

IWOCL 2026



Update on SYCL Usage in the Shamrock Framework and Exascale Scalability

Timothée David--Cléris, Université Grenoble Alpes

Timothée David--Cléris, UGA / CNRS



- What is Shamrock ?
- Performance overview (& aurora scalability)
- Our usage of SYCL
- Migrating away from `sycl::buffer(s)`



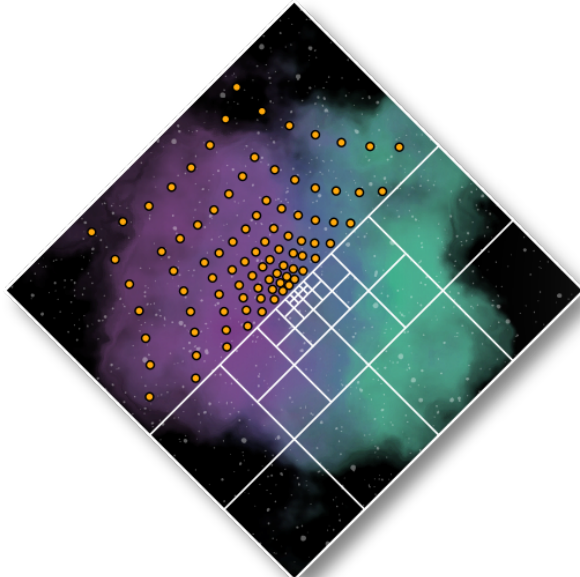
Github public repo

Acknowledgments :



No. 101053020 (Dust2Planets)

DD Allocation Shamrock on Aurora

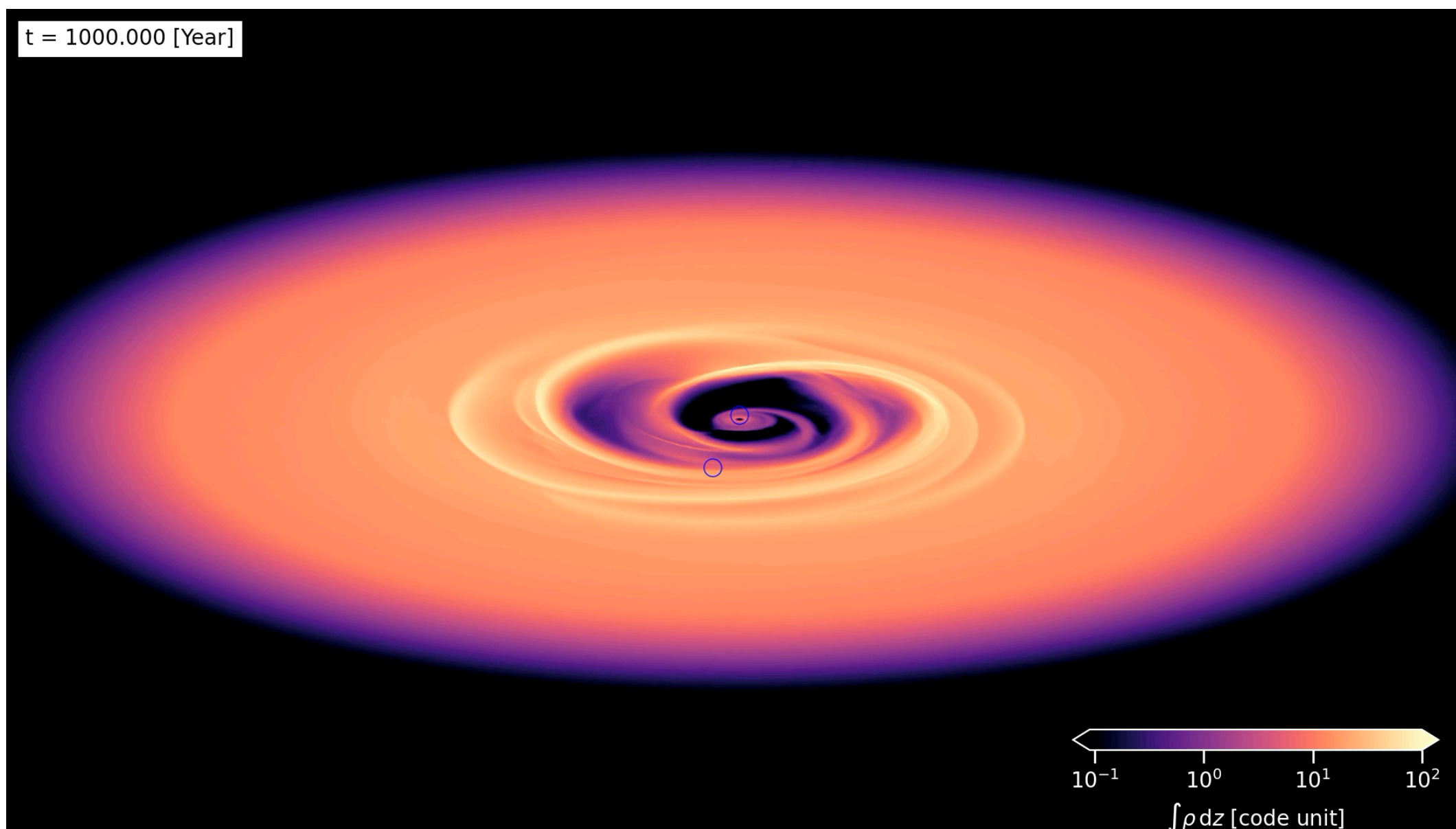
What is  Shamrock

GOAL : run hydro on exascale !

Which method ?

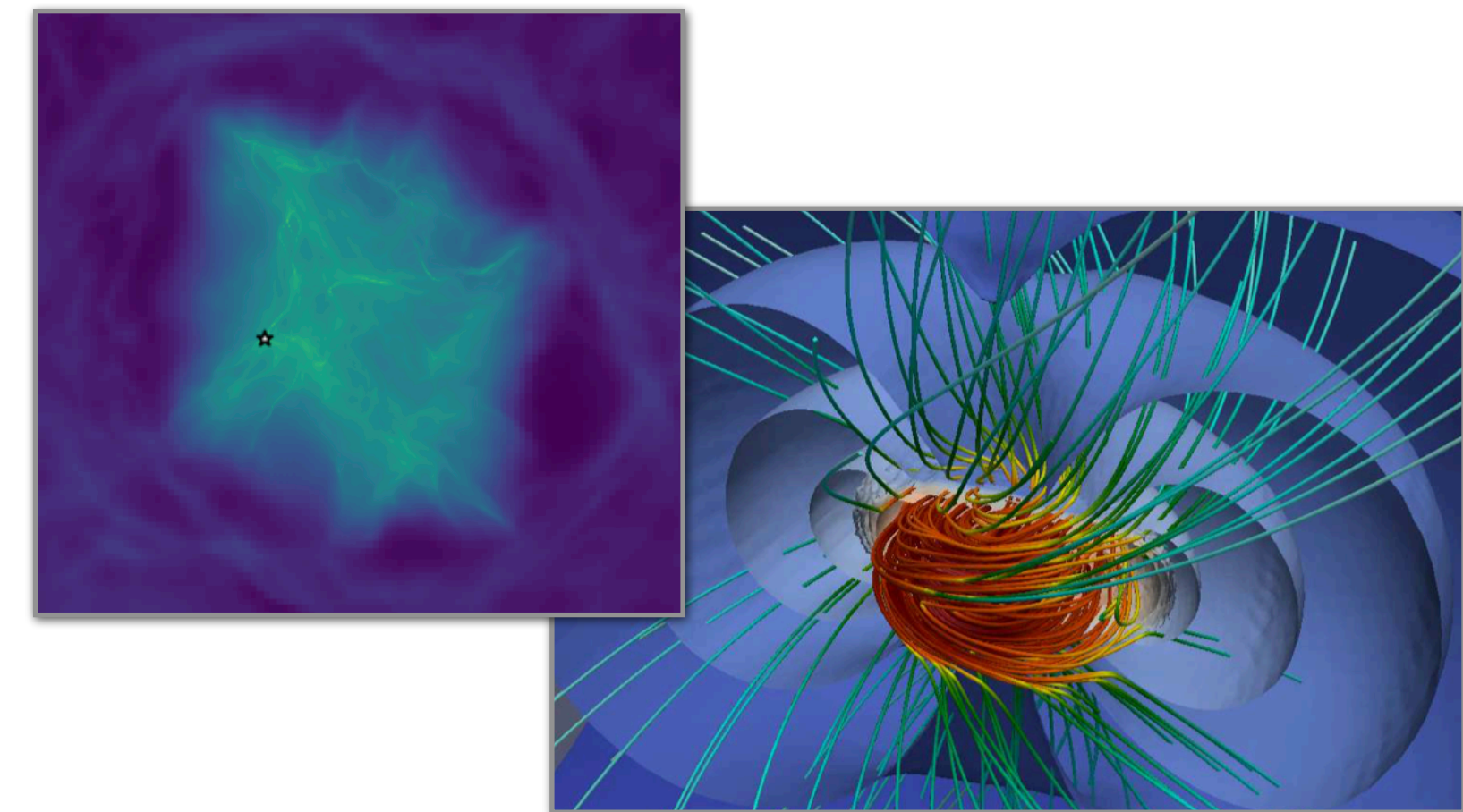
Meshless (SPH) ?

The best method obviously



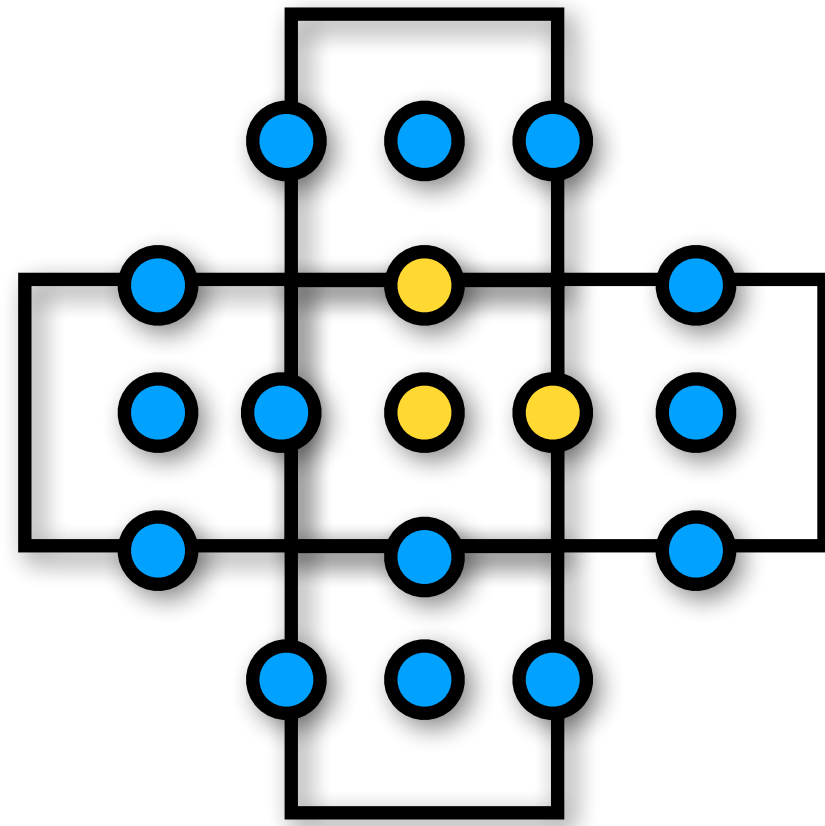
Grid / AMR ?

Look at my shocks !

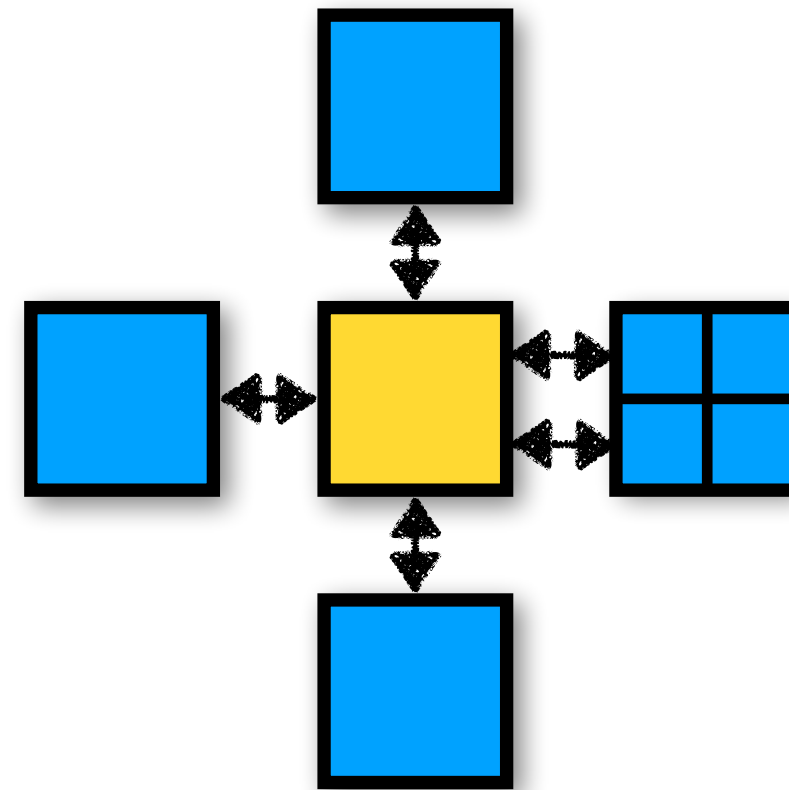


3 main numerical methods in astro :

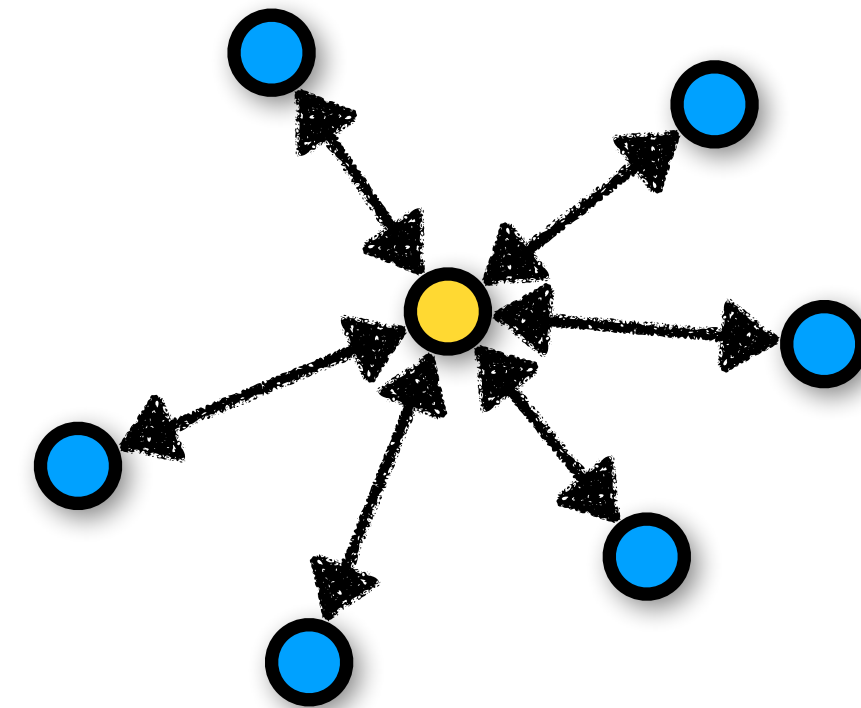
Finite elements



Finite Volumes



SPH



$$\gamma(\text{yellow circle}, \text{blue circle})$$

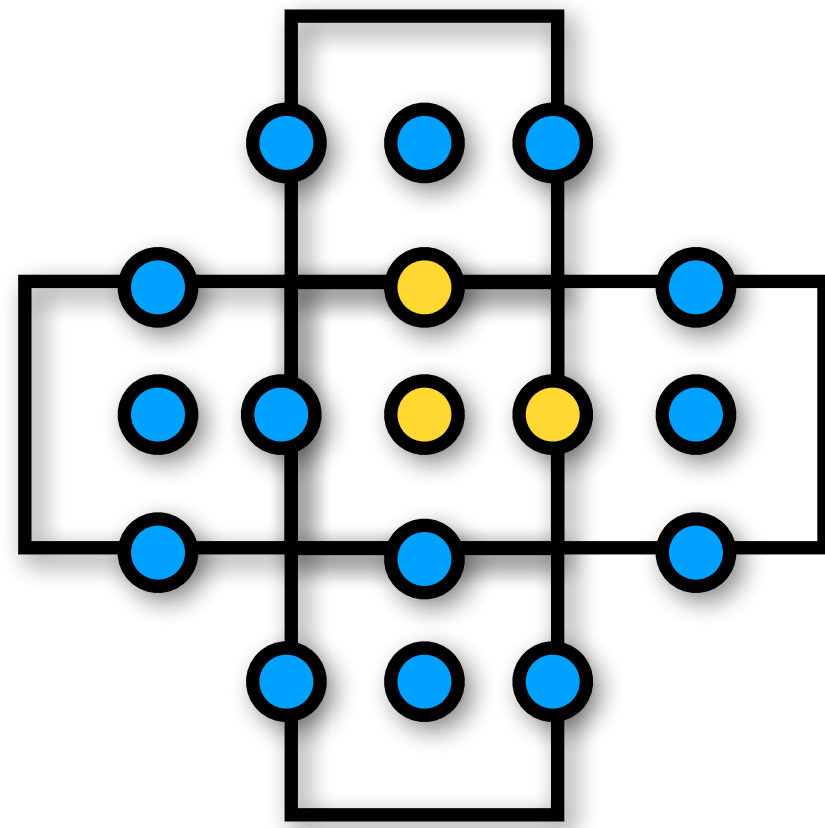
&

$$\mathbb{U}^{t+1} = F \left(\{ \mathbb{U}_i^t \}_{i \in \{\text{blue circle}\}} \right)$$

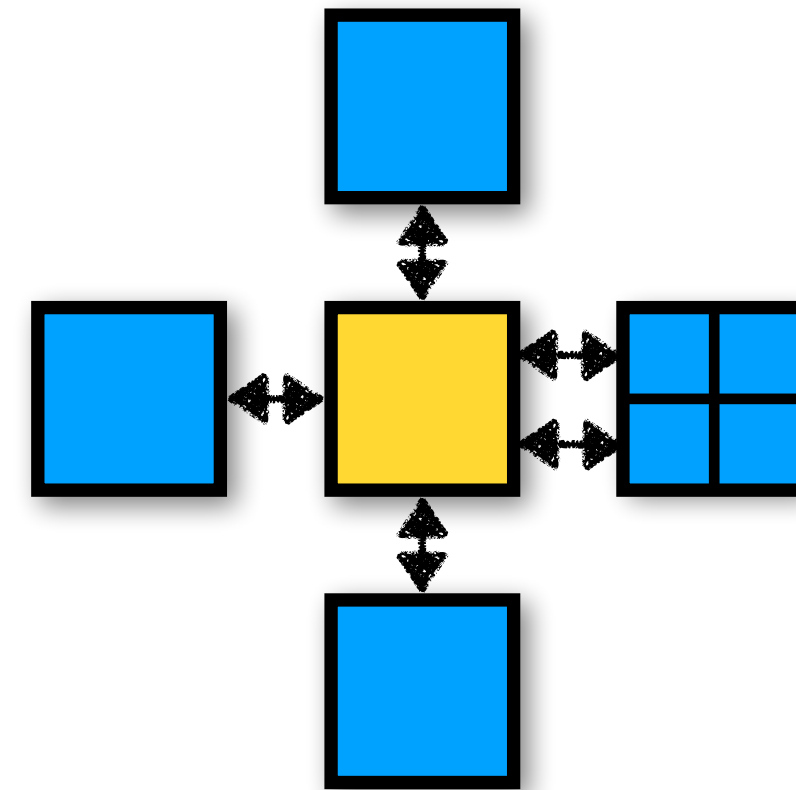
Interaction criterion

Numerical scheme

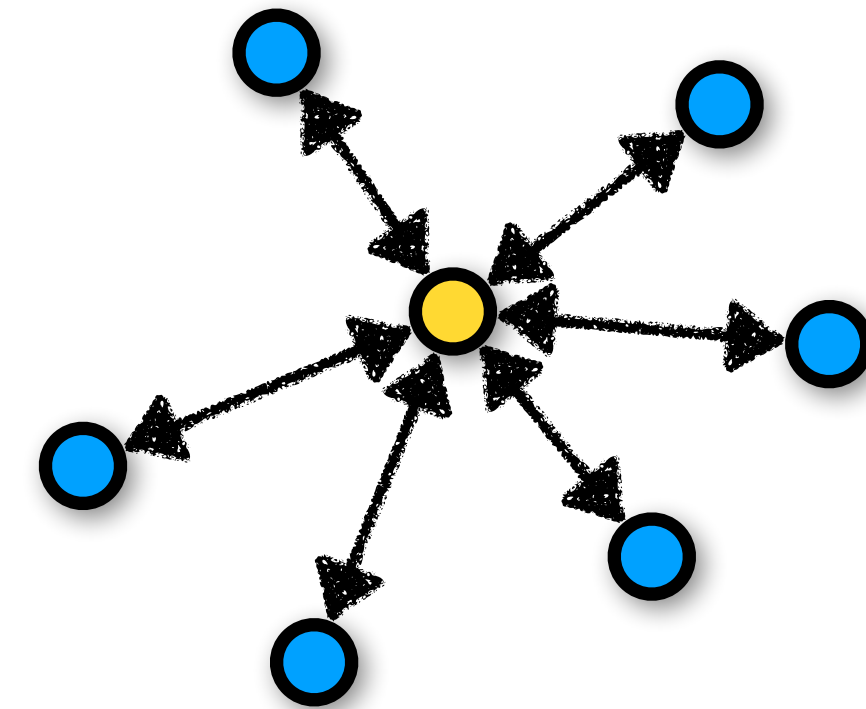
Finite elements



Finite Volumes



SPH



$$\gamma\left(\begin{array}{c} \text{yellow circle} \\ \text{yellow square} \end{array}, \begin{array}{c} \text{blue circle} \\ \text{blue square} \end{array}\right)$$

&

$$\begin{array}{c} \text{yellow circle} \\ \text{yellow square} \end{array}^{t+1} = F\left(\left\{\begin{array}{c} \text{blue circle} \\ \text{blue square} \end{array}^t\right\}_{i \in \{\text{blue circle}, \text{blue square}\}}\right)$$

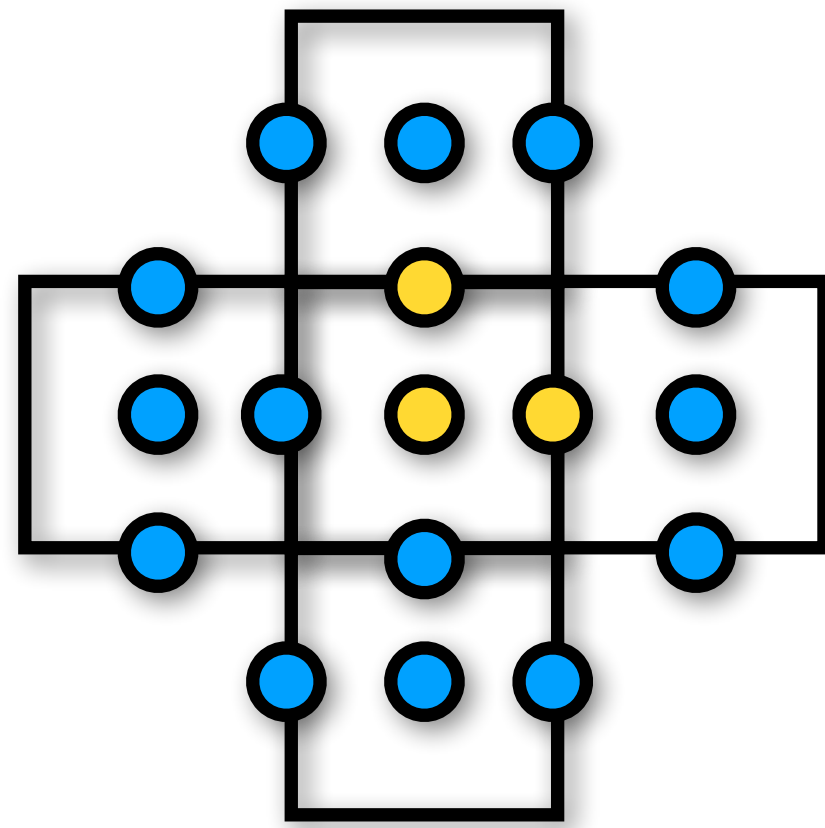
Interaction criterion

Numerical scheme

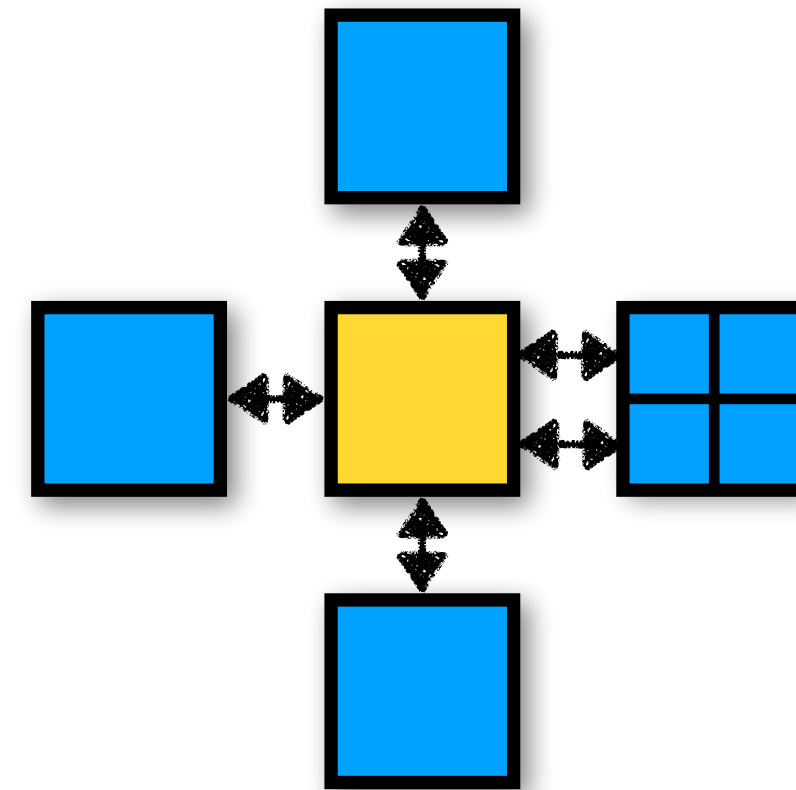
Criterion γ
Scheme F

Abstraction

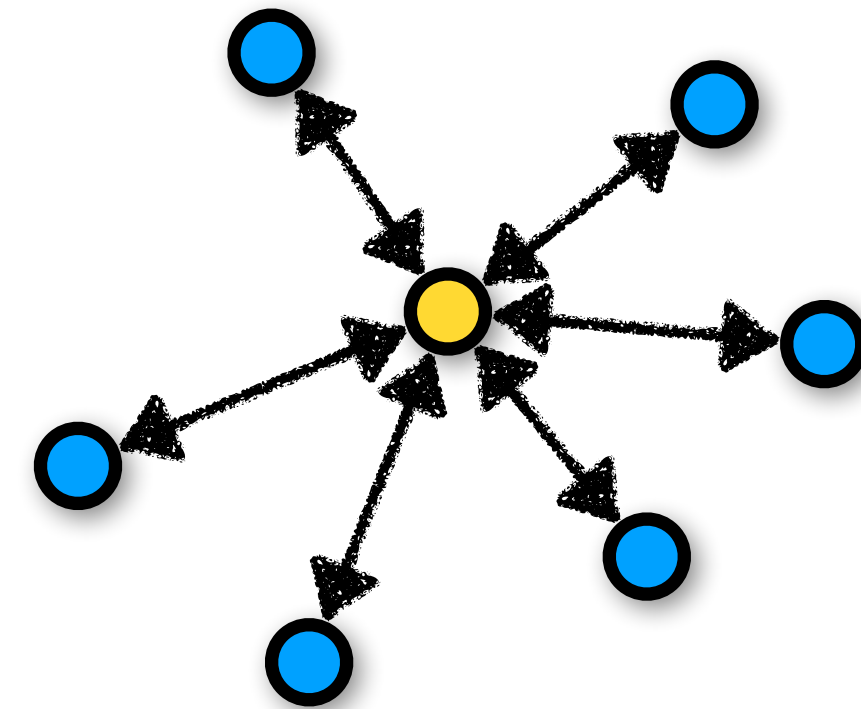
Finite elements



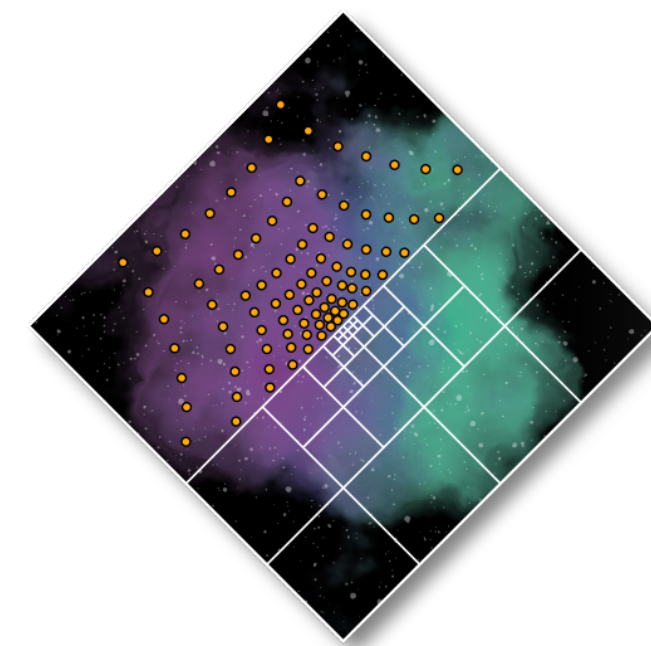
Finite Volumes



SPH



Criterion γ
Scheme F



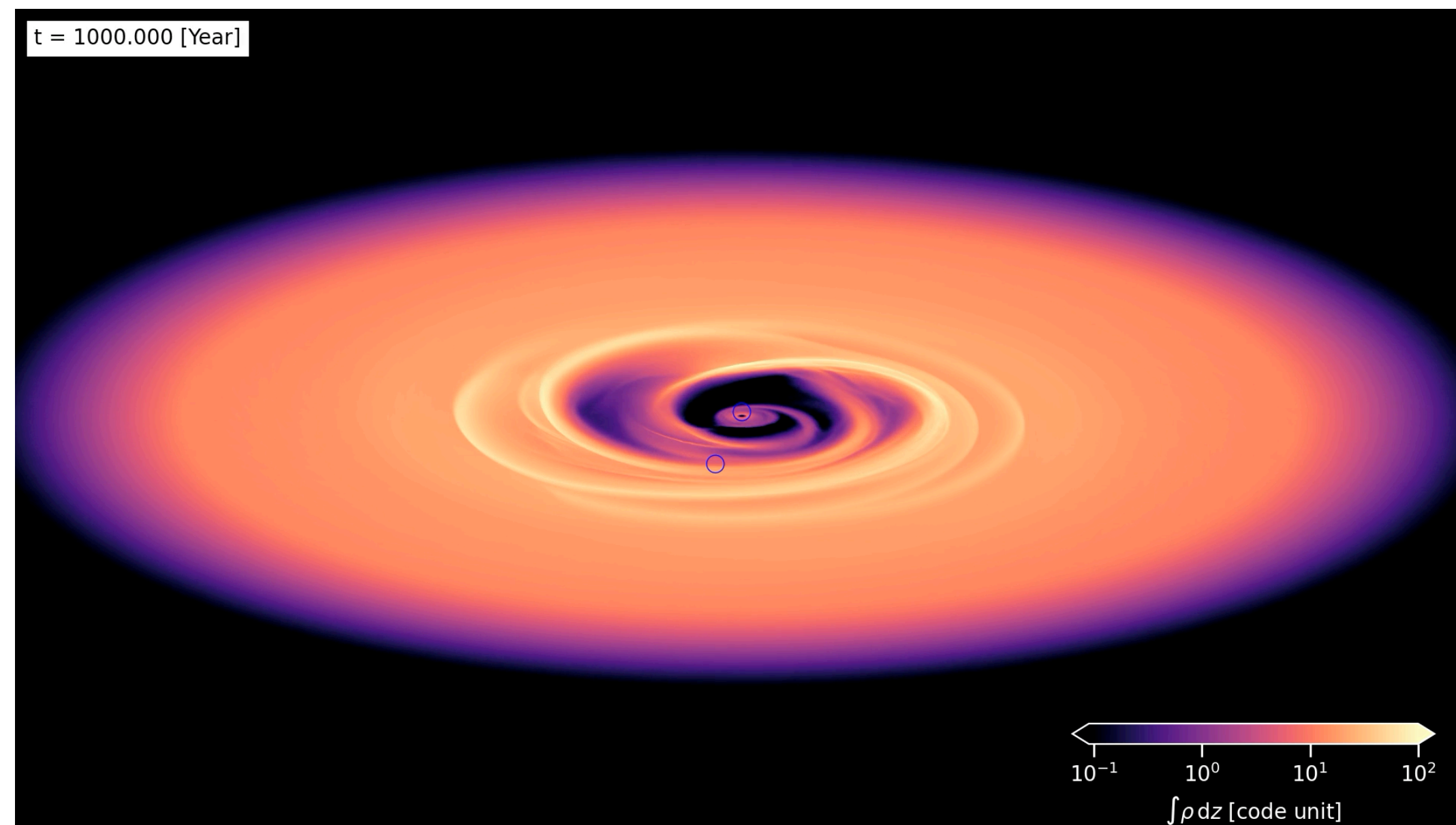
Shamrock

Generalized sparse communications
Generalized neighbor finding
Generalized load balancing

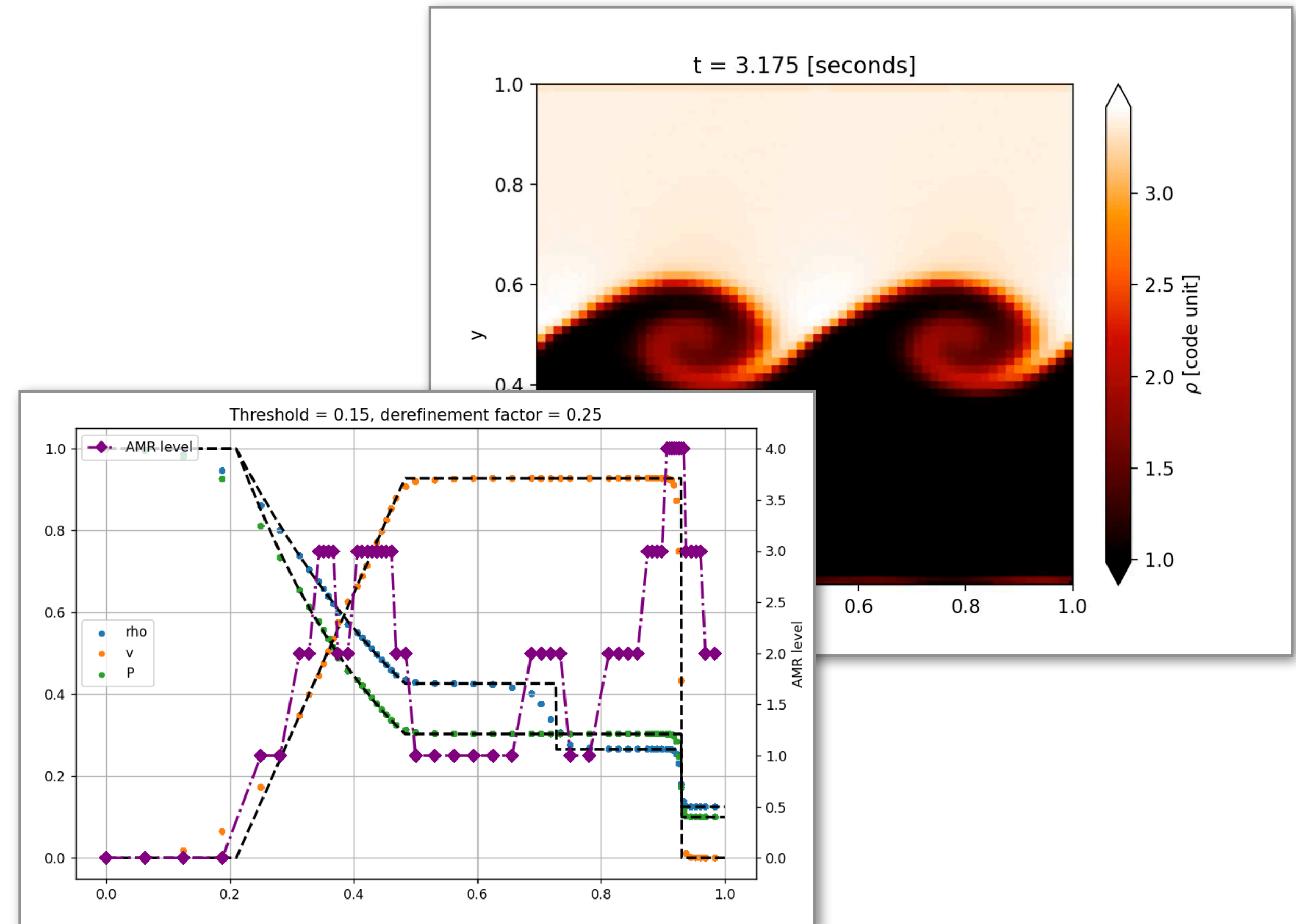
Solver
+ scalability

PHD = write this from scratch

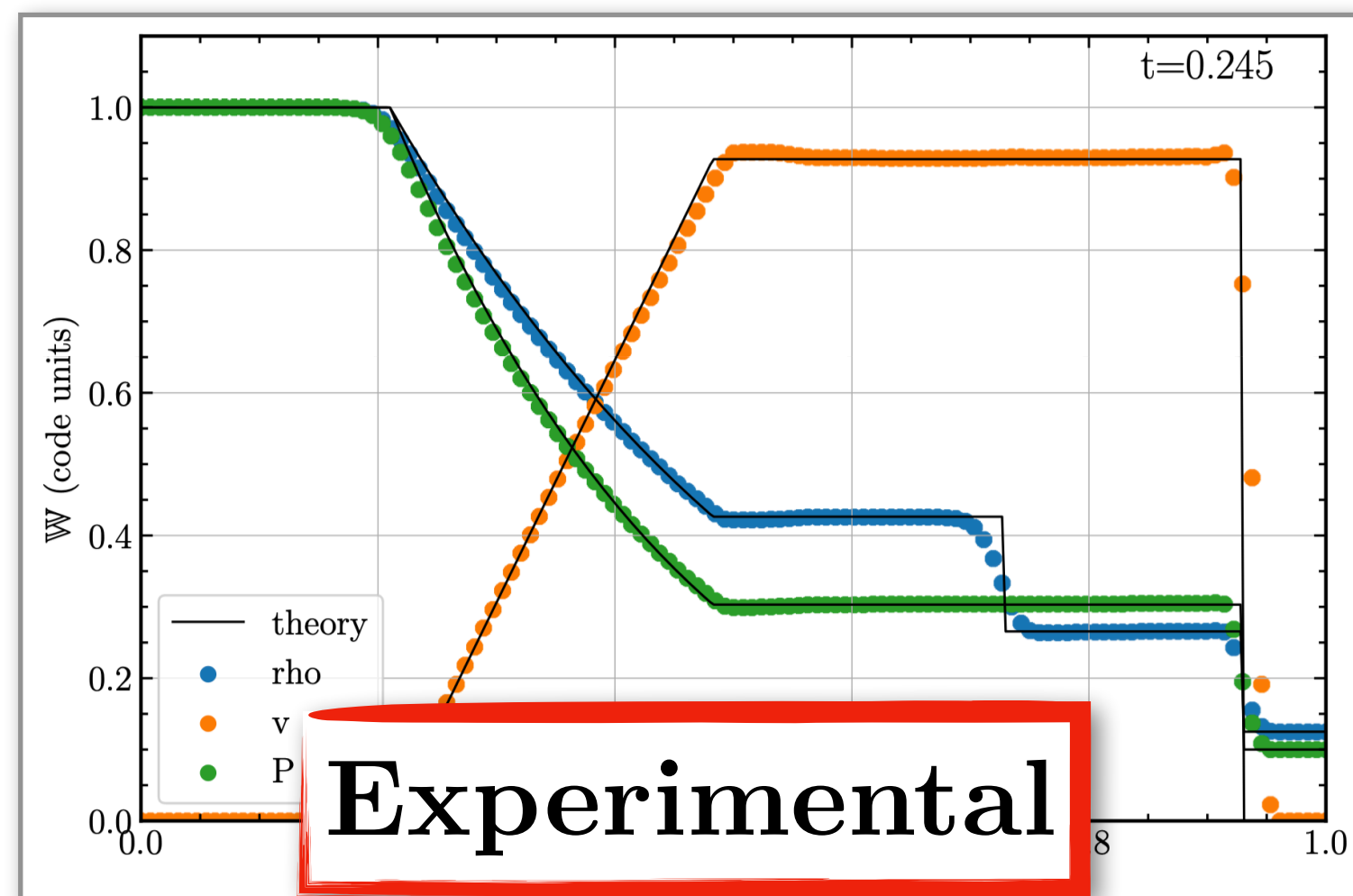
SPH (Phantom kind)



Finite volume (RAMSES kind)



Finite elements (Zeus/Fargo kind)



+ Godunov-SPH (since 2 months ago)

From S. Inutsuka group (Nagoya, Japan)

Enable multi-method studies with a single code !

● Interoperable with Python (Simple user interface)

```
[1]: import shamrock

[2]: shamrock.change_loglevel(1)

-> modified loglevel to 0 enabled log types :
log status :
- Loglevel: 1, enabled log types :
[xxx] Info: xxx ( logger::info )
[xxx] : xxx ( logger::normal )
[xxx] Warning: xxx ( logger::warn )
[xxx] Error: xxx ( logger::err )

[4]: shamrock.sys.init("0:0")

[5]: shamrock.sys.world_size()

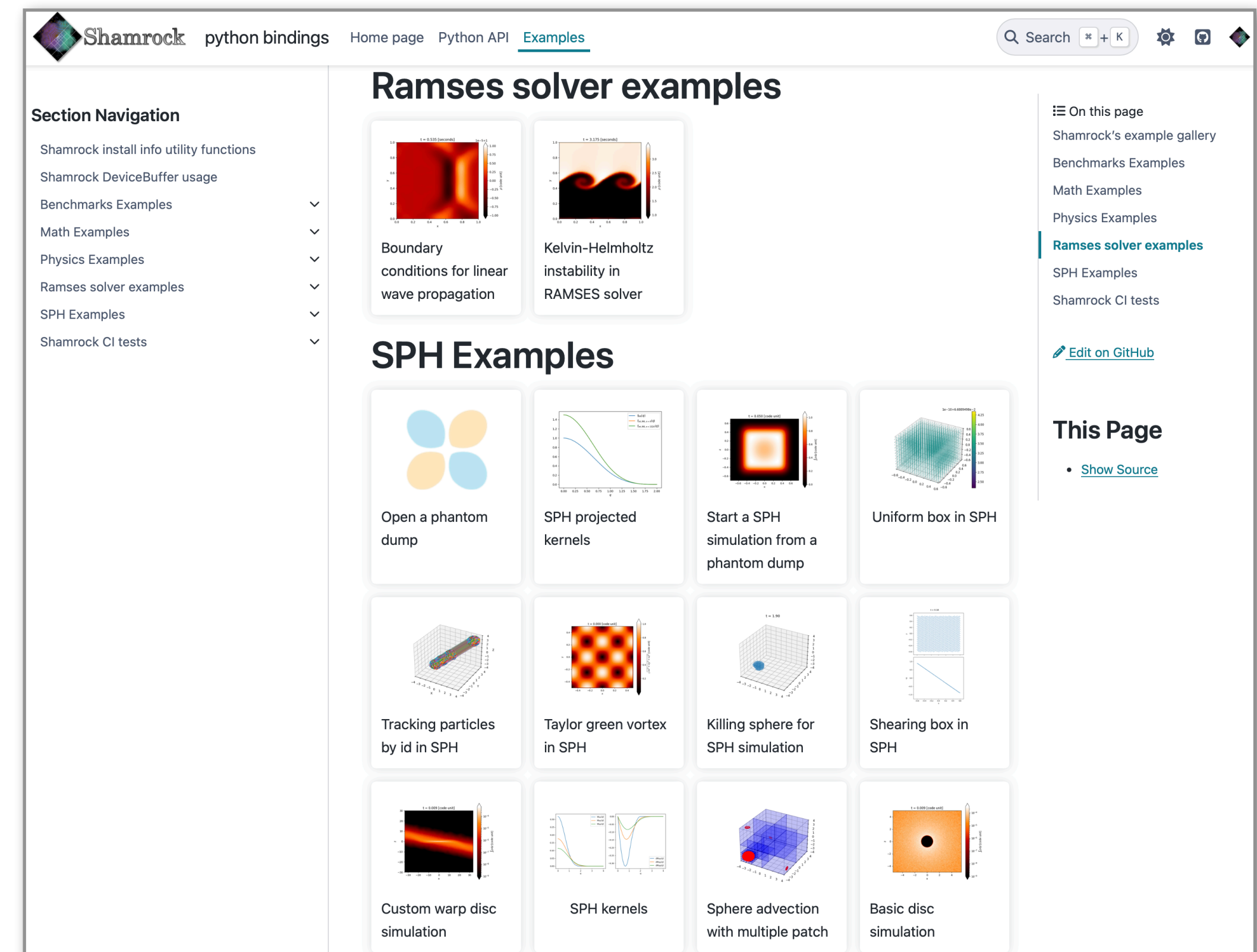
[5]: 1
```

```
[1]: import shamrock

[ ]: shamrock.get_M
```

f	get_Model_Ramses	function
f	get_Model_SPH	function
f	get_Model_Zeus	function

Documentation:



Matplotlib inspired doc, [Link](#)



Versions for shamrock

Versions Packages Information History Badges Report

4 package(s) known

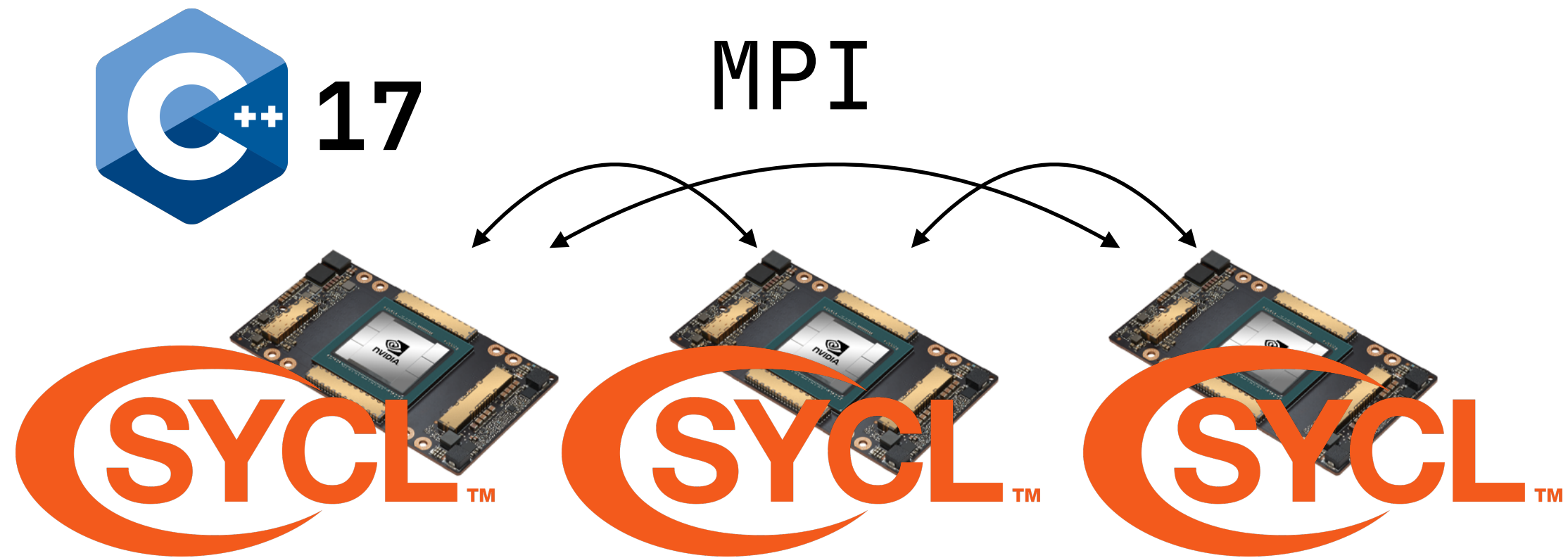
Repository	Package name	Version	Category	Maintainer(s)
Homebrew	shamrock	HEAD	-	-
Homebrew	shamrock	2025.10.0	-	-
Spack	shamrock	main	-	tdavidcl@spack
Spack	shamrock	2025.10.0	-	tdavidcl@spack

AdaptiveCpp OpenMP
Contributed last year in homebrew !

AdaptiveCpp, Icpix, Intel/llvm
(All backends)

- Everything in a single executable / no #ifdefs
- Packaged using AdaptiveCpp (Also icpx on Spack)

How ?
Do everything on GPUs



Papers :

SHAMROCK: Exascale hydrodynamics for astrophysics using SYCL.

Timothée David-Cléris
Univ. Grenoble Alpes, CNRS, IPAG
38000 Grenoble, France
davidclt@univ-grenoble-alpes.fr

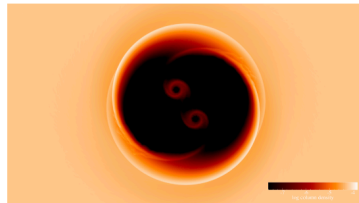


Figure 1: Snapshot of an SPH simulation of a circum-binary disc using SHAMROCK with one billion particles, made using 32 AMD-MI250X on the Adastra supercomputer.

Abstract

We present SHAMROCK, a native SYCL framework for astrophysics, designed to implement various numerical methods for modelling hydrodynamic flows, in particular Smoothed Particle Hydrodynamics (SPH). At the core of SHAMROCK lies a fast radix tree building algorithm that allows the tree to be rebuilt at each timestep with minimal cost, eliminating the need for tree communications or updates. Additionally, a domain decomposition method is used on top of the radix tree, allowing for a nearly linear multi-GPU weak scalability, resulting in 92% weak scaling efficiency on 1024 MI250X AMD graphical accelerators for large SPH simulations.

CCS Concepts

- Applied computing → Physics • Computing methodologies → Massively parallel and high-performance simulations.

Keywords

High performance computing, Astrophysics, SYCL, MPI, GPU, exascale, Smoothed Particle Hydrodynamics, SPH.

ACM Reference Format

Timothée David-Cléris, 2025. SHAMROCK: Exascale hydrodynamics for astrophysics using SYCL. In *Proceedings of International Workshop on OpenCL*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components that may not be registered with the Copyright Clearance Center (CCC) are not included in this notice. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/authors. Publication rights licensed to ACM. ACM ISBN 978-1-60558-XXXXXX-0/2025/03.

<https://doi.org/XXXXXX.XXXXXXX>

Monthly Notices
ROYAL ASTRONOMICAL SOCIETY
MNRAS **539**, 1–33 (2025)
Advance Access publication 2025 March 18
<https://doi.org/10.1093/mnras/stad444>

The SHAMROCK code: I – smoothed particle hydrodynamics on GPUs

T. David-Cléris^{1,2*}, G. Laibis^{1,3} and Y. Lapeyre¹
¹ENS de Lyon, CNRS UMR5574, Université Claude Bernard Lyon 1, CNRS, Lyon F-69007, France
²CNRS, IPAG, Université Grenoble Alpes, F-38000 Grenoble, France
³Institut Universitaire de France, France

Accepted 2025 March 11. Received 2025 March 11; in original form 2024 May 23

ABSTRACT

We present SHAMROCK, a performance portable framework developed in C++ 17 with the SYCL programming standard, tailored for numerical astrophysics on Exascale architectures. The core of SHAMROCK is an accelerated parallel tree with negligible construction time, whose efficiency is based on binary algebra. The smoothed particle hydrodynamics algorithm of the PHANTOM code is implemented in SHAMROCK. On-the-fly tree construction circumvents the necessity for extensive data communications. In tests displaying a uniform density with global time-stepping with tens of billions of particles, SHAMROCK completes a single time-step in a few seconds using over the thousand of GPUs of a supercomputer. This corresponds to processing billions of particles per second, with tens of millions of particles per GPU. The parallel efficiency across the entire cluster is larger than ~ 90 per cent.

Key words: methods: numerical.

1 INTRODUCTION

The study of the formation of structures in the Universe is a field in which non-linear, non-equilibrium physical processes interact at many different scales, requiring ever greater computing resources to simulate them, right up to Exascale (one quintillion operations per second). To increase energy efficiency with acceptable CO₂ emissions, recent supercomputers have been designed with specialized hardware such as ARM central processing units (CPUs) or graphics processing units (GPUs). GPUs involve multiple computational units that perform the same operation on multiple data simultaneously through specific instructions (Single Instruction Multiple Data, or SIMD parallel processing). This type of hardware differs radically from standard x86 CPUs, requiring a complete rewrite of CPU-based codes.

Considerable efforts have recently been invested into developing codes adapted to the new hybrid architectures aimed at Exascale (e.g. LINTX; Lenz et al. 2023; PARTHENOS; Gots et al. 2022; QUOKKA; Winkler & Krumholz 2022). Most of the codes in the community targeting Exascale are grid-based. However, these methods can be inadequate for systems with complex geometries or misaligned flows, highlighting the need for smoothed particle hydrodynamics (SPH) and, more broadly, Lagrangian-based approaches on Exascale architectures. Currently, problems involving complex geometries, such as multiple distorted systems, are primarily studied using SPH methods. However, resolution is often limited by a lack of MESSAGE PASSING INTERFACE (MPI) scalability and GPU support, preventing efficient execution on Exascale architectures. In particular, simulating small-scale instabilities embedded within global structures requires at least a billion SPH particles, which is an impractical number for codes without Exascale capabilities. The performance of hydrodynamical codes is conditioned by the rate at which data involved in the solver can be prepared, explaining the efficiency of grid-based Eulerian codes developed to date. For example, the multiphysics Godunov code I3DYN uses a fixed grid, so no overhead is required when executing the numerical scheme. On the other hand, simulating moving disordered particles on Exascale architectures is a tremendous challenge, regardless of whether they are tracers for Eulerian methods, super particles for Lagrangian methods or interpolation points for fast multiple moments (FMMs). The rule of thumb is that performance decreases when the number of neighbours increases and when they are unevenly distributed. Our code SHAMROCK is a performance portable framework aiming at hybrid CPU-GPU multinode Exascale architectures. The design of SHAMROCK makes it appealing for with particle-based methods such as SPH (e.g. Hopkins 2015; Price et al. 2018; Springel et al. 2021), while remaining inherently compatible with any distribution of numerical objects (grids and particles) and numerical schemes (grid-based or Lagrangian). Our strategy in SHAMROCK is that the tree used for neighbour search is never updated. Instead, we are aiming for a highly efficient fully parallel tree algorithm that allows on-the-fly building and traversal, for any distribution of cells or particles. The specific nature of GPU architectures calls for a different design from the state-of-the-art methods developed for CPUs (e.g. Gaffon & Rosswog 2011). The simulation domain undergoes an initial partitioning into subdomains, fostering communication and interface exchange through an coarse layer of MPI parallelism. The core of SHAMROCK is its fine layer of parallelism, which consists in processing each subdomain on GPUs using the SYCL standard. This includes a tree algorithm that delivers the performance required for handling any number of objects compatible with current GPU capabilities. It combines state-of-the-art algorithms based on Morton codes (Morton 1966; Warren & Salmon 1993; Lauterbach et al. 2009)

* E-mail: davidclt@univ-grenoble-alpes.fr

© 2025 The Author(s).
Published by Oxford University Press on behalf of Royal Astronomical Society. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

Talk at IWOCCL 2025

13th International Workshop on OpenCL and SYCL

IWOCCL 2025 OpenCL SYCL

Shamrock: Exascale Hydrodynamics for Astrophysics Using SYCL

Timothée David Cléris, University of Grenoble.



April 7–11, 2025 | Heidelberg, Germany | iwocl.org

Shamrock: Exascale Hydrodynamics for Astrophysics Using SYCL

IWOCCL 827 subscribers

Like Share Save



Github public repo
CECILL License

- Public development on GitHub
- Hosted by the Shamrock-code organization
- $\simeq$ 210.000 lines of code

A screenshot of the Shamrock GitHub repository page. The page shows the repository name 'Shamrock', the license 'CECILL-2.1 license', and the status 'On Push passing'. The description states: 'Shamrock is a general purpose HPC hydrodynamics simulation code focused on astrophysical contexts. The code is C++17 by default, and all accelerated parts are done using SYCL, which can be directly compiled to native CUDA, ROCm, OpenMP, OpenCL and much more. Shamrock also supports multiple GPUs and heterogeneous clusters using MPI. This code aims to be Modern, Modular, Fast, Portable.' The 'Packaging status' section shows versions for Homebrew (2025.10.0) and Spack (2025.05.0). The right sidebar shows the organization 'Shamrock-code' with 26 repositories, 45 stars, and 17 forks. The 'Pinned' section shows the 'Shamrock' repository and a 'containers' repository. The 'Repositories' section lists 'Shamrock' and 'shamrock-code.github.io'.



PODCAST (PI: G. Laibe)

Dust2Planets (PI: F. Ménard)

PEPR Origins (PI: A. Morbidelli)



⇒ 1 of the 3 Exascale codes
identified in the project

EuroHPC CoE Space project



⇒ port RAMSES AMR solver
in the Shamrock framework



⇒ Investigate/optimize Shamrock
Exascale scalability
& MPI tuning

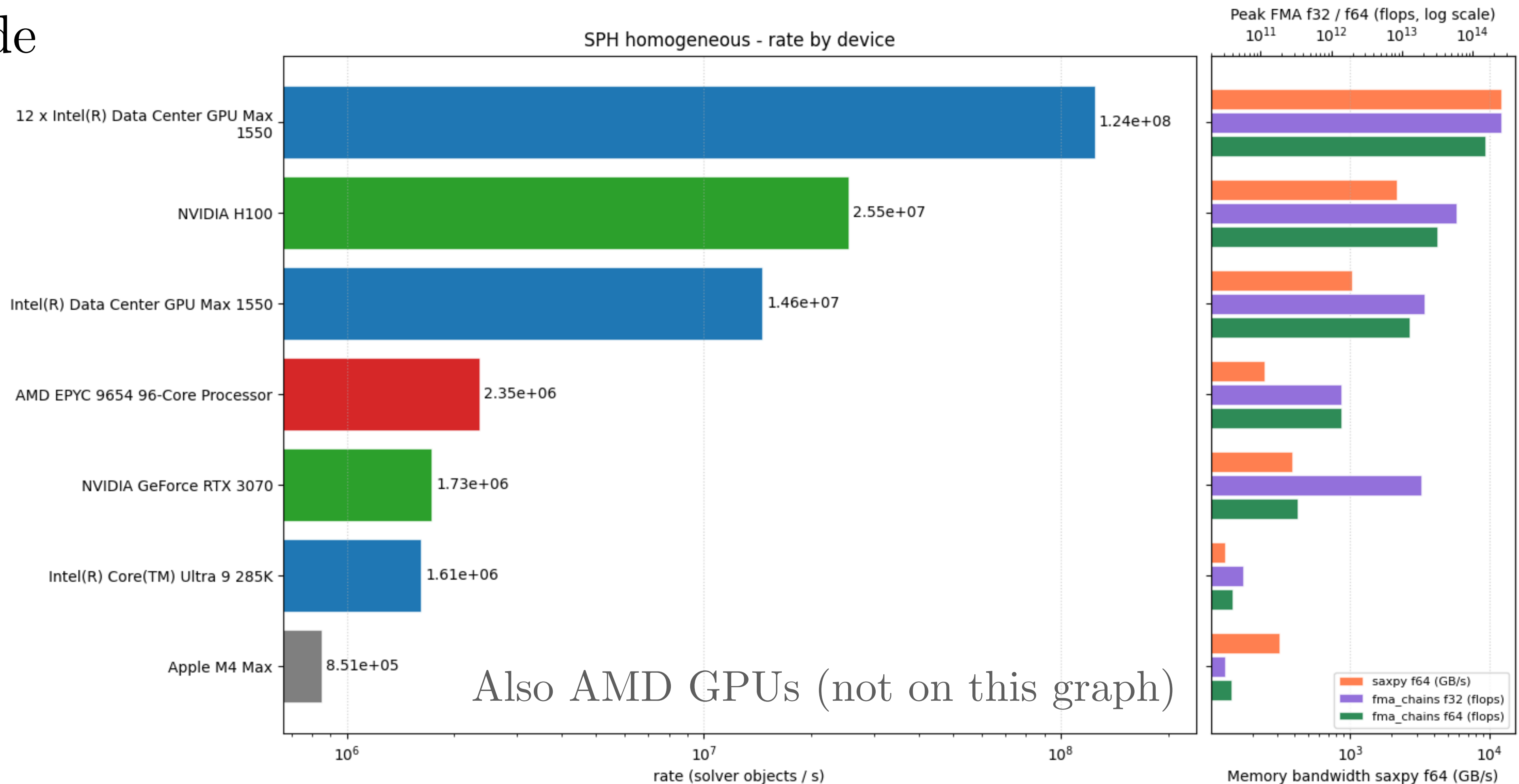
AIRBUS

⇒ Using Shamrock for aerodynamic
design optimisation

Performance overview

SPH Weak scaling test

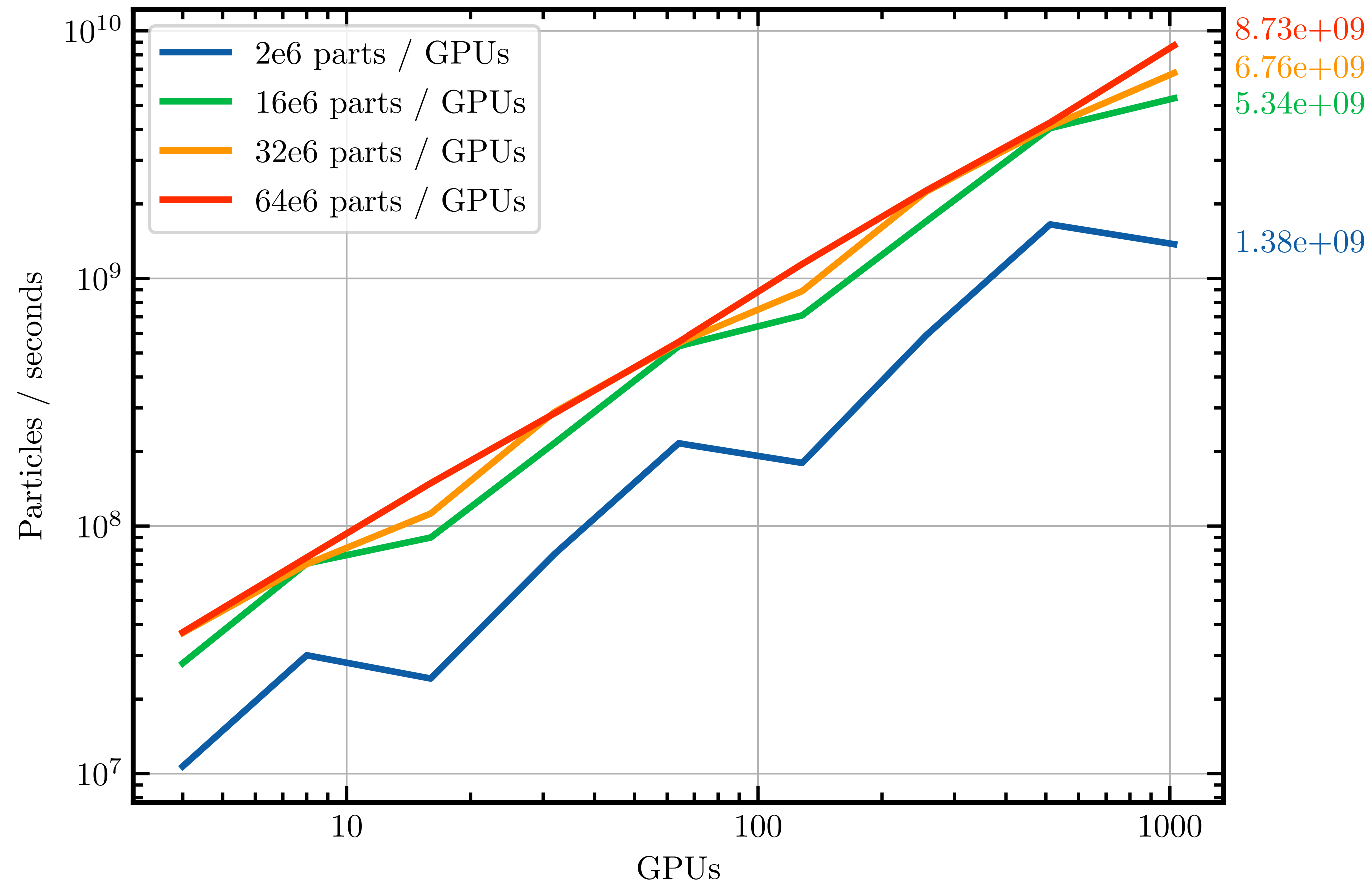
Aurora node



Aurora = icpx | Apple M4 max = Acpp OpenMP | Other = Acpp SSCP

Weak scaling test:

Sedov Taylor blast 32M part / MPI ranks



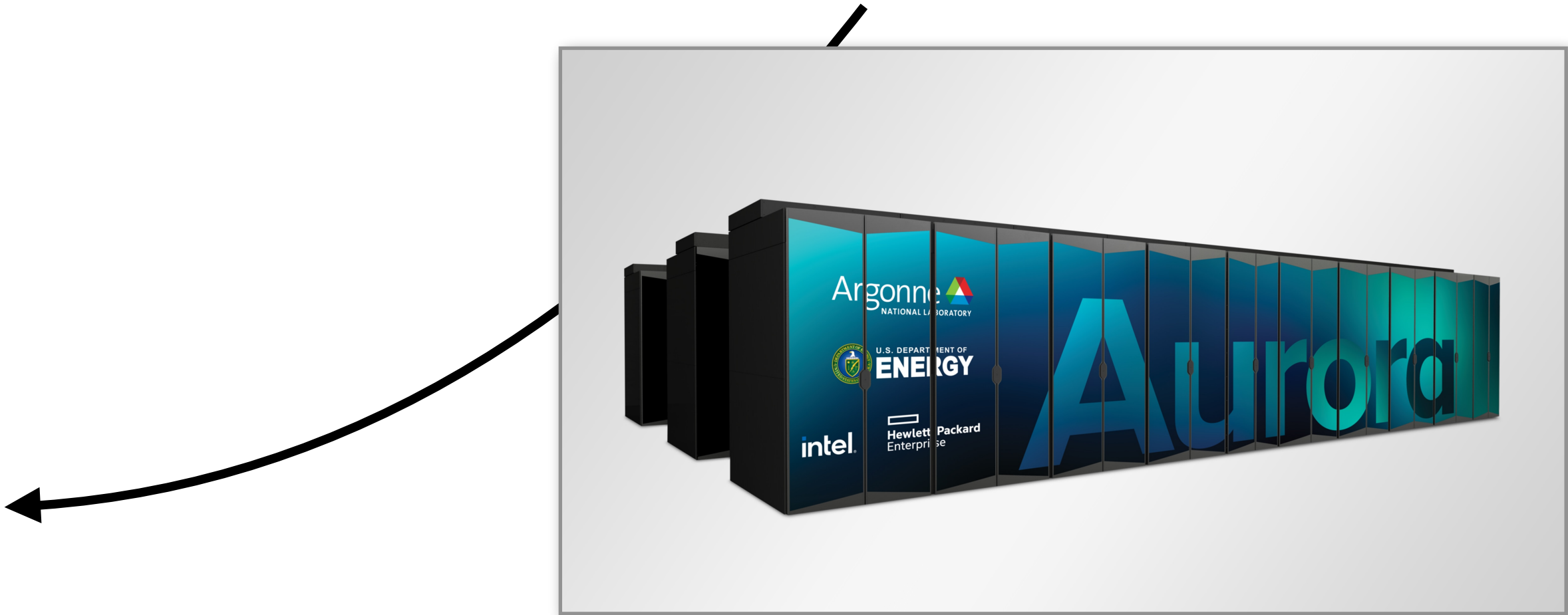
Shamrock :

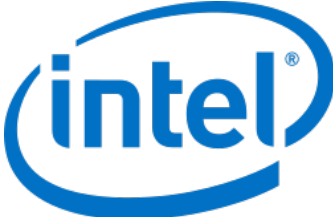
- 65.000.000.000 particles (4000^3)
- 9G part / seconds
- 7 sec / iterations
- 2048 GPU tiles (MI250x)

Top 500 list :

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	El Capitan - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
5	HPC6 - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461

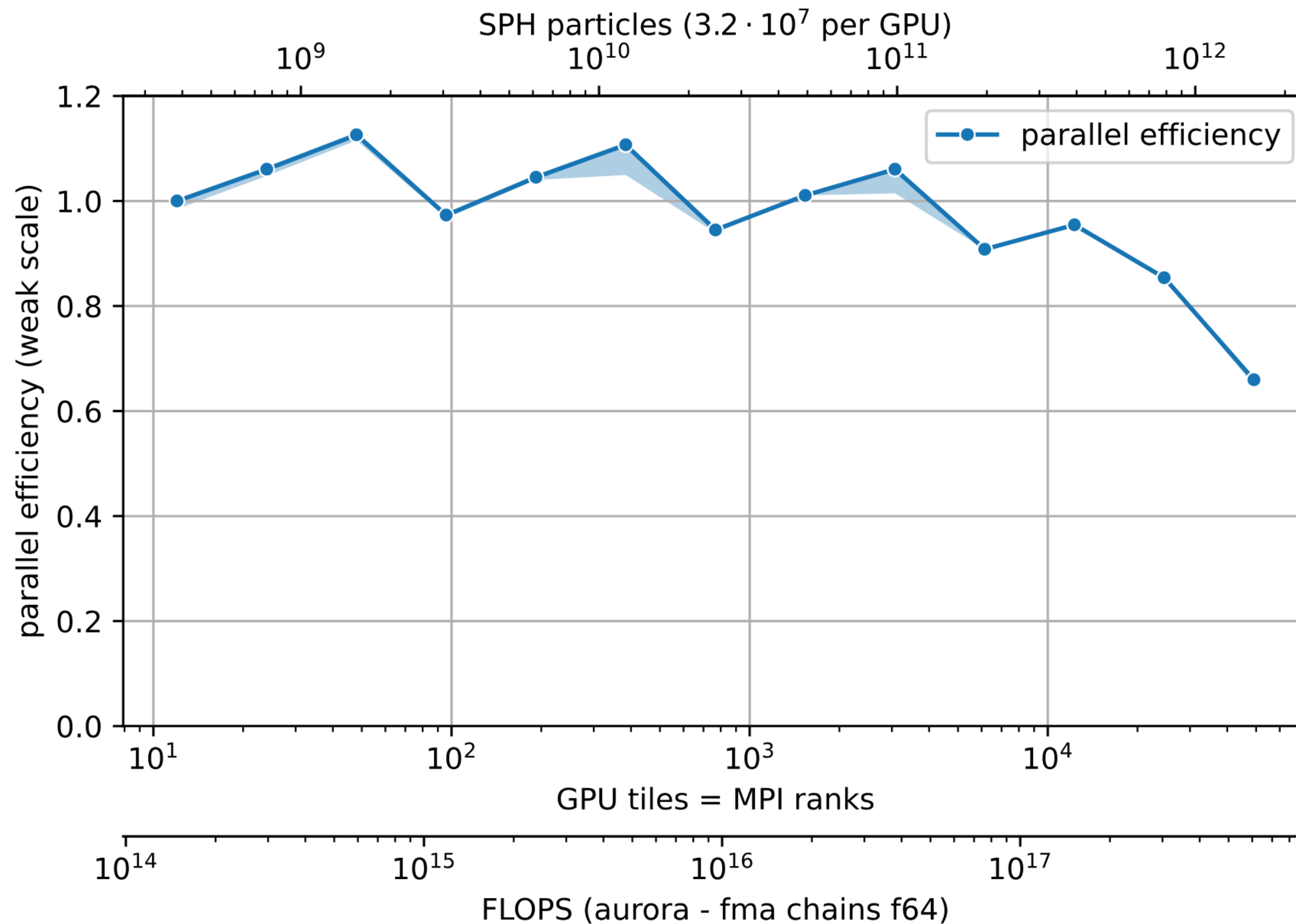
We have access to this one



120 000 GPU_s 

1 Exaflop

10 MCPU hours (GENCI) / realtime hours



Shamrock :

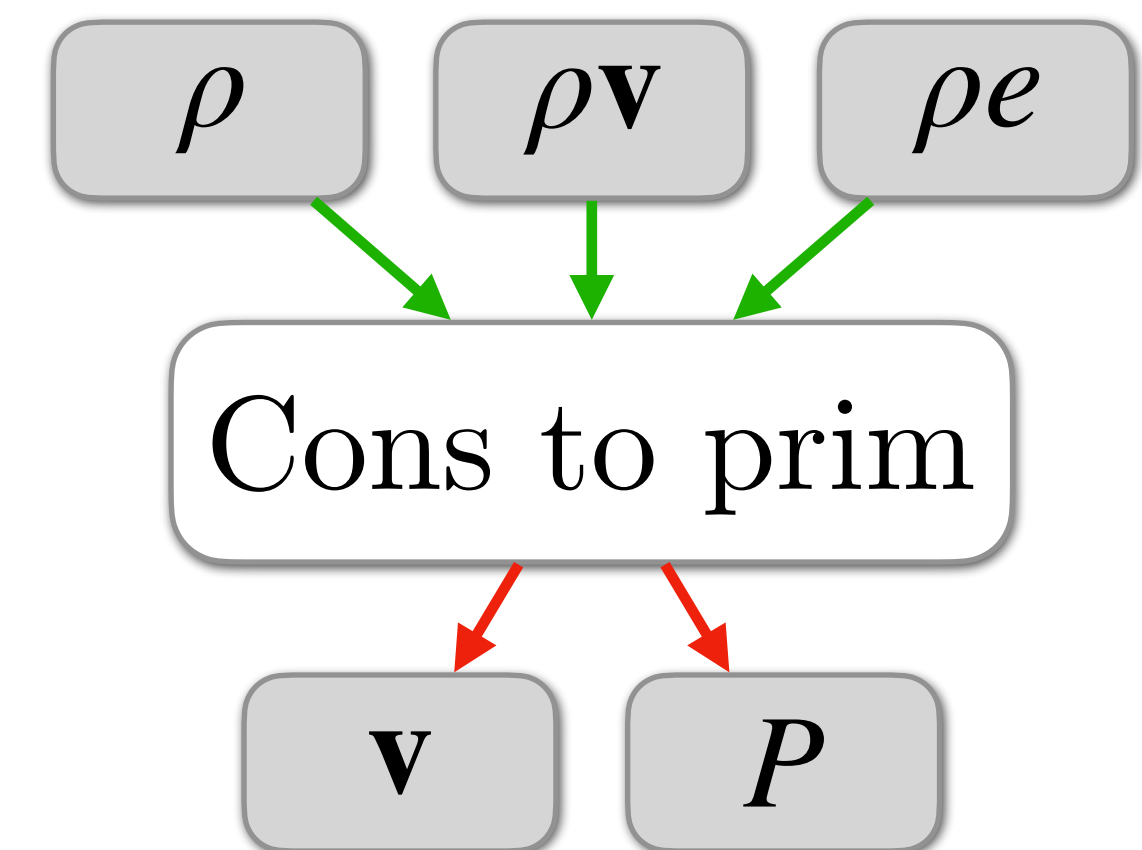
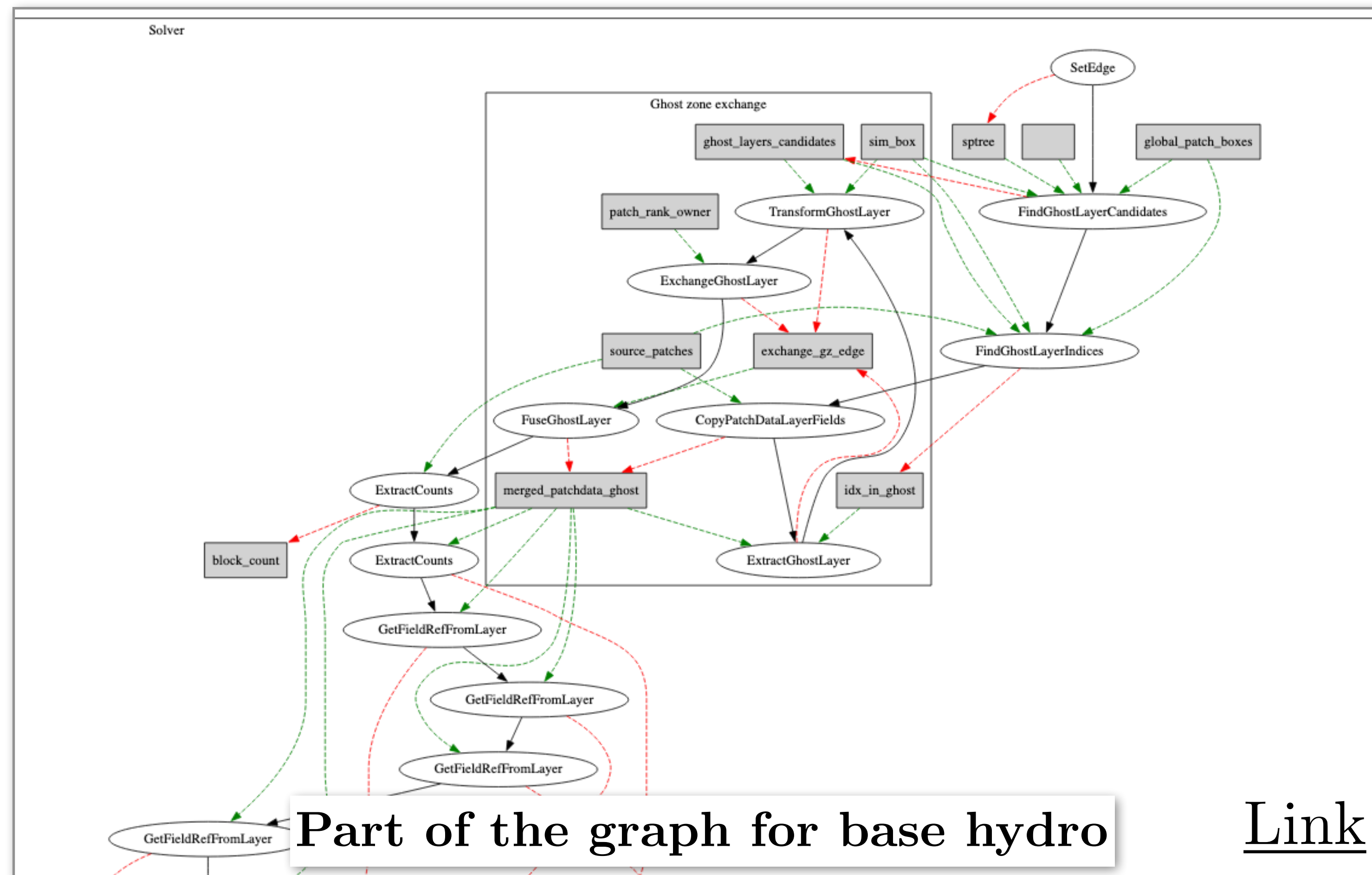
- 1.574.517.452.288 particles
- 328 Gpart / seconds
- 5 sec / iterations
- 49152 GPU tiles
- 1.3 PB of VRAM

Our  usage

- Support AdaptiveCpp (SMCP & SSCP) & icpx / Intel llvm, all backends
- Only SYCL-2020 without vendor extensions (not enough human time)
- Used through CMake with changing CMAKE_CXX_COMPILER always treat SYCL implementation as custom compiler. (Very efficient to make Aksel angry)
- Preconfigured environment scripts for many machines

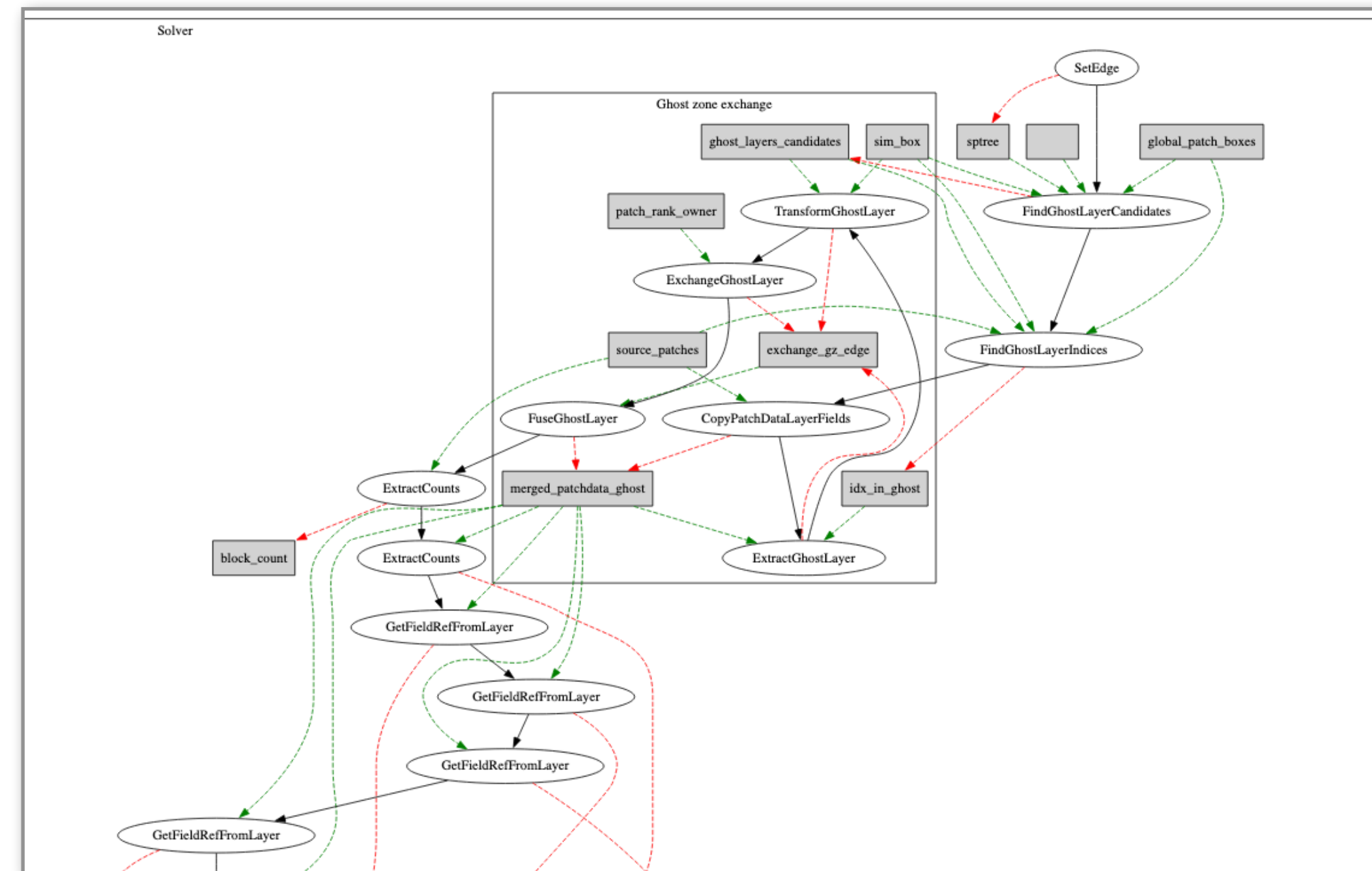
```
1 ./env/new-env --machine argonne.aurora --builddir build --
2 cd build
3 source ./activate # load the correct modules and ENV vars
4 shamconfigure      # alias to the correct cmake command
5 shammake           # alias to ninja build (or make if ninja is not avail)
```

Hydro Solver $\simeq$ DAG (with sub cycles)



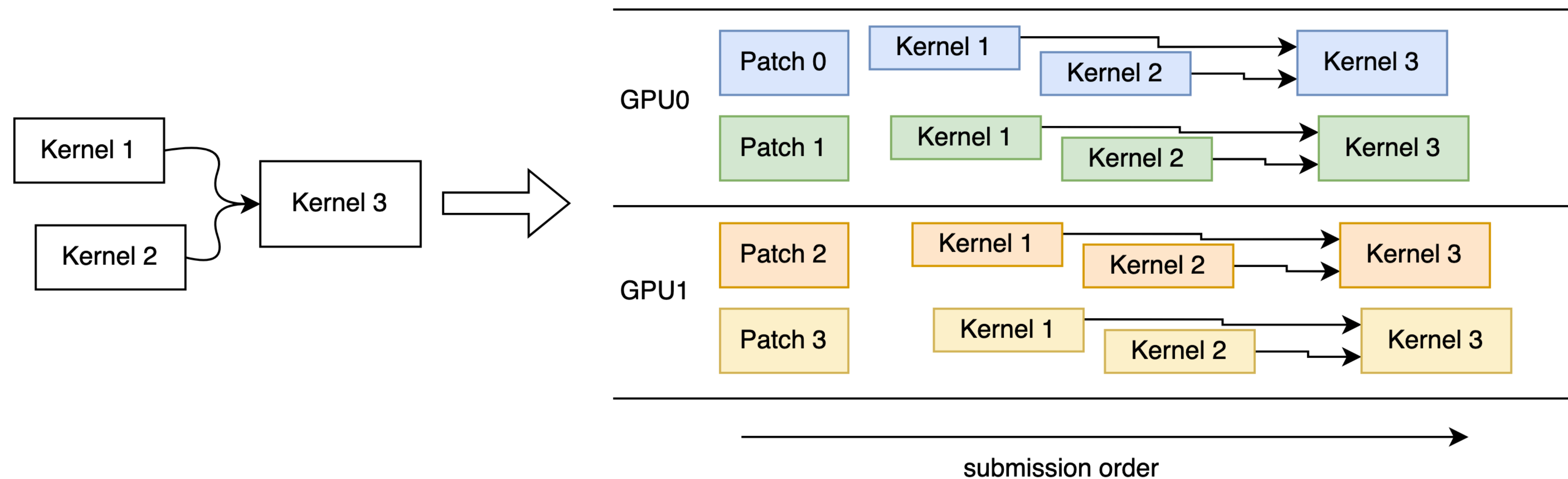
Can dump the **TEX** & **graph** of the solver

Very complex dependency chain & too many possibilities!



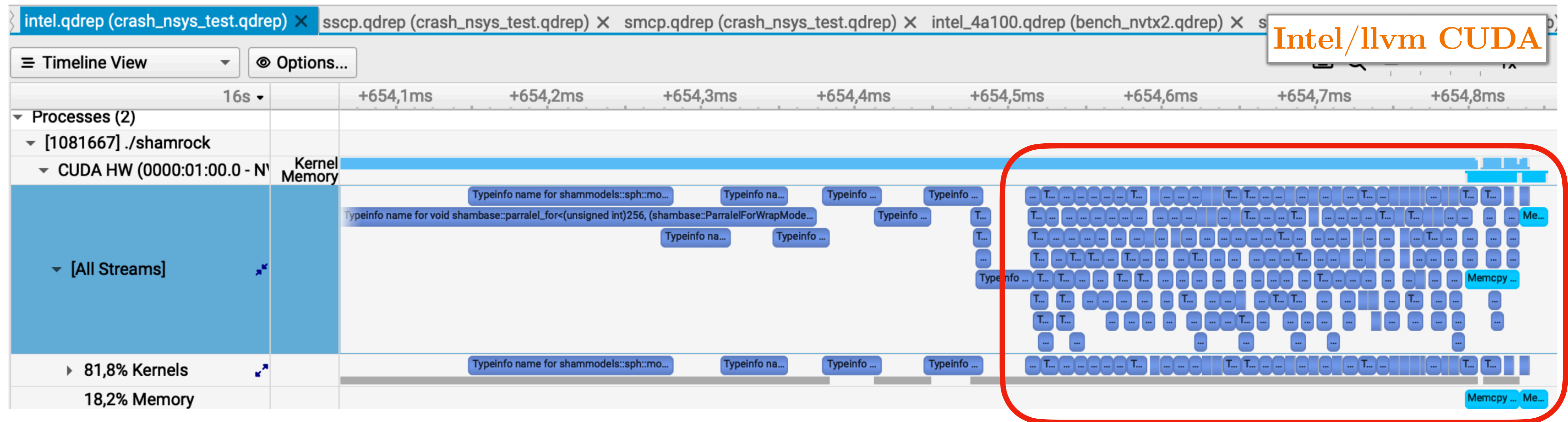
- Want to leverage **concurrent kernel execution**.
- **Impossible to track all data flows** (depends on enabled physics & choice of solvers)
 - No explicit scheduling → out-of-order queue.
- **Avoid user error in scheduling** → implicit scheduling → `sycl::buffer`

Kernels are submitted for each patches on each GPUs
using out-of-order queue ($\simeq 1$ DAG per patch)

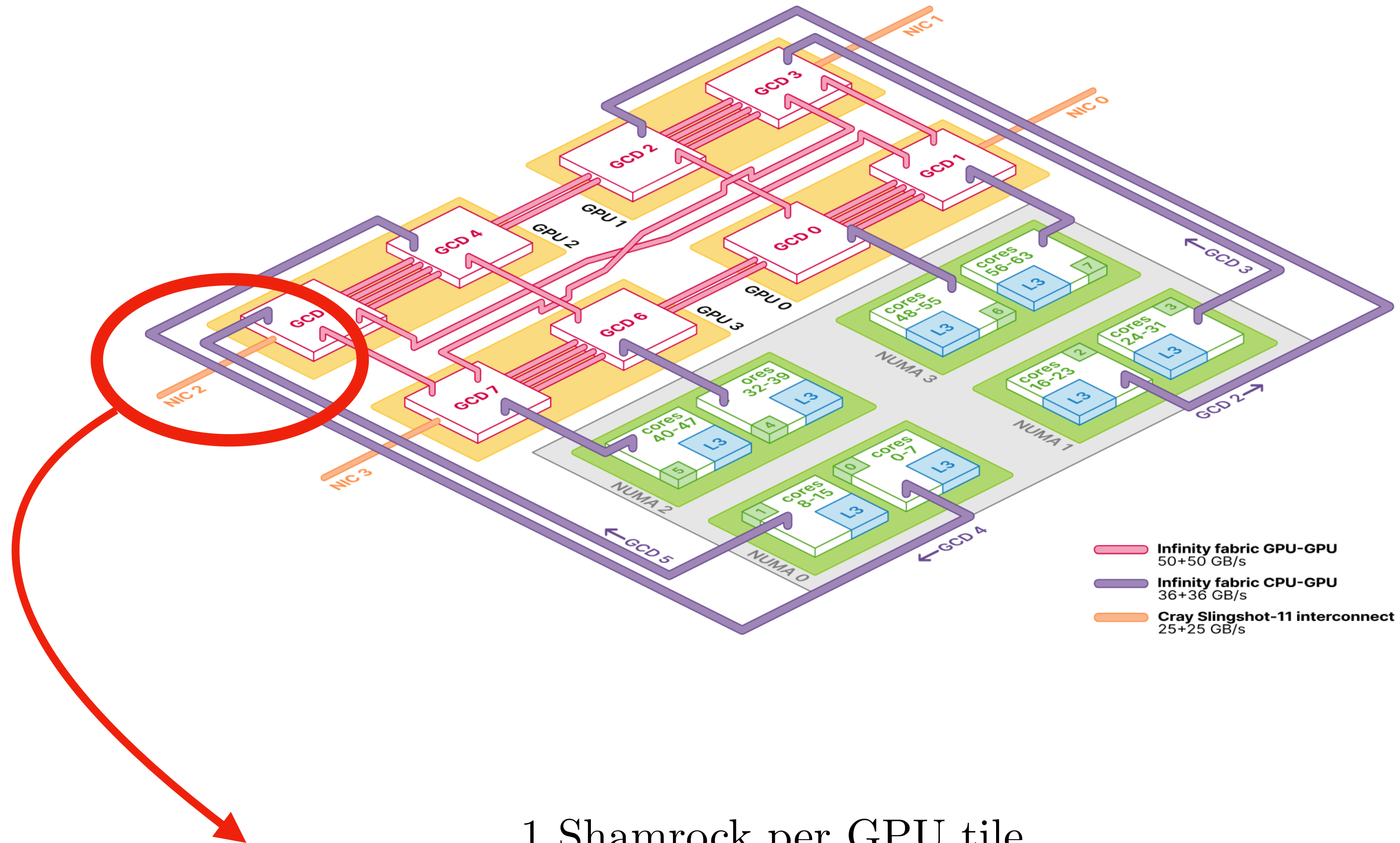


Patch = part of the simulation domain

Profiling of a small section of the graph :

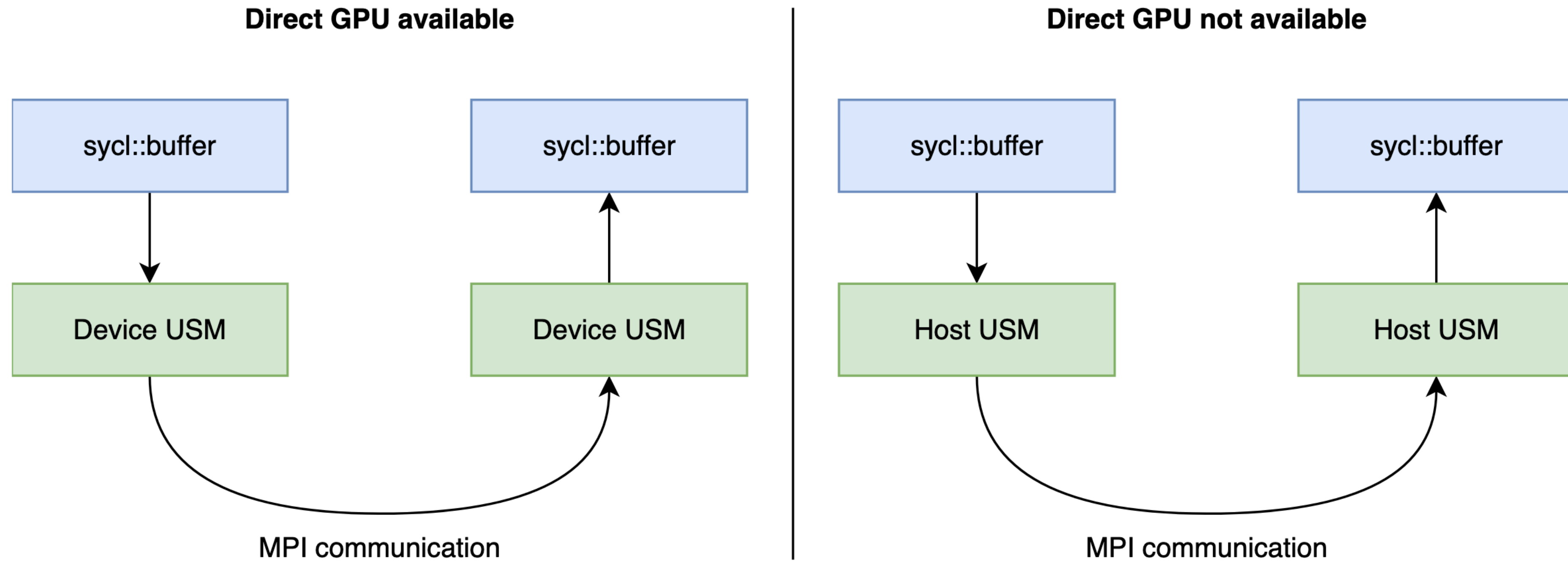


Heavy use of out-of-order 🙌



1 Shamrock per GPU tile.

MPI communications can bypass the CPUs (direct GPU)



Need extra copy since MPI wants a USM pointer for DirectGPU.

Algorithms libs are very fragmented in SYCL

⇒ Custom implementation for each algorithm

Algorithm	Implementation
Prefix scan	Decoupled scan (Duane, Garland 2016)
Sort	Bitonic sort \ Onesweep (Adinets, Merrill, 2022)

Segsort, histogram, reduction, etc. (Still looking for a good FFT)

Can switch implementations at runtime using Shamrock API calls in Python

3D physics $\Rightarrow$ 3D math $\Rightarrow$ sycl::vec

```
Tvec w = sycl::cross(x: k, y: xyz_a);
Tvec wv = sycl::cross(x: k, y: vxyz_a);
// Rodrigues' rotation formula
xyz_a = xyz_a * sycl::cos(x: effective_inc) + w * sycl::sin(x: effective_inc)
|      | + k * sycl::dot(x: k, y: xyz_a) * (1. - sycl::cos(x: effective_inc));
vxyz_a = vxyz_a * sycl::cos(x: effective_inc) + wv * sycl::sin(x: effective_inc)
|      | + k * sycl::dot(x: k, y: vxyz_a) * (1. - sycl::cos(x: effective_inc));
```

Good enough & simple usage for users

Migrating away from `sycl::buffer` to USM

- Want to stick to SYCL-2020 API (no extensions for buffer-USM interop)
- Interop with native APIs, especially **MPI**.
- Allow for more memory manipulations (e.g. reusing, fragmenting, ...)
- Buffers + memcpy do not allow partial memory copies (sub-buffers unsupported).
- Can use `__restrict`
- Lower submission latency (see Gromacs)
- Avoid performance regression due to `sycl::accessor` overhead

Basic GPU kernel ($b = 2*a$)

```
1  sycl::buffer<int> buf_a(N);
2  sycl::buffer<int> buf_b(N);
3
4  sycl::queue{}.submit([&](sycl::handler &h) {
5      sycl::accessor a{buf_a, h, sycl::read_only};
6      sycl::accessor b{buf_b, h, sycl::write_only, sycl::no_init};
7
8      h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> indx) {
9          b[indx] = 2 * a[indx];
10     });
11 });
```

Basic GPU kernel ($b = 2*a$)

```
1  sycl::buffer<int> buf_a(N);
2  sycl::buffer<int> buf_b(N);
3
4  sycl::queue{}.submit([&](sycl::handler &h) {
5      sycl::accessor a{buf_a, h, sycl::read_only};
6      sycl::accessor b{buf_b, h, sycl::write_only, sycl::no_init};
7
8      h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> indx) {
9          b[indx] = 2 * a[indx];
10     });
11 });
```

- `sycl::buffer` → `sham::DeviceBuffer`
- `sycl::accessor` → USM pointers

In an ideal world ...

```
1 namespace sham {
2     template<class T, USMKindTarget target = device>
3     class DeviceBuffer; // skip implementation
4 }
5
6 sham::DeviceBuffer<int> buf_a(N);
7 sham::DeviceBuffer<int> buf_b(N);
8
9 sycl::queue{}.submit([&](sycl::handler &h) {
10     const int* a = buf_a.get_read_access(h);
11     int* b       = buf_b.get_write_access(h);
12
13     h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> indx) {
14         b[indx] = 2 * a[indx];
15     });
16 });
```

```
1 const T* sham::DeviceBuffer<T>::get_read_access(sycl::handler &h) const {
2     h.depends_on(this->read_dependencies());
3     this->register_read_access(/* get current event */);
4     return this->internal_pointer();
5 }
```

In an ideal world ...

```
1 const T* sham::DeviceBuffer<T>::get_read_access(sycl::handler &h) const {  
2     h.depends_on(this->read_dependencies());  
3     this->register_read_access(/* get current event */);  
4     return this->internal_pointer();  
5 }
```

⚠ No way of getting the resulting event inside a `.submit(...)`.
→ Must move the accesses out of the `.submit(...)`

→ Must move the accesses out of the `.submit(...)`

```
1 sham::DeviceBuffer<int> buf_a(N);
2 sham::DeviceBuffer<int> buf_b(N);
3
4 sham::EventList deps{};
5 const int* a = buf_a.get_read_access(deps); //buf_a left in partial state
6 int* b      = buf_b.get_write_access(deps); //buf_b left in partial state
7
8 sycl::event e = sycl::queue{}.submit([&](sycl::handler &h) {
9     h.depends_on(deps.events());
10    // ... kernel code ...
11 });
12
13 buf_a.complete_event_state(e);
14 buf_b.complete_event_state(e);
```

```
1 ////////////// simplified implementation //////////////////
2 inline const T *get_read_access(sham::EventList &deps) const {
3     events_hdl->read_access(deps);
4     return hold.template ptr_cast<T>();
5 }
6
7 void complete_event_state(sycl::event e) const {
8     events_hdl->complete_state(e);
9 }
10
```


→ Must move the accesses out of the `.submit(...)`

```
1 sham::DeviceBuffer<int> buf_a(N);
2 sham::DeviceBuffer<int> buf_b(N);
3
4 sham::EventList deps{};
5 const int* a = buf_a.get_read_access(deps);
6 int* b      = buf_b.get_write_access(deps);
7
8 deps.consumed = true;
9 sycl::event e = sycl::queue{}.submit([&](sycl::handler &h) {
10     h.depends_on(deps.events());
11     // ... kernel code ...
12 });
13
14 buf_a.complete_event_state(e);
15 buf_b.complete_event_state(e);
```

- `.complete_event_state(...)` throw if no prior access.
- `EventList` destructor raise an error if not consumed.

→ Wrap it !

```
sham::DeviceBuffer<int> buf_a(N);
sham::DeviceBuffer<int> buf_b(N);

sham::EventList deps{};
const int* a = buf_a.get_read_access(deps);
int* b       = buf_b.get_write_access(deps);

sham::DeviceQueue{}.submit(deps, [&](sycl::handler &h) {
    h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> indx) {
        b[indx] = 2 * a[indx];
    });
});

buf_a.complete_event_state(e);
buf_b.complete_event_state(e);
```

Very few changes to migrate:

- Add a sham::EventList
- Move accesses out of the .submit(...)
- Use a sham::DeviceBuffer instead of a sycl::buffer
- Use a sham::DeviceQueue instead of a sycl::queue

```
1 sham::DeviceBuffer<int> buf_a(N);
2 sham::DeviceBuffer<int> buf_b(N);
3
4 sycl::event e = khr::make_event(myQueue.get_context());
5
6 // No partial state anymore !!! => can do scoped accesses
7 sham::EventList deps{};
8 const int* a = buf_a.get_read_access(deps, e);
9 int* b      = buf_b.get_write_access(deps, e);
10
11 auto reqs = khr::requirements(khr::signal_event(e), deps.get_requirement());
12 khr::launch(myQueue, sycl::range{N}, reqs, [=](sycl::id<1> index) {
13     // ... kernel code ...
14 });
```

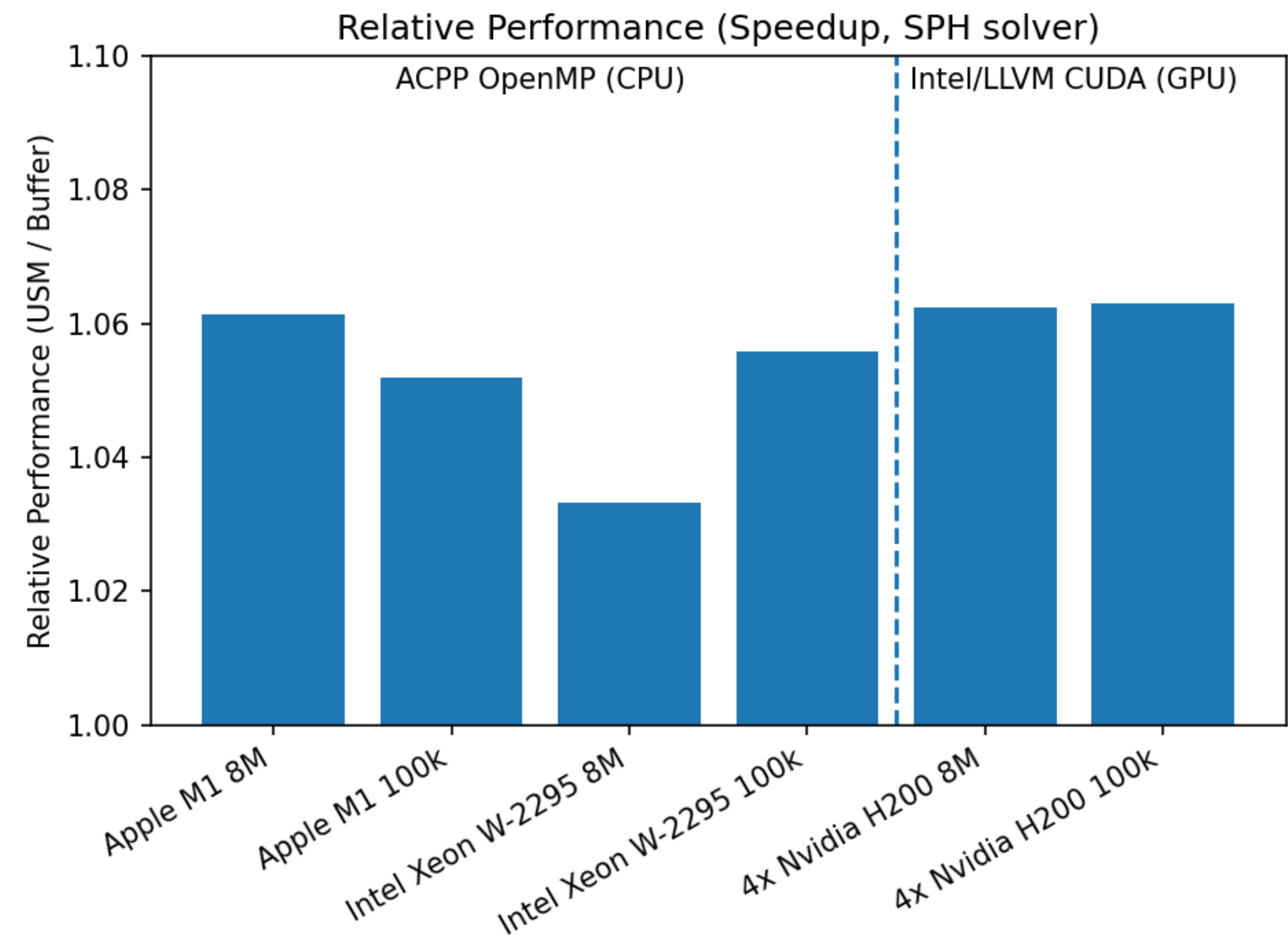
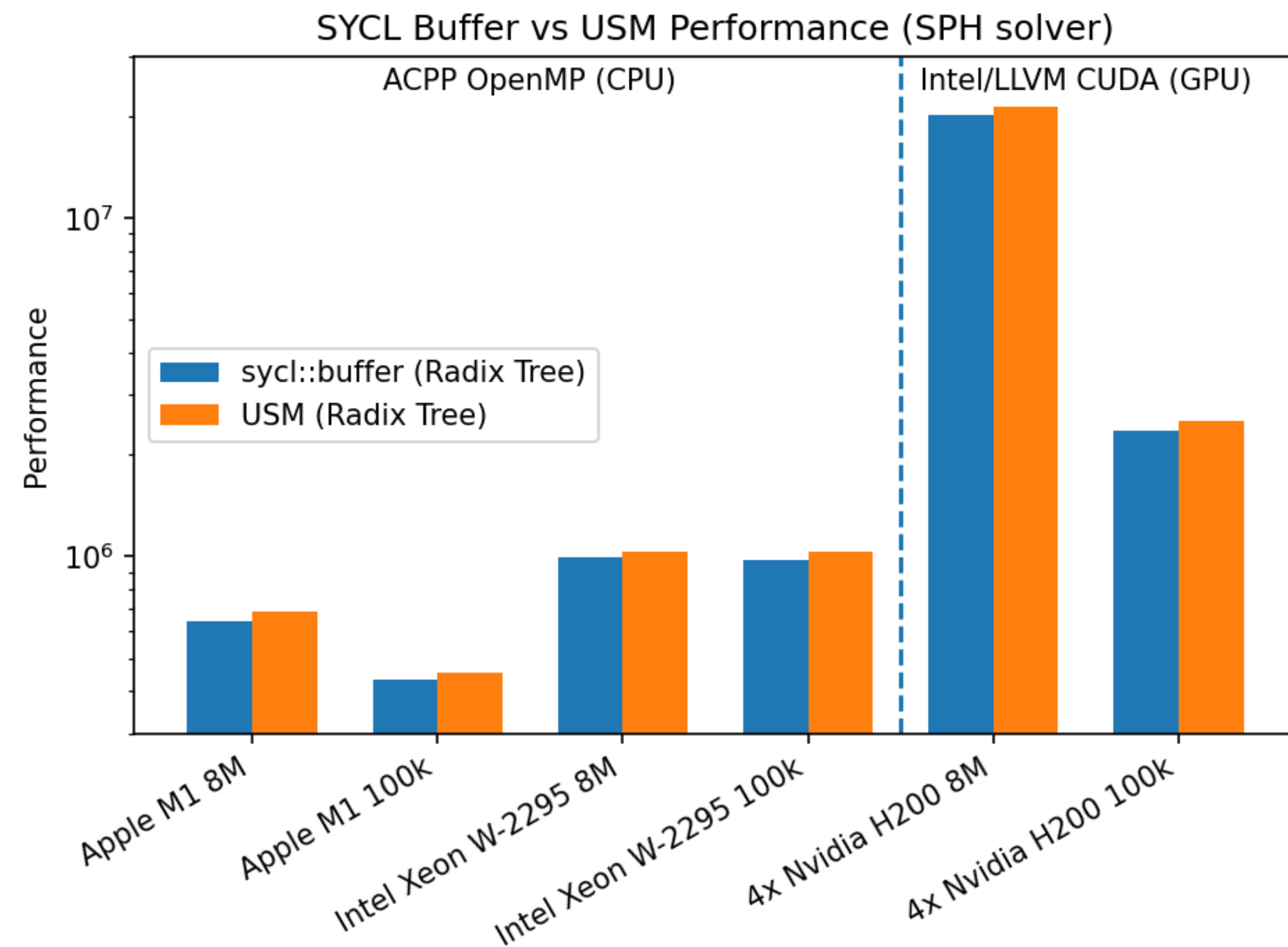
We will use it as soon as we can !

→ Wrap it (even more) !

```
1 sham::kernel_call(  
2   get_queue(),                // device queue  
3   sham::MultiRef{rho, rhov, rhoe}, // read accesses  
4   sham::MultiRef{vel, P},      // write accesses  
5   N,                          // number of threads  
6   [gamma](  
7     u32 i,  
8     const f64 *__restrict rho,  
9     const f64_3 *__restrict rhov,  
10    const f64 *__restrict rhoe,  
11    f64_3 *__restrict vel,  
12    f64 *__restrict P) {  
13  
14    auto prim_state = shammath::cons_to_prim(  
15      shammath::ConsState<f64_3>{rho[i], rhoe[i], rhov[i]}, gamma);  
16  
17    SHAM_ASSERT(prim_state.press >= 0.0);  
18  
19    vel[i] = prim_state.vel;  
20    P[i]    = prim_state.press;  
21  });
```

Still ongoing $\Rightarrow$ hard to isolate

Impact on the Radix Tree (BVH + compressed leaves):



Gains (5%) = lower latency + small performance gains in kernels

Migrating away from `sycl::buffer` to USM

- Need to use wrappers to ease the migration.
- Hard to do a 1:1 wrapper since no access to the `.submit(...)` event.
- Performance gains $\simeq 5\%$ in our case.

We can work to release it if anyone is interested

Almost everything in Shamrock is async, except for ...

Currently forced synchronizations :

- Allocations (no `sycl::malloc_device_async` / `free_async`)
- No MPI integration in SYCL DAG
- No `sycl::future` to recover async data from GPU (e.g. `memcpy` + `alloc`)

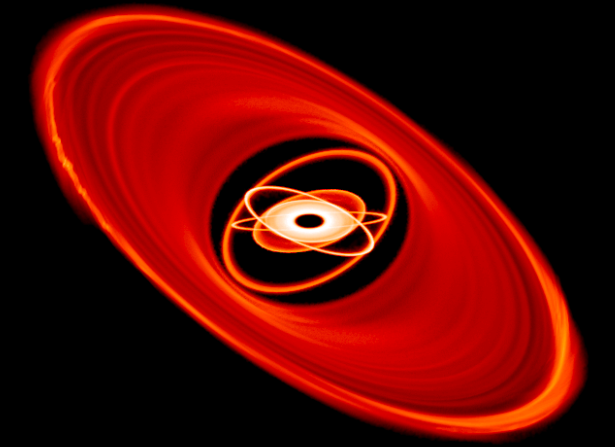
Our usage

- Out-of-order `sycl::queue`
- ~~`sycl::buffer`~~ (migrated to USM, +5% perf)
- `sycl::vec`
- SYCL implementation used through `CMAKE_CXX_COMPILER`
- Wish for more async capabilities in SYCL

$t = 1400.250$ (code unit) $dt = 0.026$ (code unit)

If you wait longer ... + more resolution

Y. Lapeyre in prep



(c) 2025 Yona Lapeyre

log column density [g/cm²]
3 4 5 6

