CS420 – Lecture 4

Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Improving cache performance

- Cache is not controlled by software
- But cache performance depends on how memory accesses are organized.
- *Cache hit ratio*: ratio between number of accesses serviced by cache and total memory accesses
- More specifically, will have L1 hit ratio, L2 hit ratio, L3 hit ratio.

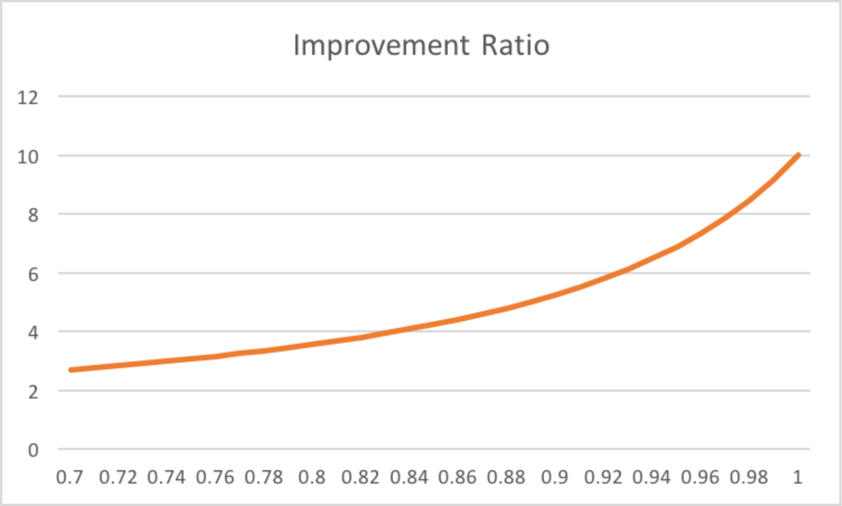
A simple performance model

T memory access time
τ cache access time
β cache hit ratio
Average access time without cache is T
Average access time with cache is $(1-\beta)T + \beta\tau$.
The ratio is

$$\frac{T}{(1-\beta)T + \beta\tau} = \frac{T/\tau}{(1-\beta)(T/\tau) + \beta}$$

Improvement in access time, as function of cache hit rate

Assume $T/\tau = 10$



- *We need high cache-hit ratios!*
- If accesses are random then cache hit ratio is – essentially zero
- Need *Temporal locality*: If word is accessed, then it is reaccessed in close time proximity
- Need *Spatial locality*: if word in cache line is accessed then other words in same cache line are accessed in close time proximity

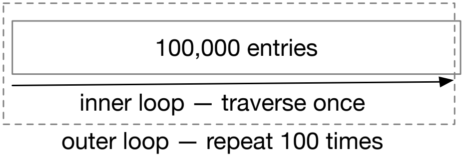
Experiment – temporal locality

```c
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 100000
#define M 100
#define s 1103515245
#define t 12345
#define rmax 2147483648
long int a[N];
int main()
{
  long int i,j,k;
  long int m = 1;
  time_t time;
  for(i = 0; i < N; i++) a[i] = i;
```

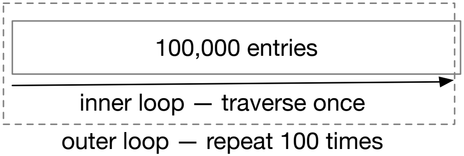
```c
for (k = 0; k < 10; k++) {
  time = clock();
  for (j = 0; j < M; j++) {
    for(i = 0; i < N; i++) {
      // assign a random number
      m = (m*s+t)%rmax;
      a[i]=m%N;
    }
  }
  time = clock()-time;
  printf("%lu ",
    1000*time/CLOCKS_PER_SEC);
}
printf("\n");
}
```

```
for (j = 0; j < M; j++) {
  for(i = 0; i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    a[i] = m%N;
  }
}
```

```
for (j = 0; j < M; j++) {
  for(i = 0; i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    a[i] = m%N;
  }
}
```

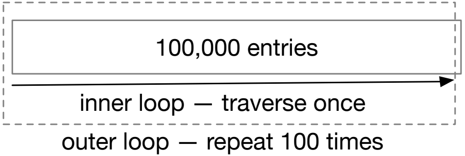


```
for (j = 0; j < M; j++) {
  for(i = 0; i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    a[i] = m%N;
  }
}
```



Running time 153 ms; 100 L1 misses per line

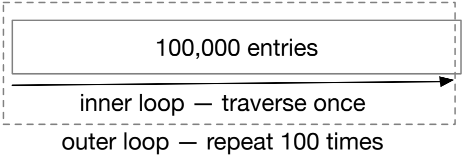
```
for (j = 0; j < M; j++) {
  for(i = 0; i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    a[i] = m%N;
  }
}
```



Running time 153 ms; 100 L1 misses per line

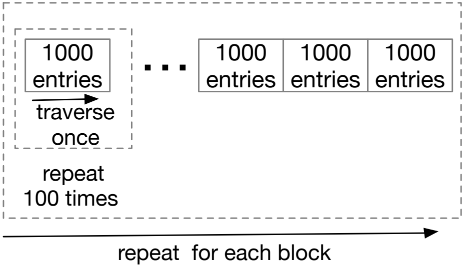
```
long int L=N/M;
...
for (i = 0; i < N; i += L) {
  for (j = 0; j < M; j++) {
    for(r = i; r < i+L; r++) {
      // assign a random number
      m = (m*s+t)%rmax;
      a[r] = m%N;
    }
  }
}
```

```
for (j = 0; j < M; j++) {
  for(i = 0; i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    a[i] = m%N;
  }
}
```

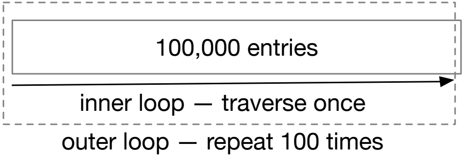


Running time 153 ms; 100 L1 misses per line

```
long int L=N/M;
...
for (i = 0; i < N; i += L) {
  for (j = 0; j < M; j++) {
    for(r = i; r < i+L; r++) {
      // assign a random number
      m = (m*s+t)%rmax;
      a[r] = m%N;
    }
  }
}
```

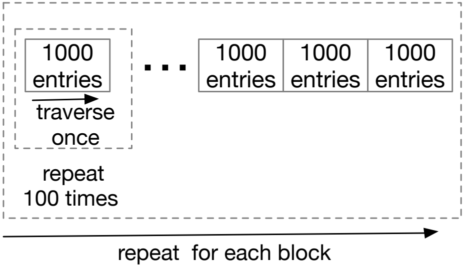


```
for (j = 0; j < M; j++) {
  for(i = 0; i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    a[i] = m%N;
  }
}
```



Running time 153 ms; 100 L1 misses per line

```
long int L=N/M;
...
for (i = 0; i < N; i += L) {
  for (j = 0; j < M; j++) {
    for(r = i; r < i+L; r++) {
      // assign a random number
      m = (m*s+t)%rmax;
      a[r] = m%N;
    }
  }
}
```



Running time 40 ms; 1 L1 miss per line

Experiment – spatial locality

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 10000000
#define s 1103515245
#define t 12345
#define rmax 2147483648
long int a[N]
  __attribute__((aligned(64)));
//align to line boundary
int main()
{
  long  int i,j,k,l;
  long int m=1;
  time_t time;
  for(i = 0; i < N; i++) a[i] = i;
```

```
for (k = 0; k < 10; k++) {
  time = clock();
  for(i = 0 ;i < N; i++) {
    // assign a random number
    m = (m*s+t)%rmax;
    l = m%N;
    a[l]=i;
  }
  time = clock()-time;
  printf("%lu ",
    1000*time/CLOCKS_PER_SEC);
}
printf("/n");
}
```

```
for(i = 0; i < N; i++) {
  // assign a random number
  m = (m*s+t)%rmax;
  l = m%N;
  a[l]=i;
}
```

```
long int n = N/8;
...
for(i = 0; i < n; i++) {
  // assign a random number
  m = (m*s+t)%rmax;
  l = 8*(m%n);
  for(j = 0; j < 8; j++) {
    a[l+j]=i;
  }
}
```

```
for(i = 0; i < N; i++) {
  // assign a random number
  m = (m*s+t)%rmax;
  l = m%N;
  a[l]=i;
}
```

Random access to 10,000,000 words in an array of size 10,000,000
average execution time 166.7 ($\sigma = 8.8$)

```
long int n = N/8;
...
for(i = 0; i < n; i++) {
  // assign a random number
  m = (m*s+t)%rmax;
  l = 8*(m%n);
  for(j = 0; j < 8; j++) {
    a[l+j]=i;
  }
}
```

Random access to 10,000,000/8 lines; words in each line are accessed sequentially
Average execution time 14.3 ($\sigma = 1.1$)

Example

Find how many numbers in a list of 50M numbers divide evenly by some number in a list of dividers (count with multiplicities)

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 50000000
#define L 8
int divs[L] =
    {2,3,5,7,11,13,17,19};
int num[N];
int sum[L];
int main()
{
  int i,j,k;
  time_t time;
```

```
  for (i = 0; i < N;i++) num[i] = random();
  for (k = 0; k < 9; k++) {
    time = clock();
    for (i = 0; i < L; i++) {
      for (j = 0; j < N; j++) {
        if(num[j]%divs[i]==0)
          sum[i]++;
      }
    }
    time=clock()-time;
    printf("%lu ",
    1000*time/CLOCKS_PER_SEC);
  }
printf("/n");
}
```

```
for (i = 0 ; i < L; i++)
 for (j = 0; j < N; j++)
  if(num[j]%divs[i]==0) sum[i]++;
```

```
for (j = 0; j < N; i++)
 for (i = 0; i < L; j++)
  if(num[j]%divs[i]==0) sum[i]++;
```

```
for (i = 0 ; i < L; i++)
 for (j = 0; j < N; j++)
  if(num[j]%divs[i]==0) sum[i]++;
```

execution time $= 2531 \pm 77$

```
for (j = 0; j < N; i++)
 for (i = 0; i < L; j++)
  if(num[j]%divs[i]==0) sum[i]++;
```

execution time $= 2358 \pm 62$

```
for (i = 0 ; i < L; i++)
 for (j = 0; j < N; j++)
  if(num[j]%divs[i]==0) sum[i]++;
```

execution time $= 2531 \pm 77$

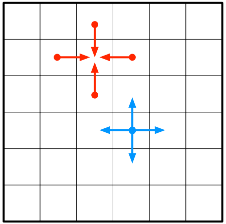
```
for (j = 0; j < N; i++)
 for (i = 0; i < L; j++)
  if(num[j]%divs[i]==0) sum[i]++;
```

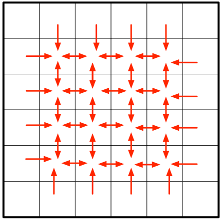
execution time $= 2358 \pm 62$

- First version traverses array `num[]` 8 times. Second version traverses it once
- Which version is better depends on the relative length of arrays `num[]` and `divs[]`
- Compiler can do *loop interchange optimizations* – but it may need guidance

Case study: Jacobi Algorithm

- Iterative algorithm in 2D; value at a point is repeatedly updated using value of neighbors.
- $a_{i,j}^{(k+1)} = 0.25(a_{i-1,j}^{(k)} + a_{i+1,j}^{(k)} + a_{i,j-1}^{(k)} + a_{i,j+1}^{(k)})$
- 4-neighbor *stencil*





Cannot update array in place – need two copies

Code

Boundary values do not change

```
double a[N][N], b[N][N];
...

while (!converged){
  for (i = 1; i < N-1; i++) {
    for(j = 1; j < N-1; j++) {
      a[i][j] = 0.25*(b[i-1][j]+b[i+1][j]
        +b[i][j-1]+b[i][j+1]);
    }
  }
  for (i = 1; i < N-1; i++) {
    for(j = 1; j < N-1; j++) {
      b[i][j] = a[i][j];
    }
  }
}
```

Code

Can avoid the copying by "playing ping-pong" with the two arrays

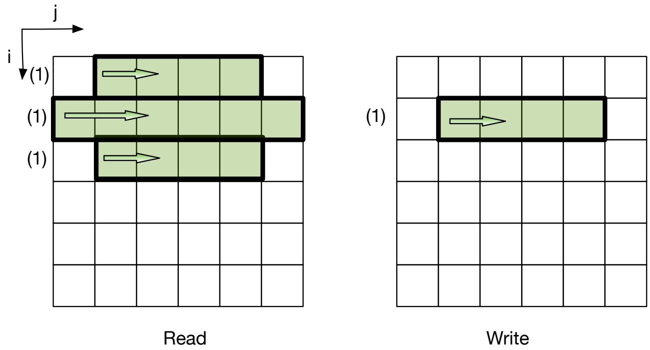
```
double a[2][N][N];
  ...

while (!converged) {
  for (i = 1; i < N-1; i++) {
    for(j = 1; j < N-1; j++) {
      a[1-k][i][j]=0.25*(a[k][i-1][j]
        +a[k][i+1][j]+a[k][i][j-1]
          +a[k][i][j+1]);
    }
  }
  k = 1-k;
}
```

Traversal order for current code

- Good spatial locality: Both matrices are traversed in storage order

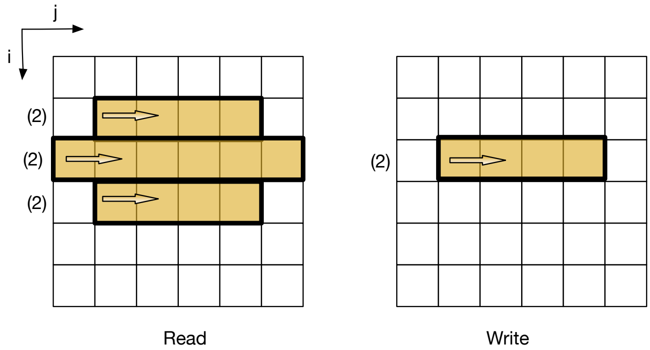
Traversal order for current code

- Good spatial locality: Both matrices are traversed in storage order



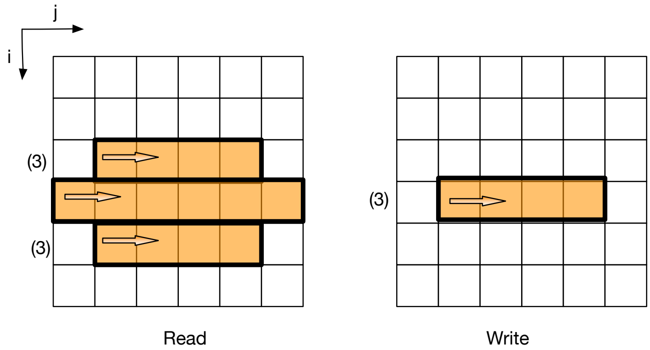
Traversal order for current code

- Good spatial locality: Both matrices are traversed in storage order



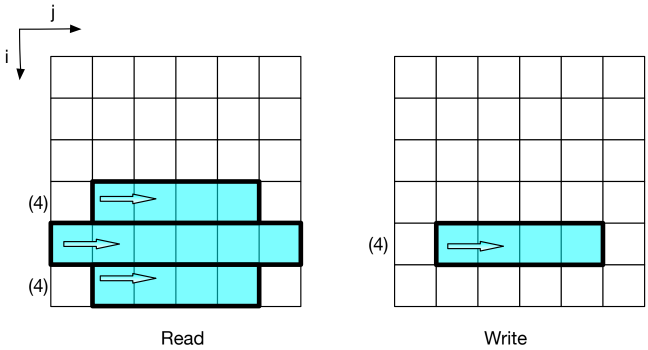
Traversal order for current code

- Good spatial locality: Both matrices are traversed in storage order

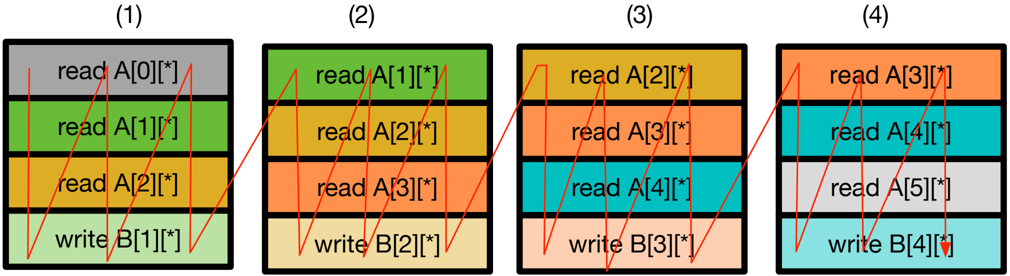


Traversal order for current code

- Good spatial locality: Both matrices are traversed in storage order



How many cache misses?



- Each row has $\sim N$ words
- If cache size $\gg 4N$ words ($32N$ bytes) then each word is read once.
  - Number of misses is $\sim 2N^2/8 = N^2/4$.
- If cache size $< 4N$ words each row in `A` is read 3 times (except row at top and bottom of matrix)
  - Number of misses is $\sim 4N^2/8 = N^2/2$

Jacobi

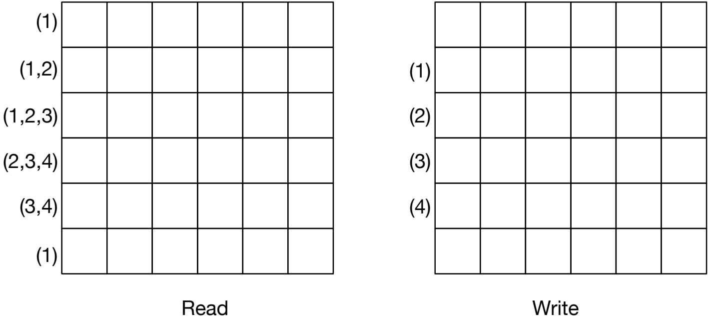
```
double a[2][M+2][N+2];
...

while (!converged){
  for (i = 1; i < N-1; i++) {
    for(j = 1; j < N-1; j++) {
      a[1-k][i][j]=0.25*(a[k][i-1][j]+a[k][i+1][j]
        +a[k][i][j-1]+a[k][i][j+1]);
    }
  }
  k = 1-k;
}
```

- Each row has $\sim N$ words
- If cache can hold 4 rows than each line is accessed once and we have $\sim N^2/4$ misses.
- Otherwise each row in `A` is read 3 times and number of misses is $\sim N^2/2$

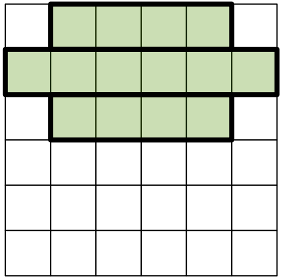
```
double a[2][M][N];
...

while (!converged){
 for (i = 1; i < N-1; i++) {
   for(j = 1; j < M-1; j++) {
     a[1-k][i][j]=0.25*(a[k][i-1][j]
        +a[k][i+1][j]+a[k][i][j-1]
        +a[k][i][j+1]);
   }
 }
 k = 1-k;
}
```

Assume cache can hold more than 4 rows

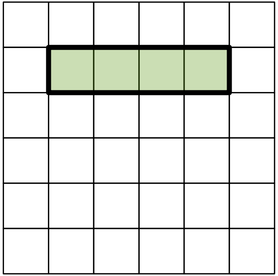
What row is accessed at each iteration of `i`



Cache content after 1st iteration

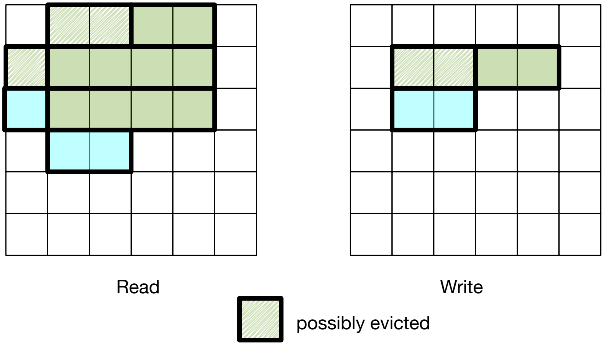


Read

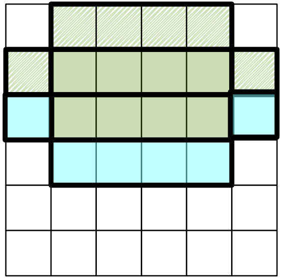


Write

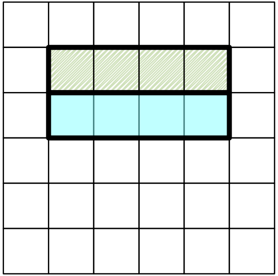
Cache content middle of 2nd iteration



Cache content after 2nd iteration

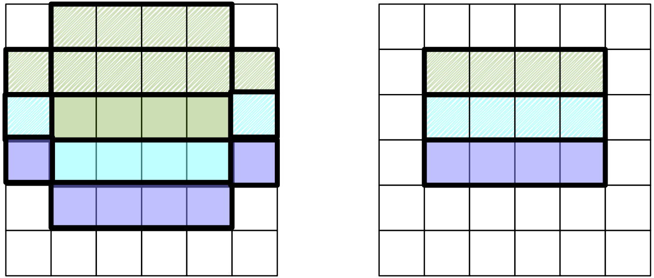


Read



Write

Cache content after 3rd iteration

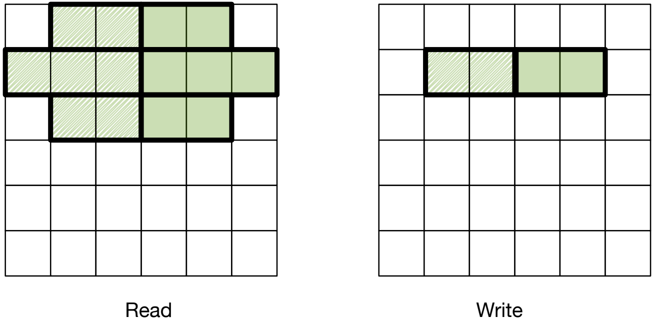


Read Write

- If cache can hold more than 4 rows than each line is accessed once
- Number of misses is $\sim 2N^2/8 = N^2/4$
- Assume ideal cache – actual performance may vary

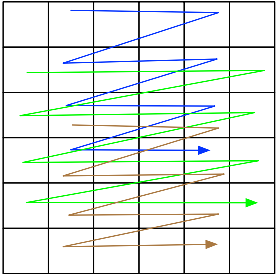
Cache cannot hold 4 rows

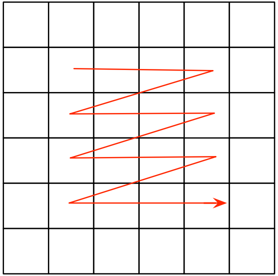
Cache content middle of 1st iteration



- Each row of left matrix is read from memory again at each iteration.
- $\sim 3N^2/8$ misses on left matrix
- $N^2/8$ misses on right matrix
- Total of $\sim N^2/2$ misses

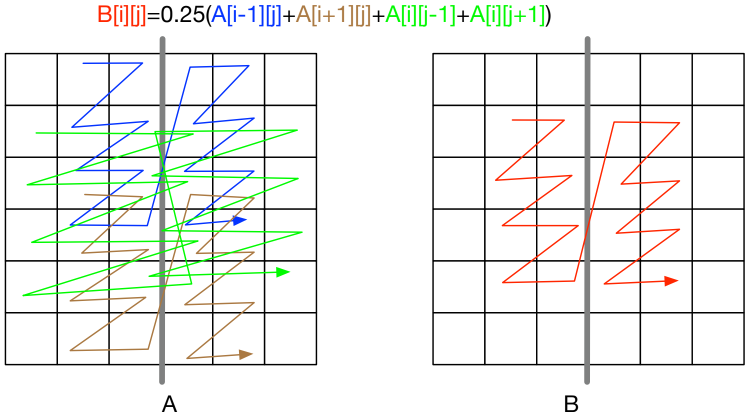
$B[i][j] = 0.25(A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])$





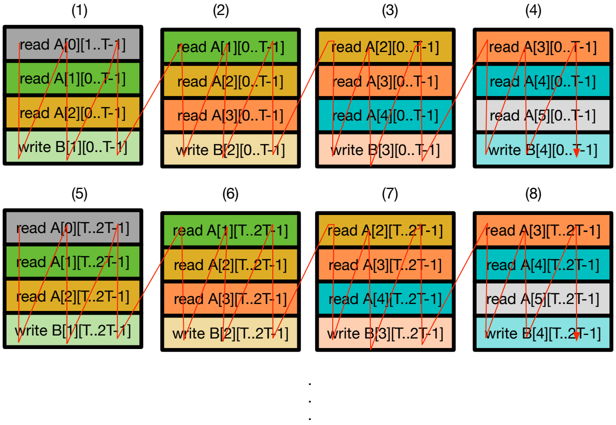
Tiling

- Bad performance if matrix is wide, i.e., if $N \gg C/32$, where C is size of cache in bytes.
- Idea: Divide matrix into narrower submatrices (tiles)



- Traversal order is tile:row:column

Another view



Code

```
double a[2][N][N];
\* N = n*T+2, T is tile size *\
...
for(jj = 0; jj < n; jj++) {
  jfirst = jj * T + 1;
  for (i = 1; i < N-1; i++) {
    for(j = jfirst; j < jfirst+T; j++) {
      a[k-1][i][j] = 0.25*(a[k][i-1][j]+a[l][i+1][j]
        +a[k][i][j-1]+a[k][i][j+1]+a[k][i][j]);
    }
  }
}
```

Pick largest tile width T so that 4 tile rows fit in cache (longer tile = better pipelining and prefetching)

- $T \approx C/32 \text{ bytes} = C/4 \text{ words}$
- Number of caches misses is $\sim N^2/4+$ (have additional misses at the “seams” between tiles).

- Number of memory accesses is $\sim 2N^2$
- Number of arithmetic operations is $\sim 4N^2$
- 2 operations per memory access
- Code has low *compute intensity* – memory bandwith will most likely be the bottleneck on performance — even after optimization