Summary of Julia projects in CERN EP R&D and Next Generation Triggers

Mateusz Fila*[1]

[1]CERN, EP-ADP-OS, Switzerland

November 2025

Abstract

This document reports on a series of R&D projects at CERN EP R&D and Next Generation Triggers that explored the use of Julia for selected high-energy physics applications. The projects include: (1) ahead-of-time compilation of JetReconstruction.jl, a Julia implementation of FastJet algorithms; (2) development of a scheduling demonstrator for event-processing applications frameworks; and (3) porting the CMS Patatrack pixel track reconstruction to Julia.

While Julia offers attractive ergonomics and a rich package ecosystem, these studies revealed multiple limitations. Ahead-of-time compilation remains incomplete, and residual just-in-time compilation may persist in produced binaries, limiting their suitability for latency-sensitive applications. Trimming binaries to reduce size is currently difficult, so generated binaries are very large and could further increase the footprint of software stacks if adopted at larger scale. Multi-threaded performance is significantly constrained by Julia's stop-the-world garbage collector, impacting applications such as event reconstruction. Additionally, several widely used packages have exhibited serious maintenance issues, raising concerns about ecosystem sustainability. The community's strong enthusiasm and active promotion of the language which, while not uncommon for programming language communities, can make it difficult to obtain a clear picture of its actual maturity, highlighting the need for a critical assessment. Julia can be a useful tool for specific tasks, but caution is warranted when considering it for latency-sensitive, high-parallel, high-throughput applications or long-term HEP projects.

1 Introduction

High-energy physics (HEP) software development faces significant challenges. Typical experimental frameworks comprise several million lines of code with combined library sizes reaching tens of gigabytes. While C++ provides the required performance, scalability, and low-level control, its complexity hinders productivity and long-term maintainability, motivating interest in simpler yet equally performant alternatives. Although HEP applications often run for hours, suggesting just-in-time (JIT) compilation overhead might be acceptable, reproducibility remains critical, in some cases execution latency is also a major concern. These factors, combined with the need for collaborative development across large teams with varying levels of software expertise, create a demanding set of requirements for any programming language in HEP.

Julia [1] is a general purpose programming language that has attracted significant attention in scientific computing due to its combination of ease of use, high performance, and support for many modern computing paradigms. Earlier studies [2, 3, 4] highlighted Julia's potential as a future language for HEP, particularly for data analysis and simulation tasks.

This document summarizes the R&D projects conducted within CERN EP R&D and Next Generation Triggers, aimed at exploring Julia's potential in HEP applications that were not addressed in previous studies. Specifically, the projects assessed its suitability for the integration of standalone Julia packages into existing HEP C++-based codebases, scheduling in event-processing frameworks, and the implementation of trigger and reconstruction applications.

*E-mail: mateusz.jakub.fila@cern.ch

2 Ahead-of-time compilation of JetReconstruction.jl

JetReconstruction.jl [5] is a native Julia reimplementation of the FastJet [6] jet clustering algorithms. The package is openly developed on GitHub [7] and already attracting community contributions [8]. It achieves single-threaded performance matching or exceeding FastJet (Figure 1) while being written in idiomatic Julia, requiring minimal manual optimization. Only one loop is explicitly optimized using LoopVectorization.jl [9] macros for vectorization and unrolling.

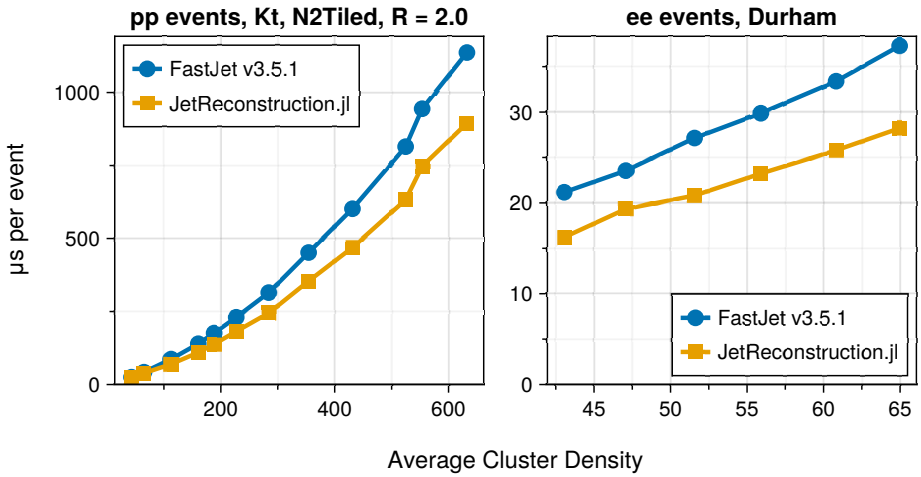

Figure 1: Comparison of jet reconstruction time for different cluster densities using FastJet 3.5.1 and JetReconstruction.jl. Left: pp kT-algorithm with the N2Tiled strategy was used. Right: e^+e^- Durham algorithm was used. JetReconstruction.jl consistently outperforms FastJet. Measurements were performed on an AMD Ryzen 7 5700G running AlmaLinux 9. Each measurement was repeated 16 times; the minimum time was reported.

Although JetReconstruction.jl outperforms FastJet, integration with non-Julia software without compromising the performance is non-trivial. Its current use case is end-user analysis in Julia for events already processed by an experimental framework, but C-bindings could enable its use in latency-sensitive HEP applications for instance usage at trigger farms. This project explored development and ahead-of-time (AOT) compilation of such bindings as a proof of concept for integrating Julia algorithms into C++-based frameworks.

2.1 LoopVectorization.jl

While LoopVectorization.jl provides significant performance benefits and has proven very effective for our needs, its long-term maintenance remains a concern. The package previously experienced periods of low maintenance, and its compatibility with Julia v1.12 was uncertain. A new version compatible with Julia v1.12 has since been released; however, ongoing development currently depends on limited grant-based support. One of its direct dependencies, VectorizationBase.jl [10], continues to be marked as seeking a new maintainer or deprecated for Julia v1.11 and beyond.

Alternatives, including hand-guided SIMD implementations, were evaluated but did not achieve comparable performance across platforms.

2.2 Compilation

Experimental C-bindings for pp algorithms were written and then compiled using PackageCompiler.jl [11] and juliac [12] (Julia v1.12). The PackageCompiler.jl executes custom script, then saves the compilation data from that execution, while juliac statically analyzes code. Both approaches are capable of producing shared libraries or executables, but neither guarantees a binary free of just-in-time (JIT) compilation at runtime. A comparison of compilation times and output sizes is presented in Table 1. Compilation with juliac is noticeably faster than with PackageCompiler.jl, though still significantly slower than compiling FastJet with a C++ compiler. Both Julia-based approaches produce significantly larger binaries than FastJet. Additionally, PackageCompiler.jl was broken in newer Julia v1.12.0 pre-releases, an issue that was also reported by other users.

Table 1: Compilation times and binary sizes for JetReconstruction.jl and FastJet 3.5.1. JetReconstruction.jl was compiled using juliac (Julia v.1.12.0-rc1) and PackageCompiler.jl (v.1.12.0-beta2). Both compilers support only single-threaded compilation and produce substantially larger binaries and longer compilation times compared to FastJet. The compilation timings were measured on AMD Ryzen 7 5700G system running AlmaLinux 9.

Code	Compiler	Compilation time	Binary size
JetReconstruction.jl	juliac	~ 120 s	~ 300 MB
	PackageCompiler.jl	~ 500 s	~ 300 MB
FastJet 3.5.1	gcc 14.2	~ 40 s	8 MB
		~ 20 s (parallel)	

2.2.1 Trimming

A notable advantage of juliac over PackageCompiler.jl is its experimental trimming feature, which reduces binary size by including only code paths that are statically proven to be reachable. However, trimming currently works only for a restricted subset of Julia, with the exact language constraints undocumented.

JetReconstruction.jl cannot be compiled in trimming mode, as it relies on unsupported features such as exception handling (the primary error-handling mechanism in Julia), standard logging, and CPU feature detection (required by LoopVectorization.jl). The additional static analysis performed during trimming revealed a few minor type instabilities that were subsequently fixed. Despite efforts to interpret the error messages and attempt fixes, the exact causes of some of the compilation errors remain unresolved. These findings suggest that trimmability should be considered from the start, as retrofitting it into existing packages can be challenging. While still experimental, trimming is expected to mature in future Julia releases, potentially lifting some of the current limitations, particularly for exceptions and logging.

2.3 Performance

As the first observation, PackageCompiler.jl was found incompatible with LoopVectorization.jl, producing slower binaries when vectorization was enabled. Without it, binaries from both compilers showed equivalent performance.

Single-threaded benchmarks comparing FastJet, native JetReconstruction.jl, and juliac-compiled C-bindings (Figure 2) showed minimal overhead from the C interface. The C-bindings incur a small overhead compared to the native Julia version, although in practice this overhead may be negligible depending on the parameters of the reconstruction algorithm. The exact origin of this overhead is not completely understood yet as many Julia-specific tools are not compatible with compiled libraries. The most likely sources are data conversions, as the Julia data types do not always map well to C-types. Input data structures have already been reworked to minimize copying, but some unavoidable conversions remain. Another issue concerns returning memory from Julia to C: since Julia uses a garbage collector (GC), memory must be copied to avoid invalidation. Allocating memory directly in C and filling it in Julia would avoid this but also would require intrusive changes to the package. Overall, while the current approach introduces some conversion and copying overhead, it does not significantly degrade performance; nevertheless, code designed from the outset with C-interoperation in mind could potentially avoid it entirely.

In repeated trials, the performance of binaries compiled with juliac was consistent, but JIT-compilation is not fully eliminated. Each trial consisted of 100 iterations, with each iteration reconstructing the same sequence of 100 events. The first iteration, during which JIT-compilation occurs, is more than six times slower than subsequent iterations. Distributions of execution times across repeated trials are shown in Figure 3. In practice, requiring the code to be *trimmable* could provide a practical means to ensure JIT-compilation free execution, since trimming mandates that the code be fully statically analyzable with no runtime type inference. However, this could not be tested because JetReconstruction.jl is not currently trimmable (Section 2.2.1). In contrast, no JIT-compilation was observed in binaries produced with PackageCompiler.jl. This approach is, however, fragile, as it relies on a precompilation script that must be carefully prepared to cover all relevant code-paths.

AOT-compiled C-bindings cannot yet be considered production-ready. While performance in steady-state execution exceeds the performance of its C++ counterpart, the remaining JIT-compilation represents a significant limitation for low-latency use-cases such as the offline trigger environment which was the original target application. At present, there is little that can be done to address this issue, since it largely depends on the future evolution of AOT-compilation in Julia. JIT-compilation remains an integral part of Julia's design, closely tied to its dynamic type system and method dispatch, whereas AOT-compilation is a recent

and secondary mechanism supporting only a subset of the language.

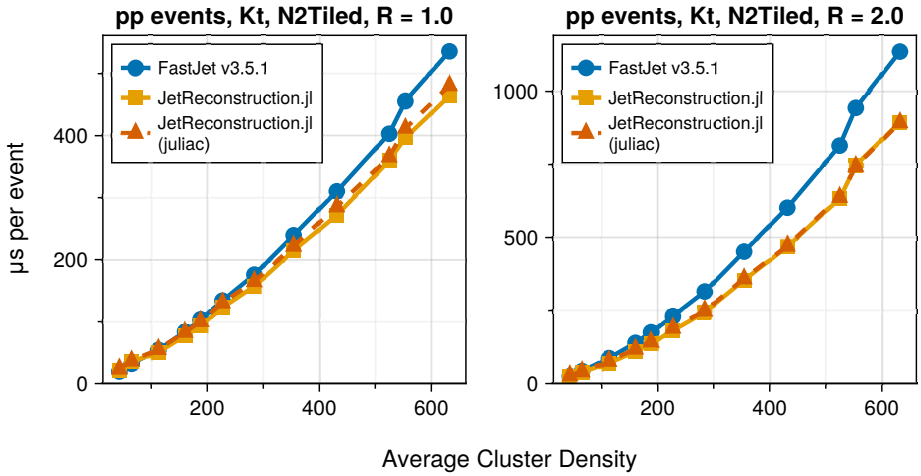

Figure 2: Comparison of jet reconstruction time for different cluster densities with FastJet 3.5.1, JetReconstruction.jl and JetReconstruction.jl C-bindings AOT-compiled with juliac. Single-threaded pp kT-algorithm and N2Tiled strategy at different jet radius values are used. JetReconstruction.jl exhibits comparable or better performance than FastJet. The overhead of juliac compiled C-bindings is minimal with respect to native JetReconstruction.jl. Each measurement was repeated 16 times, and the minimum time was reported. Measurements were done on AMD Ryzen 7 5700G system running AlmaLinux 9.

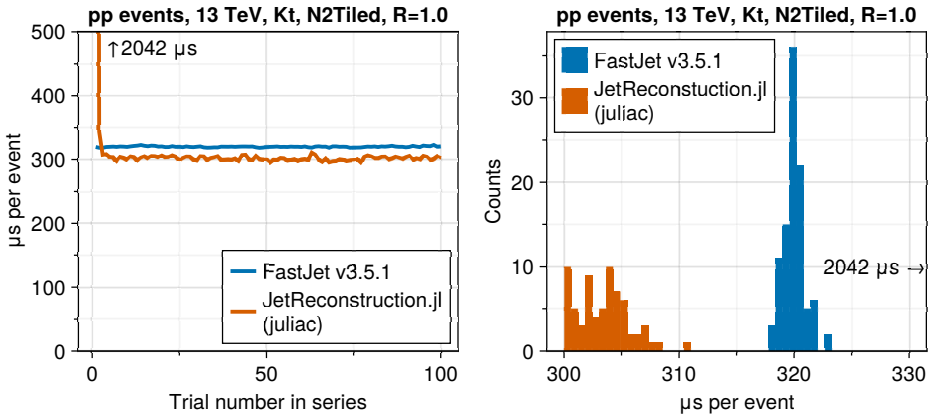

Figure 3: Distribution of jet reconstruction time in a series of 100 trials for pp kT-algorithm with N2Tiled strategy. FastJet 3.5.1 and JetReconstruction.jl C-bindings compiled with juliac are compared. Each trial reconstructs the same sequence of 100 events within a single process. Left: execution time for each trial, showing JetReconstruction.jl C-bindings are substantially slower in the first execution. Right: Histograms of execution times, the first execution for JetReconstruction.jl C-bindings is outside of range. The first execution of JetReconstruction.jl C-bindings is significantly slower than subsequent executions due to residual JIT-compilation. The subsequent executions show slightly more variability but remain consistently faster than FastJet. Measurements were done on an AMD Ryzen 7 5700G system running AlmaLinux 9.

3 Demonstrator for scheduling in event-processing frameworks

Julia is often praised for its strong support for parallel and heterogeneous computing. Motivated by these claims, this study investigated Julia's suitability for implementing an event-processing application framework such as Gaudi [13] or CMSSW [14]. Particular emphasis was placed on the scheduler, the component responsible for orchestrating event processing, with the objective of enabling multi-threading, GPU offloading, and distributed execution across nodes. To avoid the complexity of porting existing applications, the methodology established by GaudiHive [15] was adopted, in which CPU-crunching mockup

algorithms are used to emulate representative workflows. Among others, ATLAS Athena q449 standard test reconstruction workflow was extracted in a form of data-flow graph and timings using Gaudi tools. The demonstrator FrameworkDemo.jl [16] was then developed to reproduce the extracted workflow and perform scheduling based on the recorded measurements.

3.1 Julia graphs

The extracted data-flow graphs were stored in GraphML [17], a widely used format for graph-information exchange. However, the JuliaGraphs [18] ecosystem offers only rudimentary GraphML support, not allowing the reading of properties which is an essential feature of the format. More generally, graph representations have several use-cases in HEP software, including event reconstruction data models, event generator process modeling, and dependency management in event-processing frameworks, making reliable graph packages valuable for the community. Beyond GraphML serialization support, some general utilities of interest, such as longest path algorithm, were not available in JuliaGraphs when this project began. In addition, some JuliaGraphs packages, such as GraphViz.jl, have been broken since Julia v1.11, and response times for issues or pull requests often span several months. The adoption of JuliaGraphs is not ubiquitous, and packages such as the QuantumElectrodynamics.jl ecosystem [19] have instead developed their own graph implementations.

3.2 Dagger.jl

Initially, the scheduler development was designed around Dagger.jl [20], a framework for parallel computing across threads, GPUs and nodes, endorsed on the official Julia website [21]. The implementation plan was to begin with multi-threading, then extend to distributed and multi-process execution, and finally incorporate GPU support — all leveraging Dagger.jl.

3.2.1 Multi-threading

At the time of implementation, Dagger.jl lacked support for parallel-pipeline execution, requiring the task-graph to be rebuilt for each event. The scheduler was implemented using the "task-dependencies" API and benchmarked on the ATLAS Athena q449 workflow mockup. Throughput scaling was compared with a C++ demonstrator using Taskflow [22] (see Figure 4). The Dagger-based version showed poor scaling and became unstable above 16 threads. After debugging and discussions with the Dagger.jl developers, the instability was attributed to internal data-races inside Dagger.jl. Despite these discussions, no practical way to improve performance was identified. The developers also noted that Dagger's internal memory management may be more optimized for distributed execution. A new Dagger.jl API introducing "streaming tasks" was later added to support persistent task-graphs; however it is not expected to resolve the underlying performance and stability issues observed in this study.

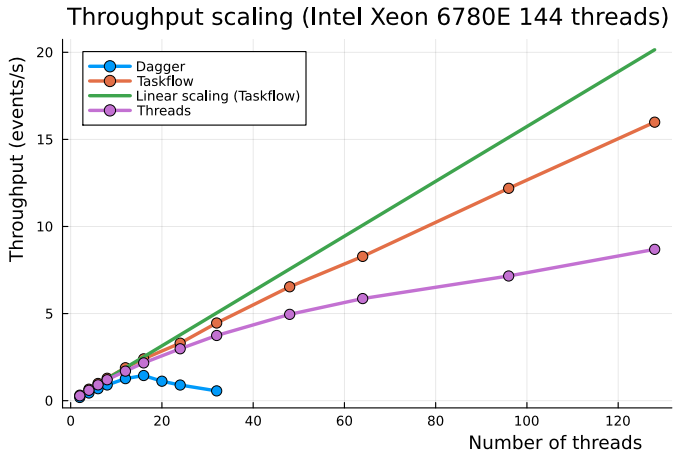

Figure 4: Multi-threaded scaling behavior of Julia (Dagger.jl, Threads) and C++ (Taskflow) demonstrators. All demonstrators mock the data-flow graph of Athena standard test reconstruction using CPU-cruncher algorithms with 20 total events per thread and 0.5 concurrent events per thread. The measured run is preceded with warm-up run with the same settings. The CPU-cruncher algorithms are written such that Julia does not allocate any memory within them. Measurements were done on a Intel Xeon 6780E system running AlmaLinux 9.

3.2.2 Distributed computing

A preliminary evaluation was also conducted on Dagger.jl's distributed computing capabilities, which are built on Julia's standard library Distributed.jl. Running multiple threads in remote processes revealed persistent thread-safety issues originating from Distributed.jl. According to discussions with the Dagger.jl developers, these issues were reported upstream, but due to long review process, they forked the library into DistributedNext.jl [23]. Following their recommendation, the demonstrator was migrated to DistributedNext.jl, which did not fully resolve the observed problems. Since then, several thread-safety improvements from DistributedNext.jl have been merged back into the standard Distributed.jl library, and maintenance activity has improved with new maintainers joining the effort. Nonetheless, these developments highlight persistent concerns regarding the long-term reliability and maintainability even for Julia's standard libraries.

Julia also offers alternative packages for distributed computing that may be more robust, such as MPI.jl [24], which provides Julia bindings to established standards rather than relying on a custom Julia-based design. While Dagger.jl continues to be promoted and endorsed as the primary framework for bridging distributed and multi-threaded computing in Julia, the Julia HPC community appears to generally favor MPI.jl-based solutions.

3.2.3 Profiling

Dagger.jl's built-in tracing initially caused up to 700% overhead, later reduced to 40% by optimizations suggested by the Dagger developers. NVTX.jl [25] was therefore adopted for low-overhead profiling (<1%), capturing also Julia-specific activity such as JIT-compilation and garbage collection. The built-in Julia sampling profiler was tested but proved limited, as suspended tasks were not captured. Julia v1.12 introduces a walltime profiler addressing this issue. PProf.jl [26], a Julia wrapper for Google's pprof endorsed in official Julia documentation [27], was also evaluated but it was broken since Julia v1.10 with no response from maintainers [28].

3.3 Standard multi-threading

Due to persistent performance and stability problems, Dagger.jl was abandoned by the scheduling project. The focus shifted from heterogeneous computing to assessing native multi-threading performance using Julia's built-in Threads module. OhMyThreads.jl [29], a performance oriented alternative emphasizing data parallelism, was also tested. Benchmarks using allocation-free CPU-crunchers showed similar scaling for both libraries, with only marginal advantages for OhMyThreads.jl. Both, however, remained notably slower than the C++ Taskflow demonstrator (Figure 4), despite considerably more time and effort devoted to profiling and optimizing the overheads of scheduling mechanism on the Julia side.

3.3.1 Memory allocations

Subsequent experiments introduced memory allocations inside the CPU-crunching algorithms. Example results with allocations enabled are presented in Figure 5. Both single-block and multiple small allocations were tested, but in all cases, performance dropped significantly and garbage collection time increased with the number of threads. This behavior points to a potentially fundamental limitation for adopting Julia in high-performance, multi-threaded HEP applications. While Julia's guidelines advise minimizing allocations — a principle that applies to any programming language — allocations might not always be avoidable in realistic HEP applications. Strategies such as pre-allocating memory are limited in Julia because only arrays can adopt such buffers, and there is no equivalent of C++'s placement new for other types that allocate memory internally. Furthermore, the memory storage of some Julia types is considered an implementation detail, which restricts low-level control over memory placement.

4 Porting CMS Patatrack to Julia

The Patatrack pixel track reconstruction [30] is a stand-alone project extracted from the CMS reconstruction software. It has been used to evaluate a wide range of CPU and GPU programming models [31], making it an ideal candidate for a Julia port to test the language in a realistic HEP environment.

The goal of this project was to assess Julia's feasibility for large-scale HEP applications by porting the CMS pixel-only track reconstruction pipeline from C++ to Julia. The work was carried out by students with no prior Julia experience, providing insight into Julia's accessibility for non-experts. The port closely mirrored the original C++ constructions rather than being fully redesigned in idiomatic Julia, preserving the structure and logic of the existing implementation. The study primarily evaluated single-threaded and multi-threaded CPU performance, while GPU support is still under development.

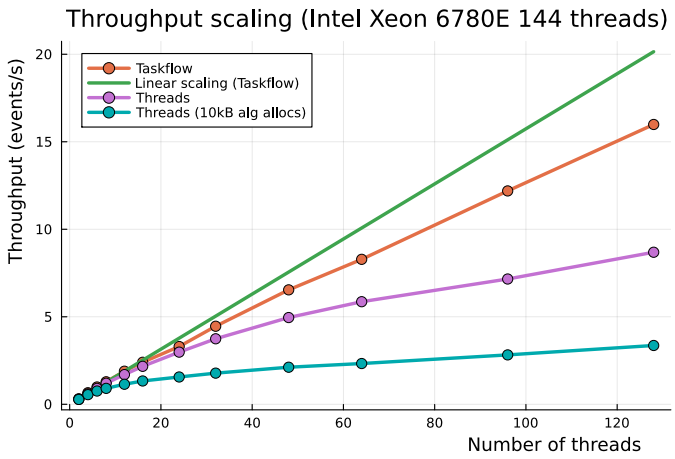

Figure 5: Effect of memory allocations on demonstrators' event throughput. The demonstrators mock the Athena standard test reconstruction using the same configuration as in Figure 4, with an additional series where the Julia demonstrator using Threads performs a single 10 kB memory allocation during execution of each algorithm. Adding allocations noticeably degrades the performance of the Julia Threads demonstrator due to increased garbage collection time.

The serial CPU port was completed, including all modules and framework components, with validated results. An initial attempt using Dagger.jl for scheduling confirmed the poor performance observed in Section 3.2.1, leading to its replacement with Julia's built-in Threads module.

4.1 Performance

The performance of the Julia port of the pixel track serial algorithm shows mixed behavior across modules. Some modules, such as clustering and rechit building, execute faster in Julia than in C++, while others, particularly ntuplet construction, are significantly slower. Despite extensive optimization efforts, the slower modules continue to dominate the total runtime, resulting in overall execution times higher than the C++ implementation (Figure 6).

All performance measurements were preceded by a warm-up execution to mitigate JIT-compilation overhead and ensure more stable timing results. Even after warm-up, a small residual fraction of time (<1%) was still spent in JIT-compilation. During the project, memory allocation was identified as a major source of overhead and mitigated where possible by using immutable structs and stack-allocated fixed-size arrays instead of Julia's default dynamic arrays.

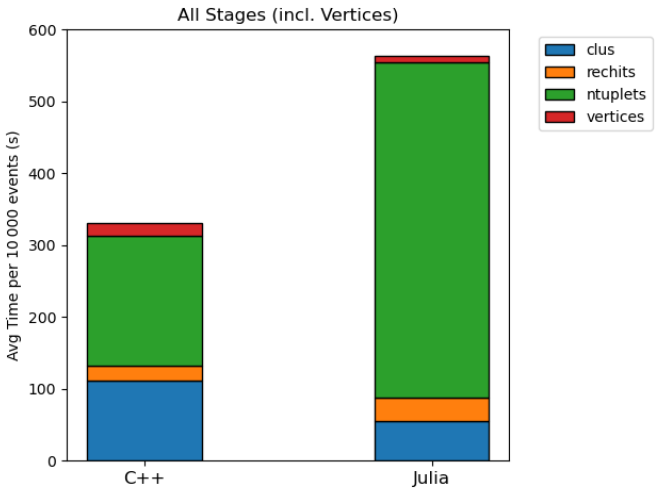

Figure 6: Comparison of average execution time per 10,000 events between the C++ pixel track serial implementation and its Julia port. The timing is broken down by module (clus, rechits, ntuplets, vertices). While some modules are faster in Julia, the ntuplets stage dominates the runtime, leading to a significantly higher total execution time compared to C++.

Throughput scaling with multiple threads is considerably worse than in C++ (see Figure 7). This is primarily due to an increasing fraction of time spent on garbage collection, as measured by Julia's profiling tools. Julia's stop-the-world garbage collector pauses all threads for collection, and the effect increases with thread count. This observation appears to be consistent with problems encountered independently in Section 3.3.1. The Julia documentation [27] recommends several approaches to reduce garbage collection overhead, including lowering the allocation rate and tuning garbage collector parameters, which were explored in this project. Adjusting the number of marking threads yielded no measurable benefit, while enabling concurrent garbage collection sweep provided the most significant improvement by reducing pause times at high thread counts. Nevertheless, the overall throughput and scaling of Julia remain far below the C++ implementation.

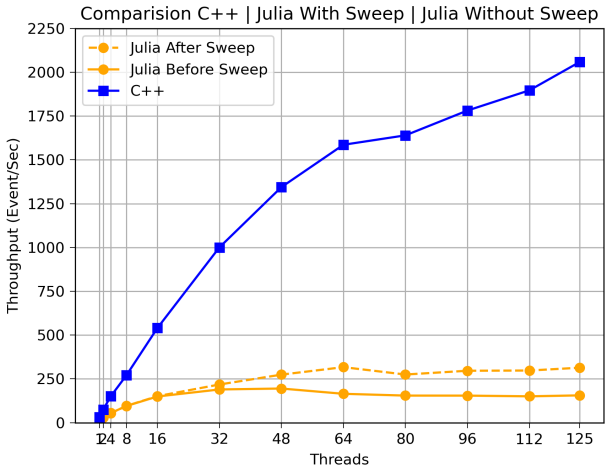

Figure 7: Throughput scaling of the pixel track sequential reconstruction as a function of thread count, comparing C++ with the Julia port before and after garbage collector sweep optimization. Julia shows significantly lower throughput due to garbage collection. Concurrent garbage collector sweep improves performance at high thread-counts but the performance remains significantly below C++. Benchmarks were performed on AMD EPYC 9534 system, with a warm-up of 1,000 events and main tests of 20,000 events (C++) and 10,000 events (Julia), repeated four times per configuration.

4.2 Compilation

As a complementary exercise, the ability to generate standalone binaries was explored. This aimed to improve the reproducibility and predictability of runtime behavior, allow smoother integration with the existing project organization, and enable more direct performance comparisons with other ports. In particular, deterministic runtime behavior is a general consideration for many HEP projects. Moreover, deploying compiled code at scale is generally preferred, since JIT-compilation would otherwise occur separately in every process on every node, leading to a significant and unnecessary waste of computing resources.

Two AOT-compilation methods were tested: PackageCompiler.jl and juliac from the Julia 1.12 release, similarly to Section 2.2. PackageCompiler.jl requires a pre-compilation file to capture the executed code paths; here, it was generated by running the program in single-threaded mode on 1,000 events.

The resulting binaries were verified with single-threaded execution. Both AOT-compiled versions were slightly slower than the JIT-compiled baseline (see Table 2), primarily due to increased garbage collection times, though the precise cause remains unclear. The executable produced with PackageCompiler.jl showed no residual JIT-compilation. In contrast, juliac reduced the JIT-compilation time during the warm-up execution by approximately 90% , but still a small fraction (<1%) of time was still spent on JIT-compilation during the main measurement.

Juliac trimming (discussed in Section 2.2.1) was also explored. However, it failed due to multiple errors originating from dependencies, and no further effort was made to enable it.

5 General findings

5.1 Distribution

The default method for installing Julia is a user-level installation, but the centralized distribution available with CERN LCG Releases was also tested and worked without issues. Running Julia on CERN infrastructure

Table 2: Comparison of JIT- and AOT-compiled Julia port of the pixel track serial algorithm. PackageCompiler.jl causes larger binaries and longer compile times. Each timing measurement was preceded by a warm-up run of 1,000 events using a single thread. Single-threaded tests for 1,000 events show that AOT-compiled code is slightly slower due to higher garbage collection overhead. Measurements were performed on an AMD Ryzen 7 5700G running AlmaLinux 9.

Compiler	Julia version	Compile time	Binary size	Runtime	GC time
JIT (baseline)	1.12.0-rc1	-	-	54.80 s	1.8 s (3.3%)
PackageCompiler.jl	1.11.5	~ 600 s	~ 500 MB	57.28 s	3.1 s (5.4%)
juliac	1.12.0-rc1	~ 120 s	~ 300 MB	58.37 s	4.4 s (7.5%)

is generally straightforward, although there are minor considerations: by default, Julia stores artifacts, logs, environments, and other files in the user's home directory, which might be not desired when the home directory is mounted from remote filesystem (AFS, EOS). It is possible to redirect Julia to another directory via an environment variable, although some auxiliary files may still be written to the home directory.

5.2 Versioning

Julia's public API, defined as the documented behavior of public bindings in the base module and the standard libraries, is intended to follow semantic versioning [32]. Nevertheless, breaking changes are occasionally introduced in minor releases. Furthermore, while Julia allows bindings to be explicitly marked as public, it is still not always clear which aspects of their behavior are formally documented, causing packages to fail when developers unintentionally rely on internal or undocumented behavior.

5.3 Basic data analysis

A substantial portion of data analysis and plotting accompanying the discussed project, such as analyzing benchmark results, was also performed in Julia. Overall, the experience is positive, offering ergonomics similar to Python. In addition, Julia provides a more fully featured environment and package management system, which allows finer control over dependencies, environments, and reproducibility. For this type of task, all the necessary packages were readily available, making the workflow straightforward and efficient.

5.4 Community and maintenance

The Julia community is small but ambitious and appears to be strongly identity-oriented. Positive results are often amplified, while critique is sometimes challenged or dismissed implying resistance to innovation.

Although the Julia ecosystem is split across many self-governing GitHub organizations, in practice a small group of contributors maintains much of the code across multiple organizations. This concentration of responsibility can lead to long response times for issues or pull requests. Direct pings on Julia Slack [33] are often the most reliable way to gain attention.

While this may not be representative of the entire ecosystem, a noticeable portion of the packages that were tested or used in these studies exhibited maintenance issues.

6 Conclusions

Julia is an interesting language with appealing ergonomics and many features that deserve recognition. However, none of the projects reviewed in this work achieved fully satisfactory results. Residual JIT-compilation remained even in AOT-compiled binaries and some of the essential performance optimizations seem to prevent the compiler from fully analyzing the code statically. Both the scheduling demonstrator and the Patatrack pixel track reconstruction port exhibited significantly lower performance than their C++ counterparts. Additionally, both projects independently highlighted Julia's garbage collector behavior under multi-threaded execution as a potentially critical limitation for certain HEP applications, such as event reconstruction.

The Julia ecosystem is already rich and offers a wide range of useful packages. However, several of the packages used in this work showed serious maintenance issues, including standard library, packages endorsed by the community or highlighted on the official Julia website. This raises broader concerns about the sustainability of the ecosystem, particularly for long-term projects. The Julia community is highly enthusiastic and often actively advocates for and promotes the language, a dynamic also observed in

other programming language communities, which can make it difficult to judge which claims reflect robust capabilities and which are more aspirational.

A deeper concern lies in several fundamental language design choices, such as the reliance on JIT compilation and garbage collection, appear to conflict with key HEP requirements. In particular, the limitations of Julia's garbage collector under multi-threaded workloads may pose a major challenge for certain high-performance applications, potentially restricting the language's ability to fully address HEP use cases. While ahead-of-time compilation is being added as an option, it is retrofitted onto a language that heavily depends on dynamic and reflective features. It remains unclear whether this approach can deliver the fully deterministic and reproducible behavior required for HEP workflows without undermining the performance and flexibility.

Overall, Julia is a powerful tool when applied in suitable domains. At the same time, the language and its ecosystem still appears immature, and it may be too early to draw definitive conclusions about its long-term suitability for high-parallel, high-throughput or large-scale HEP projects.

Acknowledgments

The author acknowledges that a substantial part of their contribution and involvement in the projects described in this report took place during a previous affiliation with the CERN EP-SFT group.

This work has been partially funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.

The work has been supported by the CERN Strategic Programme on Technologies for Future Experiments. https://ep-rnd.web.cern.ch/

References

[1] J. Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671.

[2] G. A. Stewart et al. "Julia in HEP". In: (Mar. 2025). arXiv: 2503.08184 [hep-ex].

[3] M. Stanitzki and J. Strube. "Performance of Julia for High Energy Physics Analyses". In: *Comput. Softw. Big Sci.* 5.1 (2021), p. 10. DOI: 10.1007/s41781-021-00053-3. arXiv: 2003.11952 [physics.comp-ph].

[4] J. Eschle et al. "Potential of the Julia Programming Language for High Energy Physics Computing". In: *Comput. Softw. Big Sci.* 7.1 (2023), p. 10. DOI: 10.1007/s41781-023-00104-x. arXiv: 2306.03675 [hep-ph].

[5] G. A. Stewart et al. *Polyglot Jet Finding*. 2024. DOI: 10.1051/epjconf/202429505017. arXiv: 2309.17309 [hep-ex].

[6] M. Cacciari, G. P. Salam, and G. Soyez. "FastJet User Manual". In: *Eur. Phys. J. C* 72 (2012), p. 1896. DOI: 10.1140/epjc/s10052-012-1896-2. arXiv: 1111.6097 [hep-ph].

[7] G. A. Stewart et al. *JetReconstruction.jl*. DOI: 10.5281/zenodo.12671414. URL: https://github.com/JuliaHEP/JetReconstruction.jl.

[8] E. Dimitrova and M. LeBlanc. *Julia-based study of SoftKiller performance in high-pileup conditions*. Boost 2025 poster. July 2025. URL: https://indico.physics.brown.edu/event/18/contributions/430/.

[9] JuliaSIMD. *LoopVectorization.jl*. Accessed: 2025-09-26. URL: https://github.com/JuliaSIMD/LoopVectorization.jl.

[10] JuliaSIMD. *VectorizationBase.jl*. Accessed: 2025-11-05. URL: https://github.com/JuliaSIMD/VectorizationVase.jl.

[11] JuliaLang. *PackageCompiler.jl*. Accessed: 2025-09-26. URL: https://github.com/JuliaLang/PackageCompiler.jl.

[12] J. Bezanson and G. Baraldi. *New Ways to Compile Julia*. Talk at JuliaCon 2024. Accessed: 2025-09-26. 2024. URL: https://www.youtube.com/watch?v=MKdobiCKSu0.

[13] G. Barrand et al. "GAUDI — A software architecture and framework for building HEP data processing applications". In: *Computer Physics Communications* 140.1 (2001). CHEP2000, pp. 45–55. ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(01)00254-5. URL: https://www.sciencedirect.com/science/article/pii/S0010465501002545.

[14] C. D. Jones et al. “Using the CMS Threaded Framework In A Production Environment”. In: *J. Phys. Conf. Ser.* 664.7 (2015), p. 072026. DOI: 10.1088/1742-6596/664/7/072026.

[15] B. Hegner, P. Mato, and D. Piparo. “Evolving LHC data processing frameworks for efficient exploitation of new CPU architectures”. In: *2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC)*. 2012, pp. 2003–2007. DOI: 10.1109/NSSMIC.2012.6551463.

[16] M. Fila et al. *FrameworkDemo.jl*. URL: https://github.com/key4hep/key4hep-julia-fwk.

[17] U. Brandes et al. “GraphML Progress Report Structural Layer Proposal”. In: *Graph Drawing*. Springer Berlin Heidelberg, 2002, pp. 501–512. ISBN: 978-3-540-45848-7.

[18] JuliaGraphs. *JuliaGraphs*. Accessed: 2025-09-26. URL: https://juliagraphs.org/.

[19] QuantumElectrodynamics.jl. *QuantumElectrodynamics.jl*. Accessed: 2025-10-23. URL: https://github.com/QEDjl-project.

[20] R. Alomairy et al. “Dynamic Task Scheduling with Data Dependency Awareness Using Julia”. In: *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2024, pp. 1–7.

[21] Julia Language. *The Julia Programming Language website*. Accessed: 2025-09-26. URL: https://julialang.org/.

[22] T.-W. Huang et al. “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (2022), pp. 1303–1320. DOI: 10.1109/TPDS.2021.3104255.

[23] JuliaParallel. *DistributedNext.jl*. Accessed: 2025-09-26. URL: https://github.com/JuliaParallel/DistributedNext.jl.

[24] S. Byrne, L. C. Wilcox, and V. Churavy. “MPI.jl: Julia bindings for the Message Passing Interface”. In: *Proceedings of the JuliaCon Conferences* 1.1 (2021), p. 68. DOI: 10.21105/jcon.00068. URL: https://doi.org/10.21105/jcon.00068.

[25] JuliaGPU. *NVTX.jl*. Accessed: 2025-09-26. URL: https://github.com/JuliaGPU/NVTX.jl.

[26] JuliaPerf. *PProf.jl*. Accessed: 2025-09-26. URL: https://github.com/JuliaPerf/PProf.jl.

[27] Julia Language. *Julia Documentation*. Accessed: 2025-09-26. URL: https://docs.julialang.org/en/v1/.

[28] *https://github.com/JuliaPerf/PProf.jl/issues/102*. GitHub issue, Accessed: 2025-10-23.

[29] JuliaFolds2. *OhMyThreads.jl*. Accessed: 2025-09-26. URL: https://github.com/JuliaFolds2/OhMyThreads.jl.

[30] CMS Patatrack. *PixelTrack Standalone*. Accessed: 2025-09-26. URL: https://github.com/cms-patatrack/pixeltrack-standalone.

[31] Andriotis, Nikolaos et al. “Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code”. In: *EPJ Web of Conf.* 295 (2024), p. 11008. DOI: 10.1051/epjconf/202429511008. URL: https://doi.org/10.1051/epjconf/202429511008.

[32] T. Preston-Werner. “Semantic Versioning”. In: *URL: http://semver. org* ().

[33] Julia Language. *The Julia Language Slack*. Accessed: 2025-09-26. URL: https://julialang.org/slack/.