CS420 – Lecture 3

Marc Snir

Fall 2018

Grading

Type	Frequency	%
Quizzes	every 2 weeks	5%
MPs	every 3 weeks	25%
Midterm Exam	(Tentative: 10/12)	25%
Final Exam		35%

Grading Corrected

Type	Frequency	%
Quizzes	every 2 weeks	10%
MPs	every 3 weeks	30%
Midterm Exam	(Tentative: 10/12)	25%
Final Exam		35%

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.

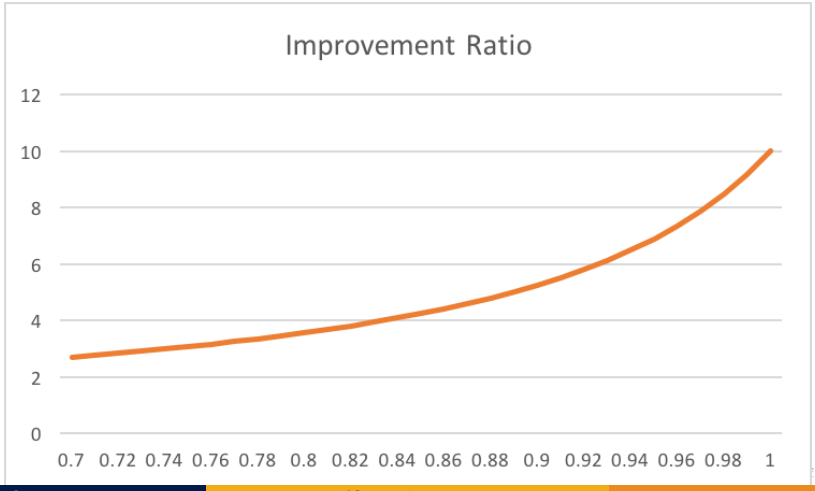
A simple performance model

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 $T/\tau = 10$



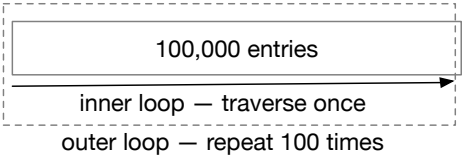
- *We need high cache-hit ratios!*
- If accesses are random then cache hit ratio is $= cache\_size/memory\_size$ – essentially zero (last lecture experiment)
- Need *Temporal locality*: If word is accessed, then it is reaccessed in close time proximity
- Need *Spatial locality*: if word in cache line is accessed then other words in same cache line are accessed in close time proximity

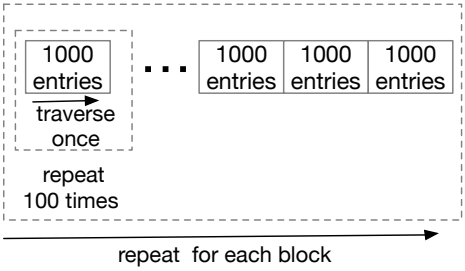
Experiment – temporal locality

```
#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;
for (k=0;k<10;k++) {
time = clock();
for (j=0;j<M;j++)
for(i=0;i<N;i++) {
m = (m*s+t)%rmax;
a[i]=m%N;
}
time = clock()-time;
printf("%lu ", 1000*time/CLOCKS_PE
}
printf("/n");
exit((int)a[N-1]);
}
```

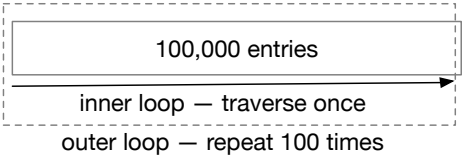
```
for (j=0;j<M;j++)
for(i=0;i<N;i++) {
m = (m*s+t)%rmax;
a[i]=m%N;
}
```

```
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++) {
m = (m*s+t)%rmax;
a[r]=m%N;
```



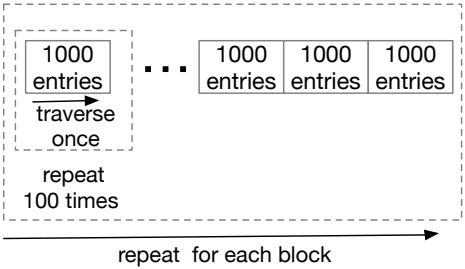


```
for (j=0;j<M;j++)
for(i=0;i<N;i++) {
m = (m*s+t)%rmax;
a[i]=m%N;
}
```



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++) {
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++) {
 m = (m*s+t)%rmax;
 l = m%N;
 a[l]=i;
}
time = clock()-time;
printf("%lu ",
   1000*time/CLOCKS_PER_SEC);
}
printf("/n");
exit((int)a[N-1]);
}
```

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

```
long int n = N/8;
  // number of cache lines
...
for(i=0;i<n;i++) {
 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++) {
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;
  // number of cache lines
...
for(i=0;i<n;i++) {
 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 ±77

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

execution time = 2358 ±62

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

execution time = 2531 ±77

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

execution time = 2358 ±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

example

$$[a[0] \dots a[M-1]] \times \begin{bmatrix} B[0][0] & \dots & B[0][N-1] \\ \dots & \dots & \dots \\ B[M-1][0] & \dots & B[M-1][N-1] \end{bmatrix} = [c[0] \dots c[N-1]]$$

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 10000
int a[N], B[N][N], c[N];
int main() {
 int i,j,k;
 time_t time;
 for(k=0;k<9;k++) {
  time = clock();
  for (j=0;j<N;j++)
   for (i=0;i<N;i++)
    c[j] += a[i]*B[i][j];
  time = clock()-time;
  printf("%lu ",
     1000*time/CLOCKS_PER_SEC);
 }
 printf("/n");
}
```

In C/C++ matrices are stored in row-major order:
`B[0][0],B[0][1],...,B[0][N-1],B[1][0],B[1][1],...`
`B[1][N-1],B[2][0],...,B[N-1][N-1]`

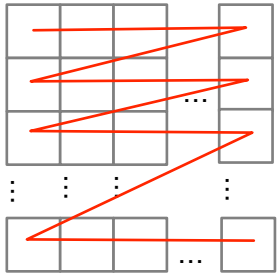
$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    c[j] += a[i]*B[i][j];
```

```
for (j=0;i<N;i++)
  for (i=0;j<N;j++)
    c[j] += a[i]*B[i][j];
```

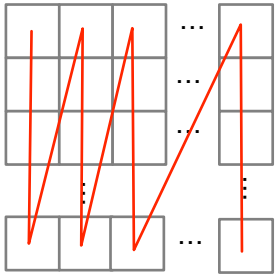
```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    c[j] += a[i]*B[i][j];
```

Matrix traversed in row-major order



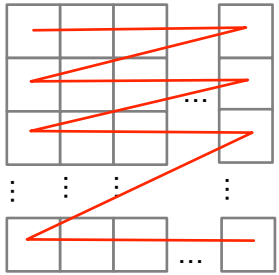
```
for (j=0;i<N;i++)
  for (i=0;j<N;j++)
    c[j] += a[i]*B[i][j];
```

matrix traversed in column-major order



```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    c[j] += a[i]*B[i][j];
```

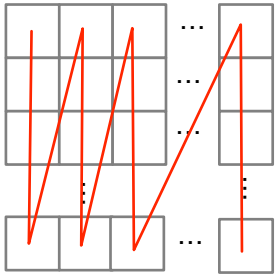
Matrix traversed in row-major order



Running time 34.9 ± 2

```
for (j=0;i<N;i++)
  for (i=0;j<N;j++)
    c[j] += a[i]*B[i][j];
```

matrix traversed in column-major order



Running time 821.4 ± 10.9

If matrix is traversed in column-major order, and matrix is tall, we get bad spatial locality.
Need to traverse in storage order.
I "cheated" with my measurements: The actual numbers where:
1157 834 811 806 835 841 797 812 835
257 42 35 36 34 33 33 33 33
I ignored first measurement.

If matrix is traversed in column-major order, and matrix is tall, we get bad spatial locality.
Need to traverse in storage order.
I "cheated" with my measurements: The actual numbers where:
1157 834 811 806 835 841 797 812 835
257 42 35 36 34 33 33 33 33
I ignored first measurement.
More time may be spent first time data is touched.

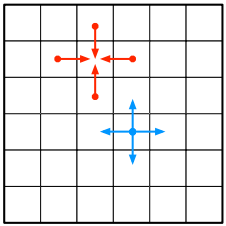
Prefetching

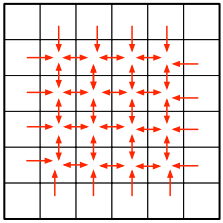
```
...
for (i=0;i<M;i++)
  for (j=0;j<N;i++)
    c[j] += a[i]*B[i][j];
...
```

- Accesses to B and c are to consecutive locations in memory
- The hardware will notice that after a few accesses and decide that the pattern will continue
- It will *prefetch* cache lines it guesses will be accessed in the near future
- Thus, "hiding" memory latency (loads will be serviced from cache)
- This does not change the results of the computation, but will often improve performance
- But wrong guesses will increase load on memory bus and pollute the cache

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;
}
```

Code

Another way to play ping-pong (C style)

```
double a[N][N], b[N][N], *p, *q *r;
p= &a[0][0]; q=&b[0][0]
...
while (!converged){
 for (i=1;i<N-1;i++)
  for(j=i*N+1;j<(i+1)*N-1;j++)
   *(p+j)=0.25*(*(q+j-1) + *(q+j+1)
          *(q+j-N)+*(q+j+N));
  r=p; p=q; q=r;
 }
```

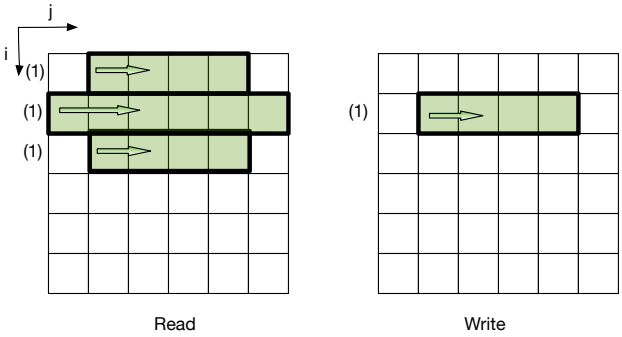
Array can be traversed in any order (all iterations are independent); what is a good order?

Traversal order for current code

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

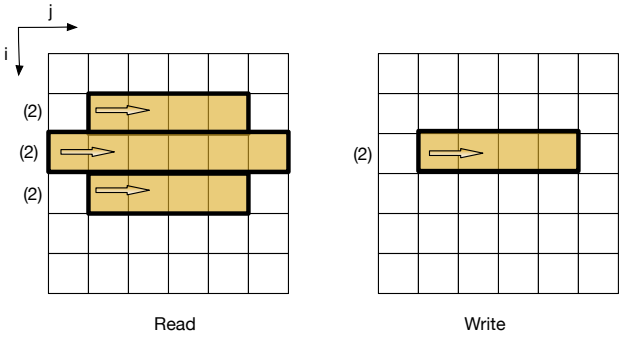
Traversal order for current code

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



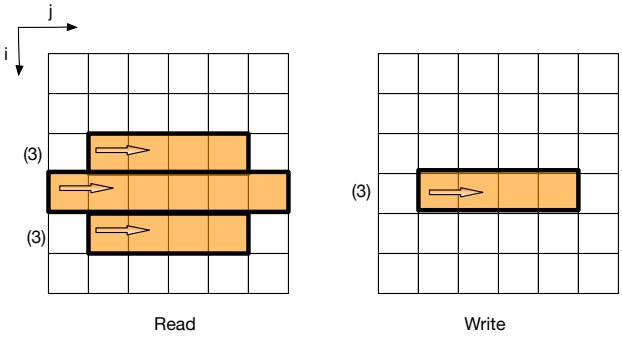
Traversal order for current code

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



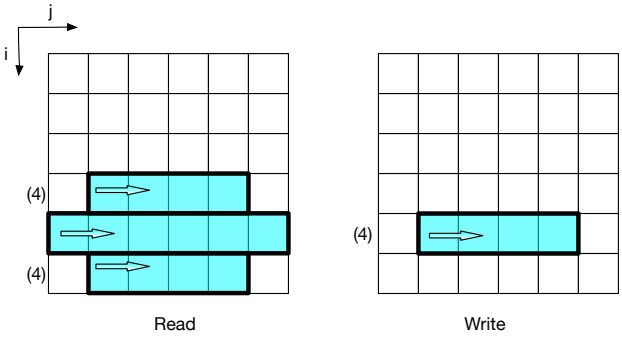
Traversal order for current code

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

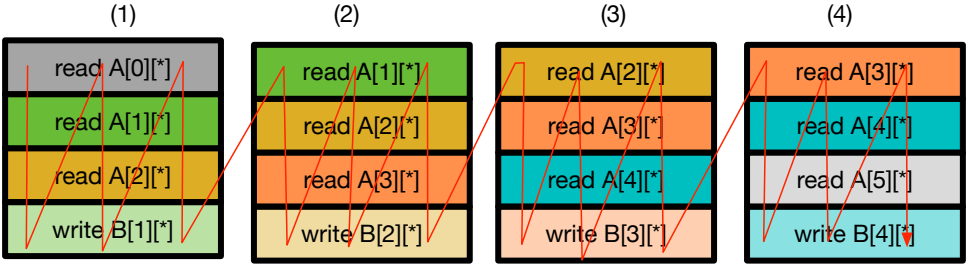


Traversal order for current code

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



How many cache misses?



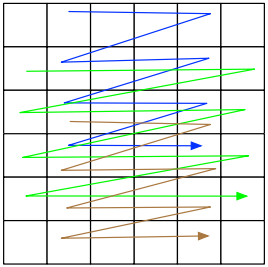
- Each row has $\sim N$ words
- If cache size $\gg 4N$ words ($32N$ bytes) then each word is read once.
  - Number of misses is $\sim 2N^2/8 = N^2/4$.
- If cache size $< 4N$ words each row in `A` is read 3 times (except row at top and bottom of matrix)
  - Number of misses is $\sim 4N^2/8 = N^2/2$

Jacobi

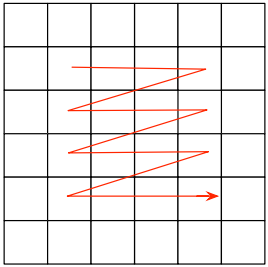
```
double a[2][M+2][N+2];
...

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 $\sim N^2/2$

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



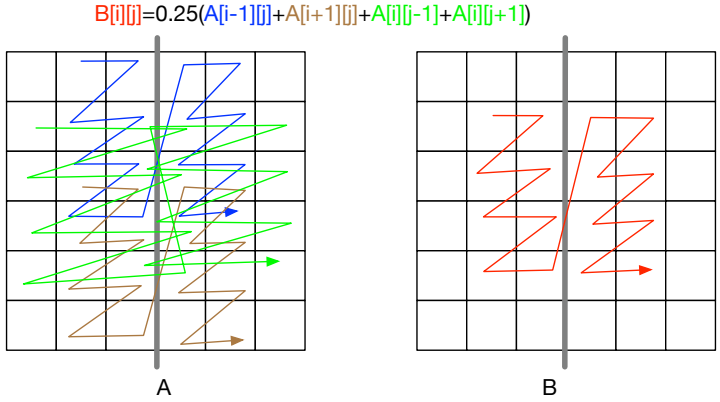
A



B

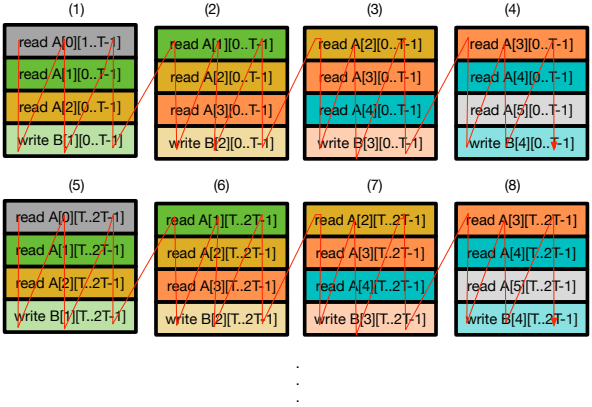
Tiling

- 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-l][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 $\sim N^2/4 + N^2/(4T)$ (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