Evaluating the performance of high-level approaches to speeding up slow code

Ove Haugvaldstad, Eirik Gallefoss

December 19, 2019

Abstract

We investigate high-level approaches to increasing the speed of python code using Numba and Cython, using an implementation of Jacobis method for finding eigenvalues as a benchmark. The results are also compared to an c++ implementation using armadillo. We show that python code containing heavy use of loops can gain a considerable speed up by using Numba and Cython, with Numba requiring the least amount of effort.

Introduction

High-level programming languages have in the recent years become increasingly popular in scientific programming. The high-level syntax renders the code very readable, allowing the code for instance to closely resemble mathematics. The nice syntax combined with excellent and easy to use libraries for plotting and mathematics makes Python or Matlab the programming languages of choice for many scientists. The nice high-level features does not come without penalty, for instance running large loops in Python or Matlab is known to be notoriously slow. Therefore in high performance computing languages low level languages like C++/C or Fortran are still the standard.

Fortunately there exists solutions which tries to offer both the readable syntax and ease of use of a high-level language and performance of low level languages. We will look at three different solutions, the just in time (JIT) compiler for python Numba, Cython which is a C based extension of python and Armadillo a library for scientific computing a in c++ offering a more high level syntax.

First we will give an short description of Numba, Cython and Armadillo. Next we will describe our experimental setup, where we implement the Jacobi method for finding eigenvalues using the aforementioned approaches and benchmarking them against a standard python approach. The specific problem we will solve is the wave equation for a buckling beam.

Numba

Numba is an open source just in time (JIT) compiler which translates a subset of python code into fast machine code. Currently Numba works best with NumPy, functions and loops. Enabling numba in a python script is very simple and is done by simply typing in a @jit declaration above the python function you want to speed up. If the @jit(nopython = true) Numba will compile the entire python function without involving python at all, resulting in maximum speed up. Still to gain any increase in speed at all, the code needs to consist of numpy functions or loops.

Cython

Cython is an optimising static compiler for Cython, which is a superset of the python programming language. The cython compiler translate Python into C

code by adding static type decelerations. Cython offers better readability of high-level python code combined with the speed of low level C and C++. This makes Cython ideal for creating fast modules which can speed up otherwise slow python code. One of the great advantages with Cython when developing code compared to for instance c++ or C, is that since Cython is a superset of Python we can actually start with our slow Python code and then optimize the code iteratively. Compare this approach to translating an entire python program into c++/c, which will probably lead to mistakes during translation and unnecessary time spent debugging.

Armadillo

Armadillo is an open-source linear algebra library for c++. Armadillo aims to be efficient while at the same time providing an easy-to-use interface. Armadillo has a large variety of different linear algebra functions provided through integration of the linear algebra package (LAPACK).

Methods

The physical problem we will look at is the one dimensional wave function

$$\gamma \frac{d^2u(x)}{dx^2} = -Fu(x) \tag{1}$$

with u(x) being the vertical displacement of a beam in the y direction, where x $\in [0, L]$, with L being the length of the beam. γ is a material constant, and F is a force towards the origin being applied at the right hand side of the beam. We will use the boundary conditions u(0) = u(L) = 0, meaning that the beam is fixed at the end points.

Since we want to solve this equation numerically we scale it. Introducing $\rho = \frac{x}{L}$ limits ρ to $[0, 1]$ and scales the equation to

$$\frac{d^2u(\rho)}{d\rho^2} = -\frac{FL^2}{\gamma}u(\rho) = -\lambda u(\rho).$$

We discretize ρ on a grid with N+1 points, fixing the step length as $h = \frac{\rho_N - \rho_0}{N}$ with $\rho_N = 1$ and $\rho_0 = 0$.

Using the three point formula for the second derivative $u'' = \frac{u(\rho+h)-2u(\rho)+u(\rho-u))}{h^2} + O(h^2)$, the initial equation can be discretized as $\frac{-u_{i+1}+2u_i-u_{i-1}}{h^2} = \lambda u_i$ or, in ma-

trix form $A\vec{u} = \lambda\vec{u}$ with A being a tridiagonal matrix with $a = \frac{-1}{h^2}$ on the upper and lower diagonal, and $d = \frac{2}{h^2}$ on the diagonal. $\vec{u}^T = [u_1, u_2, ..., u_{n-1}]$

Our equation is now in the form of an eigenvalue problem, which in this case has analytical eigenvalues $\lambda_j = d + 2a\cos(\frac{j\pi}{n+1})$ for $j \in [1, 2, ..., n]$ where n is the size of A. [1]

Properties of orthogonal transformations

An orthogonal transformation U has the property $U^TU = UU^T = I$.

Assuming we have an orthonormal basis consisting of $\vec{v}_i^T = [v_{1i}, v_{2i}, ..., v_{ni}]$ we know that $\vec{v}_i^T\vec{v}_j = \vec{v}_i \cdot \vec{v}_j = \delta_{ij}$. Let $\vec{w}_i = U\vec{v}_i$. $\vec{w}_i^T\vec{w}_j = (U\vec{v}_i)^T(U\vec{v}_j) = \vec{v}_i^TU^TU\vec{v}_j = \vec{v}_i^T\vec{v}_j = \delta_{ij}$. This shows that a orthogonal transformation preserves the dot product and orthogonality.

Jacobi´s method

The idea behind Jacobi´s method is to diagonalize A by applying an orthogonal rotation R^T a repeated number of times. By choosing a specific angle θ we can zero out one element of the transformed matrix. We will use the shorthand c = $\cos\theta$, s = $\sin\theta$ and t = $\tan\theta$ where θ is the angle of rotation.

$$R(k,l,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

where k and l is the row number containing c and -s, and s and c respectively.

After one transformation $R^TARR^T\vec{x} = B(R^T\vec{x}) = \lambda(R^T\vec{x})$ we see that the new eigenvector is $R^T\vec{x}$.

Performing the transformation results in the following matrix [1].

$$
\begin{aligned}
b_{ii} &= a_{ii}, i \neq k, i \neq l \\
b_{ik} &= a_{ik}c - a_{il}s, i \neq k, i \neq l \\
b_{il} &= a_{il}c + a_{ik}s, i \neq k, i \neq l \\
b_{kk} &= a_{kk}c^2 - 2a_{kl}cs + a_{ll}s^2 \\
b_{ll} &= a_{ll}c^2 + 2a_{kl}cs + a_{kk}s^2 \\
b_{kl} &= (a_{kk} - a_{ll})cs + a_{kl}(c^2 - s^2)
\end{aligned}
$$

Since A is symmetric, the orthogonal transformation of A will also be symmetric which leads to

$$
\begin{aligned}
b_{ki} &= b_{ik} \\
b_{li} &= b_{il} \\
b_{lk} &= b_{lk}
\end{aligned}
$$

From this we see that the only changes between B and A will be in the columns and rows k,l, and we will not have to perform the full matrix multiplication.

Setting b_{kl} equal to zero and defining $\tau = \frac{a_{kk}-a_{ll}}{2a_{kl}}$ gives

$$
\begin{aligned}
\tau &= -\frac{cos(2\theta)}{sin(2\theta)} \\
&= \frac{1}{tan(2\theta)}
\end{aligned}
$$

Solving for t leads to $t^2 + 2t\tau - 1 = 0$, with roots $t = -\tau \pm \sqrt{(\tau^2 + 1)}$. We will use this equation to fix θ to the angle that eliminates element a_{kl}. Having fixed the angle we find $c = \frac{1}{\sqrt{(1+t^2)}}$ and $s = tc$, which we need to build the transformation matrix.

We need a strategy for choosing which element to eliminate. Our goal is to transform A into a diagonal matrix, meaning all offdiagonal elements are zero (or very close to zero). We can achieve this by minimizing the sum of the squared nondiagional elements, off(B)^2. Since the frobenius norm is preserved under an orthogonal transformation we must have

$$
|B|_F^2 = |A|_F^2 = off(B)^2 + \sum_i^n b_{ii} = off(A)^2 + \sum_i^n a_{ii}.
$$

Utilizing this we show that

$$\begin{aligned} off(B)^2 &= |B|_F^2 - \sum_{i=1}^{n} b_{ii}^2 \\ &= |A|_F^2 - \sum_{i=1}^{n} b_{ii}^2 \\ &= |A|_F^2 - \sum_{i \neq k \neq l}^{n} b_{ii}^2 - (b_{kk}^2 + 2b_{kl}^2 + b_{ll}^2) \\ &= |A|_F^2 - \sum_{i \neq k \neq l}^{n} a_{ii}^2 - (a_{kk}^2 + 2a_{kl}^2 + a_{ll}^2) \\ &= |A|_F^2 - \sum_{i=1}^{n} a_{ii}^2 - 2a_{kl}^2 \\ &= off(A)^2 - 2a_{kl}^2 \end{aligned}$$

This means that the difference in off(A)^2 and off(B)^2 is the greatest when we choose k and l to correspond with the largest offdiagonal element in A.

Our algorithm is then:
while $a_{ij}^2 > \epsilon$ **do**
    calculate: τ, t, s, c
    Perform transformation
    Find largest nondiagonal element
end while
Read off diagonal elements to find eigenvalues

Our implementation of the jacobi method can be found at our github [2].

Experimental setup

To have our benchmark consistent every run was conducted using the same pc. We ran each implementation was ran five times for different n ranging from 0 to 100. We also ran a python version for the same values of n. For n larger than 100 the python version was so slow that we decided to only run the other implementations. To investigate performance for large n we ran the c++, Numba and Cython implementations for n ranging from 100 to 350 and for each n we ran the program five times. We only timed the part of the code which calculated the eigenvalues.

Results

The implementation in cython, numba and c++ was run for matrix sizes n ranging from 5 to 350. Since the python version was quite slow it was only run with n in the range of 5 to 100. All values used are average values over 5 runs.

Number of iterations

We will first look into the number of transformations needed to make all off diagonal elements smaller than the tolerance level ϵ, in our case $\epsilon = 10^{-9}$, and find an estimate as a function of n, with n being the dimension of the square matrix we are diagonalizing. In this part we will analyze the results from the c++ implementation of Jacobis method, and compare our results with results from the literature of between $3n^2$ - $5n^2$ [1] transformations.

To look into the n dependence fig. 1 shows the number of transformations scaled by n^2. The ratio flattens for n larger than 100. The horisontal line shows the mean value of the ratio for n larger than 200. Using the calculated mean value a possible equation for transformations as a function of n is $1.56 \cdot n^2$.

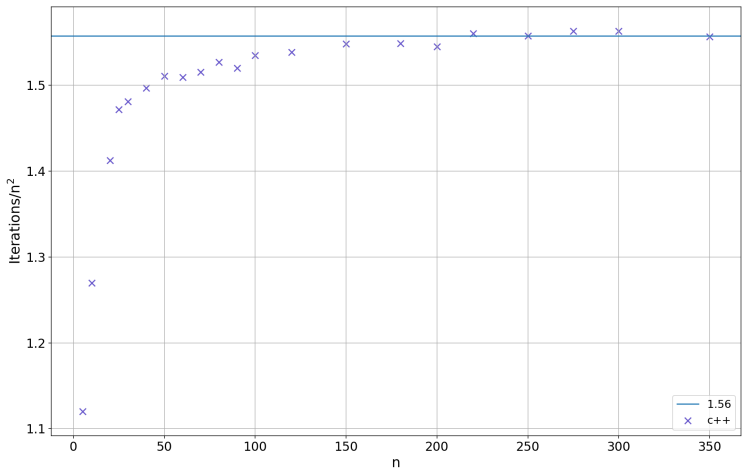

Figure 1: Runs of the Jacobi method implemented in c++. Number of iterations needed to make all offdiagonal elements smaller than 10^{-9} as function of matrix size n. The horisontal line shows the mean value of iterations for n $\geq$ 200.

Figure 2 shows the actual number of transformations, our estimated relation

and the values from the literature. This seems to be a good fit to the data. Our results shows a faster convergence than the expected values, but we are unsure of the tolerance level that was used. If we used a lower tolerance our number of transformations would be higher.

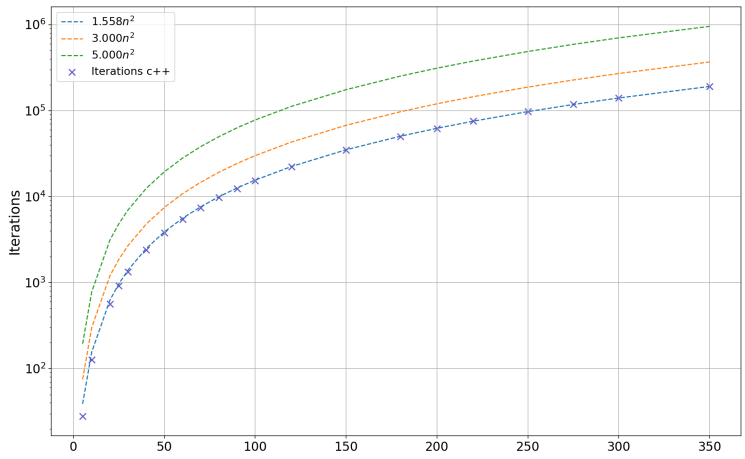

Figure 2: Runs of the Jacobi method implemented in c++. Number of iterations needed to make all offdiagonal elements smaller than 10^{-9} as function of matrix size n.

Timing Results

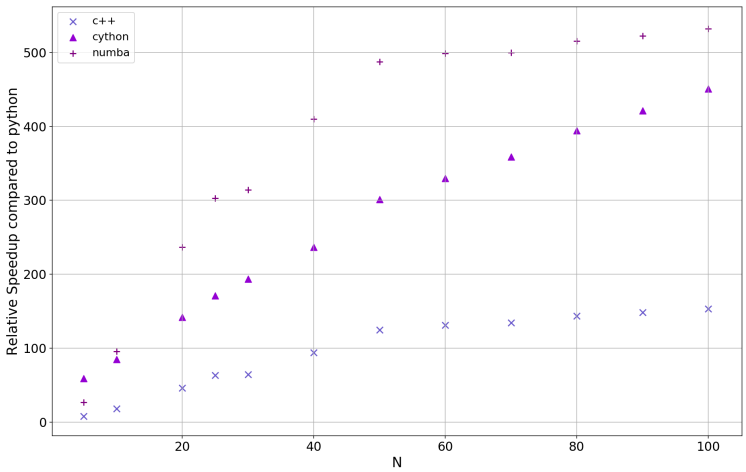

Figure 3: The average timing of 5 runs, divided by the average of the standard python timings. Starting from $n = 5$ until $n = 100$.

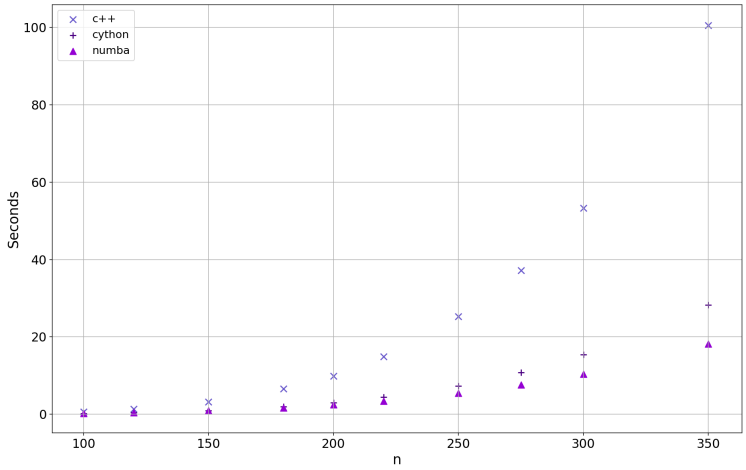

Figure 4: Average timing of 5 runs for the jacobi method implemented in c++, cython, numba. Starting from $n = 100$ until $n = 350$.

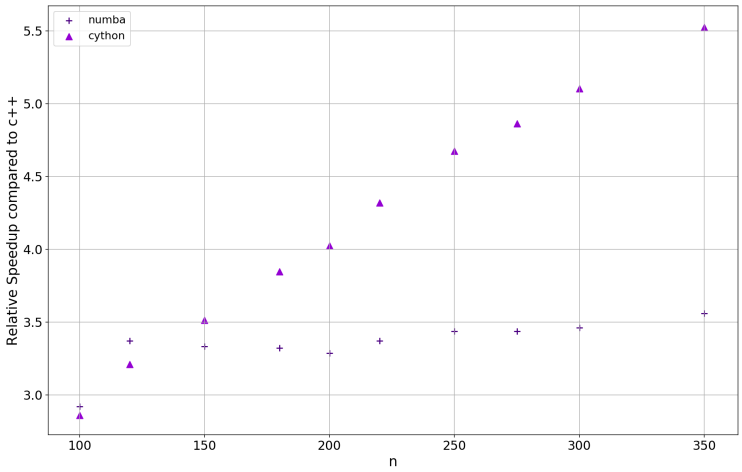

Figure 5: Relative speed up compared to c++

Discussion

The results of benchmark we showed a huge increase in performance with Numba, Cython and Armadillo c++. Performing between 100 to 500 times faster than the standard python implementation. For n less than 100 the Numba version were the fastest being around 20 % faster than Cython and around 50 % faster than Armadillo (fig. 3). This may be due to the efficiency of machine code compiled by Numba jit. For large n the Cython version showed the best performance being around five times faster than Armadillo. The Numba version was around 3.5 times faster than Armadillo (fig. 5). The Numba version did stay constant at around 3.5 times the speed of Armadillo. While Cython version show a steady increase in relative speed up of around from 3 for n = 100 to 5.5 for n = 350.

Surprising to us was that both the Numba and Cython beat Armadillo by such a large margin. The performance of Numba was especially impressive considering that only change we did compared to the baseline python code was adding the @jit(nopython=True) deceleration at the top of our functions. This results may be consequence of the high level features that Armadillo offers, which could compromise some performance. This something that would be interesting to investigate further by comparing an pure C++ implementation with the Armadillo implementation.

Conclusion

We have investigated the performance of Cython and Numba, tools to easily optimise python code and compared them with Armadillo. From our result we have shown that it is possible to achieve low-level performance with a high-level language. We where able to make our baseline python Jacobi's method implementation around up to 500 times faster through only slight modification and even beating our c++ Armadillo implementation.

Cython and Numba are quite different. With Numba the @jit compiler does all the work for you, which means doing any further optimization beyond exploits in the physical problem is difficult. Cython offers the user much more control (and more work), giving much more flexibility in terms of optimizations. This is something we think is important to consider when making optimizations. If you have larger and more complex program, then you might achieve the best result by using Cython or Armadillo. If instead you only need to optimise a small part of the code then Numba might be the best choice, offering great performance for little effort.

References

1. Hjort-Jensen, H. *Computational physics, lecture notes fall 2015* https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf.
2. Haugvaldstad, O. & Gallefoss, E. *Source files for this project* https://github.com/Ovewh/Computilus/tree/master/Project2.