A Modified Strassen Algorithm to Accelerate Numpy Large Matrix Multiplication with Integer Entries

SciPy 2023 – July 10-16, 2023

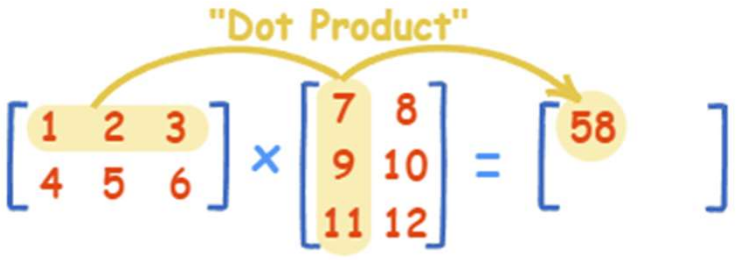
Anthony Breitzman, PhD
Assoc. Professor Computer Science Dept.
Data Science Program Coordinator
Rowan University, Glassboro, NJ, USA

Rowan University

Introduction

- Numpy is a popular Python library widely used in the math and scientific community because of its speed and convenience.
- We present a Modified Strassen type algorithm for multiplying large matrices with integer entries.
- For large Matrices with Integer entries the algorithm is **5 to 30** times faster than standard Numpy.matmul
- Although there is no apparent advantage for real entries, there are applications where integer matrices are used extensively such as:
  - Combinatorics
  - Graph Theory
  - Precision evaluation of so-called holonomic functions (e.g. exp,log, sin, Bessel functions, and hypergeometric functions) [See D. Harvey and J. V. der Hoeven, 2018]
  - Algebraic Geometry

Review of Matrix Multiplication



[https://www.mathsisfun.com/algebra/matrix-multiplying.html]

- This leads to the following elementary Python Implementation where A,B,C are NxN matrices

```
#multiply matrix A and B and put
#the product into C.
#A,B,C are assumed to be square
#matrices of the same dimension.
#No error checking is done.
def multiply(A, B, C):
  for i in range(N):
    for j in range( N):
      C[i][j] = 0
      for k in range(N):
        C[i][j]+=A[i][k]*B[k][j]
```

Rowan University

SciPy 2023

Review of Matrix Multiplication (2)

- The code shown on previous slide works for only square matrices.

- With a little modification we can use it to multiply any 2 matrices where the column dimension of the first matches the row dimension of the second.
- We show the simple code on the previous slide to illustrate the complexity is $O(N^3)$ as evidenced by the 3 nested loops
- Strassen's algorithm reduces the complexity to $O(N^{2.81})$
- In general, the complexity of implementing Strassen's algorithm is not usually worth the small gain in computational complexity, however we will show that combining Strassen with Numpy will greatly speed up Numpy's computation on large integer matrices

Divide and Conquer

- Matrices with an even number of rows and columns can be subdivided and multiplied as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

[https://www.geeksforgeeks.org/strassens-matrix-multiplication/]

Divide and Conquer (Strassen)

- Volker Strassen's Algorithm [Strassen, 1969] reduces the number of sub-matrix multiplications to 7 at the cost of additional matrix additions and matrix subtractions

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

[https://www.geeksforgeeks.org/strassens-matrix-multiplication/]

Divide and Conquer Complexity

- It's fairly easy to convince ourselves that the computational complexity of the regular divide and conquer (2 slides back) version of matrix multiplication is still $O(N^3)$.

- Since Addition and Subtraction of two matrices takes $O(N^2)$ time. The complexity of Strassen (1 slide back) is: $T(N) = 7T(N/2) + O(N^2)$

- From the Master Theorem, complexity of above method is $O(N^{Lg\ 7})$ which is approximately $O(N^{2.8074})$

Baseline Timings

	Numpy		Strassen 1		Standard Multiply	
Matrix Size	Time (seconds)	Current/ Previous	Time (seconds)	Current/ Previous	Time (seconds)	Current/ Previous
128x128	0.002	-	3.777	-	1.869	-
256x256	0.02	8.728	26.389	6.986	15.031	8.043
512x512	0.222	10.999	188.781	7.154	125.279	8.334

TABLE 1: *Timing for Base Algorithms on Matrices with Integer Entries. (Intel Core I7-9700 CPU @ 3.00 GHz, 8 Cores)*

- Numpy is much better than the obvious implementation of matrix multiply from slide 3 and from the default Strassen algorithm described in slide 6.

- On 512 x 512 matrices, Numpy takes a fraction of a second while the standard algorithm takes over 2 minutes and the Strassen algorithm takes over 3 minutes.
- The current/previous column illustrates that if we double the size of the matrices, then Strassen will take 7 times as long as the previous case and the Standard algorithm will take 8 times as long. Numpy takes 11 times as long so it is clear that Numpy has optimizations that favor smaller matrices
- However even with the favorable growth rates, the matrix size would have to reach $N=10^{16}$ before the Strassen algorithm had an advantage over Numpy.
- In the next slide we'll show a modified Strassen algorithm that has an advantage over Numpy as soon as N exceeds 128

Strassen With Crossover at N

- Rather than continuing to recurse on the Strassen algorithm until we get to 2 x 2 we suggest crossing over to a regular Numpy.matmul as soon as N gets to a smaller value such as 64, 128, or 256.

- Python Code for the Strassen with Crossover is shown below and on next slides
- First, we need some functions for dividing a matrix into quadrants and for zero-padding rows and columns if the dimensions are not even

```
def split(matrix):
    """
    Splits a given matrix into quarters a,b,c,d.
    Input: mxn matrix with m and n both even
    Output: tuple containing a,b,c,d
    """
    row, col = matrix.shape
    row2, col2 = row//2, col//2
    return matrix[:row2, :col2], matrix[:row2, col2:],
           matrix[row2:, :col2], matrix[row2:, col2:]
```

```
def padRow(m):
  x = []
  for i in range(len(m[0])):
    x.append(0)
  return(np.vstack((m,x)))

def padColumn(m):
  x = []
  for i in range(len(m)):
    x.append(0)
  return(np.hstack((m,np.vstack(x))))
```

Rowan University

SciPy 2023

Strassen With Crossover at N

- The code is straightforward:
- a,b,c,d,e,f,g,h and p1 – p7 are exactly as in slide 6.

- The only difference is this algorithm stops recursing and calls Numpy when one of the matrix dimensions is smaller than the crossoverCutoff

- This also contains some padding routines that allows the matrix multiplication to work for arbitrary size matrices rather than just square matrices

```
def strassenCrossover(x, y,crossoverCutoff):
#Split matrix into quadrants and recurse as usual until one of the matrix dimensions is < crossoverCutoff

    # Base case when size of matrices is <= crossoverCutoff
    if len(x) <= crossoverCutoff:
        return np.matmul(x,y)
    if len(x[0])<= crossoverCutoff:
        return np.matmul(x,y)

    rowDim = len(x)
    colDim = len(y[0])
    if (rowDim & 1 and True):  #if odd row dimension then pad
        x = padRow(x)
        y = padColumn(y)

    if (len(x[0]) & 1 and True):  #if odd column dimension then pad
       x = padColumn(x)
       y = padRow(y)

    if (len(y[0]) & 1 and True):
        y = padColumn(y)

    # Splitting the matrices into quadrants.
    a, b, c, d = split(x)
    e, f, g, h = split(y)

    # Computing the 7 products, recursively (p1, p2...p7)
    if (len(x) >crossoverCutoff):
     p1 = strassenCrossover(a, f - h,crossoverCutoff)
     p2 = strassenCrossover(a + b, h,crossoverCutoff)
     p3 = strassenCrossover(c + d, e,crossoverCutoff)
     p4 = strassenCrossover(d, g - e,crossoverCutoff)
     p5 = strassenCrossover(a + d, e + h,crossoverCutoff)
     p6 = strassenCrossover(b - d, g + h,crossoverCutoff)
     p7 = strassenCrossover(a - c, e + f,crossoverCutoff)
    else:
     p1 = np.matmul(a, f - h)
     p2 = np.matmul(a + b, h)
     p3 = np.matmul(c + d, e)
     p4 = np.matmul(d, g - e)
     p5 = np.matmul(a + d, e + h)
     p6 = np.matmul(b - d, g + h)
     p7 = np.matmul(a - c, e + f)

    # Computing the values of the 4 quadrants of the final matrix c
    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7

    # Combining the 4 quadrants into a single matrix by stacking horizontally and vertically.
    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))

    x = len(c) - rowDim
    if (x > 0):
        c = c[:-x, :]  #delete padded rows

    x = len(c[0]) - colDim
    if (x > 0):
        c = c[:,:-x]  #delete padded columns

    return c
```

Timings at Various Crossover values

Matrix Size	Numpy	Strassen	Strassen16	Strassen32	Strassen64	Strassen128	Strassen256	Strassen512	Standard
128 x 128	0.00	3.88	0.02	0.00	0.00	0.00	0.00	0.00	1.32
256 x 256	0.03	26.85	0.13	0.03	0.01	0.01	0.01	0.01	10.67
512 x 512	0.27	188.09	0.90	0.19	0.09	0.08	0.11	0.20	86.63
1024 x 1024	3.75	-	6.70	1.41	0.64	0.63	0.82	1.45	-
2048 x 2048	82.06	-	44.03	9.29	4.24	4.23	5.44	9.84	-
4096 x 4096	988.12	-	322.82	68.06	31.61	31.10	40.14	72.56	-
8192 x 8192	14722.33	-	2160.77	457.28	211.77	211.02	270.69	483.54	-

Table 2: Timings (seconds) for matrix multiplication square matrices with integer entries. MacBook Pro 16 with Core i7 @ 2.6 GHz

- In the table above, Strassen is the non-crossover algorithm, Standard is the standard matrix multiply, and Strassen64, Strassen128 etc. suggest we stop recursing at 64, 128, etc. and call Numpy on the remaining sub-matrices.
- We see that crossing over at 128 is optimal on the test machine. As expected, both the non-crossover Strassen and the standard algorithm are not competitive and stopped for sizes above 512 x 512.
- Strassen128 is almost 70 times faster than Numpy for matrices of size 8192 x 8192.
- We will see a more modest (but still significant) advantage of 5 to 30 times faster for arbitrary size non-square matrices soon.

Questions the Audience Should have at this Point

- What happens when the matrices are not square and not of dimensions that are powers of 2? That is, is there an advantage over Numpy for arbitrary size matrix multiplications?

- Would a divide and conquer approach without Strassen (slide 5) provide an advantage over Numpy?

- Does the advantage only exist on the test machine (Macbook with i7)?

- We explore these answers (which are yes, yes, no) in the next several slides

Arbitrary Matrices

- Note the code shown in slides 9 and 10 doesn't have to change to multiply arbitrary size non-square matrices.

- We also note that we could use the regular Divide and Conquer algorithm from slide 5 by replacing the products p1-p7 in the Strassen code with p1-p8 from slide 5.

- For fun we also will test whether there is any difference when calling Numpy via '@', via dot-product, or via Numpy.matmul.

- 4 Results are on the next slide

Python Outputs for 4 arbitrary size matrix multiplies

```
(1701 x 1267) * (1267 x 1678)
numpy (seconds)  15.43970187567
numpyDot (seconds)  15.08170314133
a @ b (seconds)  15.41474305465
strassen64 (seconds)  3.980883831158
strassen128 (seconds)  2.968686999753
strassen256 (seconds)  2.88325377367
DC64 (seconds)  6.42917919531
DC128 (seconds)  4.37878428772
DC256 (seconds)  4.12086373381

(1659 x 1949) * (1949 x 1093)
numpy (seconds)  33.79341135732
numpyDot (seconds)  33.8062295187
a @ b (seconds)  33.6903500761
strassen64 (seconds)  2.929703416
strassen128 (seconds)  2.54137444496
strassen256 (seconds)  2.75581365264
DC64 (seconds)  4.581859096884
DC128 (seconds)  4.08950223028
DC256 (seconds)  4.01872271299
```

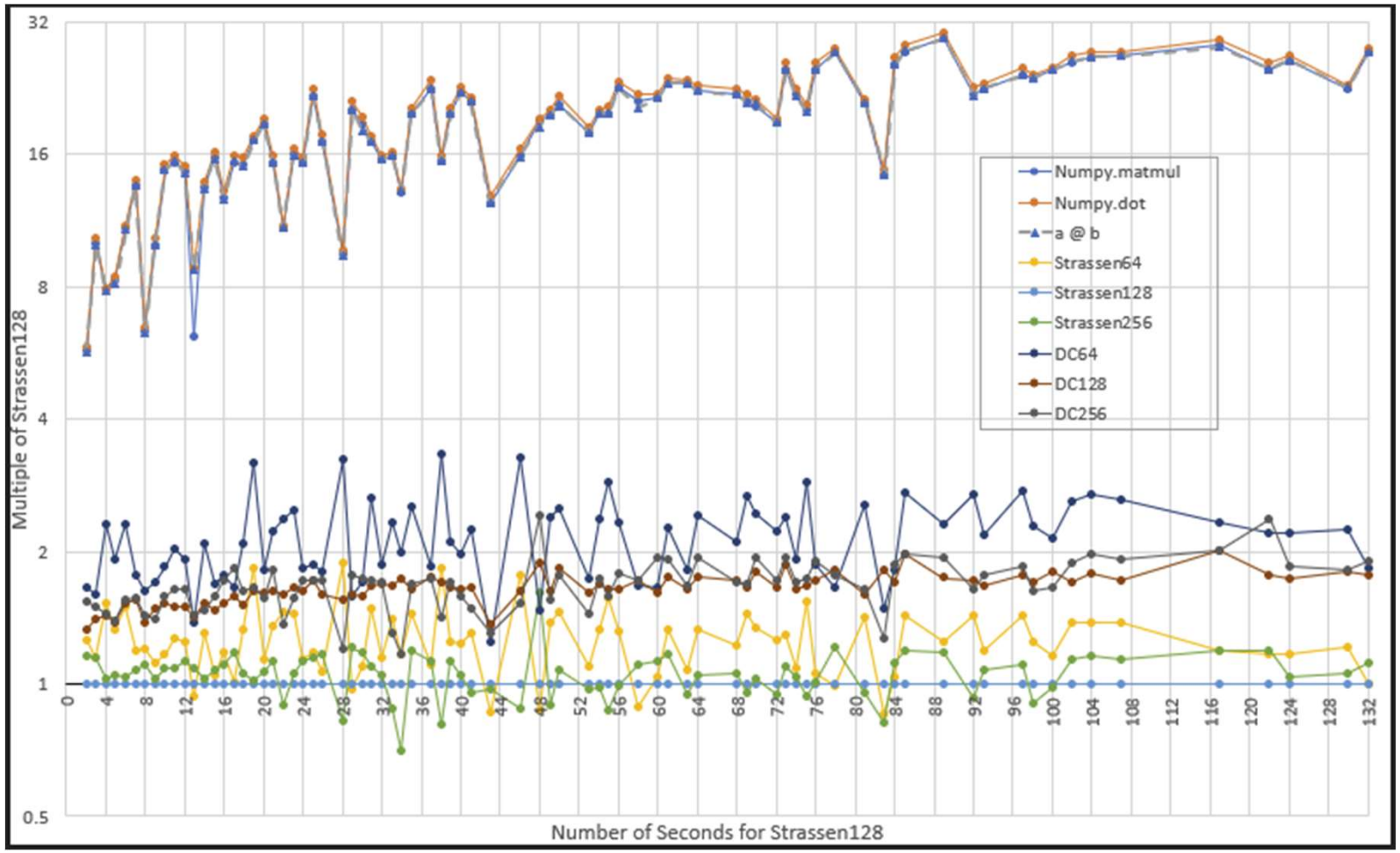
```
(1386 x 1278) * (1278 x 1282)
numpy (seconds)  7.96956253983
numpyDot (seconds)  7.54114297591
a @ b (seconds)  8.81335245259
strassen64 (seconds)  2.425855960696
strassen128 (seconds)  1.823907148092
strassen256 (seconds)  1.74107060767
DC64 (seconds)  3.8810345549
DC128 (seconds)  2.672704061493
DC256 (seconds)  2.603429134935

(1534 x 1150) * (1150 x 1439)
numpy (seconds)  11.16470945253
numpyDot (seconds)  10.07360629551
a @ b (seconds)  10.1722311042
strassen64 (seconds)  2.23611851967
strassen128 (seconds)  1.798768131062
strassen256 (seconds)  1.93347921967
DC64 (seconds)  3.40701649896
DC128 (seconds)  2.765258671715
DC256 (seconds)  2.890671111643
```

Rowan University

Notes on Previous Slide

- The output is 4 cases out of over a million cases where large random sizes are chosen for each matrix.
- In the first case we multiply a 1701 x 1267 matrix by a 1267 x 1678 matrix.
- It takes 15.4 seconds using Numpy and 2.97 seconds using Strassen with a crossover value of 128.
- So the Strassen case is 5 times faster than Numpy
- We also test the Divide and Conquer algorithm for various crossover values and it tends to be slower than Strassen and faster than Numpy
- Results of a million cases on next slide.

Results: MacBook Pro 16 with Core i7 @ 2.6 GHz



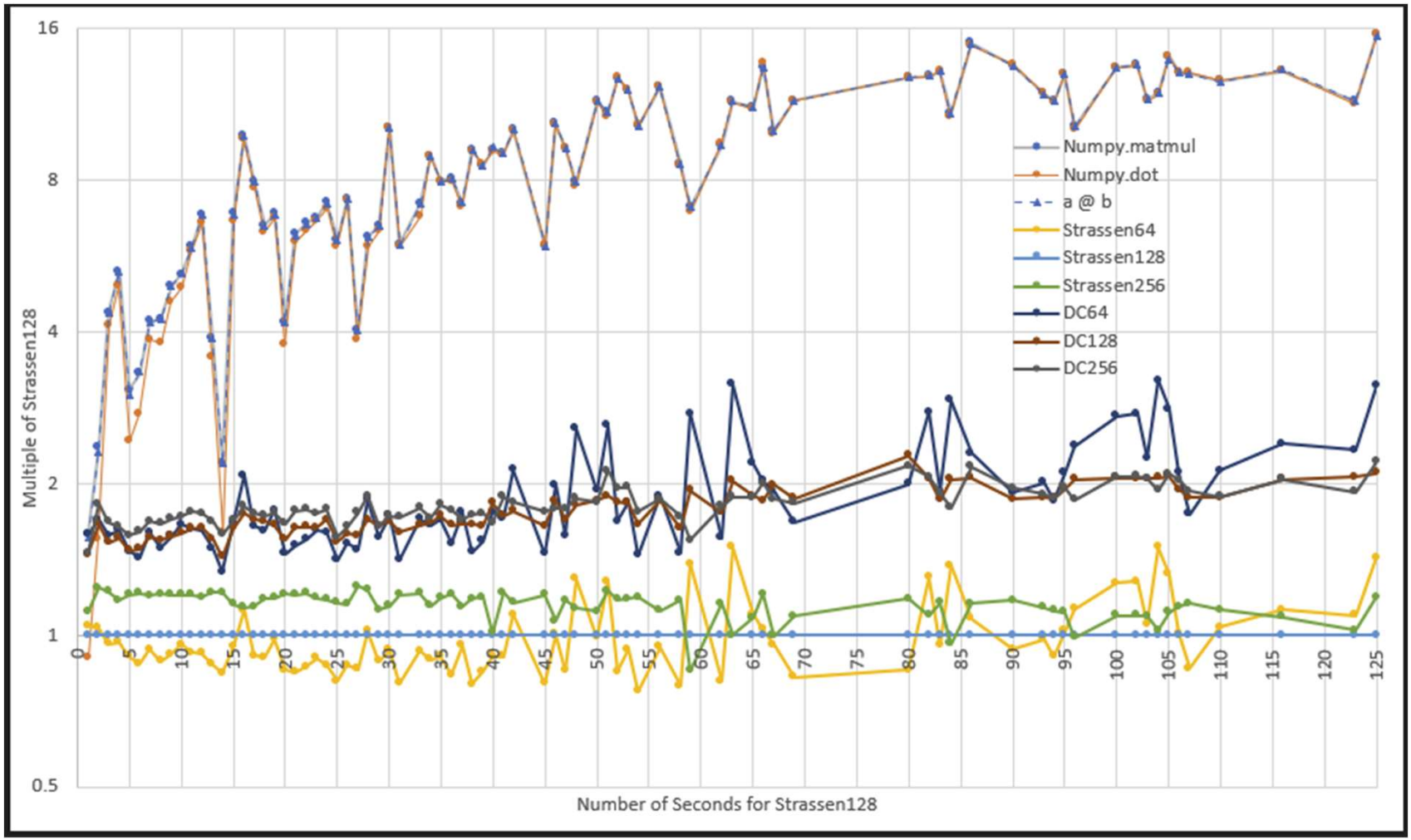
Rowan University

SciPy 2023

Interpretation of Previous Slide

- In order to display the results of a million cases we take all of the cases for each time value of Strassen128 and average them for each other method.
- For example, if we have 10 cases of Strassen 128 that take 2.99 seconds, we take the average time value of those 10 cases for Numpy (30.72) and plot the time for Numpy divided by the time for Strassen128 (2.99) and plot 30.72/2.99 = 10.27 for the Numpy graph.
- Some key results of the previous slide:
  - Regular Numpy is 5 to 30 times slower than Strassen128 and when Strassen128 takes longer, Numpy will usually take much longer.
  - For example when Strassen128 takes 2 minutes, the Numpy cases will take almost an hour.
  - The Divide and Conquer algorithms (DC128…) usually take about twice as long as Strassen128
  - There is no difference between Numpy.matmul, Numpy.dot, and '@' as expected.

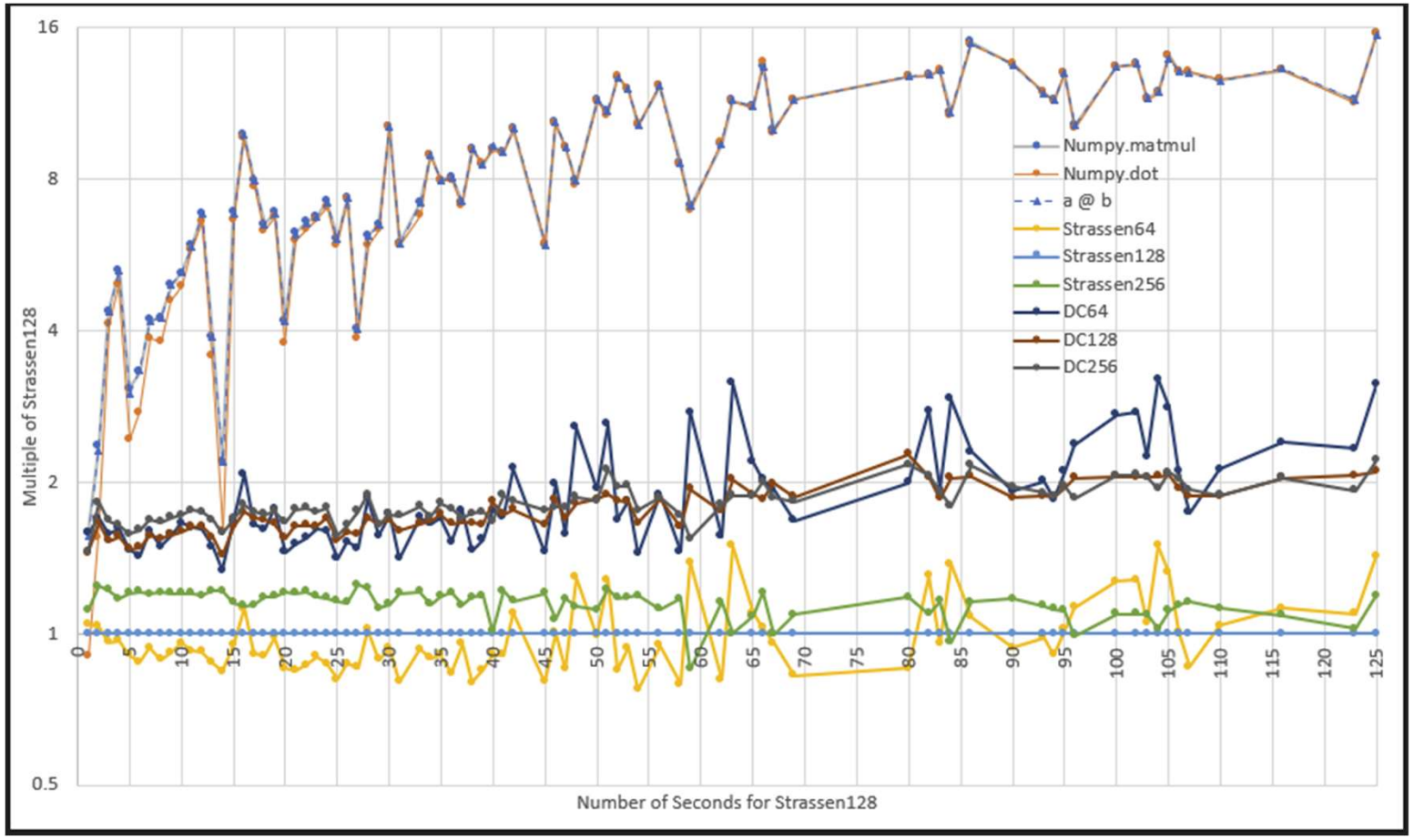
Results: Windows 11 with Core i7 @ 3.0 GHz



Rowan University

SciPy 2023

Results: Windows 11 with Core i7 @ 3.0 GHz



- Results similar to Macbook except Strassen64 often outperforms Strassen128 on this machine.

Rowan University

SciPy 2023

Summary

- Numpy is widely used in the math and scientific community because of its speed.
- In this project we created a Strassen based matrix multiplication algorithm that crosses over to Numpy for a fixed N of 64 to 128 will outperform Numpy.matmul for large matrices with integer entries.
- We showed on multiple machines that the presented algorithm can be 5 to 30 times faster for large (thousands of rows or columns) and the difference will be more pronounced as the matrices get larger,
- In one example a multiplication that took 2 minutes for the Strassen-based algorithm, took nearly an hour for Numpy.
- While the algorithm significantly outperforms Numpy on integer matrices, there is no significant performance improvement for matrices with floating point entries.

References

1. Z. Fink, S. Liu, J. Choi, M. Diener, and L. V. Kale, “Performance evaluation of python parallel programming models: Charm4py and mpi4py,”
2. 2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pp. 38–44, 2021.
3. V. Strassen, “Gaussian elimination is not optimal,” Numerische Mathematik, pp. 354–356, 1969.
4. D. Harvey and J. V. der Hoeven, “On the complexity of integer matrix multiplication,” Journal of Symbolic Computation, pp. 1–8, 2018.
5. Python.org, “Pep 465: A dedicated infix operator for matrix multiplication.” Available at https://peps.python.org/pep-0465/, 2014.
6. GeeksforGeeks, “Strassen’s matrix multiplication - geeksforgeeks.” Available at https://www.geeksforgeeks.org/strassens-matrix-multiplication/ , 2022.
7. T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms. Cambridge, MA: MIT Press, 2009.

Rowan University