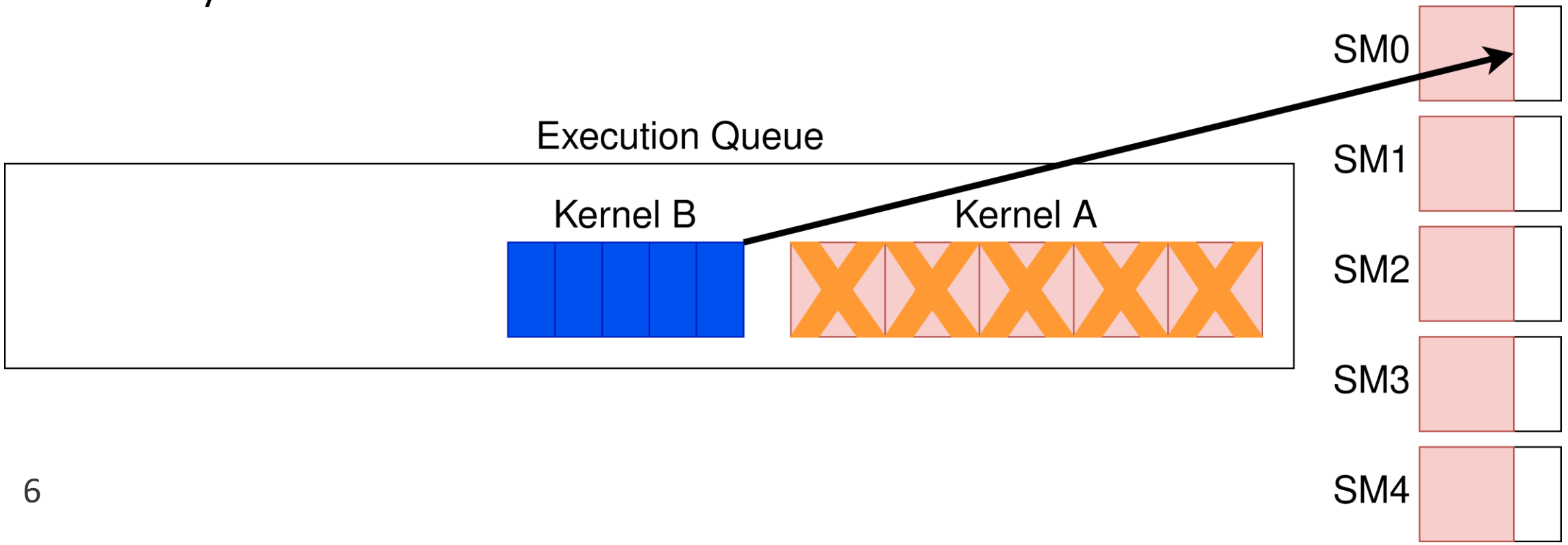
The Leftover Policy

- 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



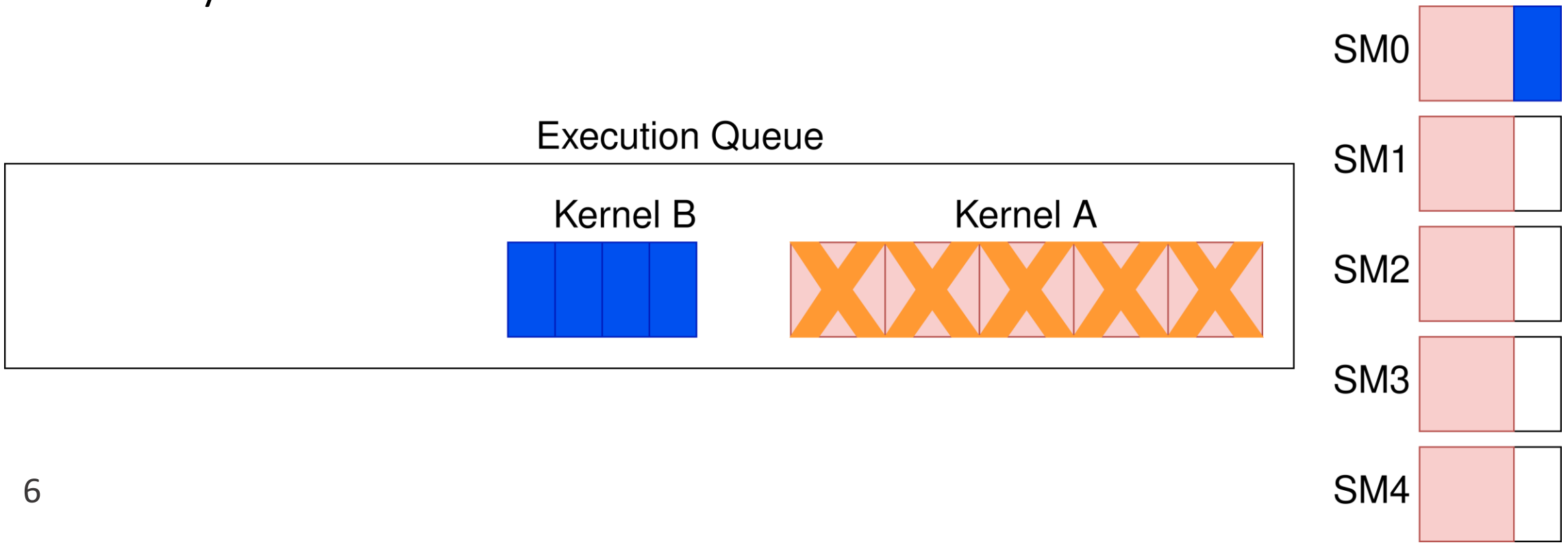
The Leftover Policy

- 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



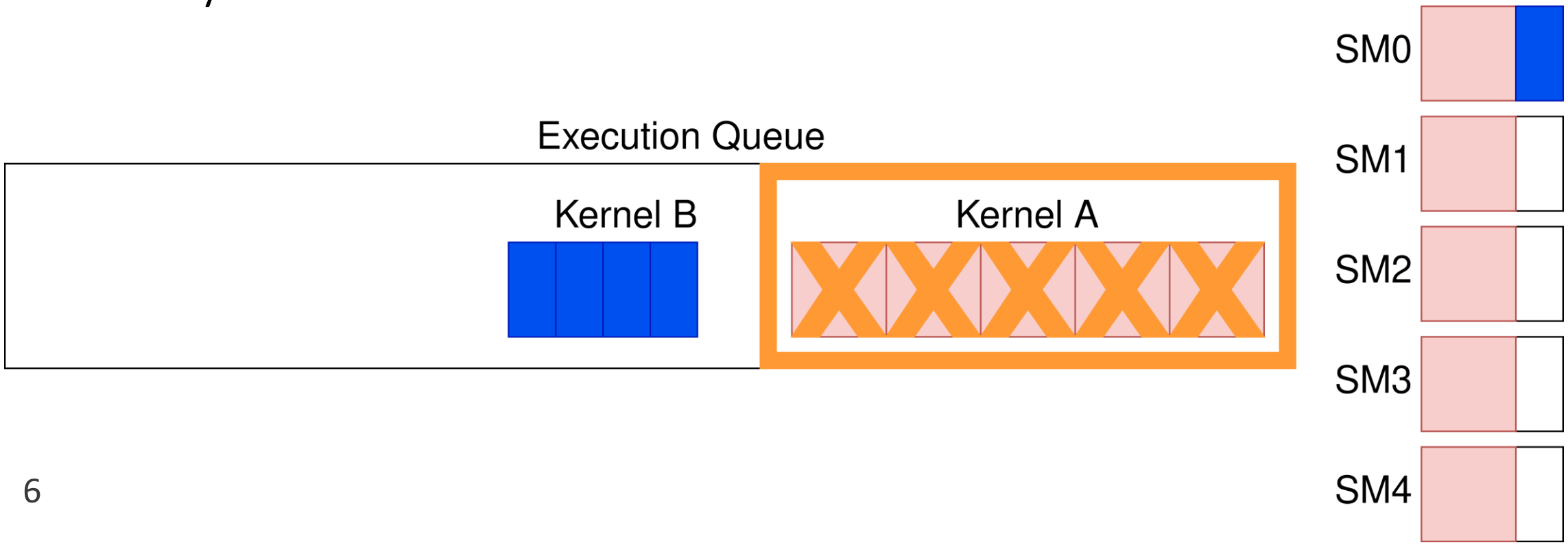
The Leftover Policy

- 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



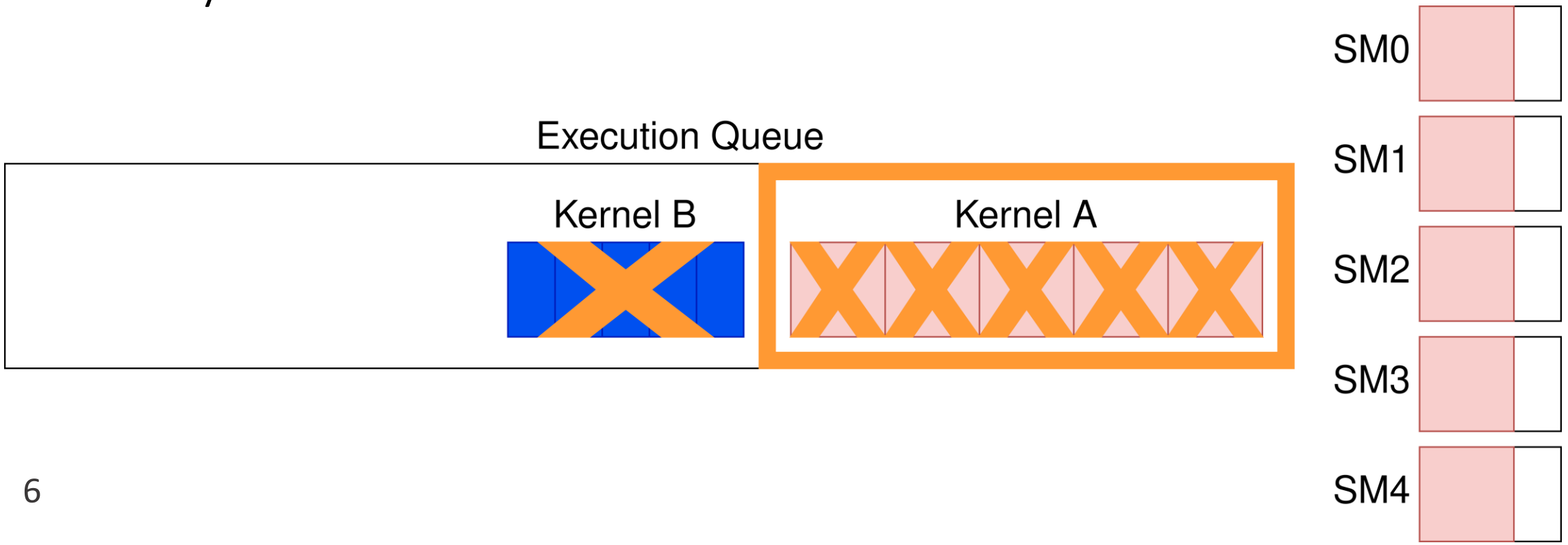
The Leftover Policy

- 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

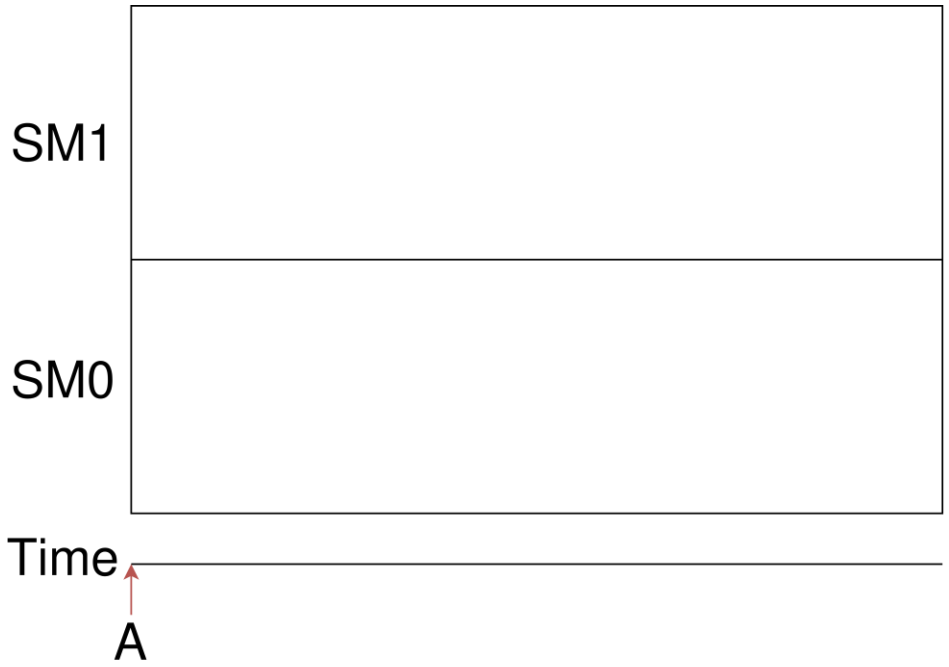


The Most-Room Policy

- 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

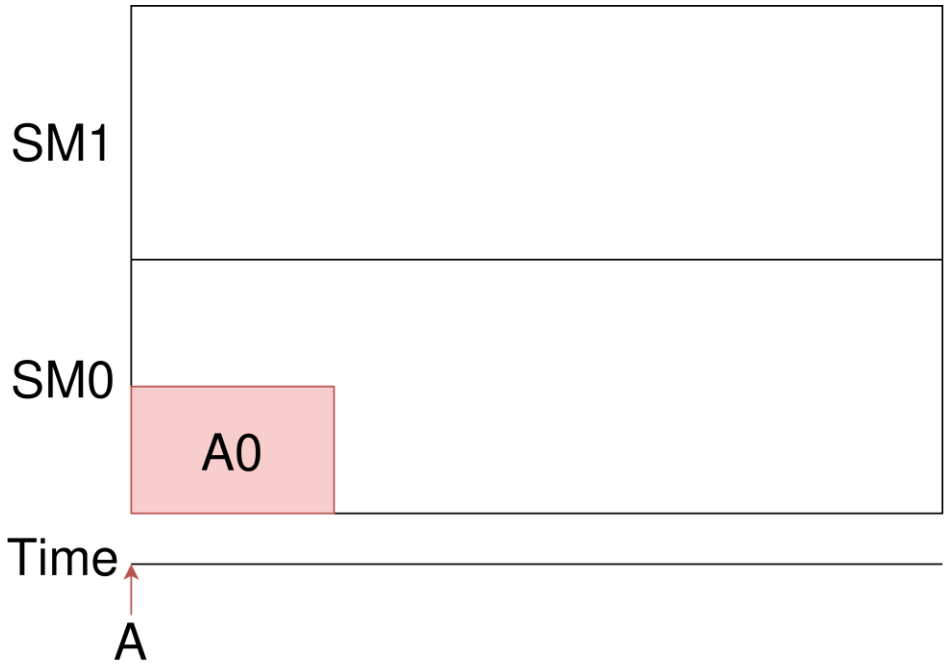
The Most-Room Policy

- 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



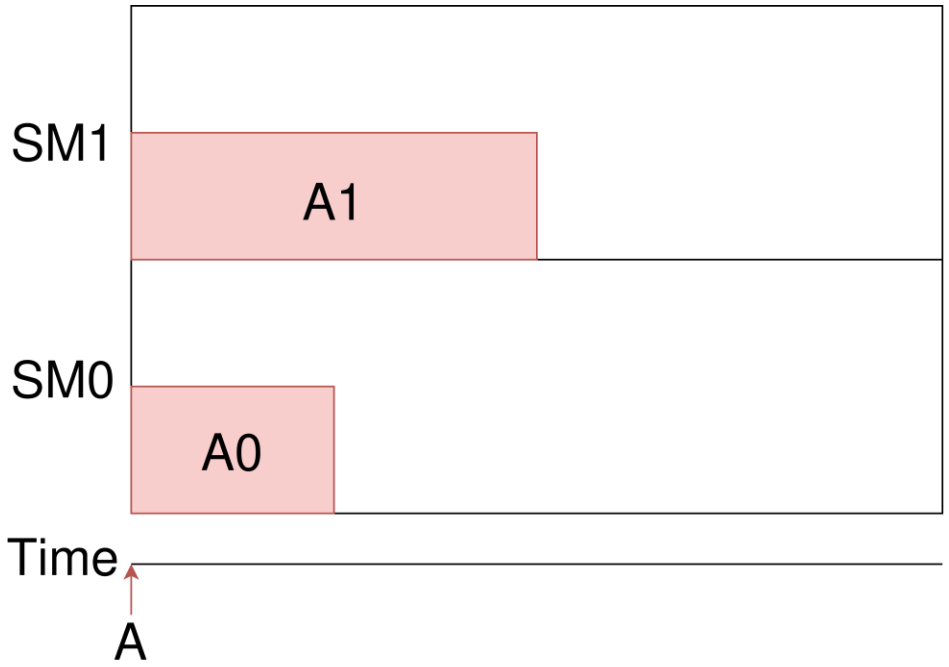
The Most-Room Policy

- 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



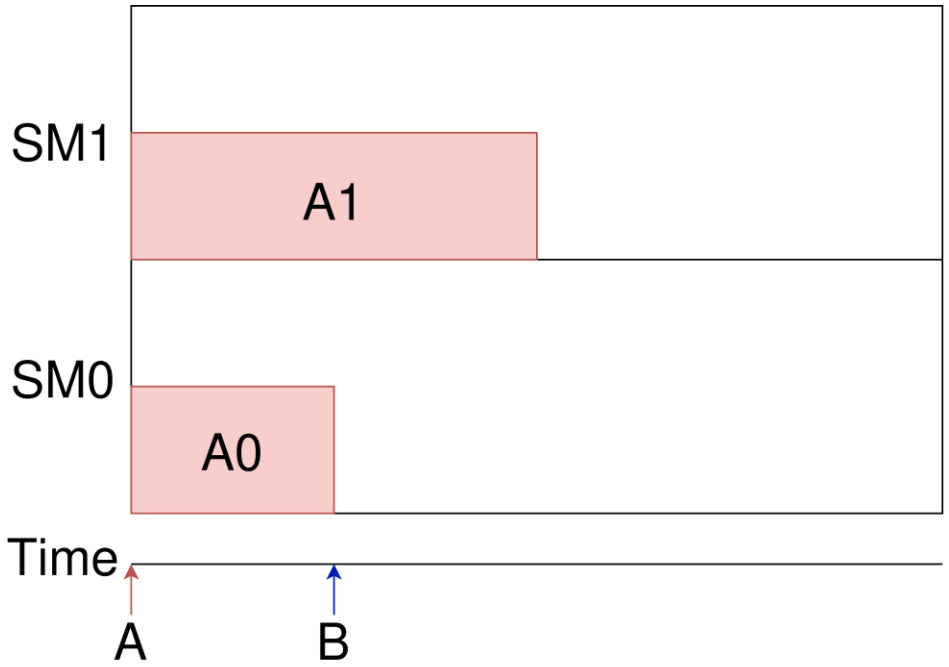
The Most-Room Policy

- 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



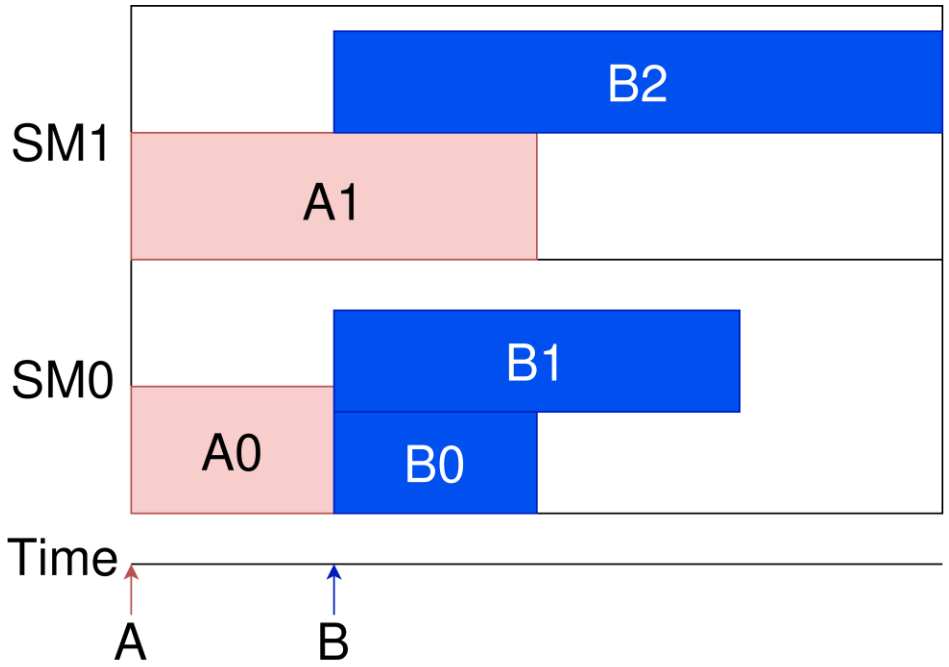
The Most-Room Policy

- 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



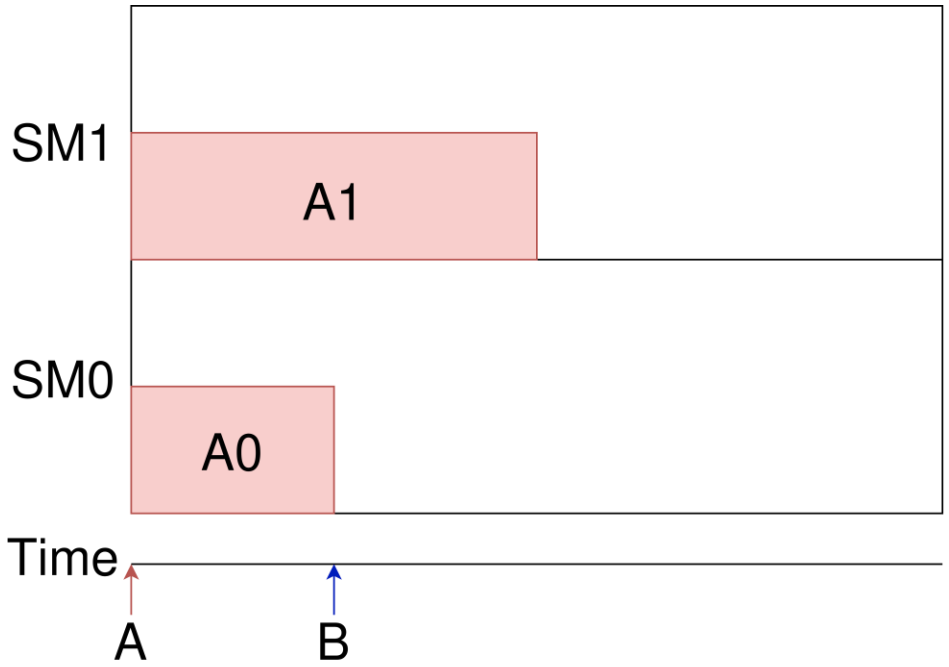
The Most-Room Policy

- 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



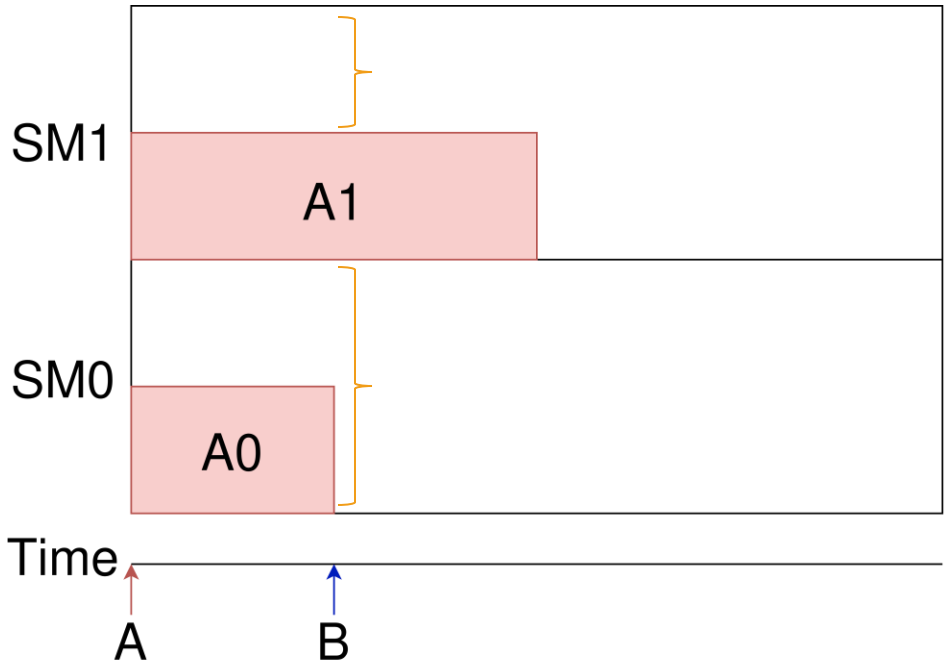
The Most-Room Policy

- 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



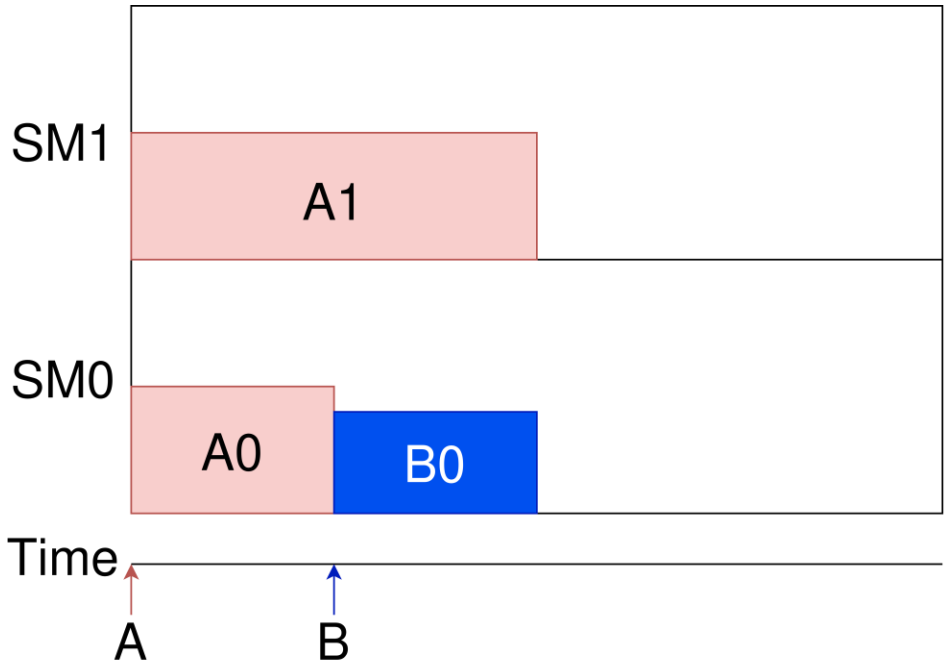
The Most-Room Policy

- 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



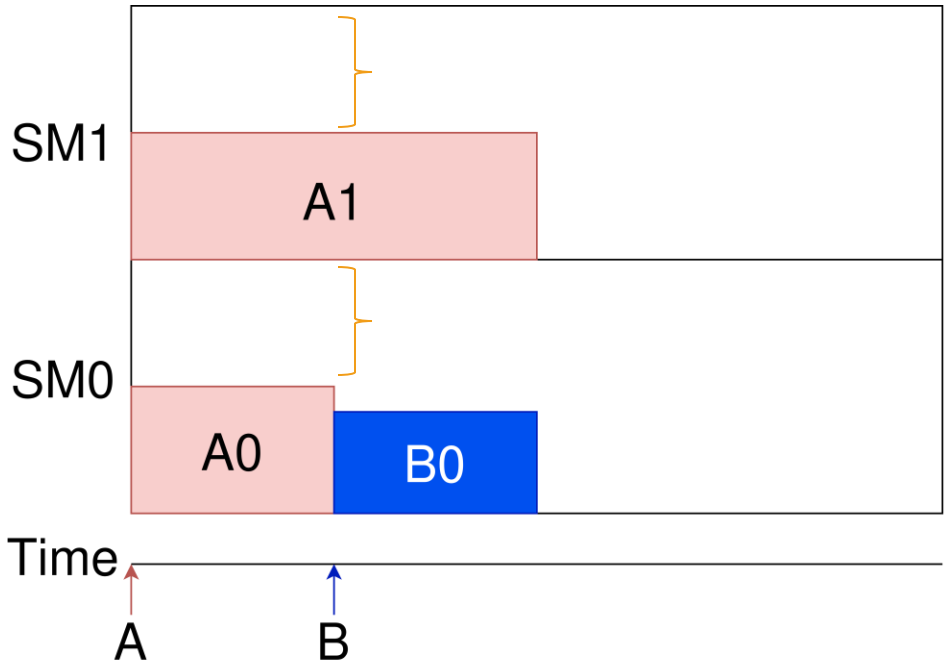
The Most-Room Policy

- 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



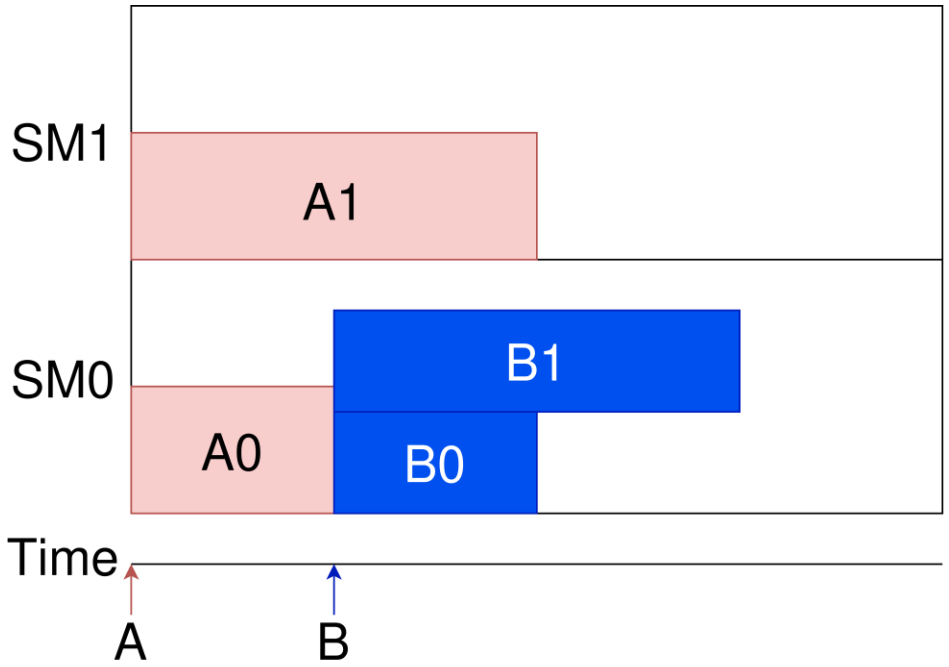
The Most-Room Policy

- 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



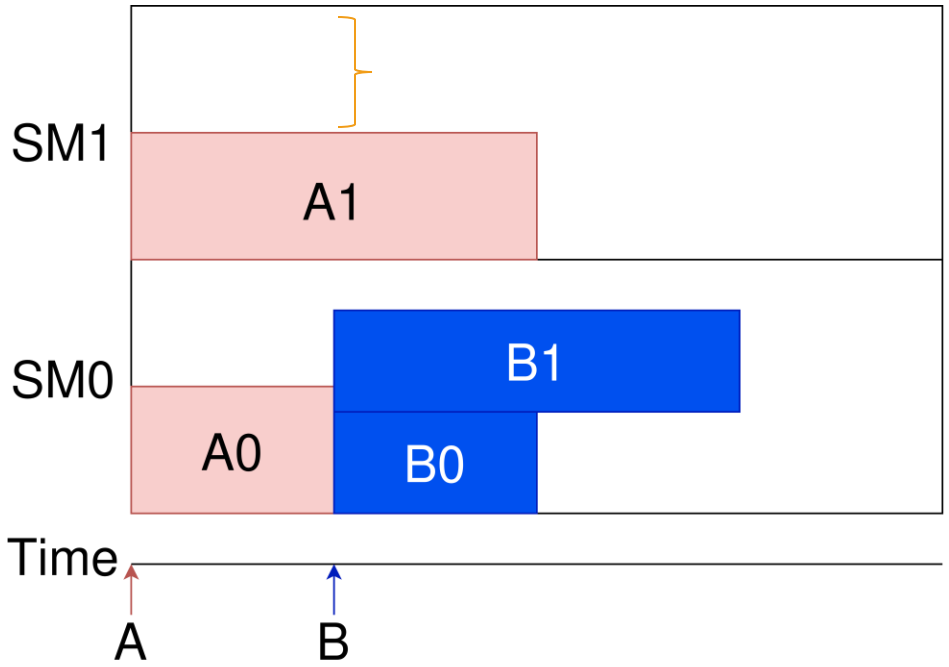
The Most-Room Policy

- 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



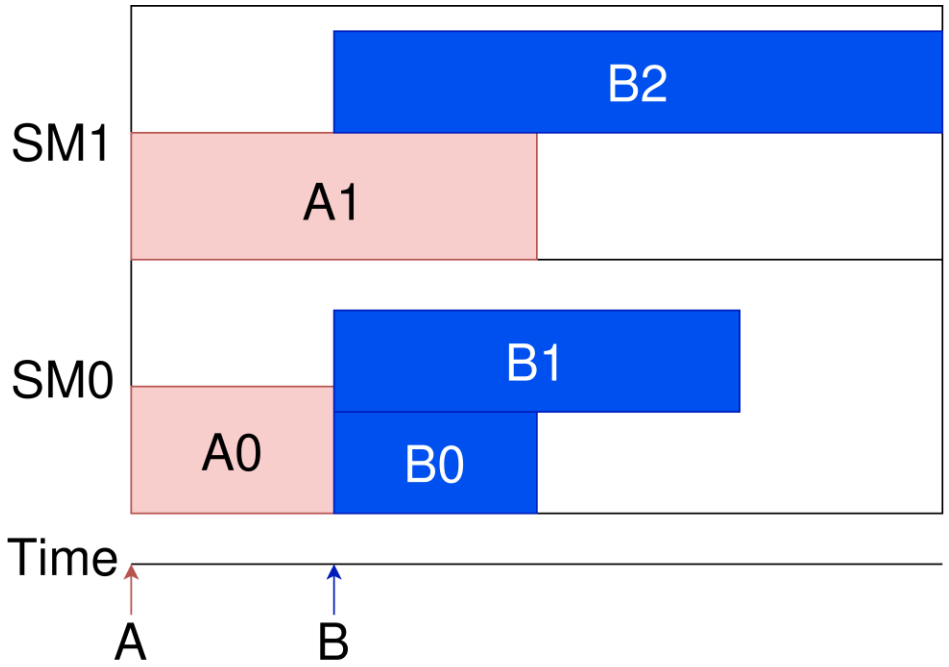
The Most-Room Policy

- 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

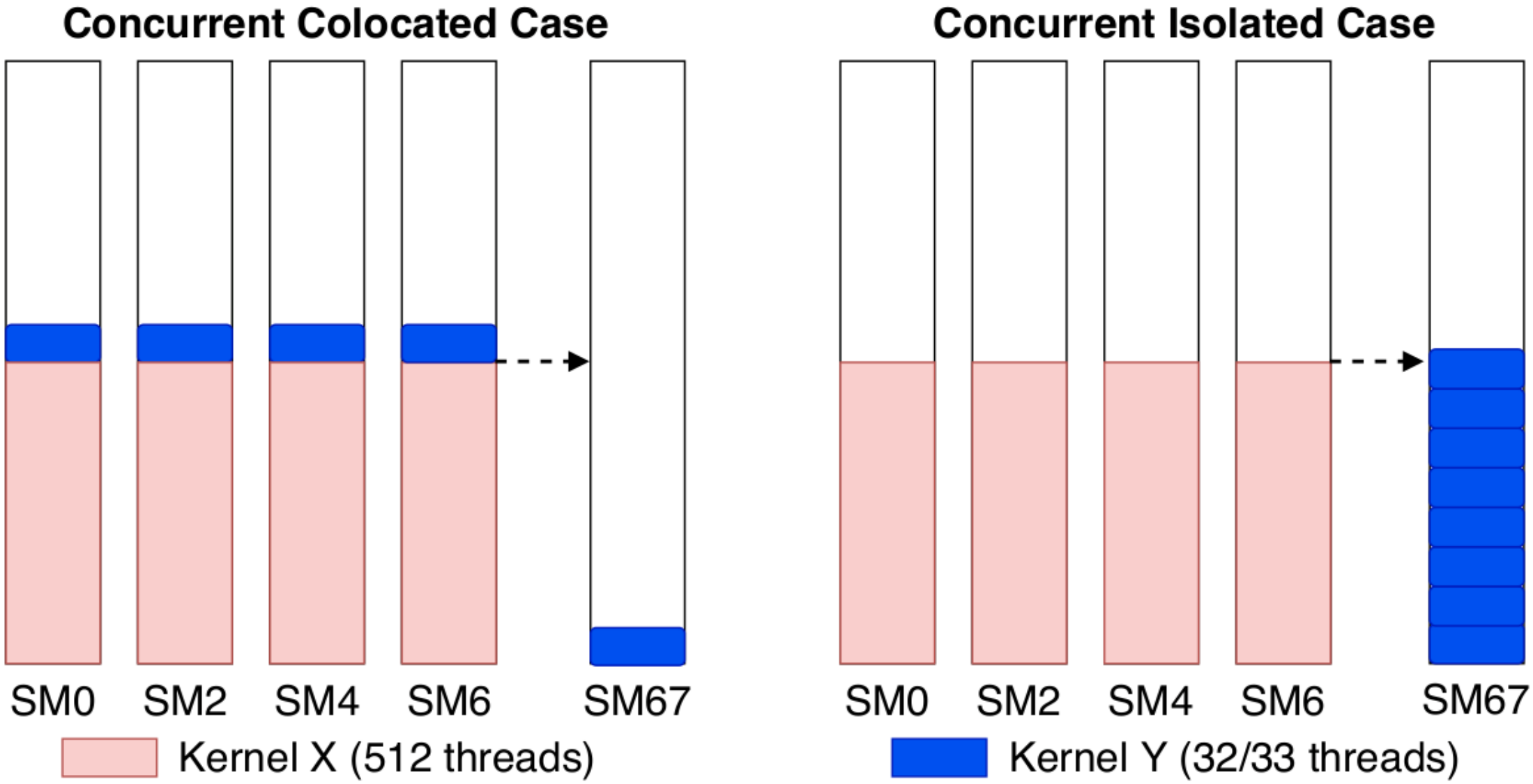


The Most-Room Policy

- 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



Performance Implications


Concurrent Colocated Case
SM0
SM2
SM4
SM6
SM67
Kernel X (512 threads)
Concurrent Isolated Case
SM0
SM2
SM4
SM6
SM67
Kernel Y (32/33 threads)

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)

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel X	Kernel Y	Total	Kernel X	Kernel Y	Kernel X	Kernel Y
L1-Cache-Dependent	85	79	164	85	79	105 (1.24X)	105 (1.33X)
Compute-Intensive	523	365	888	527	529 (1.45X)	530	676 (1.85X)
Memory-Intensive	949	10	959	951	224 (22.4X)	955	961 (96.1X)
PCIe-Bandwidth-Dep.	369	130	499	385 (1.04X)	355 (2.73X)	388 (1.05X)	466 (3.58X)

Results (Turing GPU)

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel X	Kernel Y	Total	Kernel X	Kernel Y	Kernel X	Kernel Y
L1-Cache-Dependent	85	79	164	85	79	105 (1.24X)	105 (1.33X)
Compute-Intensive	523	365	888	527	**529 (1.45X)**	530	**676 (1.85X)**
Memory-Intensive	949	10	959	951	**224 (22.4X)**	955	**961 (96.1X)**
PCIe-Bandwidth-Dep.	369	130	499	385 (1.04X)	355 (2.73X)	388 (1.05X)	466 (3.58X)

Results (Turing GPU)

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel X	Kernel Y	Total	Kernel X	Kernel Y	Kernel X	Kernel Y
L1-Cache-Dependent	85	79	164	85	79	**105 (1.24X)**	**105 (1.33X)**
Compute-Intensive	523	365	888	527	529 (1.45X)	530	676 (1.85X)
Memory-Intensive	949	10	959	951	224 (22.4X)	955	961 (96.1X)
PCIe-Bandwidth-Dep.	369	130	499	**385 (1.04X)**	**355 (2.73X)**	**388 (1.05X)**	**466 (3.58X)**

Results (Turing GPU)

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel X	Kernel Y	Total	Kernel X	Kernel Y	Kernel X	Kernel Y
L1-Cache-Dependent	85	79	**164**	**85**	79	105 (1.24X)	105 (1.33X)
Compute-Intensive	523	365	**888**	527	**529 (1.45X)**	530	676 (1.85X)
Memory-Intensive	949	10	**959**	**951**	224 (22.4X)	955	961 (96.1X)
PCIe-Bandwidth-Dep.	369	130	**499**	**385 (1.04X)**	355 (2.73X)	388 (1.05X)	466 (3.58X)

Results (Turing GPU)

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel X	Kernel Y	Total	Kernel X	Kernel Y	Kernel X	Kernel Y
L1-Cache-Dependent	85	79	**164**	85	79	**105 (1.24X)**	**105 (1.33X)**
Compute-Intensive	523	365	**888**	527	529 (1.45X)	530	**676 (1.85X)**
Memory-Intensive	949	10	**959**	951	224 (22.4X)	955	**961 (96.1X)**
PCIe-Bandwidth-Dep.	369	130	**499**	385 (1.04X)	355 (2.73X)	388 (1.05X)	**466 (3.58X)**

Results (Turing GPU)

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel X	Kernel Y	Total	Kernel X	Kernel Y	Kernel X	Kernel Y
L1-Cache-Dependent	85	79	164	85	79	105 (1.24X)	105 (1.33X)
Compute-Intensive	523	365	888	527	529 (1.45X)	530	676 (1.85X)
Memory-Intensive	949	10	959	951	224 (22.4X)	955	961 (96.1X)
PCIe-Bandwidth-Dep.	369	130	499	385 (1.04X)	355 (2.73X)	388 (1.05X)	466 (3.58X)

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