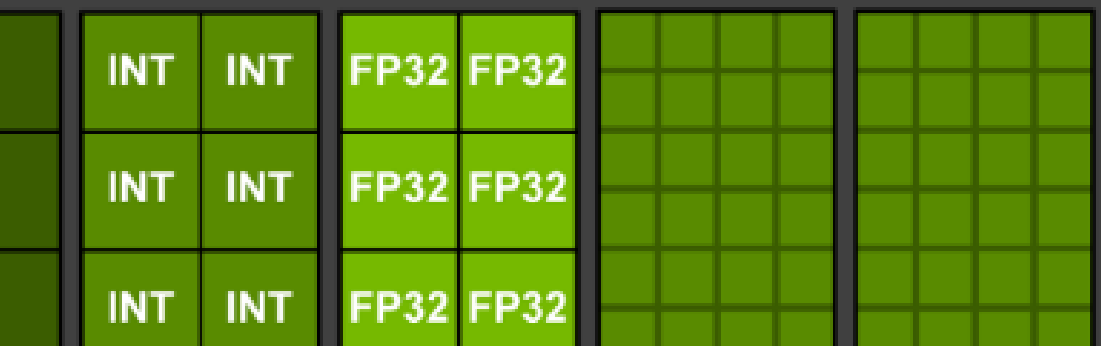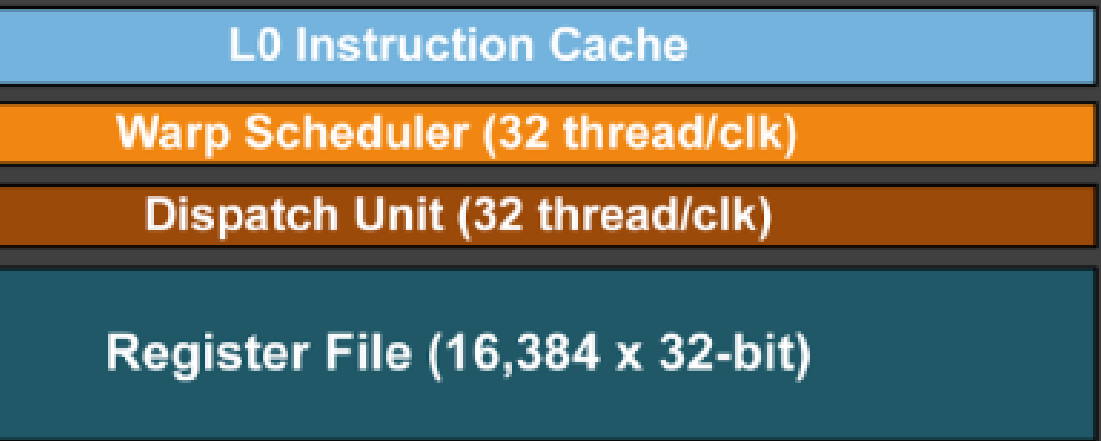# *GPUs going once…*
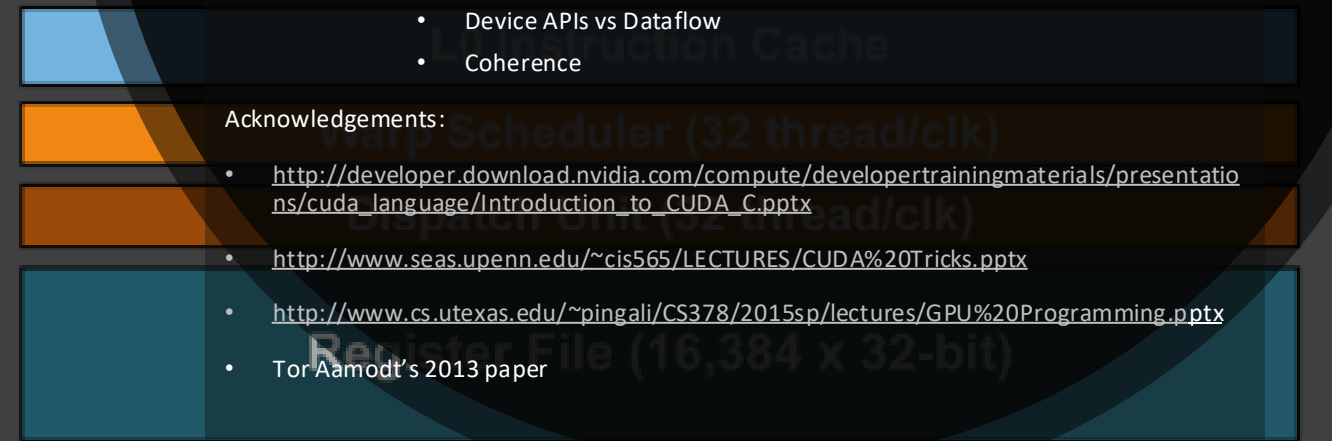# *GPUs going twice…*
you get the idea

Chris Rossbach

cs378
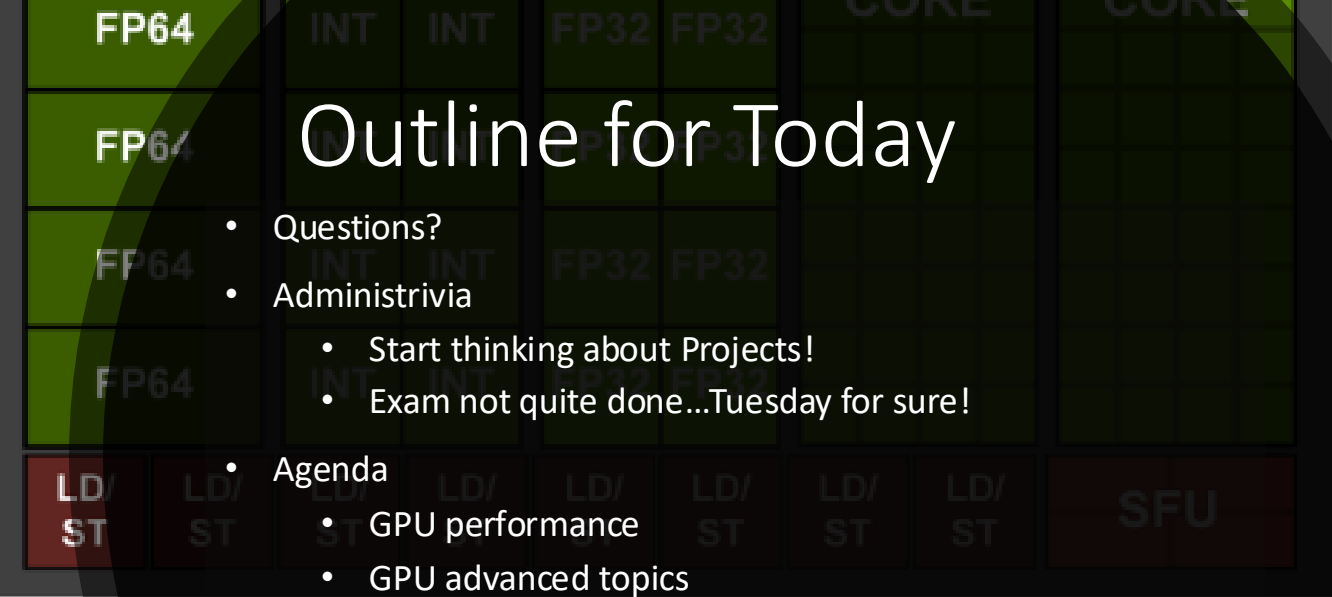
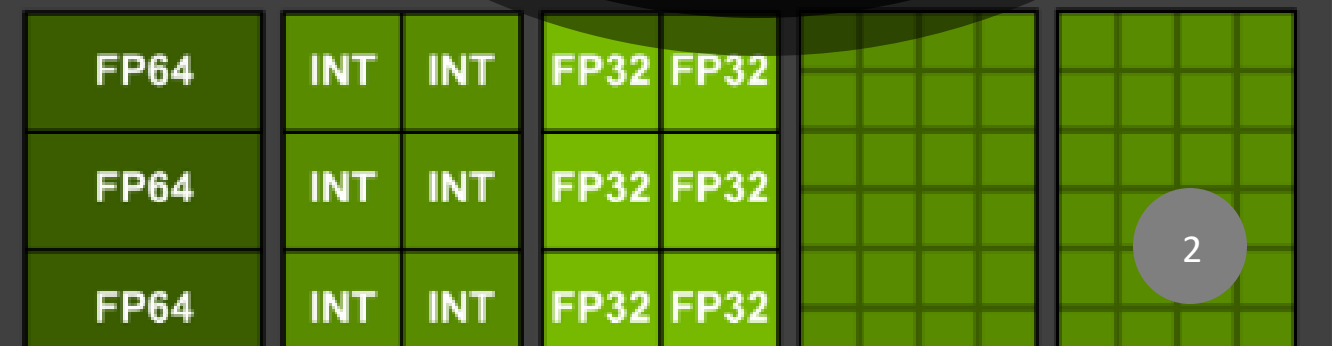# Outline for Today

- Questions?
- Administrivia
  - Start thinking about Projects!
  - Exam not quite done…Tuesday for sure!
- Agenda
  - GPU performance
  - GPU advanced topics
    - Divergence
    - Device APIs vs Dataflow
    - Coherence

Acknowledgements:
- http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx
- http://www.seas.upenn.edu/~cis565/LECTURES/CUDA%20Tricks.pptx
- http://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/GPU%20Programming.pptx
- Tor Aamodt's 2013 paper

2

Faux Quiz Questions

- How is occupancy defined (in CUDA nomenclature)?
- What's the difference between a block scheduler (e.g. Giga-Thread Engine) and a warp scheduler?
- Modern CUDA supports UVM to eliminate the need for cudaMalloc and cudaMemcpy*. Under what conditions might you want to use or not use it and why?
- What is control flow divergence? How does it impact performance?
- What is a bank conflict?
- What is work efficiency?
- What is the difference between a thread block scheduler and a warp scheduler?
- How are atomics implemented in modern GPU hardware?
- How is __shared__ memory implemented by modern GPU hardware?
- Why is __shared__ memory necessary if GPUs have an L1 cache? When will an L1 cache provide all the benefit of __shared__ memory and when will it not?
- Is cudaDeviceSynchronize still necessary after copyback if I have just one CUDA stream?

How many threads/blocks?

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

How many threads/blocks?

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

How many threads/blocks?

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

- Usually things are correct if grid*block dims >= input size
- Getting good performance is another matter
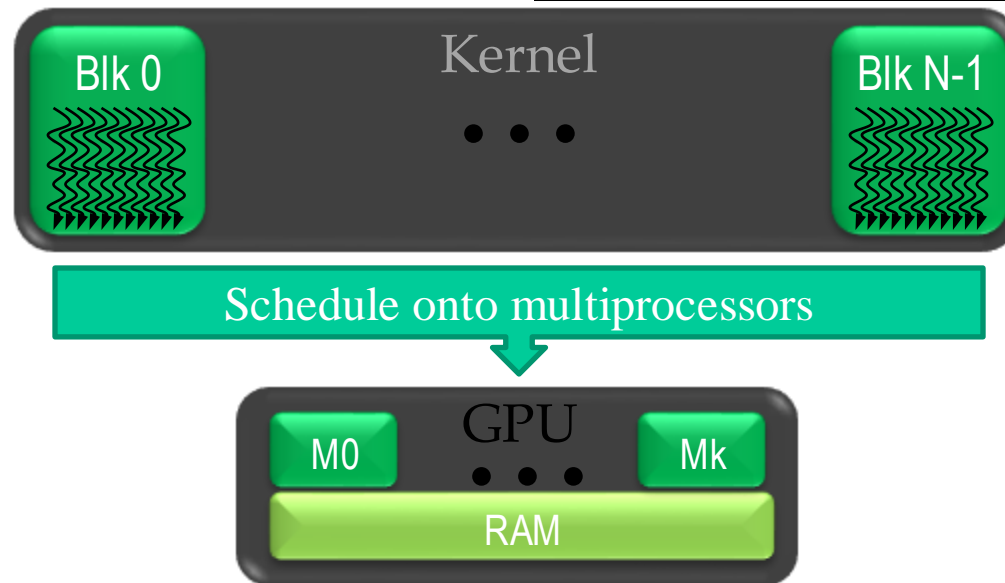
Review: Internals

```
__host__
void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
  addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
  int i = blockIdx.x * blockDim.x
               + threadIdx.x;
  if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```

Review: Internals

```
__host__
void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
  addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
  int i = blockIdx.x * blockDim.x
              + threadIdx.x;
  if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```
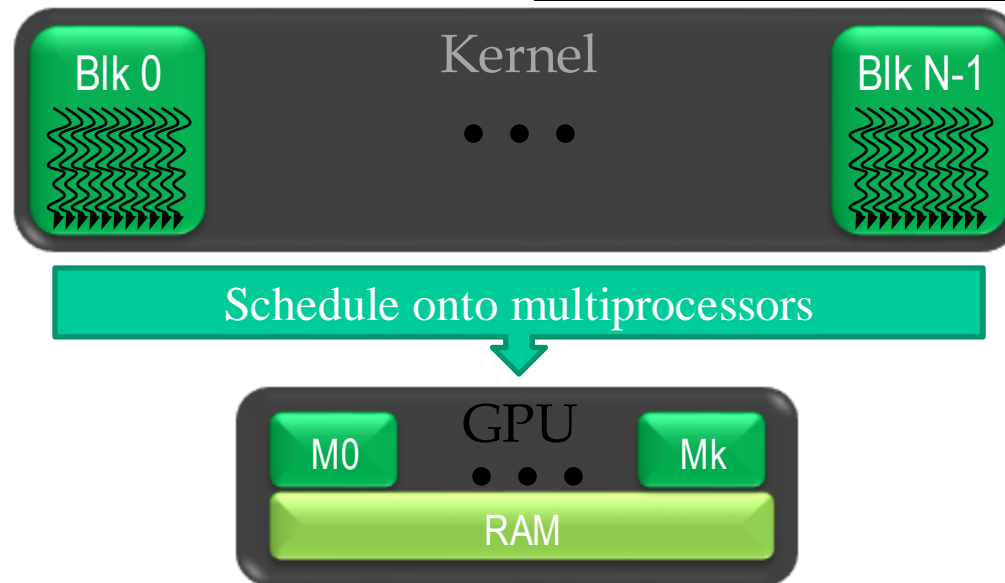

Kernel

Blk 0   . . .   Blk N-1

Schedule onto multiprocessors

GPU

M0   . . .   Mk

RAM

Review: Internals

```
__host__
void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
  addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
  int i = blockIdx.x * blockDim.x
          + threadIdx.x;
  if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```


Kernel

Blk 0   • • •   Blk N-1

Schedule onto multiprocessors

GPU

M0   • • •   Mk

RAM

How are threads scheduled?
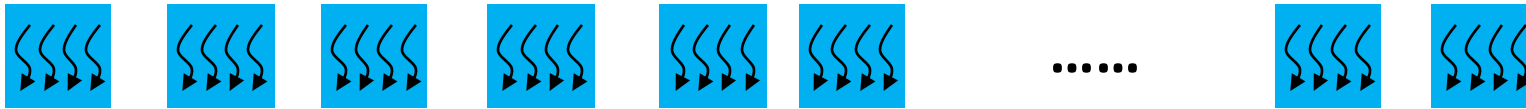
Thread Blocks, Warps, Scheduling

Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

Thread Blocks

SMs

Register File

Cores

L1 Cache/Shared Memory

SM_0

Register File

Cores

L1 Cache/Shared Memory

SM_1

Register File

Cores

L1 Cache/Shared Memory

SM_12

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each

Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

Thread Blocks

SMs

Register File
Cores
L1 Cache/Shared Memory

SM_0

Register File
Cores
L1 Cache/Shared Memory

SM_1

......

Register File
Cores
L1 Cache/Shared Memory

SM_12

- SMs split blocks into warps
- Unit of HW scheduling for SM
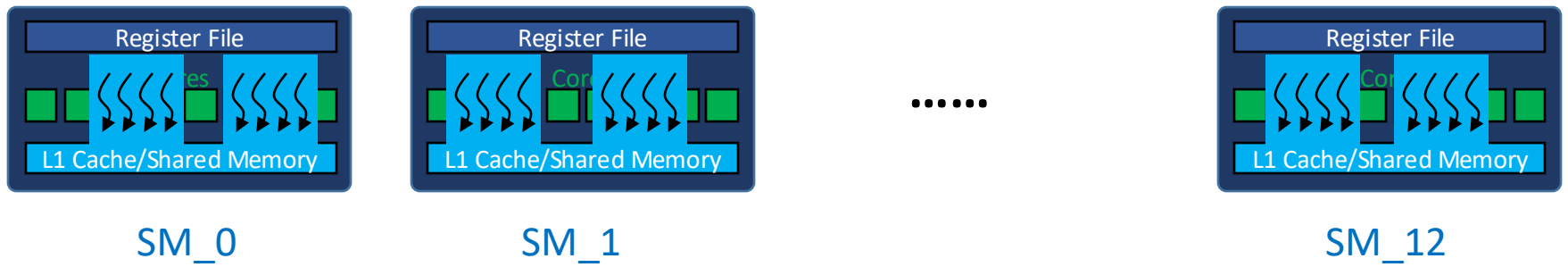- 32 threads each

Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

Thread Blocks

SMs

SM_0   SM_1   SM_12

- SMs split blocks into warps
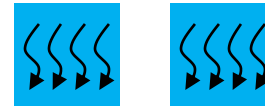- Unit of HW scheduling for SM
- 32 threads each

Thread Blocks, Warps, Scheduling

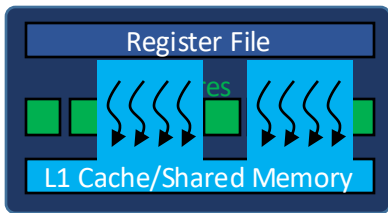Suppose one TB (threadblock) has 64 threads (2 warps)

Thread Blocks

......

SMs

Register File

Cores

L1 Cache/Shared Memory

SM_0

Register File

Cores

L1 Cache/Shared Memory

SM_1

......

Register File

Cores

L1 Cache/Shared Memory

SM_12

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each

Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)
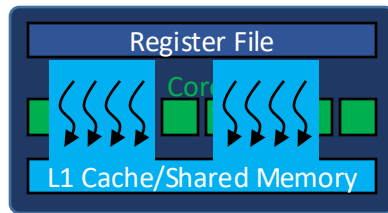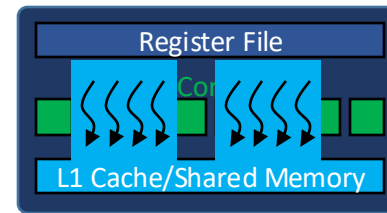
Thread Blocks

Remaining TBs are queued

......

SMs

| Register File |
| L1 Cache/Shared Memory |

SM_0

| Register File |
| L1 Cache/Shared Memory |

SM_1

......

| Register File |
| L1 Cache/Shared Memory |

SM_12

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each

GPU Performance Metric: *Occupancy*

GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized

GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

Shouldn't we just create as many threads as possible?

A Taco Bar

A Taco Bar

A Taco Bar

- Where is the parallelism here?

GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar
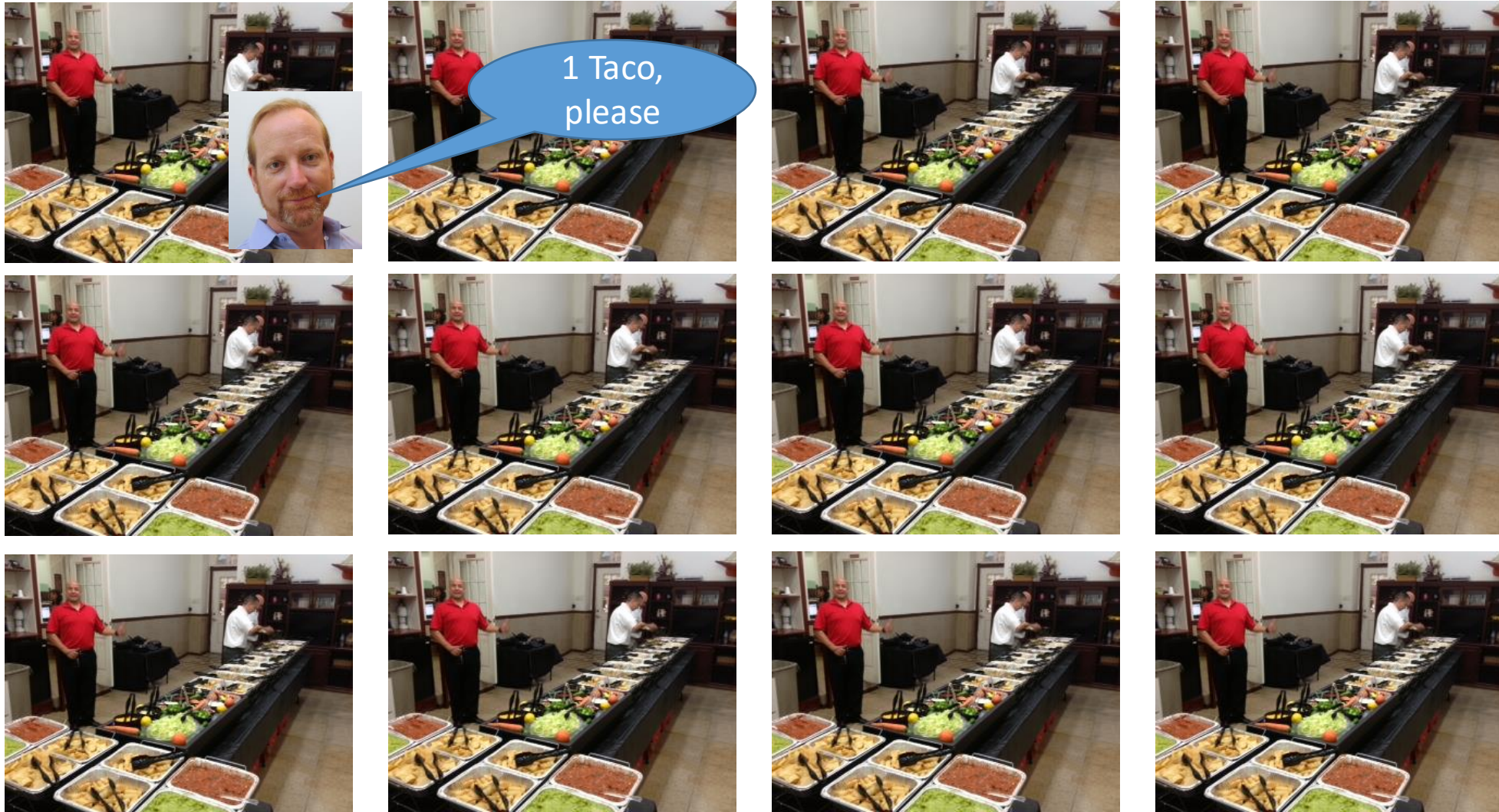
GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

1 Taco, please

Where is the parallelism here?
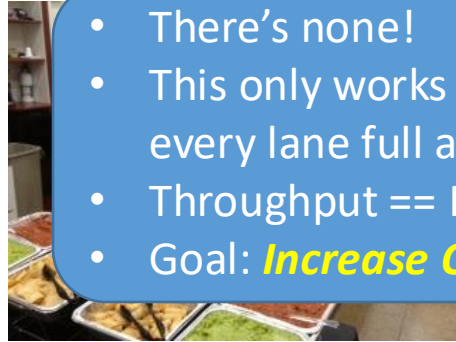
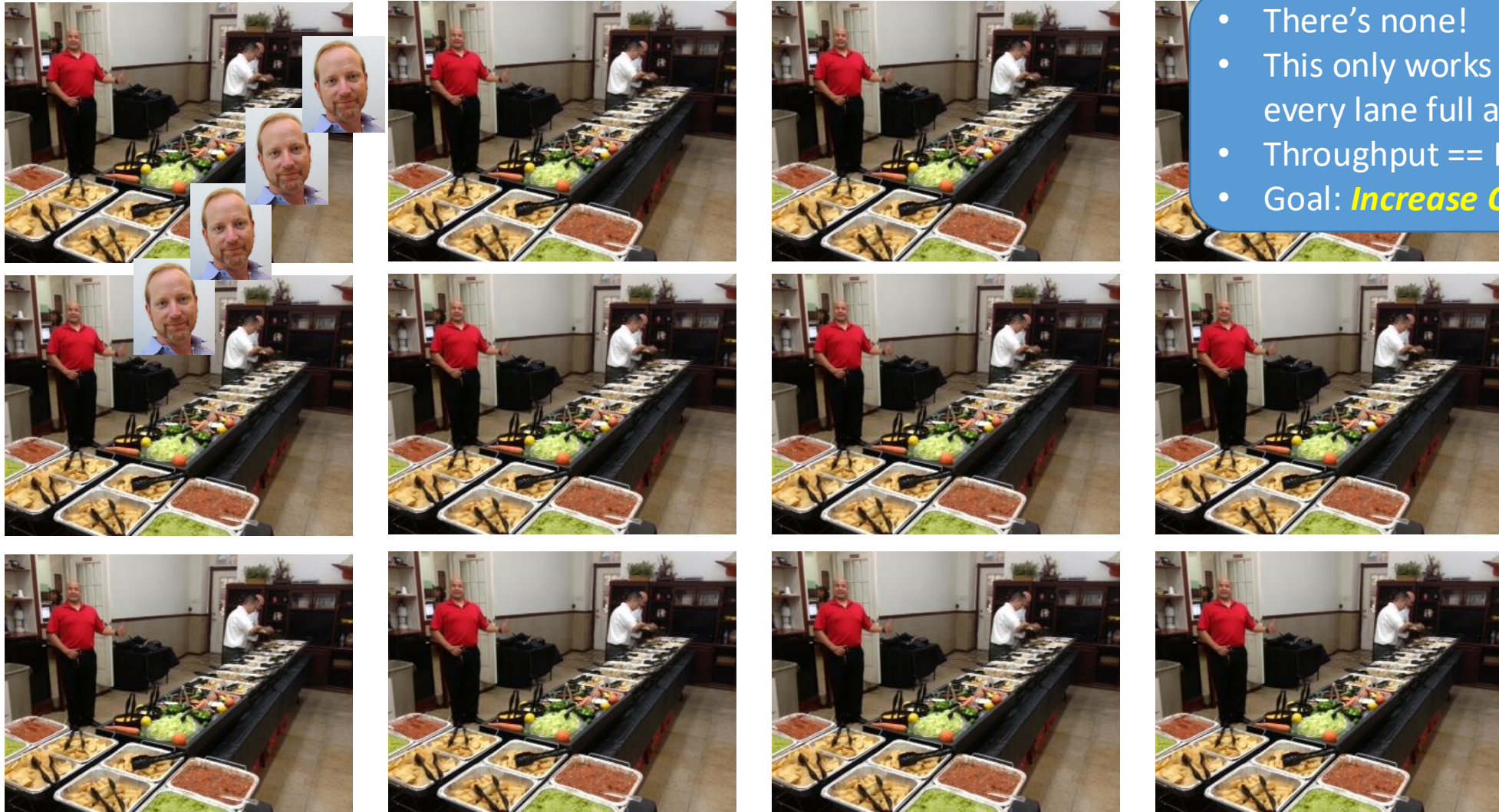# GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

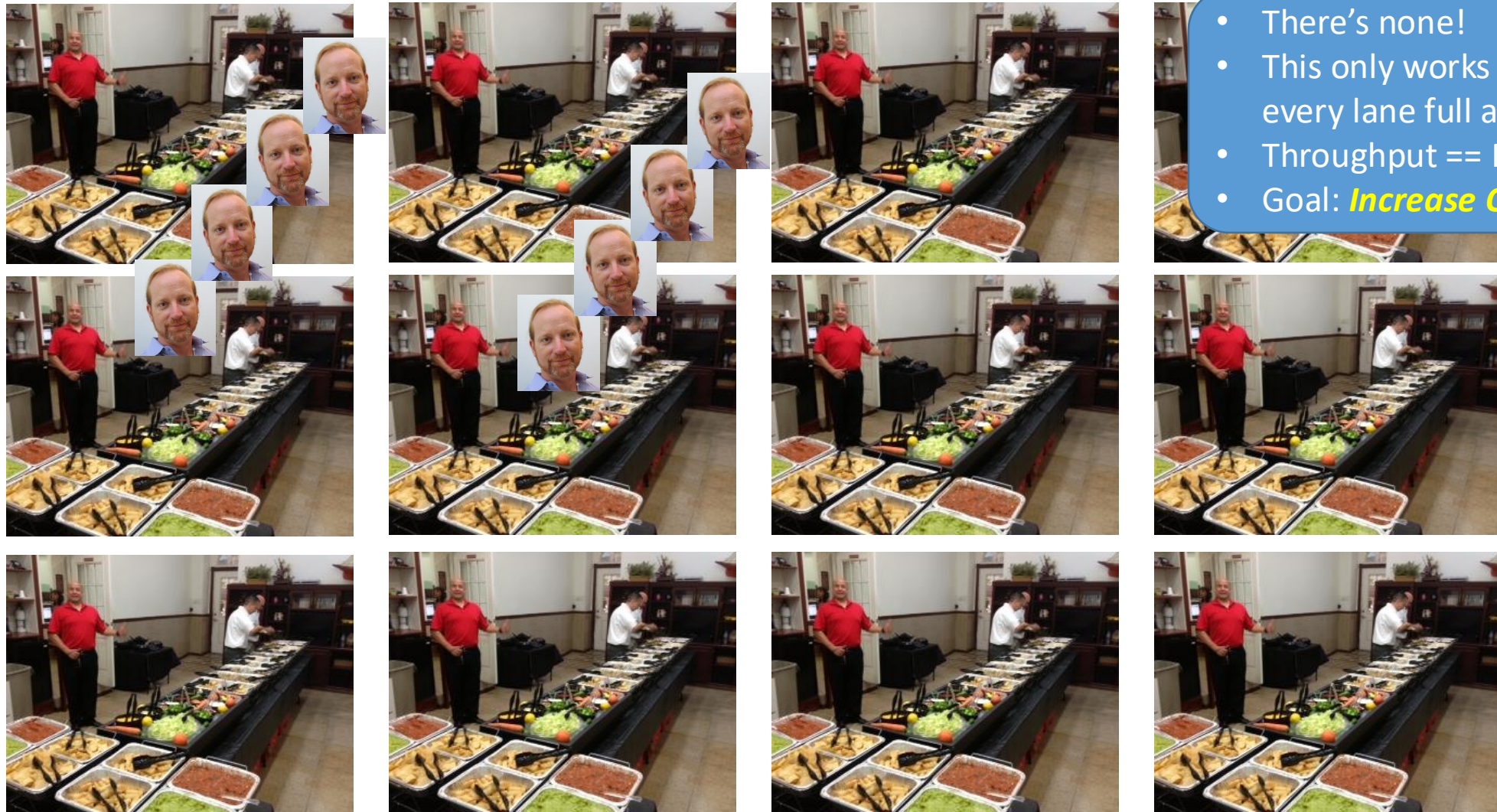1 Taco, please

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

1 Taco, please

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

GPU: a multi-lane Taco Bar
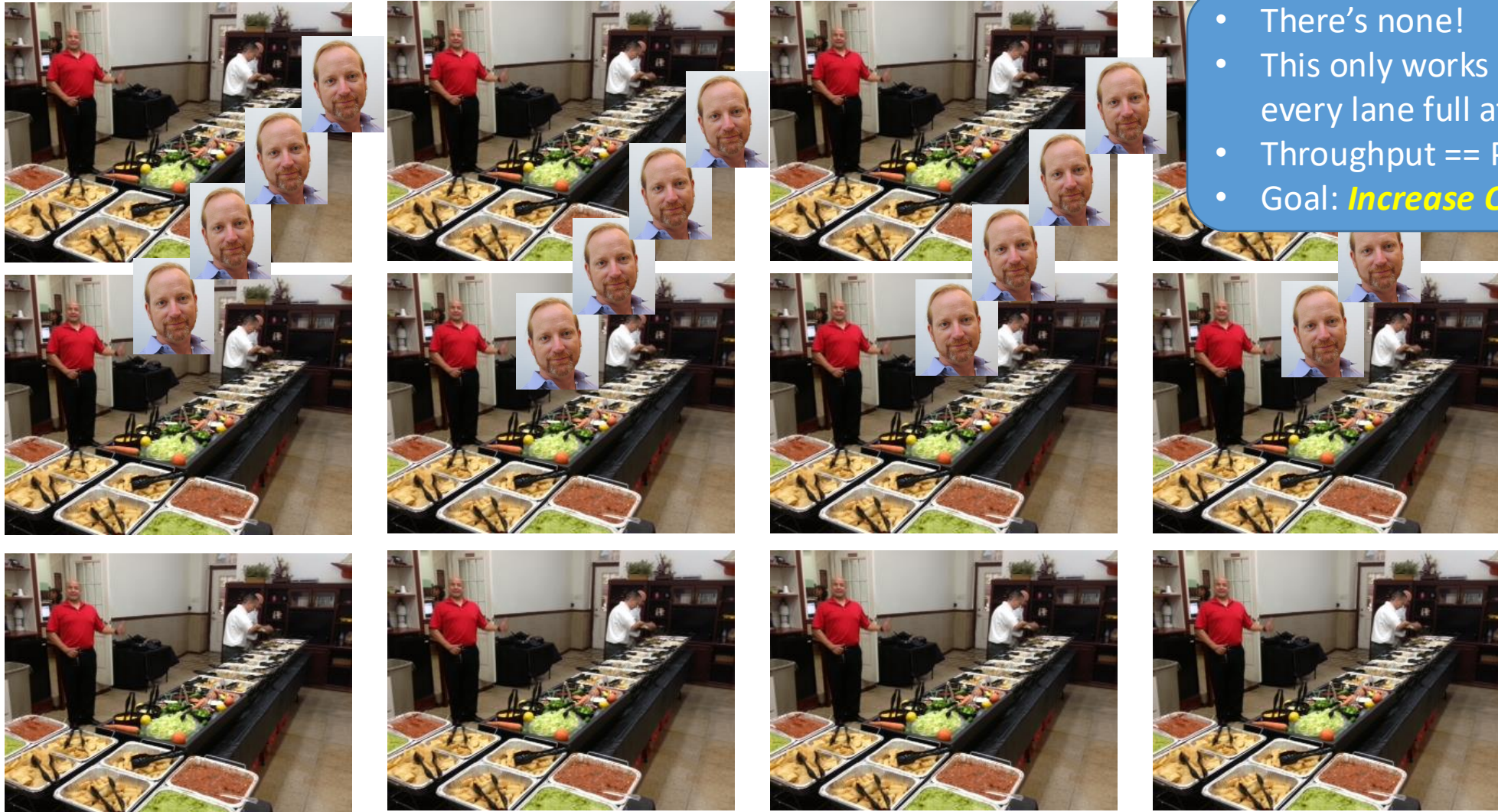

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

11

GPU: a multi-lane Taco Bar
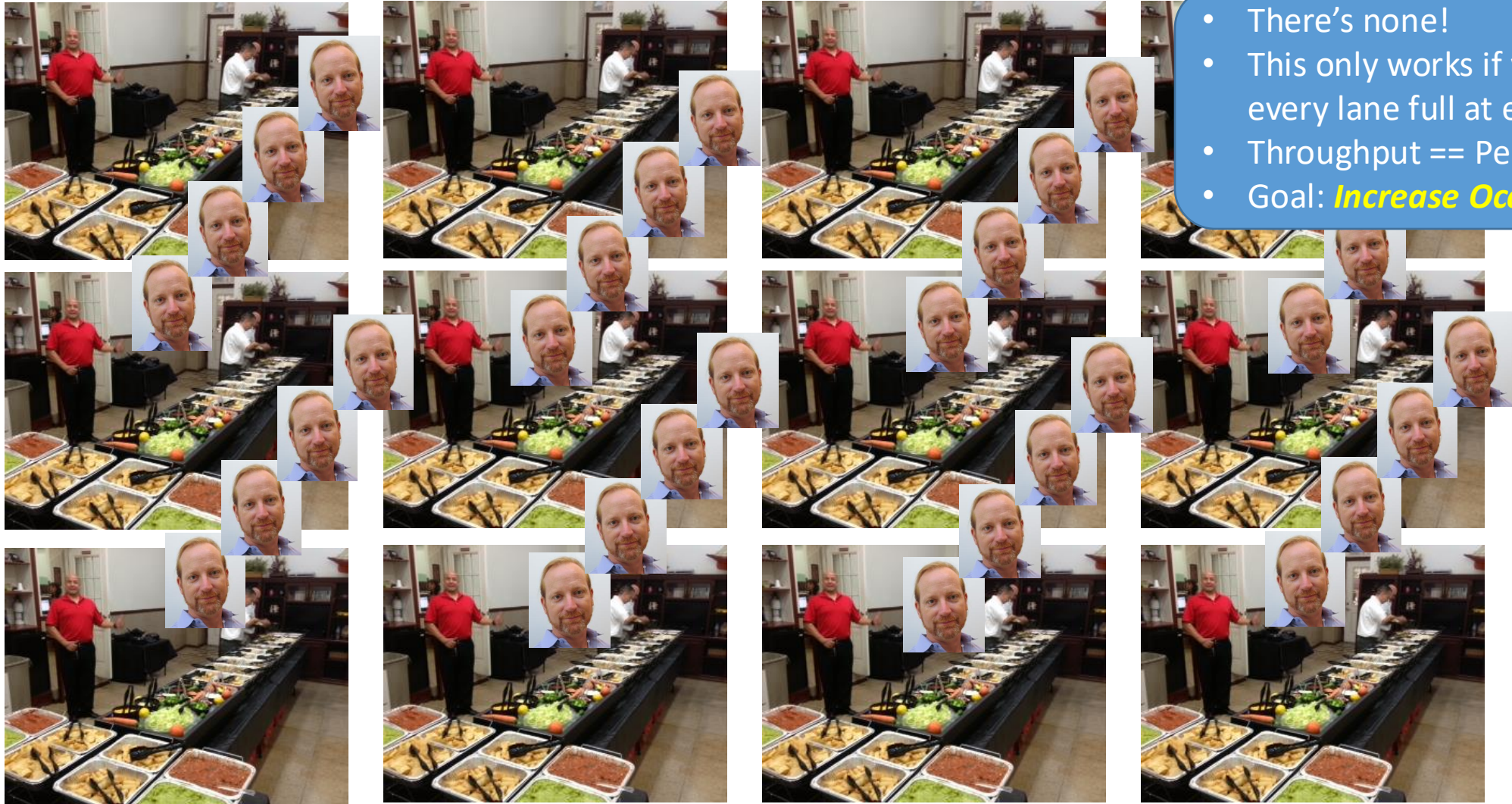

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

11

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

11

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy!*

11

Review: GPU Performance Metric: *Occupancy*

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
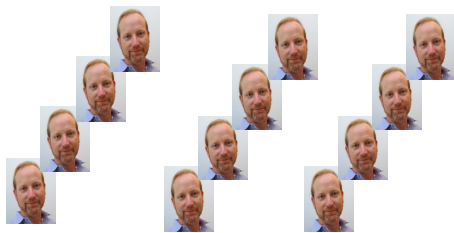 - Enables device-specific online tuning of kernel parameters

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
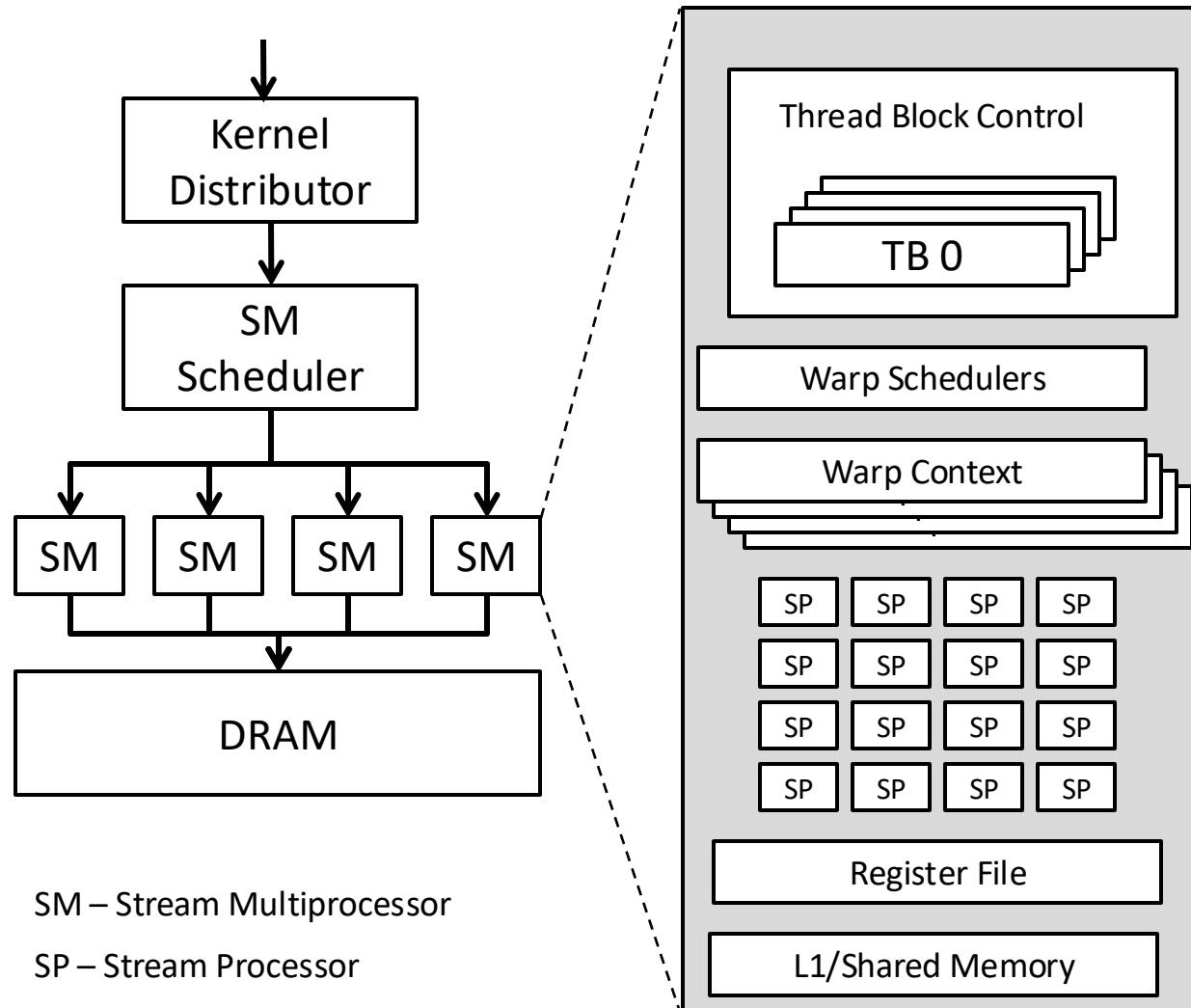 - Enables device-specific online tuning of kernel parameters

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
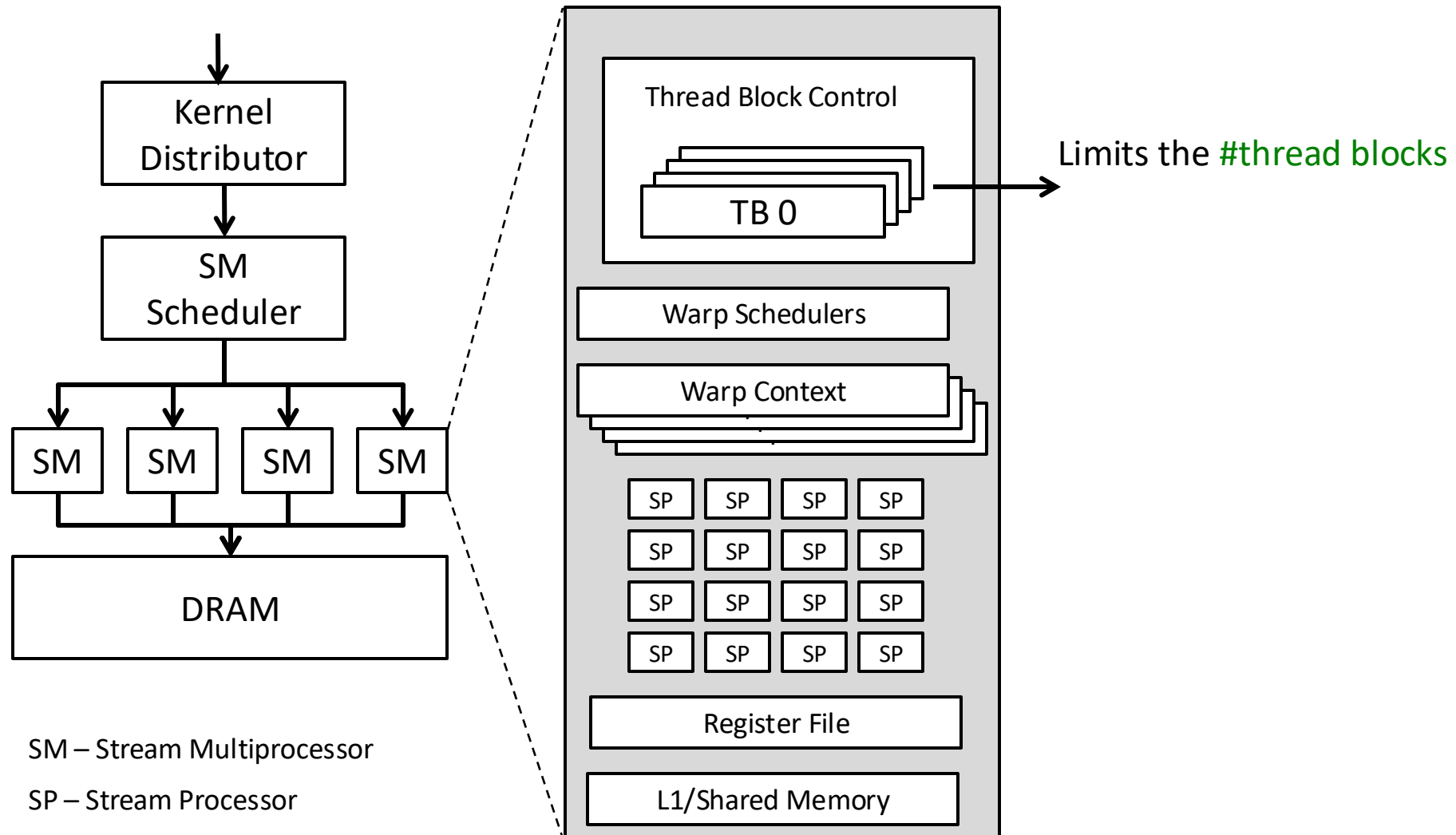 - Enables device-specific online tuning of kernel parameters

Review: GPU Performance Metric: *Occupancy*

- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

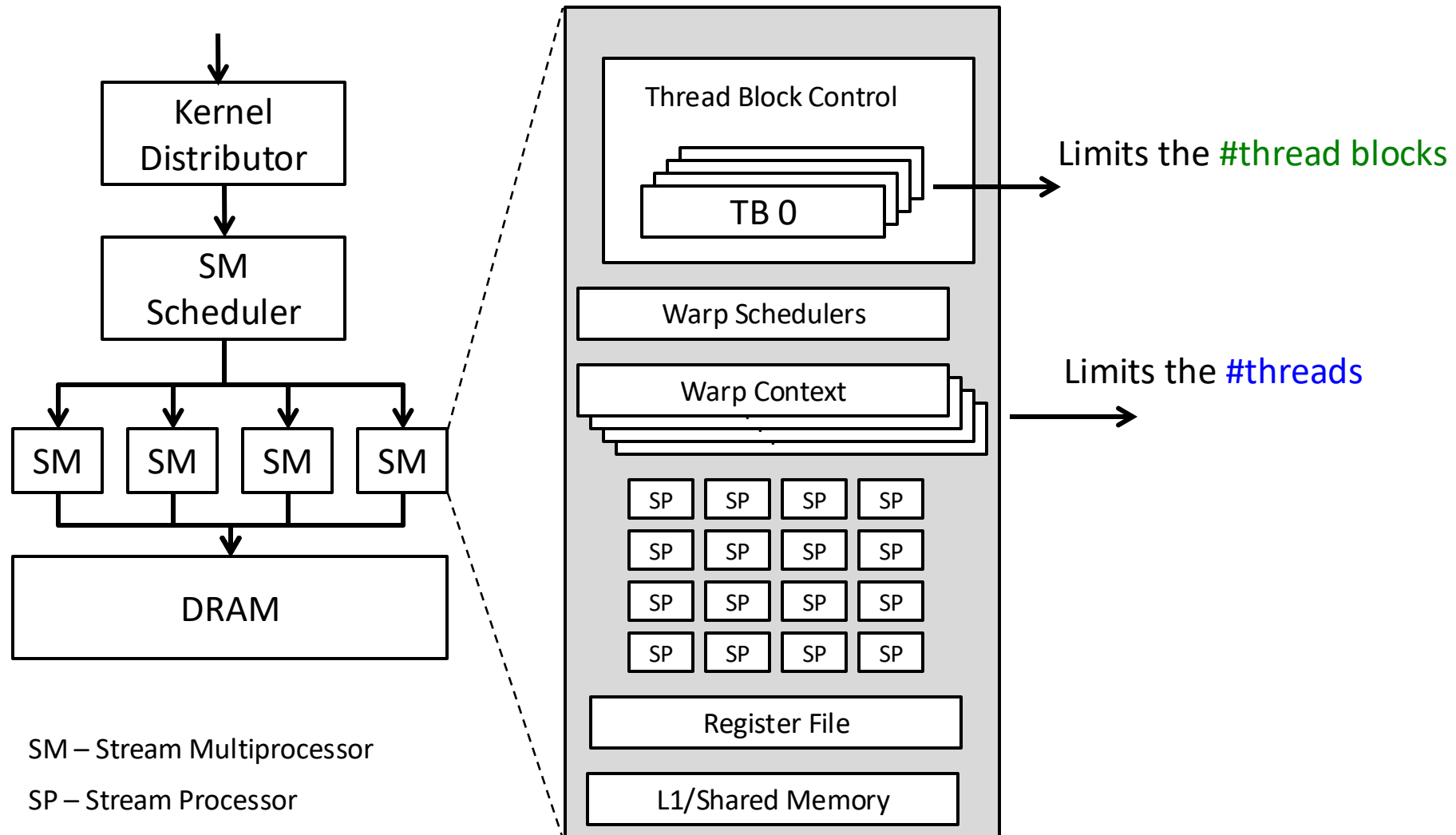Shouldn't we just create as many threads as possible?
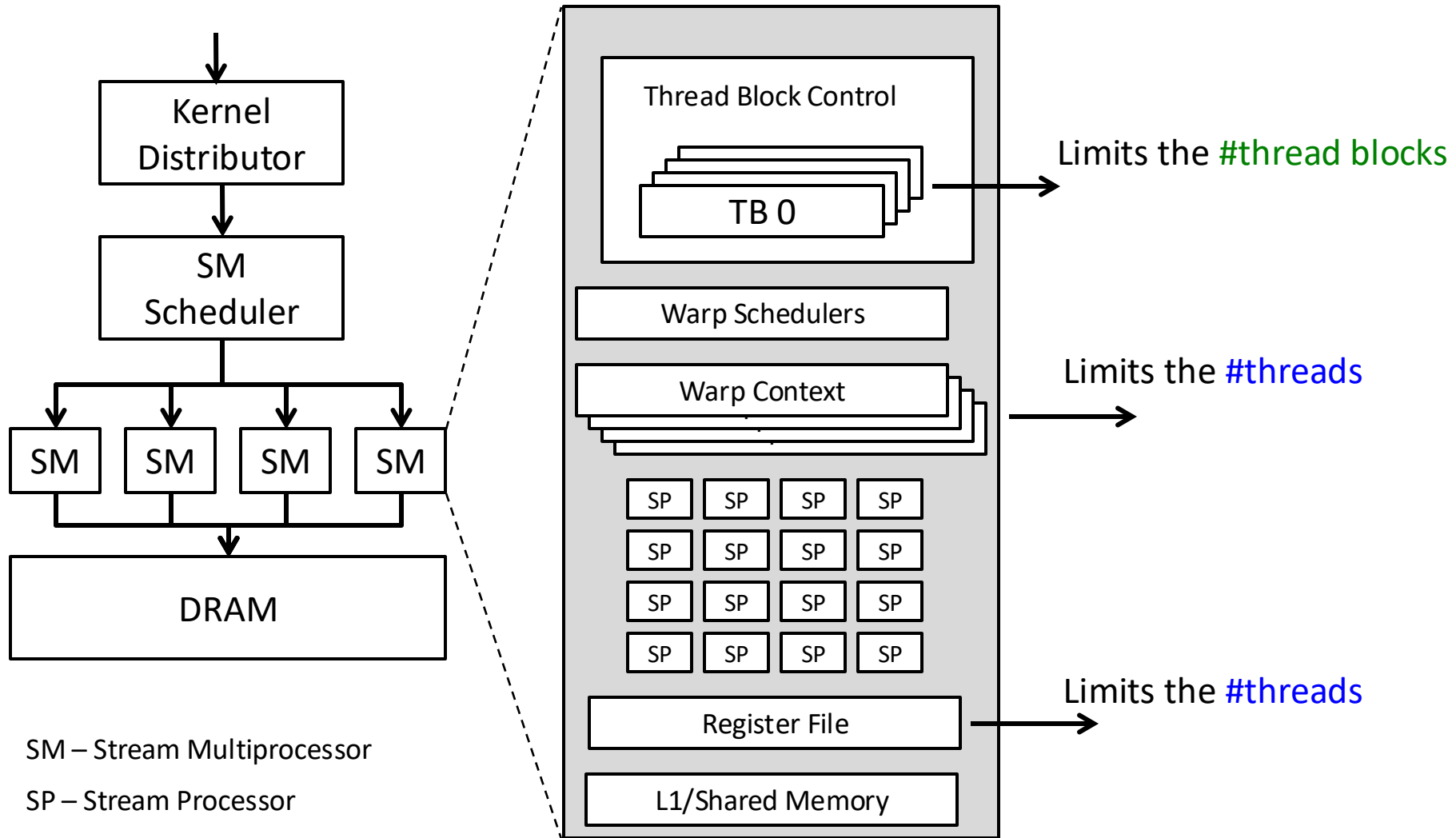
Hardware Resources Are Finite

Kernel Distributor

SM Scheduler

SM   SM   SM   SM

DRAM

SM – Stream Multiprocessor

SP – Stream Processor

Thread Block Control

TB 0

Warp Schedulers

Warp Context

SP SP SP SP
SP SP SP SP
SP SP SP SP
SP SP SP SP

Register File

L1/Shared Memory

Hardware Resources Are Finite

SM – Stream Multiprocessor

SP – Stream Processor

Hardware Resources Are Finite

SM – Stream Multiprocessor

SP – Stream Processor

Hardware Resources Are Finite

SM – Stream Multiprocessor

SP – Stream Processor

Hardware Resources Are Finite

Kernel Distributor

SM Scheduler

SM SM SM SM

DRAM

SM – Stream Multiprocessor

SP – Stream Processor

Thread Block Control

TB 0

Limits the #thread blocks

Warp Schedulers

Warp Context

Limits the #threads

SP SP SP SP
SP SP SP SP
SP SP SP SP
SP SP SP SP

Register File

Limits the #threads

L1/Shared Memory

Limits the #thread blocks

Hardware Resources Are Finite

Kernel Distributor

SM Scheduler

Thread Block Control

TB 0

Limits the #thread blocks

Warp Schedulers

Warp Context

Limits the #threads

SP SP SP SP
SP SP SP SP
SP SP SP SP
SP SP SP SP

Register File

Limits the #threads

L1/Shared Memory

Limits the #thread blocks

Occupancy:

- (#Active Warps) /(#MaximumActive Warps)

- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps

- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots

Hardware Resources Are Finite
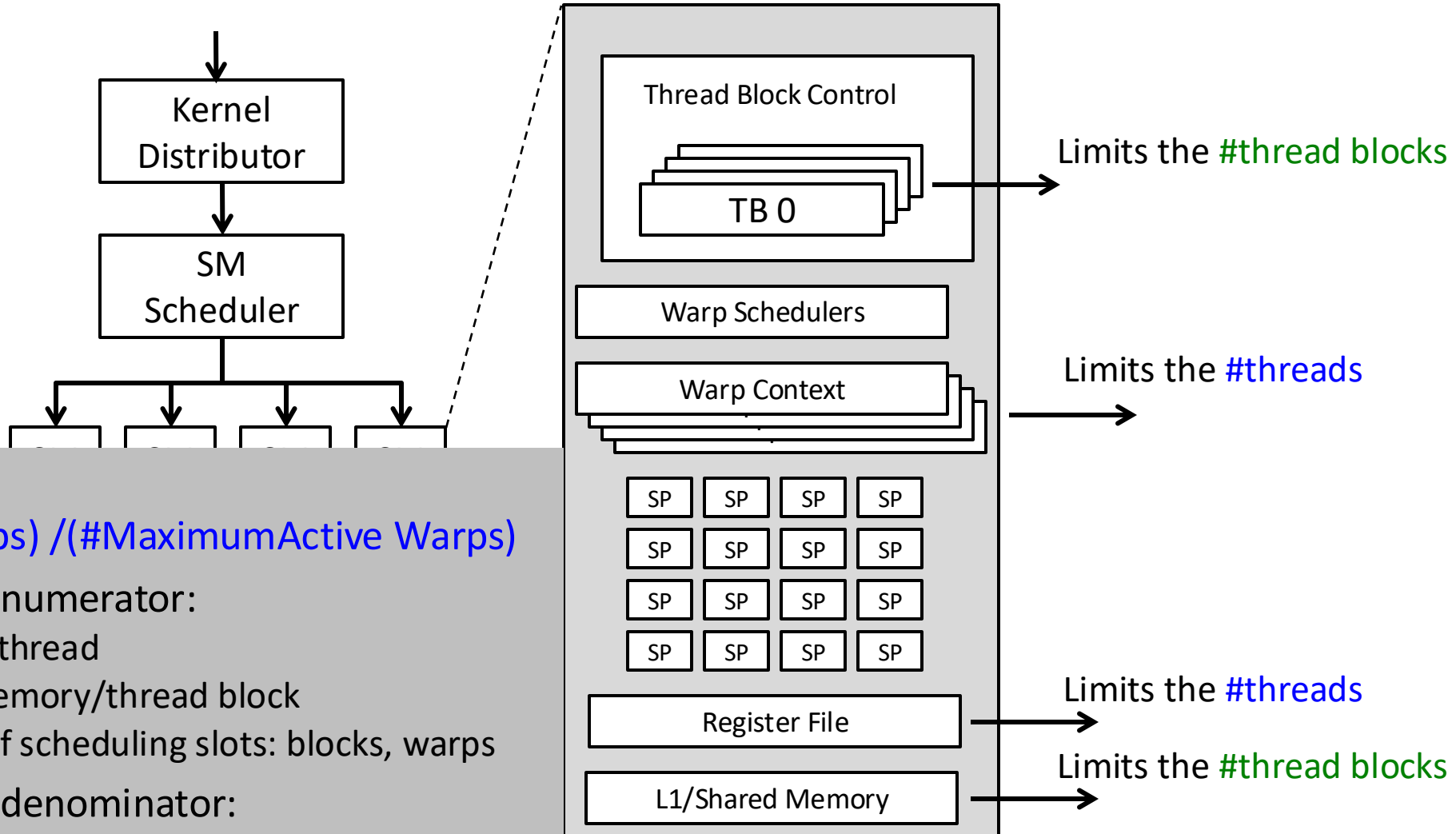
Kernel Distributor

SM Scheduler

Thread Block Control

TB 0

Limits the #thread blocks

Warp Schedulers

Warp Context

Limits the #threads

SP SP SP SP
SP SP SP SP
SP SP SP SP
SP SP SP SP

Register File

Limits the #threads

L1/Shared Memory

Limits the #thread blocks

Occupancy:

- **(#Active Warps) /(#MaximumActive Warps)**

- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps
- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots

What is the performance impact of varying kernel resource demands?

Impact of Thread Block Size
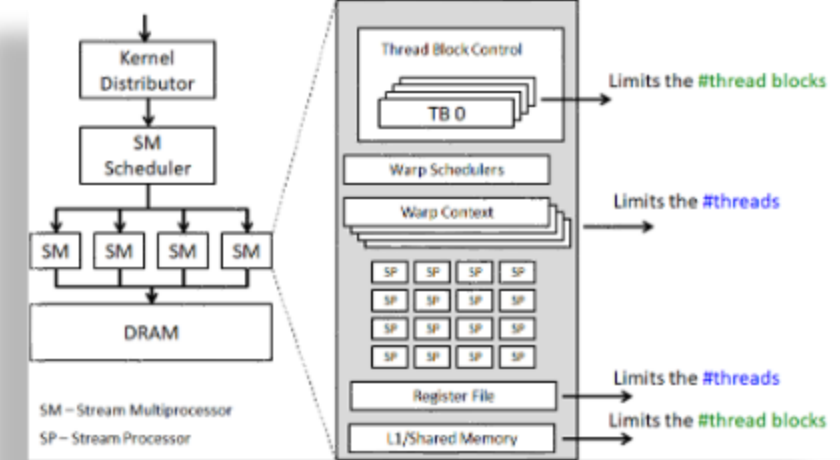
Impact of Thread Block Size
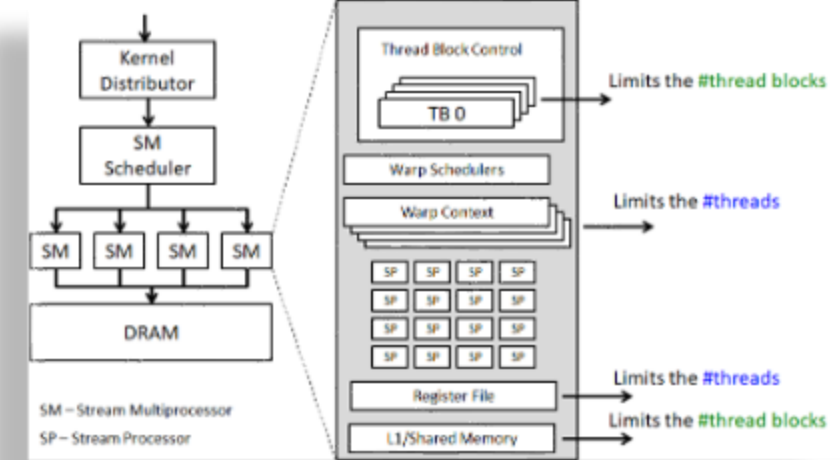
Example: v100:

Impact of Thread Block Size
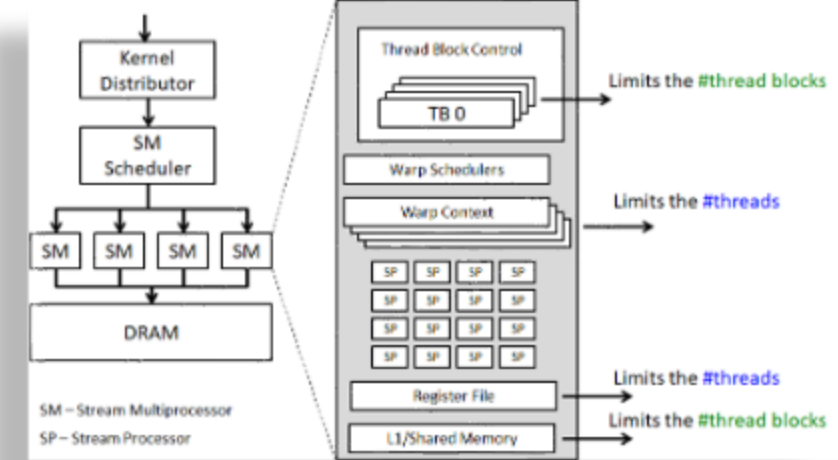
Example: v100:
- max active warps/SM == 64 (limit: warp context)

Impact of Thread Block Size

Example: v100:
- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
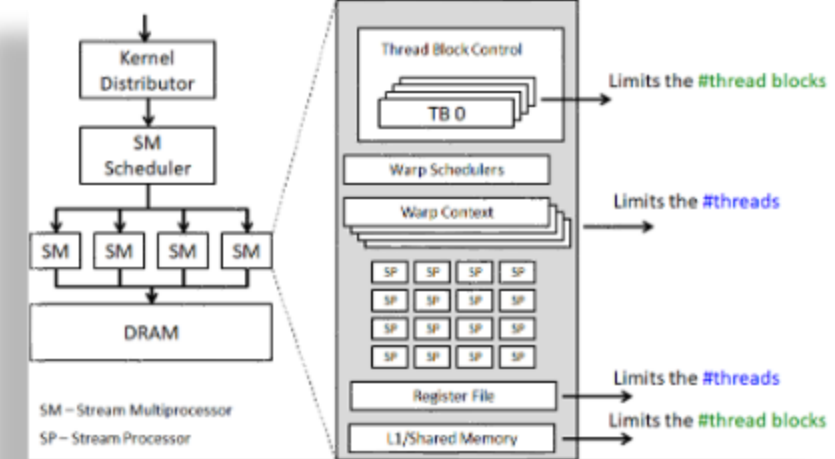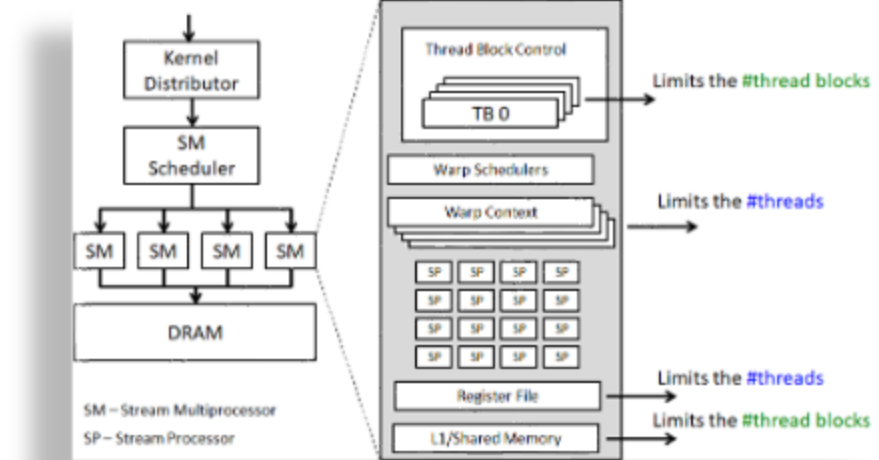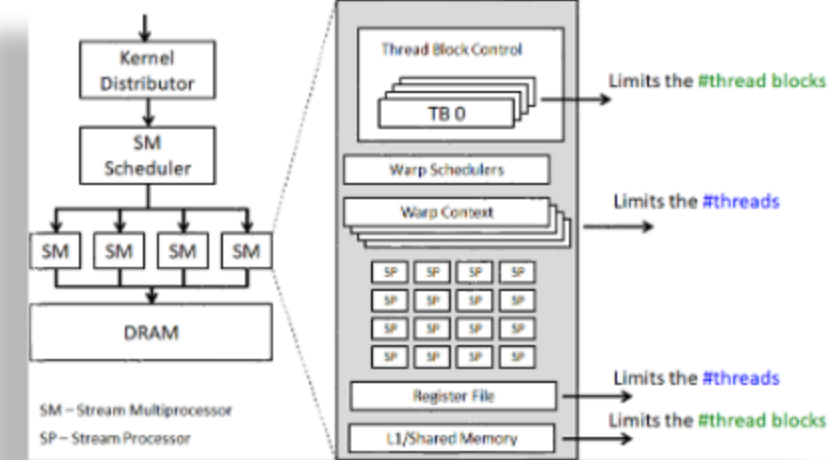
Impact of Thread Block Size
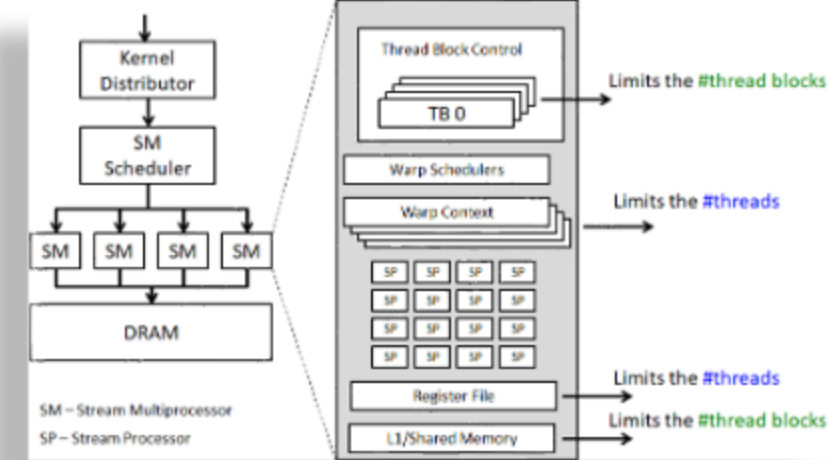

Example: v100:
- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads →
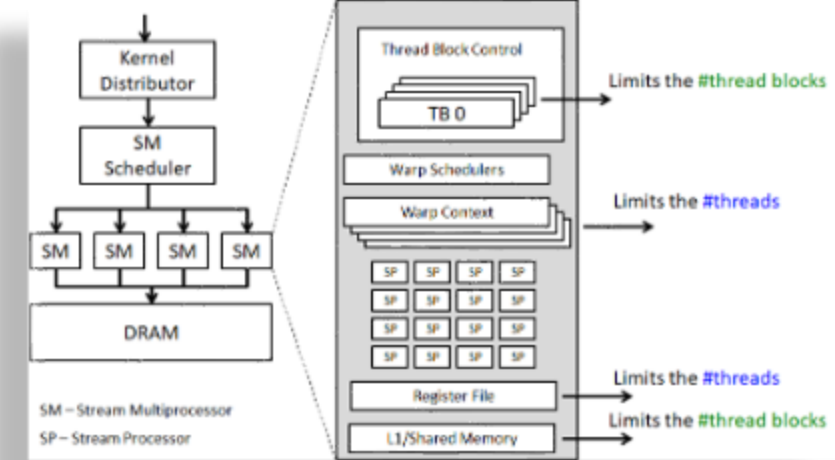
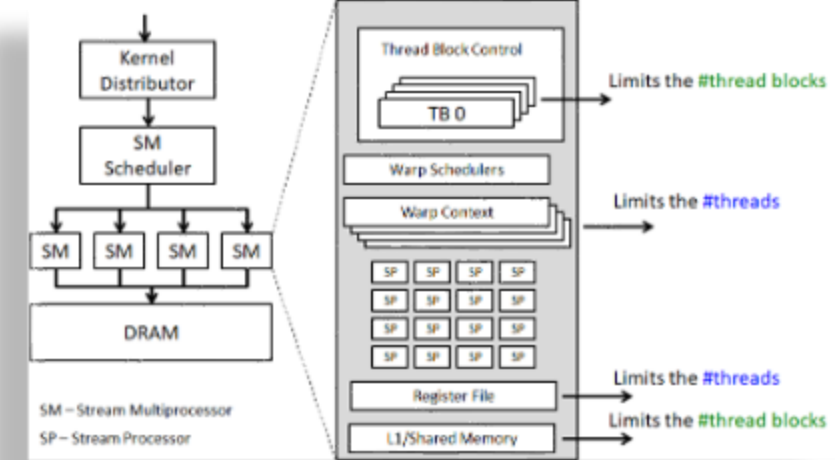# Impact of Thread Block Size

Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4

Impact of Thread Block Size


Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
  - With 128 threads/block? →

Impact of Thread Block Size

Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
 - With 128 threads/block? → 16

Impact of Thread Block Size

Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
  - With 128 threads/block? → 16
- Consider HW limit of 32 thread blocks/SM @ 32 threads/block:
  - Blocks are maxed out, but max active threads = 32*32 = 1024
  - Occupancy = .5 (1024/2048)

Impact of Thread Block Size

Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
  - With 128 threads/block? → 16
- Consider HW limit of 32 thread blocks/SM @ 32 threads/block:
  - Blocks are maxed out, but max active threads = 32*32 = 1024
  - Occupancy = .5 (1024/2048)
- To maximize utilization, thread block size should balance
  - Limits on active thread blocks vs.
  - Limits on active warps
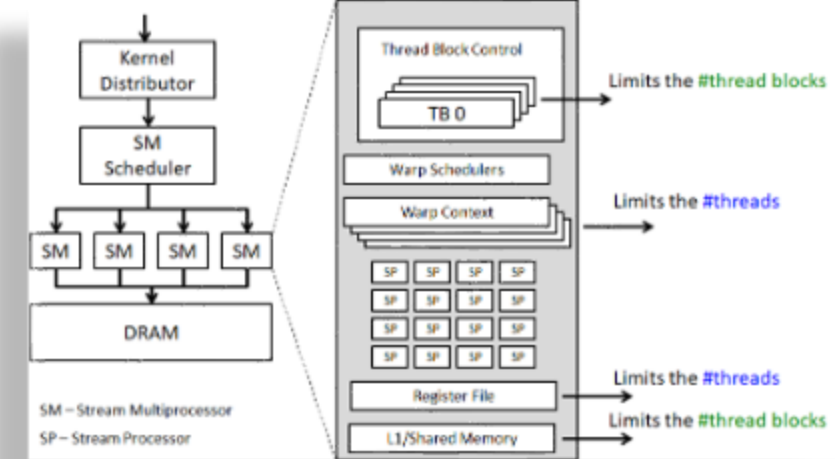
Impact of #Registers Per Thread
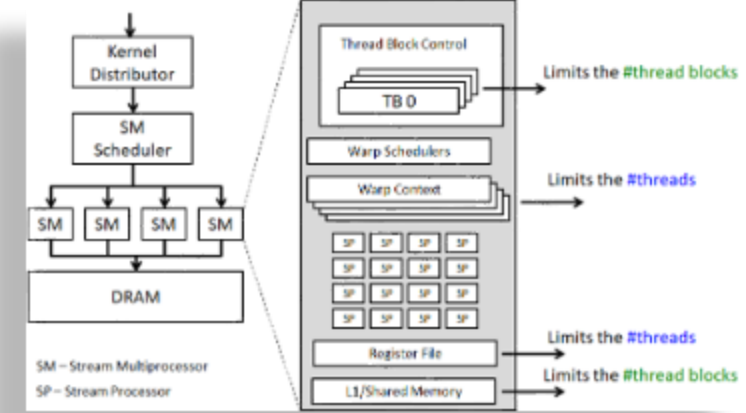
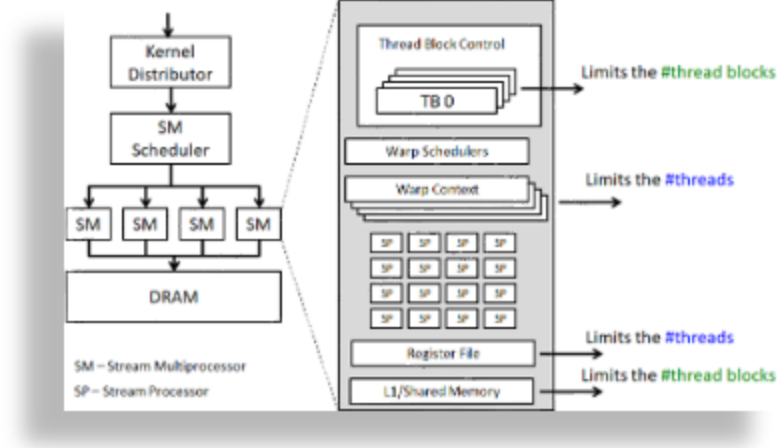# Impact of #Registers Per Thread

Registers/thread can limit number of active threads!
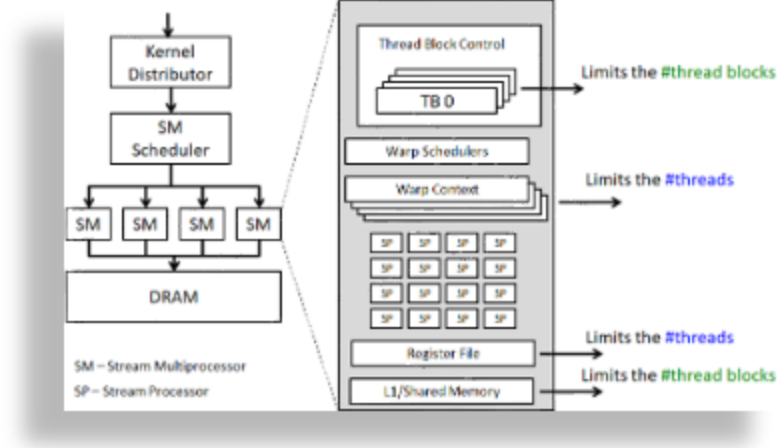
Impact of #Registers Per Thread

Registers/thread can limit number of active threads!
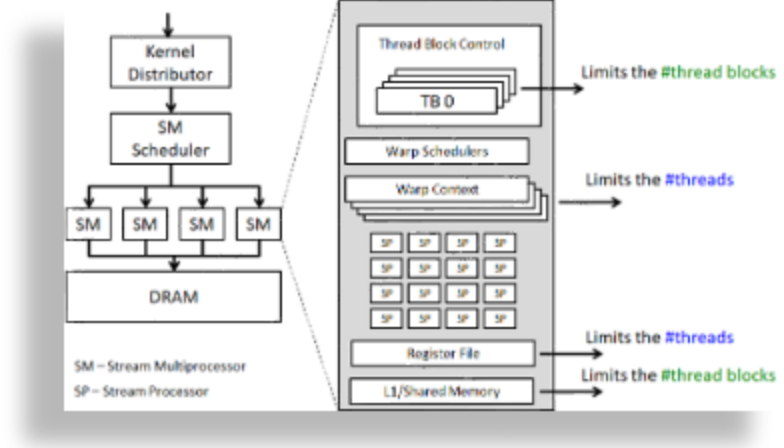
V100:

Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:
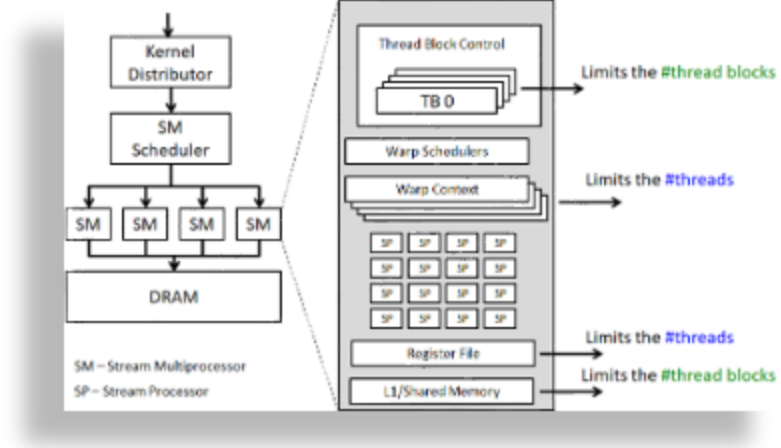
- Registers per thread max: 255

Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
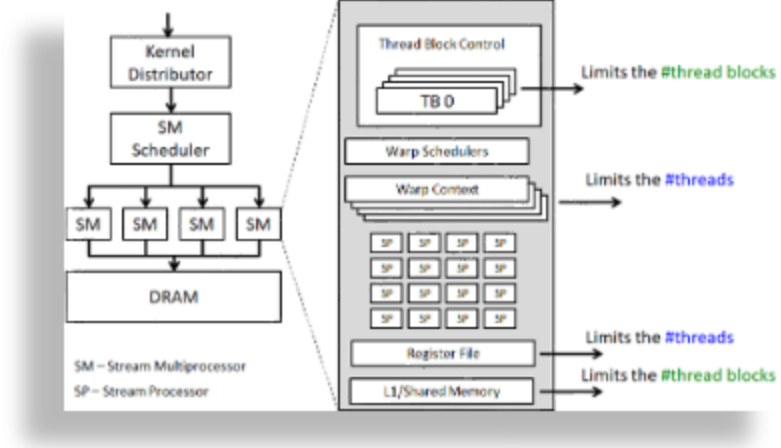- 64K registers per SM
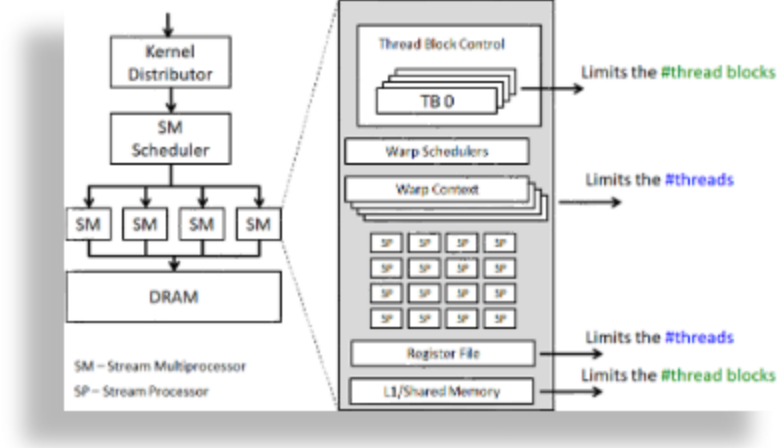
Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
- 64K registers per SM

Assume a kernel uses 32 registers/thread, thread block size of 256

Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
- 64K registers per SM

Assume a kernel uses 32 registers/thread, thread block size of 256

- Thus, A TB requires 8192 registers for a maximum of 8 thread blocks per SM
  - Uses all 2048 thread slots (8 blocks * 256 threads/block)
  - 8192 *regs/block * 8 block/SM = 64k registers*
  - *FULLY Occupied!*

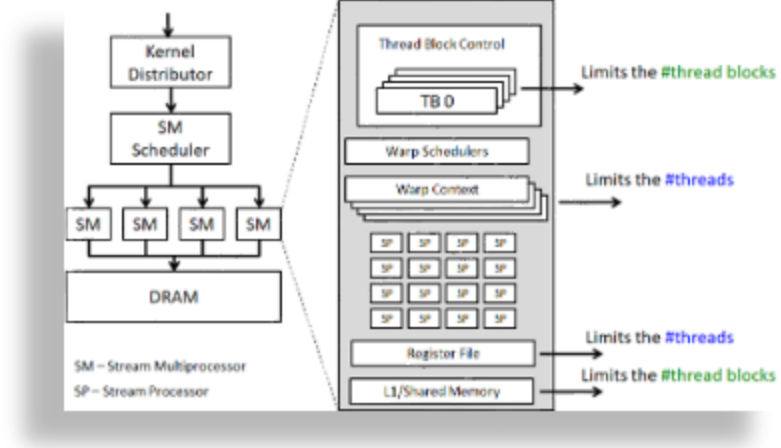# Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
- 64K registers per SM

Assume a kernel uses 32 registers/thread, thread block size of 256

- Thus, A TB requires 8192 registers for a maximum of 8 thread blocks per SM
  - Uses all 2048 thread slots (8 blocks * 256 threads/block)
  - 8192 *regs/block * 8 block/SM = 64k registers*
  - *FULLY Occupied!*
- What is the impact of increasing number of registers by 2?

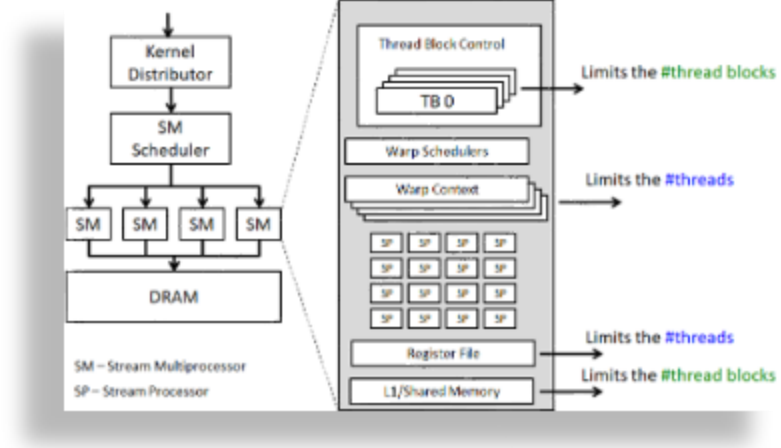# Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255

- 64K registers per SM

Assume a kernel uses 32 registers/thread, thread block size of 256

- Thus, A TB requires 8192 registers for a maximum of 8 thread blocks per SM
  - Uses all 2048 thread slots (8 blocks * 256 threads/block)
  - 8192 *regs/block * 8 block/SM = 64k registers*
  - *FULLY Occupied!*
- What is the impact of increasing number of registers by 2?
  - Recall: granularity of management is a thread block!

# Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
- 64K registers per SM

Assume a kernel uses 32 registers/thread, thread block size of 256

- Thus, A TB requires 8192 registers for a maximum of 8 thread blocks per SM
  - Uses all 2048 thread slots (8 blocks * 256 threads/block)
  - 8192 *regs/block * 8 block/SM = 64k registers*
  - *FULLY Occupied!*
- What is the impact of increasing number of registers by 2?
  - Recall: granularity of management is a thread block!
  - Loss of concurrency of 256 threads!
  - *34 regs/thread * 256 threads/block * 7 blocks/SM = 60k registers,*
  - *8 blocks would over-subscribe register file*
  - *Occupancy drops to .875!*
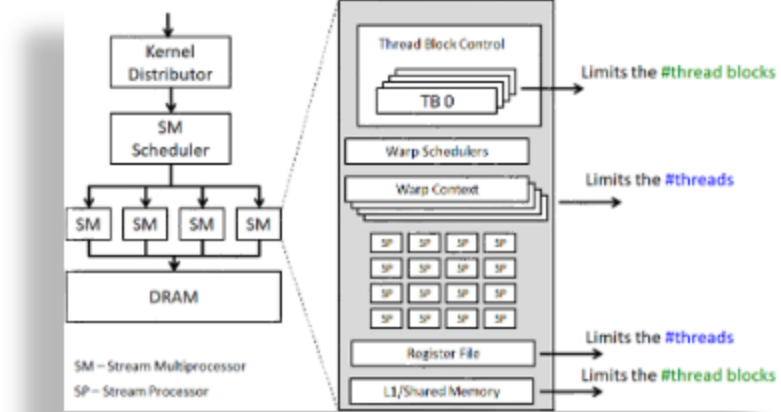
Impact of Shared Memory

- Shared memory is allocated per thread block
  - Can limit the number of thread blocks executing concurrently per SM
  - Shared mem/block * # blocks <= total shared mem per SM
- gridDim and blockDim parameters impact demand for
  - shared memory
  - number of thread slots
  - number of thread block slots

Balance
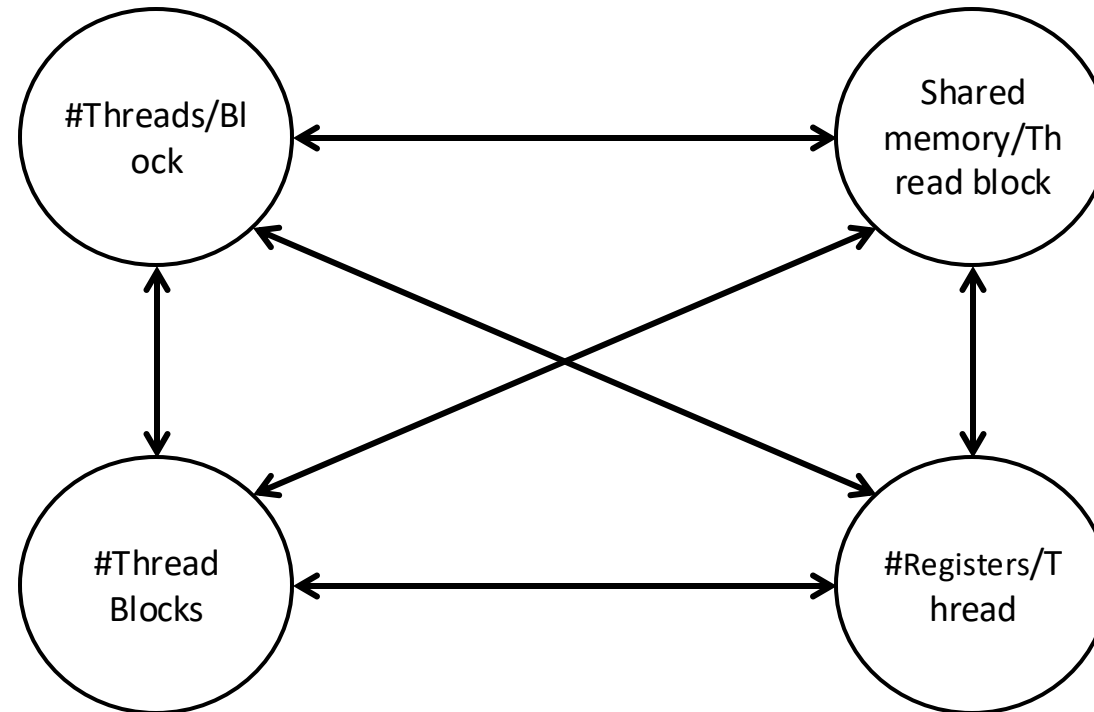
- Navigate the tradeoffs
  - ❖ maximize core utilization and memory bandwidth utilization
  - ❖ Device-specific
- Goal: Increase occupancy until one or the other is saturated

Balance

```
template < class T >
__host__ cudaError_t cudaOccupancyMaxActiveBlocksPerMultiprocessor ( int* numBlocks, T func, int blockSize, size_t dynamicSMemSize ) [inline]
```

Returns occupancy for a device function.

Parameters

numBlocks
    - Returned occupancy
func
    - Kernel function for which occupancy is calulated
blockSize
    - Block size the kernel is intended to be launched with
dynamicSMemSize
   - Per-block dynamic shared memory usage intended, in bytes

- Navigate the tradeoffs
  - ❖ maximize core utilization and memory bandwidth utilization
  - ❖ Device-specific
- Goal: Increase occupancy until one or the other is saturated

Parallel Memory Accesses

- Coalesced main memory access (16/32x faster)
  - HW combines multiple warp memory accesses into a single coalesced access

- Bank-conflict-free shared memory access (16/32)
  - No alignment or contiguity requirements
   - CC 2.x+3.0 : 32 different banks + 1-word broadcast each

Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth

- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7

Bank 15
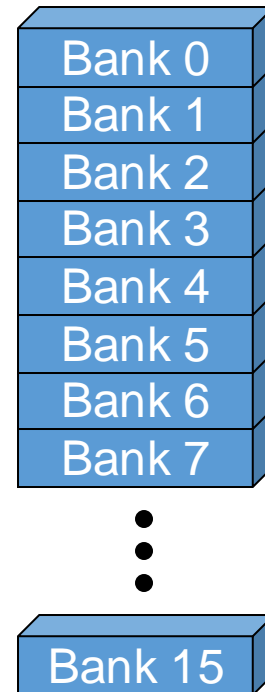
Coalesced Main Memory Accesses

single coalesced access
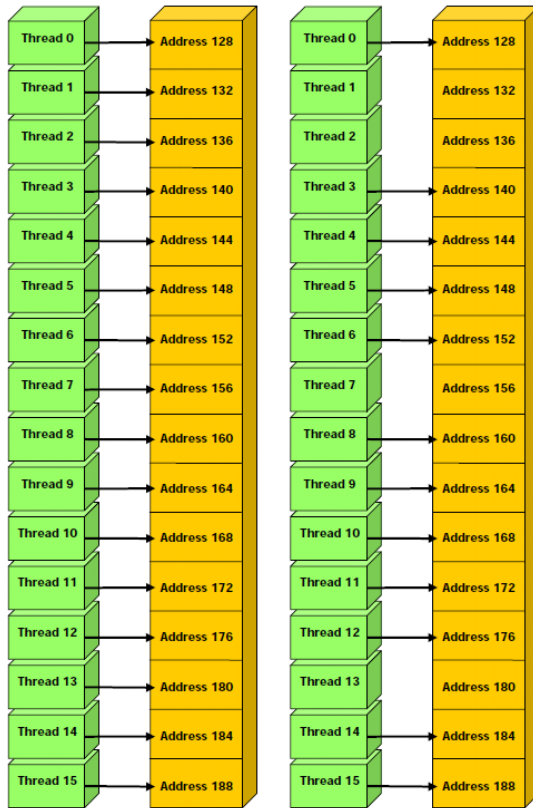
one and two coalesced accesses*

Bank Addressing Examples

No Bank Conflicts
- Linear addressing stride == 1

No Bank Conflicts
- Random 1:1 Permutation

Bank Addressing Examples

2-way Bank Conflicts
- Linear addressing stride == 2

8-way Bank Conflicts
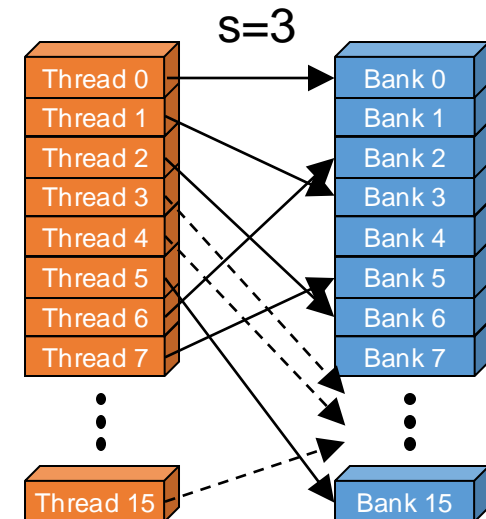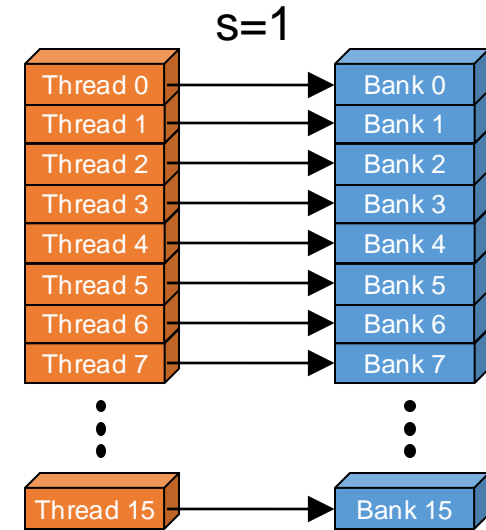- Linear addressing stride == 8
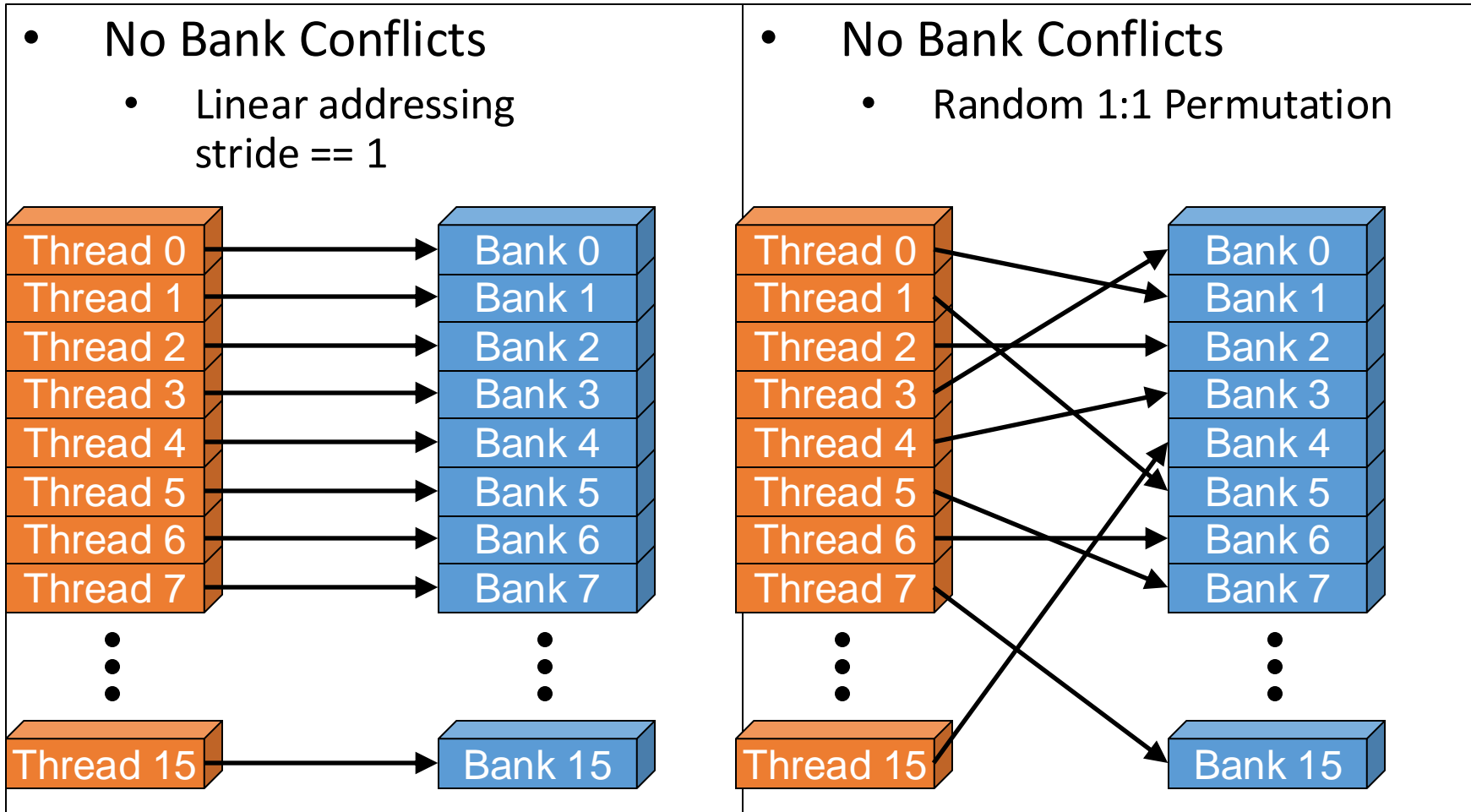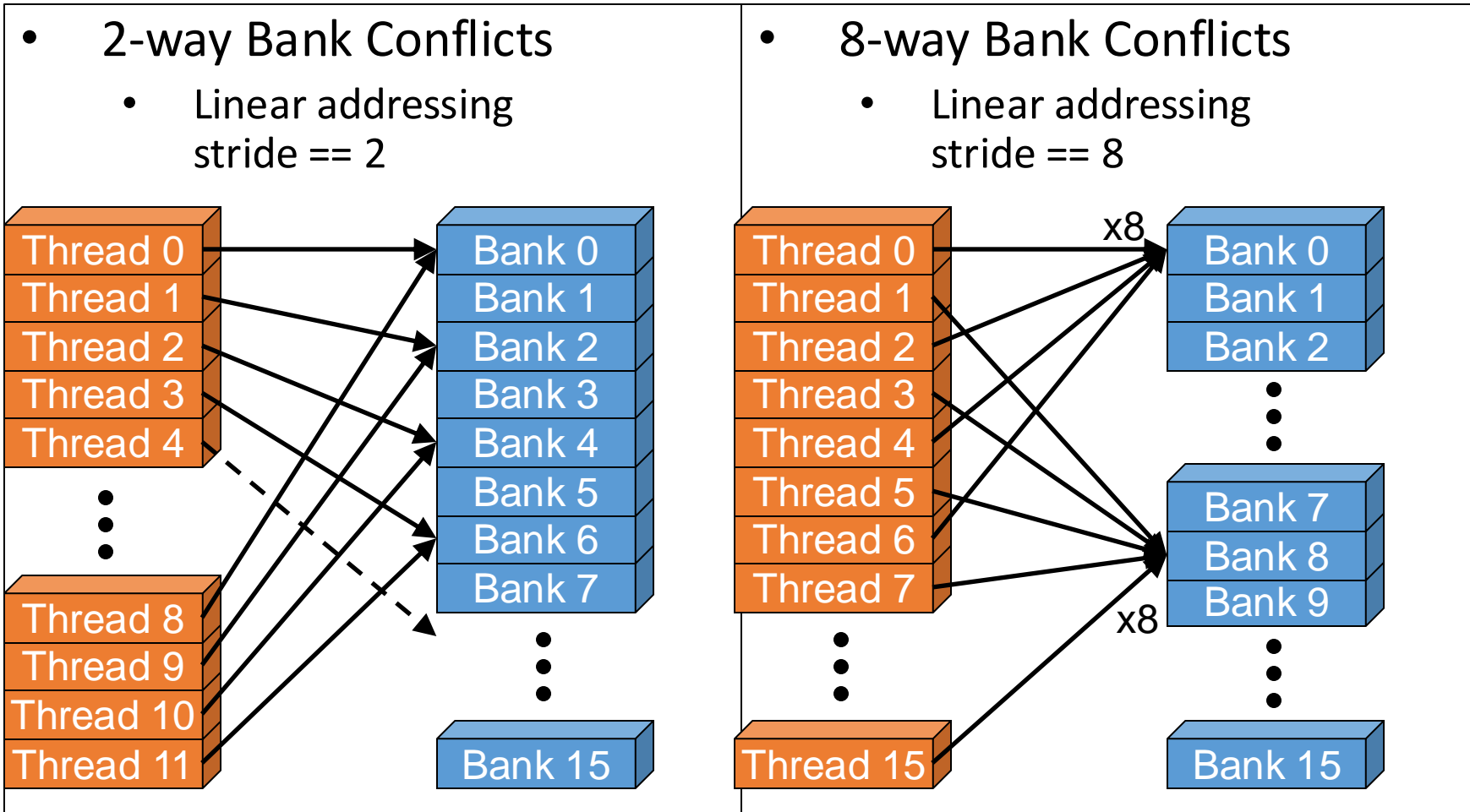
Linear Addressing

- Given:

```
__shared__ float shared[256];
float foo =
    shared[baseIndex + s *
    threadIdx.x];
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so s must be odd

23

GPU Atomics & Divergence

GPU Atomics & Divergence

Race conditions –
- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

```
atomicAdd()          atomicInc()
atomicSub()          atomicDec()
atomicMin()          atomicExch()
atomicMax()          atomicCAS()
```

Implemented as write-through to L2
- "Fire-and-forget"

GPU Atomics & Diverg

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –
- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

```
atomicAdd()              atomicInc()
atomicSub()              atomicDec()
atomicMin()              atomicExch()
atomicMax()              atomicCAS()
```

Implemented as write-through to L2
- "Fire-and-forget"
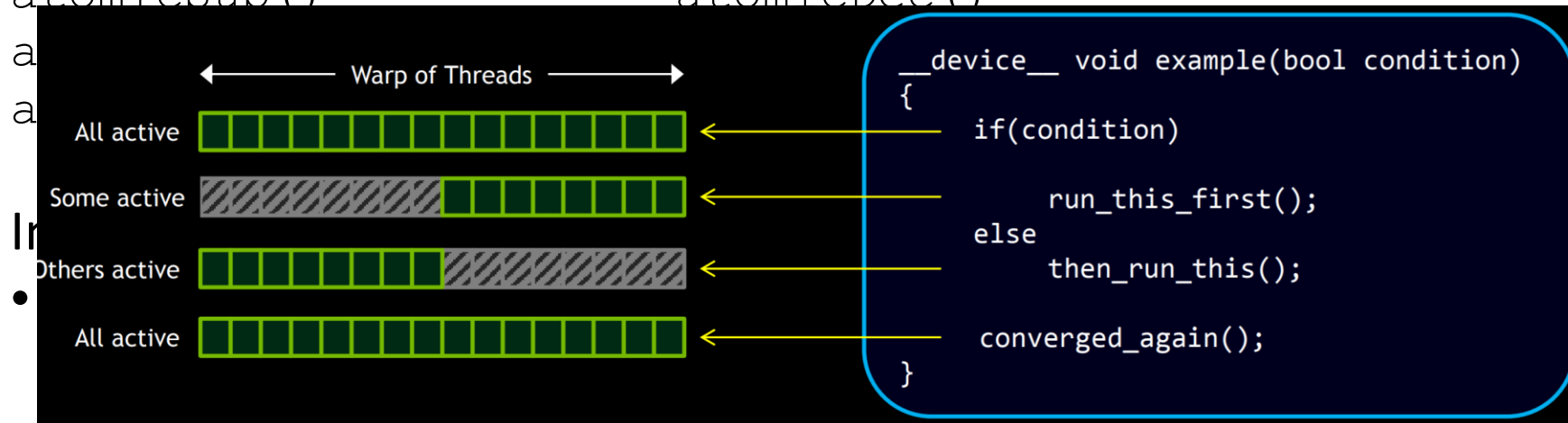
GPU Atomics & Diverg

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;       // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Is this a good idea?

Read-Modify-Write – atomic

```
atomicAdd()             atomicInc()
atomicSub()             atomicDec()
atomicMin()             atomicExch()
atomicMax()             atomicCAS()
```

Implemented as write-through to L2
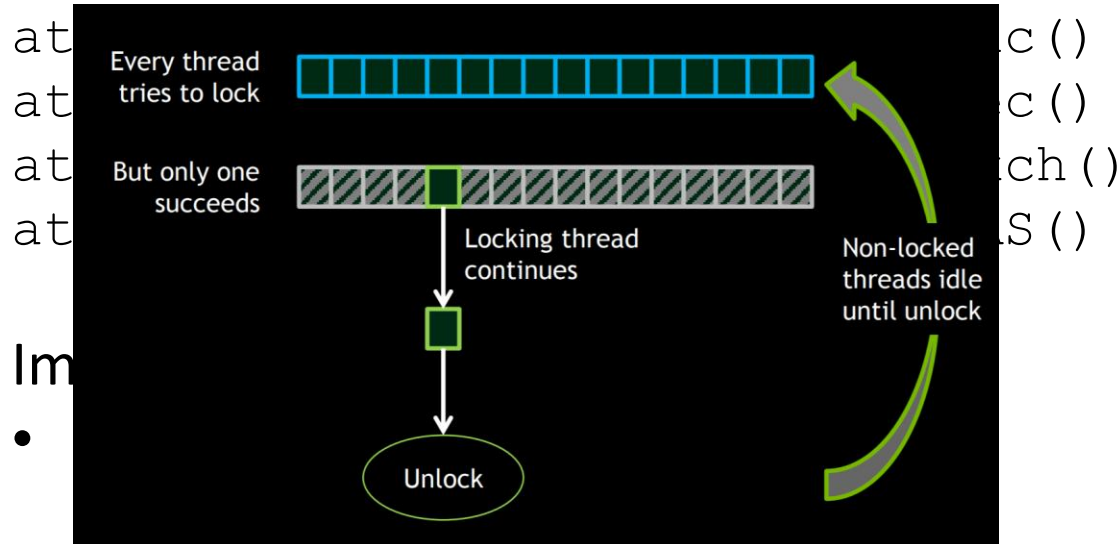
- "Fire-and-forget"

GPU Atomics & Diverg

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;      // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –
- Traditional locks: avoid!
- How do we synchronize?

Is this a good idea?

Read-Modify-Write – atomic

atomicAdd()   atomicInc()
atomicSub()   atomicDec()


```
__device__ void example(bool condition)
{
    if(condition)

        run_this_first();
    else
        then_run_this();

    converged_again();
}
```

GPU Atomics & Diverg

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;      // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –
- Traditional locks: avoid!
- How do we synchronize?

Is this a good idea?

Read-Modify-Write – atomic

```
atomicAdd()          atomicInc()
atomicSub()          atomicDec()
atomicMin()          atomicExch()
atomicMax()          atomicCAS()
```

Implemented as write-through to L2
- "Fire-and-forget"

GPU Atomics & Diverg...

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;     // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```
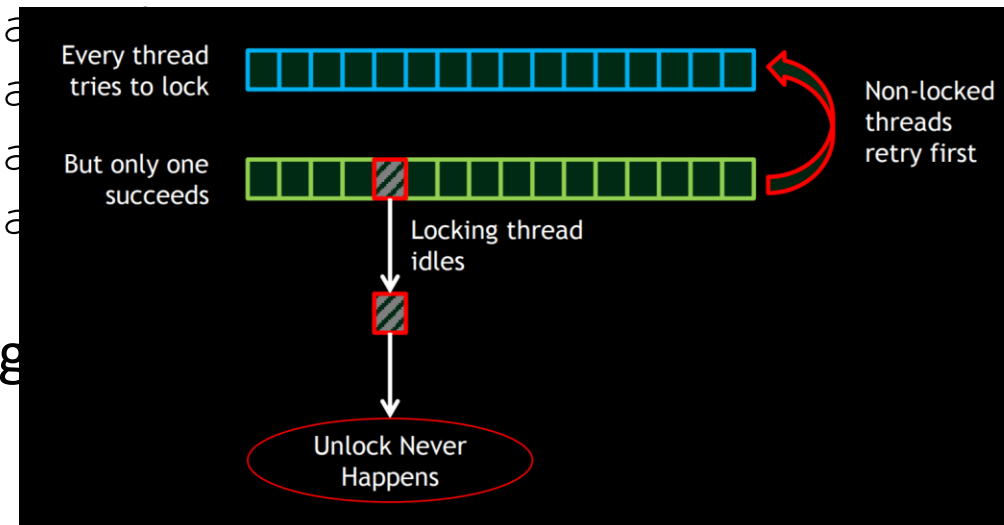
Is this a good idea?

atomic...c()
atomic...c()
atomic...ch()
atomic...S()

Im...

-

Every thread tries to lock

But only one succeeds

Locking thread continues

Non-locked threads idle until unlock

Unlock

GPU Atomics & Diverg

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;     // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

Is this a good idea?

atomicAdd()
atomicSub()
atomicMin()
atomicMax()

Implemented as write-throug

- "Fire-and-forget"

Every thread tries to lock

But only one succeeds

Non-locked threads retry first

Locking thread idles

Unlock Never Happens

Advanced Topic: GPU Programming Models

Layered abstractions

HW | CPU | I/O dev | DISK | NIC

Layered abstractions

Hardware
interface

HW

| CPU | I/O dev | DISK | NIC |

2/26/25

Layered abstractions

Applications

Hardware interface

HW

CPU   I/O dev   DISK   NIC

2/26/25
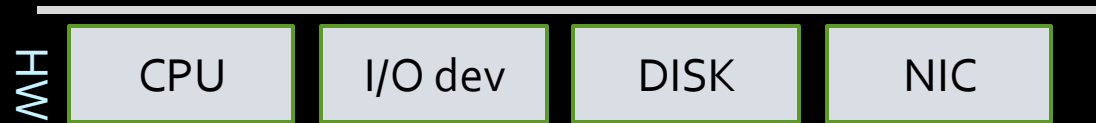
Layered abstractions

Layered abstractions

Applications

programmer-
visible interface

process · files · pipes

user

LIBC/CLR

user-mode
Runtimes/libs

OS interface

process · files · pipes

kernel

OS-level
abstractions

vendor driver · vendor driver · vendor driver

HAL

HW

CPU · I/O dev · DISK · NIC
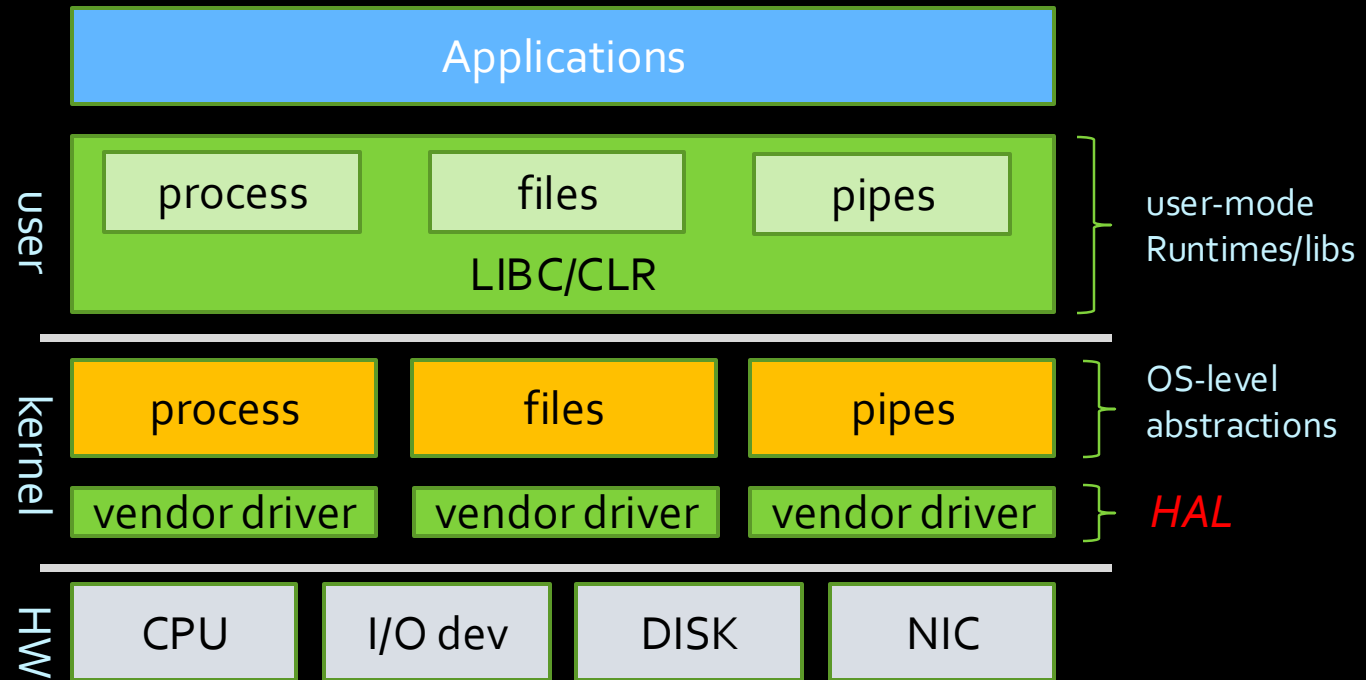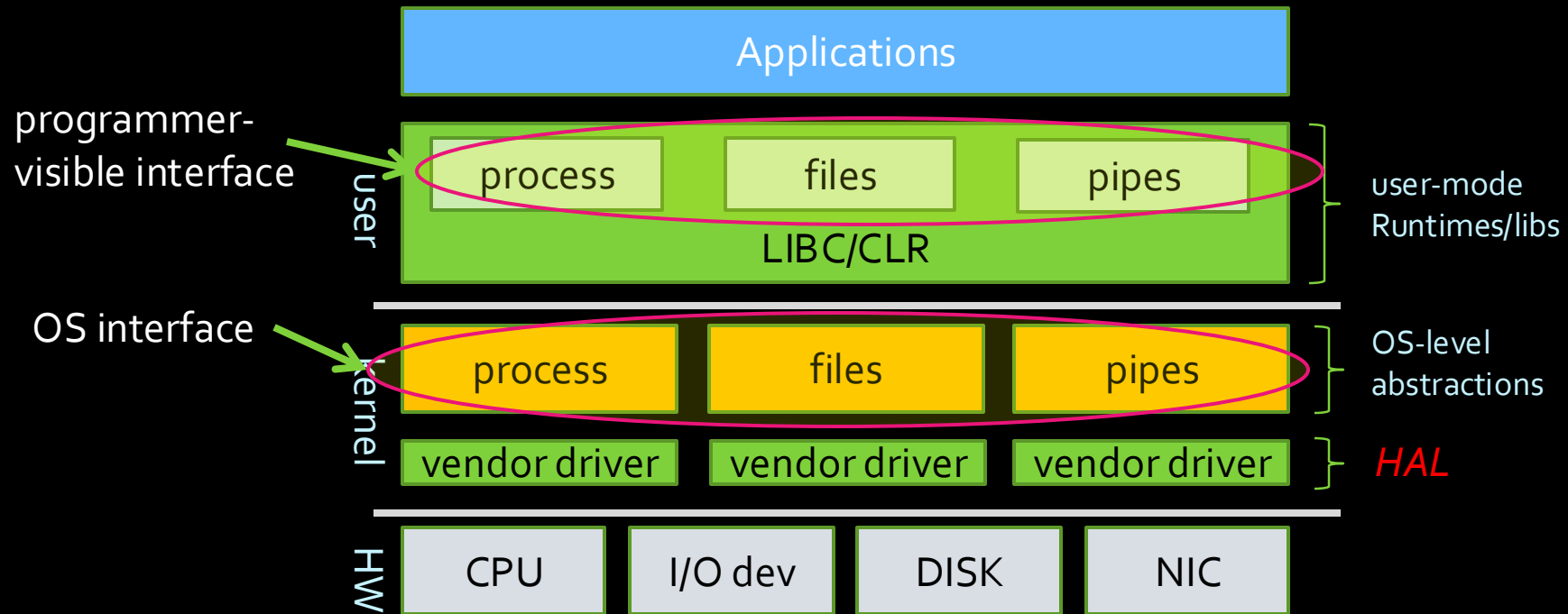
* 1:1 correspondence between OS-level and user-level abstractions
* Diverse HW support enabled HAL

2/26/25

GPU abstractions

GPU

HW

GPU abstractions

Hardware
interface

HW

GPU

GPU abstractions

Applications

user

GPGPU APIs | shaders kernels | language integration

GPU Runtime (e.g. OpenCL)

kernel

ioctl

Vendor-specific
driver

HW

GPU

Runtime support

2/26/25

GPU abstractions

Applications

programmer-
visible interface

| GPGPU APIs | shaders kernels | language integration |

GPU Runtime (e.g. OpenCL)

user

ioctl

kernel

Vendor-specific
driver

HW

GPU

Runtime
support

GPU abstractions

Applications

programmer-visible interface

GPGPU APIs

shaders kernels

language integration

user

GPU Runtime (e.g. OpenCL)

1 OS-level abstraction!

ioctl

mmap

kernel

Vendor-specific

driver

Fat driver, proprietary interfaces

HW

GPU

Runtime support

GPU abstractions

Applications

programmer-
visible interface

user

GPGPU
APIs | shaders
kernels | language
integration

GPU Runtime (e.g. OpenCL)

1 OS-level
abstraction!

kernel

ioctl | mmap

Vendor-specific

driver

Fat driver,
proprietary
interfaces

HW

GPU

Runtime
support

1. No kernel-facing API
2. OS resource-management limited
3. *Poor composability*

2/26/25

No OS support → No isolation

GPU benchmark throughput

invocations per second

value
1200
1000
800
600
400
200
0

no CPU load   high CPU load

Higher is better
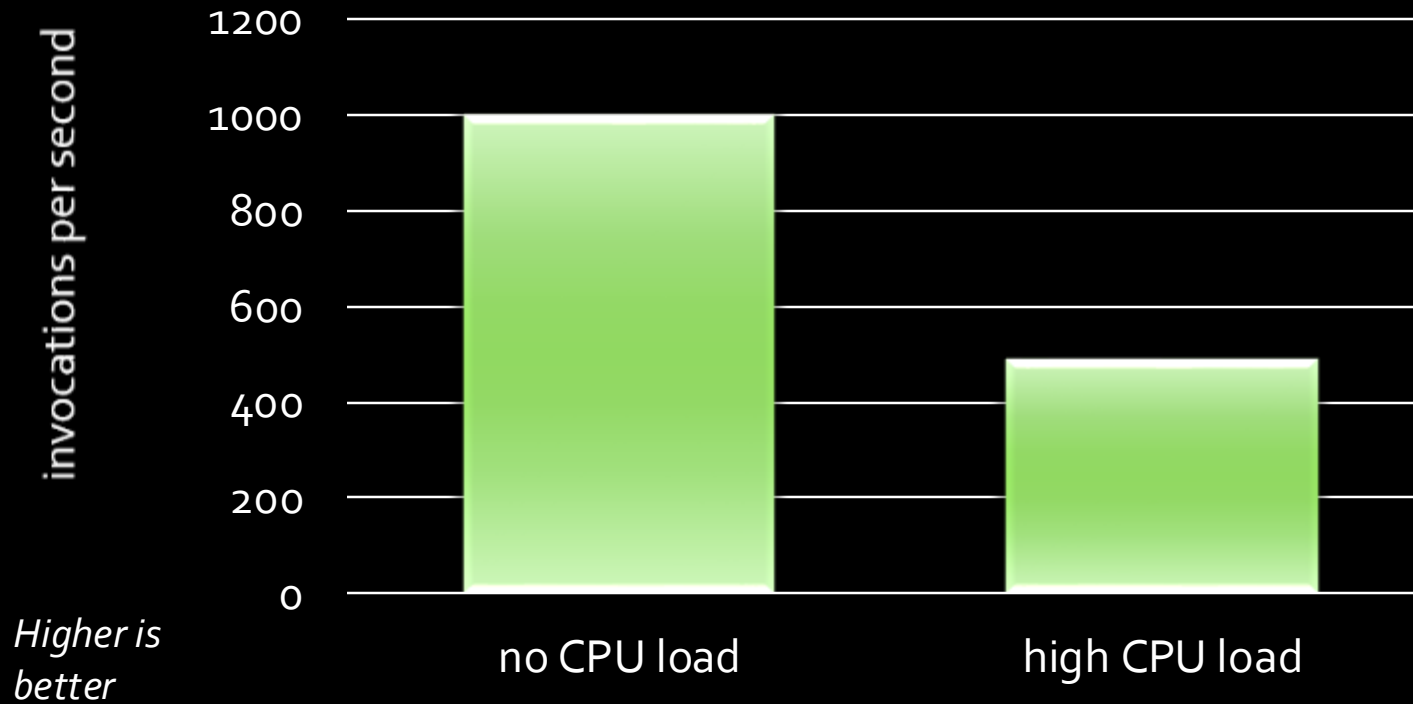
- Image-convolution in CUDA
- Windows 7 x64 8GB RAM
- Intel Core 2 Quad 2.66GHz
- nVidia GeForce GT230

2/26/25

No OS support → No isolation

GPU benchmark throughput

invocations per second

1200
1000
800
600
400
200
0

no CPU load   high CPU load

Higher is better

ge-convolution in CUDA
dows 7 x64 8GB RAM
el Core 2 Quad 2.66GHz
dia GeForce GT230
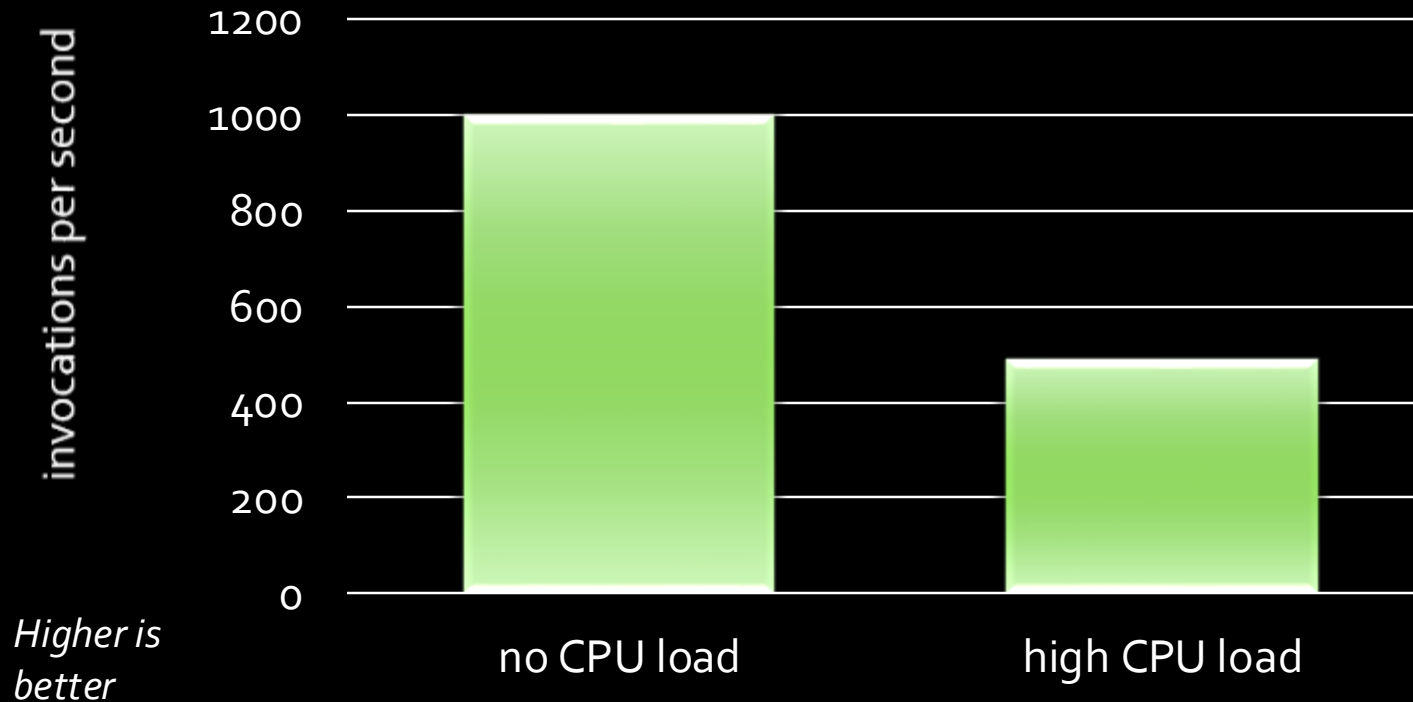
CPU+GPU schedulers not integrated! …other pathologies abundant

2/26/25

Composition: Gestural Interface

Raw images

capture

noisy point cloud

"Hand" events

detect

xform

filter

Composition: Gestural Interface

Raw images

noisy point cloud

"Hand" events

capture

capture camera images

xform

detect

filter

2/26/25

Composition: Gestural Interface

Raw images

capture

noisy point cloud

xform

filter

detect

"Hand" events

geometric transformation

2/26/25

Composition: Gestural Interface

Raw images

capture

noisy point cloud

xform

filter

"Hand" events

detect

noise filtering

Composition: Gestural Interface

Raw images

capture

noisy point cloud

"Hand" events

detect

detect gestures

xform

filter

Composition: Gestural Interface

Raw images

"Hand" events

capture

noisy point cloud

detect

xform

filter

Composition: Gestural Interface

Raw images

capture

noisy point cloud

"Hand" events

detect

xform

filter

▸ Requires OS mediation

▸ High data rates

▸ Abundant data parallelism
  ...use GPUs!

What We'd Like To Do

`#> capture | xform | filter | detect &`

- Modular design
  - flexibility, reuse
- Utilize heterogeneous hardware
  - Data-parallel components → GPU
  - Sequential components → CPU
- Using OS provided tools
  - processes, pipes

What We'd Like To Do

#> capture | xform | filter | detect &

    CPU   GPU   GPU   CPU

‣ Modular design
- ‣ flexibility, reuse

‣ Utilize heterogeneous hardware
- ‣ Data-parallel components → GPU
- ‣ Sequential components → CPU

‣ Using OS provided tools
- ‣ processes, pipes

GPU Execution model

- GPUs cannot run OS:
  - different ISA
  - Memories have different coherence guarantees
    - **(disjoint, or require fence instructions)**
- Host CPU must "manage" GPU execution
  - Program inputs explicitly transferred/bound at runtime
  - Device buffers pre-allocated

Main memory

CPU

Copy inputs

Copy outputs

Send commands

GPU memory

GPU

GPU Execution model

- GPUs cannot run OS:
  - different ISA
  - Memories have different coherence guarantees
    - **(disjoint, or require fence instructions)**
- Host CPU must "manage" GPU execution
  - Program inputs explicitly transferred/bound at runtime
  - Device buffers pre-allocated

Main memory

CPU

User-mode apps must implement

Copy inputs   Copy outputs   Send commands

GPU memory

GPU

Data migration

`#>` `capture` `|` `xform` `|` `filter` `|` `detect` `&`

user

capture	xform	filter	detect

OS executive

kernel

camdrv	GPU driver	HIDdrv

GPU

HW

Data migration

`#>` **capture** | **xform** | **filter** | **detect** **&**

user

capture

xform

filter

detect

kernel

OS executive

camdrv

GPU driver

HIDdrv

HW

GPU

2/26/25

Data migration

`#>` `capture` | `xform` | `filter` | `detect` `&`

user

capture

xform

filter

detect

read()

OS executive

kernel

camdrv

GPU driver

HIDdrv

HW

GPU

2/26/25

Data migration

user

capture

xform

filter

detect

read()

OS executive

kernel

camdrv

GPU driver

HIDdrv

HW

GPU

2/26/25

Data migration

`#> capture | xform | filter | detect &`

user

capture

xform

filter

detect

read()   write()

kernel

OS executive

camdrv

GPU driver

HIDdrv

HW

GPU

Data migration

`#>` `capture` | `xform` | `filter` | `detect` `&`

Data migration

`#> ` **capture** ` | ` **xform** ` | ` **filter** ` | ` **detect** ` &`

user

capture

xform

filter

detect

read()   write()   read()

copy
to
GPU

OS executive

kernel

camdrv

GPU driver

HIDdrv

HW

GPU

Data migration

Data migration

user

capture

xform

filter

detect

`read()`   `write()`   `read()`

kernel

copy from GPU

camdrv

GPU driver

HIDdrv

HW

copy-xfer

GPU

2/26/25

Data migration

`#>` `capture` `|` `xform` `|` `filter` `|` `detect` `&`

user

capture

xform

filter

detect

read()   write()   read()   write()

kernel

copy from GPU

camdrv

GPU driver

HIDdrv

copy-xfer

HW

GPU

2/26/25

Data migration

Data migration

`#>` `capture` `|` `xform` `|` `filter` `|` `detect` `&`

capture	xform	filter	detect

user

read() write() read() write() read()

copy
to
GPU

kernel

camdrv   GPU driver   HIDdrv

HW

GPU

Data migration

```
#> capture | xform | filter | detect &
```

Data migration

capture

xform

filter

detect

user

read()   write()   read()   write() read()

kernel

copy
from
GPU

camdrv   GPU driver   HIDdrv

copy-xfer

HW

GPU

2/26/25

Data migration

`#> ` `capture` `| ` `xform` `| ` `filter` `| ` `detect` `&`

capture	xform	filter	detect

user

read()　　write()　read()　　write() read()　　write()

kernel

copy
from
GPU

camdrv　　　　　　GPU driver　　　　　　　HIDdrv

copy-xfer

HW

GPU

2/26/25

Data migration

Data migration

Device-centric APIs considered harmful

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

Device-centric APIs considered harmful

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

What happens if I want the following?

Matrix D = A x B x C

Composed matrix multiplication

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
```

Composed matrix multiplication

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
```

Composed matrix multiplication

AxB copied from
GPU memory...

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
```

2/26/25

Composed matrix multiplication

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}
```

…only to be copied right back!

2/26/25

What if I have many GPUs?

```
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

What if I have many GPUs?

```
Matrix
gemm(GPU dev,Matrix A, Matrix B) {
    copyToGPU(dev, A);
    copyToGPU(dev, B);
    invokeGPU(dev);
    Matrix C = new Matrix();
    copyFromGPU(dev, C);
    return C;
}
```

What if I have many GPUs?

```
Matrix
gemm(GPU dev,Matrix A, Matrix B) {
    copyToGPU(dev, A);
    copyToGPU(dev, B);
    invokeGPU(dev);
    Matrix C = new Matrix();
    copyFromGPU(dev, C);
    return C;
}
```

What happens if I want the following?

Matrix D = A x B x C

Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(Matrix A,B,C) {
    Matrix AxB = gemm(???, A,B);
    Matrix AxBxC = gemm(???, AxB,C);
    return AxBxC;
}
```

Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(GPU dev, Matrix A,B,C) {
    Matrix AxB = gemm(dev, A,B);
    Matrix AxBxC = gemm(dev, AxB,C);
    return AxBxC;
}
```

Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

Rats...now I can only use 1 GPU.
How to partition computation?

```
Matrix
AxBxC(GPU dev, Matrix A,B,C) {
    Matrix AxB = gemm(dev, A,B);
    Matrix AxBxC = gemm(dev, AxB,C);
    return AxBxC;
}
```

Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

```
Matrix
AxBxC(GPU devA, GPU devB, Matrix A,B,C) {
    Matrix AxB = gemm(devA, A,B);
    Matrix AxBxC = gemm(devB, AxB,C);
    return AxBxC;
}
```

Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

This will never be manageable for *many* GPUs. *Programmer implements scheduling using static view!*

```
Matrix
AxBxC(GPU devA, GPU devB, Matrix A,B,C) {
    Matrix AxB = gemm(devA, A,B);
    Matrix AxBxC = gemm(devB, AxB,C);
    return AxBxC;
}
```

Composition with many GPUs

```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

This will never be manageable for many GPUs. Programmer implements scheduling using static view!

```
Matrix
AxBxC(GPU devA, GPU devB, Matrix A,B,C) {
    Matrix AxB = gemm(devA, A,B);
    Matrix AxBxC = gemm(devB, AxB,C);
    return AxBxC;
}
```

Why don't we have this problem with CPUs?

Dataflow: a better abstraction

Matrix: A

Matrix: B

gemm

Matrix: C

gemm
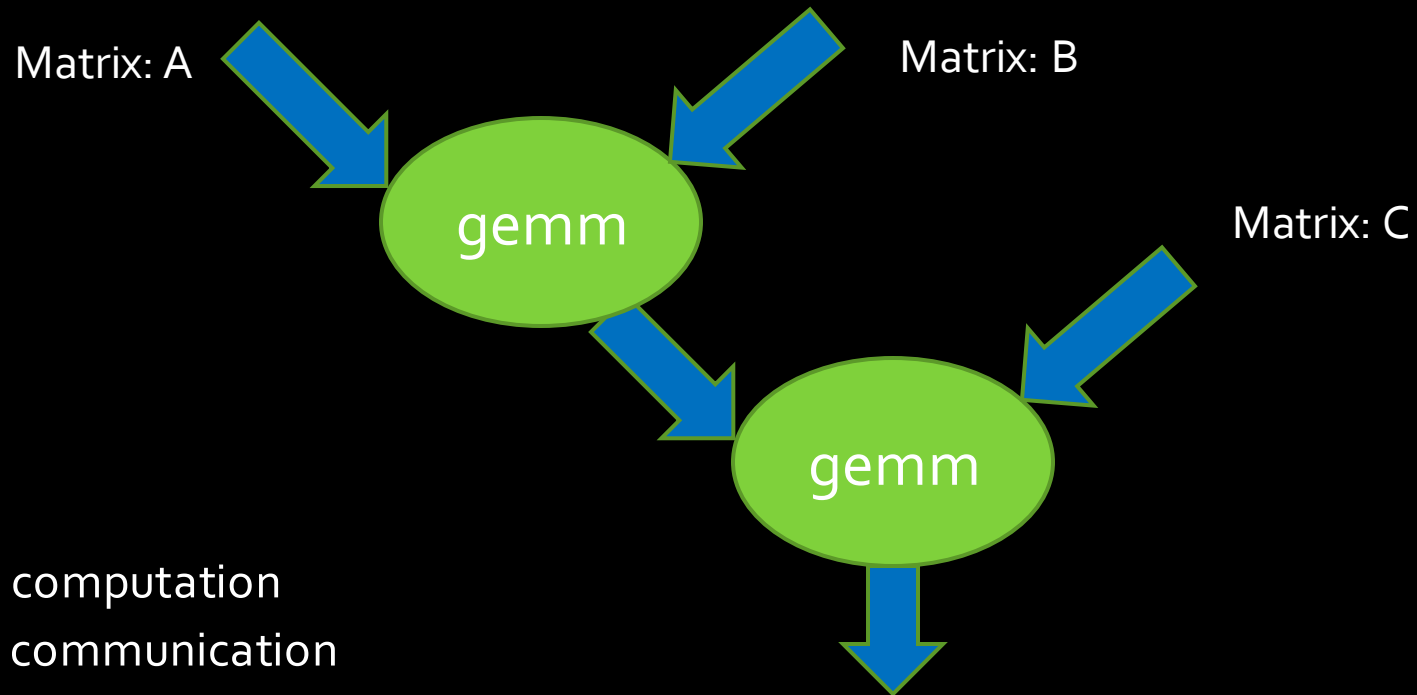
- nodes → computation
- edges → communication
- Expresses parallelism explicitly
- Minimal specification of data movement: runtime does it.
- asynchrony is a runtime concern (not programmer concern)
- No specification of compute→device mapping: like threads!

Advanced Topic: GPU Coherence

Review: Cache Coherence
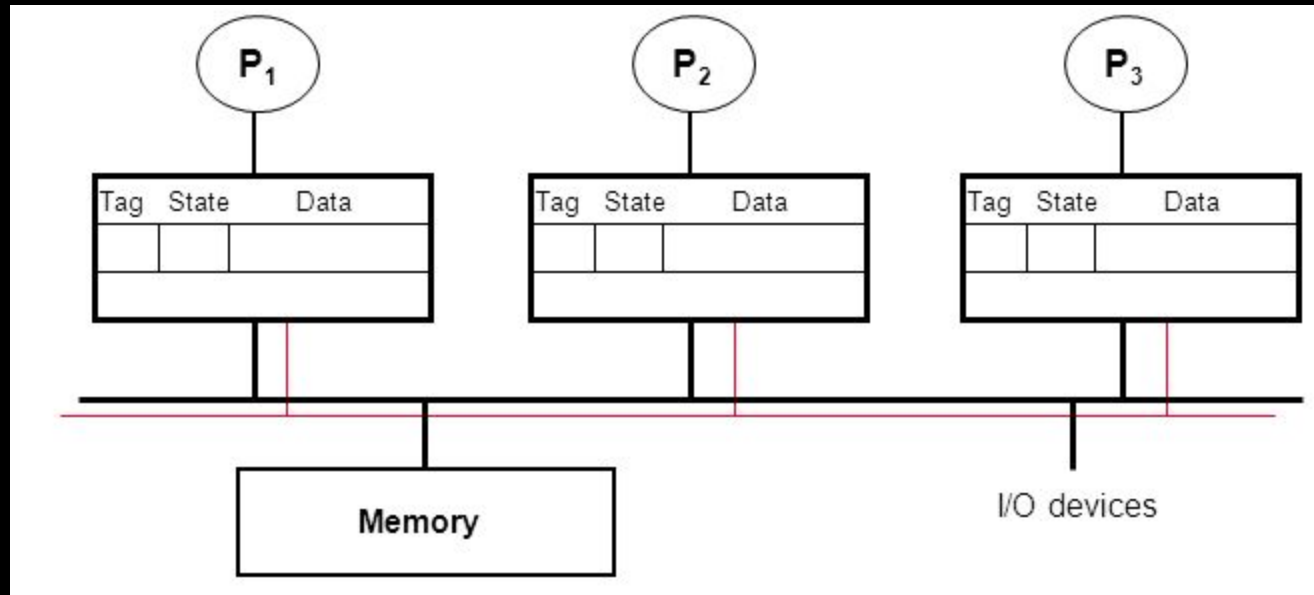
Review: Cache Coherence
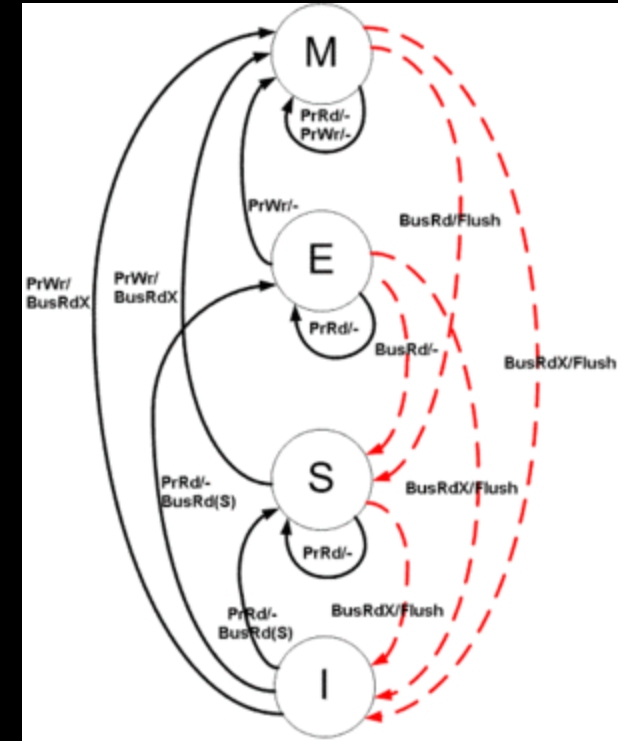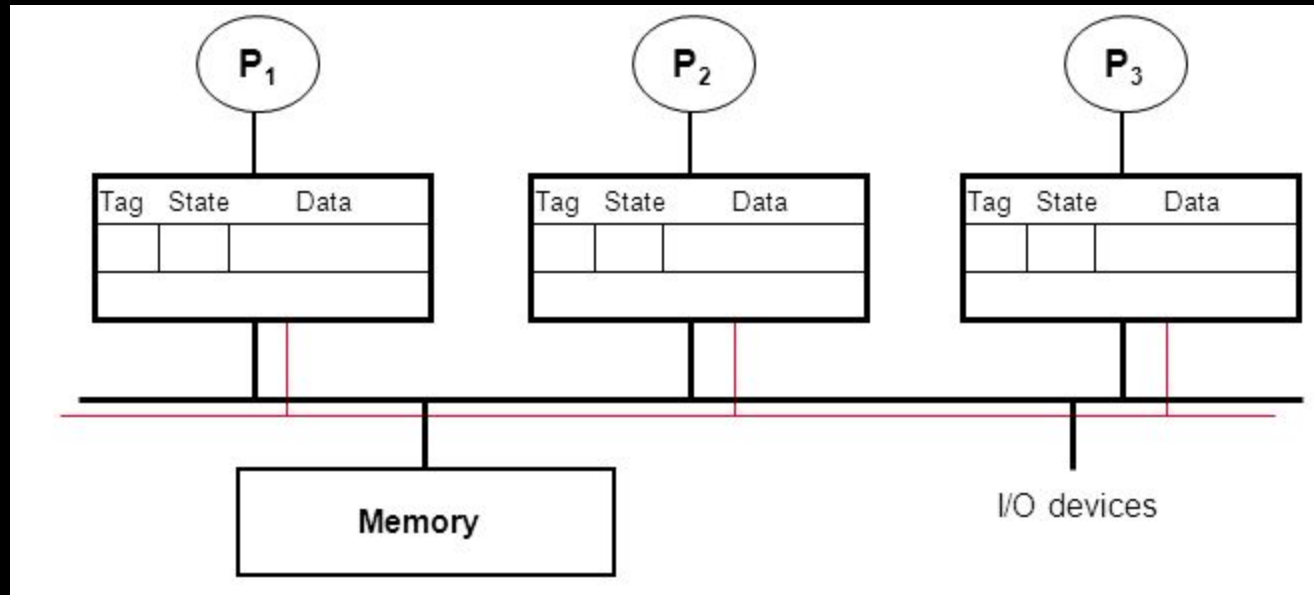
Review: Cache Coherence
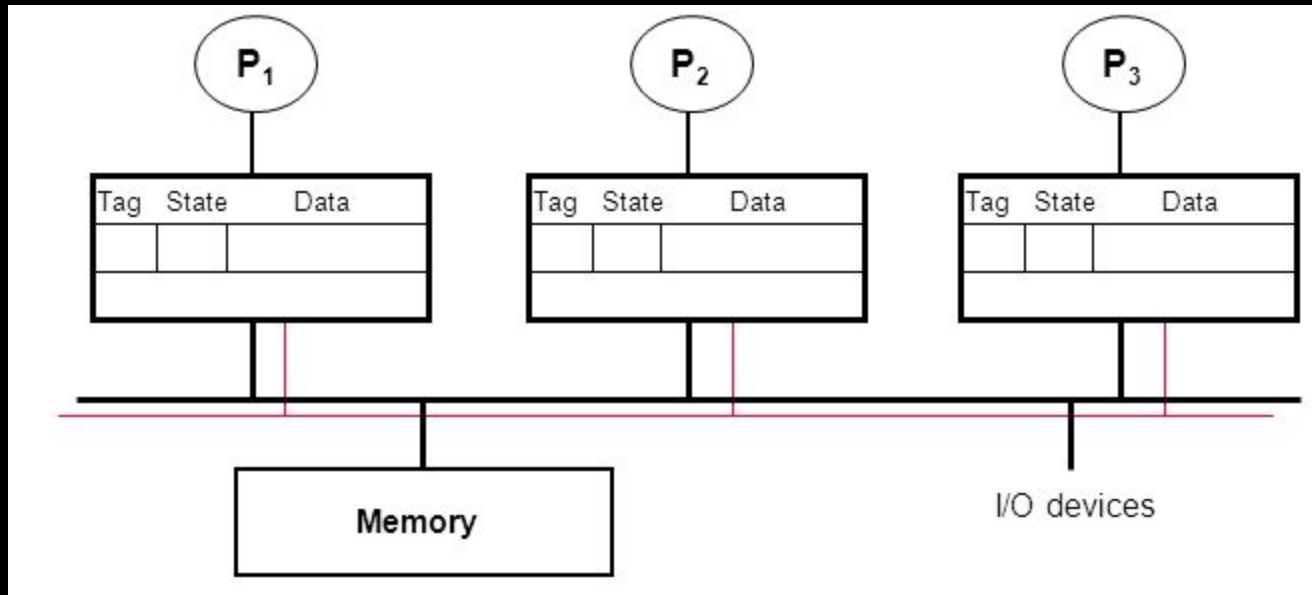
Each cache line has a state (M, E, S, I)

.

.

.

.

.
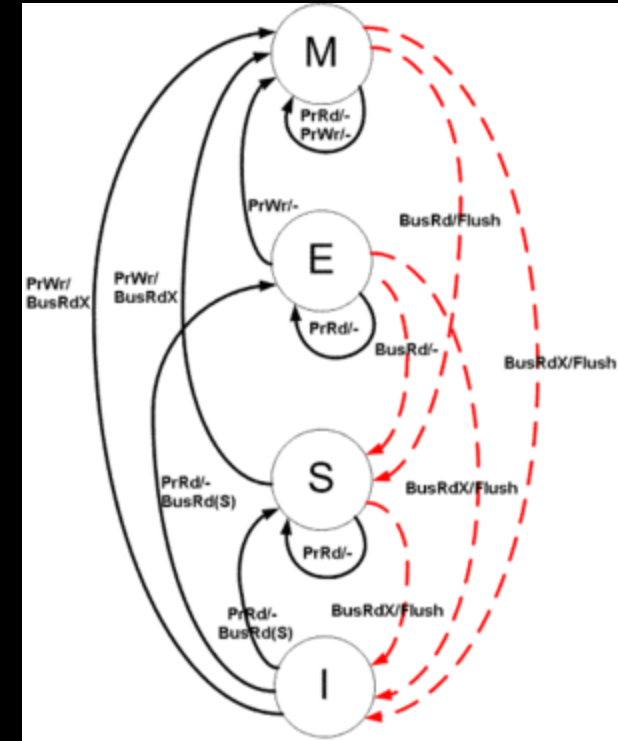
Review: Cache Coherence
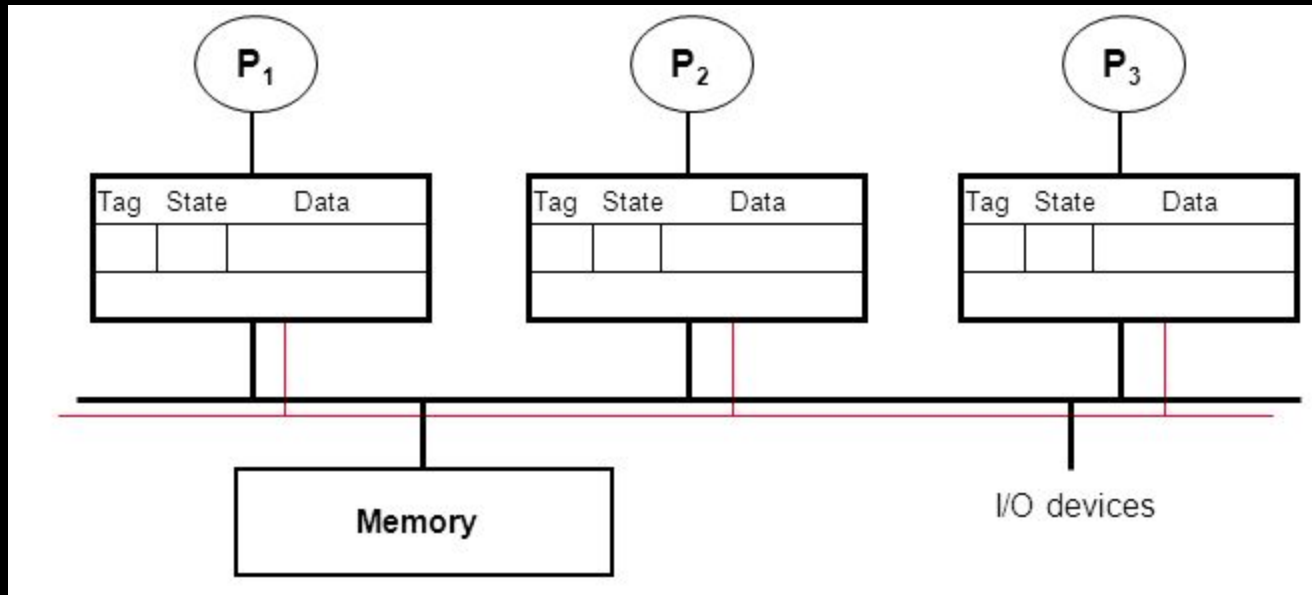
Each cache line has a state (M, E, S, I)
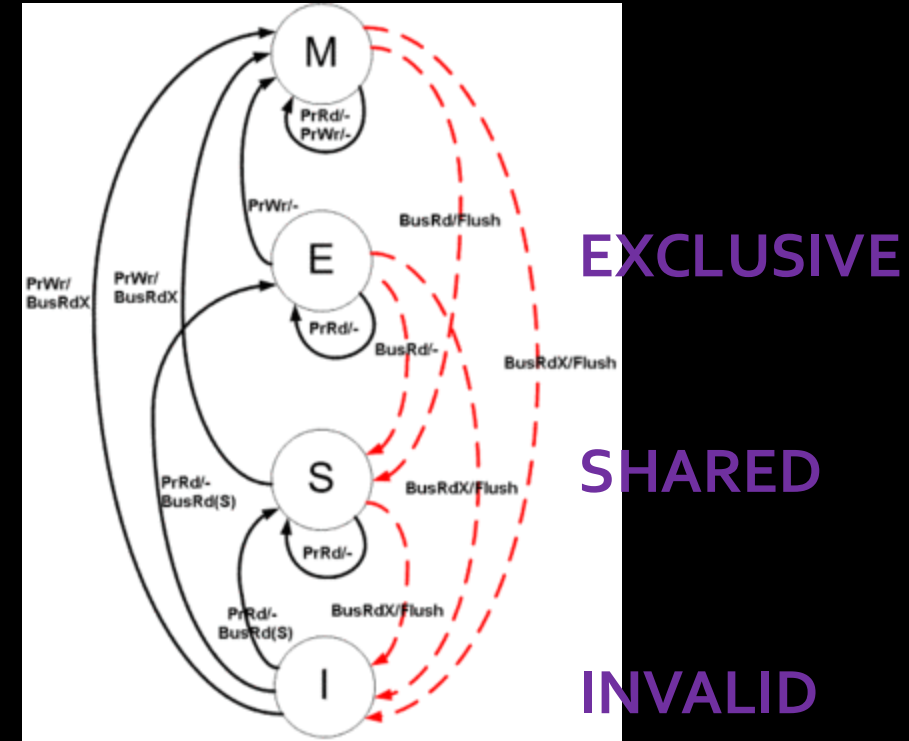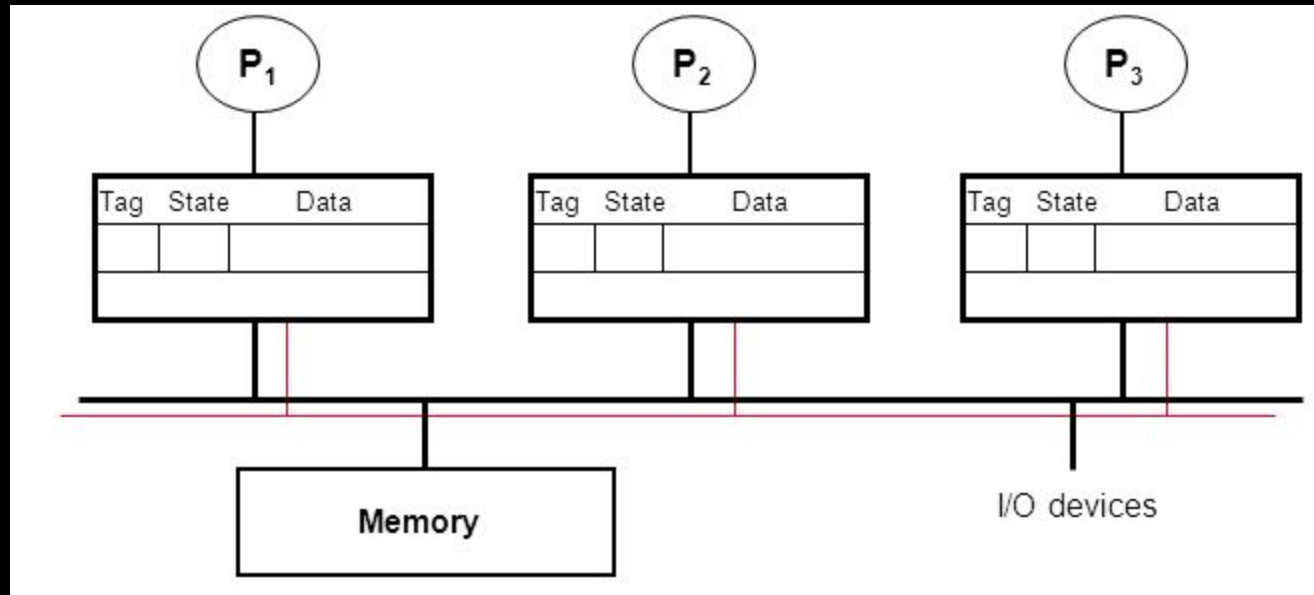
▪ Processors "snoop" bus to maintain states
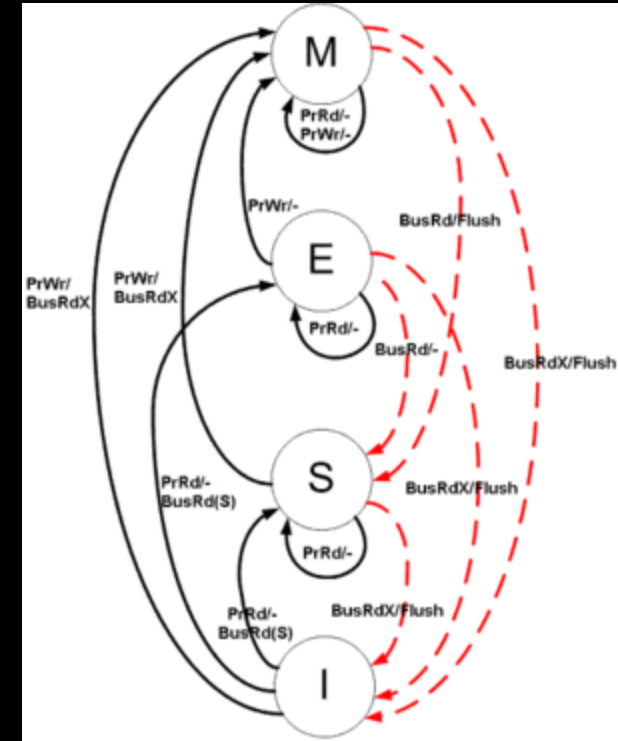
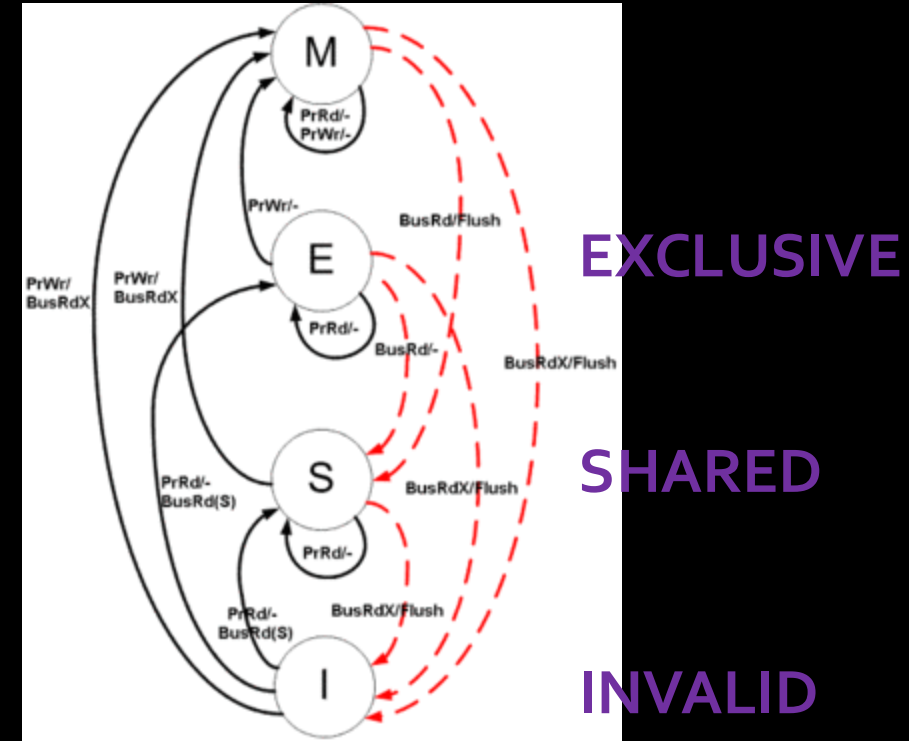# Review: Cache Coherence
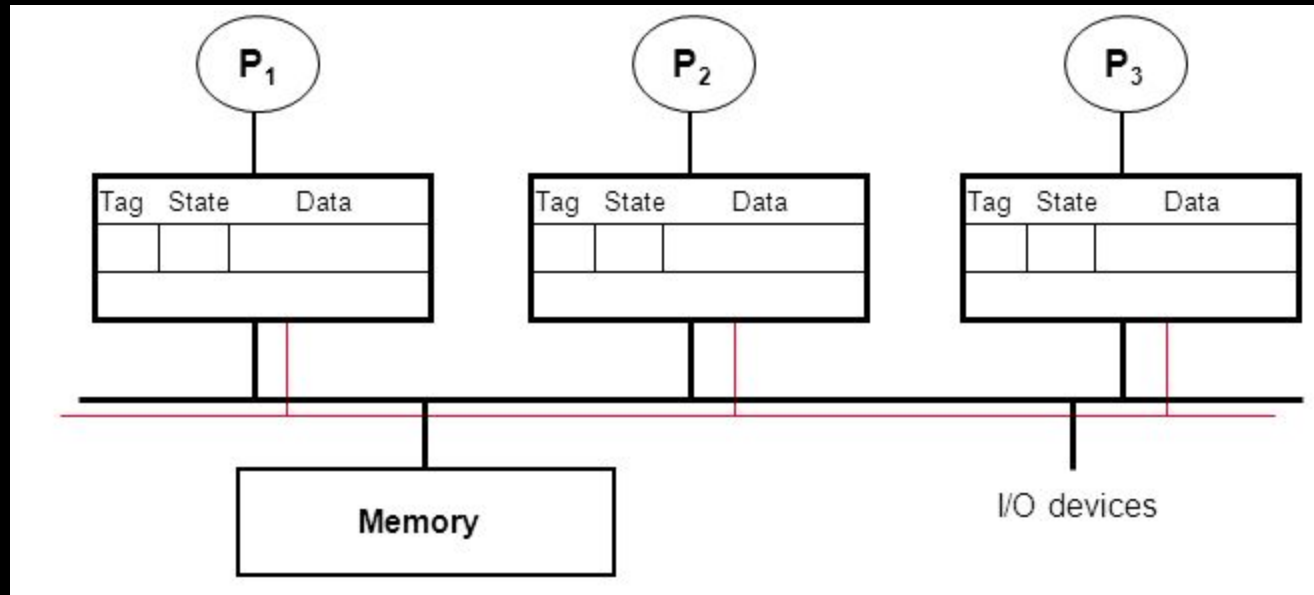
Each cache line has a state (M, E, S, I)

- Processors "snoop" bus to maintain states
- Initially → 'I' → Invalid

INVALID

Review: Cache Coherence
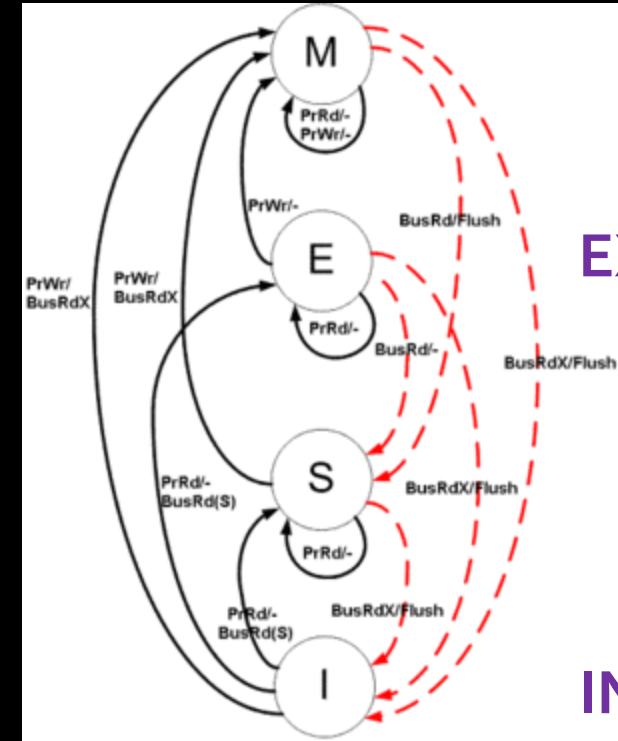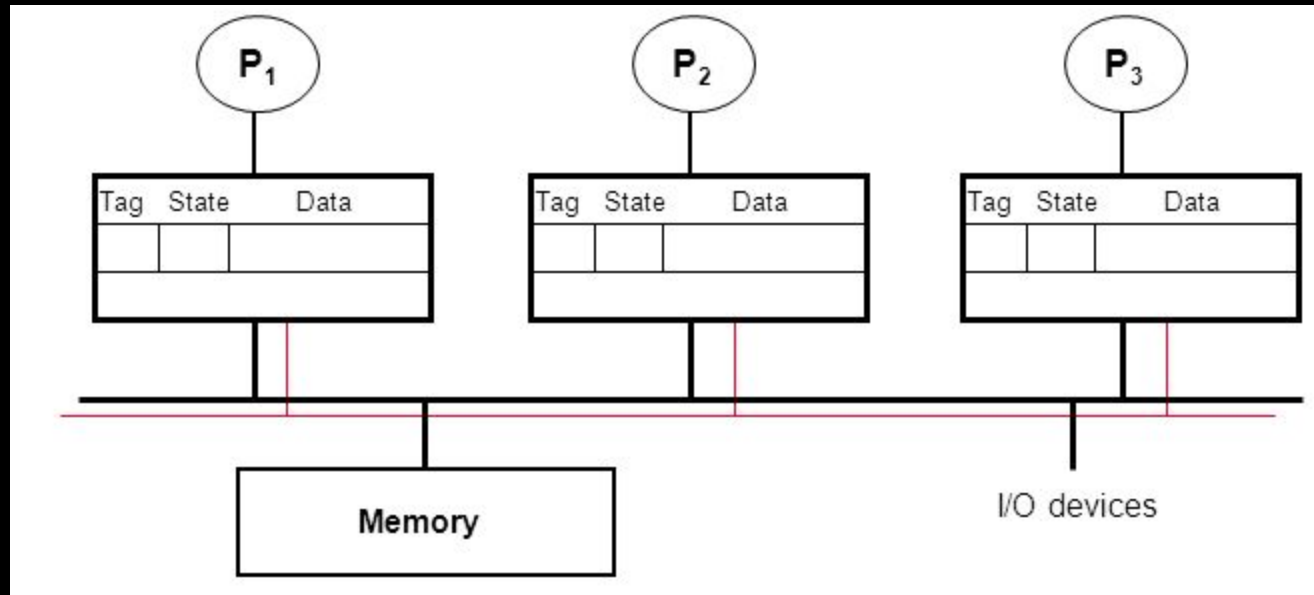
Each cache line has a state (M, E, S, I)
- Processors "snoop" bus to maintain states
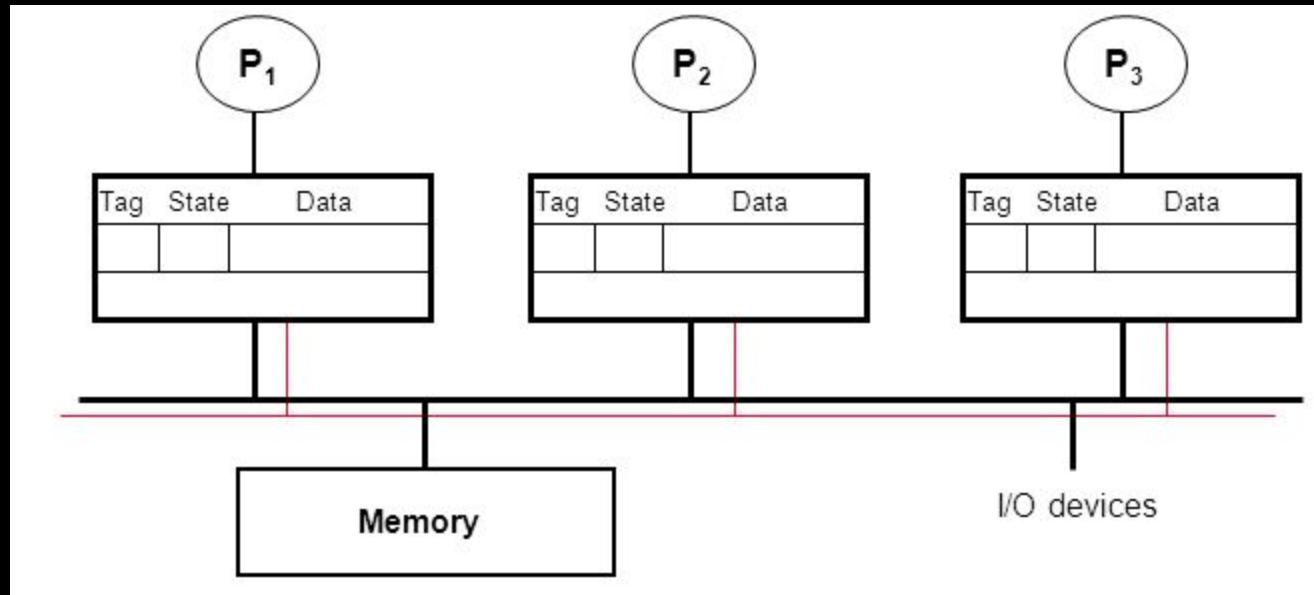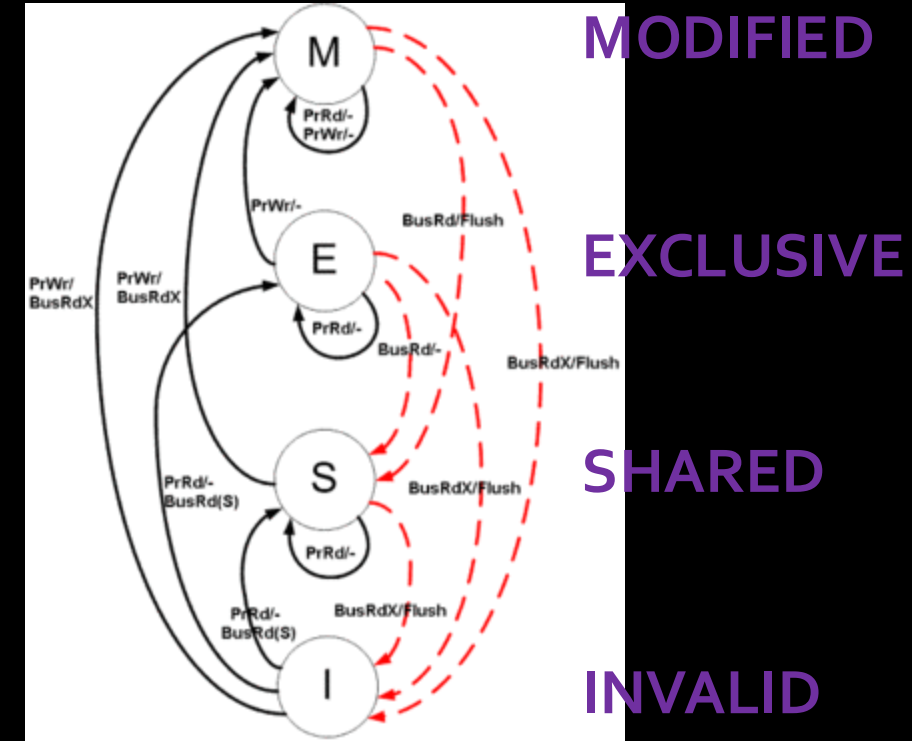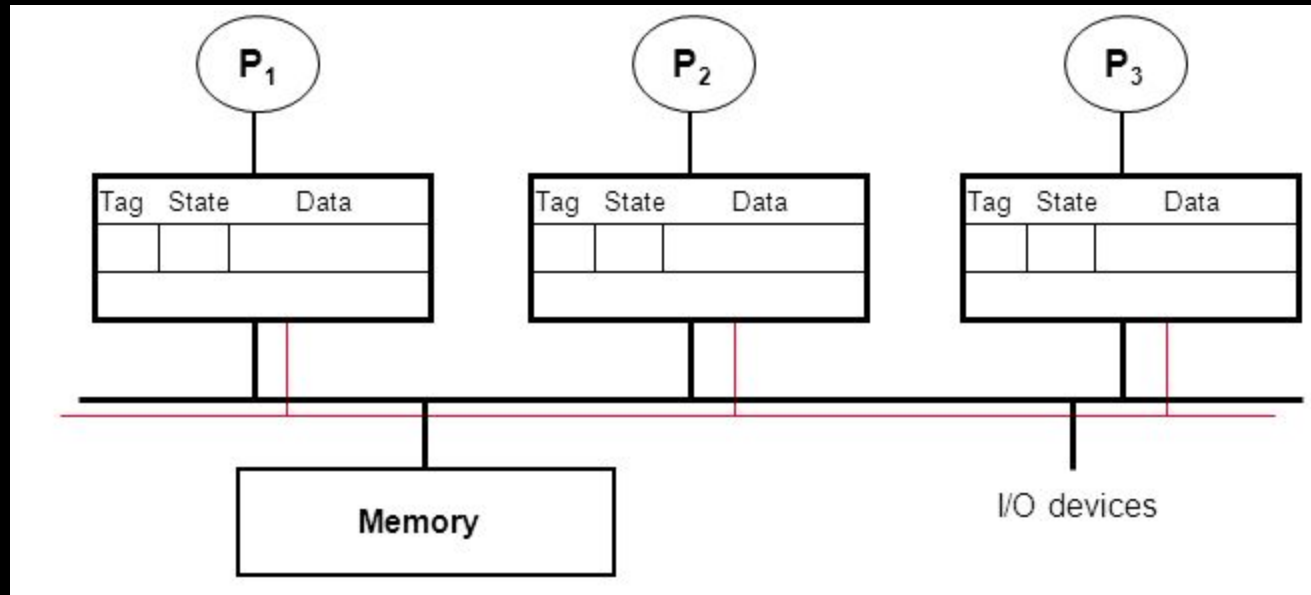- Initially → 'I' → Invalid
- Read one → 'E' → exclusive

EXCLUSIVE

INVALID

Review: Cache Coherence

EXCLUSIVE

SHARED

INVALID

Each cache line has a state (M, E, S, I)

▪ Processors "snoop" bus to maintain states

▪ Initially → 'I' → Invalid

▪ Read one → 'E' → exclusive

▪ Reads → 'S' → multiple copies possible

Review: Cache Coherence
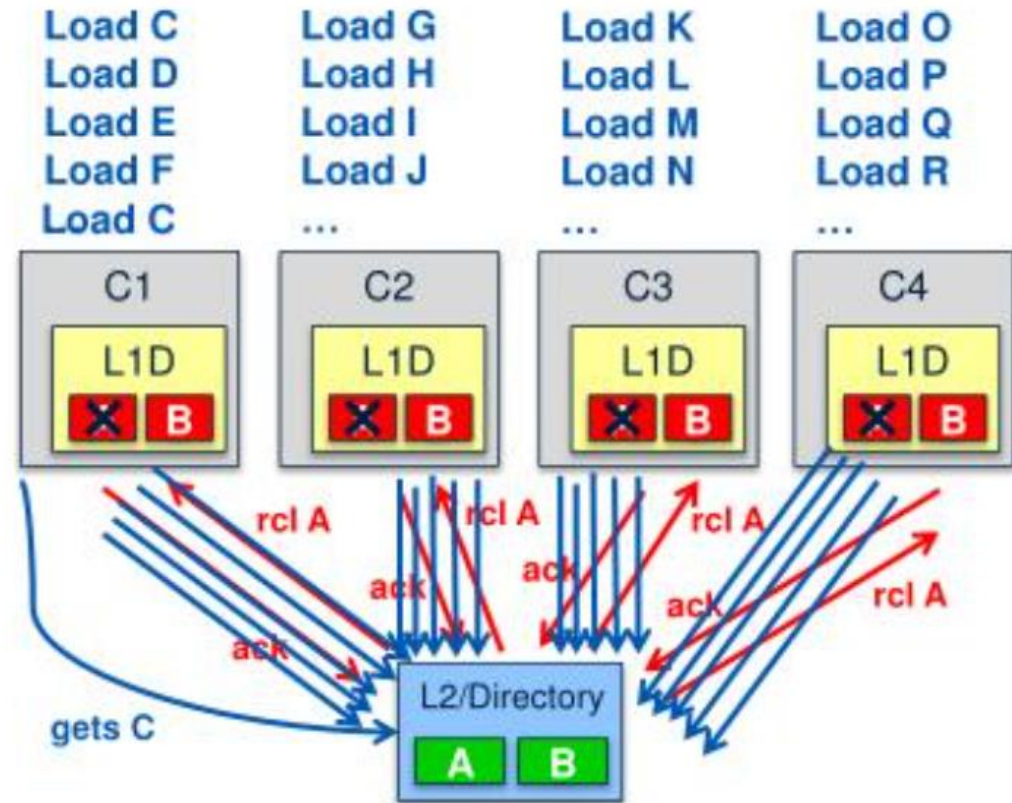

EXCLUSIVE

SHARED

INVALID

Each cache line has a state (M, E, S, I)
- Processors "snoop" bus to maintain states
- Initially → 'I' → Invalid
- Read one → 'E' → exclusive
- Reads → 'S' → multiple copies possible
- Write → 'M' → single copy → lots of cache coherence traffic

Review: Cache Coherence
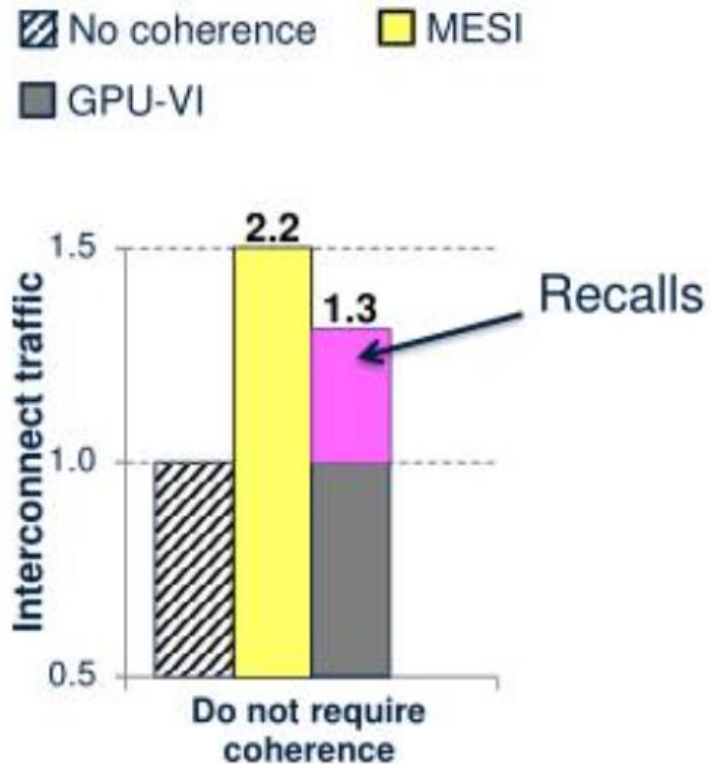

MODIFIED

EXCLUSIVE

SHARED

INVALID

Each cache line has a state (M, E, S, I)

- Processors "snoop" bus to maintain states
- Initially → 'I' → Invalid
- Read one → 'E' → exclusive
- Reads → 'S' → multiple copies possible
- Write → 'M' → single copy → lots of cache coherence traffic

GPU Cache Coherence Challenges

GPU Cache Coherence Challenges
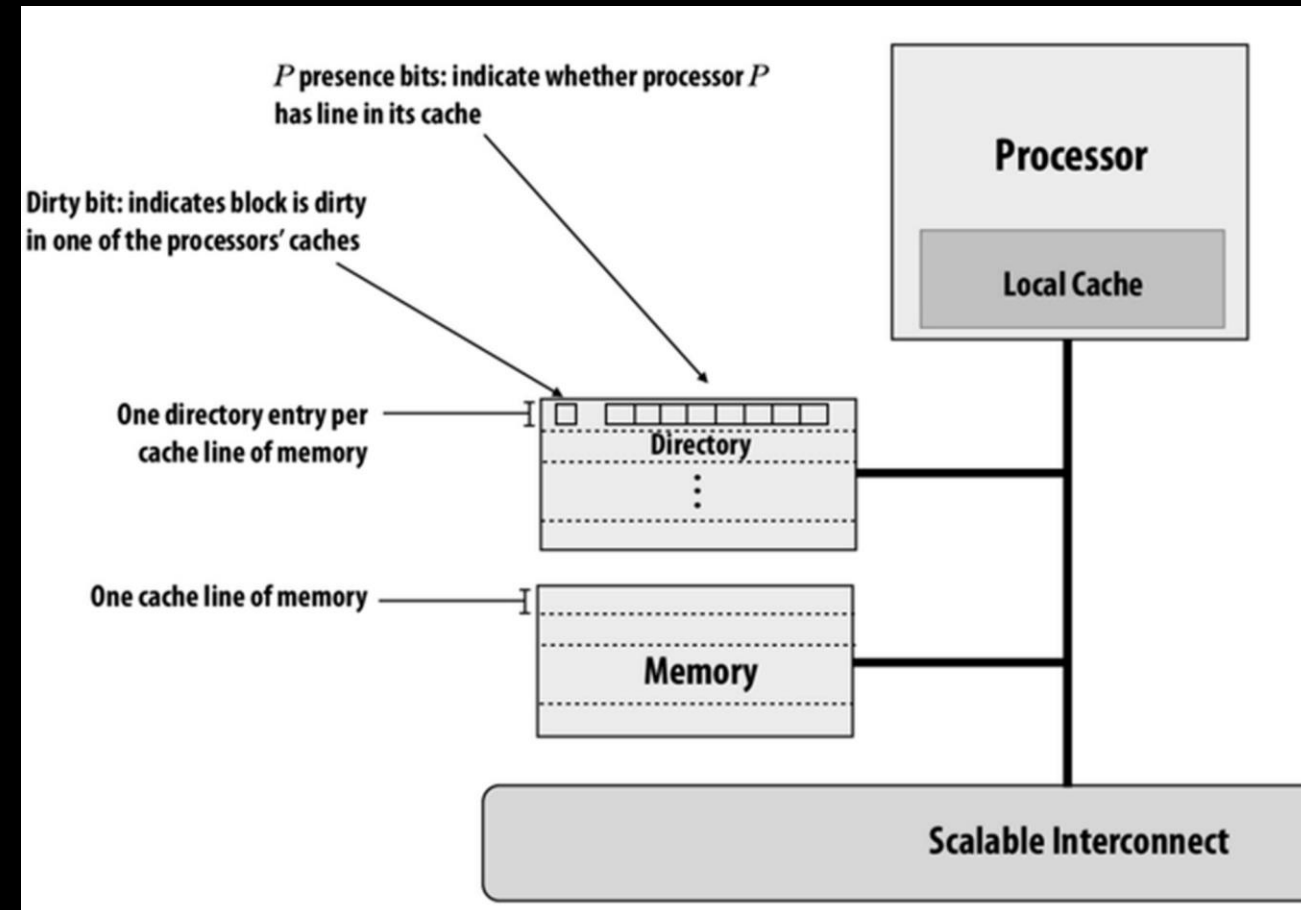

- Challenge 2: Tracking in-flight requests
  - Significant % of L2

S
Shared

S_M

M
Modified

L2 / Directory

MSHR

Background: Directory Protocol

- For each block: centralized "directory" for state in caches

- Directory is co-located with some global view of memory

- Requests are no longer seen by everyone
  - *Writes are serialized through directory*

P presence bits: indicate whether processor P has line in its cache

Dirty bit: indicates block is dirty in one of the processors' caches

One directory entry per cache line of memory — Directory

One cache line of memory — Memory

Processor

Local Cache
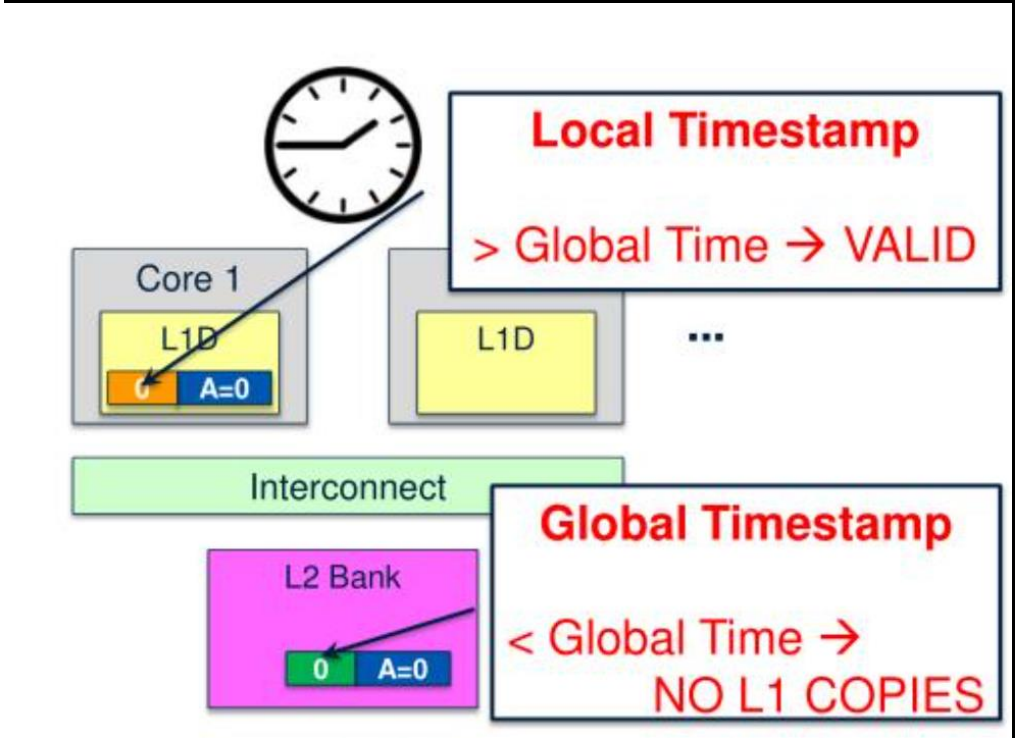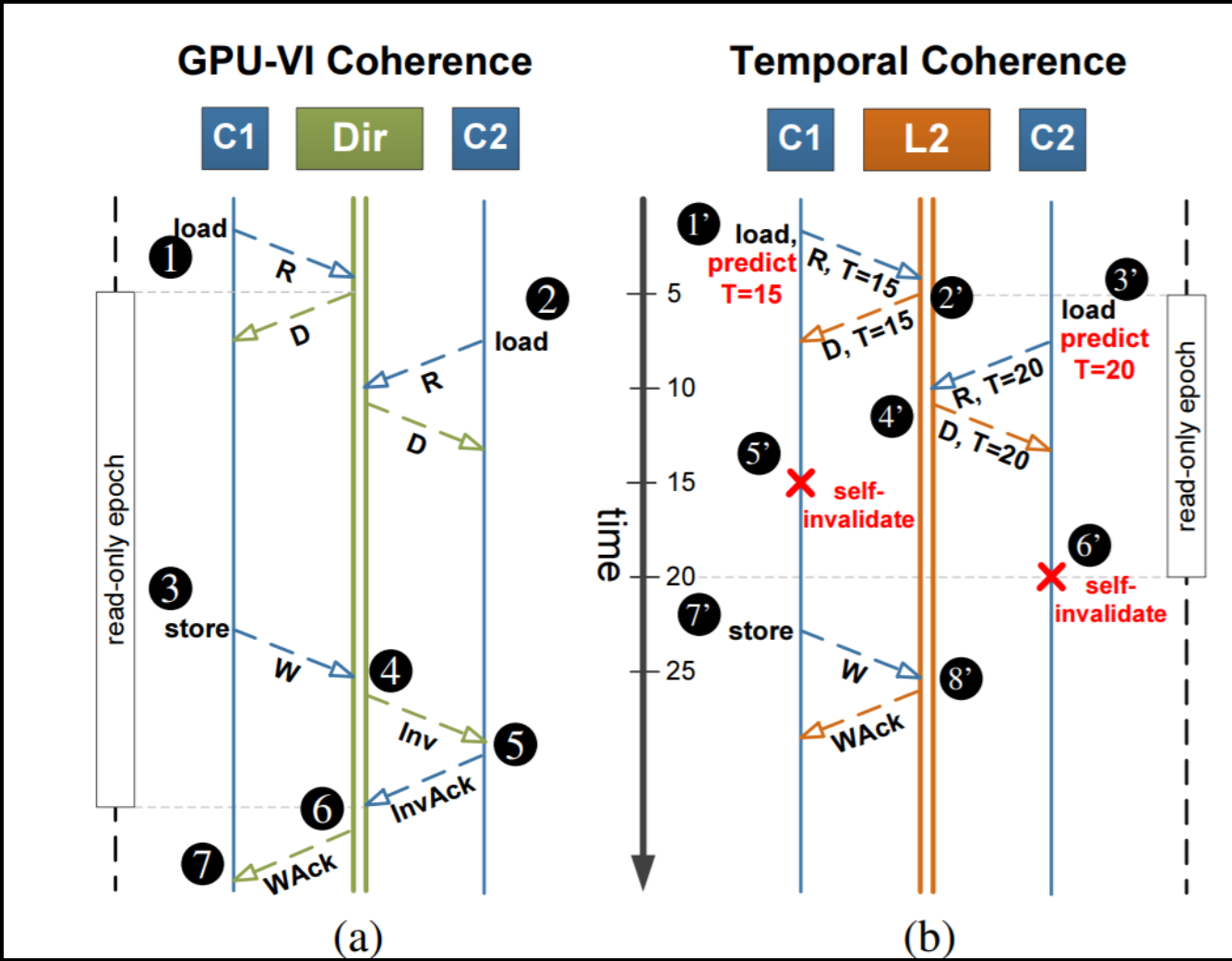
Scalable Interconnect

GPU-VI

- Directory-Based
  - Different from snoop-model
  - Global directory metadata at L2
- Two states
  - Valid
  - Invalid
- Writes invalidate other copies
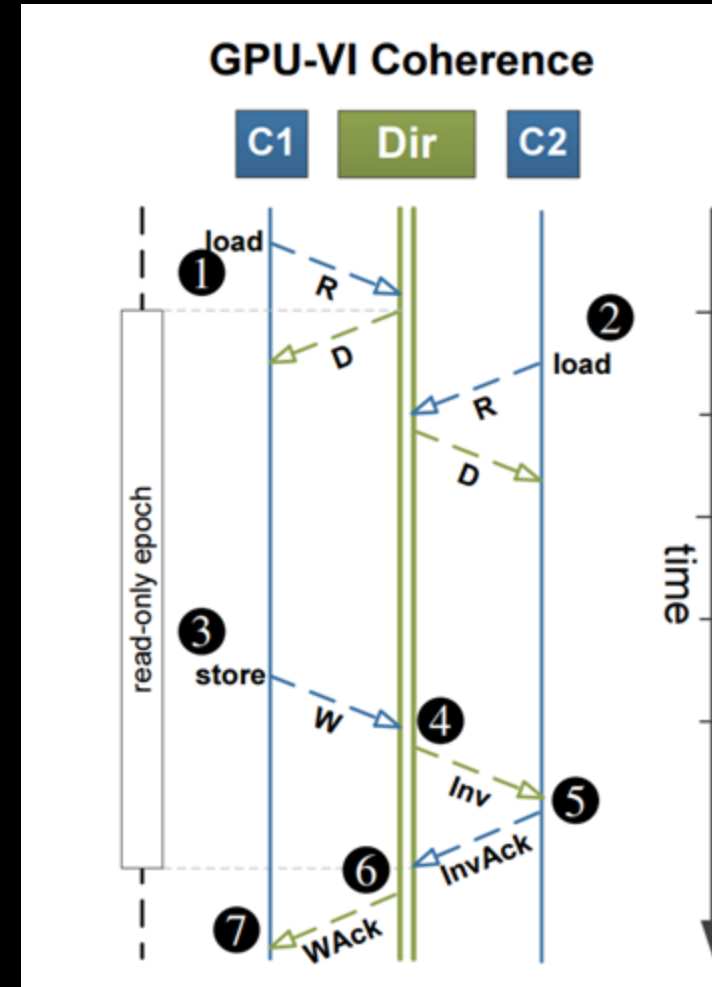
Temporal Coherence (TC)

TC-Strong vs TC-Weak