Connecting The Dots 2022
7th International CTD Workshop
Princeton University, Princeton, USA
May 31 - June 2, 2022

Vecpar - A portability parallelization library

Georgiana Mania[1,2] Nicholas Styles[1] Michael Kuhn[3] Andreas Salzburger[4] Beomki Yeo[5]
Thomas Ludwig[2]

[1]DESY [2]University of Hamburg [3]Otto von Guericke University [4]CERN [5]University of California, Berkeley

Outline

Computing challenges[1] of the HL-LHC

- 7-10x increase in data acquisition rate
- 10x more events (both real and simulated)
- higher degree of event complexity (mostly due to an increase in track multiplicity)
- higher physics accuracy is required
- increased efficiency needed in exploiting the available hardware resources

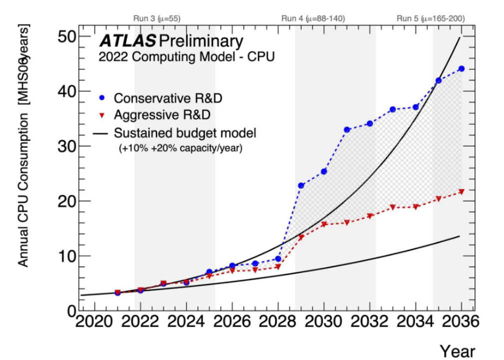


Figure 1: Computing model for ATLAS

[1] ATLAS Software and Computing HL-LHC Roadmap [Col22]

Hardware architectures

World largest supercomputers use accelerators to boost performance and reduce electricity consumption per computation, with **NVIDIA** being the most high-ranked GPU vendor [SDSM]

- Summit (US#1 in 2021) has 27,648 NVIDIA V100s
- JUWELS Booster (EU#1 in 2021) has 2,744 NVIDIA A100

Some of the next exascale supercomputers will employ other vendors like

- **AMD** for Frontier Supercomputer at ORNL (currently world's #1), LUMI (EU#1) and El Capitan Supercomputer at LLNL
- **Intel** for Aurora Supercomputer at ANL

Code portability is now more important than ever!

Software ecosystem

Many general solutions to target accelerators with different levels of complexity, portability and performance through

- language extensions and proprietary frameworks: NVIDIA CUDA, AMD HIP, Intel oneAPI
  (*limited portability & code rewrite needed using new language*)
- compiler directives (vendor-agnostic): OpenMP, OpenACC
  (*limited portability when using single source code*)
- portability libraries: Kokkos, Alpaka, NVIDIA stdpar
  (*code rewrite needed to fit the abstractions or limited portability*)

Choosing the right solution depends on the application and the accepted compromise.

Requirements

- Easy to use C++ API which requires limited code changes to be adopted
- Abstract away the notion of architecture by using (a) single-source code and (b) automatically generated code for platform optimizations
- Compilable by mainstream C++ compilers (e.g. clang)
- Portable (shared-memory CPU and NVIDIA GPU) but also easily extendable to new architectures like AMD/Intel GPUs and potentially Big Data platforms (Google Cloud/AWS)
- Guarantee improved wall-clock performance over the initial (sequential) implementation with little or no penalty over native parallelization solutions

We designed vecpar having all these in mind.

Outline

Regardless of the platform, data parallelism

- means executing the same task for several elements of a collection in parallel (e.g. OpenMP parallel-for loop)
- requires the data to be partitioned and distributed among the workers (e.g. threads of any kind: OpenMP, TBB, CUDA, etc)

Moreover, to offload computations to a GPU, some preconditions have to be met

- the function pointer to the actual parallel task has to be copied to the GPU
- the data used for the computations needs to be accessible from the device

Vecpar handles both data distribution and host↔device transfers out of the box.

Vecpar builds on the map-filter-reduce paradigm from functional programming

- (map f coll) - apply function **f** to each element of the **coll**ection to produce a different collection of the same size, with the same or different type of elements
- (filter p coll) - keep in the output collection only the elements from the input **coll**ection which satisfy the predicate **p**
- (reduce f coll init) - reduce the elements of the **coll**ection using function **f** and store it in the result initialized with **init**

```
;; transform a collection of measurements into space points
(map toSpacePoints `(meas1 meas2.. measN)) ;; (sp1 sp2.. spN)
;; keep only the space points (sp) with activation above a threshold
(filter isAboveThreshold? `(sp1 sp2 sp3)) ;; (sp1 sp2)
;; gather volumes during propagation; ps = propagation state
(reduce addVolume `(ps1 ps2.. psN) []) ;; `(volume1 volume2.. volumeM)
```

Listing 1: Code samples in Clojure programming language [Hic20] vecpar algorithms offer an abstraction layer over low-level specializations enabled by compile-time polymorphism for the two backends: CPU OpenMP and GPU CUDA

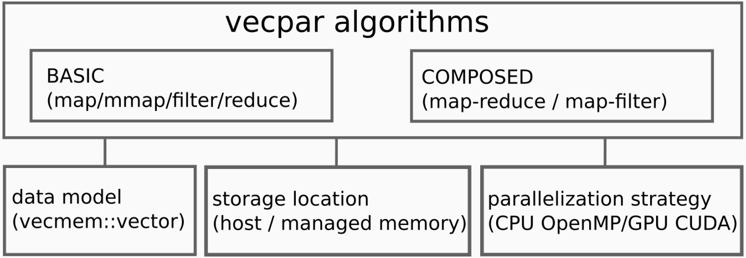


Figure 2: Vecpar architecture

Code compilation

C++ code which extends vecpar algorithms can be compiled (with clang) for different architectures. cmake examples:

```
find_package(vecpar REQUIRED)

target_link_libraries(cpu_exe
    vecpar::all
    vecmem::core vecmem::cuda
    OpenMP::OpenMP_CXX
```

Listing 2: cmake compile for CPU configuration

```
find_package(vecpar REQUIRED)

target_link_libraries(gpu_exe
    vecpar::all
    vecmem::core vecmem::cuda
    CUDA::cudart)

target_compile_options(gpu_exe PUBLIC
    $<$<COMPILE_LANGUAGE:CXX>:-x cuda
                         --offload-arch=sm_XY>)
```

Listing 3: cmake compile for GPU configuration

Outline

- Runge-Kutta-Nyström (RKN) stepper is used to estimate the position and momentum of charged particles by integrating the equation of motion
- It is a compute intense algorithm, yet ideal for parallelization because it can be invoked for several tracks in parallel since the data is independent
- A **simplified**[2] RKN implementation used for this talk is available in detray project [3]

[2] with a fixed number of integration steps

[3] `https://github.com/acts-project/detray`

Baseline CUDA code

```
__global__ void rk_stepper_test_kernel (vecmem::data::vector_view<free_track_parameters> tracks_data, const
    ↪ vector3 B) {
    int gid = threadIdx.x + blockIdx.x * blockDim.x;
    vecmem::device_vector<free_track_parameters> tracks(tracks_data);
    if (gid >= tracks.size()) { // Prevent overflow
        return;
    }
    auto& traj = tracks.at(gid); // Get a track
    // Define RK stepper
    rk_stepper_type rk(B);
    // Forward direction
    rk_stepper_type::state forward_state(traj);
    for (unsigned int i_s = 0; i_s < rk_steps; i_s++) {
        rk.step(forward_state);
    }
    // Backward direction
    traj.flip();
    rk_stepper_type::state backward_state(traj);
    for (unsigned int i_s = 0; i_s < rk_steps; i_s++) {
        rk.step(backward_state);
    }
}
```

Listing 4: CUDA implementation of the RKN stepper in detray project (Feb 2022)

vecpar implementation

```
struct rk_stepper_algorithm :
    public vecpar::algorithm::parallelizable_mmap<free_track_parameters, vector3>{

    TARGET free_track_parameters& map(free_track_parameters& traj, vector3 B) override {

        // Define RK stepper
        rk_stepper_type rk(B);
        // Forward direction
        rk_stepper_type::state forward_state(traj);
        for (unsigned int i_s = 0; i_s < rk_steps; i_s++) {
            rk.step(forward_state);
        }
        // Backward direction
        traj.flip();
        rk_stepper_type::state backward_state(traj);
        for (unsigned int i_s = 0; i_s < rk_steps; i_s++) {
            rk.step(backward_state);
        }
        return traj;
    }
};
```

No index-based calculation needed → portable code

Listing 5: C++ implementation of the RKN stepper using vecpar library

Outline

Test configuration 10,000 simulated tracks using $\theta \in [0, \pi]$ and $\phi \in [-\pi, \pi]$, with origin position (0,0,0), charge -1, RKN computes 100 integration steps for each track, using a uniform magnetic field of B=(0,0,2T)

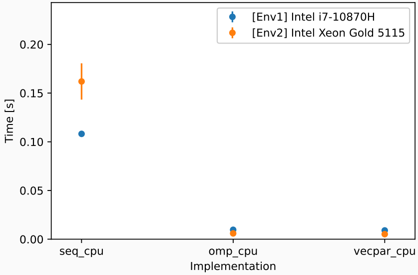
Code setup detray's backends cmath and eigen are being investigated

Config	Environment 1	Environment 2 [4]
Arch	1 socket x 8 cores x 2 threads	2 sockets x 10 cores x 2 threads
CPU	Intel(R) Core(TM) i7-10870H @ 2.20GHz	Intel(R) Xeon(R) Gold 5115 @ 2.40GHz
GPU	NVIDIA GeForce RTX 3060	NVIDIA Tesla V100
CUDA Driver	510.47.03	510.47.03
CUDA Version	11.6	11.6
C++ compiler	clang 14	clang 14

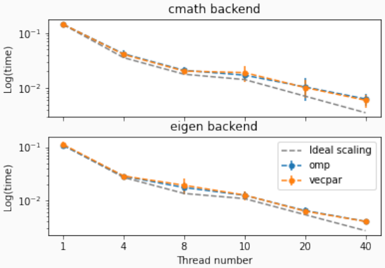
Table 1: Hardware environments used for performance evaluation

[4] ATLAS-GPU01 node at the National Analysis Facility (NAF) at DESY

CPU results



(a) Comparison between Env1 (16 OpenMP threads) and Env2 (40 OpenMP threads), double precision, cmath backend



(b) Strong-scaling evaluation: Multi-threading implementations in simple precision for cmath/detray backends, on Env2

Figure 3: Mean and standard deviation for Runge-Kutta-Nyström stepper

CPU results - Speedup

Vecpar implementation speedups over the sequential implementation for different math backends

- 28x - cmath
- 34x - eigen

Vecpar implementation adds no overhead in comparison to a hard-coded OpenMP pragma implementation.

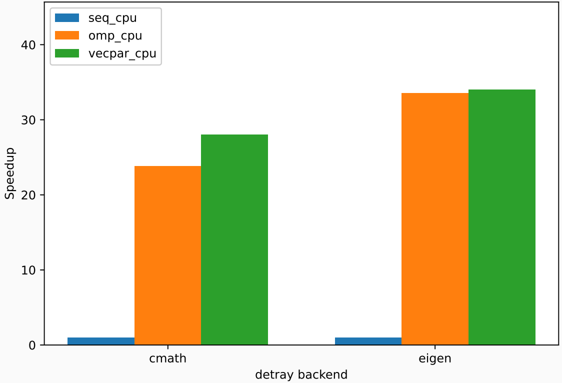
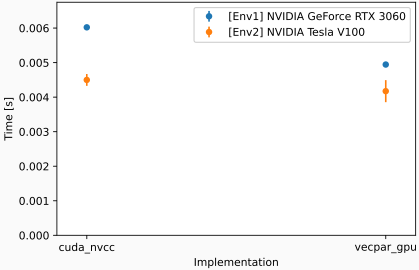
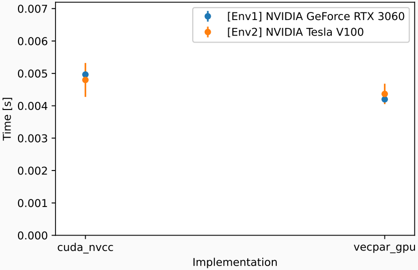


Figure 4: Speed-up factors over initial sequential version (seq_cpu) for the multi-threading hard-coded OpenMP (omp_cpu) and vecpar (vecpar_cpu)implementations using detray cmath/eigen backends, double precision on Env2

GPU results - Different environments



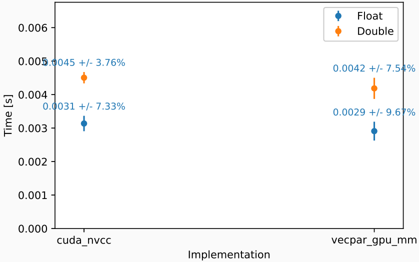
(a) Fastmath disabled



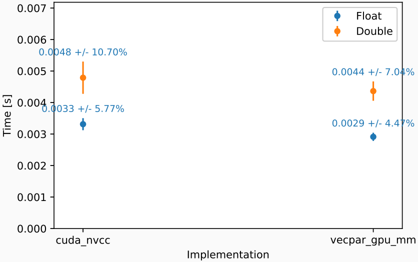
(b) Fastmath enabled

Figure 5: Mean and standard deviation for Runge-Kutta-Nyström stepper with grid configuration: 157x64 CUDA threads, clang as host compiler for nvcc

GPU results - Different precision



(a) Fastmath disabled



(b) Fastmath enabled

Figure 6: Mean and standard deviation for Runge-Kutta-Nyström stepper, cmath backend, simple /double precision, Env2

Similar results were obtained with the eigen backend.

GPU results - Speedup

Vecpar implementation speedups over the sequential implementation for different math backends

- 38x - cmath
- 27x - eigen

Vecpar implementation compiled with clang is faster than the CUDA implementation compiled with nvcc.

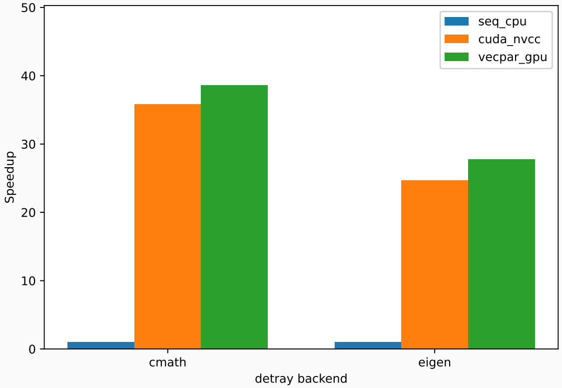
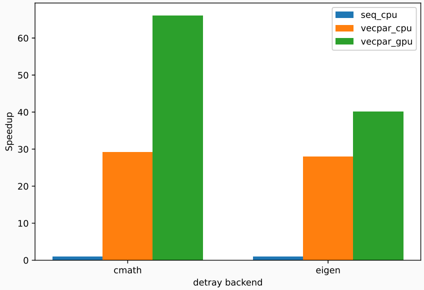
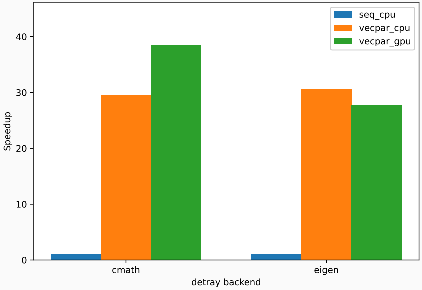


Figure 7: Speed-up factors over initial sequential version (seq_cpu) for GPU CUDA (cuda_nvcc) and vecpar (vecpar_gpu), detray cmath/eigen backends in double precision, with fastmath disabled, on Env2

Aggregated results



(a) Simple precision



(b) Double precision

Figure 8: Speedup for vecpar implementation over the base (sequential) CPU implementation, using cmath/eigen math backends, in simple/double precision, on Env2

Extreme load - 1 million tracks

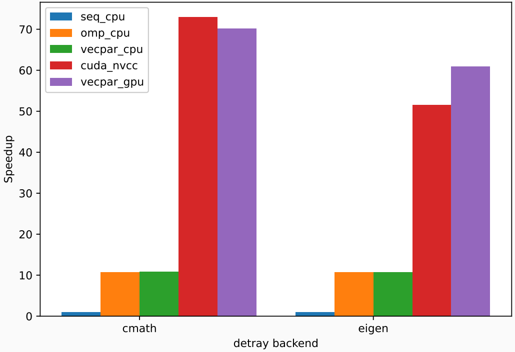


Figure 9: Speedup diagram for simplified RKN stepper descriebed on slide 16, cmath/eigen backends, in simple precision, 1 million tracks, fastmath disabled, on Env1

GPU Kernel statistics (via Nsight Compute Tool)

- Theoretical / achieved occupancy: 66.67% / 65.62%
- Execution time: 128ms
- Memory throughput: 130GB/s
- L1 / L2 cache hit rate: 84.91% / 92.63%

Aggregated results

Evaluation for a simplified RKN integrator

- Speedups of 27-70x over sequential implementation using the same code base compiled for different platforms
- Vecpar shows comparable wall-clock times to the hand-tuned OpenMP and CUDA implementations while vecpar is using one single-source file.

Current state and known limitations

vecpar library

- is open-source on github [5]
- supports x86 and NVIDIA GPUs
- uses vecmem library [6] to handle data structures and memory allocations.
  - currently only vecmem::vectors are supported as iterable collections for parallel loops
  - managed memory is fully supported by both backends while host memory is supported for CPU backend only (GPU work in progress)
- is covered by automated tests using googletest infrastructure [7]

Disclaimer: vecpar library is in early development phase!

[5] `https://github.com/wr-hamburg/vecpar`
[6] `https://github.com/acts-project/vecmem`
[7] `https://github.com/google/googletest`

Future development

- Offloading of an algorithm chain on a GPU
- Use of more complex data types (e.g. 2D vectors)
- Automatic performance optimizations
- A new (HIP) backend to target AMD GPUs

Suggestions and contributions are highly welcomed! Email me at georgiana.mania@desy.de

References

[Col22] ATLAS Collaboration. **ATLAS Software and Computing HL-LHC Roadmap.** Technical report, CERN, Geneva, Mar 2022.

[Hic20] Rich Hickey. **A history of clojure.** *Proc. ACM Program. Lang.*, 4(HOPL), jun 2020.

[SDSM] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. **Top500 list.** `https://www.top500.org/lists/top500/2021/11/`. Accessed: 2022-05-30.