Porting Eiron's domain-decomposed Monte Carlo algorithm to GPUs

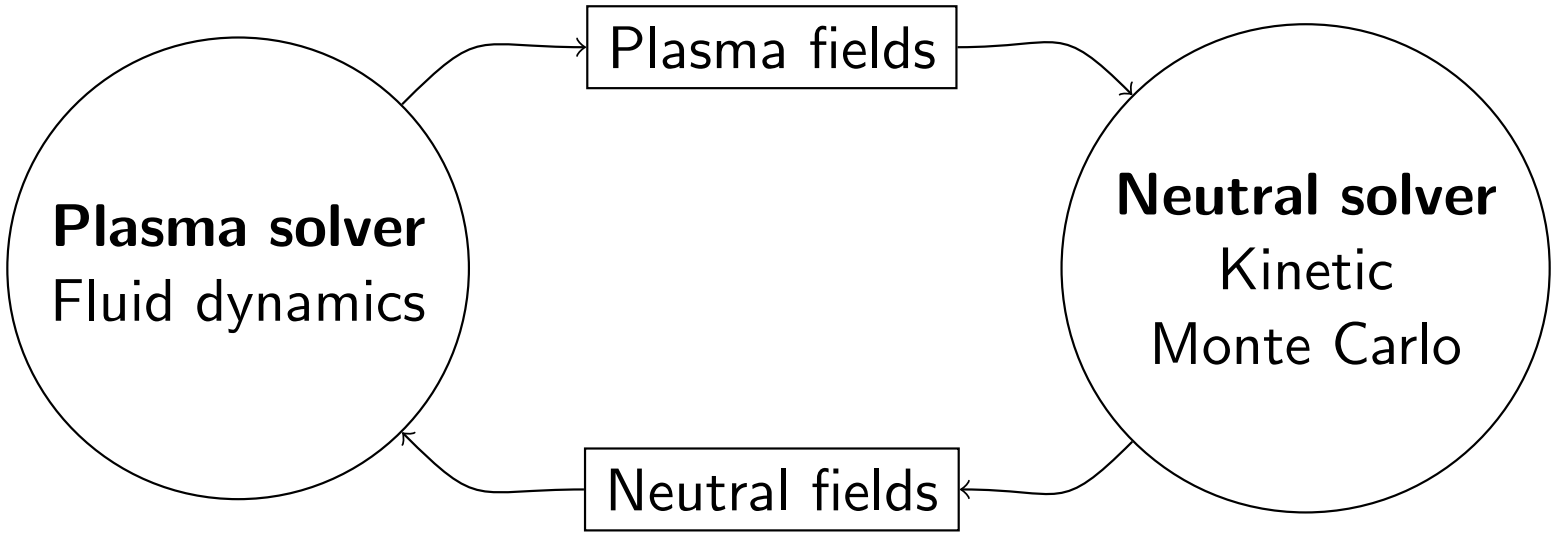
Oskar Lappi

27.10.2026

Presentation outline

- Recap of Eiron and particle Monte Carlo
- Correction to last years presentation (performance bug)
- GPU port of the Monte Carlo API
- GPU algorithms, what's done and todos

Background: The edge and scrape-off-layer (SOL)



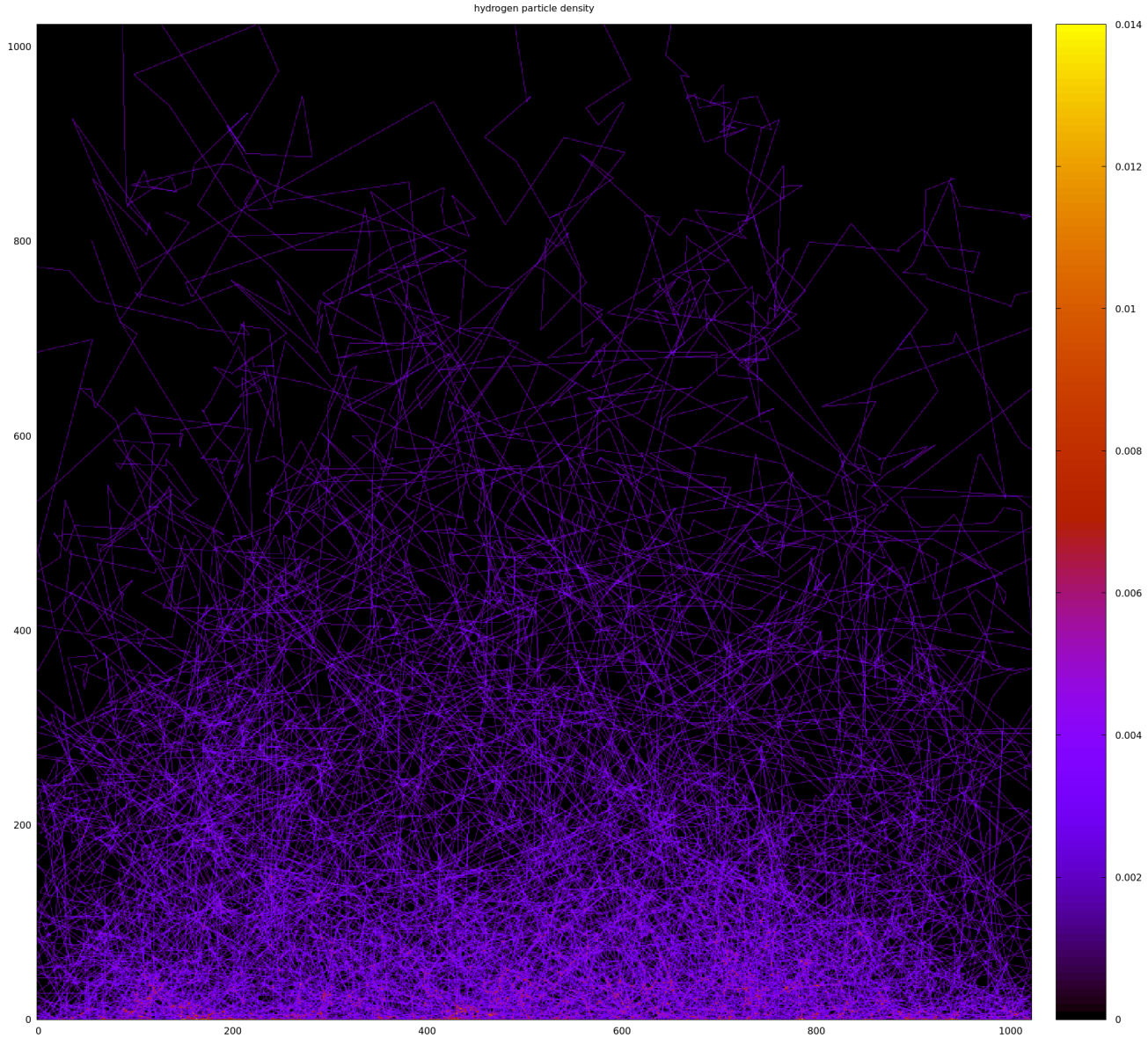
Simulating the edge and SOL regions in a fusion reactor is often done with a fluid-kinetic model

- Plasma has a short mean free path (mfp), fluid model valid
- Neutral gas surrounding it has a long mfp, fluid model invalid

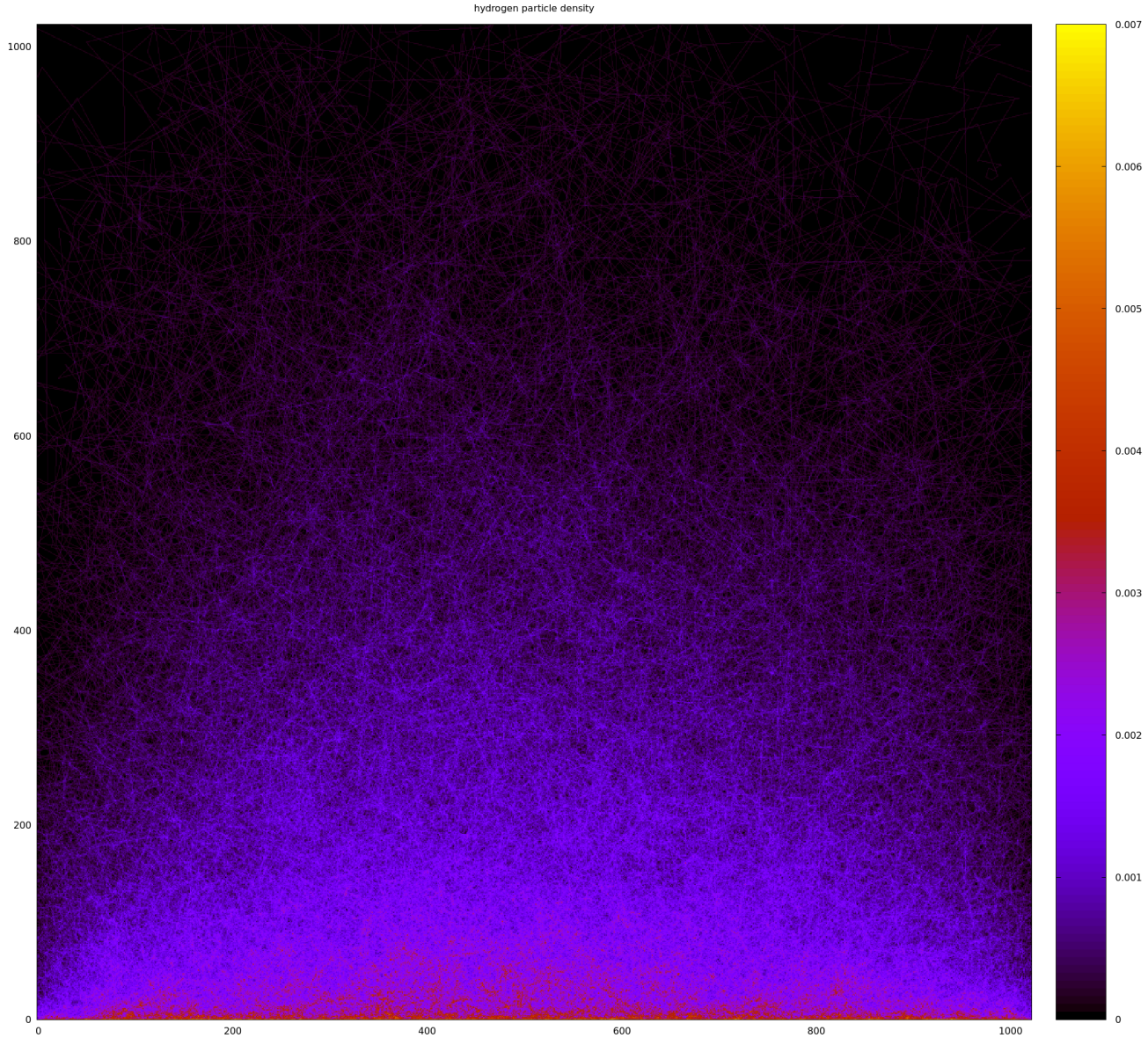
---

- Plasma simulated using CFD, DD & GPU easier
- Neutrals simulated using MC, DD & GPU less easy
- EIRENE is the kinetic neutral solver of choice in EUROfusion

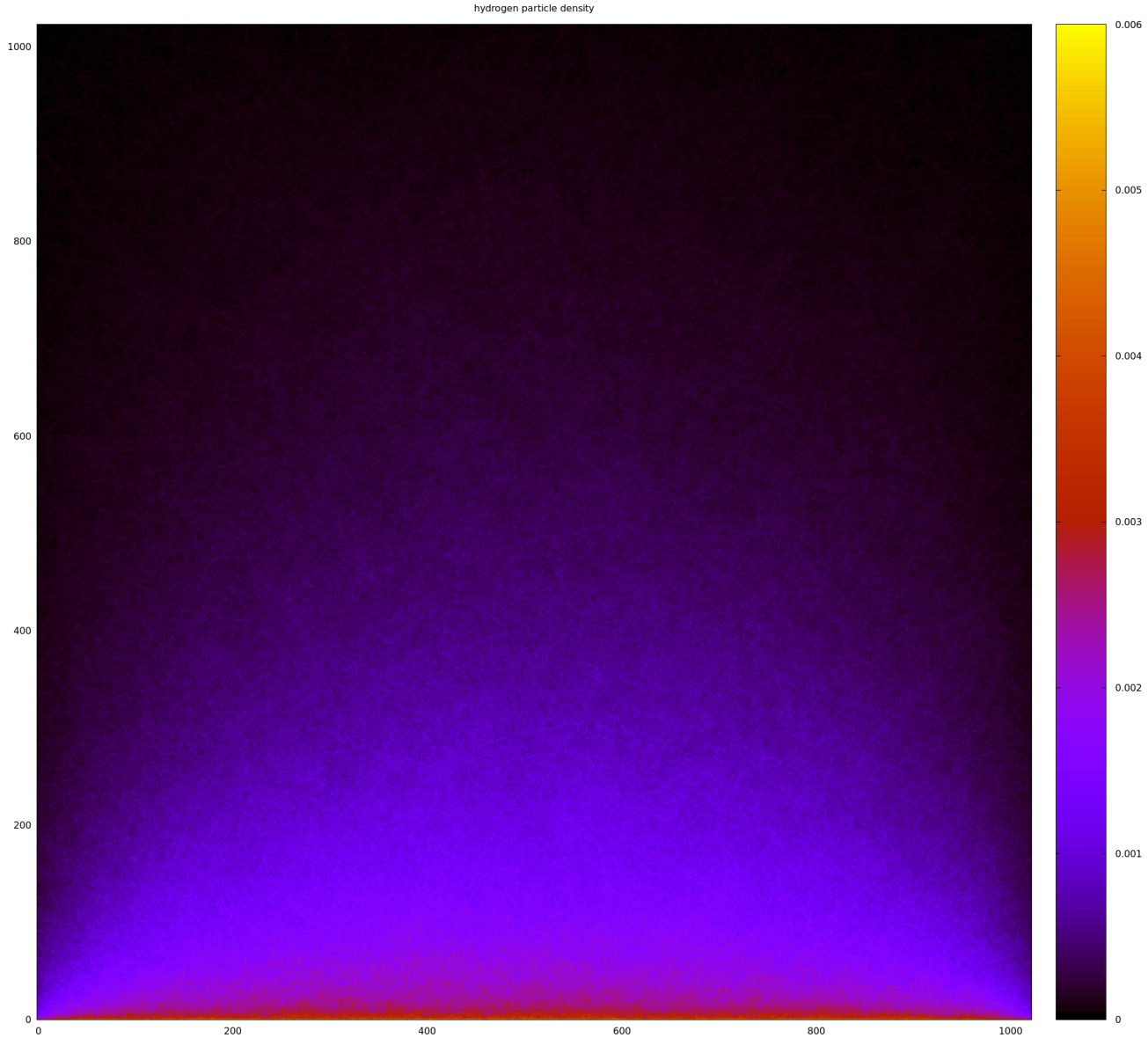
Wall source example, 1K parts



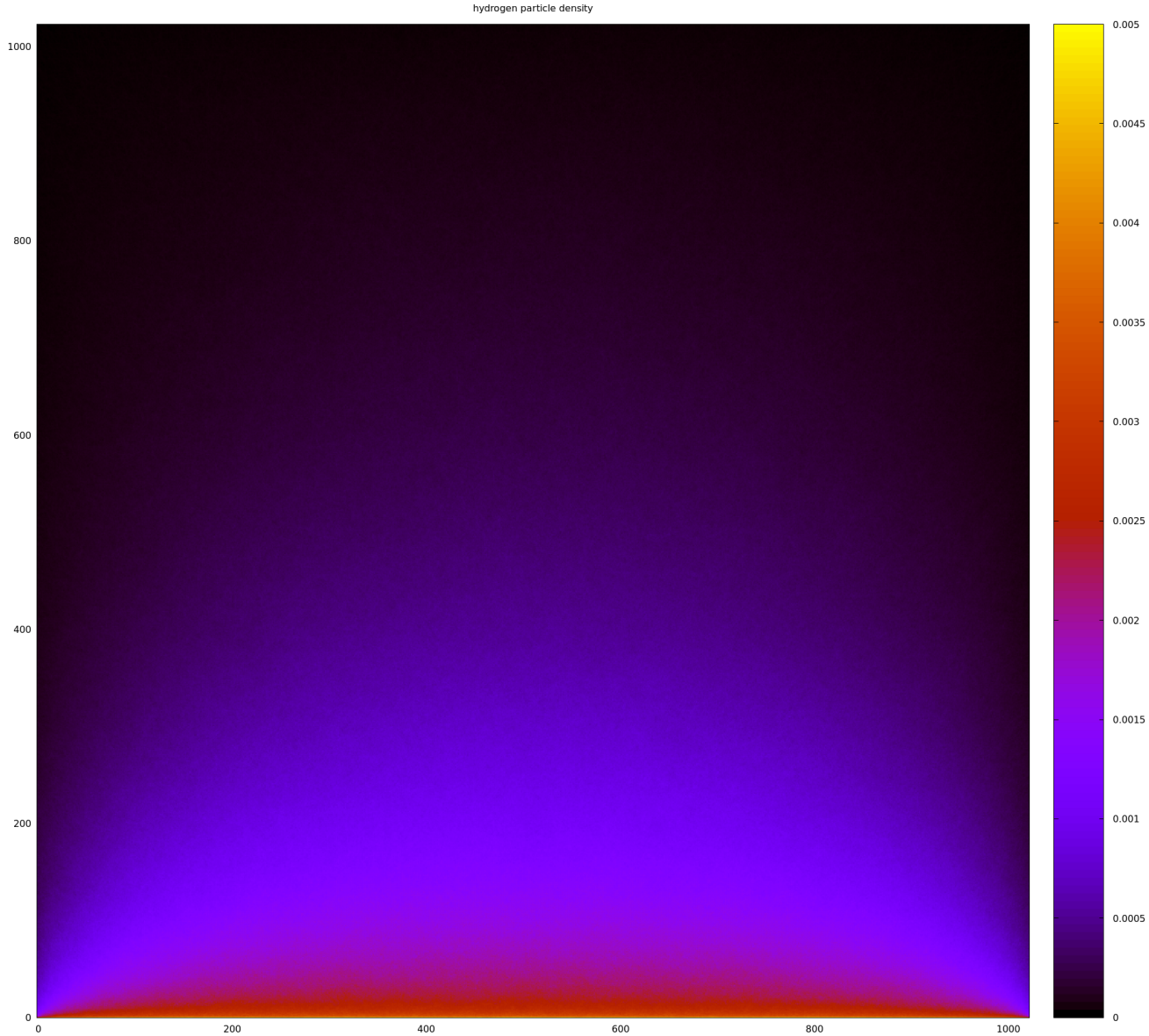
Wall source example, 10K parts



Wall source example, 100K parts



Wall source example, 1M parts



Background: EIRENE and Eiron

- EIRENE is FORTRAN, does not support GPUs or domain-decomposed Monte Carlo (DDMC)
- $\Longrightarrow$ EIRENE is the bottleneck in the fluid-kinetic models

---

- Eiron is a simpler Monte Carlo model built to develop algorithms that support DDMC and GPU
- DDMC is done, and a performance study has been completed (although last years results had a performance bug)
- GPU support added this year, but still needs work

Correction to last years presentation

Correction to last years presentation

- Last year, I presented results with very high superlinear strong scaling
- It turns out we had a severe performance bug. We were *copying the grid* in the simulation loop

Correction to last years presentation

The bug

```
const eiron::grid::CartGrid2D simulation_grid =
  sim_ctx.simulation_subdomain->simulation_grid.at(
    state.particle_event.species_idx
  );
```

The fix

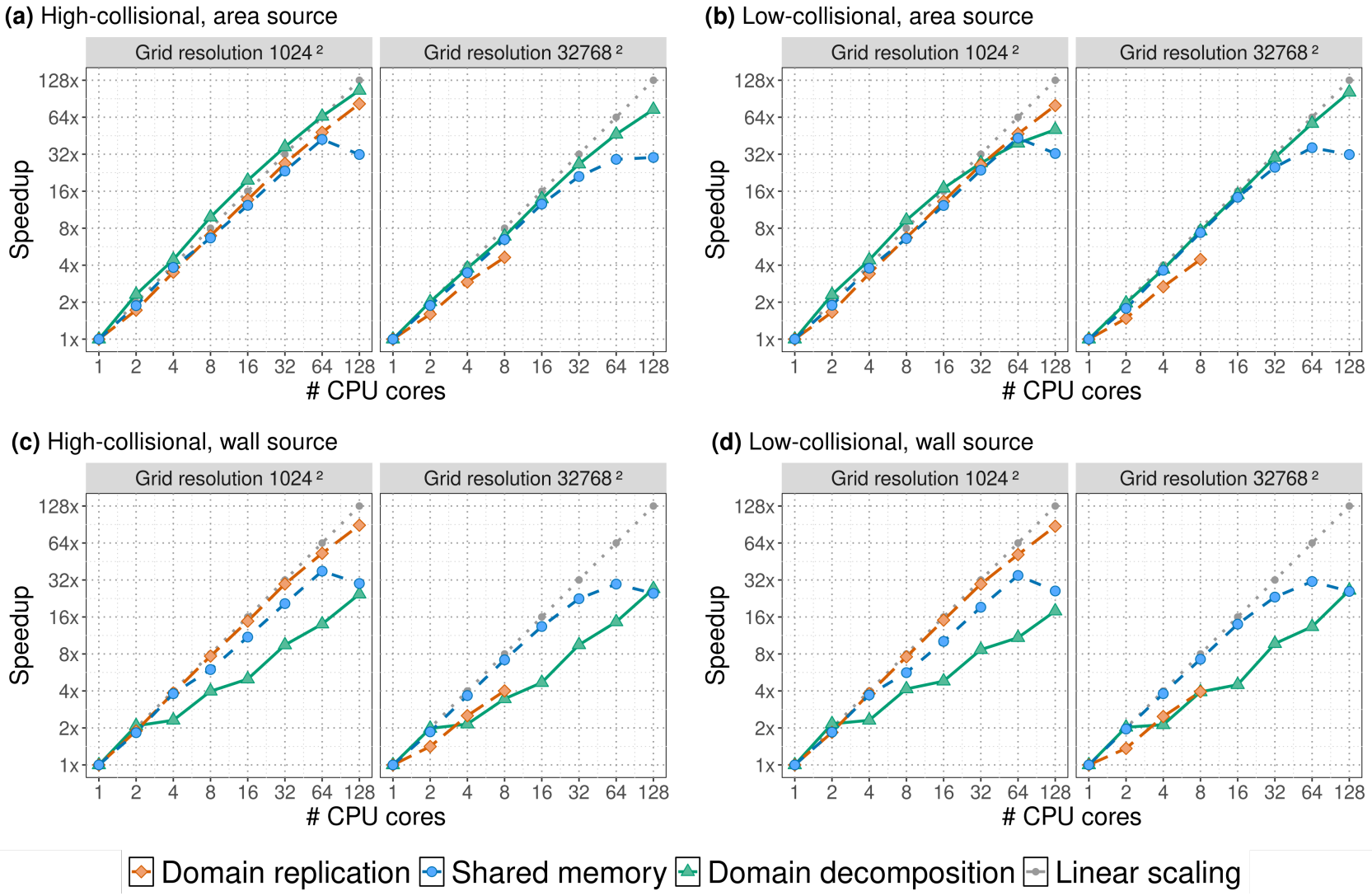
```
const eiron::grid::CartGrid2D & simulation_grid =
  sim_ctx.simulation_subdomain->simulation_grid.at(
    state.particle_event.species_idx
  );
```

- Because of the grid's deep copy semantics, the data buffer backing it was being copied
- This makes performance worse, and scalability better

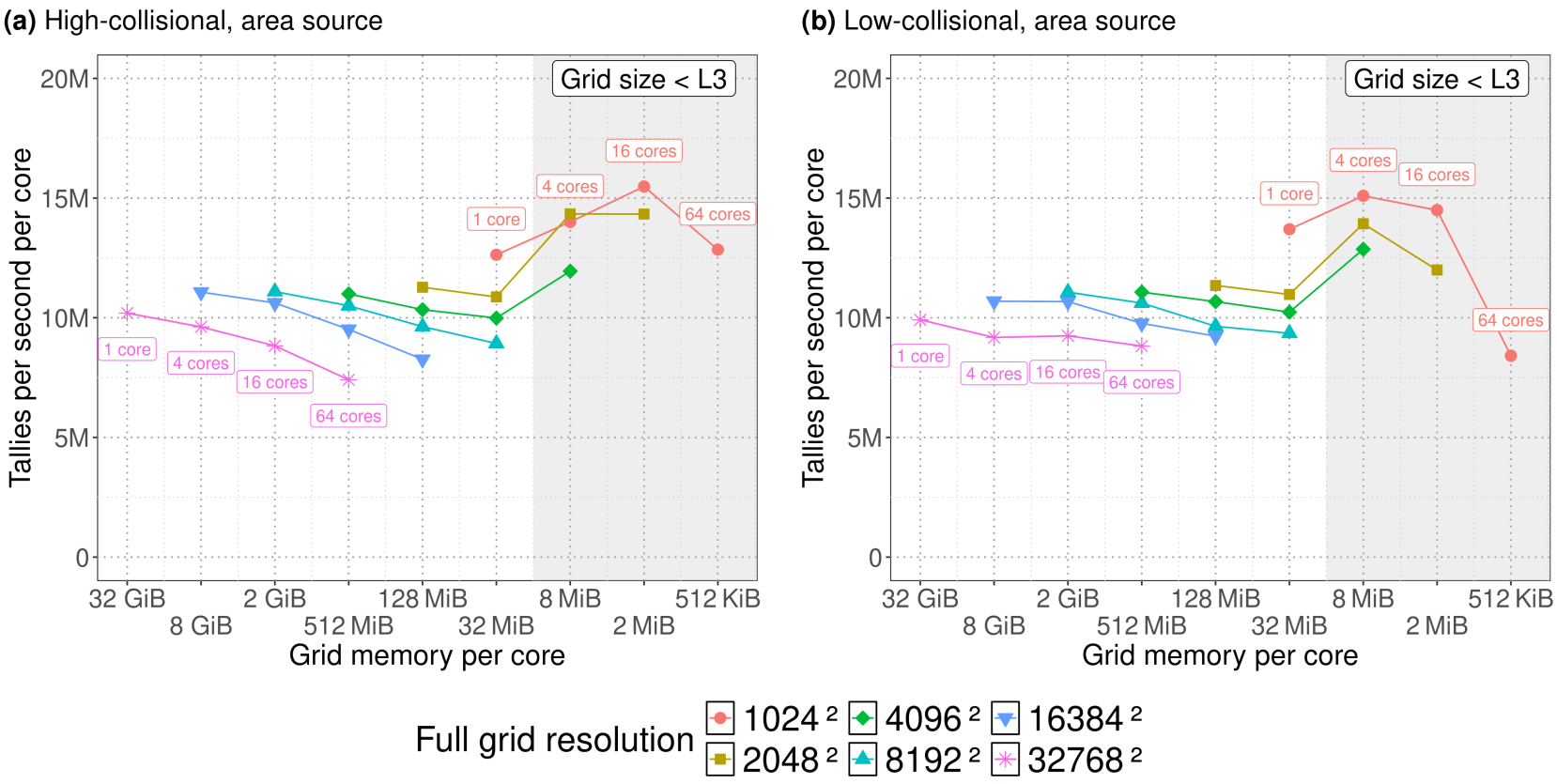
Correction to last years presentation

- The bug has been fixed
- Grid copy semantics changed to avoid similar bugs in future
- Results are now much more reasonable

After bug fix: strong scaling

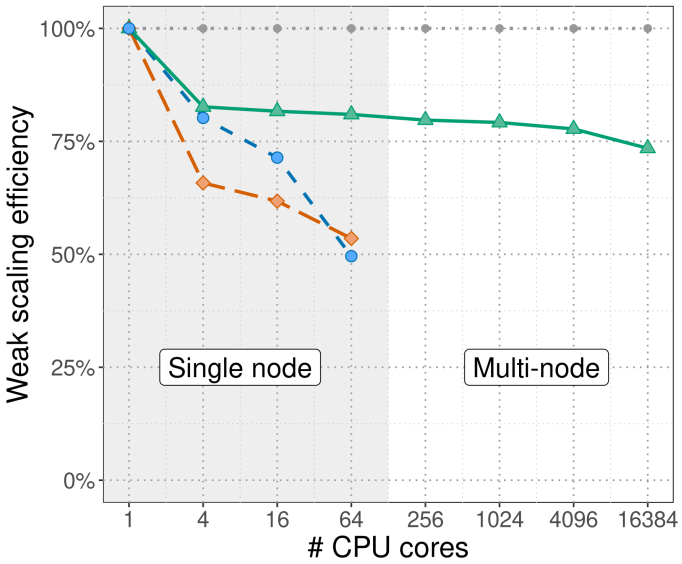


After bug fix: grid size effect

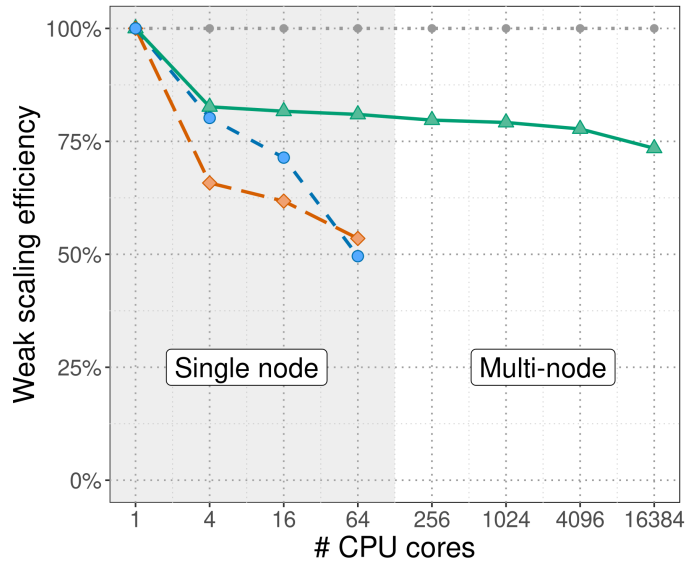


After bug fix: weak scaling

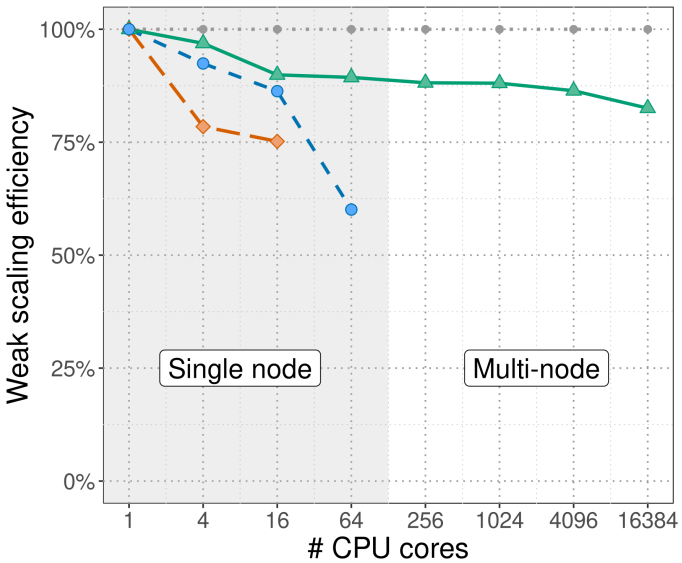
(a) High-collisional, area source, 1024^2 per core



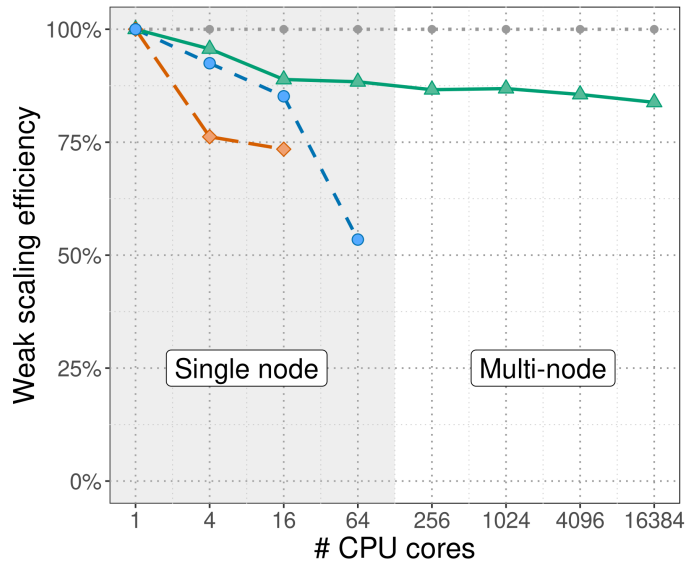
(b) Low-collisional, area source, 1024^2 per core



(c) High-collisional, area source, 4096^2 per core



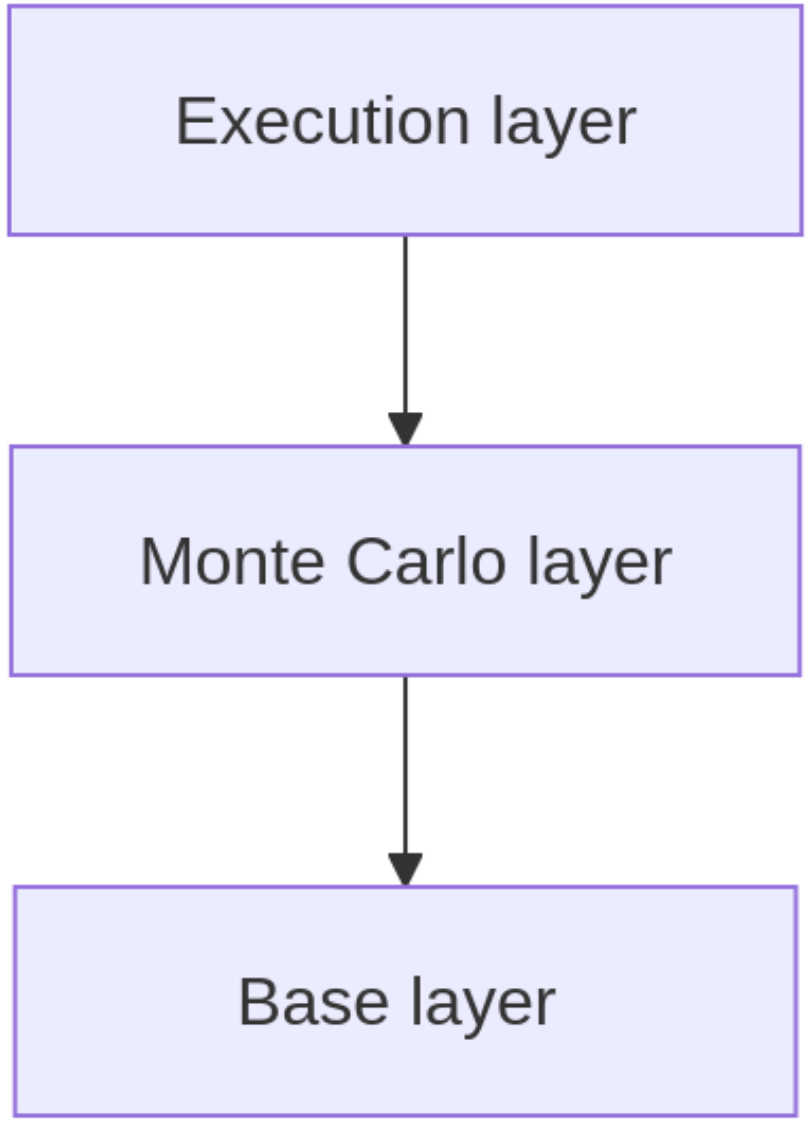
(d) Low-collisional, area source, 4096^2 per core



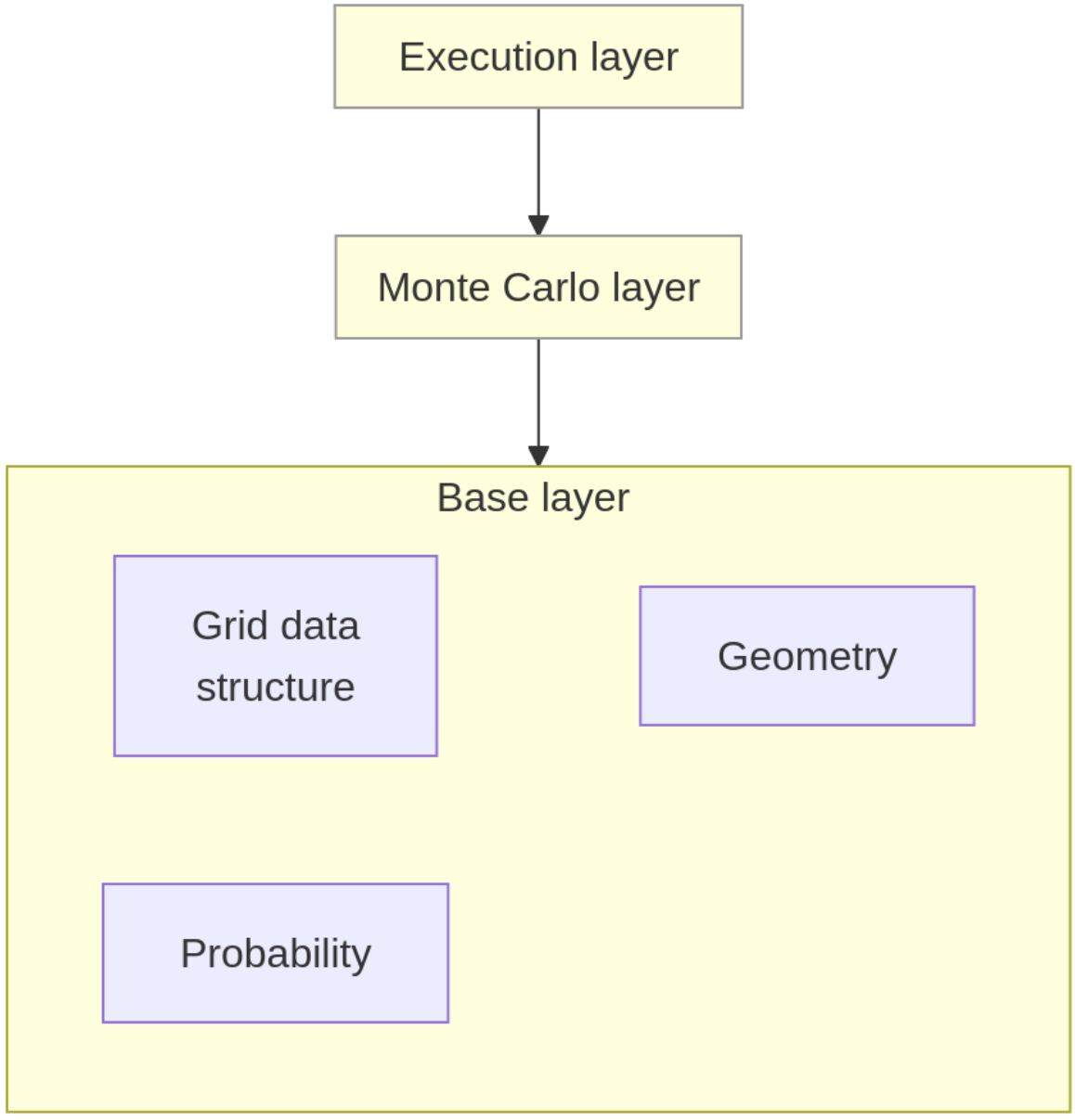
Domain replication | Shared memory | Domain decomposition | Linear scaling

GPU port

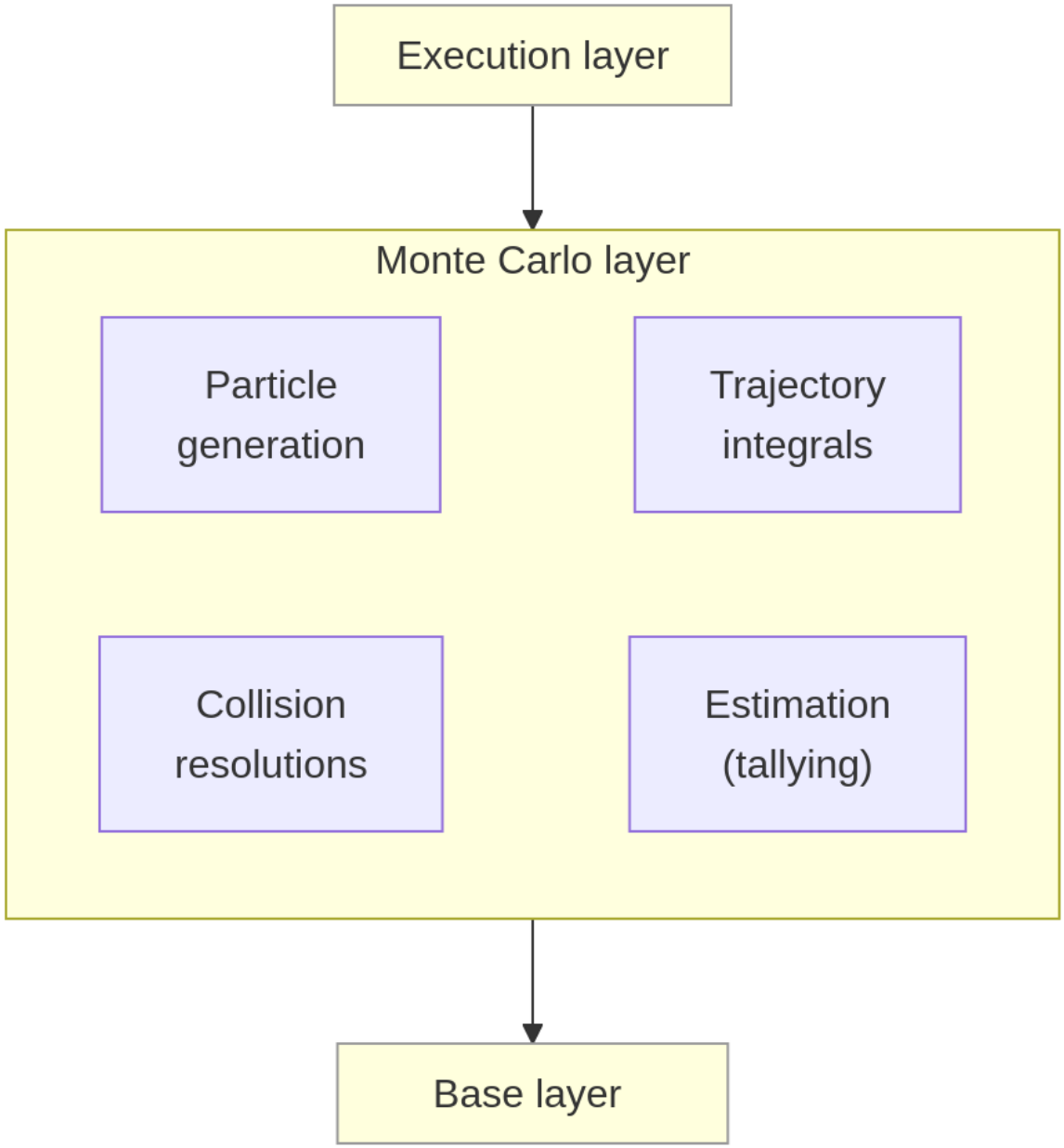
Eiron API layers



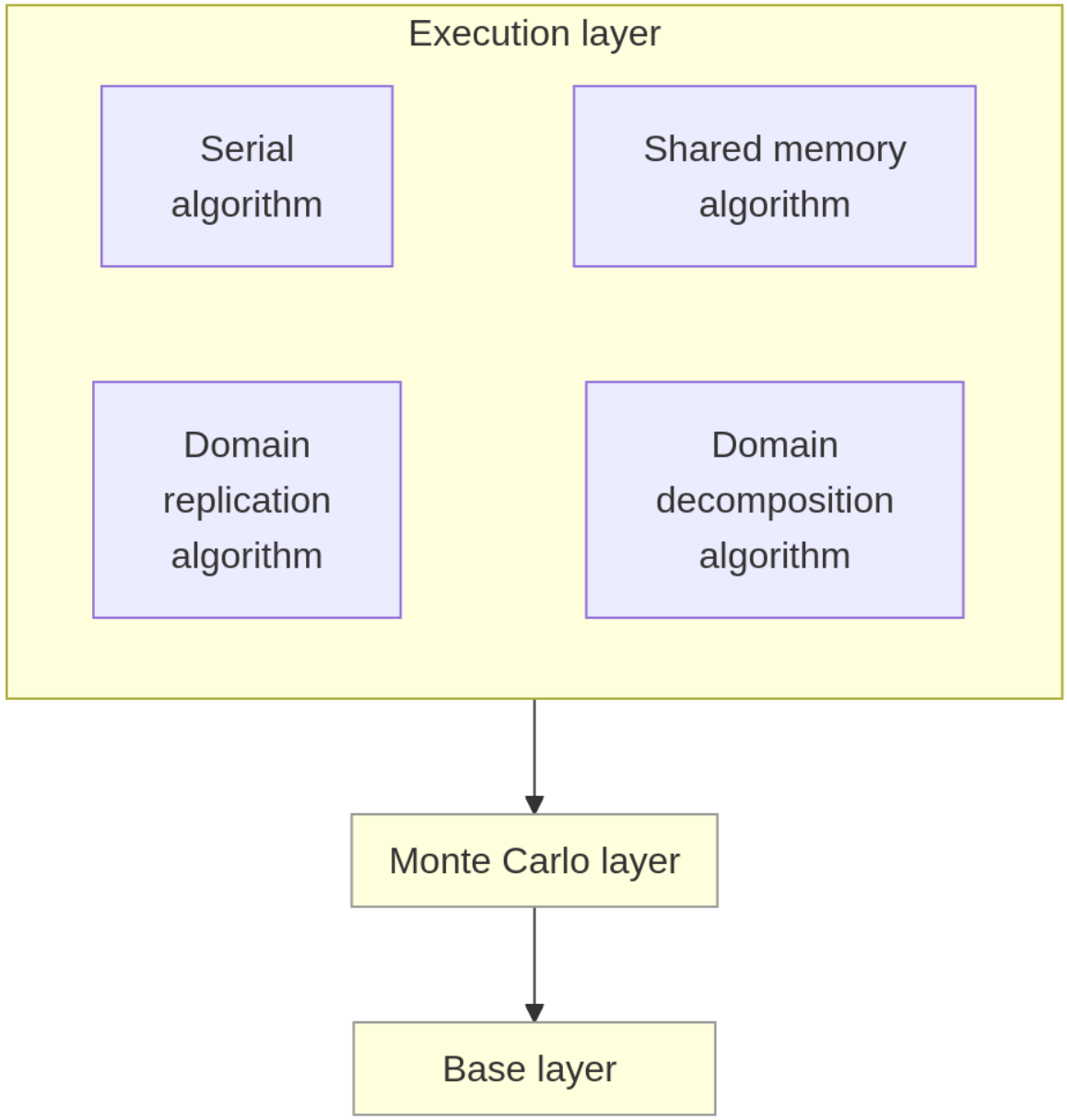
Eiron API layers



Eiron API layers



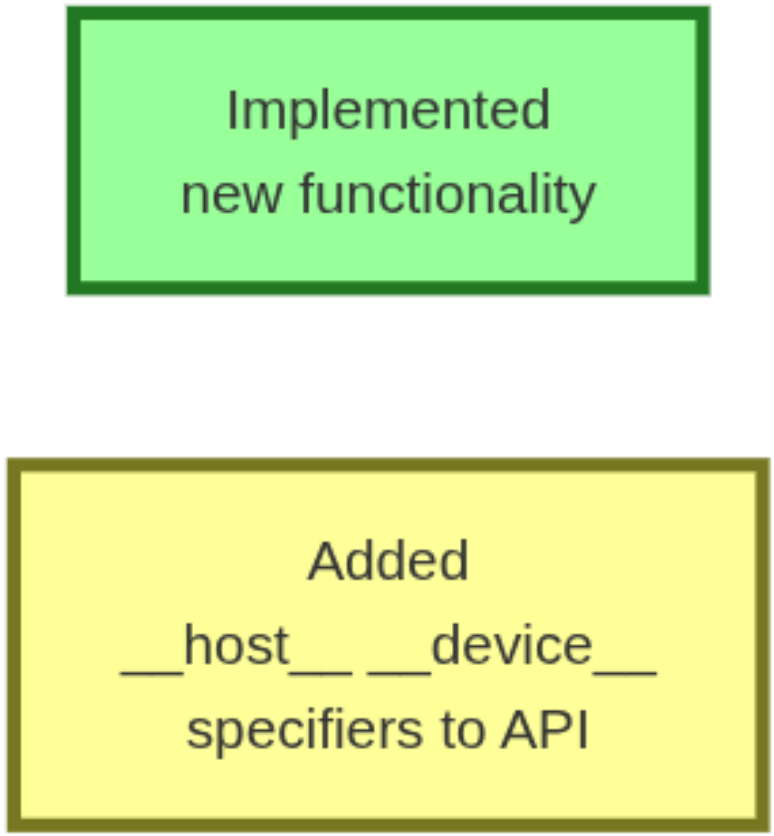
Eiron API layers



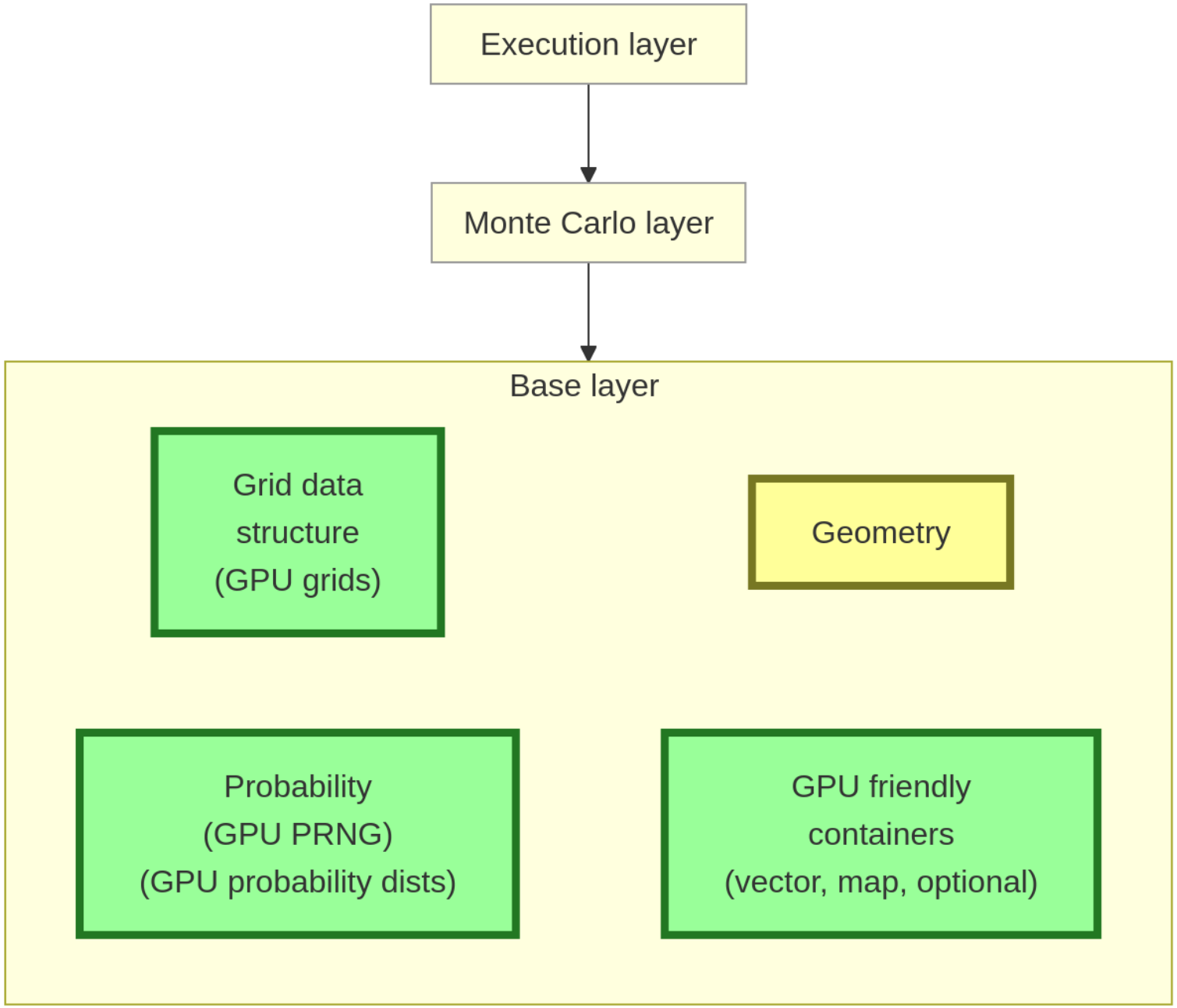
GPU port

- Approach: re-usable Monte Carlo API for both CPU and GPU
  - Refactor the Monte Carlo API for CUDA/HIP support
  - Compile MC API as *__host__ __device__*
- MC API always the same in both CPU & GPU impl.
- Execution API uses MC API to implement parallel algos:
  - Shared-memory, Domain replication, Domain decomposition
  - CPU, GPU
  - History-based MC, event-based MC

GPU port, legend for changes



GPU port


Execution layer
Monte Carlo layer
Base layer
Grid data structure (GPU grids)
Geometry
Probability (GPU PRNG) (GPU probability dists)
GPU friendly containers (vector, map, optional)

GPU port

No GPU support	Custom alternative	
`std::vector`	`eiron::inplace_vector`	Fixed capacity vector
`std::map`	`eiron::const_size_map`	Fixed capacity map
`std::optional`	`eiron::optional`	

GPU port: eiron::inplace_vector

```
template <typename T, size_t N>
struct inplace_vector {
    size_t m_size = 0;
    T      m_data[N];

    using element_t         = T;
    using iterator          = element_t *;
    using const_iterator    = const element_t *;
    using reference         = element_t &;
    using const_reference   = const element_t &;
    using rvalue_reference = element_t &&;

    // std::vector interface: push_back, emplace_back, pop_back, erase, begin/
        end, front/back, operator[], etc.
    __HOSTDEVICE__
    void
    push_back(const_reference value)
    {
        assert(m_size < N);
        m_data[m_size++] = value;
    }

    template <typename... Args>
    __HOSTDEVICE__ reference
    emplace_back(Args &&...args)
    {
        assert(m_size < N);
        return *new (m_data + m_size++) element_t(std::forward<Args>(args)...);
    }
    //...
}
```

GPU port: eiron::const_size_map

```
template <typename KeyType, typename ValueType, std::size_t N>
struct const_size_map {
    using key_t    = KeyType;
    using value_t  = ValueType;
    using iterator       = const_size_map_iterator<key_t, value_t, N>;
    using const_iterator = const_size_map_iterator<key_t, value_t, N, true>;

    std::size_t spec;
    value_t     array[N];

    // std::map interface: contains, insert, try_emplace, erase, begin/end,
         front/back, at(), operator[], etc.
    __HOSTDEVICE__
    bool
    contains(key_t key) const
    {
        auto        idx = static_cast<std::size_t>(key);
        std::size_t bit = 1 << idx;
        return !((bit & spec) == 0u);
    }

    __HOSTDEVICE__
    void
    insert(key_t key, const value_t &value)
    {
        auto idx = static_cast<std::size_t>(key);
        assert(idx < N);
        spec |= 1 << idx;
        array[idx] = value;
    }
    //...
}
```

GPU port: eiron::const_size_map_iterator

```
template <typename KeyType, typename ValueType, std::size_t N, bool Const>
struct const_size_map_iterator {
    using map_t                = const_size_map<KeyType, ValueType, N>;
    using iterator_category    = std::random_access_iterator_tag;
    using key_t                = KeyType;
    using value_t              = ValueType;
    using difference_type      = std::ptrdiff_t;
    using iterator_ref_t       = std::pair<key_t, value_t &>;
    using const_iterator_ref_t = std::pair<key_t, const value_t &>;

    map_t                       &map;
    mutable std::size_t          pos;
    mutable iterator_ref_t       ref;
    mutable const_iterator_ref_t const_ref;

    // random_access_iterator API: operator++, operator==, operator*, operator->
    __HOSTDEVICE__
    iterator_ref_t *
    operator->()
    {
        ref = iterator_ref_t(static_cast<key_t>(pos), map.at_pos(pos));
        return &ref;
    }

    __HOSTDEVICE__
    const iterator_ref_t *
    operator->() const
    {
        const_ref = const_iterator_ref_t(static_cast<key_t>(pos), map.at_pos(pos
            ));
        return &const_ref;
    }
    //...
};
```

GPU port

The PRNG we use (pcg64) and various probability distributions are not supported on GPUs

- Implemented an internal pcg64 PRNG with GPU support
- Implemented various probability distributions for GPU

GPU port: pcg64

```
#define MAKE_UINT128(hi, lo) (((__uint128_t)(hi)) << 64 | (lo))

struct prng {
    static constexpr __uint128_t PCG_MULTIPLIER = MAKE_UINT128
         (2549297995355413924ULL, 4865540595714422341ULL);
    static constexpr __uint128_t PCG_INCREMENT  = MAKE_UINT128
         (6364136223846793005ULL, 1442695040888963407ULL);

    __uint128_t seed;
    __uint128_t state;

    __HOSTDEVICE__           prng(__uint128_t seed_);
    __HOSTDEVICE__ void      advance(__uint128_t delta);
    __HOSTDEVICE__ uint64_t output() const;
    __HOSTDEVICE__ __uint128_t operator()();
};
```

GPU port: pcg64

```
__HOSTDEVICE__ uint64_t
prng::operator()()
{
    state = state * PCG_MULTIPLIER + PCG_INCREMENT;
    return output();
}

__HOSTDEVICE__ uint64_t
prng::output() const
{
    // PCG64 xsl_rr permutation family, 128-bit state, 64-bit output
    uint64_t value = state ^ (state >> 64);
    uint8_t  rot   = uint8_t(state >> (122)) & 63;
    return (value >> rot) | (value << ((-rot) & 63));
}

__HOSTDEVICE__ void
prng::advance(__uint128_t delta)
{
    __uint128_t acc_mult = 1;
    __uint128_t acc_inc  = 0;
    auto        mult     = PCG_MULTIPLIER;
    auto        inc      = PCG_INCREMENT;
    while (delta > 0) {
        if (delta & 1) {
            acc_mult *= mult;
            acc_inc = acc_inc * mult + inc;
        }
        inc = (mult + 1) * inc;
        mult *= mult;
        delta >>= 1;
    }
    state = acc_mult * state + acc_inc;
}
```

GPU port

Grid data structure changes:

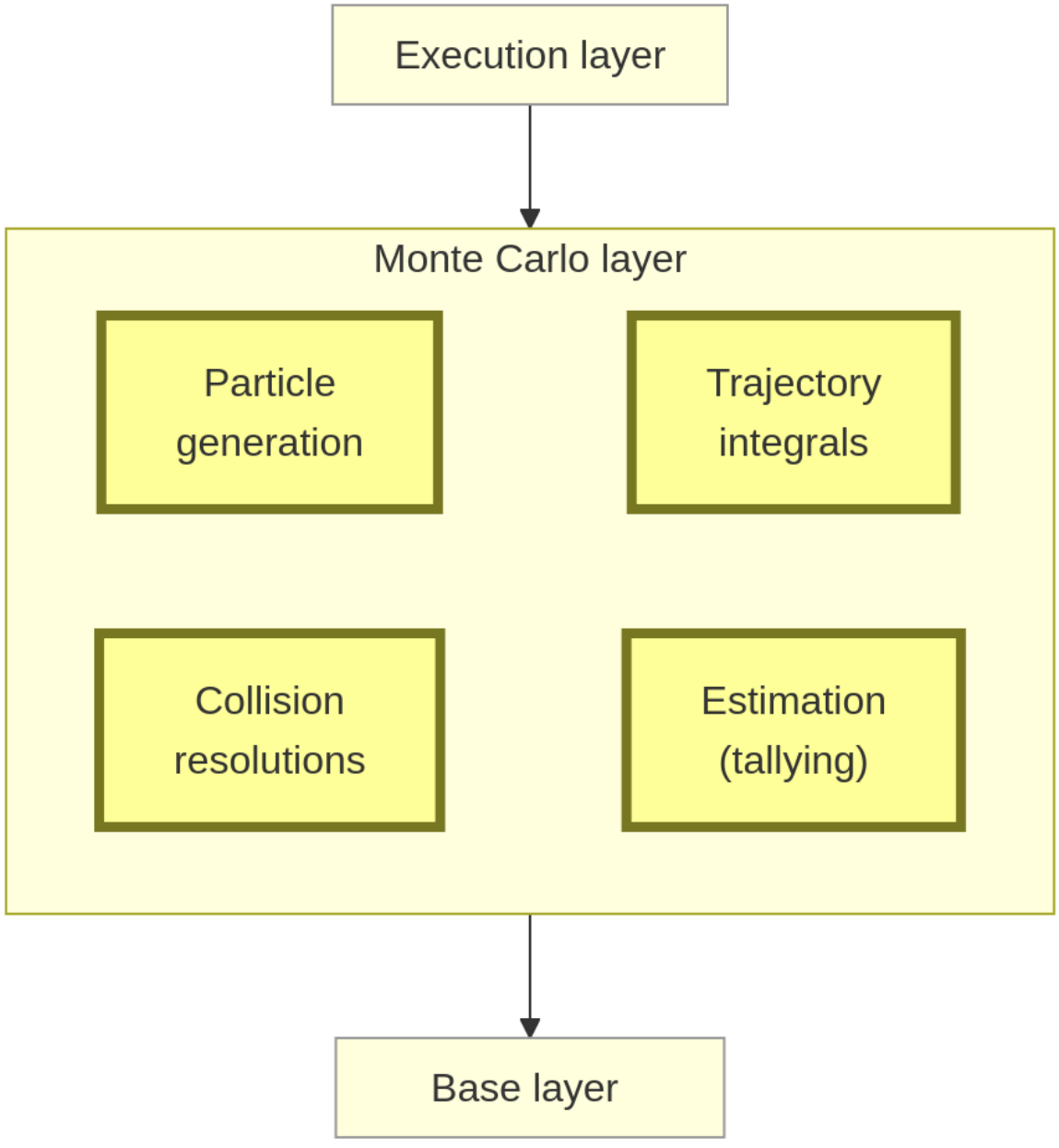
- Grids take a *device_type* parameter, grid data is allocated on CPU or GPU depending on the value
- If we want to pass-by-value to CUDA kernels, copy semantics need to be shallow copies (reference to data same in copy)

$\Longrightarrow$ Ended up with a reference counting solution

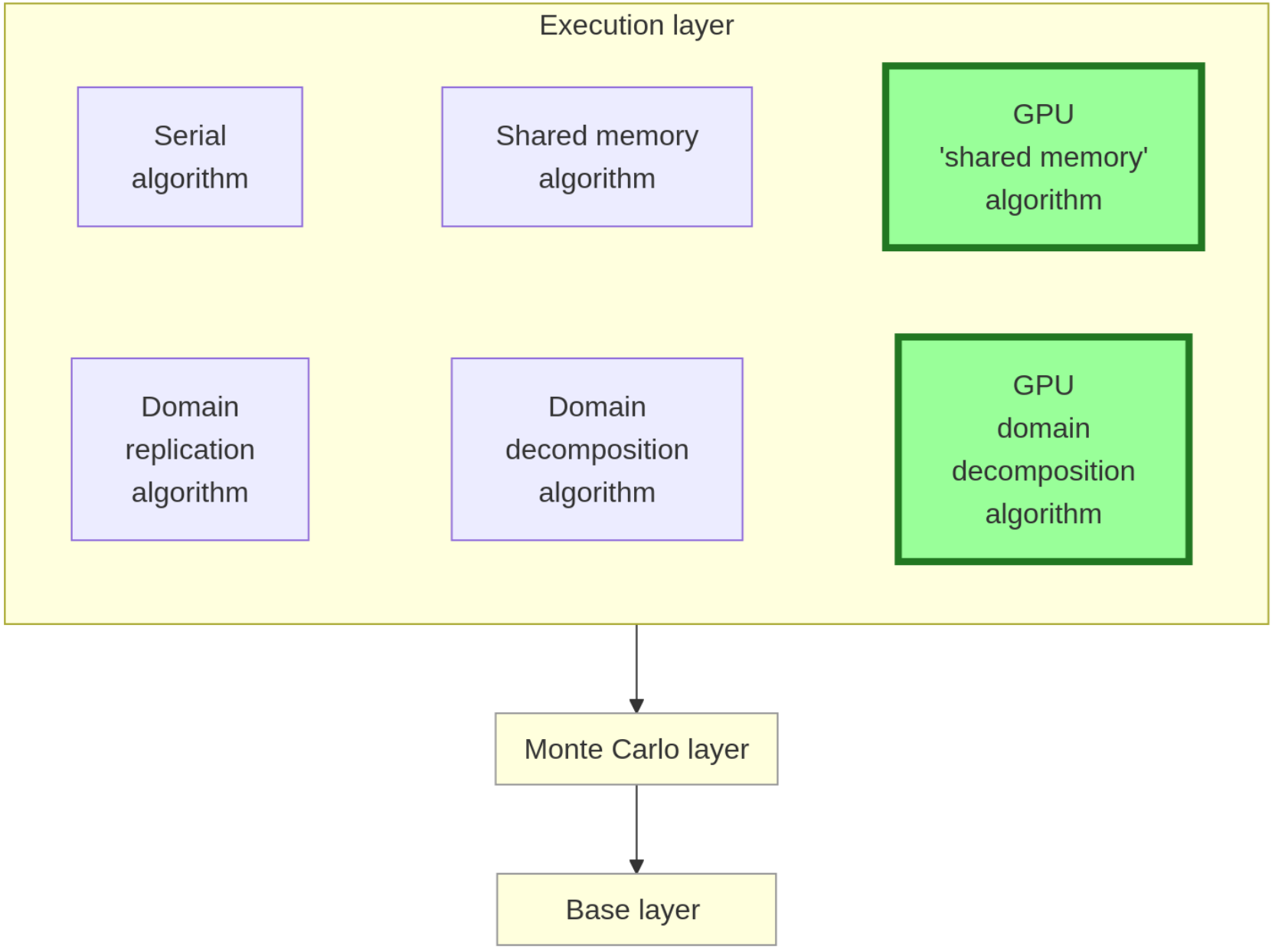
- Added member functions to create (deep) CPU copies or GPU copies of grids

Other internal data structures were also changed, but there is little point in recounting the changes here.

GPU port


Execution layer
Monte Carlo layer
Particle generation
Trajectory integrals
Collision resolutions
Estimation (tallying)
Base layer

GPU port


Execution layer
Serial algorithm
Shared memory algorithm
GPU 'shared memory' algorithm
Domain replication algorithm
Domain decomposition algorithm
GPU domain decomposition algorithm
Monte Carlo layer
Base layer

GPU MC algorithms

GPU MC algorithms

- Shared memory algorithm ported to GPU
- DDMC algorithm ported to GPU
- Correctness tested on NVIDIA & AMD, same results as on CPU
- ... but performance is bad, slower than parallel CPU algorithms

GPU MC algorithms, CPU algo issues

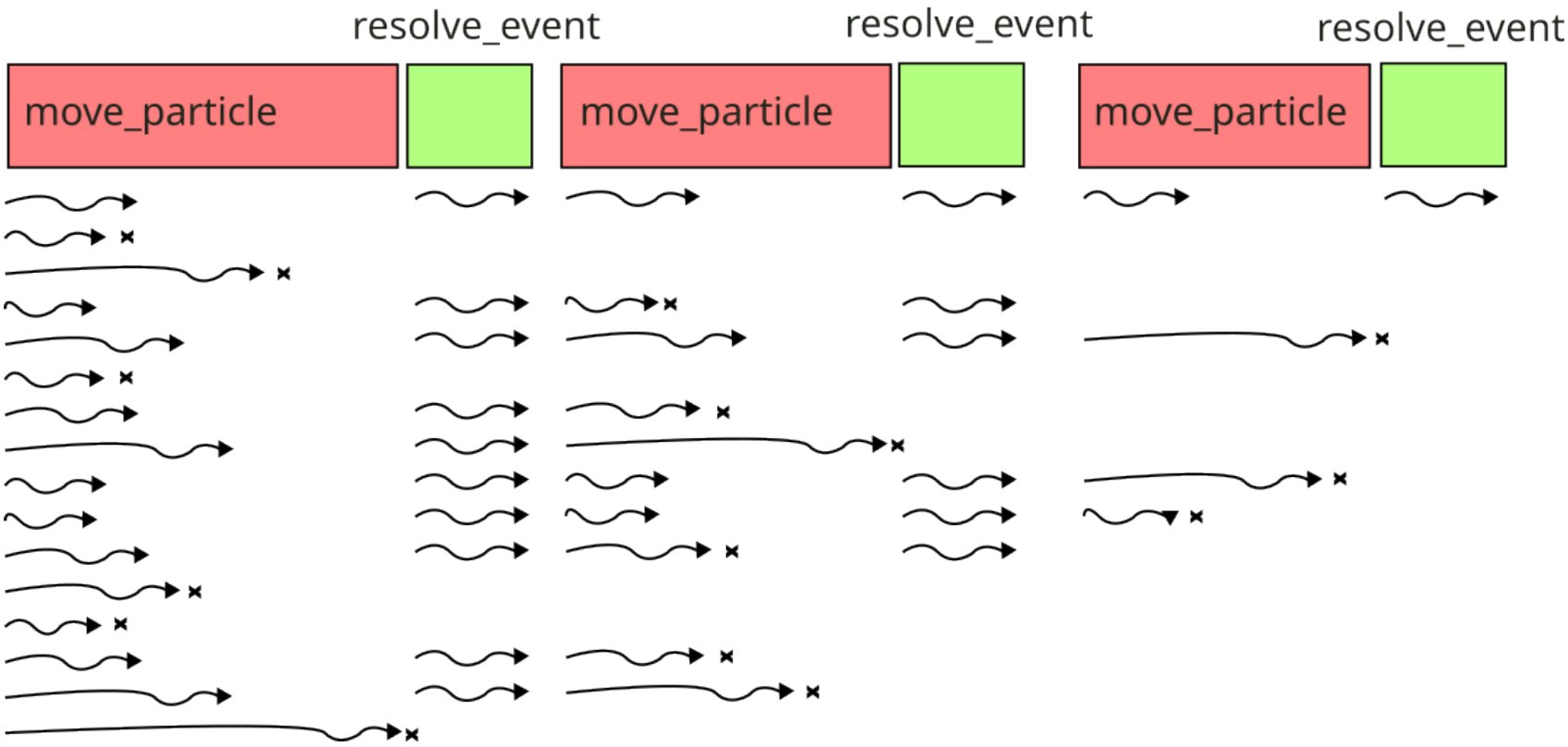
- The algorithms that have been ported are history-based MC
- This has thread divergence issues on GPUs

$\Longrightarrow$ Performance is not great

$\Longrightarrow$ We are working on event-based MC and other optimizations

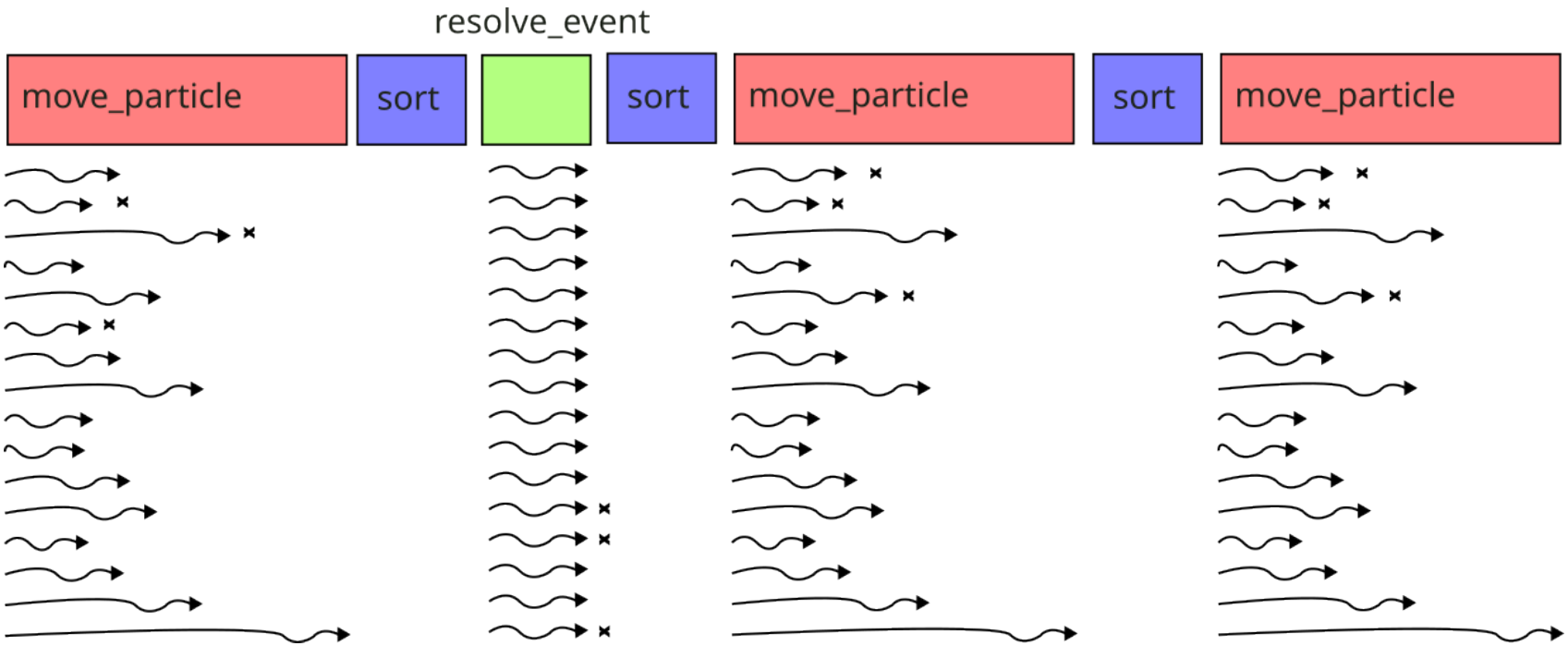
Event-based Monte Carlo

The issue with history-based MC

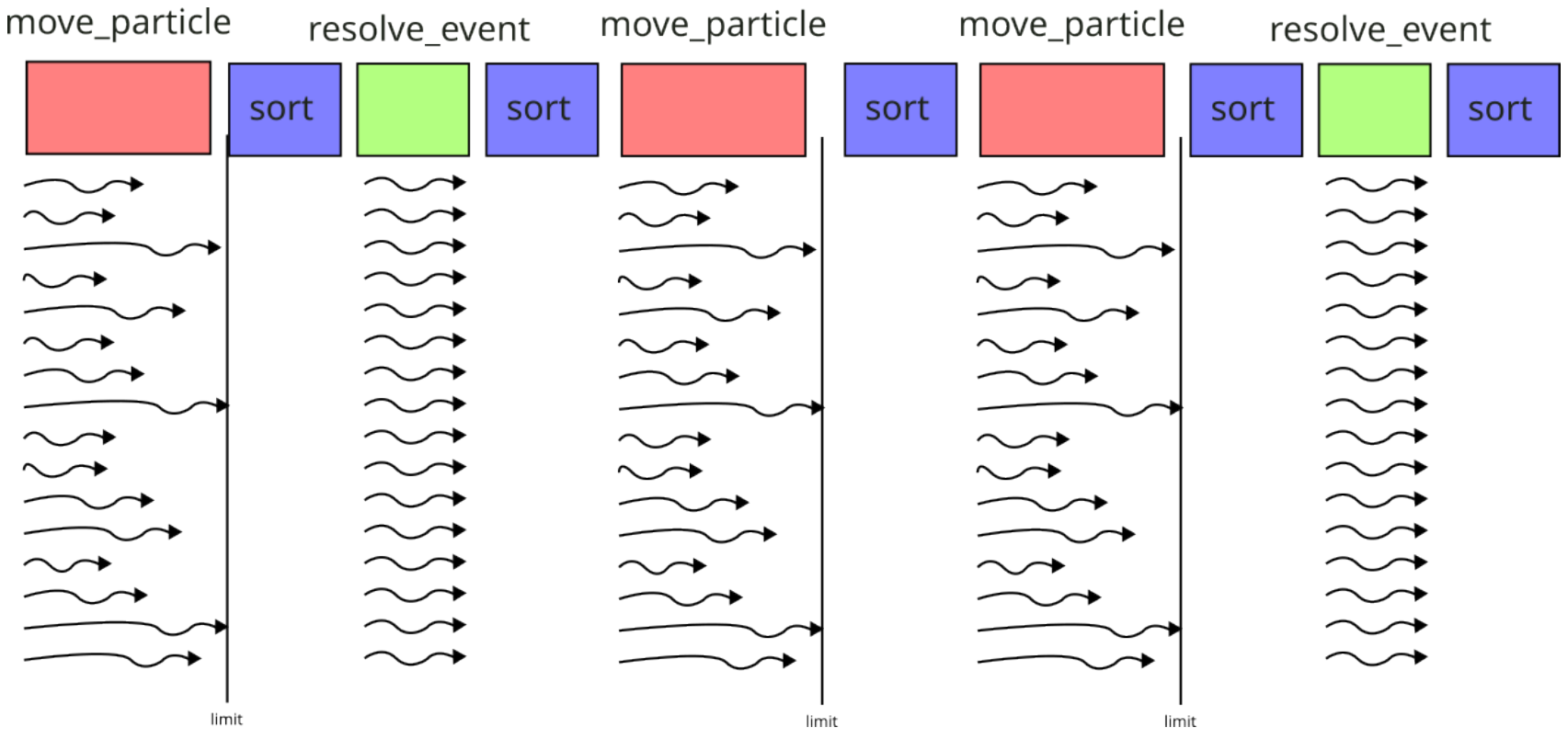


Thread divergence $\Longrightarrow$ low resource use (thread occupancy).

Event-based better for thread divergence



Adding a limit may be even better



Conclusion

GPU porting

- Dynamic runtime allocations are not GPU-friendly
- The C++ STL is not well supported on CUDA/HIP yet (although some support exists)
- Easiest to initialize on CPU, copy to GPU, copy back to CPU at the very end
- History-based MC bad for GPUs, need event-based

Open questions

- How do we sort particles efficiently on GPUs?
- GPUs and event-based algo add more tuning params. How to deal with 5 or so tuning parameters?

Any comments are appreciated!

Thank you!

https://version.helsinki.fi/lapposka/eiron.git