DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. Reference herein to any social initiative (including but not limited to Diversity, Equity, and Inclusion (DEI); Community Benefits Plans (CBP); Justice 40; etc.) is made by the Author independent of any current requirement by the United States Government and does not constitute or imply endorsement, recommendation, or support by the United States Government or any agency thereof.

Improving Runtime Performance of Tensor Computations using Rust From Python

Kimmie Harding* Daniel M. Dunlavy†

Abstract

In this work, we investigate improving the runtime performance of key computational kernels in the Python Tensor Toolbox (`pyttb`), a package for analyzing tensor data across a wide variety of applications. Recent runtime performance improvements have been demonstrated using Rust, a compiled language, from Python via extension modules leveraging the Python C API—e.g., web applications, data parsing, data validation, etc. Using this same approach, we study the runtime performance of key tensor kernels of increasing complexity, from simple kernels involving sums of products over data accessed through single and nested loops to more advanced tensor multiplication kernels that are key in low-rank tensor decomposition and tensor regression algorithms. In numerical experiments involving synthetically generated tensor data of various sizes and these tensor kernels, we demonstrate consistent improvements in runtime performance when using Rust from Python over 1) using Python alone, 2) using Python and the `Numba` just-in-time Python compiler (for loop-based kernels), and 3) using the `NumPy` Python package for scientific computing (for `pyttb` kernels).

1 Introduction

Tensors, or multi-dimensional (d-way) arrays [2], can be used to represent complex relationships in a variety of data analysis applications, such as deep learning [15], quantum computing [12], quantum chemistry [9], signal processing [24], neuroscience [17], scientific computing [7], and others. Several key tensor algorithms and operations, such as the sparse tensor times vector (TTV) product and sparse matricized tensor times Khatri-Rao product (MTTKRP), require computation that scales exponentially with d. Therefore, improving the runtime performance of such computational kernels and other tensor operations is critical for effective tensor data analysis.

The Python Tensor Toolbox, `pyttb` [4], is a recently developed software package for tensor operations. In the work presented here, we explore the possibility of improving runtime performance of `pyttb` on CPUs by implementing key tensor kernels in the compiled language Rust [8] that are then called from Python. Several strategies for improving runtime performance in Python include leveraging compiled-code extensions through the Python Foreign Function Interface (FFI) [6] and just-in-time (JIT) compilation [10], and we compare several such approaches in the numerical experiments presented below. We focus on using Rust from Python via the Python FFI, as this approach has shown improved performance for a variety of

*New Jersey Institute of Technology (`kah46@njit.edu`); Sandia National Laboratories (`kahardi@sandia.gov`)
†Sandia National Laboratories (`dmdunla@sandia.gov`)

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

applications—e.g., web applications, databases, file I/O, data parsing/validation, etc. [23]. However, there is limited research available regarding using Rust from Python for numerical computing in general and tensor operations specifically. Although there exist many examples of improving runtime performance of Python software using C from Python, including previous work involving tensor algorithms [18], we focus primarily on using Rust from Python, as a detailed comparison using Rust from Python and using C from Python is beyond the scope of this work.

We test the hypothesis that using Rust from Python [20] for tensor operations leads to faster runtime performance over using Python alone by implementing several tensor kernels of increasing complexity and comparing average runtimes over multiple trials of each approach. Specifically, our contributions are as follows:

- Demonstration of runtime performance improvements greater than 2 orders of magnitude using Rust from Python over Python for two simple tensor kernels: vector dot product and dense matrix-vector product; and
- Demonstration of runtime performance improvements by approximately 1 order of magnitude using Rust from Python over Python for a more advanced tensor kernel: sparse tensor times vector product.

The remainder of the paper is structured as follows: Section 2 provides background material on tensor computations and related work; Section 3 describes the approach for comparing runtime performance of Python, Rust, and using Rust from Python for several tensor kernels; and Section 4 presents the numerical experiments. We then present our conclusions and ideas for potential future work in Section 5.

2 Background

In this section, we provide an overview of tensors and the `pyttb` software package, including the necessary background for using Rust from Python to achieve runtime performance improvements for tensor kernels. In addition, we discuss alternative approaches for potential runtime performance improvements of tensor kernels that could be considered in future research.

2.1 Tensors, Python, and Rust

Tensors refer to general data arrays with $d \geq 0$ dimensions. Thus, scalars, vectors, and matrices—as well as arrays with $d > 2$—are all examples of tensors, as illustrated in Figure 1. Throughout this paper, we use standard tensor notation from [2] as follows: scalars ($d = 0$) are denoted by lowercase letters (e.g. x), vectors ($d = 1$) are denoted as bold lowercase letters (e.g., $\mathbf{x}$), matrices ($d = 2$) are denoted as bold uppercase letters (e.g., $\mathbf{X}$), and general tensors ($d \geq 3$) are denoted as bold Euler script letters (e.g., $\boldsymbol{\mathcal{X}}$). All tensor elements in our current work are real-valued (i.e., taken from the set of real values, $\mathbb{R}$). Furthermore, tensor elements are defined by their indices ranging from 1 to the size of their dimensions; for example, vector elements are denoted as x_{i_1} for $\mathbf{x} \in \mathbb{R}^{n_1}$ with $i_1 \in \{1, \ldots, n_1\}$; matrix elements are denoted as x_{i_1,i_2} for $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2}$ with $i_1 \in \{1, \ldots, n_1\}$ and $i_2 \in \{1, \ldots, n_2\}$; and general tensor elements are denoted as $x_{i_1,i_2,\ldots,i_d}$ for $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ with $i_1 \in \{1, \ldots, n_1\}$, $i_2 \in \{1, \ldots, n_2\}$, and $i_d \in \{1, \ldots, n_d\}$.

The Python Tensor Toolbox (`pyttb`) [4] is a Python implementation of the highly cited Tensor Toolbox for MATLAB (`TTB`) [1]. Previous work indicated that the implementation of several tensor algorithms in Python can lead to faster runtimes over MATLAB [18], motivating the creation of `pyttb` to support tensor research and data analysis consistent with `TTB` for Python users. Currently, tensor data classes in `pyttb` are defined using the `ndarray` data structure from the `NumPy` Python package. Instances of `ndarrays` are stored in memory as contiguous blocks and most functions involving `ndarrays` utilize vectorized operations for improved performance [5]. Dense tensors are defined using an `ndarray` for the data values, and sparse tensors are defined using an `ndarray` for the indices of the nonzero values and an `ndarray` for the values corresponding to each of those indices. The *shape*, or sizes of the dimensions of the tensor, is defined using a tuple for both dense and sparse tensors. These attributes of the `pyttb` implementation are important for

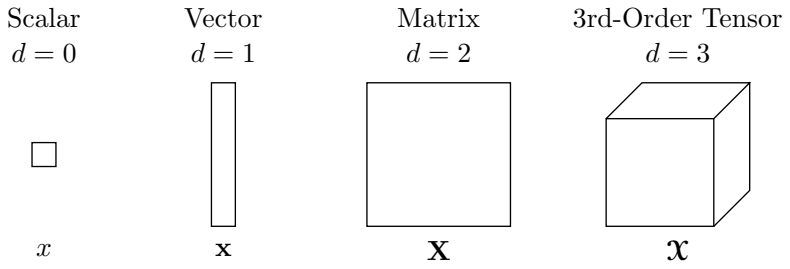


Figure 1: Illustration of tensors of various dimensions.

understanding how we are using Rust from Python for the various tensor kernels we investigated in our work and will be discussed in more detail in Section 3.

The Python language manages memory through reference counting and garbage collection, incurring overhead costs and potentially impacting runtime performance. Reference counting is maintained through Python's Global Interpreter Lock (GIL), which only allows one thread to control the Python interpreter at a time[1]. In CPython, which we use exclusively in the research presented here, the Python C API [22] handles the interaction between C (or C-compatible) code and the Python interpreter by managing data type conversions and reference counting across the FFI boundary [25]. In Rust, memory is managed through *ownership*, a set of rules checked at compile time to ensure memory safety [8]. We use the `PyO3` Rust crate[2] to provide interoperability between the different memory managers in Rust and the Python C API [20]. Python-compatible code is created and packaged from the `PyO3`-enabled Rust code using the `Maturin` [19] Python package.

2.2 Related Work

As mentioned above, the `NumPy` Python package provides vectorized operations of its data structures that utilize compiled C code, avoiding explicit loops/indexing and resulting in improved performance [5]. As some of these vectorized functions are already used in `pyttb` (via methods involving `ndarrays`), we can use current implementations of tensor kernels in `pyttb` as baselines for measuring runtime performance improvements, thus comparing using Rust from Python to using C from Python. Although, as noted above, the goal of our work presented here is not a comprehensive comparison of using Rust from Python and using C from Python, but rather if using Rust from Python is a viable option for improving runtime performance of tensor kernels originally implemented in Python.

The `Numba` Python package includes a just-in-time (JIT) compiler that optimizes Python code containing loops and `NumPy` data structures [10]. `Numba` provides function decorators to denote Python functions to be compiled into machine code during the initial call to each function. This approach can reduce overall development cost and complexity but may not actually lead to improved performance if `Numba`'s JIT compiler cannot successfully compile a function into machine code as requested. For example, if the intermediate representation of variables defined in a function cannot be identified at compilation time, the function is compiled in *object mode*, which has lower performance than when compiling fully into machine code using *no python mode* [10]. A detailed comparison between using `Numba` and using Rust from Python is beyond the scope of this work, but we include several examples of comparisons in the numerical experiments in Section 4 to illustrate some of the performance improvements that can be gained using `Numba`.

As stated above in Section 1, using Rust from Python for improved runtime performance has been demonstrated in various applications, including the `Polars` DataFrame package [16], the `TikToken` natural

[1] Although recent versions of Python provide support for an experimental feature of free-threading by releasing the GIL [21], which can leverage parallelism for runtime performance improvements, the use of this feature is beyond the scope of this work.

[2] Crates are packages external to the core Rust language that are available via Rust's Package Registry: `https://crates.io`
.

language processing package [14], and the `Pydantic` data validation package [3]. However, there is little published evidence of runtime performance improvement associated with packages focused on numerical computing (and specifically on tensor operations), thus motivating our work presented here.

3 Methodology

We test our hypothesis that using Rust from Python leads to faster runtime performance of tensor operations in `pyttb` using computational kernels of increasing complexity run over a range of various data sizes. The computational kernels are implemented in Python, Rust, and using Rust from Python, and the average runtimes over multiple trials are measured as a function of the size of the data. Specifically, we focus here on the following tensor kernels:

- vector dot product (Section 3.1);
- dense matrix-vector product (Section 3.2); and
- sparse tensor times vector product (Section 3.3).

For each tensor kernel, we run an experiment consisting of n_{trials} trials for each combination of kernel implementation and size of data to account for system runtime variability. All data arrays contain randomized 64-bit floating-point values (i.e., `numpy.float64` in Python and `f64` in Rust) uniformly sampled from $[0, 1]$ to reflect the standard data types used in numerical computing in general and `pyttb` specifically. New data arrays are generated for each trial of the loop-based kernels: vector dot product and dense matrix-vector product. However due to generation costs, one data array is generated for all trials of the sparse tensor times vector product. All the values are sampled using the `uniform` function from the `random` module in the `NumPy` Python package and using the `Uniform` struct from the `rand_distr` Rust crate. The data sizes chosen in our experiments were determined as those sizes that resulted in runtimes greater than milliseconds for the Python tensor kernel implementations to reduce the impact of noise in the timing instrumentation.

We implement all tensor kernels using the `NumPy` Python package `ndarray` data structure. However, only the sparse tensor times vector product kernel in `pyttb` leverages some of the vectorized functions from the `NumPy` Python package. Similarly, the Rust implementations use data structures from the `Ndarray` Rust crate [13], and only the sparse tensor times vector product kernel employs various `Ndarray` Rust crate methods for comparable functionality to the methods in the `NumPy` Python package.

The timing instrumentation in Python and Rust both use monotonic clocks that measure the duration between the two reference points. In Python, we use the `perf_counter` function from the `time` module. In Rust, we use the `now` and `elapsed` methods of the `Instant` struct from the `std` Rust crate. Thus, there could be some minor timing discrepancies when comparing runtimes between Rust and Python in our experiments. However, all timing results within Python (i.e., for the Python loop-based kernels, `NumPy`, `Numba`, and `pyttb`) are consistent across experiments. When using Rust from Python with the `PyO3` Rust crate, overhead costs are included in the runtimes of the initial call per Python script execution. The specific sources of the overhead cost for the first call are uncertain—possibilities include crossing the FFI boundary, loading the Python module, data type coversions, etc.—however, future research is required to determine the contribution of each source. To control for these overhead costs in our experiments, we compute each kernel once before measuring runtimes. Furthermore, we estimate this overhead when using Rust from Python for the vector dot product and the dense matrix-vector product at each of the data sizes by computing the difference in average runtimes of the first call and the average runtimes of the n_{trials} calls after the first call for each experiment. Reports of these estimated costs are reported in the numerical results presented in Section 4.

3.1 Vector Dot Product

The vector dot product computes the scalar product of two vectors (first-order tensors) as the summation of the products of the elements of each vector at the same indices. The vectors must have the same number of elements in order to calculate the dot product. The equation for scalar output is as follows:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i_1=1}^{n_1} x_{i_1} y_{i_1} = x_1 y_1 + x_2 y_2 + \cdots + x_{n_1} y_{n_1} \ . \tag{1}$$

We implement Equation 1 using a single loop over the two vectors in Python, Rust, and using Rust from Python. We also compare the vector dot product using `Numba`'s JIT-compiled version of the Python implementation. When using Rust from Python and `Numba`, we estimated the costs of the first call of using Rust from Python and the `Numba` JIT compiler, respectively, to illustrate the additional overhead in using these approaches. Note, the overhead of the first call of using the `Numba` Python package includes both the JIT compile time in addition to the other overhead costs mentioned above.

3.2 Dense Matrix-Vector Product

The dense matrix-vector product computes an output vector whose elements consist of the vector dot product between the rows of a matrix (second-order tensor) and a vector (first-order tensor). Mathematically, a dense matrix-vector product of $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2}$ and $\mathbf{x} \in \mathbb{R}^{n_2}$ is defined as follows:

$$\mathbf{A} \times \mathbf{x} = \left[\sum_{n_2=1}^{n_2} a_{1 i_2} x_{i_2}, \ \sum_{i_2=1}^{n_2} a_{2 i_2} x_{i_2}, \ \ldots, \ \sum_{i_2=1}^{n_2} a_{n_1 i_2} x_{i_2} \right]^\top , \tag{2}$$

where the result is a column vector of length n_1. We implement Equation 2 using a nested loop in Python, Rust, and using Rust from Python. The outer loop computes the elements of the resulting column vector (first-order tensor)—i.e., the elements of the right hand side of Equation 2 above. The inner loop computes the sums in each of those elements of the resulting vector.

We define multiple experiments involving the dense matrix-vector product corresponding to matrices with an 1) increasing number of rows (n_2 constant), 2) increasing number of columns (n_1 constant), and 3) increasing number rows and columns (n_1 and n_2 equal). Note that computing with second-order tensors requires the additional consideration of *memory layout*, as the elements of matrices can be stored in memory in row-major or column-major order—defined in the `NumPy` Python package as "`C`" and "`F`" (denoting Fortran) order, respectively. All data structures in the current implementation of `pyttb` use column-major order (for consistency with `TTB`), whereas the corresponding data structures in the `Ndarray` Rust crate use row-major order, and these differences may impact the runtime performance comparisons. Further research could be conducted to assess whether `pyttb` could be improved by providing more flexible memory layout options. Although preliminary studies demonstrated that memory layout can impact runtime performance, a detailed assessment of the impact of memory layout in tensor operations is beyond the scope of the current work. As in the vector dot product experiment described above in Section 3.1, we also compare runtime performance using JIT-compiled versions of the Python implementation via `Numba`, and runtime costs associated with the first call of using Rust from Python and the `Numba` JIT compiler are estimated in a similar way.

3.3 Sparse Tensor Times Vector Product

The sparse tensor times vector (TTV) product multiples a d-way tensor with n_v vectors resulting in a tensor with $d - n_v$ dimensions, where $n_v \in \{1, \ldots, d\}$ and the size of each vector must correspond to the size of one of the dimensions of the d-way tensor [2]. For example, for the TTV product of a sparse third-order tensor ($d = 3$) and a single vector ($d = 1$ and $n_v = 1$), the result is a tensor of dimension

$d - n_v = 3 - 1 = 2$ (i.e., a matrix). Mathematically, the TTV product of a d-way tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and a vector $\mathbf{x} \in \mathbb{R}^{n_k}$ can be defined element-wise as follows:

$$(\boldsymbol{\mathcal{A}} \times_k \mathbf{x})_{i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_d} = \sum_{i_k=1}^{n_k} a_{i_1,i_2,\ldots,i_d} x_{i_k} \ , \tag{3}$$

where $\times_k$ denotes mode-k tensor multiplication [2, Sec. 3.3] and $k \in \{1, \ldots, d\}$.

Sparse tensors contain a large percentage of zero values; therefore, only the nonzero elements are stored to support efficient use of memory. This can be beneficial for data analysis, as loops over the sparse tensor instances only iterate over the indices associated with nonzero values. The sparse TTV serves as the foundation of the sparse matricized tensor times Khatri-Rao product (MTTKRP) kernel, the main computational kernel used in important tensor algorithms, such as low-rank tensor decompositions [2] and tensor-on-tensor regression [11]. Therefore, improving the runtime performance of the sparse TTV kernel in `pyttb` will also improve the runtime performance of the sparse MTTKRP and these important tensor algorithms.

For the experiment involving the TTV kernel, we use the sparse tensor function `sptensor.ttv` from `pyttb` (`pyttb.sptensor.ttv`) as the Python implementation. Although `pyttb` supports general dimensions for sparse tensors (e.g. $d = 1$ or $d = 2$) and the ability to compute against multiple vectors in one operation, this experiment only focuses on the case of a third-order (d=3) tensor times one vector as an illustration of typical runtime performance for the sparse TTV kernel. Furthermore, this experiment isolates the runtime performance of the TTV kernel by removing all error checking from the `pyttb` code. Although `pyttb` converts the output to a dense tensor when the percentage of nonzero elements exceeds 50%, this experiment removes this conversion step as well.

The sparse TTV kernel in `pyttb` leverages several vectorized functions of the `ndarray` class in the `NumPy` Python package that are not implemented in the `Ndarray` Rust crate: `setdiff1d`, `arrange`, `unique`, `accumarray`, `flatten`, `squeeze`, `nonzero`. Therefore, our implementation in Rust does not correspond exactly line-by-line to the implementation in `pyttb`. Instead, our implementation provides equivalent functionality of the TTV kernel using a combination of available functions provided in the `Ndarray` Rust crate, Rust iterators, and several new functions created specifically for this experiment. Note that we made no attempt to improve the current sparse TTV implementation in `pyttb`, and future work could attempt to improve the `pyttb` implementation to provide a more equitable comparison. Furthermore, as no sparse tensor implementation currently exists in Rust, our implementation of the TTV kernel when using Rust from Python returns the individual arrays for `newsubs`, `newvals`, and `newshape`, which are used to instantiate a new instance of `pyttb.sptensor` after the return to Python (which is equivalent to the final step of the `pyttb.sptensor.ttv` implementation). This runtime cost of the `sptensor` instantiation in `pyttb` is included in the average runtimes presented in the numerical results in Section 4.

The sparse TTV kernel is tested using a constant density of nonzero elements across the various data sizes investigated in the experiment. The data tensors, $\boldsymbol{\mathcal{A}}$, are also instantiated with random values using the `pyttb.sptenrand` method, which generates the nonzero elements by sampling uniformly from $[0, 1]$. For consistency across the experiment, in all trials for all data sizes, the sparse TTV is computed using third-order tensors with dimensions of equal sizes, $n_1 = n_2 = n_3$, multiplied by a single vector (first-order tensor) of size n_2 along dimension $k = 2$.

4 Numerical Experiments

All numerical experiments were performed on a system with an AMD 1.8 GHz EPYC 9345P processor and 768GB RAM, running Red Hat Enterprise Linux Release (RHEL) 9.5 and the following versions of Python, Rust, and their respective packages and crates:

- Python 3.12 and the following packages:

  `pyttb` 1.8.2, `NumPy` 2.2.6, `Maturin` 1.9.4, `Numba` 0.61.2; and

- Rust 1.81 and the following crates:

  `PyO3` 0.25.0, `Ndarray` 0.16.1, `rust-numpy` 0.25.0, `Rand` 0.9.2, `rand_distr` 0.5.1.

For each tensor kernel, we computed the average runtimes across $n_{trials} = 30$ trials for each size of data.

4.1 Vector Dot Product

The vector dot product was calculated for vectors of sizes $n_1 \in \{10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$. The average runtimes versus vector size for Python, Rust, using Rust from Python, and using a `Numba`-compiled version of the Python code are shown in Figure 2. The results illustrate a runtime improvement greater than 2 orders of magnitude between using Rust from Python over using Python alone. At larger vector sizes, the plot illustrates negligible differences between the average runtime performance between using Rust from Python, Rust, and using `Numba`. However, at smaller vector sizes, such as $n_1 = 10^4$, the Rust implementation is faster than using Rust from Python and using `Numba`, potentially indicating runtime variability in these other methods at smaller vector sizes.

Estimated overhead costs for the first call of using Rust from Python and the `Numba` JIT compiler as a function of the size of the vector are shown in Table 1. The runtime overhead for the first call of using Rust from Python is approximately 3 orders of magnitude faster than the first call of using `Numba`. However, the estimated overhead for using Rust from Python increases slightly with the size of the vector, while the estimated overhead using `Numba` is relatively constant.

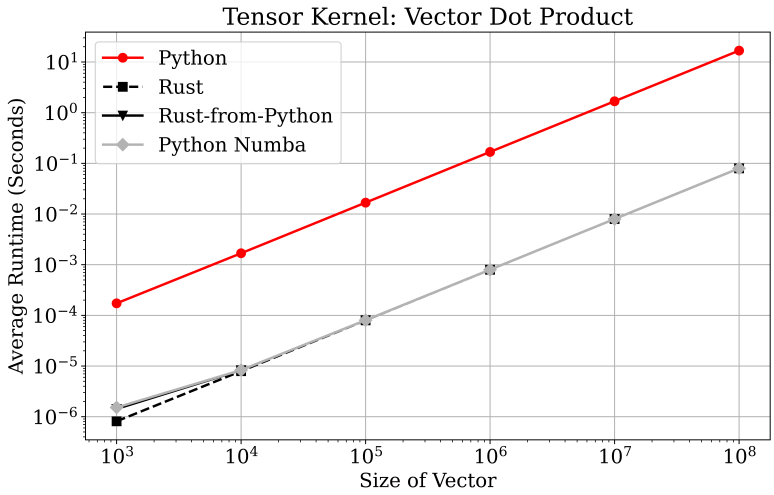


Figure 2: Average runtime of the vector dot product as a function of vector size.

Table 1: Estimated overhead costs (in seconds) for using Rust from Python and using the `Numba` Python JIT compiler for the vector dot product.

Size of Vector	Rust from Python	`Numba`
10^3	1.44×10^{-4}	3.17×10^{-1}
10^4	1.44×10^{-4}	3.17×10^{-1}
10^5	1.49×10^{-4}	3.18×10^{-1}
10^6	1.58×10^{-4}	3.16×10^{-1}
10^7	2.42×10^{-4}	3.19×10^{-1}
10^8	3.07×10^{-4}	3.19×10^{-1}

4.2 Dense Matrix-Vector Product

The dense matrix-vector product was tested using matrices with an 1) increasing number of rows, 2) increasing number of columns, and 3) equal, increasing number of rows and columns of the following sizes:

- $n_1 \in \{10^2, 10^3, 10^4, 10^5, 10^6\}$ with constant $n_2 = 10^2$;
- $n_2 \in \{10^2, 10^3, 10^4, 10^5, 10^6\}$ with constant $n_1 = 10^2$; and
- $n_1, n_2 \in \{10^2, 10^3, 10^4\}$.

The average runtimes for Python, Rust, using Rust from Python, and using a `Numba`-compiled version of the Python code for the various cases are shown in Figure 3. At the largest matrix size, as demonstrated in all subplots, using Rust from Python presents a runtime performance improvement greater than 2 orders of magnitude. Note that the reasons for the increased average runtimes for Rust and `Numba` over using Rust from Python for the larger matrix sizes in all three experiments is unclear from our experiments. In future work, we plan to investigate the differences in more detail.

As in the results for the vector dot product discussed in Section 1, we estimated the overhead as a function of matrix size for the first call of using Rust through the FFI and JIT compiling with `Numba`. The results were calculated for matrices with equal numbers of rows and columns, with the total number of elements in the square matrices displayed in Table 2. The results illustrate lower overhead from the first call of using Rust from Python in comparison to the first call when using `Numba`. However, the estimated overhead cost when using Rust from Python has greater variability as the size of the matrix increases in comparison to using `Numba`.

Table 2: Estimated overhead costs (in seconds) for using Rust from Python and using the `Numba` Python JIT compiler for the dense matrix-vector product.

Number of Matrix Elements	Rust from Python	`Numba`
10^4	1.50×10^{-4}	3.58×10^{-1}
10^6	1.87×10^{-4}	3.58×10^{-1}
10^8	6.41×10^{-2}	3.06×10^{-1}

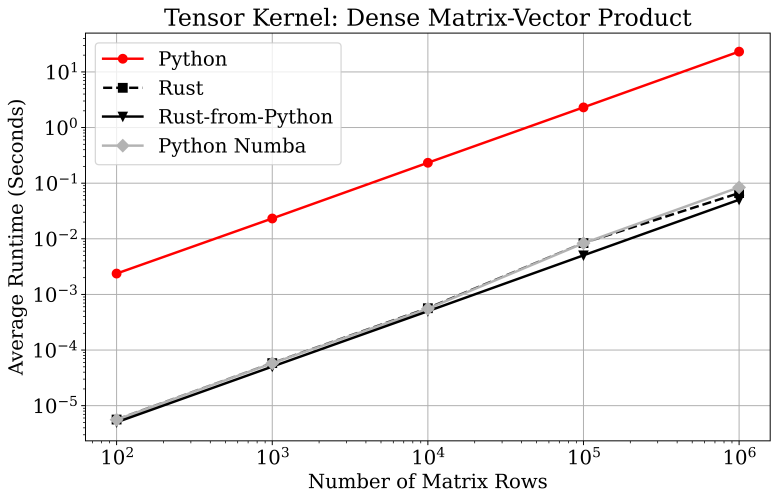
4.3 Sparse Tensor Times Vector (TTV) Product

The sparse tensor times vector product was tested using third-order tensors containing 1% nonzero values (specified as `density=0.01` when calling `pyttb.sptenrand`) and equal sizes of dimensions, with $n_1, n_2, n_3 \in \{100, 200, \ldots, 1200\}$. In all experiments, the sparse tensors are multiplied along the second-dimension by a dense vector of random floating point values.

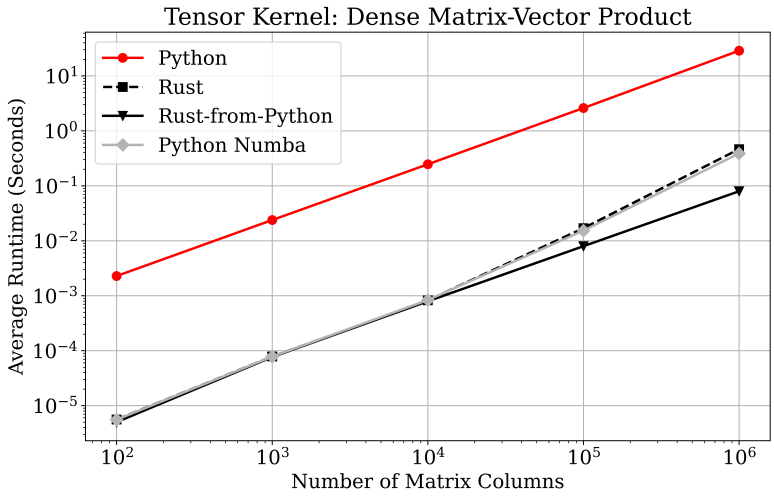
The results shown in Figure 4 demonstrate an improved runtime performance of approximately 1 order of magnitude when using Rust from Python over using the Python implementation in `pyttb`. The plot illustrates that the difference between the runtime performance of using Rust from Python to Python increasing slightly with the size of the tensor. There are potentially many differences in the implementations leading to these runtime performance differences, such as memory layout, compiler optimizations (e.g., Rust iterators versus `NumPy` vectorized optimizations), and FFI boundary crossing overhead (Rust via `PyO3` versus `NumPy`, which uses C-based extension modules). In future work, we plan to investigate the specific causes of these runtime improvements across these various factors.

5 Conclusion and Future Work

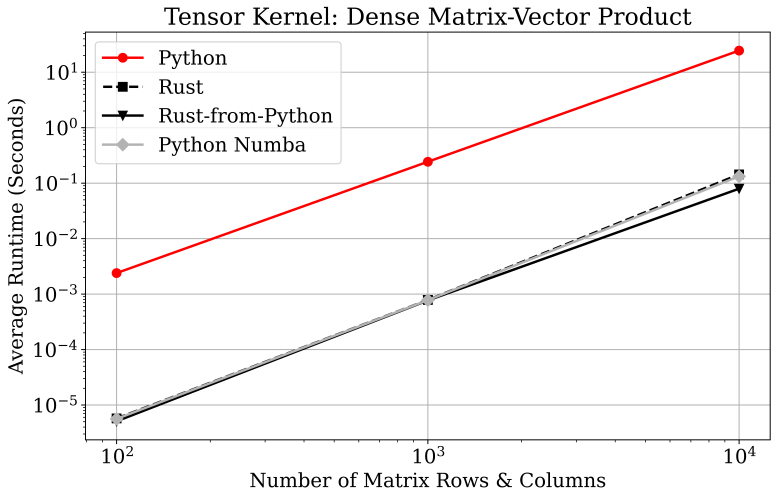
We demonstrated that using Rust from Python can result in improved runtime performance for several tensor kernels available in the Python Tensor Toolbox (`pyttb`). Our results indicate runtime improvements



(a)



(b)



(c)

Figure 3: Average runtimes of dense matrix-vector product for (a) increasing number of columns, (b) increasing number of columns, and (c) equal, increasing number of rows and columns.

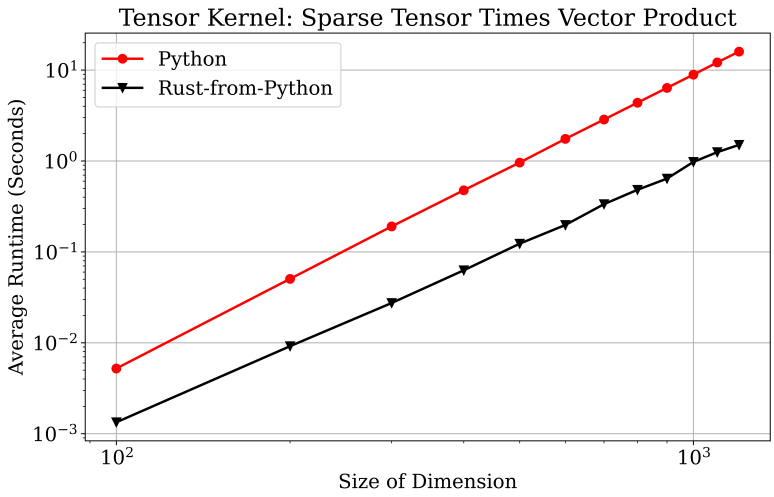


Figure 4: Average runtime of sparse tensor times vector product as a function of tensor dimension size.

greater than 2 orders of magnitude for two simple tensor kernels—the vector dot product and the dense matrix-vector product—and runtime improvements of approximately 1 order of magnitude for a more advanced tensor kernel, the sparse tensor times vector (TTV) product. For the two former kernels, we also compared to versions of the Python implementations that were compiled into machine code using the just-in-time compiler from the `Numba` Python package, and results indicate comparable runtime performance improvements to that of using Rust from Python. Future work could consider experiments leveraging `Numba` on more advanced computation, like that of the sparse TTV kernel. There is a trade-off between lower development complexity versus higher one-time execution overhead cost in using just-in-time compilation in `Numba` that could also be explored in detail in future work. Other ideas for future work include investigation into the role of memory layout, compiler optimizations, using iterators over loop-based implementations, and increased computational complexity (e.g., as in the matricized tensor times Khatri-Rao product [MTTKRP]) when searching for opportunities that could lead to improved runtime performance. Lastly, although we focused here on runtime improvements on a single CPU core, we could also investigate the use of advanced computing architectures—e.g., GPUs (graphics processing units), TPUs (tensor processing units), etc.—and the use of on-node concurrency (via Rust or solutions provided in several Python packages) or distributed-memory computation to take advantage of multiple computing resources simultaneously.

Acknowledgements

We would like to express our sincere gratitude to Joshua B. Teves, Carolyn Mayer, Carlos Llosa, Rich Lehoucq, and Jeremy Myers for their invaluable feedback and suggestions during the preparation of this manuscript. Their insights greatly enhanced the quality of this work.

References

[1] Brett W. Bader and Tamara G. Kolda. Tensor Toolbox for MATLAB (Version 3.6). https://www.tensortoolbox.org/, 2023.

[2] Grey Ballard and Tamara G. Kolda. *Tensor Decompositions for Data Science*. Cambridge University Press, 2025.

[3] Samuel Colvin. Pydantic V2 Plan. https://pydantic.dev/articles/pydantic-v2, 2022. Accessed: 2025-07-02.

[4] Daniel M. Dunlavy, Nicholas T. Johnson, et al. pyttb: Python Tensor Toolbox (Version 1.8.2). https://github.com/sandialabs/pyttb, January 2025.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[6] Price D. Johnson and Douglas D. Hodson. PyO3: Building Python extension modules in native Rust with performance and safety in mind. In Douglas D. Hodson, Michael R. Grimaila, Hamid R. Arabnia, Leonidas Deligiannidis, and Torrey J. Wagner, editors, *Scientific Computing and Bioinformatics and Computational Biology*, pages 23–30, Cham, 2025. Springer Nature Switzerland.

[7] Boris N. Khoromskij. *Tensor Numerical Methods in Scientific Computing*. De Gruyter, Berlin, Boston, 2018.

[8] Steve Klabnik, Carol Nichols, et al. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2022.

[9] Taichi Kosugi, Xinchi Huang, Hirofumi Nishi, and Yu-ichiro Matsushita. Tensor-decomposition technique for qubit encoding of maximal-fidelity lorentzian orbitals in real-space quantum chemistry. *Phys. Rev. A*, 111:052615, May 2025.

[10] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Eric F Lock. Tensor-on-tensor regression. *Technometrics*, 60(4):480–492, 2018.

[12] Linjian Ma and Chao Yang. Low rank approximation in simulations of quantum algorithms. *Journal of Computational Science*, 59:101561, 2022.

[13] Ndarray Developers. Ndarray: an N-dimensional array for general elements and numerics (version 0.16.1). https://github.com/rust-ndarray/ndarray, 2025.

[14] OpenAI. tiktoken. https://github.com/openai/tiktoken, 2023.

[15] Yannis Panagakis, Jean Kossaifi, Grigorios G Chrysos, James Oldfield, Taylor Patti, Mihalis A Nicolaou, Anima Anandkumar, and Stefanos Zafeiriou. Tensor methods in deep learning. In *Signal Processing and Machine Learning Theory*, pages 1009–1048. Academic Press, 2024.

[16] Ulrik Thyge Pederson. Better together four examples of how Rust makes Python better. https://towardsai.net/p/l/better-together-four-examples-of-how-rust-makes-python-better, 2023. Accessed: 2025-07-02.

[17] A. Pellegrino, H. Stein, and N.A. Cayco-Gajic. Dimensionality reduction beyond neural subspaces with slice tensor component analysis. *Nature Neuroscience*, 27:1199–1210, 2024.

[18] Matthew G. Peterson and Daniel M. Dunlavy. Tensor Toolbox: Wrapping to Python using SWIG, December 2014. Technical Report Number SAND2015-3829O.

[19] PyO3 Project and Contributors. Maturin. https://github.com/PyO3/maturin, 2025.

[20] PyO3 Project and Contributors. PyO3 (Version 0.25.0). https://github.com/PyO3/pyo3, 2025.

[21] Python Software Foundation. Python experimental support for free threading. https://docs.python.org/3/howto/free-threading-python.html. Accessed: 2025-07-02.

[22] Python Software Foundation. Python C API Reference Manual. https://docs.python.org/3/c-api/index.html, 2025. Version 3.x (refer to specific version if applicable).

[23] Prashanth Rao. How Rust is supercharging Python from the ground up. https://thedataquarry.com/blog/rust-is-supercharging-python, 2023. Accessed: 2025-07-02.

[24] Neriman Tokcan, Shakir Showkat Sofi, Van Tien Pham, Clémence Prévost, Sofiane Kharbech, Baptiste Magnier, Thanh Phuong Nguyen, Yassine Zniyed, and Lieven De Lathauwer. Tensor decompositions for signal processing: Theory, advances, and applications. *Signal Processing*, 238:110191, 2026.

[25] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.