ECRTS
OSPERT 2025

Compute Kernels as Moldable Tasks

Towards Real-Time Gang Scheduling in GPUs

Attilio Discepoli Mathias Huygen Antonio Paolillo

VUB SOFTWARE LANGUAGES LAB

GPUs

- Massive parallelism
- Are everywhere

Example of GPU code

```
__device__ compute_edges(Image P) {
  // Compute the edges of the image
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  // -- snip --
}

while (active) {
  P = get_picture();
  compute_edges<<<N>>>(P);
  wait period(30ms);
}
```

Run on GPU

Run on CPU

Unpredictability

Not really suitable for real-time systems
where deadlines are important

Reason: lack of fine-grained control over GPU resource scheduling (kernel)

The CUDA default execution model is focused on performance and not deadlines

Even with the compute capabilities of the GPU, deadlines can still be missed due to poor prioritization

Question

What if we model GPU kernels as jobs with deadlines?

CUDA executes all the kernels in FIFO order, giving all the resources

➡ All-out scheduling policy

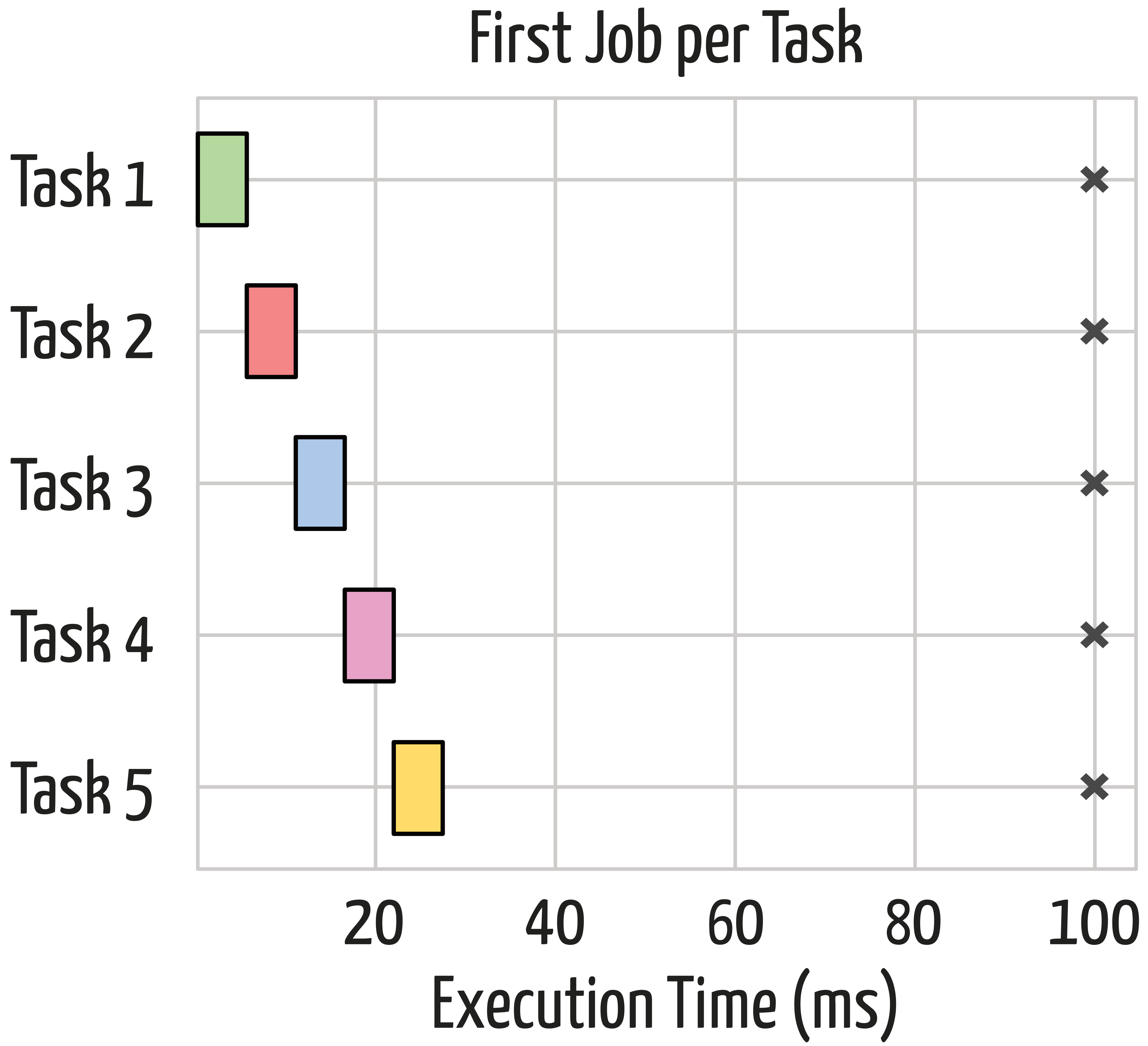
« All-out » because it takes all the GPU resources available

All-out scheduling policy

The CUDA default execution model

Example with 5 tasks (D=100ms)

No problem when the tasks have ample time-to-deadline



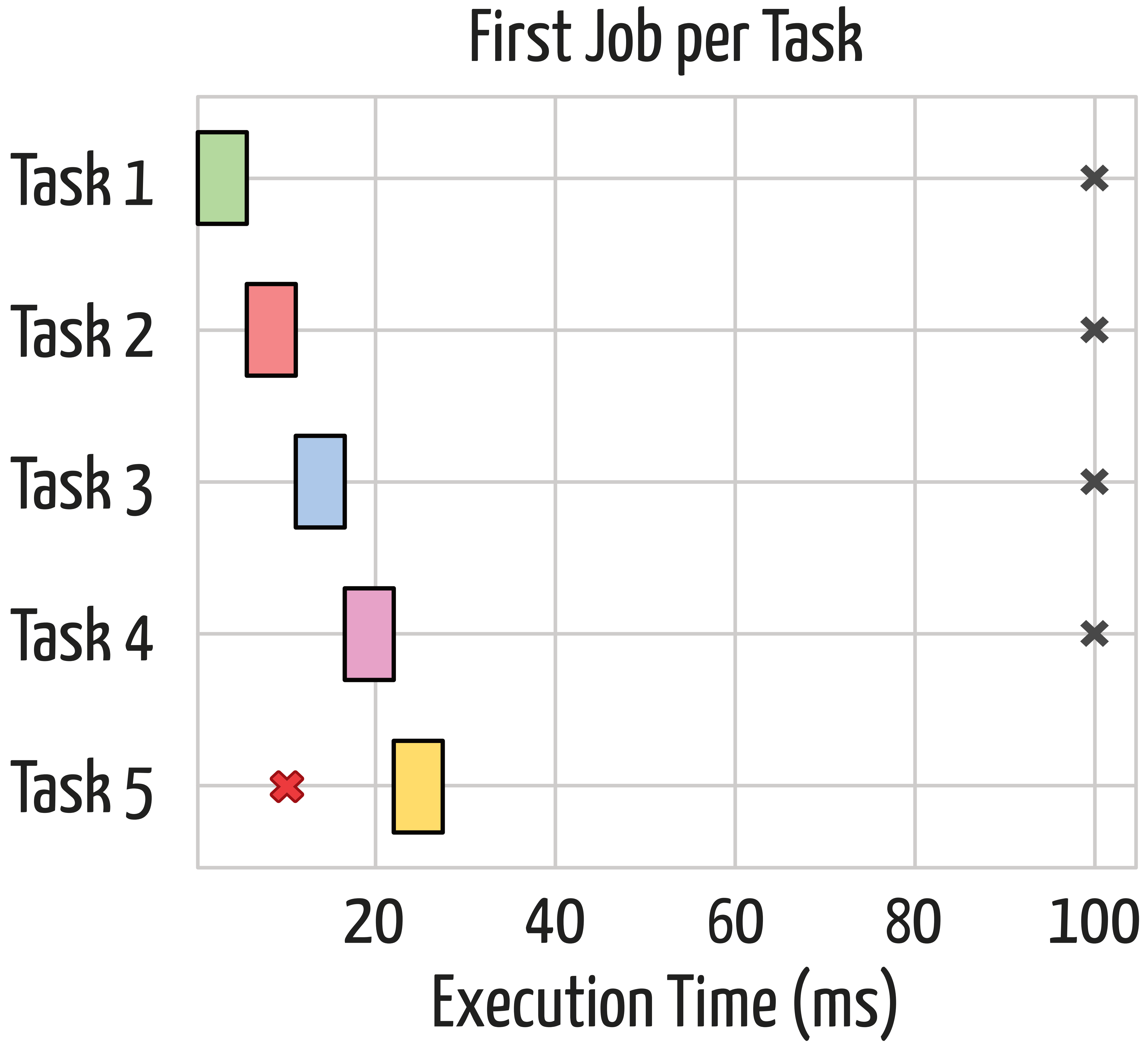
All-out scheduling policy

But when a task has a higher priority ⇒ deadline missed

Example with

4 tasks (D=100ms)

1 urgent task (D=10ms)



As fast as possible is not real-time

What if we use a real-time scheduling algorithm?

We will use EDF (Earliest Deadline First)

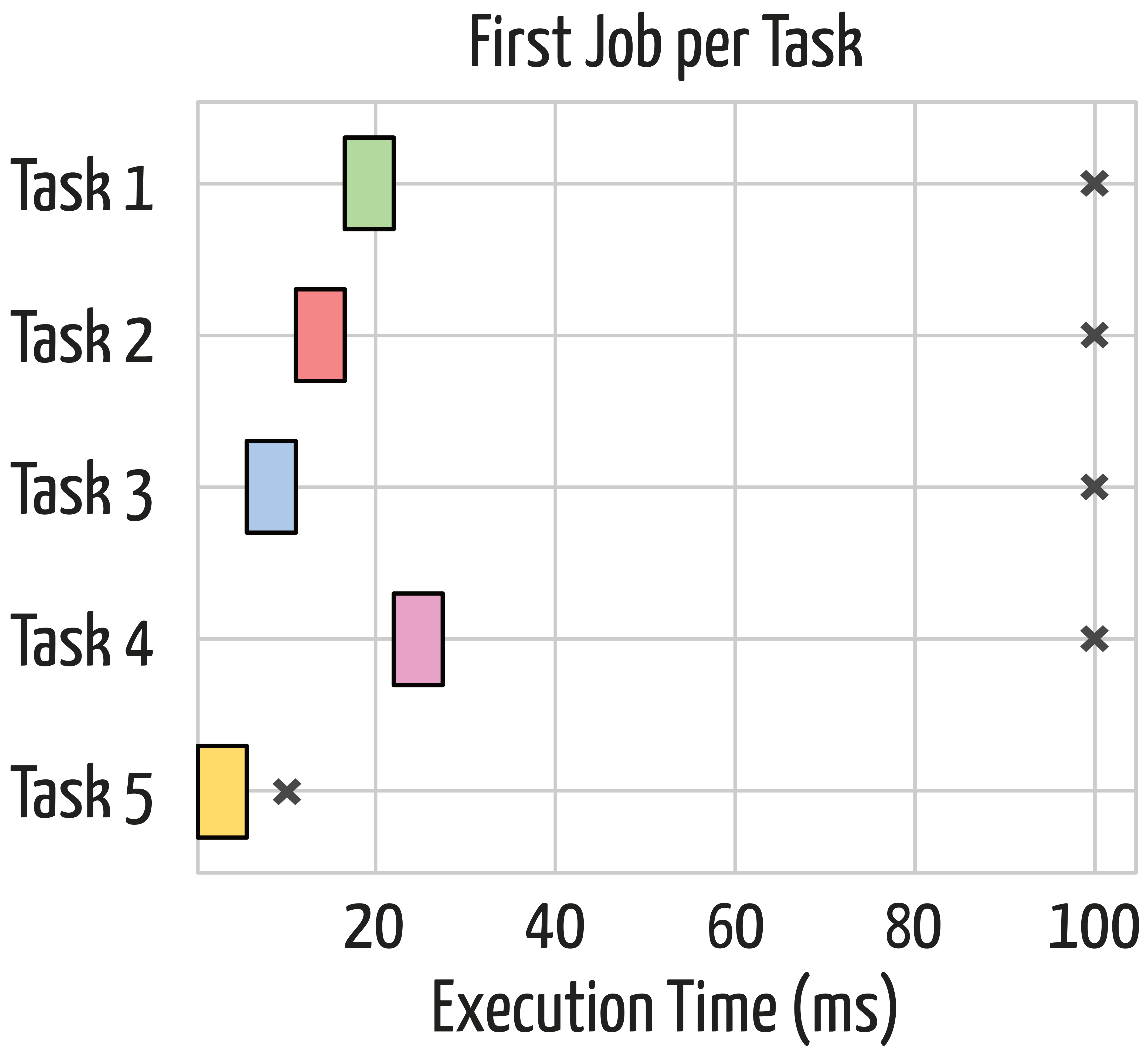
Sequential-EDF scheduling policy

Scheduler is now deadline-aware

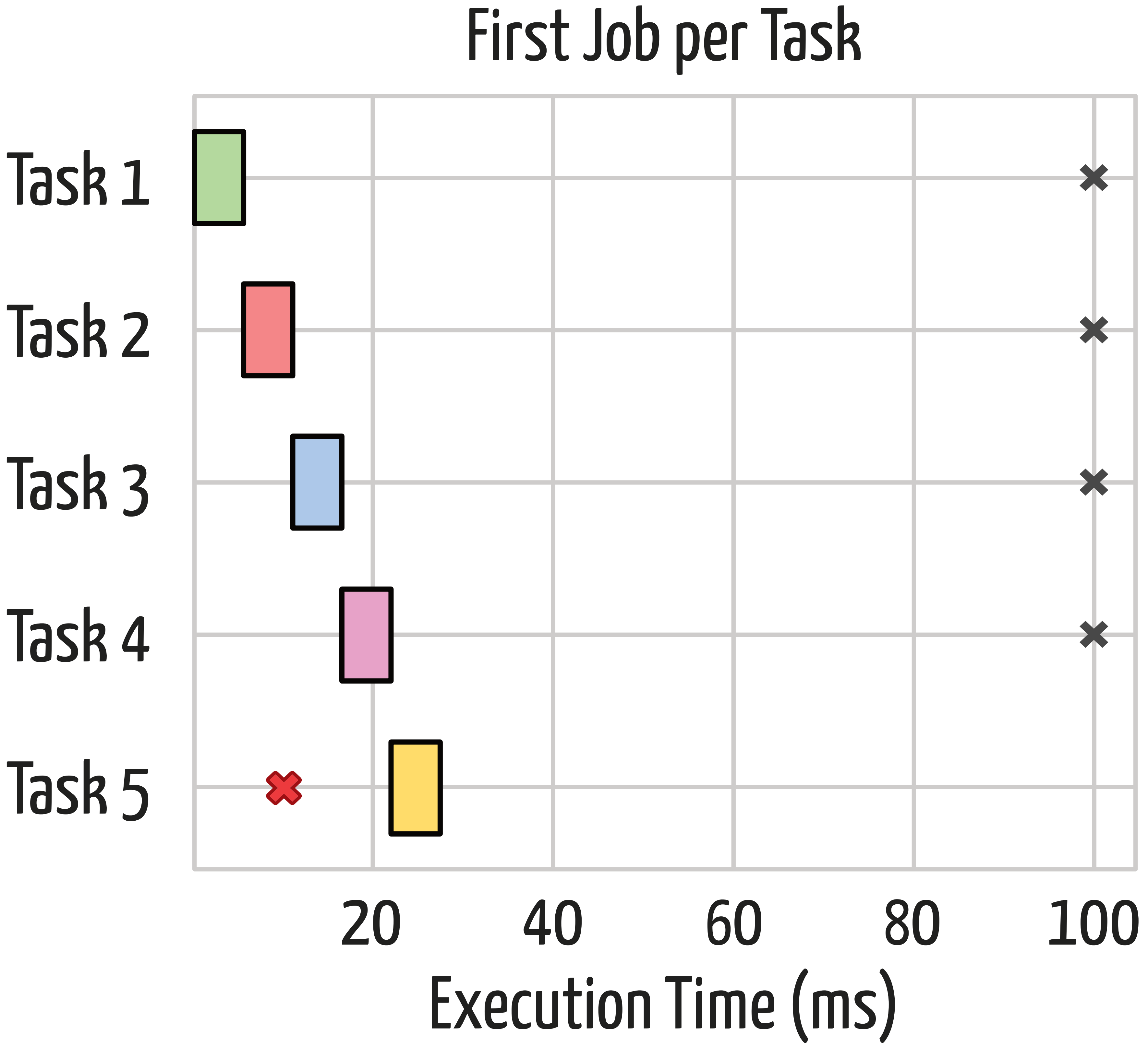
Example with (Same as the second all-out exp.)

- 4 tasks (D=100ms)
- 1 urgent task (D=10ms)

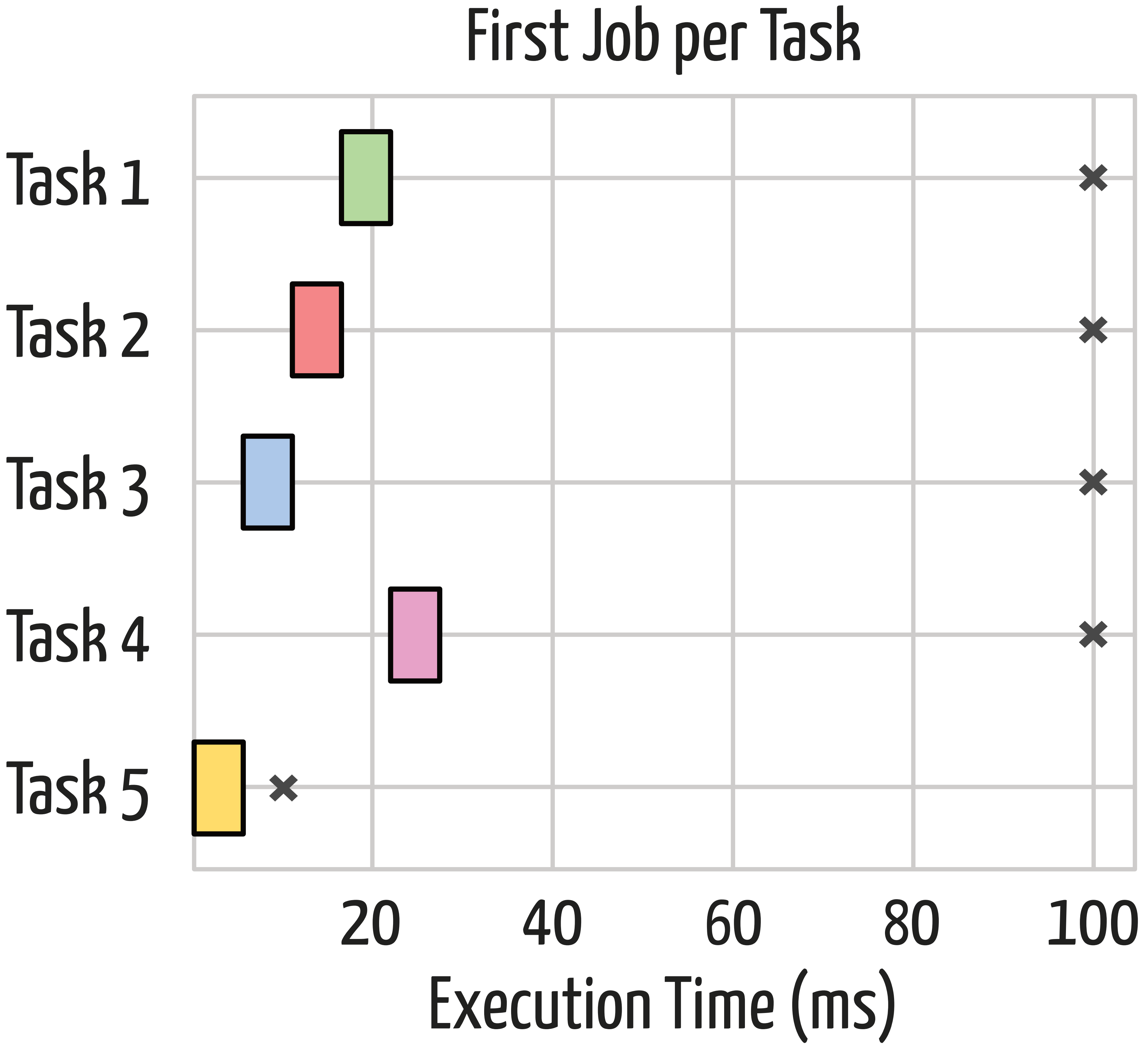
The highest priority task is now scheduled correctly



Recap: All-Out vs Sequential-EDF



✖ All-Out



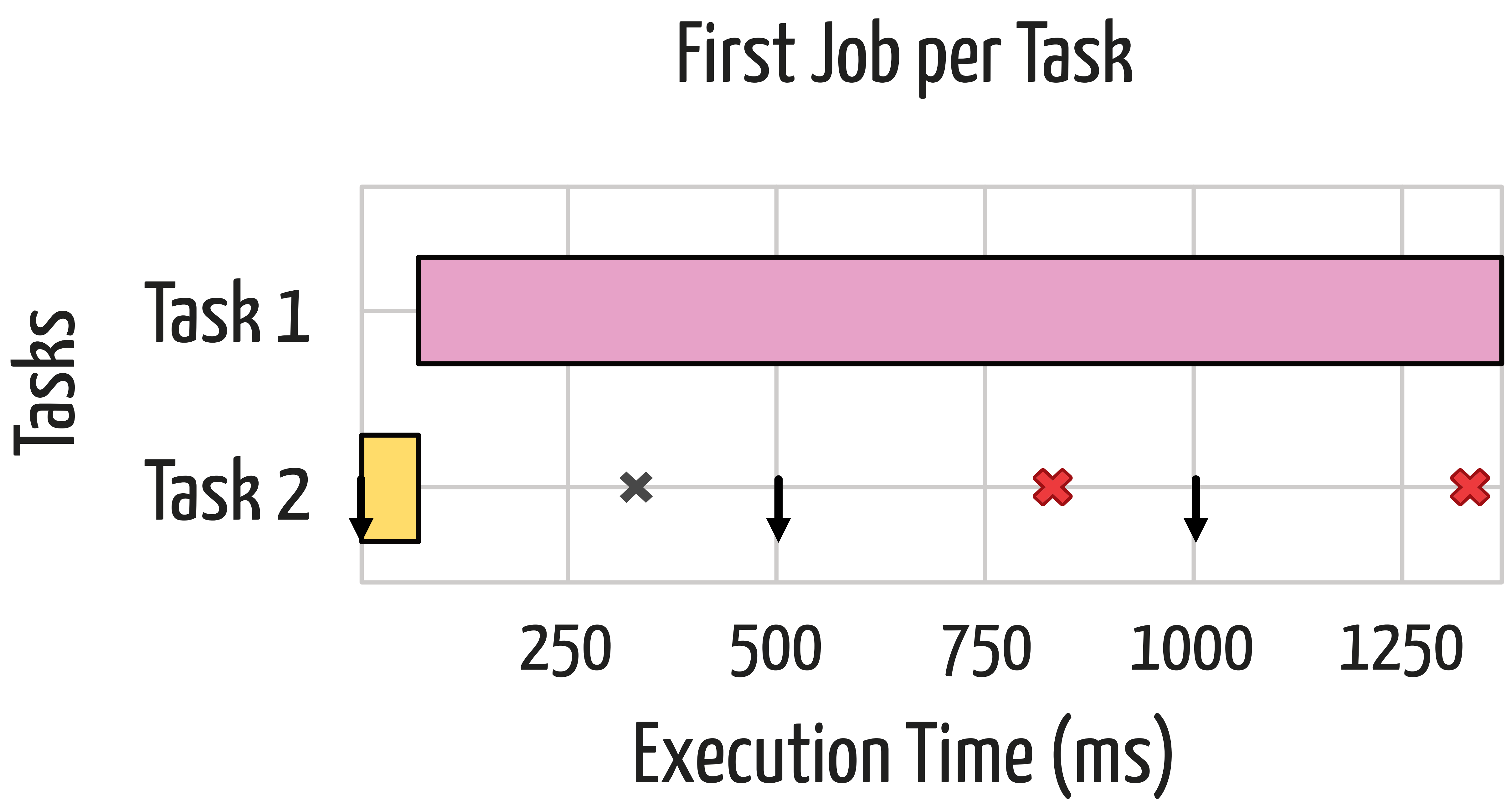
✔ Sequential-EDF

What if we insert a « greedy » task in the system ?

Sequential-EDF scheduling policy

Example with

- 1 greedy task (D=10s)
- 1 urgent task (D=300ms, T=500ms)



N.B. : Tasks are non-preemptive

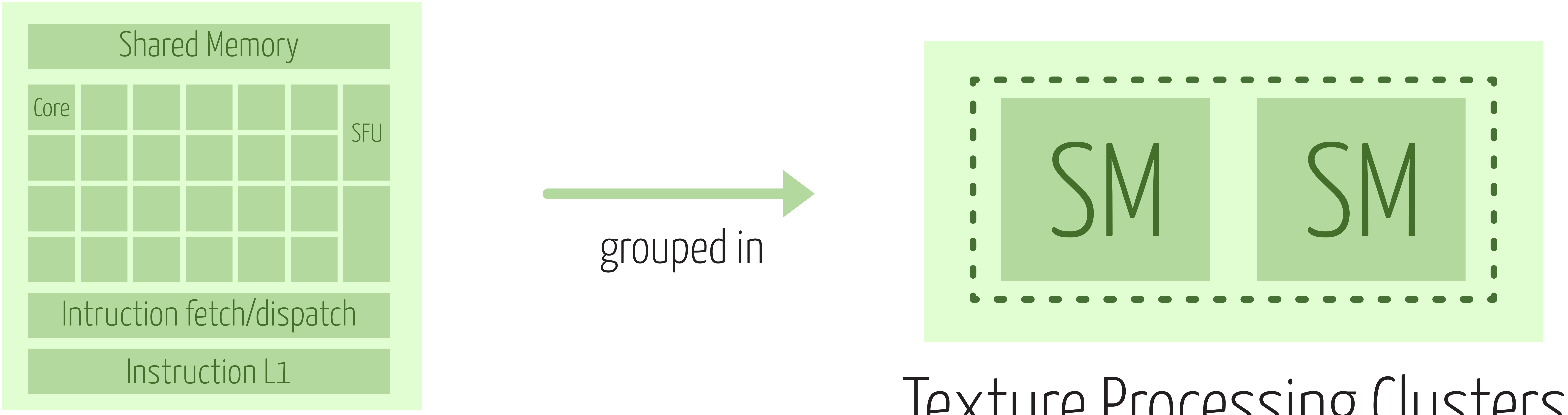
The greedy task takes all the resources

The scheduler uses the GPU as a single core machine

Need to find a way of limiting a task so that it uses only the amount of resources required to meet its deadline

What about leveraging partitioning ?

Intermezzo: Anatomy of a GPU



Streaming Multiprocessor
(SM)

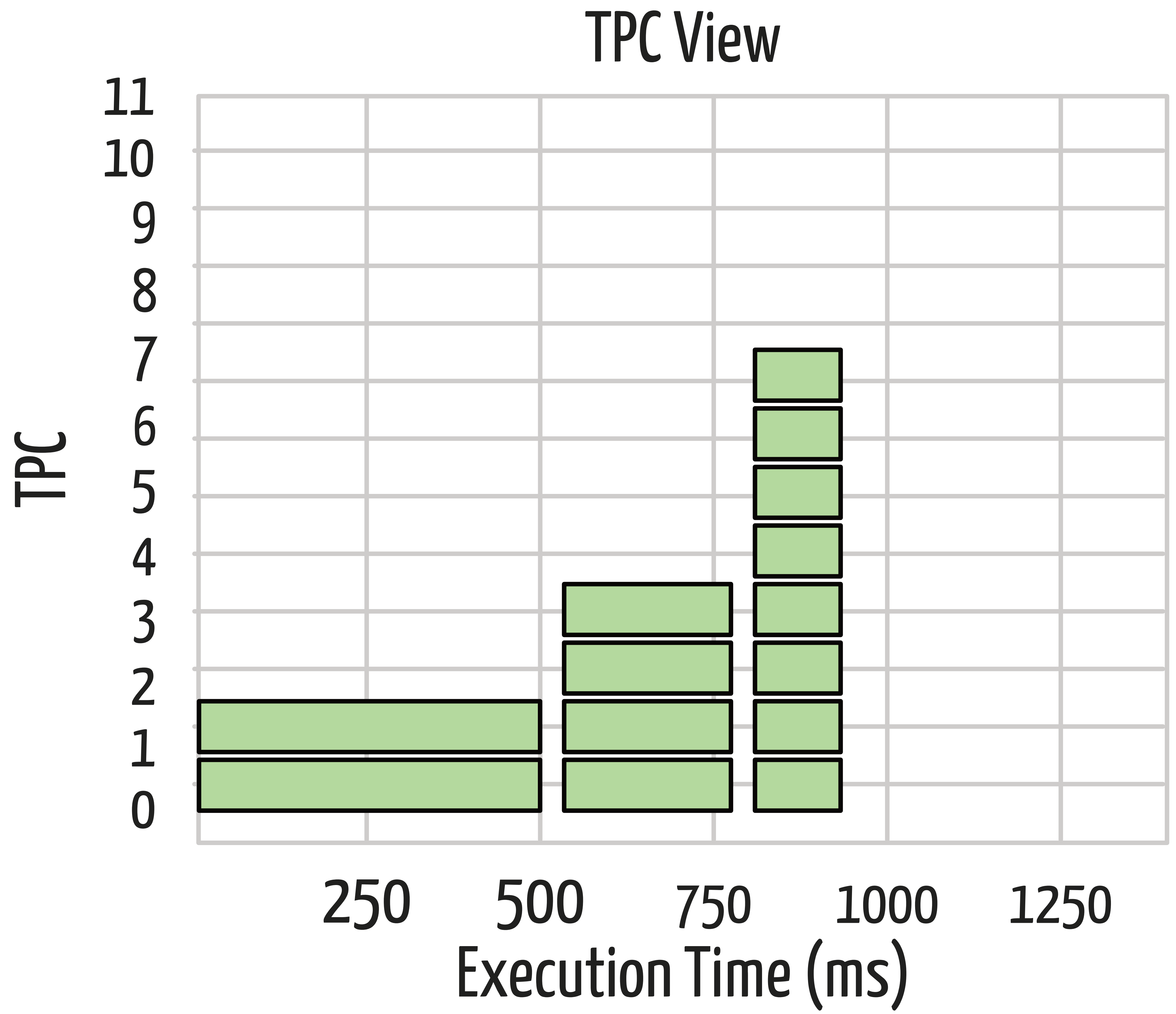
Texture Processing Clusters
(TPC)

We can use the TPCs as processors in a multiprocessor system and partition them among tasks

Intermezzo: Partitioning of GPU

The `libsmctrl`* enables to select on which TPCs a kernel is executed

A task will not consume the same execution time depending on the number of TPCs used



Same task run on various TPCs count

*[Bakita RTAS23]

Gang scheduling terminology [Goossens 2010]

“A job is said to be

rigid if its processor allocation is fixed externally and never changes,

moldable if the scheduler decides the allocation at release time, and

malleable if the allocation can change during execution”

But how to achieve « processor allocation » in GPUs?

Gang scheduling terminology applied to GPUs

"A job is said to be

rigid if its **TPC** allocation is fixed externally and never changes,

moldable if the scheduler decides the **TPC** allocation at release time, and

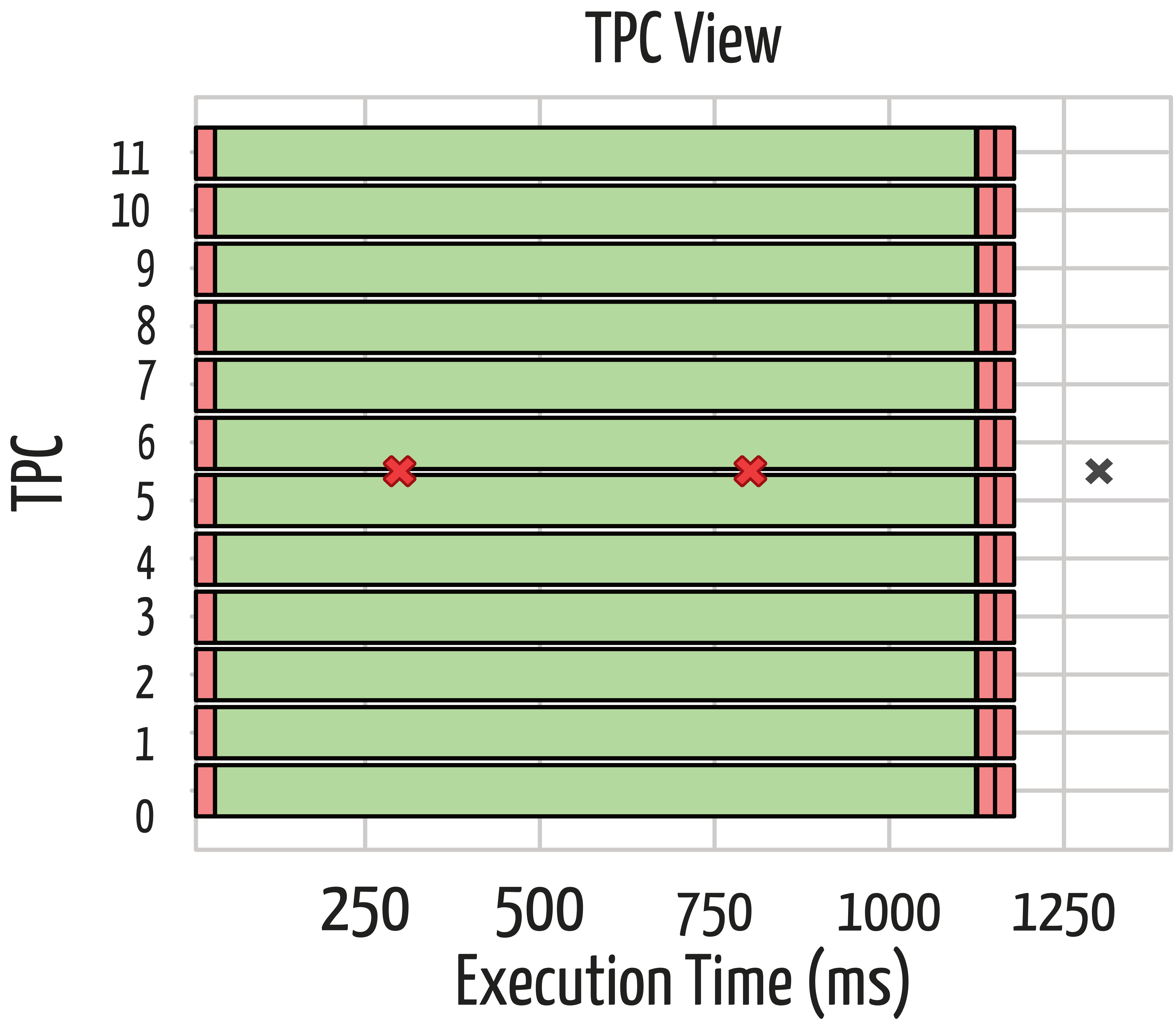
malleable if the **TPC** allocation can change during execution"

Sequential-EDF is rigid, we will introduce Moldable-EDF

Sequential-EDF scheduling policy

Like all-out, the sequential-EDF (rigid) gives all the computational resources to each task

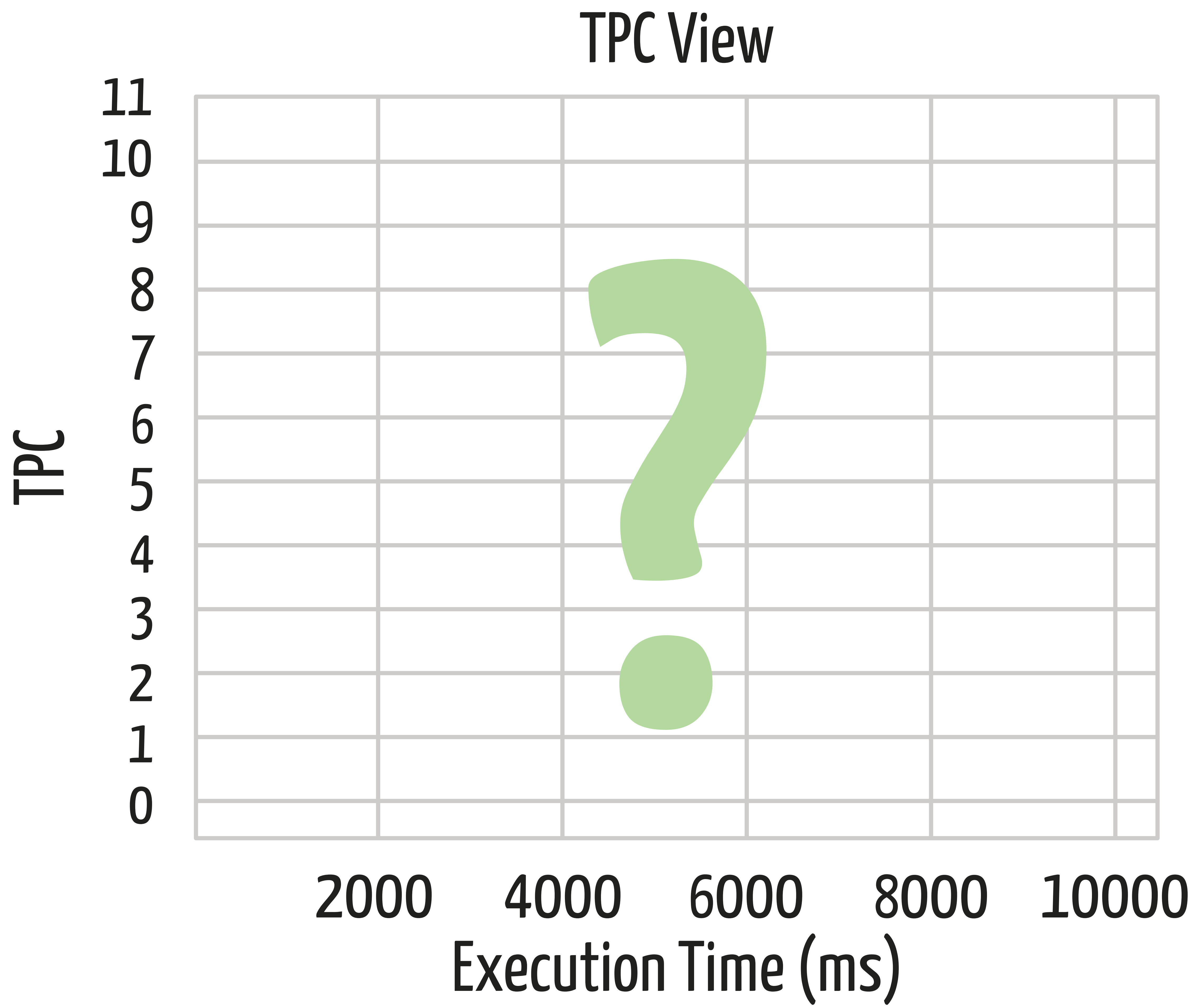
If there is a job with higher priority when another longer one is running ⇒ deadline missed



Idea: Moldable-EDF scheduling policy

A moldable EDF would only assign the needed amount of TPCs to finish the job

With a moldable policy, together with libsmctrl, the scheduler can decide the number of TPCs per kernel at job release time



How can we know how many TPCs are needed to respect the deadline?

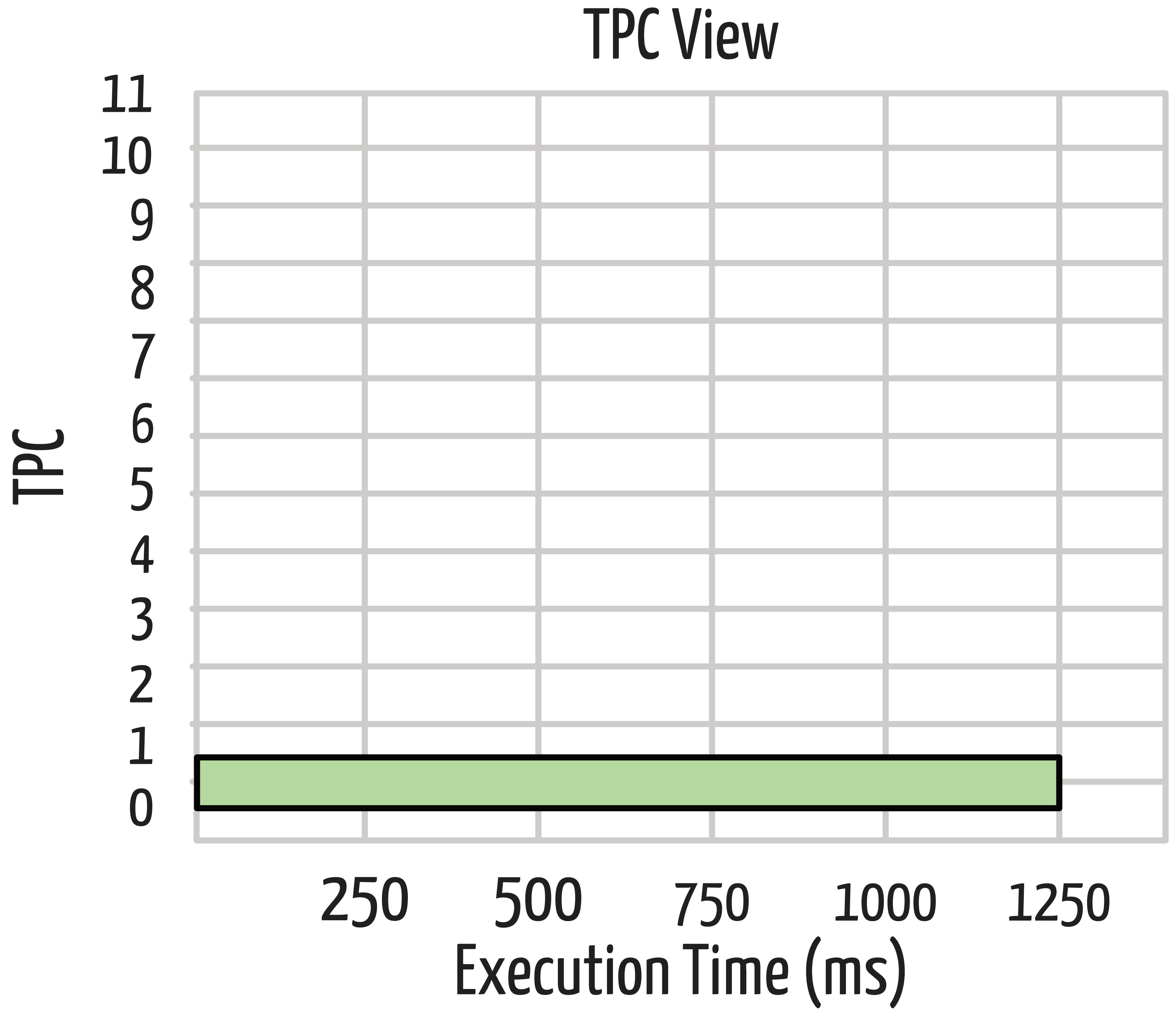
WCET Profiling

We empirically compute the kernel's WCET
on each number of TPC

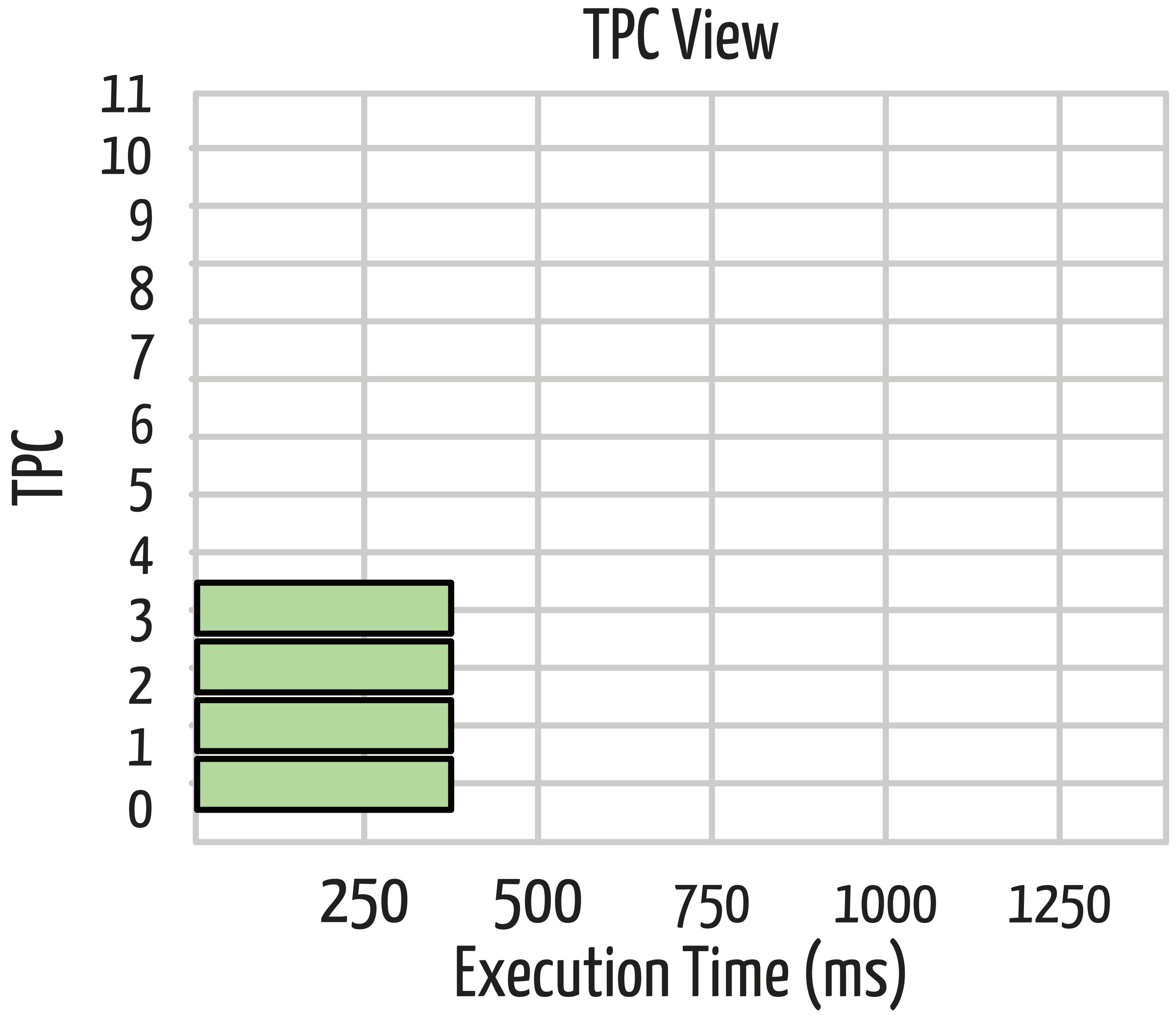
1. Warmup the kernel several times
2. Run the kernel several times, by using 1 to 12 TPCs
3. Record the worst observed execution time and use it as a reference in the scheduler

C_i^m WCET of jobs of task i when assigned m TPCs

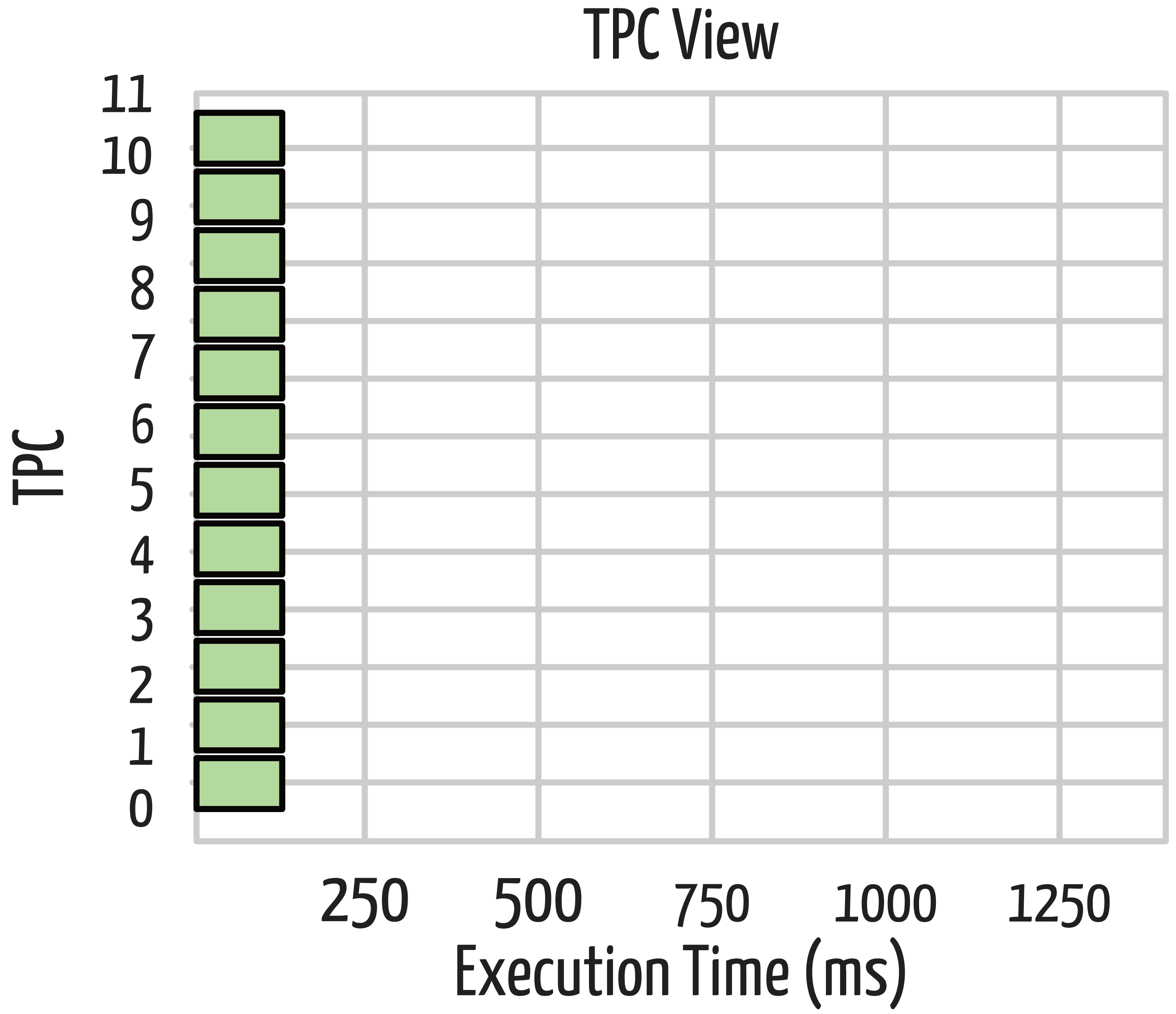
C_i^1



C_i^4



C_i^{12}



Now we have partitioning information
that can be used by our moldable scheduler
to know the number of TPCs needed to
execute the job correctly

Moldable-EDF scheduling policy

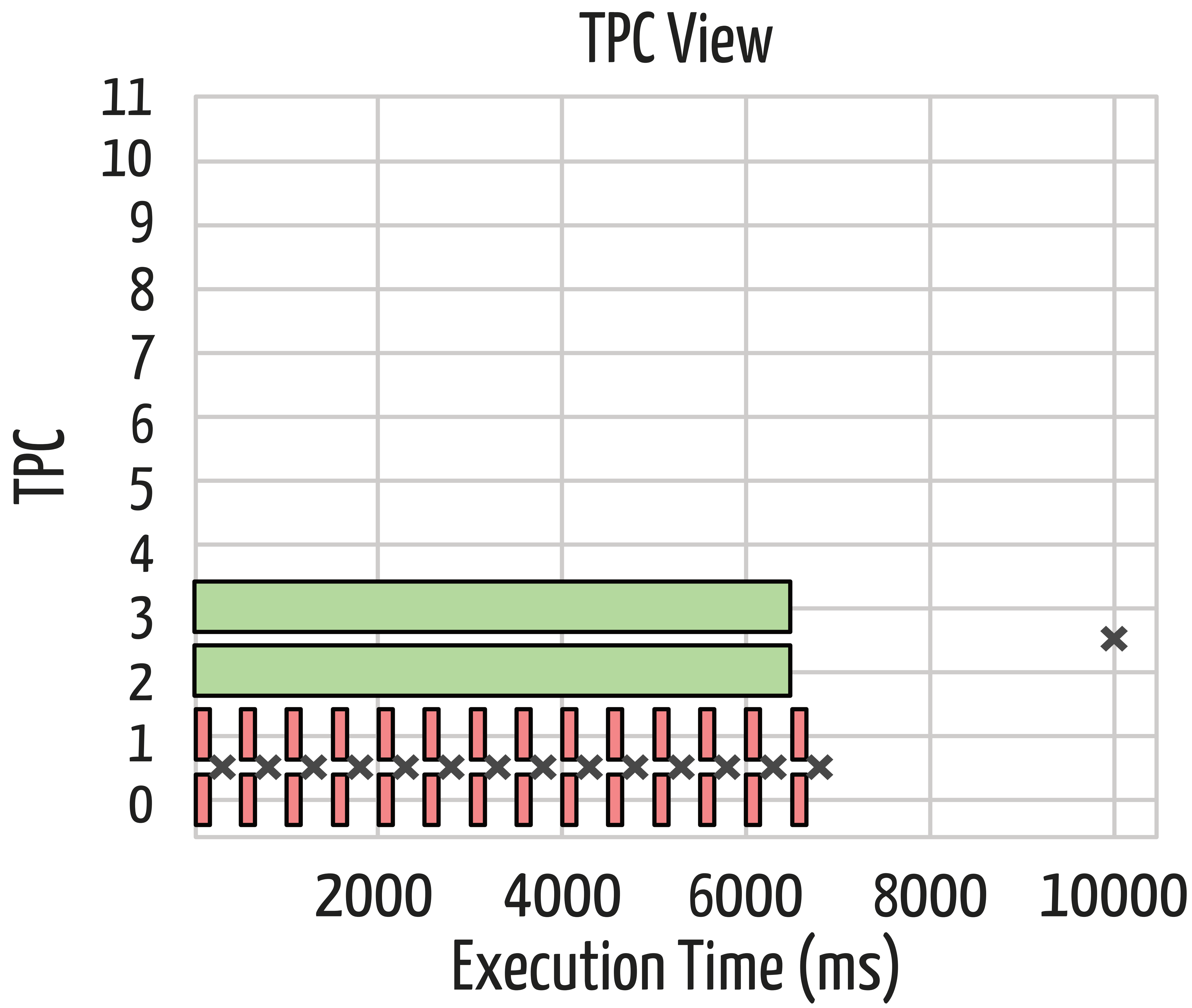
Scheduler is now deadline and resource-aware

Example with

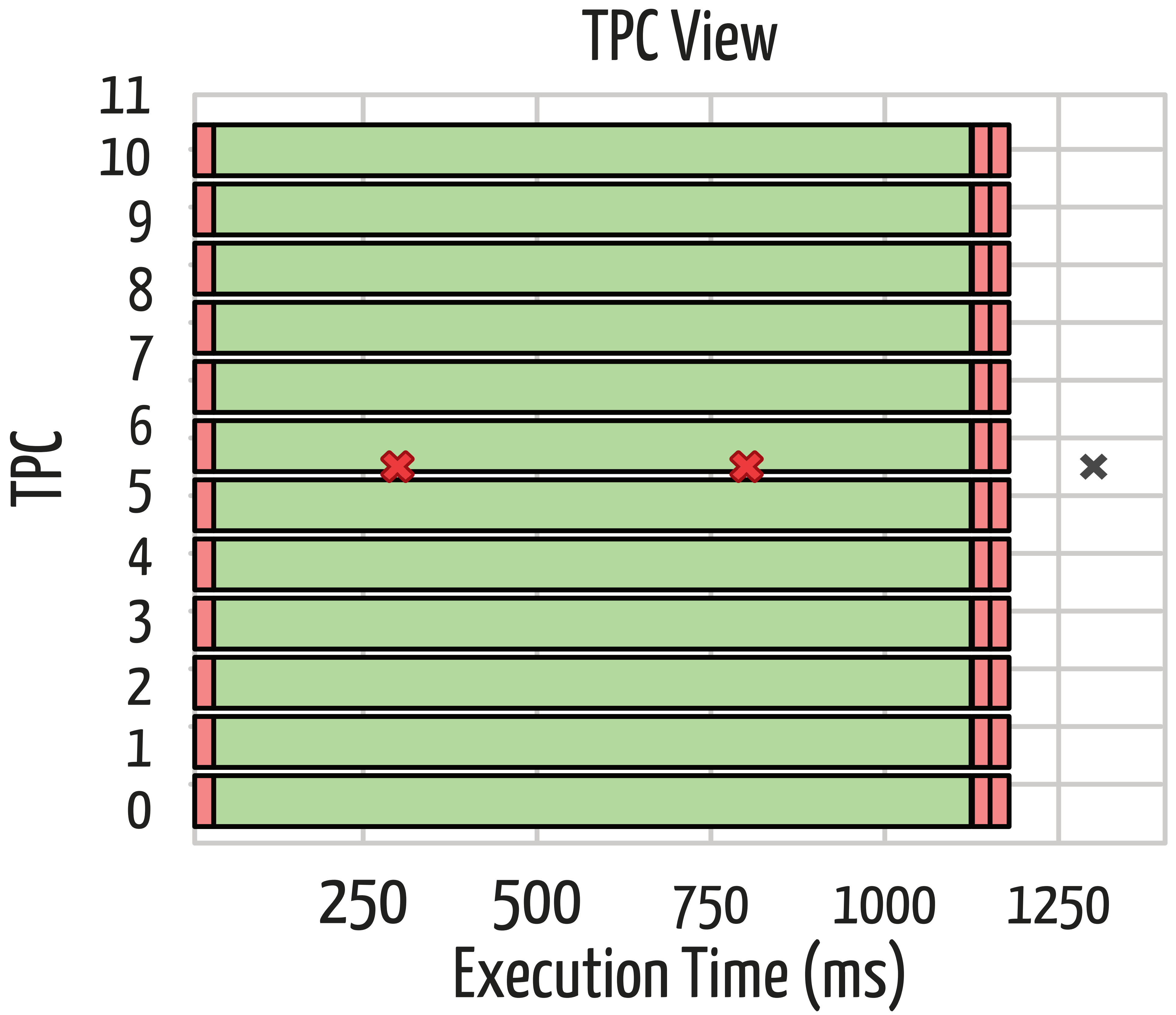
1 greedy task (D=10s)

1 urgent task (D=300ms, T=500ms)

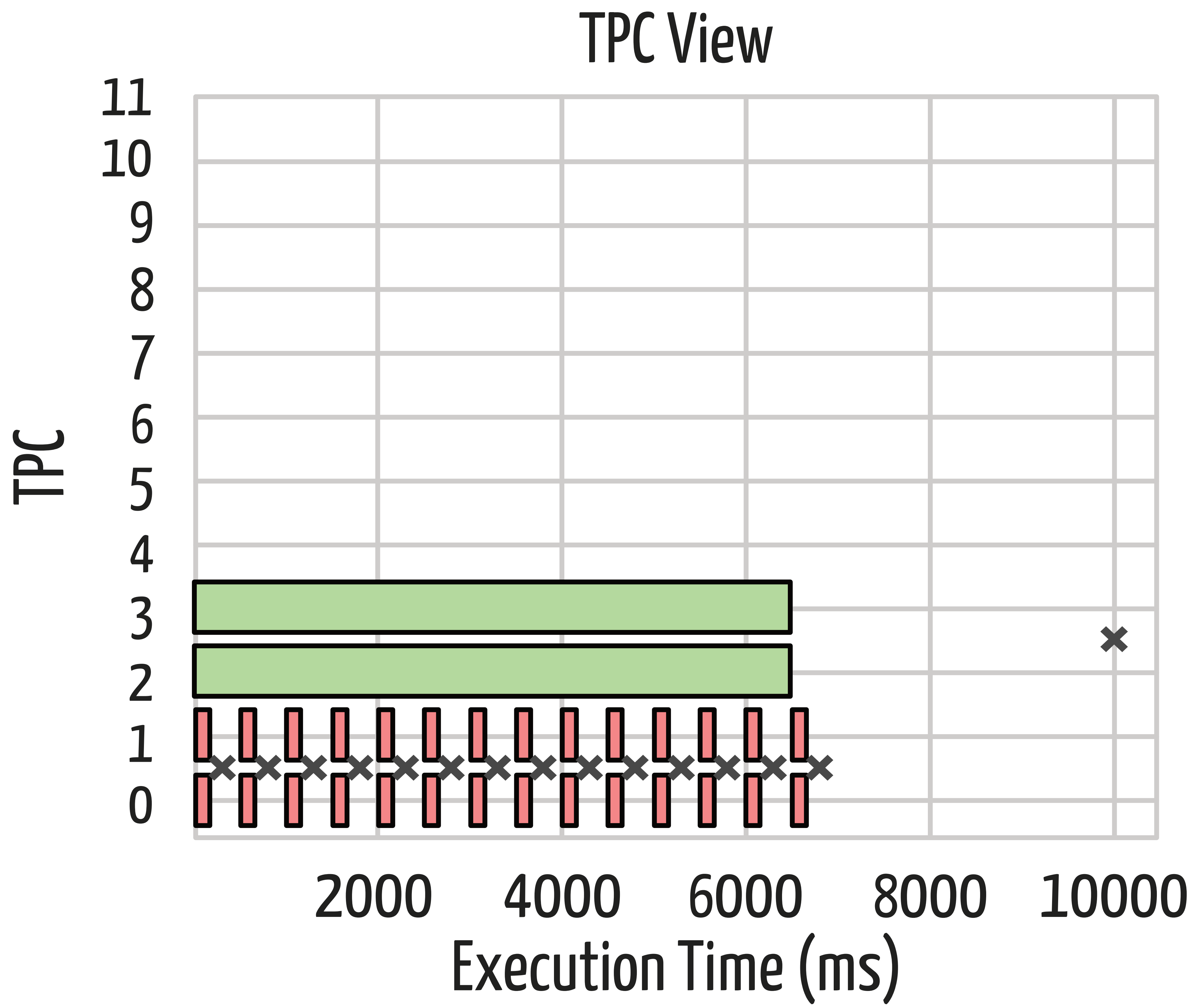
(Same as the second seq-edf exp.)



Recap: Sequential-EDF vs Moldable-EDF



✖ Sequential-EDF



✔ Moldable-EDF

Limitations

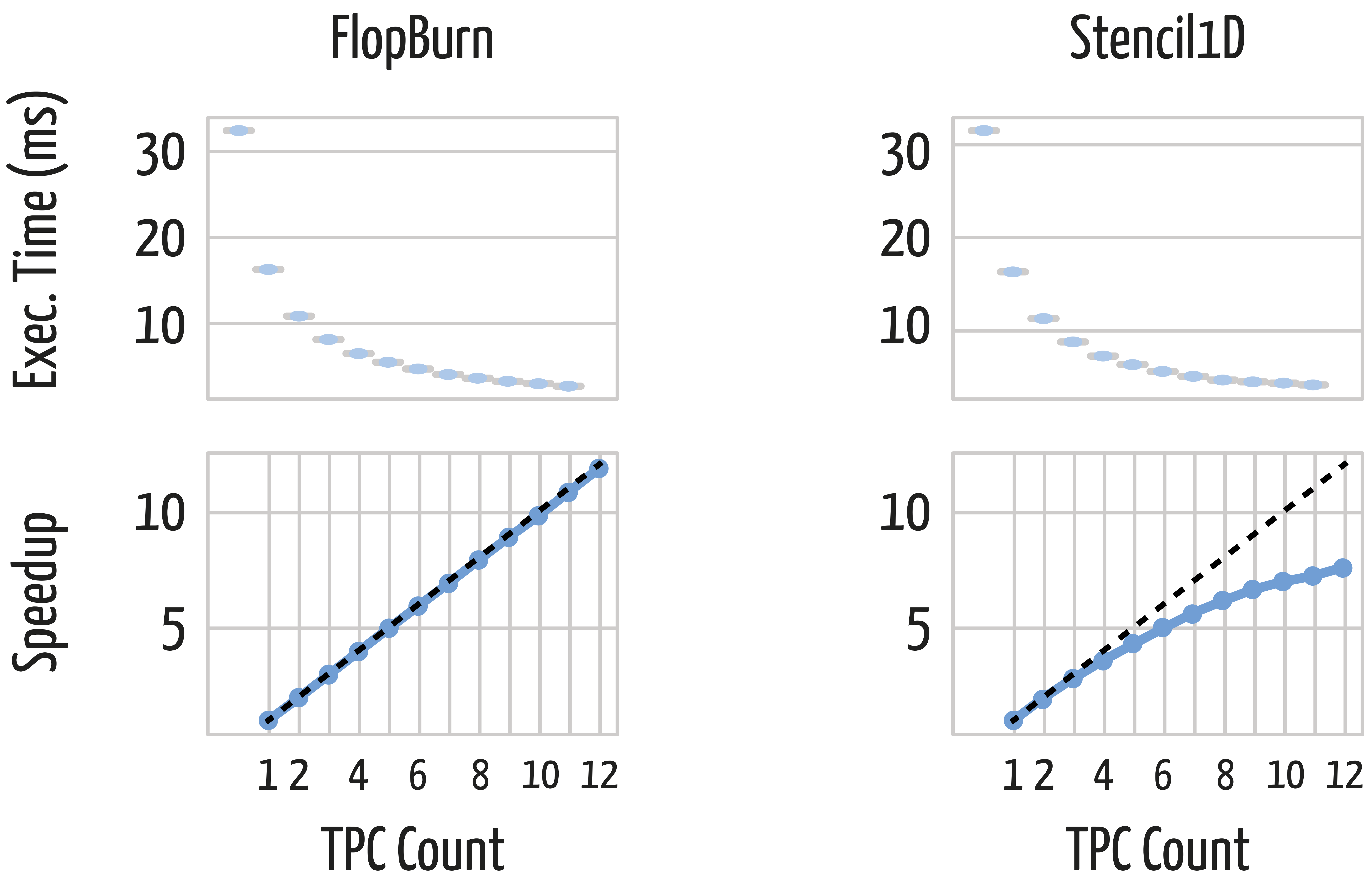
- Kernel executions are calibrated and evaluated in **isolation**
- We do not explore **alternative CUDA configurations** (e.g: static priority assignement)
- Evaluation is limited to **synthetic kernels** without complex memory accesses.

What's next?

- Use real-world kernels (AI, image processing, etc.) and study the memory model of NVIDIA GPUs more in depth (Copy engine)
- Explore the feasibility of malleable scheduling in GPUs
- Execute our experiments on NVIDIA Jetson Orin
- Analyze formally the scheduler

Thank you!

Questions?


FlopBurn
Stencil1D
Exec. Time (ms)
30
20
10
Speedup
10
5
1 2 4 6 8 10 12
TPC Count

Placement sensitivity