



Schedulers

Table of contents

Greedy Scheduler

Multithread Scheduler


Event-Based Scheduler

The Scheduler component is a critical part of the system responsible for governing the execution of operators in a graph by enforcing conditions associated with each operator. Its primary responsibility includes orchestrating the execution of all operators defined in the graph while keeping track of their execution states.

The Holoscan SDK offers multiple schedulers that can cater to various use cases. These schedulers are:

1. Greedy Scheduler: This basic single-threaded scheduler tests conditions in a greedy manner. It is suitable for simple use cases and provides predictable execution. However, it may not be ideal for large-scale applications as it may incur significant overhead in condition execution.
2. MultiThread Scheduler: The multithread scheduler is designed to handle complex execution patterns in large-scale applications. This scheduler consists of a dispatcher thread that monitors the status of each operator and dispatches it to a thread pool of worker threads responsible for executing them. Once execution is complete, worker threads enqueue the operator back on the dispatch queue. The multithread scheduler offers superior performance and scalability over the greedy scheduler.
3. Event-Based Scheduler: The event-based scheduler is also a multi-thread scheduler, but as the name indicates it is event-based rather than polling based. Instead of having a thread that constantly polls for the execution readiness of each operator, it instead waits for an event to be received which indicates that an operator is ready to execute. The event-based scheduler will have a lower latency than using the multi-thread scheduler with a long polling interval (`check_recession_period_ms`), but without the high CPU usage seen for a multi-thread scheduler with a very short polling interval.

It is essential to select the appropriate scheduler for the use case at hand to ensure optimal performance and efficient resource utilization. Since most parameters of the schedulers overlap, it is easy to switch between them to test which may be most performant for a given application.

 **Note**

Detailed APIs can be found here: [C++/](#)

```
<a  
href="../api/python/holoscan_python_api_schedulers.html#module-  
holoscan.schedulers">Python</a>
```

).

Greedy Scheduler

The greedy scheduler has a few parameters that the user can configure.

- The [clock](#) used by the scheduler can be set to either a `realtime` or `manual` clock.
 - The realtime clock is what should be used for applications as it pauses execution as needed to respect user specified conditions (e.g. operators with periodic conditions will wait the requested period before executing again).
 - The manual clock is of benefit mainly for testing purposes as it causes operators to run in a time-compressed fashion (e.g. periodic conditions are not respected and operators run in immediate succession).
- The user can specify a `max_duration_ms` that will cause execution of the application to terminate after a specified maximum duration. The default value of `-1` (or any other negative value) will result in no maximum duration being applied.
- This scheduler also has a boolean parameter, `stop_on_deadlock` that controls whether the application will terminate if a deadlock occurs. A deadlock occurs when all operators are in a `WAIT` state, but there is no periodic condition pending to break out of this state. This parameter is `true` by default.
- When setting the `stop_on_deadlock_timeout` parameter, the scheduler will wait this amount of time (in ms) before determining that it is in deadlock and should stop. It will reset if a job comes in during the wait. A negative value means no stop on deadlock. This parameter only applies when `stop_on_deadlock=true`.

Multithread Scheduler

The multithread scheduler has several parameters that the user can configure. These are a superset of the parameters available for the `GreedyScheduler` (described in the section above). Only the parameters unique to the multithread scheduler are described here. The multi-thread scheduler uses a dedicated thread to poll the status of operators and schedule any that are ready to execute. This will lead to high CPU usage by this polling thread when `check_recession_period_ms` is close to 0.

- The number of worker threads used by the scheduler can be set via `worker_thread_number`, which defaults to `1`. This should be set based on a consideration of both the workflow and the available hardware. For example, the topology of the computation graph will determine how many operators it may be possible to run in parallel. Some operators may potentially launch multiple threads internally, so some amount of performance profiling may be required to determine optimal parameters for a given workflow.
- The value of `check_recession_period_ms` controls how long the scheduler will sleep before checking a given condition again. In other words, this is the polling interval for operators that are in a `WAIT` state. The default value for this parameter is `5` ms.

Event-Based Scheduler

The event-based scheduler is also a multi-thread scheduler, but it is event-based rather than polling based. As such, there is no `check_recession_period_ms` parameter, and this scheduler will not have the high CPU usage that can occur when polling at a short interval. Instead, the scheduler only wakes up when an event is received indicating that an operator is ready to execute. The parameters of this scheduler are a superset of the parameters available for the `GreedyScheduler` (described above). Only the parameters unique to the event-based scheduler are described here.

- The number of worker threads used by the scheduler can be set via `worker_thread_number`, which defaults to `1`. This should be set based on a consideration of both the workflow and the available hardware. For example, the topology of the computation graph will determine how many operators it may be possible to run in parallel. Some operators may potentially launch multiple threads internally, so some amount of performance profiling may be required to determine optimal parameters for a given workflow.

© Copyright 2022-2024, NVIDIA.. PDF Generated on 06/06/2024