Lecture 11

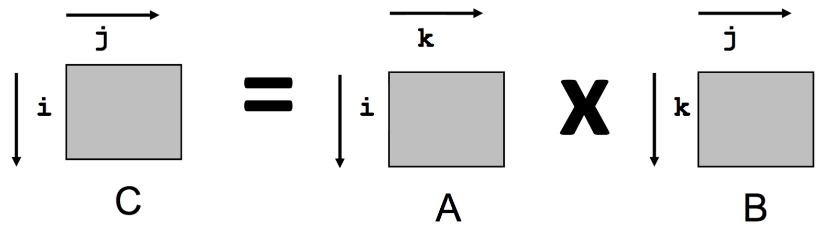
<2016-05-04 Wed>

Contents

1 Matrix Multiplication

- multiply n × n matrices
- elements are double
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination

– but may be able to hold in register



1.0.1 Basic

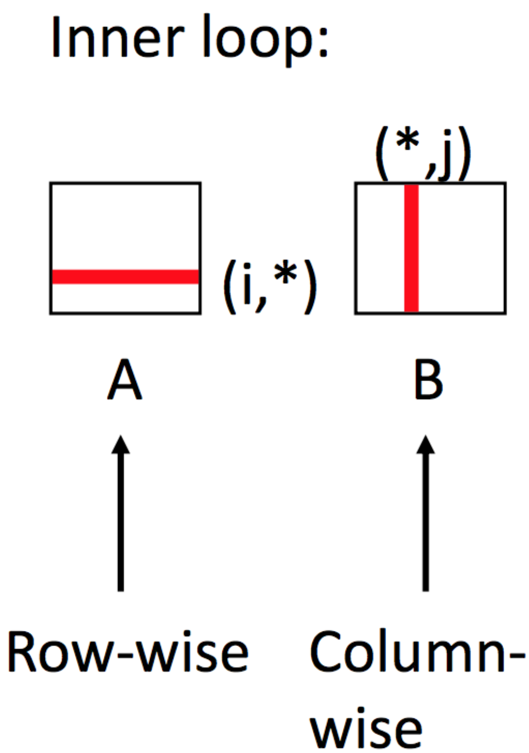
- assumptions
  - block size 32 bytes (4 doubles)
  - matrix dimension (N) very large
    - approximate 1 / N as 0.0
  - cache is not big enough to hold multiple rows
  - n × n matrix
- review
  - assume block size (B) > sizeof ($a_{i,j}$)
  - C arrays allocated in row-major order
    - each row in contiguous memory locations

	step through columns in 1 row	step through rows in 1 column
access	access *successive* elements	access *distant* elements
locality	spatial locality	no locality
miss rate	sizeof($a_{i,j}$) / B	1 (100%)

1.0.2 ijk / jik

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++) {
      sum += a[i][k] * b[k][j];
    }
```

```
    c[i][j] = sum;
  }
}
```

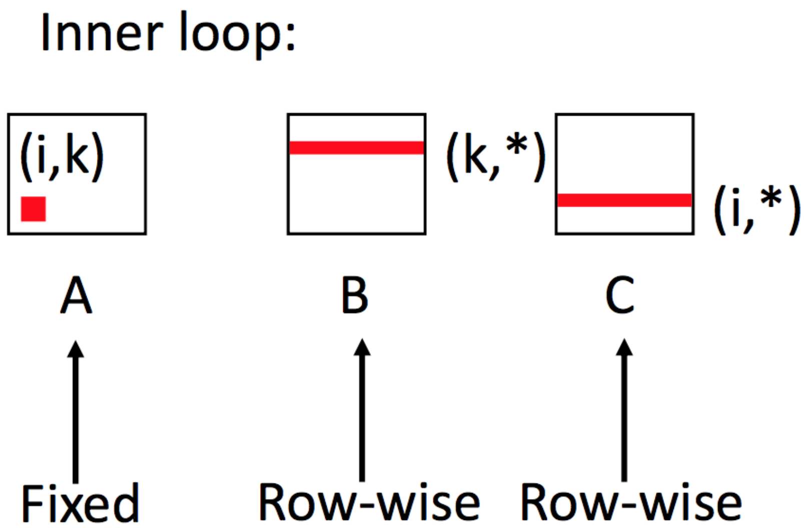


	miss per inner loop iteration	explanation
A	0.25	sizeof(double) / block size = 8 / 32
B	1.0	step by column, stride too big
C	0.0	local variable, temporal locality

1.0.3 kij / ikj

```
for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++) {
    r = a[i][k];
    for (j = 0; j < n; j++) {
      c[i][j] += r * b[k][j];
    }
  }
```

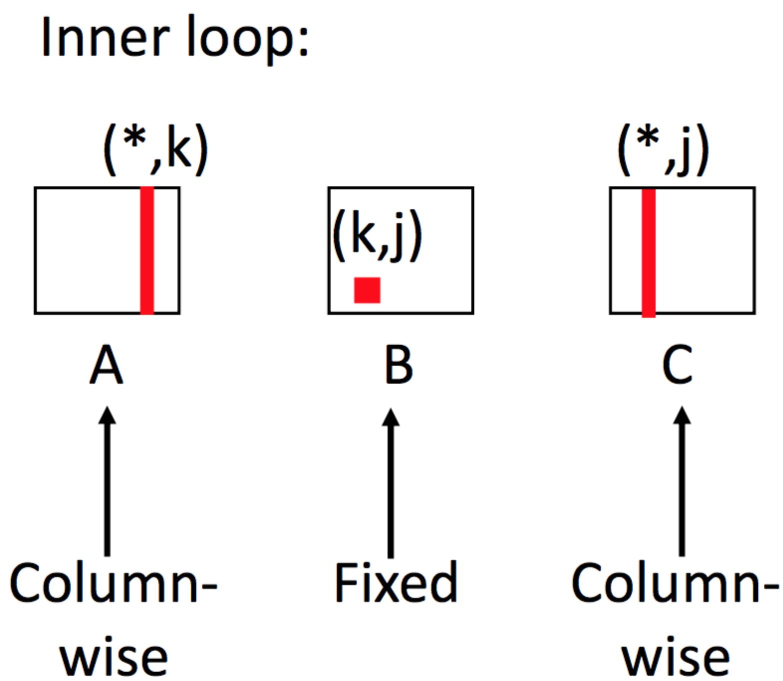
```
}
```



	miss per inner loop iteration	explanation
A	0.0	local variable, temporal locality
B	0.25	sizeof(double) / block size = 8 / 32
C	0.25	sizeof(double) / block size = 8 / 32

1.0.4 jki / kji

```
for (j = 0; j < n; j++) {
  for (k = 0; k < n; k++) {
    r = b[k][j];
    for (i = 0; i < n; i++) {
      c[i][j] += a[i][k] * r;
    }
  }
}
```



	miss per inner loop iteration	explanation
A	1.0	step by column, stride too big
B	0.0	local variable, temporal locality
C	1.0	stride too big, always miss

1.0.5 Summary

	ijk / jik	kij / ikj	jki / kji
loads & stores	2 loads, 0 strores	2 loads, 0 stores	2 loads, 1 stores
misses per iteration	1.25	0.5	2.0

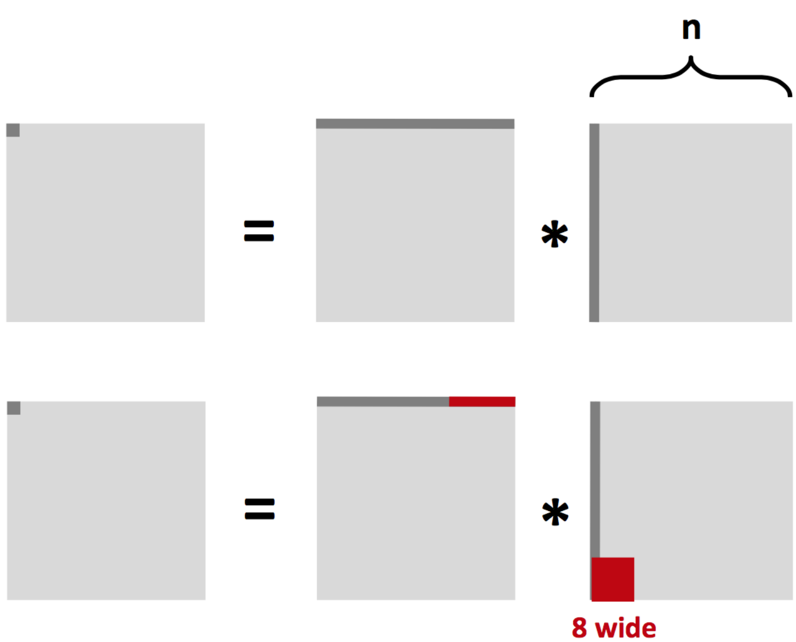
1.1 Matrix Multiplication Without Block

```
void mmm(double *a, double *b, double *c, int 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*n + j] += a[i*n + k] * b[k*n + j];
}
```

1.1.1 Cache Miss Analysis

- assume
  - matrix elements are double
  - cache block = 8 doubles (64)
  - cache size C much smaller than n
- misses each iteration
  - n / 8 + n = 9n/8
- total miss
  - $9n/8 \times n^2 = (9/8) \times n^3$

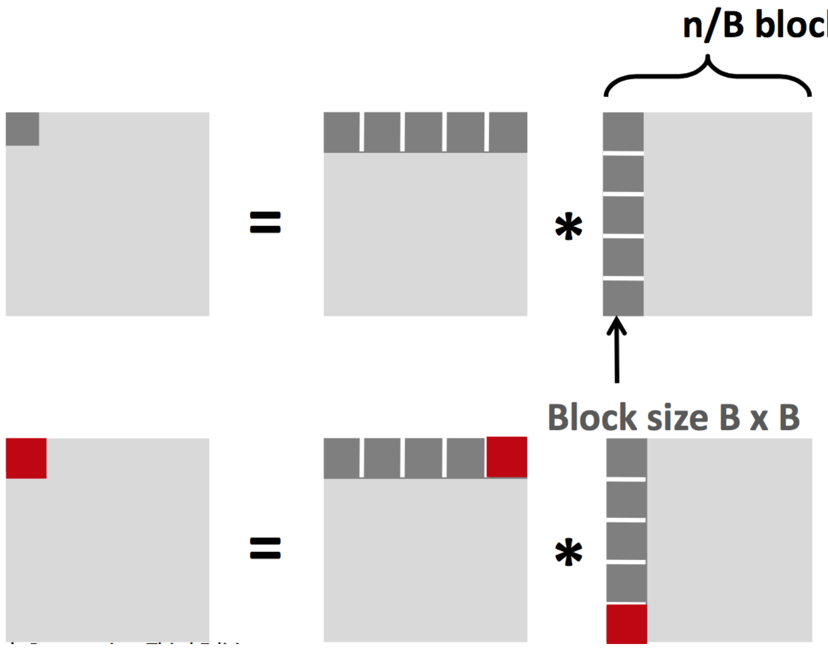


1.2 Blocked Matrix Multiplication

```
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k, i1, j1, k1;
```

```
  for (i = 0; i < n; i += B)
    for (j = 0; j < n; j += B)
      for (k = 0; k < n ; k += B)
          /* B by B mini matrix multiplications */
        for (i1 = i; i1 < i+B; i1++)
          for (j1 = j; j1 < j+B; j1++)
            for (k1 = k; k1 < k+B; k1++)
              c[i1*n + j1] += a[i1*n + k1] * b[k1*n + j1];
}
```

- assume
  - cache block = 8 doubles
  - cache size C much smaller than n
  - $(n/B) \times (n/B)$ as a mini matrix, $B \times B$ mini matrix multiplications
  - 3 blocks can fit into cache: $3B^2 < C$
- misses per iteration
  - $B^2/8$ misses per block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix c)
- total misses
  - $nB/4 \times (n/B)^2 = n^3/(4B)$



1.2.1 Summary

- no blocking: $(9/8) \times n^3$
- blocking: $1/(4B) \times n^3$
- B has limit $3B^2 < C$
- reason for dramatic difference
  - matrix multiplication has inherent temporal locality
    - input data: $3n^2$, computation $2n^3$
    - every element used $O(n)$ times
  - but program has to be written properly

1.3 Summary

- the speed gap between CPU, memory and mass storage continues to widen
- well written programs exhibit a property called **locality**

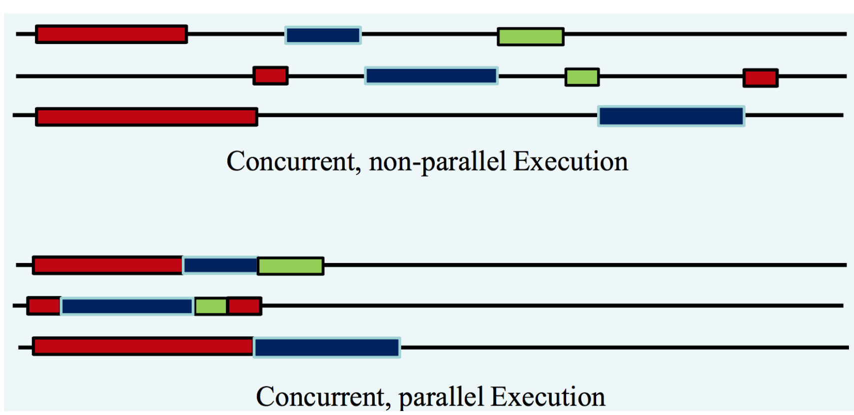
- memory hierarchies based on caching close the gap by exploiting locality
  - cache memories can have significant performance impact
- you can write your programs to exploit this
  - focus on the inner loops, where bulk of computations and memory accesses occur
  - try to maximize *spatial locality* by reading data objects with sequentially with stride 1
  - try to maximize *temporal locality* by using data objects as often as possible once it's read from memory

2 OpenMP

- OpenMP is one of the most common parallel programming models in use today

2.1 Concurrency vs. Parallelism

- concurrency
  - a condition of a system in which multiple tasks are *logically* active at one time
- parallelism
  - a condition of a system in which multiple tasks are *actually* active at one time



- concurrent application
  - an application for which computations *logically* execute simultaneously due to the semantics of the application
- parallel application
  - an application for which the computations *actually* execute simultaneously in order to complete a problem in less time

2.2 OpenMP

- an API for writing multithreaded applications
  - a set of compiler directives and library routines for parallel application programmers
  - greatly simplifies writing multi-threaded programs in Fortran, C and C++
- compiler directives `#pragma omp contruct [clause [clause] ... ]`
- function prototypes and types in the file `omp.h`

```
#pragma omp parallel num_threads(4)
#include <omp.h>
```

- most OpenMP constructs apply to a structured block
  - structured block
    - a block of one or more statements with one point of entry at the top and one point of exit at the bottom
    - it's OK to have an `exit()` within the structured block
- compiler flag
  - `gcc -fopenmp foo.c`
  - `export OMP_NUM_THREADS=4` (for bash shell)

2.2.1 Example

```
#include <omp.h>

int main() {

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d) ", ID);
        printf("world(%d)\n", ID);
    }
}
```

2.2.2 Shared Memory

- shared memory computer
  - any computer composed of multiple processing elements that share an address space
    - symmetric multiprocessor (SMP)
      - a shared address space with "equal-time" access for each processor, and the OS treats every processor the same way
    - non uniform address space multiprocessor (NUMA)
      - different memory regions have different access costs ... think of memory segmented into "near" and "far" memory
- shared memory program
  - an instance of a program
    - one process and lots of threads
    - threads interact through reads/writes to a shared address space
    - OS scheduler decides when to run which threads
      - interleaved for fairness
    - synchronization to assure every legal order results in correct results