iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 10 **Issue:** III **Month of publication:** March 2022

DOI: https://doi.org/10.22214/ijraset.2022.40949

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 10 Issue III Mar 2022- Available at www.ijraset.com

Design of a New Language Seeks Literature Survey

B R Gagan[1], Shivaprakash T[2], Thirumalai Shaktivel C[3], Vaishak P[4], Kushal Kumar B. N[5]
[1, 2, 3, 4]*B. E Student, Dept. of Computer Science, K. S Institute of Technology. Bengaluru, Karnataka*
[5]*Assistant Professor, Dept. of Computer Science, K. S Institute of Technology, Bengaluru, Karnataka*

Abstract: ***In a scientific study, computing is a must-have tool. In general, scientists have various difficulties, requirements, and views when it comes to computation, which need to be addressed by the programming language that they use, this cannot be satisfied by general-purpose languages. Also, researchers need to concentrate on the issue they are working on rather than the optimizations for the calculations, so instead of using a general-purpose language, if there exists a language whose compiler would take care of those optimizations, it would make their work easier and faster. This is a survey of the work undertaken to design the programming language and its compiler. The primary goal of this research is to examine the function of work, implementation strategy, steps taken for improving the performance, the procedure of benchmarking, and finally, the outcome of the papers studied. The survey's main conclusions are that: the most common language mentioned among the papers was Python which appears to be more popular among developers due to its simple syntax and library support for computing. On the other hand, Python lacks performance, to compensate for this performance issue, the community has developed tools like Cython, Numba, Pythran, etc, which can be used to speed up Python. Domain-specific languages such as Wolfram, Seq, and ELI highlighted various methods for overcoming problems. Some languages like Wolfram and ELI moved from interpreter to compiler to get the performance boost. Most of the compilers use LLVM as the backend for optimizations and code generation.***
Keywords: scientific computation, compiler, programming language

I. INTRODUCTION

Scientific Computing is a multidisciplinary field that contains many aspects of Computer Science, Applied Mathematics, and Engineering. Scientific Computing has emerged as the "third approach" for addressing fundamental research questions, alongside "theory" and "experiment." Many new researchers have joined in recent decades and are eager to contribute with their ideas. Researchers working in computational science generally encounter a difficulty i.e., primarily they have to concentrate on procedures like implementing, experimenting, and extracting information for solving the problem. At the same time, they have to convert this idea into programs, that include implementation, testing and debugging. While converting their problems into programs they should also concentrate on optimizing the program which is a time-consuming process. This suggests that an optimized high-performance computing language is required to assist researchers. To achieve this, we must first examine previous implementations and the work undertaken by the community to design one.

Before we proceed, let us understand what is a programming language and compiler: A programming language is a means of communicating with computers that consist of a set of grammatical rules (program) for instructing a computer to perform a specific task. The computer does not understand this program because it is written in a 'high-level' language; this is where a 'compiler' comes in. The compiler can convert a user-written program into machine code that computers can understand. In addition to these basic requirements, a good compiler should be able to compile faster to achieve good performance which increases efficiency, provide understandable error messages to aid developers in debugging the code, include runtime libraries, and also perform all necessary optimizations.

Why develop a new language for Scientific Computing? ALGOL, APL, Fortran, MATLAB, R, and other programming languages are used for scientific computation. Although these languages are better suited for computations than other 'General Purpose Languages,' some of them lack features such as readability, ease of understanding, and performance. That being said there is a need for a programming language for numerical and scientific computations with simple, readable syntax and high performance.

In general, developing a new language entails designing the syntax, building a compiler, and adding support for standard libraries. Creating a new language from scratch has fallen out of fashion; instead, the developer community offers a variety of useful tools ranging from parsing to code generation. Let us explore the various tools and methodologies used by different programming languages through this paper. In this context, the purpose of this paper is to facilitate the first step i.e., literature survey in the design of a new language.

©IJRASET: All Rights are Reserved

II. LITERATURE SURVEY

The following literature survey (each paper) is divided into 4 parts, namely:
1) Problem focus, 2) Implementation, 3) Performance, 4) Outcome.

Abdul Dakkak, Tom Wickham-Jones, et al., [2] proposed "The Design and Implementation of the Wolfram Language Compiler". This paper talks about 1) Constructing a Wolfram Language production compiler that enables easy scalability and simplifies the compilation of big programs. The constraints of the existing bytecode Wolform compiler are also addressed in this work. The compiler's feature set makes it a powerful and versatile tool for solving a wide range of technical and computation issues. 2) A typical compiler pipeline design is used to create this compiler, which is: Wolfram Source code is parsed and converted to Wolfram Language AST, then transferred to the untyped intermediate representation, then turned into typed-annotated compiler intermediate representation and finally generating the code. 3) An internal benchmark suite is used to evaluate the new Wolfram compiler's performance. They chose 7 easy benchmarks from the benchmark suite, such as Blur, Mandelbrot, QSort, Histogram, and Dot, to demonstrate the performance factors of the compiled operations. It outperforms similar compilers by 16 times in the Mandelbrot benchmark. 4) The Wolfram Language has a long history of helping people solve issues in areas including optimization, mathematics, and data science. Create a compiler that boosts programmer efficiency while delivering code that is as good as optimized C code. It demonstrates that careful compiler design preserves Wolfram Language productivity and expected behavior.

Ariya Shajii, Ibrahim Numanagić, et al., [4] proposed "Seq: A High-Performance Language for Bioinformatics". This paper talks about 1) A domain-specific language for computational biology (Bio-informatics) that combines Python's ease of use and productivity with performance comparable to C. 2) Seq is designed based on Python (statically-typed) syntax, with domain-specific data types and a complete reimplementation of all of Python's language functionalities and built-in services. For general-purpose optimizations, the Seq compiler uses an LLVM backend. Seq applications also employ a lightweight runtime library for memory allocation and I/O, with the Boehm garbage collector replacing CPython's reference counting. 3) Seq improves performance by 160 times over normal Python and by up to 7 times over C++. 4) Seq links the gap between biologists whose daily workflow entails quick prototyping and the development of new techniques and computationalists who want to build optimized code for a particular application.

Denis Merigoux, Raphaël Monat, et al., [5] proposed "A Modern Compiler for the French Tax Code". This paper discusses 1) Designing Mlang, an open-source compiler toolset aimed at replacing the current system for compiling the complex French Tax Code. Mlang is built on top of the DGFiP's architecture that has been reverse-engineered. 2) Clarifying M's semantics and proposing a new DSL–M++, to solve the language's inadequacies. Mlang takes an M syntax-code and an M++ script as input. Mlang is based on OCaml syntax and has about 9,000 code lines. The M files and M++ program are processed and converted into intermediate representations first. All of these intermediate representations are merged into a single backend intermediate representation (BIR). Finally, the backend translates BIR to C and Python. 3) Each of the M variable values were stored in an array (globally accessed) by the DGFiP while using the legacy code. The identical strategy is used in this paper and discovered that with -O1, it was nearly as quick as the legacy code. 4)The authors were able to transform legacy, confidential scripts into a retractable, general asset which can then be disseminated into nearly any programming environment using modern compiler building approaches.

Jeff Bezanson, Alan Edelman, et al., [6] proposed "Julia: A Fresh Approach to Numerical Computing". According to this paper, 1) Julia is an open-source, dynamic, interactive, compiled, and JIT compiler whose unique feature is its blend of performance and productivity, i.e., C-like performance and Ruby-like dynamism. Many of its features make it ideal for numerical and scientific computing. 2) Julia's core is written in Julia and C, FemtoLisp is used for parsing. Depending on the platform Julia runs on, the LLVM compiler infrastructure (uses C++) is utilized as the back end for generating 64-bit or 32-bit efficient machine code. Julia itself is used for designing the standard library. 3) In Binary STL benchmark: Julia is 2 times faster than C++ and 120 times faster than python.[1] 4) Julia was created to fulfill numerical computing needs, and it turns out that many people had the same desire.

Hanfeng Chen, Wai-Mee Ching, et al., [8] proposed "An ELI-to-C Compiler: Design, Implementation, and Performance". This paper discusses 1) The construction of a compiler that converts ELI to C, which is bootstrapped and optimized for good performance as C. All the built-in features and functionalities of the ELI language, are supported by this compiler. 2) This ELI compiler was designed in ELI. Firstly, it parses the syntax (top-down parser) and produces the parse tree on the front end. The shape and type information has also been included in the parse trees. Then, to transform a parse tree into C code, it uses the treewalk function in the backend. 3) Over the ELI interpreter, C code (generated using ECC) resulted in speedups ranging from roughly 2 to over 40 times. In terms of performance, ECC-generated code is often comparable to optimized C code. Benchmark for hotspots: It considerably enhances the ECC-produced code's performance, with speedups ranging from 1.4 to 1.16 times. 4) This study introduced a new ELI compiler that outperforms the interpreter.

©IJRASET: All Rights are Reserved

Hanwen Yin, Yun Pan, et al., [9] proposed "The Implementation of a Simple Smart Contract Language and Its Compiler Based on Ethereum Platform". This paper talks about 1) The typical smart Contract languages, such as Solidity and LLL, have complicated syntax and are difficult to learn and use for beginners. This paper describes the steps involved in creating a simple smart contract language, from language design to intermediate code execution. 2) The compiler designed in this paper can parse the Python code, and the target executing platform is the Ethereum virtual machine. The front-end of the compiler generates an AST, then the back-end processes this by converting the AST into target code that can be executed on the EVM. 3) Since this paper was in the exploratory stage, the compiler's performance was not demonstrated. 4) This paper creates a simple smart contract language, its compiler, and conducts an in-depth evaluation of each module's implementation. This compiler can implement some programs and simple smart contract instances while maintaining good grammar learnability.

Roshan Dathathri, Blagovesta Kostova, et al., [10] proposed "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation". This paper 1) Introduces Encrypted Vector Arithmetic is a programming language and IR for computing Fully-Homomorphic Encryption. EVA is also built to make things easy for domain-specific languages. EVA features an efficient compiler that targets the state-of-the-art SEAL package and delivers accurate, reliable, and speedy code. 2) EVA has a Python frontend that allows programmers to develop complex applications with minimum effort while hiding all technical information from them. 3) When comparing the performance of EVA with CHET (artificial neural network compiler), on average EVA beats its unaltered version by 5.3 times. 4) EVA includes domain-specific and auto-vectorizing compilers for working with encrypted information, as well as a powerful platform for a larger range of Fully-Homomorphic Encryption applications.

Pierrick Brunet, Serge Guelton§∗, et al., [12] proposed "Pythran: Enabling Static Optimization of Scientific Python Programs". This paper presents 1) Pythran is an efficient static compiler that converts implicit modules (statically typed) into parametric C++ code. The Pythran supports only a subset of Python syntax. It helps many of the Python language's high-level constructs in version 2.7. It also optimizes the compiler in a variety of ways. 2) In the Pythran compiler: The front-end translates Python source code into an AST, which is subsequently converted into a Pythran IR that is type-agnostic. On this IR, the middle-end performs different code optimizations. The backend functions in three stages: i) parametric C++ program is generated from Pythran's IR. ii) instantiating the yielded C++ code for the necessary types; iii) executing the produced C++ code into native one; Pythran additionally includes Numpy expression back-end optimizations and explicit parallelization using OpenMP-like interpretation. 3) Benchmark hyantes kernel (Numpy version): Pythran is 90 times quicker than Python. On the other hand, Pythran + OpenMP is 190 times quicker than Python. 4) Converting Python to C++ with the Pythran optimizer, compiler, and translator. Pythran, unlike other static Python compilers, uses several module-level or function-level studies to give many Python-centric or generic code optimizations. It also makes use of a C++ library that heavily relies on template programming to give an API that is comparable to a fraction of Python's built-in library

Siu Kwan Lam, Antoine Pitrou, et al., [13] proposed "Numba: A LLVM-based Python JIT Compiler". This paper talks about 1) Python code optimization for scientific computing, with a focus on NumPy arrays and functions. Users can annotate a performance-critical Python function for execution without having to rewrite the program in another language with Numba. 2) Numba is a CPython Just-in-Time compiler that runs functions one at a time. Numba uses the industry-standard LLVM compiler library to transform Python functions into efficient machine code at runtime. The compilation process begins by transforming Python source code into an intermediate representation, which is then used to perform type inference. This IR is reduced to LLVM, which subsequently generates machine code if the type of each value in the IR can be inferred. Numba assumes all values in the function are Python objects if it can't figure out the type of one of the items in the IR. In these instances, Numba must use CPython and depend on the Python environment for compilation. 3) Numba is more than 100 times faster than plain Python for this application when compared to the Wolfram model. In fact, converting simple Python code to C++ directly is slower than using Numba. The Numba version could be defeated with more optimization in C++. 4) Numba is a tool for Python that prioritizes performance, which is crucial in numerous numerical and scientific uses. Deferred loop specialization, array expression modification, and support for numerous back-ends are all included.

Eric Petit and Yohann Uguen [14] proposed "PyGA: A Python to FPGA Compiler Prototype". This paper discusses 1) PyGA is a compiler prototype that converts Python to FPGA, which is built on top of the Numba (Just In Time) Python compiler and the Intel FPGA SDK. PyGA makes it simple for any programmer to use an FPGA card as a Python booster. This prototype is designed for Python or more precisely for Scipy's Numpy arrays. 2) Implementation: When utilizing PyGA, the user calls his method without any annotations and with the correct arguments. The tool then obtains the Numba generated LLVM IR by using Numba's type inference to obtain the function's signature. Using the function's signature, an OpenCL wrapper is created with the right types and parameters. We get the associated LLVM IR from the wrapper.

This Numba's IR is then transformed to Intel's IR using Numba's LLVM-IR module. The LLVM IR of the updated wrapper can then be executed into an FPGA stream of bits and transformed into the FPGA. Meanwhile, the signature of the function is used to generate the host code. Finally, Python executes the program by invoking the host code. 3) Deriche edge detector (algorithm for benchmark): On the CPU, the PyGA kernel is roughly 8 times slower than the Numba JIT version, but 125 times quicker than the Python interpreter. 4) HLS technologies were used to investigate the possibility of accelerating compute-intensive kernels in interpreted languages. The proposed solution is to create an offline FPGA stream of bits that are used for the program kernel, which would subsequently be called during the Python application compilation.

TABLE I. Comparison Among Some Popular Languages

Languages	Compilation	Domain [4]	High Performance	Less Complexity	Optimization w.r.t Scientific Computation
C++	Ahead of time	General	✓	✗ [7]	✓
Python	Interpreted [4]	General	✗ [11]	✓	✗ [3]
Seq [4]	Ahead of time	Bioinformatics	✓	✓	✗ [4]

TABLE 1 gives a visual comparison between the pros and cons of different programming languages like C++, Python, and Seq. Among these languages, we can see that C++ is a complex language to learn and understand, Python does not deliver high performance and Seq does not have optimizations for scientific computations.

III. CONCLUSION

From the above findings, we conclude that: currently both domain-specific languages and general-purpose languages are used in scientific computations. Among them, most people, when asked to choose a language, choose popular languages like Python for its exceptional friendliness and ease of use. While domain-specific languages (Wolfram, ELI) do provide optimizations for scientific computing. Due to them being unpopular, most people do not prefer them. Instead, they move towards the trend (like Python). Popular general-purpose languages like C++ and Python are used in Scientific Computation; they use libraries to achieve this, which hinders the performance. New technologies arise every day and continuing to express these new technologies in the old way of thinking might not be adequate. From all these, we can infer that there is a need for a domain-specific language that is exclusively designed for their Domain. The main key takeaways are that the programming language must be easy with simple syntax. Compilers must have speedy compilation (which includes faster parsing, effective conversions, and well-organized package linking) and improved optimizations to achieve a good performance. Most of the programming languages examined here use LLVM as a backend, which can be used for both code generation and optimization.

REFERENCES

[1] Aaron Ang, October 2018, "STL Benchmark Comparison: C++ vs. Julia", Accessed on: January 2022 Available: [online] https://aaronang.github.io/2018/stl-benchmark-comparison-cpp-vs-julia/

[2] Abdul Dakkak, Tom Wickham-Jones, and Wen-mei Hwu. 2020. "The Design and Implementation of the Wolfram Language Compiler." In Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22-26, 2020, San Diego, CA, USA. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3368826.3377913

[3] Antonio Cuni "High-performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages" July 2010

[4] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. 2019. "Seq: A High-Performance Language for Bioinformatics." Proc. ACM Program. Lang. 3, OOPSLA, Article 125 (October 2019), 29 pages. https://doi.org/10.1145/3360551

[5] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021. "A Modern Compiler for the French Tax Code". In Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2-3, 2021, Virtual, Republic of Korea. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446804.3446850

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah. 2017. "Julia: A Fresh Approach to Numerical Computing" November 2017 [online] https://julialang.org/assets/research/julia-fresh-approach-BEKS.pdf

[7] Joab Jackson "Google executive frustrated by Java, C++ complexity" July 23, 2010

[8] Hanfeng Chen Wai-Mee Ching Laurie Hendren "An ELI-to-C Compiler: Design, Implementation, and Performance" June 2017 https://dl.acm.org/doi/abs/10.1145/3091966.3091969

[9] Hanwen Yin, Yun Pan, Han Jiang. 2020. “The Implementation of Simple Smart Contract Language and Its Compiler Based on Ethereum Platform” August 2020. https://doi.org/10.1145/3418994.3419010

[10] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. “EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation”. In Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI ’20), June 15–20, 2020, London, UK. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386023

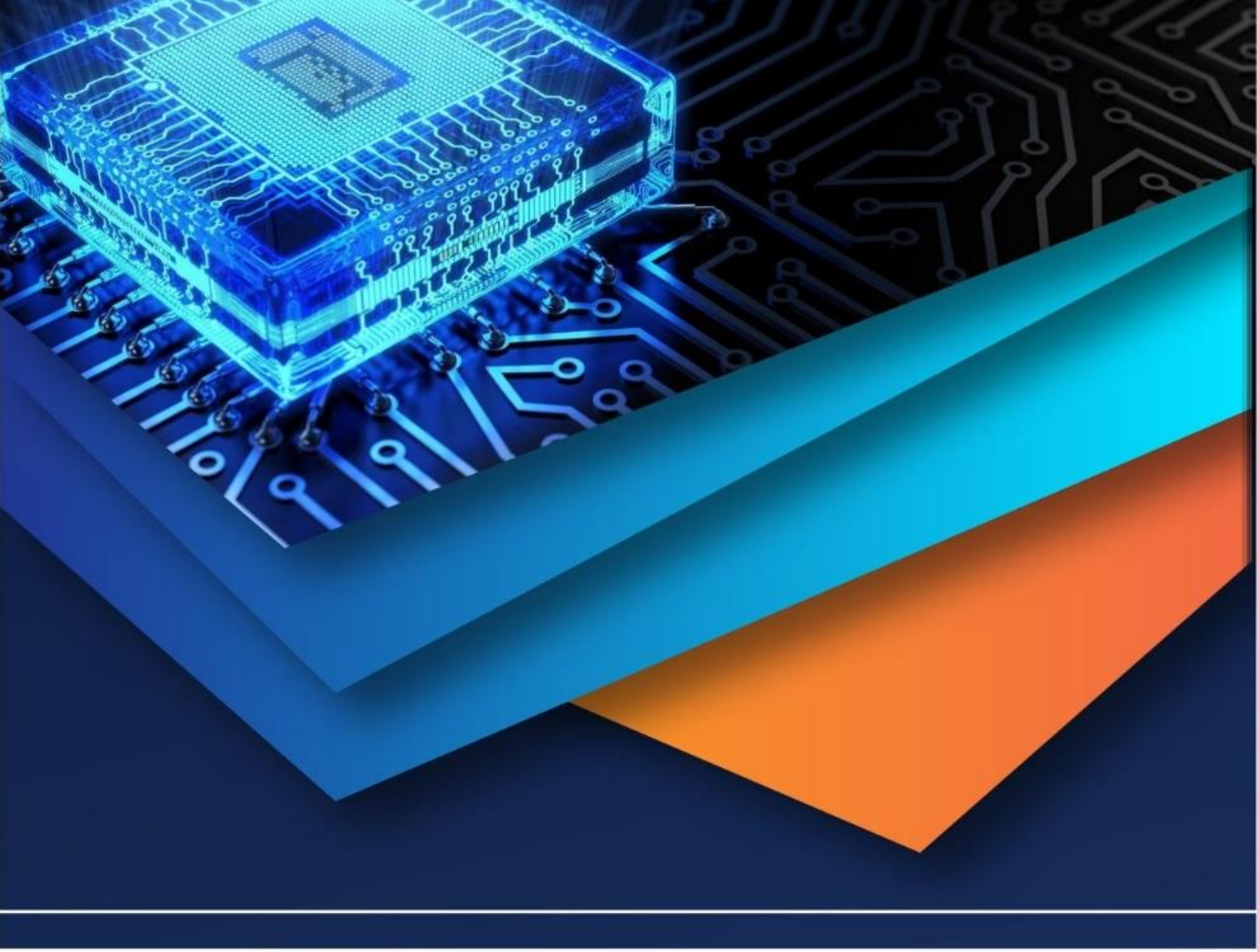
[11] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jacome, Cunha, Jo~ao Paulo Fernandes, Jo~ao Saraiva “Ranking Programming Languages by Energy Efficiency” January 4, 2021

[12] Serge Guelton§∗ , Pierrick Brunet‡, Alan Raynaud‡, Adrien Merlini‡, Mehdi Amini “Pythran: Enabling Static Optimization of Scientific Python Programs” January 2015 Available: [online] http://conference.scipy.org/proceedings/scipy2013/pdfs/guelton.pdf

[13] Siu Kwan Lam, Antoine Pitrou, Stanley Seibert “Numba: A LLVM-based Python JIT Compiler” November 15 - 20 2015 http://dx.doi.org/10.1145/2833157.2833162

[14] Yohann Uguen and Eric Petit. 2018. “PyGA: A Python to FPGA Compiler Prototype.” In Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems (AI-SEPS ’18), November 6, 2018, Boston, MA, USA. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3281070.3281072

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)