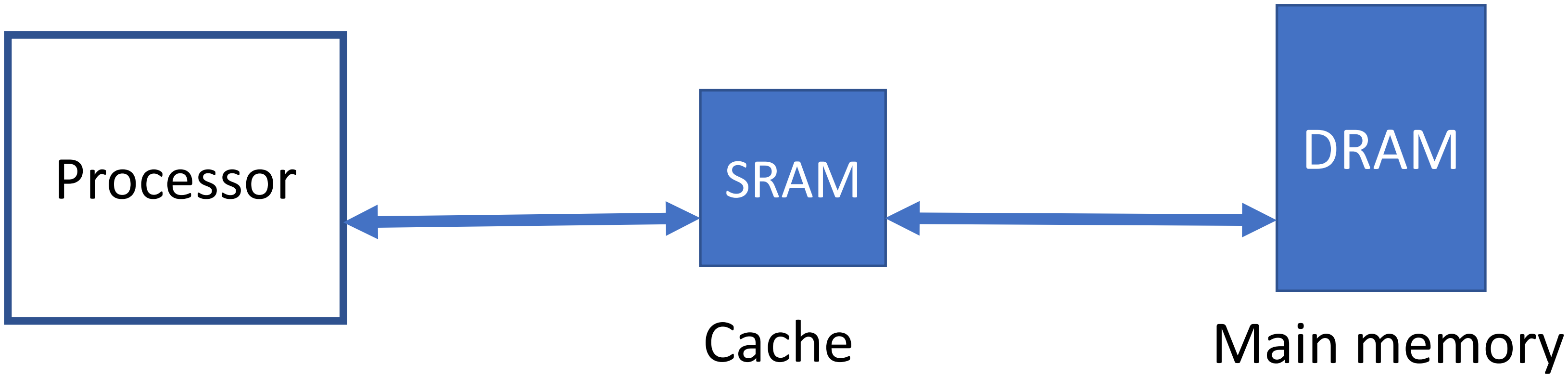
Application of memory management in C: Cache-efficient algorithms Matrix Multiplication

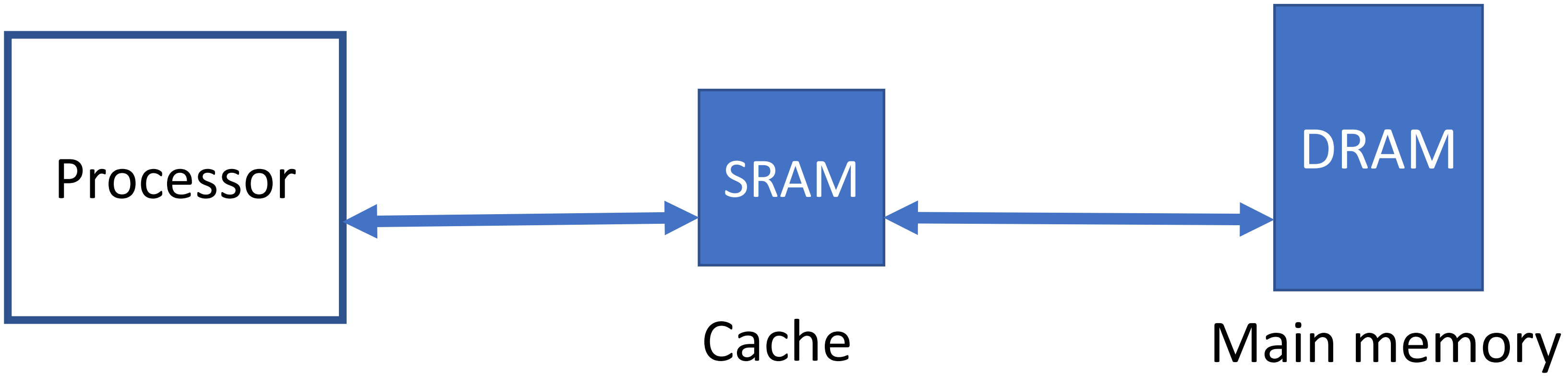
Sujoy Sinha Roy
School of Computer Science
University of Birmingham

Hypothetical computer



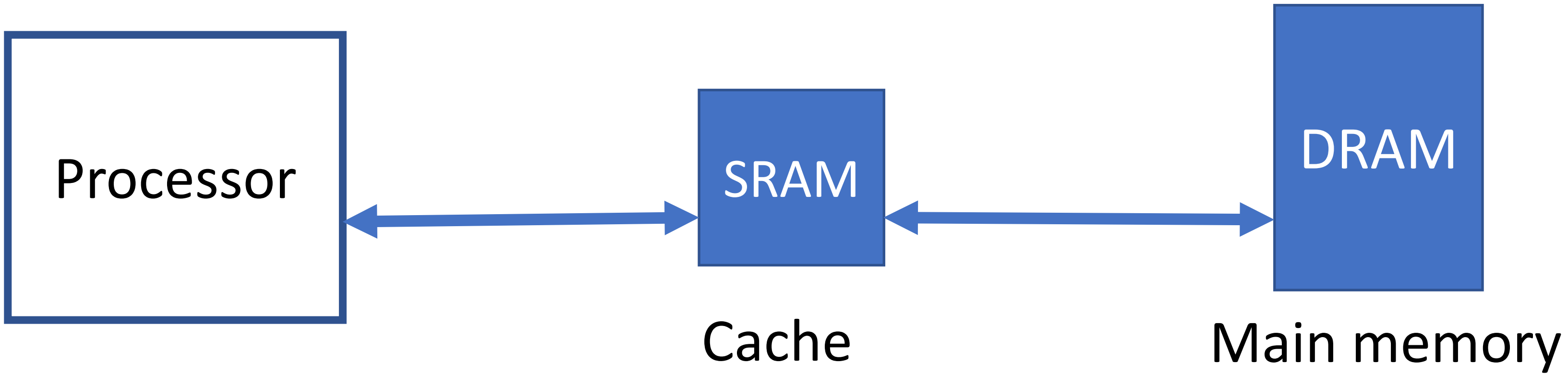
Assumption: processor can access fast cache in negligible time

Simplified model for computation-communication tradeoff



- Assume a generic program which performs
  - m = number of data movements between fast and slow memory
  - f = number of arithmetic operations

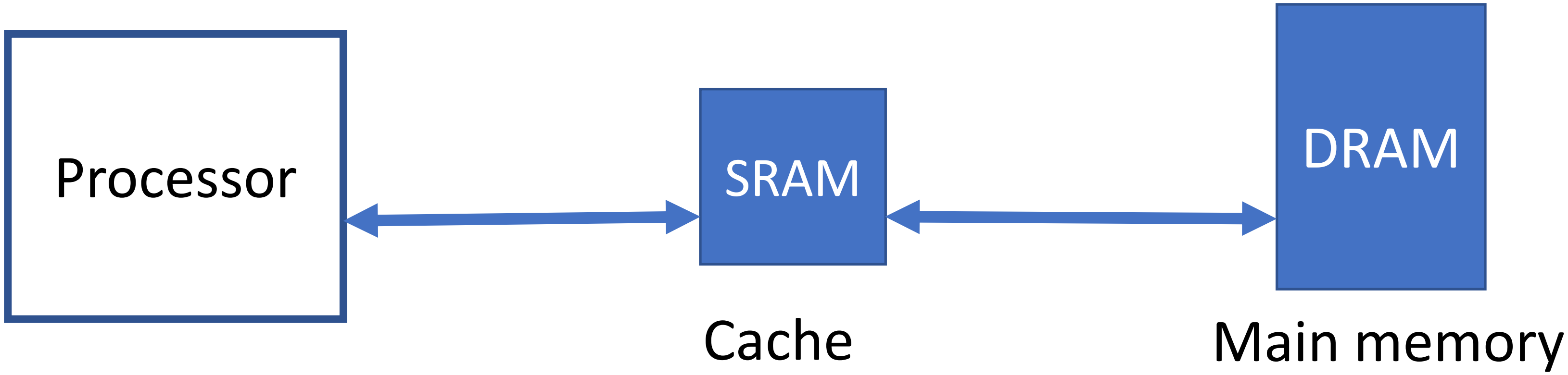
Simplified model for computation-communication tradeoff



- Assume a generic program which performs
  - m = number of data movements between fast and slow memory
  - f = number of arithmetic operations
  - t_m = time per data movement
  - t_f = time per arithmetic operation

- Total time for program execution = $f * t_f + m * t_m$

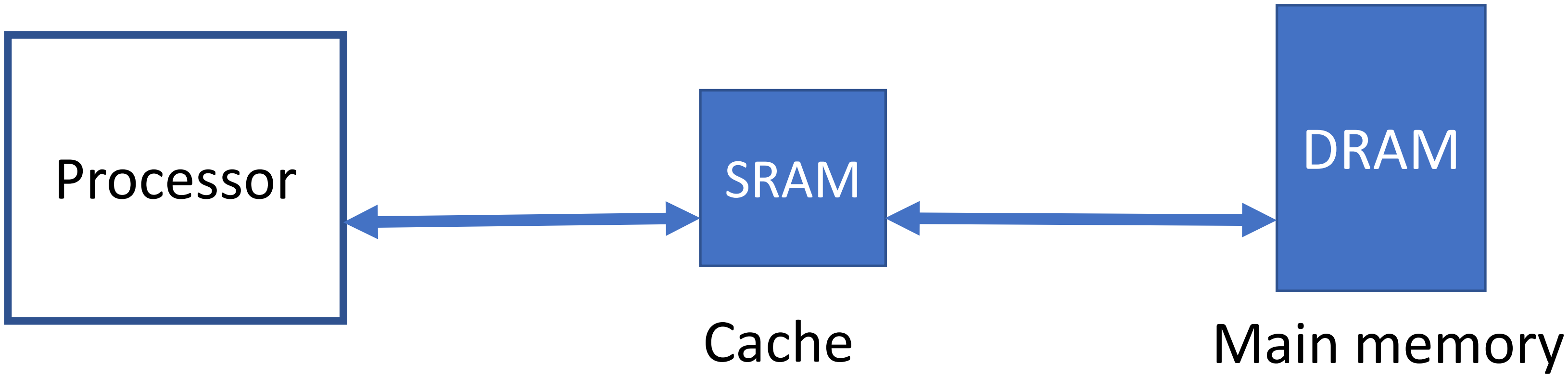
Simplified model for computation-communication tradeoff



- Assume a generic program which performs
  - m = number of data movements between fast and slow memory
  - f = number of arithmetic operations
  - t_m = time per data movement
  - t_f = time per arithmetic operation
  - q = f/m average number of arithmetic operations per slow memory access

Higher q indicates that the algorithm is ***computation-centric*** and performs less data movement.
So, a higher q results in a better system-performance.

Simplified model for computation-communication tradeoff



- Assume a generic program which performs
  - m = number of data movements between fast and slow memory
  - f = number of arithmetic operations
  - t_m = time per data movement
  - t_f = time per arithmetic operation
  - q = f/m average number of arithmetic operations per slow memory access

- Total time for program execution = $f * t_f + m * t_m$

$$= f * t_f * (1 + t_m/t_f * \mathbf{1/q})$$

Total time for program execution = $f * t_f + m * t_m$

$= f * t_f * (1 + t_m/t_f \; * \; \mathbf{1/q})$

where

- t_m = time per data movement (speed of memory)
- t_f = time per arithmetic operation (speed of processor)

Hardware constants (programmer cannot control)

Goal: Cache-efficient algorithms increase **q** by reducing the number of slow main-memory access → improves system performance.

Case study: Arithmetic on dynamically allocated matrices

Assume we have three matrices:

$$A = \begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \qquad B = \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

$$C = \begin{vmatrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{vmatrix}$$

We want to compute [C] = [C] + [A]*[B]

Creating a dynamically-allocated matrix

We can create a matrix with *n* rows and *n* columns as

```
T *A;

A = (T *) malloc(n*n*sizeof(T));
```

If we want to store data in row-major order then

```
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j)
      scanf("%f", p+i*n+j);
}
```

If we want to store data in column-major order then

```
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j)
      scanf("%f", p+i+j*n);
}
```

Case study: Arithmetic on dynamically allocated matrices

Assume that matrices are stored in column-major order.

$$A=\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \qquad B=\begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

Logical view

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Memory layout

Note: All data is initially in slow Main memory

Compute C = C + A*B

Logical view

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

A and B are stored in slow memory in column-major order

Processor

-	-	-	-
-	-	-	-

Cache (fast but small)

Compute C = C + A*B

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

A and B are stored in slow memory in column-major order

Processor

-	-	-	-
-	-	-	-

Cache (fast but small)

Compute C = C + A*B

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

1. Read A00 into fast memory
2. Read B00 into fast memory

A and B are stored in slow memory in column-major order

Processor

-	-	-	-
-	-	-	-

Cache (fast but small)

Compute C = C + A*B

Logical view

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Spatial locality

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

A and B are stored in slow memory in column-major order

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

1. Read A00 into fast memory
2. Read B00 into fast memory

Processor

B00	B10	B20	B30
A00	A10	A20	A30

Cache (fast but small)

Due to spatial locality, not just A00 and B00, but also nearby elements get stored in the cache.

Compute C = C + A*B

A00	A01	A02	A03		B00	B01	B02	B03
A10	A11	A12	A13		B10	B11	B12	B13
A20	A21	A22	A23		B20	B21	B22	B23
A30	A31	A32	A33		B30	B31	B32	B33

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = **A00*B00**+A01*B10+A02*B20+A03*B30

A and B are stored in slow memory in column-major order

Processor

B00	B10	B20	B30
A00	A10	A20	A30

Cache (fast but small)

Processor can read only {A00, B00} from fast mem. So, C00 = A00*B00

Compute C = C + A*B

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = (A00*B00)+A01*B10+A02*B20+A03*B30

A and B are stored in slow memory in column-major order

Processor

B00	B10	B20	B30
A00	A10	A20	A30

Cache (fast but small)

Processor can get only B10 from Cache.
A01 is needed from Main memory.

Compute C = C + A*B

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

A and B are stored in slow memory in column-major order

Compute C00 = (A00*B00)+**A01*B10**+A02*B20+A03*B30

3. Read A01 into fast memory (cache miss)

Processor

B00	B10	B20	B30
A01	A11	A21	A31

Cache (fast but small)

Processor can read only {A01, B10} from fast mem

C00=C00+A01*B10

Compute C = C + A*B

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = (A00*B00+A01*B10)+**A02*B20**+A03*B30

4. Read A02 into fast memory (again cache miss!)

A and B are stored in slow memory in column-major order

Processor

B00	B10	B20	B30
A02	A12	A22	A32

Cache (fast but small)

Processor can read only {A02, B10} from fast mem

C00=C00+A02*B20

Compute C = C + A*B

A00	A01	A02	A03		B00	B01	B02	B03
A10	A11	A12	A13		B10	B11	B12	B13
A20	A21	A22	A23		B20	B21	B22	B23
A30	A31	A32	A33		B30	B31	B32	B33

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

A and B are stored in slow memory in column-major order

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

5. … and so on

Processor

B00	B10	B20	B30
A03	A13	A23	A33

Cache (fast but small)

Always cache miss for A[]

Observation:

Processor cannot exploit spatial locality

What if A and B are in row-major order?

$$
\left|\begin{matrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{matrix}\right|
\left|\begin{matrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{matrix}\right|
$$

Logical view

Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B01	B02	B03	B10	B11	B12	B13	B20	B21	B22	B30	B30	B31	B32	B33

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

A and B are stored in slow memory in **row-major** order

Processor

-	-	-	-
-	-	-	-

Cache (fast but small)

Can processor exploit spatial locality?

What if A and B are in row-major order? (Answer)

Logical view

A00	A01	A02	A03
A10	A11	A12	A13
A20	A21	A22	A23
A30	A31	A32	A33

B00	B01	B02	B03
B10	B11	B12	B13
B20	B21	B22	B23
B30	B31	B32	B33

Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B01	B02	B03	B10	B11	B12	B13	B20	B21	B22	B30	B30	B31	B32	B33

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

A and B are stored in slow memory in **row-major** order

Processor

-	-	-	-
-	-	-	-

Cache (fast but small)

Can processor exploit spatial locality?
Always cache miss for B[]

Column-major order: #comp, #comm

A00	A01	A02	A03		B00	B01	B02	B03
A10	A11	A12	A13		B10	B11	B12	B13
A20	A21	A22	A23		B20	B21	B22	B23
A30	A31	A32	A33		B30	B31	B32	B33

Logical view

Memory layout

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

To compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

m_{C00} = read B(, 0) once: #n memory access
+ read entire A(,): #n^2 memory access (due to cache misses)
+ read/write C00 once: #2 memory access
= (n^2+n+2)

f_{C00} = n multiplications + n additions = 2n

There are n^2 elements in matrix multiplication result, so total is

$m_{total} = n^2(n^2+n+2) \approx O(n^4)$ $f_{total} = 2n^3$ $\rightarrow$ $q = f_{total}/m_{total} = 1/n$

Better approach

A(,) in row-major and B(,) in column-major orders

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Logical view

Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Processor

-	-	-	-
-	-	-	-

Cache (fast but small)

A(,) in row-major and B(,) in column-major orders

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Logical view

Spatial locality

Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

1. Read A00 into fast memory
2. Read B00 into fast memory

Processor

B00	B10	B20	B30
A00	A01	A02	A03

Cache (fast but small)

A(,) in row-major and B(,) in column-major orders

$$\begin{vmatrix} \boxed{A00 \quad A01 \quad A02 \quad A03} \\ A10 \quad A11 \quad A12 \quad A13 \\ A20 \quad A21 \quad A22 \quad A23 \\ A30 \quad A31 \quad A32 \quad A33 \end{vmatrix} \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

Logical view

Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

3. Compute C00=A00*B00
4. Compute C00=C00+A01*A02
5. and the rest

Processor

B00	B10	B20	B30
A00	A01	A02	A03

Cache (fast but small)

Processor finds all required elements in cache ☺
[Very few cache miss]

[Row]-[Column] major orders: #Comm and #Comp

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

Logical view

Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

To compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

m_{C00} = read A(0,) once: #n memory access
+ read B(,0) once: #n memory access
+ read/write C00 once: #2 memory access
= (2n+2)

f_{C00} = n multiplications + n additions = 2n

There are n^2 elements in matrix multiplication result, so total is

$m_{total} = n^2(2n+2) \approx O(2n^3)$ $f_{total} = 2n^3$ → $q = f_{total}/_{mtotal} \approx 1$

n times better!

+ exploit temporal locality of A(i,)

[Row]-[Column] major orders: #Comm and #Comp

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30
C01 = A00*B01+A01*B11+A02*B21+A03*B31
C02 = A00*B02+A01*B12+A02*B22+A03*B32
C03 = A00*B03+A01*B13+A02*B23+A03*B33

Compute one row of C

$m_{C(0,\ )}$ = read A(0,) once: #n memory access
+ read B(,i) n times: #n^2 memory access
+ read/write C(0,) once: #2n memory access
= $(n^2 + 3n)$

$f_{C00} = n^2$ multiplications + n^2 additions = $2n^2$

Total cost for matrix multiplication

$m_{total} = n(n^2 + 3n) \approx O(n^3)$ $f_{total} = 2n^3$ → $q = f_{total}/m_{total} =$ **2**

even better!

What assumptions did we make?

$$\left|\begin{array}{cccc} \boxed{A00 \quad A01 \quad A02 \quad A03} \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array}\right| \left|\begin{array}{cccc} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{array}\right|$$

1. Cache memory is sufficiently large to store one row of A
2. and one column of B
3. Cache memory access has 0 overhead

In a real system, the advantage will be smaller compared to our hypothetical machine

Achieving $q > 2$

Partition matrix into blocks

$$\begin{vmatrix} \mathbf{A00} & \mathbf{A01} \\ \mathbf{A10} & \mathbf{A11} \end{vmatrix} \longleftarrow \begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix}$$

where

$$\mathbf{A00} = \begin{vmatrix} A00 & A01 \\ A10 & A11 \end{vmatrix} \qquad \mathbf{A01} = \begin{vmatrix} A02 & A03 \\ A12 & A13 \end{vmatrix} \qquad \text{etc.}$$

Bold font is used to represent a sub-matrix.

Multiplication after partitioning

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \quad \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

$$\begin{vmatrix} \mathbf{A00} & \mathbf{A01} \\ \mathbf{A10} & \mathbf{A11} \end{vmatrix} \qquad \begin{vmatrix} \mathbf{B00} & \mathbf{B01} \\ \mathbf{B10} & \mathbf{B11} \end{vmatrix}$$

Then matrix multiplication result is

$$\begin{vmatrix} \mathbf{C00} & \mathbf{C01} \\ \mathbf{C10} & \mathbf{C11} \end{vmatrix} = \begin{vmatrix} \mathbf{A00*B00+A01*B10} & \mathbf{A00*B01+A01*B11} \\ \mathbf{A10*B00+A11*B10} & \mathbf{A10*B01+A11*B11} \end{vmatrix}$$

Multiplication after partitioning

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \quad \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

$$\begin{vmatrix} \mathbf{A00} & \mathbf{A01} \\ \mathbf{A10} & \mathbf{A11} \end{vmatrix} \qquad \begin{vmatrix} \mathbf{B00} & \mathbf{B01} \\ \mathbf{B10} & \mathbf{B11} \end{vmatrix}$$

$$\begin{vmatrix} \mathbf{C00} & \mathbf{C01} \\ \mathbf{C10} & \mathbf{C11} \end{vmatrix} = \begin{vmatrix} \mathbf{A00*B00+A01*B10} & \mathbf{A00*B01+A01*B11} \\ \mathbf{A10*B00+A11*B10} & \mathbf{A10*B01+A11*B11} \end{vmatrix}$$

Assume we can fit three such sub-matrices in the cache.

Cache (fast but small)

Multiplication after partitioning

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \quad \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

$$\begin{vmatrix} \boxed{\mathbf{A00}} & \mathbf{A01} \\ \mathbf{A10} & \mathbf{A11} \end{vmatrix} \qquad \begin{vmatrix} \boxed{\mathbf{B00}} & \mathbf{B01} \\ \mathbf{B10} & \mathbf{B11} \end{vmatrix}$$

$$\begin{vmatrix} \boxed{\mathbf{C00}} & \mathbf{C01} \\ \mathbf{C10} & \mathbf{C11} \end{vmatrix} = \begin{vmatrix} \mathbf{A00*B00+A01*B10} & \mathbf{A00*B01+A01*B11} \\ \mathbf{A10*B00+A11*B10} & \mathbf{A10*B01+A11*B11} \end{vmatrix}$$

Assume we can fit three such sub-matrices in the cache.

Processor

C00
A00
B00

Cache (fast but small)

We can compute **entire** sub-matrix operation smoothly!
C00=A00*B00

Multiplication after partitioning

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \quad \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

$$\begin{vmatrix} \mathbf{A00} & \boxed{\mathbf{A01}} \\ \mathbf{A10} & \mathbf{A11} \end{vmatrix} \qquad \begin{vmatrix} \mathbf{B00} & \mathbf{B01} \\ \boxed{\mathbf{B10}} & \mathbf{B11} \end{vmatrix}$$

$$\begin{vmatrix} \boxed{\mathbf{C00}} & \mathbf{C01} \\ \mathbf{C10} & \mathbf{C11} \end{vmatrix} = \begin{vmatrix} \mathbf{A00*B00+A01*B10} & \mathbf{A00*B01+A01*B11} \\ \mathbf{A10*B00+A11*B10} & \mathbf{A10*B01+A11*B11} \end{vmatrix}$$

Assume we can fit three such sub-matrices in the cache.

Processor

C00
A01
B01

Cache (fast but small)

We can compute **entire** sub-matrix operation smoothly!
C00=C00+A01*B10

Multiplication after partitioning

$$\begin{vmatrix} A00 & A01 & A02 & A03 \\ A10 & A11 & A12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{vmatrix} \quad \begin{vmatrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{vmatrix}$$

$$\begin{vmatrix} \mathbf{A00} & \mathbf{A01} \\ \mathbf{A10} & \mathbf{A11} \end{vmatrix} \qquad \begin{vmatrix} \mathbf{B00} & \mathbf{B01} \\ \mathbf{B10} & \mathbf{B11} \end{vmatrix}$$

$$\begin{vmatrix} \mathbf{C00} & \mathbf{C01} \\ \mathbf{C10} & \mathbf{C11} \end{vmatrix} = \begin{vmatrix} \mathbf{A00*B00+A01*B10} & \mathbf{A00*B01+A01*B11} \\ \mathbf{A10*B00+A11*B10} & \mathbf{A10*B01+A11*B11} \end{vmatrix}$$

What can we do if these sub-matrices do not fit in Cache?

Answer: partition into multiple smaller sub-matrices such that they can be placed in the cache.

(this is the idea behind Tiled Matrix Multiplication)

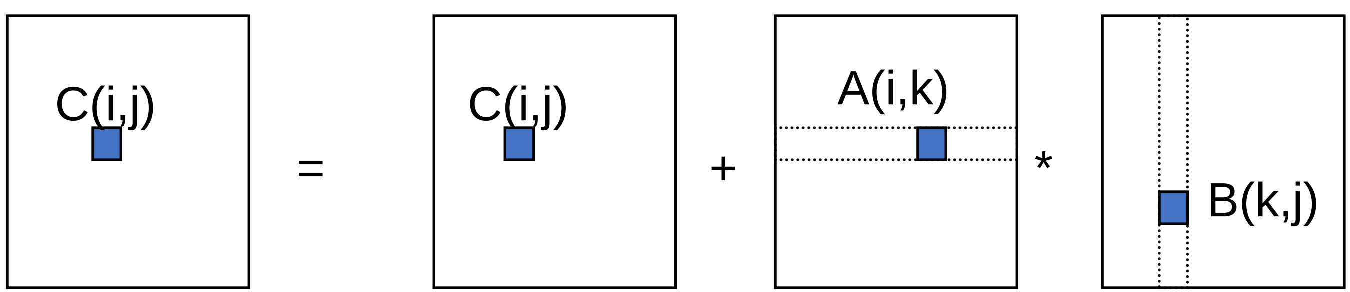
Tiled matrix multiplication

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

b=n / N is called the **block size**

$$\begin{vmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,N-1} \\ A_{1,0} & A_{1,1} & \dots & A_{1,N-1} \\ & & \dots & \\ A_{N-1,0} & & \dots & A_{N-1,N-1} \end{vmatrix}$$

```
for(i = 0; i<N; i++)
  for(j = 0; j<N; j++)
   {read block C(i,j) into fast memory}
   for(k = 1; k<N; k++)
         {read block A(i,k) into fast memory}
         {read block B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
   {write block C(i,j) back to slow memory}
```



Tiled matrix multiplication: #Comm and #Comp

$m = N^*n^2$ read blocks of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N^*n^2$)

$+ N^*n^2$ read blocks of A N^3 times

$+ 2n^2$ read and write each block of C once

$= (2N + 2) * n^2$

$f = 2n^3$

So computational intensity **$q = f / m = 2n^3 / ((2N + 2) * n^2)$**

$\approx n / N = b$ for large n

- By choosing b>2 we can achieve q>2
- Additionally q is proportional to block size b

Advantages of tiled matrix multiplication

Computational intensity **$q = f / m \approx b$ for large n**

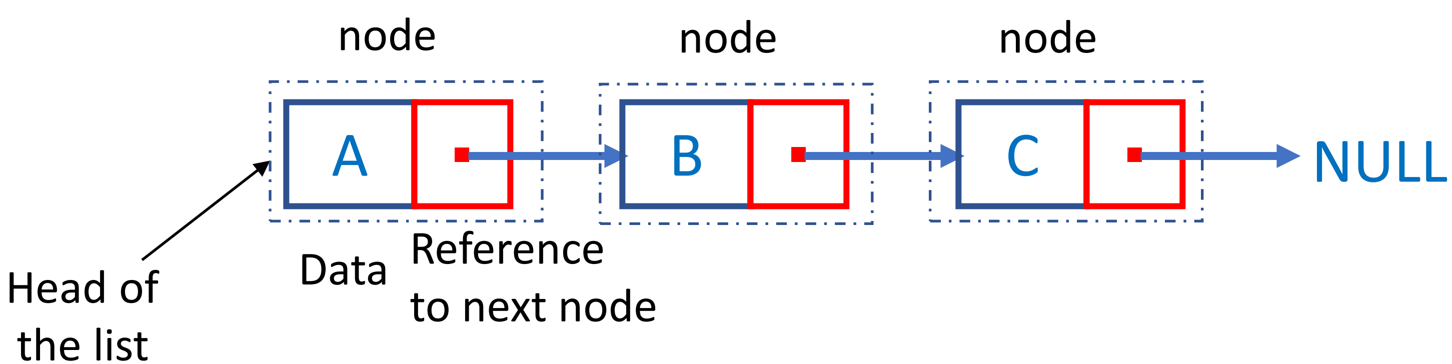
Advantages:

- Enhancement in computational intensity
- Flexibility depending on size of fast memory in machine (choose block size b according to system cache)

Linked List (Recap from last week)

A 'linked list' is a

- linear collection of data elements called 'nodes'
- each node points to the next node in the list
- unlike arrays, linked list nodes are not stored at contiguous locations; they are linked using pointers as shown below.



Can we have a cache-efficient linked-list? (or any other data structures)

Try yourself