Univerza v Ljubljani
Fakulteta za elektrotehniko

Matrix multiplication via NumPy

Name: Danilo Pejović

Date: 06.02.2022

Abstract

This paper will provide an overview of matrix multiplication in Python. It will aim to explain the difference between compiled and scripted languages, as well as explain a few basic algorithms for matrix multiplication. This paper will explain the difference in how objects are stored/initialized in both types of languages and how they are used during calculations.

In order to sufficiently explain such a topic, it will very briefly touch on problems of “starving” CPUs, processor design, vectorization, cache sizes, and financial limitations that designers of commonly used processors are bound by.

1. Introduction

The best starting point in understanding how NumPy works is understanding the reasons why we need external libraries in the first place. Let's try and figure out what happens when we run a python loop over matrices:

```
def regular_multiplication(X,Y):


    result = [[sum(a * b for a, b in zip(X_row, Y_col)) for Y_col in zip(*Y)] for X_row in X]
```

Figure 1. Matrix multiplication in python

Python is an interpreted language – meaning that the interpreter goes line by line and translates code to machine language. The biggest drawback of interpreted languages is execution time – as the interpreter doesn't see the "bigger picture" of the code. In our, very naive, code interpreted would have to run (at least) the following checks every time it does through this loop:

- To what memory address do I need to jump to find the current element of the array
- What file type are variables a and b
- What does operator "*" do for that file type

Memory allocation for matrices is also far from perfect, severely limiting the use of cache memories – meaning that most of the time will be spent in bringing variables from the main memory into registers while only a negligible amount of time will be spent doing the calculations. Unsurprisingly, such an algorithm gave me bad results and it took me as much as two minutes to multiply two 1000x1000 matrices. That said, interpreted languages are not always inferior to compiled languages – they work across systems better and don't need that extra step of compilation in order to work – but in programs where you need to do a lot of relatively simple operations on data of the same type, they tend to perform orders of magnitude worse than compiled languages.

On the other hand, compiled languages get translated into machine code before they are used. In C arrays can only contain objects of the same type and need to be initialized before they are used – meaning that the program is able to have efficient memory allocation. Knowing what type objects are in advance will mean that operator is known in advance and won't have to be translated during every loop.

NumPy is a package for scientific computing in Python, that includes multidimensional array objects, various derived objects, and routines for fast operations on those array objects. NumPy calculations are actually run in C and there are few key differences between NumPy arrays and Python sequences [1], mainly making NumPy arrays act like C arrays (having fixed size and having files all be of the same type that is determined on creation). This allows for more efficient memory allocation, vectorization, and more efficient use of cache memory.

2. Memory allocation in NumPy

Having to "jump around" in memory to find the next member of our array is very inefficient and that is why programs will want to employ concepts of spatial locality – where a dataset is accessed sequentially from memory. On a fundamental level, an N-dimensional array is just a one-dimensional sequence of memory with indexing that maps that N-dimensional array into a one-dimensional index [2]. Next picture shows an example of the most commonly used array types and their mapping of memory sequence:

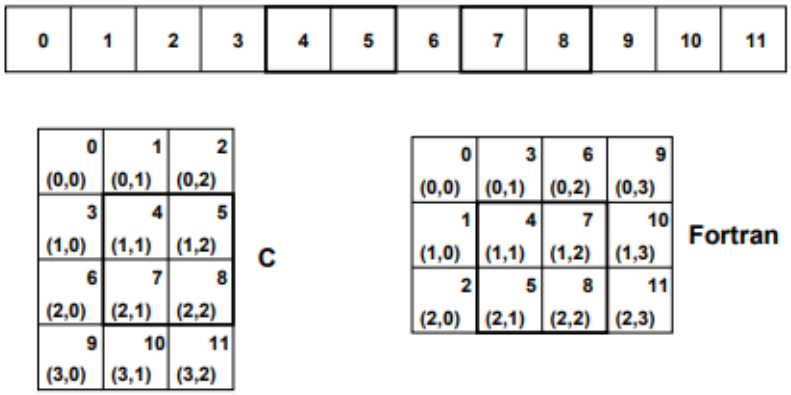

Figure 2. Mapping of C array and Fortran array [2]

Each numbered block represents one object and in bottom left corner of each block in the matrix is indexing of that block. In the C array (left) we can see that the last index varies fastest, when we are dealing with matrices that would mean that we move row-wise. Fortran array is opposite and the first index varies fastest, meaning that we access a matrix column-wise. It is important to use the right array type and as a quick demo I summed C array row-wise and collum-wise and got the following results:

```
1641777944.3962674
135548
1641777945.0779657
Time Elapsed:  0.6816983222961426
```

```
1641777891.256739
134990
1641777891.2948325
Time Elapsed:  0.03809356689453125
```

Figure 3. Summing 10000x10000 array row (right) and columns (left) wise

The reason for that is that in column-wise sum it doesn't get to access memory sequentially – it has to access memory addresses in increment of 3 and then return to the start of next column when it reaches the end of the current column.

As we know that during matrix multiplication matrices are accessed differently (one is accesses row-wise and one column-wise) simply having matrices be of different array styles is the first step towards optimizing the performance of matrix multiplication. Having tested out naïve implementation of matrix multiplication in I got a performance that is considerably better than Python code but still ways off

performance NumPy gives.

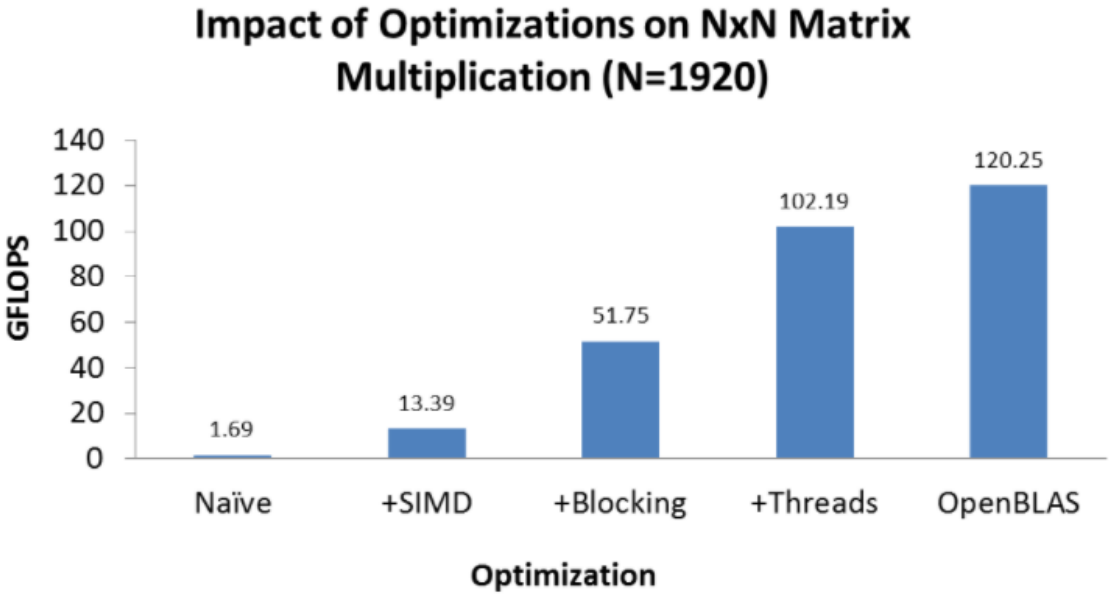

Figure 4. Comparison of matrix multiplication implementations [3]

We have covered SIMD (single instruction multiple data) processors during our lectures, and that implementation (that most modern processors support) makes it so that algorithm works on a set of values at the time rather than on a single value. This optimization is called vectorization and is heavily used by NumPy.

3. Blocking Algorithm for matrix multiplication

For the last 20 years, the biggest bottleneck in the performance of CPUs has not been frequency on our CPU cores, but rather a memory operation [4]. Memory operations on addresses in main memory can be up to 1000 times slower than floating-point operations when data is already in the registers. Meaning that the performance of our algorithm will mostly depend on how efficiently we do memory operations. In order to be as efficient as possible, we would need to have as few operations from main memory to registers as possible and do all of our memory operations on objects in Layers 1 or Layer 2 cache. That is a problem with the naïve implementation of matrix multiplication. For naïve implementation (where we loop through the whole matrix) to work, the program will need access to whole matrices simultaneously and there is simply not enough space in cache memory to store big objects there. 32 kB can only roughly fit three 40x40 matrices. That results in us bringing basically all our objects directly from the main memory to registers which is a very expensive process. One of the more efficient ways to solve those problems is the use of blocking algorithms [5].

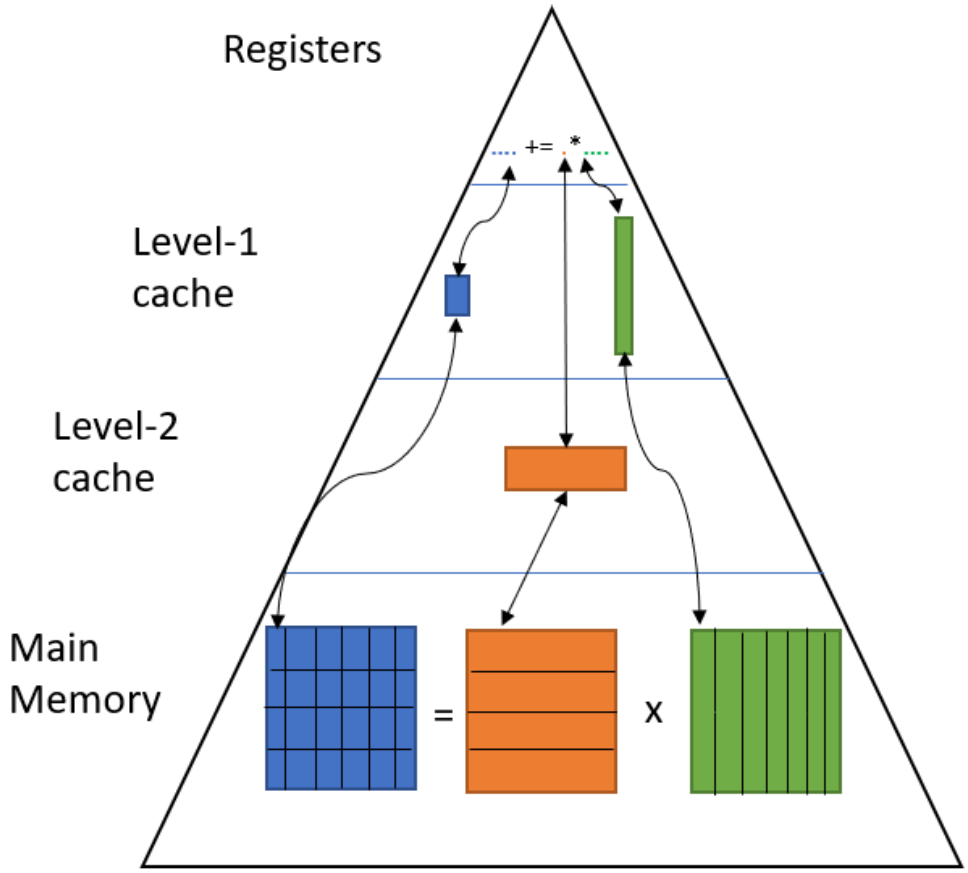

Figure 5. Sketch of a blocking algorithm

In Figure 5. we have a sketch of how the blocking algorithm works on CPU with L1 and L2 cache. The size of each block depends on the size of L1 and L2 cache on a CPU – blocks should be as big as memory available allows.

As an example, I will use a relatively simple situation that can efficiently fit both matrices A block in L2 and have a (smaller) slice of B matrix in L1 cache. In my example slice of B is 4 objects wide, so our algorithm will be to go row-wise through a slice of matrix A (1 object at a time) and go column-wise through matrix B (4 objects at a time). Doing that will give us matrix multiplication of those slices, which we will then save at the corresponding spot in matrix C. Since L2 cache is slower than L1 cache, we will want to limit writing in L2 cache as much as possible – so we will not remove a slice of matrix A from L2 cache until we have gone through every slice from matrix B and recorded our calculations in matrix C.

The method behind the process doesn't change when we are dealing with a more complicated situation where rows of A can't fit into L2 cache and columns of B can't fit in L1 cache – we will then split slices from my example into even more slices and go trough them in a similar fashion [6]. In that example, we will have to do more memory operations since we will have to reuse slices but as seen in Figure 4 blocking brings enormous benefits in our calculations.

4 . References

[1] NumPy user guide; NumPy; 2021

https://numpy.org/doc/stable/user/whatisnumpy.html

[2] Guide to NumPy; Travis E. Oliphant; NumPy; 7.12.2006

[3] BLAS-level CPU Performance in 100 Lines of C; Stefan Hadjis; Stanford; 2015

[4] Why modern CPUS are starving and what can be done about it; Francest Alted; IEEE Computer society

[5] A high-performance matrix-matrix multiplication methodology for CPU and GPU architectures; Kelefouras Vasilios, Angeliki Kritikakou, Iosif Mporas, Vasilios Kolonias; Sheffield Hallam University; 2016

[6] LAFF-on Programmin for High Performance; Robert van de Gejin, Margaret Myers, Devangi Parikh; ULAFF; 25.11.2021; https://www.cs.utexas.edu/users/flame/laff/pfhp/