A C++ Cherenkov photons simulation in CORSIKA 8

Matthieu Carrère[1,2,*], *Luisa* Arrabito[1], *Johan* Bregeon[4], *David* Parello[2,3], *Philippe* Langlois[2,3], and *Georges* Vasileiadis[1]

[1]LUPM : Laboratoire Univers et Particules de Montpellier, France
[2]DALI : Digits, Architectures et Logiciels Informatiques, Perpignan, France
[3]LIRMM : Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, France
[4]LPSC : Laboratoire de Physique Subatomique et de Cosmologie, Grenoble, France

Abstract. CORSIKA is a standard software for simulations of air showers induced by cosmic rays. It has been developed mainly in Fortran 77 continuously over the last thirty years. It has become very difficult to add new physics features to CORSIKA 7. CORSIKA 8 aims to be the future of the CORSIKA project. It is a framework in C++17 which uses modern concepts in object oriented programming for an efficient modularity and flexibility. The CORSIKA 8 project aims to attain high performance by exploiting techniques such as vectorization, gpu/cpu parallelization, extended use of static polymorphism and the most precise physical models available. In this paper, we focus on the Cherenkov photon propagation module of CORSIKA, which is of particular interest for gamma-ray experiments, like the Cherenkov Telescope Array. First, we present the optimizations that we have applied to the Cherenkov module thanks to the results of detailed profiling using performance counters. Then, we report our preliminary work to develop the Cherenkov Module in the CORSIKA 8 framework. Finally, we will demonstrate the first performance comparison with the current CORSIKA software as well as physics validation.

Acknowledgments

This work was conducted in the context of the CTA Consortium then was performed within and together with the CORSIKA 8 Collaboration.

1 Introduction

CORSIKA[2] is a software for the simulation of air shower development written in C/Fortran 77. It is used by a wide community in the fields of gamma-ray astronomy, neutrino, radio astronomy and cosmic rays. In this paper we focus in particular on the use case of the Cherenkov Telescope Array[3] (CTA) project and on the module processing the propagation of the Cherenkov photons through the atmosphere. CTA is the next generation ground-based observatory for gamma-ray astronomy at very-high energies up to about 300 TeV. It is composed of two arrays of Cherenkov telescopes deployed in the two hemispheres in La Palma (Spain) and Paranal (Chile). CTA regularly performs massive simulation campaigns

*e-mail: matthieu.carrere@lupm.in2p3.fr

using EGI[1] resources. It represents more than 200 million HS06 CPU hours[2] per year with about 70% spent in CORSIKA. In order to reduce this computing time, we have studied different strategies of code optimization[1]. In Section 2, we present the final steps of this optimization work applied to the most recent CORSIKA 7 version. We will show how our optimizations allow us to better exploit the capabilities of modern processors, in particular enhancing auto-vectorization and efficient memory usage. Moreover, our approach in optimizing the code aimed to keep the same numerical accuracy and physics results with respect to the original version. The optimized code has been successfully used in production during the last large-scale campaign performed by the CTA consortium in 2020. For the current production Prod5, which started in July 2020, we estimate that the optimized code has saved us around 12 million HS06 CPU hours for 72 million HS06 CPU hours consumed so far. During 30 years, the core of CORSIKA did not change and adding new features or optimizations has become a complex task. In this context, the CORSIKA 8 project [5] aims to entirely rewrite CORSIKA in modern C++ (C++ 17) to obtain a modular, flexible and efficient code. It is always a challenge to re-write an old scientific code in a modern framework, while keeping compatible physics results and high performance. For the use case of CTA, the Cherenkov module was the most CPU consuming part and we expect the same behavior in CORSIKA 8. Indeed for each shower, billions of Cherenkov photons are produced and their positions, directions and arrival times must be calculated taking into account the properties of the traversed atmosphere. In the framework of CORSIKA 8, we have thus started the development of the Cherenkov module and obtained the first physics and performance comparisons with CORSIKA 7 (our reference). This work is presented in Section 3. Table 1 summarizes the different CORSIKA versions (that we have) developed and compared both for the optimization done in CORSIKA 7 and in the Cherenkov module development in CORSIKA 8. Finally, in Section 4 we present conclusions and prospects to continue to improve the performance and the quality of the simulations.

All the simulations in this paper have been produced with input parameters typically used in CTA productions. In particular, we have been using gamma incident particles in an energy range of 3 GeV - 330 TeV and for the conditions of the CTA South site at Paranal (geomagnetic field, altitude). In order to obtain reproducible results, we have also fixed the pseudo-random number generator seeds. Finally, CORSIKA 7 optimizations have been tested with runs of 5000 showers, whereas CORSIKA 8 Cherenkov module has been validated by injecting particle tracks from 10 showers generated with CORSIKA 7.

Table 1: CORSIKA Versions

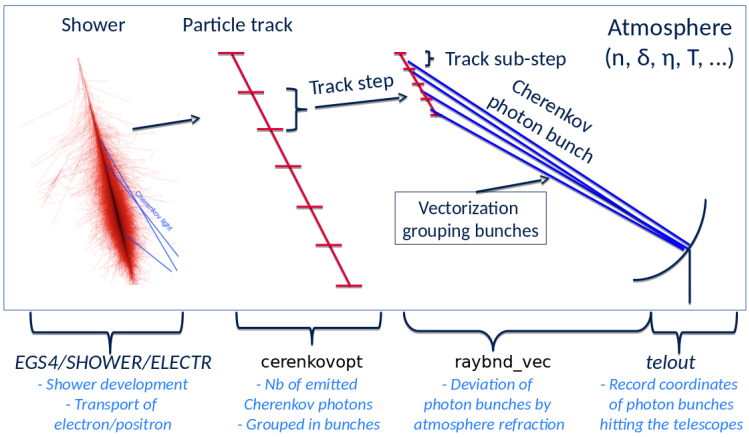
Version	Description
CORSIKA 7.6900-ORG	Original 7.6900 version with scalar operations
CORSIKA 7.6900-OPT-V1	Optimized 7.6900 with vectorial operations (with *vector-libm*)
CORSIKA 7.6900-OPT-V2	7.6900-OPT-V1 with memory and vector size optimizations
CORSIKA 7.7100	Similar version of 7.6900-OPT-V2
CORSIKA 8 CherenkC7Test Module	7.7100 rewrites with oriented object to read CORSIKA 7 inputs (with/without *vector-libm*)

[1]European Grid Infrastructure
[2]https://w3.hepix.org/benchmarking.html

2 CORSIKA 7 - Optimizations

2.1 Cherenkov Simulation in CTA

Figure 1: Cherenkov photons simulation in CORSIKA 7.6900-OPT-V1



The main steps of Cherenkov simulation in CORSIKA and the associated functions are illustrated in Figure 1. Each particle produced in the shower undergoes a series of physics interactions. Between two interactions the particle is transported from one point to another (track step). For each track step, the number of Cherenkov photons induced is calculated. In order to save computing time, Cherenkov photons are grouped into bunches of 5 photons and then propagated through the atmosphere. Track steps are thus further subdivided into sub-steps, so that one photon bunch is emitted for each sub-step. Given the fact that the different bunches propagate independently from one another, it is possible to group their coordinates into vectors and vectorize the code. A precise description of the atmosphere allows us to take into account the effect of refraction on the photon bunches directions. In particular, the values of air density, thickness and refraction index are tabulated at different altitudes. Interpolation is then used to obtain the properties of the atmosphere at intermediate altitudes. Telescope arrays are represented by a grid of telescope positions, each telescope being represented by a fiducial sphere. Once propagated to the ground, only the photon bunches intersecting a telescope are saved in the final output. In CORSIKA, the Cherenkov photon propagation, the atmospheric model and the geometry of the telescope arrays are handled in a dedicated package (IACT/atmo). By profiling CORSIKA 7 with *Linux Perf* tool, we have identified the most CPU consuming functions. These can be grouped into 3 categories each representing about 1/3 of the overall CPU consumed: elementary mathematical functions, Cherenkov simulation functions and a set of other smaller functions. All the measurements reported in this section have been done on a dedicated server[1].

2.2 Vectorization

The first optimization work done in CORSIKA 7 consisted in enhancing SIMD (Single Instruction Multiple Data) auto-vectorization (CORSIKA 7.6900-OPT-V1 in Table 1), as de-

[1]Intel Xeon Gold 5122/3.6Ghz/L3:16.5MB, RAM:128GB with gcc/gfortran 8.4 and optimization flags: -O3 -mavx2

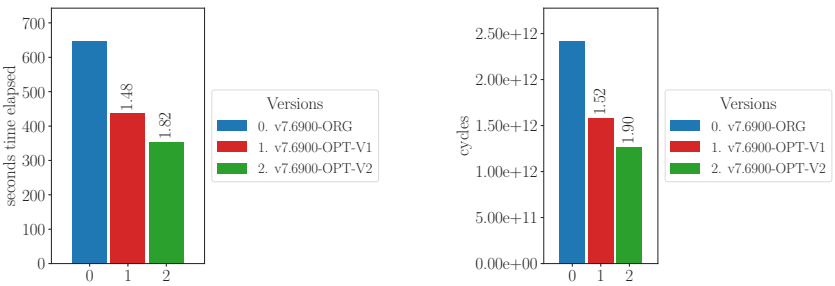
tailed in Ref. [1]. Indeed, all modern processors perform operations in parallel thanks to vector SIMD registers. These specialized registers allow us to use vector instructions performing one type of operation for multiple data in one cycle. Many technologies exist for different architectures. In our case, we have considered the x86 instruction set[1], and more specifically AVX2[2] (AVX-512 did not provide here significant speed-up and is not yet widespread in the grid computing centers). In order to benefit from SIMD vectorization, there are two main options. The first is automatic vectorization by the compiler (auto-vectorization) while the second is to use manually intrinsic functions which depend on the target architecture. Auto-vectorization has been chosen because it allows us to easily obtain a portable code with good performance. Writing code using intrinsic functions is a difficult task and it requires to write a specialized version for each instruction set.

In version 7.6900-OPT-V1, Cherenkov functions have been rewritten to enable auto-vectorization and the vectorization ratio τ_{vec} (cf. Equation (1)) increased from 0.57% to almost 10%. The main bonus comes from the application of *vector-libm*[4][3], a specialized vector mathematical library where the overall vectorization ratio τ_{vec} increased to 59.52%. Indeed as 1/3 of computing time is used in elementary function call, using the *vector-libm* for the relevant parts of the code increased the vectorization rate to a significant speed-up. We measure a 1.48 overall speed-up for the vectorized version 7.6900-OPT-V1 using AVX2 instructions.

In principle, we expect even better performances by enabling AVX-512 instructions, since it allows us to execute twice more operations in parallel compared to AVX2. However, we measure slightly better elapsed time performances with AVX2 whereas AVX512 execution had a lower number of cycles. This behavior is explained as the clock frequency is automatically reduced to avoid chip heating on Intel CPUs when AVX512 is enabled. A similar effect also appears with AVX2 : Figure 2 exhibits some speedup differences between the number of cycles and the elapsed time.

$$\tau_{vec} = \frac{\sum 2 * I_{128,dbl} + 4 * I_{256,dbl} + 8 * I_{512,dbl}}{\sum 2 * I_{128,dbl} + 4 * I_{256,dbl} + 8 * I_{512,dbl} + \sum I_{scal,dbl}} \tag{1}$$

Equation 1: Vectorization ratio with the number of instructions for each instruction set (scalar operations, vectorized operations in 128 bits, 256 bits and 512 bits) in double precision.



(a) Total time elapsed (b) Total cycles

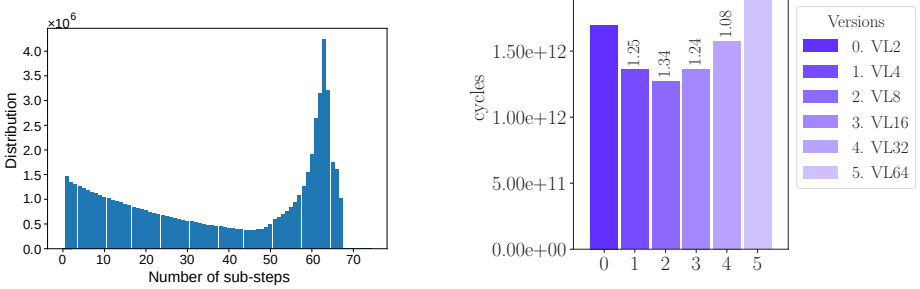
Figure 2: Comparision on CORSIKA 7.6900-ORG/OPT-V1/OPT-V2

[1] instruction collection to perform basic math operations
[2] 256 bits registers to calculate in parallel 4 floaty doubles or 8 floaty singles
[3] https://gitlab.com/cquirin/vector-libm

2.3 Vector length tuning

The vector length of static arrays used in the vectorized Cherenkov function of CORSIKA 7.6900-OPT-V1 can be tuned at the compilation time. The Cherenkov function contains two main loops over particle sub-steps (see Figure 1), to compute the propagation that include several calls to the *vector-libm*. The two loops perform the same computations but the first time is vectorized loop while the second one remains scalar. For a given particle step, the vector length parameter also determines how the workload is subdivided between the vector and the scalar loop. If static arrays cannot be entirely filled, the remaining computations will be done in the scalar loop. We can thus argue that the vector length parameter has an impact on overall performances.

However, the optimal value of this parameter depends on several factors and is difficult to predict. For instance among these factors, the number of sub-steps is not uniform, as shown by the distribution in Figure 2(a). A too high value of vector length would imply a larger fraction of computations performed in the scalar loop. On the other side, a too low value implies a large number of loops and branches, which has a negative impact on performances. Finally, the *vector-libm* delivers best performances for the vector length equals to 8 as reported in Ref [4] and by our experiments presented in Figure 2(b).

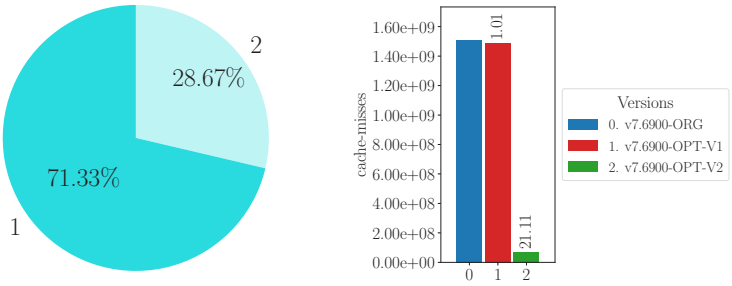


(a) Sub-steps distribution (b) Vector length comparison

Figure 3: Analyses on CORSIKA 7.6900-OPT-V1/OPT-V2

2.4 Memory usage optimization

Since further vectorization required profound changes in the code, either the logic of algorithms or the data structures, we have examined other paths for optimization and in particular regarding memory access. Using the *Perf Linux* tool, we have collected data from several performance counters. By analyzing counters statistics, we identify a single function that generates 71% of the cache misses as shown in Figure 4(a). This *memset_sse2* function is used to initialize memory blocks to zero. Fortunately in the present case, we found that *memset_sse2* was in fact called by a *calloc* function call that allocates the memory for structures of photon bunches. After having verified that such initialization was not necessary in this case, a simple *malloc* function drastically reduced the number of cache misses as shown in Figure 4(b). The impact on overall performances was also significant, bringing the overall speed-up to 1.82 after changing, as shown in Figure 2.



(a) CORSIKA 7.6900-OPT-V1 - By function : 1.*memset_sse2*, 2.rest

(b) CORSIKA 7.6900-ORG/OPT-V1/OPT-V2

Figure 4: Cache-misses analysis

3 CORSIKA 8 - Cherenkov Module Development

CORSIKA 8 is a collaborative project for the development of a modern C++ version of CORSIKA[1] using different modern tools for version control, testing and continuous integration. In the CORSIKA 8 framework it is already possible to simulate hadronic and electromagnetic showers. The support for electromagnetic cascades has been recently introduced via an interface with the PROPOSAL software.[2] At the moment only some analytical atmospheric models are supported, but in the future more precise models, e.g., atmospheric tables as in CORSIKA 7, will be included. In the rest of the paper we present our contribution to the development of a Cherenkov module in the CORSIKA 8 framework to handle the generation and the propagation of the Cherenkov photons through the atmosphere.

Measurements reported in this section have been done on a laptop configuration[3]. For the first development phase of CORSIKA 8, we considered that the compiler optimizations should be disabled. But in this phase of tests on the Cherenkov module presented below, we analyze data compiled with the optimizations enabled.

3.1 Cherenkov module - Overview

We follow three main steps to develop a Cherenkov module in the CORSIKA 8 framework with compatible physical results and performances with respect to the current CORSIKA 7. The first step consists of writing a version very close to the CORSIKA 7 Cherenkov module, to provide a good basis for comparison. Then, we develop a version in the general style of CORSIKA 8, i.e. using different C++ 17 concepts and removing all pointers.

Finally, we explore different possibilities of optimization, testing different vector libraries for elementary functions or applying single precision to certain parts of the code, thanks to a numerical error analysis or using vectorized operations with templates. Indeed, instead of writing a loop with operations that could be not vectorized, we could create an operator which performs automatically some vectorial operations .

The first step is currently almost achieved. The CherenkovC7Test module (cf. Table 1) generates and propagates Cherenkov photons to the ground using the same algorithms as in CORSIKA 7 except one missing function that tests photons crossing the telescopes.

[1] https://gitlab.ikp.kit.edu/AirShowerPhysics/corsika
[2] https://github.com/tudo-astroparticlephysics/PROPOSAL
[3] Intel i7-8665U/4.8Ghz/L3:8MB, RAM:16GB with gcc/g++/gfortran 9.3 and optimization flags: -O3 -mavx2

In order to do an accurate comparison between this new CherenkovC7Test version and CORSIKA 7, we have slightly modified the latter to save particle tracks coordinates which are used as entries by the Cherenkov function. Then, we inject these entries in the CherenkovC7Test module and we compare the results in terms of photon coordinates after the propagation to the ground.

In Tables 2 and 3 we explain how the module is organized.

Table 2: CherenkovC7Test Module in CORSIKA 8 - Simulation

Steps in Simulation	Main methods
1. Read and init tabulated atmosphere model	*loadAtmosphereProfile()*
2. Loop: Read particle data created in CORSIKA7	*initParticleValues()*
3. Loop: Generates and propagates photons	*generatePhotons()*,*propagatePhotons()*
4. Loop: Save data	*savePhotonsInformationBeforeInpact()*

Table 3: CherenkovC7Test Module in CORSIKA 8 - Different classes

Class	Description	Dependencies
CherenkovC7Test	generate and propagate photons	AtmosphereTabulated
AtmosphereTabulated	generate atmosphere and interaction	Interpolation
Interpolation	linear and cubic spline interpolations	

It was necessary to add interpolation because C functions in CORSIKA 7 used only double precision type. In fact, CORSIKA 8 uses macro custom types for physical values (energies, lengths, etc) to keep safe physics dimensions with zero runtime cost dimension checking. Interface code was added to transform these unit types consistently between CORSIKA 8 and CORSIKA 7 code. Thanks to this new rewriting with templates, methods of the class Interpolation can take any type in input/output.

3.2 Cherenkov module - Performances

In order to accurately estimate the performances related to specific parts of the code, we have used the *PAPI* API[6] to collect hardware performance counters in some selected areas for CORSIKA 7 and CORSIKA 8. Indeed contrary to *Linux Perf*, *PAPI* allows us to instrument the code to isolate the areas where it sums the collected events for one or multiple counters. So thanks to *PAPI*, we can easily isolate the different parts of the code that we want to compare between CORSIKA 7 and 8, where the selected counters come from the *Linux Perf* library.

We compare for some selected hardware counters the Cherenkov module in CORSIKA 7 and CORSIKA 8 (CherenkovC7Test) with and without the *vector-libm*. We recall that CORSIKA 7 always uses here the *vector-libm*.

Table 4: General hardware counter ratios (C8 Cherenkov module / C7 Cherenkov module)

Version	Instructions	Cycles	Wall clock time
C8 / C7	1.26	1.67	1.48
C8+vlibm / C7	1.02	1.57	1.39

The results reported in Table 4 exhibit that the Cherenkov module in CORSIKA 8 is always slower than in CORSIKA 7. Nevertheless the wall clock time difference is reduced with the *vector-libm*. Indeed adding the *vector-libm* in CORSIKA 8 removes 20% of instructions and reduces the number of cycles. Nevertheless, it seems that the *vector-libm* library is bottle-necked as explained in the next paragraphs.

Table 5: Distribution of double precision floating-point instructions in Mi (mega-instructions)

Version	Scalar	Vector 256bits	Vectorization Ratio
C7	108	53	66.4%
C8	236	10	14.6%
C8+vlibm	219	29	34.8%

In Table 5 we report the number of scalar and vector instructions as well as the vectorization ratio. In CORSIKA 8 the number of scalar instructions increases by about a factor 2 while the number of vector instructions decreases by about 45% despite the *vector-libm* being enabled. Here are some explanations for such reduction of vectorized instructions. To add the *vector-libm* in CORSIKA 8, we had to remove some memory alignment functions in the vectorized functions, as C++ does not know them (old C functions). Moreover the "constraint" clause (mainly used for function parameters) is not used in C++ whereas it is in C. For the scalar instructions, the code in CORSIKA 8 is slightly different : we add some operations to convert double to "custom types", for some comparisons, isolated calculations (coefficients,etc) and mathematical functions (absolute value function, pow function,etc). Two multiplication are necessary for each function (custom type to double and double to custom type). These technical differences may explain the observed reduction of the vectorized ratio. This scalar part is the previously mentionned bottleneck for the *vector-libm* efficiency.

Table 6: Branch and cache miss ratio ratio

Version	BM/B[1]	CM/CR[1]	LLCM/LLC[1]
C7	3.0%	6.0%	9.8%
C8	46.8%	33.7%	41.0%
C8+vlibm	3.3%	32.7%	41.8%

We have then measured the events for different counters giving indications to the memory access pattern, as reported in Table 6.[1] First, we observe that in CORSIKA 8, the ratio of cache misses to cache references increases from 6% to about 33%. Secondly, branch misses ratio[2] is stable between CORSIKA 7 and 8, except for CORSIKA 8 without *vector-libm*, where we observe an increase. In fact, the use of *vector-libm* has a branch cost because we need to test if we can use *vector-libm*'s functions or not during the simulation. It generates more branches but it is not proportional with branch misses. It explains the important 46.8% ratio of branch misses to branches without the *vector-libm*.

LLC-loads ratio is interesting because it increases by more than 41% in comparison to 9.8% measured in CORSIKA 7. L3 cache misses is bigger than for other cache levels so it reveals that CORSIKA 8 uses more data than CORSIKA 7. Even if the implemented algorithms are the same, CORSIKA 8's classes with methods and attributes increase cache

[1] BM: branch misses, B: branch, CM: cache misses, CR: cache reference, LLCM: last level cache misses, LLC: last level cache load

[2] number of miss-predicted branches / number of executed branches

misses. Attributes enable to call functions with sending argument but contrary to C code, we have inevitably more data in memory.

Table 7: Performance events and percentage[1] for C7 and C8+vlibm

Step	Minstructions	Mcycles	KcacheMisses	KLLCMisses	Vector Ratio
Generate Photons C7	133 (12.3%)	107 (16.6%)	22 (17.0%)	4 (39.8%)	0.0%
Generate Photons C8	192 (17.6%)	200 (19.8%)	5912 (33.1%)	421 (46.8%)	0.0%
Before Propagation C7	444 (41.3%)	246 (38.1%)	7 (5.1%)	0 (3.0%)	52.3%
Before Propagation C8	343 (31.4%)	305 (30.2%)	4809 (26.9%)	399 (21.3%)	52.5%
Propagation C7	499 (46.4%)	292 (45.3%)	99 (77.9%)	5 (57.2%)	75.5%
Propagation C8	557 (51.0%)	505 (50.0%)	7142 (40.0%)	484 (31.9%)	28.7%

Finally, we have divided the Cherenkov module in 3 areas (as shown in Table 7) and compared different counters area by area. In general, we have a huge increase for cache misses and LLC misses for CORSIKA 8 but we notice some particularities for every area.

Like in CORSIKA 7, the photon generation step does not produce vectorized instructions and we observe a moderate increase of instructions and cycles. Before photon's propagation (some calculations to prepare it), we found a vector ratio slightly better for CORSIKA 8 version. In fact there are more vectorial operations and a general instruction decrease in this part thanks to a loop fusion between scalar and vectorized loops versions of CORSIKA 7. C version is faster but very close in cycles with C++ version.

Propagation function is a more specific case where vectorization does not work very well. Indeed, CORSIKA 8 vector ratio is worse than CORSIKA 7 in this case and we need to investigate further. This cycle increase is not negligible. We call a lot of getter methods from Atmosphere class to get some values that can generate additional cycles. We can also imagine a bug in the algorithm with *vector-libm* which does not work in this method and use mainly the *libm* for elementary functions (two possible ways in algorithm).

For the moment, there was not optimization work and this is clearly seen in performance results. *Vector-libm* improves general performance with an insufficient speedup of 1.06. This work is a basis to do something better in performance, readability and precision. Several optimization options were evoked like vectorial calculations in single precision, new vectorial operators and new C++ style in general.

3.3 Module Cherenkov - Physics results

Table 8: Pearson and Kolmogorov indices between CORSIKA 7/8 for the total number of photons, positions in X,Y,Z, directions in X,Y and arrival times by sub-step

	NbPhotons	PosZ	PosX	PosY	DirX	DirY	Time
Pearson	1.000	1.000	0.988	0.996	0.982	0.986	0.991
Kolmogorov	1.000	1.000	0.995	0.935	0.963	0.939	0.972

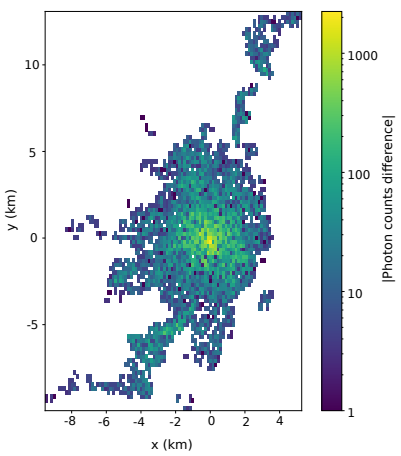
In order to validate the physical results obtained with the new Cherenkov module in CORSIKA 8 compared to CORSIKA 7, we have set up a test environment to run and compare the two codes under very similar conditions. First of all, it is important to use the same set of entries (i.e., particle track coordinates) for the two versions. Secondly, in this first phase, we

[1] Number of events / Overall number of events

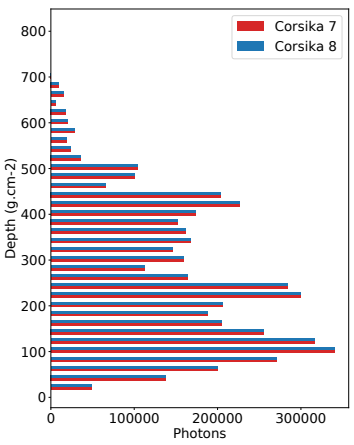
have implemented essentially the same algorithms as those used in CORSIKA 7. The main difference between the two codes concerns the random number generator, that is used to generate the ϕ angular coordinate of the emitted Cherenkov photons for a given opening angle θ. CORSIKA 7 uses a modified version of the CERN random number generator in double precision (*RMM48*), while CORSIKA 8 currently uses the uniform real distribution in double precision of the "random" library.

The output of the Cherenkov module consists of the list of photon bunches reaching the ground, with their positions, directions and arrival time. We have thus run the Cherenkov module on a set of 10 gamma-ray showers and compared the different output distributions.

As shown in Figure 5(a), the 2D map of the photon counts on the ground is very similar between CORSIKA 7 and CORSIKA 8 while the photon distribution by atmospheric depth is exactly the same in Figure 5(b). A statistical comparison of the distributions of the physical quantities confirms a very good agreement between the two versions, as indicated by the Pearson and Kolmogorov coefficients in Table 8. We have also verified that the slight differences in position and direction distributions are due to the different random generator used in the two versions. With this validated version in hands, we are now ready to start the second and third phases of our plan, i.e., writing the module in a more C++ friendly style in line with the CORSIKA 8 framework (e.g., using templates, static polymorphism, etc.) and applying different types of optimizations, also benefiting from the language capabilities.



(a) 2D map of the photon counts difference on the ground



(b) Photon distribution by atmospheric depth for CORSIKA 7 (in red) and CORSIKA 8 (in blue)

Figure 5: Comparison between CORSIKA 7 and CORSIKA 8 after propagation in atmosphere with gamma incident particles - 10 CORSIKA 7 showers with 4.9x10^6 Cherenkov photons produced

4 Conclusion

Starting with a detailed profiling of CORSIKA (version 7), we have been able to apply different kinds of optimizations while keeping a portable code. Starting from a vectorized version developed in a previous work, we have been able to further optimize CORSIKA by improving

memory access and enhancing vectorization. The overall obtained speed-up after this optimization work is 1.86 in elapsed time and 2.05 in number of cycles. All these optimizations were done without changing the logic of the algorithms and in such a way that the obtained physical results were identical to the original version.

Further CORSIKA optimization would require profound changes in algorithms or in data structures. We have thus chosen to contribute to the on-going CORSIKA 8 project for the full rewriting of CORSIKA in C++. Our contribution focuses on the Cherenkov module, which is of particular interest for the CTA community and which also appears to be the bottleneck regarding run time in CTA simulations. The work presented in this paper is the first phase toward the development of an optimized Cherenkov module in CORSIKA 8 framework.

Our goal was to first develop a version very similar to CORSIKA 7, from the algorithmic point of view, to have a good basis for comparison. We have thus performed a detailed profiling of the new and of the old code and compared several hardware counters. The overall performance of the new version is already very close to CORSIKA 7, even before any specific optimization work. Nevertheless, an excess of cache misses in the new version indicates that there is some room for improvement regarding memory access.

Then, we have set up a test environment that allowed us to compare the distributions of different physical output quantities of the Cherenkov module between the two codes. We have found that the agreement between the different distribution is excellent.

Starting from this version, the next step will consist in writing a new one in line with the CORSIKA 8 framework C++ style (e.g., using templates, static polymorphism, etc.). Then, we will investigate different optimization strategies, like using optimized libraries for different mathematical computations or developing our own vectorized operators thanks to C++ features, that could also benefit to other modules. Finally, we think that a promising strategy would be to reduce the precision of some computations from double to single in specific parts of the code, so to take even more benefit from vectorization. In order to study the numerical errors associated to the reduction of the precision, we plan to use the Shaman library[7] or equivalent tools.

References

[1] L.Arrabito, K.Bernlöhr, J.Bregeon, M.Carrère, A.Khattabi, P.Langlois, D.Parello G.Revy , *Optimizing Cherenkov Photons Generation and Propagation in CORSIKA for CTA Monte–Carlo Simulations*, Computing and Software for Big Science, **4**, 9 (2020)

[2] D.Heck et al., *CORSIKA: A Monte Carlo Code to Simulate Extensive Air Showers*, Forschungszentrum Karlsruhe Report FZKA 6019, 90 pages, 10 figures (1998)

[3] The CTA Consortium, *Science with the Cherenkov Telescope Array*, WORLD SCIENTIFIC, ISBN 9789813270091, http://dx.doi.org/10.1142/10986 (2018)

[4] C.Lauter , *A new open-source SIMD vector libm fully implemented with high-level scalar C*, 2016 50th Asilomar Conference on Signals, Systems and Computers, pp. 407-411 (2016)

[5] R.Engel et al., *Towards a Next Generation of CORSIKA : A Framework for the Simulation of Particle Cascades in Astroparticle Physics*, Comput. Softw. Big Sci., **43**, 2 (2019)

[6] S.Browne et al., *A Portable Programming Interface for Performance Evaluation on Modern Processors*, The International Journal of High Performance Computing Applications, **14**, 189-204 (2000)

[7] N.Demeure, *Gestion du compromis entre la performance et la précision de code de calcul*, HAL https://tel.archives-ouvertes.fr/tel-03116750, 169 (2021)