International Journal of Research in Engineering, Science and Management
Volume-2, Issue-4, April-2019
www.ijresm.com | ISSN (Online): 2581-5792

A Review of Existing Approaches to Increase the Computational Speed of the Python Language

M. Varsha[1], S. Yashashree[2], Drishya K. Ramdas[3], Sini Anna Alex[4]
[1,2,3]*Student, Department of CSE, M. S. Ramaiah Institute of Technology, Bengaluru, India*
[4]*Assistant Professor, Department of CSE, M. S. Ramaiah Institute of Technology, Bengaluru, India*

***Abstract*: The Python programming language has gained prominence in a wide range of fields, ranging from Computer Science to Medicine and Genetics to Astrophysics and Cosmology. The main reasons for this extensive applicability are Python's versatility, the availability of robust, built-in libraries for all computations, readability, support for multiple programming paradigms and compatibility with several platforms. However, Python being an interpreted language, is slower than native, compiled languages like C/C++. It is also memory-intensive, due to flexible data types and throws a number of runtime errors, owing to flawed design. Most of the methods that have been developed to overcome Python's low speed deal with optimizing Just-In-Time (JIT) compilers, specific to the Python language, which compile the interpreted codes on-the-fly. Some techniques describe general methodologies like optimum utilization of GPUs and increasing the efficiency of JIT Virtual Machines for dynamic programming languages. We present an expansive summary of all cutting edge approaches which aim to tackle Python's low speed, in the hope that this would further increase its reach.**

***Keywords*: JIT compilers, runtime, multiprocessing, speed, Python, efficiency, Virtual Machines.**

1. Introduction

The trend of late has been to adapt to dynamic programming languages as they are more versatile in a variety of aspects including: (a) Being compact and epigrammatic in nature; (b) Prevention of wasting time over debugging and semantic errors; (c) More tolerant to change and passing coding trends; (d) Quality performance in terms of complexity with respect to space and time; (e) Being platform independent thereby promoting the concept of portability. The most preliminary requirement is that there is an absence of a separate compilation step. The concept of an incremental, instant or partial compilation does not exist and that is why people are switching to these faster and less verbose languages. In the realm of High-performance computing and parallel processing, dynamic languages fail to rise to the occasion as it leads to detrimental consequences in the form of lack of code completion and most importantly, better performance. In lieu of the aforementioned, we are using Python as the de-facto programming language as it does not involve extending our outreach to a new programming language and is highly compatible with most of the packages and modules available in our constantly evolving world. However, speed is an extremely important factor in most applications, since a simple Python code takes 10 seconds to execute, as compared to the 4 seconds taken by its C++ counterpart. In order to surpass this hurdle, we have presented a concise and complete compendium on how to increase the efficiency of Python as a dynamic programming language, with the avant-garde mechanism of Just in time compilation (JIT). The several frameworks encompassed within Python have excelled by removing the unnecessary byte-code thereby eliminating wasteful interpretation time. Keeping in mind overall complexity with the execution time taken, our proposed methodology characterizes several solutions to overcome these obstacles, thereby retaining the simplicity of Python and the elegance of Just In Time (JIT) compilation approaches.

2. Literature survey

The paper by Hannon et. al. describes Just-In-Time compilation as the process of dynamically compiling codes during execution. It also mentions how JIT compilation takes advantage of interpretation as well as inert compilation of program codes to perform optimizations that speed up processing. The paper mainly deals with developing a parallel discrete event simulation engine (PDES) named Simian, that performs JIT compilation on interpreted languages – JavaScript, Lua, Python and others. The Pending Event Queue and Pseudo Random Number Generation aspects of the PDES engine were compared and evaluated using the La-PDES benchmark suite. Techniques used by interpreted languages to invoke C modules have been explored. It was found that JIT compilers outperform AOT (Ahead-Of-Time) compilers with native data structures while the reverse was true for operations that invoke native code. [1]

The book by G. Morra contains a chapter on how to speed up calculations in Python by making optimum use of the NumPy module, and avoiding bottlenecks like redundant loops and heavily vectorized functions. Morra has clearly illustrated how to perform efficient computations using ndarrays, n-

dimensional indexing and Boolean indexing. After introducing basic linear algebra concepts, the chapter describes the working of Cython, the superior language which compiles Python code dynamically into C code. The mpi4py, a version of the Message Passing Interface (MPI) library that enables the distribution of code across processors, has been used to write native Python code, in order to bring in parallelism into the picture. Finally, the Python accelerator, Numba is described as having offered an increase in speed of one order of magnitude, while using large NumPy arrays in the program code. [2]

Yangguang Li and Zhen Ming Jiang's paper is one of the few that mentions the effects of the various configuration parameters on the performance of PyPy, a Python implementation. Two benchmarks have been used to compare the execution of PyPy code before and after jitting the code - the PyPy benchmark suite and the Tech Empower Web Framework Benchmark suite. The systems were divided into three groups based on JIT configurations – varying default configurations, randomly generated configurations and JIT off, which disables the jitting process for the code. It was found that systems using the default JIT configuration settings perform badly, as compared to the others, and that the optimal configuration was system-dependent. To overcome this issue, the authors have proposed a tool, PyPyJITTuner, to automatically tune the JIT parameters that impact performance, using a search-based technique called the ESM-MOGA (Effect Size Measurement-based Multi-Objective Genetic Algorithm). Systems which used the auto-tuned ESM-MOGA configurations showed an improvement of upto 60% in mean peak performance, over systems which used the default settings. [3]

Yet another paper by Paras Jain et. al., points to under-utilization of hardware accelerated devices like GPUs and TPUs due to factors like different kernel dimensions, unpredictable intervals between process arrivals, and stringent latency constraints, while running ML algorithms. The authors propound an Out-of-Order (OoO) JIT compiler, built on top of VLIW that satisfies SLOs, and reschedules the kernel executions dynamically to ensure that the throughput of the device is maximized. The paper also describes why time-only and space-only GPU multiplexing fails, hence advocating a late-binding, aware-of-context approach to programming the order of execution of kernels on the GPU. Unlike conventional VLIW compilers which only modify code beforehand, this aggressive approach uses the concepts of tuning early and packing dynamically. [4]

The versatility of the Python programming language is overshadowed by its low runtime performance which makes it difficult to use for applications like astrophysics and cosmology, which involve massive training and extensive datasets. The computational speed of the Python language can be increased by either maximizing Just-In-Time (JIT) compiler performance, or improving the speed of interpreters. HOPE is a Python Just-In-Time compiler that constitutes a superset of the popular programming dialect, involving mathematical expressions mostly used in astrophysical calculations. It converts Python code to C++ to achieve speeds of native languages. It is also non-intrusive because it only needs a decorator to be added to the function definition in order to be enabled, and no unnecessary packages need to be included. The paper includes a detailed flow chart of how HOPE handles a function call. HOPE was found to increase the performance of Python, when tested on the PyCosmo project, by a factor of 2.4 × –119× x (a value that is benchmark scenario-dependent) [5].

Richard Plangger et. al. contributed to the PyPy's vectorization which is built into the tracing just-in-time compiler. Tracing JIT's are based on optimization of hot loops. Traditional vectorization methods have a lot of overhead. The auto new vectorization method proposed has less overhead and provides numerical loop speedup. It makes use of loop unrolling, efficient heuristics for instruction categorizing and scheduling, guard instruction usage reduction and guard strengthening, removal of redundant array index out of bounds check, splitting of accumulator for reduction. It also supports expansion of scalars and constants. In the algorithm proposed, parallel instruction groups are found, extended and combined together. The evaluation showed that the vectorizer provides speedups which are close to SSE4 instruction set.[6]

Serge Guelton et. al. presented the Pythran compiler, a translator and an optimizer for a portion of Python language. Pythran converts the static python modules into C++ code with parameters. It supports high level python constructs such as list, map, dictionary, lambda functions, polymorphic functions and nested functions. It does not support dynamic feature and classes. Pythran provides Python-centric code optimization unlike existing static python compilers. Pythran provides API similar to python standard library as it makes use of C++ library which uses template programming. The static Pythran compiler has a front-end where python code is converted to internal representation, a middle-end where the internal representation is optimized, and a back-end where internal representation is converted into C++ code. Pythran takes the advantage of multiple cores. Using Pythran, high level language code can be run at descent speed [7].

Wim TLP Lavrijsen in his paper speaks about optimizing the ROOT I/O in python using PyPy's tracing JIT. PyPy aims at providing a support framework for producing dynamic language implementation and emphasizes on clear separation between aspects of implementation and language specification. PyPy's tracing JIT compiles the high level python code to machine code at native speeds. The cppyy project provided C++ bindings to PyPy. It also kept the code transparent to JIT while providing bindings. By applying these cppyy techniques to ROOT I/O of python, there was a performance gain of 20x times over CPython was gained. However, it still is 2.7x times slower compared to C++ [8].

Serge Guelton in his paper introduces principles of Pythran which is an ahead-of-time compiler for scientific python.

Backward compatibility, type agnosticism, high level and pure native are the core principles around which Pythran is built. Pythran easily generates hundred percent native functions from code written in python and puts it into a PyCapsule rather than generating an extra conversion layer. PyCapsule is a simple PyObject that was introduced in Python C API. Pythran provides concurrency support by releasing the Global Interpreter Lock after converting Python objects to Pythran objects as there would be no more interaction with Python C API. Pythran helps pass the native or interpreted boundary and hence provides significant speedups without changing the original code [9].

This paper by Derek Lockhart primarily focuses on using Instruction Set Simulator (ISS) to achieve a balance between high productivity and performance levels. The ISS must be generated from Architectural Description of Languages (ADLs) promoting the fundamental concept of decimal to binary translation (DBT) techniques. To achieve this, the concept of Pydgin is used where-in directly executable pseudo codes are generated using meta-tracing of JIT compilers. To reduce the complexity of interpreter codebases, meta–tracing annotation is used to simplify the design of dynamic languages. This is achieved using the module RPython which provides instruction semantics and a modular instruction interpreter which follows the mechanism of a translation toolchain. The bytecode generated can be obtained from various phases and once the loop is finished, the control returns to the jit_merge_point.[10]

This paper by Berkin Ilbeyi elucidates on increasing the productivity and performance of dynamic programming languages thereby motivating the need for just in time compilation (JIT). However, while developing JIT optimization VM's consumes a lot of time and effort. Using JIT frameworks, abstraction of language definitions seems to be the best possible option. A new cross layer methodology is proposed between 2 frameworks to characterize a variety of standards at the application level. Interpreted code is generally 10x magnitude times slower that statically compiled code, so using JIT at runtime optimizes the bytecode fetch and decode overhead. RPython as a framework identifies hot target loops and the generates a trace to go about solving the loop. This can be used a fundamental premise to describe a recent branched-layer characterization methodology that enables inserting branched layer annotations at an upper layer and then explicate this at a lower level [11].

This paper by Juan. G. Galvez and 2 others comments on the extremely challenging task of parallel programming. Although several mannerisms are deployed to reduce the complexity, they have still remained unused for several reasons. Here, a new model is proposed namely ChamPy which is primarily modelled using the dynamic approach of Python. The salient characteristics include an intelligible model and API, better pliability and obviously the programming power of the Python language. This is a complex model-driven archetype of distributed relocatable objects. It reaps gains from several modern language features and it can run asynchronous concurrent jobs. Molecular dynamics mini apps can also be written using the same. Numba JIT compilation is an additional feature to this model. It is easy to absorb as it is an introductory level programming language. Keeping, High Performance Computing (HPC) in mind, message parsing as the de facto standard is replaced with a distributed computing framework. CharmPy uses the archetype of diffused, and varied relocatable objects with nonsynchronous distant procedure citation which is mentioned using the concept of objects keeping in mind the interlinkage among the same [12].

The work presented in this paper by Mihai Bucurica is focused on the realm of furtherance of computational orderliness of a finite state automation using the concept of cellular technology in a simulated virtual based setup. This is achieved using the 3 fundamental pillars i.e. Java, Python and Python with an additional package of Numba with Just in Time Compilation (JIT). The third option was considered to ramp up the speed of the application to ensure completion and fulfilment by dispatching it using the beauty of Python. The package Numba generated useful and efficient machine code with minimal complexity by running it on the LLVM compiler framework. It also extends its support to compilation of code on any kind of a processing unit and is developed in a manner to correlate and combine the technical aspects as well. This is implemented to specify and cater to the needs of platform-neutrality, not swaying to the specifications of the various components. It was also observed that the overall administrations in total, are superior to those obtained by using other programming practices. This was done by improving the copy_matrix function. This ensures nominal average programming time of several different sections, typically on cheaper environments. In total, the all-inclusive time complexity associated with Numba through JIT compilation superseded that of an average Python code [13].

This paper talks about how essential Python language is in the field of performance resulting in higher throughput and computerization of scientific aspects. It touches on how the language can be broadened by using one of several domains. The domains mentioned are Numpy, SciPy, PyPy, Cython and Numexpr, resulting in expanding the versatility of the language. A multitude of methods have been presented in a concise manner which can be referred to thereby promoting the speed and optimality of the same. The modularity is also explained, resulting in top-notch code thereby, being of great significance as well. It makes it clear how this is a notch above the standard programming practices as it is definitely a class apart from the rest in terms of efficiency obtained. This can be achieved using Numba as a single entity or alongside JIT with the vectorization feature also [14].

3. Discussion

The effortless nature of Python is indispensable and is what draws several users to explore and use it with utmost ease. This

being the case, Python offers an abundance of features in terms of its packages and frameworks. Some of the significant ones, that stand out much higher than the rest are:

The effortless nature of Python is indispensable and is what draws several users to explore and use it with utmost ease. This being the case, Python offers an abundance of features in terms of its packages and frameworks. Some of the significant ones, that stand out much higher than the rest are:

- PyPy – This is a unique framework-based environment that Python offers to bring about code that is optimal in all measures. With that being said, it uses the concept of compilation in the nick of the time. This results in code being immediately translated into its equivalent machine type. Scripts that are developed using this strategy are very storage-efficient as it hardly takes up significant memory. The simultaneous execution as a process becomes relatively easier, making it rapid, speedy and well-ordered. The rate of program execution is significantly faster, yielding better results;
- Numba – This is a wonderful aggregate of the two most commonly used approaches in our day-to-day programming interfaces namely NumPy and Just in Time (JIT) compilation mechanism. It was brought into existence keeping in mind all the usual coders who dealt with quantitative and data-structure directed processing units. It is an open source platform that can be easily accessed, used by one and all making it user-friendly and easy to work with. It is the conglomeration of a variety of bundles that can be used from something like a small-scale application to larger applications including distributed computing and data warehouses. It focuses on the concept of data analytics and varied sciences;
- CharmPy – This is a relatively new concept that was introduced to bring about betterment of the existing Python language by additionally boosting it with the Charm framework. It focuses on running many jobs that are anachronic in nature thereby bringing about faster execution rates. This works in association with the Numba package combining Just in Time (JIT) compilation to literally achieve the best of both worlds. It is object oriented in the sense that it concentrates primarily on the adducing of objects. The key aspect to be kept in mind here, is the processing element (PE) that sustains inter and intra class similarity. It also provides the added benefit of addition and deletion of objects without structurally damaging the infrastructure of the framework;
- Pythran - Pythran, an ahead-of-time compiler was developed for scientific python. It provided significant speedups over high-level Numpy implementation thereby relieving the scientists from the burden of using lower-level languages for scientific computing. Pythran converts the static python modules into C++ code with parameters. It supports high level python constructs such as list, map, dictionary, lambda functions, polymorphic functions and nested functions but unfortunately it does not support dynamic features and classes. Pythran provides Python-centric code optimization unlike existing static python compilers. Pythran outperformed Cython by supporting backward compatibility, type agnosticism and high level which weren't supported by Cython;
- Pydgin –This is the backbone of modern programming use case where-in a Python powered DSL is used to conjure Instruction Set Simulators. It follows a systematic procedure of encoding the rules based on semantics obtained and running it through a finite loop, delivering an annotation-based tree and parallelly a refined script. This is a first-class executable that can be spawned across several interpreter-based contexts and over suitable frameworks as well.

As Python began to be used more and more across diverse sectors, certain shortcomings were noticed, primarily, the low computation speed. Two methods were developed to overcome this issue, one was that of increasing the speed of interpreters, and the other was the optimization of Just-In-Time (JIT) compilers. The latter can be done by using parallel discrete event simulation engines, message passing interfaces (MPIs), tuning configuration parameters so as to provide the best performance, enabling complete utilization of hardware accelerators like GPUs and TPUs, or by developing JITs which are directed at a specific application.

The distribution of discrete events across computers is called parallel discrete event simulation (PDES); it incurs lesser overhead in terms of time and memory than serial event simulation. Simian is one such PDES engine that performs jitting on the codes of interpreted languages. Python packages like NumPy have features like vectorized arrays, which, if used judiciously, can speed up computation. Cython is a superset of the Python language and JIT-compiles Python code into faster C code. When the JIT tool, PyPy JIT Tuner was used to automatically tune the configuration features and compile PyPy, a popular implementation of Python, its average peak performance increased up to 60%. Finally, Python JIT compilers like HOPE are designed for specific applications, in this case, astrophysics and cosmology. By using a subset of the Python numerical expressions and compiling the code on-the-fly, HOPE has managed to raise the speed of Python.

Loop execution is one of the major reasons behind simple python codes taking lot of execution time as compared codes written in native languages like C/C++. To overcome this problem, Tracing Just-In-Time compilers were developed for the optimization of hot loops. Efficient heuristics were used in new vectorization methods which resulted in less overhead as compared to traditional vectorization methods and resulted in

speedups which were close to SSE4 instruction set. The working of a JIT model can be explained with respect to 4 different phases namely the inference phase, optimization phase, code generation phase and finally the control returns to an arbitrary point defined in the start-up routine namely the JIT merge point.

4. Conclusion

A detailed examination of the research work based on optimizing the Python language has shown that the problems once faced due to higher execution time of Python codes as compared to its native C/C++ codes is now tackled by using Python Just-In-Time compilers like Numba, Pydgin, RPython, Cython, Pythran and HOPE. Daunting problems that were terribly significant such as loop execution, parallelism, ROOT I/O can now be easily conquered using Just-In-Time compilers. The wide range of approaches and optimization techniques resulted in significant speedups which relieved the burden of using lower-level languages for scientific computing. This survey paper is drafted to provide a comprehensive summary of all revolutionary tried and tested approaches which aim to tackle the low speed of the Python language. Most of these methods deal with enhancing Just-In-Time (JIT) compilers which compile the interpreted Python codes instantly and immediately thereby reaping the best benefits of the same. The survey paper deals with all the Python frameworks, modules, packages that have increased the compilation mechanism speeds thereby proving to deliver fruitful results in the fields of High-Performance Computing and Parallel Processing. We have strived our best to propose an all-inclusive python related JIT compiler that touch upon each and every nook and corner of this particular domain. We aim to assist researchers and help them with their subsequent work by comparing their proposed methodologies with the existing ones that have been studied in our survey.

References

[1] Hannon, Christopher, Dong Jin, Nandakishore Santhi, Stephan Eidenbenz, and Jason Liu. "Just-in-time parallel simulation." In 2018 Winter Simulation Conference (WSC), pp. 640-651. IEEE, 2018.
[2] Morra, Gabriele. "Fast Python: NumPy and Cython." In Pythonic Geodynamics, pp. 35-60. Springer, Cham, 2018.
[3] Yangguang Li, Zhen Ming (Jack) Jiang, “Assessing and optimizing the performance impact of the just-in-time configuration parameters - a case study on PyPy”, Springer Science+Business Media, LLC, Springer Nature 2019.
[4] Jain, Paras, Xiangxi Mo, Ajay Jain, Alexey Tumanov, Joseph E. Gonzalez, and Ion Stoica. "The OoO VLIW JIT Compiler for GPU Inference." 2019.
[5] Akeret, Joël, Lukas Gamper, Adam Amara, and Alexandre Refregier. "HOPE: A Python just-in-time compiler for astrophysical computations." Astronomy and Computing 10 (2015): 1-8.
[6] Plangger, Richard, and Andreas Krall. "Vectorization in PyPy's Tracing Just-In-Time Compiler." In Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, pp. 67-76. ACM, 2016.
[7] Guelton, Serge, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. "Pythran: Enabling static optimization of scientific python programs." Computational Science & Discovery 8, no. 1 (2015): 014001.
[8] Lavrijsen, Wim TLP. "Optimizing python-based ROOT I/O with PyPy's tracing just-in-time compiler." In Journal of Physics: Conference Series, vol. 396, no. 5, p. 052046. IOP Publishing, 2012.
[9] Serge, Guelton. "Pythran: Crossing the Python Frontier." Computing in Science & Engineering 20, no. 2 (2018): 83.
[10] J. Diaz-Montes, C. Muñoz Caro, A. Niño, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era", IEEE Transactions on Parallel and Distributed Systems, vol. 23, pp. 1369-1386, 2012.
[11] B. Chamberlain, D. Callahan, H. Zima, "Parallel Programmability and the Chapel Language", The International Journal of High Performance Computing Applications, vol. 21, no. 3, pp. 291-312, 2007.
[12] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, "HPX: A Task Based Programming Model in a Global Address Space", Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, 2014.
[13] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice", ser. SC, 2014.
[14] C. F. Bolz, Meta-Tracing Just-In-Time Compilation for RPython, 2012.