Python vs Cython

Rohan Bhatia, Marius Fleischer

December 2021

1 Motivation

The programming language Python has two major strengths. It is easy to learn Python, no matter of the previous experience, and Python enables programmers to develop prototypes quickly. A downside of Python is that it is slow compared to other languages since it is usually executed using interpretation. Especially computation-heavy programs struggle with this limitation. A possibility to overcome this problem is to compile the Python program to native code before execution. In the case of a naive compilation without any other changes the resulting program will still be slower than C programs, for example, since the runtime system has to be compiled into the binary, too. Therefore, the overhead from the runtime system is still present in the executable.

Gal et al. investigated this issue for JavaScript. They found that using runtime type information during the compilation results in a significant speedup since it enables the compiler to use the additional information for more specific optimizations. Knowing the types also helps to reduce the number of checks which need to be performed.

Python3 offers the possibility to provide type hints for variables and function definitions. Combining this fact with the results of Gal et al., it seems beneficial to use this type information together with type propagation and potentially even lightweight type inference to determine the types of variables and arguments. Then, we can provide the compiler with this additional information and enable it to use type specific optimizations and reduce the number of checks which we expect to result in a good speedup compared to naive compilation. Our project focused on the second part, finding out whether and if, how we get a good speedup.

For the compilation, we use Cython [1]. If Cython is new to the reader, see an example of a Cython program in the Appendix or refer to the documentation of Cython[1]. For our tests, we chose a set of programs to look at a variety of tasks. We will discuss each of the programs in the following. For investigation of differences in the execution time of the different program versions, we used profilers which we will introduce in the next chapter.

2 Profilers

Two famous profilers exist for Python: cProfile and line_profiler. Both profilers are written in C and are deterministic. cProfile is a lightweight function-level profiler, i.e., cProfile can tell which function is slow but can't tell why it is slow. line_profiler overcomes this limitation by profiling at the line-level. line_profiler's output consists of the lines in a set of functions to profile. However, as line_profiler is more detailed, it incurs more profiling overhead compared to cProfile. Nevertheless, it is extremely useful for identifying hotspots in the program for small inputs.

[1] `https://cython.readthedocs.io/en/latest/`

3 Sieve of Eratosthenes

Sieve of Eratosthenes is used to find all the prime numbers from 2 up to a particular given number. This program constructs an array with length equal to the given input. Then, it starts with the base fact that 2 is a prime number and so it marks 2 as a prime in the array. Then, it traverses all the multiples of 2 and marks them as not-prime. It then does the same for the next prime, which is 3, then 5, and so on.

This program doesn't rely on much computation. Instead, the hotspots for this program are array allocation, garbage collection, array indexing and element assignment. It is a simple yet very useful program for understanding where Cython speedups come from.

We write the base program in *Python*. Then, we compile it with *Cython*. Then, we add Cython types to the program before compilation. We call this variant *Cython Typed*. To see if Cython lets us reach the speed of pure C, we also write the program in *C*.

In the *Cython Typed* variant, we declare an array with Python's `array` module. This module creates efficient typed arrays. However, these arrays are stll Python objects and operations like indexing and element assignment are done at Python speed. We get a reference to the underlying C array by using a memory view [2]. With the C array reference, operations on it are performed at C speed.

We plot the speedup of the variants compared to pure Python in the appendix (Figure 1). We see that Cython improves the execution time by a factor of ~ 1.5, Cython Typed improves the performance by a factor of ~ 2.5. However, pure C offers a speedup of ~ 40. In the following section, we make a deep-dive to understand why C is much faster than Cython Typed. Then, we make a few optimizations to get C-like performance with Cython Typed.

3.1 Deep-Dive: Cython v C

From the HTML generated by Cython while compiling the Sieve program, we find several bottlenecks that limits Cython from achieving C speed.

Usually, a Python's `range` for-loop is compiled to C if all of the loop variables are typed. However, we still see that the first for-loop is not compiled to C. It remains a Python for loop after compiling it with Cython. Whereas, the second for-loop is compiled to C. The difference between the first and the second loop in C is that the first loop's step size argument for the `range` function is a variable while for the second loop, it is a constant 1. Cython isn't able to determine if the loop moves forward or backward during the runtime. As Cython currently can only handle simple cases, it's not able to produce a corresponding C loop.

Cython first converts all the loop variables of the first loop back to Python because it has to pass them as arguments to Python's `range` function. Then, it gets the reference to the `range` function through Python Runtime API. Then, it makes the function call. Then, Cython inserts various checks to see if the return value of the `range` function is a list, a tuple or an iterator. Even though, we know that Python `range` always returns an iterator, Cython isn't smart enough to know that. All the logic mentioned in this paragraph uses Python objects. So, Cython also has to insert reference counting statements for the variables/references. This adds to a lot of overhead considering that the for-loop is called inside a while loop. We bypass all these overheads by replacing the for loop with a while loop, which is easier for Cython to compile.

Another optimization is related to function calls. Before we add any types, calling a function involves a lot of calls to the Python runtime. First of all, we need to retrieve the global name of the function which we want to call. The next step is to determine the type of the function, e.g., whether it is an instance method. Depending on the function, there might also be some calls to the garbage collector or type checking. After this, we are ready to call the function. If we compare this to function calls in C, there is a lot of overhead. In normal C, we just call the function directly. To allow Cython to generate a normal C-like function call, we need to provide type information for all parameters and the variables used as parameters, the types need to match, and the function needs to be defined as a

pure C function.

Defining the function as pure C function and annotating all parameters also has another advantage. When we enter or leave a function, there is also some overhead in the non-typed version. Python objects need to be converted to the Cython representation upon function entry and be converted back to the Python objects when leaving, we need to instantiate a lot of objects for use in the function and sometimes check the types of the objects we received as parameters. In addition, to be able to call the function from Python, there is a wrapper function which helps with these conversions and makes the native function accessible to Python code. By defining the function as pure C function and annotating the types, we remove the need for a wrapper function, type checking can be done by the compiler, conversions are no longer needed, and we can often also reduce the number of instantiations. The combination of these two improvements can give us a good speedup when we call a function repeatedly.

Another place we can optimize are bound checks. We know that C isn't memory safe and it allows accessing arbitrary memory locations. Cython on the other hands adds some logic for bound checking and negative-indexing (indexing with negative values) each time an indexing operation is made. For bringing Cython performance closer to C, we disable the bound checks and negative indexing for the sieve function.

The way we created an array earlier in Cython using Python's `array` module is very convenient. However, we also note that it's a cause of slowdown when declaring large arrays. It's because the allocation, initialization and reference counting for each array element happens at Python speed. Another way to create an array using the C API that Cython provides. We can declare the array with `malloc`, but we would also have to manually release the memory with `free`.

After all these optimizations, we see that in Figure 2, for the sieve program, most of the speedup comes from optimizing the for loop and using C to declare and initialize an array. We also note that different programs may respond differently to these optimizations depending on where the hotspots lie.

4 IO

This example program models IO-heavy tasks and is therefore completely different to the rest of the programs. The program first writes a large number of characters, for example 2 million, to a file and then reads the same characters back into memory. We annotate as many variables and function arguments as possible and then measure the execution times. You can find a plot of the speedups for the compilation with and without types in the appendix (Figure 3). As we can see, the speedup is very small. Even worse, there is no noteworthy difference in performance between the Cython and the typed Cython version. Investigating this phenomena shows very quickly that about 90% of the execution time are spent either actually writing to or reading from disk. This is not bound by the execution speed of the Python or Cython program but by the speed of the disk. Therefore, compilation cannot achieve a speedup in this part of the program. The remaining part of the program in the Cython versions is slightly faster than in Python which leads to the observed small speedup. In summary it can be said, that the effort of optimizing a program by adding types and compiling it with Cython is not worth if the program is IO-heavy.

There is an additional lesson we learned while looking into this program. The first version of the program generated the characters randomly within the method responsible for writing to disk. The random generation is slow compared to writing a few thousand characters to disk or reading them. Since we often started the profiling by running line_profiler on the main function it seemed as if writing was a few magnitudes slower than reading. Since this is in line with usual performance of older disks, this seemed reasonable and we nearly overlooked the fact, that the slowdown in fact comes from the random generation instead of writing. So, the lesson learned is to track down the reasons for results even if they seem in line with what you expected.

5 Heap Sort

Heap Sort is another one of the computation-heavy test cases. Opposed to the Sieve, it consists mainly of comparisons and swaps inside the array. This also makes it different from the typical examples of computation-heavy programs, matrix operations. After trying to annotate as much variables and function arguments with types, and converting functions to C functions, we get the performance improvements shown in Figure 4 in the Appendix. Apparently, the performance of Heap Sort can be improved very well.

A first look at the HTML provided by Cython shows that the typed version has significantly less interactions with Python than the non-typed version. Using the type annotations, we achieve native loops similar to what we have seen for the Sieve already. We are also able to get completely native function calls as we did for the Sieve. The source line which is responsible for swapping the array entries seems to be similar for both versions judging from the shade of yellow. When we look deeper into the code corresponding to this line, we see a different picture. While the non-typed version obviously needs a few calls to Python to perform this task, the typed version actually does not. It only interacts with the Python runtime if there is an indexing error. If not, the only difference to an array access in C is, that there are some checks concerning the index value to prevent out-of-bounds accesses. This difference can also be seen in the profiling data.

6 Matrix Multiplication

Numpy is a popular Python library used for scientific computing. Numpy offers vectorized operations as first-class citizen which makes it super-easy to write intricate programs. To understand how Cython can improve the performance of Numpy code, we wrote a simple numpy program. The matrix multiplication program first creates two 2D numpy matrices initialized randomly. Then, the program multiplies the two matrix.

Figure 5 shows the effect of Cython compilation on Matrix multiplication. We see that Cython and Cython Typed suffers from a slight slowdown. It happens because before calling numpy function, Cython makes a lot of calls to Python Runtime API. It gets a new reference to numpy module every time it calls a function. Then, it gets the function reference from its name. Then, Cython makes multiple checks to determine if the function is an instance method, a static method, a function, or a C function. For all the temporary variables introduced because of this logic, it adds reference counting code. Adding types further complicates the program as now Cython must perform type checks and checks on the shapes of numpy arrays.

7 Matrix Inversion

Matrix inversion is another typical computation-heavy task related to matrices. For this task, we decided not to use numpy since we could not really do much to optimize the execution time when we only call the numpy function for inverting a matrix. The implementation we are using handles matrices using two-dimensional lists. We tried to find a conversion from lists to numpy arrays which could also be performed automatically later on but even in this small program it became obvious that this is not feasible. Because of that and Cython offering no option to annotate lists, we were not able to add type information to the matrices in this program. After annotating the rest of the program, we achieved the speedup which can be seen in Figure 6 in the Appendix.

We can see, that there is a good speedup, but it is not as good as the one seen for Heap Sort. Investigating the sources of the speedup shows, that we can, in most cases, achieve pure C loops. In some cases, for example when the number of iterations depends on the dimension of the matrix, pure C loops are not possible. In addition, we were able to get pure C function calls within our program. Problematic are all source lines that interact in some way with the matrices stored in our

two-dimensional lists. Each interaction with a list results in a call to the Python runtime and prevents a speedup or even slows down the execution. In addition to these findings, we noticed something interesting that we did not see for any of the other programs. In the function `getMatrixDeternminant`, there is a source line which consists of one typed and two non-typed operators. Comparing the C code of the typed and the non-typed version of this source line shows that the same number of interactions with the Python runtime are is needed but the two versions use different functions. The functions in the typed versions were more specific to the types that we annotated the one typed variable with. In addition, profiling this function showed that the typed version is indeed faster than the non-typed one. Therefore, even if it is not possible to provide type annotations for all variables involved in an operation, partial type information can still help the compiler to choose a more specific function of the runtime which might be faster than the general one.

8 Stochastic Gradient Descent

We implement a stochastic gradient descent algorithm on a Neural Network with 2 hidden layers. We train the neural network for 5 epochs on the MNIST dataset. We see in Figure 7 that Cython only slows down the program because of reasons mentioned in the Matrix Multiplication section.

9 Edge Detection

With these programs we intended to apply what we learned by looking at the small example programs to real-world programs. We chose the two edge detection algorithms Marr-Hildreth and Canny. Both are implemented using numpy, therefore, our first step was to add the appropriate set up for numpy. In addition, we defined as many functions as possible as pure C functions. Since both implementations have only one major function each, the whole algorithm runs inside the native function. As a next step, we tried to achieve native for-loops by providing types for all variables involved in any loop definition. Finally, we tried to annotate variables which are used together so that the corresponding source lines can be compiled to pure C code.

As you can see in the appendix in Figure 8 for Canny and in Figure 9 for Marr-Hildreth, the Cython version is roughly as fast as the Python version. The typed Cython version is only slightly faster. While we investigated the reason for these poor improvements, we noticed that roughly 70% of the execution time are spent in numpy operations. As seen before, we cannot speedup numpy operations. They actually often result in a slowdown since we have to access numpy via the Python runtime, which leads to an overhead. For the remaining lines, it is hard to get any speedup except from the for loops, which are indeed pure C loops. These two factors together, a large part of the program being impossible to speedup with typing and the remaining part offering only limited options, lead to the small speedup compared to the one we achieved for Heap Sort or the Sieve of Eratosthenes.

10 Conclusion

We tested the speedup effect of using Cython on Python programs using several benchmark programs. We successfully find the cases where using type information to generate type-specialized native code offers huge speedups. Therefore, we set the ground for an efficient Python runtime which generates type-specialized code using type-annotations and lightweight type-inference. Such a Python runtime must intelligently estimate the speedup that is possible and the cost (in terms of time spent) to pay to achieve that speedup.

References

[1] Cython. https://cython.org/. Last accessed on 12/02/2021.

[2] Memoryview objects. URL `https://docs.python.org/3/c-api/memoryview.html`.

[3] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542528. URL `https://doi.org/10.1145/1542476.1542528`.

Appendix

Graphs

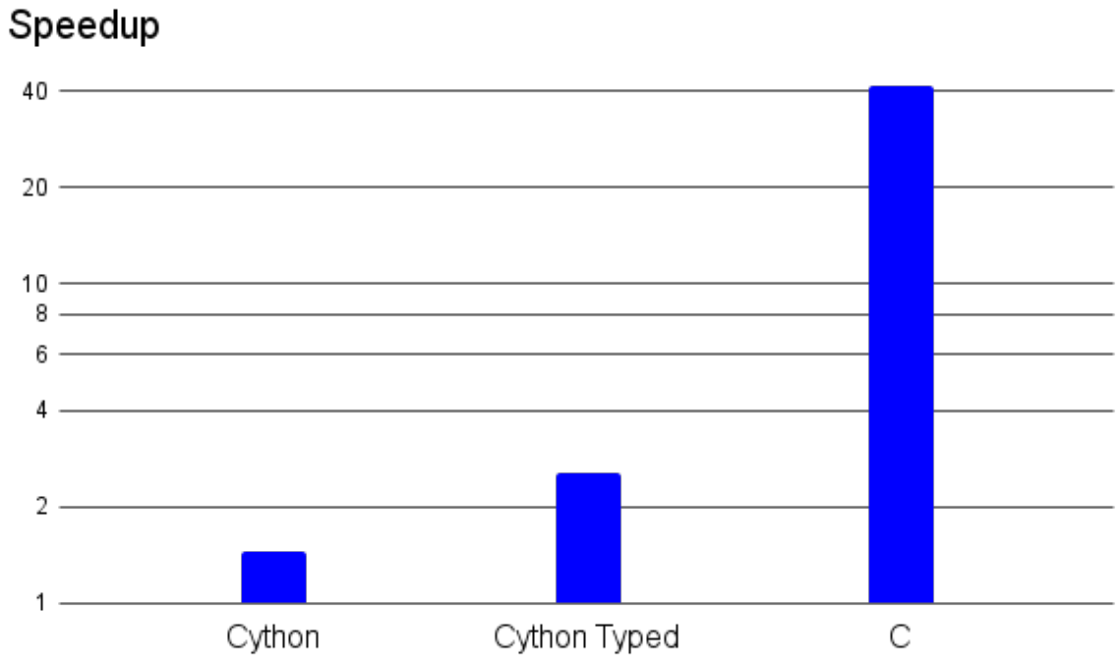

Figure 1: Speedup gained by direct compilation and by compilation of the typed version for the sieve program

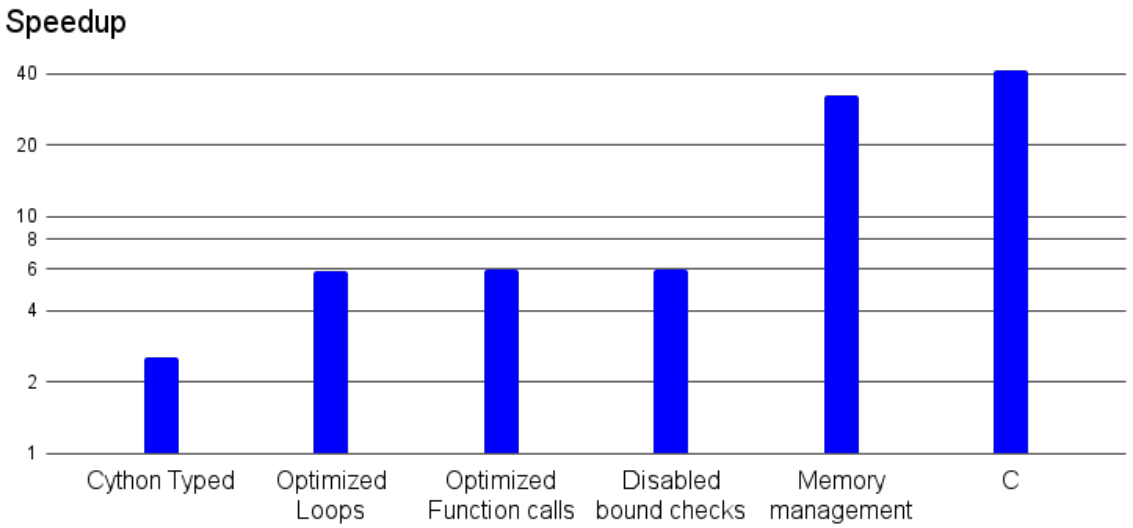

Figure 2: Effect of several optimization on the Sieve programs. The optmizations are stacked from left to right.

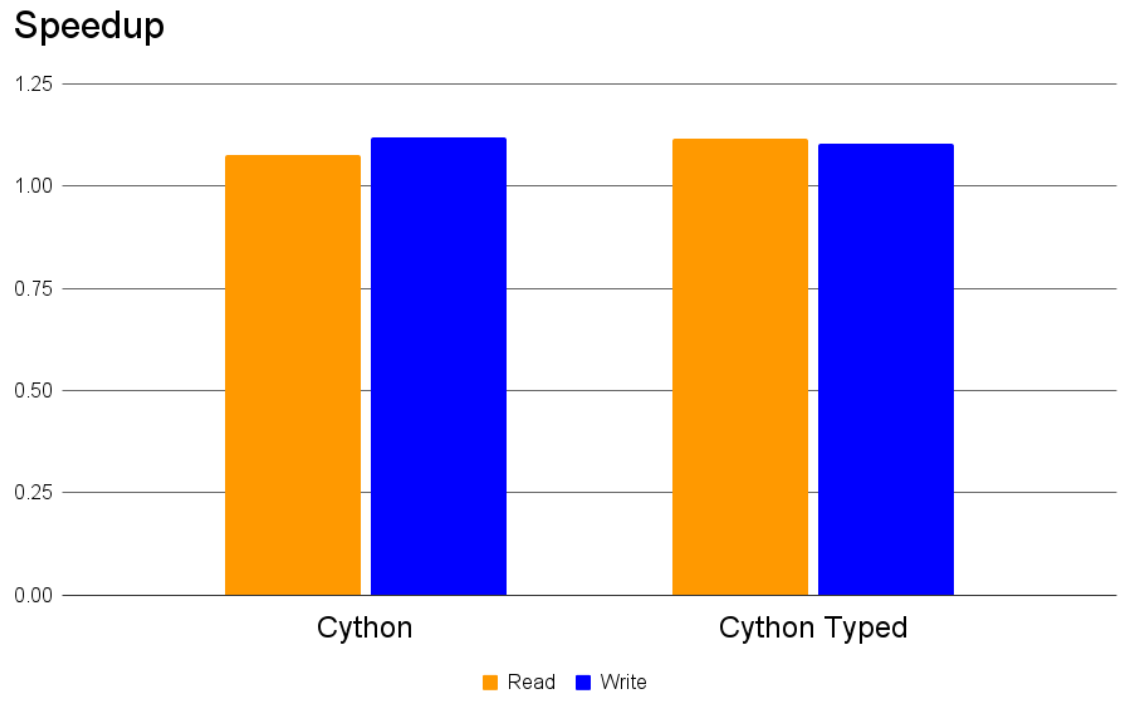

Figure 3: Speedup gained by direct compilation and by compilation of the typed version for the IO-heavy test program.

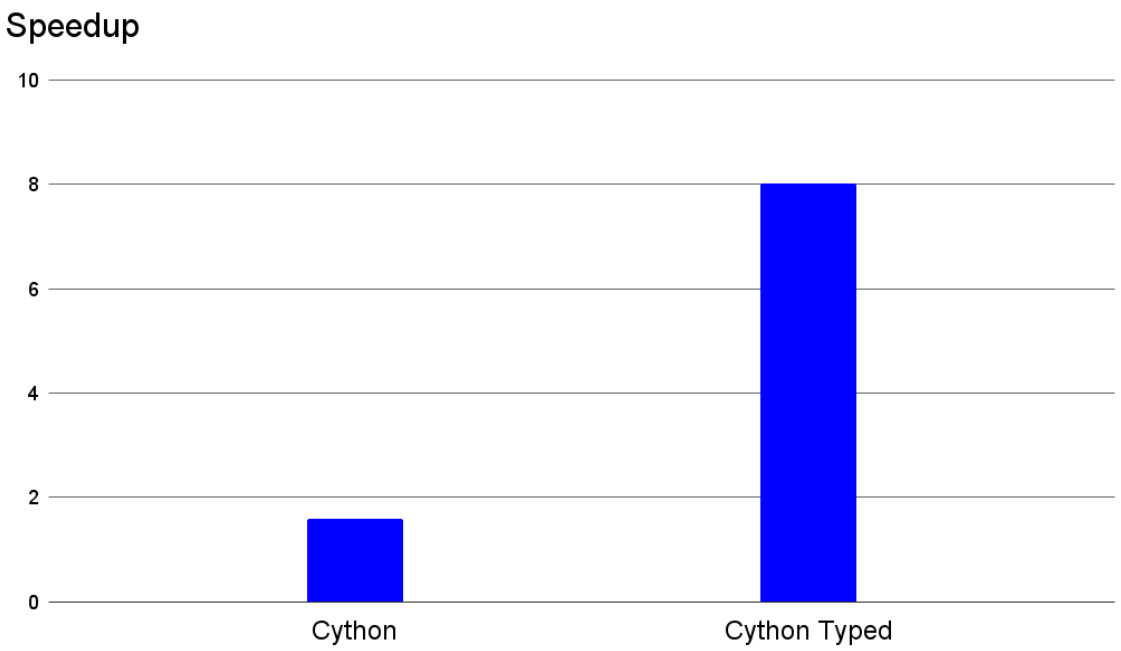

Figure 4: Speedup gained by direct compilation and by compilation of the typed version for Heap Sort.

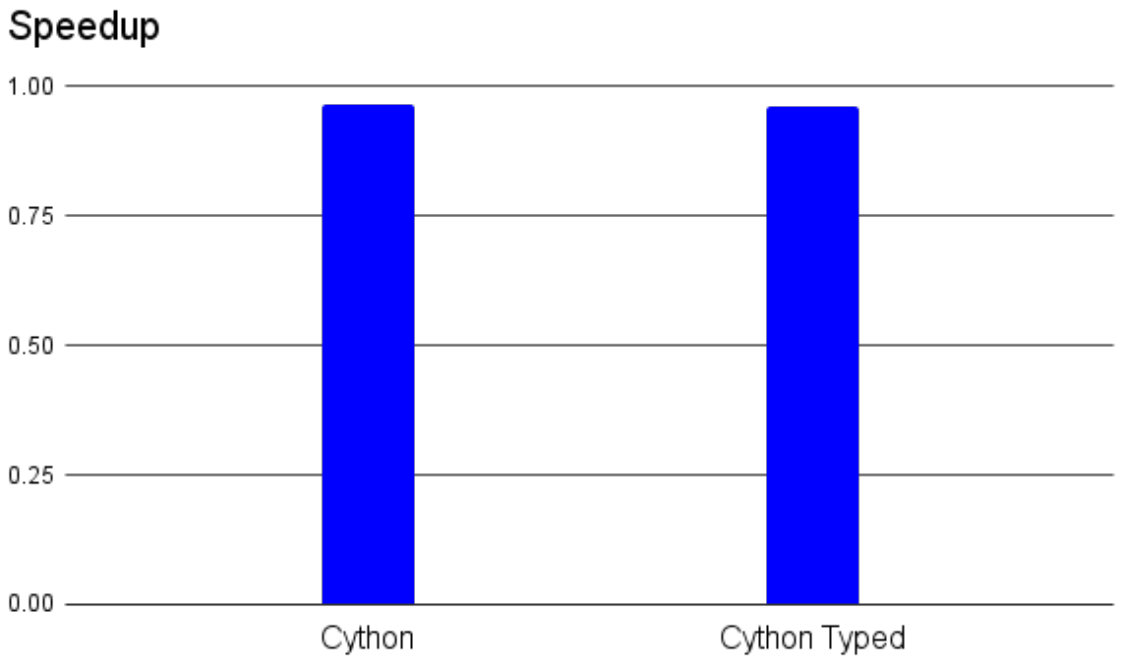

Figure 5: Speedup gained by direct compilation and by compilation of the typed version for Matrix Multiplication program written with Numpy.

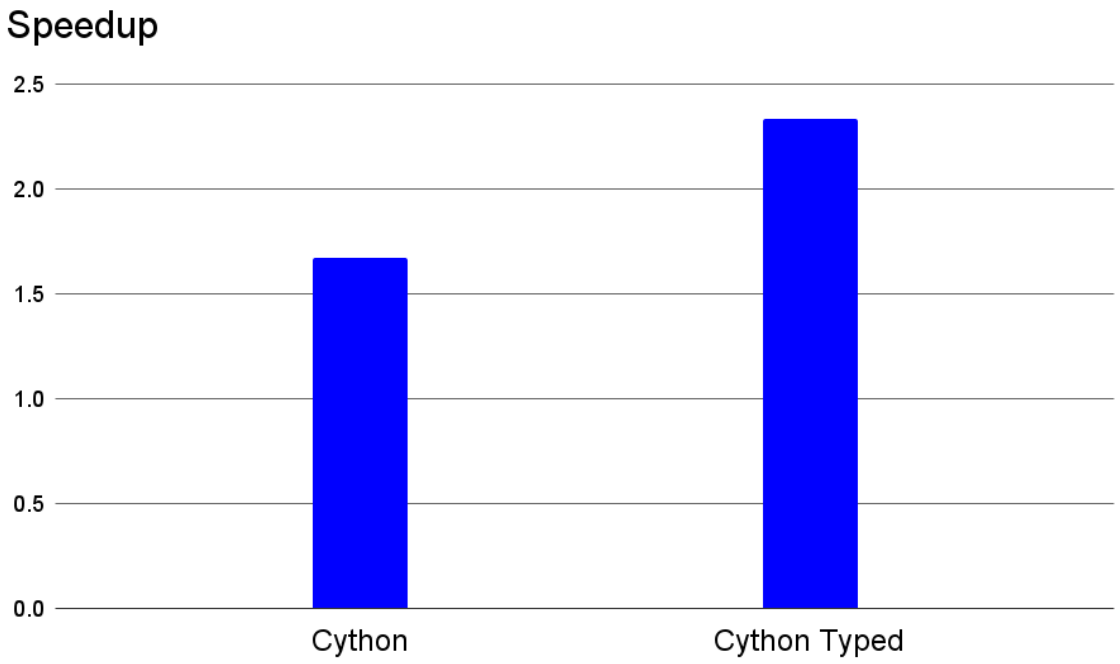

Figure 6: Speedup gained by direct compilation and by compilation of the typed version for the matrix inversion.

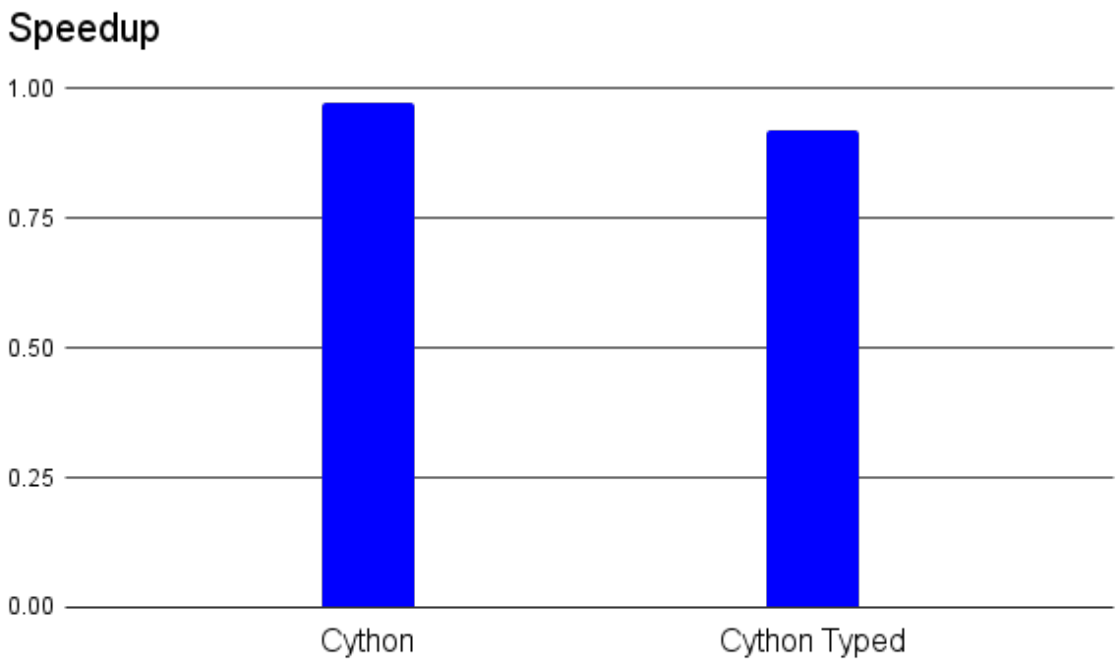

Figure 7: Speedup gained by direct compilation and by compilation of the typed version for the Stochastic gradient descent algorithm.

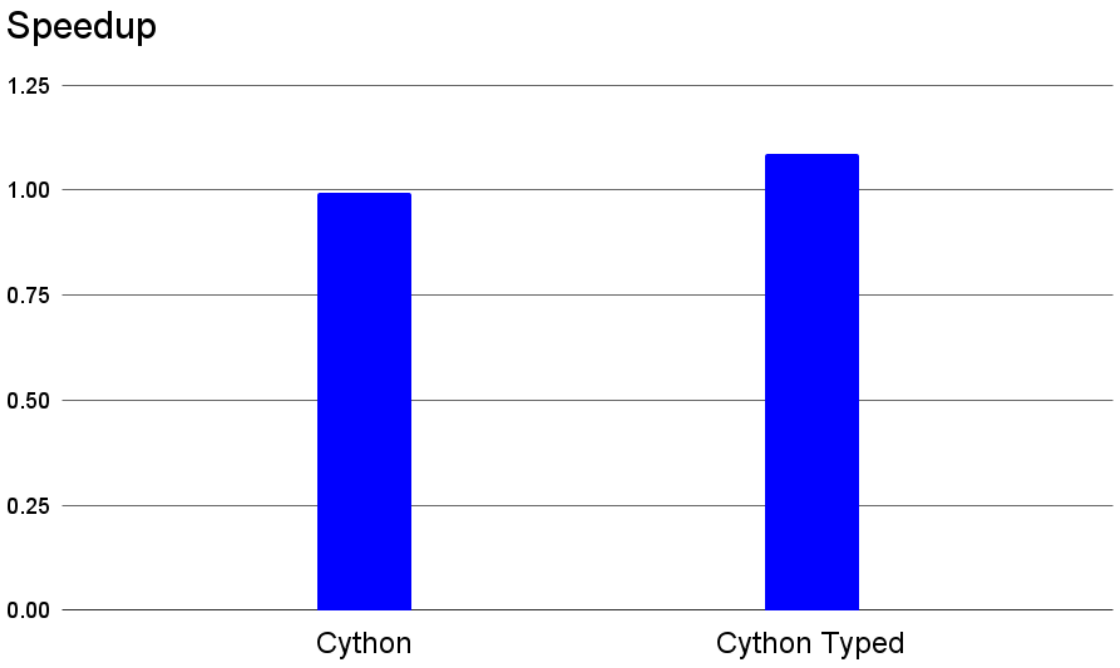

Figure 8: Speedup gained by direct compilation and by compilation of the typed version for the Canny edge detection algorithm.

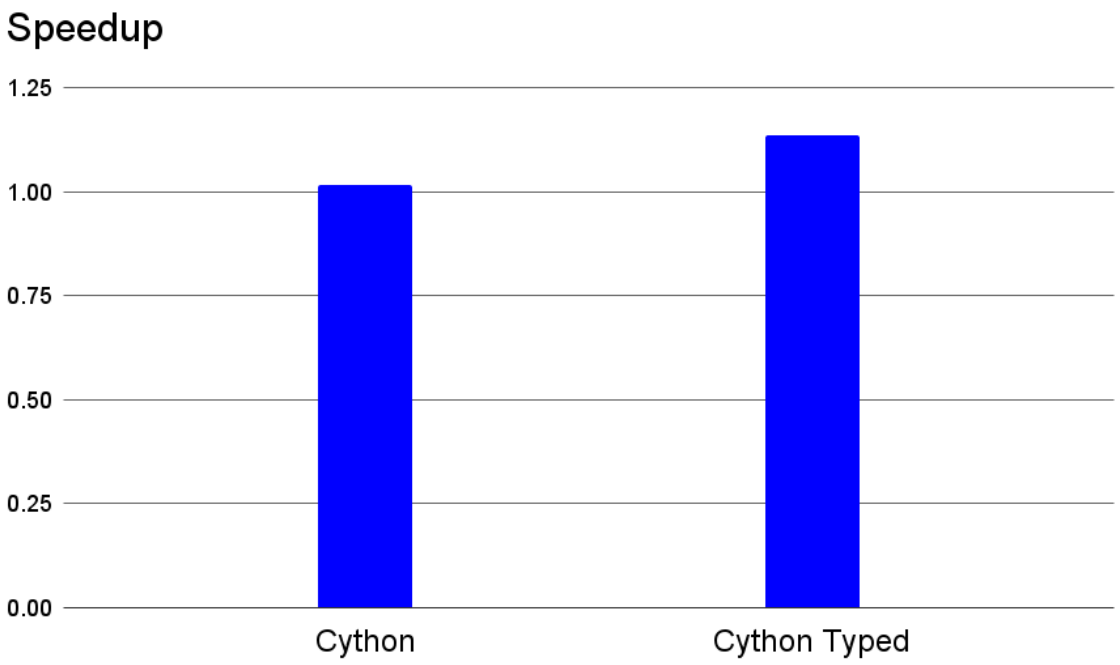

Figure 9: Speedup gained by direct compilation and by compilation of the typed version for the Marr-Hildreth edge detection algorithm.

Code

```
cdef int[:] create_array(int n):
    cdef int[:] arr = array.array('i', [0] * n)
    cdef int i
    for i in range(n):
        arr[i] = (random.randint(0,20000))
    return arr
```