Parallelizing the Naughty Dog engine using fibers

Christian Gyrling
Lead Programmer @ Naughty Dog

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

Evolution of our PS4 engine

- Solve the issues and limitations with our job system
  - All code need to be able to be jobified
  - Fibers
- CPU utilization is nowhere near 100%!
  - Process multiple frames at once
  - Memory lifetime

Background: PS3 Engine

- Single threaded engine (30Hz)
  - Game logic followed by command buffer setup
- SPUs were used as worker threads
  - Most engine systems were running on the SPUs
- Very little gameplay code ran on SPUs

Issues with our PS3 job system

- Jobs always ran to completion without ever yielding.
  - Complex to move gameplay onto SPUs
- User of the job system had to allocate/free resources
  - Job definitions and job lists (lifetime issues)
- State of a job list was confusing
  - Possible to add jobs while the list was running/stopped
- Job synchronization through marker index in job array
  - Had to reset job array between uses because the index would get reused

Design goals for new job system

- Allow jobifying code that couldn't be moved to SPUs
- Jobs can yield to other jobs in the middle of execution
  - Example: Player update with kick and wait for ray casts
- Easy to use API for gameplay programmers
- No memory management for the user
- One simple way to synchronize/chain jobs
- Performance was secondary to ease-of-use of the API

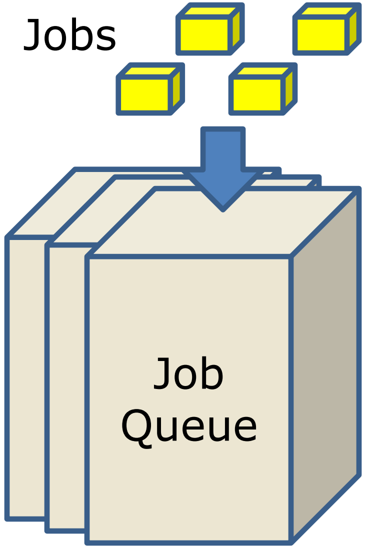
Fibers

- Like a partial thread
  - User provided stack space
  - Small context containing state of fiber and saved registers
- Executed by a thread
- Cooperative multi-threading (no preemption)
  - Switching between fibers is explicit (sceFiberSwitch on PS4)
    - Other operating systems have similar functionality
- Minimal overhead
  - No thread context switching when changing between fibers. Only register save/restore. (program counter, stack pointer, gprs…)

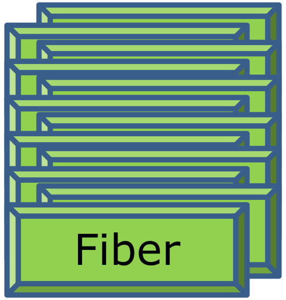
Our job system

- 6 worker threads
  - Each one is locked to a CPU core
- A thread is the execution unit, the fiber is the context.
- A job always executes within the context of a fiber
- Atomic counters used for synchronization
- Fibers
  - 160 fibers (128 x 64 KiB stack, 32 x 512 KiB stack)
- 3 job queues (Low/Normal/High priority)
  - No job stealing

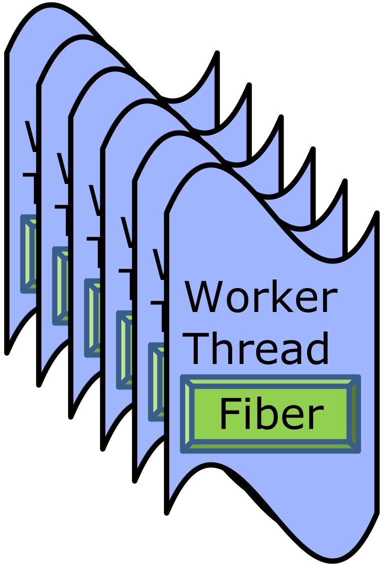
Job System



3 Job Queues
Low, Normal, High Priority

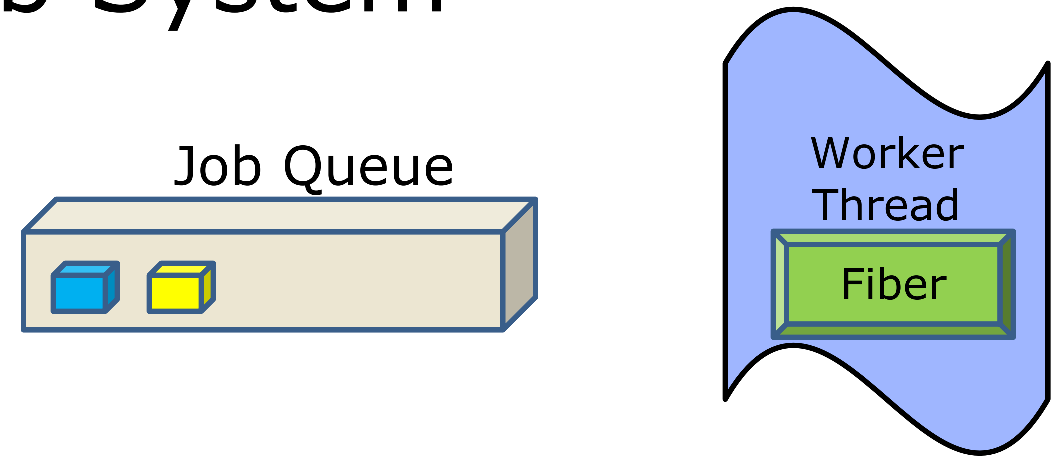


Fiber Pool - 160 Fibers
Stack & Registers

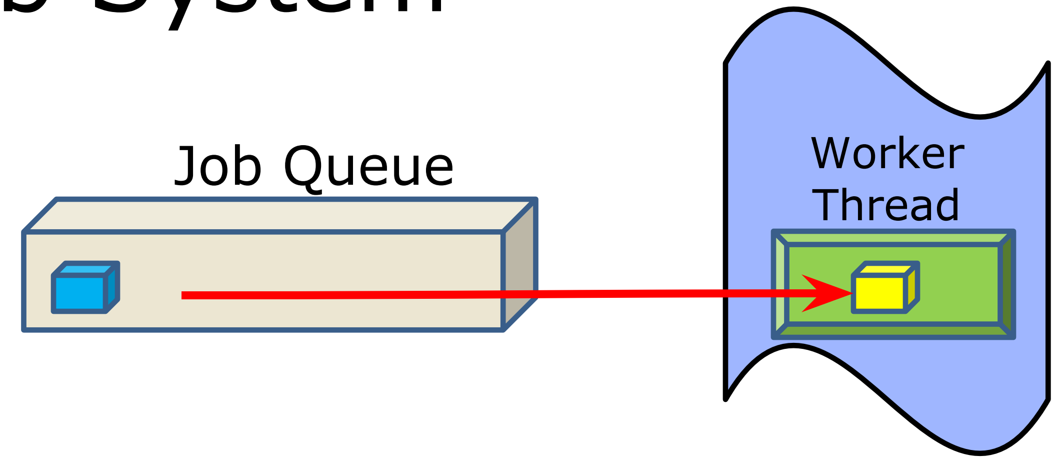


6 Worker Threads

Job System

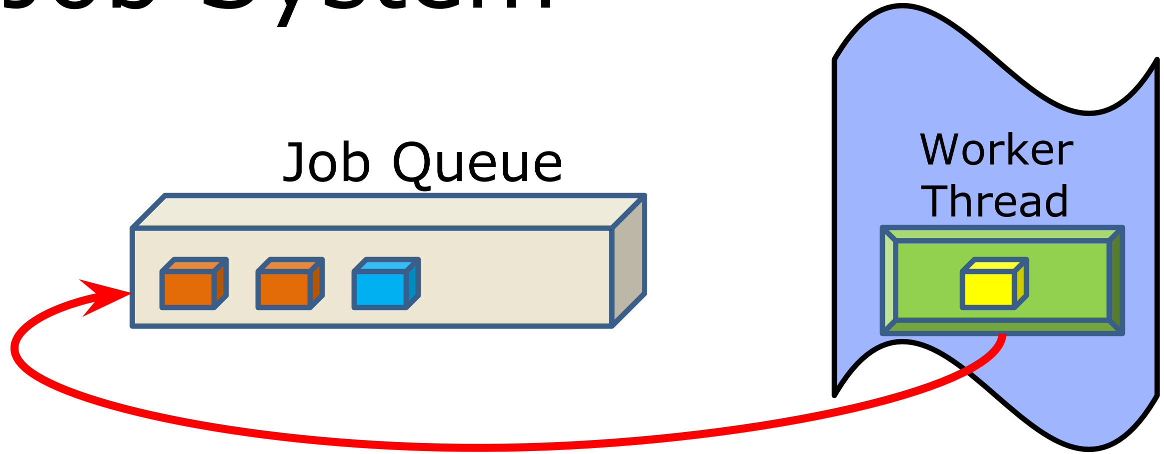


Job System



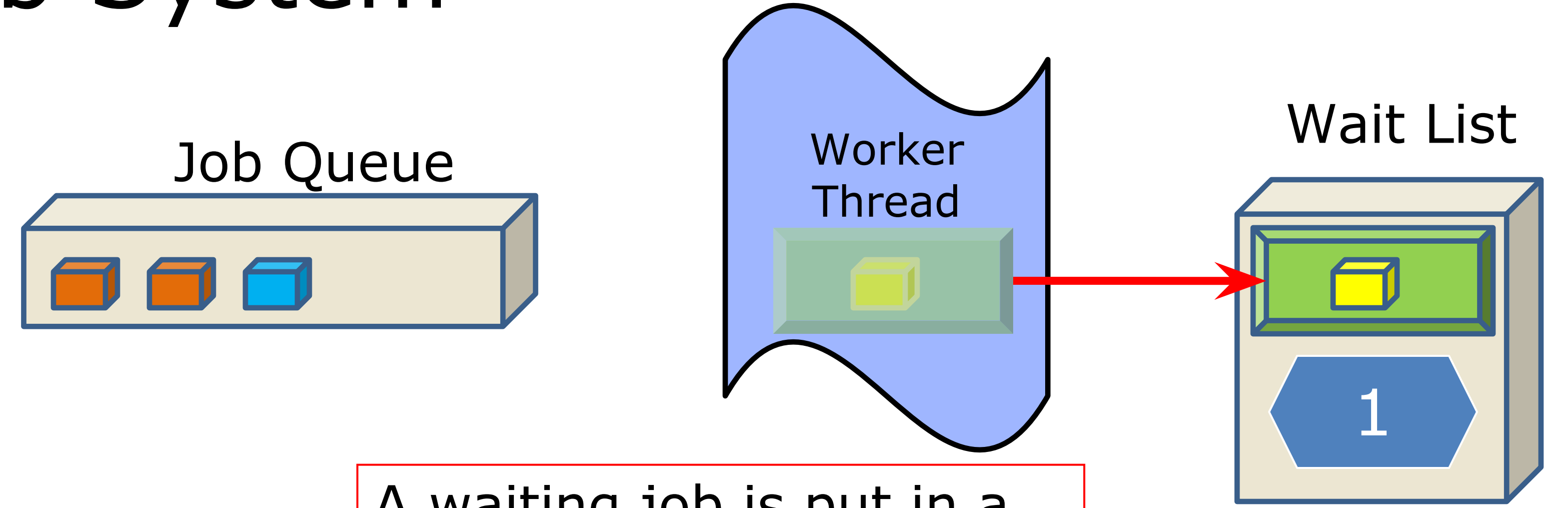
Jobs always execute inside a fiber context

Job System



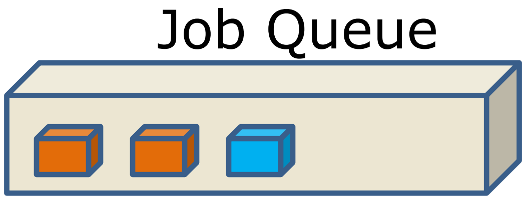
Jobs can add more jobs

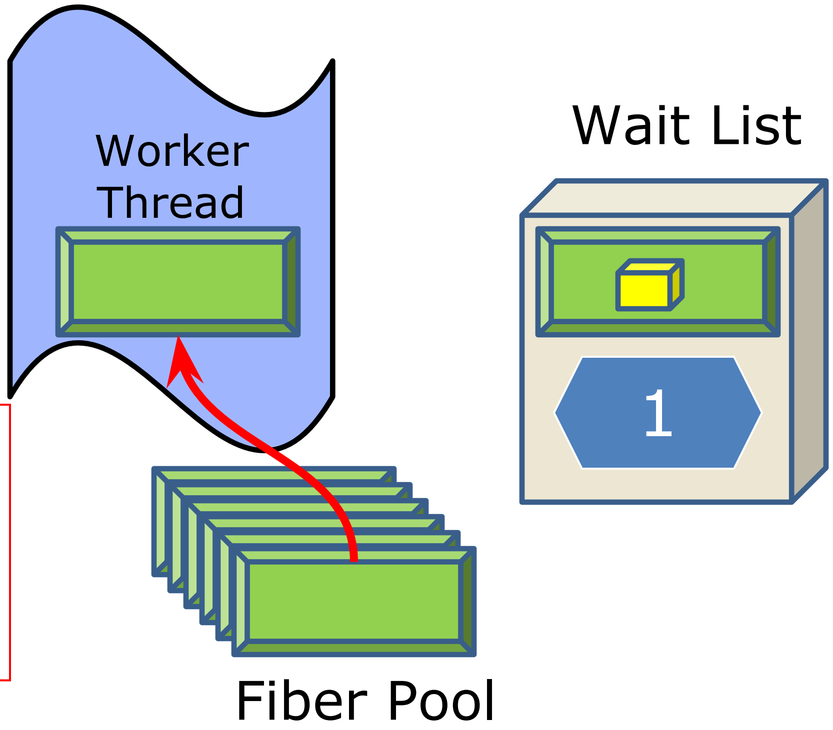
Job System



A waiting job is put in a wait list associated with the counter waited on

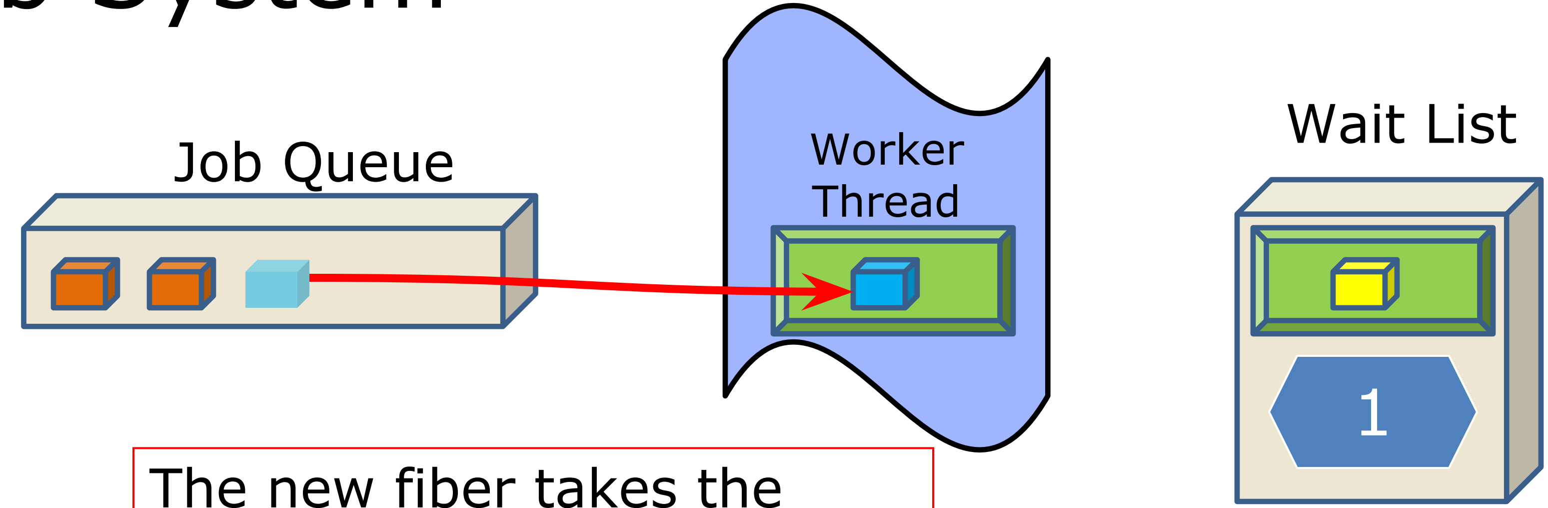
Job System





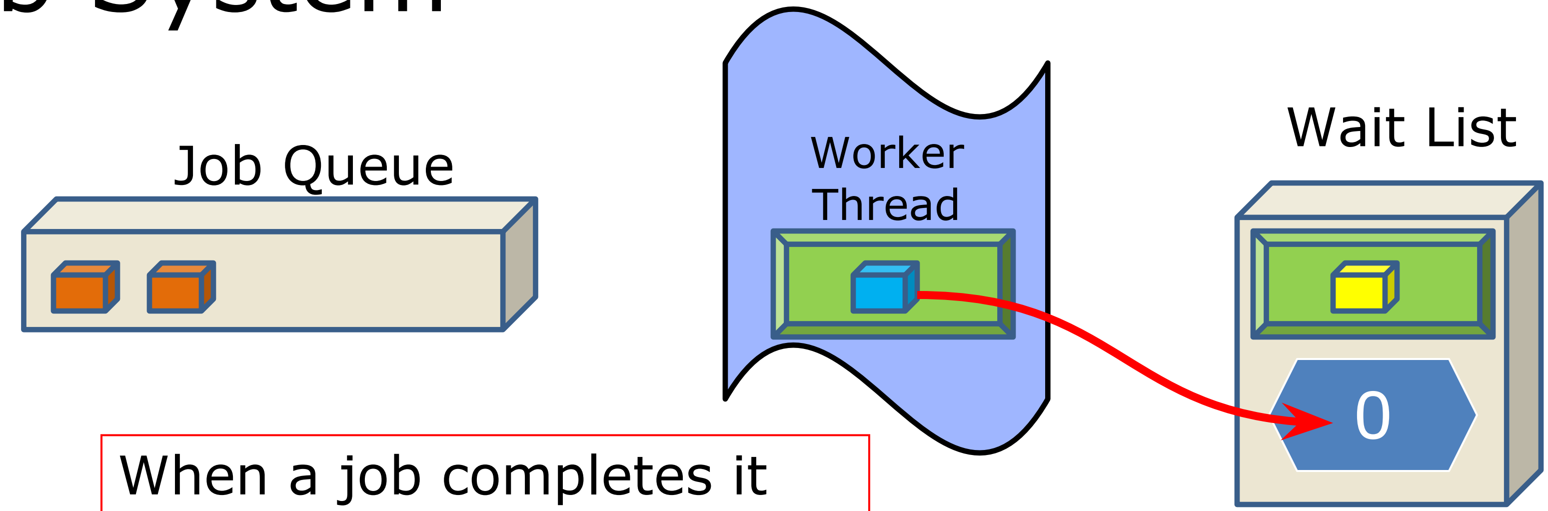
A new fiber is used to allow another job to begin executing. The stack of the waiting job is preserved in the fiber

Job System



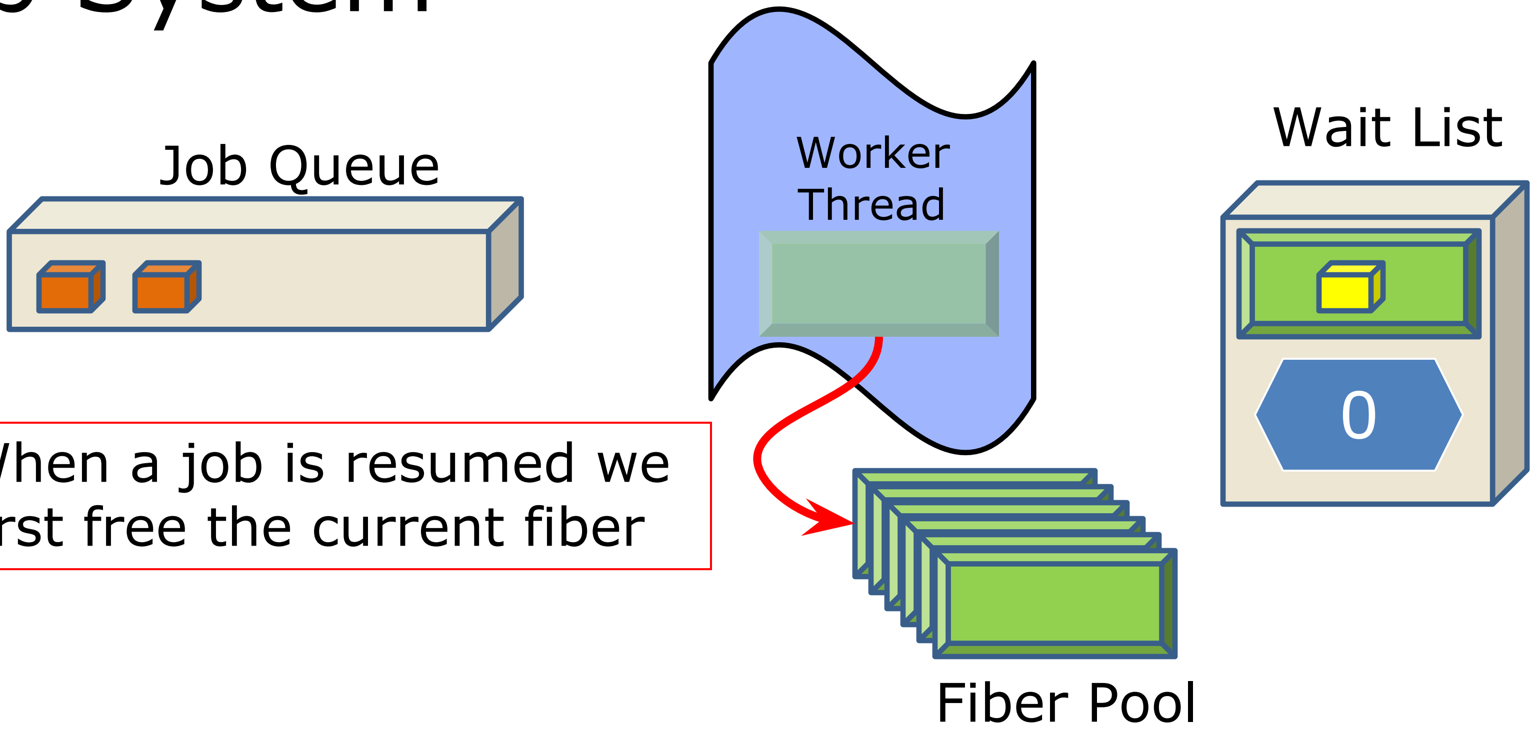
The new fiber takes the next job from the job queue and begins executing it

Job System



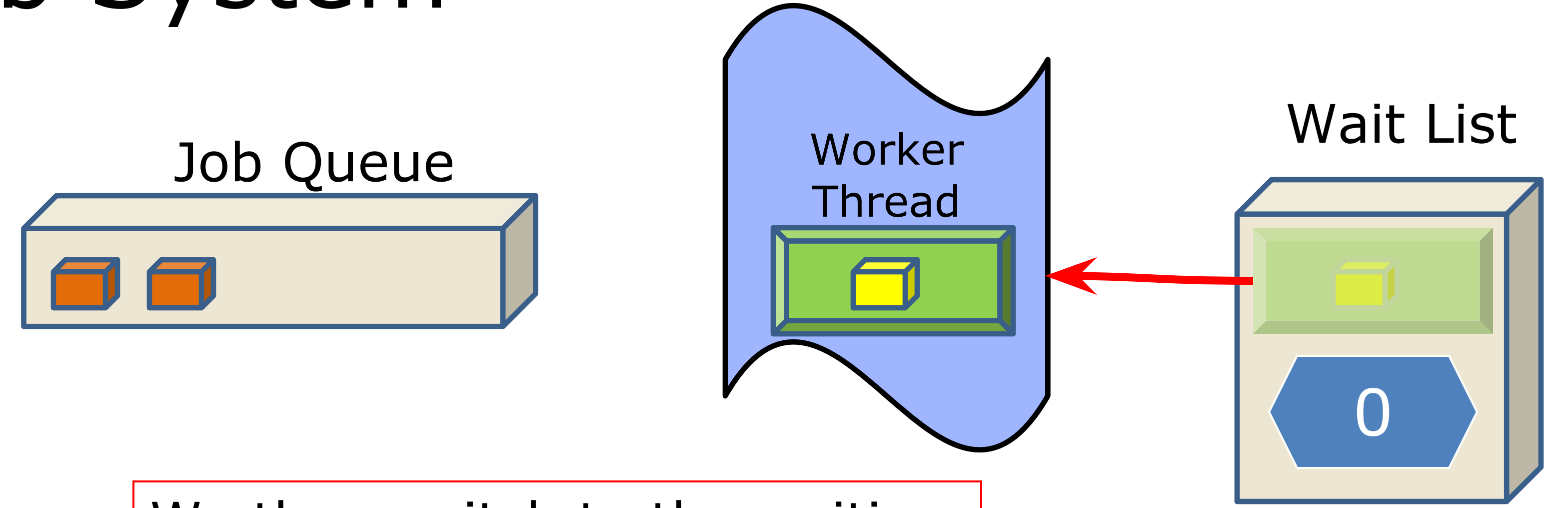
When a job completes it will decrement an associated counter and wake up any waiting jobs

Job System



When a job is resumed we first free the current fiber

Job System



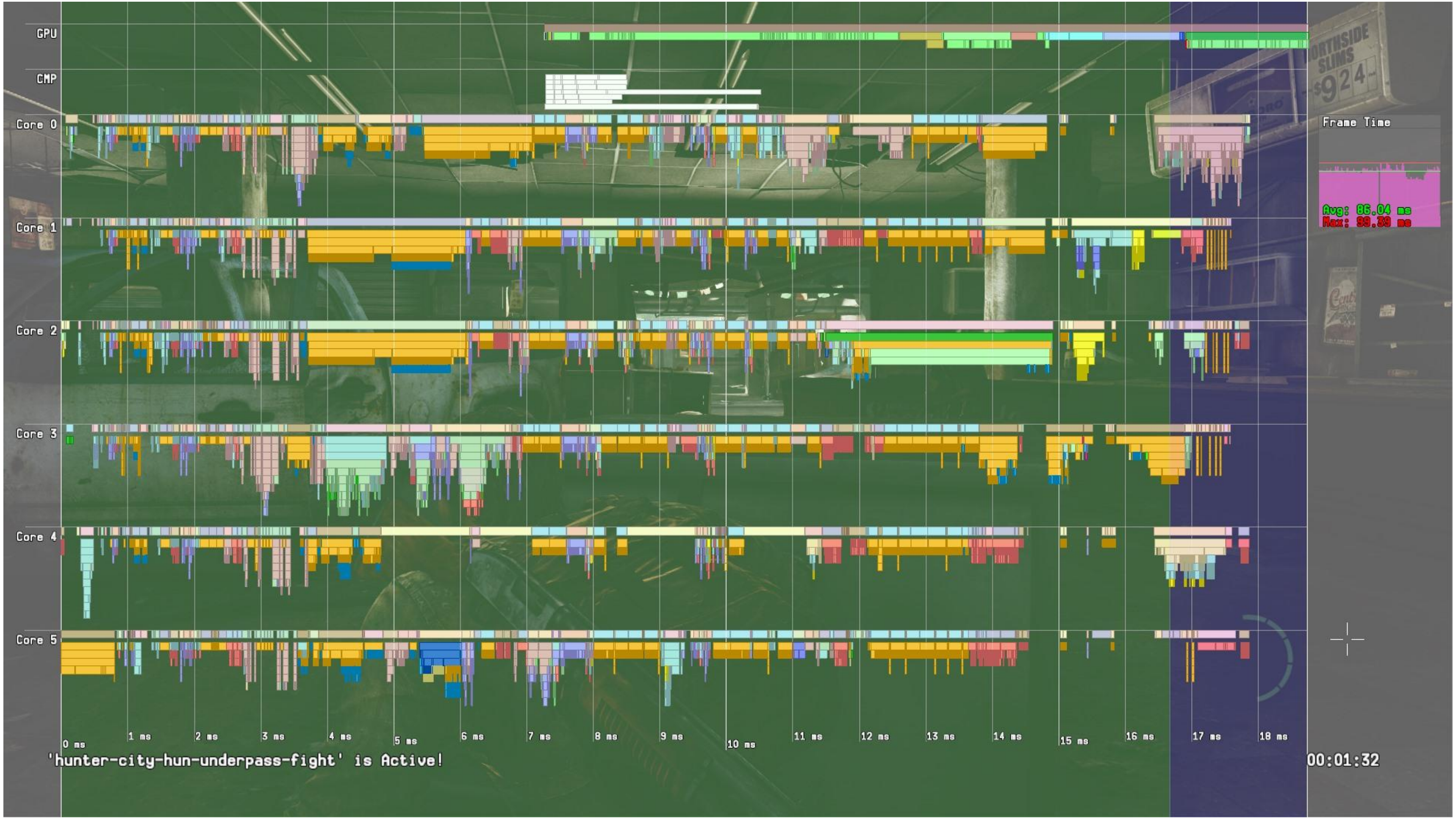
We then switch to the waiting fiber and resume execution

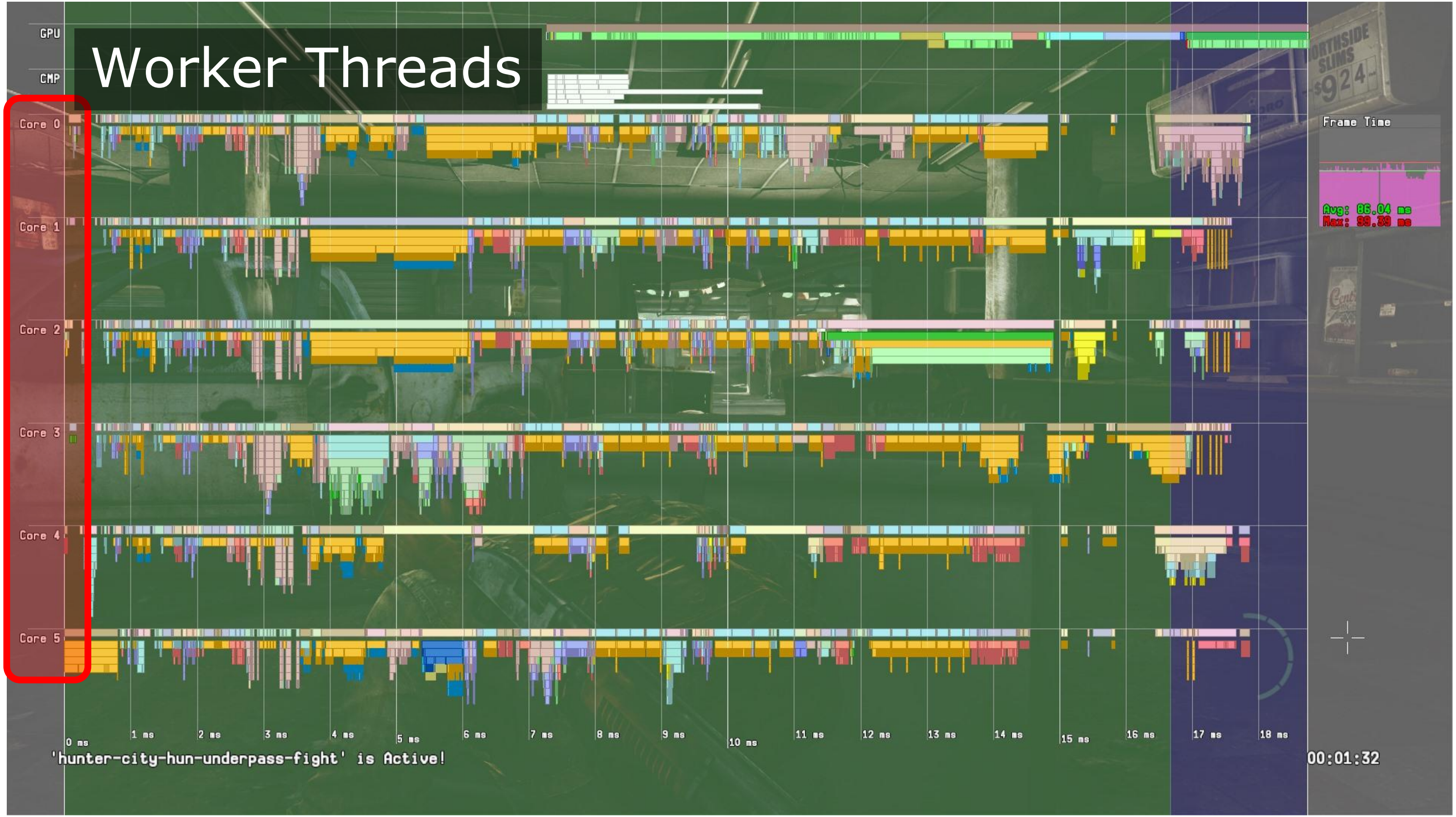
Job System Details

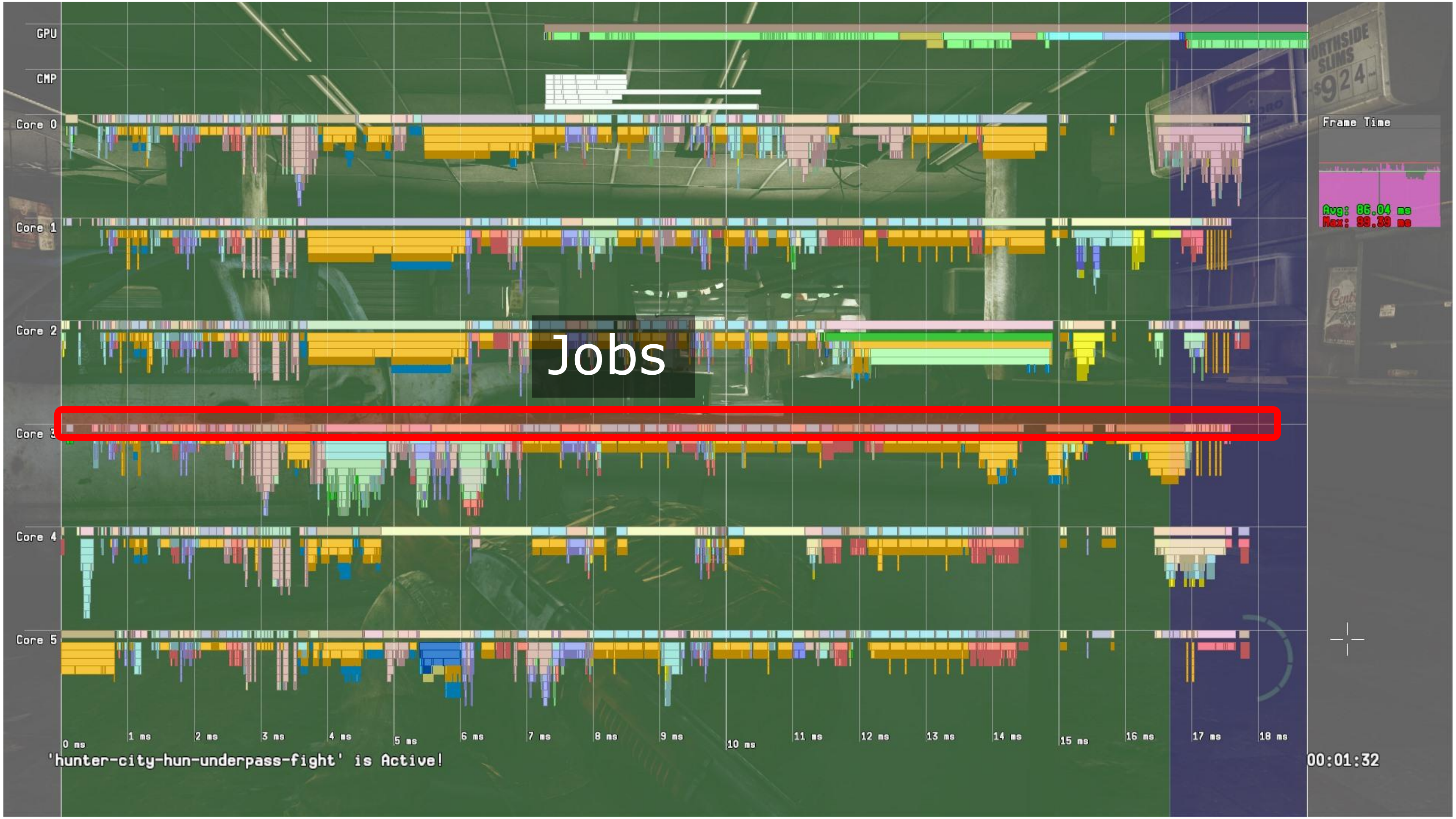
- Worker threads are locked to cores
  - Avoid context switches and unwanted core switching
  - Kernel threads can otherwise cause ripple effects across the cores
- ndjob::WaitForCounter(counter, 0)
  - Can be waited on by any job (moves fiber to wait list)
  - Only way of synchronizing jobs
  - Being able to yield a job at any time is extremely powerful!
- ~800-1000 jobs per frame in ‘The Last of Us: Remastered’

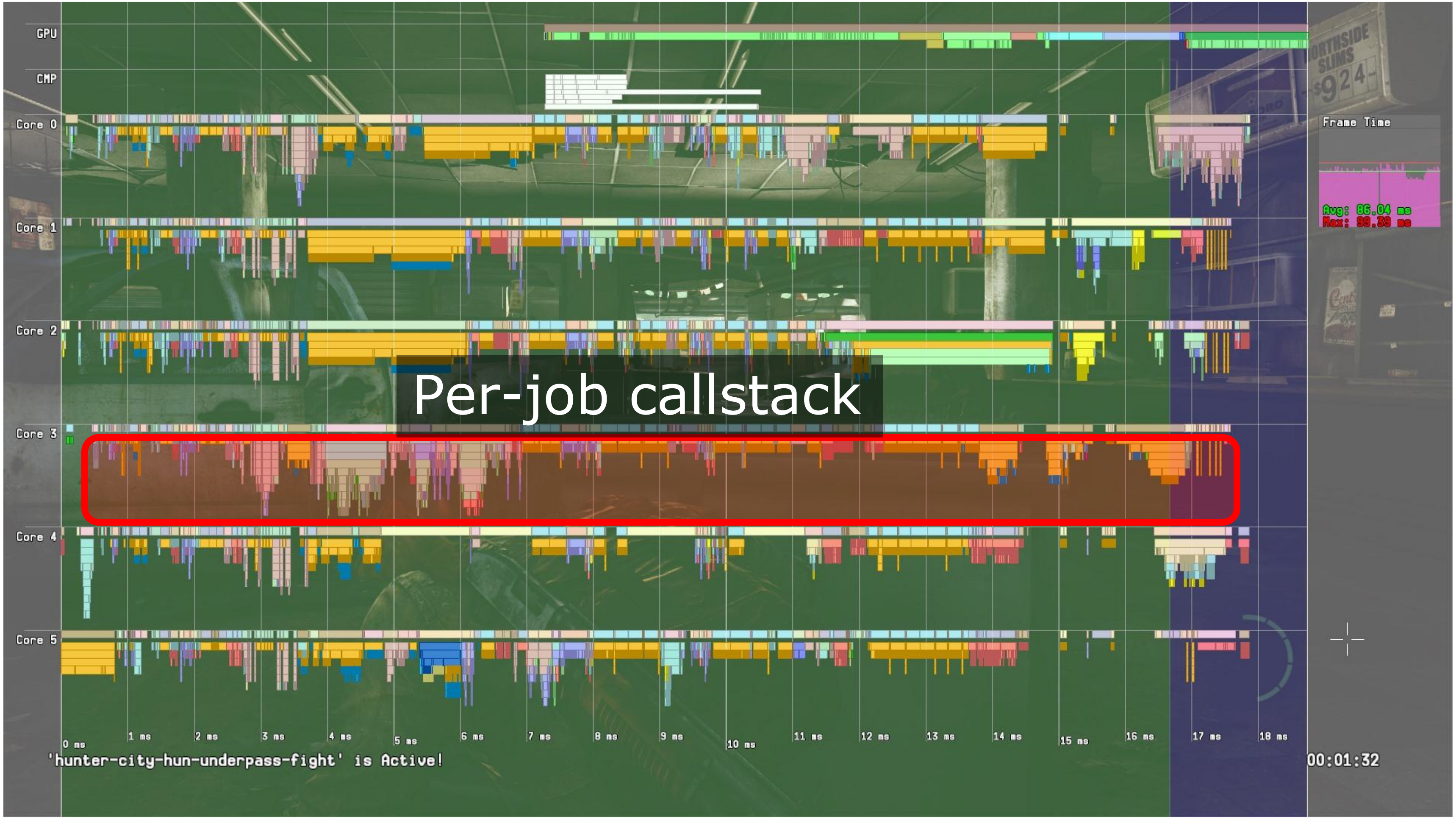
Everything is a job

- Game object updates
- Animation updates & joint blending
- Ray casts
- Command buffer generation
- Except "I/O threads" (sockets, file I/O, system calls…)
    - These are system threads
    - Implemented like interrupt handlers (Read data, post new job)
    - Always waiting and never do expensive processing of the data


GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Avg: 86.04 ms
Max: 99.39 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
6 ms
7 ms
8 ms
9 ms
10 ms
11 ms
12 ms
13 ms
14 ms
15 ms
16 ms
17 ms
18 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:32


Worker Threads
GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Avg: 86.04 ms
Max: 99.39 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
6 ms
7 ms
8 ms
9 ms
10 ms
11 ms
12 ms
13 ms
14 ms
15 ms
16 ms
17 ms
18 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:32


GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Jobs
Frame Time
Avg: 86.04 ms
Max: 99.39 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
6 ms
7 ms
8 ms
9 ms
10 ms
11 ms
12 ms
13 ms
14 ms
15 ms
16 ms
17 ms
18 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:32


GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Per-job callstack
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
6 ms
7 ms
8 ms
9 ms
10 ms
11 ms
12 ms
13 ms
14 ms
15 ms
16 ms
17 ms
18 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:32

Example – Animate All Objects

```
Object g_Objects[100];

void AnimateObject(void* pObj)
{
	…
}

void AnimateAllObjects()
{
	for (int objIndex = 0; objIndex < 100; ++objIndex)
	{
		AnimateObject(&g_Objects[objIndex ]);
	}
}
```

Example – Animate All Objects

```
Object g_Objects[100];

void AnimateObject(void* pObj)
{
	…
}

void AnimateAllObjects()
{
	for (int objIndex = 0; objIndex < 100; ++ objIndex )
	{
		AnimateObject(&g_Objects[objIndex]);
	}
}
```

Example – Animate All Objects

```
Object g_Objects[100];

JOB_ENTRY_POINT(AnimateObjectJob)
{
	...
}

void AnimateAllObjects()
{
	JobDecl jobDecls[100];
	for (int jobIndex = 0; jobIndex < 100; ++jobIndex)
	{
		jobDecls[jobIndex] = JobDecl(AnimateObjectJob, &g_Objects[jobIndex]);
	}
	Counter* pJobCounter = NULL;
	RunJobs(&jobDecls, 100, &pJobCounter);
	WaitForCounterAndFree(pJobCounter, 0);
}
```

Example – Animate All Objects

```
Object g_Objects[100];

JOB_ENTRY_POINT(AnimateObjectJob)
{
	…
}

void AnimateAllObjects()
{
	JobDecl jobDecls[100];
	for (int jobIndex = 0; jobIndex < 100; ++jobIndex)
	{
		jobDecls[jobIndex] = JobDecl(AnimateObjectJob, &g_Objects[jobIndex]);
	}
	Counter* pJobCounter = NULL;
	RunJobs(&jobDecls, 100, &pJobCounter);
	WaitForCounterAndFree(pJobCounter, 0);
}
```

Jobs can be created on the stack

Example – Animate All Objects

Fill in the job declarations with entry point and parameter

```
Object g_Objects[100];

JOB_ENTRY_POINT(AnimateObjectJob)
{
	...
}

void AnimateAllObjects()
{
	JobDecl jobDecls[100];
	for (int jobIndex = 0; jobIndex < 100; ++jobIndex)
	{
		jobDecls[jobIndex] = JobDecl(AnimateObjectJob, &g_Objects[jobIndex]);
	}
	Counter* pJobCounter = NULL;
	RunJobs(&jobDecls, 100, &pJobCounter);
	WaitForCounterAndFree(pJobCounter, 0);
}
```

Example – Animate All Objects

```
Object g_Objects[100];

JOB_ENTRY_POINT(AnimateObjectJob)
{
	…
}

void AnimateAllObjects()
{
	JobDecl jobDecls[100];
	for (int jobIndex = 0; jobIndex < 100; ++jobIndex)
	{
		jobDecls[jobIndex] = JobDecl(AnimateObjectJob, &g_Objects[jobIndex]);
	}
	Counter* pJobCounter = NULL;
	RunJobs(&jobDecls, 100, &pJobCounter);
	WaitForCounterAndFree(pJobCounter, 0);
}
```

The job entry point is easily defined

Example – Animate All Objects

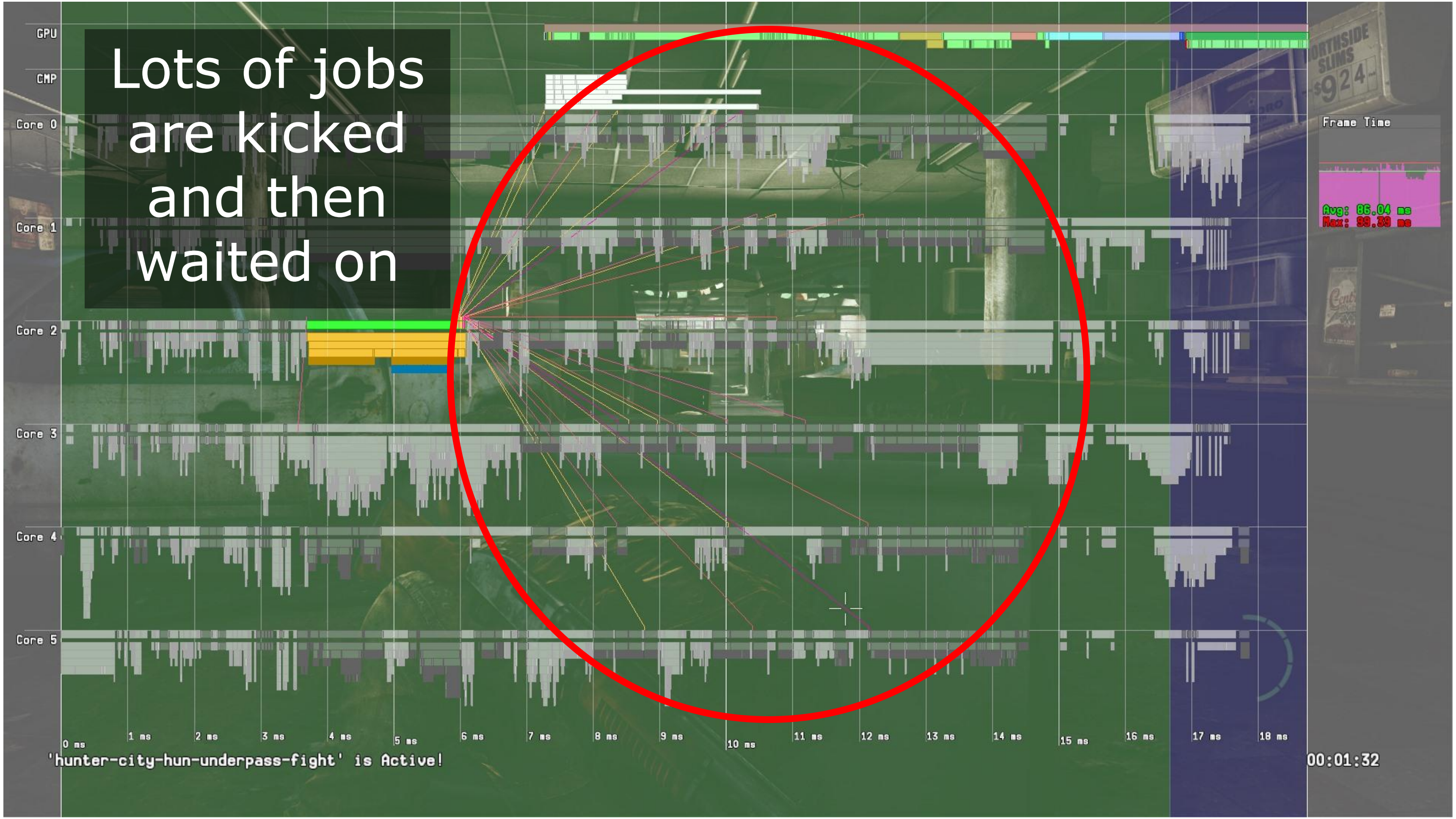
```
Object g_Objects[100];

JOB_ENTRY_POINT(AnimateObjectJob)
{
	…
}

void AnimateAllObjects()
{
	JobDecl jobDecls[100];
	for (int jobIndex = 0; jobIndex < 100; ++jobIndex)
	{
		jobDecls[jobIndex] = JobDecl(AnimateObjectJob, &g_Objects[jobIndex]);
	}
	Counter* pJobCounter = NULL;
	RunJobs(&jobDecls, 100, &pJobCounter);
	WaitForCounterAndFree(pJobCounter, 0);
}
```

Schedule all jobs and receive a counter that you can wait for

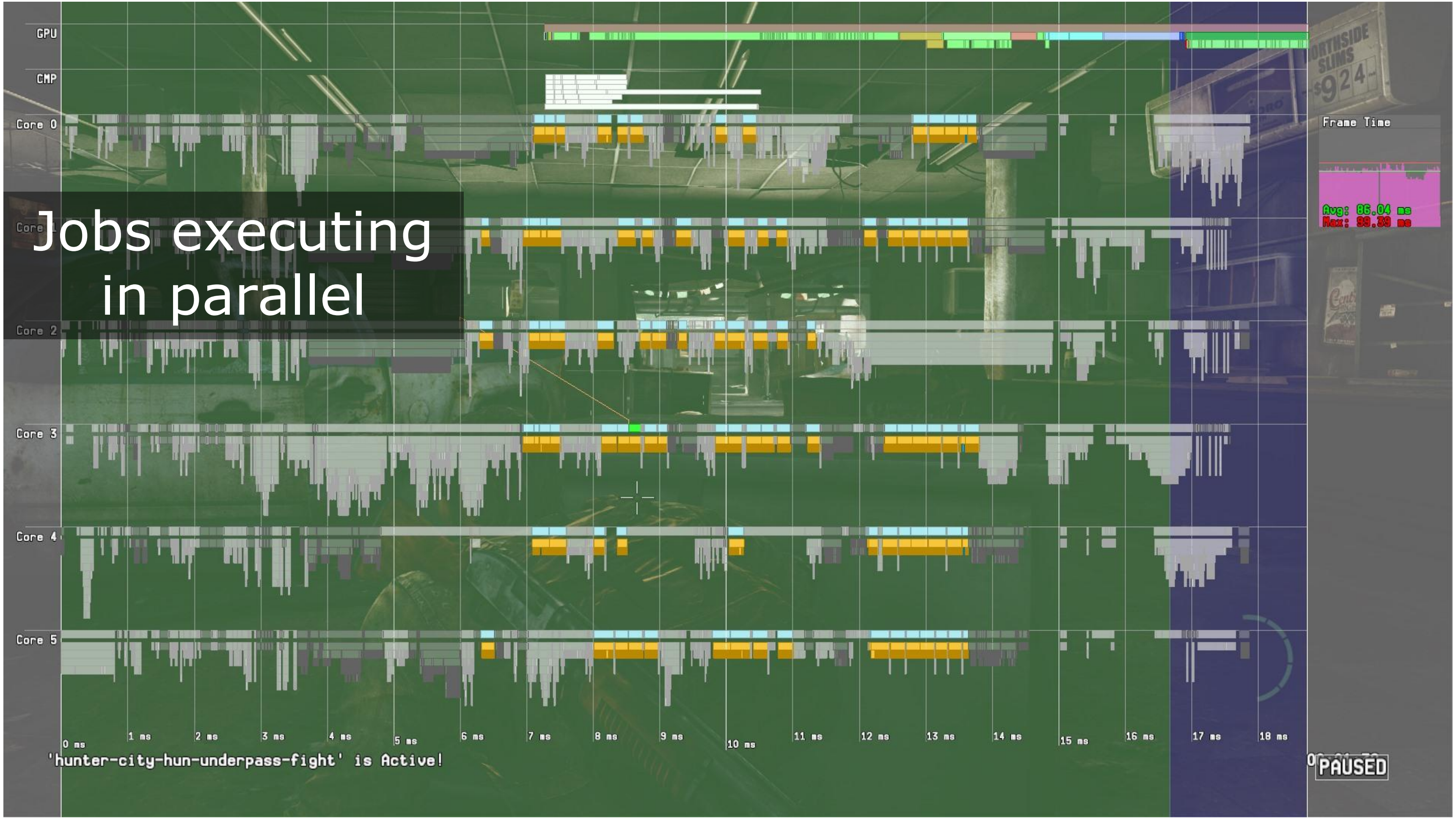
Example – Animate All Objects

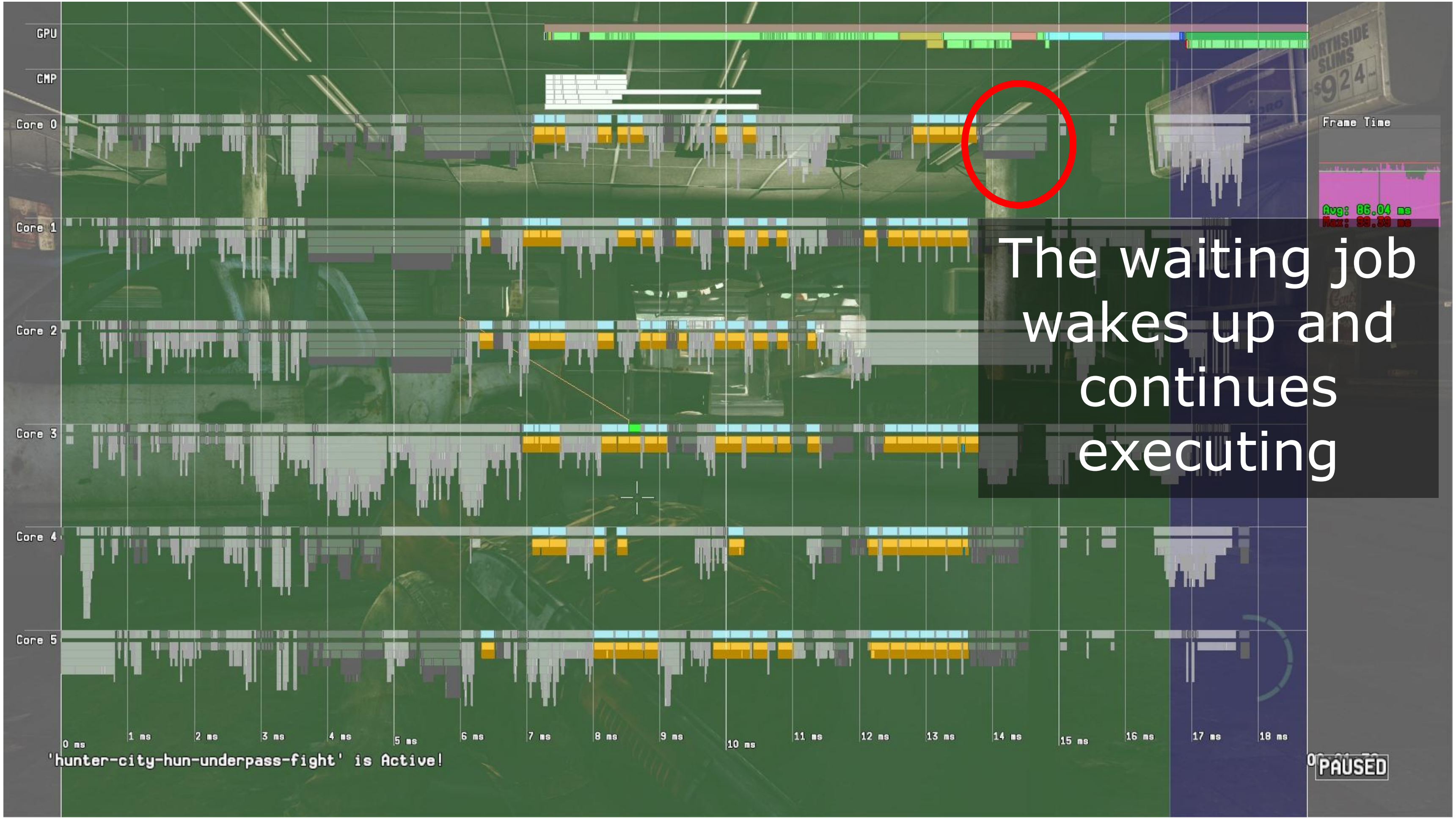
```
Object g_Objects[100];

JOB_ENTRY_POINT(AnimateObjectJob)
{
        …
}

void AnimateAllObjects()
{
        JobDecl jobDecls[100];
        for (int jobIndex = 0; jobIndex < 100; ++jobIndex)
        {
                jobDecls[jobIndex] = JobDecl(AnimateObjectJob, &g_Objects[jobIndex]);
        }
        Counter* pJobCounter = NULL;
        RunJobs(&jobDecls, 100, &pJobCounter);
        WaitForCounterAndFree(pJobCounter, 0);
}
```

Wait for all jobs by waiting for the counter to be zero. Job goes to sleep.


Lots of jobs are kicked and then waited on
GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
6 ms
7 ms
8 ms
9 ms
10 ms
11 ms
12 ms
13 ms
14 ms
15 ms
16 ms
17 ms
18 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:32

Jobs executing in parallel





The waiting job wakes up and continues executing

Pros of new job system

- Extremely easy to jobify existing gameplay updates
  - Deep call stacks are no problem
- Having one job wait for another is straight-forward
  - WaitForCounter(…)
- Super-lightweight to switch fibers
  - System supported operation - sceFiberSwitch() on PS4
  - Save the program counter and stack pointer…
    - …and all the other registers
  - Restore the program counter and stack pointer…
    - …and all the other registers

Cons of new job system

- System synchronization primitives can no longer be used
  - Mutex, semaphore, condition variables…
  - Locked to a particular thread. Fibers migrate between threads
- Synchronization has to be done on the hardware level
  - Atomic spin locks are used almost everywhere
  - Special job mutex is used for locks held longer
    - Puts the current job to sleep if needed instead of spin lock

Support For Fibers

- Fibers and their call stacks are viewable in the debugger
  - Can inspect fibers just like you would inspect threads
- Fibers can be named/renamed
  - Indicate current job
- Crash handling
  - Fiber call stacks are saved in the core dumps just like threads

Support For Fibers…

- Fiber-Safe Thread Local Storage (TLS) Optimizations
  - Issue: TLS address is allowed to be cached for the duration of the function by default. Switch fiber in the middle of function and you wake up with wrong TLS pointer. ☹
  - Currently not supported by Clang
  - Workaround: Use separate CPP file for TLS access
- Use adaptive mutexes in job system
  - Jobs can be added from normal threads
    - Spin locks -> Dead locks
  - Spin and attempt to grab lock before doing system call
  - Solves priority inversion deadlocks
  - Can avoid most system calls due to initial spin

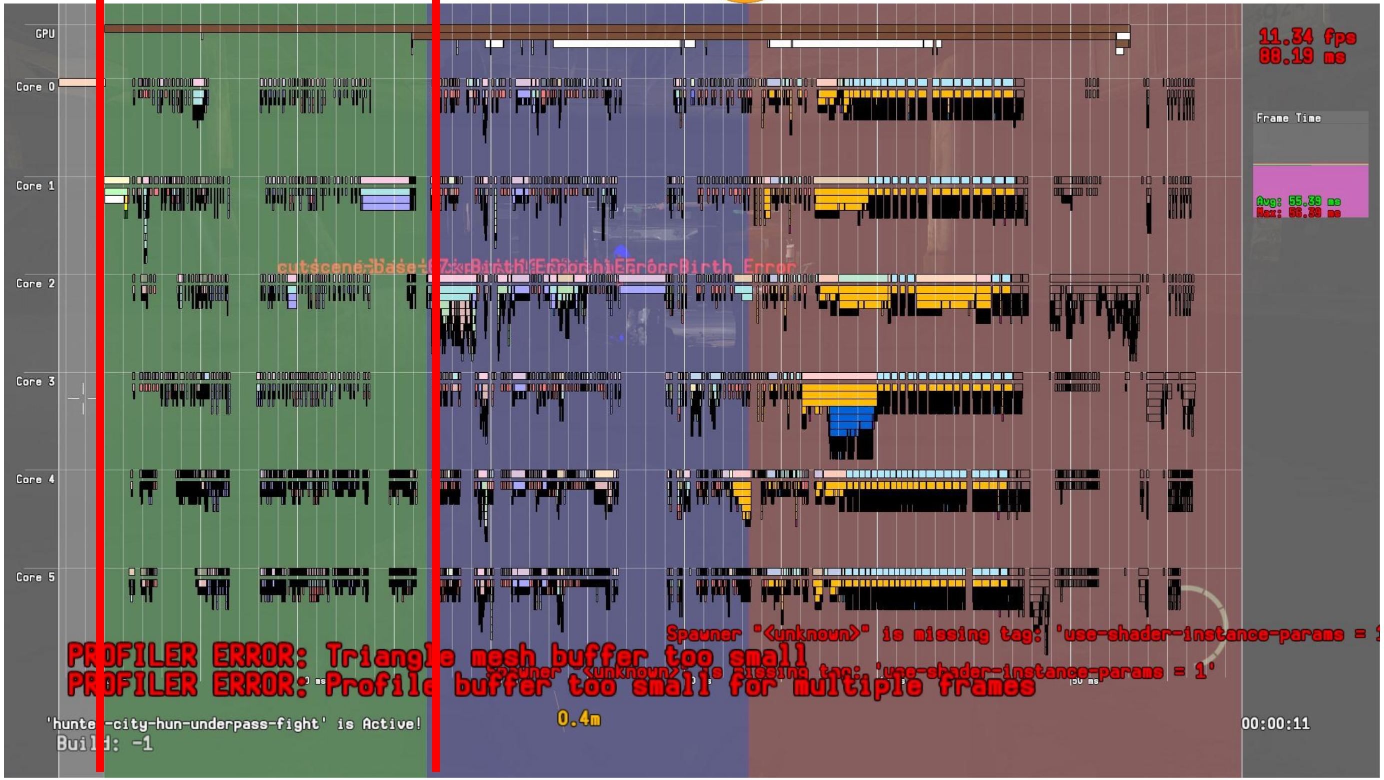
Other job systems

- Intel Thread Build Blocks (TBB) is popular
- Solves dependencies between jobs and chaining but requires every job to execute until completion…
  - … unless you want context switching
- OR allow job nesting while a job is waiting.
  - This will prevent the first job from resuming until the second job has completed.
- A fiber-based job system has none of these issues

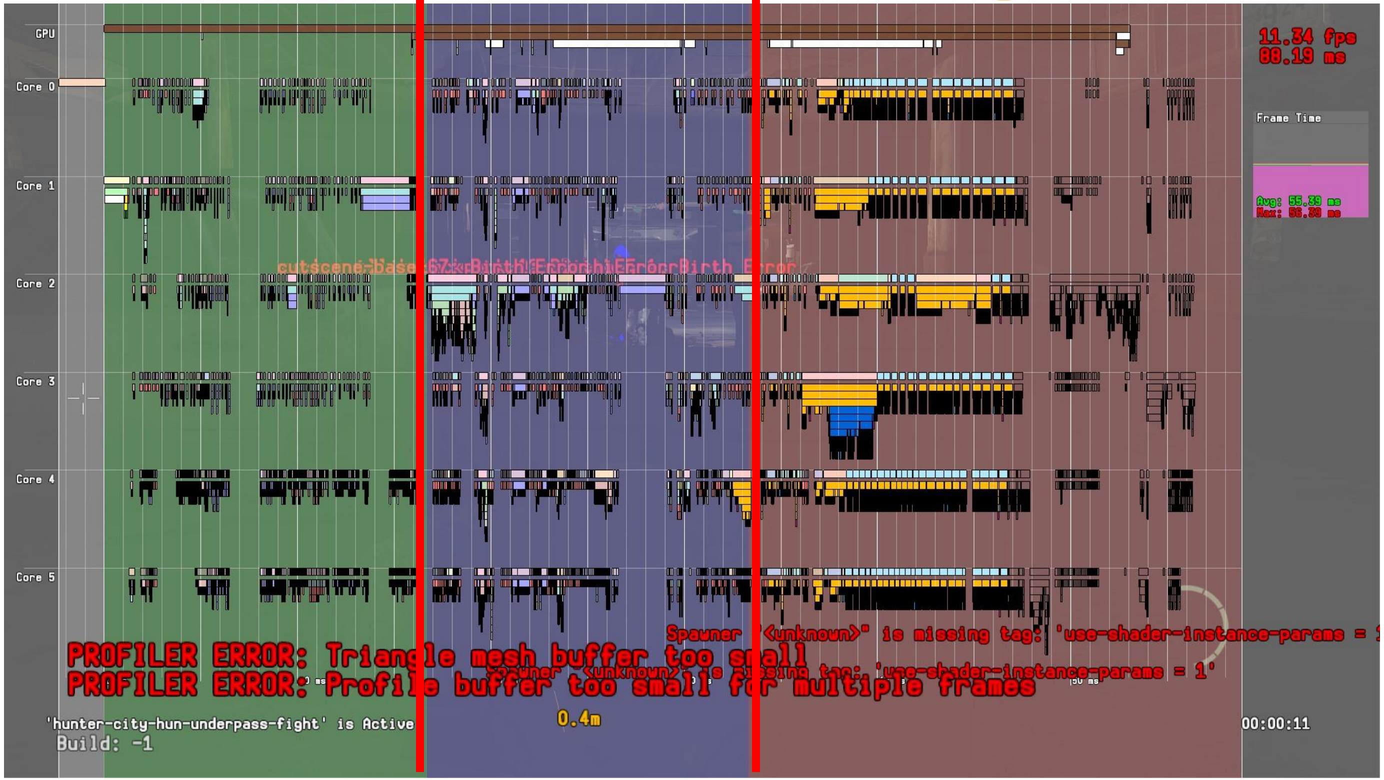
Moving on…

- We have our new job system
- All code can now be jobified
- Let's do it!

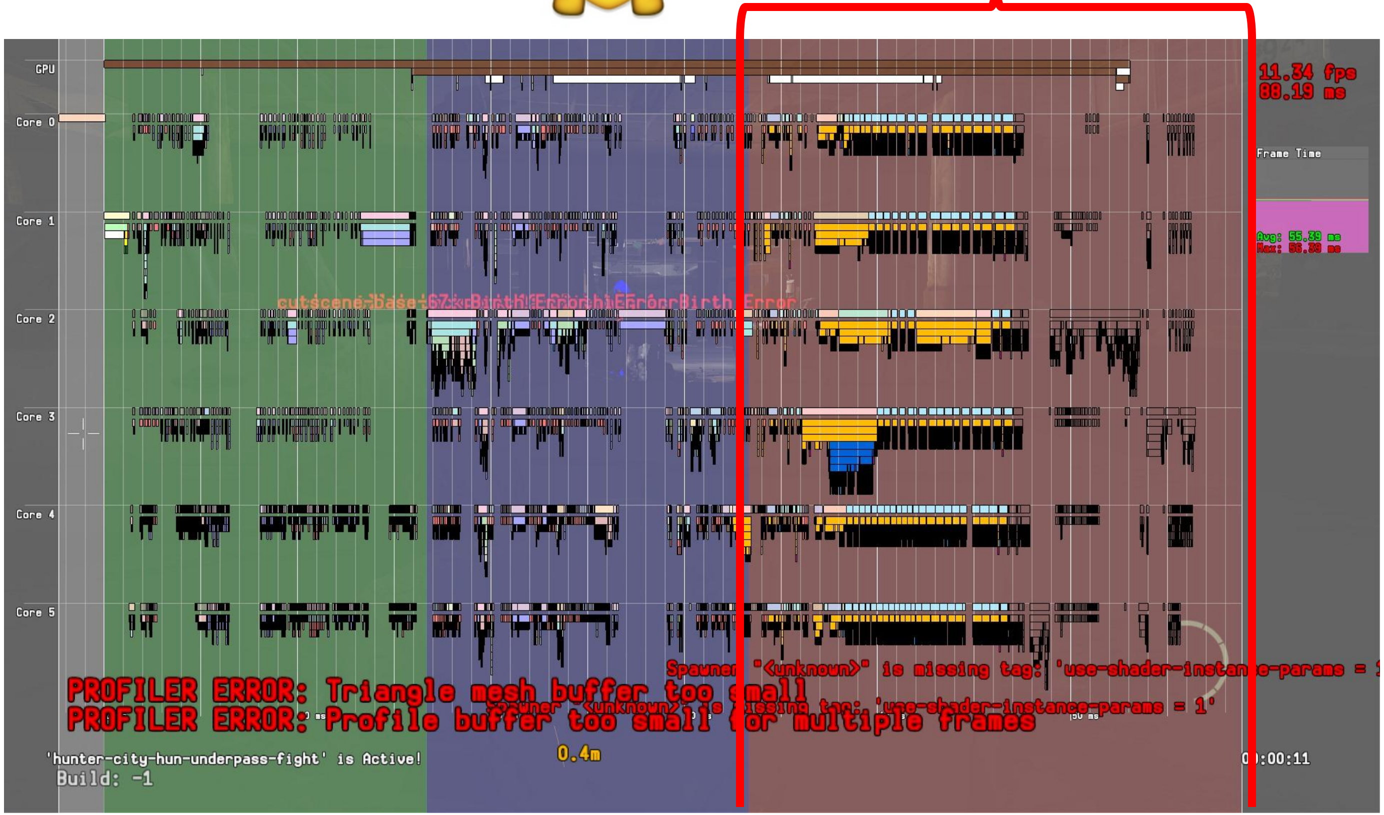
16.66 ms(60 fps)



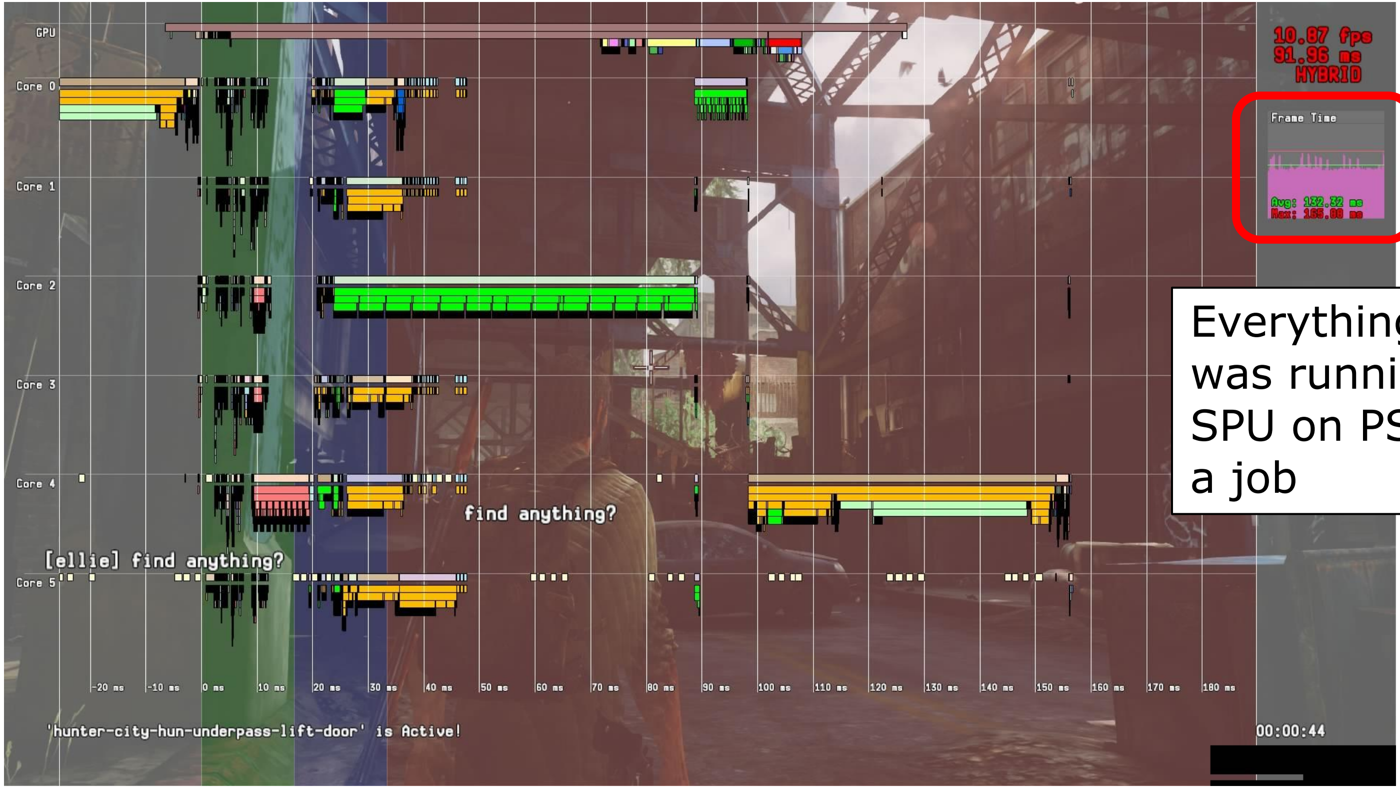
33.33 ms(30 fps)



Less than 30 fps



Initial Port



132 ms

Everything that was running on SPU on PS3 is now a job

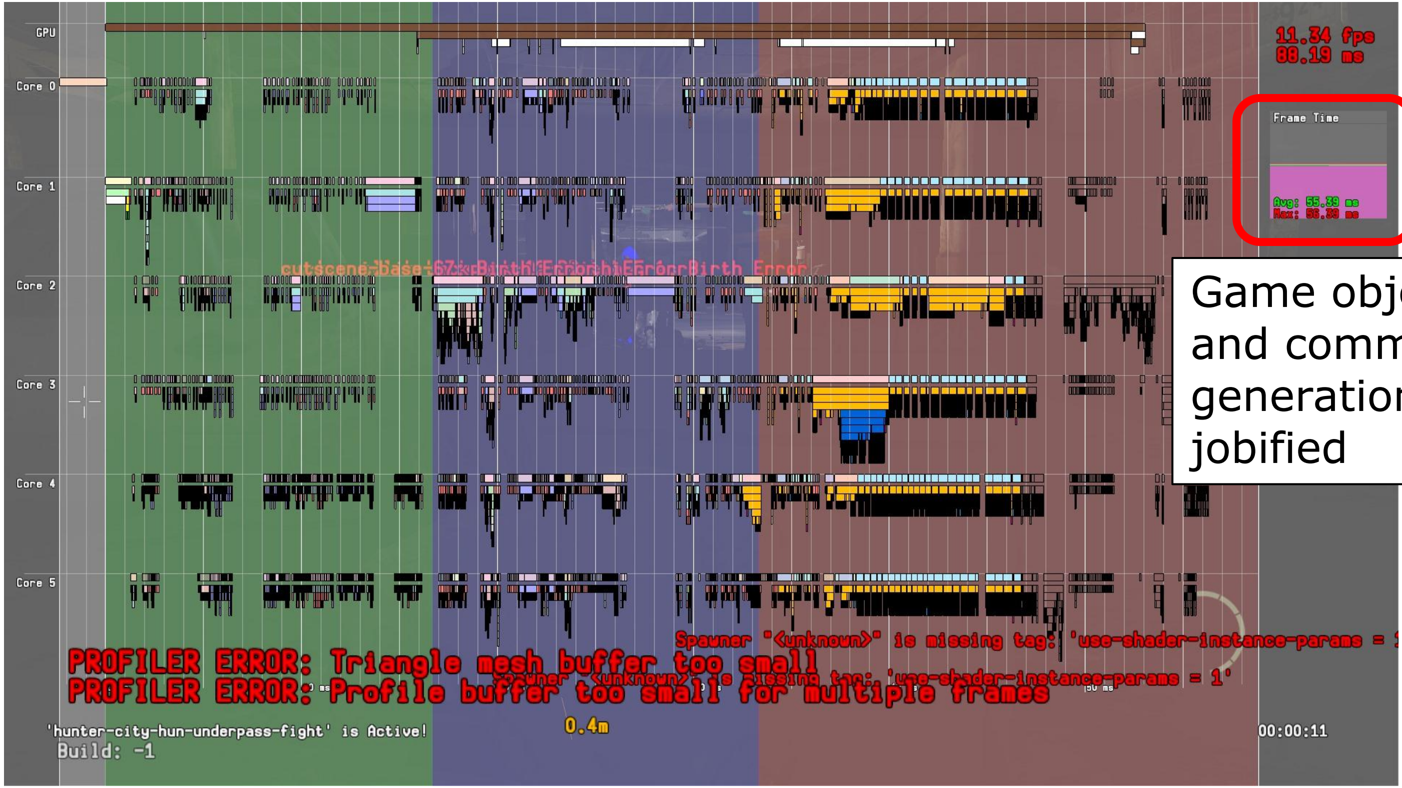
Initial Port

But we have a new rendering engine.

132 ms

Everything that was running on SPU on PS3 is now a job

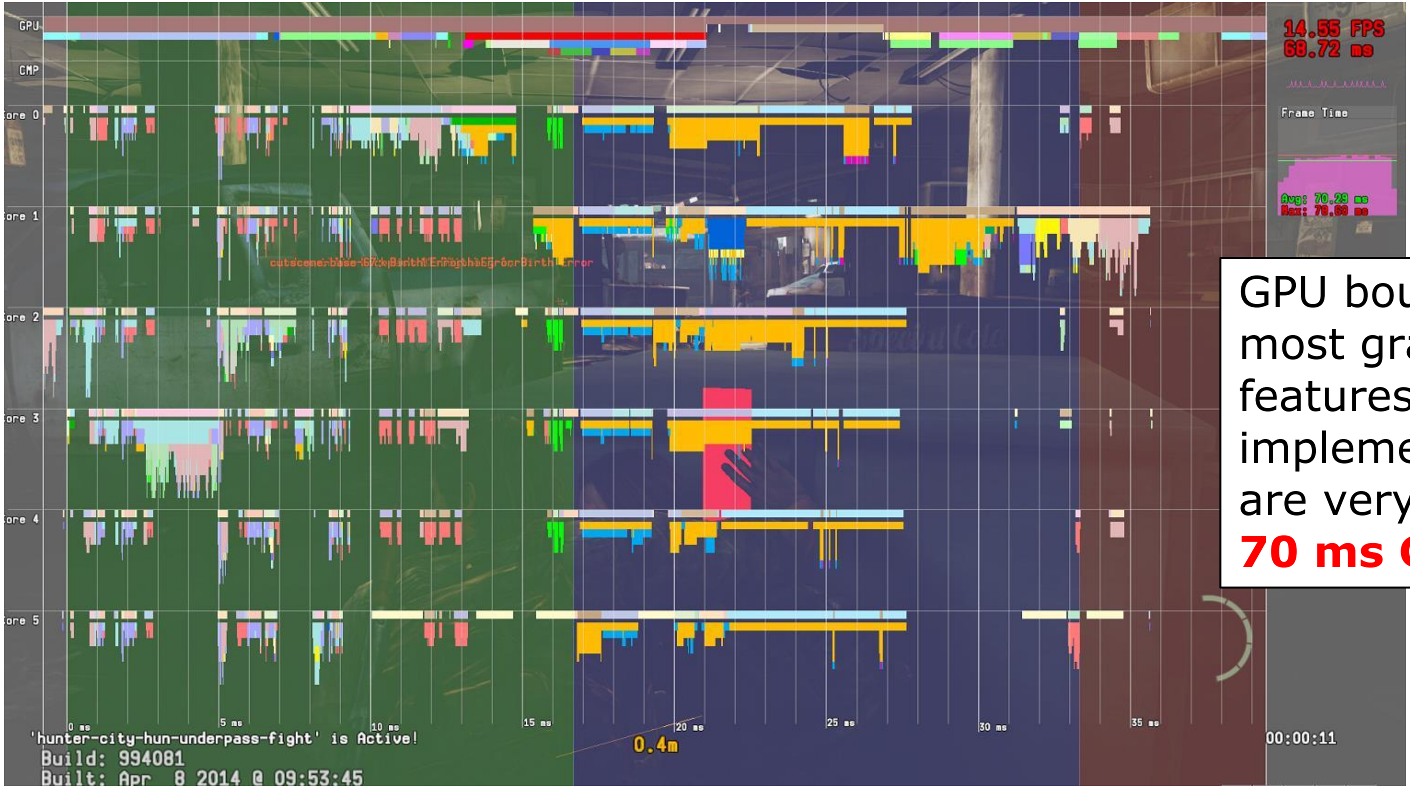
Jobify everything



55 ms

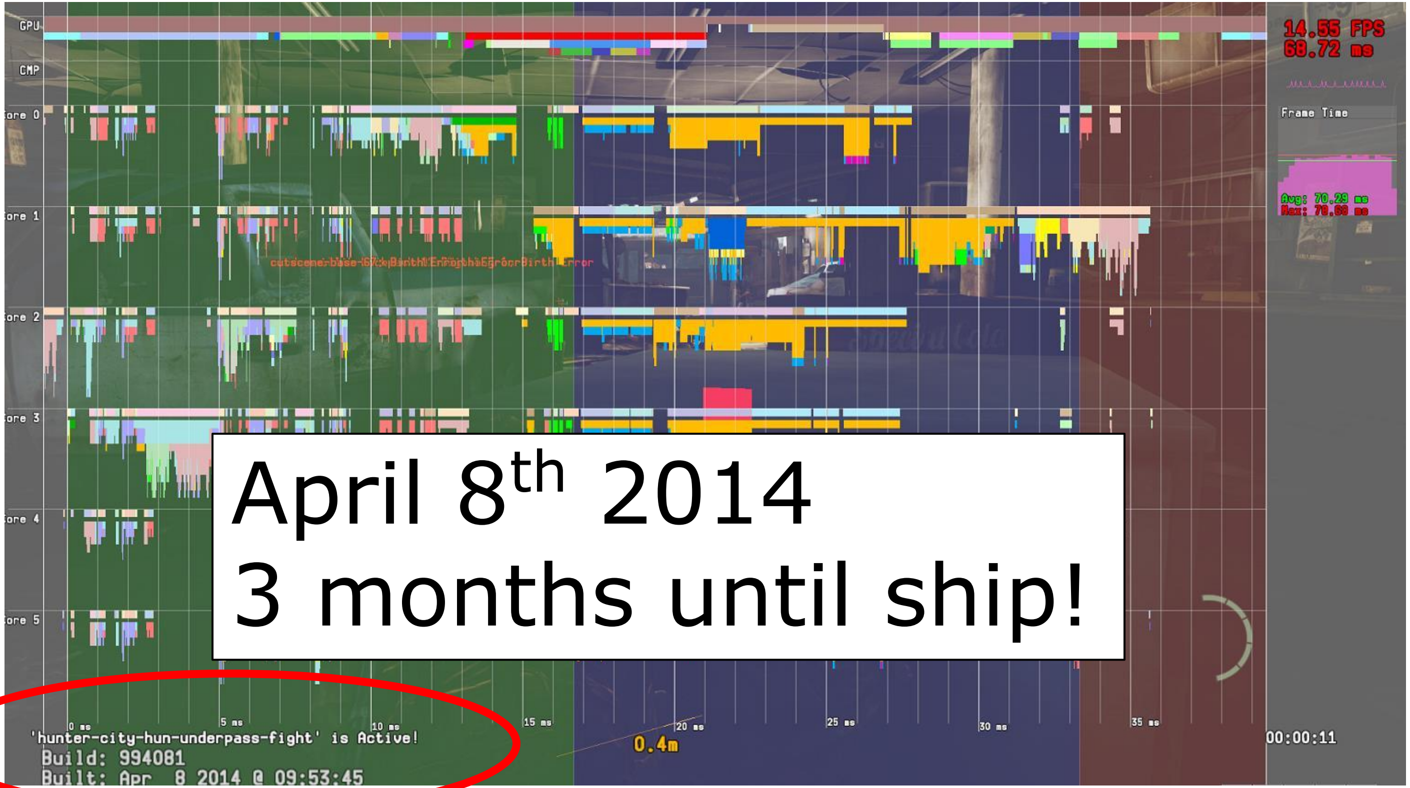
Game object updates and command buffer generation is now jobified

More jobs and fewer locks



36 ms

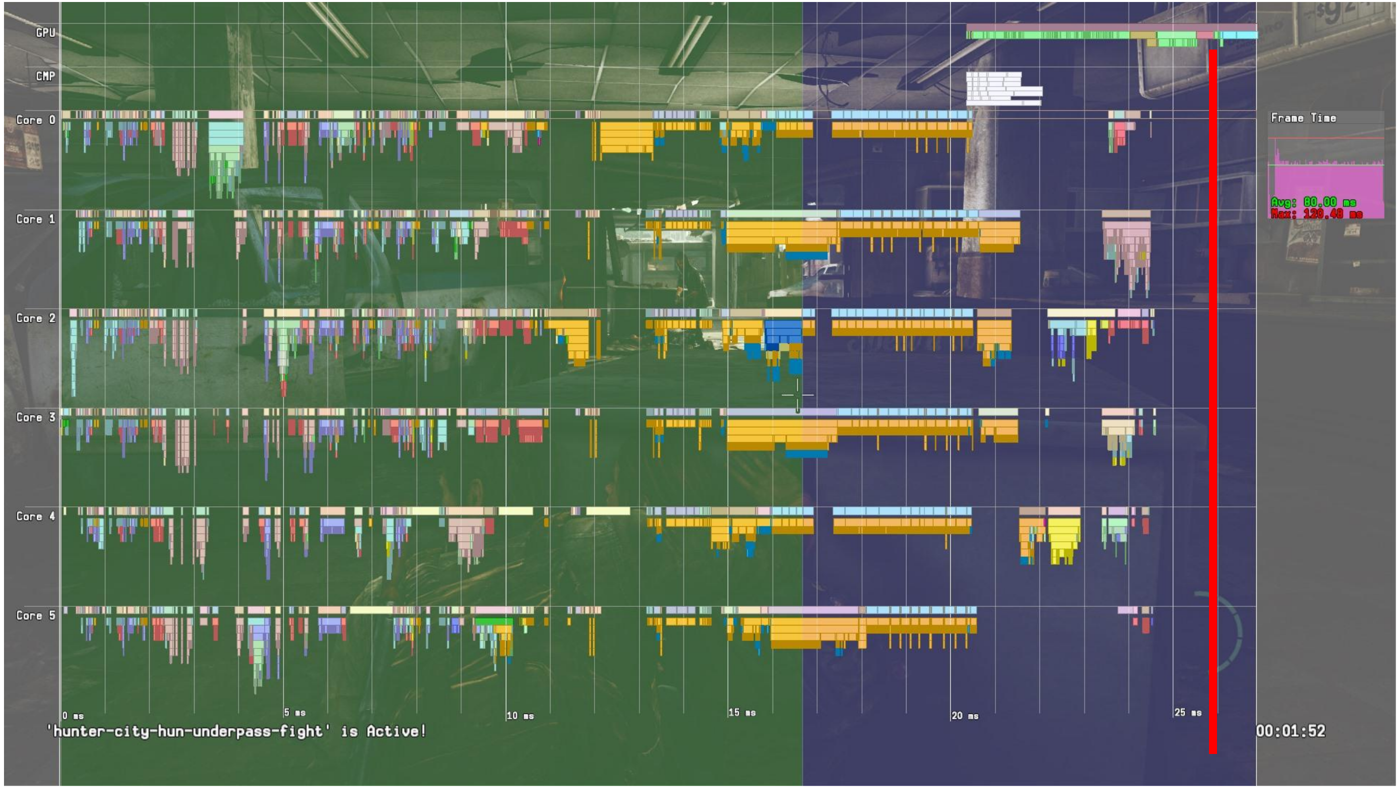
GPU bound due to most graphics features being implemented, but are very slow.
70 ms GPU frame



April 8^{th} 2014

3 months until ship!

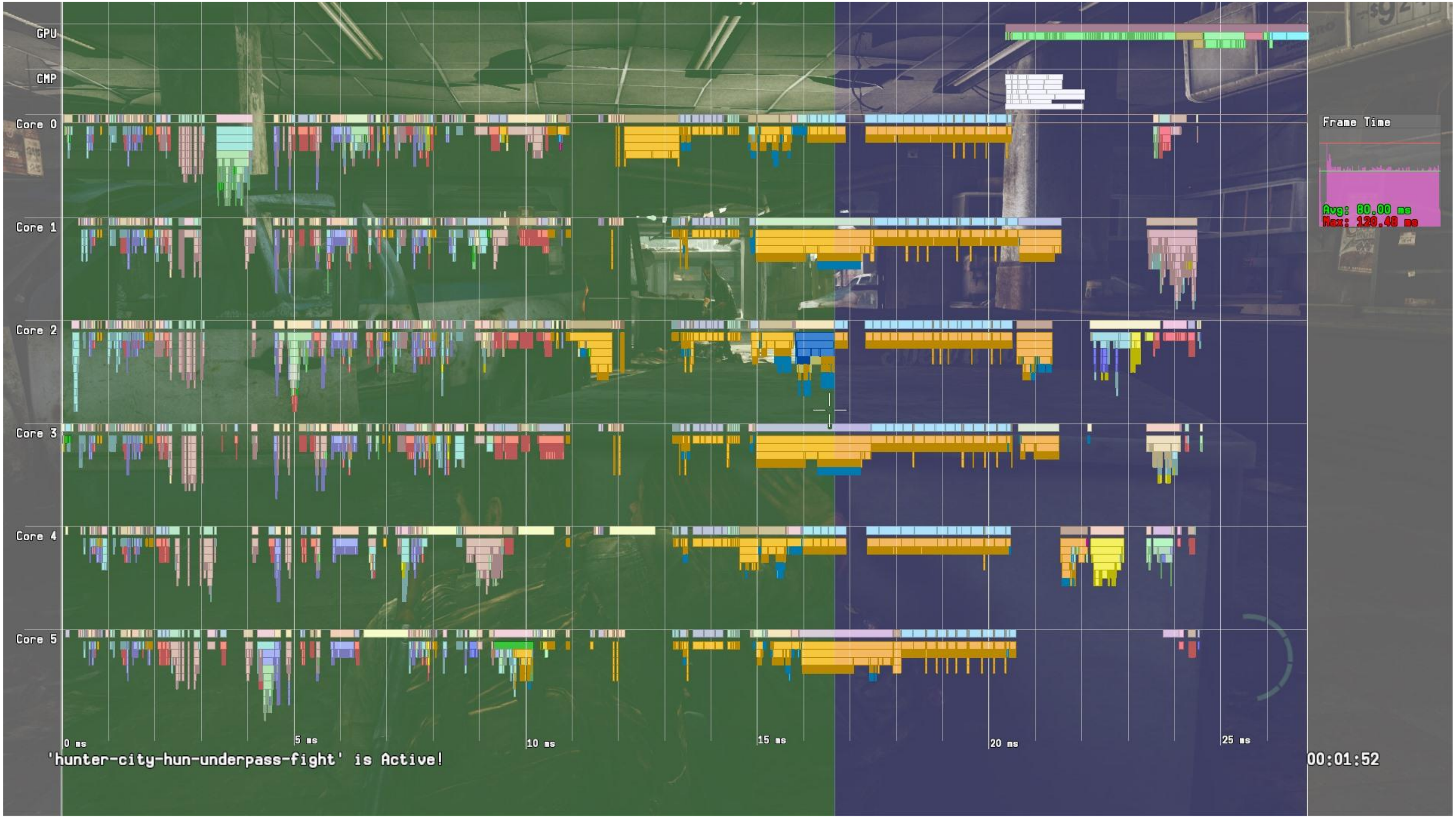
Diminishing returns…

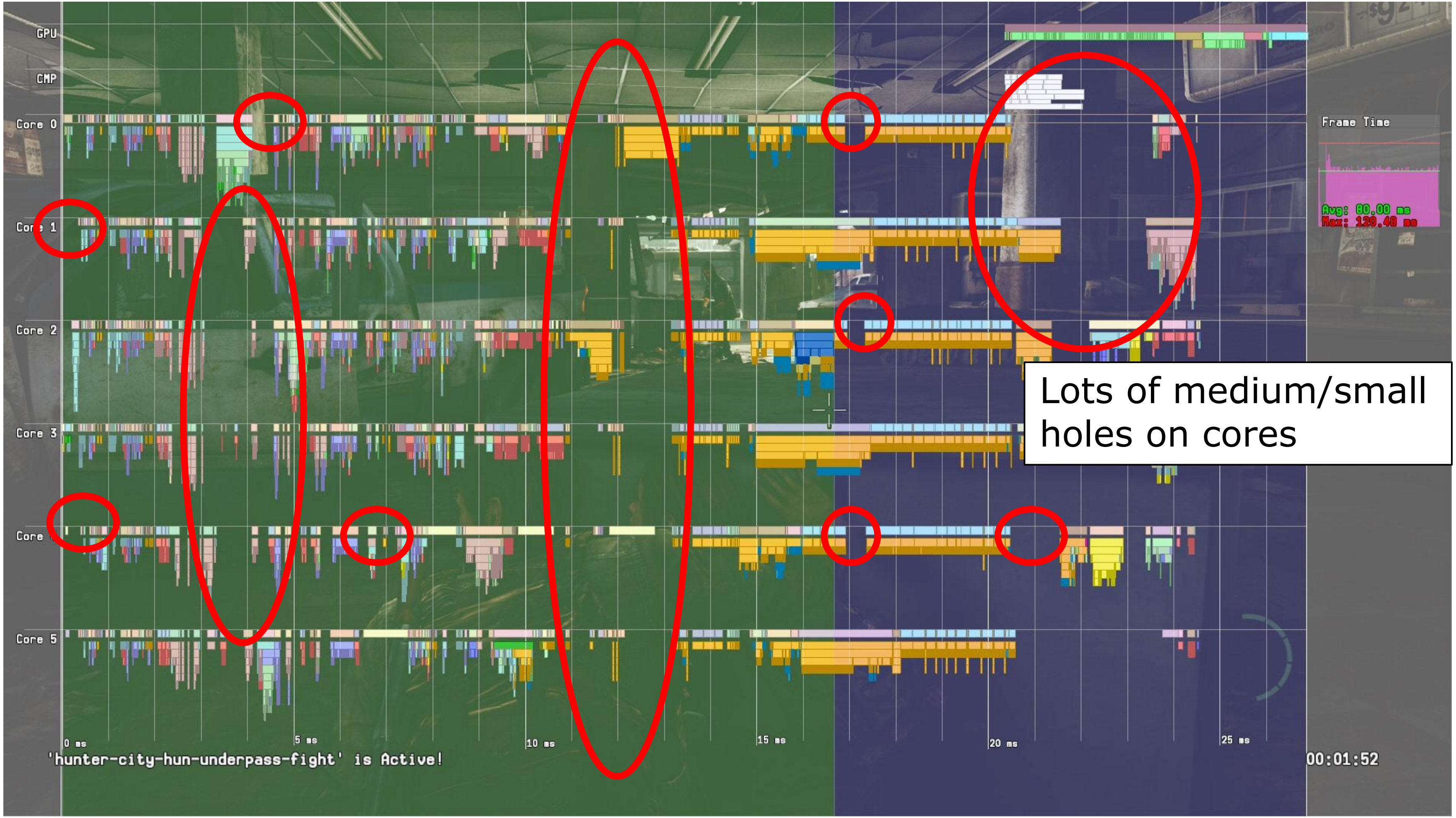


25 ms

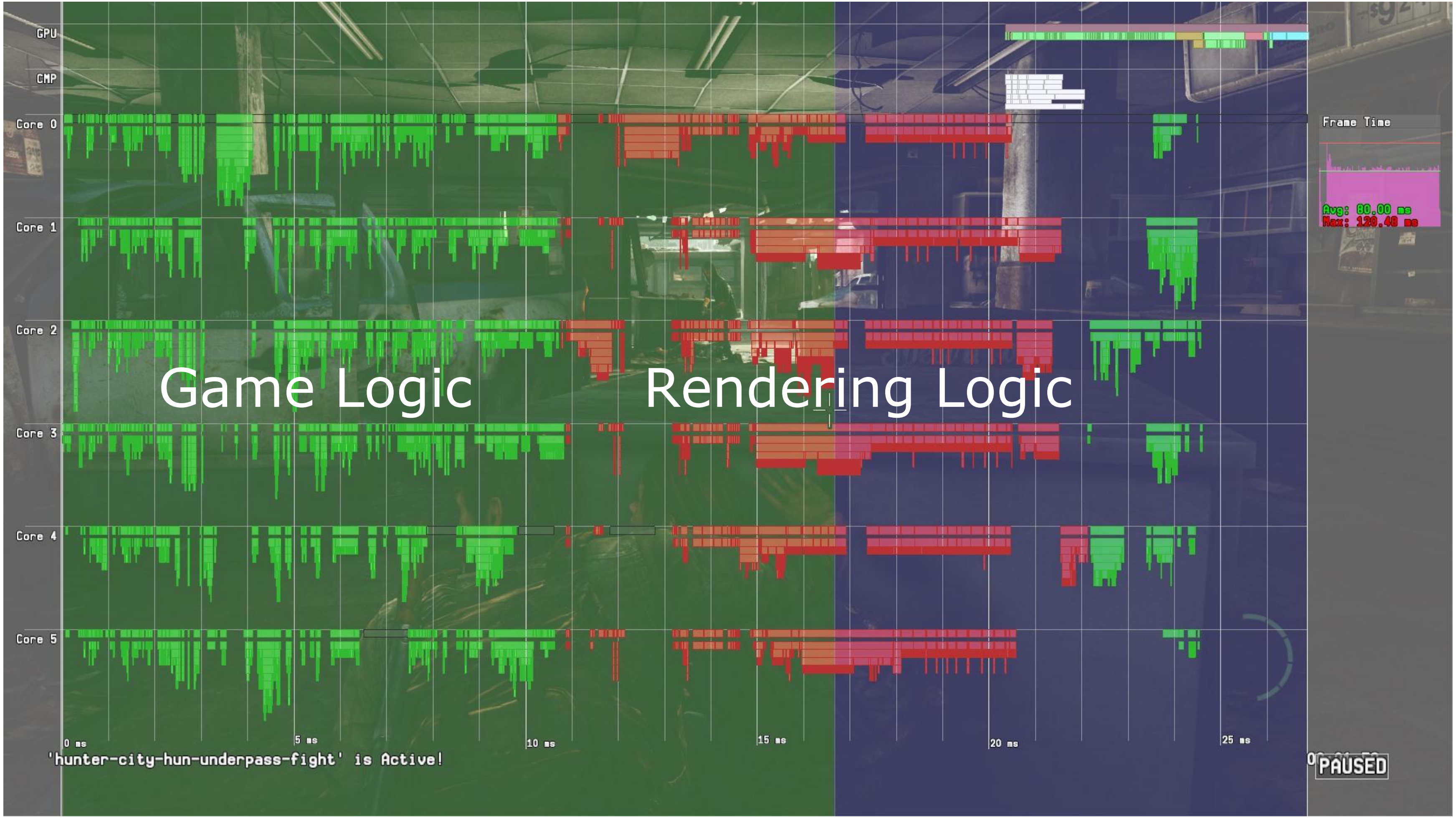
Harsh realization

- Most systems are now jobified.
- GPU optimizations are going really well
- CPU bound
  - Critical path on CPU takes ~25 milliseconds.
- Lots of idle time on CPU cores
- At this point we are a little over 2 months away from shipping


GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Avg: 80.00 ms
Max: 120.40 ms
0 ms
5 ms
10 ms
15 ms
20 ms
25 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:52



Lots of medium/small holes on cores


Game Logic
Rendering Logic
GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Avg: 80.00 ms
Max: 120.40 ms
0 ms
5 ms
10 ms
15 ms
20 ms
25 ms
'hunter-city-hun-underpass-fight' is Active!
PAUSED

Is it possible to reach 60 fps?

- We had ~100 ms worth of work to do on the CPU for a single frame
- If we can manage to fill all the cores 100% of the time, then we can make it run at 60 fps.
  - 16.66 ms of work * 6 cores -> ~100 ms of work
- Ok, theoretically possible… but how do we do that?

Let’s cut it in half!!!

How?

- Current design
  - Game logic runs first
  - Rendering logic and command buffer generation runs after
- New design
  - Run game and render logic at the same time
  - ***BUT** processing different frames!*

Engine pipeline – Feed forward

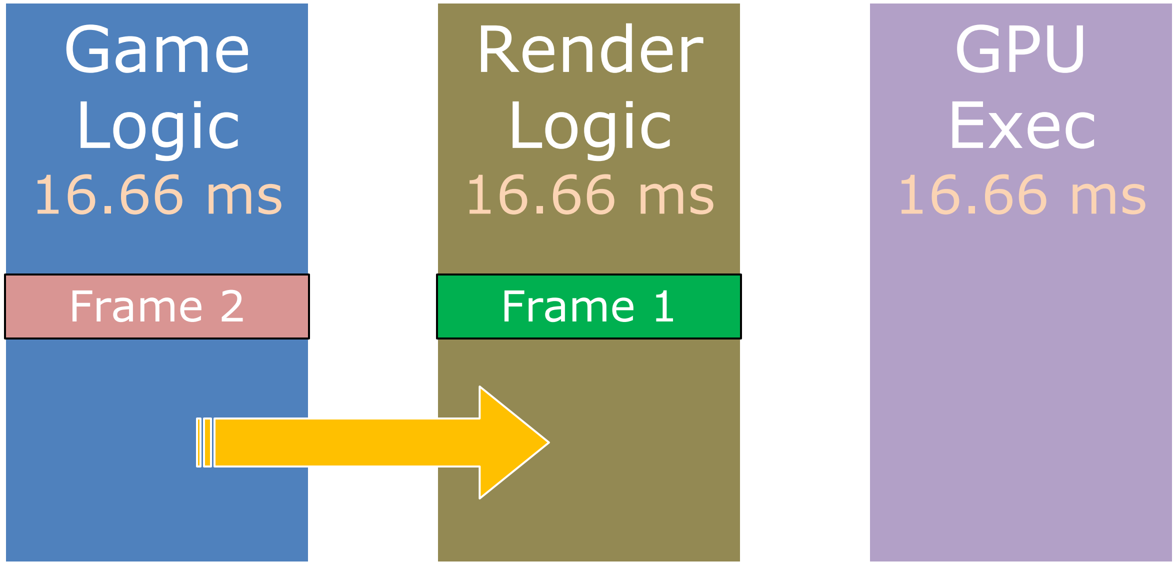
Game Logic
16.66 ms

Frame 1

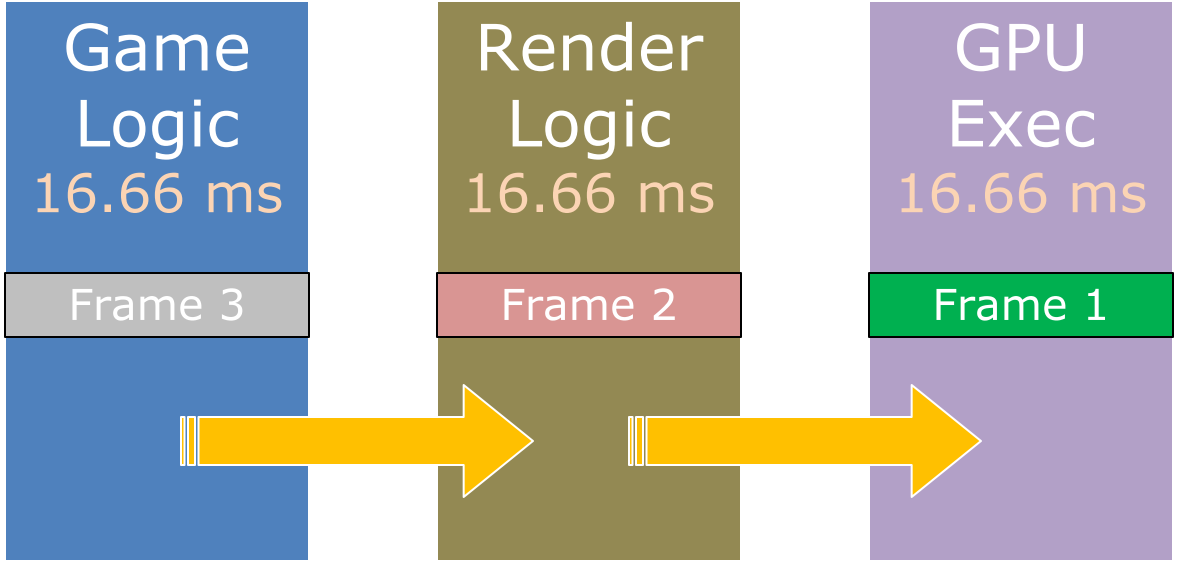
Render Logic
16.66 ms

GPU Exec
16.66 ms

Engine pipeline – Feed forward

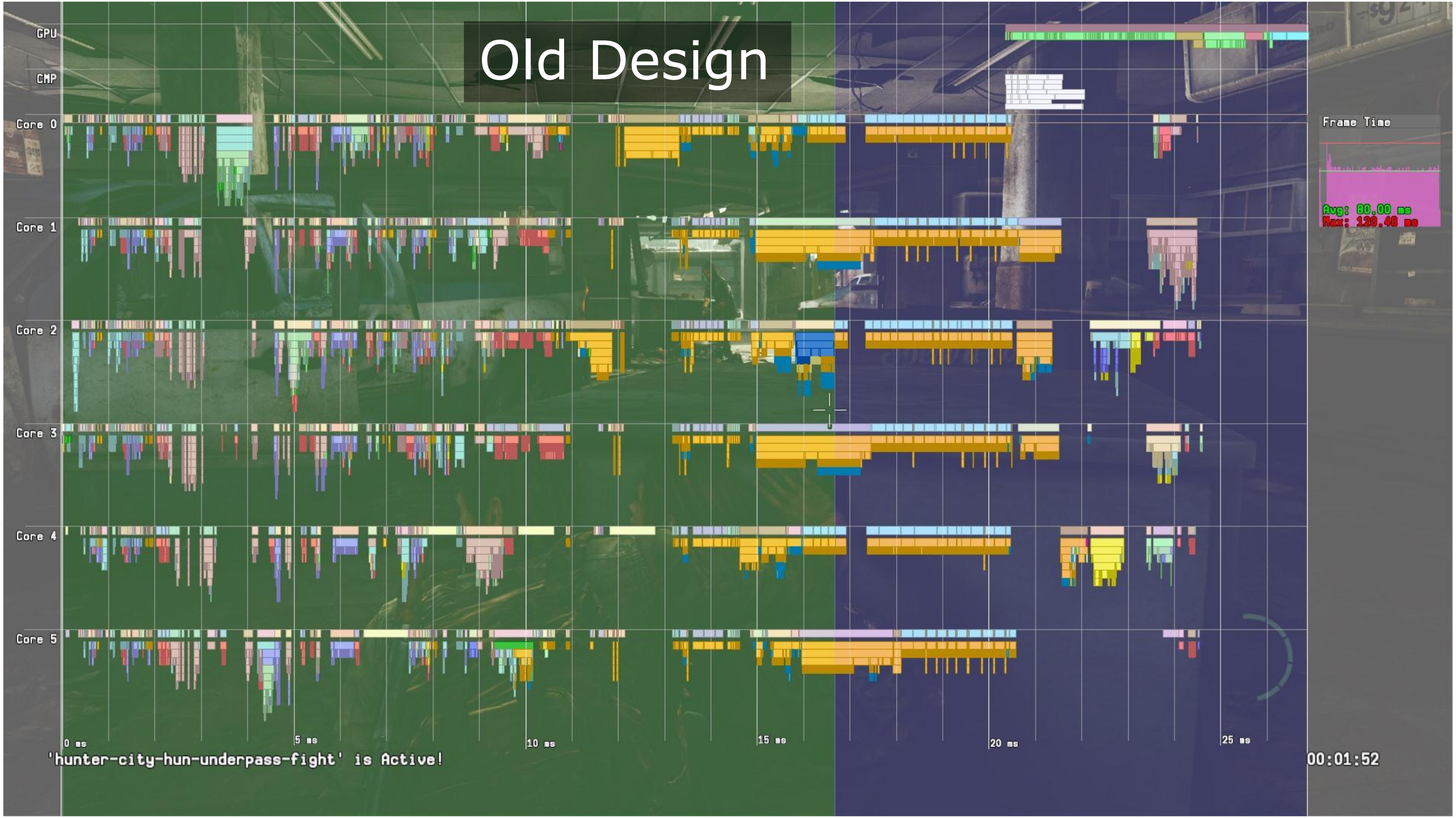


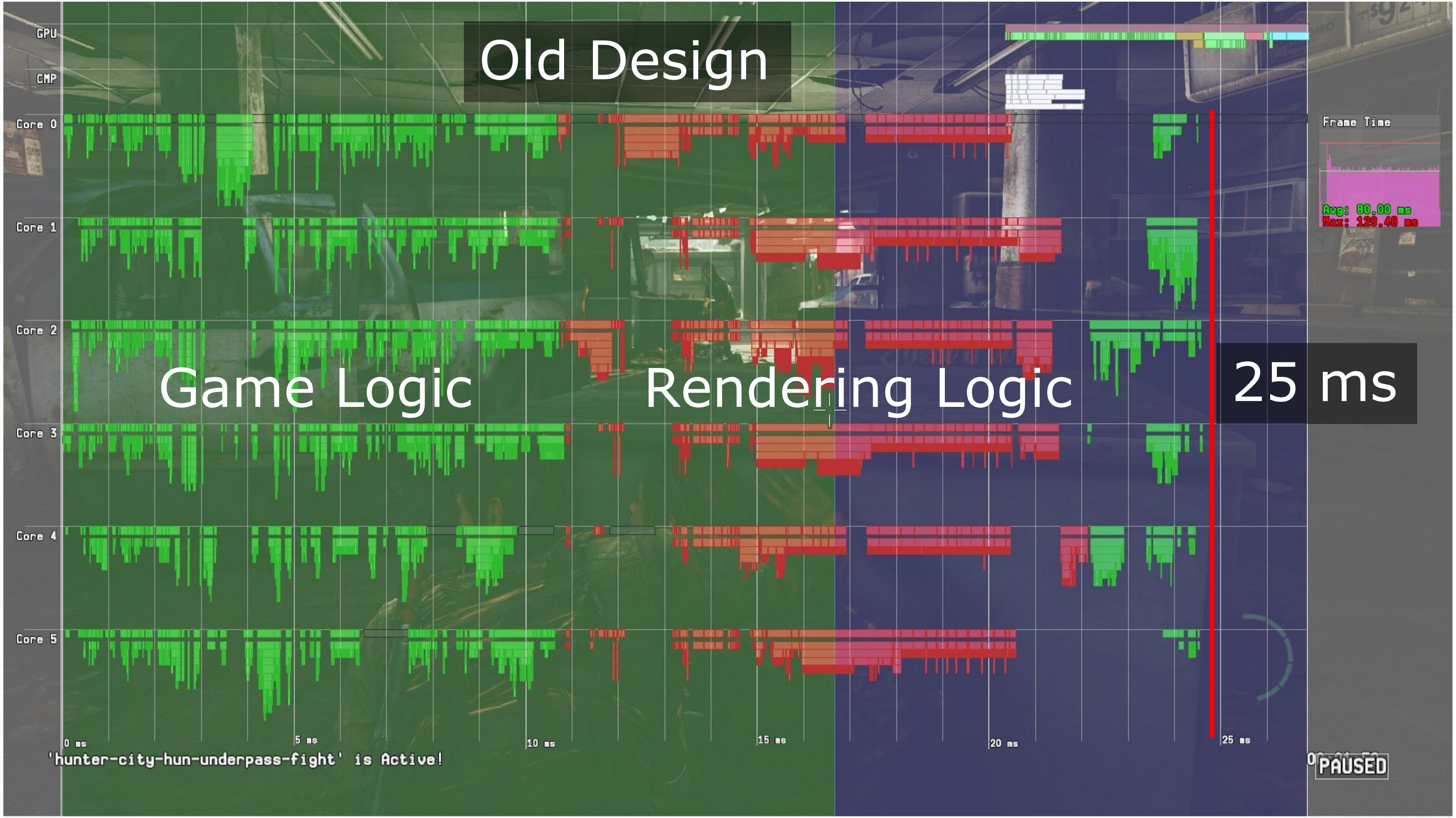
Engine pipeline – Feed forward



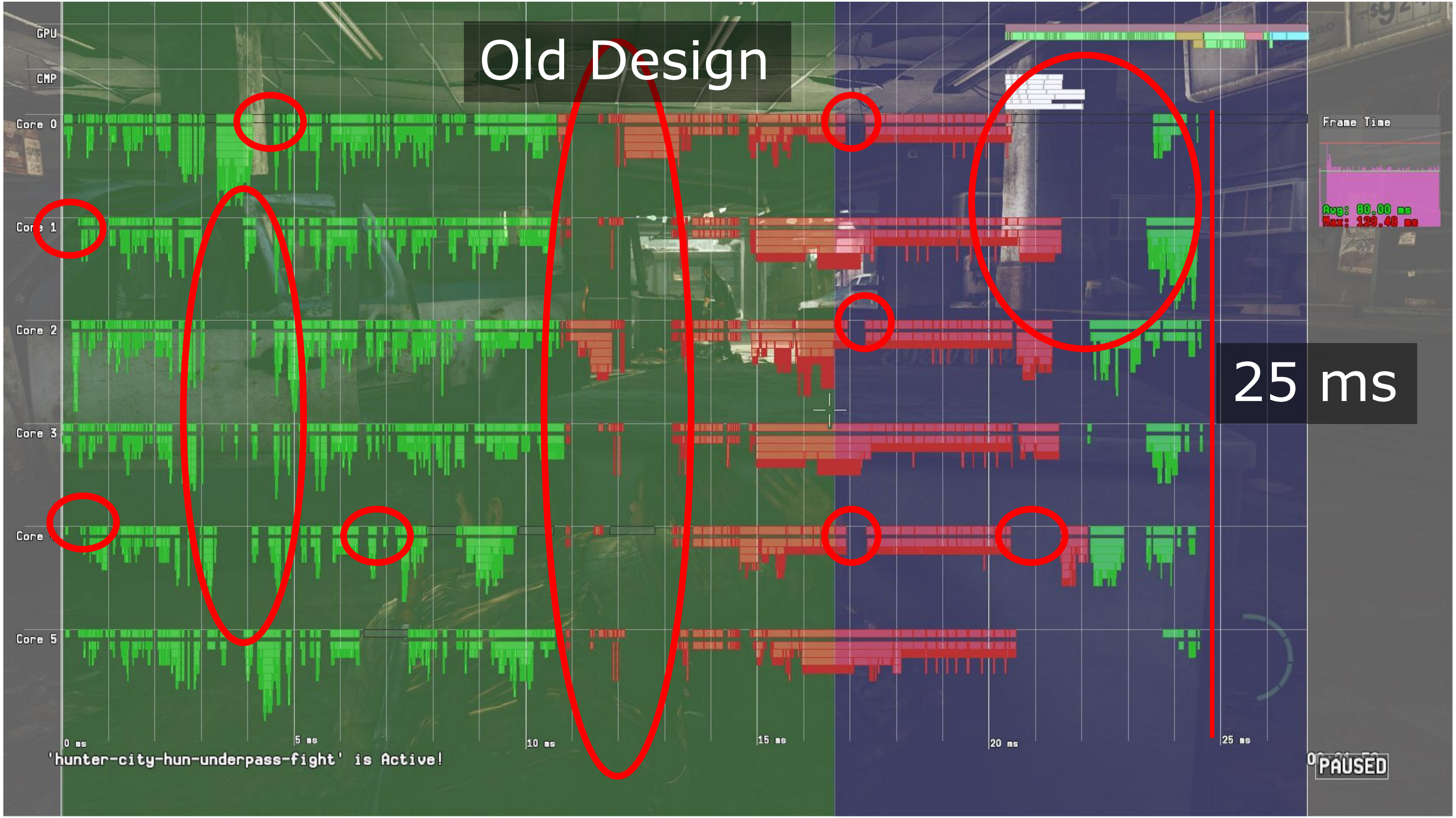
Frame centric design

- Each stage is running completely independent
  - No synchronization required
- A stage can process the next frame right away
- Complexity in engine design is simplified
  - Very few locks needed due to parallelism
  - Locks are only used to synchronize **within** heavily jobified stage updates


Old Design
GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Avg: 80.00 ms
Max: 120.40 ms
0 ms
5 ms
10 ms
15 ms
20 ms
25 ms
'hunter-city-hun-underpass-fight' is Active!
00:01:52


Old Design
GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Frame Time
Avg: 80.00 ms
Max: 120.40 ms
Game Logic
Rendering Logic
25 ms
0 ms
5 ms
10 ms
15 ms
20 ms
25 ms
'hunter-city-hun-underpass-fight' is Active!
PAUSED

Old Design


GPU
CMP
Core 0
Core 1
Core 2
Core 3
Core
Core 5
Frame Time
Avg: 80.00 ms
Max: 120.48 ms
25 ms
0 ms
5 ms
10 ms
15 ms
20 ms
25 ms
'hunter-city-hun-underpass-fight' is Active!
PAUSED

New Design

15.5 ms!!!
Ship it!

Memory allocation on PS3

- Linear allocators
  - Contiguous block of memory + offset in block
  - Allocation is as simple as updating the offset
  - 'Free' all allocations by setting the offset to 0
- Single/Double/Triple frame
  - At the beginning of the frame we rotate to a new block and set the offset to 0
- Works great when you process ONE frame at a time

Multi-frame difficulties

- “When should I free my memory?”
  - Memory is consumed in a later asynchronous stage
- “I need a new buffer every frame.”
  - “How many buffers do I need to rotate between to avoid memory stomping?”
- “Which delta time should I use?”
- Every programmer was solving the same problem
  - … often incorrectly
- Hard to understand everything that is going on

Frames, frames, frames…

- What is a ‘frame’ anyway?
  - Game logic frame, GPU frame, display frame?
- Memory lifetime
  - When do I safely free memory?
  - Can we double/triple buffer memory?
    - What does the even mean now?
- Need a new way of thinking about frames

Our definition of a frame

- “A piece of data that is processed and ultimately displayed on screen”
- The main point here is ‘piece of data’.
- It is NOT a length of time.
- A frame is defined by the stages the data goes through to become a displayed image

Introducing: FrameParams

- Data for each displayed frame
  - One instance for each new frame to eventually be displayed
  - Sent through all stages of the engine
- Contains per-frame state:
  - Frame number
  - Delta time
  - Skinning matrices
- Entry point for each stage to access required data

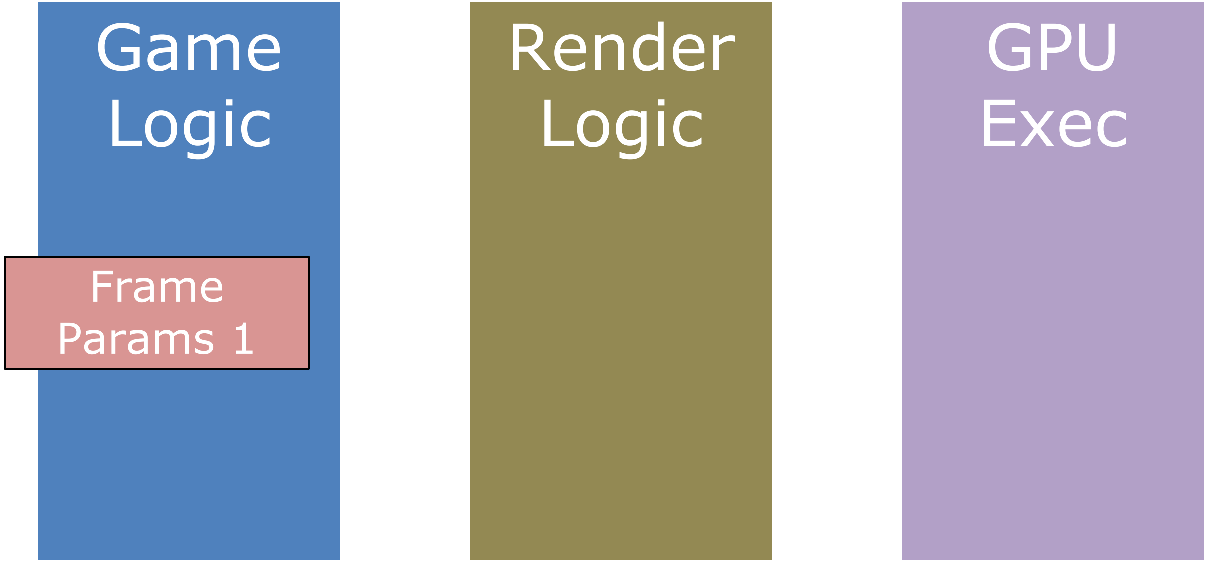
FrameParams...

- Uncontended resource
  - No locks needed as each stage works on a unique instance
- State variables are copied into this structure every frame
  - Delta time
  - Camera position
  - Skinning matrices
  - List of meshes to render
- Stores start/end timestamps for each stage
  - Game, Render, GPU and Flip

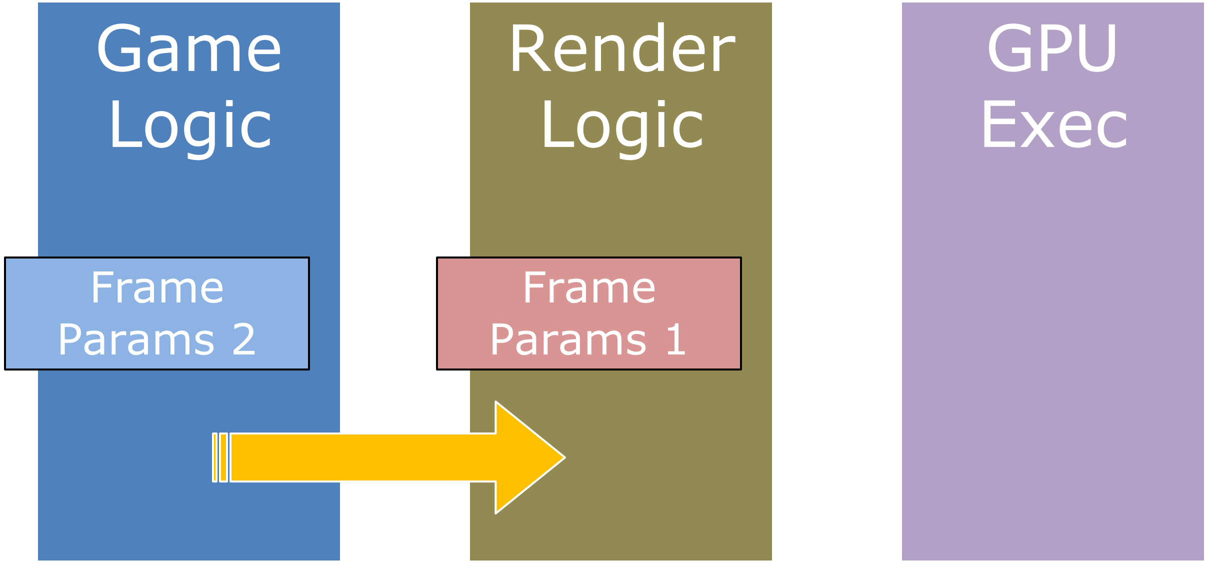
FrameParams…

- Easy to test if a frame has completed a particular stage
  - HasFrameCompleted(frameNumber)
- Memory lifetime is now easily tracked
  - If you generate data to be consumed by the GPU in frame X, then you wait until HasFrameCompleted(X) is true.
- We have 16 FrameParams that we rotate between
  - You can only track the state of the last 15 frames

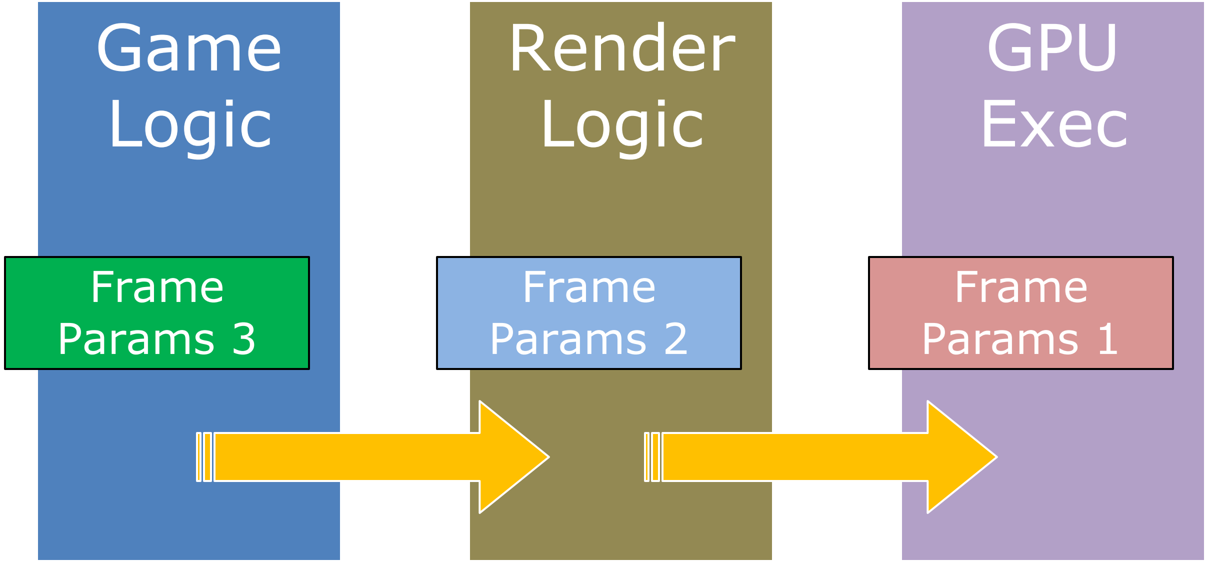
Engine pipeline – FrameParams



Engine pipeline – FrameParams



Engine pipeline – FrameParams



Memory lifetimes

- Single Game Logic Stage (scratch memory)
- Double Game Logic Stage (low-priority ray casts)
- Game To Render Logic Stage (Object instance arrays)
- Game To GPU Stage (Skinning matrices)
- Render To GPU Stage (Command Buffers)
- … for both Onion(CPU) and Garlic(GPU) memory!

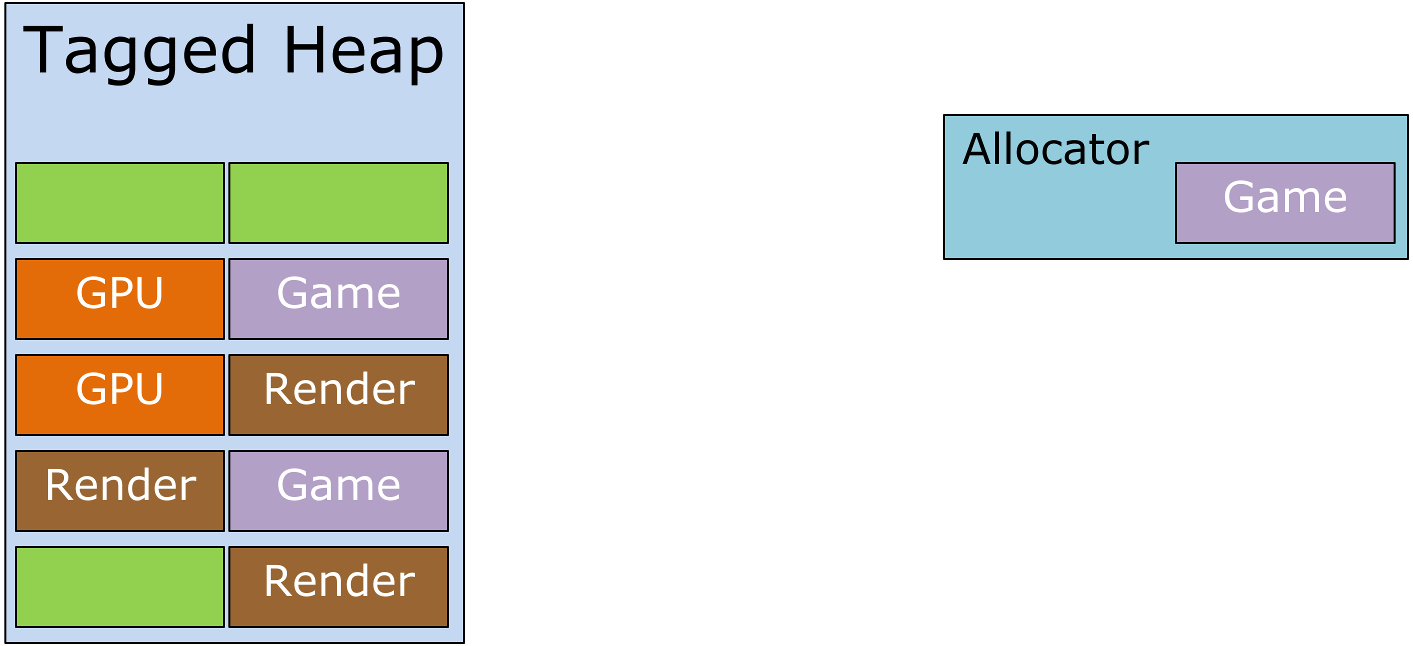
Running out of memory

- Many different linear allocators
- Many different life times
- All sized for worst case
- We never hit worst case for all allocators at the same time
- 100-200 MiB of wasted memory

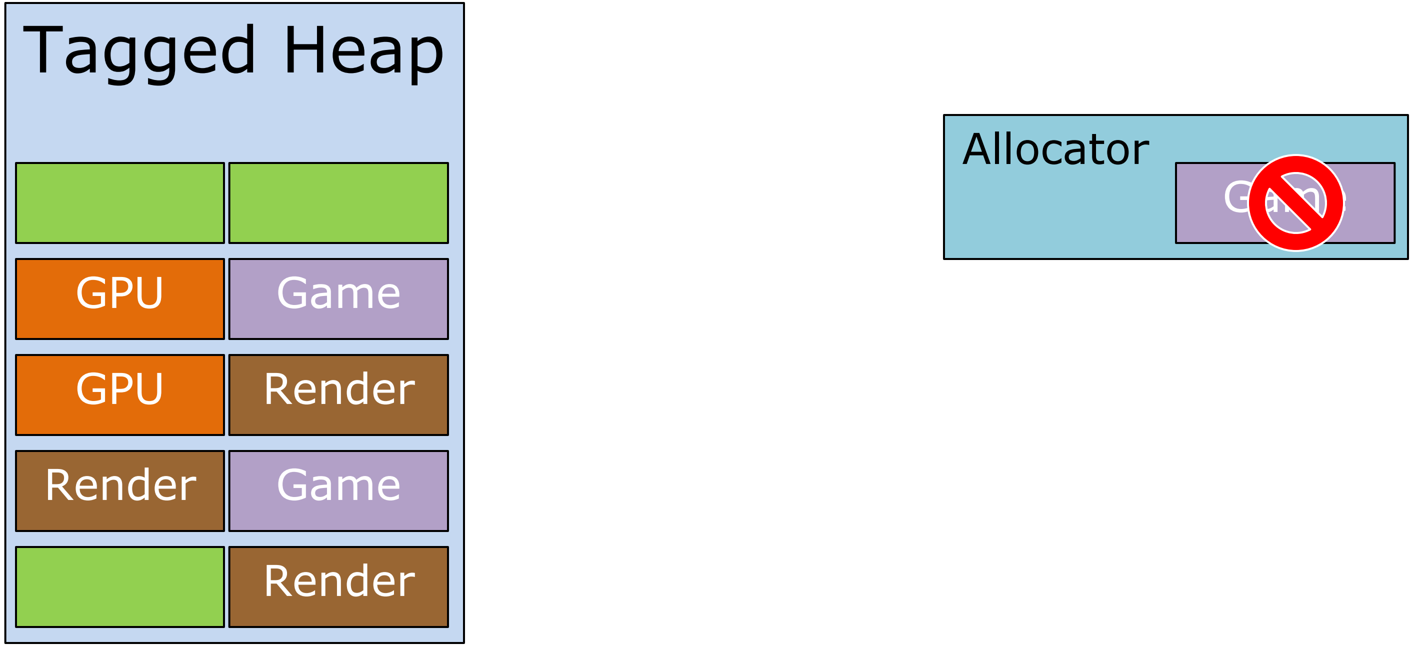
Introducing: Tagged Heap

- Block based allocator
  - Block size of 2 MiB
  - 2 MiB is a 'Large Page' on PS4 –> 1 TLB entry
- Each block is owned by a tag
  - uint64_t
- No 'Free(ptr)' interface
- Free all blocks associated with a specific tag

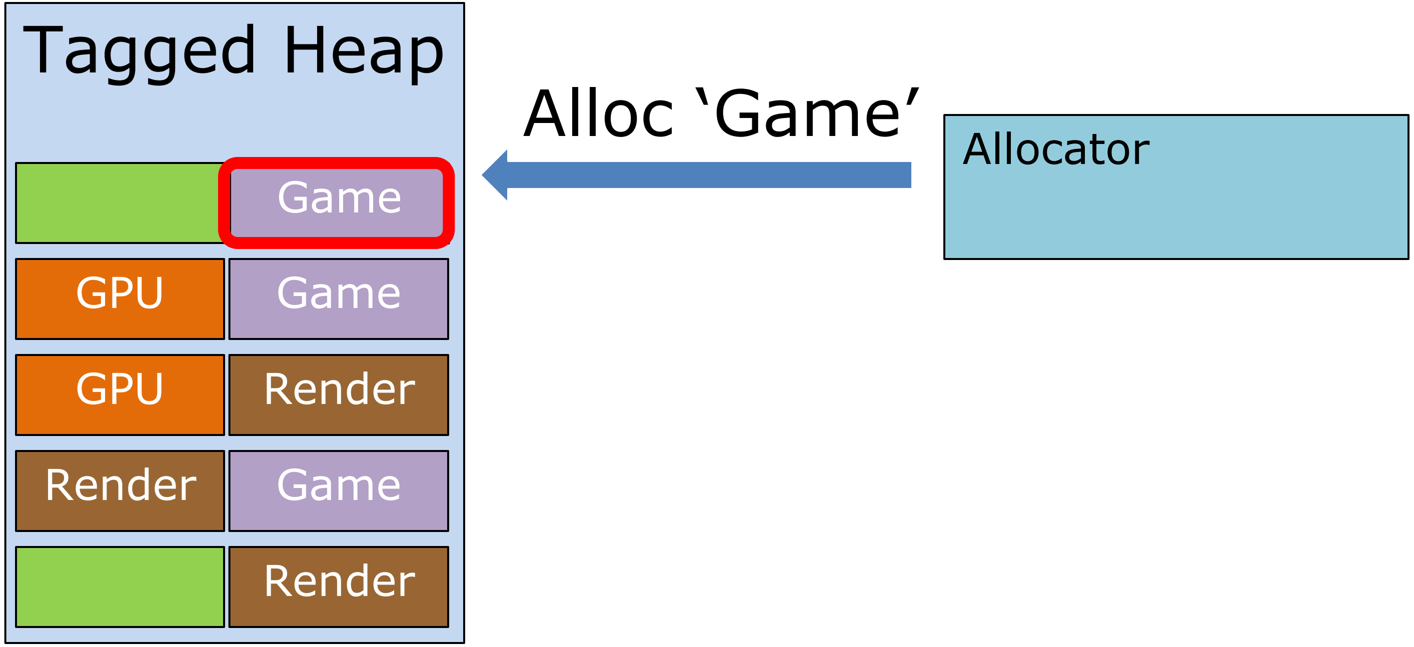
Allocate from Tagged Heap



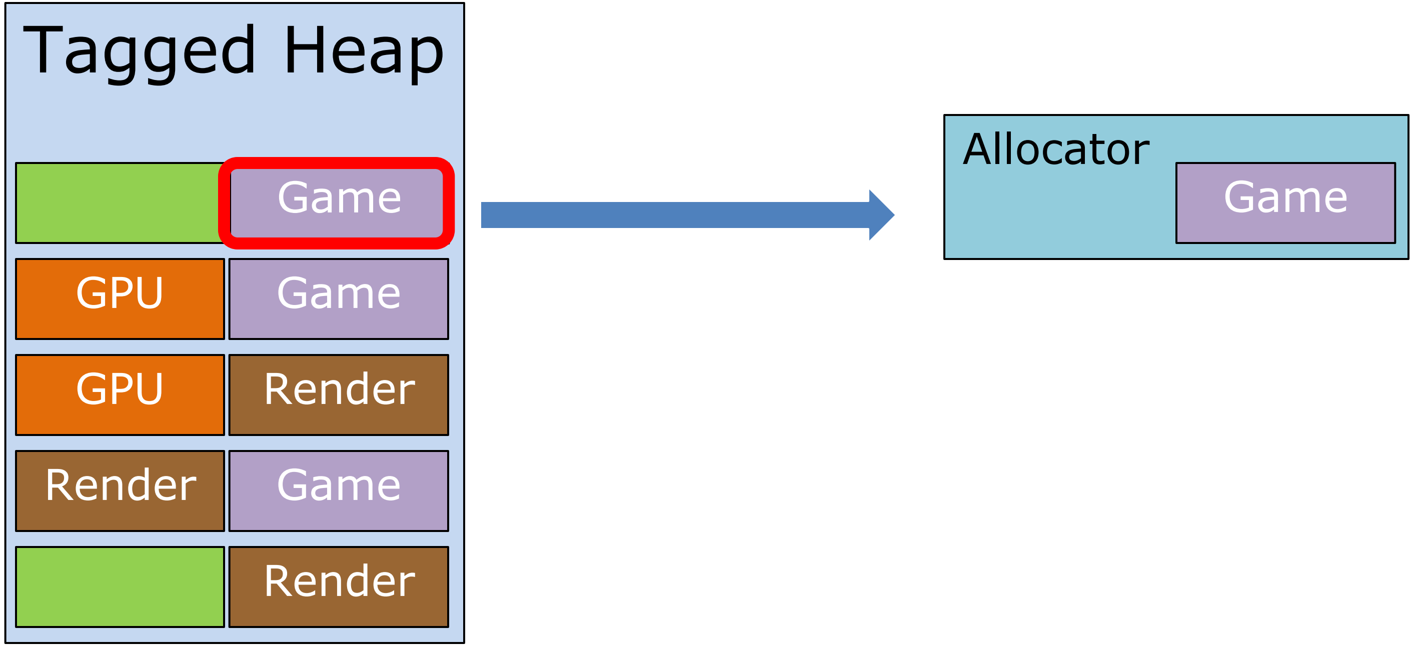
Allocate from Tagged Heap



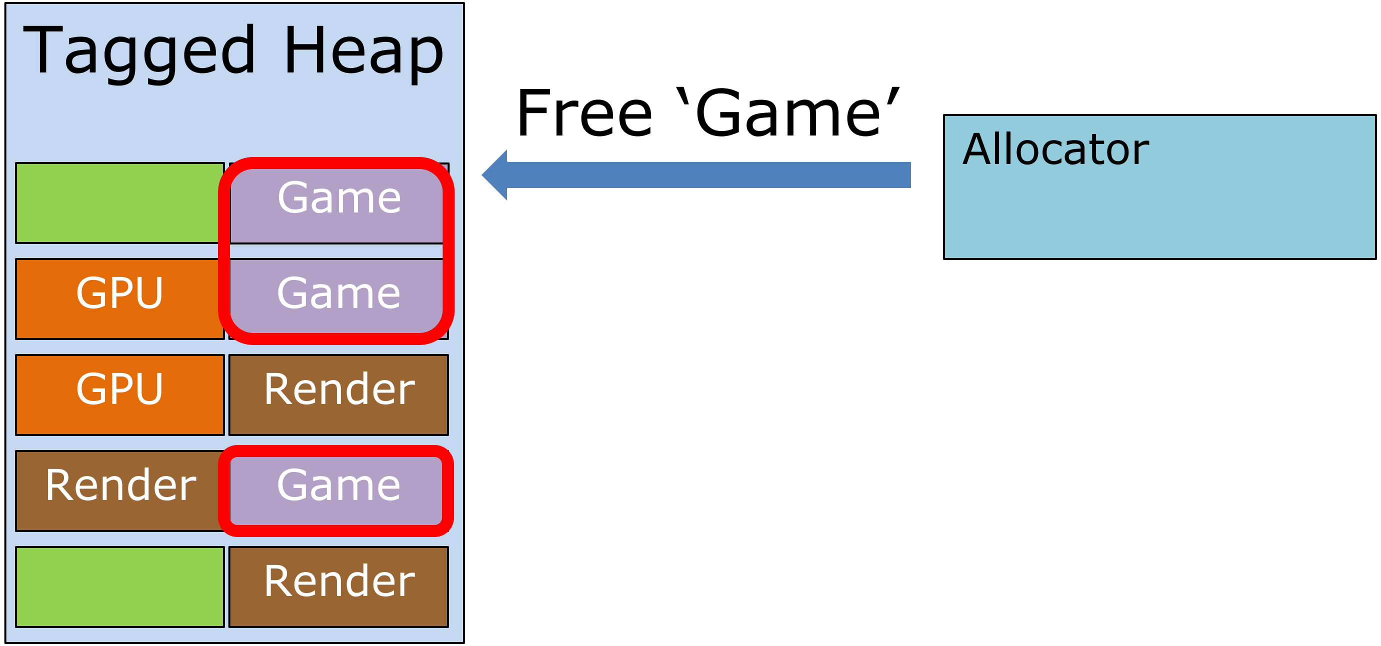
Allocate from Tagged Heap



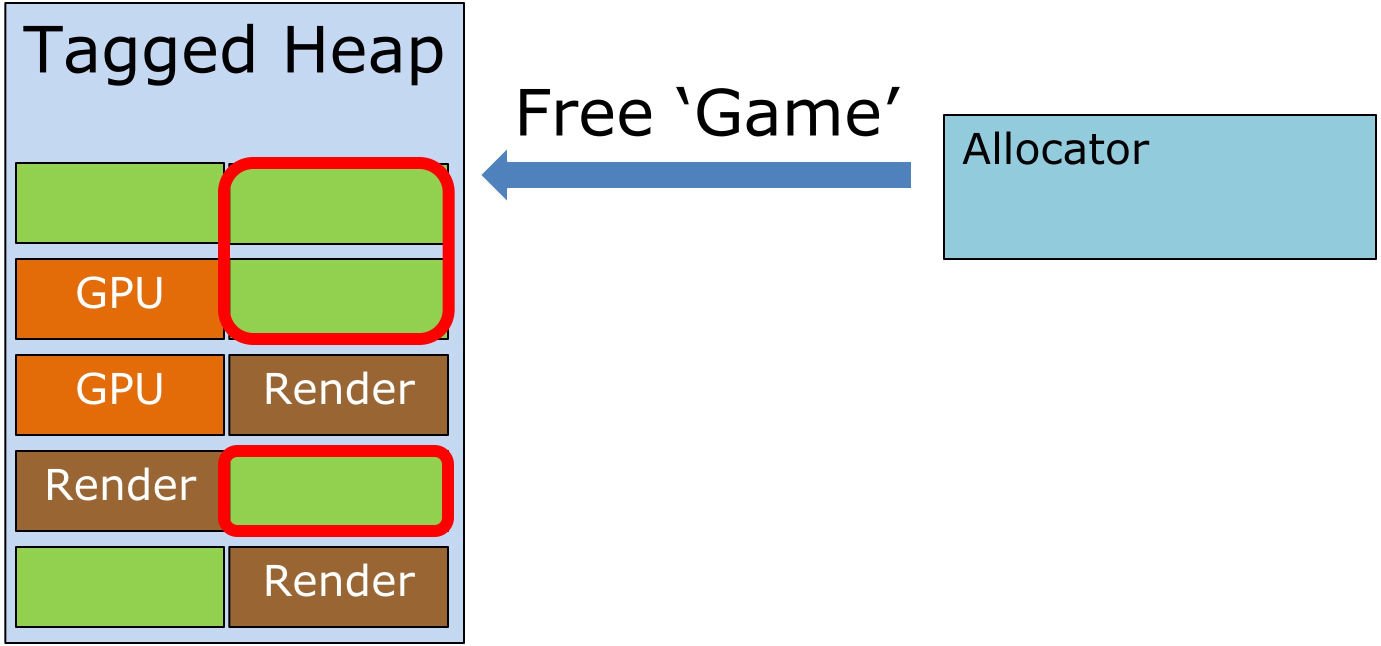
Allocate from Tagged Heap



Bulk free


Tagged Heap
Game
GPU
Game
GPU
Render
Render
Game
Render
Free 'Game'
Allocator

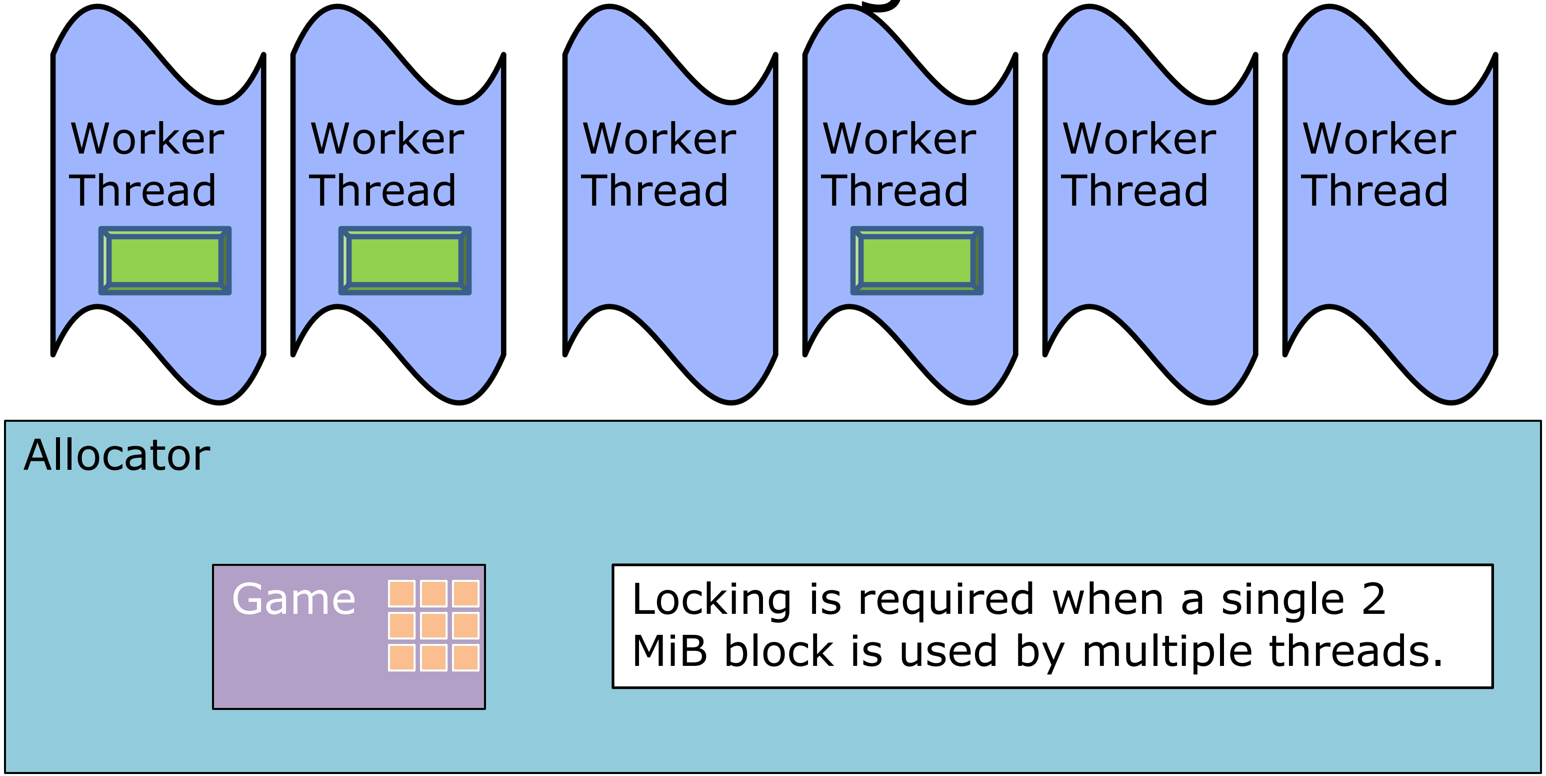
Bulk free


Tagged Heap
GPU
GPU
Render
Render
Render
Free ‘Game’
Allocator

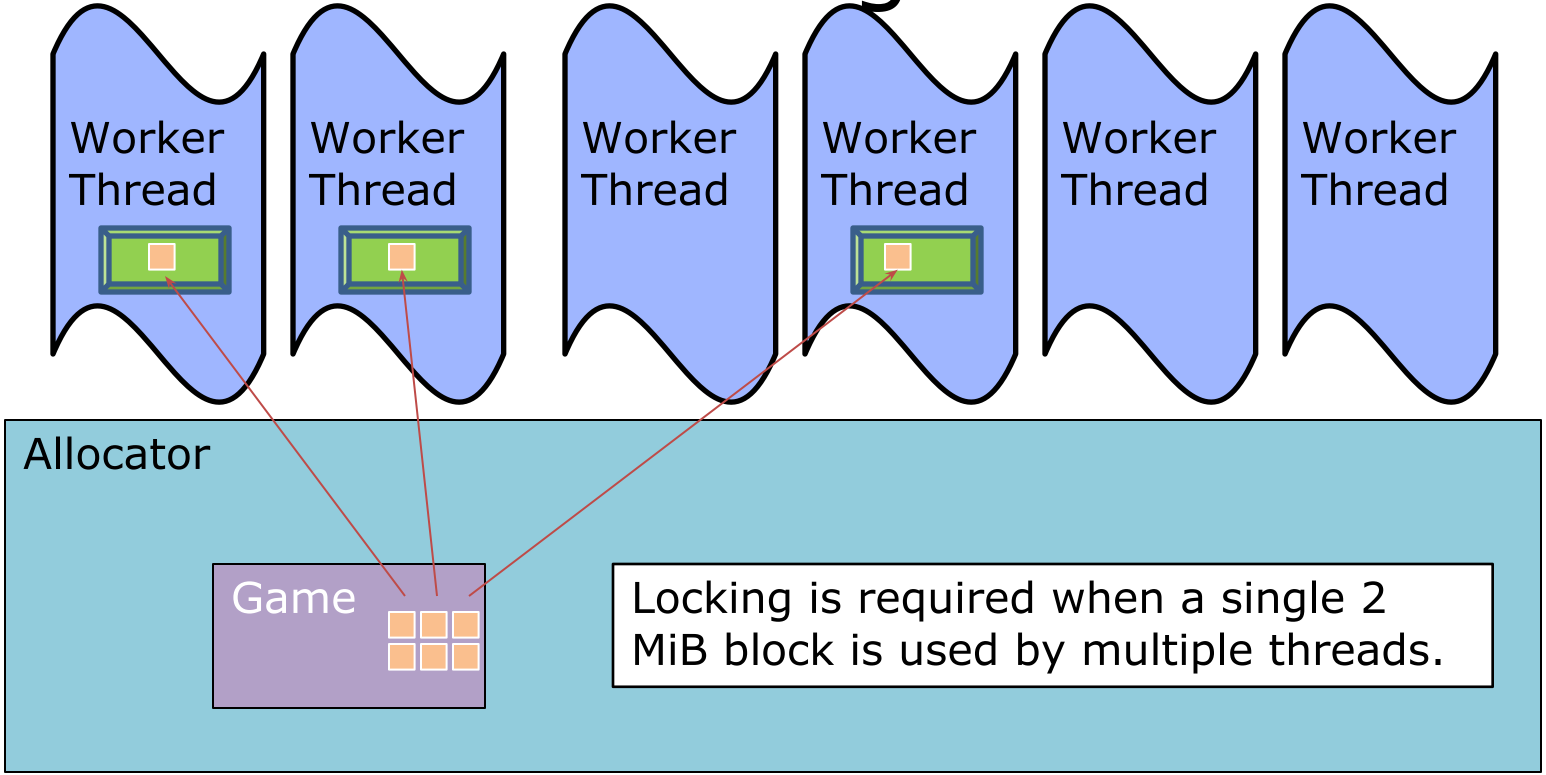
All allocators use the tagged heap

- Allocate 2 MiB block from shared tagged heap and store locally in the allocator
- 99% of all of our allocations are less than 2MiB
  - Larger than 2 MiB allocations get consecutive 2 MiB blocks allocated from the tagged heap
- Make allocations from this local block until empty
- Sharing a common pool of blocks like this allows for dynamic sizing of allocators

Allocator with single block



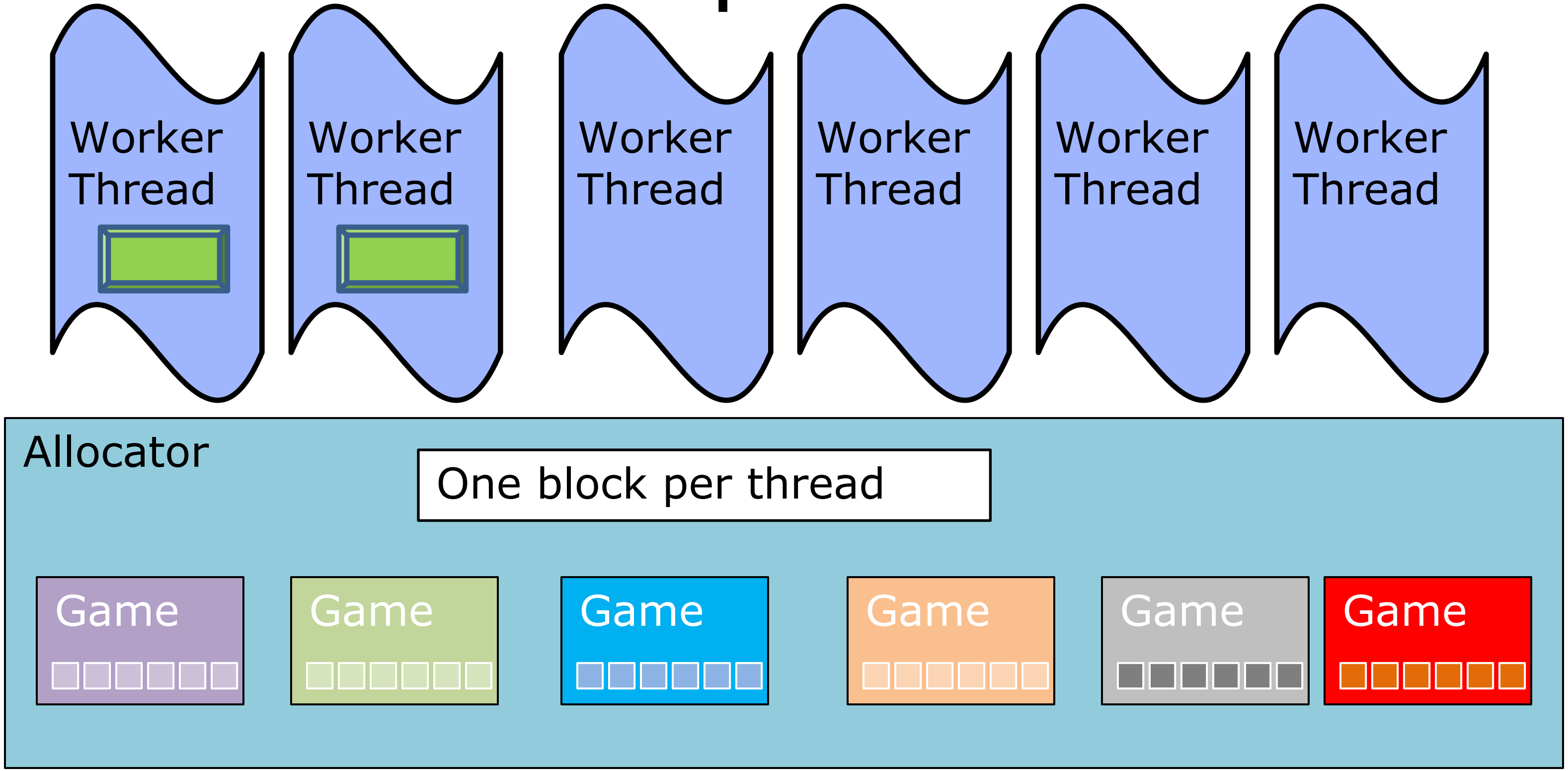
Allocator with single block



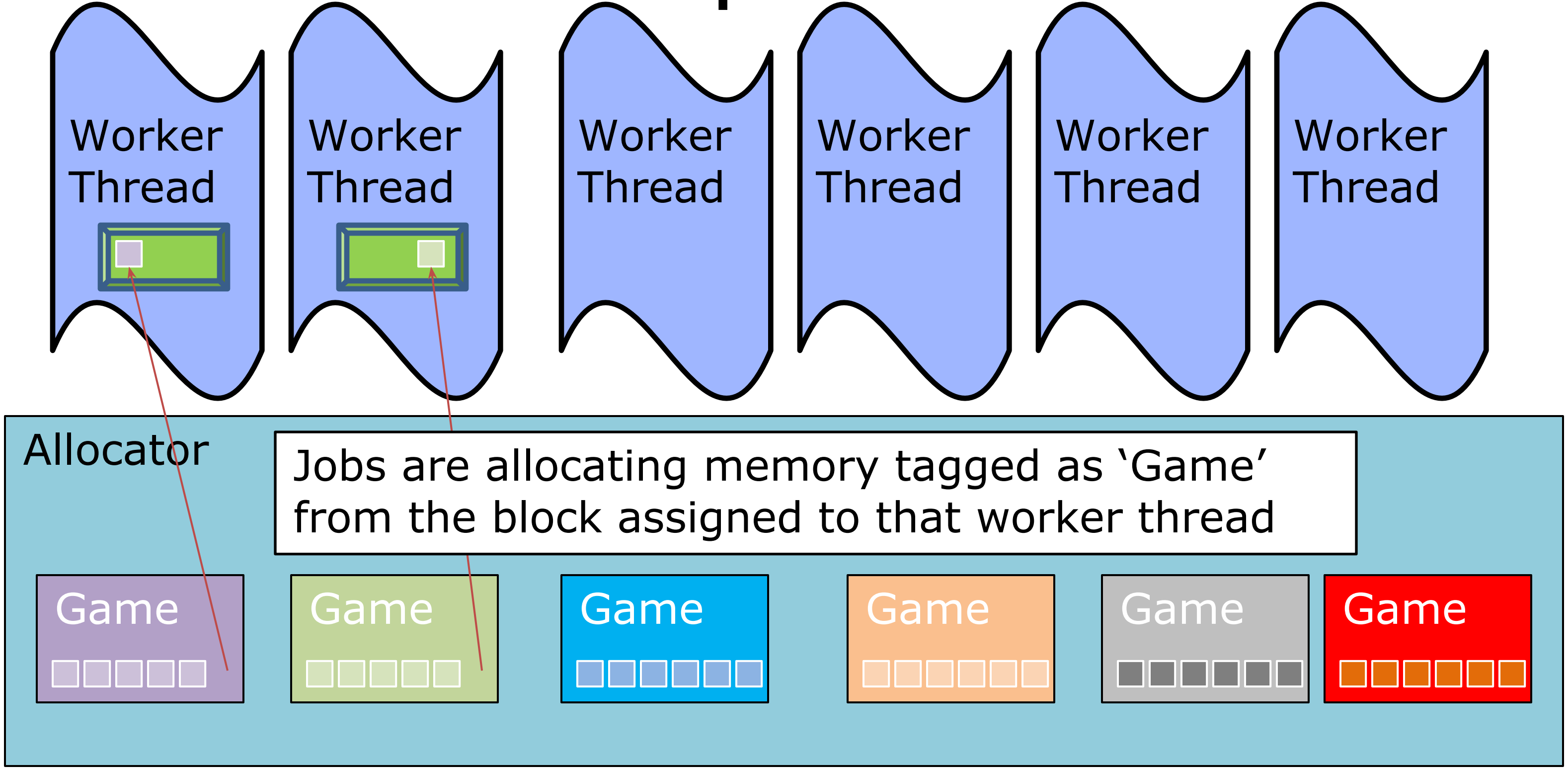
Faster…

- Store one 2 MiB block per worker thread in the allocator
  - We use worker thread index to select which block to use
- All allocations on that thread go to that thread's block
  - No contention
- No locks required for 99.9% of memory allocations
- High volume, high-performance allocator

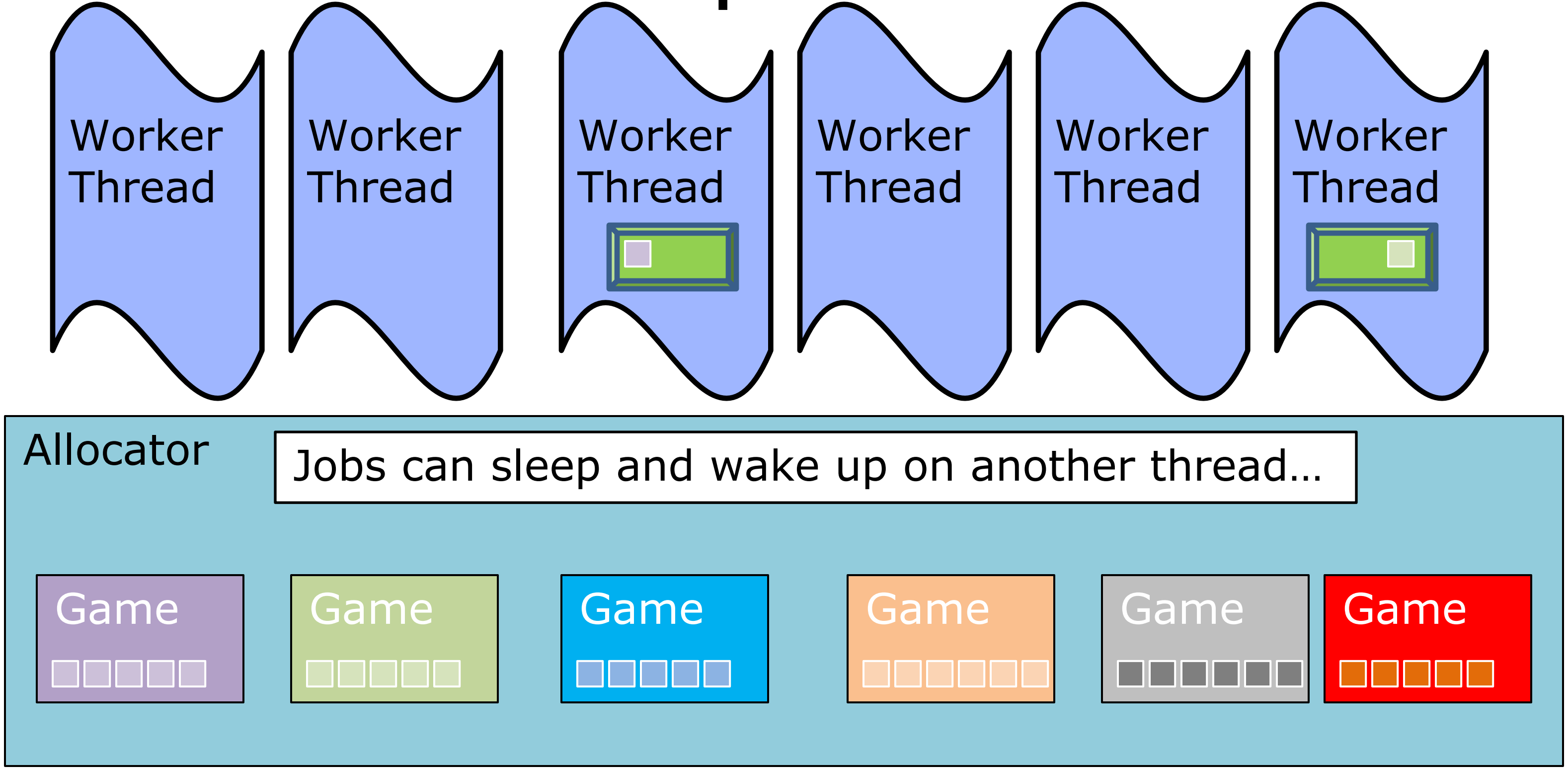
Allocator with per-thread blocks



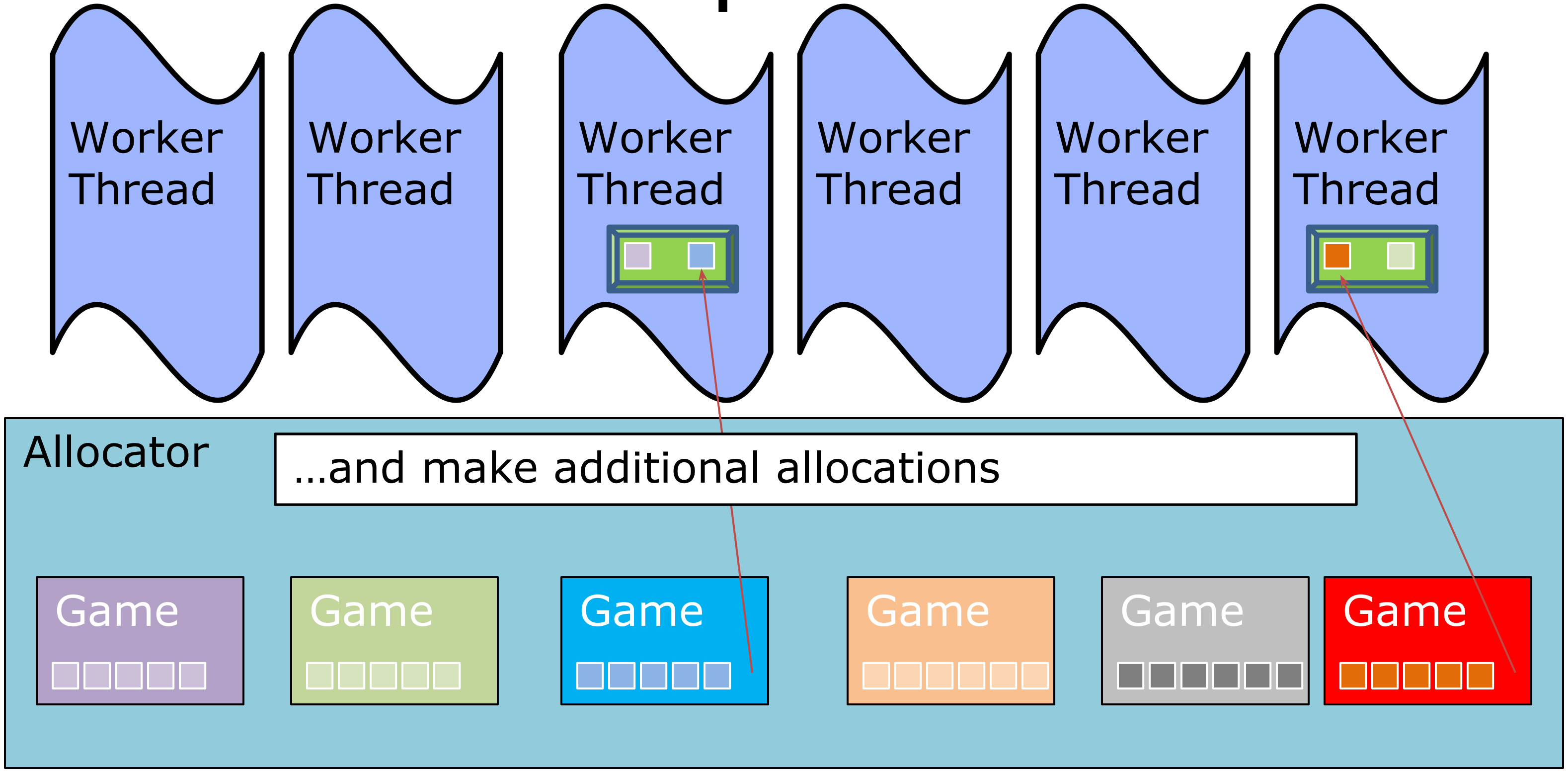
Allocator with per-thread blocks



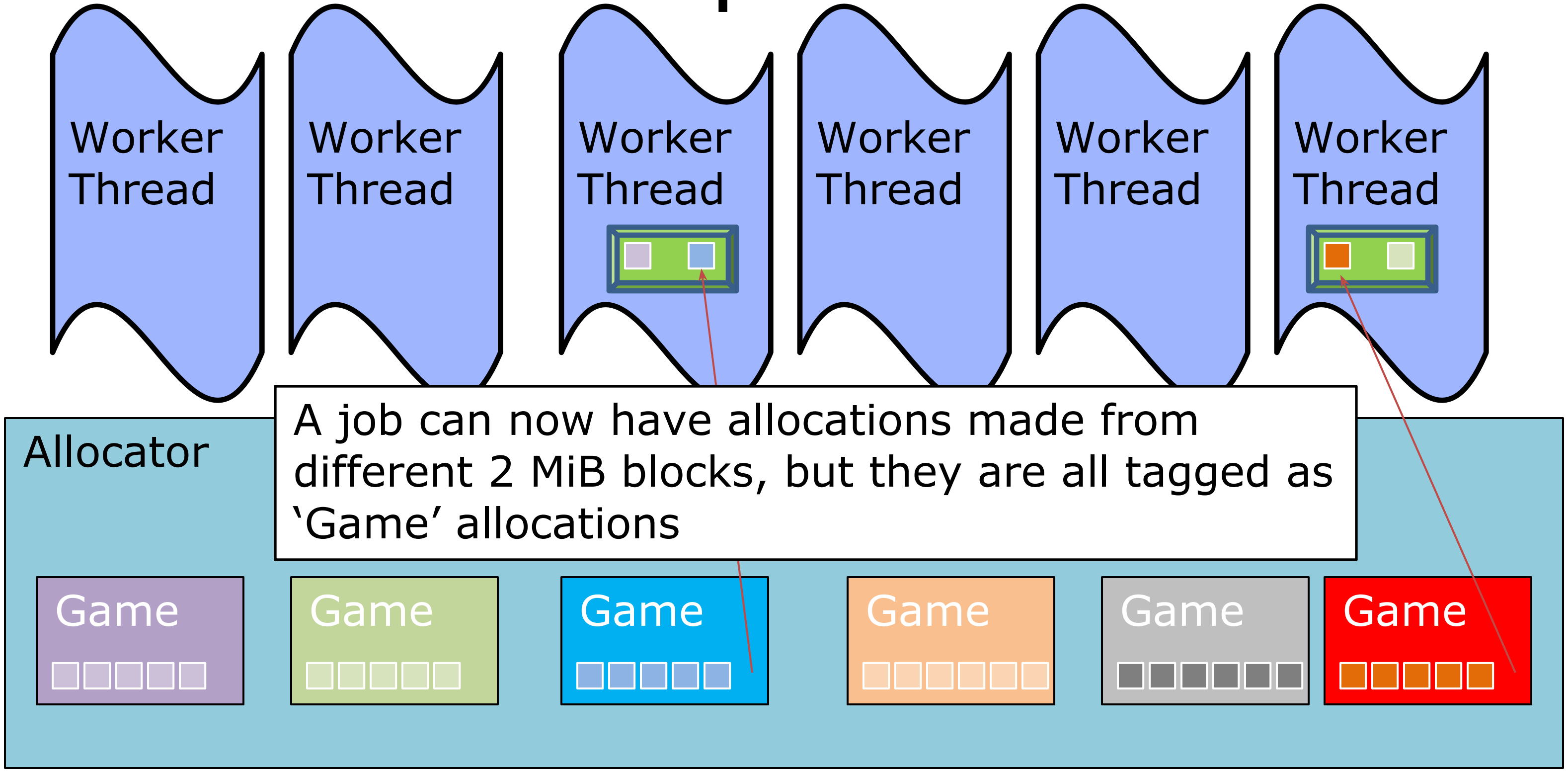
Allocator with per-thread blocks



Allocator with per-thread blocks



Allocator with per-thread blocks



Summary

- Fibers are awesome
- Frame-centric design simplifies your engine
  - Using something like FrameParams greatly simplifies data lifetime and memory management
- Tag-based block allocators are great when dealing with a multi-frame engine design

Thank you!

Questions?

Christian Gyrling
christian_gyrling@naughtydog.com