ECE 454 Computer Systems Programming

Optimizing for Caches

Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

Content

- Cache basics and organization
- Understanding/Profiling memory
- Optimizing for caches (this lecture)
  - Loop reordering
  - Tiling/blocking
- Prefetching (later)
- Virtual memory (later)

Optimizing for Caches

Memory Optimizations

- Write code that has locality
  - Spatial: access data that is contiguous as much as possible
  - Temporal: access the same data within short intervals of time
- How to achieve locality?
  - Proper choice of algorithm
  - Loop transformations

Background: Array Allocation

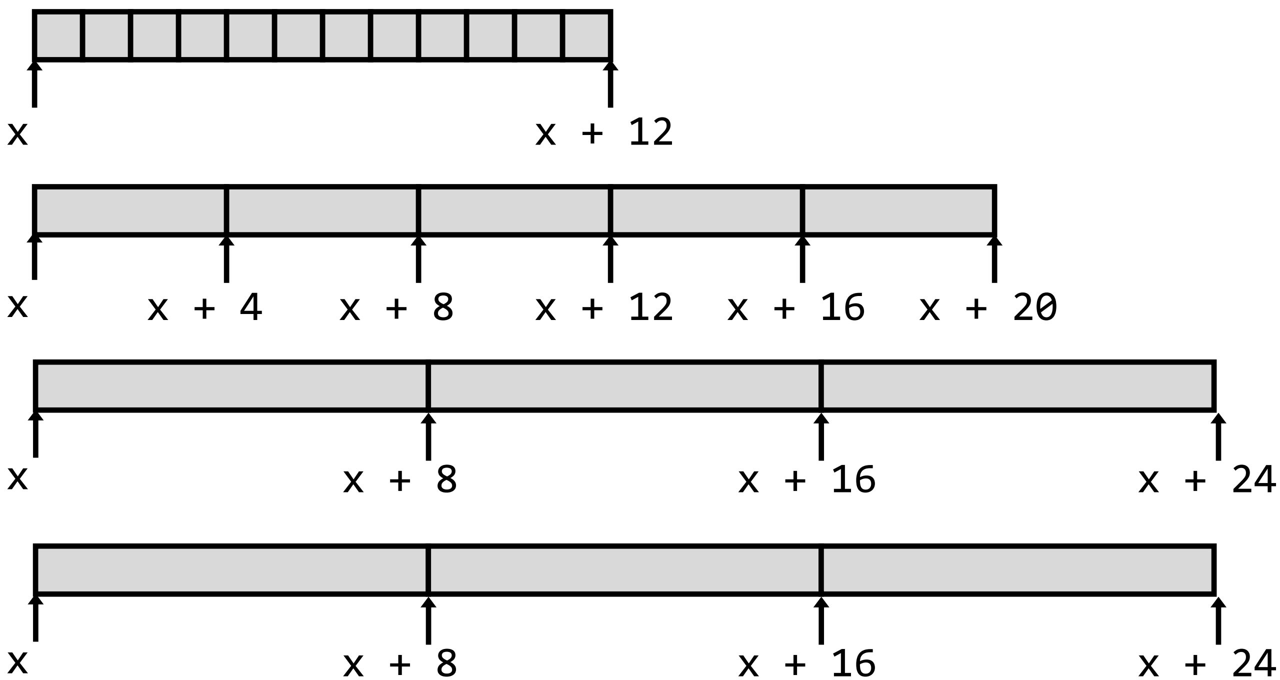
- `T  A[L];`
  - Array of data type `T` and length `L`
  - Contiguously allocated region of `L * sizeof(T)` bytes

```
char string[12];

int val[5];

double a[3];

char *p[3];
  (64 bit)
```

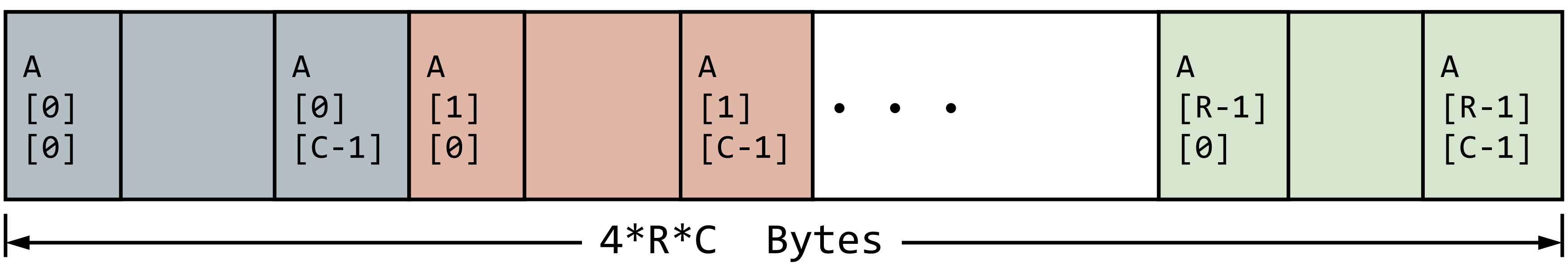


Multidimensional (Nested) Arrays

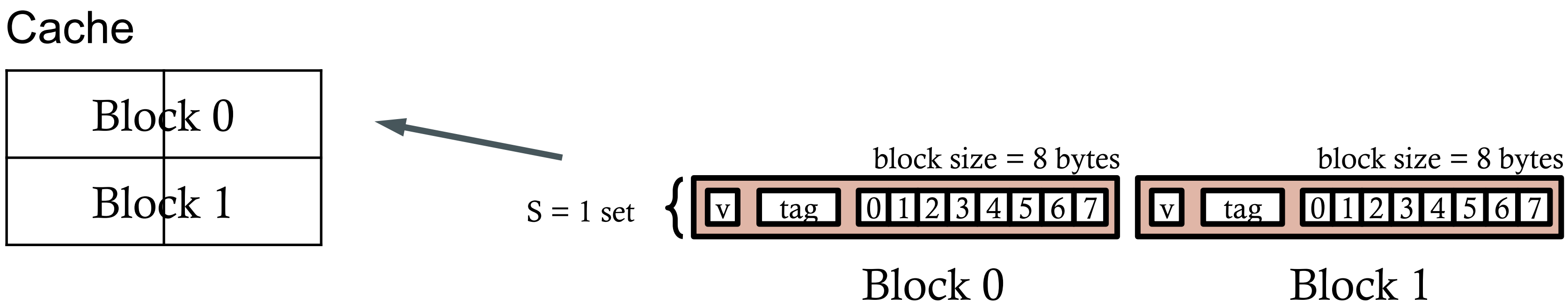
- Declaration: T   A[R][C];
  - 2D array of data type T
  - R rows, C columns
  - T element requires K bytes
- Arrangement
  - Row-Major Ordering (C code)

$$\begin{bmatrix} A[0][0] & \cdots & A[0][C-1] \\ \vdots & & \vdots \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{bmatrix}$$

Array Size: `R * C * K` bytes



Assumed Simple Cache



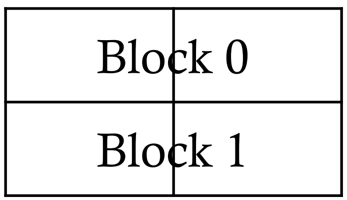
- 2-way set associative (2 blocks per set)
- 8 bytes per block, i.e., 2 4-byte integers
- 1 set
- Total size = 2 blocks, i.e., same as fully associative
- Replacement policy: Least Recently Used (LRU)

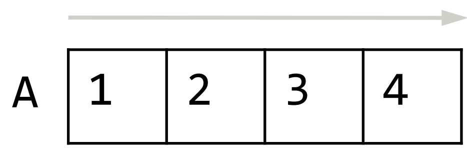
Some Key Questions

- How many elements are there per block?
- Does the data structure fit in the cache?
- Do I reuse blocks over time?
- In what order am I accessing blocks?

Simple Array

Cache



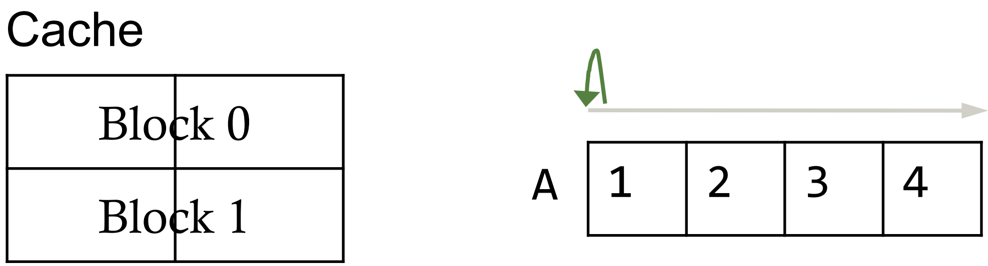



```
int A[N]; // 4 byte
          // elements

for (i=0; i<N; i++){
   … = A[i];
}
```

- Miss rate = #misses / #accesses = (N/2) / N = 1/2 = 50%

Simple Array with Outer Loop, Fits in Cache

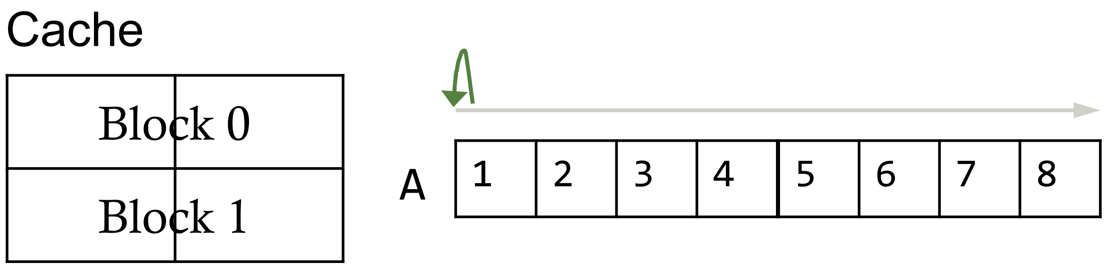



```
for (k=0; k<P; k++) {
  for (i=0; i<N; i++){
    … = A[i];
  }
}
```

- Assume array A fits in the cache
- Miss rate = #misses / #accesses = (N/2) / N*P = 1/(2P)

Lesson: for sequential accesses with re-use,
if data fits in the cache, first visit suffers all the misses

Simple Array with Outer Loop, Doesn't Fit in Cache




```
for (k=0; k<P; k++) {
  for (i=0; i<N; i++){
    … = A[i];
  }
}
```

- Assume array A does **not** fit in the cache
- Miss rate = #misses / #accesses = (N/2) / N = 1/2 = 50%

Lesson: for sequential accesses with re-use,
if the data doesn't fit, **same** miss rate as no-reuse

2D Array, Fits in Cache

Cache

Block 0	
Block 1	

A

1	2	3	4
5	6	7	8

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++){
    … = A[i][j];
  }
}
```

- Assume matrix A fits in the cache
- Miss rate = #misses / #accesses = $(N^2/2) / N^2 = 1/2 = 50\%$

2D Array, Doesn't Fit in Cache

Cache

Block 0
Block 1

A

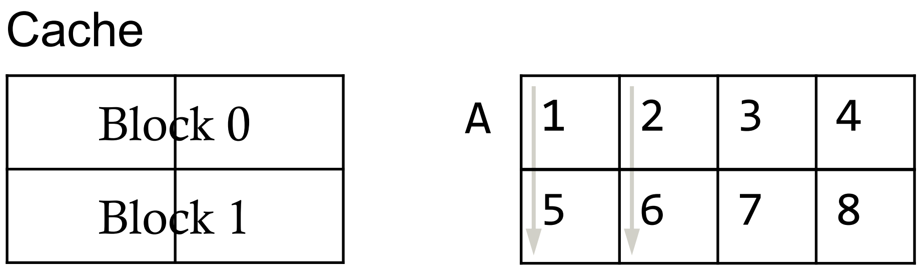
1	2	3	4
5	6	7	8

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++){
    … = A[i][j];
  }
}
```

- Assume matrix A does **not** fit in the cache
- Miss rate = #misses / #accesses = $(N^2/2) / N^2 = 1/2 = 50\%$

Lesson: for 2D accesses, row-order, no re-use, whether data fits or not, same rate as sequential

2D Array, Column Order, Column Fits in Cache




```
for (j=0; j<N/2; j++) {
  for (i=0; i<N; i++){
    … = A[i][j];
  }
}
```

- Assume matrix A fits in the cache
- Miss rate = #misses / #accesses = $(N^2/2) / N^2 = 1/2 = 50\%$

Lesson: for 2D accesses, column-order, no re-use, when column data fits in cache, same rate as sequential

2D Array, Column Order, Column Doesn't Fit in Cache

Cache

Block 0	
Block 1	

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

```
for (j=0; j<N; j++) {
  for (i=0; i<N; i++){
    … = A[i][j];
  }
}
```

- Assume matrix A does not fit in the cache
- Miss rate = #misses / #accesses = $N^2 / N^2 = 100\%$

Lesson: for 2D accesses, column-order, no re-use, when column data doesn't fit in cache, 100% miss rate

Loop Reordering/Interchange

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++){
    A[j][i] = i * j;
  }
}
```

```
for (j=0; j<N; j++) {
  for (i=0; i<N; i++){
    A[j][i] = i * j;
  }
}
```

- Initial code accesses data in column order
  - 100% miss rate, when column doesn’t fit in cache
- Loop reordering allows accessing data in row order
  - 50% miss rate

Matrix Multiplication

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
    B[k][j];
  }
}
```

2 2D Arrays

Cache

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

- Assume matrix A and matrix B do not fit in cache
- Assume a column of B does not fit at same time as a row of A
- Assume cache is fully set associative, LRU replacement policy
- Miss rate = #misses / #accesses =

2 2D Arrays

Cache		Time
		~~1~~ 3
		2
		4
		5

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
The inner most loop (i=j=0):}
A[0][0] * B[0][0], A[0][1] * B[1][0],
   A[0][2] * B[2][0], A[0][3] * B[3][0]
```

2 2D Arrays

Cache		Time
1	2	3
25	26	6
21	22	4
3	4	~~5~~ 7

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
The inner most loop (i=j=0):}
A[0][0] * B[0][0] A[0][1] * B[1][0]
   A[0][2] * B[2][0] A[0][3] * B[3][0]
```

2 2D Arrays

Cache		Time
29	30	8
25	26	6
21	22	4
3	4	7

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
The inner most loop (i=j=0):}
A[0][0] * B[0][0], A[0][1] * B[1][0],
   A[0][2] * B[2][0], A[0][3] * B[3][0]
```

2 2D Arrays

Cache		Time
29	30	8
25	26	6
21	22	4
3	4	7

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
    A[0][2] * B[2][1], A[0][3] * B[3][1]
```

2 2D Arrays

Cache		Time
29	30	8
25	26	6
1	2	9
3	4	7

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
   … = A[i][k] *
    B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
    A[0][2] * B[2][1], A[0][3] * B[3][1]
```

2 2D Arrays

Cache		Time
29	30	8
17	18	10
1	2	~~9~~ 11
3	4	7

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
    A[0][2] * B[2][1], A[0][3] * B[3][1]
```

2 2D Arrays

Cache		Time
29	30	8
17	18	10
1	2	11
21	22	12

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
    A[0][2] * B[2][1], A[0][3] * B[3][1]
```

2 2D Arrays

Cache / Time

Cache		Time
3	4	13
17	18	10
1	2	11
21	22	12

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
    A[0][2] * B[2][1], A[0][3] * B[3][1]
```

2 2D Arrays

Cache		Time
3	4	~~13~~ 15
25	26	14
1	2	11
21	22	12

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
    A[0][2] * B[2][1], A[0][3] * B[3][1]
```

2 2D Arrays

Cache		Time
3	4	15
25	26	14
29	30	16
21	22	12

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

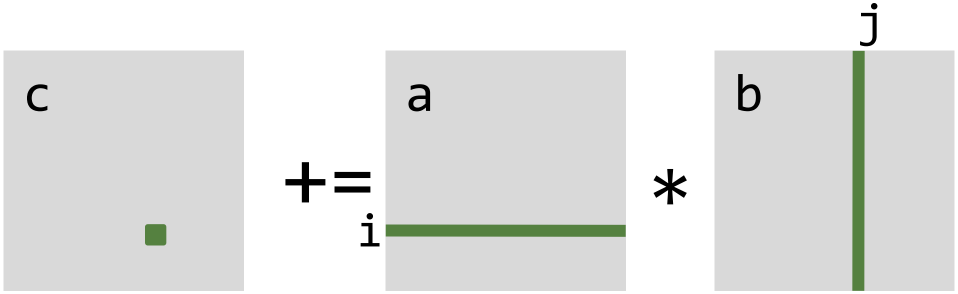
```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    for (k=0;k<N;k++){
    … = A[i][k] *
     B[k][j];
  }
}
```

```
Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],
  A[0][2] * B[2][1], A[0][3] * B[3][1]
```

- Assume matrix A and matrix B do not fit in cache
- Assume a column of B does not fit at same time as a row of A
- Miss rate = #misses / #accesses = 75%

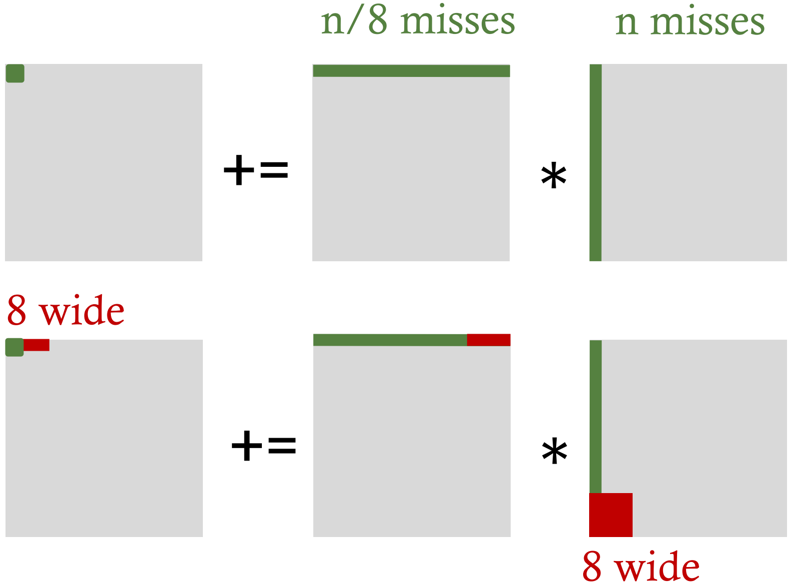
Matrix Multiplication

```
/* Multiply N x N matrices a and b  */
mmm(double a[][N], double b[][N], double c[][N])
{
    int i, j, k;
    for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
           c[i][j] += a[i][k]*b[k][j];
}
```



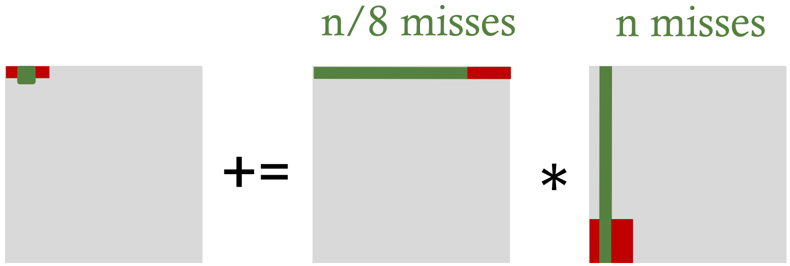
Cache Miss Analysis, First Iteration

- Assume:
  - Matrix elements are doubles
  - Cache block 64B = 8 doubles
  - Cache capacity << n (much smaller than n)
    - i.e., can't even hold an entire row in the cache!
- On first iteration:
  - How many misses?
  - n/8 + n = 9n/8 misses
- At the end of first iteration:
  - in cache



Cache Miss Analysis, Second Iteration

- Assume:
  - Matrix elements are doubles
  - Cache block 64B = 8 doubles
  - Cache capacity << n (much smaller than n)
    - i.e., can't even hold an entire row in the cache!
- On second iteration:
  - How many misses?
  - n/8 + n = 9n/8 misses

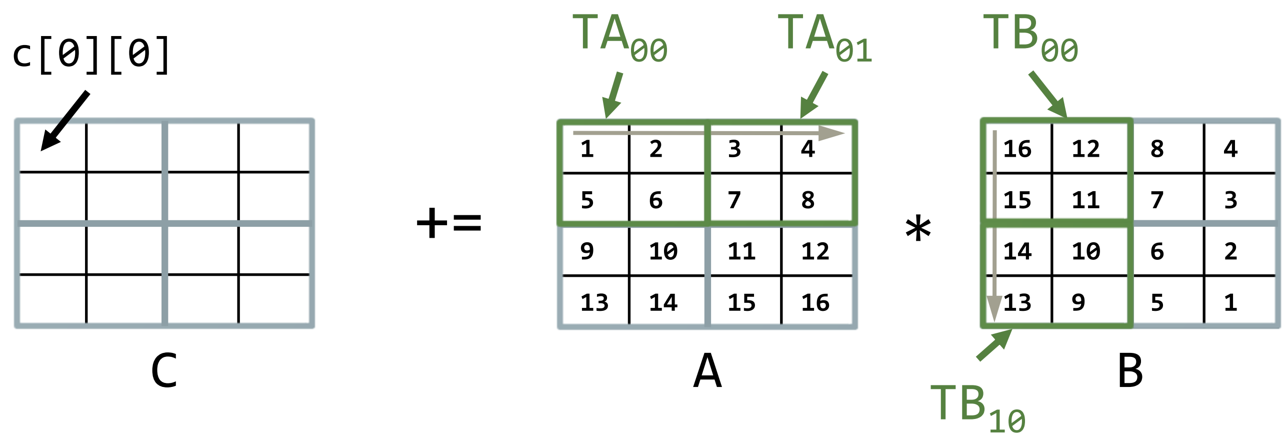


- Total misses for matrix multiplication:
  - $(9n/8) * n^2 = (9/8)*n^3$

Improving Cache Reuse

- Misses are expensive
  - L1 cache reference: 1-4 ns (L1 cache size: 32 KB)
  - Main memory reference: 100 ns (memory size: 4-256 GBs)
- Matrix multiplication has lots of data re-use
  - Key idea: Try to use entire cache block once it is loaded
  - Challenge: We need to work with both rows and columns
- Solution:
  - Operate in sub-squares of the matrices
    - One sub-square per matrix should fit in cache simultaneously
    - Heavily re-use the sub-squares before loading new ones
  - Called Tiling or Blocking (a sub-square is a tile)

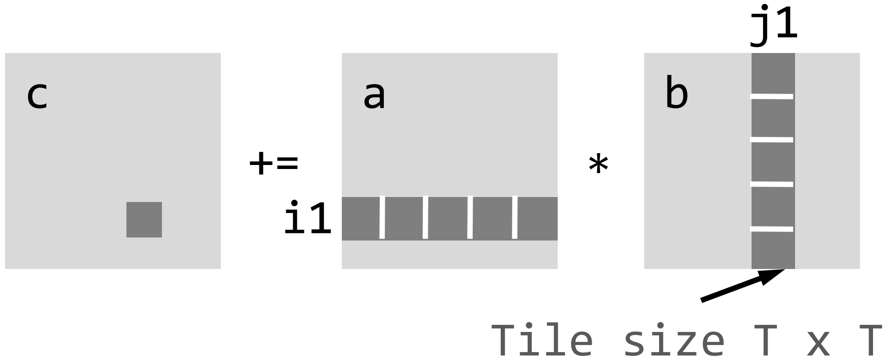
Idea for Tiled Matrix Multiplication



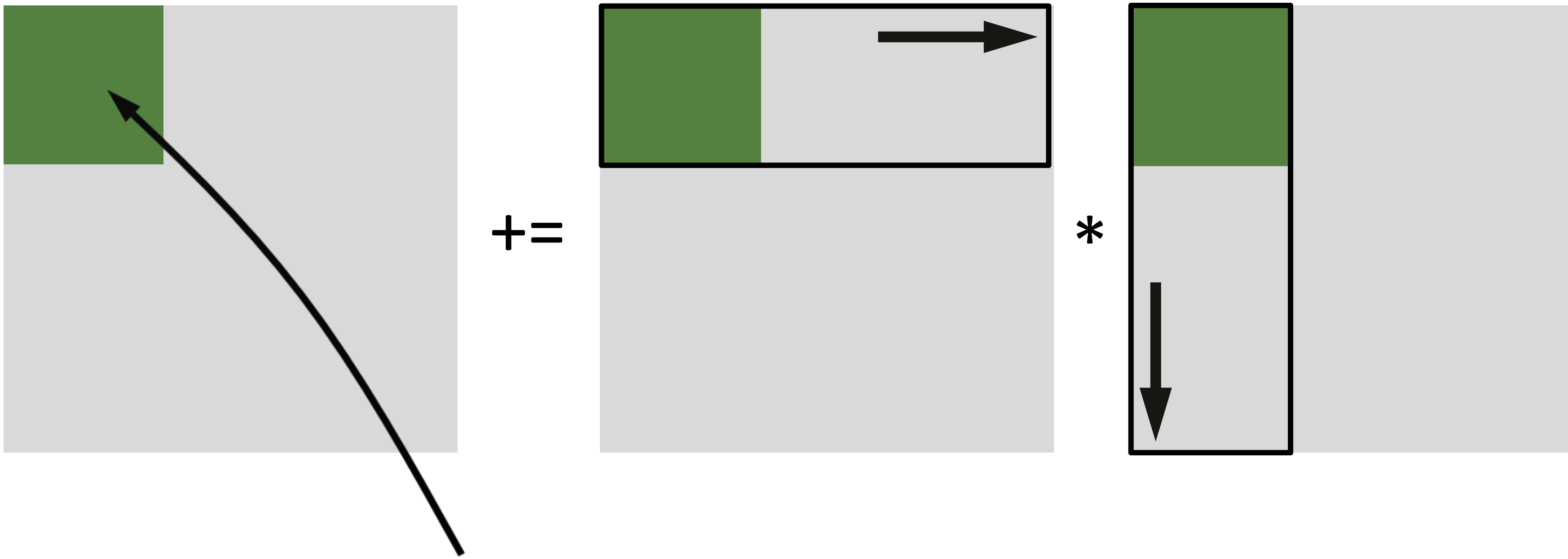
- Perform tile based mini-multiplications
  - E.g., $TA_{00} * TB_{00}$, $TA_{00} * TB_{01}$
- $$C[0][0] = \underbrace{1*16 + 2*15}_{\text{Perform as part of } TA_{00} * TB_{00}} + \underbrace{3*14 + 4*13}_{\text{Perform as part of } TA_{01} * TB_{10}}$$

Tiled Matrix Multiplication

```
/* Multiply N x N matrices a and b  */
void mmm(double a[][N], double b[][N], double c[][N], int T) {
    int i, j, k;
    for (i = 0; i < n; i+=T)
      for (j = 0; j < n; j+=T)
        for (k = 0; k < n; k+=T)
          /* T x T mini matrix multiplications */
          for (i1 = i; i1 < i+T; i1++)
            for (j1 = j; j1 < j+T; j1++)
              for (k1 = k; k1 < k+T; k1++)
                  c[i1][j1] += a[i1][k1]*b[k1][j1];
}
```

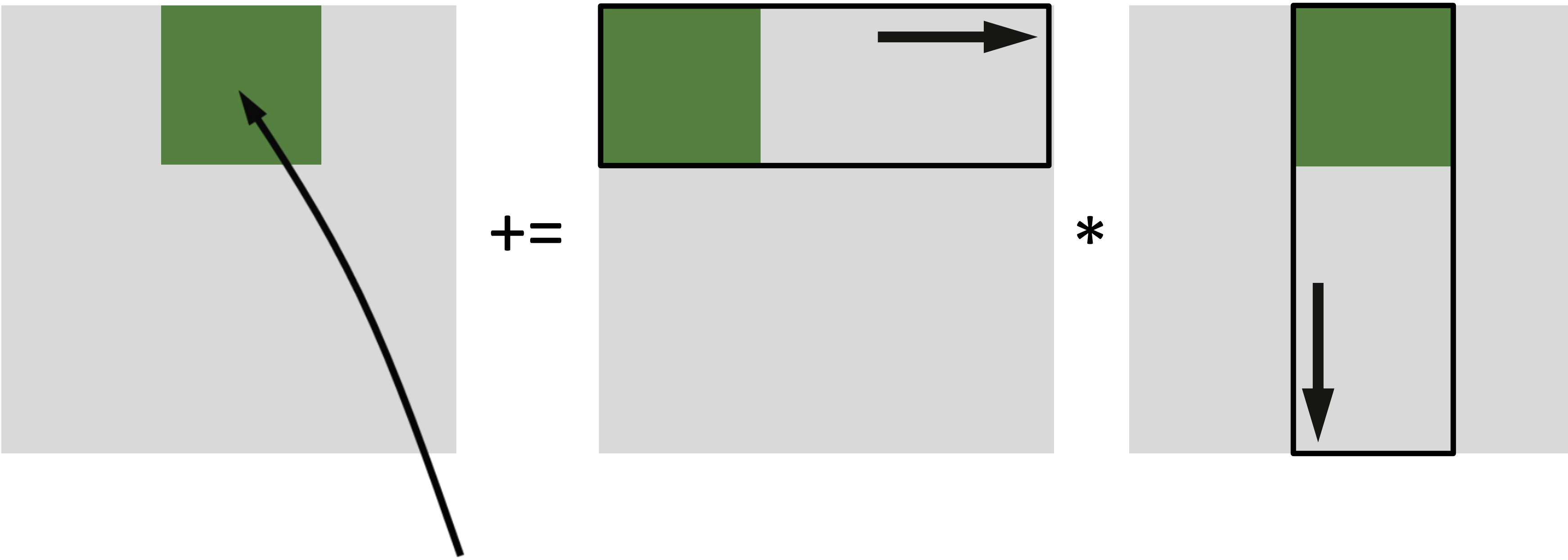


Visualization



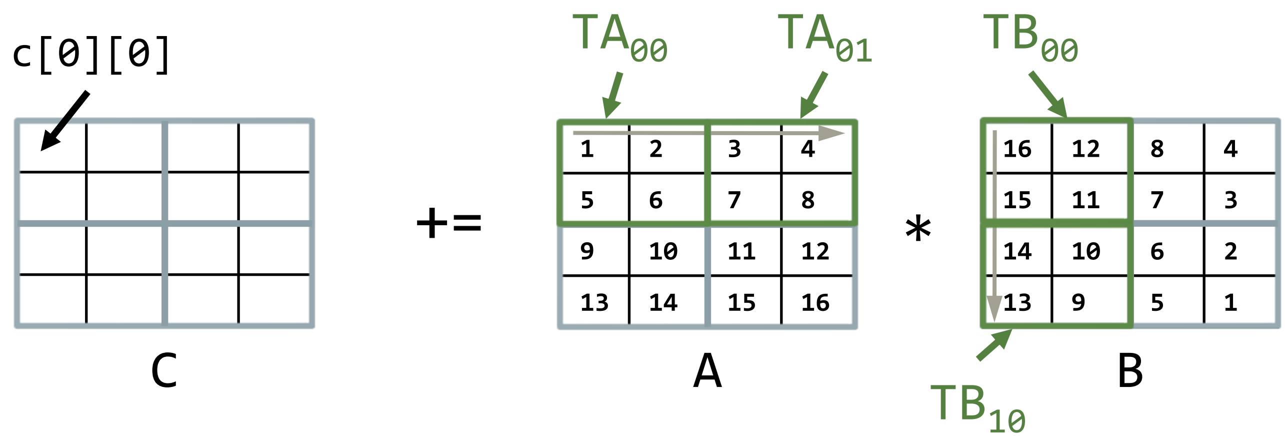
- First calculate TC_{00} = C[0][0] – C[T-1][T-1]

Visualization



- Next calculate TC_{01} = C[0][T] – C[T-1][2T-1]

Why Does Tiling Work?



- $$C[0][0] = \underbrace{1*16 + 2*15}_{\text{Perform as part of } TA_{00} * TB_{00}} + \underbrace{3*14 + 4*13}_{\text{Perform as part of } TA_{01} * TB_{10}}$$
- Still have to access `B[ ]` column-wise
- But now `B[ ]`’s cache blocks don’t get replaced

Tiling Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache capacity << n (much smaller than n)
  - With 3 arrays, need to fit 3 tiles in cache
    - Cache capacity > $3T^2$
- Misses per tile-iteration:
  - $T^2/8$ misses to load tile
  - $(2n/T) * (T^2/8) = nT/4$
- Total misses:
  - Tiled: $(nT/4) * (n/T)^2 = (1/(4T)) * n^3$
  - Untiled: $(9/8) * n^3$