Comparative efficiency of HEP codes across languages and architectures

Samuel Cadellin Skipsey[1,*] and *Graeme Andrew* Stewart[2]

[1]School of Physics & Astronomy, University of Glasgow, Glasgow, G12 8QQ
[2]CERN, Esplanade des Particules 1, Geneva, Switzerland

Abstract. Recently, interest in measuring and improving the energy (and carbon) efficiency of computation in HEP, and elsewhere, has grown significantly. Measurements have been, and continue to be, made of the efficiency of various computational architectures in standardised benchmarks, but those benchmarks tend to compare only implementations in single programming languages. Similarly, comparisons of the efficiency of various languages tend to focus on a single architecture, although it is the case that some abstractions in a given language can match specific architectural choices (in, say, memory ordering strictness) better than others. The existence of the JetReconstruction.jl project, implementing a subset of the FastJet C++ code's functionality in performant Julia, allows us to usefully compare how the relative efficiencies of implementations in the two languages are influenced by the architecture they are executed on. We report on the results of comparing benchmarks on these codes, and others, on AMD64 and various aarch64 implementations, amongst others.

1 Introduction

The effects of anthropogenic climate change are increasingly apparent across the world, driving responsible communities to work to reduce their energy usage (albeit that most significant changes need to be driven at a governmental level). In data-driven research communities, like those of the Worldwide LHC Compute Grid (WLCG), this means that we are increasingly interested in improving the energy-efficiency of our computational workloads - to first order, this is approximated by improving the time-efficiency of a workload [14], but more subtle effects (such as the higher power-load from SIMD instructions) complicate this somewhat. Simultaneously, we also must consider the language in which we write our code. The High-Energy Physics (HEP) community has passed through multiple phases in which a particular language or languages were dominant, and currently uses Python (with NumPy, Numba etc) in combination with C++ (for the "low level" parts that need to be fast). It has been argued that [1] the newer language Julia [2] is a better fit for the needs of scientific computing in general, and HEP in particular, for a range of reasons. In this paper, we will not explore these in detail, save to note that Julia's implementation choices mean that it can provide a syntax somewhat like NumPy with a performance similar to that of a compiled language like C++ [3].

*e-mail: Samuel.Skipsey@glasgow.ac.uk

It is not a novel concept that choice of programming language influences the efficiency of code compiled from it. In the 1980s, it was often noted that FORTRAN77's first-class arrays allowed a compiler to make stronger optimisations than for an equivalent implementation in C - the stronger classes of invariants guaranteed by FORTRAN allowed the compiler to prove that arguments to a function would not alias each other (overlap in memory), and thus track the state of the function's logic more reliably. Julia, which also provides built-in array and other mathematical classes, can provide similar invariants to a compiler. Contrariwise, modern processor architectures such as x86 and AMD64 have been influenced by the dominance of the C (and C++) "virtual machine" to present interfaces that directly optimise for C-like function calls and array addressing. This presents an advantage to languages - like C++ - that are aligned with C's virtual machine model. There, however, remain differences between architectures, at the level of memory-ordering consistency and other higher-order considerations. [4]

Thus, it is of interest to compare implementations in multiple languages, across multiple architectures, in order to determine the strength of these effects.

In this paper, we report on some results from benchmarking Julia and C implementations of algorithms relevant to HEP. The core of our report is a series of benchmarks comparing the FastJet [8] package, a well-known and mature library in C++ for jet clustering, with JetReconstruction.jl [9], an implementation of a growing subset of FastJet's functionality in Julia. As compared to ubiquitous microbenchmarks, we hope this provides a more contextually useful comparison of the scaling of code in Julia and C++ for HEP workloads.

1.1 Previous Work

This work was partly inspired by previous work performed to show the effectiveness of the Julia language for HPC workloads. Giordano, Klöwer and Churavy [10] performed a series of matrix-mathematics benchmarks on the ARM A64FX based Fugaku supercomputer, using Julia's standard library and optimised C++ linear algebra packages. They showed that Julia, in 2022, could outperform even Fujitsu's own optimised libraries for some workloads on ARM architectures.

In addition, Pereira, Ribeiro, et al [11–13] published a series of analyses of the resource efficiency of programming languages, on multiple axes, running on x86 architectures. Whilst their original paper did not consider Julia as a language, their later update added it amongst many others. Considering the pareto-optimal set of languages, whilst compiled languages like C and Rust place highly, Julia also places extremely well (competitive with Fortran and Chapel). This clearly motivates more exploration of the specific usefulness of Julia in our domain. As these papers only studied power efficiency on x86 architectures, it is a natural extension to consider others here, especially as the studies including Julia are now four years old (and as a young language, Julia has made more progress in that time than more "mature" languages it was compared against have).

Work on JetReconstruction.jl has been reported in previous CHEPs, with a longer update in this proceedings [5]. That paper covers mostly performance measurements on x86 architectures and does so in more detail than we cover here.

2 Methodology

The Glasgow WLCG Tier-2 site has been performing energy-efficiency benchmarking on a wide range of systems for some years [6] across both AMD64 and ARM ("aarch64") architectures of multiple generations. (We have also benchmarked RISC-V systems, but Julia does

Table 1. Summary table of the systems used for benchmarking codes in this paper. The bottom two systems are ARM based, whilst the remainder are AMD64.

System ID	CPU	HEPSCORE (/thread)
h16	2xIntel Xeon 8-core E5-2630	9.69 (HT)
h17	2xIntel Xeon 10-core E5-2630v4	9.64 (HT)
d22	2xAMD EPYC 32-core 7513	14.74 (HT)
a23	2xAmpere Altra 80-core Q80-28	15.75
m24	Ampere Altra Max 120-core M120-30	16.57

not currently support RISC-V as a stable target, so we did not include the RISC-V system here). Much of our benchmarking has been supported by the HEPSCORE [7] project, which provides a well-tested suite of composite benchmarks covering real-life workloads from the WLCG experiments, as well as infrastructure to host benchmark results indexed by machine configuration, and ongoing development work to add additional instrumentation, such as energy use per benchmark.

For all of the following benchmarks, we took a single representative machine from the generations currently in production at Glasgow. Table 1 shows the different machines, with their code-name we will use to refer to them. These machines were all installed with Alma Linux 9, with a fully updated system image at the time of benchmarking. For C and C++ code, the distribution's standard version of Clang was used, to minimise the issues of different compiler optimisation flows (as Julia's JIT is also LLVM-derived). Julia ships its own version of LLVM (tied to the Julia release), and so it may be slightly ahead of the conservative LLVM releases present in RedHat packaging. Julia was installed via the 'juliaup' tool, pinning to release 1.11.1 . Selected benchmarks were evaluated on Julia 1.10.5 and 1.11.0, but no significant changes in relative performance were measured.

3 Results

3.1 Microbenchmarks

Plentiful microbenchmarks exist for programming languages, although it is not always clear what aspects of the language they stress. We constructed a purpose-built microbenchmark for the purposes of validation of the principle of this paper. The task is to generate a $10^4 \times 10^4$ sample image of the Julia set of the iterated Newton's method solution of $x^3 - 1 = 0$. Specifically, we require to store the intermediate state of the whole image in memory - which requires on the order of 10^8 bytes (~100MB), comfortably exceeding the cache size on all of the CPUs tested, and ensuring that any garbage collection flows in the language are exercised. Julia does, by default, perform automatic garbage collection, via a parallel mark-and-sweep algorithm, and so this overhead is relevant for large memory operations. The actual process per pixel is a tight iterated loop, testing the efficiency of iterative control flow in a language (Python, for example, has famously slow for loops, and is hampered in these kinds of benchmarks by that performance limitation).

Figure 1 shows the resulting measured time of execution of idiomatic implementations in Julia and C, using the system default release of clang for compilation; both implementations are single-threaded.

As can be seen, the same sets of code produce markedly different relative performance on the different nodes. Relative performance on the AMD64 nodes is proportional to the per-thread HEPSCORE of the system, and shows an advantage to the C implementation. Conversely, and surprisingly, the relative performance on the ARM systems is reversed - the

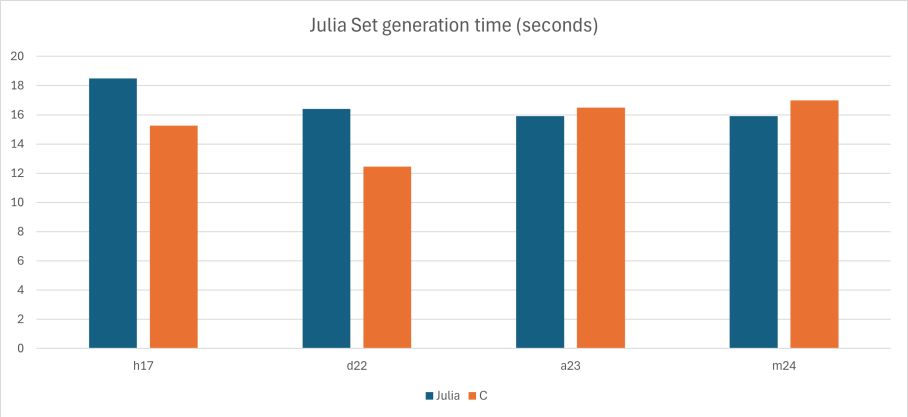

Figure 1. Absolute execution times for a large Julia set iteration as described in text body, implemented in Julia and C, single-threaded. System h16 is not included due to time constraints.

Julia implementation is consistently faster — and there is no clear scaling with HEPSCORE per thread.

Examination of the assembly code generated by the Julia JIT, compared to that produced by Clang for C, on ARM, suggests that this is mostly due to poorer optimisation by the system Clang instance. As Julia ships its own version of LLVM, as mentioned above, we assume that this is the confounding factor.

3.2 Jet Clustering Algorithms

Unlike the deliberately trivial code compared in the previous section, both JetReconstruction.jl and FastJet are mature codes, utilising all of the features of their respective languages to achieve high performance. For example, FastJet uses C++ memory fences to fine-tune concurrency of operations across its datastructures, and JetReconstruction.jl leverages Julia features such as the @simd macro to allow automatic vectorisation of loops at the machine-code level. As such, benchmarks here show how the two codebases can exploit a range of different architectures via their languages' affordances.

3.2.1 Type Concreteness

During the development of this work, JetReconstruction.jl went through multiple minor releases. We highlight briefly here an interesting aside about how correct use of generics in Julia brings significant performance improvements. JetReconstruction v0.4.0, the first release to use a type-generic form for event containers, also incurred a significant performance degradation. Attempts to mitigate this in v0.4.1 by introducing cache-friendly representations were disappointing, as they only resulted in performance improvements on Apple Silicon (indeed, testing on our Ampere ARM CPUs at Glasgow showed no effect, in common with AMD64 systems). Type genericity in function signatures should not result in a performance hit in the first place, however — Julia specialises methods according to the types they are called with, so there should be no overhead in the compiled code that executes. The problem, in fact, was that the code was not doing what was intended.

Listing 1. Example of inefficient and efficient ways to genericise a function argument over a container type

```
function inefficient(Vec{Number})
    #this function takes a Vector of boxed values
    #constrained so each value is a subtype of Number
    #this is slow to access as each value is resolved
    #by pointer indirection
end

function efficient(Vec{T}) where T<:Number
    #this function takes a uniform Vector of
    #   unboxed concrete type T
    #where T must be a concrete subtype of Number
    #this is fast to access, and cache-friendly
    #as each value is concrete and packed
end
```

As listing 1 demonstrates, there are subtleties regarding how Julia interprets specification of container types with non-concrete element types. A function specified as taking a container of non-concrete elements will assume that it is passed a container using indirection to box the values, which makes access and update of those elements slow. The fast way to provide flexibility - if the container will, in fact, only ever contain one element type for any invocation - is to specify the function as taking a container of generic type T, and then use a where clause to provide the restriction of possible values of T. This produces functions which take packed, uniform containers with unboxed types, significantly improving efficiency.

3.2.2 Relative Performance Tests

Relative performance tests using the standard set of test event files provided with JetReconstruction.jl v0.4.2 were performed across all the available test frameworks in Table 1.

Figure 2a shows the relative performance, as the number of events increases, for FastJet and JetReconstruction.jl using a "Plain" strategy - in which the event space is not segmented - and AntiKt clustering. It can be seen that on all architectures, the Julia implementation is significantly more performant, with a steady-state ratio of around 80% to 66% of the time needed by FastJet. The relative advantage scales according to the SIMD-width and cache-sizes of the host architecture - Julia's JIT can auto-vectorise code at the time of execution to match the host's capability, which gives it an advantage here versus FastJet, which is dependant on choices made at compile time (to match a generic set of machine features).

Figure 2b shows a similar comparison, using the Tiled strategy, in which search space is segmented into localities of scale R, distribution of the clustering problem. Here there is a notable difference across architectures: the ARM-based Ampere nodes both show FastJet with a small (~5%) advantage over the Julia implementation, whilst all of the AMD64 architectures show a slightly larger (~10%) advantage *to* Julia.

Figure 3, for comparison, shows the behaviour of the same Tiled strategy, but for a much smaller R (thus reducing the maximum size of a cluster). In this scenario, again, all architectures show a similar trend, and all have the advantage to the Julia implementation. As R increases, the main effect on the algorithm is to increase the stress on cache (as more particles need to be considered and re-read when clustering), which suggests that the relatively small caches on the ARM nodes is the cause of performance regression for ARM relative to AMD64.

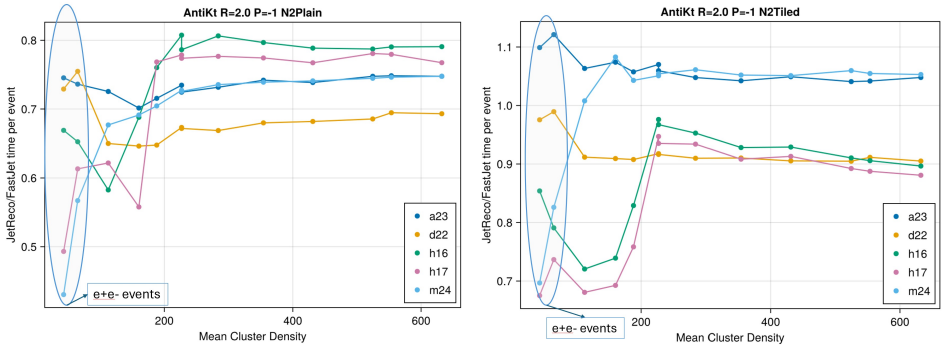

Figure 2. Relative performance of FastJet and JetReconstruction.jl with AntiKt clustering, R=2.0, for the Plain (left) and Tiled (right) strategies.

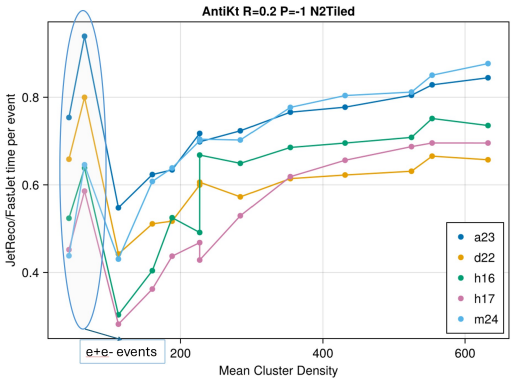

Figure 3. Relative performance of the FastJet and JetReconstruction.jl codes for the AntiKt clustering, Tiled strategy for small R=0.2

Comparing absolute performance metrics for FastReco and JetReconstruction.jl for the Kt-like "Durham" algorithm, in Figure 4, shows how the variation in relative performance observed for "small event" samples is due to significant fluctuation in the performance of the FastJet code between the two ARM nodes. This is surprising - the main difference between the a23 and m24 nodes is that the latter's Altra Max has effectively more cache contention due to the larger numbers of cores, but neither tested implementation is explicitly multithreaded.

4 Conclusions

Performance testing across multiple architectures is an increasingly relevant task, in a world where x86 architectures are no longer the obvious target. Even with a fixed algorithm and compiler, performance can noticeably vary between architectures, and editions of architectures, and not in a linear fashion. When you include the effects of porting an algorithm between languages, we have shown that the dependencies merely grow more complex, due to the applicability of assumptions exposed by each language to a destination machine code. For the FastJet and JetReconstruction.jl dyad, however, there is an overall trend that the Julia implementation is slightly more performant, even on the more energy-efficient ARM archi-

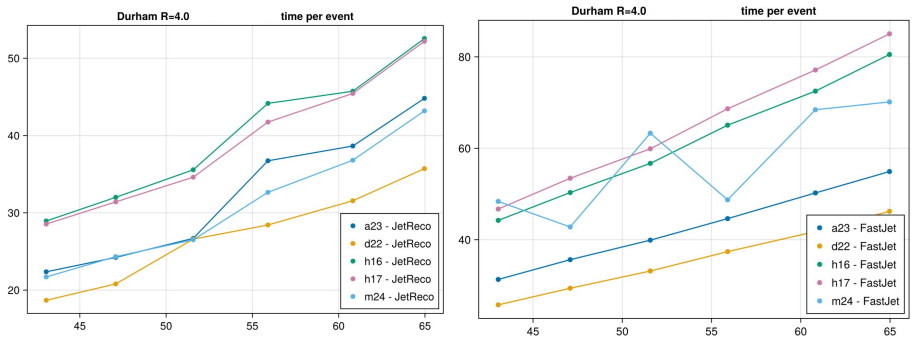

Figure 4. JetReconstruction.jl (left) and FastJet (right) absolute execution times for the Durham clustering algorithm.

tectures tested. We believe that this is a positive indicator, as the Julia language is also easier to teach than C++ and has better safety characteristics.

5 Acknowledgements

The authors acknowledge the support of the JuliaHEP community in this work. Additionally, they would like to thank Emanuele Simili for assistance in benchmarking for the specific nodes at Glasgow.

References

[1] Graeme Andrew Stewart, et al, *Julia in HEP*, Proceedings of the CHEP 2024 conference (to be published)

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral Shah. *Julia: A fresh approach to numerical computing*, SIAM review **59** pp65-98 (2017) https://doi.org/10.1137/141000671

[3] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. *Julia: dynamism and performance reconciled by design.* Proc. ACM Program. Lang. 2, OOPSLA, Article 120 (2018), https://doi.org/10.1145/3276490

[4] Gupta, Khushi, and Tushar Sharma. *Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors.* International Journal of Scientific Research in Computer Science, Engineering and Information Technology **7** 619-631 (2021). https://doi.org/10.32628/CSEIT2173188

[5] Graeme Andrew Stewart, *Fast Jet Reconstruction in Julia*, Proceedings of the CHEP 2024 conference (to be published)

[6] Simili, E., Stewart, G., Skipsey, S., Spiteri, D., Borbely, A. and Britton, D. *ARMing HEP for the future Energy Efficiency of WLCG sites (ARM vs. x86)* EPJ Web of Conferences **295** 11007 https://doi.org/10.1051/epjconf/202429511007

[7] Giordano, D., Barbet, J.M., Boccali, T., Borge, G.M., Hollowell, C., Innocente, V., Lampl, W., Michelotto, M., Meinhard, H., Ondris, L. and Sciabà, A. *HEPScore: A new CPU benchmark for the WLCG* EPJ Web of Conferences **295** 07024 (2024) https://doi.org/10.1051/epjconf/202429507024

[8] Matteo Cacciari, Gavin P. Salam, Gregory Soyez. *The FastJet User Manual*, https://arxiv.org/abs/1111.6097, (2011)

[9] Graeme Andrew Stewart and Philippe Gras and Benedikt Hegner and Atell Krasnopolski. *Polyglot Jet Finding* https://arXiv.org/abs/2309.17309 (2023)

[10] Giordano, Mosè and Klöwer, Milan and Churavy, Valentin *Productivity meets Performance: Julia on A64FX* 2022 IEEE International Conference on Cluster Computing (CLUSTER), pp549-555 (2022) https://doi.org/10.1109/CLUSTER51413.2022.00072

[11] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. *Energy Efficiency across Programming Languages: How does Energy, Time and Memory Relate?*, Proceedings of the 10th International Conference on Software Language Engineering (SLE '17) (2017) https://doi.org/10.1145/3136014.3136031

[12] *ibid.* "Energy Efficiency of Languages Updated" https://sites.google.com/view/energy-efficiency-languages/updated-functional-results-2020?authuser=0 (last accessed 28 Feb 2025)

[13] *ibid.* "Ranking programming languages by energy efficiency" Science of Computer Programming 205:102609 (2021) https://doi.org/10.1016/j.scico.2021.102609

[14] Nicolas van Kempen, Hyuk-Je Kwon, Dung Tuan Nguyen, Emery D. Berger "It's Not Easy Being Green: On the Energy Efficiency of Programming Languages" https://arxiv.org/abs/2410.05460 (2024) https://doi.org/10.48550/arXiv.2410.05460