Understanding GPU Kernels with Instrumentation

Yumeng Liu
Advisor: John Mellor-Crummey
Rice University

Scalable Tools Workshop 2025

Performance Analysis on a GPU

- Current tools can provide
  - Traces of GPU functions (kernel launch, memory copies, etc.)
  - High-level metrics of GPU kernels (GPU utilization, bandwidth utilization, etc.)
  - Detailed metrics per instruction (instruction counts, etc.)
  - Traces of metrics
- A missing feature
  - Traces of GPU **device** function calls **inside** kernels over kernel execution time
    - Why? Behavior of complex kernels may differ greatly across threads

Complex GPU-Accelerated Application: Quicksilver

- A proxy application for a dynamic Monte Carlo particle transport code *Mercury*, **consisting of a single GPU kernel with thousands of lines of code that invokes other GPU device functions**
- High-level structure of the GPU kernel
  - Each GPU thread works on one particle
  - Each particle keeps evolving until a termination condition is met
  - **Work for particles differ between threads**

```
cycle_tracking() {
  for all particles {
    do {
      compute distance to census
      compute distance to facet
      compute distance to reaction
      do segment with shortest distance
      increment tallies
    } until census, absorbed, escaped
  }
}
```

Potential load imbalance, optimization opportunity

Our Goals

- Trace GPU device function calls within kernels to understand thread behavior
- Explore potential optimizations

Outline

- Our trace tool
- Potential optimization opportunities of Quicksilver found with the trace tool
- Exploration of an optimization opportunity
- Work stealing algorithm among GPU threads
- Summary

Outline

- **Our trace tool**
- Potential optimization opportunities of Quicksilver found with the trace tool
- Exploration of an optimization opportunity
- Work stealing algorithm among GPU threads
- Summary

Instrumentation of GPU Binaries

- Use **NVBit** for dynamic binary instrumentation on NVIDIA GPUs
- Add instrumentation to trace function invocations on each thread
  - Instrumentation points
    - Before a call (CALL)
    - Upon function entry
    - Before function exit (RET, EXIT)

Instrumentation of GPU Binaries

- At each instrumentation point, emit a record containing
  - Warp ID + Mask => Thread ID - who
  - Timestamp - when
  - Address (Special: EXIT will be 0) - where

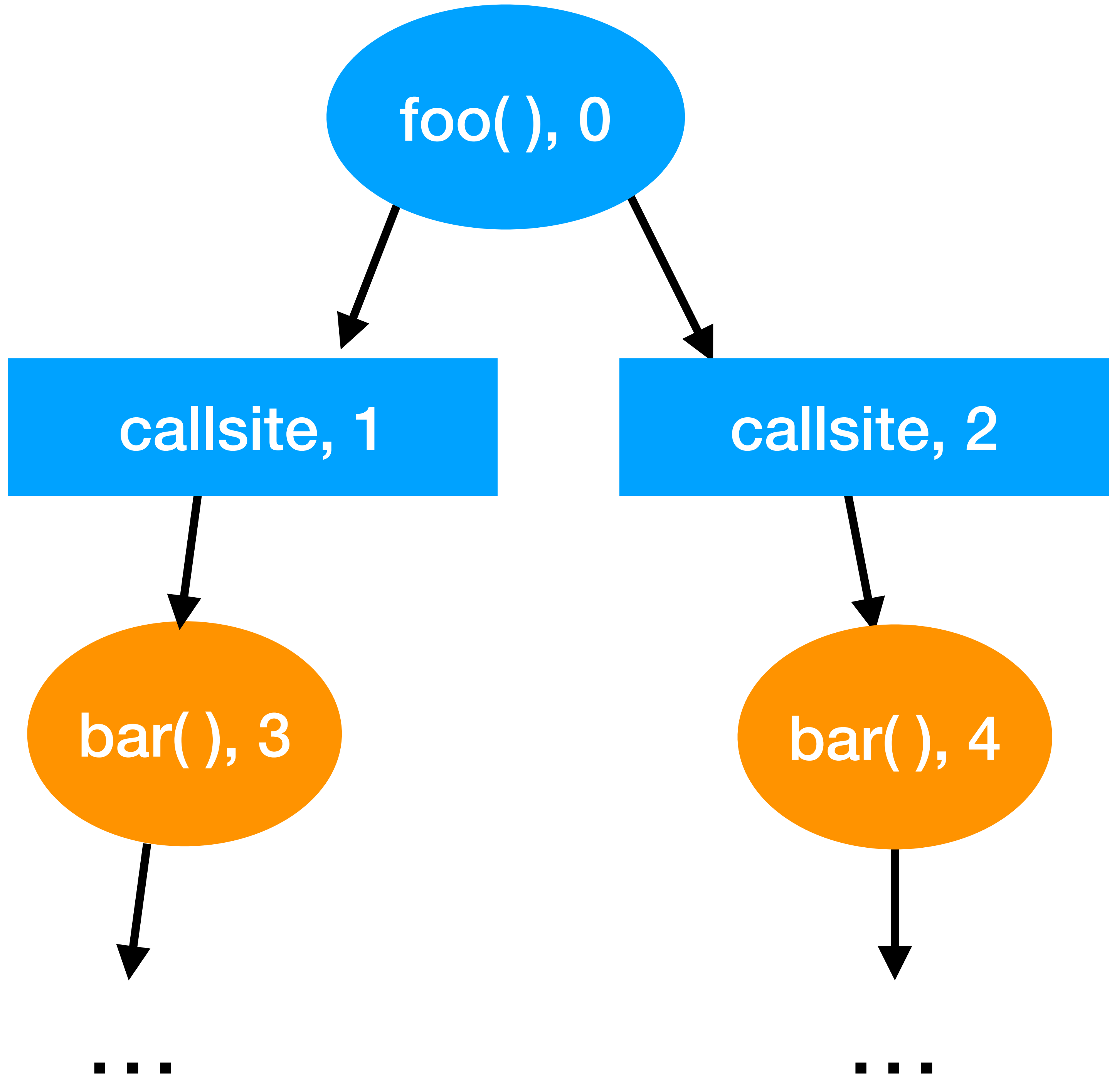
Moving Trace Records to CPU

- Use ChannelDev/ChannelHost objects from NVBit to move records back to CPU
    - One buffer on GPU, one buffer on CPU, flush the buffer when it’s full

Processing Trace Stream on CPU

- Input: a stream of records containing who (thread id), where (function), when (timestamp)
- Output: **Calling Context Tree (CCT)** and trace, using HPCToolkit measurement format

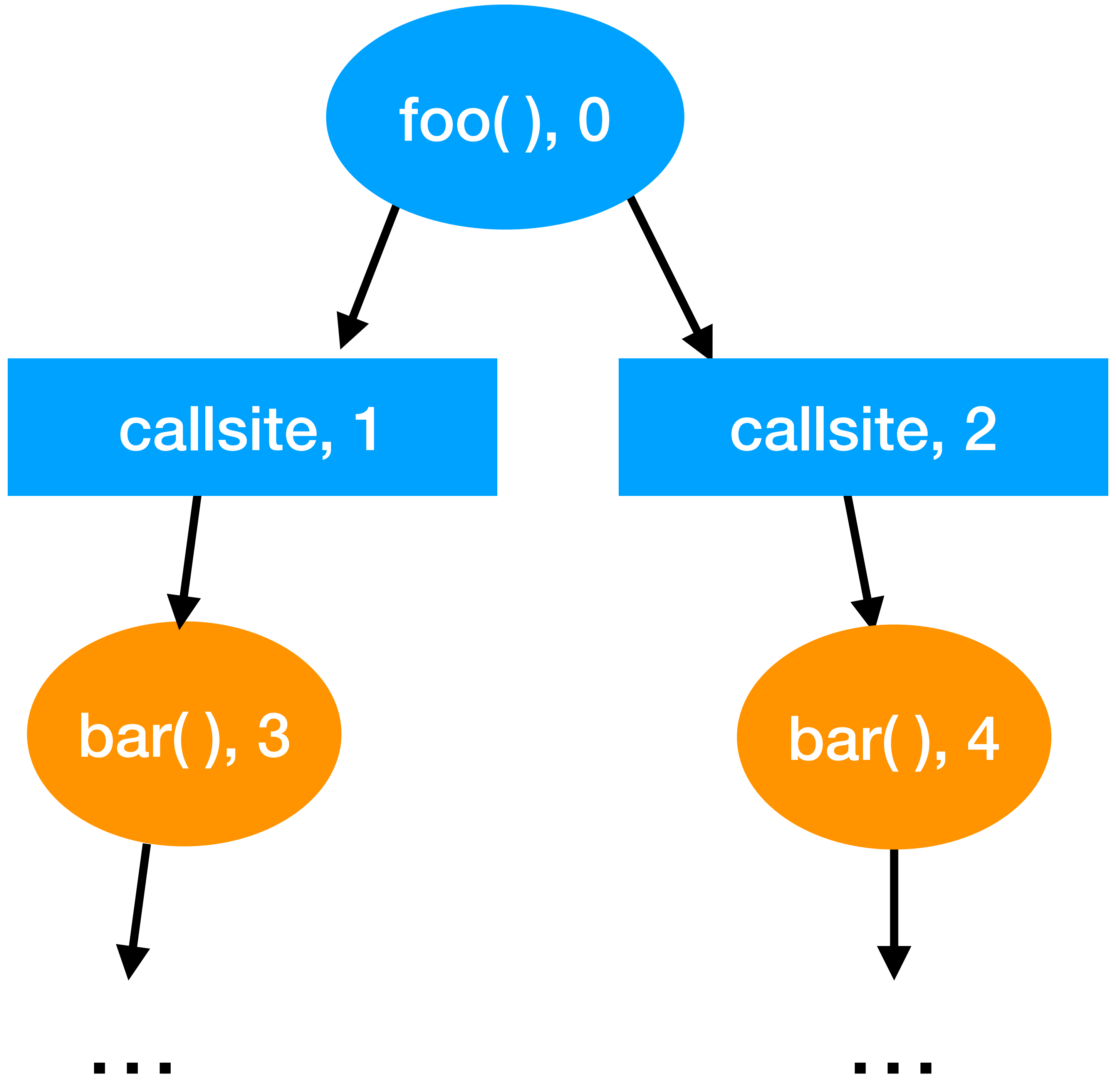
```
void foo() {
    // ...
    bar();
    // ...
    bar();
    // ...
}
```



Callsite information differentiates multiple calls by a function to the same callee

Processing Trace Stream on CPU

- Input: a stream of records containing who (thread id), where (function), when (timestamp)
- Output: Calling Context Tree (CCT) and **trace**, using HPCToolkit measurement format



[timestamp, CCT Node ID]

[t0, 0] [t1, 1] [t2, 3] [t3, 1] [t4, 0] [t5, 2] [t6, 4] …

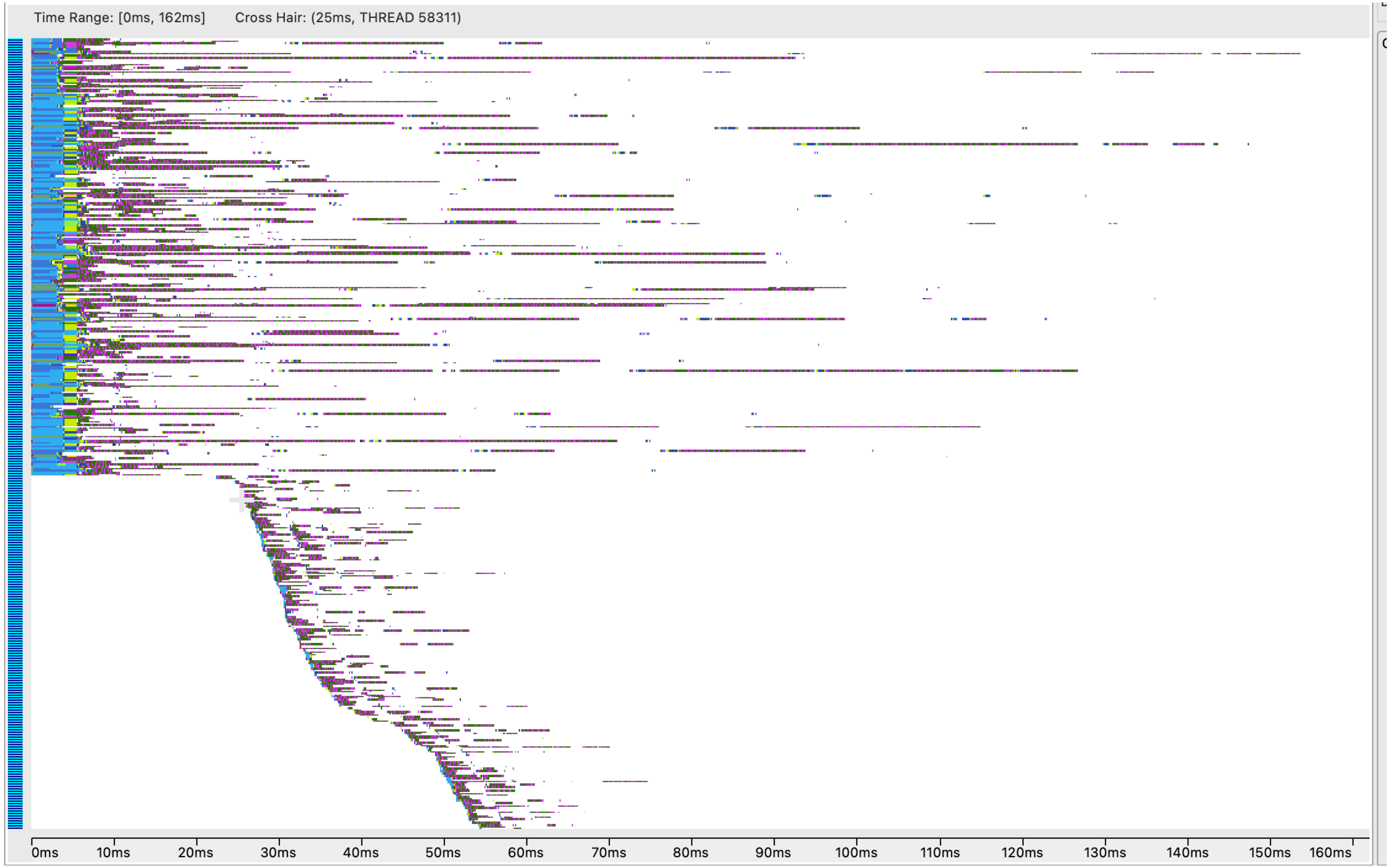
Post-mortem Analysis

- Analyze the GPU binary with *hpcstruct*
- Interpret the trace data with *hpcprof*
- View the traces in *hpcviewer*

Outline

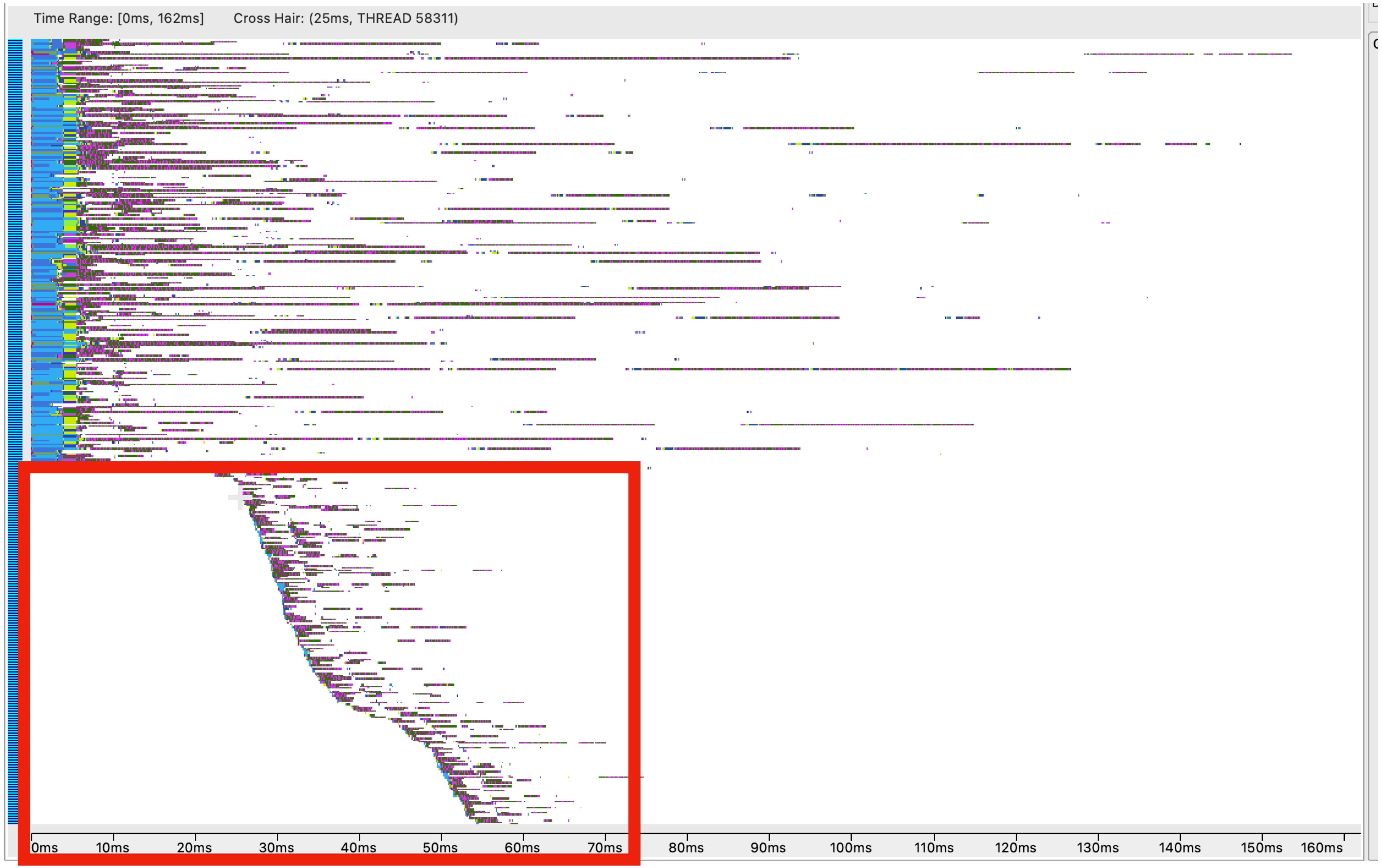
- Our trace tool
- **Potential optimization opportunities of Quicksilver found with the trace tool**
- Exploration of an optimization opportunity
- Work stealing algorithm among GPU threads
- Summary

Quicksilver in Our Tool



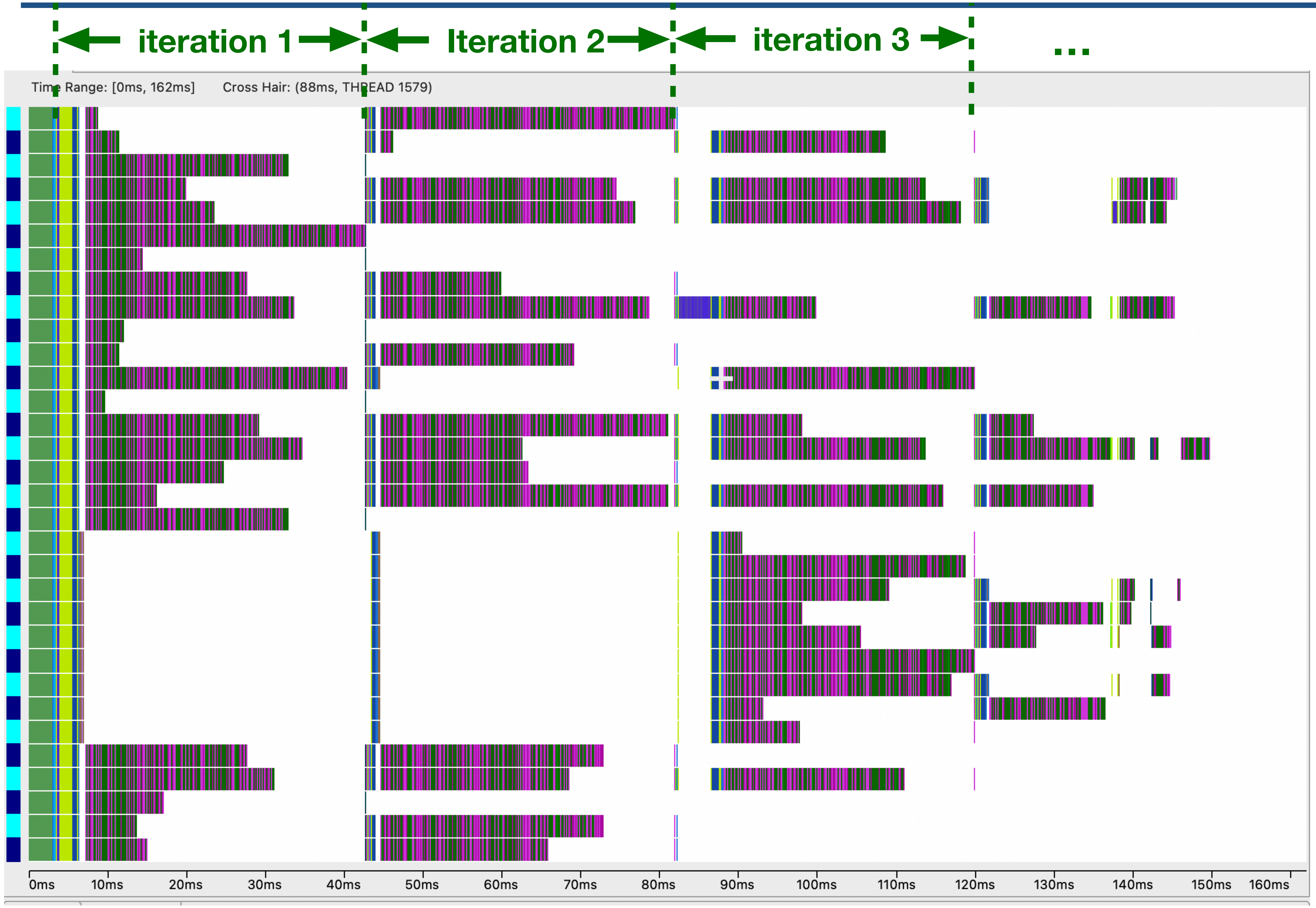
- 100,000 particles
- 100,096 threads
- First impression
  - A lot of idleness
  - Obvious load imbalance

Quicksilver in Our Tool



- 100,000 particles
- 100,096 threads
- First impression
  - A lot of idleness
  - Obvious load imbalance
  - Roughly half threads start late

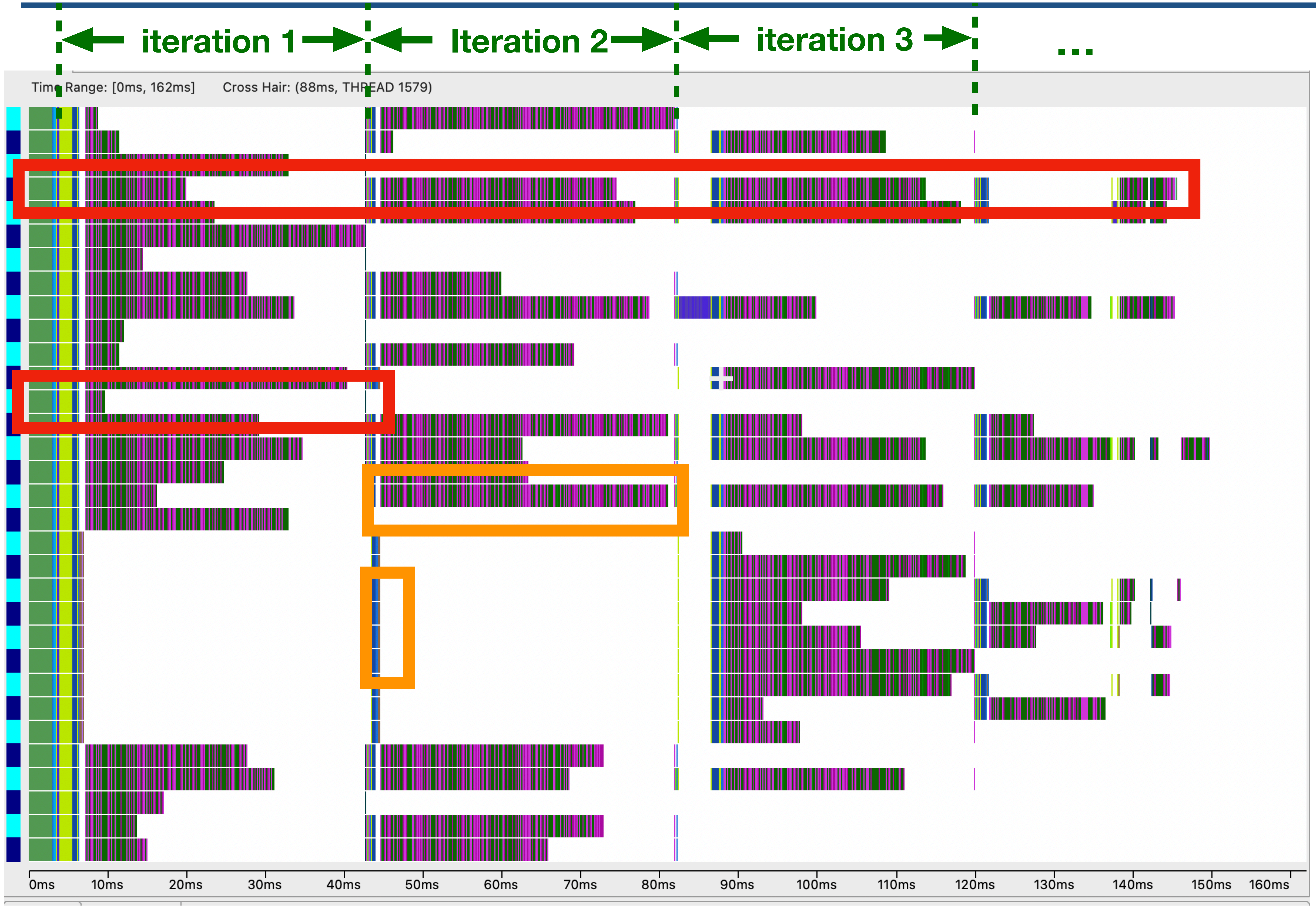
Quicksilver in Our Tool




```
CycleTrackingFunction(){
    do{
        calculate an outcome
        switch(outcome){
            case 1
            case 2
            …
        }
    }while(keepTracking);
}
```

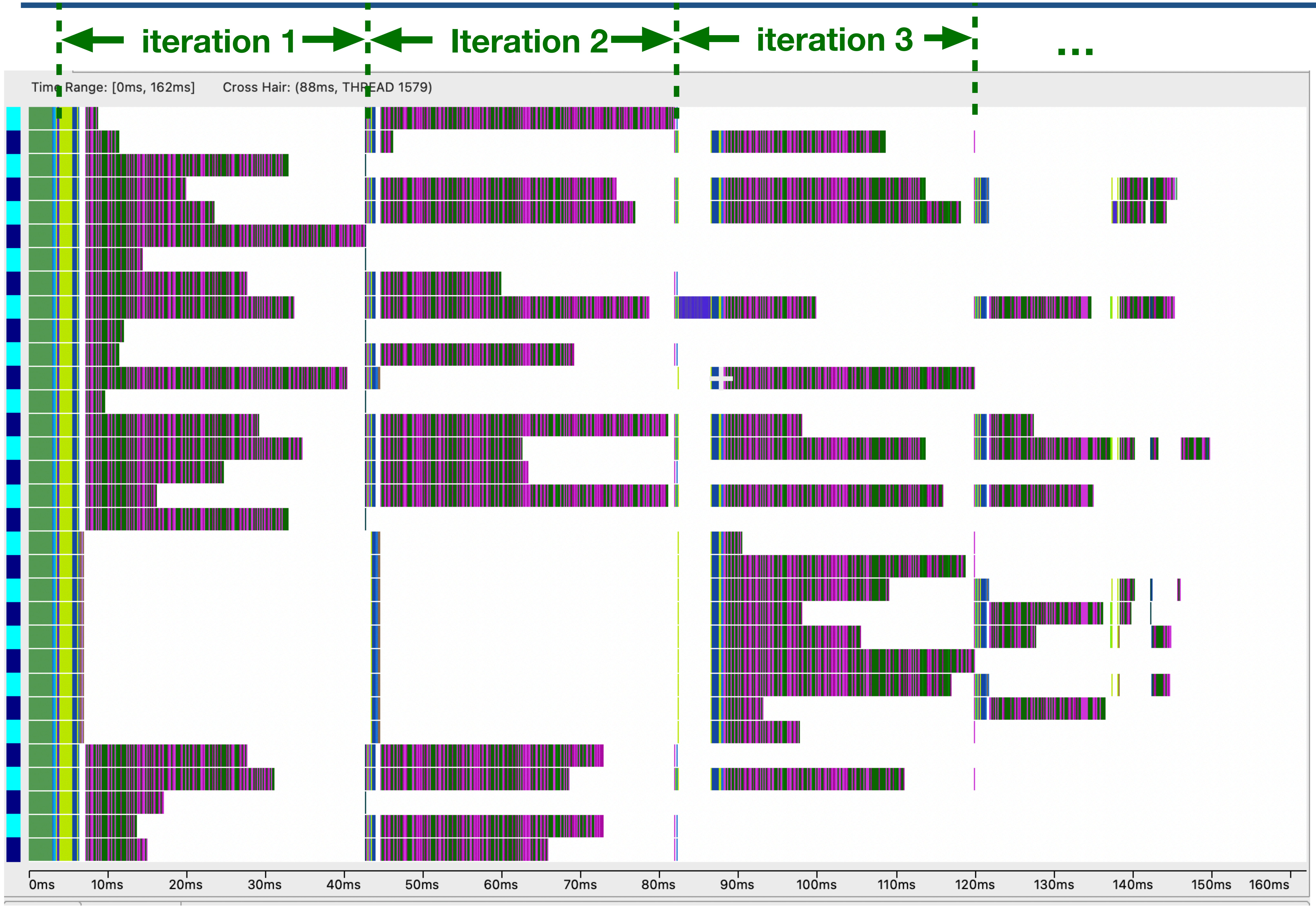
- Each case
    - Some computation
    - Decide keepTracking

Quicksilver in Our Tool



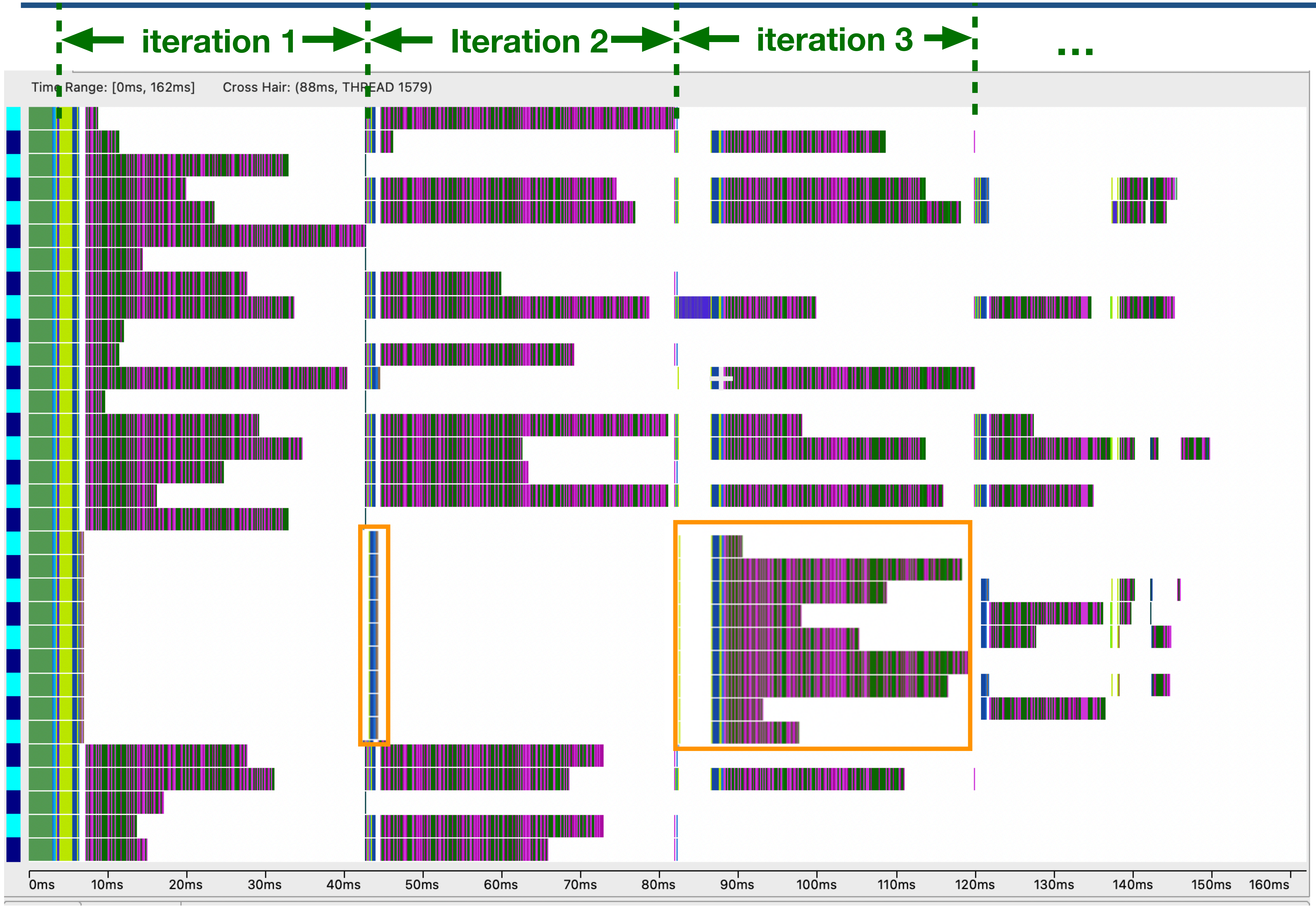
- Two imbalance
  - Imbalance **between** tasks
  - Imbalance between cases **within** each iteration

Quicksilver in Our Tool



- Two imbalance
  - Imbalance **between** tasks
  - Imbalance between cases **within** each iteration
- Two optimization opportunities
  - Compact tasks among threads

Quicksilver in Our Tool



- Two imbalance
  - Imbalance **between** tasks
  - Imbalance between cases **within** each iteration
- Two optimization opportunities
  - Compact tasks among threads
  - Each thread executes short cases until the next case is the long one, and all the threads work on the long case together

Outline

- Our trace tool
- Potential optimization opportunities of Quicksilver found with the trace tool
- **Exploration of an optimization opportunity**
- Work stealing algorithm among GPU threads
- Summary

Exploration with Mini Proxy App

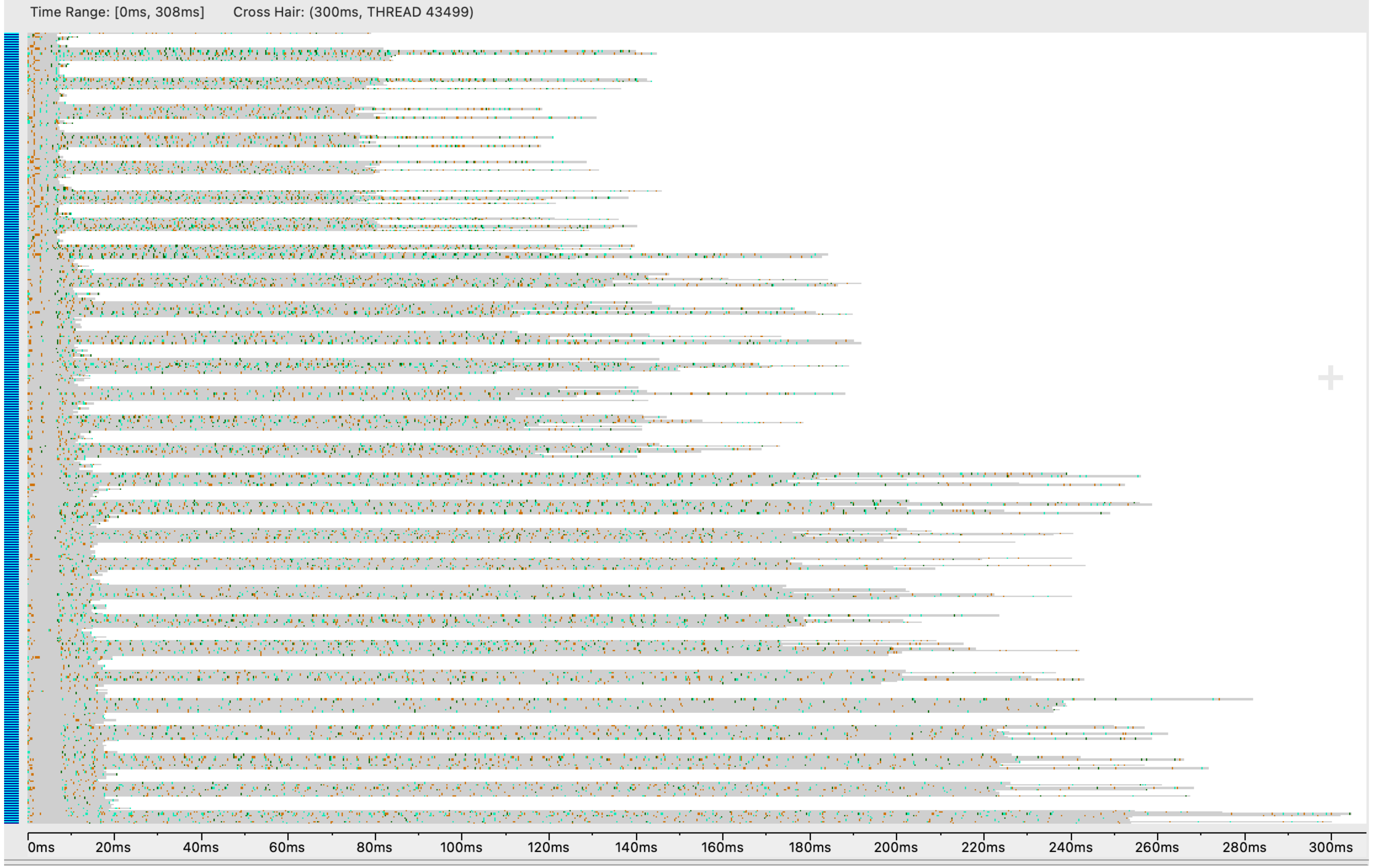
- N threads and N tasks, each thread works for one task, N = 100,096
- Each task can be either SHORT (5 * 100,000 additions) or LONG (100 * 100,000 additions)
- Terms:
    - Even threads: threads with even id
    - Odd threads: threads with odd id
    - First half threads: threads with id in [0, N/2)
    - Second half threads: threads with id in [N/2, N)

Mini Proxy App - Potential Benefits

- Same amount of total workload, different threads assignment (who get what task)

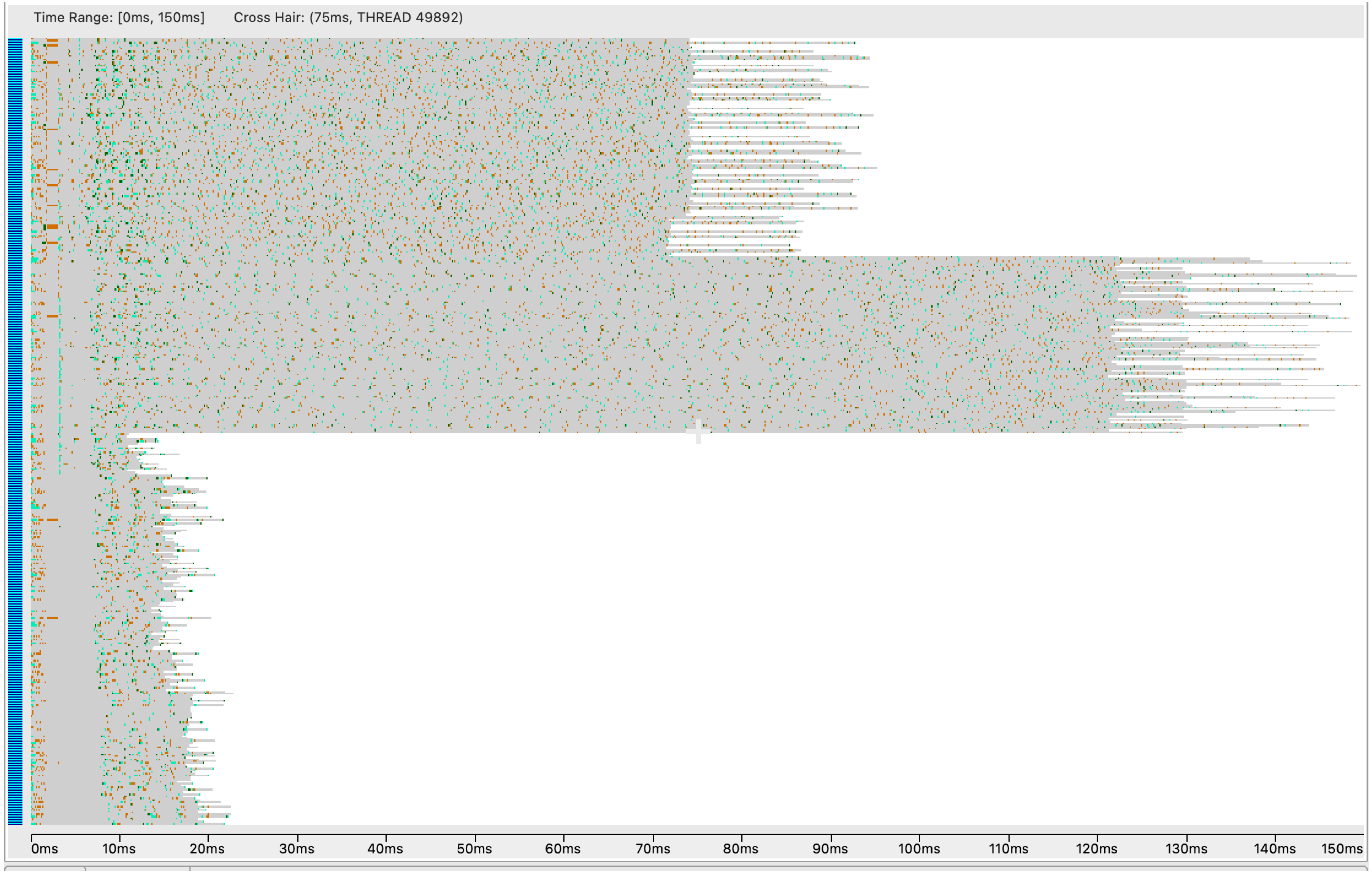
Version	LONG task	SHORT task
odd-even	even threads	odd threads
long-short	first half	second half
short-long	second half	first half

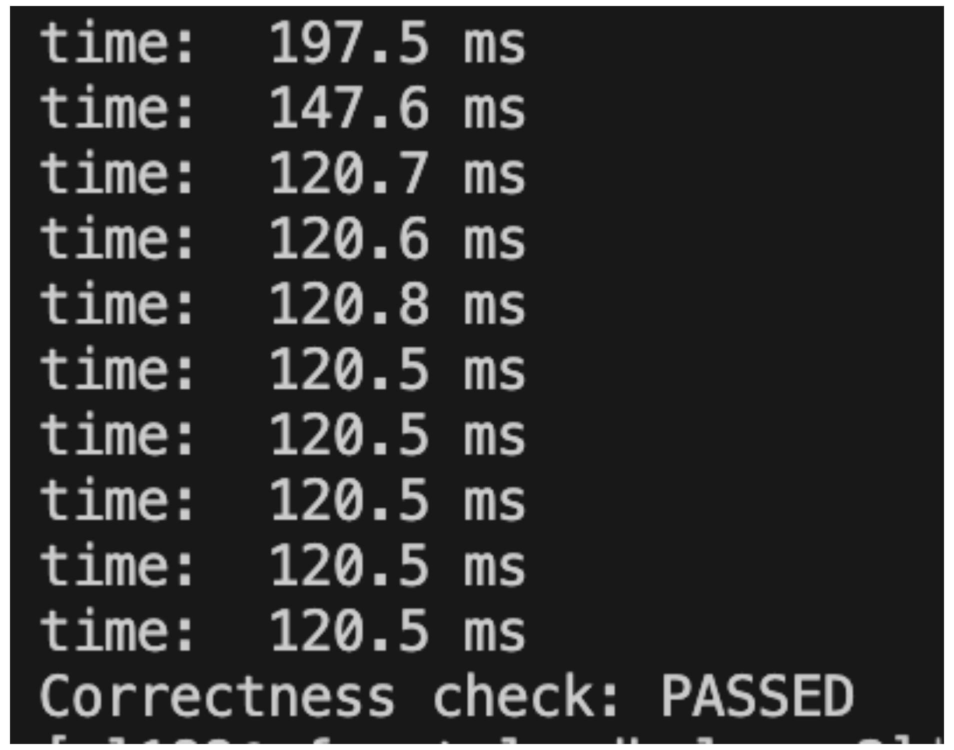
Mini Proxy App - Version odd-even



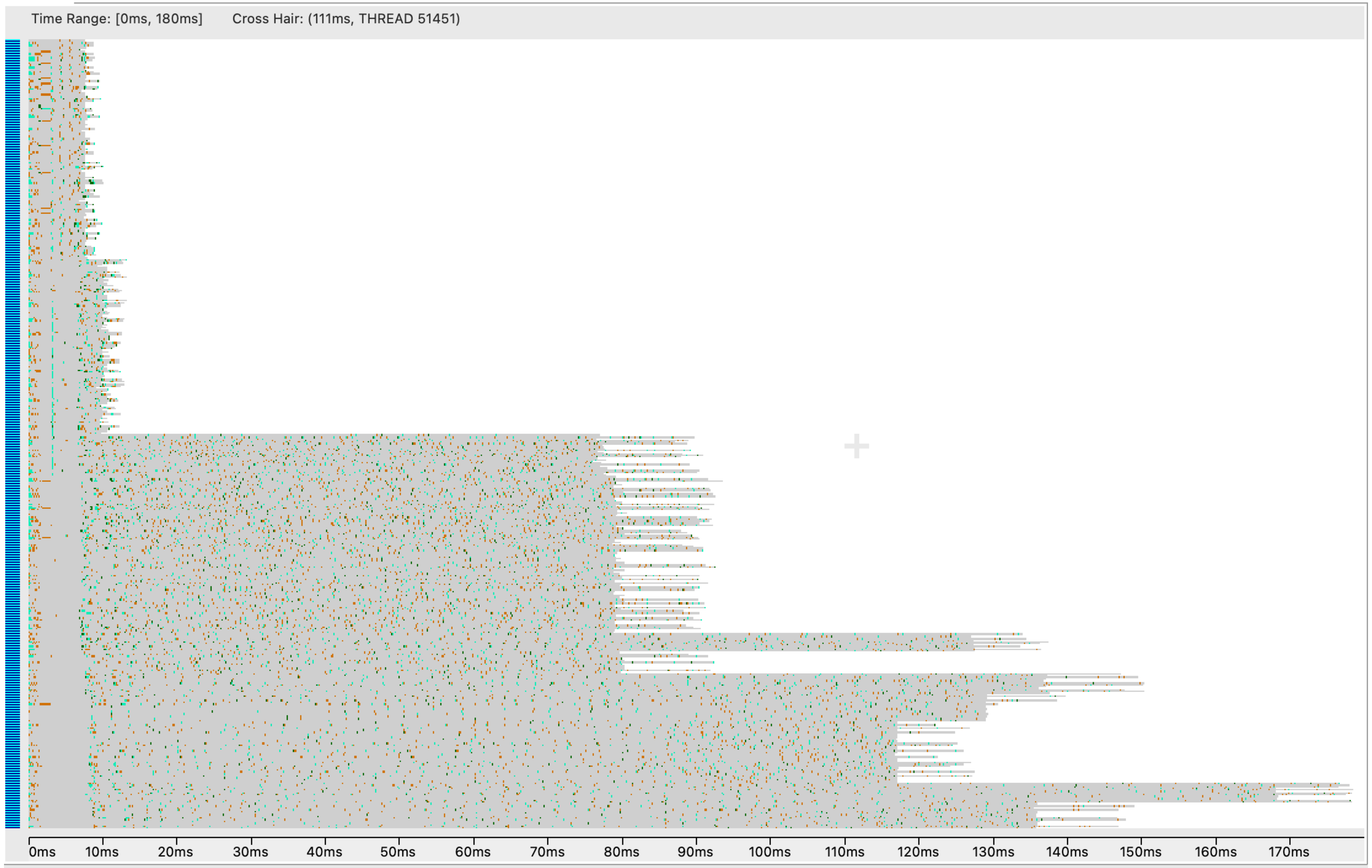

```
time:  336.0 ms
time:  234.2 ms
time:  233.5 ms
time:  233.2 ms
time:  233.5 ms
time:  233.5 ms
time:  233.5 ms
time:  233.5 ms
time:  233.5 ms
time:  233.5 ms
Correctness check: PASSED
```

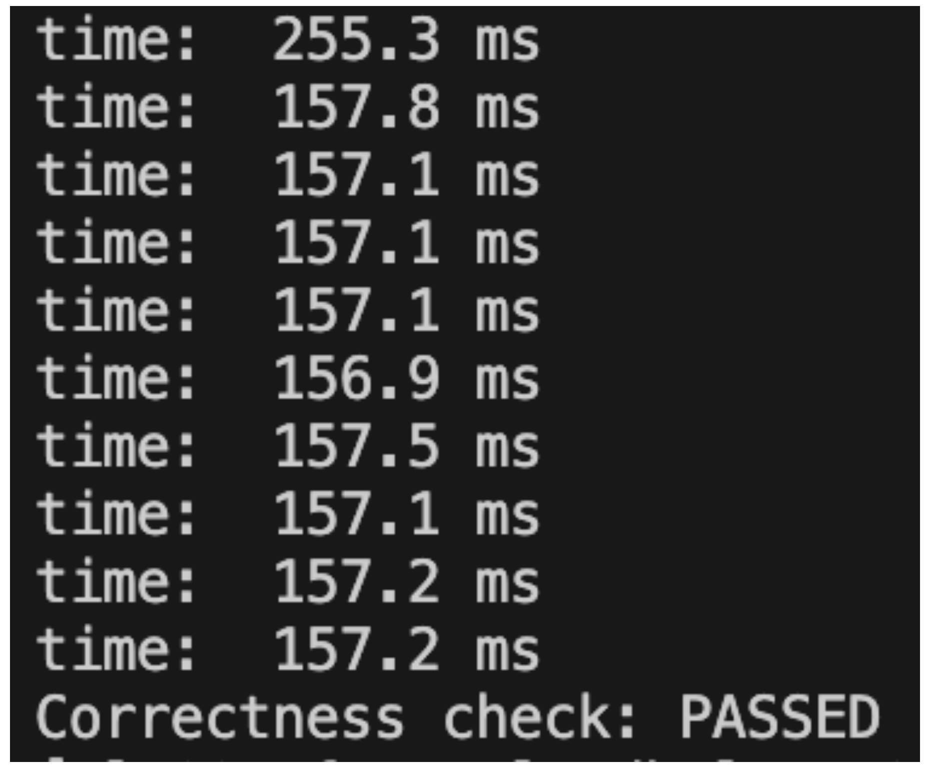
Mini Proxy App - Version long-short





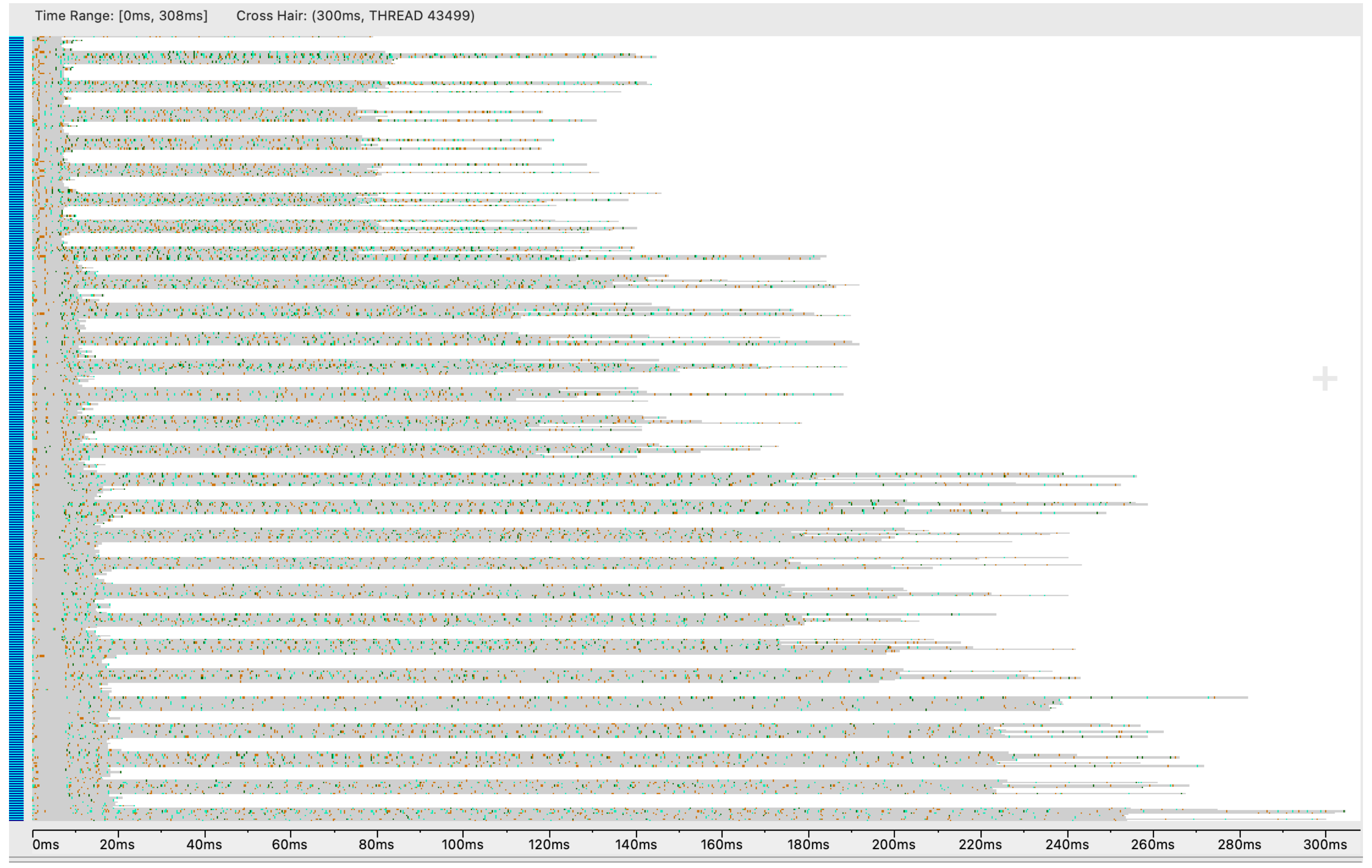
Mini Proxy App - Version short-long





Compact Work is Faster

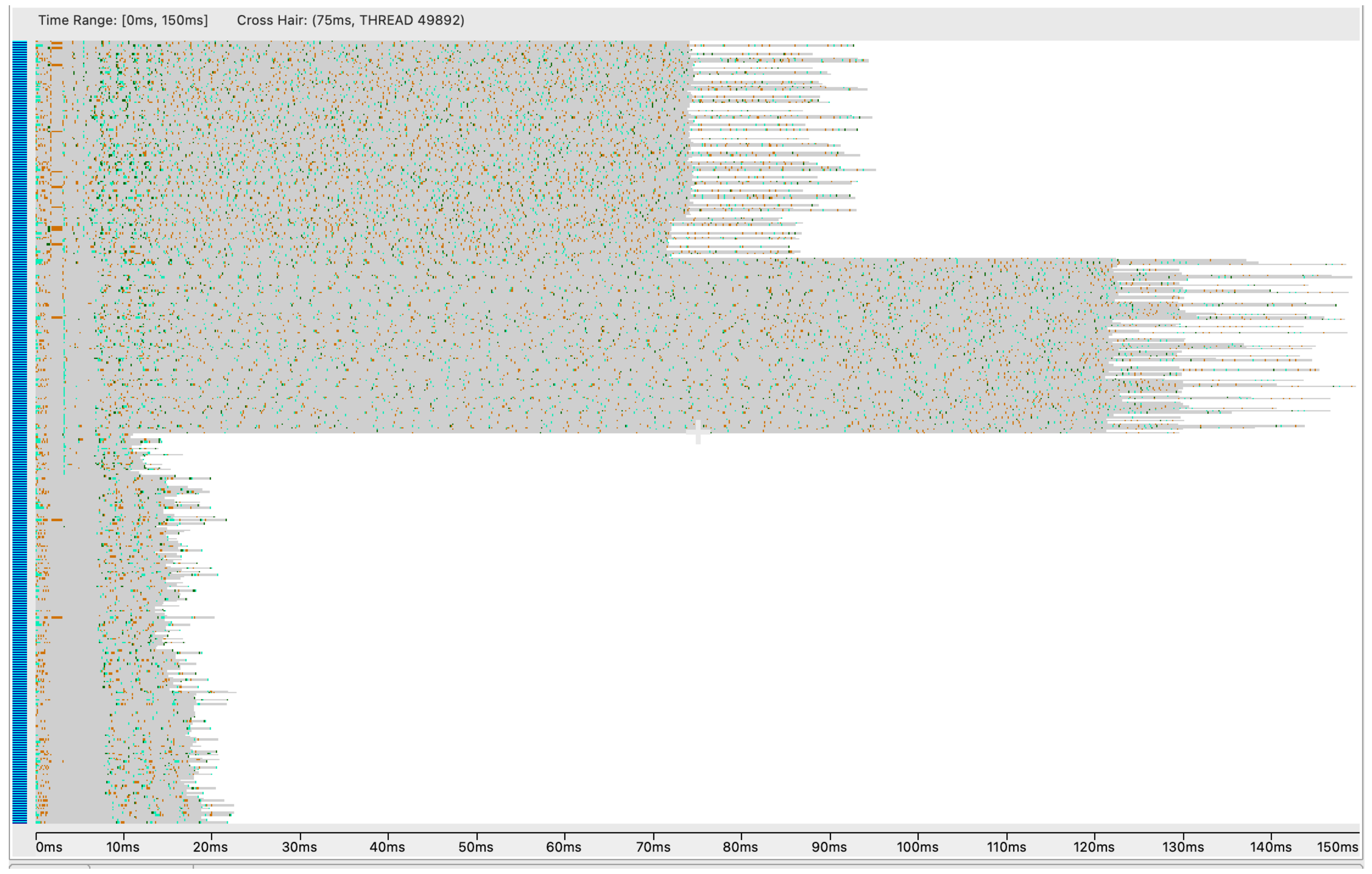
odd-even



```
time:   336.0 ms
time:   234.2 ms
time:   233.5 ms
time:   233.2 ms
time:   233.5 ms
time:   233.5 ms
time:   233.5 ms
time:   233.5 ms
time:   233.5 ms
time:   233.5 ms
Correctness check: PASSED
```

Slow

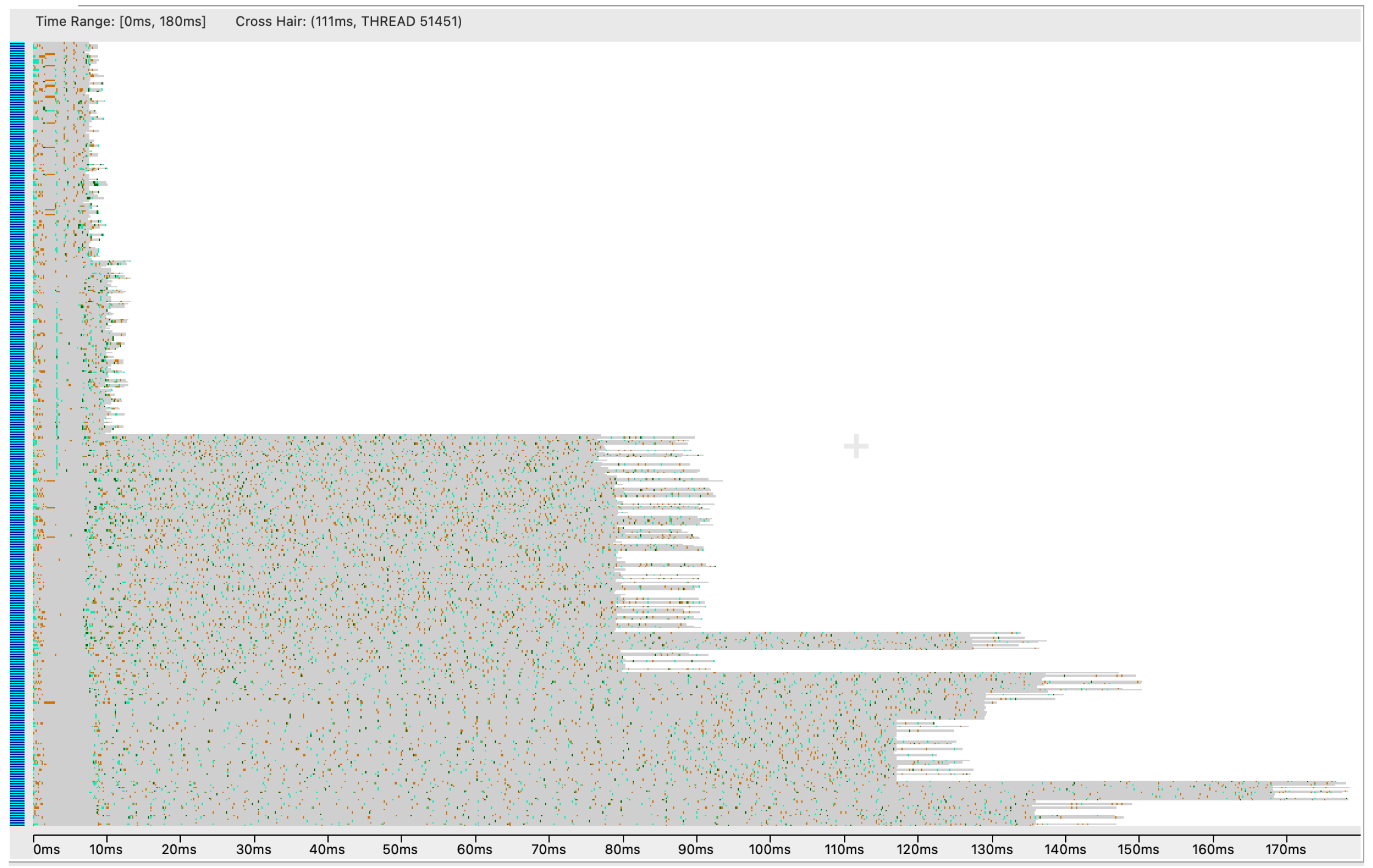
long-short



```
time:   197.5 ms
time:   147.6 ms
time:   120.7 ms
time:   120.6 ms
time:   120.8 ms
time:   120.5 ms
time:   120.5 ms
time:   120.5 ms
time:   120.5 ms
time:   120.5 ms
Correctness check: PASSED
```

1.94x Faster

short-long



```
time:   255.3 ms
time:   157.8 ms
time:   157.1 ms
time:   157.1 ms
time:   157.1 ms
time:   156.9 ms
time:   157.5 ms
time:   157.1 ms
time:   157.2 ms
time:   157.2 ms
Correctness check: PASSED
```

1.49x Faster

Outline

- Our trace tool
- Potential optimization opportunities of Quicksilver found with the trace tool
- Exploration of an optimization opportunity
- **Work stealing algorithm among GPU threads**
- Summary

Work Stealing Among GPU Threads

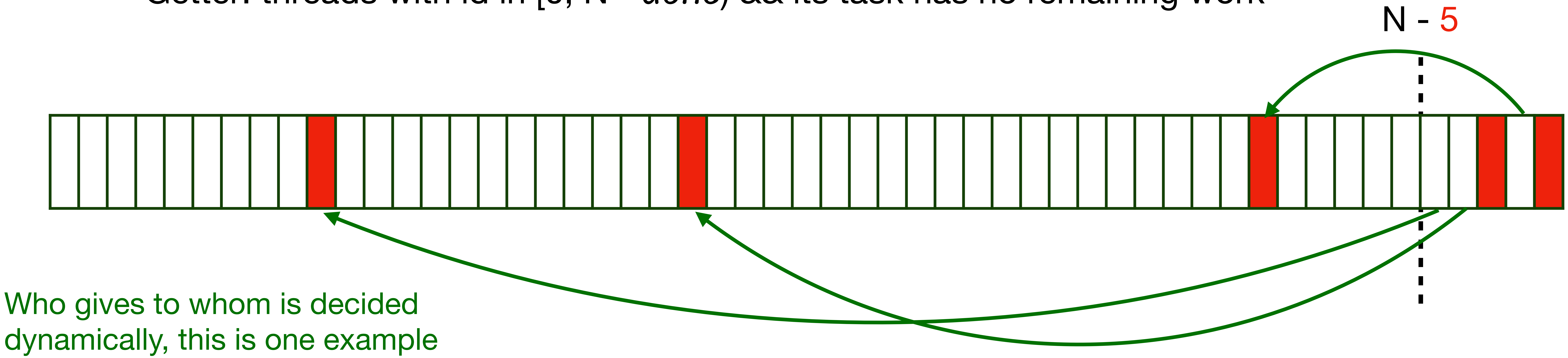
- In Quicksilver, unknown workload imbalance between threads motivates dynamic work stealing
- Pay attention
    - Avoid thread-level spin loop
        - Otherwise, one thread spinning may block the rest in its warp
    - Use __syncwarp( )
    - Consider the scenario that not all the threads start at the same time

Work Stealing Among GPU Threads

- Each thread starts with its own task, and in each iteration
  - Step 1: works on its task
  - Step 2: decides if it should
    - give another thread its task
    - get another thread’s task
    - exit
    - nothing
  - Step 3: behaves according to the decision from Step 2
  - Step 4: __syncwarp()

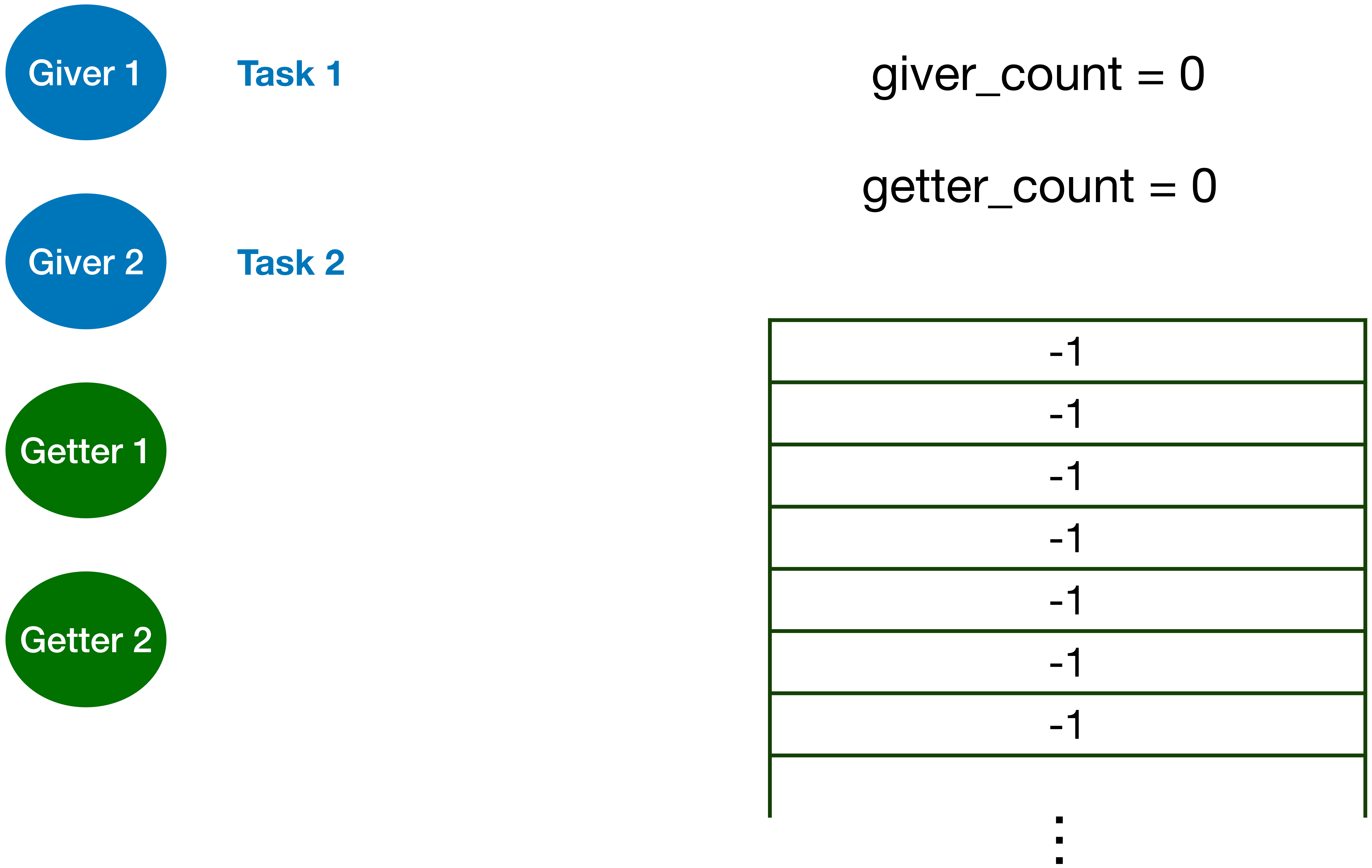
Work Stealing Among GPU Threads

- How to decide if the thread is a giver or getter or none?
  - Global variable *done*: number of finished tasks
  - Giver: threads with id in [N - *done*, N) && its task still has remaining work
  - Getter: threads with id in [0, N - *done*) && its task has no remaining work



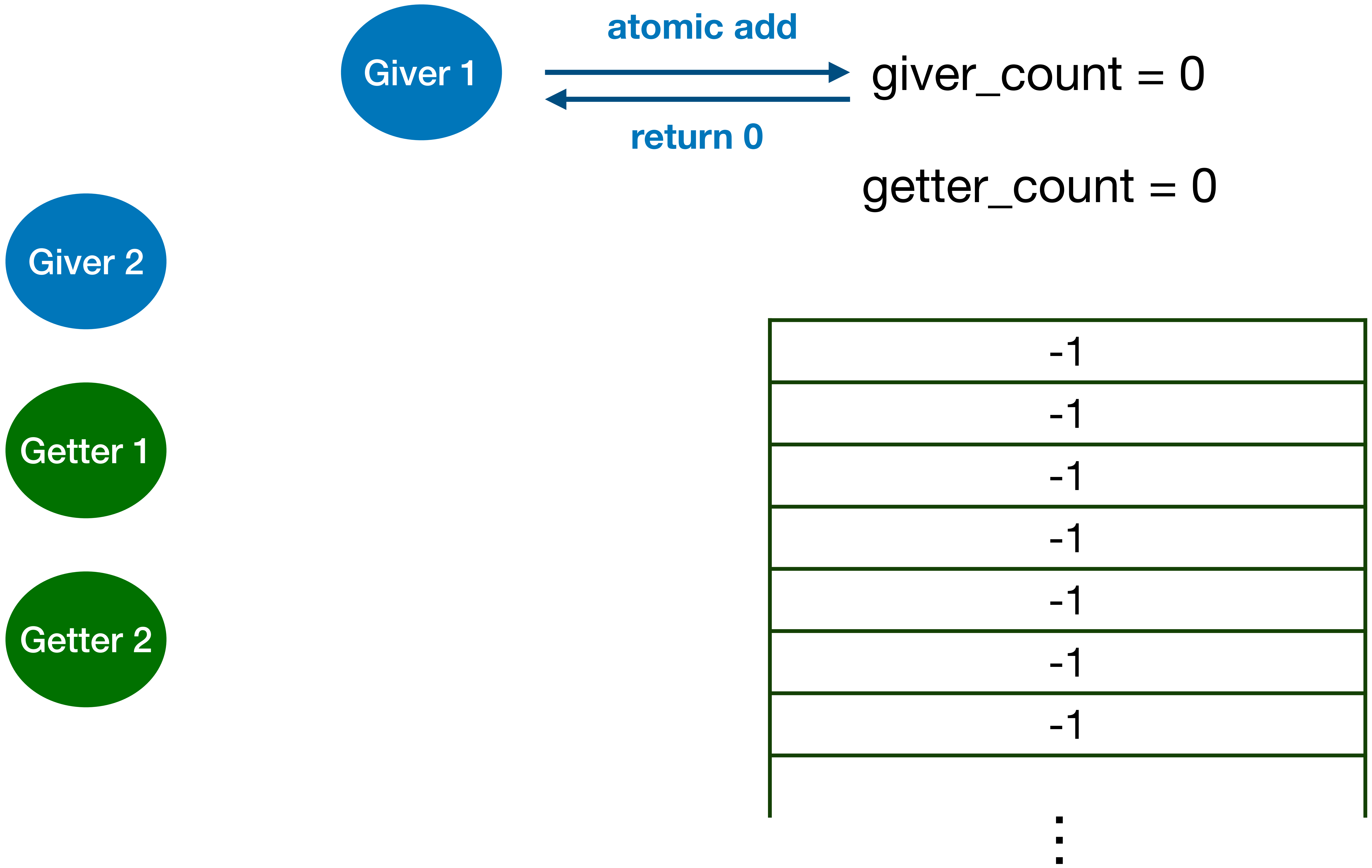
Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



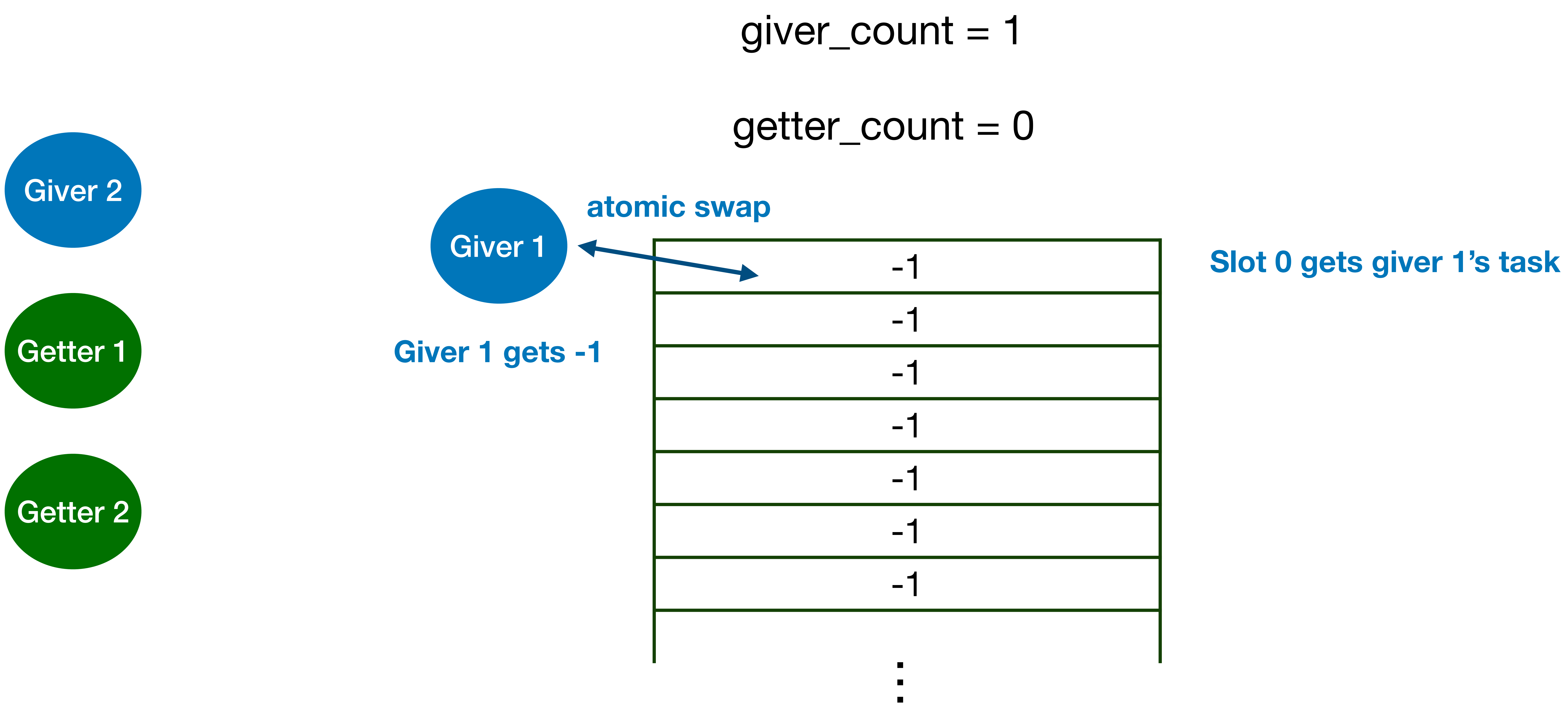
Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



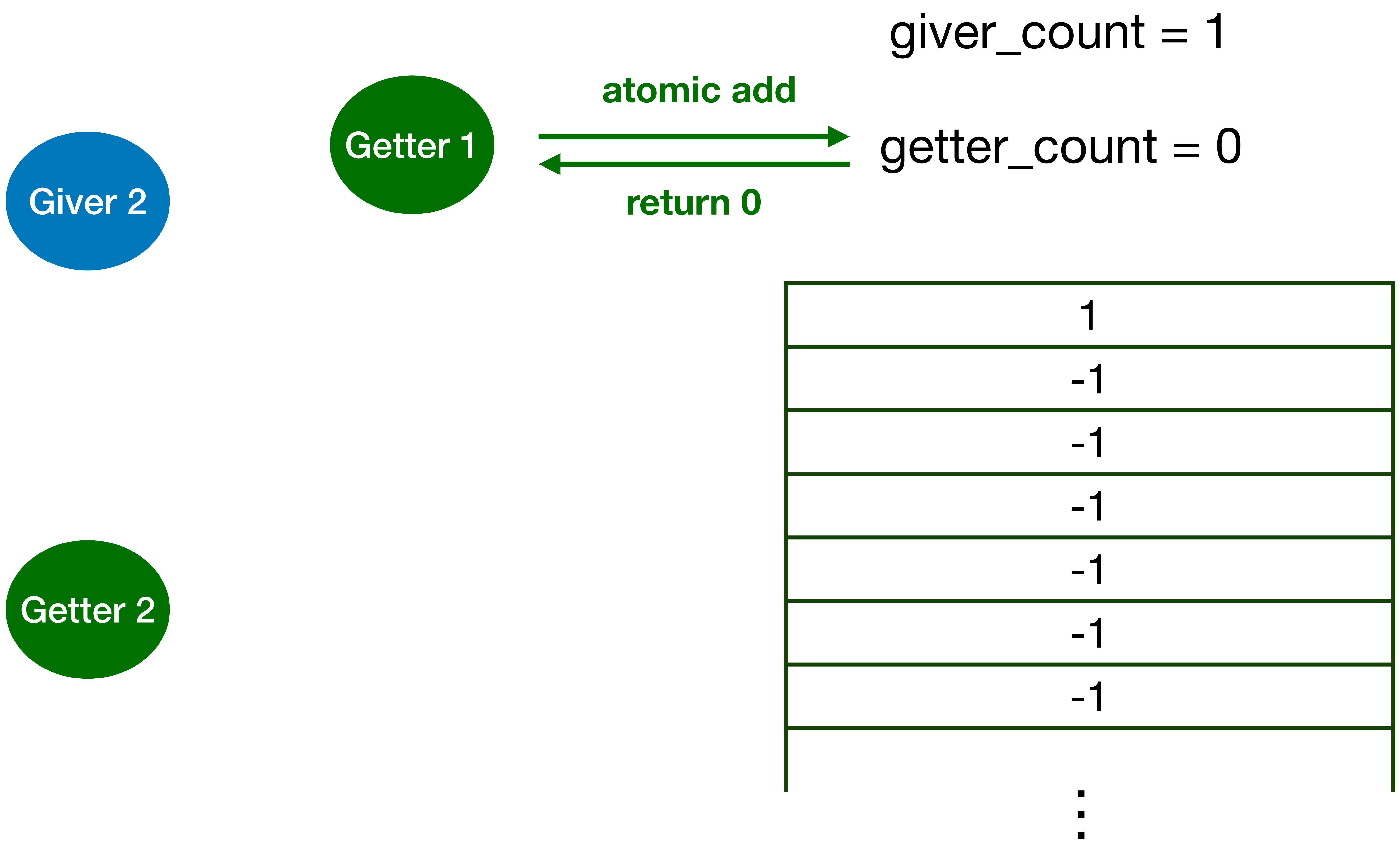
Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



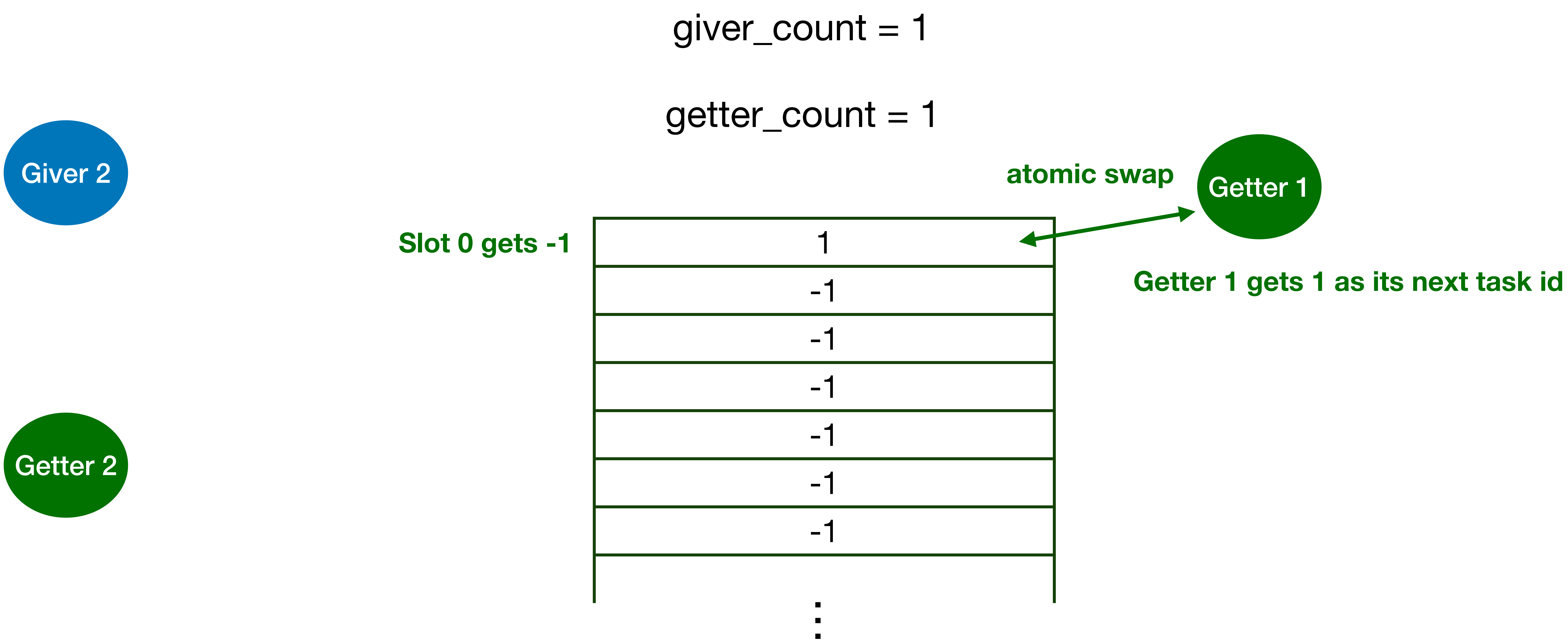
Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



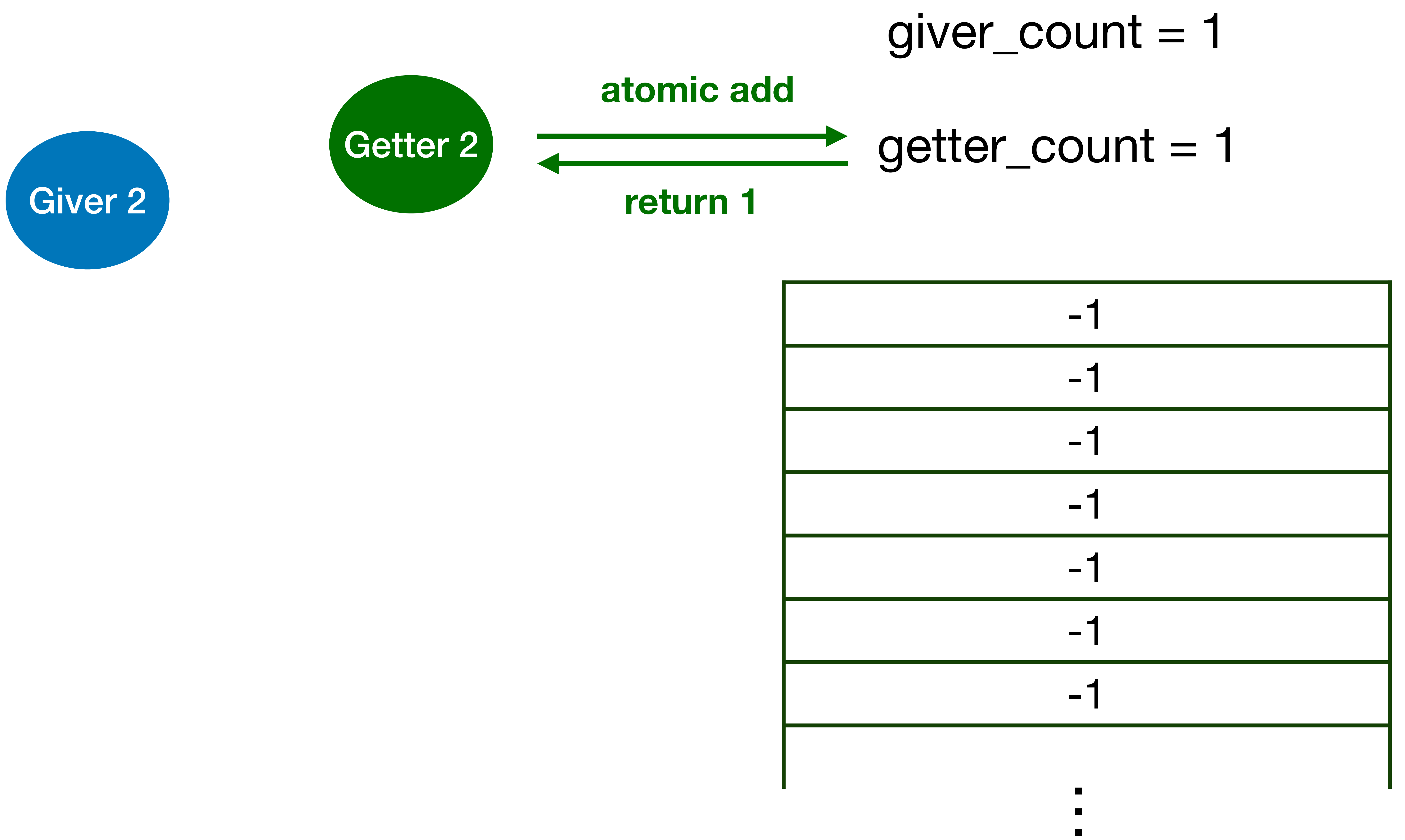
Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



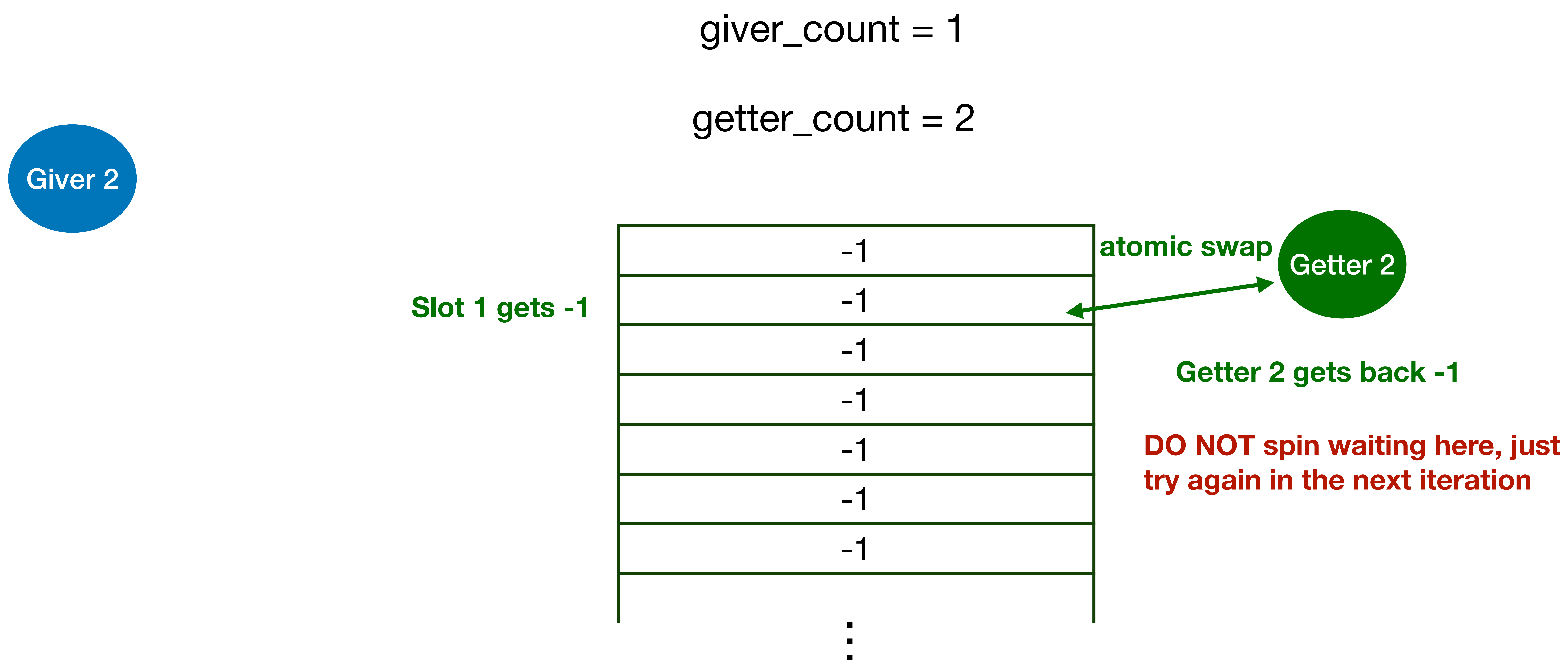
Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



Work Stealing Among GPU Threads

- How to give or get? **Use atomic swap**



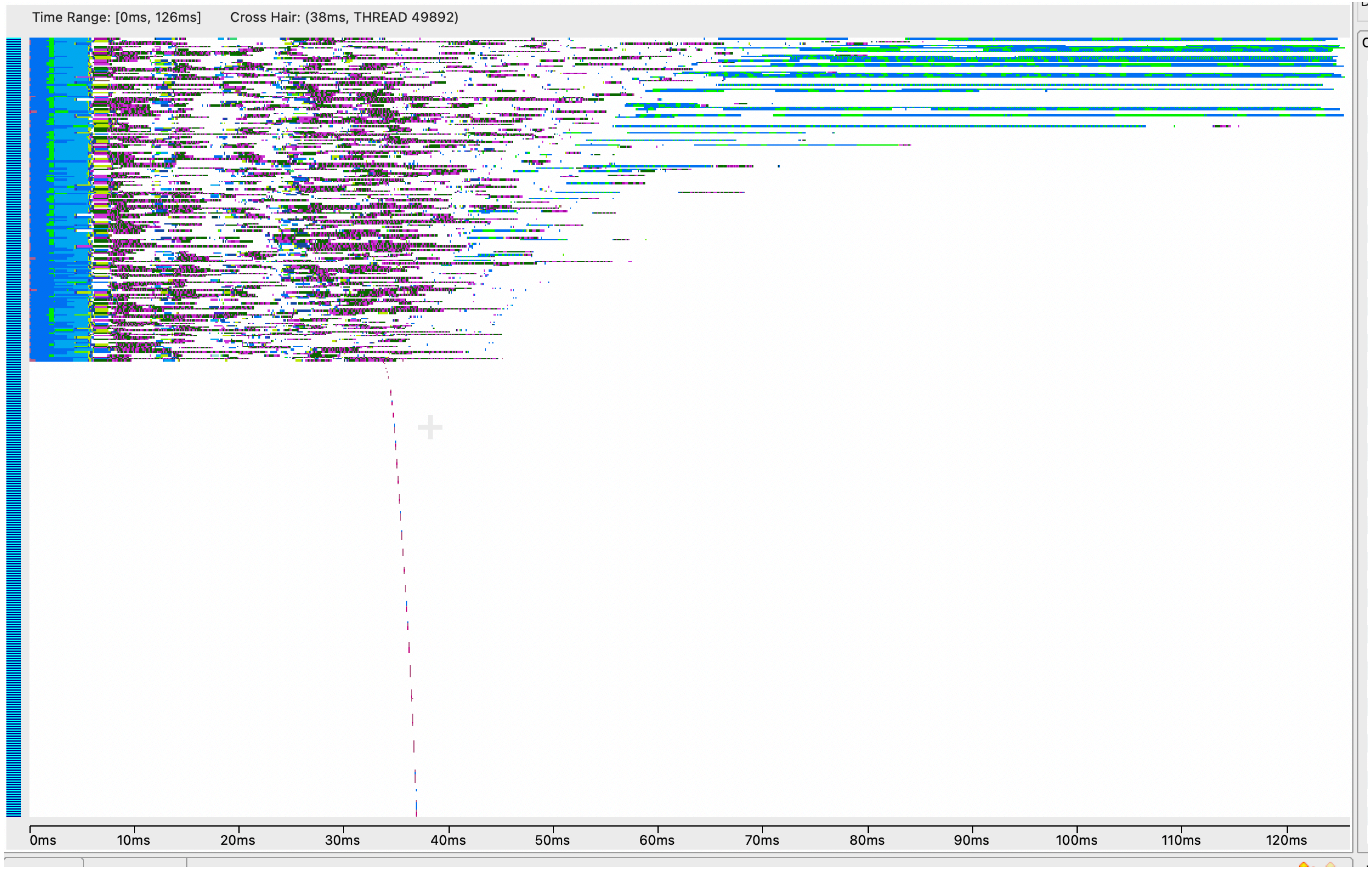
Work Stealing Among GPU Threads

- Each thread starts with its own task, and in each iteration
  - Step 1: works on its task
  - Step 2: decides if it should
    - give another thread its task
    - get another thread's task
    - exit
    - nothing
  - Step 3: behaves according to the decision from Step 2
  - Step 4: __syncwarp()

Only if it is NOT trying to get or give

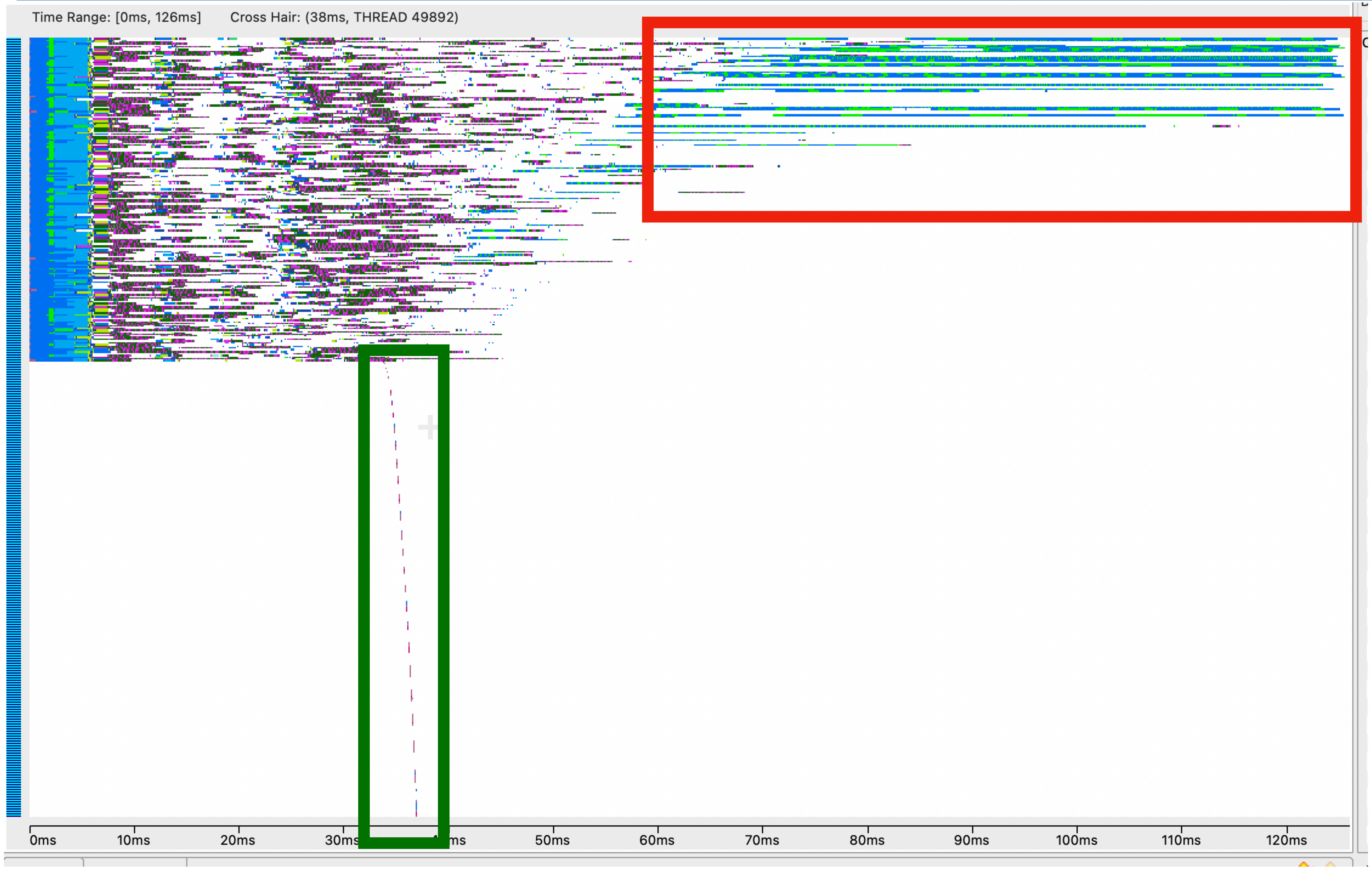
Only try one time of get( ) or give( ) in each iteration

Work Stealing Among GPU Threads - Quicksilver



- Successfully compacted tasks

Work Stealing Among GPU Threads - Quicksilver



- Successfully compacted tasks
- Threads started late find no work to do and just exit
- Threads keep trying to get( ) when only small amount of task remained

Outline

- Our trace tool
- Potential optimization opportunities of Quicksilver found with the trace tool
- Exploration of an optimization opportunity
- Work stealing algorithm among GPU threads
- **Summary**

Summary

- Binary instrumentation of GPU device functions within complex kernels reveals potential optimization opportunities
- Compacting threads with work into small amount of active blocks improves the performance
- A study of Quicksilver revealed two opportunities for optimizations
    - Imbalance between tasks
    - Imbalance between cases within each iteration
- Work stealing algorithm on GPU threads seems promising to accelerate the kernel