(19 - 22)-11-2021

Implementation of Square Matrix-Matrix Multiplication with Python

By: Oluwatosin S. Oluseyi

Key words: matrix-matrix multiplication, square matrix, loop unrolling, optimization, python, numpy.

About: The overall aim of this experiment is to compare the timing performance of a looped native and optimized (loop optimization technique used is loop unrolling) square matrix-matrix multiplication implementation with Python's numpy.dot() function matrix-matrix multiplication implementation. Specifically, the experiment aims to achieve the following objectives:

1. Execute a native square matrix-matrix multiplication with arrays assigned using list of lists;
2. Execute a native square matrix-matrix multiplication with arrays assigned using numpy.ndarrays;
3. Execute an optimized square matrix-matrix multiplication with arrays assigned using list of lists;
4. Execute an optimized square matrix-matrix multiplication with arrays assigned using numpy.ndarrays;

5. Execute a matrix-matrix multiplication with numpy.dot() function;

Python Environment: IDLE 3.10.0

Machine specification:

a. Computer model: HP-15-R029WM.

b. Processor: Intel® Pentium® CPU N3540 @2.16GHz, 4 Core(s), 4 Logical Processor(s)

c. Installed RAM: 4.00GB

d. System type: 64-bit operating system, x64-based processor

Definition of terms:

What is Matrix-Matrix Multiplication (MM)? There are two ways of considering a matrix-matrix multiplication -

1. **As a linear combination of matrix-vector products:** Suppose **A** is a $\boldsymbol{m} \times \boldsymbol{n}$ matrix and $\mathbf{B_1}, \mathbf{B_2}, \mathbf{B_3}, \ldots, \mathbf{B_p}$ are the column entries of a $\boldsymbol{n} \times \boldsymbol{p}$ matrix B. Then the matrix product of **A** with **B** is the $\boldsymbol{m}\times\boldsymbol{p}$ matrix where column i is the matrix-vector product ABi.

$$AB = A\,[B_1|B_2|B_3| \ldots |Bp] = [AB_1|AB_2|AB_3| \ldots |ABp] \tag{i}$$

For example =

$$\text{Let } A = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \text{, and, let } B = \begin{bmatrix} 1 & 2 \\ 3 & 2 \end{bmatrix}$$

Then, AB = [A [B(column 1)] | A [B(column 2)]]

$$AB = [\, A \begin{bmatrix}1\\3\end{bmatrix} \quad A \begin{bmatrix}2\\2\end{bmatrix} \,]$$

Let's first consider column 1 of AB : A $\begin{bmatrix}1\\3\end{bmatrix}$, this is : $\begin{bmatrix}2 & 3\\4 & 5\end{bmatrix}$ x $\begin{bmatrix}1\\3\end{bmatrix}$ =

$$1\begin{bmatrix}2\\4\end{bmatrix} + 3\begin{bmatrix}3\\5\end{bmatrix} = \begin{bmatrix}(1x2)+(3x3)\\(1x4)+(3x5)\end{bmatrix}=\begin{bmatrix}2+9\\4+15\end{bmatrix}=\begin{bmatrix}11\\19\end{bmatrix}$$

Now, column 2 of AB: A $\begin{bmatrix}2\\2\end{bmatrix}$, this is : $\begin{bmatrix}2 & 3\\4 & 5\end{bmatrix}$ x $\begin{bmatrix}2\\2\end{bmatrix}$ =

$$2\begin{bmatrix}2\\4\end{bmatrix} + 2\begin{bmatrix}3\\5\end{bmatrix} = \begin{bmatrix}(2x2)+(2x3)\\(2x4)+(2x5)\end{bmatrix}=\begin{bmatrix}4+6\\8+10\end{bmatrix}=\begin{bmatrix}10\\18\end{bmatrix}$$

Therefore, AB = $\begin{bmatrix}11 & 10\\19 & 18\end{bmatrix}$

2. **As a summation of Entry-by-Entry products:** Suppose **A** is a $\boldsymbol{m \times n}$ matrix and **B** is a $\boldsymbol{n \times p}$ matrix. Then for $1 \le i \le m$, $1 \le j \le p$, the individual entries of AB are given by

$$[AB]_{ij} = [A]_{i1}[B]_{1j} + [A]_{i2}[B]_{2j} + [A]_{i3}[B]_{3j} + \cdots + [A]_{in}[B]_{nj}$$

(i = row counter, j = column counter)

$$= \sum_{k=1}^{n} [A]ik \;\; [B]kj$$

(ii)

Using the example matrices in 1, let's treat each entry of the result matrix AB as a summation of entry-by-entry products -

: A = 2 x 2 matrix and B = 2 x 2 matrix, therefore, AB = 2x2 matrix

: Naming each entry of A and B, we have: $\begin{bmatrix} [A]11 & [A]12 \\ [A]21 & [A]22 \end{bmatrix}$, $\begin{bmatrix} [B]11 & [B]12 \\ [B]21 & [B]22 \end{bmatrix}$

AB = $\begin{bmatrix} [AB]11 & [AB]12 \\ [AB]21 & [AB]22 \end{bmatrix}$

: Using the formula in (ii),

Entry of AB in first row, first column -

[AB]11 = [A]11 [B]11 + [A]12 [B]21 = (2 x 1) + (3 x 3) = 2 + 9 = 11

Entry of AB in first row, second column -

[AB]12 = [A]11 [B]12 + [A]12 [B]22 = (2 x 2) + (3 x 2) = 4 + 6 = 10

Entry of AB in second row, first column -

[AB]21 = [A]21 [B]11 + [A]22 [B]21 = (4 x 1) + (5 x 3) = 4 + 15 = 19

Entry of AB in second row, second column -

[AB]22 = [A]21 [B]12 + [A]22 [B]22 = (4 x 2) + (5 x 2) = 8 + 10 = 18

Therefore, AB = $\begin{bmatrix} [AB]11 & [AB]12 \\ [AB]21 & [AB]22 \end{bmatrix} = \begin{bmatrix} 11 & 10 \\ 19 & 18 \end{bmatrix}$

In summary, the multiplication of an A-by-B matrix and a B-by-C matrix is an A-by-C matrix. In a matrix-matrix multiplication, it is necessary for the column size of the first matrix to be equal to the row size of the second matrix, as in :

X[a][b] x Y[b][c] = Z[a][c]. Where X, Y and Z are matrices

X has [a] rows and [b] columns

Y has [b] rows and [c] columns

Z has [a] rows and [c] columns

What is a Square Matrix(SM)? A matrix with equal number of rows and columns.

What is loop unrolling? Loop optimization technique that steps a loop iteration by some integer, $i > 1$, i can either be a multiple of the number of iterations or not. The aim of loop unrolling is to speed up the execution time of a program by reducing the number of instructions to be executed by the loop by n/i, where n = number of loop iterations and i = unroll factor.

Manually, unrolling is applied to a loop by explicitly stating out the instructions being executed in loop iterations using a repetition of similar instructions that do not depend on each other. The instructions being stated are with respect to the unroll factor $(i > 1)$. For example, if a loop needs to execute an instruction 50 times, and an unroll factor of 10 is applied to the step of the loop, the transformed loop only needs to make 5 iterations instead of 50.

Loop unrolling yields more opportunity for parallelization at the instruction level (ILP – Instruction Level Parallelism), since the instructions in the loop iteration are independent of each other, it also increases the efficiency of a program and reduces loop overhead since the loop iterations are reduced. However, it increases the size of a program.

Example:

Original loop:

```
for i in range (n):
        x = x + a[i] + b[i]
```

Transformed loop A: (if unroll factor is a multiple of n)

```
for i in range (0, n, 3):#
        x = x + a[i] + b[i]
        x = x + a[i+1] + b[i+1]
        x = x + a[i+2] + b[i+2]
```

Transformed loop B: (if unroll factor is not a multiple of n

```
m = n – (n % 3)
for i in range (0, m, 3):
        x = x + a[i] + b[i]
        x = x + a[i+1] + b[i+1]
        x = x + a[i+2] + b[i+2]
#tail loop to finish the last iteration
for i in range (m,n,1):
        x = x + a[i] + b[i]
```

Algorithm: Pseudocode of square matrix-matrix multiplication

```
procedure square_matrix_multiply(A, B, C, n)
    for i = 0 to n do
        for j = 0 to n do
            Cij = 0
            for k = 0 to n do
                Cij = Cij + Aik * Bkj
            end for
        end for
    end for
    return C
end procedure
```

Where: (for the square matrix)

n = size of row or column;

A = input matrix A of size n x n;

B = input matrix B of size n x n;

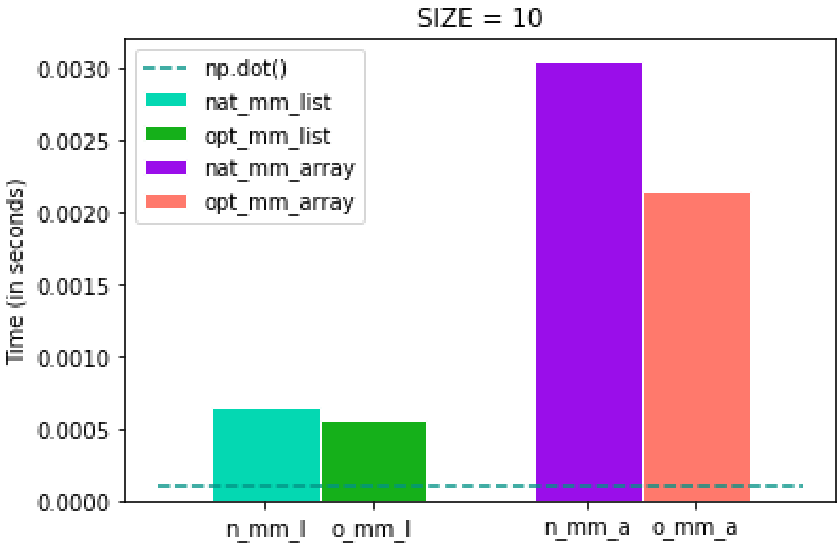
C = output matrix C of size n x n;

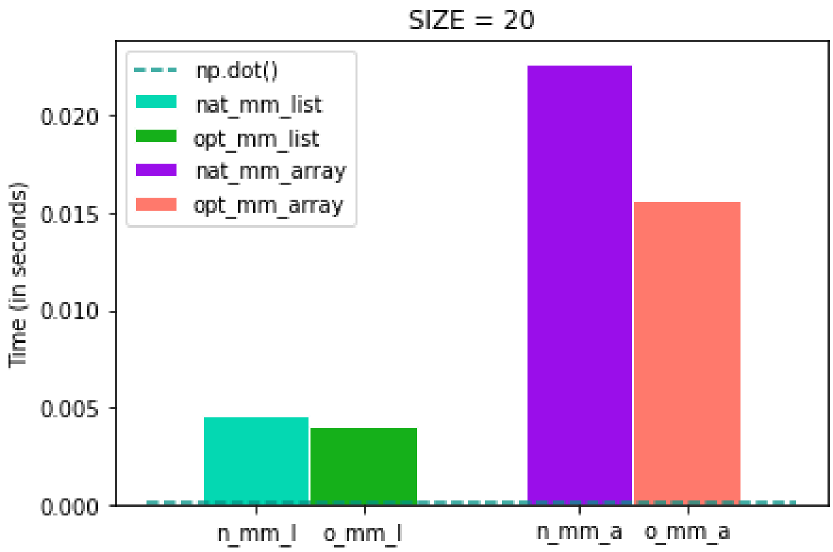
Tabular representation of results

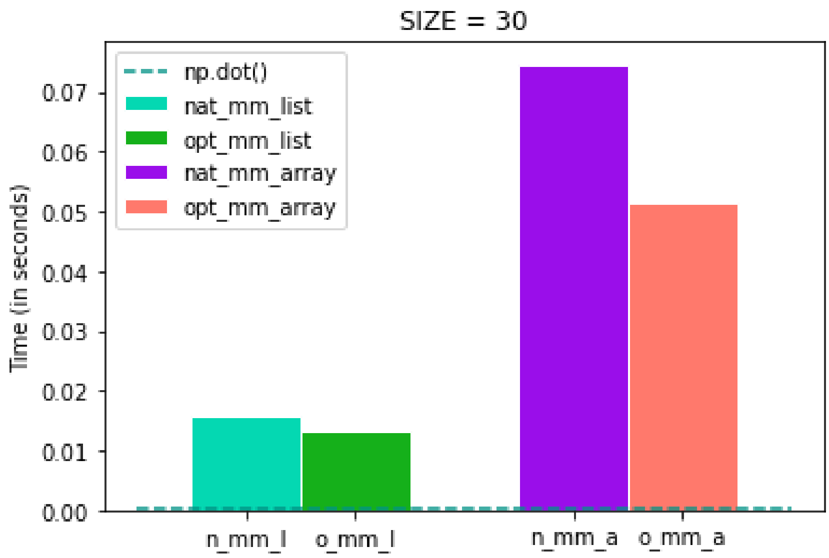
Rounding = Use 4 significant figures						
Size = 10						
Algorithm and Function	Array assignment	Unroll factor	T1 (seconds)	T2 (seconds)	T3 (seconds)	Tavg (seconds)
Native MM	List of Lists	-	0.0006491	0.0006652	0.0006405	0.0006516
Optimized MM	List of lists	5	0.0005675	0.0005519	0.0005582	0.0005592
Native MM	np.array	-	0.003014	0.003115	0.003031	0.003053
Optimized MM	np.array	5	0.002162	0.002147	0.002158	0.002156
Numpy.dot()	np.array	-	0.00009509	0.0001314	0.00009329	0.0001066
Size = 20						
Native MM	List of Lists	-	0.004634	0.004630	0.004631	0.004632
Optimized MM	List of lists	10	0.004117	0.003975	0.003991	0.004028
Native MM	np.array	-	0.02252	0.02255	0.02291	0.02266
Optimized MM	np.array	10	0.01562	0.01566	0.01574	0.01567
Numpy.dot()	np.array	-	0.0001121	0.0001609	0.0001097	0.0001276
Size = 30						

Native MM	List of Lists	-	0.01559	0.01587	0.01549	0.01565
Optimized MM	List of lists	15	0.01321	0.01352	0.01321	0.01331
Native MM	np.array	-	0.07422	0.07489	0.07471	0.07461
Optimized MM	np.array	15	0.05120	0.05184	0.05143	0.05149
Numpy.dot()	np.array	-	0.0001487	0.0001805	0.0001589	0.0001627

Graphical representation of results







Conclusion:

1. Optimized matrix-matrix multiplication done with loop unrolling executed faster than the native (not optimized) matrix-matrix multiplication in both array assignment cases – array assigned as type lists and type numpy.ndarray.

2. The for loop construct matrix-matrix multiplication(optimized and not optimized) with arrays of the type lists executed faster than that with arrays of the type numpy.ndarray .

3. Matrix multiplication with the numpy.dot() function executed faster than the for loop construct matrix multiolication (optimized and not optimized). NumPy is fast.

4. It would be interesting to see the effect of different unroll factors, but I do not report results for this.