Too Big to Fail:

Massive Scale Linear Algebra with Chapel and Arkouda

Chris Hollis

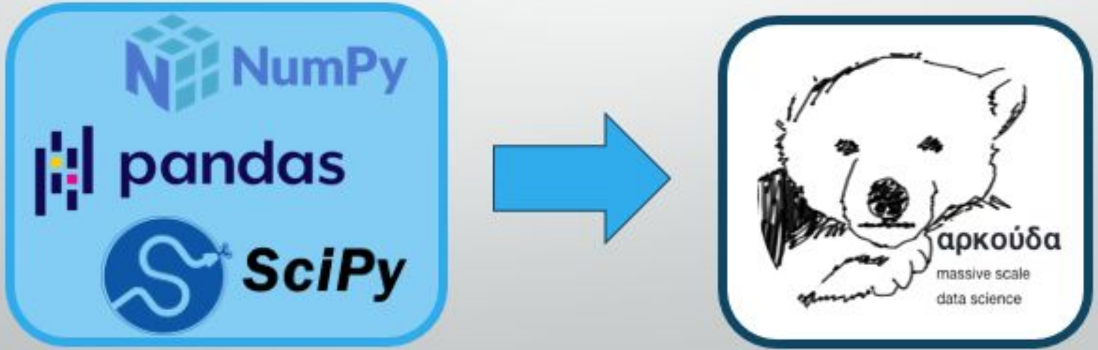
Software Engineer – Department of Defense

CHIUW 2023

June 2

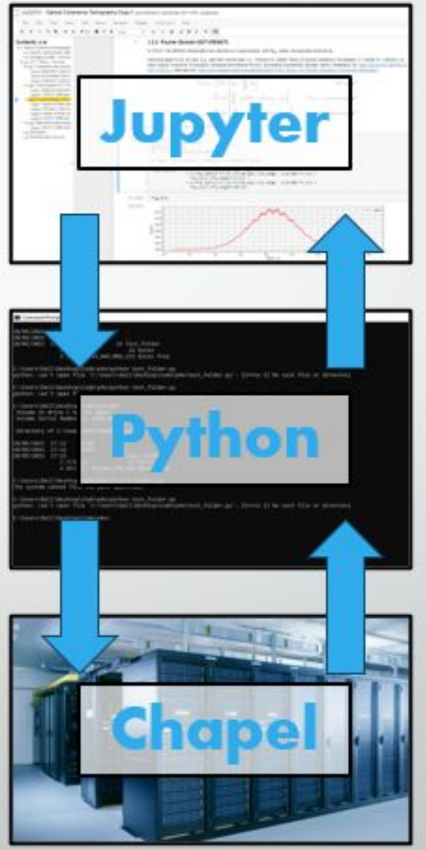
Objective

- Exploratory data analysis (EDA) requires open-ended and frictionless interaction with data
  - Pandas -> NumPy/SciPy -> Linear algebra -> Pandas
- Arkouda allows interactive EDA at scale
  - 10's of TBs of data
  - Distributed memory allows for large array allocation



Arkouda Overview

- **What is Arkouda?**
  - A NumPy-like Python app that utilizes Chapel for its backend server
    - Abstracts powerful Chapel functions with a familiar Python interface
  - Prioritizes compatibility with existing data science workloads
    - Jupyter notebooks
    - Mirrors Pandas/NumPy usage
  - Open-scource and can be found at:
    - https://github.com/Bears-R-Us/arkouda



AkSparse Overview

- **What is AkSparse?**
  - Sparse linear algebra library built with Arkouda
  - Emulates SciPy's ".sparse" library
  - Supports COO, CSR, and CSC formats
    - Basic matrix arithmetic
    - Matrix-Vector multiplication
    - Sparse General Matrix Multiplication (SpGeMM)

	Sparse matrix object	Format conversion	Sparse General Matrix Multiplication
Aksparse	Aksparse.coo_matrix()	A.tocsc()	C = A.spgemm(B)
scipy	scipy.coo_matrix()	A.tocsr()	C = A.dot(B)

Algorithm

- **Sparse matrix multiplication is hard**
  - No way to know how large solution will be beforehand
  - Load balancing
  - Communication cost
- **Focus on large unstructured data**
  - Need distributed-scale computing
  - Communication cost is a bottleneck

$$\begin{pmatrix} & A & \end{pmatrix}\begin{pmatrix} & B & \end{pmatrix} = \begin{pmatrix} & ? & \end{pmatrix}$$

Algorithm

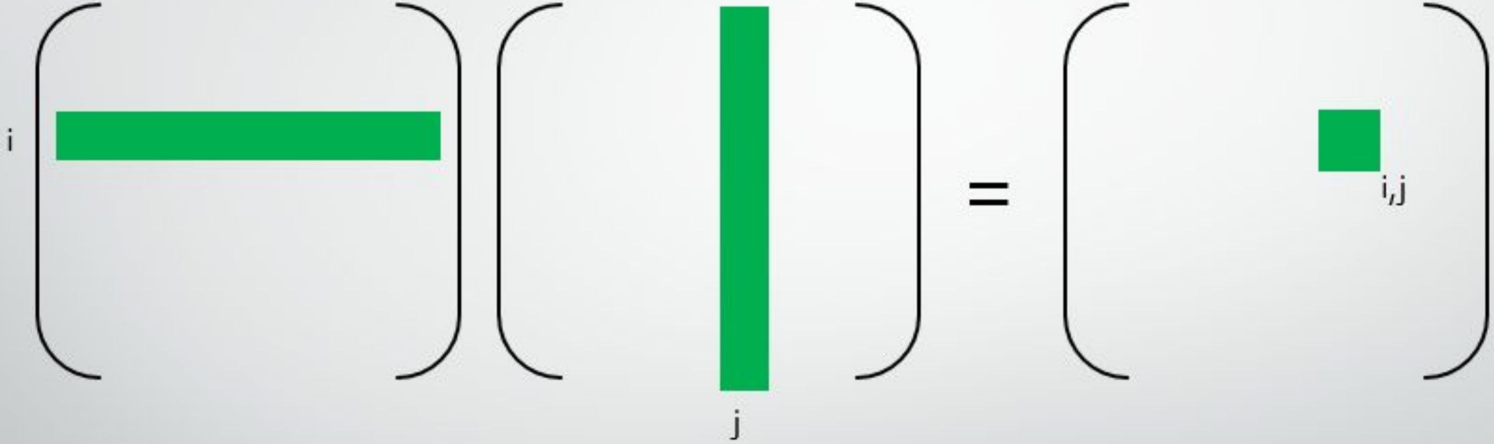
- **Sparse matrix multiplication is hard**
  - No way to know how large solution will be beforehand
  - Load balancing
  - Communication cost
- **Focus on large unstructured data**
  - Need distributed-scale computing
  - Communication cost is a bottleneck

- **How is AkSparse's SpGeMM different?**
  - Leverage Arkouda's optimized sorting and groupby capabilities on HPC hardware
    - Interactive manipulation of TB scale data
  - "Outer product" formulation of SpGeMM
    - Reveals size of work needed before any computations
    - Minimize communication cost through Arkouda's message aggregation

Algorithm

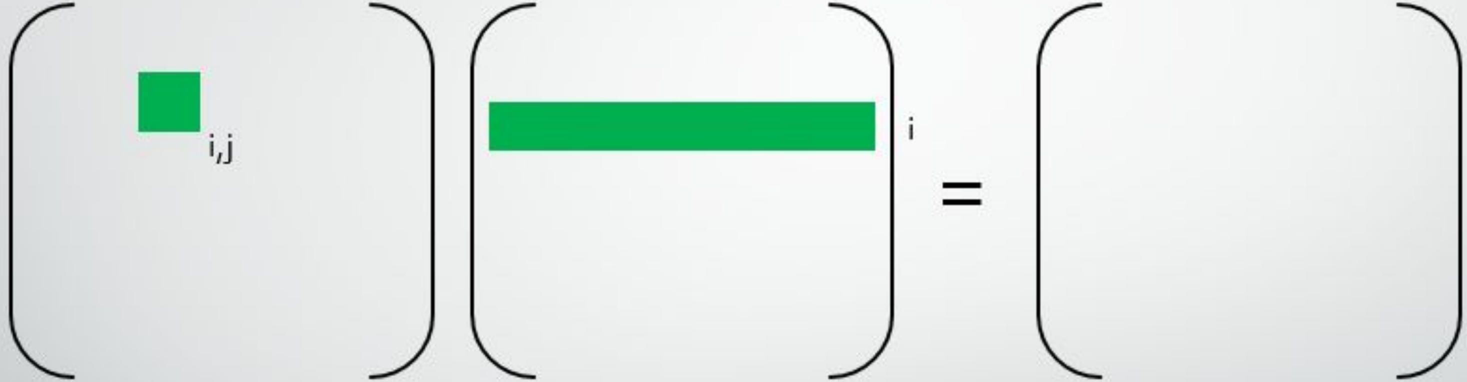
$$\begin{pmatrix} \\ & A & \\ \\ \end{pmatrix} \begin{pmatrix} \\ & B & \\ \\ \end{pmatrix} = \begin{pmatrix} \\ & C & \\ \\ \end{pmatrix}$$

Algorithm

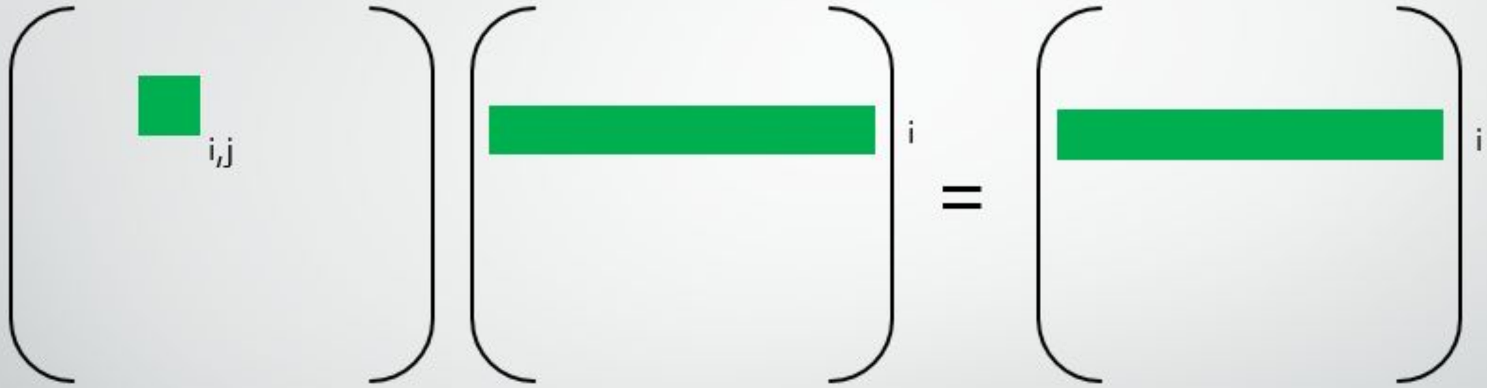


Algorithm

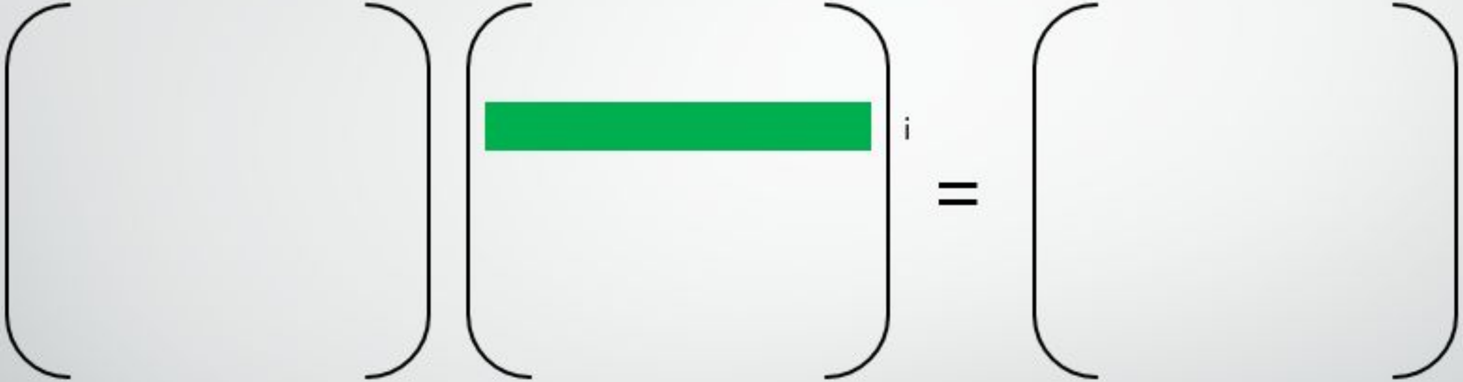
Algorithm



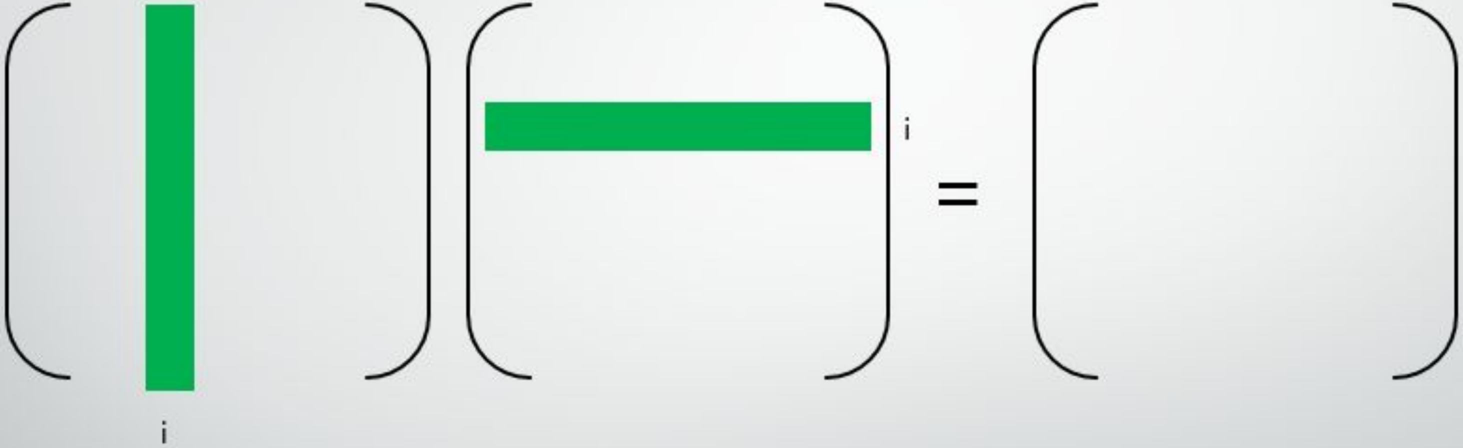
Algorithm



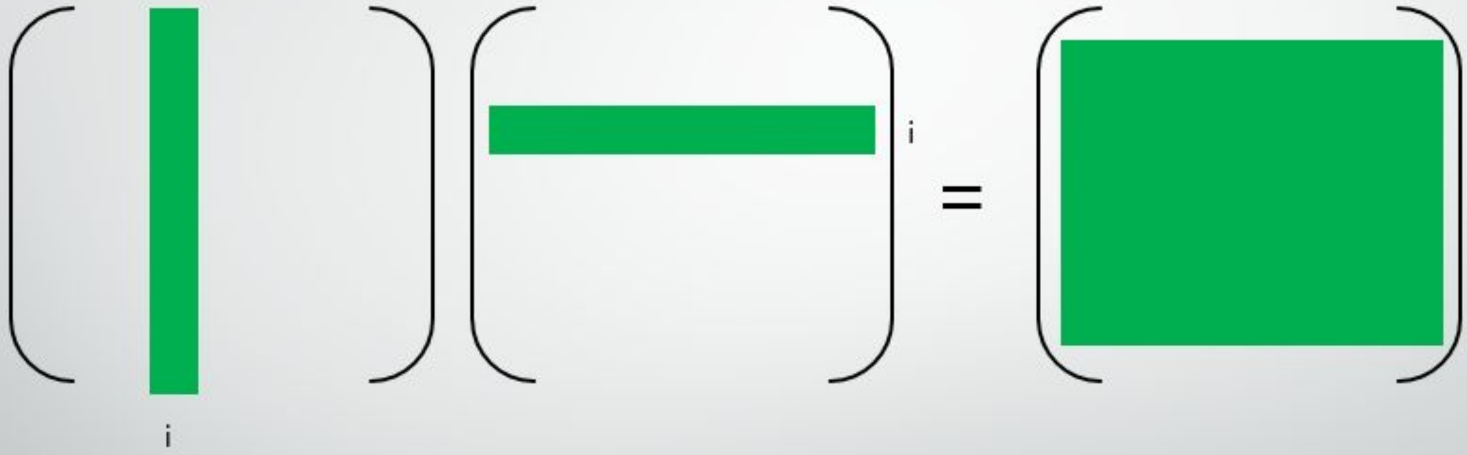
Algorithm



Algorithm



Algorithm



Algorithm

- SpGeMM in Aksparse (A.spgemm(B))
  - Convert A to CSC
  - Convert B to CSR
  - Find all 'hits' between nonzero entries in the columns of A and corresponding rows of B
  - Generate a single Arkouda array for all the multiplications of A.spgemm(B)
  - Perform a GroupBy on the matrix indices implied by the full multiplication array
  - Perform a sum aggregate on the full multiplication array results to yield the final matrix C

```
def spgemm(self: CSC, other: CSR):

#Identify number of multiplications needed
starts = other.indptr[self._gb_col_row.unique_keys[0]]
ends = other.indptr[self._gb_col_row.unique_keys[0] + 1]
lengths = (ends - starts)
fullsize = lengths.sum()
segs = ak.cumsum(lengths) - lengths
slices = ak.ones(fullsize, dtype=ak.akint64)
diffs = ak.concatenate((ak.array([starts[0]]),
                        starts[1:] - ends[:-1] + 1))

#Set up arrays for multiplication
slices[segs] = diffs
nonzero = (ends > starts)
fullsegs, ranges =  segs, ak.cumsum(slices)
fullBdom = other._gb_row_col.unique_keys[1][ranges]
fullAdom = ak.broadcast(fullsegs,
                        self._gb_col_row.unique_keys[1][nonzero],
                        fullsize)
fullBval = other.data[ranges]
fullAval = ak.broadcast(fullsegs, self.data[nonzero], fullsize)
fullprod = fullAval * fullBval

#GroupBy indices and perform aggregate sum
proddomGB = ak.GroupBy([fullAdom, fullBdom])
result = proddomGB.sum(fullprod)
return Csr(result[1],
           result[0][1],
           result[0][0],
           shape = (self.shape[0], other.shape[1]))
```

Algorithm

- SpGeMM in Aksparse [A.spgemm(B)]
  - Convert A to CSC
  - Convert B to CSR
  - Find all 'hits' between nonzero entries in the columns of A and corresponding rows of B
  - Generate a single Arkouda array for all the multiplications of A.spgemm(B)
  - Perform a GroupBy on the matrix indices implied by the full multiplication array
  - Perform an aggregate on the full multiplication array results to yield the final matrix C

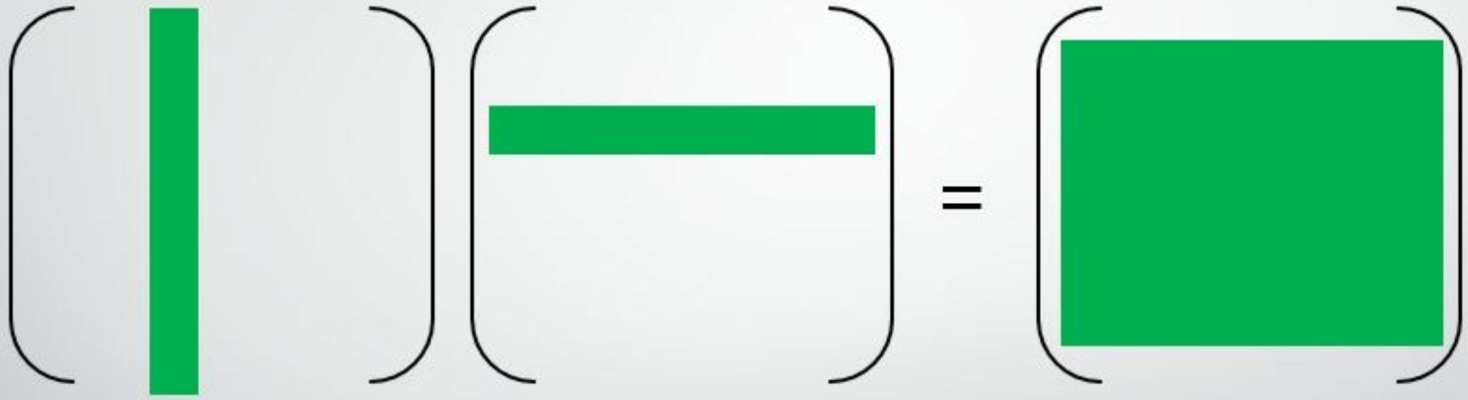
Why it's an outer product

This is easy!

Effectively yields N rank 1 COO matrices

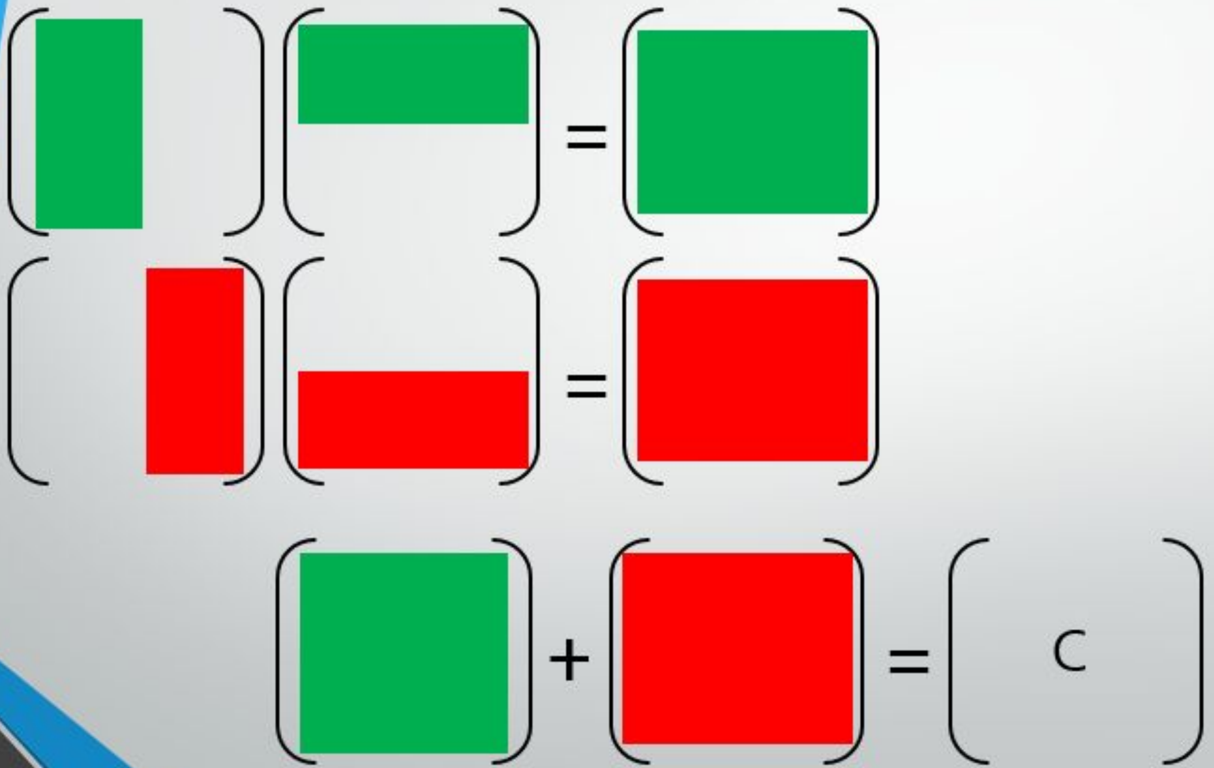
This is hard!

Algorithm



Algorithm

Algorithm



Algorithm

- Only "difficult" computation: O(# mults) groupby
  - O(# mults) in general is much bigger than nnz
  - Arkouda is tuned to handle large sorts
- Counting # of mults needed is easy
  - Can be done without forming the array
  - This means we know if splitting the problem is necessary before attempting the calc
- Avoids the load balancing issue
  - Recursive splitting
- Runs on large distributed memory system
  - Multiple 2TB nodes with dual-socket InfiniBand interconnect
  - Can handle MUCH larger nnz amounts in the output

Results/Benchmarks

*Results in seconds

SciPy on home computer

NNZ/Size(NxN)	100k	1mil	10mil	100mil
100k	0.02	0.04	0.14	1.02
1mil	0.34	0.35	0.46	2.01
10mil	311.14	11.64	6.28	4.97
100mil	X	X	X	X
1bil	X	X	X	X

SciPy on HPC

NNZ/Size(NxN)	100k	1mil	10mil	100mil
100k	0.02	0.04	0.24	2.52
1mil	0.28	0.23	0.53	3.07
10mil	24.57	6.59	4.56	6.84
100mil	X	X	101.50	55.68
1bil	X	X	X	X

Arkouda on HPC

NNZ/Size(NxN)	100k	1mil	10mil	100mil
100k	3.19	3.10	3.03	2.98
1mil	3.22	3.18	3.19	3.11
10mil	7.51	3.46	3.22	3.14
100mil	41.79	42.01	7.25	3.50
1bil	X	X	X	44.99

Final Test case:
~10bil nnz in output
~10bil multiplications

Results/Benchmarks

- Adjacency matrix A for a "small world" graph
  - #rows = #columns = ~77 mil
  - NNZ = ~620 mil
- Computing $A*A^T$
  - 40 compute nodes
  - ~440 billion multiplies
  - ~300 billion NNZ in C
    - ~22TB of memory to compute and store solution
  - ~**4 mins**

Future Work

- AkSparse is open source and available here:
  - https://github.com/Bears-R-Us/arkouda-contrib/tree/main/aksparse
- Additional linear algebra functionality
- Optimization
  - Improved load balancing
  - Implement outerproduct SpGemm Chapel kernel
- Problems too big to store in memory (write to disk)
  - Target "big" problem:
    - #edges = O(1 bil)
    - NNZ = (100 bil)
    - # multiplies = O(100 trillion)

Questions?