C++ Standard Parallelism for GPU Programming in a Particle-In-Cell Application

Ester El Khoury[1,3], Mathieu Lobet[1], Julien Bigot[1], and Laurent Colombet[2,3]

C++ Standard Parallelism, also known as **"StdPar"**, offers a vendor-independent parallel programming model, originally introduced for CPUs, and now extended to GPUs.

Does the C++17 Standard Parallelism model provide an effective balance between Performance, Portability, and Productivity when used on a single GPU for large-scale plasma physics simulations?

C++ Standard Parallelism

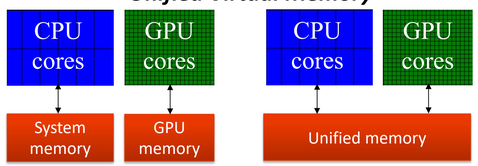
C++17 introduced high-level parallelism with four execution policies that specify how code may be executed :

- `seq` sequential execution (default)
- `par` has potential for parallel execution, `par_unseq` for parallel and vectorized execution and `unseq` for vectorized execution (C++20)

These policies are used with standard algorithms such as : `for_each, transform, reduce, transform_reduce, sort, count, fill …`

C++ assumes a unified memory model, but CPUs and GPUs have distinct physical memories. To facilitate GPU support, NVIDIA introduced Unified Virtual Memory (UVM) [1], a heterogeneous memory management system that was later also adopted by AMD.

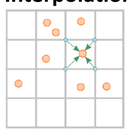
Unified Virtual Memory



MiniPIC

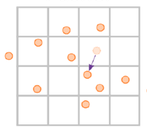
- **Particle-in-Cell (PIC) mini-application [2]**
- **Goal :** Explore different programming models (SYCL, Thrust, Kokkos, OpenMP Target and OpenACC)
- **Structure :** PIC loop with separate kernels for each physics operator

Interpolation



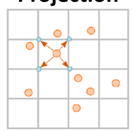
- Random memory access on the grid
- Cache miss

Pusher



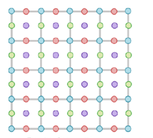
- Indices are independent
- Contiguous memory access

Projection



- Random access
- Memory contention

Maxwell Solver



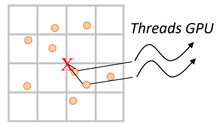
- Stencil problem

(1) Université Paris-Saclay, UVSQ, CNRS, CEA, Maison de la Simulation, Gif-sur-Yvette, France
(2) CEA, DAM, DIF, Arpajon, France
(3) Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, Bruyères-le-Châtel, France

Parallel Particle Projection

- While all kernels introduce programming challenges, the primary difficulty stems from ensuring atomic reductions.
- Each particle is handled by a GPU thread.
- Concurrent accesses to shared grid cells may occur.

➢ Atomic operations are needed to protect memory.



… but they can negatively impact performance

```
double *Jx = patch.vector.get_raw_pointer(device);

std::for_each(std::execution::par_unseq, counting_iterator(0),
counting_iterator(n_particles), [=](int ip) {
    atomicAdd(&Jx[…],∝ * Jxp1); atomicAdd(&Jx[…], β * Jxp2);
});
```

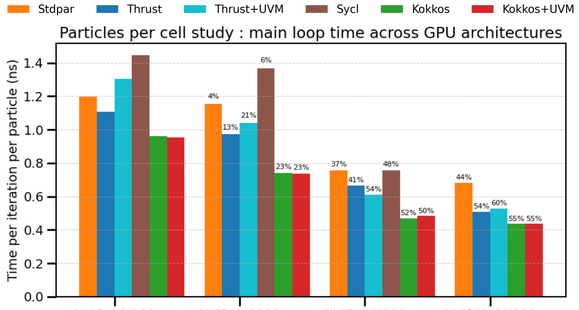
```
kokkos::View<double***, Kokkos::Atomic> Jx =Jxs;

Kokkos::parallel_for(n_particles,KOKKOS_LAMBDA(const int ip) {
    Jx(…,…,…) +=  ∝ * Jxp1; Jx(…,…,…) +=  β * Jxp2;
});
Kokkos::fence();
```

In C++, just add an execution policy, no need to rewrite the algorithm!
With Kokkos, you need to adapt both data structures and parallel loops.

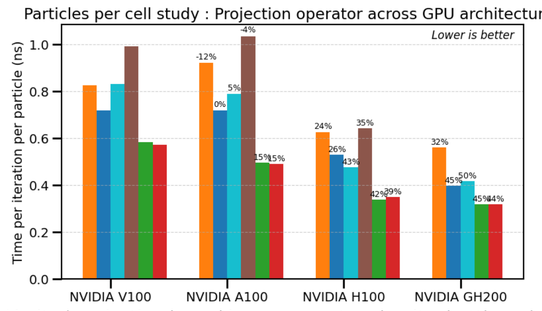
Results

- Interpolation, Pusher, and Maxwell Solver account for only 10-25% of the total runtime, justifying a focus on the Projection kernel, which dominates across all models.
- The comparison is performed with 134 million particles.
- Percentages indicate speedup relative to NVIDIA V100 and AMD MI250 (e.g., +33% means 33% faster than V100 or MI250 baseline; –20% means 20% slower).

NVIDIA GPUs*

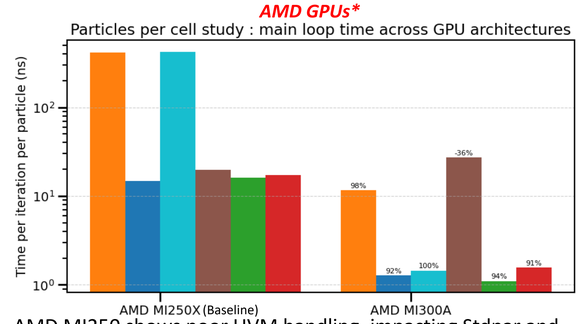


- On newer GPUs, execution time decreases, but Stdpar shows different gains due to backend differences.
- From V100 to GH200, Stdpar execution time improves by 44%, while Thrust with UVM achieves a 60% speedup.
- The limited acceleration in Stdpar is primarily due to the projection kernel, where atomic reductions use global-scope memory, an inefficient pattern on GPUs.
- Therefore, we aim to focus our results on analyzing the behavior of this kernel.

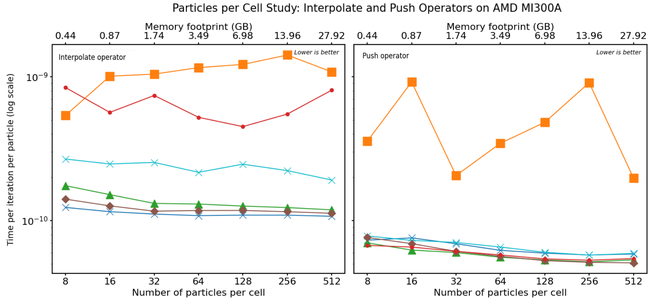


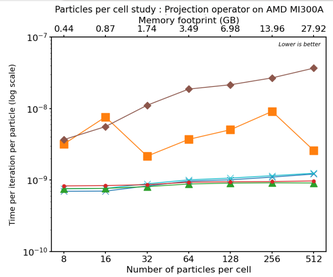
Limited projection kernel improvement makes it a bottleneck, limiting hardware upgrade benefits.

AMD GPUs*



- AMD MI250 shows poor UVM handling, impacting Stdpar and Thrust UVM performance.
- MI300 shows noticeable improvement; however, this gain is mostly due to MI250's weak UVM performance.





- On AMD GPUs, Stdpar shows consistently higher execution times across all kernels, not just projection.
- Main loop slowdown : 10 ns/particle on MI300A vs 0.8 ns on NVIDIA V100.
- MI300A improves overall performance for Thrust UVM and Stdpar, but some issues persist.
- UVM remains unstable: Kokkos shows instability in the first kernel, and Stdpar has issues across all four kernels.

Conclusion

Depends on your goal and use case, but this approach offers:

✓ Strengths

- Accessible to non-experts
- No major rewrite of existing C++ code
- Portable across GPUs (NVIDIA & AMD)
- High performance for simple loops

✗ Limitations

- Limited optimization for experts
- CPU model not suited for GPU architecture
- Execution policy implementation depends on compiler developers

⇒ *Potential for further improvement*

Future work

- Understand performance gaps between Stdpar and Thrust
- Analyze performance variability across different AMD GPUs
- Optimize the Projection kernel
- Rewrite the entire mini-application using CUDA
- Improve the performance of existing CUDA kernels
- Explore asynchronous programming for further acceleration
- Propose enhancements for existing C++ libraries for GPU model

References

MiniPIC Repo

[1] Allen, T., & Ge, R. (2021). In-depth analyses of unified virtual memory system for GPU accelerated computing. SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, 1–14. https://doi.org/10.1145/3458817.3480855

[2] Silva-Cuevas, J., Zych, M., et al. : Towards a complete task-based implementation of a 3d particle-in-cell code: Performance studies and benchmarks. Computer Physics Communications 313, 109647 (2025). https://doi.org/https://doi.org/10.1016/j.cpc.2025.109647

* Experiments were conducted using :
- CUDA 12.2.1 with nvcc++ 23.7 for NVIDIA V100, A100, and H100 GPUs
- CUDA 12.3 with nvcc++ 24.1 for the GH200
- ROCm 6.1.2 with clang++ 17.0.0 for AMD GPUs