CoE 163

Computing Architectures and Algorithms

Matrix-Matrix Multiplication (part 1)

Why Matrix-Matrix Multiplication (MMM)?

- At the heart of many linear algebra algorithms
- Optimizing MMM is valuable to optimizing many applications, especially in the applied sciences

Warm up: Write some MMM code (practice on your own)

Three people denoted by P_1, P_2, P_3 intend to buy some rolls, buns, cakes, and bread. Each of them needs these commodities in differing quantities and can buy them in two shops S_1, S_2.

TRY IT YOURSELF (warm up): Using Python, write a program that determines which shop is the best for every person P_1, P_2, P_3 to pay as little as possible. The individual prices and desired quantities of the commodities are given in the following tables:

Demand quantity of foodstuff:

	roll	bun	cake	bread
P_1	6	5	3	1
P_2	3	6	2	2
P_3	3	4	3	1

Prices in shops S_1 and S_2

	S1	S2
roll	1.50	1.00
bun	2.00	2.50
cake	5.00	4.50
bread	16.00	17.00

How do we optimize our MMM algorithm?

Improving locality is the key

- Goal: write our program such that temporal and spatial locality is maximized
    - Cache misses are minimized
    - Contents of the cache are used up immediately
- Matrix multiplication has inherent locality
    - Spatial locality: matrices are stored as 2d arrays (see next: row major vs column major)
    - Temporal locality: Implemented as nested loops
- Try to optimize cache behavior in our MMM algorithm's innermost loop

Preliminaries: How are matrices stored in memory?

- Column-major vs Row-Major
  - Example shows how a matrix of floats (assume 8 bytes) is stored row-major (such as in C language) or column-major (such as in Fortran)

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

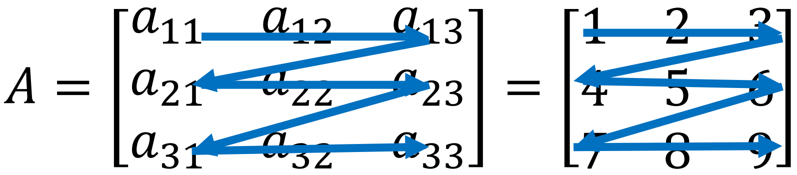
Sample code:

```
float A[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

Memory Address	Data (assume 8 bytes word length)
0x00000000	
0x00000008	
0x00000010	
0x00000018	
0x00000020	
0x00000028	
0x00000030	
0x00000038	
0x00000040	

Preliminaries: How are matrices stored in memory?

- Column-major vs Row-Major
  - Example shows how a matrix of floats (assume 8 bytes) is stored row-major (such as in C language) or column-major (such as in Fortran)



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Sample code:

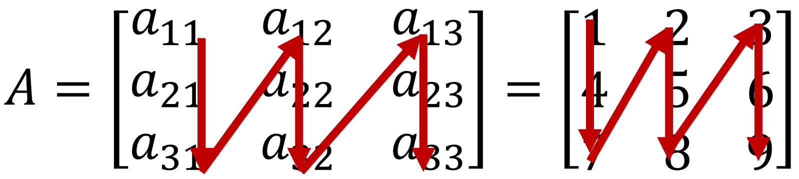
```
float A[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

ROW MAJOR

Memory Address	Data (assume 8 bytes word length)
0x00000000	`A[0][0]` = $a_{11} = 1$
0x00000008	`A[0][1]` = $a_{12} = 2$
0x00000010	`A[0][2]` = $a_{13} = 3$
0x00000018	`A[1][0]` = $a_{21} = 4$
0x00000020	`A[1][1]` = $a_{22} = 5$
0x00000028	`A[1][2]` = $a_{23} = 6$
0x00000030	`A[2][0]` = $a_{31} = 7$
0x00000038	`A[2][1]` = $a_{32} = 8$
0x00000040	`A[2][2]` = $a_{33} = 9$

Preliminaries: How are matrices stored in memory?

- Column-major vs Row-Major
  - Example shows how a matrix of floats (assume 8 bytes) is stored row-major (such as in C language) or column-major (such as in Fortran)



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Sample code:

```
float A[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

COLUMN MAJOR

Memory Address	Data (assume 8 bytes word length)
0x00000000	`A[0][0]` = $a_{11} = 1$
0x00000008	`A[1][0]` = $a_{21} = 2$
0x00000010	`A[2][0]` = $a_{31} = 3$
0x00000018	`A[0][1]` = $a_{12} = 4$
0x00000020	`A[1][1]` = $a_{22} = 5$
0x00000028	`A[2][1]` = $a_{32} = 6$
0x00000030	`A[0][2]` = $a_{13} = 7$
0x00000038	`A[1][2]` = $a_{23} = 8$
0x00000040	`A[2][2]` = $a_{33} = 9$

Why does this matter?

Memory Address	Data (assume 8 bytes word length)
0x00000000	A[**0**][0] = $a_{11} = 1$
0x00000008	A[**0**][1] = $a_{12} = 2$
0x00000010	A[**0**][2] = $a_{13} = 3$
0x00000018	A[**1**][0] = $a_{21} = 4$
0x00000020	A[**1**][1] = $a_{22} = 5$
0x00000028	A[**1**][2] = $a_{23} = 6$
0x00000030	A[**2**][0] = $a_{31} = 7$
0x00000038	A[**2**][1] = $a_{32} = 8$
0x00000040	A[**2**][2] = $a_{33} = 9$

- Table shows 2d array stored in main memory **row-wise**
- Algorithm we use must take this into account to maximize spatial locality
  - Outer loop should iterate row-wise, and then iterate across elements
  - If we iterate column-wise first, we are not accessing contiguous data in memory

Let’s look at a column-major example

Memory Address	Data (assume 8 bytes word length)
0x00000000	`A[0][0]` = a_{11}
0x00000008	`A[1][0]` = a_{21}
0x00000010	`A[2][0]` = a_{31}
0x00000018	`A[3][0]` = a_{41}
0x00000020	…
0x00000028	`A[6][9]` = $a_{7\ 10}$
0x00000030	`A[7][9]` = $a_{8\ 10}$
0x00000038	`A[8][9]` = $a_{9\ 10}$
0x00000040	`A[9][9]` = $a_{10\ 10}$

- Table shows 2d array stored in main memory **column-wise**
  - Assume A is 10×10 matrix
  - If we iterate row-wise first, we are not accessing contiguous data in memory

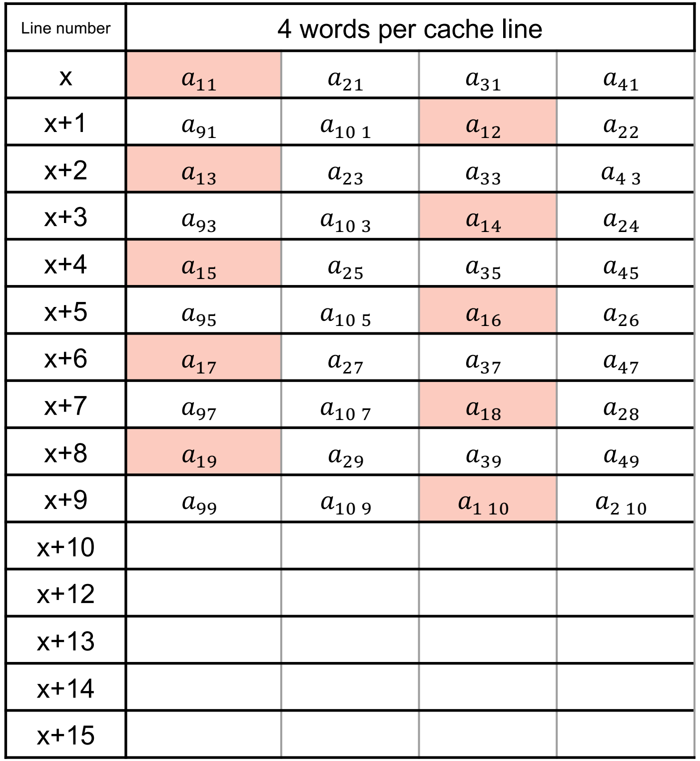
(demonstrated in next few slides)

Keep in mind: Transfers to cache are in blocks

- If we traverse one row of matrix A, we transfer other adjacent data words into cache that we are not using
- Highlighted words in diagram show row 1 of A in cache

Sample traversal pseudocode:

```
for i=1 to n
    for j=1 to n
      {perform operation with $A_{ij}$}
    end for
end for
```



Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\ 1}$	a_{12}	a_{22}
x+2	a_{13}	a_{23}	a_{33}	$a_{4\ 3}$
x+3	a_{93}	$a_{10\ 3}$	a_{14}	a_{24}
x+4	a_{15}	a_{25}	a_{35}	a_{45}
x+5	a_{95}	$a_{10\ 5}$	a_{16}	a_{26}
x+6	a_{17}	a_{27}	a_{37}	a_{47}
x+7	a_{97}	$a_{10\ 7}$	a_{18}	a_{28}
x+8	a_{19}	a_{29}	a_{39}	a_{49}
x+9	a_{99}	$a_{10\ 9}$	$a_{1\ 10}$	$a_{2\ 10}$
x+10				
x+12				
x+13				
x+14				
x+15				

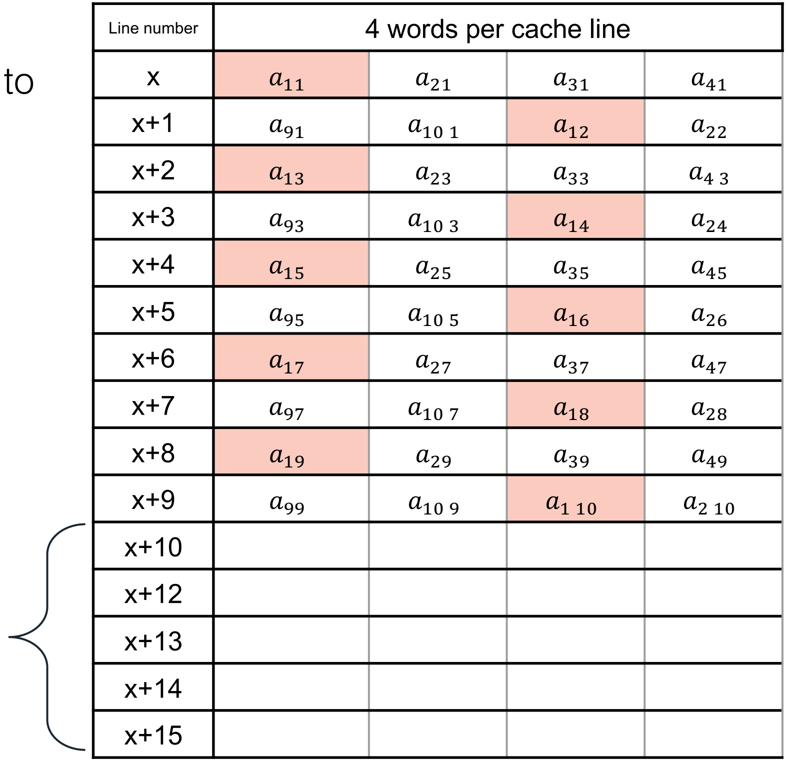
Sample Processor Cache
64 words, 4 words per line

Keep in mind: Transfers to cache are in blocks

- **Algorithm is inefficient:** many words transferred to cache that are not useful for the operation
- **Lines in cache occupied by unused words:** this increases cache misses

Sample traversal pseudocode:

```
for i=1 to n
    for j=1 to n
      {perform operation with $A_{ij}$}
    end for
end for
```



Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\,1}$	a_{12}	a_{22}
x+2	a_{13}	a_{23}	a_{33}	$a_{4\,3}$
x+3	a_{93}	$a_{10\,3}$	a_{14}	a_{24}
x+4	a_{15}	a_{25}	a_{35}	a_{45}
x+5	a_{95}	$a_{10\,5}$	a_{16}	a_{26}
x+6	a_{17}	a_{27}	a_{37}	a_{47}
x+7	a_{97}	$a_{10\,7}$	a_{18}	a_{28}
x+8	a_{19}	a_{29}	a_{39}	a_{49}
x+9	a_{99}	$a_{10\,9}$	$a_{1\,10}$	$a_{2\,10}$
x+10				
x+12				
x+13				
x+14				
x+15				

Remaining space in cache for additional data needed by the operation

Sample Processor Cache
64 words, 4 words per line

Let's look at a basic MMM algorithm (assume row-major memory)

MMM Algorithm 1 (ijk), row-major memory

- Consider the following pseudocode for MMM (let's call it "ijk"):

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
        end for
    end for
end for
```

- Multiply $n \times n$ matrices
  - For illustration, let's use small matrix $n = 4$
  - We are usually more concerned with large matrices
- Performance of algorithm: $O(n^3)$ total operations -> grows cubically with larger matrices

MMM Algorithm 1 (ijk), row-major memory

- Consider the following pseudocode for MMM (let’s call it “ijk”):

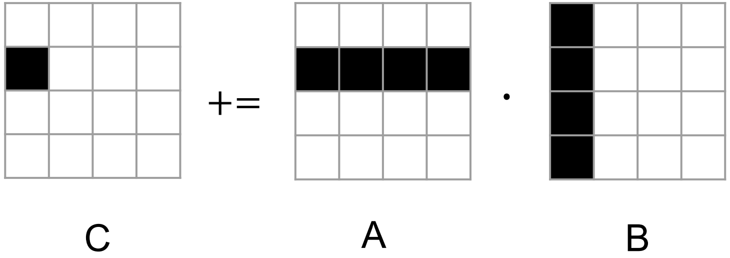
```
for i = 1 to n
    for j = 1 to n
              for k = 1 to n
                        $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
              end for
    end for
end for
```

- C_{ij} is initialized to zero matrix
- `i` keeps track of the row
- `j` keeps track of the column
- `k` iterates elements across the row of A and the column of B

MMM Algorithm 1 (ijk), row-major memory

- Consider the following pseudocode for MMM (let's call it "ijk"):

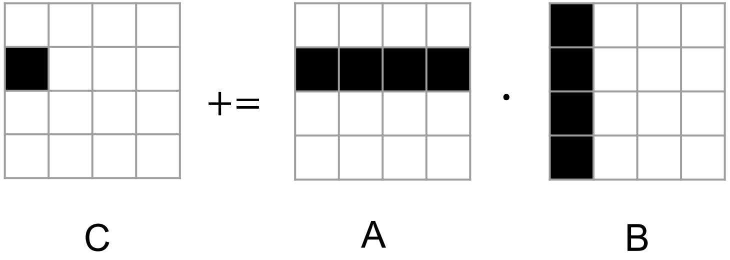
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
        end for
    end for
end for
```



- We traverse A row-wise and traverse B columnwise
  - Load row i of A (successive in main memory) into cache *once* until the entire computation for row i of C finishes
  - We load a new column j of B (costly: not successive in memory) whenever innermost loop completes an iteration k=1 to n

MMM Algorithm 1 (ijk): Cache behavior

- Suppose we are solving for C_{21}
  - Elements of row $i = 2$ of A are successively stored in main memory
  - Elements of column $j = 1$ of B are not stored successively
- Algorithm has spatial and temporal locality wrt accessing elements of A
- No spatial locality accessing elements of B (cache misses, especially for very large matrices)



What if we swap the i and j loops (“jik”)?

```
for j = 1 to n
    for i = 1 to n
            for k = 1 to n
                    $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
            end for
    end for
end for
```

- No spatial locality accessing elements of A (cache misses, especially for large matrices)
- Enjoy spatial locality accessing elements of B
- Roughly same performance

What about “kji”?

```
for k = 1 to n
    for j = 1 to n
              for i = 1 to n
                        $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
              end for
    end for
end for
```

- Innermost loop iterates something like the following:
  - $C_{11} = C_{11} + A_{11} * B_{11}$
  - $C_{21} = C_{21} + A_{21} * B_{11}$
  - $C_{31} = C_{31} + A_{31} * B_{11}$

- B is fixed, but traverse C and A column-wise
- Encounter cache misses for elements of *both* C and A at each iteration of the innermost loop
- Likely to have poorer performance

Try it yourself!

- Try to implement ijk, jik, kji, kij, and other variants of nested loops of MMM
- Time the execution of each and compare the results