The 11th International workshop on OpenCL and SYCL

IWOCL & SYCLcon 2023

Stellar Mergers with HPX-Kokkos and SYCL: Methods of using an Asynchronous Many-Task Runtime System with SYCL

Gregor Daiß, University of Stuttgart

Patrick Diehl, Hartmut Kaiser and Dirk Pflüger

HPX and SYCL

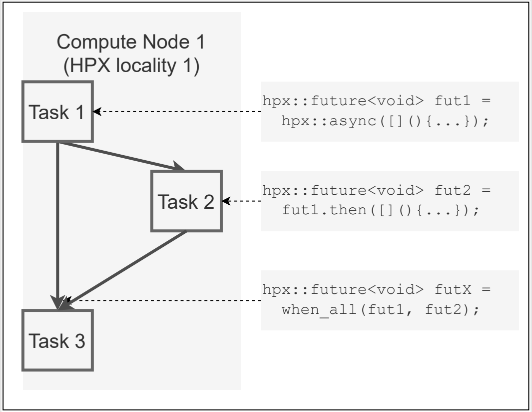
What is HPX?

- **Asynchronous, Distributed Many-Task Runtime System**

HPX and SYCL

What is HPX?

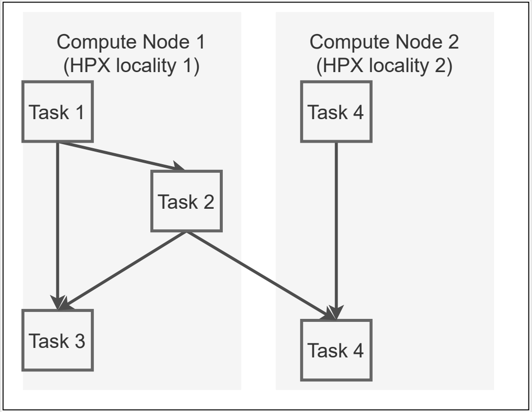
- **Asynchronous, Distributed Many-Task Runtime System**
- **Asynchronous**: Build task graph using futures and continuations (`then`, `when_all`)



HPX and SYCL

What is HPX?

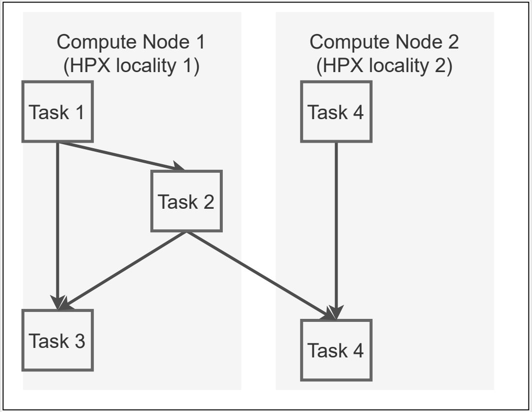
- **Asynchronous, Distributed Many-Task Runtime System**
- **Asynchronous**: Build task graph using futures and continuations (`then`, `when_all`)
- **Distributed**: Task graph across compute nodes (remote function calls , HPX channels, multiple backends available)



HPX and SYCL

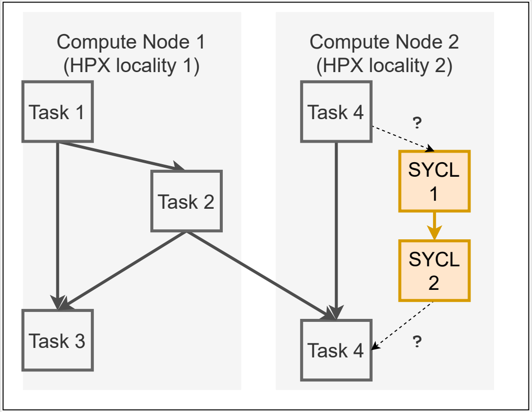
What is HPX?

- **Asynchronous, Distributed Many-Task Runtime System**
- **Asynchronous**: Build task graph using futures and continuations (`then`, `when_all`)
- **Distributed**: Task graph across compute nodes (remote function calls , HPX channels, multiple backends available)
- **Many Tasks**: Few HPX worker threads (one per core) working on millions of lightweight (suspendable) HPX tasks



HPX and SYCL

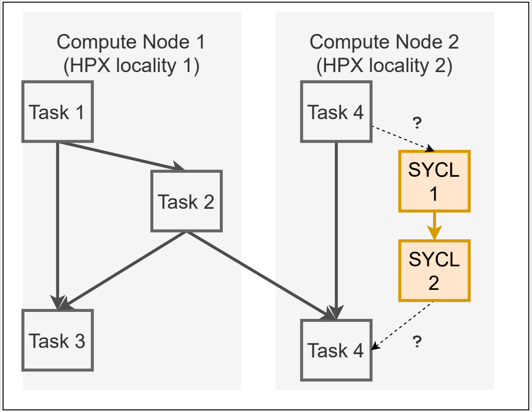
Why combine HPX with SYCL?



HPX and SYCL

Why combine HPX with SYCL?

- **More choices**: SYCL for HPX applications, HPX for distributed SYCL applications (instead of MPI)
- **Better integrations**: Better integration of HPX with other libraries that use SYCL (Kokkos)
- **More efficiency**: Complement strengths



HPX and SYCL

Why combine HPX with SYCL?

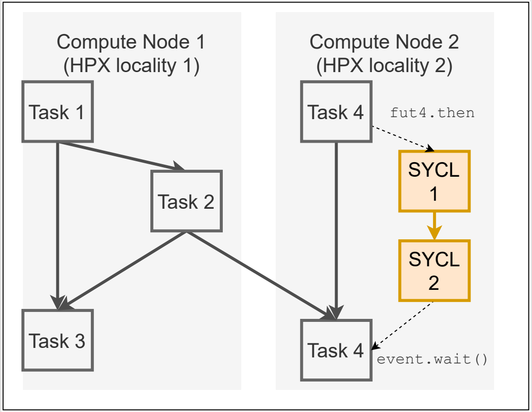
- **More choices**: SYCL for HPX applications, HPX for distributed SYCL applications (instead of MPI)
- **Better integrations**: Better integration of HPX with other libraries that use SYCL (Kokkos)
- **More efficiency**: Complement strengths

How to combine HPX with SYCL?

- The problem: Integrate task-graphs asynchronously and efficiently

  → **No active waiting** (no `event.wait()`) Avoid barriers / blocking of worker threads

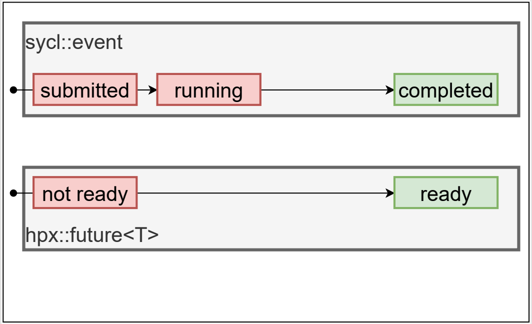
  → **Overhead**?



HPX-SYCL Integration Basics

How to combine HPX with SYCL?

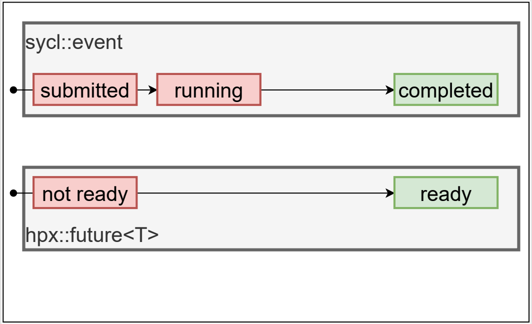
- **We have**: SYCL events to check if asynchronous SYCL actions are done
- **We need**: HPX futures to check if asynchronous SYCL actions are done
- Get an HPX future from a SYCL event without actively waiting or blocking the thread



HPX-SYCL Integration Basics

How to combine HPX with SYCL?

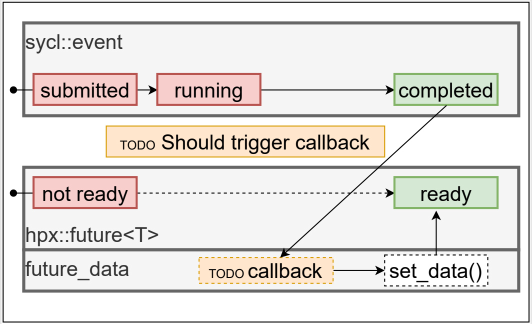
- **We have**: SYCL events to check if asynchronous SYCL actions are done
- **We need**: HPX futures to check if asynchronous SYCL actions are done
- Get an HPX future from a SYCL event without actively waiting or blocking the thread
- HPX scheduler takes care of the rest (triggering continuations)



HPX-SYCL Integration Basics

How to combine HPX with SYCL?

- **We have**: SYCL events to check if asynchronous SYCL actions are done
- **We need**: HPX futures to check if asynchronous SYCL actions are done
- Get an HPX future from a SYCL event without actively waiting or blocking the thread
- HPX scheduler takes care of the rest (triggering continuations)



TODOs for integration

- Add specialization for HPX future_data
- Add callback mechanism that is called when the SYCL event is completed
- Use it to set the future to ready

HPX-SYCL Integration Variant 1: Using SYCL `host_task`s

Use SYCL `host_task`s as callback mechanism

HPX-SYCL Integration Variant 1: Using SYCL `host_tasks`

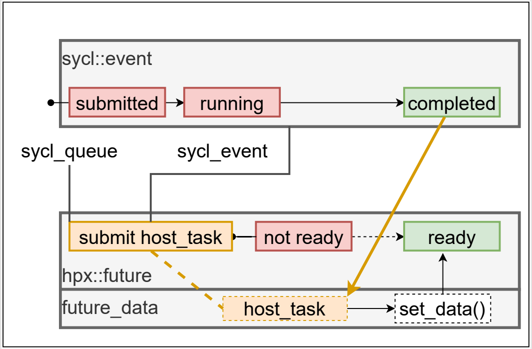
Use SYCL `host_tasks` as callback mechanism

- Advantages:
  - Easiest way to implement the HPX-SYCL integration
- Disadvantages:
  - `host_tasks` not executed by HPX workers
    $\rightarrow$ Overhead/contention problem?

HPX-SYCL Integration Variant 1: Using SYCL `host_tasks`

Use SYCL `host_tasks` as callback mechanism

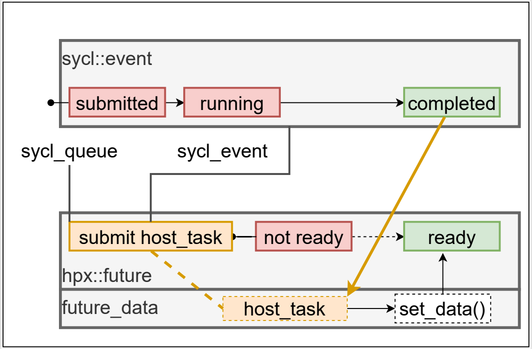
- Advantages:
  - Easiest way to implement the HPX-SYCL integration
- Disadvantages:
  - `host_tasks` not executed by HPX workers
    → Overhead/contention problem?



HPX-SYCL Integration Variant 1: Using SYCL host_tasks

Use SYCL host_tasks as callback mechanism

- Advantages:
  - Easiest way to implement the HPX-SYCL integration
- Disadvantages:
  - host_tasks not executed by HPX workers
    → Overhead/contention problem?



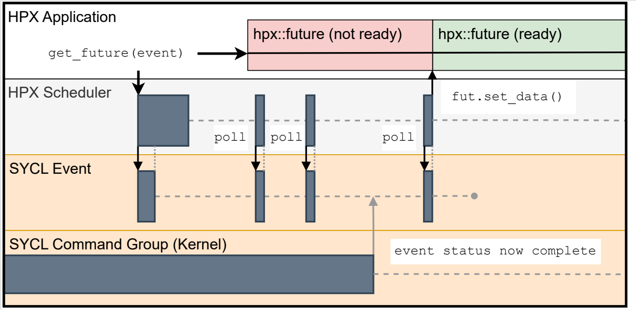
Create Callback during future_data creation

```
sycl_queue.submit([fdp =
    hpx::intrusive_ptr<future_data>(
    this),
    sycl_event](cl::sycl::handler& h) {
  h.depends_on(sycl_event);
  h.host_task ([fdp]() {
    fdp->set_data (hpx::util::unused); });
});
```

HPX-SYCL Integration Variant 2: Using Event Polling

Event polling within the HPX scheduler

- Store event-callback pairs in HPX scheduler
- Worker threads poll events in-between tasks and invoke callbacks
- Only one thread polls (others skip if mutex is already locked)
- Use concurrent queue for adding and mutex-protected vector for later checking



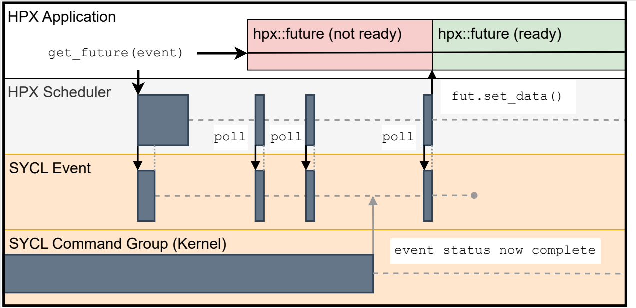
HPX-SYCL Integration Variant 2: Using Event Polling

Event polling within the HPX scheduler

- Store event-callback pairs in HPX scheduler
- Worker threads poll events in-between tasks and invoke callbacks
- Only one thread polls (others skip if mutex is already locked)
- Use concurrent queue for adding and mutex-protected vector for later checking

Advantages

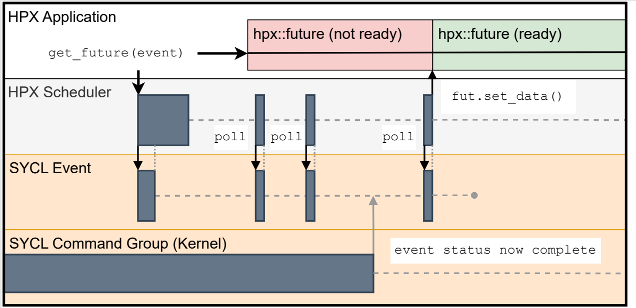
- HPX worker run callbacks themselves → One threadpool
- Works with SYCL implementations that do not yet support `host_tasks`



HPX-SYCL Integration Variant 2: Using Event Polling

Event polling within the HPX scheduler

- Store event-callback pairs in HPX scheduler
- Worker threads poll events in-between tasks and invoke callbacks
- Only one thread polls (others skip if mutex is already locked)
- Use concurrent queue for adding and mutex-protected vector for later checking



Advantages

- HPX worker run callbacks themselves → One threadpool
- Works with SYCL implementations that do not yet support `host_tasks`

Disadvantages

- Requires additions to the HPX scheduler
- Event creations, deletions and polling can cause overheads

HPX-SYCL Integration: Basic Usage and `get_future`

Dummy SYCL kernel/task

```
sycl::event my_event = queue.submit([&](sycl::handler& h) {
    /* insert SYCL dependencies */
    h.parallel_for(num_items, [=](auto i) {
            /* insert numeric code here */ });/
});
```

HPX-SYCL Integration: Basic Usage and `get_future`

Dummy SYCL kernel/task

```
sycl::event my_event = queue.submit([&](sycl::handler& h) {
    /* insert SYCL dependencies */
    h.parallel_for(num_items, [=](auto i) {
              /* insert numeric code here */ });/
```

Call HPX-SYCL integration

```
hpx::future<void> my_future =
   hpx::sycl::experimental::detail::get_future(my_event);
```

HPX-SYCL Integration: Basic Usage and `get_future`

Dummy SYCL kernel/task

```
sycl::event my_event = queue.submit([&](sycl::handler& h) {
    /* insert SYCL dependencies */
    h.parallel_for(num_items, [=](auto i) {
            /* insert numeric code here */ });/
});
```

Call HPX-SYCL integration

```
hpx::future<void> my_future =
   hpx::sycl::experimental::detail::get_future(my_event);
```

Add HPX continuation asynchronously

```
hpx::future<void> continuation_future =
   my_future.then([](auto&& fut) { /* insert CPU work,communication,... */});
```

HPX-SYCL Integration: Basic Usage and `get_future`

Dummy SYCL kernel/task

```
sycl::event my_event = queue.submit([&](sycl::handler& h) {
    /* insert SYCL dependencies */
    h.parallel_for(num_items, [=](auto i) {
                /* insert numeric code here */ });/
```

Call HPX-SYCL integration

```
hpx::future<void> my_future =
   hpx::sycl::experimental::detail::get_future(my_event);
```

Add HPX continuation asynchronously

```
hpx::future<void> continuation_future =
   my_future.then([](auto&& fut) { /* insert CPU work,communication,... */});
```

Suspend calling HPX task until everything is done

```
continuation_future.get()
```

HPX-SYCL Integration: HPX-SYCL Executor with `hpx::async`

Use HPX-SYCL Executor for convenience

- Wrapper for in-order SYCL queues
- Allows passing SYCL queue functions directly to hpx::async

Use HPX-SYCL Executor for convenience

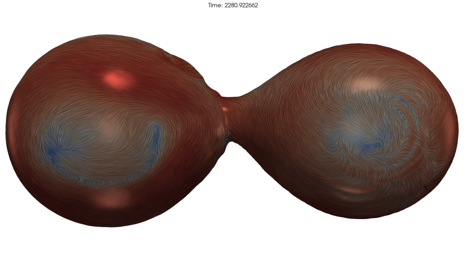
```
hpx::sycl::experimental::sycl_executor
  exec(sycl::default_selector{});
auto fut = hpx::async(exec,
      &sycl::queue::submit, [&](sycl::handler& h) {
          /* insert buffer accessesors */
          h.parallel_for(num_items, [=](auto i) {
          /* insert numeric code here */ });
      });
```

Scientific Application as a Benchmark: Octo-Tiger

Octo-Tiger: Overview

- **Simulation of interacting binary star systems and stellar mergers**
  - Double white dwarf mergers
  - Contact binary v1309 and its merger
  - R Coronae Borealis stars
- Intended for large scale, distributed runs
  - Previous runs: Cori, Piz Daint, Summit
  - Current target: Perlmutter
- Based on the HPX runtime

Octo-Tiger

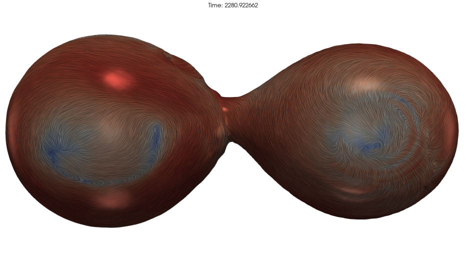


Scientific Application as a Benchmark: Octo-Tiger

Octo-Tiger: Overview

- **Simulation of interacting binary star systems and stellar mergers**
  - Double white dwarf mergers
  - Contact binary v1309 and its merger
  - R Coronae Borealis stars
- Intended for large scale, distributed runs
  - Previous runs: Cori, Piz Daint, Summit
  - Current target: Perlmutter
- Based on the HPX runtime

Octo-Tiger



Octo-Tiger as an HPX-SYCL Benchmark

- All major solvers are implemented with Kokkos

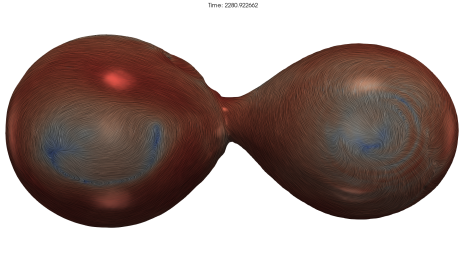
Scientific Application as a Benchmark: Octo-Tiger

Octo-Tiger: Overview

- **Simulation of interacting binary star systems and stellar mergers**
  - Double white dwarf mergers
  - Contact binary v1309 and its merger
  - R Coronae Borealis stars
- Intended for large scale, distributed runs
  - Previous runs: Cori, Piz Daint, Summit
  - Current target: Perlmutter
- Based on the HPX runtime

Octo-Tiger as an HPX-SYCL Benchmark

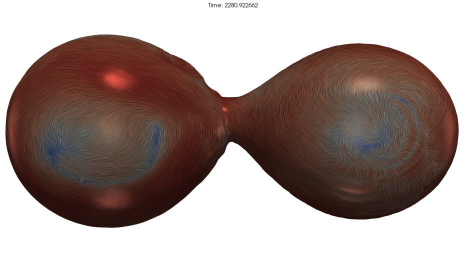
- All major solvers are implemented with Kokkos
- Kokkos supports various CPU/GPU execution and memory spaces (CUDA, HIP, HPX and **SYCL** spaces available)



Scientific Application as a Benchmark: Octo-Tiger

Octo-Tiger: Overview

- **Simulation of interacting binary star systems and stellar mergers**
  - Double white dwarf mergers
  - Contact binary v1309 and its merger
  - R Coronae Borealis stars
- Intended for large scale, distributed runs
  - Previous runs: Cori, Piz Daint, Summit
  - Current target: Perlmutter
- Based on the HPX runtime



Octo-Tiger as an HPX-SYCL Benchmark

- All major solvers are implemented with Kokkos
- Kokkos supports various CPU/GPU execution and memory spaces (CUDA, HIP, HPX and **SYCL** spaces available)
- Kokkos kernels can run a SYCL execution space

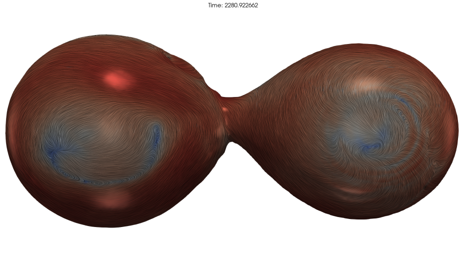
Scientific Application as a Benchmark: Octo-Tiger

Octo-Tiger: Overview

- **Simulation of interacting binary star systems and stellar mergers**
  - Double white dwarf mergers
  - Contact binary v1309 and its merger
  - R Coronae Borealis stars
- Intended for large scale, distributed runs
  - Previous runs: Cori, Piz Daint, Summit
  - Current target: Perlmutter
- Based on the HPX runtime

Octo-Tiger as an HPX-SYCL Benchmark

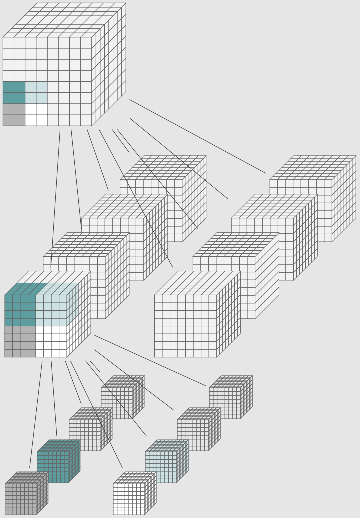
- All major solvers are implemented with Kokkos
- Kokkos supports various CPU/GPU execution and memory spaces (CUDA, HIP, HPX and **SYCL** spaces available)
- Kokkos kernels can run a SYCL execution space
- **HPX-SYCL integration $\rightarrow$ non-blocking HPX futures for Kokkos kernels running on the SYCL space**



Octo-Tiger: Datastructure and Solvers

Self-gravitating astrophysical fluids

- Inviscid Euler equations (Hydro) $\rightarrow$ Finite Volumes
- Newtonian Gravity (Gravity) $\rightarrow$ Fast Multipole Method

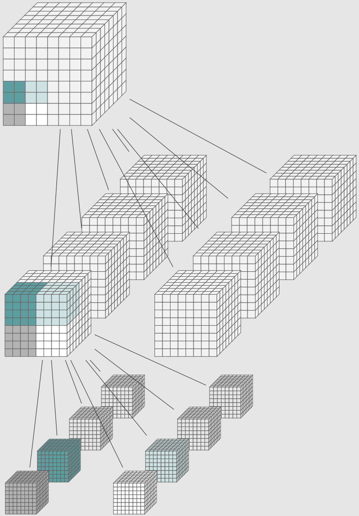


From [1]

Octo-Tiger: Datastructure and Solvers

Self-gravitating astrophysical fluids

- Inviscid Euler equations (Hydro) → Finite Volumes
- Newtonian Gravity (Gravity) → Fast Multipole Method
- Adaptive Mesh Refinement (AMR)
- Octree refined to maximize resolution for the atmosphere between the stars
- Entire sub-grid in each tree-node



From [1]

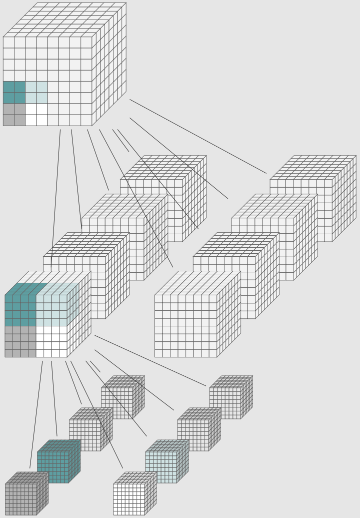
Octo-Tiger: Datastructure and Solvers

Self-gravitating astrophysical fluids

- Inviscid Euler equations (Hydro) → Finite Volumes
- Newtonian Gravity (Gravity) → Fast Multipole Method
- Adaptive Mesh Refinement (AMR)
- Octree refined to maximize resolution for the atmosphere between the stars
- Entire sub-grid in each tree-node

Kokkos Compute Kernels:

- Solvers traverse the tree, calling compute kernels on each sub-grid individually
- Each Kokkos kernel works on one sub-grid with many concurrent kernels being launched



From [1]

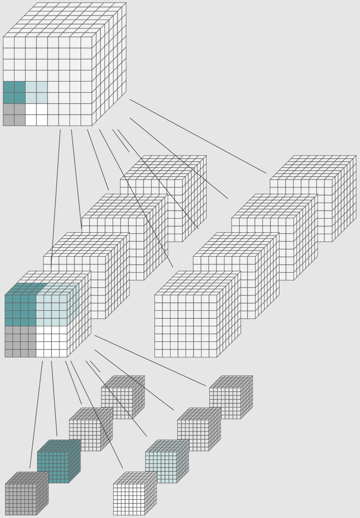
Octo-Tiger: Datastructure and Solvers

Self-gravitating astrophysical fluids

- Inviscid Euler equations (Hydro) $\rightarrow$ Finite Volumes
- Newtonian Gravity (Gravity) $\rightarrow$ Fast Multipole Method
- Adaptive Mesh Refinement (AMR)
- Octree refined to maximize resolution for the atmosphere between the stars
- Entire sub-grid in each tree-node

Kokkos Compute Kernels:

- Solvers traverse the tree, calling compute kernels on each sub-grid individually
- Each Kokkos kernel works on one sub-grid with many concurrent kernels being launched
- Even small scenarios contains thousands of kernel launches within $< 250ms$ $\rightarrow$ good stress test
- Not launching enough kernels in parallel can cause starvation (smallish kernels)

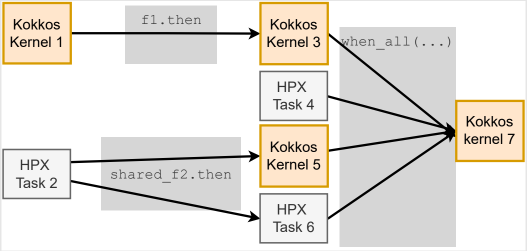


From [1]

Octo-Tiger: Execution Model

DAG of Compute Kernels

- HPX and Kokkos integrations exist
- Get futures for Kokkos kernels using **HPX-Kokkos compatibility library** (by calling `get_future` specializations within HPX)
- HPX-Kokkos only works for supported execution spaces (previously the CUDA, HIP and HPX spaces)
- Run individual Kokkos kernels either on a CPU (HPX) or GPU execution space



Adapted from [2]

Octo-Tiger: Execution Model

DAG of Compute Kernels

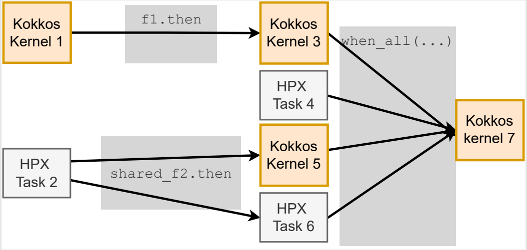
- HPX and Kokkos integrations exist
- Get futures for Kokkos kernels using **HPX-Kokkos compatibility library** (by calling `get_future` specializations within HPX)
- HPX-Kokkos only works for supported execution spaces (previously the CUDA, HIP and HPX spaces)
- Run individual Kokkos kernels either on a CPU (HPX) or GPU execution space

Kernel CPU Execution:

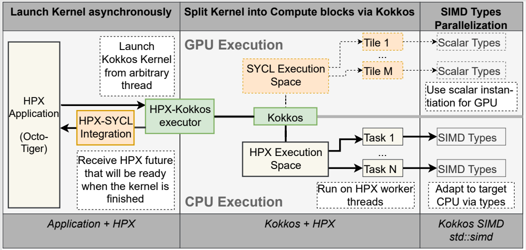
- Kernel gets split into HPX tasks
- Kernel gets instantiated with appropriate SIMD types

Kernel GPU Execution:

- SIMD template types get instantiated with scalar types
- Run on GPU execution space (CUDA, HIP, SYCL?)



Adapted from [2]



Adapted from [3]

Octo-Tiger: Execution Model

DAG of Compute Kernels

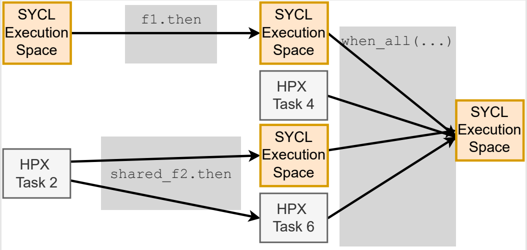
- HPX and Kokkos integrations exist
- Get futures for Kokkos kernels using **HPX-Kokkos compatibility library** (by calling `get_future` specializations within HPX)
- HPX-Kokkos only works for supported execution spaces (previously the CUDA, HIP and HPX spaces)
- Run individual Kokkos kernels either on a CPU (HPX) or GPU execution space

Kernel CPU Execution:

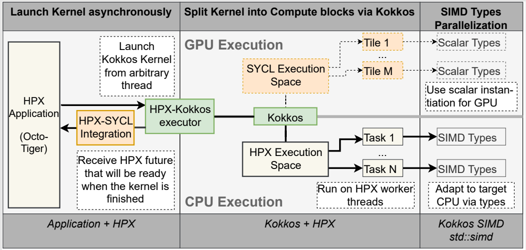
- Kernel gets split into HPX tasks
- Kernel gets instantiated with appropriate SIMD types

Kernel GPU Execution:

- SIMD template types get instantiated with scalar types
- Run on GPU execution space (CUDA, HIP, SYCL?)



Adapted from [2]



Adapted from [3]

Required (SYCL-related) Software Additions for Octo-Tiger and its Dependencies

- **HPX**:
  - **Changes:** Implemented both presented HPX-SYCL integration variants
  - **PR**: `https://github.com/STEllAR-GROUP/hpx/pull/6085`

Required (SYCL-related) Software Additions for Octo-Tiger and its Dependencies

- **HPX**:
  - **Changes:** Implemented both presented HPX-SYCL integration variants
  - **PR**: `https://github.com/STEllAR-GROUP/hpx/pull/6085`
- **HPX-Kokkos**:
  - **Purpose**: Compatibility layer for HPX and Kokkos. Allows treating Kokkos kernels as HPX tasks IF the `get_future` functionality exists for the underlying execution space.
  - **Changes**: Plug in the HPX-SYCL `get_future` call. Add `deep_copy_async` overload using the SYCL event directly
  - **PR**: `https://github.com/STEllAR-GROUP/hpx-kokkos/pull/13`

Required (SYCL-related) Software Additions for Octo-Tiger and its Dependencies

- **HPX**:
  - **Changes:** Implemented both presented HPX-SYCL integration variants
  - **PR**: `https://github.com/STEllAR-GROUP/hpx/pull/6085`
- **HPX-Kokkos**:
  - **Purpose**: Compatibility layer for HPX and Kokkos. Allows treating Kokkos kernels as HPX tasks IF the `get_future` functionality exists for the underlying execution space.
  - **Changes**: Plug in the HPX-SYCL `get_future` call. Add `deep_copy_async` overload using the SYCL event directly
  - **PR**: `https://github.com/STEllAR-GROUP/hpx-kokkos/pull/13`
- **CPPuddle**:
  - **Purpose**: Memory and executor utility library for task-based programming. Provides memory recycling allocators
  - **Changes**: Add allocators for SYCL memory pools on the device
  - **PR**: `https://github.com/SC-SGS/CPPuddle/pull/15`

Required (SYCL-related) Software Additions for Octo-Tiger and its Dependencies

- **HPX**:
  - **Changes:** Implemented both presented HPX-SYCL integration variants
  - **PR**: `https://github.com/STEllAR-GROUP/hpx/pull/6085`
- **HPX-Kokkos**:
  - **Purpose**: Compatibility layer for HPX and Kokkos. Allows treating Kokkos kernels as HPX tasks IF the `get_future` functionality exists for the underlying execution space.
  - **Changes**: Plug in the HPX-SYCL `get_future` call. Add `deep_copy_async` overload using the SYCL event directly
  - **PR**: `https://github.com/STEllAR-GROUP/hpx-kokkos/pull/13`
- **CPPuddle**:
  - **Purpose**: Memory and executor utility library for task-based programming. Provides memory recycling allocators
  - **Changes**: Add allocators for SYCL memory pools on the device
  - **PR**: `https://github.com/SC-SGS/CPPuddle/pull/15`
- **Octo-Tiger**:
  - **Changes**: Use correct SYCL execution space and memory allocators
  - **PR**: `https://github.com/STEllAR-GROUP/octotiger/pull/432`

Required (SYCL-related) Software Additions for Octo-Tiger and its Dependencies

- **HPX**:
  - **Changes:** Implemented both presented HPX-SYCL integration variants
  - **PR**: `https://github.com/STEllAR-GROUP/hpx/pull/6085`
- **HPX-Kokkos**:
  - **Purpose**: Compatibility layer for HPX and Kokkos. Allows treating Kokkos kernels as HPX tasks IF the `get_future` functionality exists for the underlying execution space.
  - **Changes**: Plug in the HPX-SYCL `get_future` call. Add `deep_copy_async` overload using the SYCL event directly
  - **PR**: `https://github.com/STEllAR-GROUP/hpx-kokkos/pull/13`
- **CPPuddle**:
  - **Purpose**: Memory and executor utility library for task-based programming. Provides memory recycling allocators
  - **Changes**: Add allocators for SYCL memory pools on the device
  - **PR**: `https://github.com/SC-SGS/CPPuddle/pull/15`
- **Octo-Tiger**:
  - **Changes**: Use correct SYCL execution space and memory allocators
  - **PR**: `https://github.com/STEllAR-GROUP/octotiger/pull/432`
- **Kokkos**:
  - Already contained SYCL execution and memory space
  - **Required changes**: Some CMake additions to allow using the SYCL execution space on AMD GPUs
  - **Optional optimization**: Removing internal execution space barriers for in-order queues
  - **PR**: Not yet upstreamed

Experiment Setup

Scenario Size, Number of Kernel Calls per Time-Step

Grid parameters			GPU metrics per time-step	
Sub-grid size	Overall number of cells	Number of (leaf) sub-grids	Kernel calls	CPU-GPU data transfers
8^3 **(512)**	**262144**	**512**	**7680**	**15360**

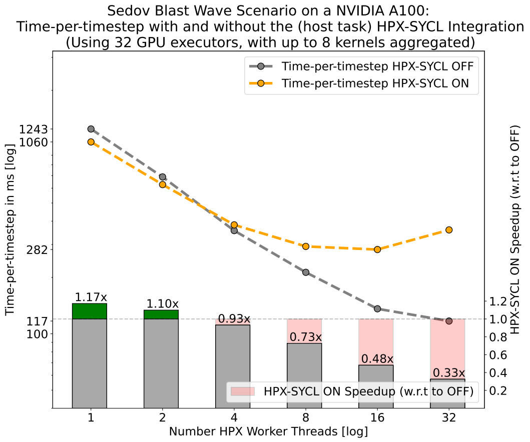
Scenario

- **Goal: Evaluate performance with and without the HPX-SYCL integration** turned on
- Use patch to turn of the integration by inserting event wait commands and returning ready futures
- Vary number of HPX worker threads (steers contention)
- **Simple Node-Level Hydro-Only Scenario**: Sedov-Taylor Blast Wave
- Using Intel DPC++/OneAPI

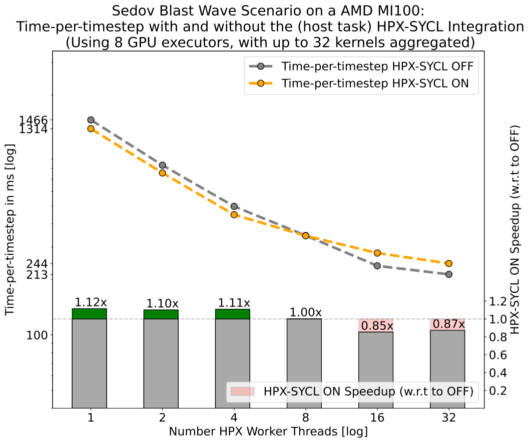
Hardware

- NVIDIA® GPU node
  - CPU: Intel® Xeon® Platinum 8358 CPU
  - NVIDIA A100 GPU
- AMD® GPU node
  - CPU: AMD EPYC™ 7H12 CPU.
  - GPU: AMD MI100 GPU
- Use **best combination of performance parameters** for each node (number of concurrent GPU executors, dynamic work aggregation limit)

Results: Host Task Integration

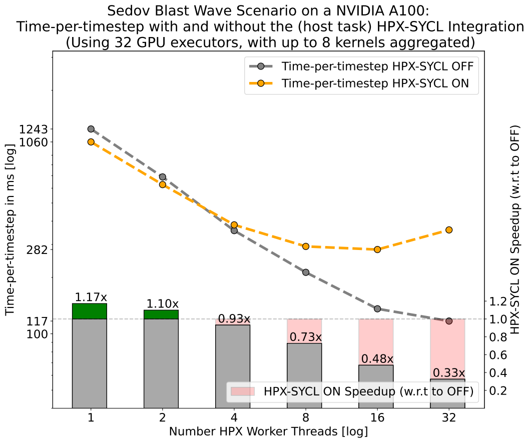


(a) A100: Best combinations

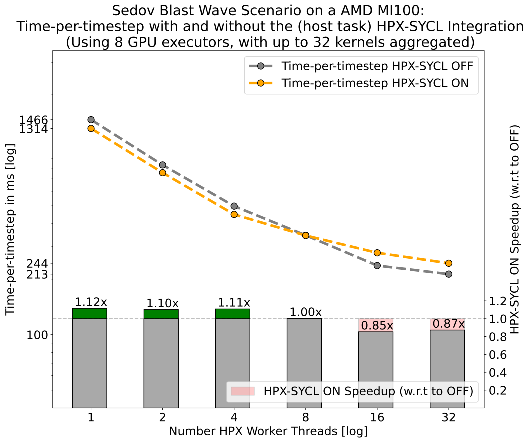


(b) MI100: Best combinations

Results: Host Task Integration



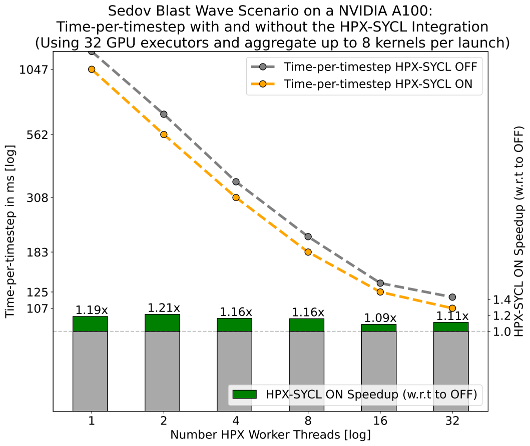
(a) A100: Best combinations



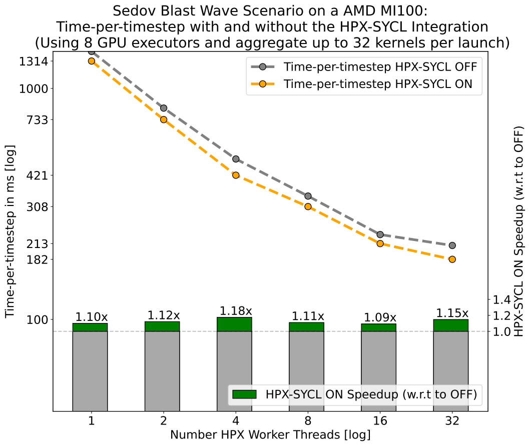
(b) MI100: Best combinations

→ Runtime degrades when using the `host_task`-based HPX-SYCL integration (at least when using all CPU cores)

Results: Event Polling Integration

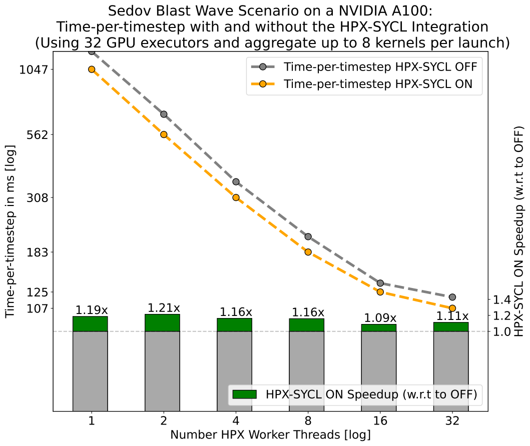


(a) A100: Best combinations

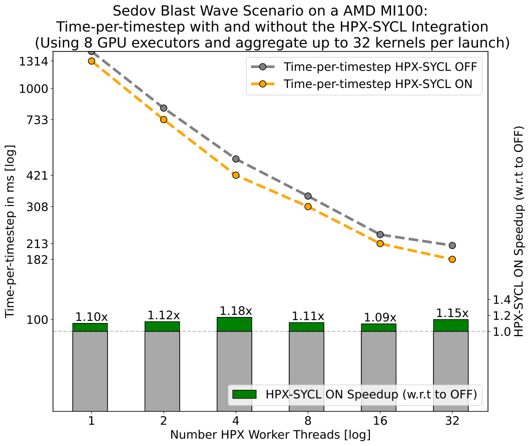


(b) MI100: Best combinations

Results: Event Polling Integration



(a) A100: Best combinations



(b) MI100: Best combinations

→ Runtime consistently improves when using the event polling HPX-SYCL integration (even for this small scenario)

Conclusion

Conclusion

- Developed HPX-SYCL integration allowing us to treat SYCL events as HPX tasks
- Adapted entire Octo-Tiger software stack for SYCL to benchmark the integration(s)
- Event polling integration performs better than (DPC++) host tasks integration
- Integration is beneficial (over synchronous execution without it), even when just running simple, single-node scenarios
- Software stack is still experimental, lots of potential for optimizations

Outlook

- Intel GPUs?
- Integration speedup with distributed runs?

Universität Stuttgart

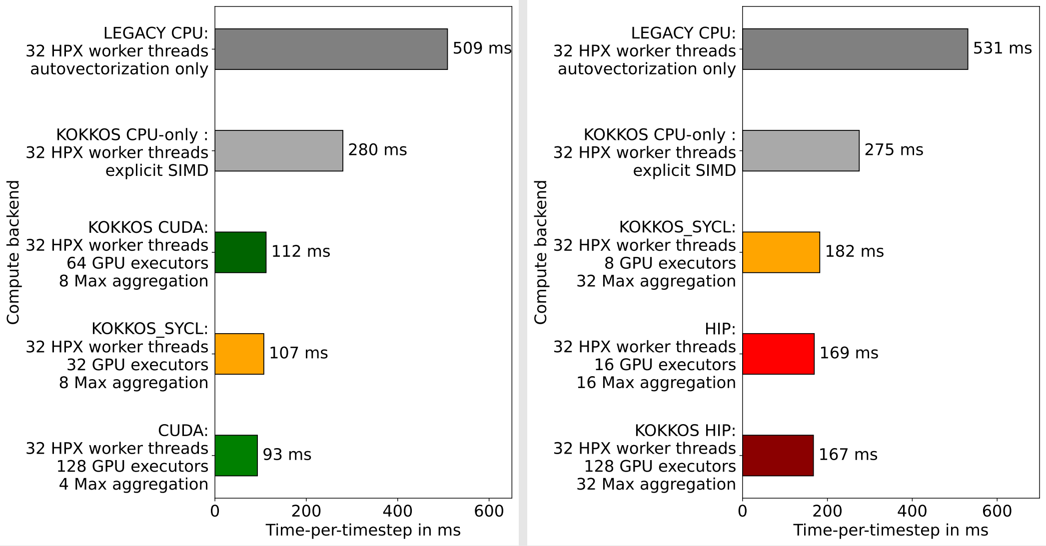
LSU | Center for Computation & Technology
Interdisciplinary | Innovative | Inventive

STE||AR GROUP

Thank you for your attention!

- D. Pfander, G. Daiß, D. Marcello, H. Kaiser, and D. Pflüger, "Accelerating Octo-Tiger: Stellar mergers on Intel Knights Landing with HPX," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '18. New York, NY, USA: ACM, 2018, pp. 19:1–19:8.
- G. Daiß *et al.* (video presentation) hips 2021: Beyond fork-join: Integration of performance portable kokkos kernels with hpx. Youtube. [Online]. Available: https://www.youtube.com/watch?v=CQaA9AYlm1I
- G. Daiß, S. Singanaboina, P. Diehl, H. Kaiser, and D. Pflüger, "From merging frameworks to merging stars: Experiences using hpx, kokkos and simd types," in *2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2022, pp. 10–19. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ESPM256814.2022.00007

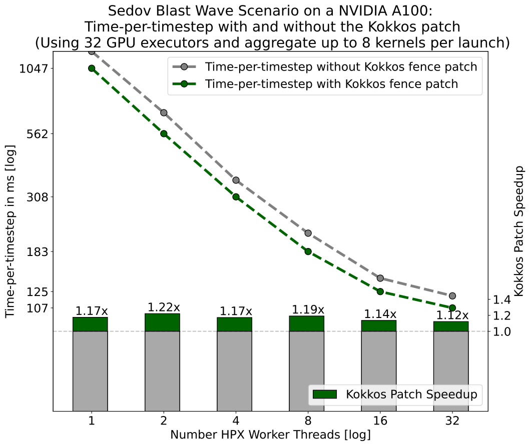
Performance using various execution spaces:



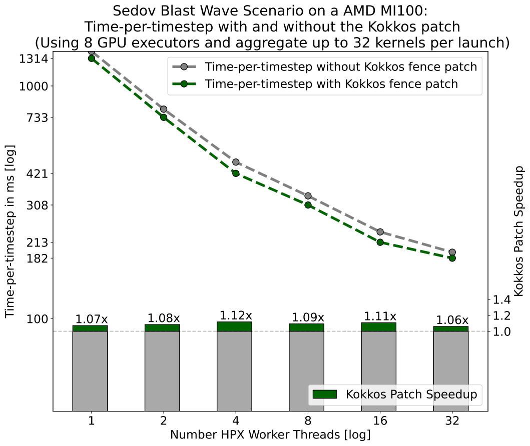
(a) Best runs on the NVIDIA A100

(b) Best runs on the AMD MI100

Speedup when removing barriers within Kokkos for in-order queues:

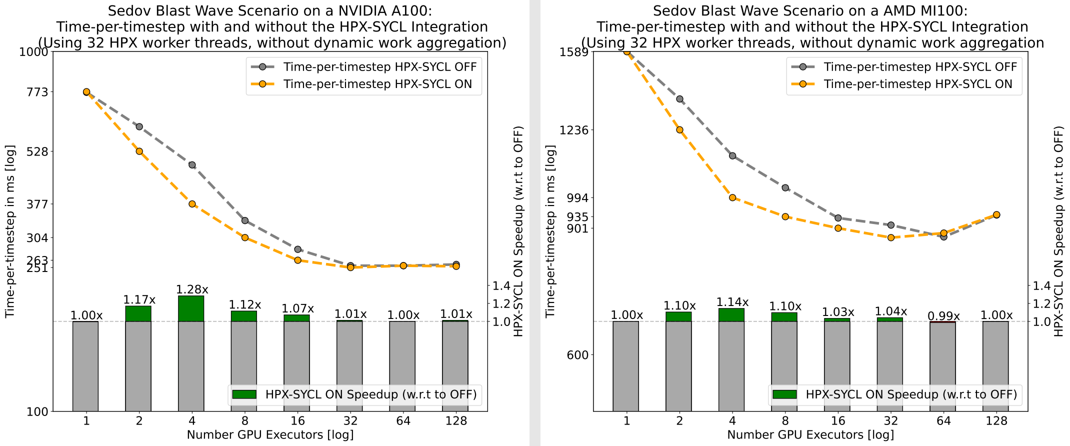


(a) A100: Best combinations



(b) MI100: Best combinations

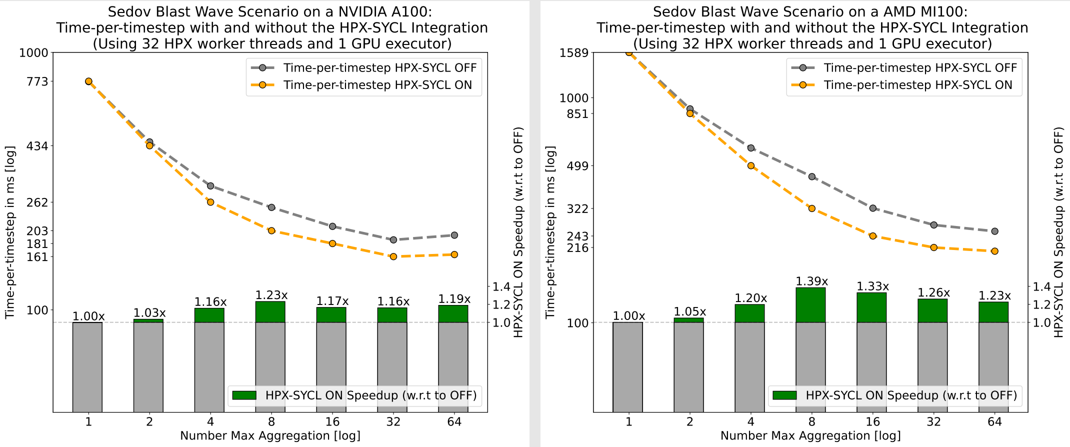
Event polling integration: Runtime with varying number of executors



(a) A100: Increasing Number of GPU executors

(b) MI100: Increasing Number of GPU executors

Event polling integration: Runtime with varying number of aggregated kernels



(c) A100: Increasing number of kernels aggregated

(d) MI100: Increasing number of kernels aggregated