P573 Computer Science

Randall Bramley

1104 Luddy Hall

8:00 – 9:15 AM, Monday & Wednesday

Matrix-vector product

- Suppose A is an n x n matrix, x is an n x 1 vector
- Want $y = A*x$ (so what are the dimensions of y?)
- Two ways of computing this (actually, there are at least three ways, but you've probably only seen two)
- I'll assume indexing starts at 1, since all linear algebra books do the same (except in signal processing)
- Version 1: compute the dotproduct of row i of A with the vector x to get $y(i)$

y = A * x

$$\begin{bmatrix} \\ \\ \\ \end{bmatrix} = \begin{bmatrix} -2 & 1 & 2 & 3 \\ 1 & 0 & -1 & 5 \\ -3 & 2 & 3 & 0 \\ -6 & 5 & 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 3 \\ -3 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ \\ \\ \\ \end{bmatrix} = \begin{bmatrix} -2 & 1 & 2 & 3 \\ 1 & 0 & -1 & 5 \\ -3 & 2 & 3 & 0 \\ -6 & 5 & 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 3 \\ -3 \\ 2 \end{bmatrix}$$

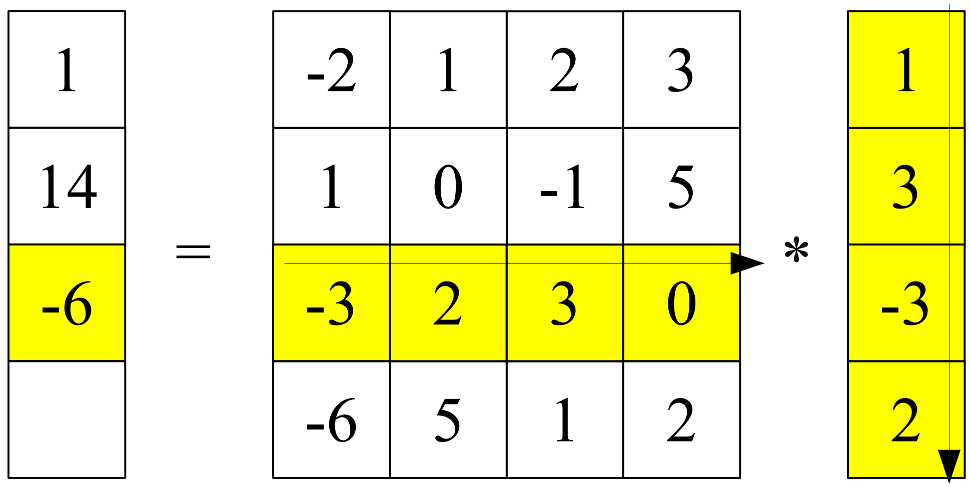
y(1) = A(1,1)*x(1) + A(1,2)*x(2) + A(1,3)*x(3) + A(1,4)*x(4)

1 = -2*1 + 1*3 + 2*-3 + 3*2

$$\begin{bmatrix}1\\14\\ \ \\ \ \end{bmatrix} = \begin{bmatrix}-2&1&2&3\\1&0&-1&5\\-3&2&3&0\\-6&5&1&2\end{bmatrix} * \begin{bmatrix}1\\3\\-3\\2\end{bmatrix}$$

$$y(2) = A(2,1)*x(1) + A(2,2)*x(2) + A(2,3)*x(3) + A(2,4)*x(4)$$

$$14 = 1*1 + 0*3 + -1*-3 + 5*2$$



y(3) = A(1,1)*x(1) + A(1,2)*x(2) + A(1,3)*x(3) + A(1,4)*x(4)

-6 = -3*1 + 2*3 + -3*-3 + 0*2

$$\begin{bmatrix} 1 \\ 14 \\ -6 \\ 10 \end{bmatrix} = \begin{bmatrix} -2 & 1 & 2 & 3 \\ 1 & 0 & -1 & 5 \\ -3 & 2 & 3 & 0 \\ -6 & 5 & 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 3 \\ -3 \\ 2 \end{bmatrix}$$

y(4) = A(1,1)*x(1) + A(1,2)*x(2) + A(1,3)*x(3) + A(1,4)*x(4)

10 = -6*1 + 5*3 + 1*-3 + 2*2

Matrix-vector product

- Leads to a simple algorithm, version *dotprod* :

```
y(1:n) = 0    // Set y to all zeros
for i = 1:n
      for j = 1:n
            y(i) = y(i) + A(i, j)*x(j)
      end for
 end for
```

- The above is pseudo-code:
  - *y(1:n) = 0* means set *y(1) = 0, y(2) = 0, …, y(n) = 0*
  - "*for i = 1:n*" is a loop setting *i = 1, 2, …, n* in turn
- We can swap the order of loops above ...

Matrix-vector product

- Swapping loops gives version *daxpy* :

```
y(1:n) = 0  // set y to be all zeros
for j = 1:n
      for i = 1:n
            y(i) = y(i) + A(i, j)*x(j)
      end for
 end for
```

- This represents y as a linear combination of the columns of A, with coefficients given by x
- If columns of A are vectors v_1, v_2, v_3, v_4 , the linear comb is $y = x(1)*v_1 + x(2)*v_2 + x(3)*v_3 + x(4)*v_4$
- In picture form ....

$$
\begin{bmatrix} 1 \\ 14 \\ -6 \\ 10 \end{bmatrix} = \begin{bmatrix} -2 & 1 & 2 & 3 \\ 1 & 0 & -1 & 5 \\ -3 & 2 & 3 & 0 \\ -6 & 5 & 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 3 \\ -3 \\ 2 \end{bmatrix}
$$

y = Col 1 of A*x(1) + Col 2 of A*x(2) + Col 3 of A*x(3) + Col 4 of A*x(4)

= A(1:4,1)*x(1) + A(1:4,2)*x(2) + A(1:4,3)*x(3) + A(1:4,4)*x(4)

$$\begin{bmatrix} 1 \\ 14 \\ -6 \\ 10 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \\ -3 \\ -6 \end{bmatrix} * \begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 2 \\ 5 \end{bmatrix} * \begin{bmatrix} 3 \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \\ 3 \\ 1 \end{bmatrix} * \begin{bmatrix} -3 \end{bmatrix} + \begin{bmatrix} 3 \\ 5 \\ 0 \\ 2 \end{bmatrix} * \begin{bmatrix} 2 \end{bmatrix}$$

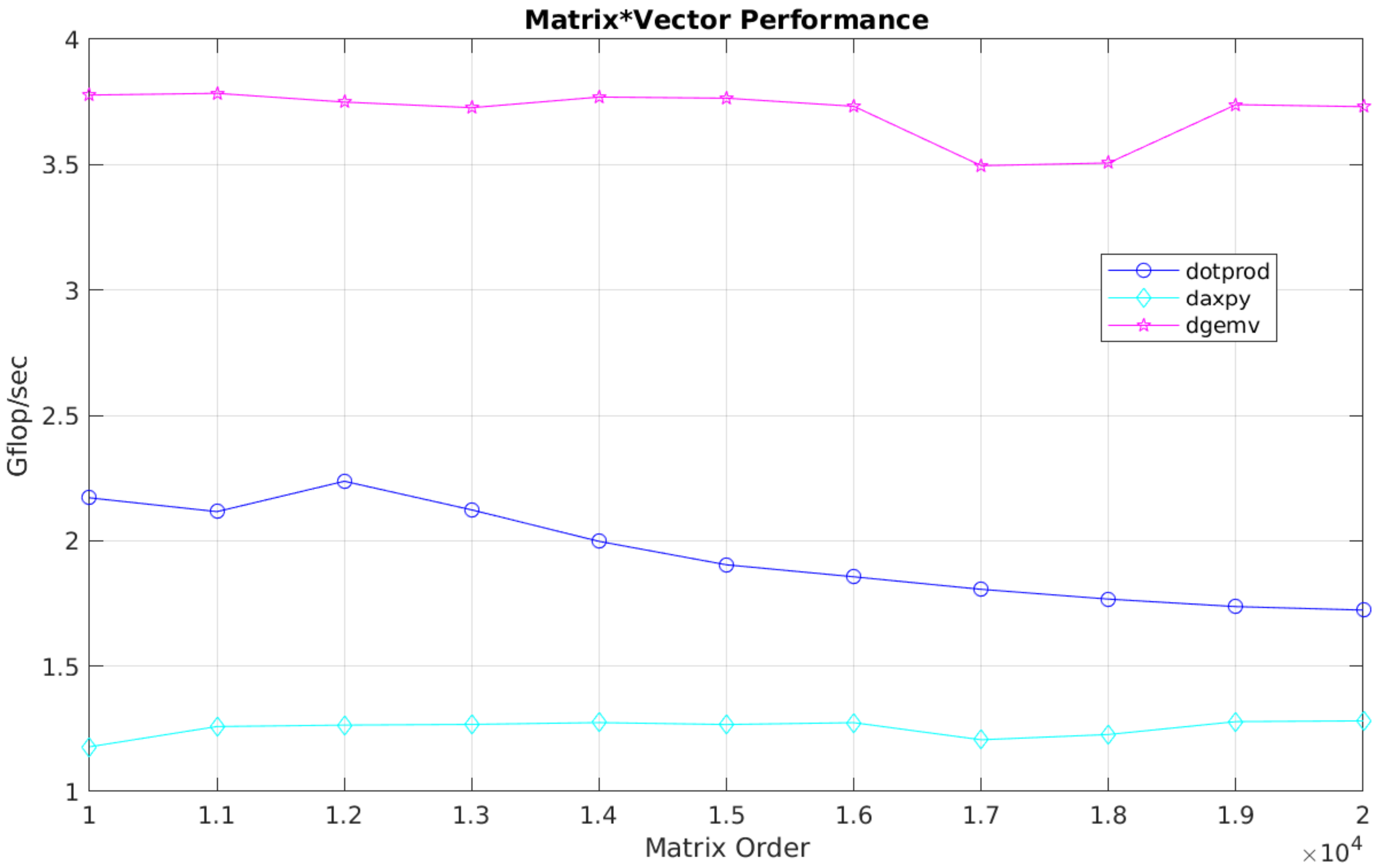
Matrix-vector product

- So big, fat, hairy deal. Who cares? (*ans: we do*)
- Load/store analysis says the first implementation (*dotprod*) is going to be 1.5 times faster than the second (*daxpy*)
- Now for the magic part of load/store: the same analysis says *some implementation* exists that will be 2 times as fast as the *dotprod* implementation
- Load/store does not say what that magic implementation would consist of, just that it exists
- Call that implementation *dgemv* for arcane reasons that will be explained later
- Big claims made above, and you should not trust Bramley (or anyone) unless that theoretical claim is backed up with actual computational results

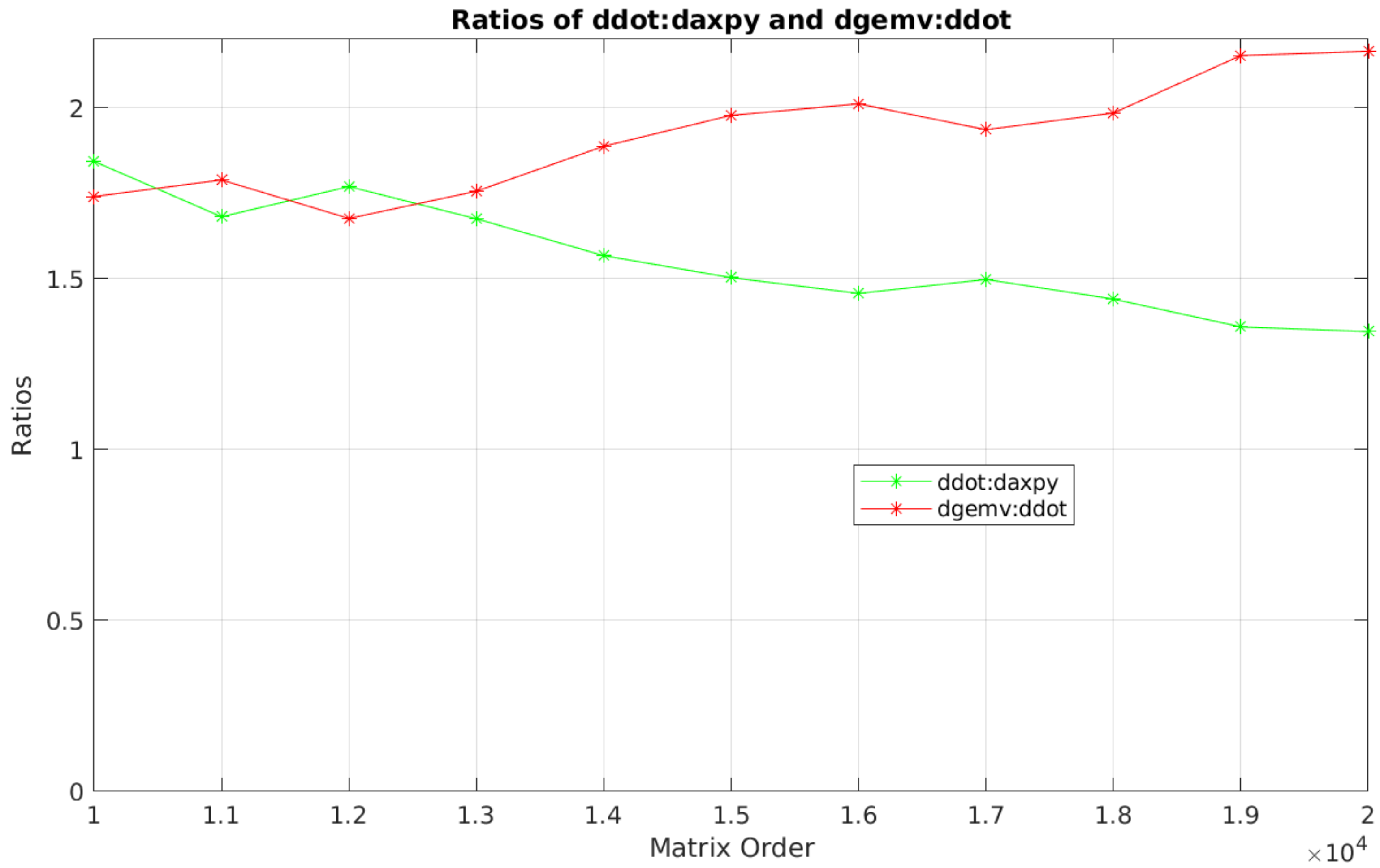
Matrix-vector product

- The mysterious third method *(dgemv)* is actually easy to do, based on some simple ideas covered later
- Implemented all three ways of computing matrix-vector product in Fortran 2018
- Language does not matter, results hold in C, C++, assembly language, Cobol, ...
- Ran on a desktop system with Intel i7 core processor
- Then plotted computational rate in Gflops/sec, against the matrix order (A is n x n, so the matrix order is n)
- n ranges from 10k to 20k

Results for matrix-vector product



Results for matrix-vector product



Matrix-vector product

- Load/store ratios of performance are not always exact, but do tell which implementation will be faster
- So if it says 1.5 times faster, actual performance may be 1.2 to 2.1 times faster, but will not be less than 1.0
- Results on previous slide shows the predicted ratios are good for this operation

Matrix-vector product

- Caveats:
  - Load/store is for large n; for $n = 1$ matrix-vector multiply is just a scalar multiply so all three versions are identical
  - Generally, "large n" means the data does not fit in cache, but in most cases $n \geq 50$ suffices
  - It's always possible to implement even a simple operation in such a stupid way that it will run abysmally slow
  - Results are for a general matrix A.
    - If A is the zero matrix, just set $y = 0$ (well, duh)
    - If A is a Fourier transform, ultrafast methods exist better than any of the three shown
    - If $A = uv^T$ is a rank-1 matrix where u and v are $n \times 1$ vectors, again far faster methods exist that take just $4n$ flops, not $2n^2$ flops