GPUs to the left GPUs to the right GPUs all day GPUs all night

Chris Rossbach

cs378

Outline for Today

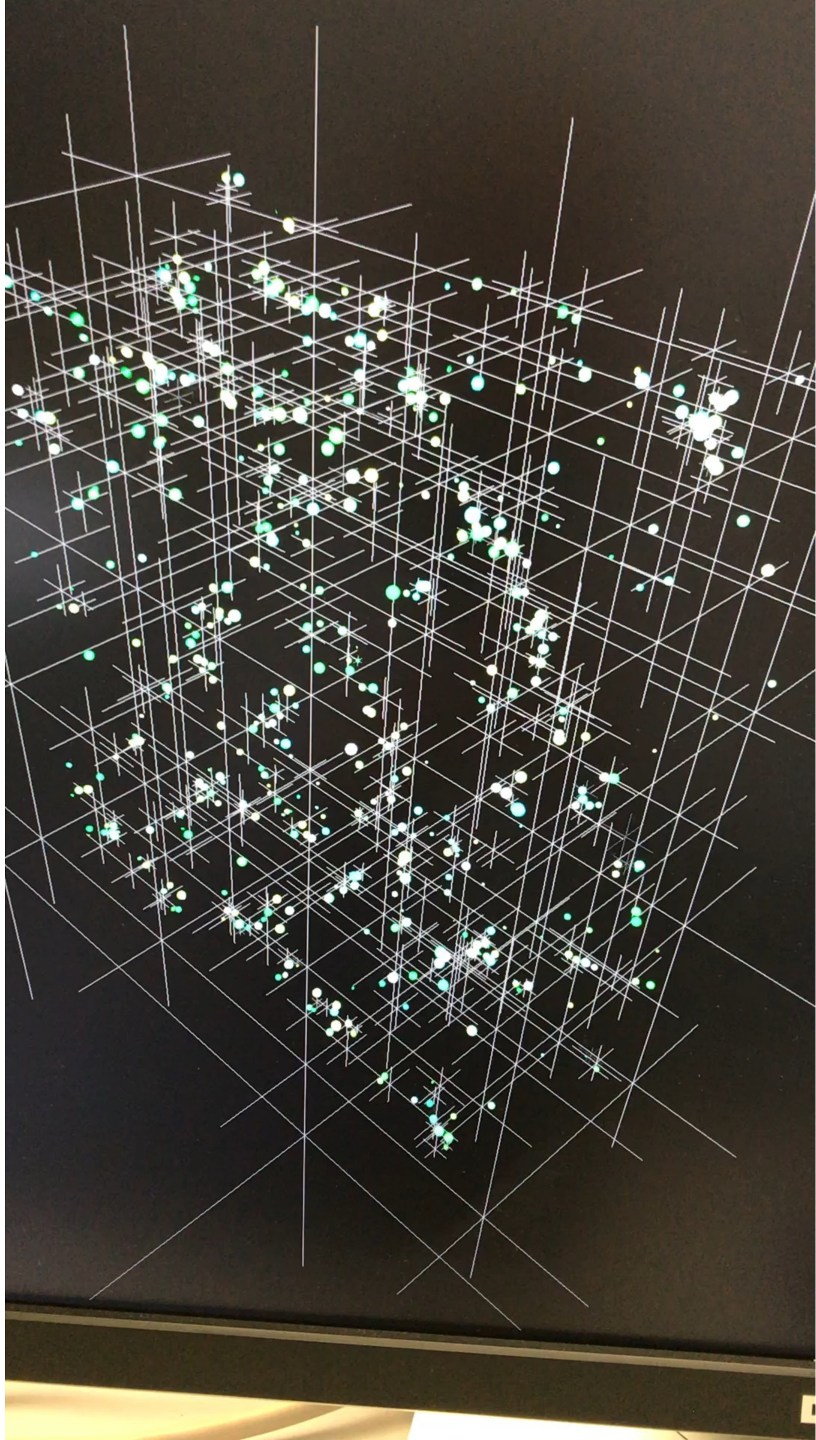
- Questions?
- Administrivia
    - Exam graded
- Agenda
    - CUDA continued
    - CUDA Performance

Acknowledgements:

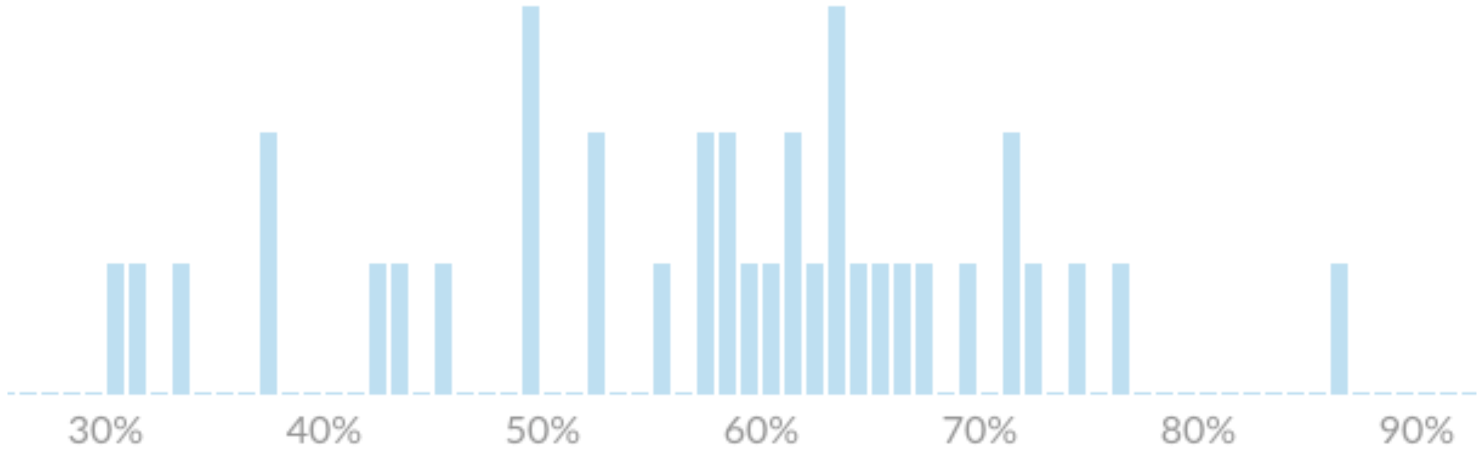
- http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx
- http://www.seas.upenn.edu/~cis565/LECTURES/CUDA%20Tricks.pptx
- http://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/GPU%20Programming.pptx

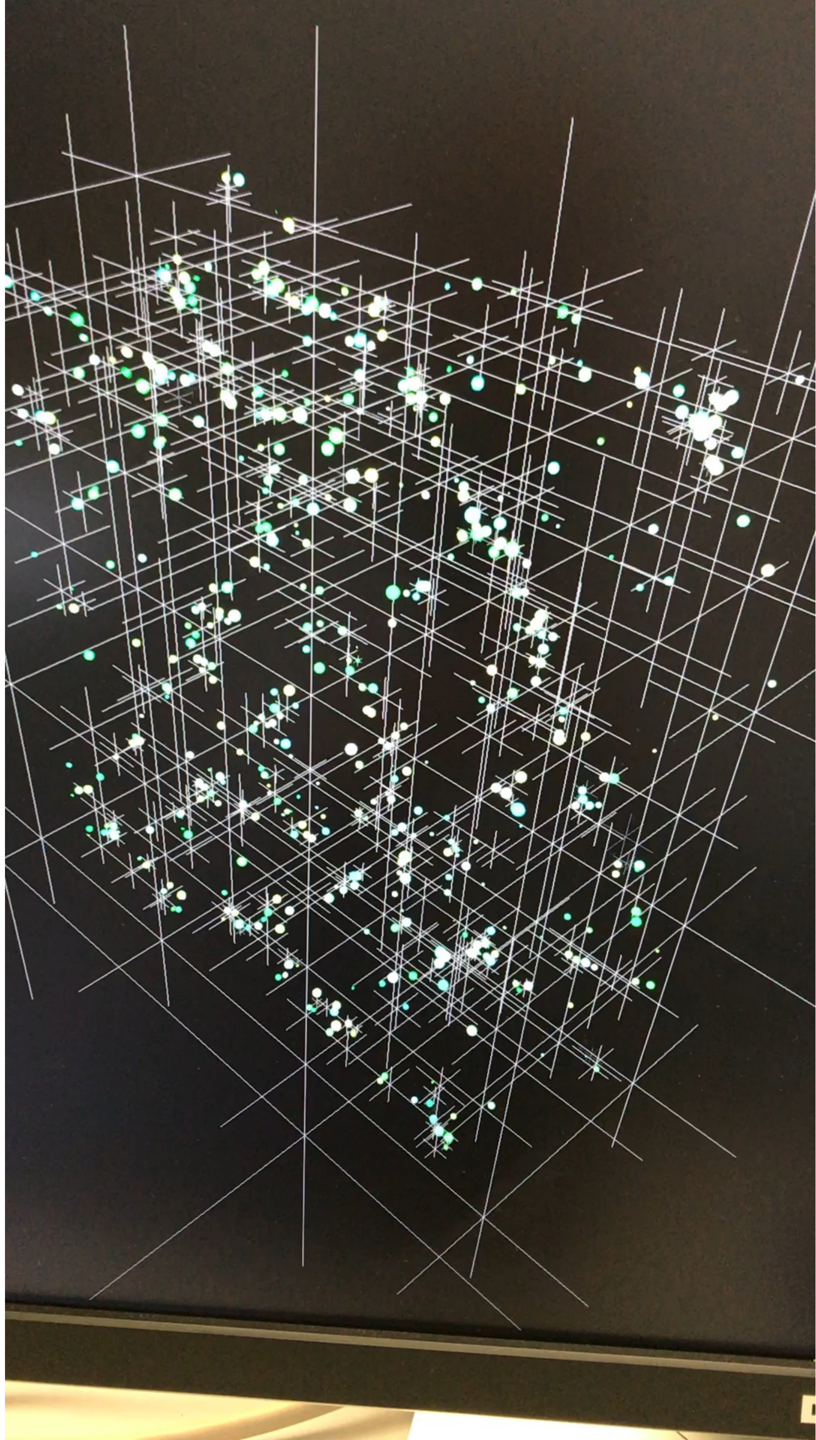
Exam Stats

Exam Stats

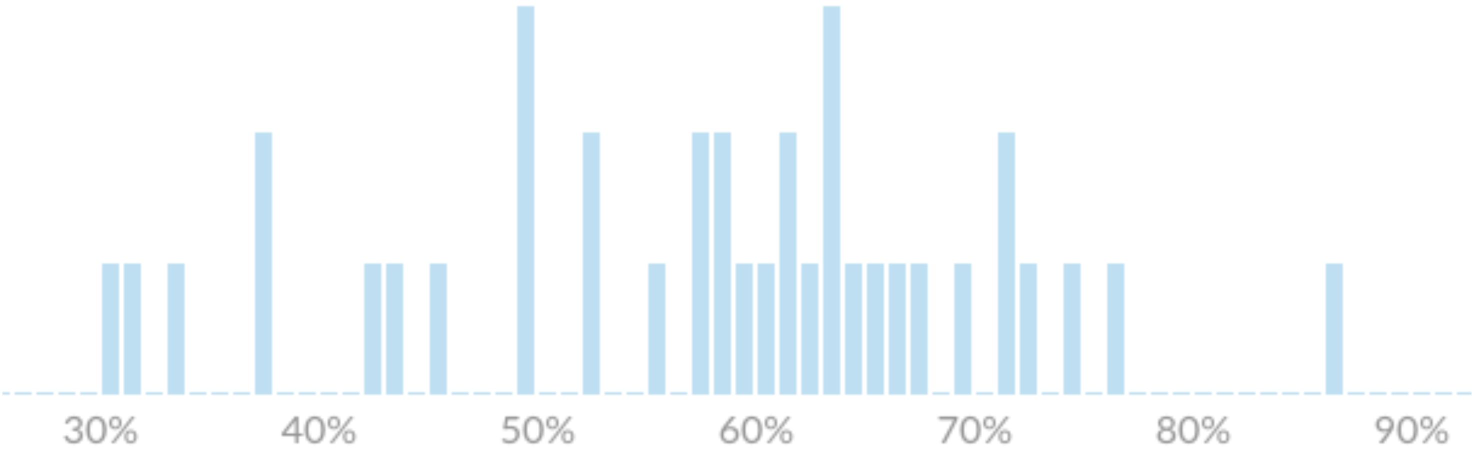


Exam Stats

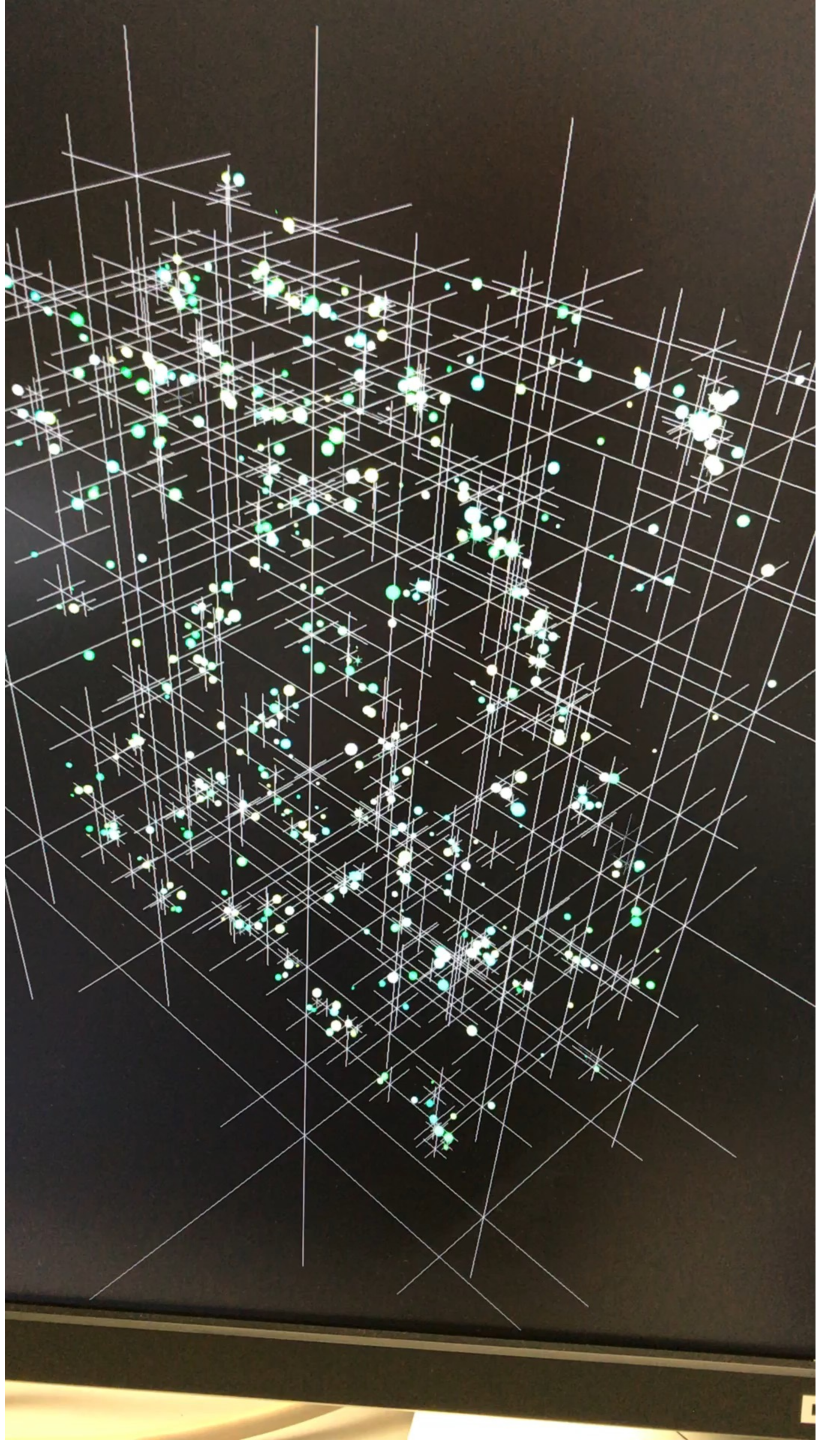




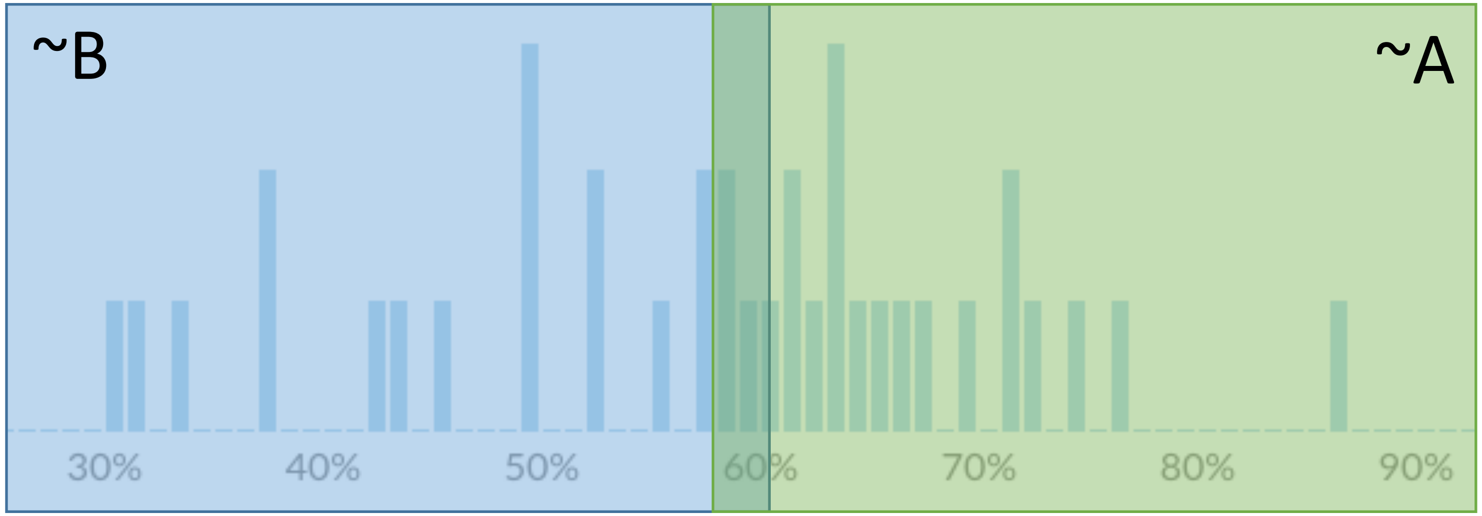
Exam Stats



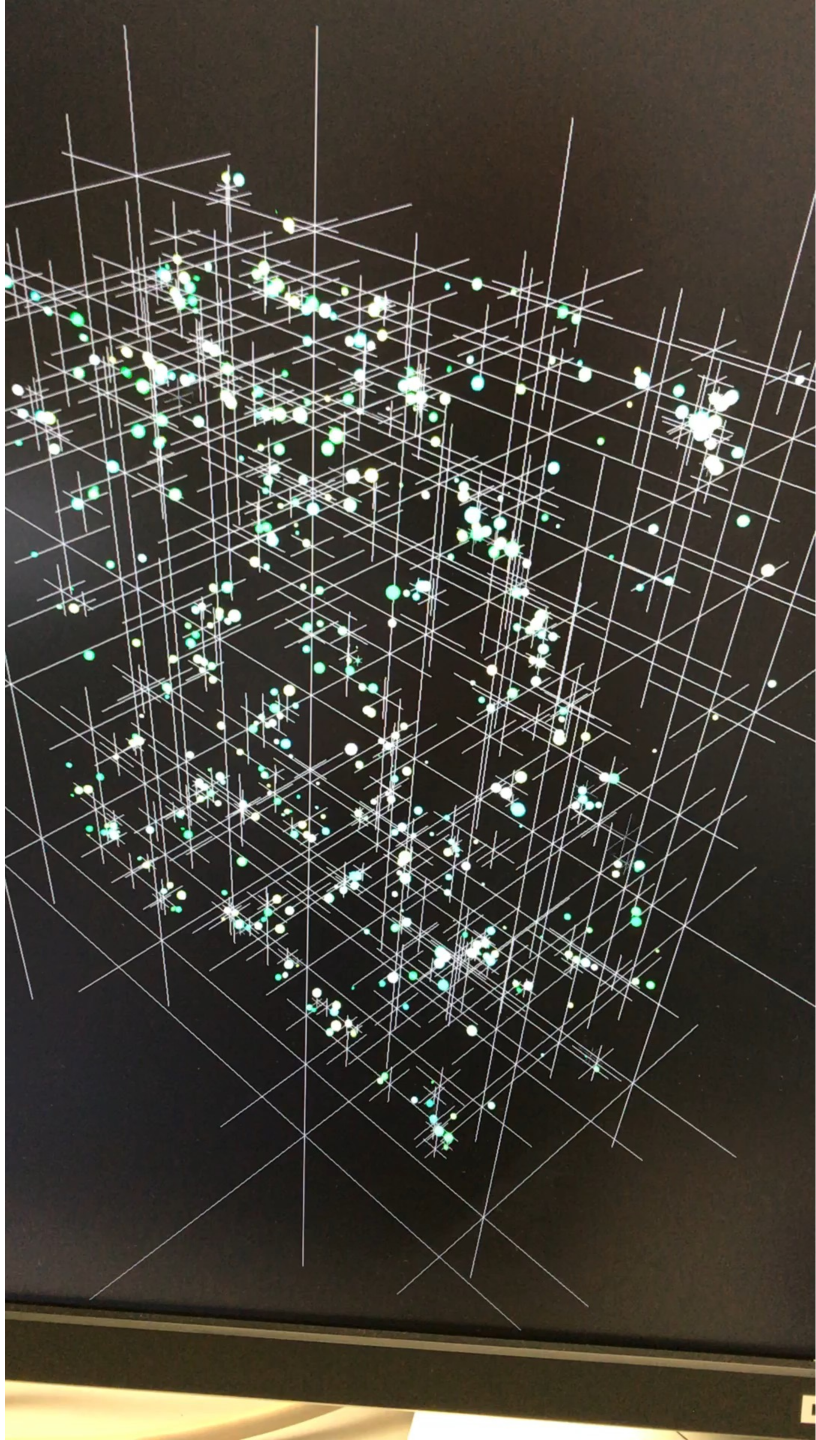
Average: ~60
High: 91
Low: 33
Stdev: ~14



Exam Stats



Average: ~60
High: 91
Low: 33
Stdev: ~14



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?

Review: Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

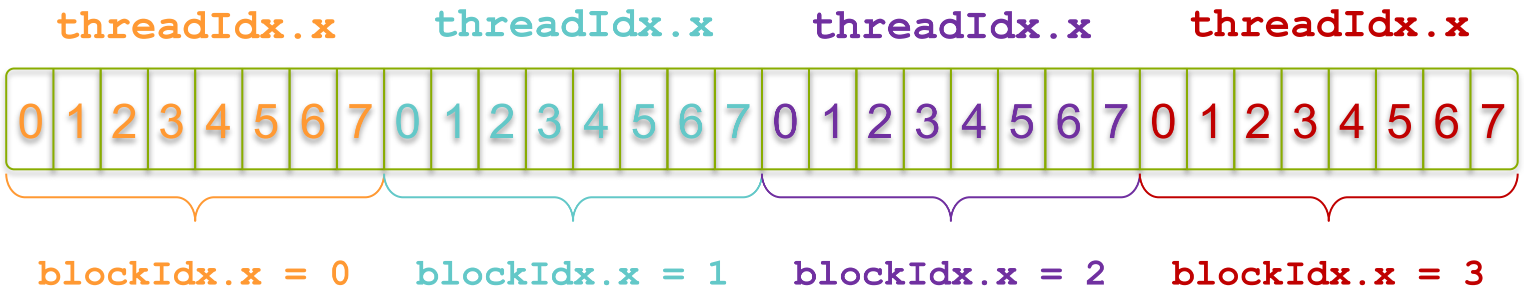
Review: Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

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

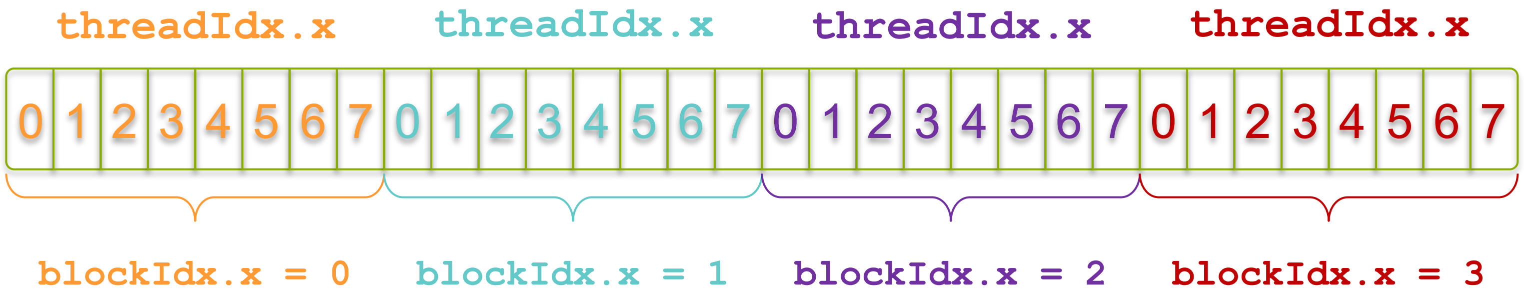
Review: Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)



Review: Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

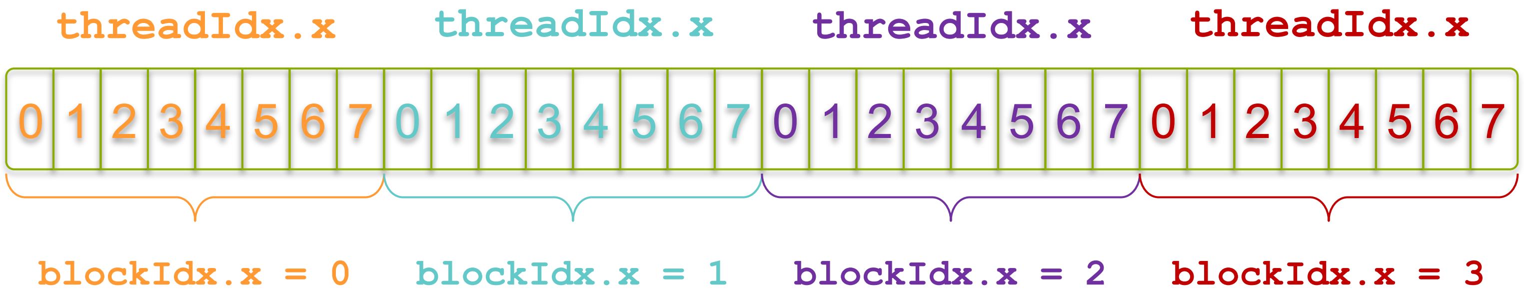


- With M threads/block, unique index per thread is :

Review: Blocks and Threads

- Mos
  - In

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```



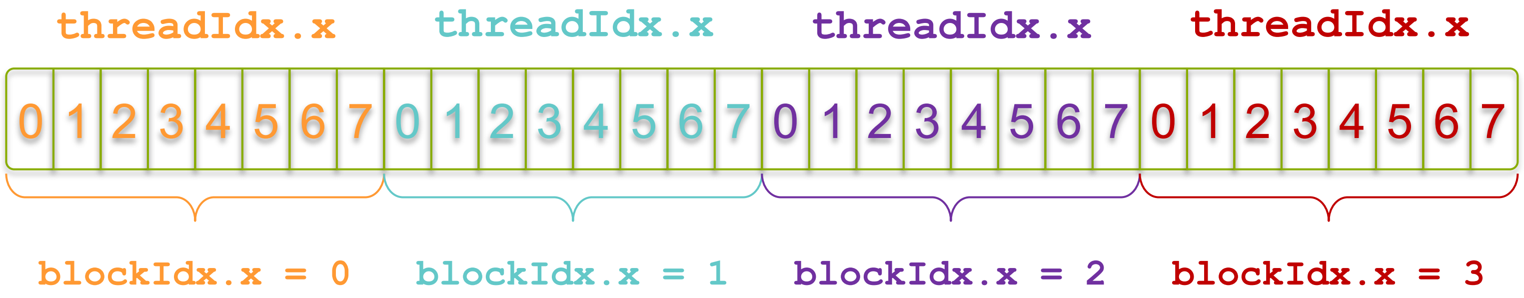
- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

Review: Blocks and Threads

- Mos
  - Ir

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```



- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

Review: Blocks and Threads

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

Update the kernel launch:

```
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```

- Mos
  - Ir

0

`blockIdx.x = 0` `blockIdx.x = 1` `blockIdx.x = 2` `blockIdx.x = 3`

- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

Review: Blocks and Threads

- Mos
  - Ir

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
```

```
>(d_a, d_b, d_c, N);
```

blockIdx.x = 0   blockIdx.x = 1   blockIdx.x = 2   blockIdx.x = 3

- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

Review: Blocks and Threads

- Mos
  - Ir

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
```

- Why have threads?

```
>(d_a, d_b, d_c, N);
```

`blockIdx.x = 0`   `blockIdx.x = 1`   `blockIdx.x = 2`   `blockIdx.x = 3`

- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

Review: Blocks and Threads

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
```

- Mos
  - In

- Why have threads?
  - Why not just blocks or just threads?

```
>(d_a, d_b, d_c, N);
```

blockIdx.x = 0   blockIdx.x = 1   blockIdx.x = 2   blockIdx.x = 3

- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

Review: Blocks and Threads

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
```

- Mos
  - In

- Why have threads?
  - Why not just blocks or just threads?
- Unlike parallel blocks, threads can:
  - Communicate
  - Synchronize

```
>(d_a, d_b, d_c, N);
```

Y

blockIdx.x = 0   blockIdx.x = 1   blockIdx.x = 2   blockIdx.x = 3

- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

What if my array size N % M != 0 !!???

How many threads/blocks should I use?

```
    // 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 should I use?

```
    // 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 should I use?

```
    // 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

Internals

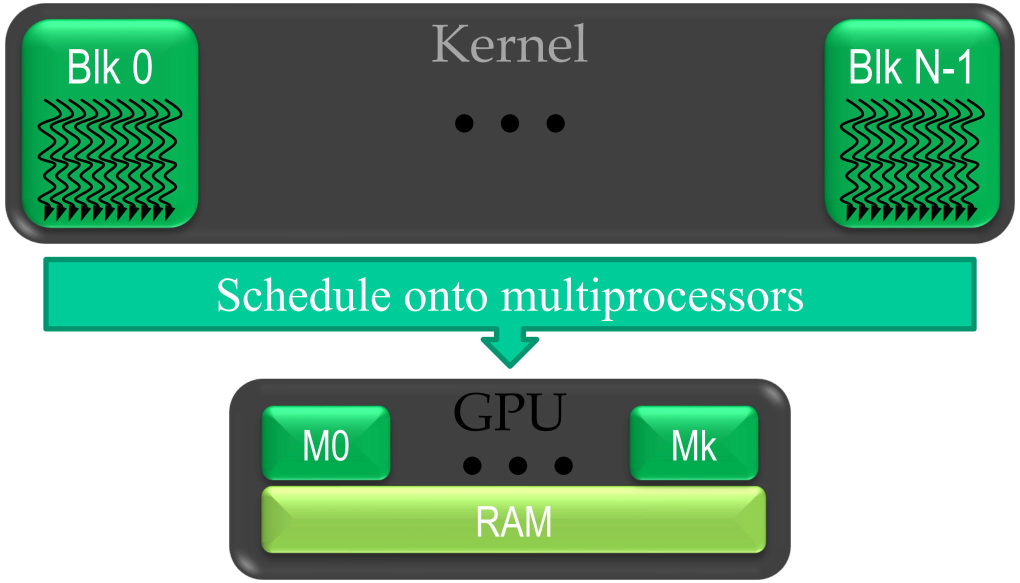
```
__host__
void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
  addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

```
__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];
}
```

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);
}
```

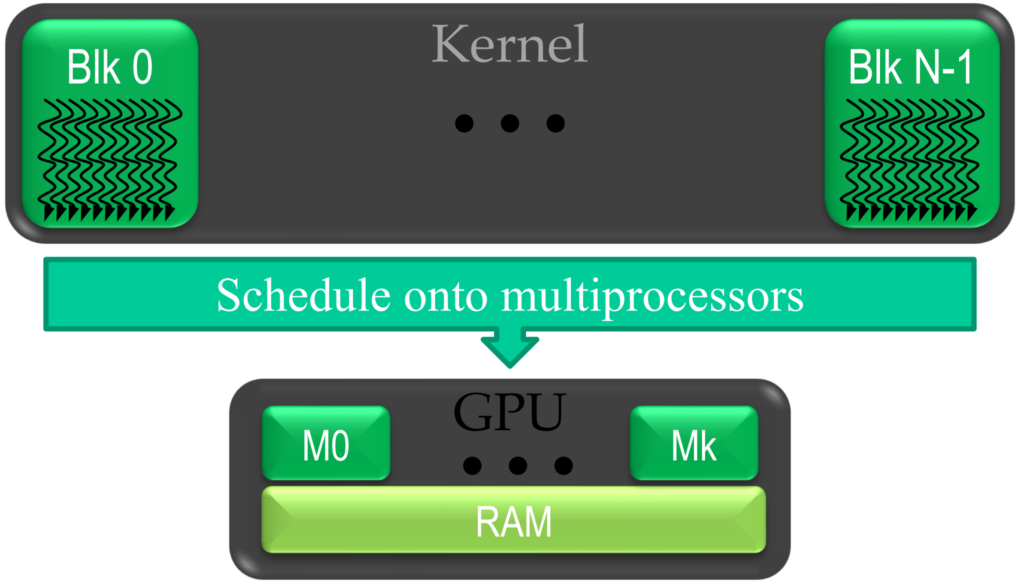
```
__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;
   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



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];
}
```



How are threads scheduled?

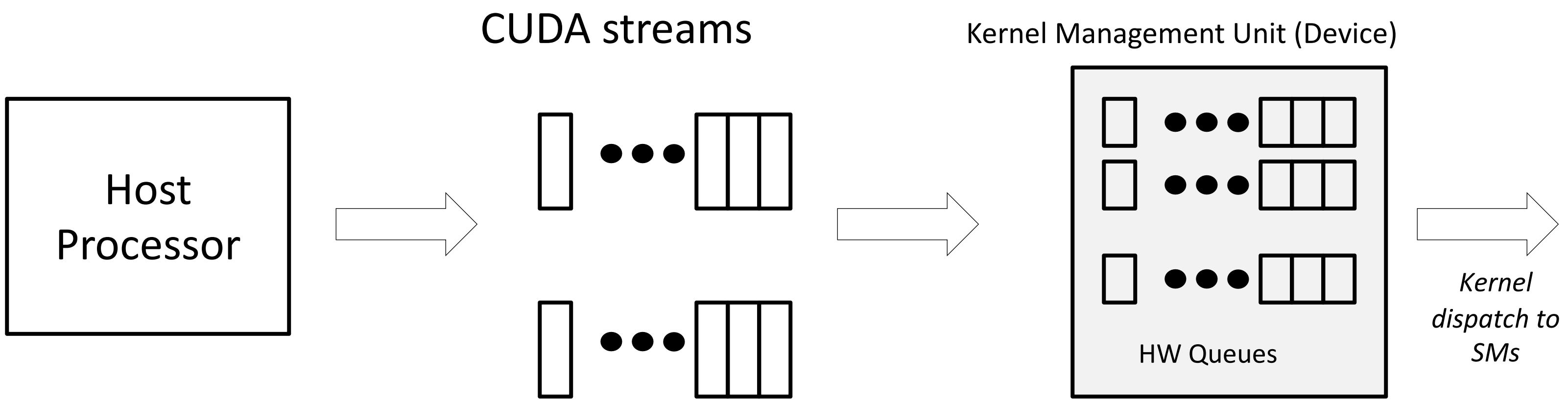
Kernel Launch

Kernel Launch

- Commands by host issued through ***streams***

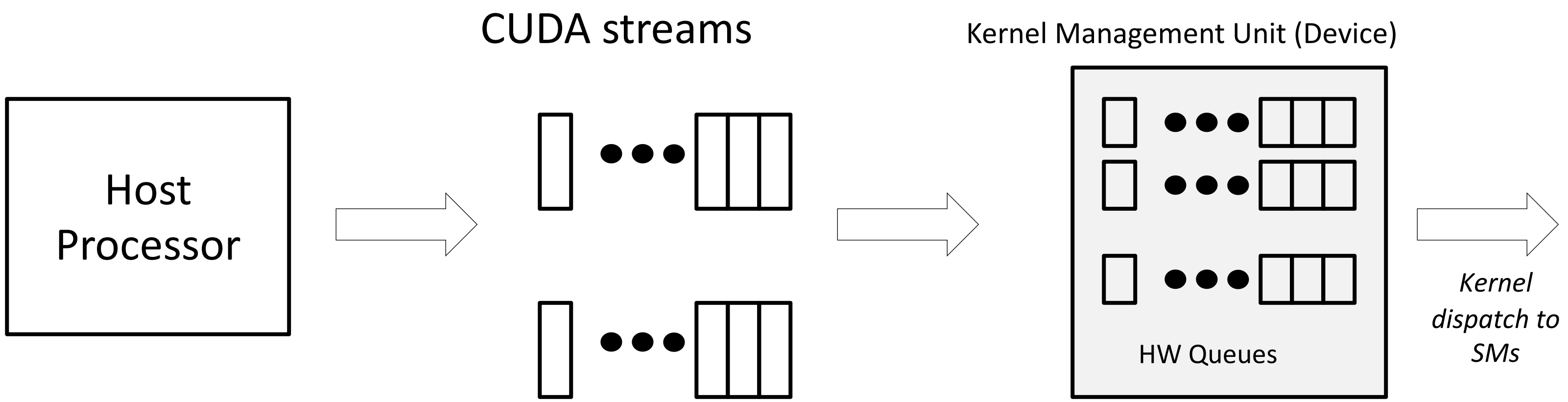
Kernel Launch

- Commands by host issued through ***streams***



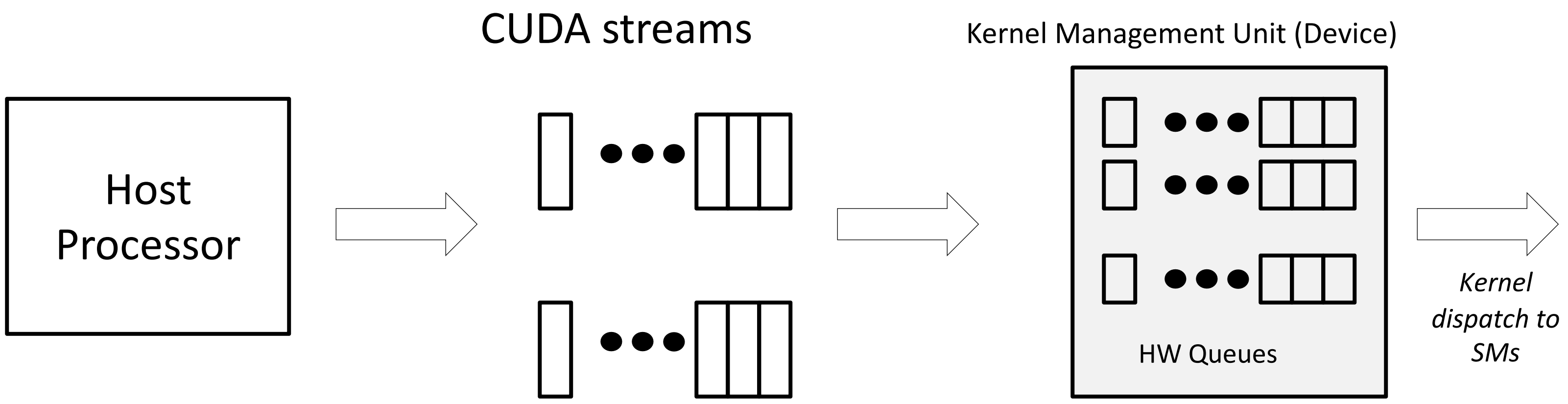
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially



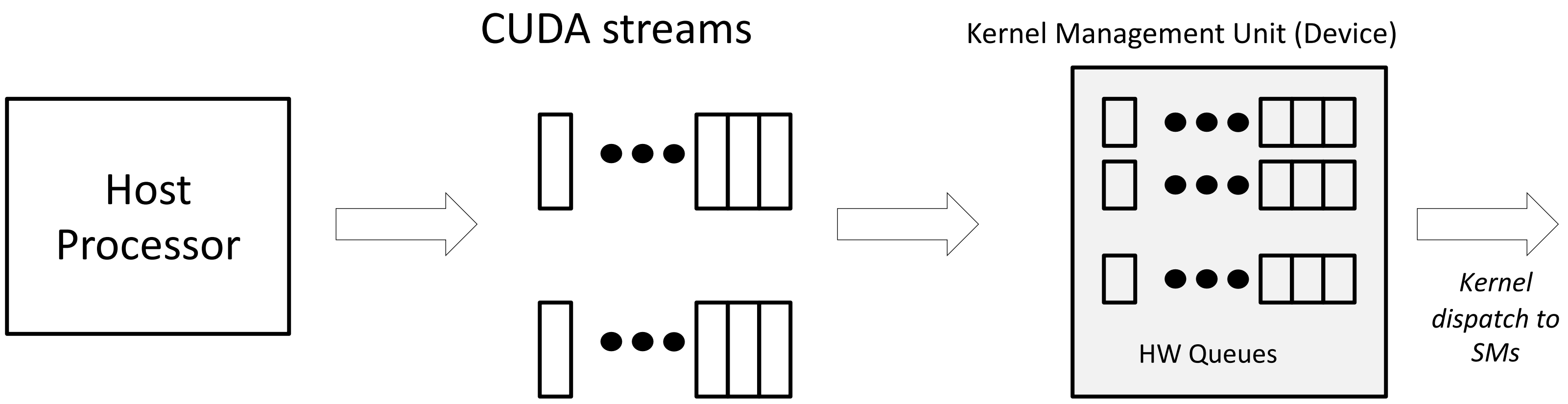
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently



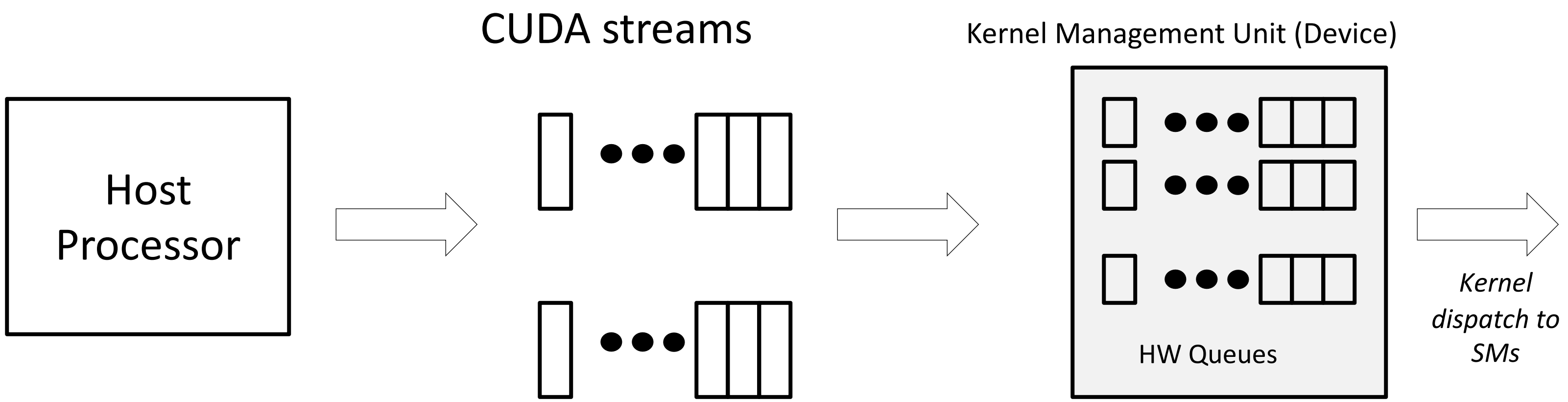
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues



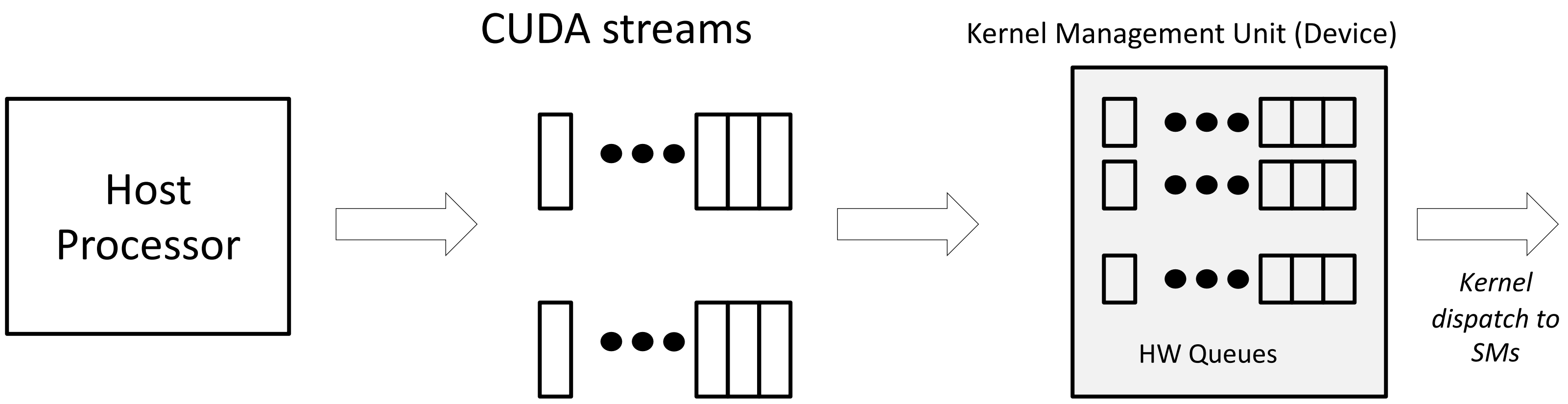
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)



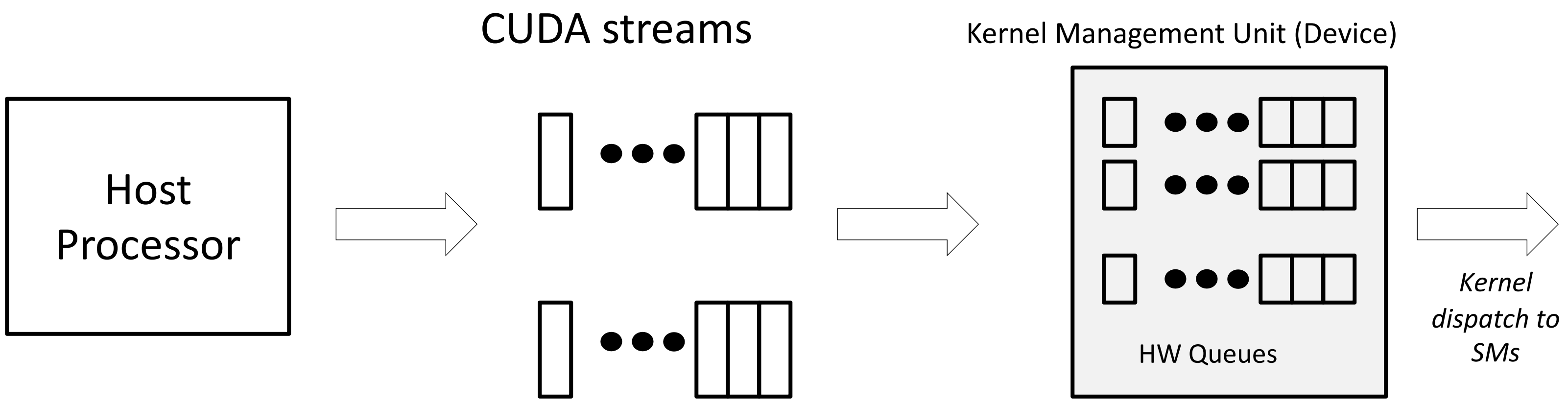
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
  - Multiple streams mapped to each queue → serializes some kernels

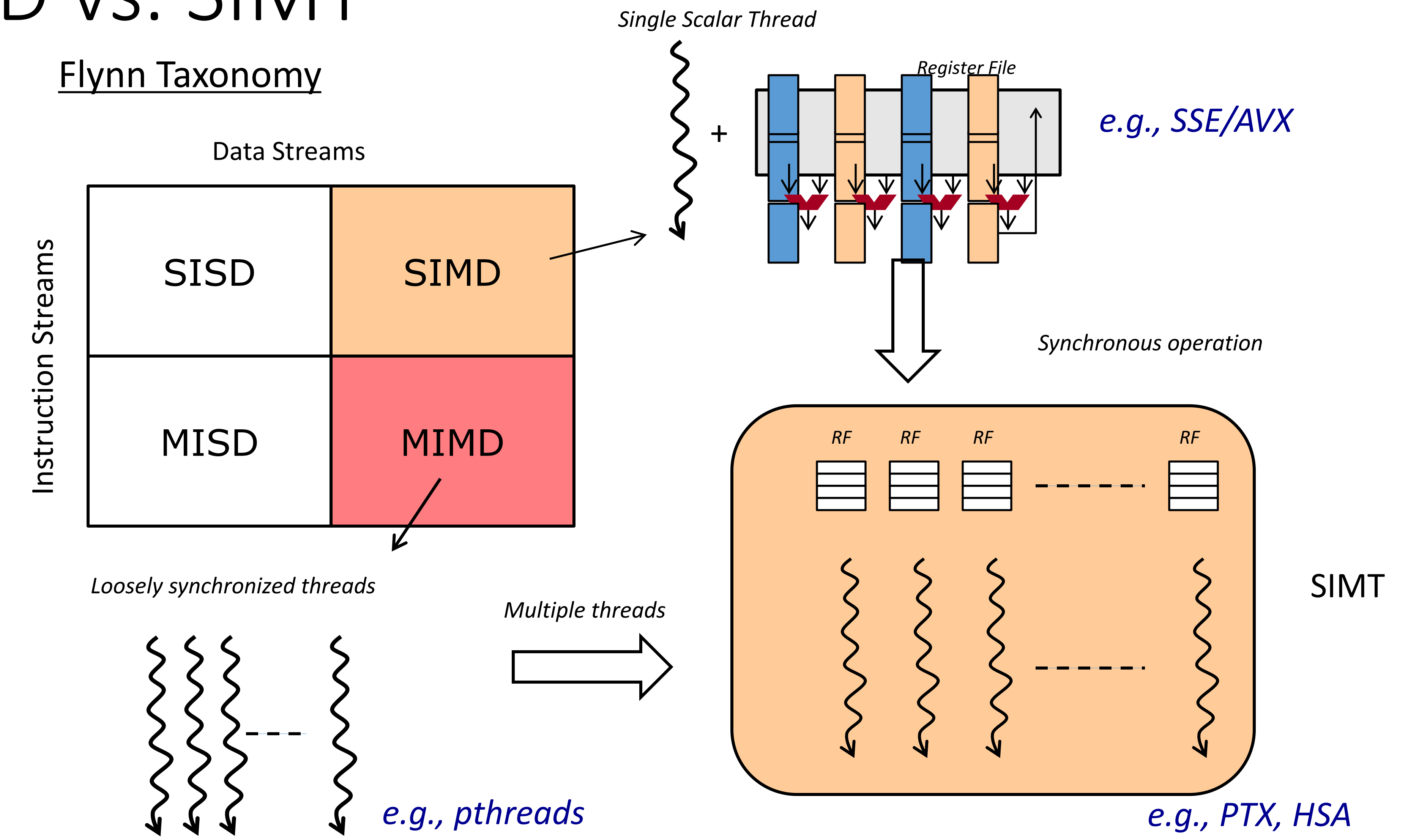


Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
  - Multiple streams mapped to each queue → serializes some kernels
- Kernel launch distributes thread blocks to SMs



SIMD vs. SIMT



Flynn Taxonomy
Data Streams
Instruction Streams
SISD
SIMD
MISD
MIMD
Single Scalar Thread
+
Register File
e.g., SSE/AVX
Synchronous operation
RF
RF
RF
RF
SIMT
Loosely synchronized threads
Multiple threads
e.g., pthreads
e.g., PTX, HSA

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

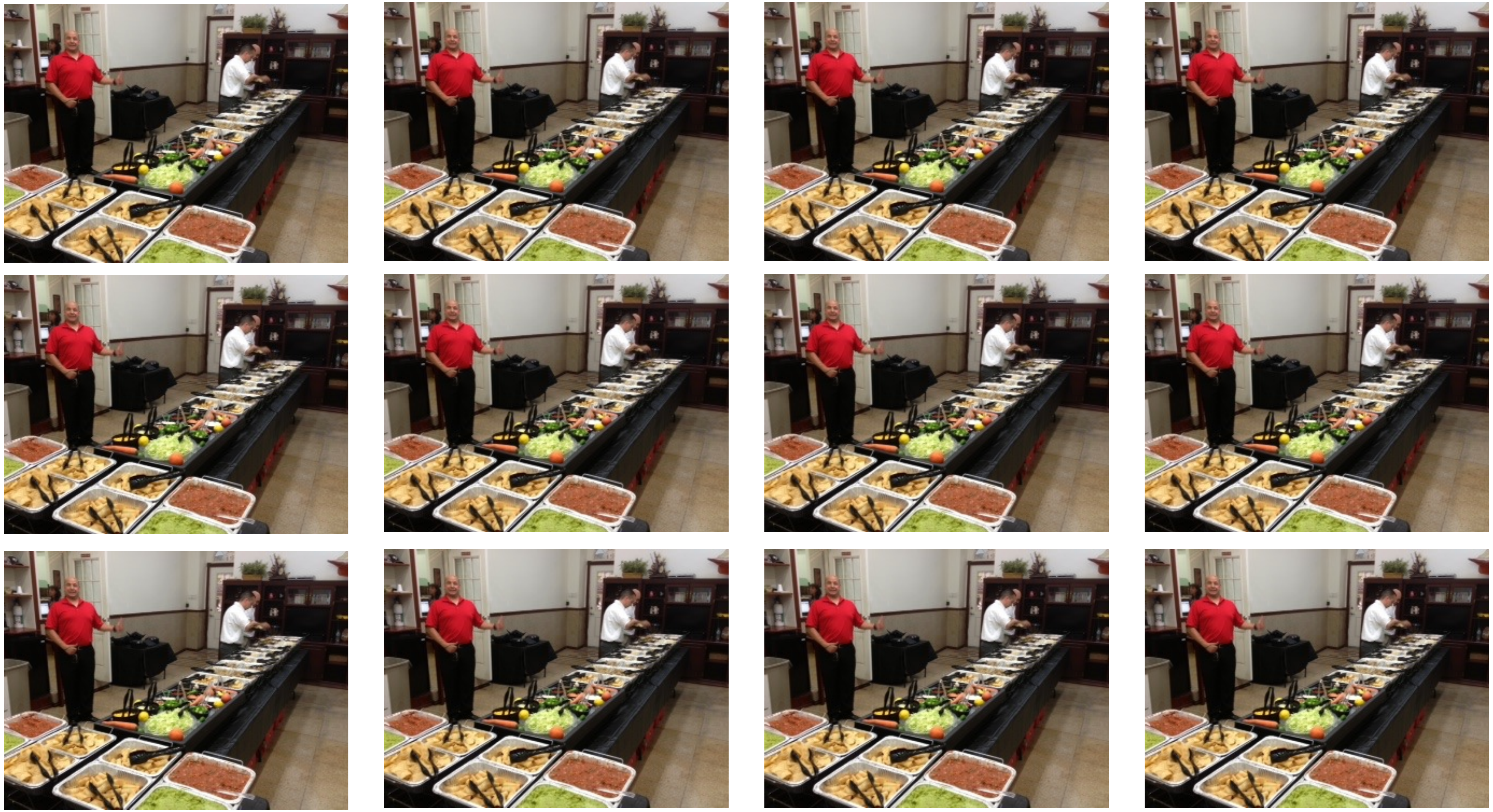
GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

GPU: a multi-lane Taco Bar



GPU: a multi-lane Taco Bar

GPU: a multi-lane Taco Bar

- Where is the parallelism here?

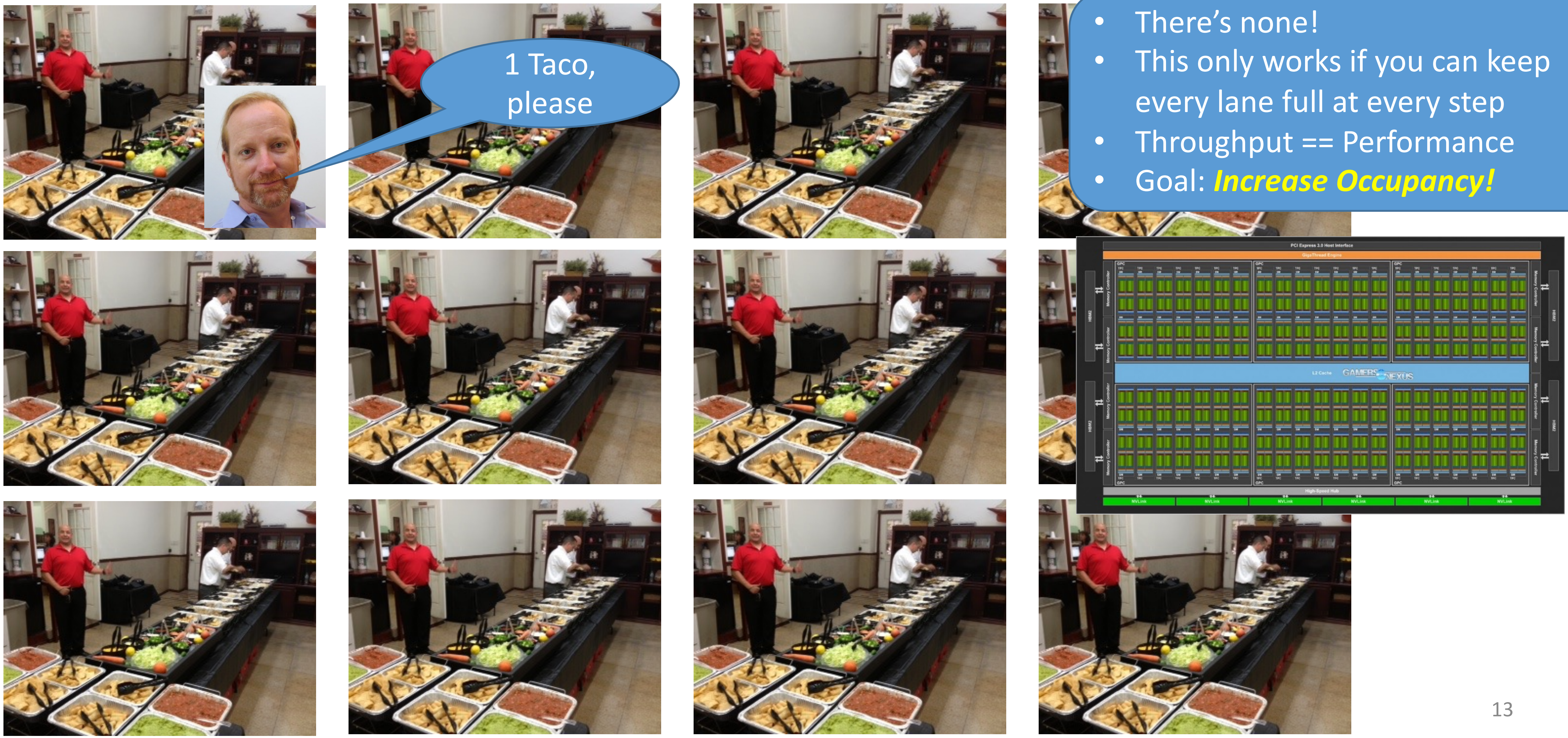
GPU: a multi-lane Taco Bar

- 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!***

GPU: a multi-lane Taco Bar

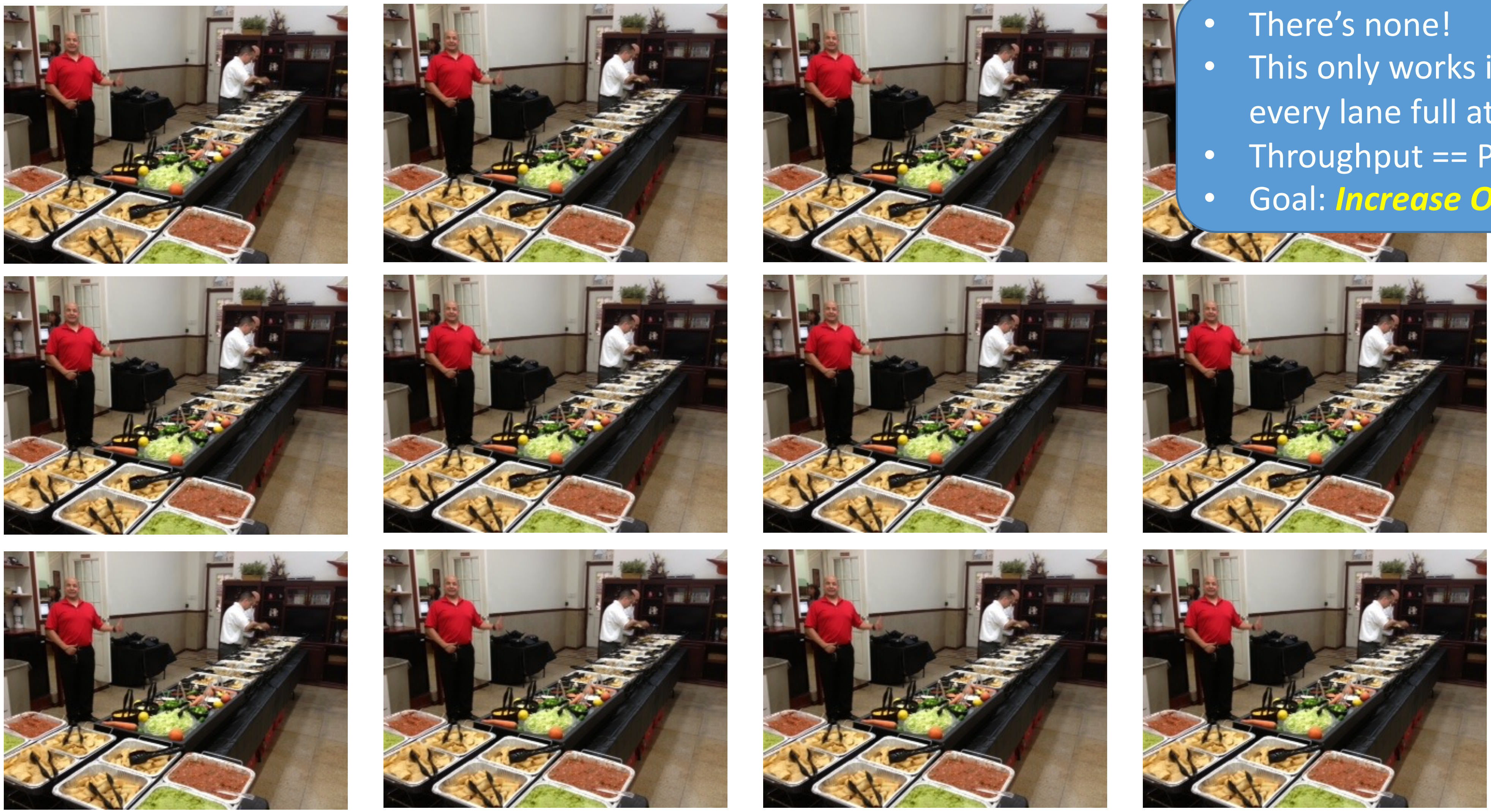
- Where is the parallelism here?



- 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

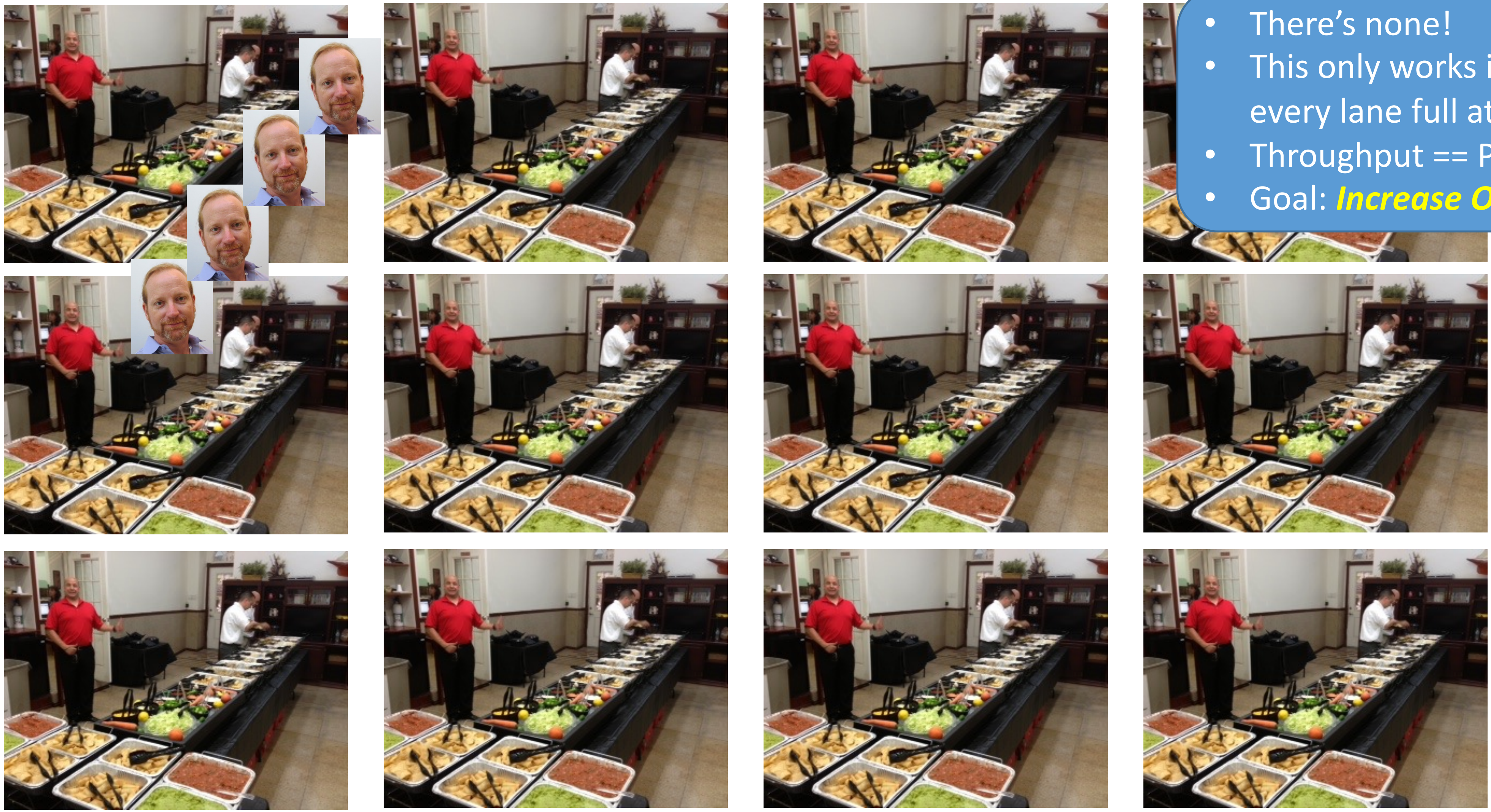
- Where is the parallelism here?



- 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

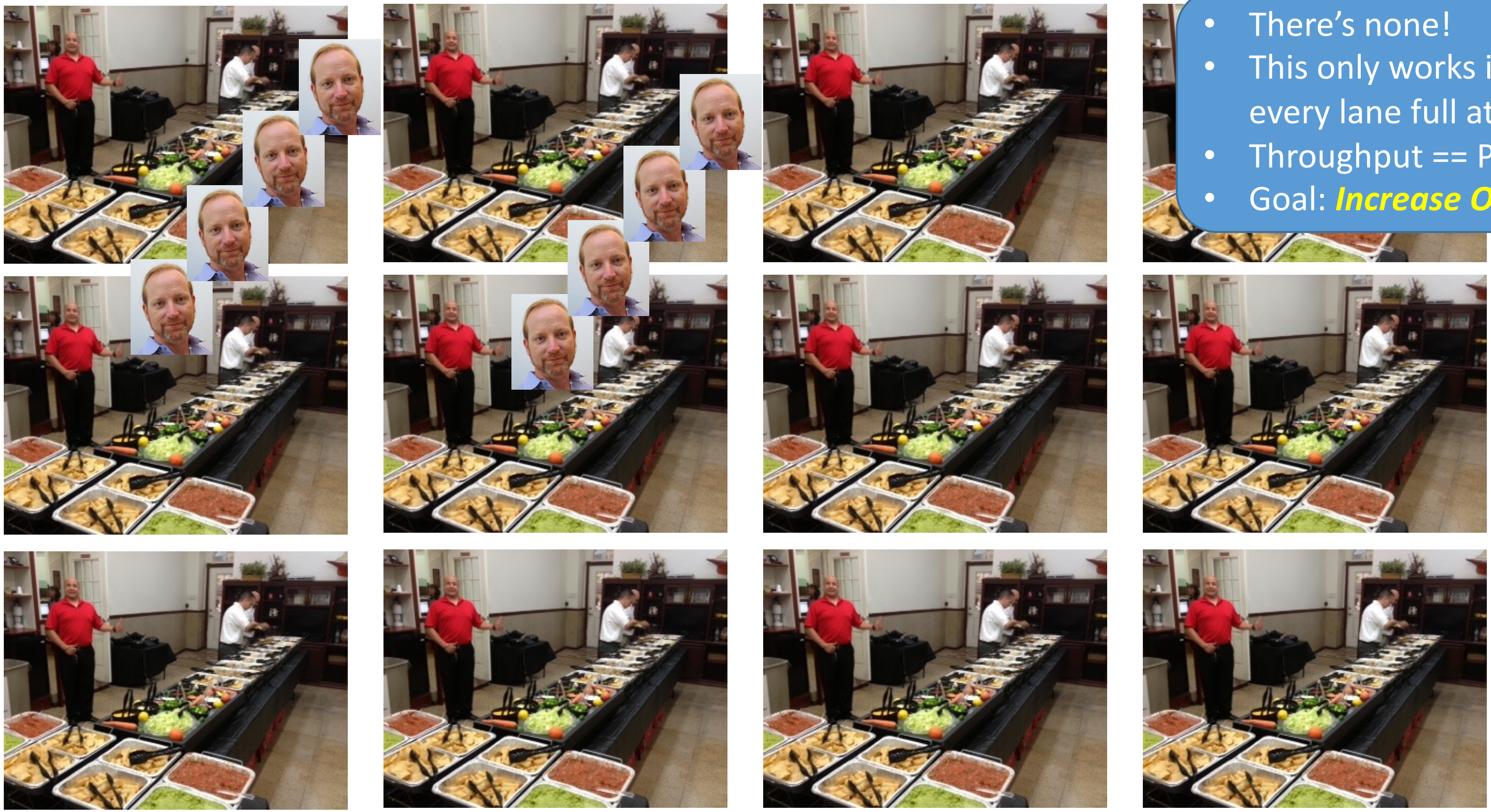
- Where is the parallelism here?



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

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!***

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 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*

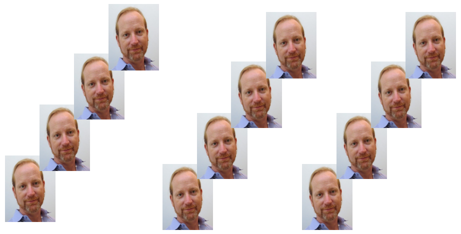
- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

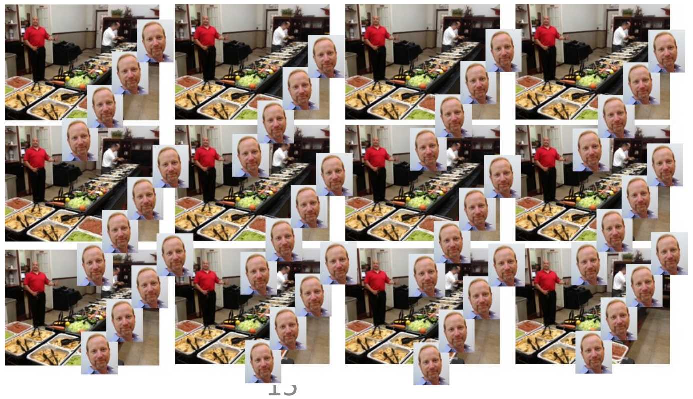
GPU Performance Metric: *Occupancy*

- 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

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*

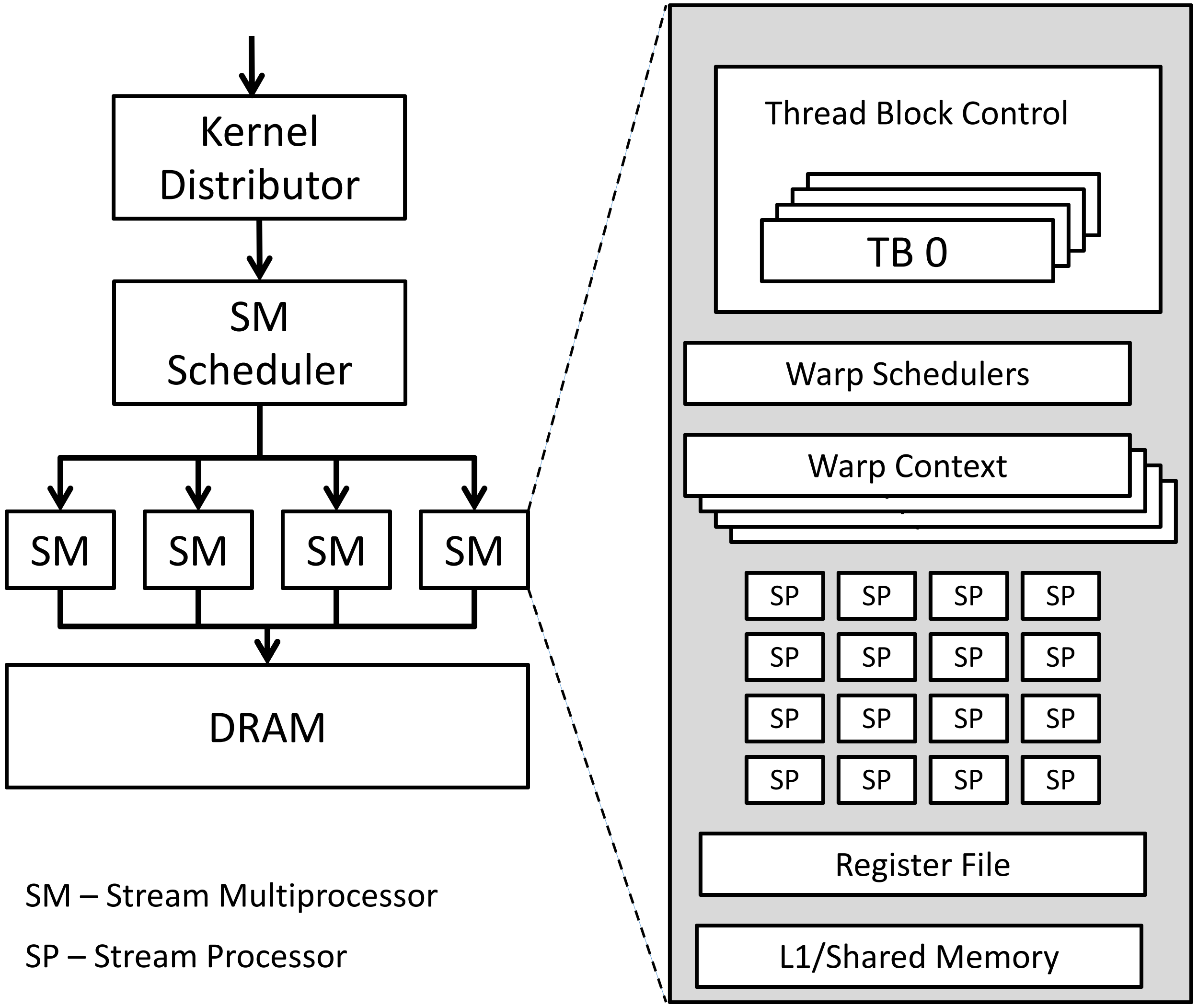
- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

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?

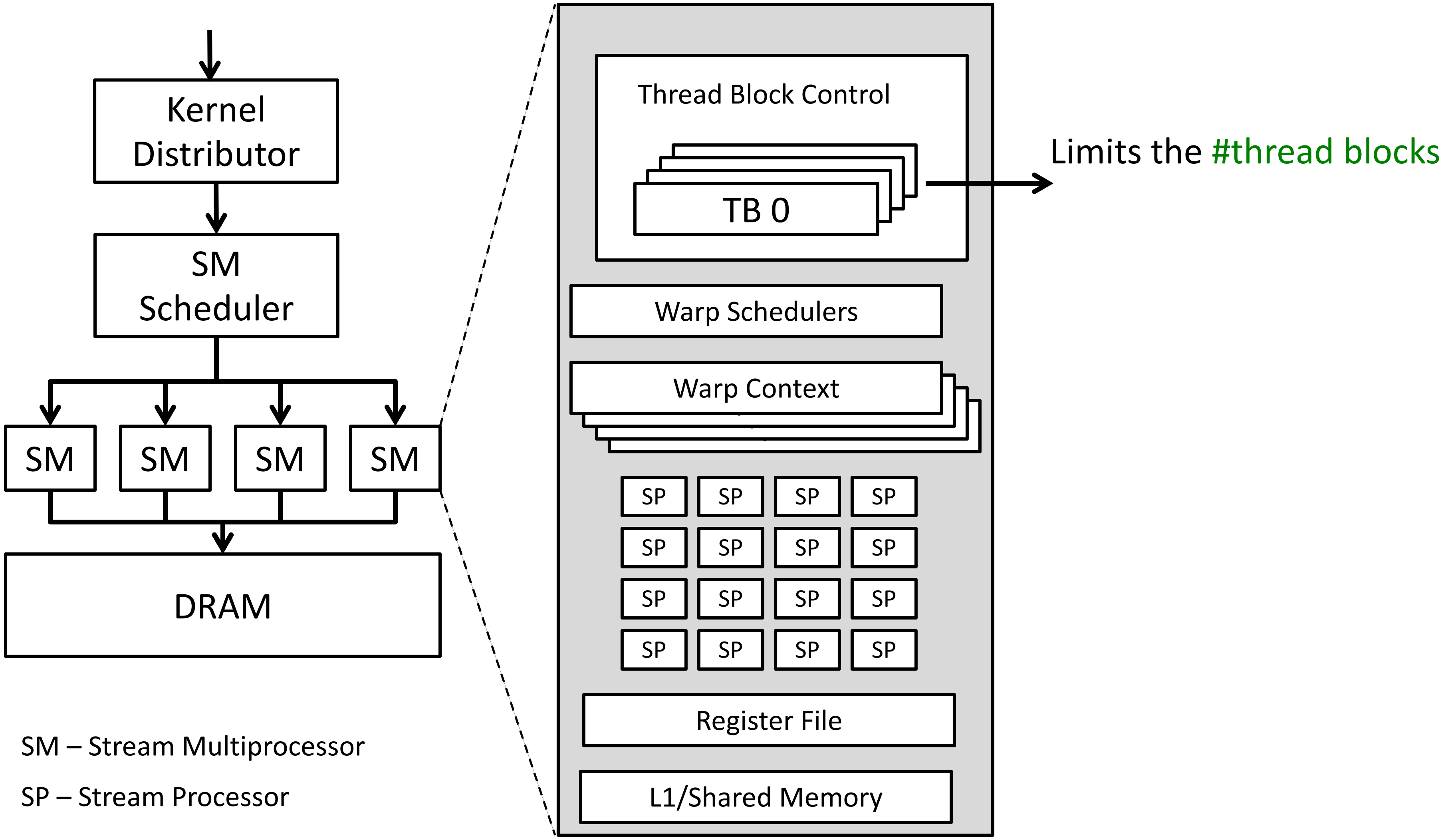
Hardware Resources Are Finite



SM – Stream Multiprocessor

SP – Stream Processor

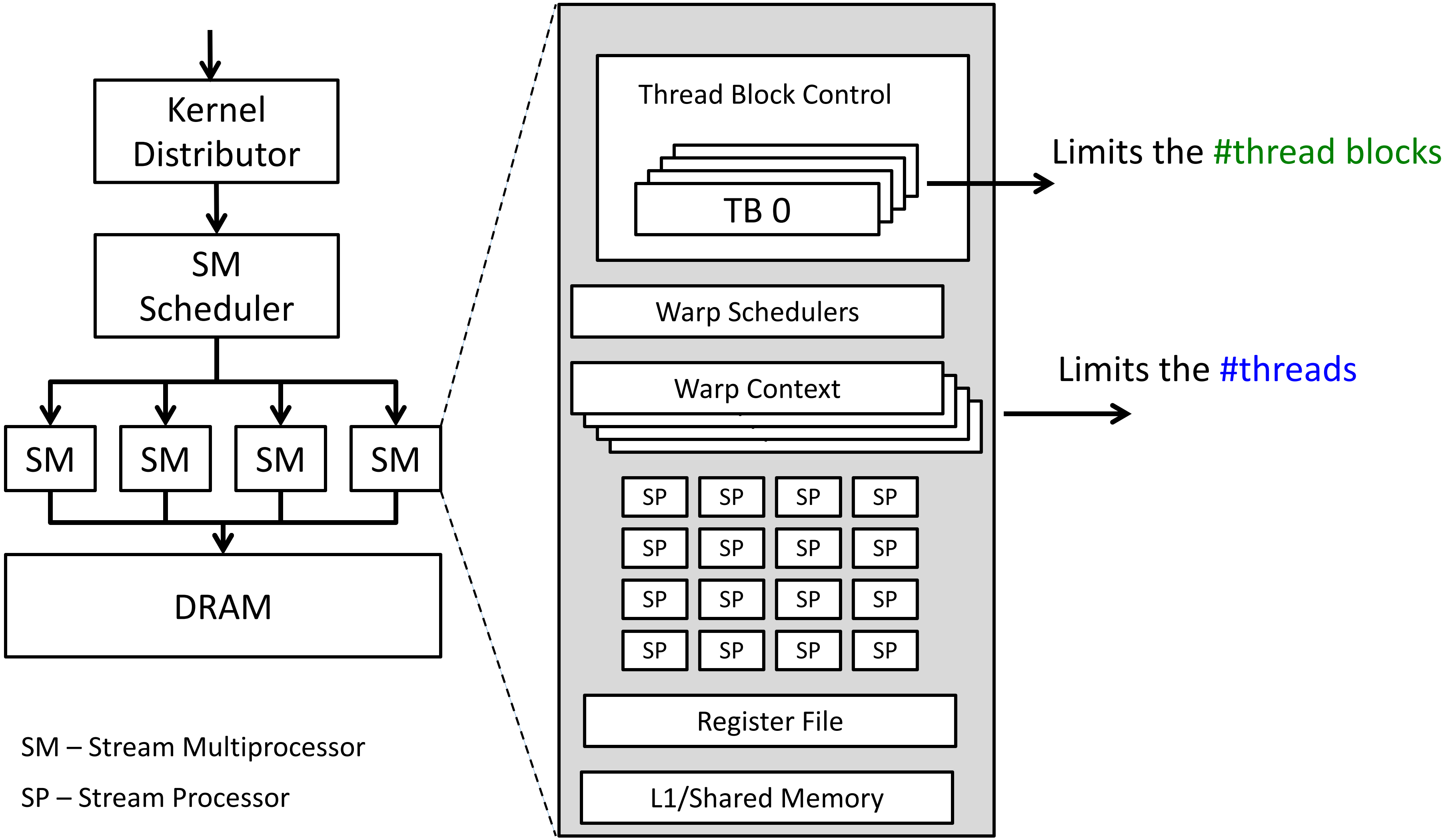
Hardware Resources Are Finite



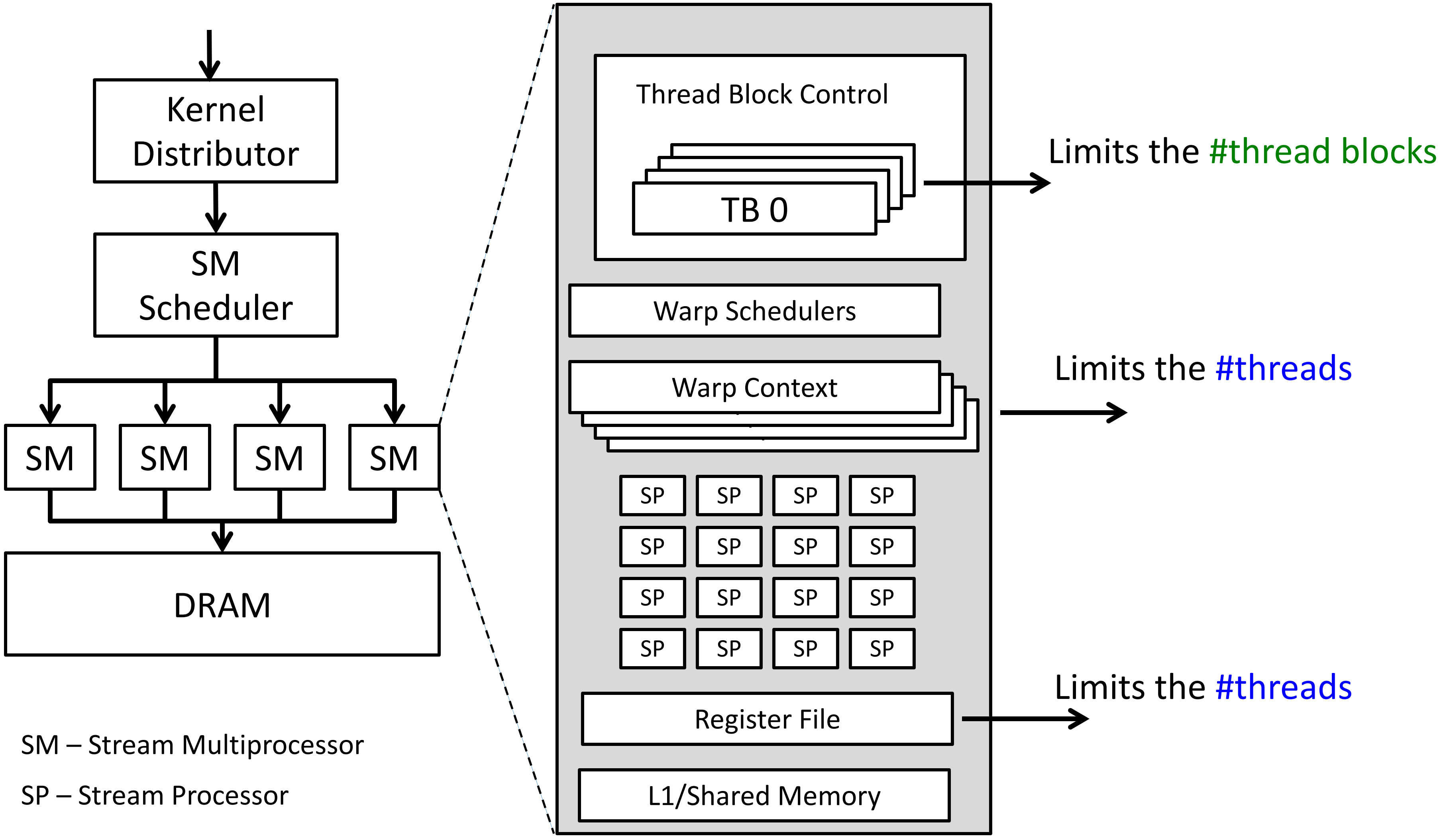
SM – Stream Multiprocessor

SP – Stream Processor

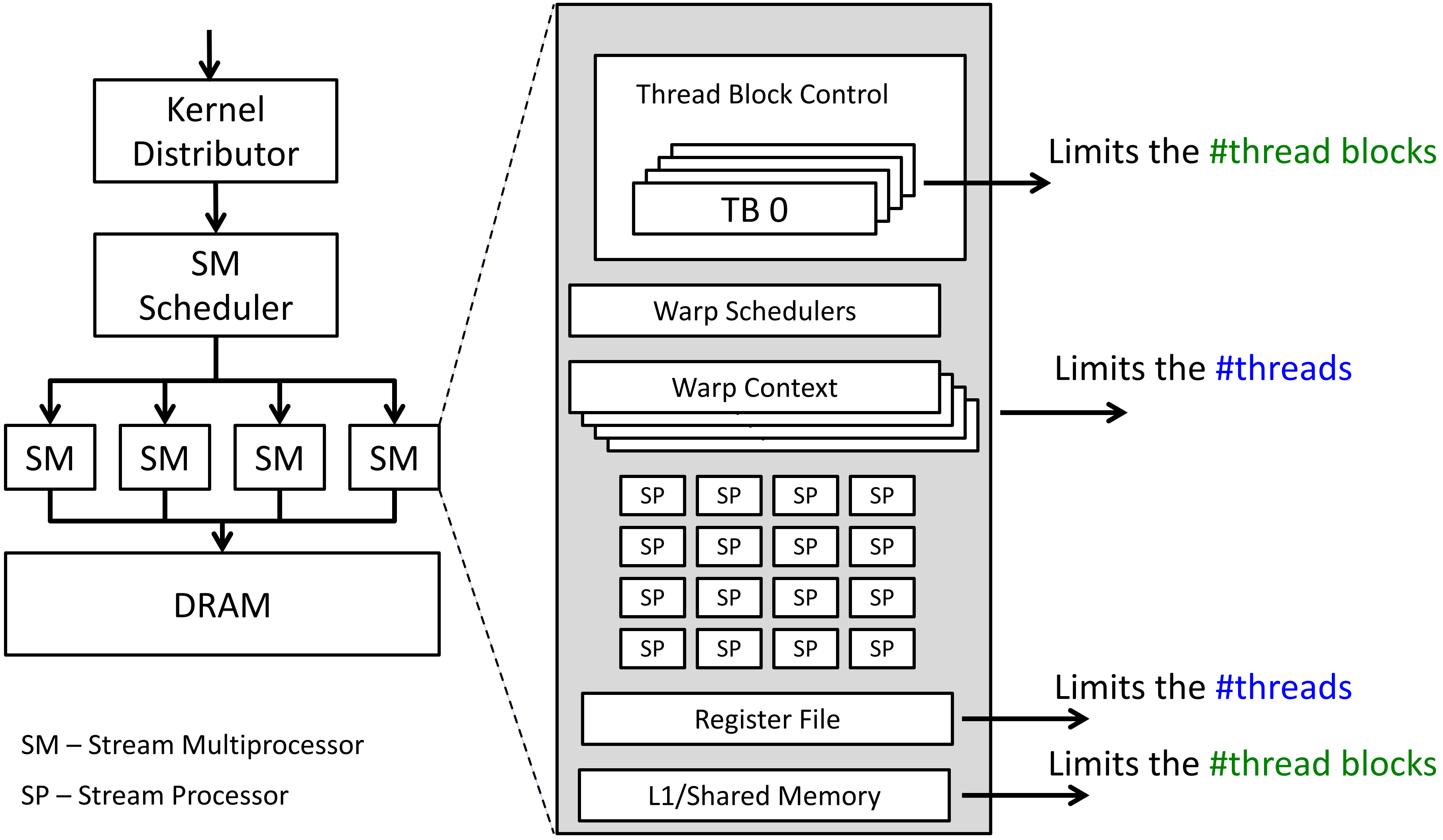
Hardware Resources Are Finite



Hardware Resources Are Finite



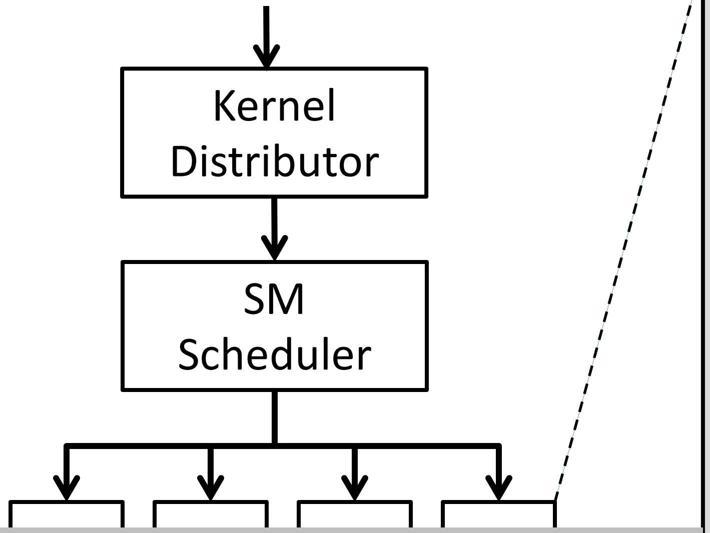
Hardware Resources Are Finite

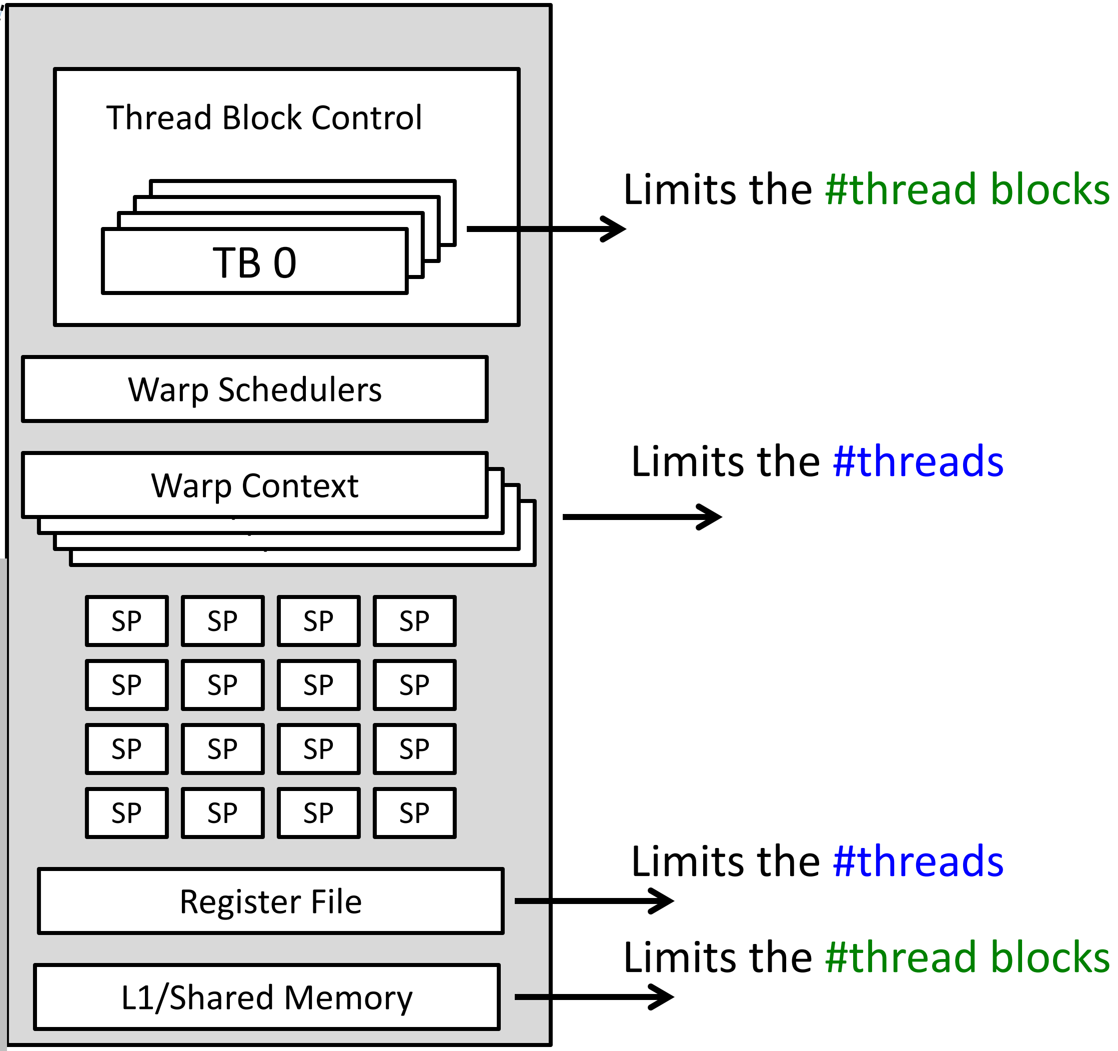


SM – Stream Multiprocessor

SP – Stream Processor

Hardware Resources Are Finite

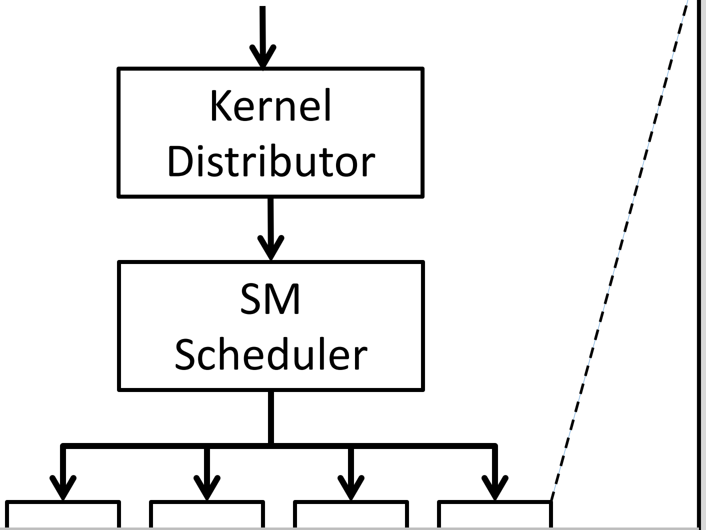


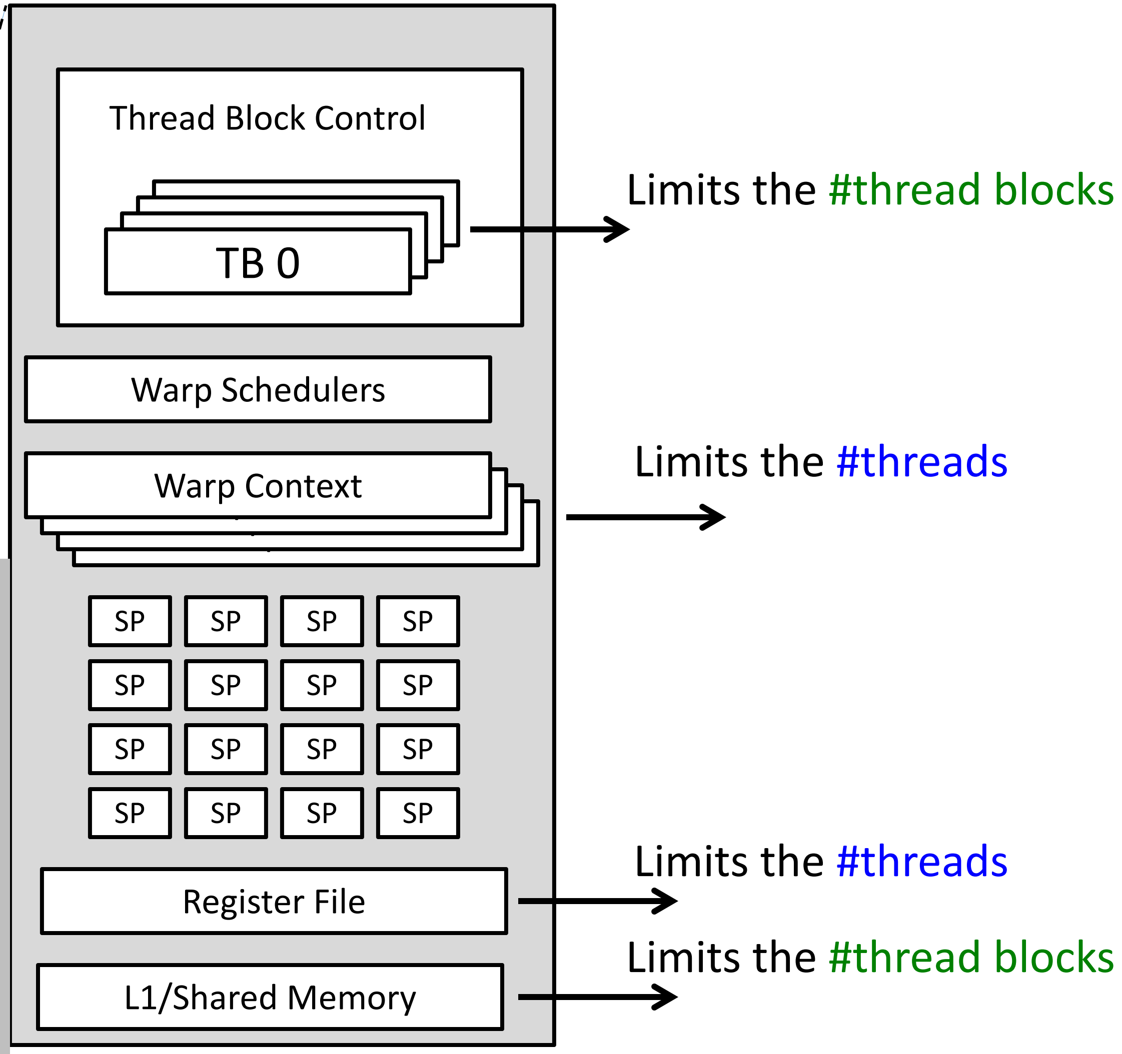


Occupancy:

- (#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



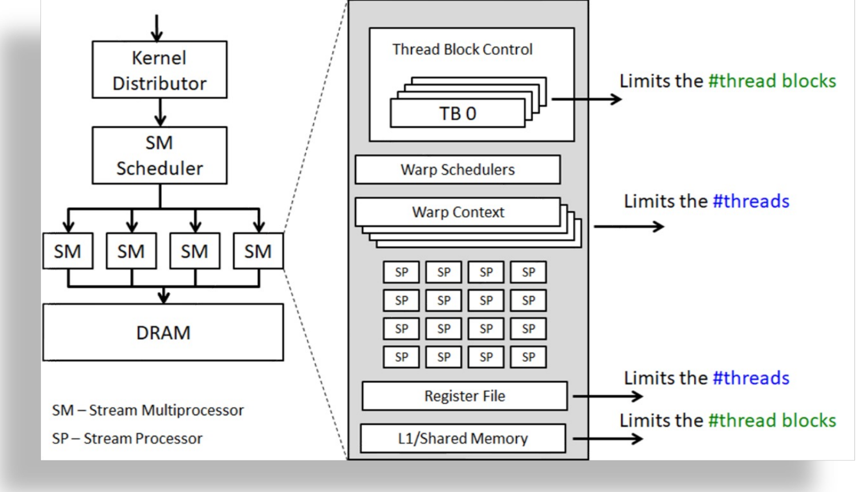


Occupancy:

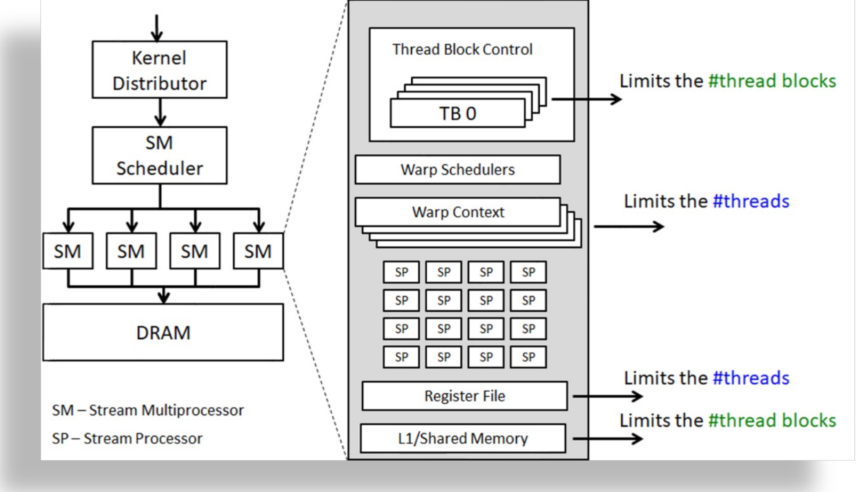
- (#Active Warps) /(#MaximumActive Warps)
- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps
- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots

What is the performance impact of varying kernel resource demands?

Impact of Thread Block Size

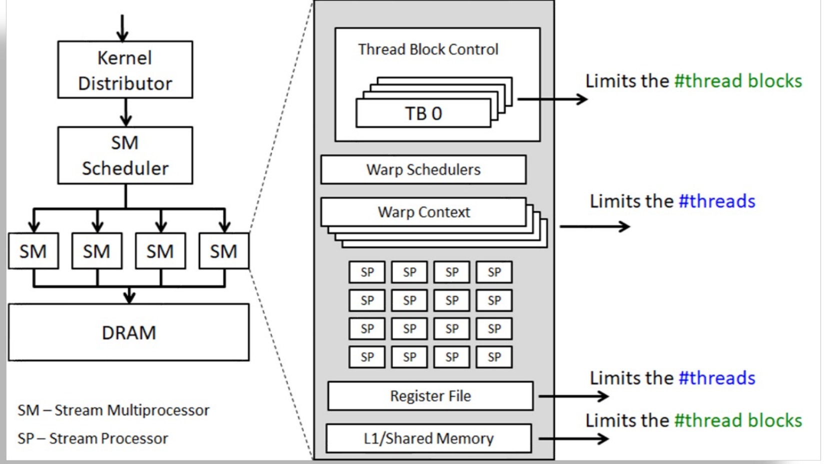


Impact of Thread Block Size



Example: v100:

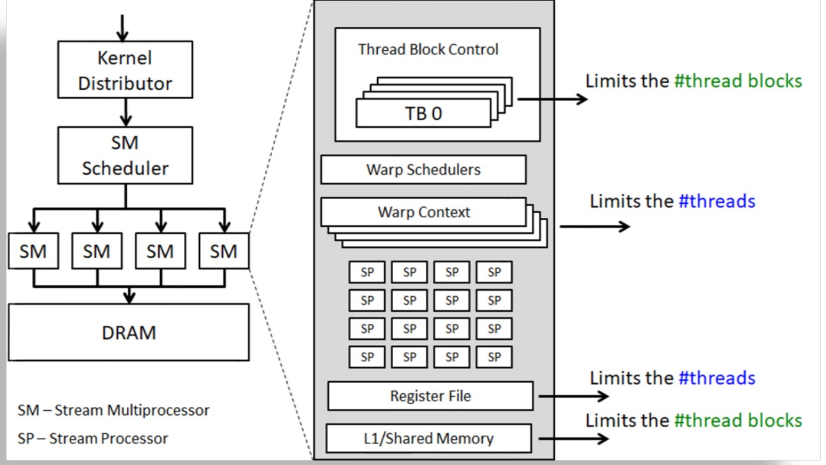
Impact of Thread Block Size



Example: v100:

- max active warps/SM == 64 (limit: warp context)

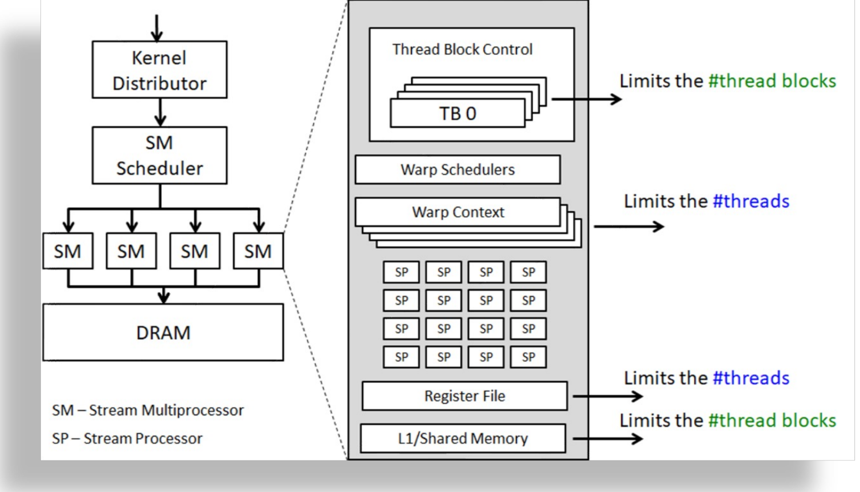
Impact of Thread Block Size



Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)

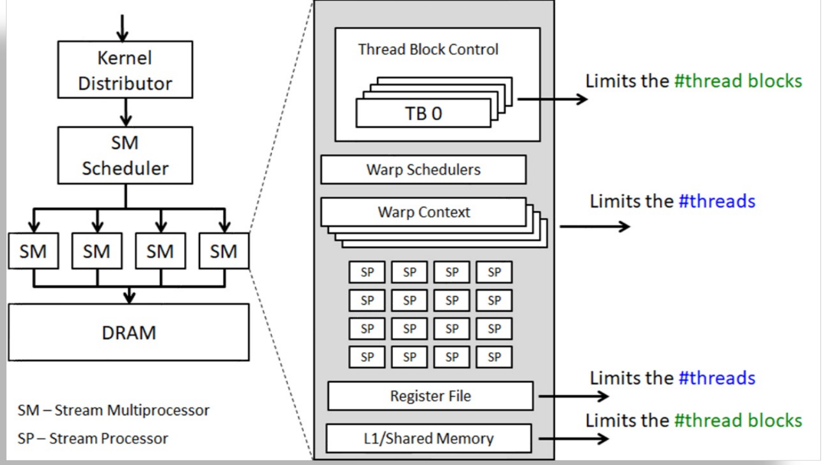
Impact of Thread Block Size



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 →

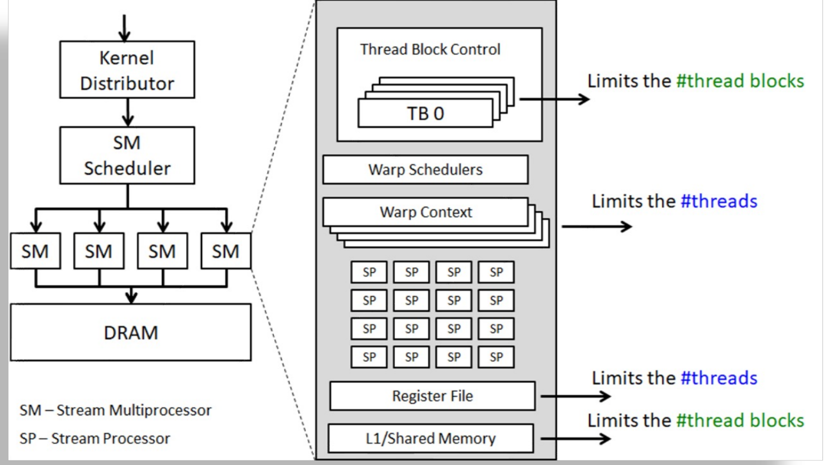
Impact of Thread Block Size



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

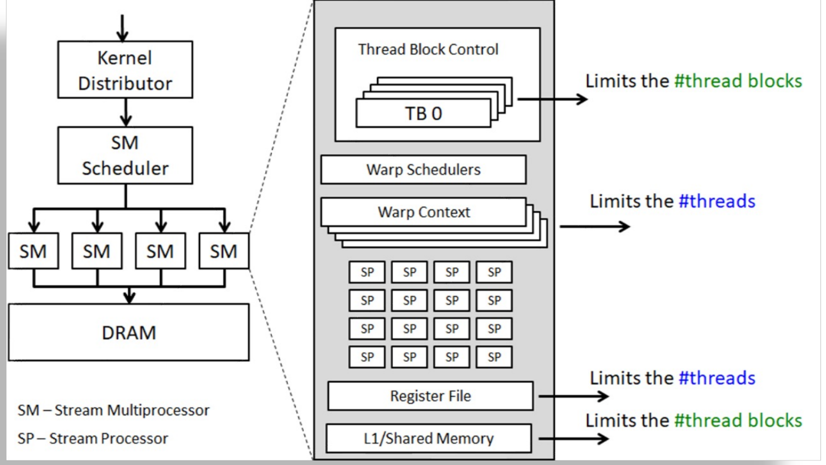
Impact of Thread Block Size



Example: v100:

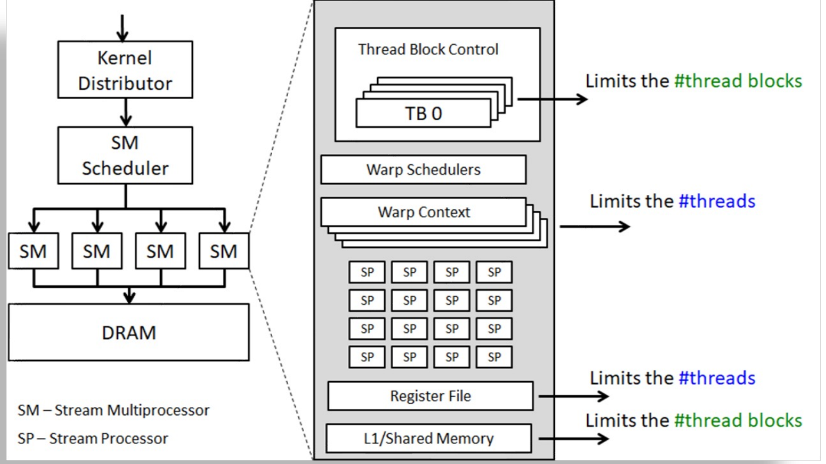
- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
  - With 128 threads/block? →

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

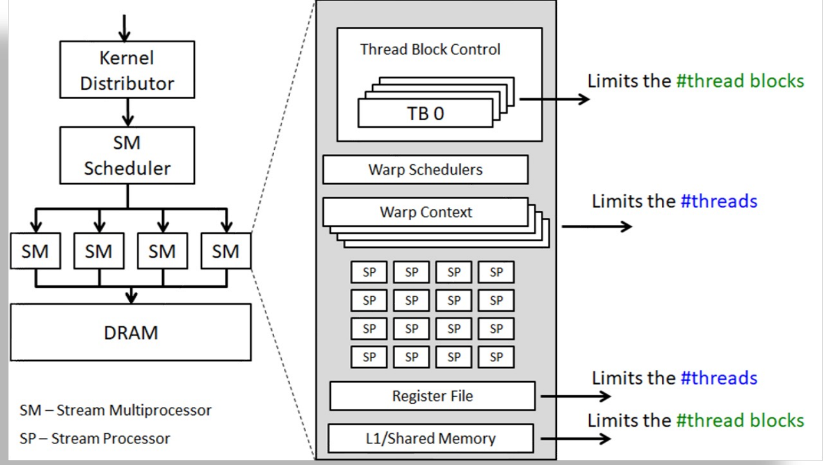


Impact of Thread Block Size

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)

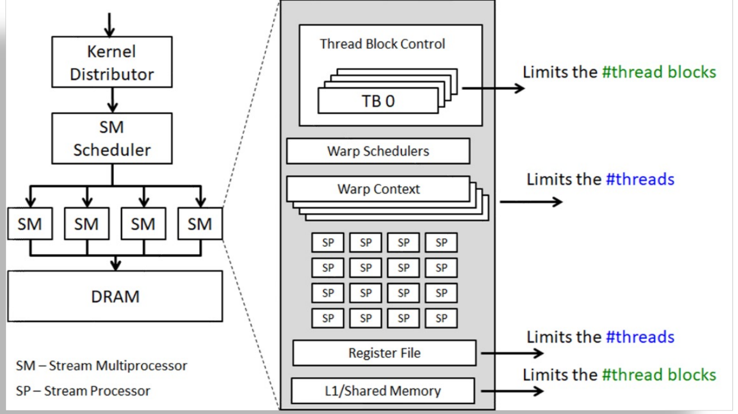
Impact of Thread Block Size



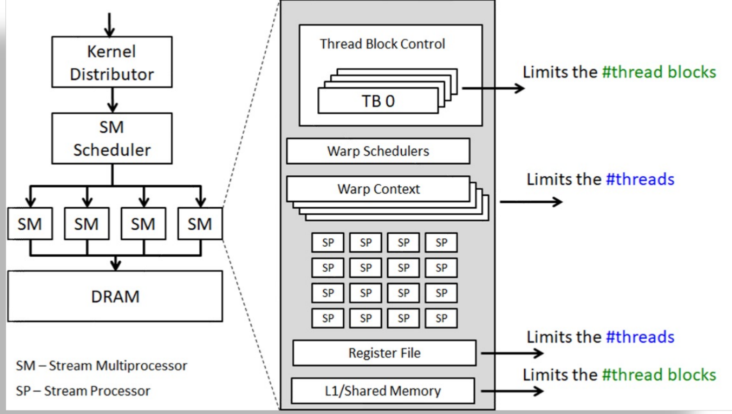
Example: v100:

- 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

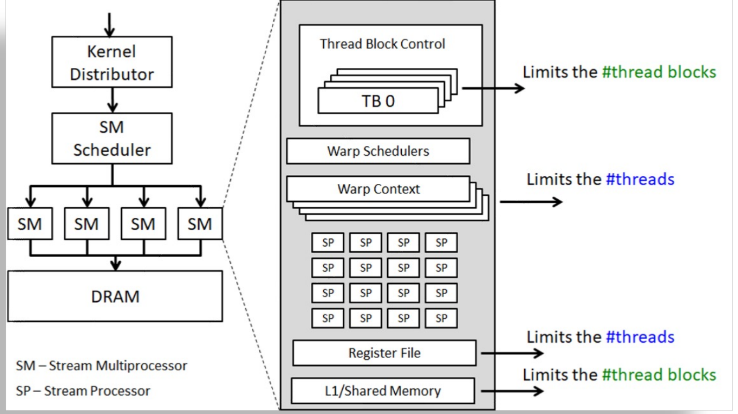


Impact of #Registers Per Thread



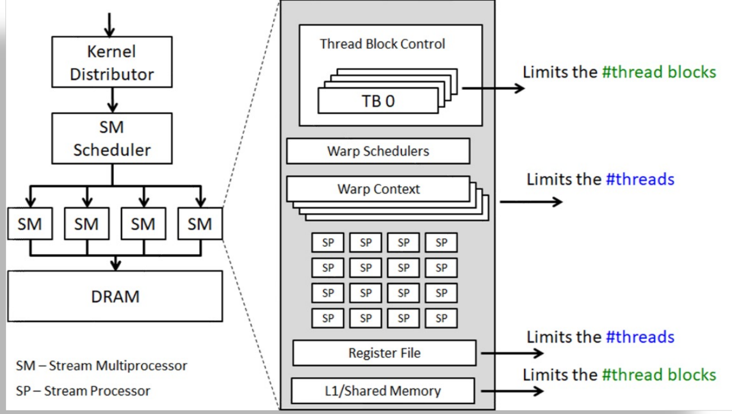
Registers/thread can limit number of active threads!

Impact of #Registers Per Thread



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

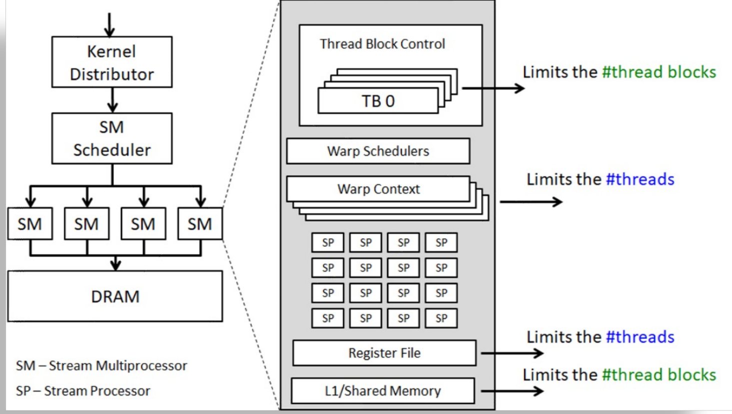
Impact of #Registers Per Thread



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

- Registers per thread max: 255

Impact of #Registers Per Thread

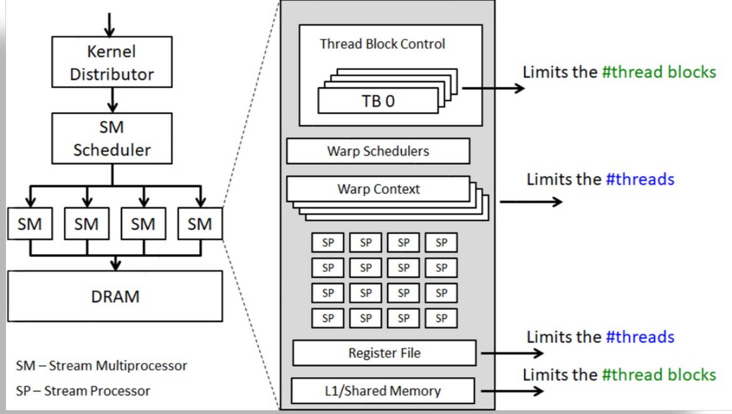


Registers/thread can limit number of active threads!

V100:

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

Impact of #Registers Per Thread



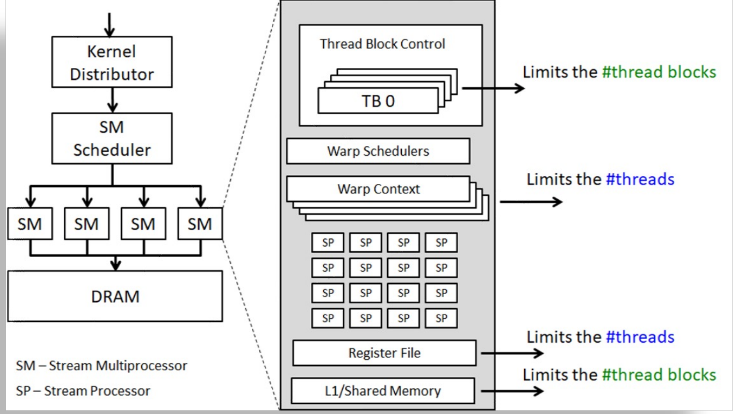
Registers/thread can limit number of active threads!

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!

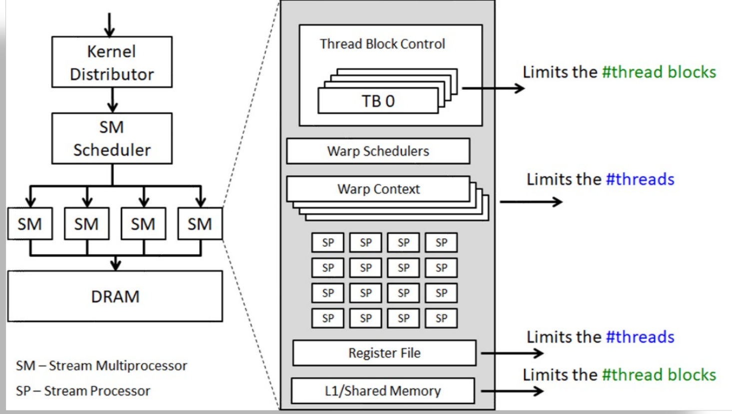
V100:

- 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!

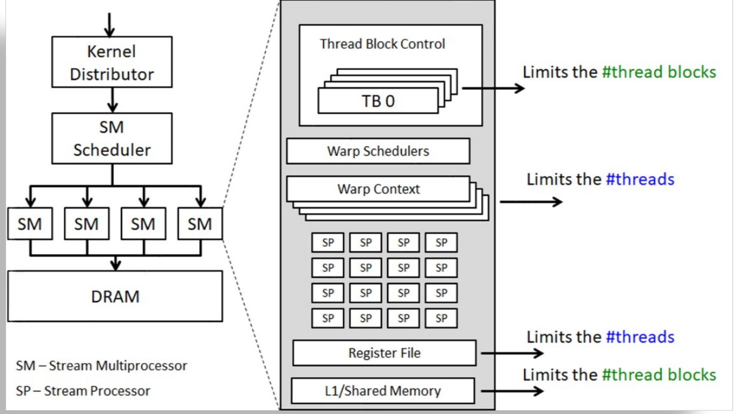
V100:

- 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!

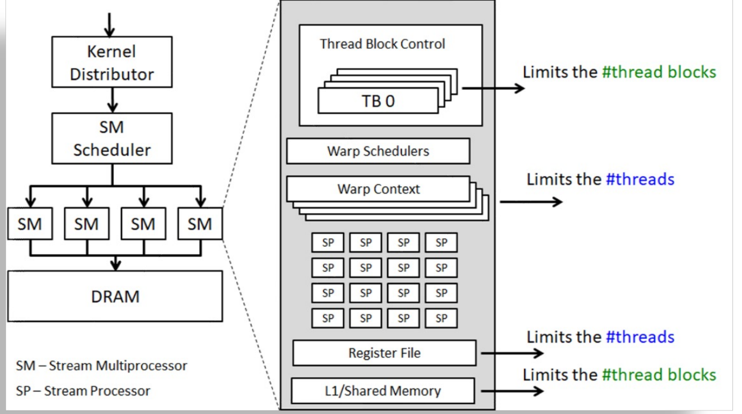
V100:

- 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

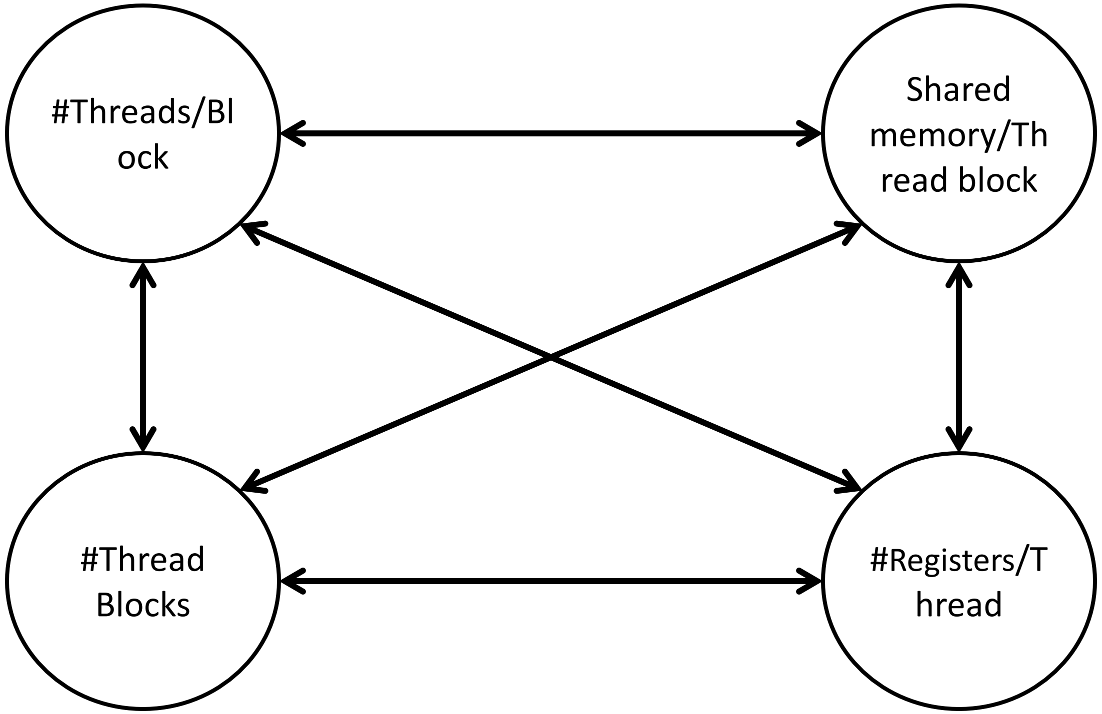
Assume a kernel uses 32 registers/thread, thread block size of 256

- 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

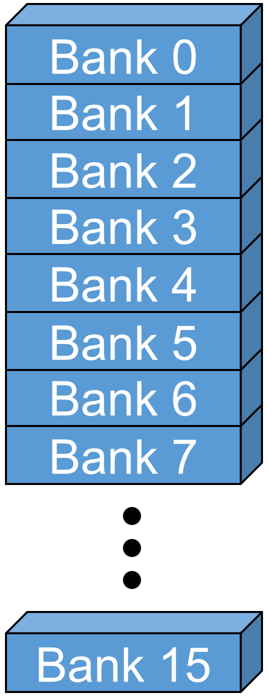
- Navigate the tradeoffs
  - maximize core utilization and memory bandwidth utilization
  - Device-specific
- Goal: Increase occupancy until one or the other is saturated

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 1.3: 16 different banks per half warp or same word
    - CC 2.x+3.0 : 32 different banks + 1-word broadcast each

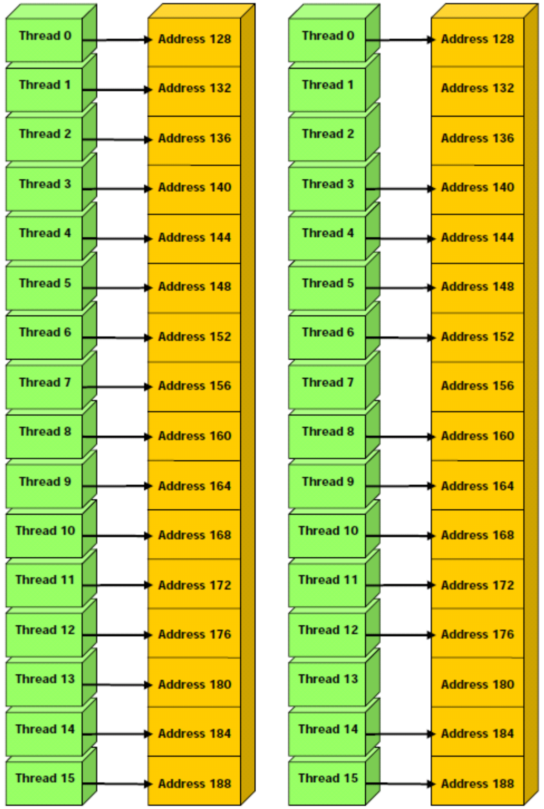
Parallel Memory Architecture

- 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



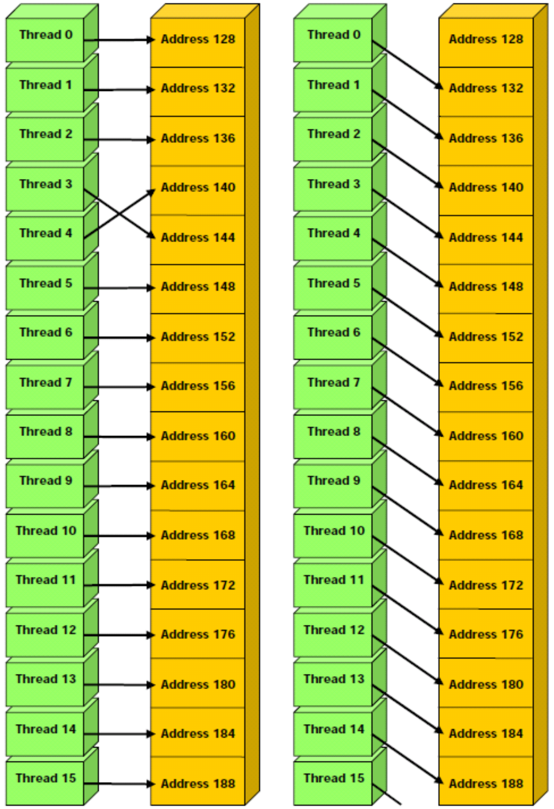
Coalesced Main Memory Accesses

single coalesced access



NVIDIA

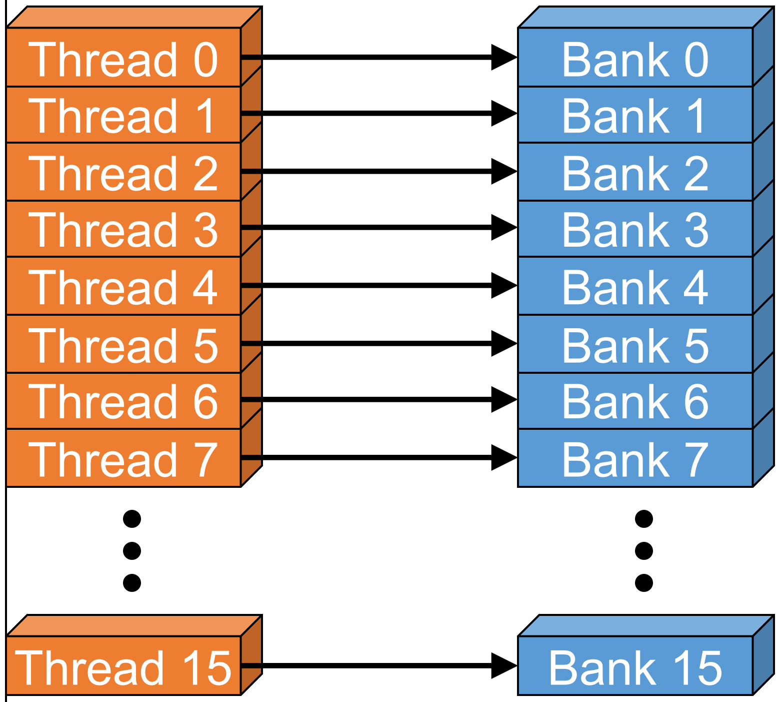
one and two coalesced accesses*



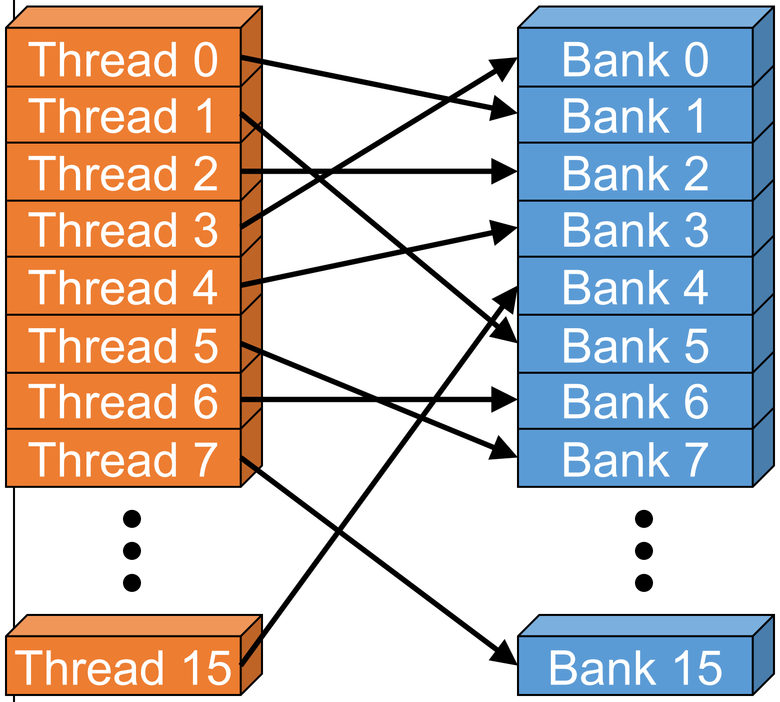
NVIDIA

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

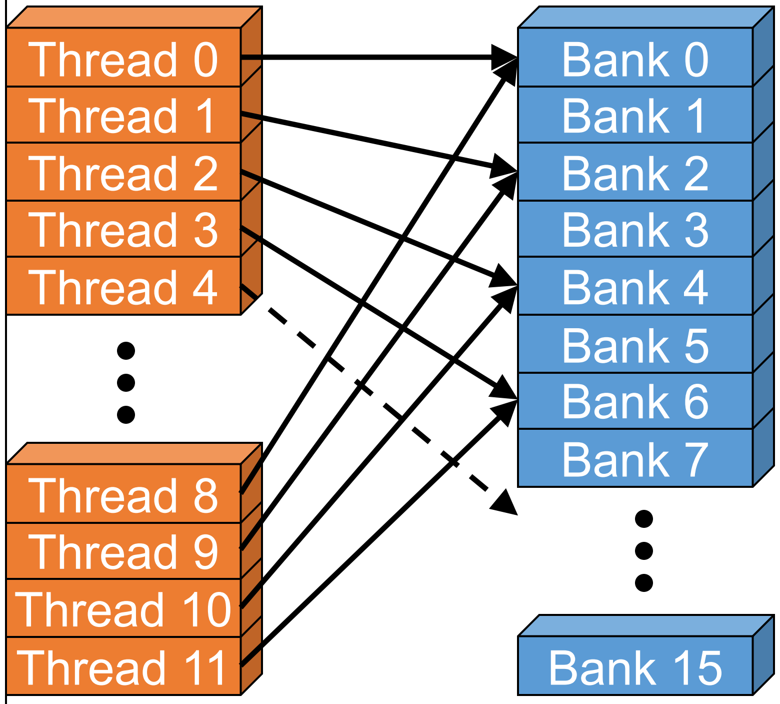


- No Bank Conflicts
  - Random 1:1 Permutation

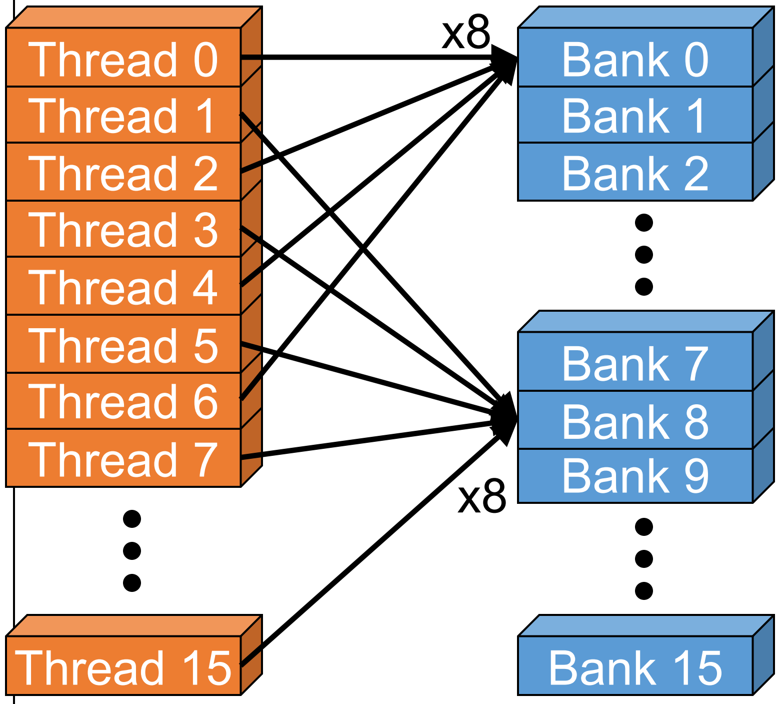


Bank Addressing Examples

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



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

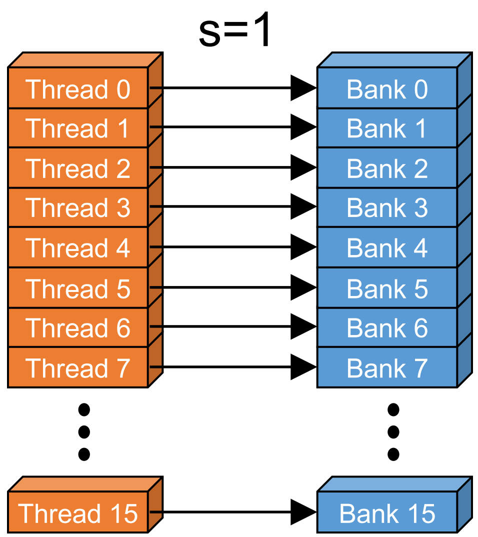


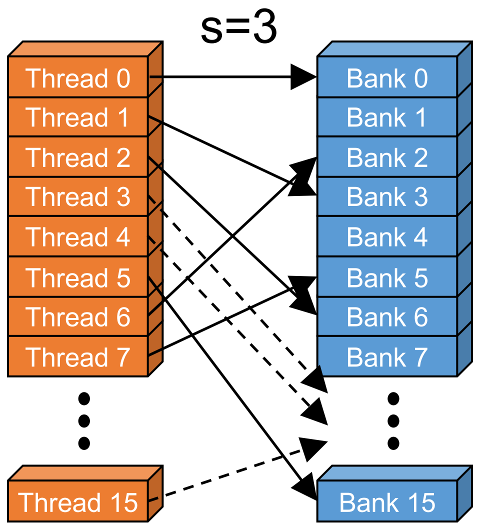
Linear Addressing

- Given:

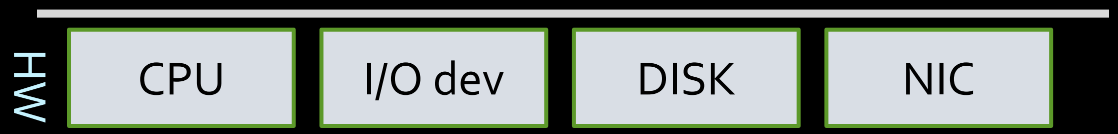
```
__shared__ float shared[256];
float foo =
   shared[baseIndex + s *
    threadIdx.x];
```

- 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



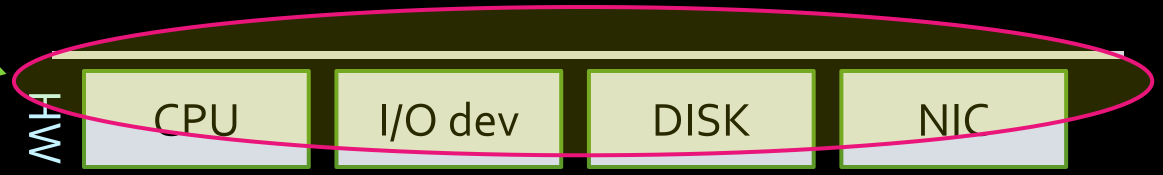


Layered abstractions

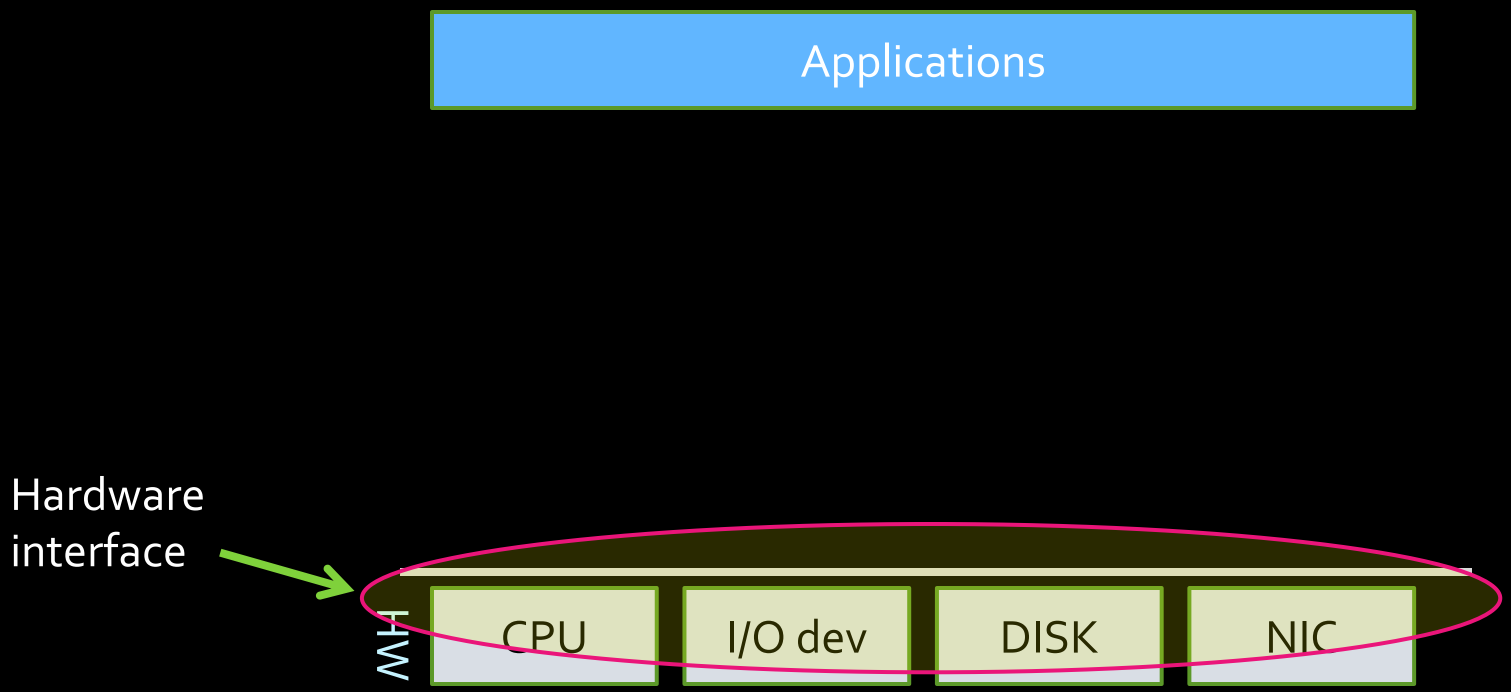


Layered abstractions

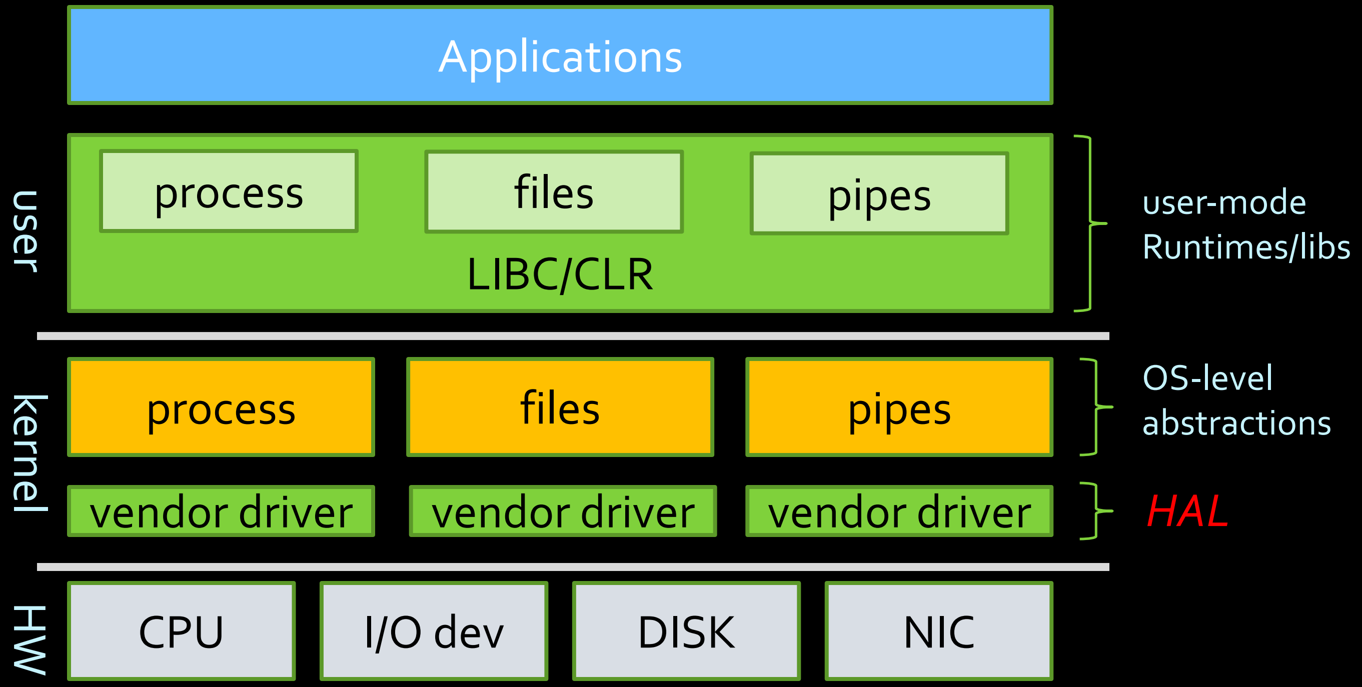
Hardware interface



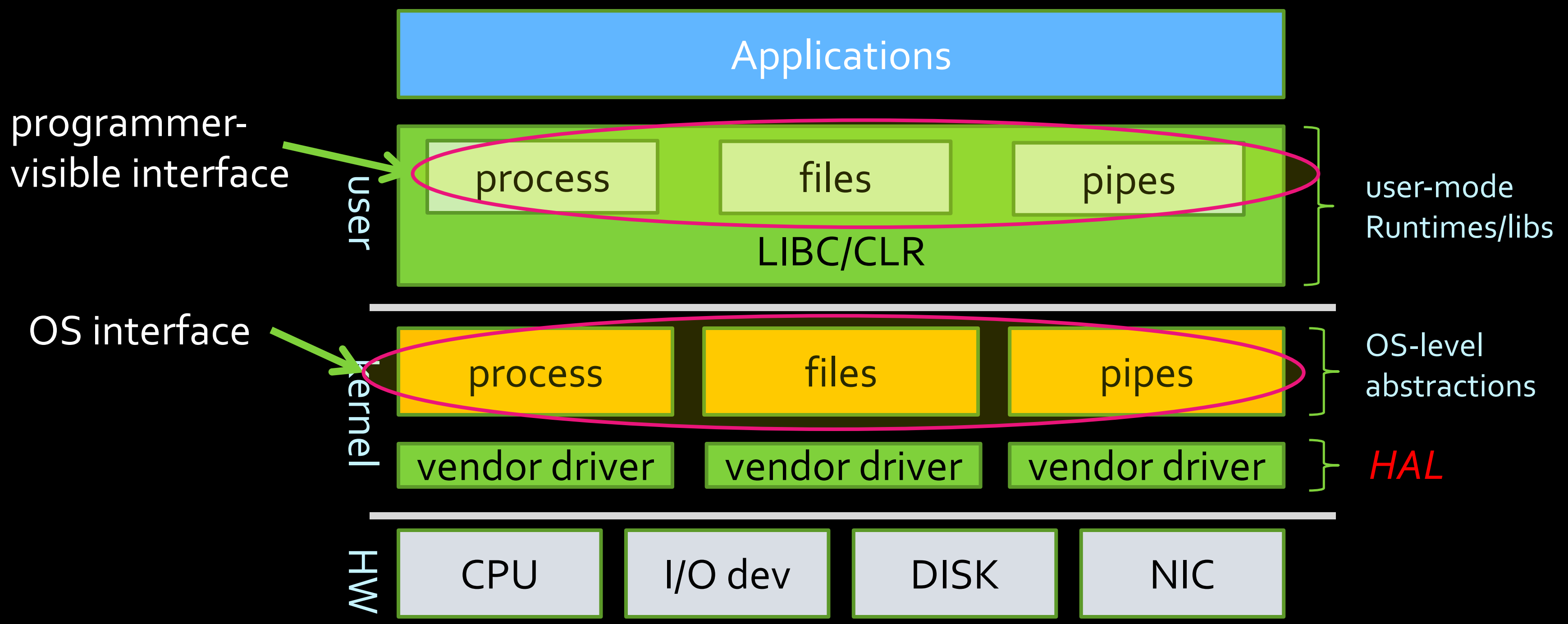
Layered abstractions



Layered abstractions

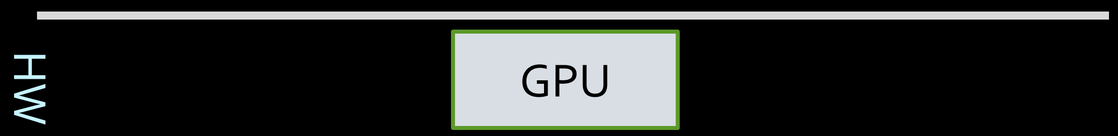


Layered abstractions

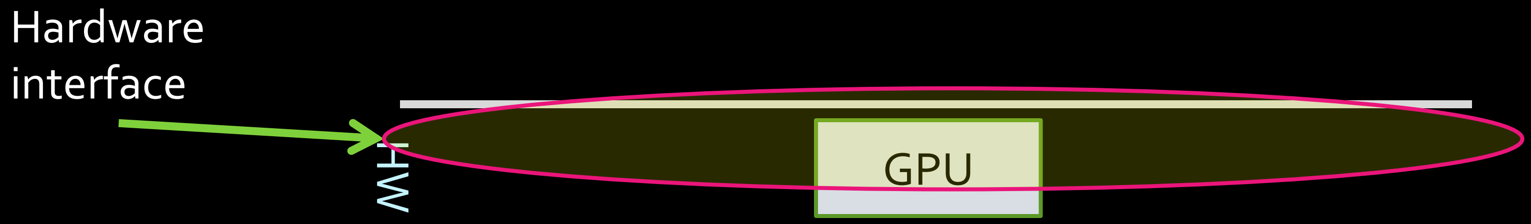


** 1:1 correspondence between OS-level and user-level abstractions*
** Diverse HW support enabled HAL*

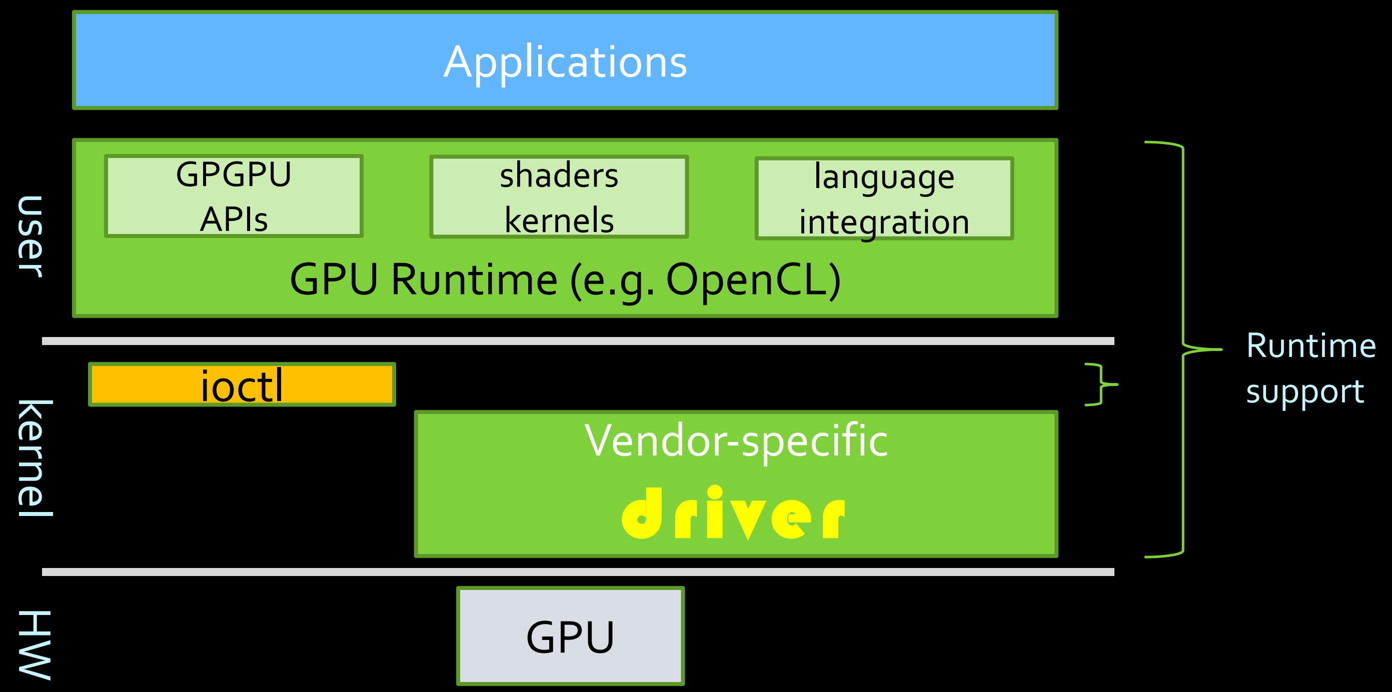
GPU abstractions



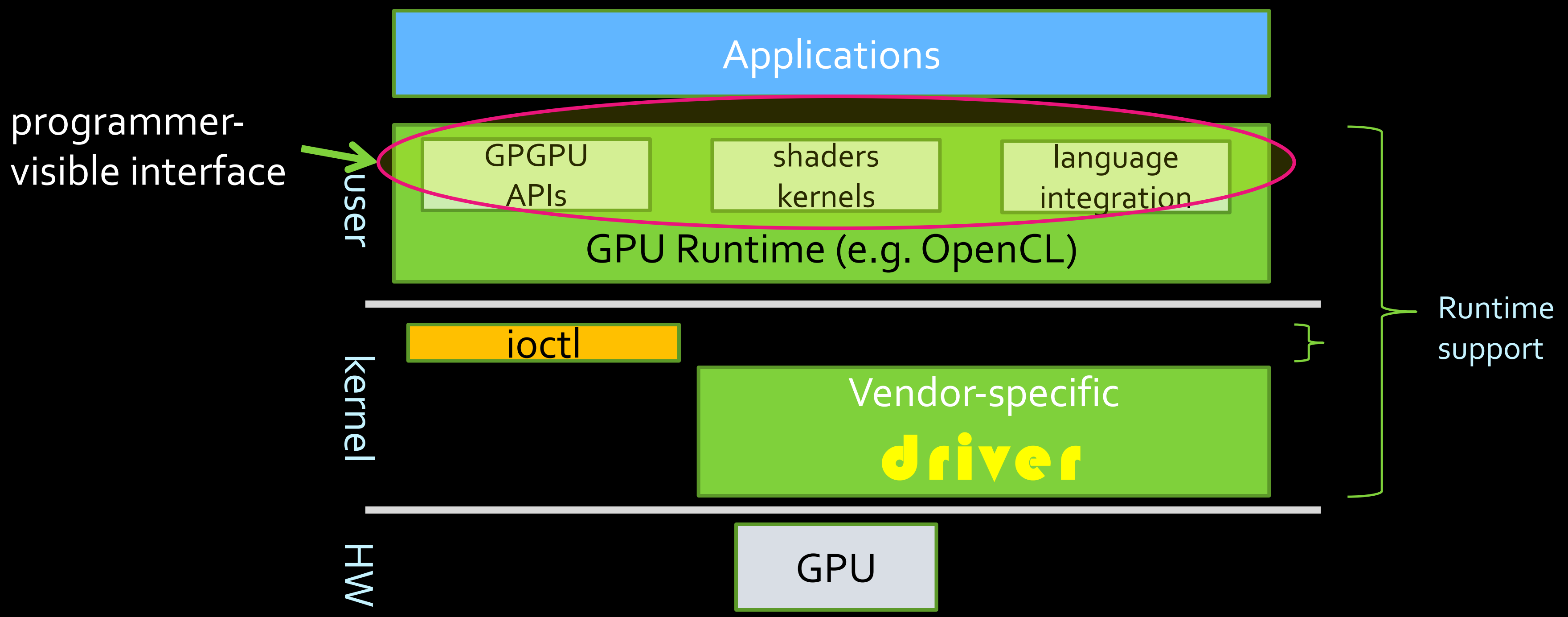
GPU abstractions



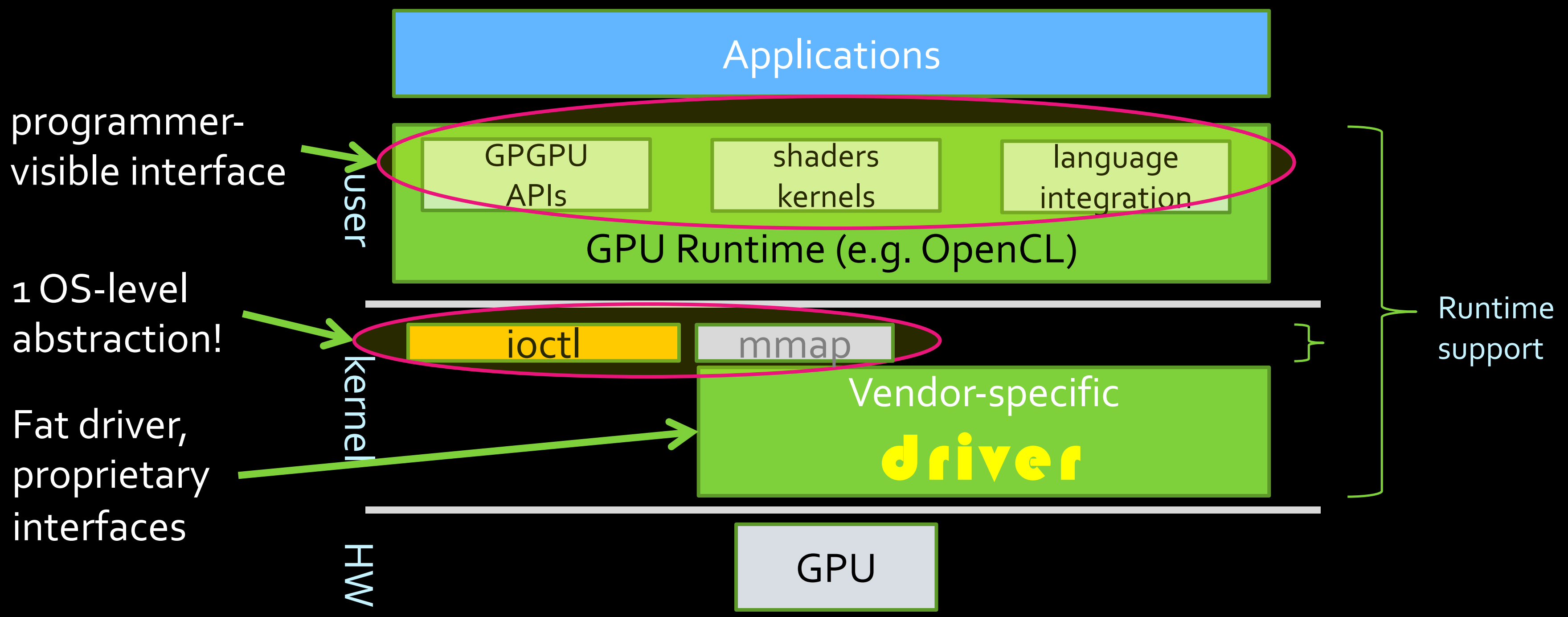
GPU abstractions



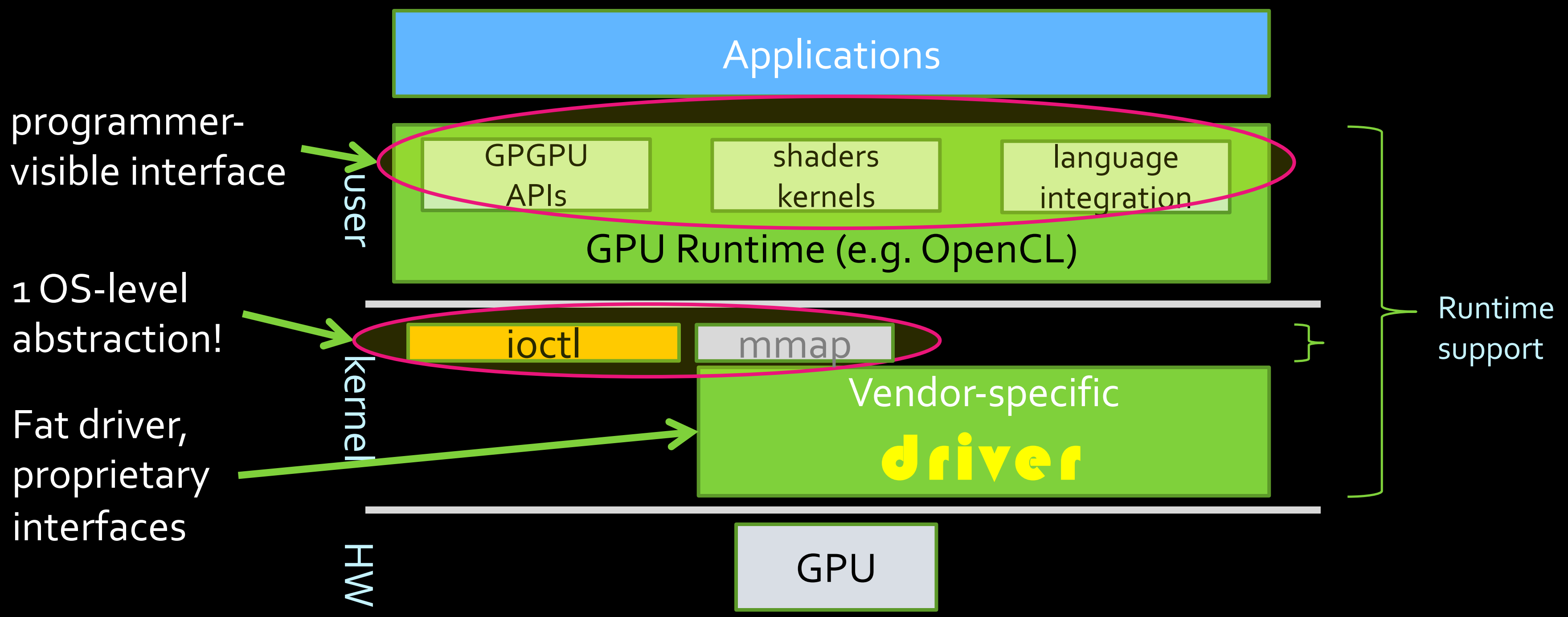
GPU abstractions



GPU abstractions

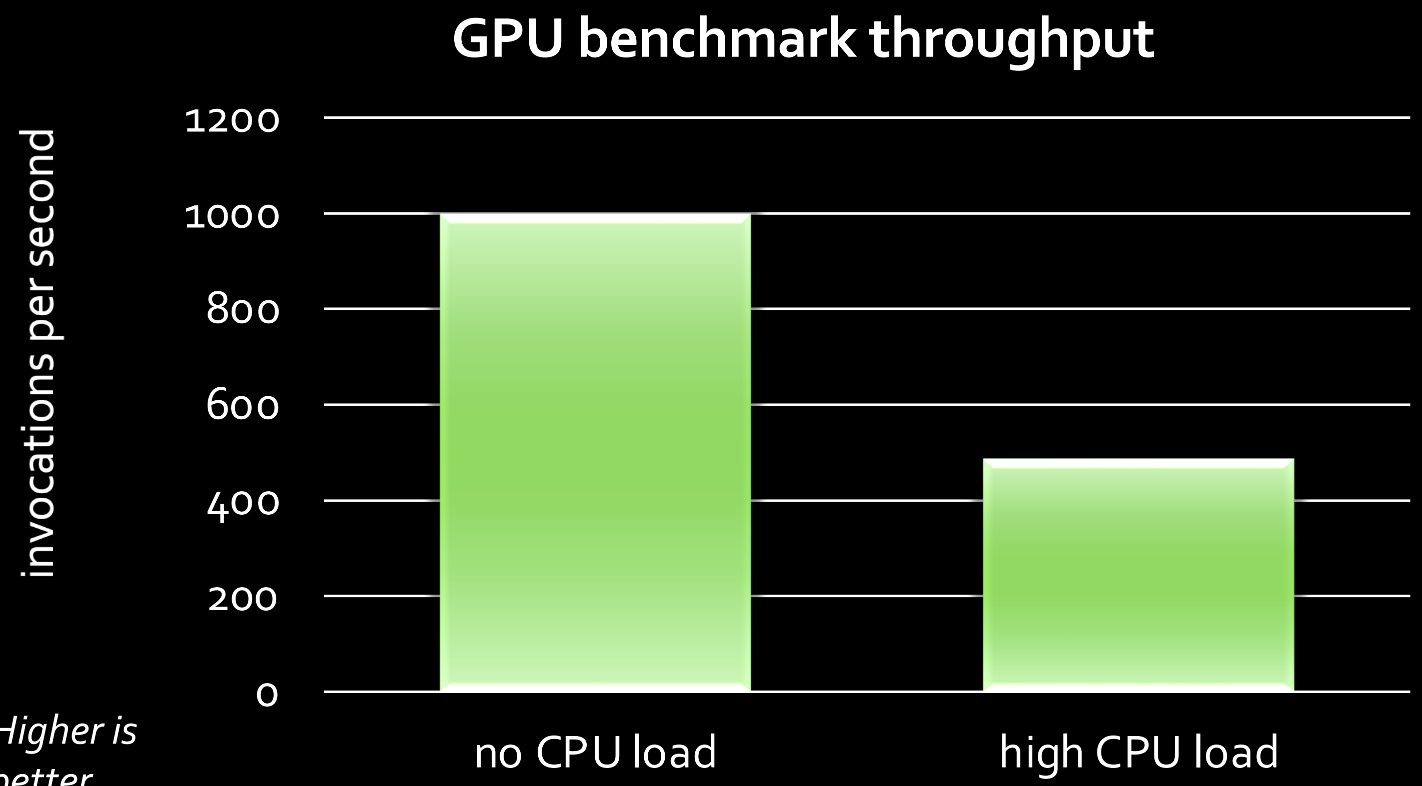


GPU abstractions



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

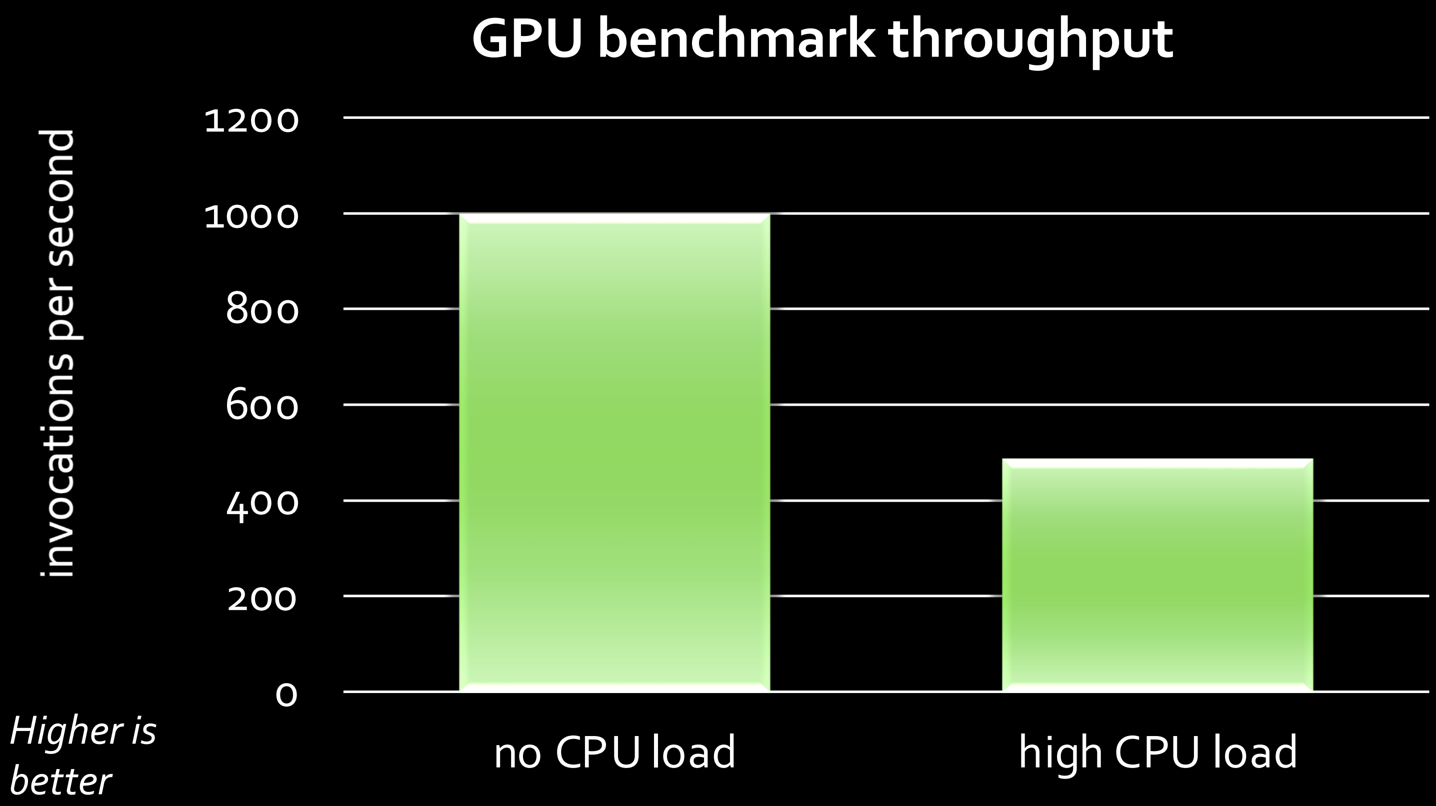
No OS support → No isolation



Higher is better

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

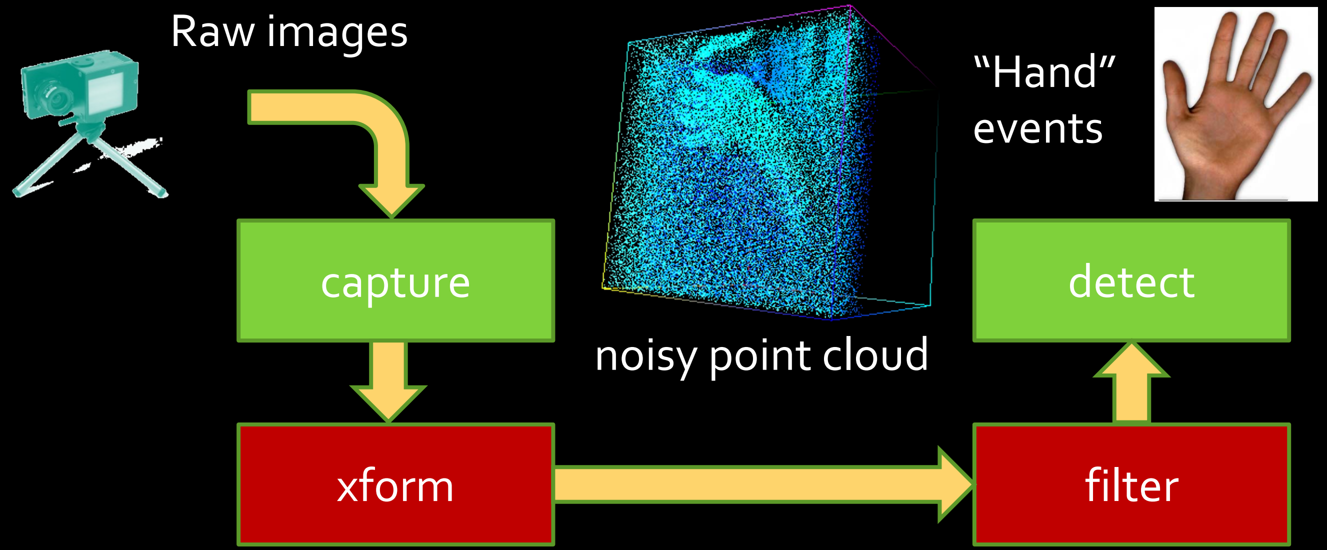
No OS support → No isolation



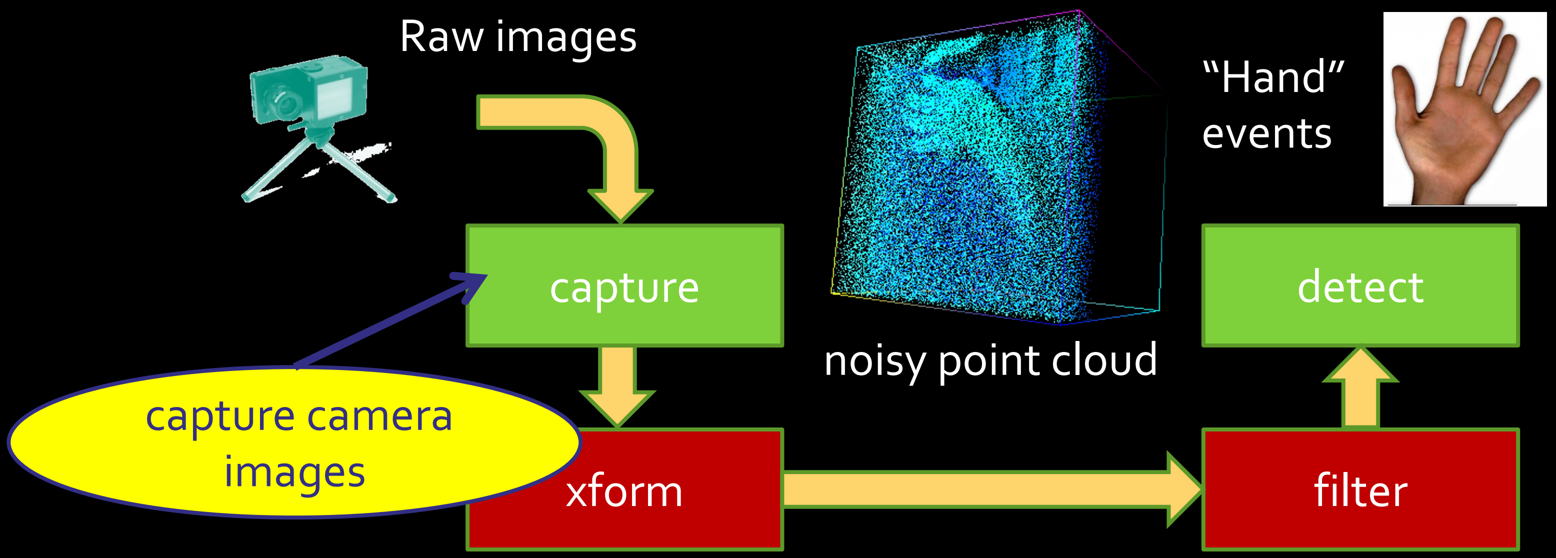
CPU+GPU schedulers not integrated!
...other pathologies abundant

ge-convolution in CUDA
dows 7 x64 8GB RAM
l Core 2 Quad 2.66GHz
dia GeForce GT230

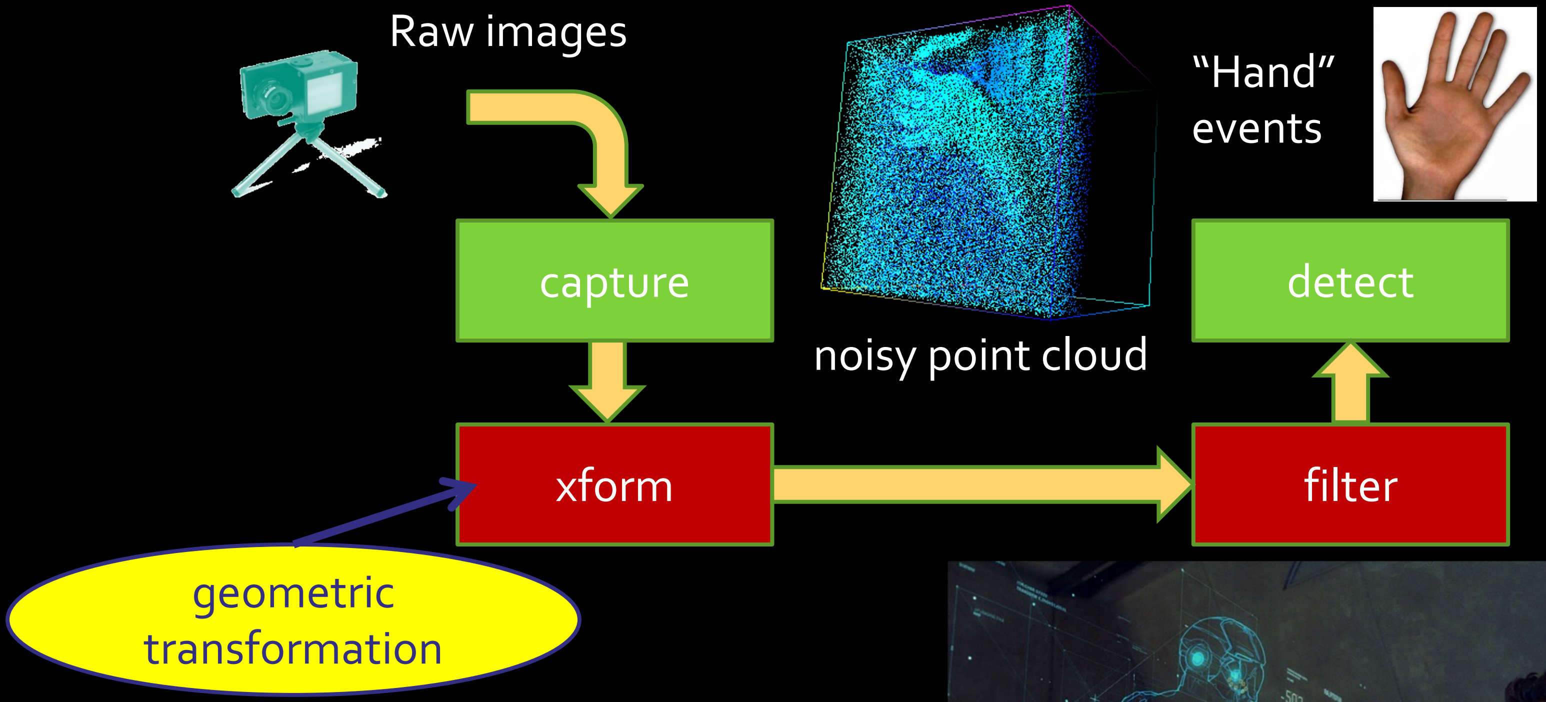
Composition: Gestural Interface



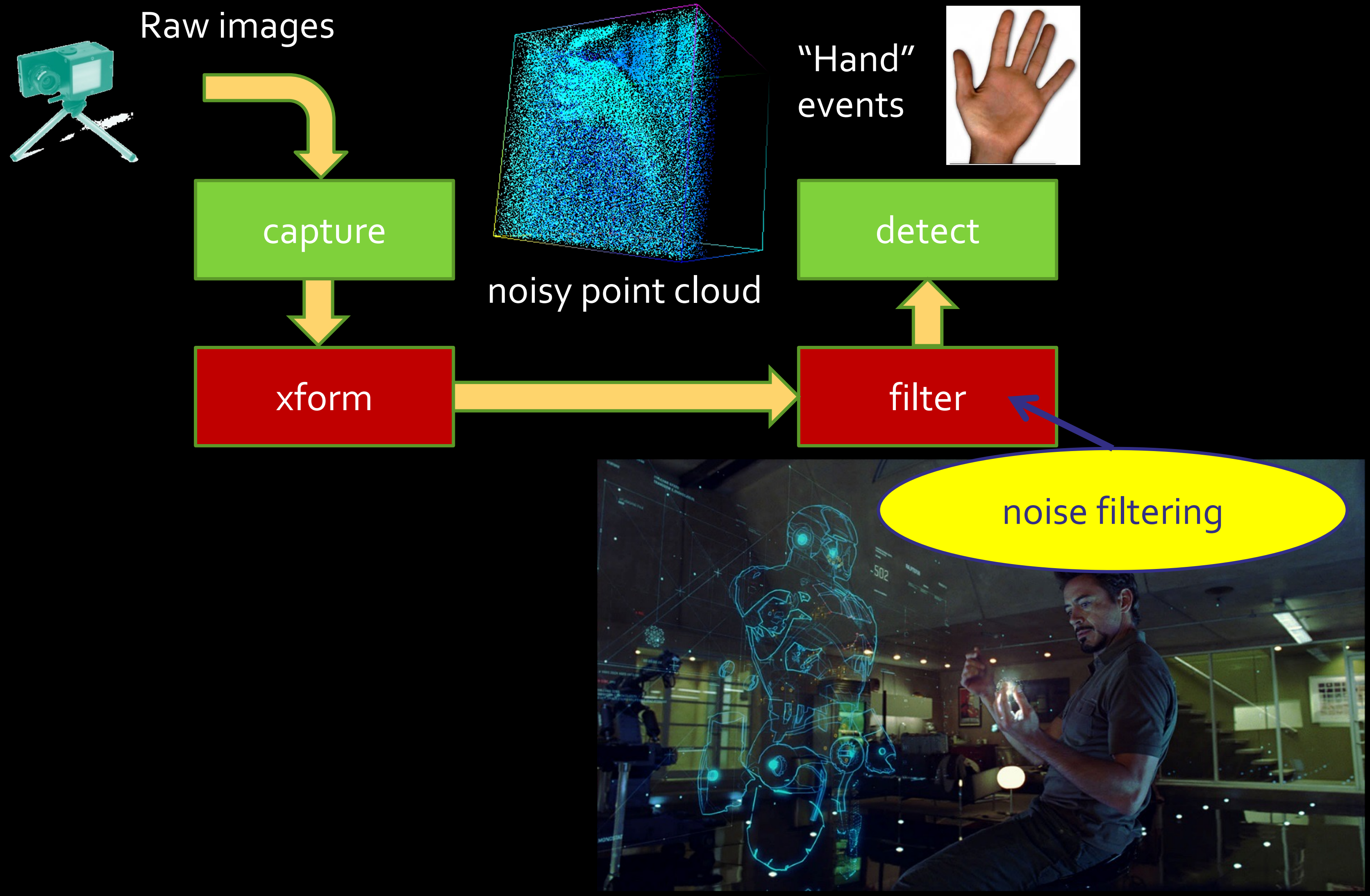
Composition: Gestural Interface



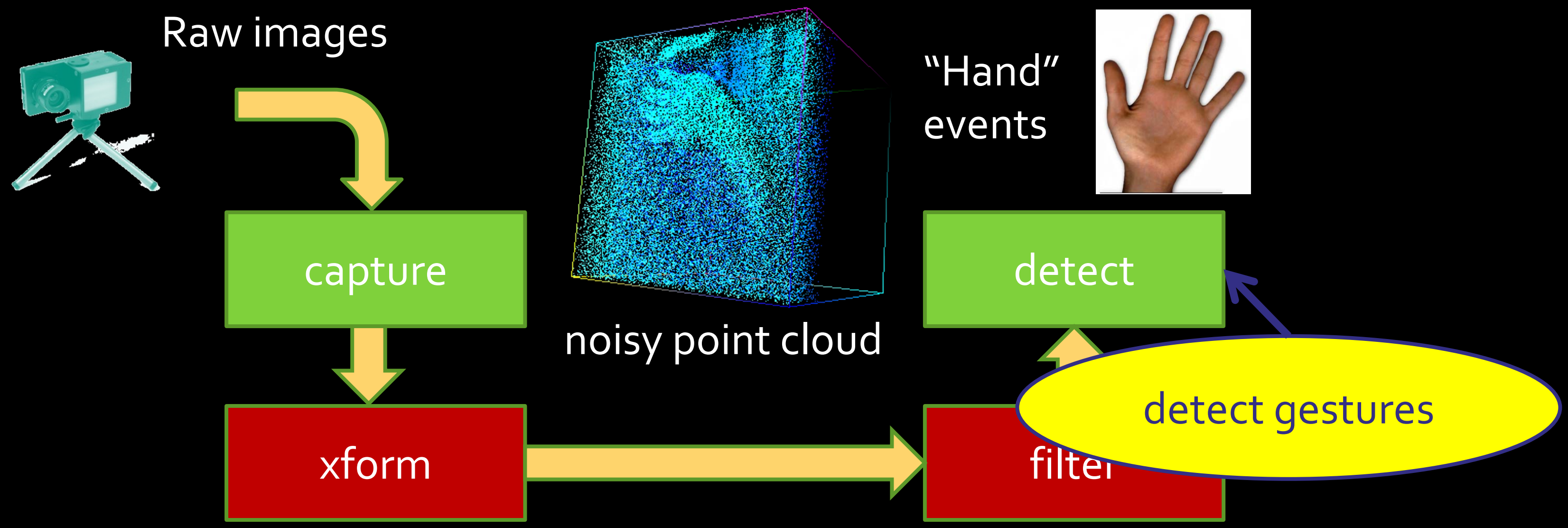
Composition: Gestural Interface



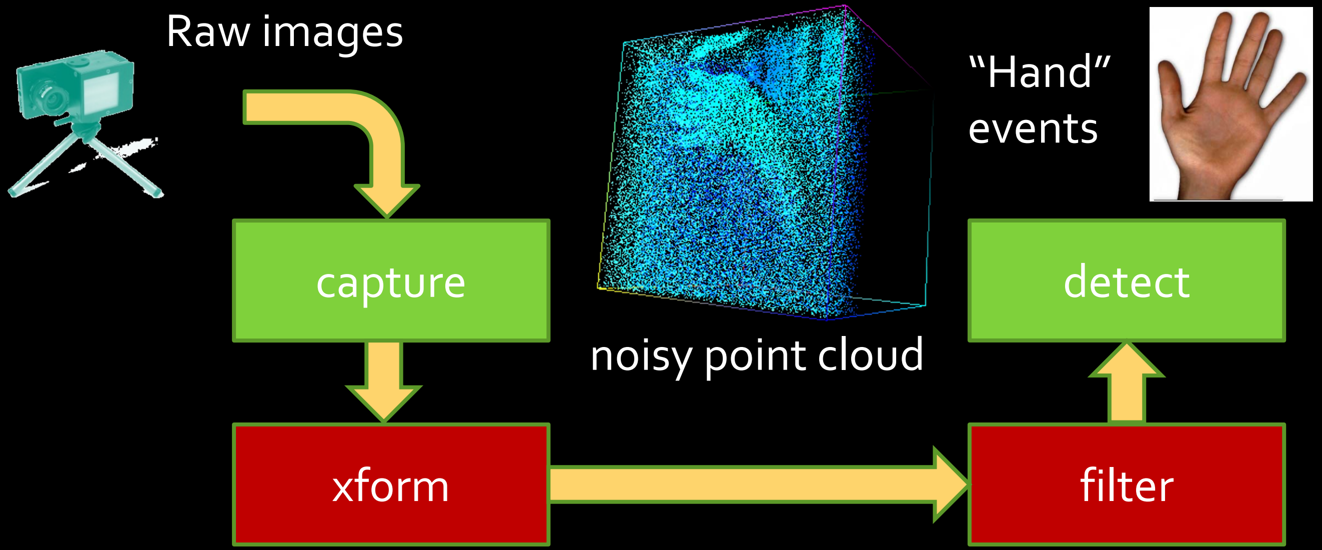
Composition: Gestural Interface



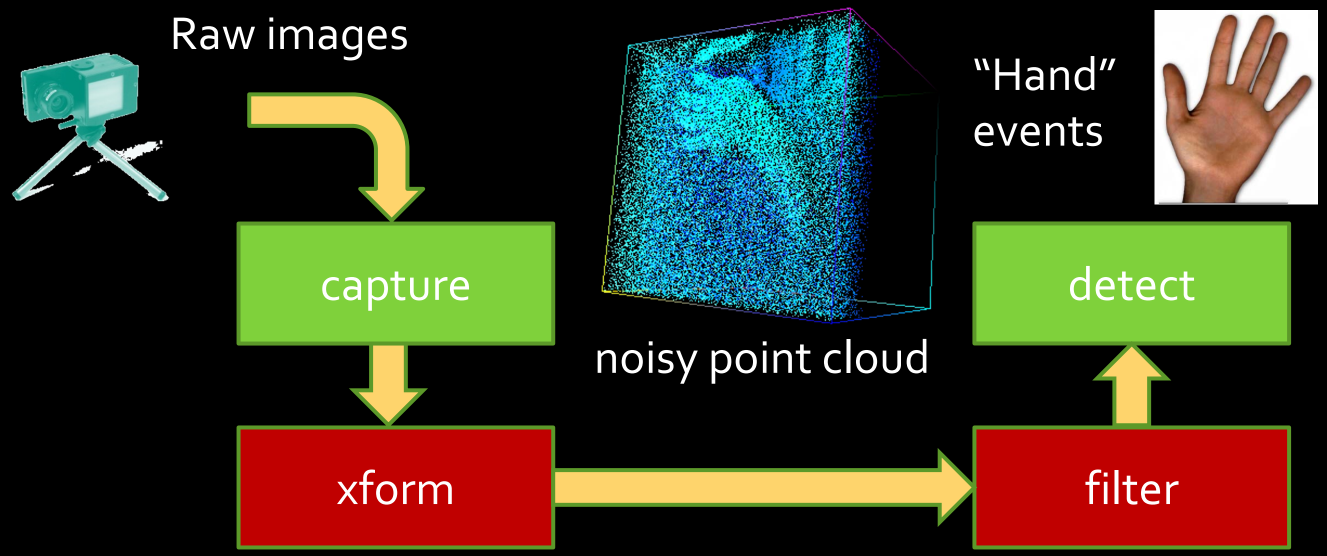
Composition: Gestural Interface



Composition: Gestural Interface



Composition: Gestural Interface



- 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

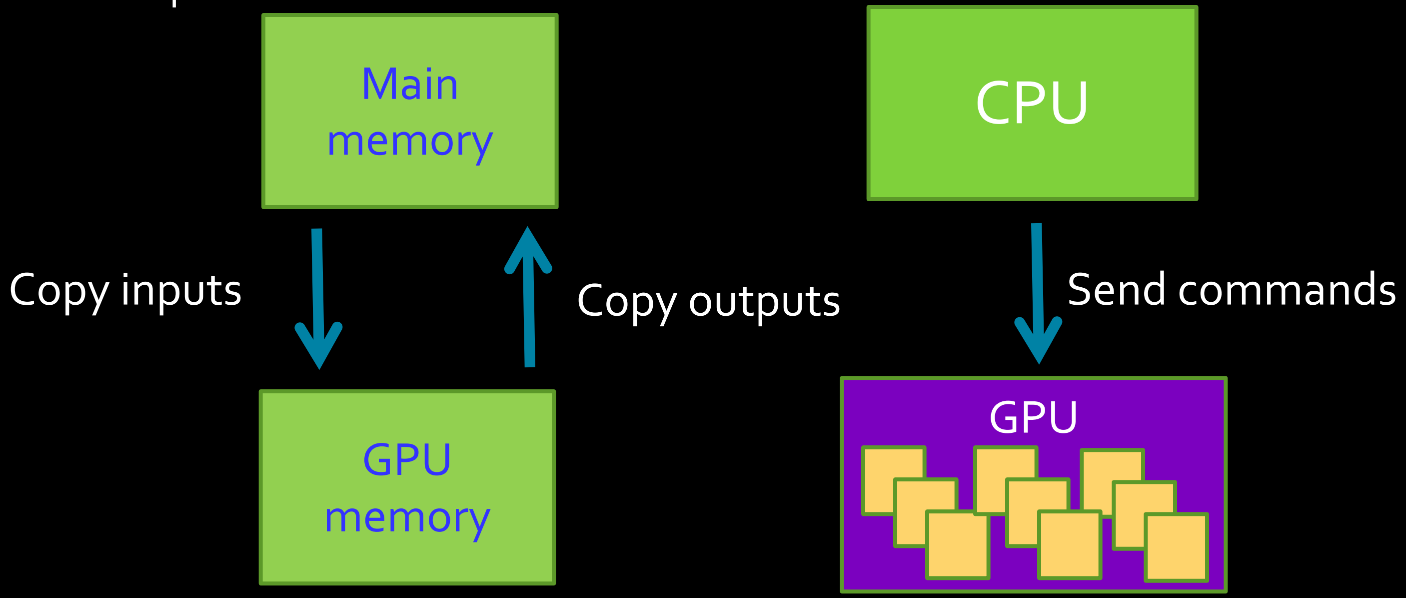
```
#> capture | xform | filter | detect &
```

CPU	GPU	GPU	CPU

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

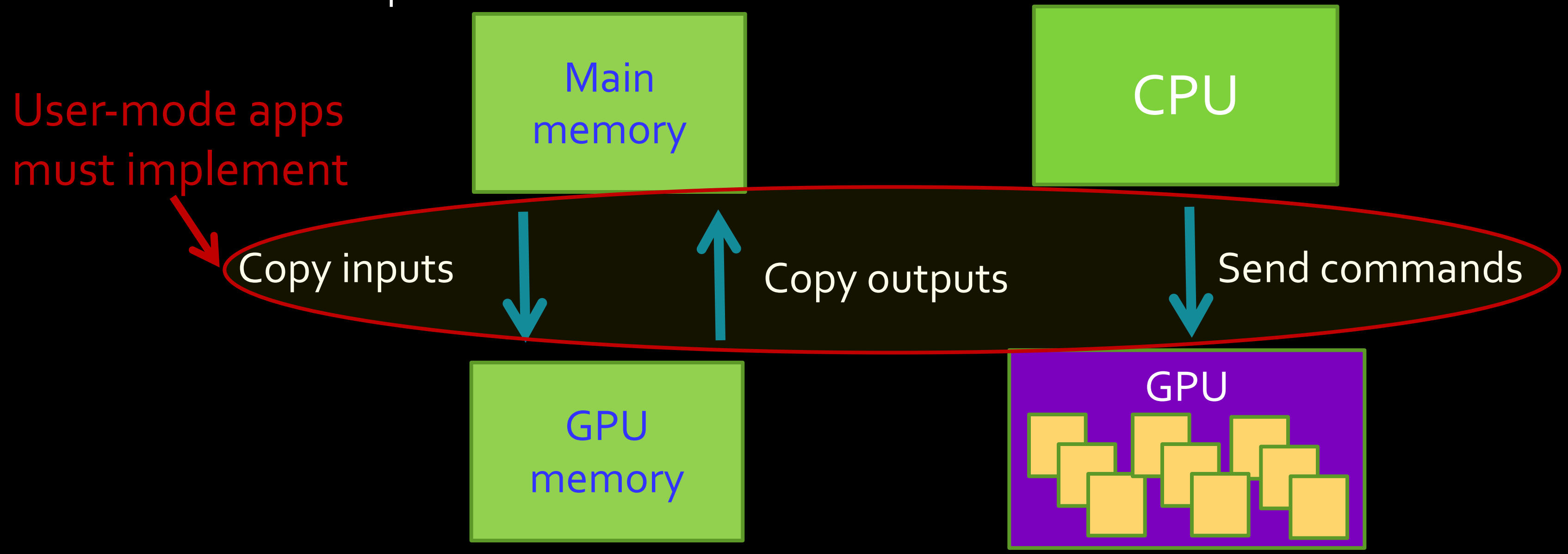
GPU Execution model

- 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



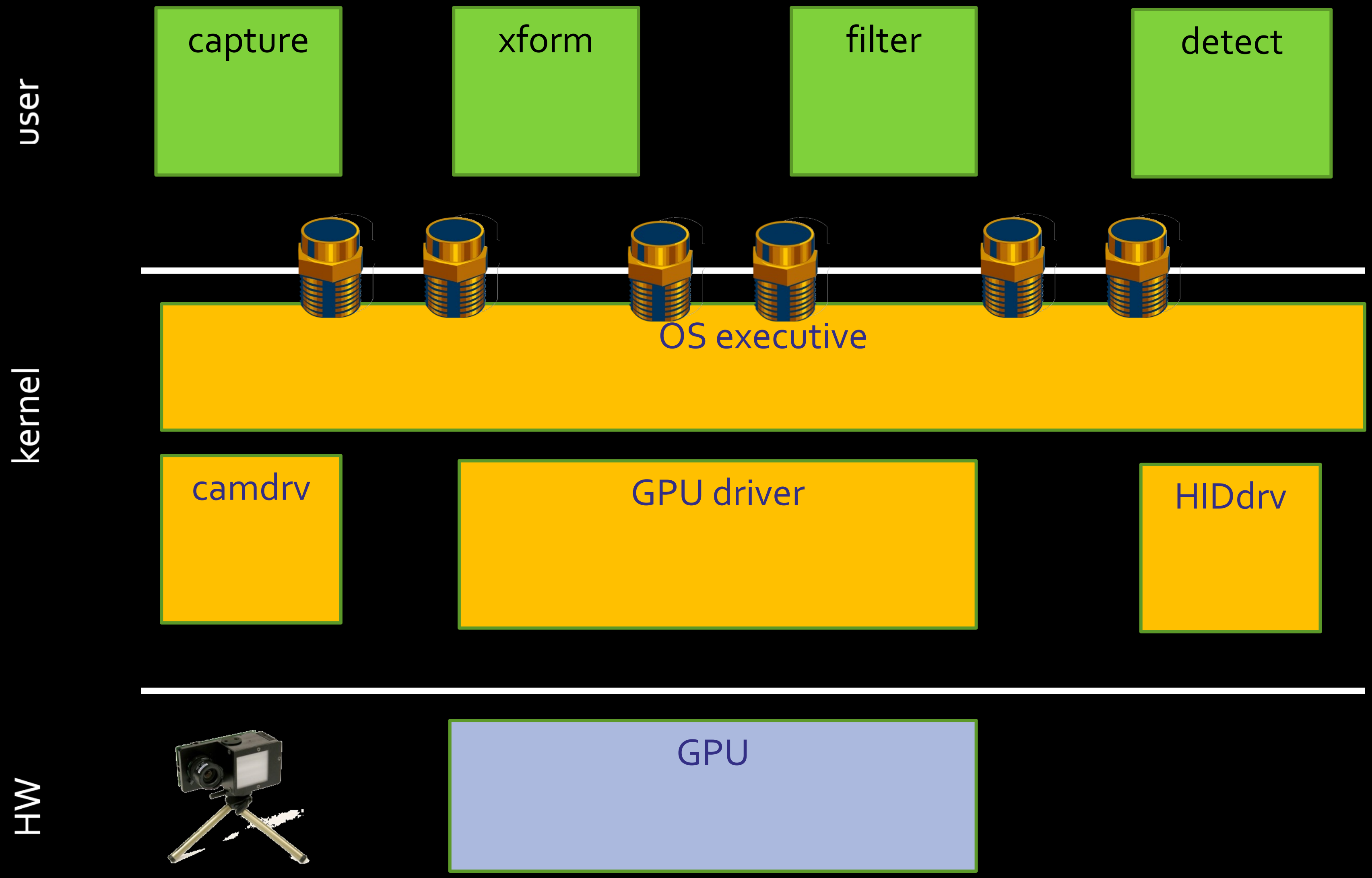
GPU Execution model

- 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



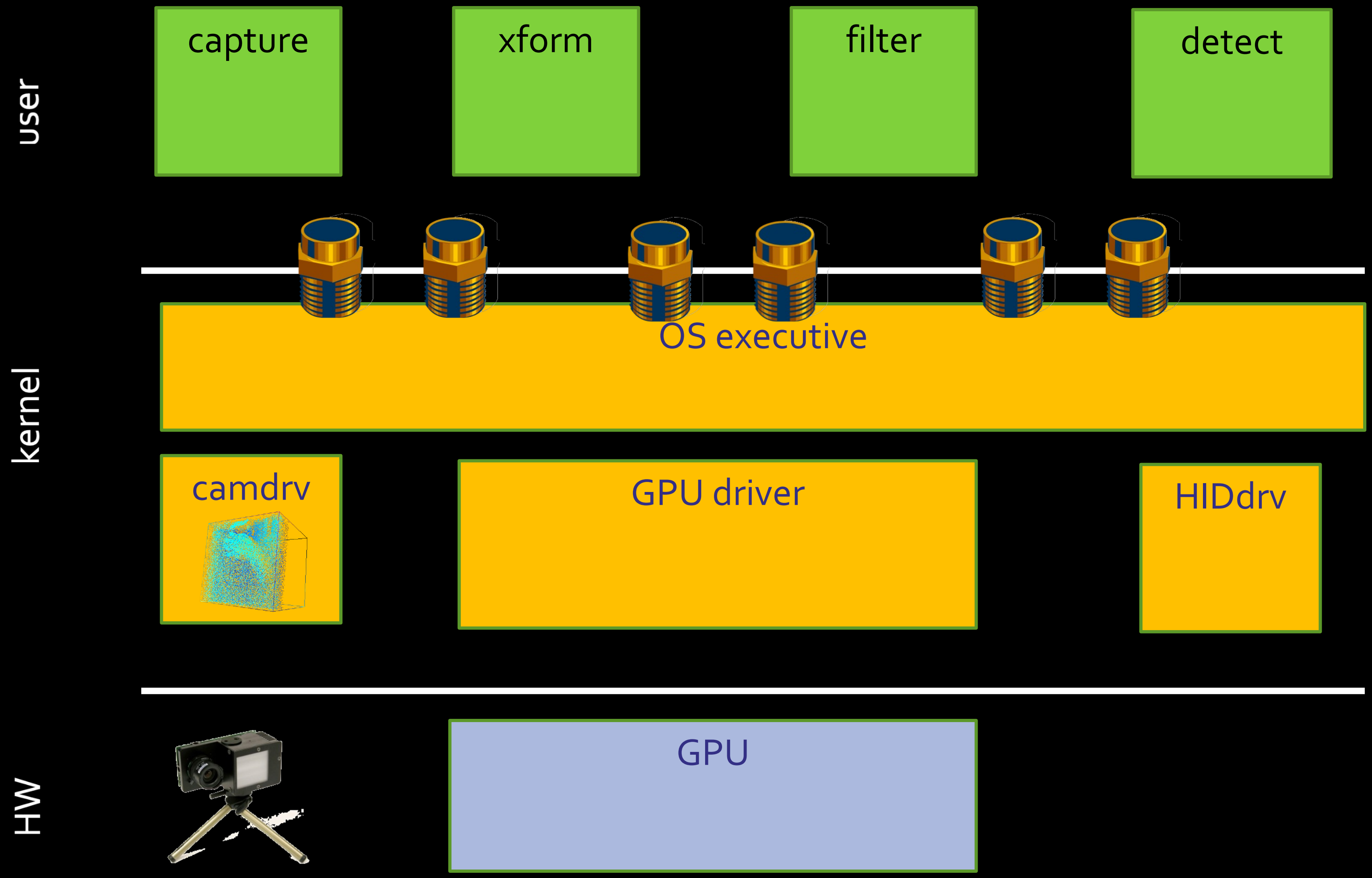
Data migration

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



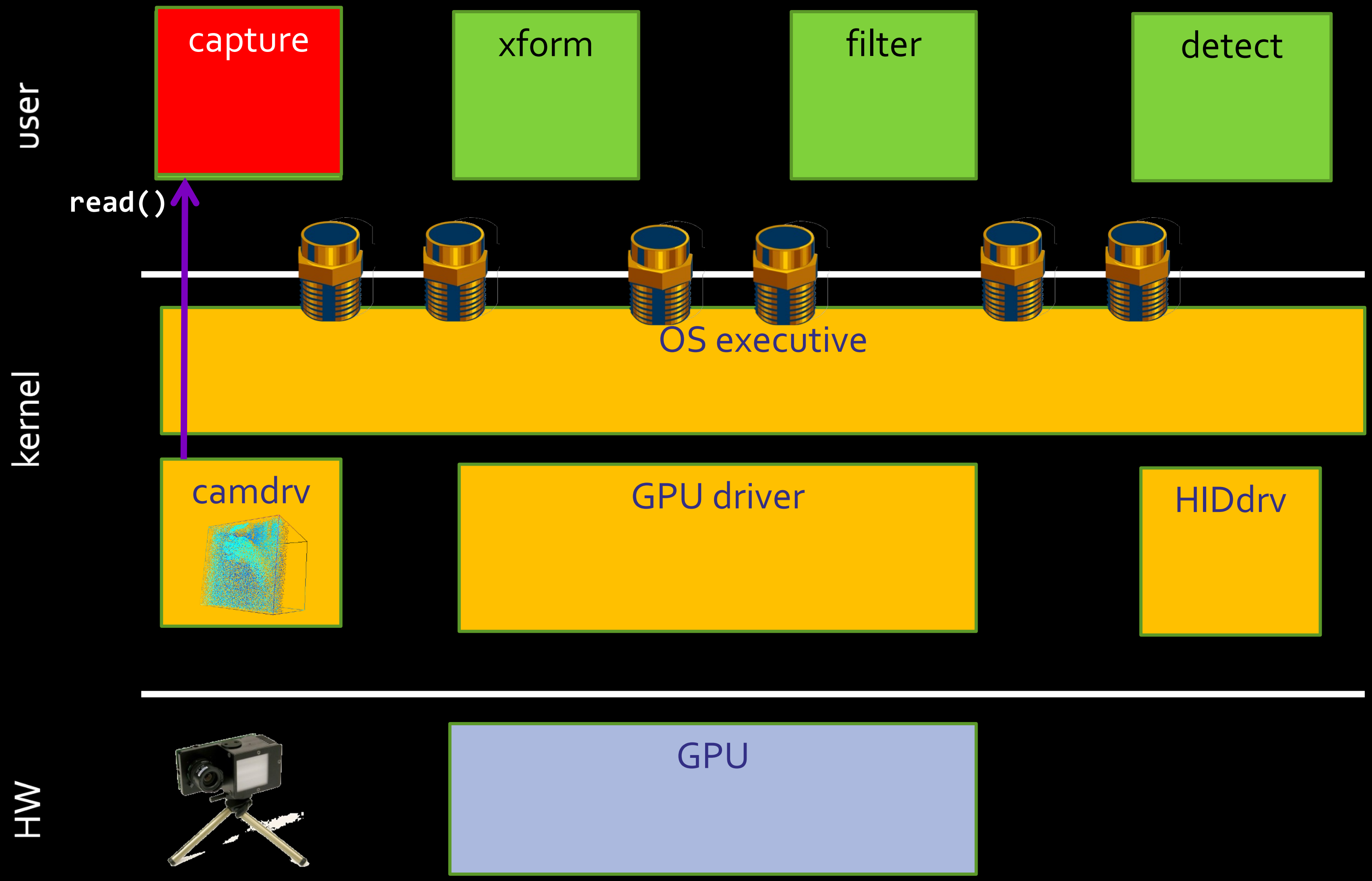
Data migration

#> capture | xform | filter | detect &



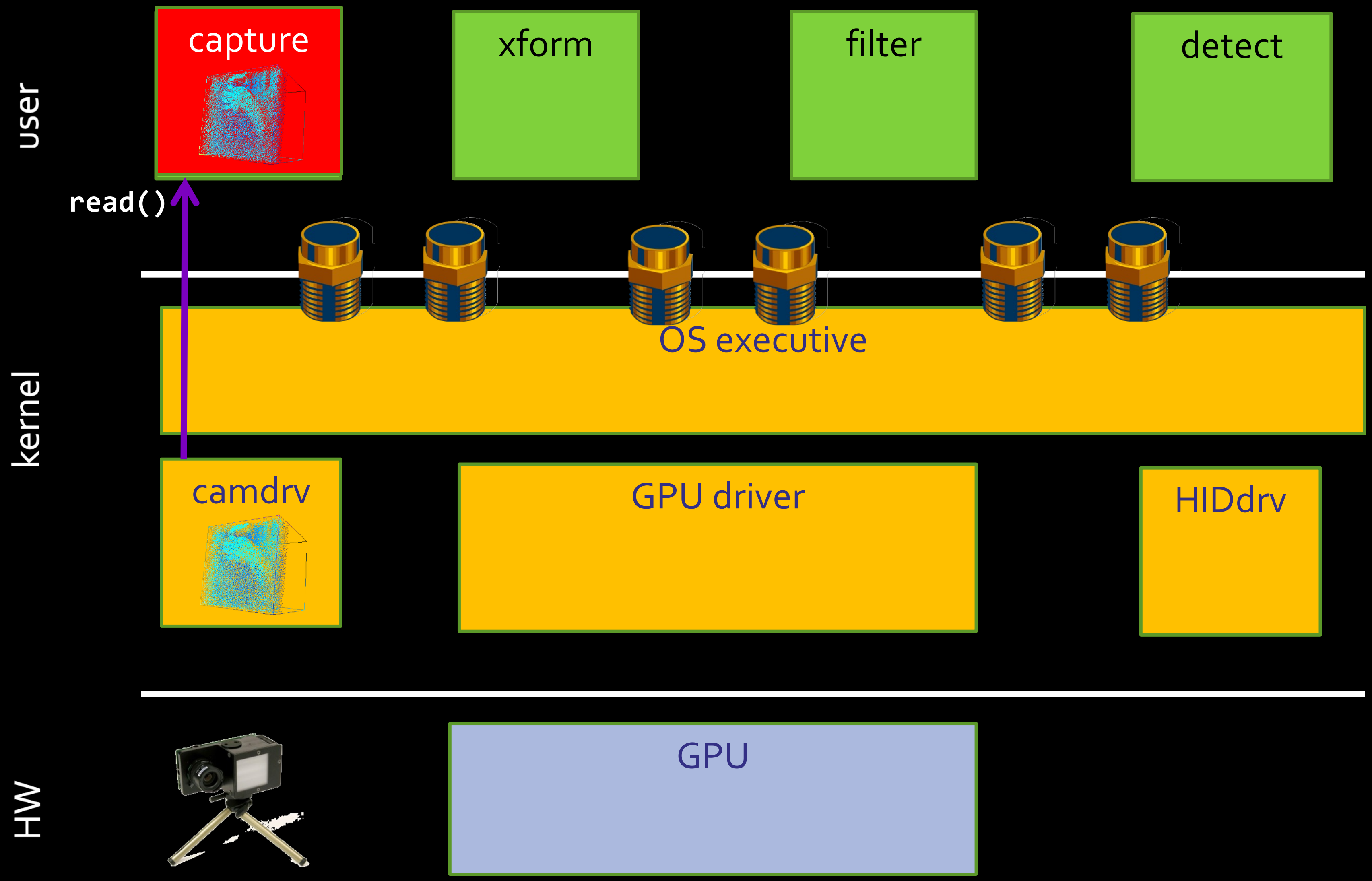
Data migration

#> capture | xform | filter | detect &



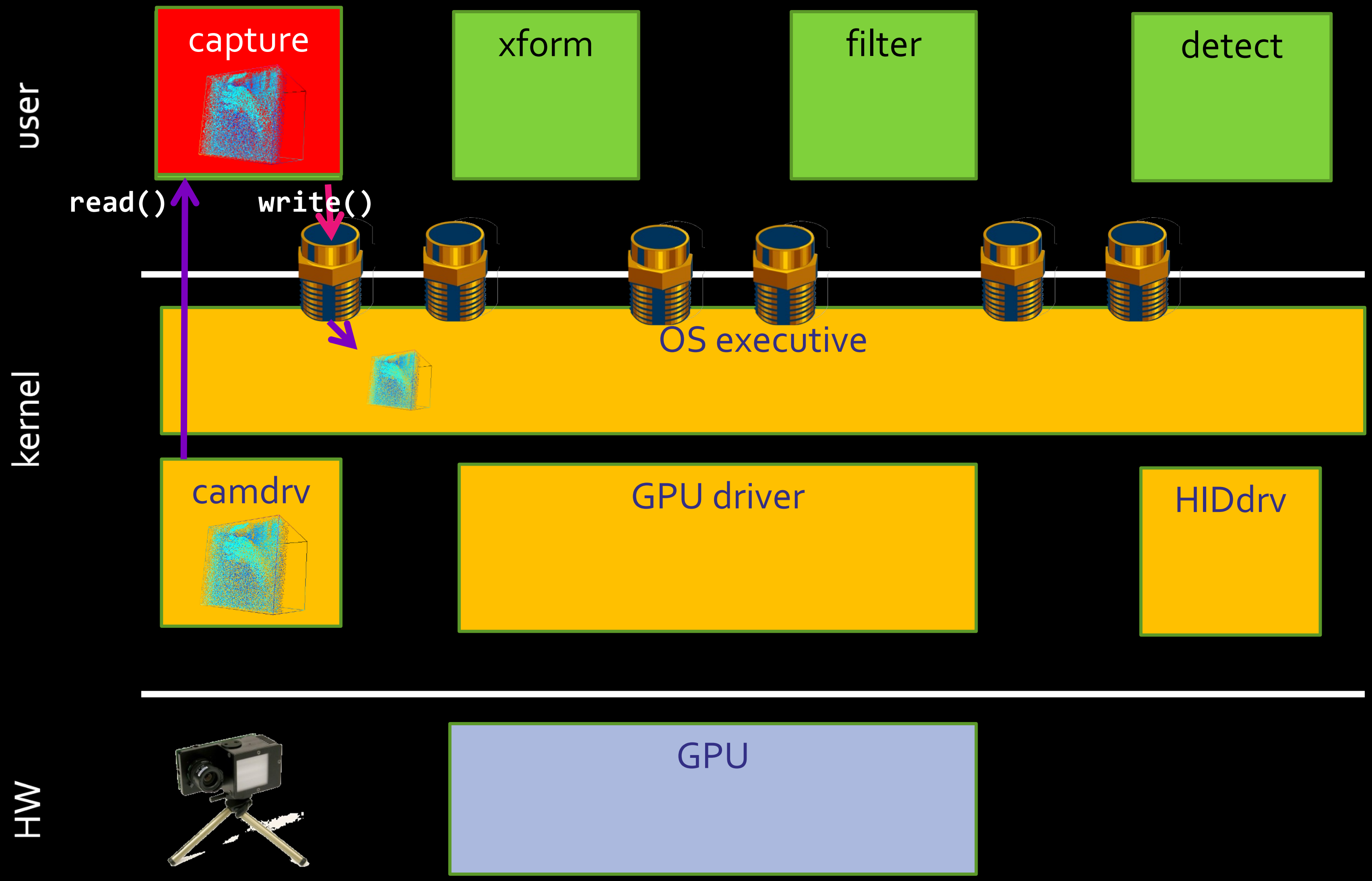
Data migration

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



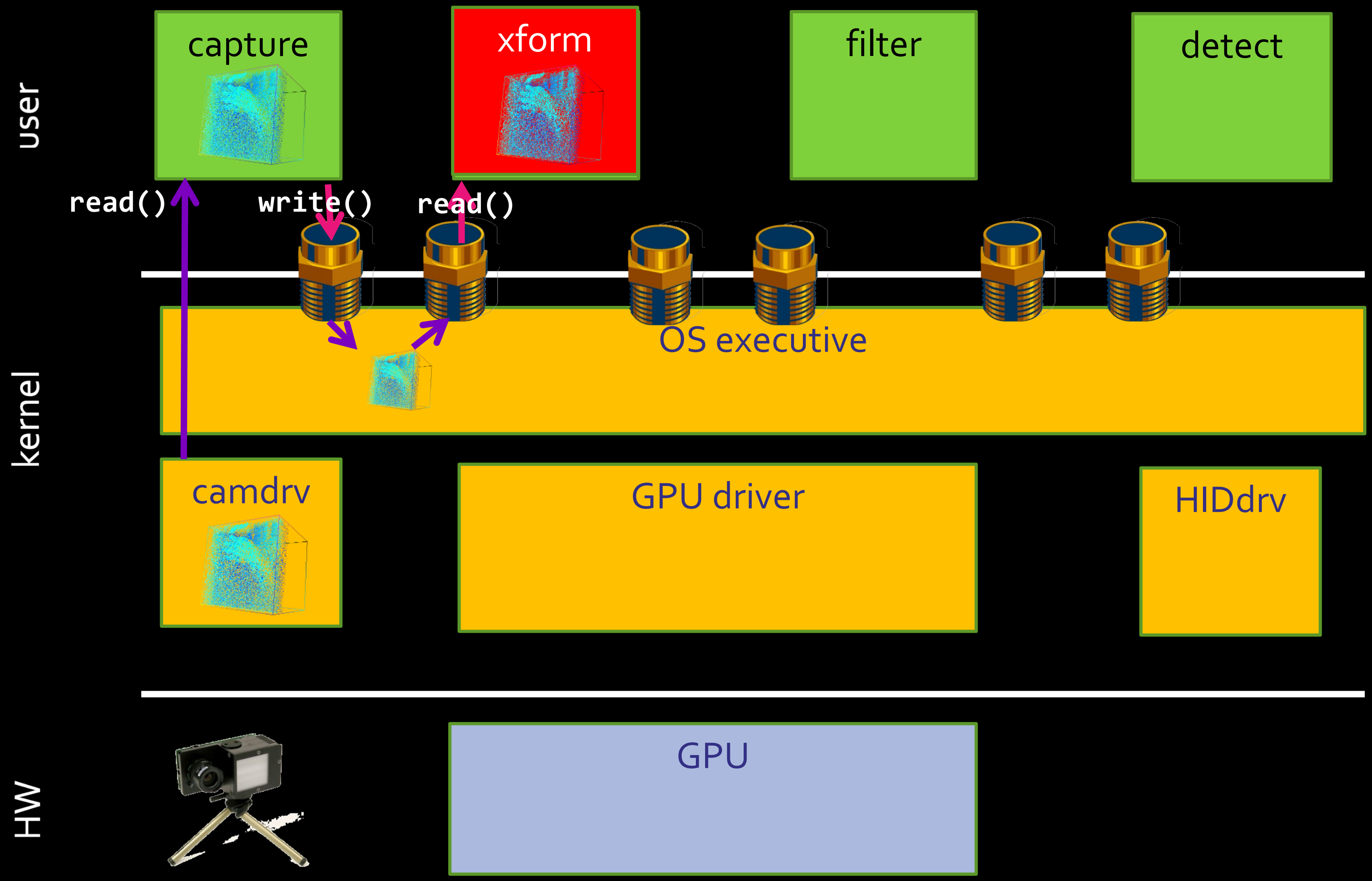
Data migration

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



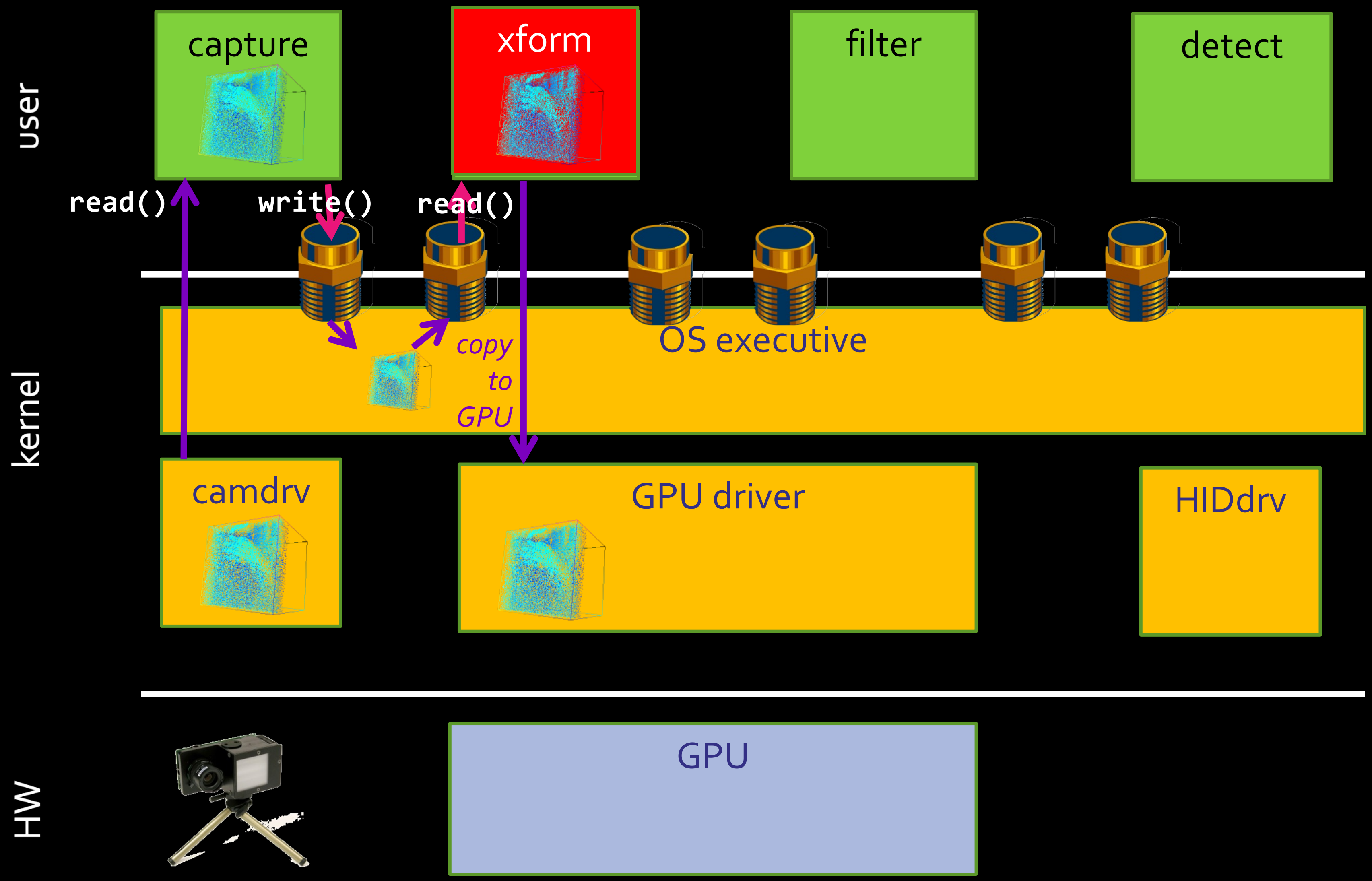
Data migration

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



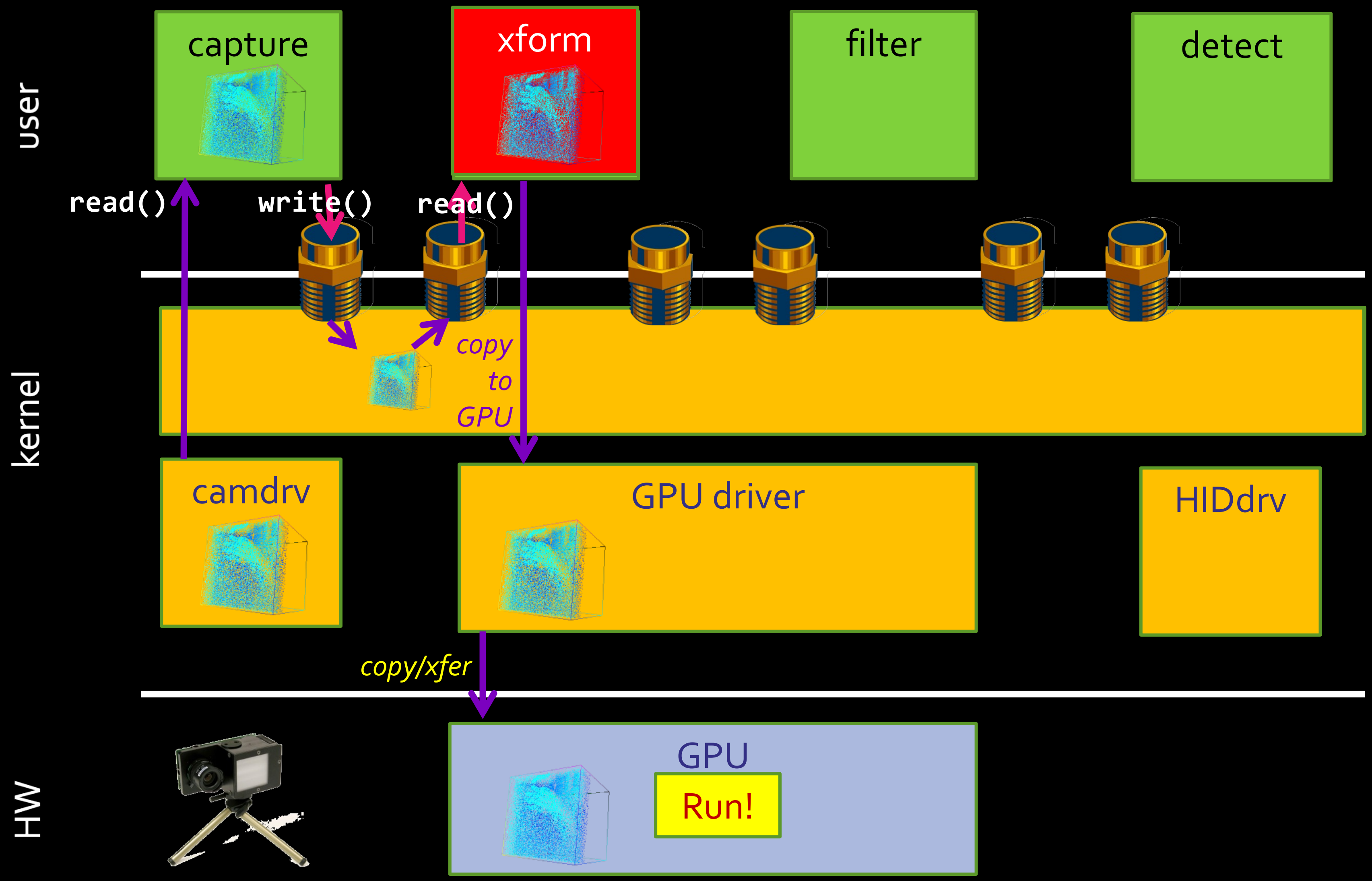
Data migration

#> capture | xform | filter | detect &



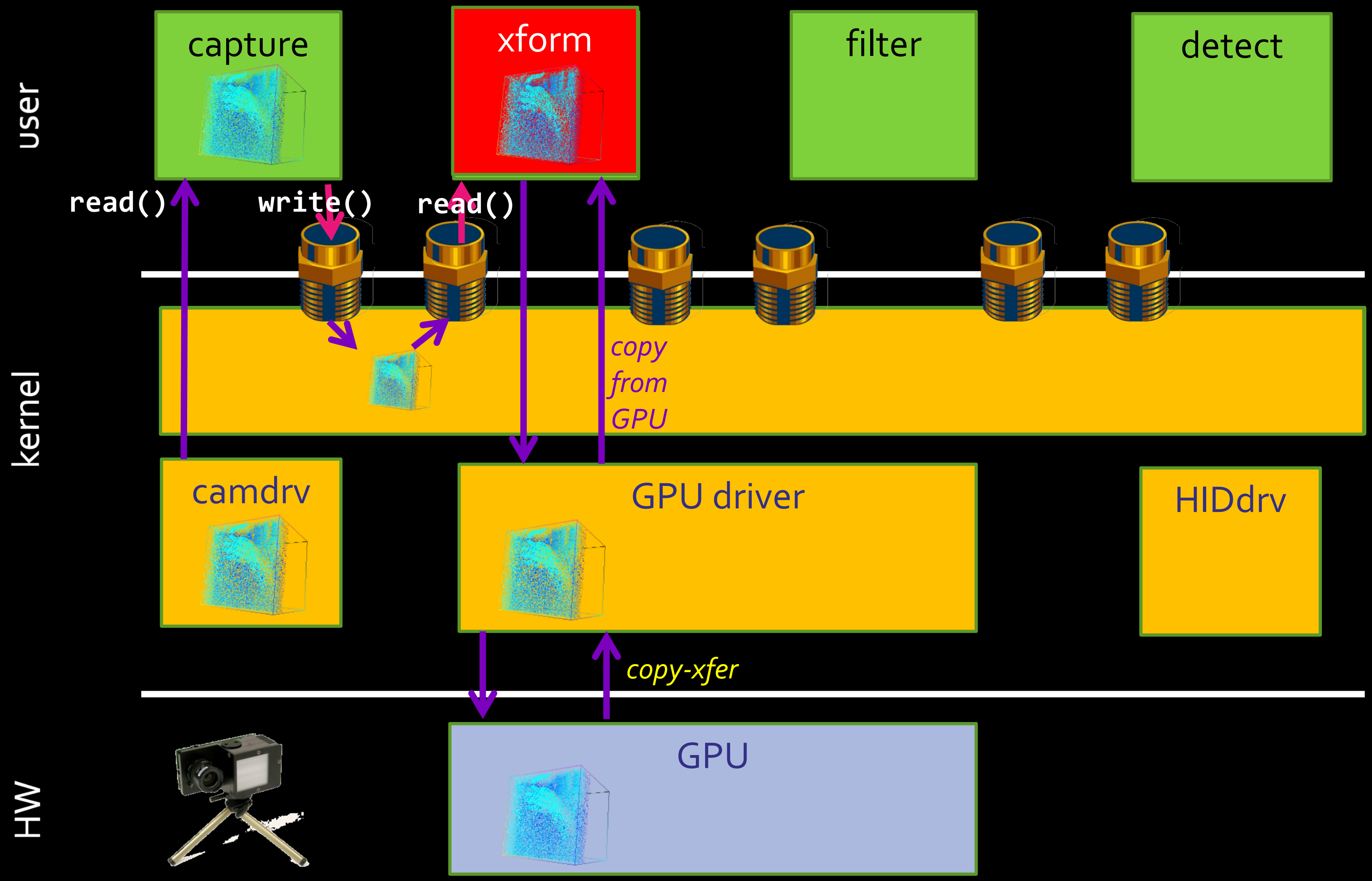
Data migration

#> capture | xform | filter | detect &



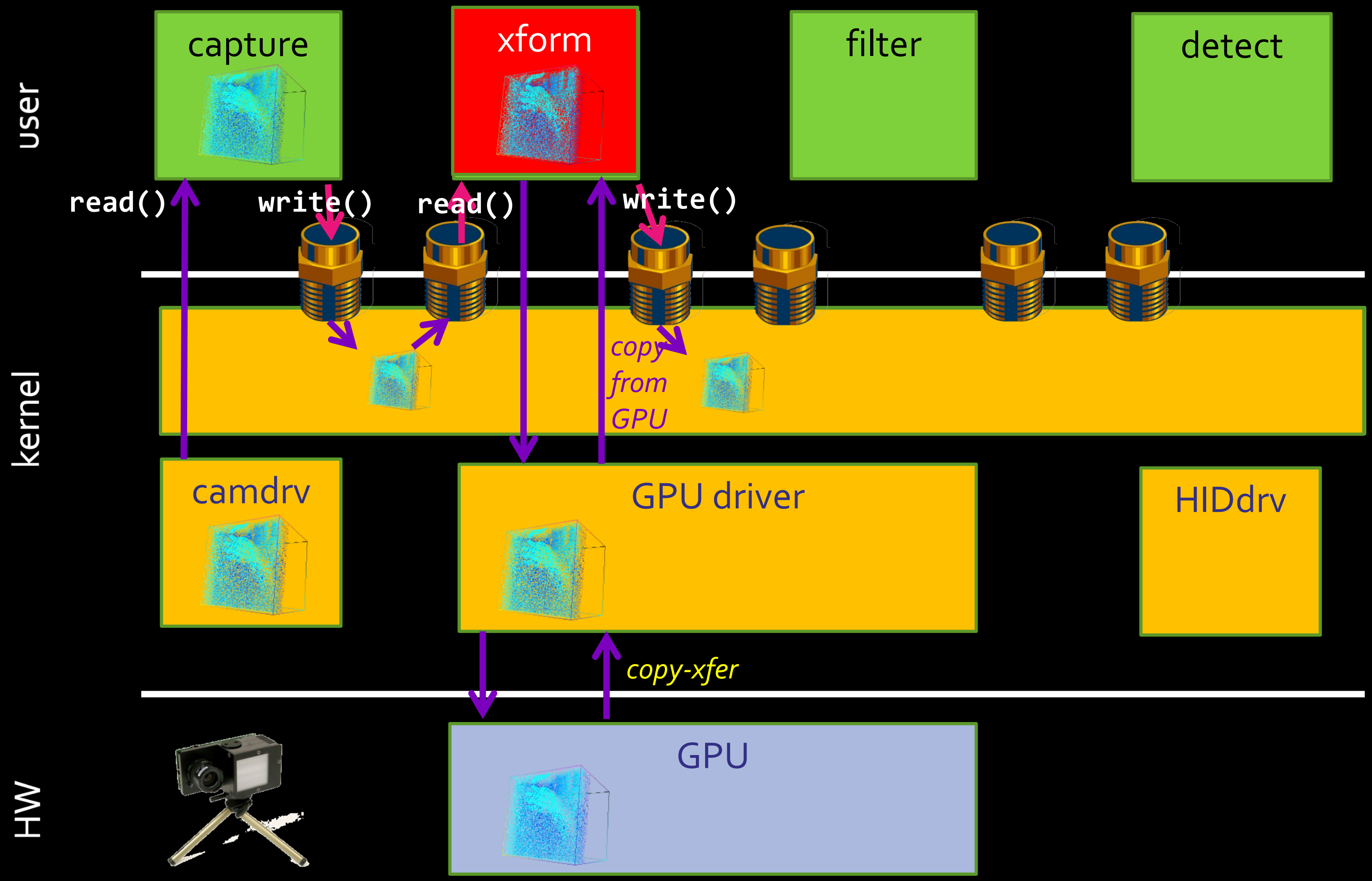
Data migration

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



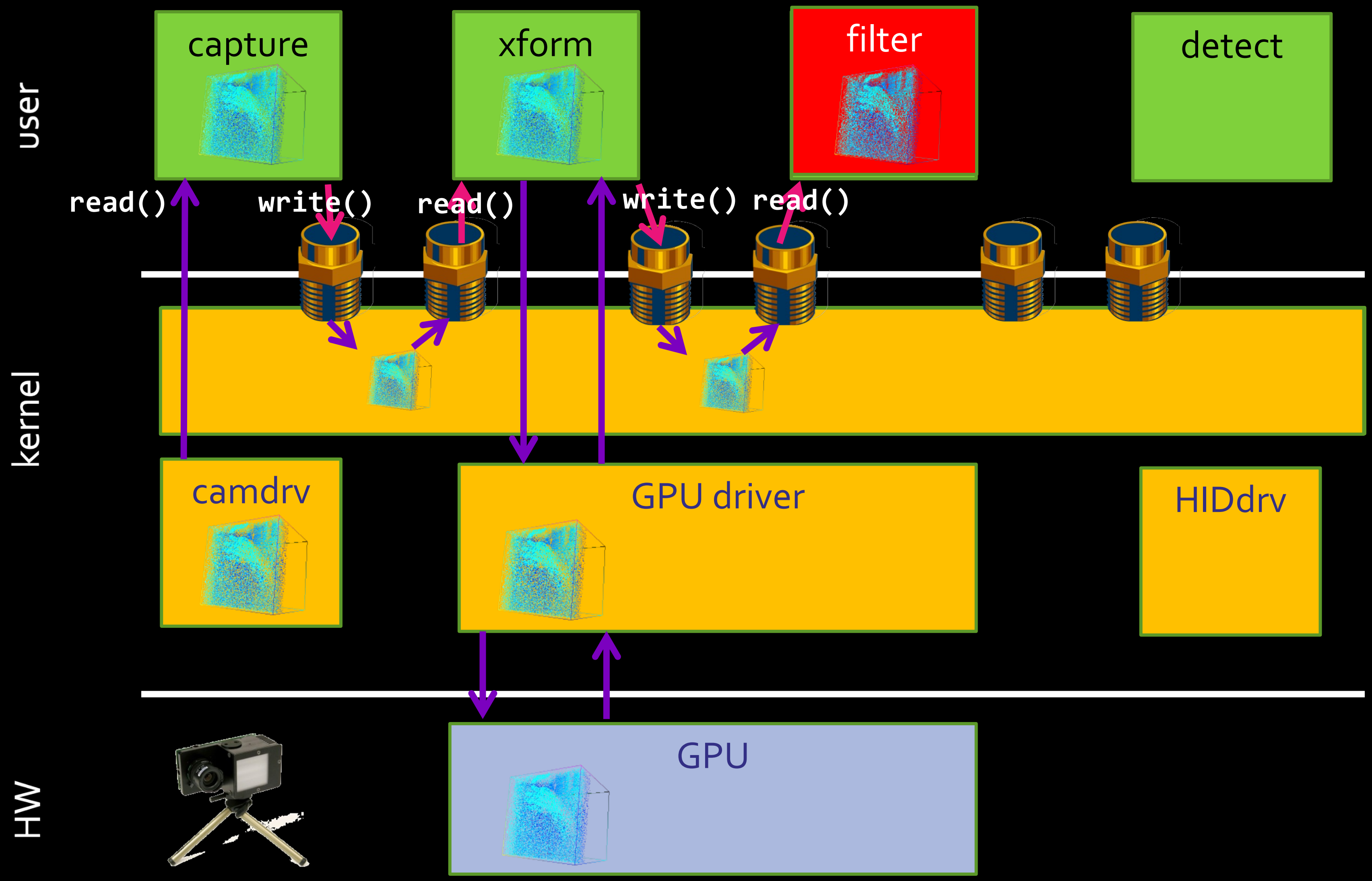
Data migration

#> capture | xform | filter | detect &



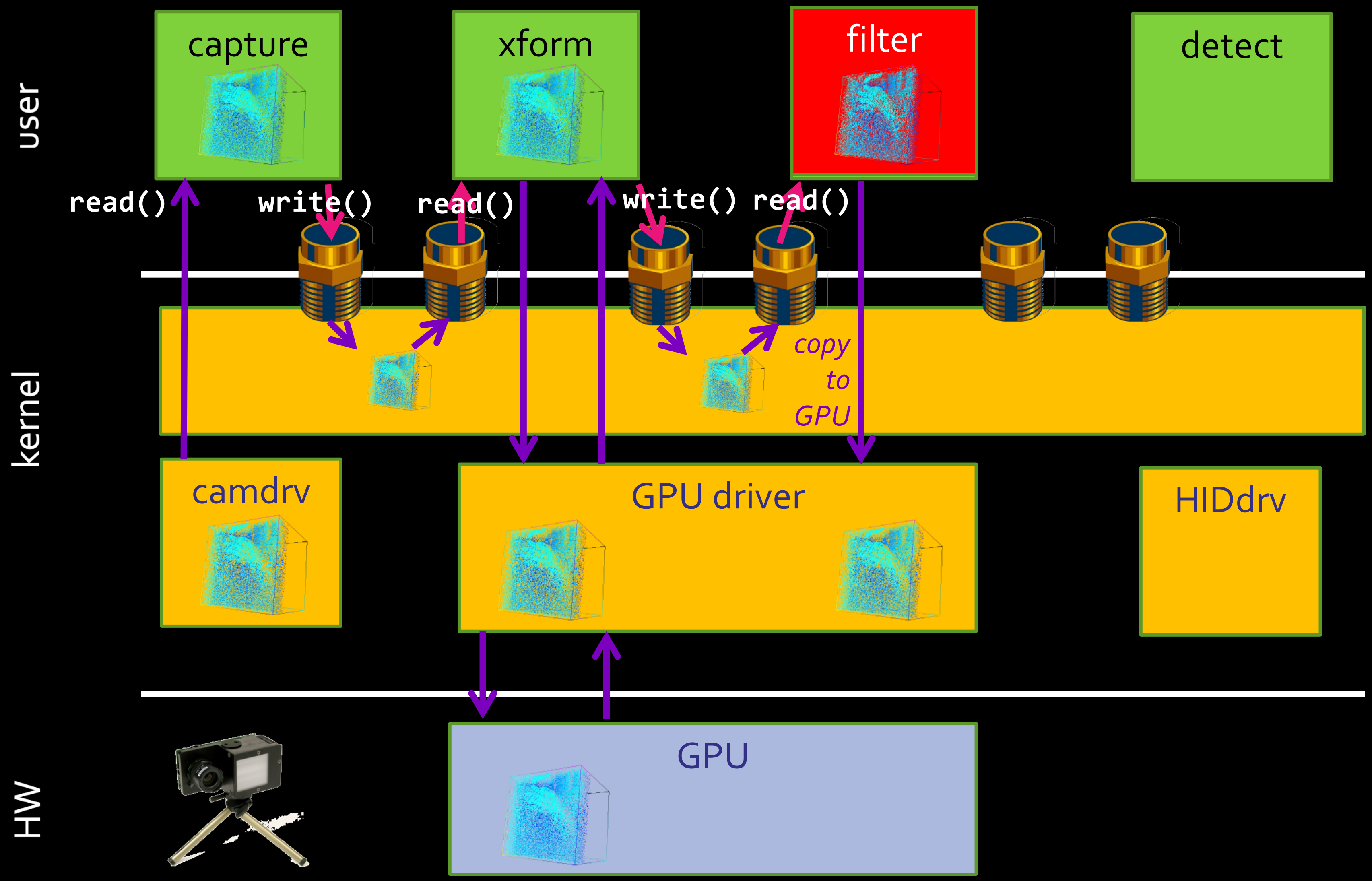
Data migration

#> capture | xform | filter | detect &



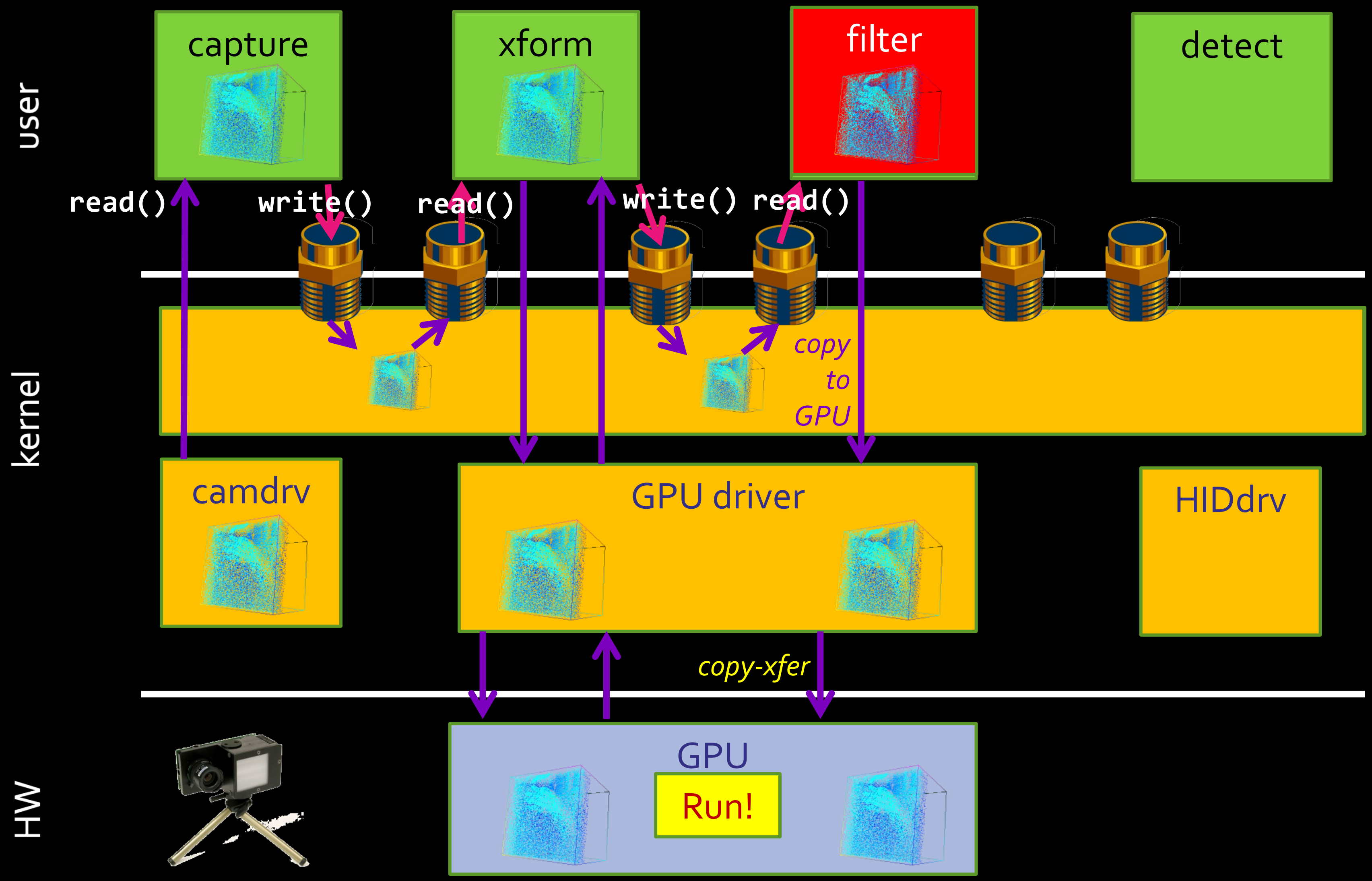
Data migration

#> capture | xform | filter | detect &



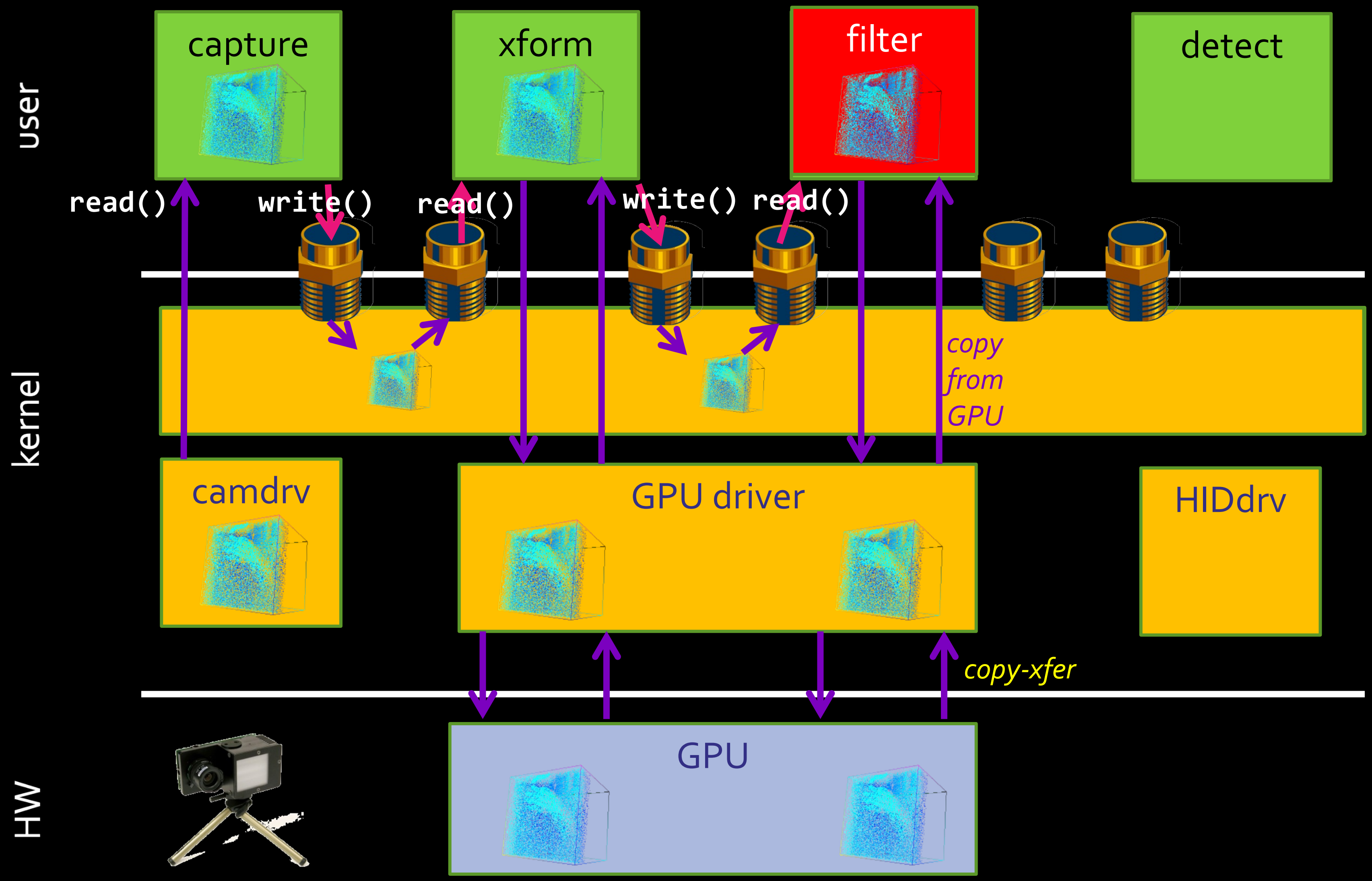
Data migration

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



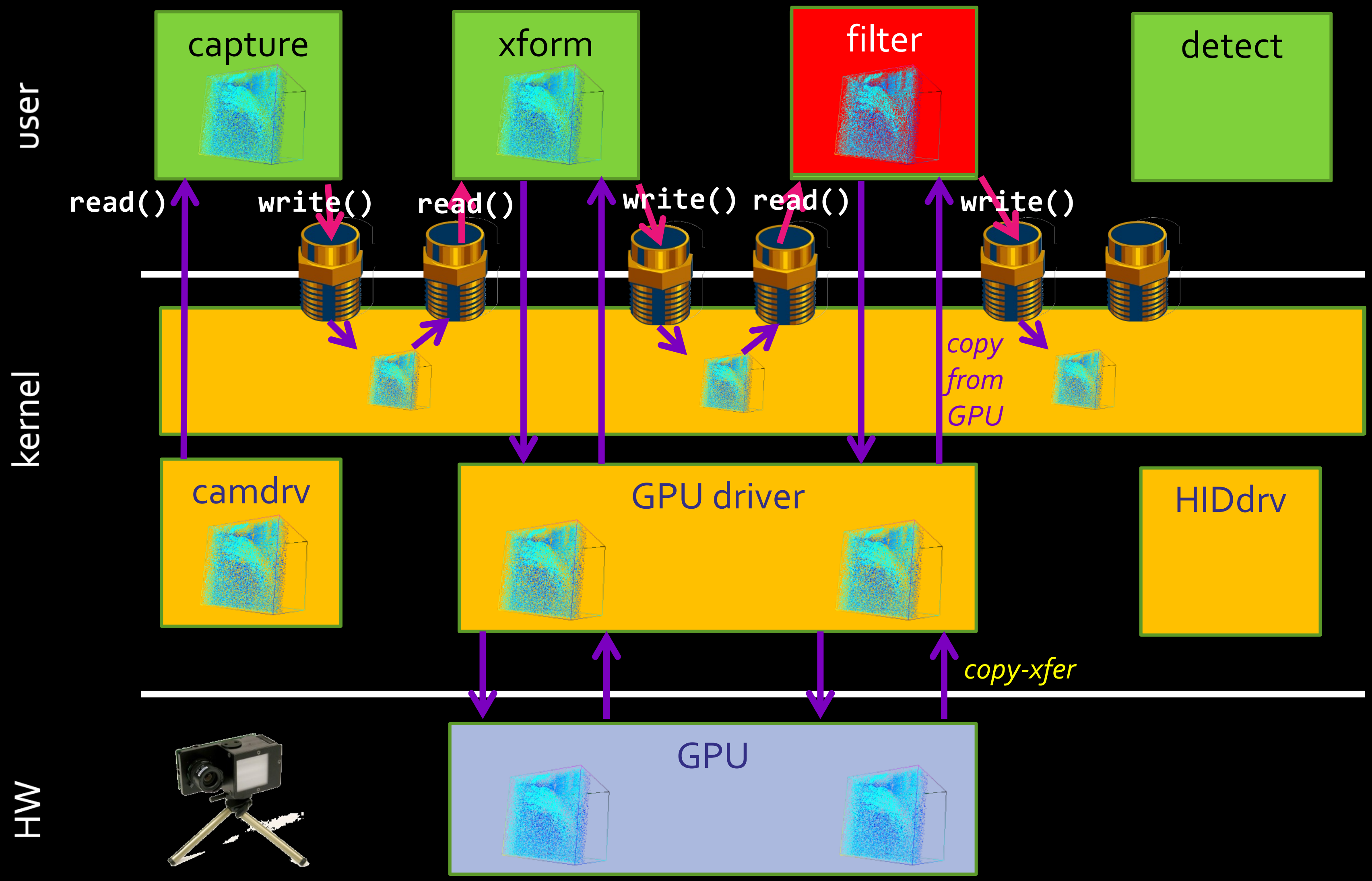
Data migration

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



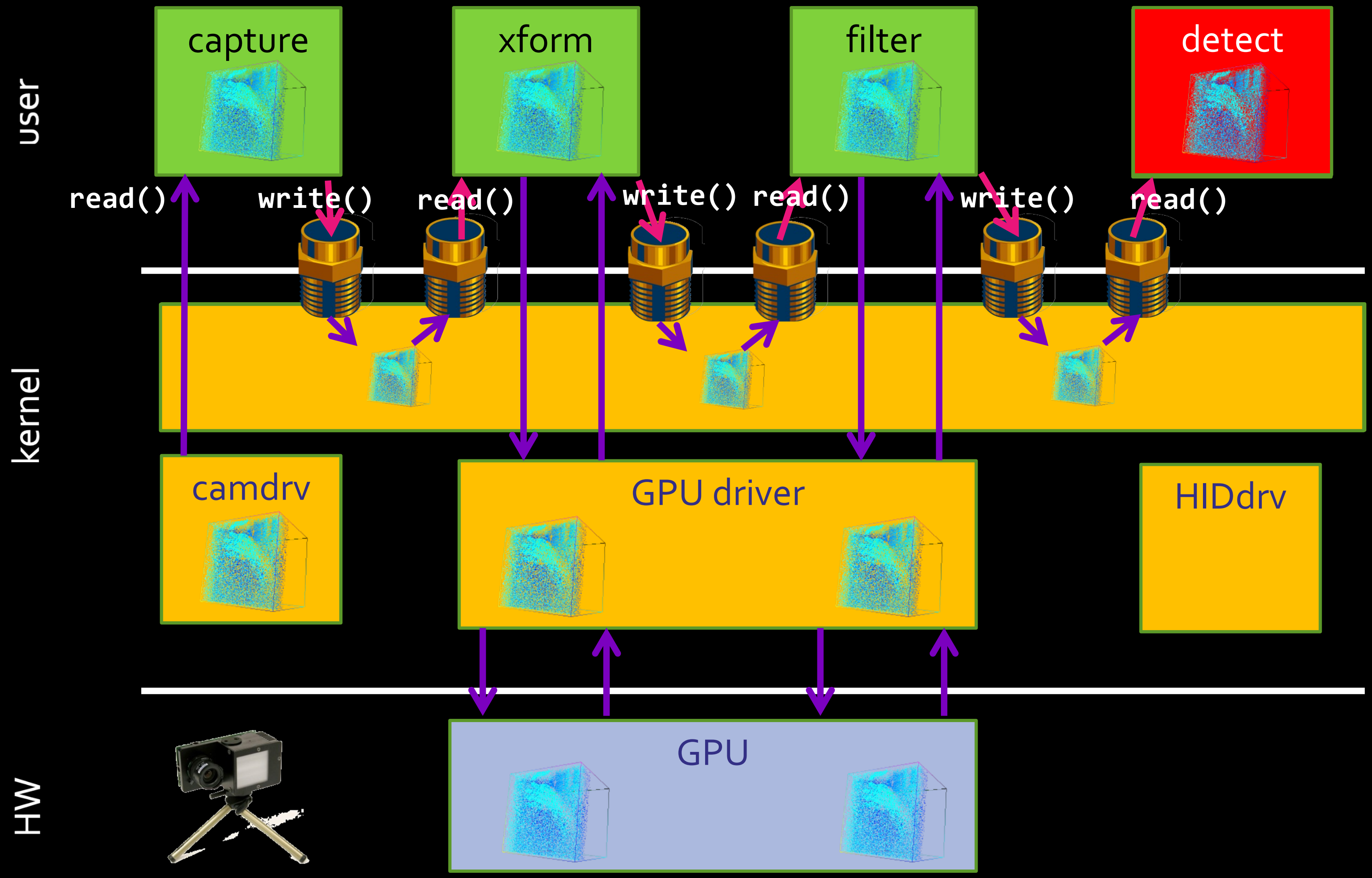
Data migration

#> capture | xform | filter | detect &



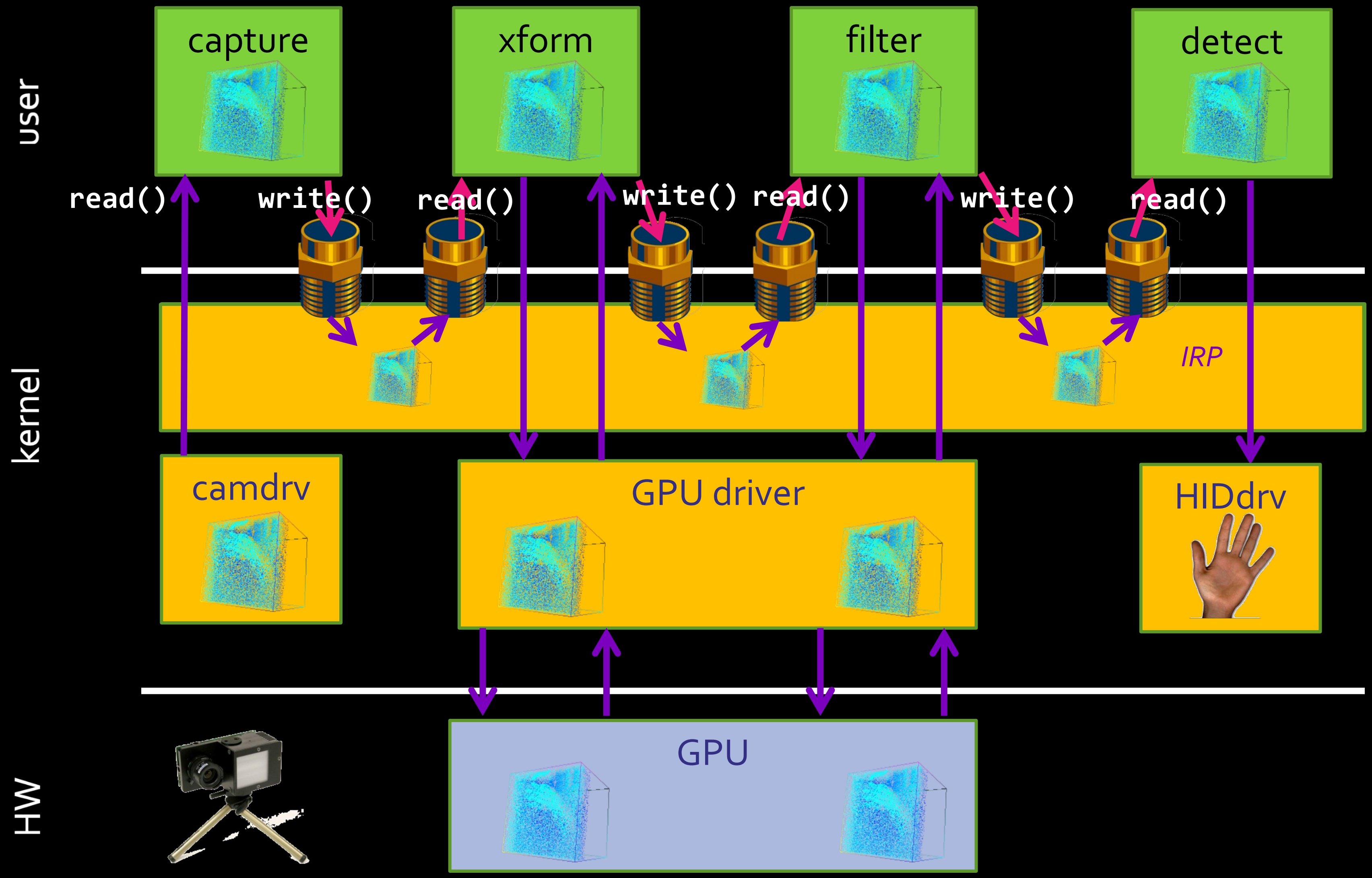
Data migration

#> capture | xform | filter | detect &



Data migration

#> capture | xform | filter | detect &



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

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

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!

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

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

```
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

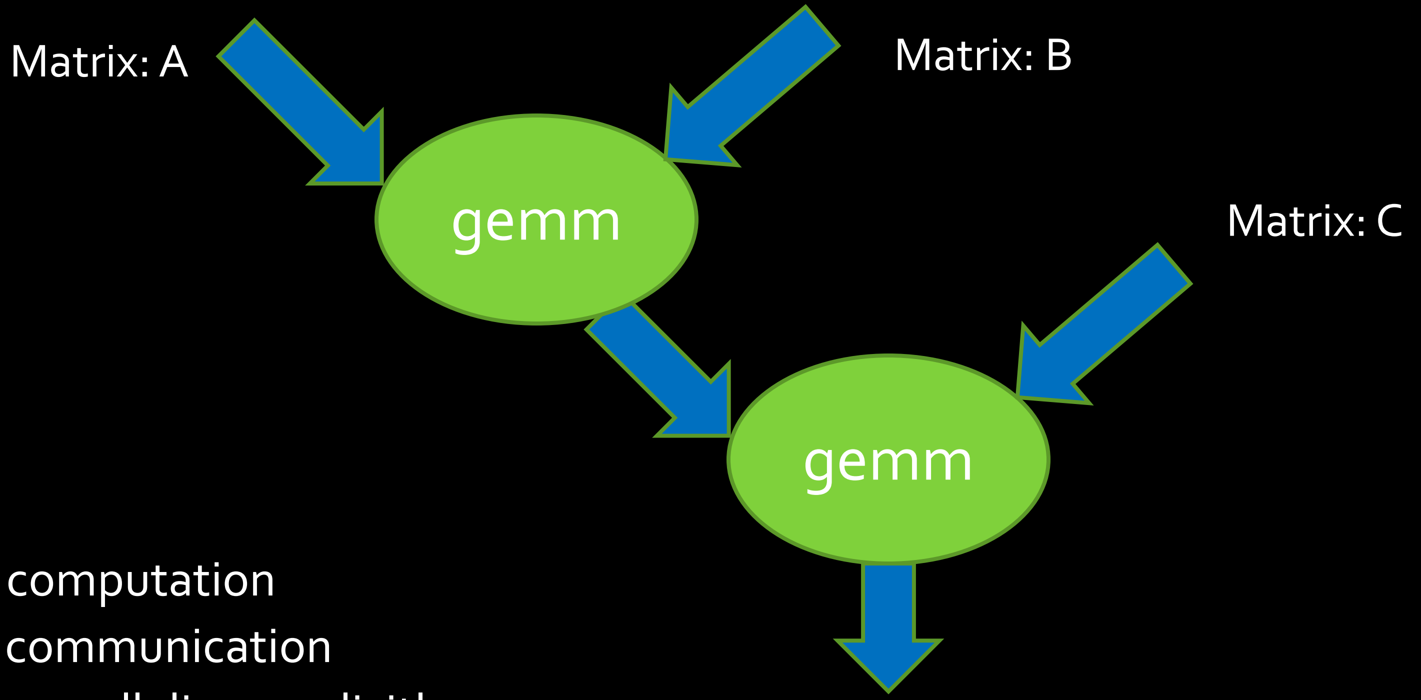
```
Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```

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;
}
```

Why don't we have this problem with CPUs?

Dataflow: a better abstraction

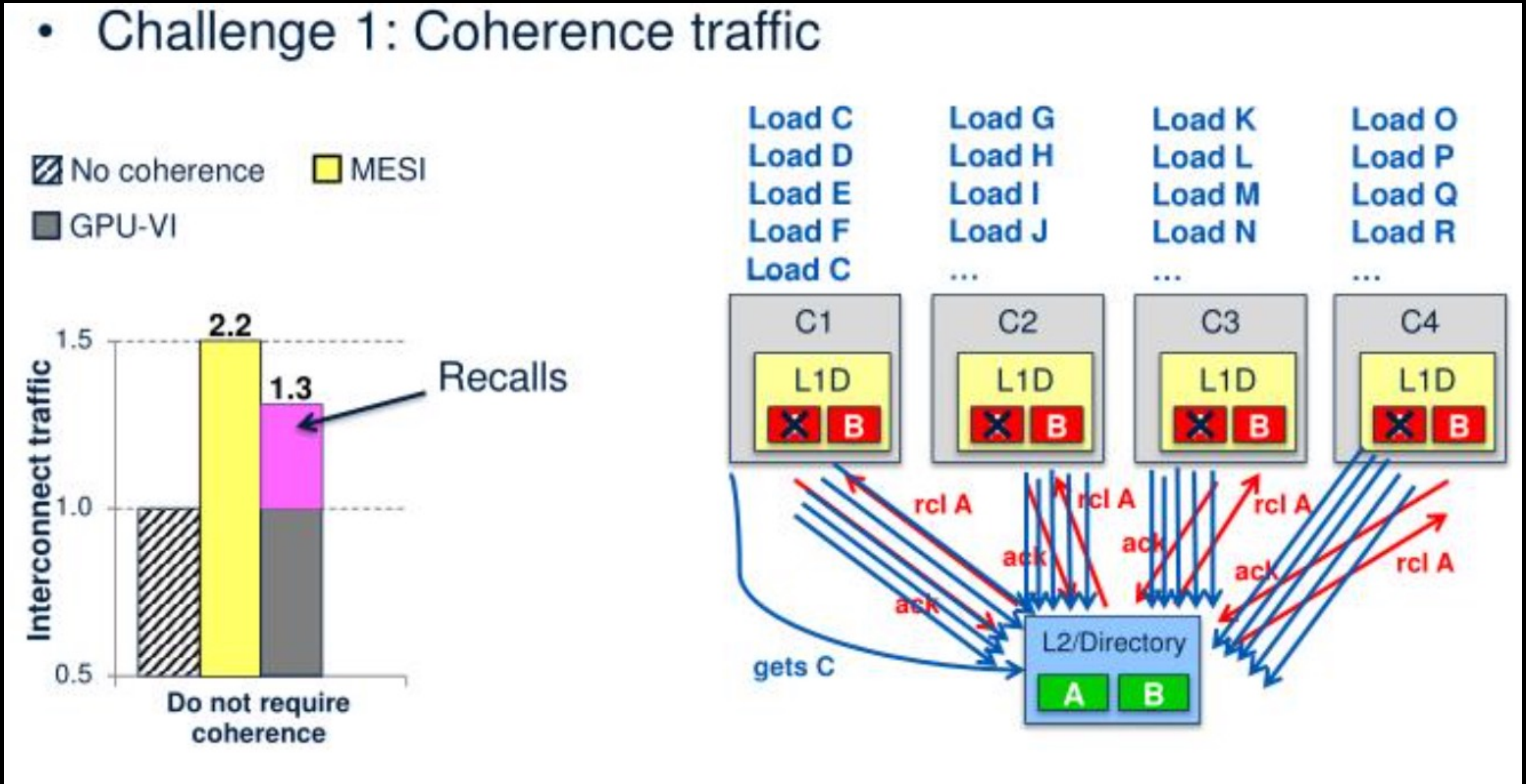


- nodes → computation
- edges → communication
- Expresses parallelism explicitly
- Minimal specification of data movement: runtime does it.
- asynchrony is a runtime concern (not programmer concern)
- No specification of compute→device mapping: like threads!

Faux Quiz Questions

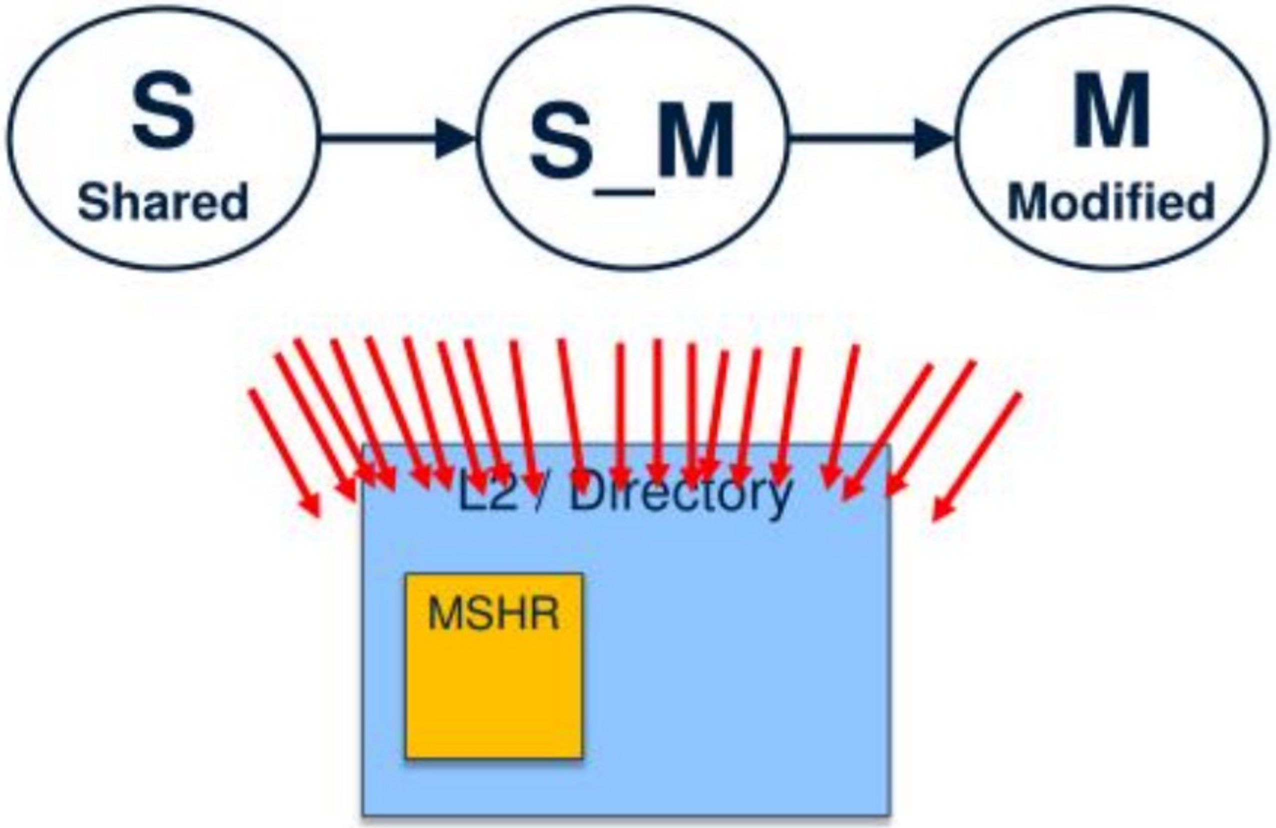
- How is occupancy defined (in CUDA nomenclature)?
- What's the difference between a block scheduler (e.g. Giga-Thread Engine) and a warp scheduler?
- Modern CUDA supports UVM to eliminate the need for cudaMalloc and cudaMemcpy*. Under what conditions might you want to use or not use it and why?
- What 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?

GPU Cache Coherence Challenges

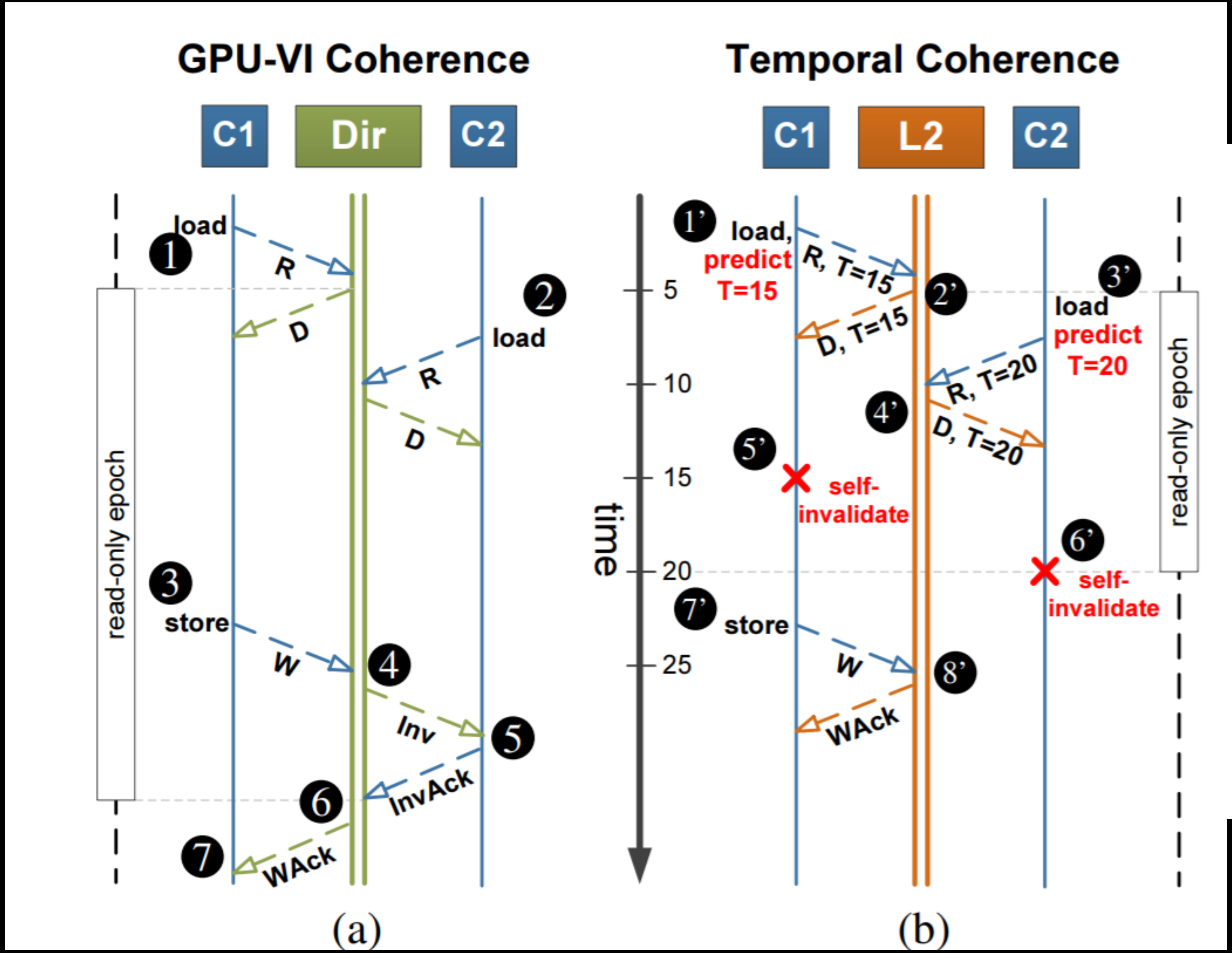


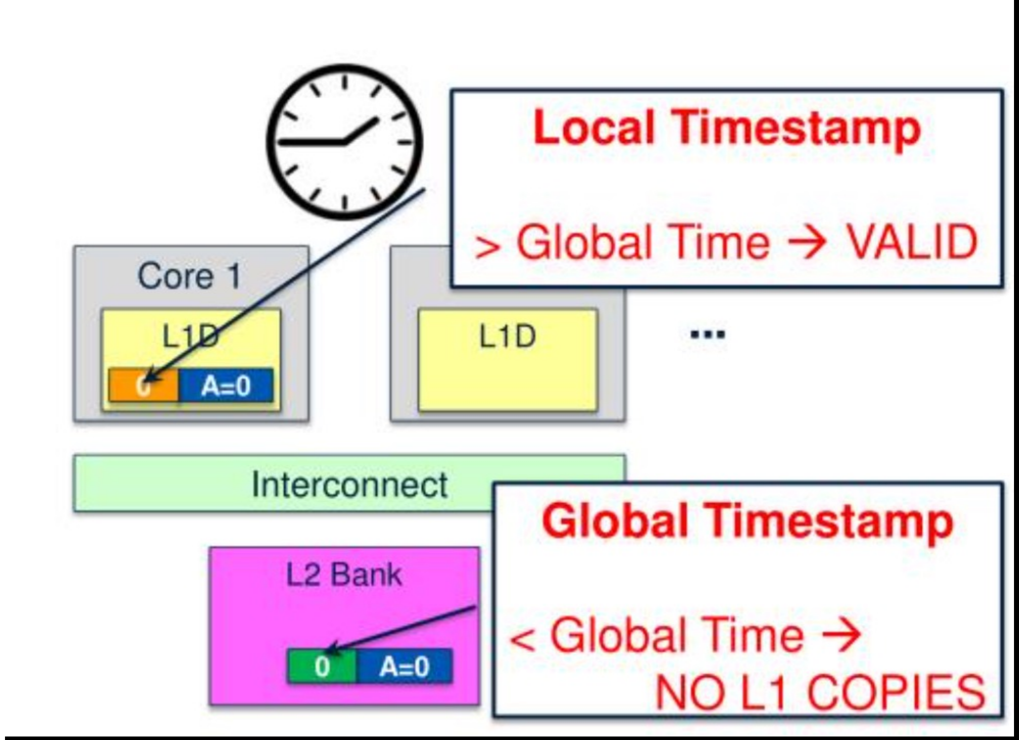
GPU Cache Coherence Challenges

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

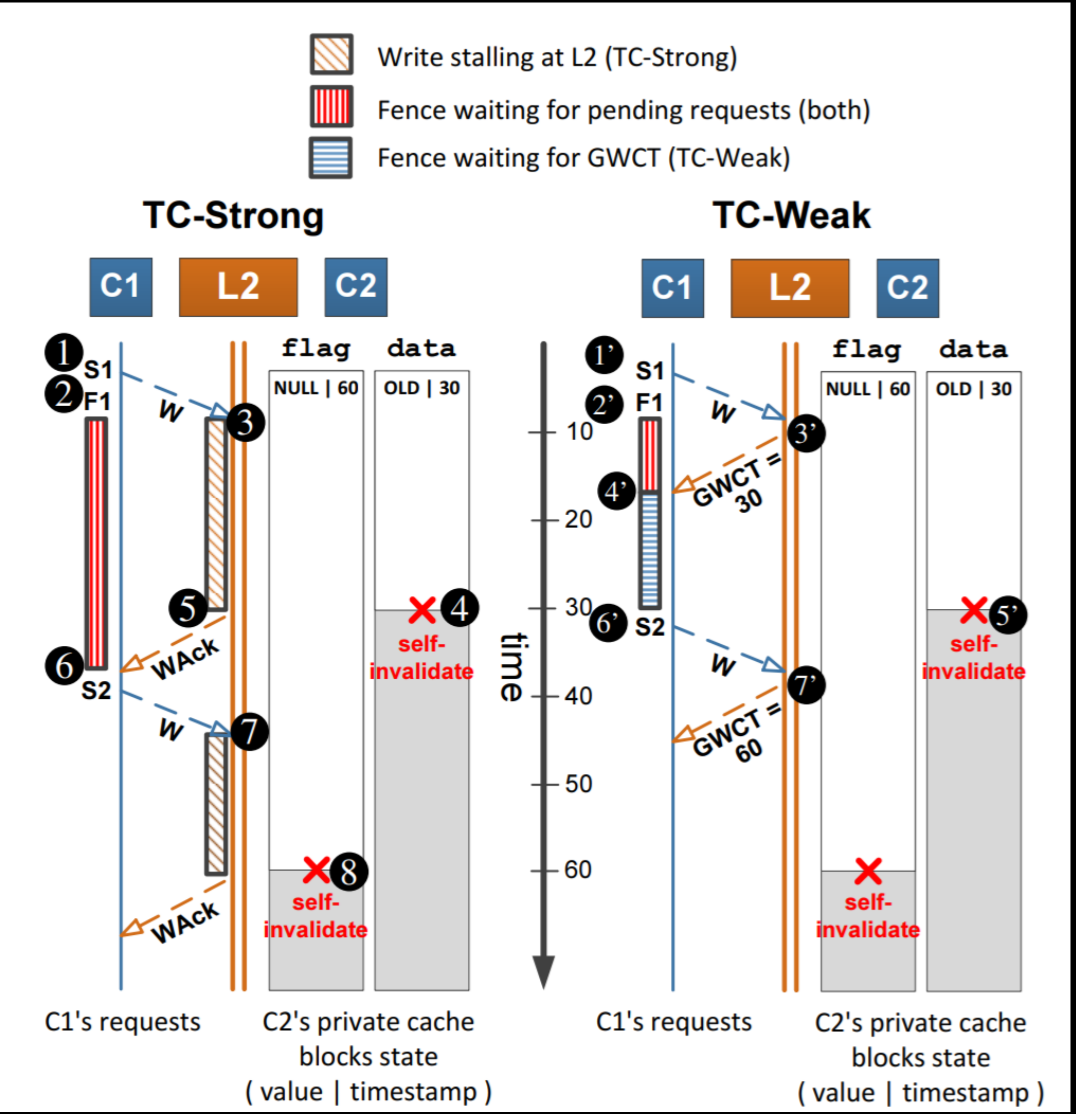


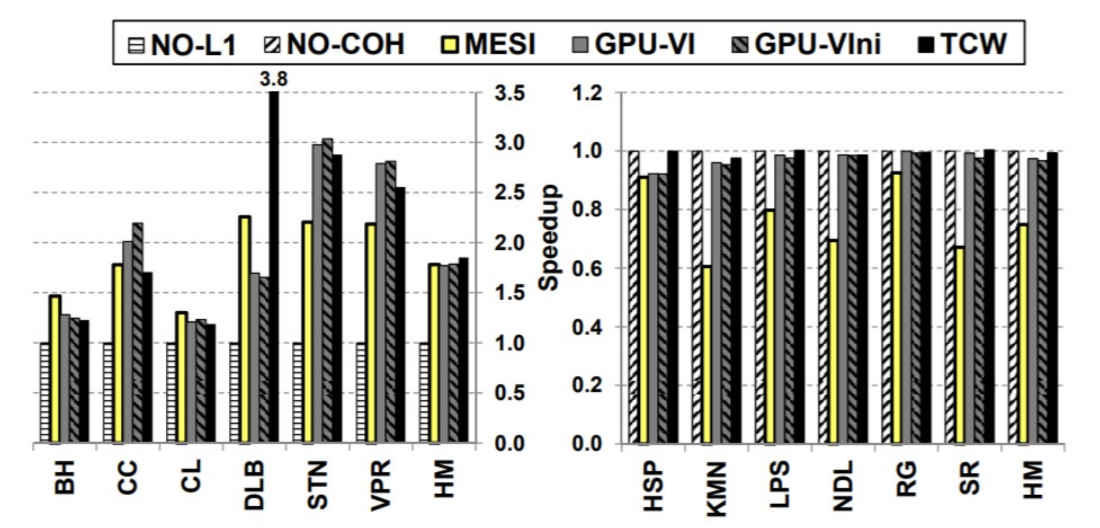
Temporal Coherence (TC)





TC-Strong vs TC-Weak





(a) Inter-workgroup comm. (b) Intra-workgroup comm.