Vecpar – A Framework for Portability and Parallelization*

Georgiana Mania[1,2][0000−0001−7536−5336], Nicholas Styles[1][0000−0001−6976−9457], Michael Kuhn[3][0000−0001−8167−8574], Andreas Salzburger[4][0000−0001−6004−3510], Beomki Yeo[5,6], and Thomas Ludwig[2,7]

[1] Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany
{georgiana.mania, nicholas.styles}@desy.de
[2] University of Hamburg, Hamburg, Germany
[3] Otto von Guericke University Magdeburg, Magdeburg, Germany
michael.kuhn@ovgu.de
[4] CERN, 1211, Geneva, Switzerland
andreas.salzburger@cern.ch
[5] Lawrence Berkeley National Laboratory, CA 94720, Berkeley, USA
[6] Department of Physics, University of California, CA 94720, Berkeley, USA
beomki.yeo@berkeley.edu
[7] Deutsches Klimarechenzentrum, Bundesstraße 45a 20146 Hamburg, Germany
ludwig@dkrz.de

Abstract. Complex particle reconstruction software used by High Energy Physics experiments already pushes the edges of computing resources with demanding requirements for speed and memory throughput, but the future experiments pose an even greater challenge. Although many supercomputers have already reached petascale capacities using many-core architectures and accelerators, numerous scientific applications still need to be adapted to make use of these new resources. To ensure a smooth transition to a platform-agnostic code base, we developed a prototype of a portability and parallelization framework named *vecpar*. In this paper, we introduce the technical concepts, the main features and we demonstrate the framework's potential by comparing the runtimes of the single-source vecpar implementation (compiled for different architectures) with native serial and parallel implementations, which reveal significant speedup over the former and competitive speedup versus the latter. Further optimizations and extended portability options are currently investigated and are therefore the focus of future work.

Keywords: Performance portability · Parallel computing · Heterogeneous computing

1 Introduction

High Energy Physics experiments, such as those at the Large Hadron Collider (LHC) at CERN [2], use complex algorithms to resolve the huge amounts of data

* This work was supported by DASHH under grant number HIDSS-0002.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

from their detectors into accurate descriptions of the particles of interest to be studied. The number of particles traversing the detector within a given time interval is frequently increasing, resulting in a significant rise in the multiplicity of discrete measurements. This will be further exacerbated in future, when the High-Luminosity LHC will provide events containing up to 10 000 charged particles to be reconstructed.

Multi-threading and accelerator support could be one possibility to alleviate this challenge. GPUs, many-core CPUs or architectures like ARM are efficient solutions in terms of FLOPS per watt, which makes them the perfect candidates for the world's most powerful supercomputers. As listed in the November 2022 edition of the TOP500, the ten highest-ranked clusters use a variety of processing units provided by different vendors including AMD, NVIDIA, IBM, Intel and ARM [23]. This great diversity comes with a major challenge for programmers: how to ensure code portability and maintainability, when each targeted platform requires different low-level assembly code that can be generated by a multitude of programming languages, language extensions, libraries or tools, some of which are vendor-proprietary.

Furthermore, there is also the question of how much the currently commonly used particle reconstruction software could benefit from the heterogeneous architectures, keeping in mind that the parallelization potential is limited in a number of places by its sequential nature. Thus, we designed *vecpar* to be an easy-to-use framework for parallelization targeting CPU and GPU with single-source C++ code, compiled for different platforms. Domain scientists can easily implement an algorithm without any previous knowledge of dedicated language extensions and test its performance on heterogeneous architecture, knowing that vecpar adds minimal overhead to native implementations. In the end, they can choose to either gradually port more complex algorithms to vecpar abstractions or to implement a specific native solution based on the performance results obtained by the prototype vecpar implementation.

This paper is organized as follows. In Section 2 we present an overview of the state of the art and related work. Then we introduce the vecpar framework in Section 3. Preliminary performance results are discussed in Section 4, while the conclusions and future work are summarized in Section 5.

2 State of the Art and Related Work

In the last 15 years, the HPC community has included support for co-processors, GPUs and FPGAs into existing state-of-the-art parallel programming standards like OpenMP [18] and developed new ones like OpenACC [17], OpenCL [7], and SYCL [20]. Even the C++ standard added concepts like parallel execution policies, polymorphic allocators, increased support for compile-time polymorphism and many others in a first step to unify the execution environment on heterogeneous devices. Additionally, each vendor provides native interfaces and dedicated compiler tools, which are optimized to ensure the best performance out of their hardware. Examples include CUDA [16], the parallel computing

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

platform for NVIDIA GPUs and Data Parallel C++ [19], the multi-architecture programming model proposed by Intel to target all their platforms: CPU, GPU and FPGA.

Unsurprisingly, these developments offer a trade-off between performance and portability [9], a difficult decision that needs to be made having several factors in mind: the available hardware for production environment, the programming skills needed to write the code, the effort to maintain potentially several native implementations targeting different platforms and last, but certainly not least, the parallelization potential of the scientific application. Consequently, this led to the development of several heterogeneous libraries and frameworks which offer the flexibility of having single-source C++ code compiled for different platforms while still ensuring good performance. With complex abstractions for memory and compute layouts, Kokkos [25] and Alpaka [26] deliver top performance but with some drawbacks in term of productivity considering that application developers are usually scientific domain experts rather than computer scientists. Other libraries like GrPPI [11] and SkePU [8] define similar abstractions to vecpar, but the former does not ensure GPU offloading support, while the latter is missing some features (like the filter skeleton) to accommodate some steps of the reconstruction flow (e.g. filtering the measurements above a specific threshold). Similar to these two libraries, vecpar aims to decouple scientific algorithms from parallelization strategies while automatically handling memory transfers, which can increase development productivity significantly.

Particle physicists from major experiments at CERN already started to adapt algorithms and event data models for parallel execution and GPU offloading in software frameworks like AthenaMT [13], CMSSW [6], ALICE O^2[21] and Allen [3]. A Common Tracking Software Project (ACTS) is an experiment-independent toolkit for track[8] reconstruction [4], which offers a realistic and thread-safe test-bed for R&D projects that explore heterogeneous architectures; these include the following three libraries. Firstly, *algebra-plugin* [1] is a linear algebra library with two available heterogeneous backends: *cmath*, which provides custom implementations in C++, CUDA and SYCL, and *eigen*, uses Eigen library [12] for data types and mathematical functions. Secondly, *vecmem* [24] is a heterogeneous memory management library with C++, CUDA, HIP and SYCL support, which defines several memory resources, iterable containers based on polymorphic allocators (e.g. `vector` and `jagged_vector`) and utility functions to support vector-related operations similar to the ones provided by the C++ Standard Library but enhanced with GPU-friendly features. Lastly, *detray* [22] is a header-only detector surface intersections based on algebra-plugin and vecmem.

While these libraries increase portability to different platforms and vendors by implementing the core algorithms inline to the GPU's restrictive requirements (e.g. no dynamic memory allocations and polymorphism), they move away from the concept of single-source repository because they provide dedicated backends for different architectures; this translates into a substantial development effort for domain scientists to maintain all of them. Moreover, the applications which

[8] A track is a charged particle trajectory through a detector.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

use these tools, must assemble a reconstruction chain for CPU and a different one for GPU. Vecpar is intended to close this gap by delivering comparable levels of performance portability when using a single-source implementation, without requiring any knowledge of parallelization strategies.

3 Proposed Approach

In this section we first introduce vecpar's underlying programming concepts and then we describe the framework's design, the Application Programming Interface (API) and associated compiler support.

Vecpar[9] is implemented as an open-source header-only library. To address the most common scenarios in particle reconstruction (but not necessarily limited to them), a series of abstractions were defined which allow the inversion of control of the execution flow from the application itself to the framework. This enables the separation of concerns between the scientific code and the parallelization strategies which can be extended, modified or replaced with minimum or even no impact on the invoking code.

Conceptually, vecpar is based on the *map-filter-reduce* notations and calculus from functional programming for specifying and manipulating computable functions over lists[5] and ensures a thread-safe environment by handling immutable data structures protected by *const correctness*[10]. It defines the main operators: `parallel_map`[11], `parallel_filter` and `parallel_reduce`, and the composed versions `parallel_map_filter` and `parallel_map_reduce`, which are implemented by each of the vecpar backends using different languages. Additionally, a generic `parallel_algorithm` operator is provided, which dispatches the execution to the appropriate implementation based on C++20 concepts and partial specializations, which were chosen because they employ compile-time polymorphism. The operators apply user-defined functions wrapped in `vecpar::algorithm`s on C++ vectors allocated either in host, device or CUDA unified memory using the vecmem library support for heterogeneous resources. For example, by extending the `vecpar::parallelizable_map` algorithm class which is templated on (a) the number of iterable collections and (b) the data types for input and output, the user has to provide an implementation for `mapping_function`, which is called by the framework at run-time by invoking a wrapper lambda function that will eventually be executed by each parallel thread. The algorithm classes are stateless and have no virtual functions. This ensures a straight-forward mapping to the GPU's memory, if needed. The infrastructure supports up to five collections of the same length that can be iterated over in the same time.

Currently, vecpar fully supports two parallel execution backends: CPU using OpenMP threads and GPU using CUDA threads. An experimental backend based

[9] https://github.com/wr-hamburg/vecpar

[10] As an exception motivated by performance optimization purposes, the vecpar API provides a limited number of mutable versions as well.

[11] A `parallel_mmap` operator which allows mutable data structures is also defined.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

on OpenMP target was recently implemented, in order to extend portability to AMD GPUs. Similarly, new backends could be easily added to the generic dispatch system. The execution flow is summarized in Fig.1. To run a vecpar algorithm in a parallel (and potentially GPU offloaded) manner, it has to be passed to a specific API function like `vecpar::parallel_map(algorithm,...)` or the generic `vecpar::parallel_algorithm(algorithm,...)` which would delegate the execution to the parallel implementation using OpenMP or CUDA, respectively. In this case, the decision is made implicitely based on evaluating internal flags like `__CUDA__` or `_OPENMP` which are set by the compiler based on information provided either by the user (e.g. through compilation flags) or inferred at compile/link time (e.g. available support for OpenMP). There is also the option to invoke a specific backend directly; for example calling `vecpar::cuda::parallel_algorithm(algorithm,...)` bypasses the dispatch system. This could be particularly useful when two imbricated levels of parallelism are required, for instance when reading multiple event data files from disk in a CPU multi-threaded environment while offloading the computations associated with each event to the GPU.

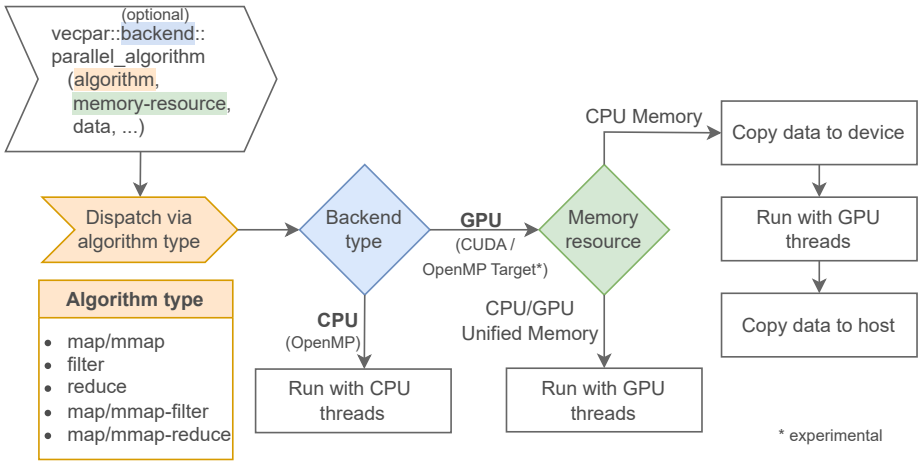


Fig. 1: Decision flow in vecpar

If an algorithm doesn't necessarily fit into one of the map-filter-reduce patterns or the input data structures are different from the ones expected by the predefined base classes, vecpar offers a generic `vecpar::parallel_map` function which can be invoked with a lambda that defines the behavior, provided by the user. While this offers more flexibility and ensures that the anonymous function can be executed as a GPU kernel, it passes the responsibility of data management and parallelization to the user implementation.

In the case of a GPU parallelization, the framework handles not only the distribution of independent work items to different threads, but also the required

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

memory transfers between host and device, using vecmem library support. Vecpar's CUDA backend provides separate implementations for executing parallel algorithms handling collections stored in managed memory and host memory. This is an important distinction since the former requires only a retrieval of a pointer to an unified memory address, while the latter requires explicit memory copy between host and device for both input and output.

The algorithm chaining functionality simplifies the implementation of a use case which requires several algorithms as intermediate steps to reach a final result. As a precondition, these algorithms need to be mathematically composable or otherwise a compile-time error will be shown. Vecpar's implementation calls `parallel_algorithm(algorithm,...)` on each algorithm, in order, using the result from the previous one as input for the current one. The execution flow of the chain is then reduced to the one described above in Fig. 1. To use this feature, an instance of a `vecpar::chain` templated on the input and output data types is needed as a first step. Then the algorithm instances and the input data associated with the first algorithm are passed to the chain through calls to its member functions as showed in Listing 1.1.

```
vecpar::chain<vecmem::host_memory_resource,
              double,
              vecmem::vector<int>> chain(mr);

chain.with_config(config) // optional
     .with_algorithms(alg_1, alg_2, alg_3)
     .execute(vecmem_vector, context_object);
```

Listing 1.1: Algorithm chaining definition example

The user has the option to configure the parallelization by providing a specific number of workers in a `vecpar::config` object when invoking a parallel function or parallel chain. Without passing it or by leaving it empty, a default configuration is generated based on the problem size for the CUDA backend and the runtime environment variables for the OpenMP backend. At the moment, the OpenMP Target backend does not support user configuration and uses a default one for optimization purposes.

The `GNU gcc` compiler can be used to compile vecpar code to target `x86_64` and `aarch64` architectures and when built with offloading capabilities, it can also target NVIDIA and AMD GPUs through vecpar's OpenMP target backend. In addition to what is offered by `gcc`, `LLVM/clang` compiler can additionally generate NVIDIA's Parallel Thread Execution assembly code (NVPTX) and therefore target NVIDIA GPUs with native assembly, which makes it more versatile for building heterogeneous code. `ROCm/aomp` can also be used to compile for CPU and AMD GPUs using the OpenMP target backend. An NVIDIA compiler which supports C++20 is required to build vecpar sources.

4 Evaluation

In this section, we demonstrate the benefits of using vecpar by evaluating it in comparison to native state-of-the-art parallel solutions like OpenMP (OMP),

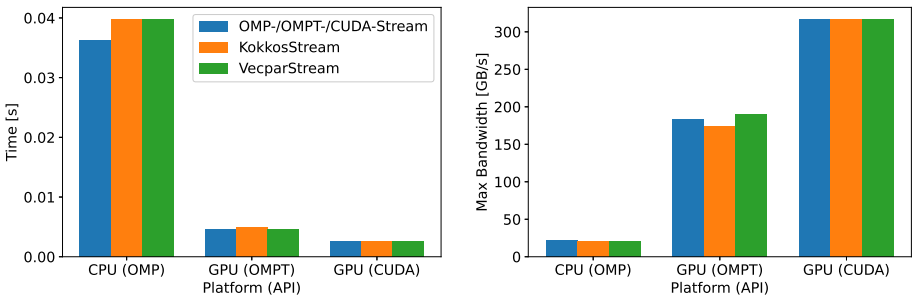
ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

OpenMP Target (OMPT) and CUDA, and to similar approaches like Kokkos, in several use cases, starting from trivial kernels to more complex scenarios from particle reconstruction software.

Four environments were used for running the experiments: **Env1** – Intel Core i7-10870H CPU and NVIDIA GeForce RTX 3060 GPU, **Env2** (HPC cluster node) – 20-cores Intel Xeon Gold 5115 CPU and NVIDIA Tesla V100 GPU, **Env3** (Raspberry Pi Cluster node) – 4-cores ARM Cortex A72 CPU, and **Env4** – Intel i5-8600K CPU and AMD Radeon RX6750 XT. `Clang 14` (with offload support) was used as default C++ compiler for the Env1 and Env2, `gcc 12` was used for the third one, while `aomp` 16 was used for Env4. CUDA 11.6 (driver v.510.47.03), ROCm 5.3.3 and OpenMP 4.5 ensured the support for parallelism.

4.1 BabelStream Benchmark

The BabelStream benchmark [10] defines a framework to evaluate (a) the wall clock execution times and (b) the memory throughput of simple mathematical operations when using different parallelization APIs and/or compilers. To achieve this, different abstractions are required for memory allocations (and GPU transfers if needed) and kernel execution. This clear separation is not transparent in a vecpar algorithm implementation since memory handling is a built-in feature of the library, as previously shown in Fig. 1; nevertheless, vecpar offers a lambda-based feature, which allows more flexibility to the user by trading off strictly single-source nature of the implementation. This lambda implementation is used for the GPU-CUDA benchmark while single-source vecpar algorithms were used for all the other scenarios. The vecpar branch for BabelStream is available open-source on github[12]. For the present results, we used the *triad* benchmark which performs $a_i = b_i + scalar \times c_i$, $\forall\, 0 \leq i < 2^{25}$, with double-precision operands. This kernel is run 100 times and the mean execution time (which does



(a) Execution time (b) Bandwidth

Fig. 2: BabelStream `triad` kernel, using vectors of 2^{25} FP64, on Env1

[12] https://github.com/wr-hamburg/BabelStream/tree/vecpar

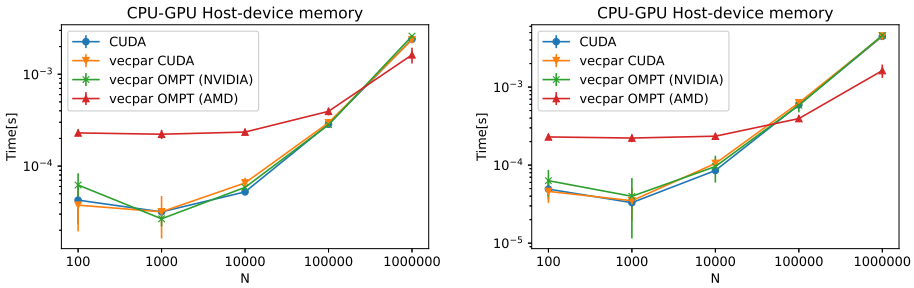
ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

not include the necessary time for memory transfers in the case of the GPU) is computed. The measurements shown in Fig. 2 prove that vecpar achieves comparable performance with native parallel implementations; for the CPU, there is a deviation of up to $\approx 10\%$ for both vecpar and kokkos OMP backends, while the GPU targeting implementations (using OMPT and CUDA) are within $\pm 1\%$ of the other benchmarks.

4.2 Vecpar Internal Benchmark

To evaluate the performance and portability of a single-source vecpar algorithm compiled for different platforms, we designed a custom benchmark, available in the vecpar repository. In this case, we include CPU-GPU memory transfer times in the total execution time of a given scenario. Each test was repeated 20 times.

The Single and Double Precision $a \times X + Y$ (SAXPY/DAXPY) benchmarks provides three implementations: OpenMP, CUDA and vecpar (using different backends), which perform the same operation: $y_i = a \times x_i + y_i,\ x_i \in X,\ y_i \in Y,\ \forall\, 0 \leq i < N$ and the code is compiled with `clang` for the NVIDIA GPU and with the `aomp` compiler for the AMD GPU. Fig. 3 shows that the overhead of vecpar in comparison to the native CUDA implementations on NVIDIA hardware is up to 0.5%. The vecpar OpenMP target code compiled for AMD GPU seems to be slower for smaller problem size while showing a 4× speedup for vectors of one million elements; this is most likely credited to the link-time optimization of the LLVM compiler.



(a) SAXPY (b) DAXPY

Fig. 3: Mean and standard deviation for SAXPY and DAXPY kernels, using vectors of different sizes N, on Env1 and Env4

4.3 Track Reconstruction Use Cases

In order to establish the position and momentum of charged particles in a magnetic field, the second order differential equation of motion is integrated to provide a numerical solution which is later corroborated with detector measurements to obtain a realistic estimate. The 4th order Runge-Kutta-Nyström (RKN) algorithm is widely used for its precision but it has the downside of being inherently

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

sequential, with each stage depending on the calculations of the previous one. The *detray* project features an implementation of an algorithm based on adaptive RKN methods [14,15] (*RKN stepper*) which provides estimates for the transport of track parameters [13] and their covariance matrices, either in global coordinates (i.e. free parameters) or in local coordinates based on an intersection surface (i.e. bound parameters) through the detector layers. For each experiment a different executable is built for every linear algebra backend from *algebra-plugin* project and then ran on different platforms. For all the tests, the wall-clock times include host-device transfers when the data is in host memory and a GPU is used to compute the results. In case the data is already in managed memory, the time spent for initialization is not factored in.

Track Parameters Estimation The first case that we explored is a simplified STEP algorithm which provides estimates for track parameters only (without computing the Jacobians) and limits the numerical integration to 100 steps in both forward and backward direction. Moreover, the magnetic field is assumed constant in z direction. For each test, we simulate 10 000 (free) tracks starting from the center of the detector, with a negative unit charge and a magnetic field of (0, 0, 2T).

To evaluate the development productivity, we first look at the implementation details. The CUDA and vecpar RKN kernels in Listing 1.2 and Listing 1.3[14]

```
__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);
    // Prevent overflow
    if (gid >= tracks.size()) {
        return;
    }
    // Get a track
    auto& traj = tracks.at(gid);
    // Define RK stepper
    rk_stepper_type rk(B);
    // Index for stepping
    unsigned int i_s=0;
    // Forward direction
    rk_stepper_type::state forward_state(traj);
    for (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 (i_s=0; i_s<rk_steps; i_s++) {
        rk.step(backward_state);
    }
}
```

Listing 1.2: CUDA

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

  TARGET free_track_parameters& mapping_function(
      free_track_parameters& traj, vector3 B) override
       {
    // Define RK stepper
    rk_stepper_type rk(B);
    // Index for stepping
    unsigned int i_s=0;
    // Forward direction
    rk_stepper_type::state forward_state(traj);
    for (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 (i_s=0; i_s<rk_steps; i_s++) {
        rk.step(backward_state);
    }
    return traj;
  }
};
```

Listing 1.3: C++/vecpar

show that while the former requires knowledge about accessing the data from unified memory and thread parallelization based on global index, the latter fits the code into vecpar's *map* abstraction templated on the input and output data

[13] A set of parameters describing the helical trajectory followed by a charged particle moving within a magnetic field.

[14] `TARGET` is a vecpar macro which adds extra qualifiers at compile time.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

types and can just focus on the actual hardware-agnostic algorithm which is **identical** in both cases, as highlighted in the rectangle. Moreover, the vecpar implementation can be compiled with `clang` for x86_64 and NVIDIA GPU platforms without further code changes, which ensures portability and productivity by avoiding to maintain distinct code sources for CPU and GPU.

In terms of performance, we compared the vecpar single-source implementation compiled for different platforms (`vecpar_cpu` and `vecpar_gpu`) with the initial sequential version (`seq_cpu`), a hard-coded OpenMP version (`omp_cpu`) and a CUDA version (`cuda_nvcc`) and we evaluated the impact of different factors like the problem size, the precision of the operands and the number of parallel workers on Env1 and Env2. Fig. 4 shows the results for the tests focusing on single precision; both OpenMP and vecpar versions provide close to ideal strong scaling performance, when using up to 40 threads, on Env2.

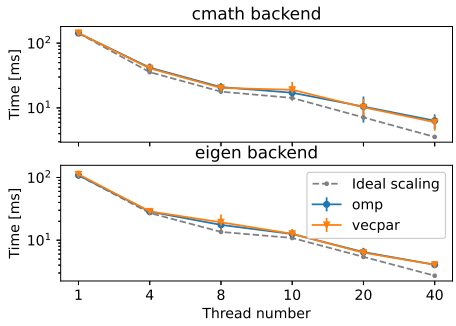


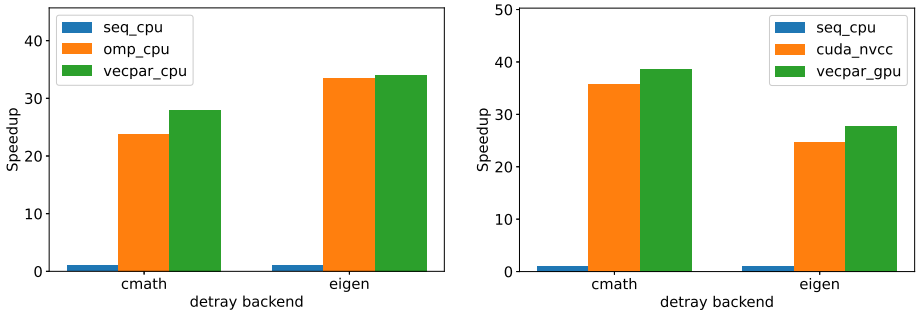
Fig. 4: Strong-scaling evaluation: Multi-threading implementations of the RKN stepper, in single precision, cmath/eigen backends, on Env2

A closer analysis shows that the parallel versions are faster than the sequential one for double precision operands on both linear algebra backends (cmath and eigen), for both CPU and GPU, as depicted in the speedup diagrams from Fig. 5. For the GPU, the advantage of the vecpar solution over the native CUDA is assumed to be due to NVPTX optimizations done by `clang` since compiling the same RKN kernel with `clang` already shows a slight speedup over the executable produced by `nvcc`.

Fig. 6 summarizes these experiments with the conclusion that for this given scenario, the vecpar single-source implementation ensures a significant speedup of $28 - 65\times$ over the initial sequential implementation, comparable to native OpenMP and CUDA implementations.

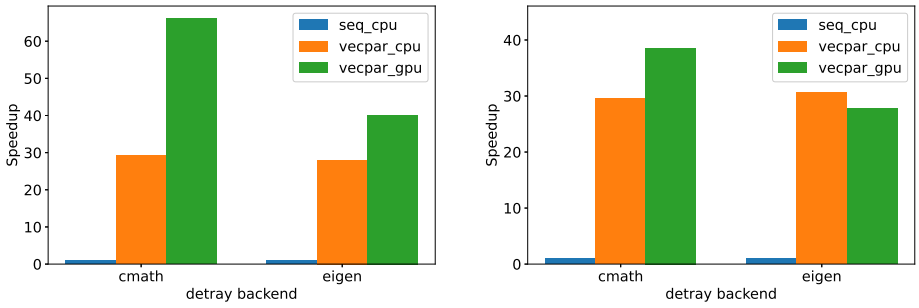
Extreme Load Use Case We evaluated the performance of the vecpar implementation in an extreme-load test of one million tracks using 16 OpenMP threads and 3907×256 CUDA threads. The results in Fig. 7 show that the vecpar implementation is comparable with native ones, showing up to $12\times$ and $70\times$ speedups for CPU and GPU respectively over the initial sequential one.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18



(a) CPU (b) GPU

Fig. 5: Speedup factors of the vecpar single-source implementation compiled for CPU and GPU over other implementations, on Env2



(a) Single precision (b) Double precision

Fig. 6: Speedup for vecpar implementation over the sequential CPU implementation, using cmath/eigen math backends, in single/double precision, on Env2

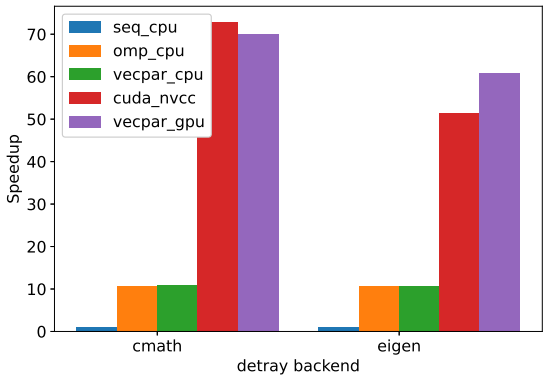


Fig. 7: Speedup diagram for simplified RKN stepper using cmath/eigen backends, in single precision, one million tracks, without fastmath support, on Env1

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

Track parameter and its covariance estimation The third set of experiments is already very close to common operations in track reconstruction and it is based on a recent implementation of a more realistic RKN stepper which also includes error-propagation while assuming the same constant magnetic field. This adds more complexity since it computes partial derivatives of the track parameters. Since the double precision use case is more realistic for a production environment, we focus the experiments on this level of precision; this also includes disabling the fastmath support features offered by some compilers.

Table 1 shows a selection of results obtained by estimating 10 000 free track parameters (including error propagation) in double precision on different platforms. The same vecpar C++ source file is compiled for CPU and GPU, and compared against the initial sequential version and against native parallel solutions (using OpenMP threads for CPU and CUDA threads for GPU). The measurements involve using the cmath backend for detray, but similar results were obtained with the eigen backend, but they are omitted due to space restrictions.

Platform	Sequential	OpenMP/CUDA	Vecpar
`ARM aarch64` (Env3)	4.54 ±0.04%	1.16 ±4.11%	1.27 ±9.78%
`Intel x86_64` (Env2)	1.69 ±5.51%	0.1058 ±4.75%	0.1052 ±5.41%
`NVIDIA V100` (Env2)	-	0.004 ±6.32%	0.02 ±1.86%

Table 1: Mean time (in seconds) and standard deviation for estimating 10 000 tracks using sequential, parallel and vecpar implementations on different CPU/GPU platforms

Firstly, we explored the extended portability to `aarch64`; the vecpar implementation is within ±9% of the native implementation and 3.55× faster than the sequential one. This is close to the maximum theoretical speedup of 4× which is limited by the number of OpenMP threads (which is 4 in this case). For the `x86_64` platform, the vecpar implementation is within ±3% of the OpenMP implementation while being 15× faster than the sequential one. Secondly, for the GPU case, regardless of the storage location, the vecpar implementation takes longer than native CUDA compiled with `nvcc` and data in managed memory. Despite much better memory and computation throughput, there are several uncoalesced global memory accesses due to unnecessary duplication of the offloaded algorithmic code. We expect this overhead to be alleviated by using a CUDA 11.7 feature: `__grid_constant__`; this will store the algorithm object into GPU constant cache which provides much better latency than global memory. Experimental validation should be possible as soon as LLVM/clang enables support for it.

Nevertheless, there is still a benefit of having a single parallel implementation which can be executed on different platforms as it is shown in Fig. 8. Speedups up to 19× and 108× can be observed for CPU and GPU respectively.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

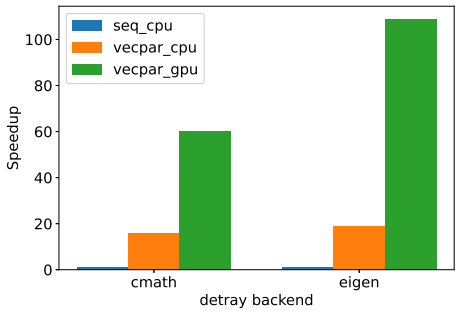


Fig. 8: Speedup of the vecpar implementation for RKN stepper (with error propagation) for 10 000 tracks over the sequential implementation for cmath and eigen backends, in double precision, on Env2

5 Conclusions and Future Work

In this paper, we presented a new framework for efficient parallelized execution of charged particle tracking[15]. It relies on new C++ features, OpenMP and CUDA to ensure improved performance and portability while decoupling the scientific algorithms from the parallelization strategies. Although vecpar is still in an early development phase, it was successfully used to port a simplified step of the track reconstruction flow and demonstrated its potential by obtaining speedups up to 108× over sequential implementations using single-source C++ code. Moreover, it provides speedups competitive with those obtained by both native and related APIs, while being easier to implement and port to different architectures. We showed its portability by testing it on different platforms like `x86_64`, `ARM`, `NVIDIA Volta` and `Ampere`, and `AMD Radeon`.

In future versions, we plan to improve the memory access patterns for the CUDA backend, to finalize the development of the OpenMP target backend and to extend the abstractions to allow more complex patterns like hierarchical parallelism and uneven distributed workloads.

Acknowledgment

We acknowledge the support by DASHH (Data Science in Hamburg - HELMHOLTZ Graduate School for the Structure of Matter) with the Grant-No. HIDSS-0002. The National Analysis Facility (NAF) at Deutsches Elektronen-Synchrotron (DESY), the University of Hamburg (UHH) and Deutsche Klimarechenzentrum (DKRZ) provided the hardware resources for the experiments.

[15] While the initial goal was to contribute to the increase of the performance portability of open-source track reconstruction software, other scientific use cases are welcome in the future.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

References

1. Algebra-plugin, https://github.com/acts-project/algebra-plugins/, Last accessed: 8 Dec 2022
2. European Organization for Nuclear Research (CERN). Nature **184**(4702), 1844–1844 (Dec 1959). https://doi.org/10.1038/1841844b0
3. Aaij, R., Albrecht, J., Belous, M., Billoir, P., Boettcher, T., et al.: Allen: A High-Level Trigger on GPUs for LHCb. Computing and Software for Big Science **4**(1) (apr 2020). https://doi.org/10.1007/s41781-020-00039-7
4. Ai, X., Allaire, C., Calace, N., Czirkos, A., Ene, I., Elsing, M., et al.: A Common Tracking Software Project. Computing and Software for Big Science (2022). https://doi.org/https://doi.org/10.1007/s41781-021-00078-8
5. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. pp. 5–42. Springer Berlin Heidelberg, Berlin, Heidelberg (1987). https://doi.org/10.1007/978-3-642-87374-4_1
6. Bocci, A., Kortelainen, M., Innocente, V., Pantaleo, F., Rovere, M.: Heterogeneous reconstruction of tracks and primary vertices with the CMS pixel tracker (2020). https://doi.org/10.48550/ARXIV.2008.13461, https://arxiv.org/abs/2008.13461
7. Breitbart, J., Fohry, C.: OpenCL - An effective programming model for data parallel computations at the Cell Broadband Engine. In: 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings. pp. 1–8. IEEE (2010). https://doi.org/10.1109/IPDPSW.2010.5470823
8. Dastgeer, U.: Skeleton Programming for Heterogeneous GPU-based Systems (2011)
9. Deakin, T., Poenaru, A., Lin, T., McIntosh-Smith, S.: Tracking Performance Portability on the Yellow Brick Road to Exascale. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 1–13 (2020). https://doi.org/10.1109/P3HPC51967.2020.00006
10. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: Evaluating attainable memory bandwidth of parallel programming models via BabelStream. International Journal of Computational Science and Engineering **17**(3), 247–262 (2018). https://doi.org/10.1504/IJCSE.2018.095847, special Issue on Novel Strategies for Programming Accelerators
11. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurrency and Computation: Practice and Experience **29**(24), e4175 (2017). https://doi.org/https://doi.org/10.1002/cpe.4175, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4175
12. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
13. Leggett, C., Baines, J., Bold, T., Calafiura, P., Farrell, S., van Gemmeren, P., et al.: AthenaMT: upgrading the ATLAS software framework for the many-core world with multi-threading. Journal of Physics: Conference Series **898**, 042009 (oct 2017). https://doi.org/10.1088/1742-6596/898/4/042009
14. Lund, E., Bugge, L., Gavrilenko, I., A., S.: Track parameter propagation through the application of a new adaptive Runge-Kutta-Nystrom method in the ATLAS experiment. Tech. rep. (Jan 2009), https://cds.cern.ch/record/1113528/files/ATL-SOFT-PUB-2009-001.pdf
15. Lund, E., Bugge, L., Gavrilenko, I., A., S.: Transport of covariance matrices in the inhomogeneous magnetic field of the ATLAS experiment by the application of a semi-analytical method. Tech. rep. (Jan 2009), https://cds.cern.ch/record/1114177/files/ATL-SOFT-PUB-2009-002.pdf

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18

16. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. In: International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2008, Los Angeles, California, USA, August 11-15, 2008, Classes. pp. 16:1–16:14. ACM (2008). https://doi.org/10.1145/1401132.1401152
17. Organization, O.: The OpenACC Application Programming Interface, Version 3.2, https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf
18. van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP - The Next Step: Affinity, Accelerators, Tasking, and SIMD. MIT Press (2017), https://mitpress.mit.edu/books/using-openmp-next-step
19. Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., Tian, X.: Data Parallel C++. Apress Berkeley, CA (2021). https://doi.org/https://doi.org/10.1007/978-1-4842-5574-2
20. Reyes, R., Lomüller, V.: SYCL: Single-source C++ accelerator programming. In: Joubert, G.R., Leather, H., Parsons, M., Peters, F.J., Sawyer, M. (eds.) Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK. Advances in Parallel Computing, vol. 27, pp. 673–682. IOS Press (2015). https://doi.org/10.3233/978-1-61499-621-7-673
21. Rohr, D., Gorbunov, S., Schmidt, M.O., Shahoyan, R.: Track Reconstruction in the ALICE TPC using GPUs for LHC Run 3 (2018). https://doi.org/10.48550/ARXIV.1811.11481, https://arxiv.org/abs/1811.11481
22. Salzburger, A., Niermann, J., Yeo, B., Krasznahorkay, A.: Detray: a compile time polymorphic tracking geometry description. Journal of Physics: Conference Series **2438**(1), 012026 (feb 2023). https://doi.org/10.1088/1742-6596/2438/1/012026
23. Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: Top500 List. https://www.top500.org, accessed: 2022-12-01
24. Swatman, S.N., Krasznahorkay, A., Gessinger, P.: Managing heterogeneous device memory using C++17 memory resources. Journal of Physics: Conference Series **2438**(1), 012050 (feb 2023). https://doi.org/10.1088/1742-6596/2438/1/012050
25. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., et al.: Kokkos 3: Programming Model Extensions for the Exascale Era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
26. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka - An Abstraction Library for Parallel Kernel Acceleration. IEEE Computer Society (May 2016), http://arxiv.org/abs/1602.08477

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-35995-8_18