Transpiling Python to Rust for Optimized Performance

✉ Henri Lunnikivi[0000−0003−4817−2939], Kai Jylkkä[0000−0002−8412−5825], and Timo Hämäläinen[0000−0002−7867−0800]

Tampere University, Tampere, Finland `tau@tuni.fi`
http://tuni.fi
{`henri.lunnikivi,kai.jylkka,timo.hamalainen`}`@tuni.fi`

Abstract. PYTHON has become the de facto programming language in machine learning and scientific computing, but high performance implementations are challenging to create especially for embedded systems with limited resources. We address the challenge of compiling and optimizing PYTHON source code for a low-level target by introducing RUST as an intermediate source code step. We show that pre-existing PYTHON implementations that depend on optimized libraries, such as `NumPy`, can be transpiled to RUST semi-automatically, with potential for further automation. We use two representative test cases, Black–Scholes for financial options pricing and robot trajectory optimization. The results show up to 12× speedup and 1.5× less memory use on PC, and the same performance but 4× less memory use on an ARM processor on PYNQ SoC FPGA. We also present a comprehensive list of factors for the process, to show the potential for fully automated transpilation. Our findings are generally applicable and can improve the performance of many PYTHON applications while keeping their easy programmability.

Keywords: Python · Rust · embedded computing · transpilation.

1 Introduction

PYTHON is the most popular language in Machine Learning (ML) and is rapidly replacing others in signal processing and scientific computing. Yet, its execution time and memory consumption are not workable in constrained systems. Additionally, energy awareness in ML computing in general is becoming a big concern. Because of the wide adoption of PYTHON, an improvement in process will have multiplied effects in the large scale, which makes the issue important. This paper addresses the challenge of how a pre-existing PYTHON implementation could be automatically compiled to a high performance and low energy implementation while retaining easy programmability.

In a typical use case, a high-level language like PYTHON is used to write domain-specific steering code that then relies on optimized libraries. Key challenges of optimizing to a target are cross-library memory optimization, platform specific optimization, and parallelization. Cross-library optimization is required

because there is a risk of needless data movement at function and library boundaries of which a typical user is not aware. Platform specific optimization is required to take advantage of the particular capabilities of the platform, and for many platforms, extracting parallelism out of the programming model is a necessary part of that.

Many of the approaches towards compiling PYTHON into low-level machine code work better for cloud-like computer infrastructure. However, large runtime systems and current end-to-end optimization technologies do not work as well for embedded devices. Runtime systems are computationally unaffordable and end-to-end optimization technologies create additional challenges for understanding application performance[14] or require changes to used libraries (eg. [3, 5, 9, 10, 15–17]).

Our proposal is based on RUST[1] as an intermediate source code step. PYTHON and RUST are both high-level languages that allow problems to be modelled using domain-specific idioms. Expressions of idioms in PYTHON can be converted to expressions in RUST at least semi-automatically, and at least one open-source implementation[8] exists. The benefit is that source-to-source transpiled code retains readability, which allows for further changes, experimentation, and optimization.

To our best knowledge, this is the first publicly available, reproducible, and systematic study on translation and transpilation of PYTHON to RUST for optimized implementations. We have two research questions: 1) How much speedup can be gained by transpiling? 2) How automatable is the process? Our contributions are as follows:

1. A workflow for quick optimization of a high-level description of an algorithm for a low-end target with support for parallel execution
2. A measurement on how functionally equivalent high-level implementations in PYTHON and RUST might perform with respect to each other
3. Evaluation of feasibility of `pyrs`[8] for transpilation
4. Propositions for further automation of transpilation
5. Two use cases for performance analysis: Financial data analytics and motion trajectory optimization for robots

This paper is organized as follows. We will introduce related work in Section 2, and our approach in Section 3. We will describe the two use cases in Section 4 and results in Section 5. We will give our conclusions and future work in Section 6.

2 Related Work

Most software is built from layers upon layers of other software. An application is described in an expressive, high-level language, using domain-specific idioms and libraries relevant to the problem being solved. The application and its libraries must then be translated to a hardware-specific stream of instructions by

---

[1] RUST. https://www.rust-lang.org/

compilers, interpreters, or both. A popular workflow involves writing domain-specific steering code in PYTHON and then relying on one of a myriad of techniques to ensure that the program performs at the required level on the target platform. Techniques include use of optimized libraries[1, 12, 13, 21], optimizing compilers[2–5, 9, 10, 15–17], and runtimes with just-in-time (JIT) compilation[3, 5, 9–11, 15–18].

Our approach is based on compiled RUST, which allows optimization of the code as a whole including the libraries. RUST is a recent, high-level alternative for performance-oriented programming, which allows a path from a high level of abstraction to optimized machine code. The approach is similar to that taken by the developers of `Cython`[2] and `MicroPython`[3]. `Cython` allows an improvement in performance by using a static compiler via C, while `MicroPython` is a Python runtime, optimized for microcontrollers. Our approach differs in that the transpiled code is compiled to a runtimeless, fully optimizable native executable without reliance on an interpreter. Our method also resembles that of the `C2Rust` tool[4], which translates C into semantically matching RUST. Our approach differs in both the source language and the fact that `C2Rust` generates RUST code using the *unsafe* subset of the RUST language, while our approach is focused on *safe* RUST, preserving the memory safety of the PYTHON implementation. We use an experimental open-source PYTHON tool called `pyrs`[8] to convert PYTHON syntax into RUST.

Compared to optimized PYTHON, RUST offers similar capabilities but with a smaller degree of separation between high-level development processes and native tooling. RUST seems to allow combining optimization processes, such as use of domain specific, optimized, native libraries and cross-platform algorithm development, with low-level native tooling such as standard Linux tools `perf`[5], `valgrind`[6], and the LLVM debugger[7]. As an LLVM-compiled language, RUST allows similar performance enhancements as an end-to-end optimizer, a staple in deployment of PYTHON. This similar optimization is achieved with compiler enforced pointer aliasing rules[8]. Additionally, RUST allows extended utilities for embedded development, such as the `no_std`[9] feature that allows the compiler to not assume the existence of heap memory, networking support or threads.

3 Methods

We summarize first the environment for conducting this work. The tools used in transpilation are `pyrs`[8], `MonkeyType`[6], and `IntelliJ IDEA`[7]. RUST 1.31 is

---

[2] `Cython`. https://github.com/cython/cython

[3] `MicroPython`. https://github.com/micropython/micropython

[4] `C2Rust`. https://github.com/immunant/c2rust

[5] `perf`. https://perf.wiki.kernel.org/

[6] `valgrind`. https://valgrind.org/

[7] The LLDB Debugger. https://lldb.llvm.org/

[8] Aliasing in Rust. https://doc.rust-lang.org/nomicon/aliasing.html

[9] RUST: No stdlib. https://doc.rust-lang.org/1.7.0/book/no-stdlib.html

used to compile the resultant RUST program using the package manager `cargo`[10] and the `rustc` compiler. The two chosen use cases are a simple Black–Scholes model and an advanced algorithm for trajectory optimization[20] abbreviated as Motion Planning. The performance was tested on a Windows PC with an AMD Ryzen 7 3700X processor and PYNQ Z1 All Programmable System-on-Chip (APSoC)[11]. Additionally, as proof-of-concept, the transpiled programs were cross-compiled to Snapdragon 835 mobile Hardware Development Kit[12], which to our best knowledge does not natively support PYTHON.

The PYNQ processing system includes a 650 MHz dual-core Cortex-A9 processor and 512 MB DDR3 memory. The device was running a Ubuntu 18.04 PYNQ Linux OS including a PYTHON 3.6 interpreter. We installed the RUST toolchain[13] on the device and compiled the source code used for the tests on the device. We also successfully verified our transpiling method on Snapdragon 835 development kit running with Android 7.1.2 OS. We cross-compiled RUST source code on Ubuntu 16.04 host PC and used `Android Debug Bridge` to run compiled binaries on the device.

3.1 Transpilation Workflow

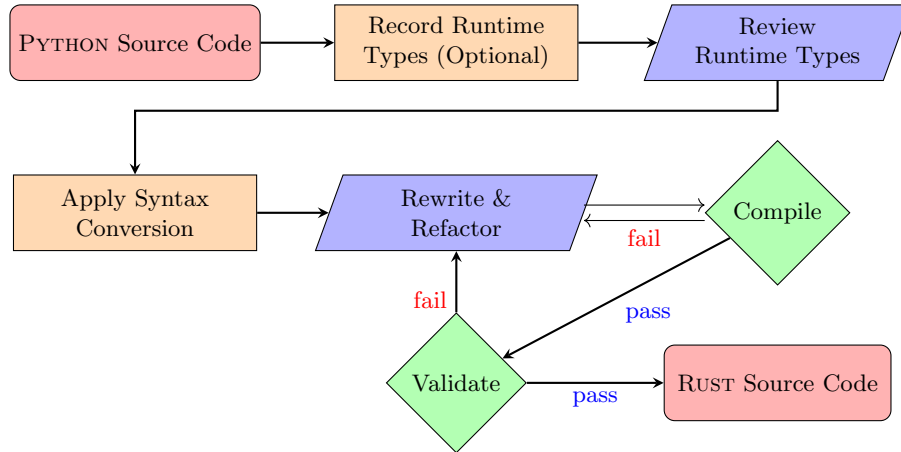PYTHON source code is transpiled to RUST. The general outline of the transpilation process is depicted in Figure 1.

Fig. 1. Transpiling Python to Rust.

---

[10] `cargo` the RUST package manager. https://doc.rust-lang.org/cargo/

[11] PYNQ Z1.
https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start

[12] Snapdragon 835 HDK. https://developer.qualcomm.com/hardware/snapdragon-835-hdk

[13] `rustup` the RUST toolchain installer. https://github.com/rust-lang/rustup

Transpilation comprises the following phases: optional application of runtime types, syntax conversion, manual refactoring, and validation testing. Before syntax conversion, the PYTHON program can be run against a tool[14] to obtain information about the concrete types used by the application. Types recorded this way allow for more information to be available in the later steps of transpilation, but may be unnecessarily constraining.

Next, the syntax conversion is applied using a publicly available automatic syntax converter that creates a PYTHON Abstract Syntax Tree (AST) from the PYTHON source code. The PYTHON AST is automatically made into RUST source code by visiting each AST node using the visitor pattern[15] and outputting the equivalent RUST code. Syntax conversion allows for rough, automatic transformation of PYTHON syntax to almost equivalent RUST. After syntax conversion, the program is unlikely to immediately compile using the RUST compiler and must be manually edited with help from the RUST compiler. Once the compiler accepts the source code, it must be further tested to validate that the code is functionally equivalent to the original. Once the program is validated, the program source code can be further edited, optimized, and compiled as a RUST program.

The performance of the compiled RUST code can be measured either by native RUST benchmarks or by dispatching library functions via its C foreign function interface (CFFI) via a PYTHON 3.8.1 interpreter. Execution on PC is measured using the CFFI and the PYTHON interpreter for a fair comparison against the original `SciPy` and `BLAS` optimized PYTHON code. Deployability on embedded targets is verified by natively compiling the program on PYNQ and by cross-compiling to Snapdragon using the cross-compilation framework `cross`[16].

4   Use Cases

Two distinct use cases were chosen for examining the difficulty and outcomes of transpiling and its feasibility for embedded implementations. The first use case is a simple Black–Scholes model for options pricing in financial markets. The second is a recent, complex algorithm for motion planning in robotics[20]. Black–Scholes is a collection of established small benchmark methods that can each be represented in around 5 lines of PYTHON with `SciPy`. Motion Planning, on the other hand, is a novel and experimental algorithm in robotics, spanning 437 lines of PYTHON with `NumPy` as formatted by `autopep8`, omitting blank lines and comments. We transpiled it *as is* from the developers during development of the algorithm.

4.1   Black–Scholes Model

Black–Scholes is a well-known model for the dynamics of financial markets. It was selected for its readability, compact source code size and good coverage of

---

[14] `MonkeyType`. https://github.com/Instagram/MonkeyType
[15] Visitor Pattern. https://en.wikipedia.org/wiki/Visitor_pattern
[16] `cross`. https://github.com/rust-embedded/cross

results in other studies[9, 15, 16, 18]. We implemented[17] the model in Python using NumPy based on an online source[19], and added validation tests. A notable feature of the implementation is that because of NumPy, it works the same for both scalar and array inputs. Our transpiled sources are publicly available[18].

The transpilation was conducted as explained earlier in Section 3. After automatic syntax conversion, compiler-assisted library mapping was straightforward. NumPy functions generally mapped to Rust standard library functions of similar names and signatures. The more complex SciPy-call to a cumulative distribution function required a manual implementation using statrs, though a direct expression-to-expression mapping was found to exist. Automatic syntax conversion introduced an incorrect calculation order issue that was captured by validation tests and corrected. Potential, automatic solutions to the described issues are discussed in Section 5.

The transpiled Rust source code and its dependencies were compiled natively on PC and PYNQ, and cross-compiled to Qualcomm Snapdragon using cross. Fixed inputs produce same results on all platforms.

4.2 Motion Planning

Multi-convex path constrained trajectory optimization for mobile manipulators[20], or Motion Planning, is a recent proposition for an algorithm in robotics, written in Python and NumPy. Motion Planning allows a robot, such as a mobile manipulator, to find a path through space while respecting kinematic constraints and avoiding collisions [20]. Key qualities of the algorithm are its parallel nature and the suitability for experimentation in the physical world. The physical aspect makes it a good candidate for optimizing and deploying on an embedded target. Singh et al. note that prototyping the algorithm in a low-level language might improve computation time[20].

The original Motion Planning implementation uses Basic Linear Algebra Subprograms (BLAS) specification to prescribe optimizations via NumPy. Our transpiled implementation maps NumPy functions to Rust community library ndarray[19]. In our experiments, the BLAS optimizations are implemented by linking to Intel MKL on PC and to OpenBLAS[20] on PYNQ. The total runtime of the original implementation is a couple of seconds on a multicore desktop PC when allowed to run for 100 iterations. In the experiments, the parameters to the algorithm are built-into the library, which reduces reliability of the measurements for both the Python and the transpiled Rust implementations. Measurements were taken with careful examination of allocations and verification of produced outputs.

After automatic syntax conversion, we used regular expressions extensively to fix remaining errors and library references. Multiple manual implementations

---

[17] Black–Scholes based on [19]. https://github.com/hegza/black-scholes-py

[18] Transpiled Black–Scholes. https://github.com/hegza/black-scholes-transpiled-rs

[19] ndarray. https://github.com/rust-ndarray/ndarray

[20] OpenBLAS. https://www.openblas.net/

of expressions (< 1 % of all expressions) were required though each was straight-forward with output from the RUST compiler. Validation tests revealed that the original PYTHON implementation converged slightly faster than the RUST implementation. This is possibly due to different linear algebra implementations in the respective libraries. Slower convergence was not taken into account in performance measurements.

The transpiled RUST source code and its dependencies were compiled natively on PC and PYNQ, and cross-compiled to Qualcomm Snapdragon using `cross`. Results converge on the same values with fixed inputs on all platforms.

5   Results and Analysis

We consider the results of transpilation in wall clock time, memory consumption, and complexity in terms of issues encountered and regularity for potential automation. We also consider what processes transpiled sources now allow. Lines of code is not considered a feasible measure because the number of lines is almost the same, with differences being due to the differing formatters of each language.

5.1   Performance

The performance of PYTHON implementations on PC is measured with `timeit` library, which avoids the overhead from PYTHON garbage collection. RUST implementations are measured with `criterion.rs`[21], a community library for RUST benchmarks. Memory use is measured with `valgrind --tool=massif`. `Intel MKL` and `OpenBLAS` are dynamically linked to the runtimes where applicable. The Black–Scholes model was calculated for 72 million *put options*[22], which were loaded from a file for the RUST implementation to prevent excessive LLVM optimizations. Motion Planning was run for 100 iterations. The performance of the Black–Scholes model is as follows:

Table 1. Execution profile of Black–Scholes on PC (allocations included).

Black–Scholes	Python (MKL)	Rust (native)
Execution time	27.29 s	11.70 s
Peak memory consumption	9.372 GB	3.456 GB

Table 1 shows that the native RUST implementation uses less memory and runs faster. Output from Valgrind's `massif` seems to suggest that the PYTHON implementation reserves the input and output arrays arrays 2–3 times. Memory use of both measurements seems realistic, as the theoretical memory consumption of the Black–Scholes algorithm for this use case is

---

[21] `criterion.rs`. https://github.com/bheisler/criterion.rs

[22] https://en.wikipedia.org/wiki/Put_option

$$\text{floating\_point\_width} \times \text{number\_of\_options}$$
$$\times (\text{number\_of\_inputs} + \text{number\_of\_outputs})$$
$$= 8 \text{ bytes} \times 72 \text{ million} \times (5 + 1)$$
$$= 3.456 \; GB \text{ (decimal)}.$$

The result presented in Table 1 shows that a naively transpiled Python / `NumPy` implementation may perform equally well as – or better than – the original implementation in memory limited scenarios. Further investigation reveals that the Rust implementation links statically into `matrixmultiply`[23], which is a native Rust implementation of matrix multiplication. This enables full memory optimization via LLVM. Static linking of `Intel MKL` is possible for both implementations, but this is not the default behavior of either of the toolchains and requires manual work that is not portable across implementations on our chosen devices.

The performance results for Motion Planning are presented in Tables 2, 3, and 4.

Table 2. Execution profile of Motion Planning on PC using `Intel MKL`.

Motion Planning	Python	Rust (CFFI)	Rust
Execution time	6.44 s	0.671 s	0.532 s
Peak memory consumption	9.40 MB	8.45 MB	6.47 MB

Table 3. Execution profile of Motion Planning on PC using `OpenBLAS`.

Motion Planning	Python	Rust
Execution time	4.10 s	2.80 s
Peak memory consumption	8.85 MB	1.93 MB

Table 2 shows that the transpiled version of the `Intel MKL` accelerated implementation speeds up $12.1\times$. All `Intel MKL` implementations are parallel and speedup is likely to be mainly gained by `LLVM` optimizations, which are better at considering all of the code compared to Python steered calls to functions. Dispatching the transpiled Motion Planning algorithm from a Python interpreter via `CFFI` shows increased memory use due to the interpreter but runs faster than accelerated Python. Memory use improves by a factor of $1.45\times$. Table 3 shows that the transpiled version speeds up $1.46 \times$ when linked against `OpenBLAS`, with a $4.2\times$ improvement in memory consumption. Further investigation with Python tooling[24] reveals that most of the memory used by the

---

[23] `matrixmultiply`. https://github.com/bluss/matrixmultiply
[24] `tracemalloc`. https://docs.python.org/3/library/tracemalloc.html

PYTHON implementations is due to the loaded PYTHON runtime and libraries, while runtime workload allocates 1.99 MB of memory. The RUST implementation slightly improves in peak memory consumption for runtime data with 1.93 MB allocated.

Table 4. Execution profile of Motion Planning on PYNQ using `OpenBLAS`.

Motion Planning	Python	Rust (CFFI)	Rust
Execution time	70.5 s	70.3 s	69.0 s
Peak memory consumption	6.9 MB	2.7 MB	1.7 MB

Table 4 shows that the `OpenBLAS` accelerated implementation runs equally fast on PYNQ when executed via either language. The higher memory consumption of the PYTHON implementation is partially explained by the included interpreter. The transpiled native RUST implementations can also be cross-compiled to Qualcomm Snapdragon. All implementations use more memory than what is necessary due to the benefits caching can provide in some circumstances. However, the right trade-off between execution time and memory consumption is important. RUST implementations generally both run as fast or faster as PYTHON implementations and consume less memory.

5.2 Transpilation Regularity and Automation

The introduced method was applicable as-is and numerous automatable conversions were identified. The used automatic syntax conversion tool (`pyrs`) introduced a calculation order error in the Black–Scholes use case. The calculation order could be more accurately traced between the languages by introducing a step for converting the PYTHON AST to a RUST AST, as shown in Figure 2.
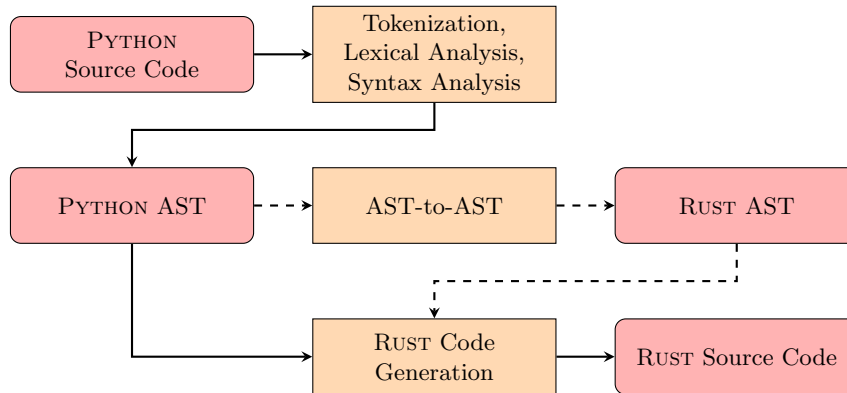

Fig. 2. Proposed AST-to-AST transpilation step (dashed arrows).

Creating such an AST-to-AST transpiler would require an implementation of a transformation from each source AST node to each target AST node. A PYTHON AST implementation in RUST is available via community library `RustPython`[25] and a RUST AST implementation in RUST is available via community library `syn`[26].

In our use cases, PYTHON library references could be converted to RUST library references in a highly regular fashion using a regex. A more sophisticated approach would be to use an AST aware transpiler to correctly map the method calls from source to target. Method calls can be mapped from a function body expression in PYTHON into a RUST path segment, and the parameters can be mapped expression-by-expression. In developed domain-specific libraries, a single declarative mapping for functions in source language to functions in target language may exist, and could be maintained via tooling to create a persistent mapping if desirable.

Issues arising from syntactic differences between the languages were all found to be automated, or automatable. However, in addition to syntax translation, conversion from PYTHON to RUST requires accounting for semantic differences. RUST ownership semantics can be at least partially accounted for by preferring use-by-reference for read-only bindings and by rebinding variable data for each use, trusting the compiler to optimize. PYTHON and RUST variable scopes do not match 1:1 and need to be corrected by hand, though these issues were found to be rare in our use cases with one semantic scope issue in Black–Scholes and none in Motion Planning. All found issues were detectable at compile-time with associated error messages, apart from the singular incorrect calculation order error in Black–Scholes. All problems were solvable via human intervention, as demonstrated by the use cases. The vast majority of issues were automated via `pyrs`, or could have been automatable.

5.3   Implications of Transpiled Source

The transpiled source code is easily programmable and compiles down to a high performance implementation that matches the native execution model of the platform, provided that an LLVM backend exists for the platform. The transpiled code adheres to RUST semantics and the language's type system. A notable example is RUST semantics regarding pointer aliasing, which allows LLVM optimizations such as loop vectorization and constant propagation. The type system guarantees the absence of race conditions in parallel execution, which allows additional paths for multicore optimization.

As the RUST source code compiles directly to target assembly without a managing runtime, its source code can be easily annotated with assembly or LLVM-IR at function level with tooling such as `cargo-asm`[27]. This is useful for comparing implementations for optimization. Profiling on target is enabled by tools

---

[25] `RustPython`. https://github.com/RustPython/RustPython

[26] `syn`. https://docs.rs/syn/1.0.16/syn/

[27] cargo-asm. https://github.com/gnzlbg/cargo-asm

such as `perf`, that allows the programmer to determine where in the code the processor spends most time. For instance, in combination with `FlameGraph`[28], profiling information generated and annotated with `perf` can be used to determine if the main workload spends time allocating heap memory that could be pre-allocated, or if the time is spent in an optimized library like `Intel MKL`. Direct access to profiling information like this allows the programmer to identify bottlenecks and make informed decisions about what to optimize next.

In addition to hardware oriented tooling, use of RUST allows the programmer to rely on type system guarantees when producing parallel implementations. In fact, the inner loop of Motion Planning is trivially convertible to a data-parallel implementation by rewriting the loop expression in terms of a parallel iterator as provided by the community library `rayon`[29], though this seems to provide no significant advantage over `Intel MKL` enabled data-parallelism in this workload.

6 Conclusions

Transpiled implementations are generally faster than accelerated PYTHON implementations up to an order of magnitude for the more complex use case. Transpilation seems automatable and generally applicable. An almost 1:1 syntactic mapping from expression to expression exists between PYTHON and RUST, and could be applied with an AST-to-AST transpiler. Library references can be mapped out and applied in a highly regular fashion. Issues arising from differing type systems and borrow checking semantics cannot be solved at the level of syntax conversion and another approach is required.

Transpiled implementations can be further developed and optimized. The implementation of the Black–Scholes model can be further accelerated by linking to `Intel MKL` statically, and bottlenecks in the motion planning algorithm can be investigated. The motion planning algorithm could also be made more portable and lighter on memory by using native RUST matrix solvers in place of `BLAS`, or faster by linking statically to `Intel MKL`.

Our future work focuses on comparing transpiled implementations to end-to-end optimized implementations, and improving the automation of transpilation. Further advancements in the transpilation process for such applications are certainly available. AST-to-AST transpilation allows improved control over syntax transformation. Library mappings can be defined declaratively and automated.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). pp. 265–283 (2016)

---

[28] FlameGraph. https://github.com/brendangregg/FlameGraph
[29] `rayon`. https://github.com/rayon-rs/rayon

2. Behnel, S., Bradshaw, R., Seljebotn, D.S., Ewing, G., et al.: Cython: C-extensions for python. published (2008)
3. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Wanderman-Milne, S.: JAX: Composable transformations of Python+NumPy programs (2018), http://github.com/google/jax
4. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al.: {TVM}: An automated end-to-end optimizing compiler for deep learning. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). pp. 578–594 (2018)
5. Frostig, R., Johnson, M.J., Leary, C.: Compiling machine learning programs via high-level tracing. Systems for Machine Learning (2018)
6. Instagram: (2020), https://github.com/Instagram/MonkeyType
7. JetBrains: (2020), https://www.jetbrains.com/idea/
8. Konchunas, J.: Python to rust transpiler (2020), github.com/konchunas/pyrs
9. Kristensen, M.R., Lund, S.A., Blum, T., Skovhede, K., Vinter, B.: Bohrium: unmodified numpy code on cpu, gpu, and cluster. In: 4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13) (2013)
10. Kristensen, M.R., Lund, S.A., Blum, T., Skovhede, K., Vinter, B.: Bohrium: a virtual machine approach to portable parallelism. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. pp. 312–321. IEEE (2014)
11. Leary, C., Wang, T.: Xla: Tensorflow, compiled. TensorFlow Dev Summit (2017)
12. Maclaurin, D., Duvenaud, D., Adams, R.P.: Autograd: Effortless gradients in numpy. In: ICML 2015 AutoML Workshop. vol. 238 (2015)
13. Oliphant, T., Peterson, P., Eric, J.: Scipy.org (2001), https://www.scipy.org/
14. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.G.: Making sense of performance in data analytics frameworks. In: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15). pp. 293–307 (2015)
15. Palkar, S., Thomas, J., Narayanan, D., Shanbhag, A., Palamuttam, R., Pirk, H., Schwarzkopf, M., Amarasinghe, S., Madden, S., Zaharia, M.: Weld: Rethinking the interface between data-intensive applications. arXiv preprint arXiv:1709.06416 (2017)
16. Palkar, S., Thomas, J., Narayanan, D., Thaker, P., Palamuttam, R., Negi, P., Shanbhag, A., Schwarzkopf, M., Pirk, H., Amarasinghe, S., et al.: Evaluating end-to-end optimization for data analytics applications in weld. Proceedings of the VLDB Endowment **11**(9), 1002–1015 (2018)
17. Palkar, S., Thomas, J.J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., Zaharia, M., InfoLab, S.: Weld: A common runtime for high performance data analytics. In: Conference on Innovative Data Systems Research (CIDR) (2017)
18. Palkar, S., Zaharia, M.: Optimizing data-intensive computations in existing libraries with split annotations. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 291–305 (2019)
19. Schlegel, A.: Black-scholes formula and python implementation (January 2018), https://aaronschlegel.me/black-scholes-formula-python.html
20. Singh, A.K., Ahonen, A., Ghabcheloo, R., Muller, A.: Inducing multi-convexity in path constrained trajectory optimization for mobile manipulators. arXiv preprint arXiv:1904.09780 (2019)
21. Walt, S.v.d., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering **13**(2), 22–30 (2011)