Chapter 6: Caching III

Topics

- The Memory Mountain
- Blocking Application Data to Improve Temporal Locality

Announcements

- Performance Lab on due Monday Nov 17
  - Sign up for time slots soon
- Recitation next week
  - TAs will introduce next shell lab
- Try to return Midterm #2 next week
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 6 (all sub-sections)

Recap…

1. Arrays are laid out in row major order
2. Caches store data in blocks

Stride-1 Access Pattern

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

- Miss on a[i][0], but cache a block of a[i][0],a[i][1],a[i][2],a[i][3]…
- Miss rate = 25%
- Exploits spatial locality

Stride-N Access Pattern

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

- Miss on a[i][0], cache a block of a[i][0],a[i][1],a[i][2],a[i][3]…
- Miss on a[i+1][0], …
- Miss rate = 100%
- No spatial locality

Recap…

Matrix multiplication example: C = A x B, 3 loops

```
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];
```

- ijk/jik had one stride-1 and one-stride-N on interior loops
- kij/ikj had stride-1 access on both interior loop arrays – lowest miss rate!
- jki/kji had stride-N access on both interior loop arrays

Recap…

Caches organized into rows and columns for fast access

S rows or sets, and E columns, B bytes/element for fast access

Each element is called a "line" and contains a block of B bytes, a tag, and a valid bit

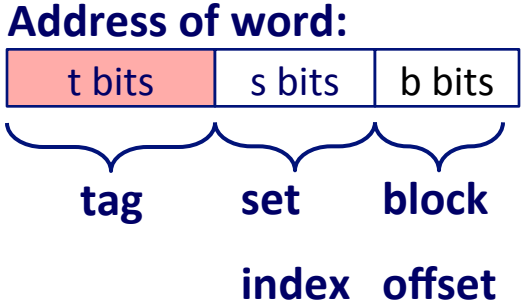
To find if data/instruction is in cache, use its address bits:

middle bits find row/set, top bits match to a line's tag, lowest bits index B bytes into a line

Direct-mapped cache: one column, E = 1

N-way Set Associative Cache: E = N

Fully Associative Cache: one row, S=1



What about writes?

Multiple copies of data exist:

- L1, L2, L3, Main Memory, Disk

What to do on a write-hit?

- Write-through (write immediately to memory)
- Write-back (defer write to memory until replacement of line)
  - Need a dirty bit (line different from memory or not)

What to do on a write-miss?

- Write-allocate (load into cache, update line in cache)
  - Good if more writes to the location follow
- No-write-allocate (writes immediately to memory)

Typical

- Write-through + No-write-allocate
- Write-back + Write-allocate

The Memory Mountain

Read throughput (read bandwidth)

- Number of bytes read from memory per second (MB/s)

Memory mountain

- Measured read throughput as a function of spatial and temporal locality.
- Compact way to characterize memory system performance.

Memory Mountain Test Function

```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                    /* warm up the cache first! */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

Memory Mountain Main Routine

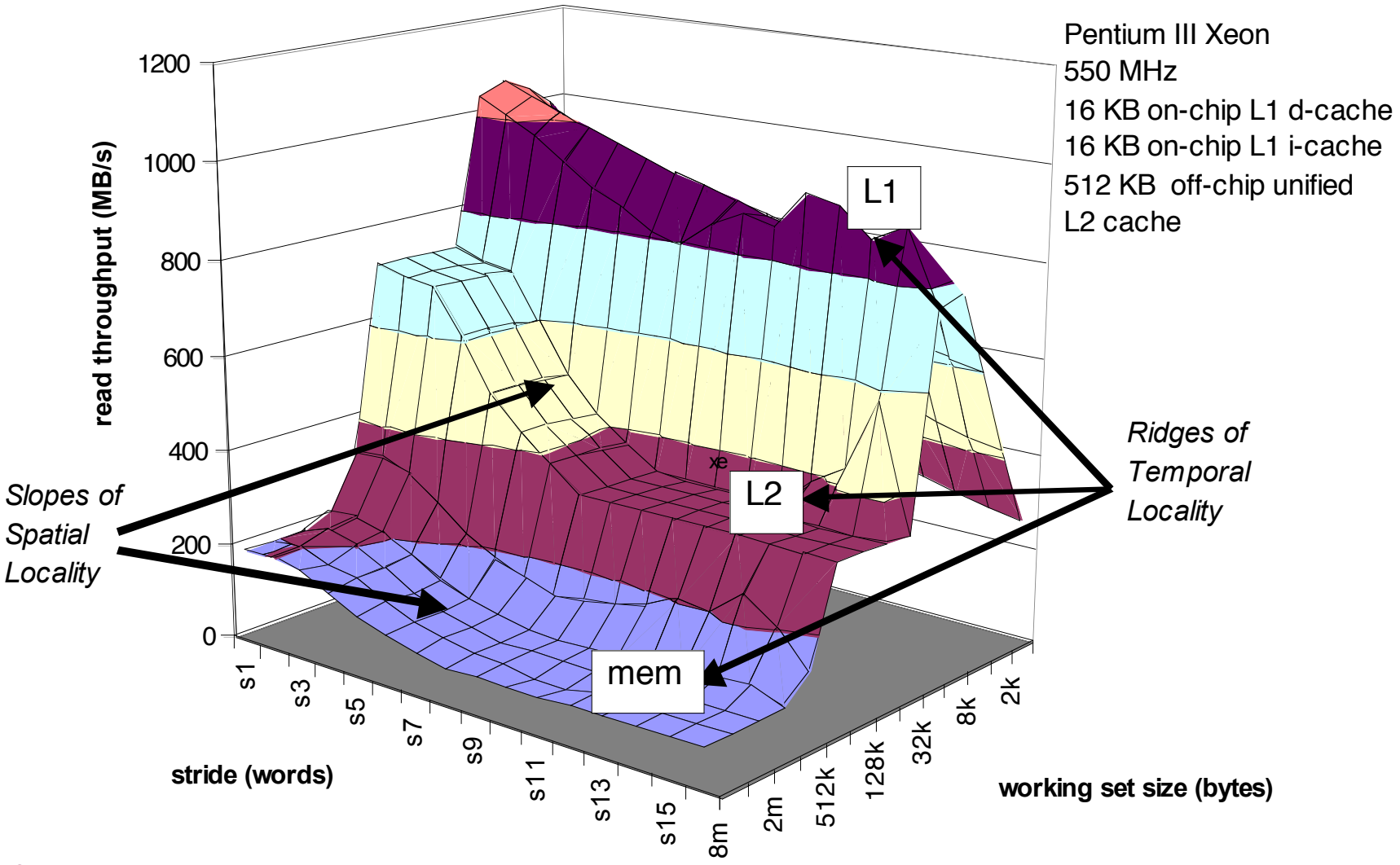
```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;         /* Working set size (in bytes) */
    int stride;       /* Stride (in array elements) */
    double Mhz;       /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
```

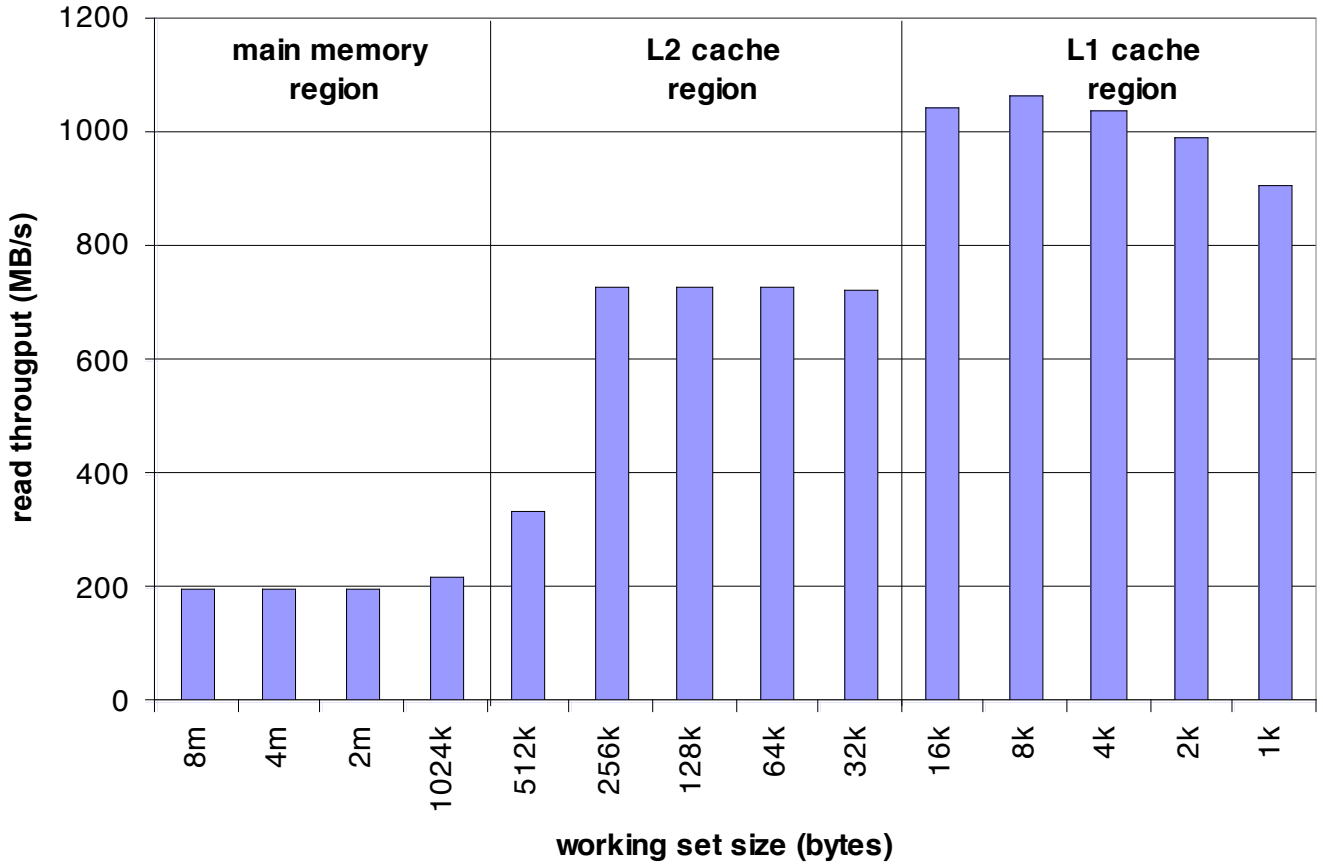
The Memory Mountain



Ridges of Temporal Locality

Slice through the memory mountain with stride=1

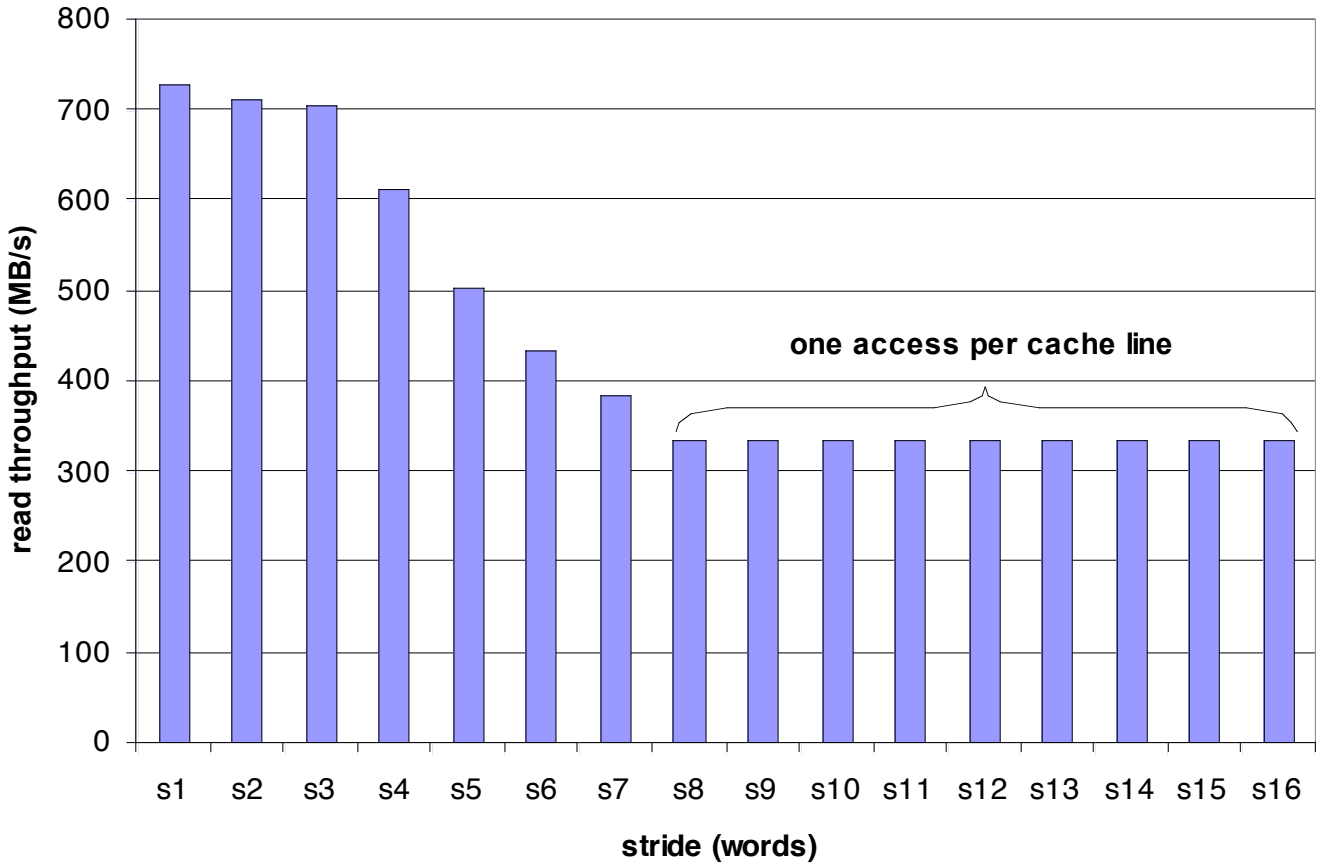
- illuminates read throughputs of different caches and memory



A Slope of Spatial Locality

Slice through memory mountain with size=256KB

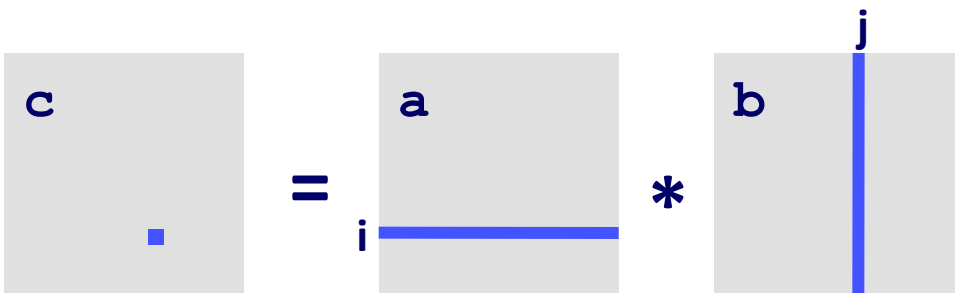
- shows cache block size.



Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
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][j] += a[i][k] * b[k][j];
}
```

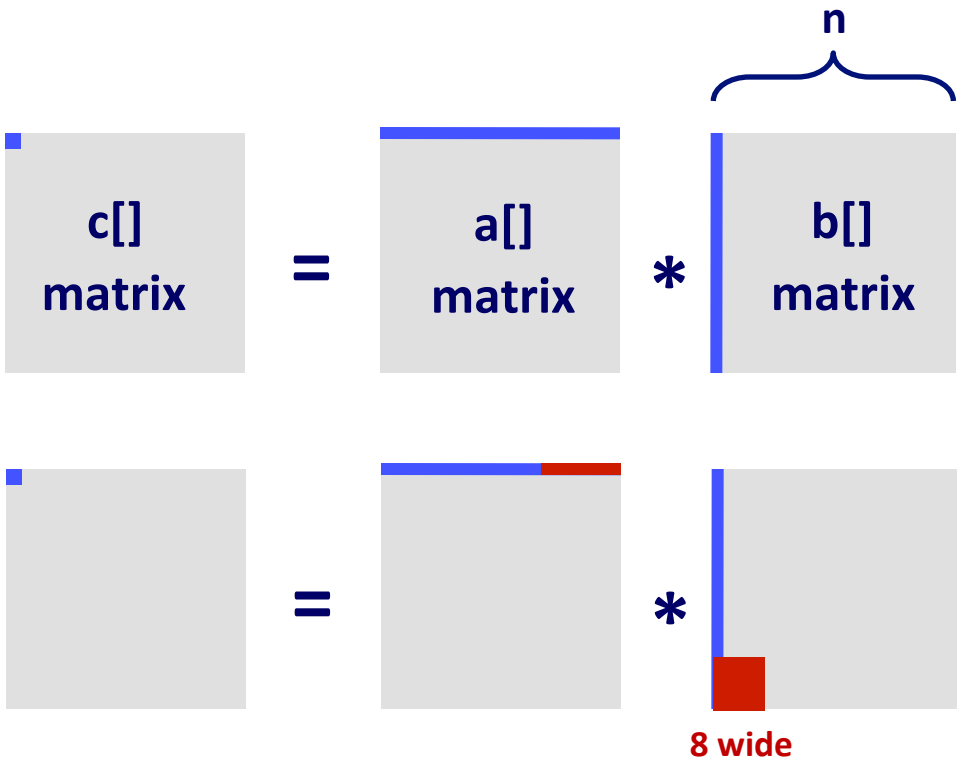


- By rearranging loop indices, exploit spatial locality
  - e.g. Stride-1
  - inner loop does not reuse the *same* data well, which is needed to exploit temporal locality
    - Once done with b's column j, don't reuse it, move on to column j+1
- Multiply in a way that exploits temporal locality, i.e. reuses the same data
  - 2-Dim Blocking

Total Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - n/8 misses (from a[])
    + n misses (from b[],
    which is stride-N)
    = 9n/8 misses
  - Afterwards in cache: (schematic)



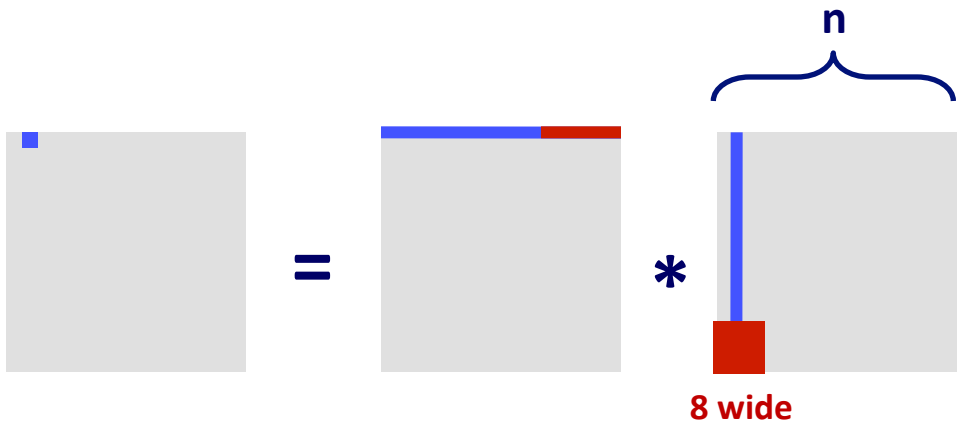
Total Cache Miss Analysis

Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

Second iteration:

- Again:
  n/8 + n = 9n/8 misses



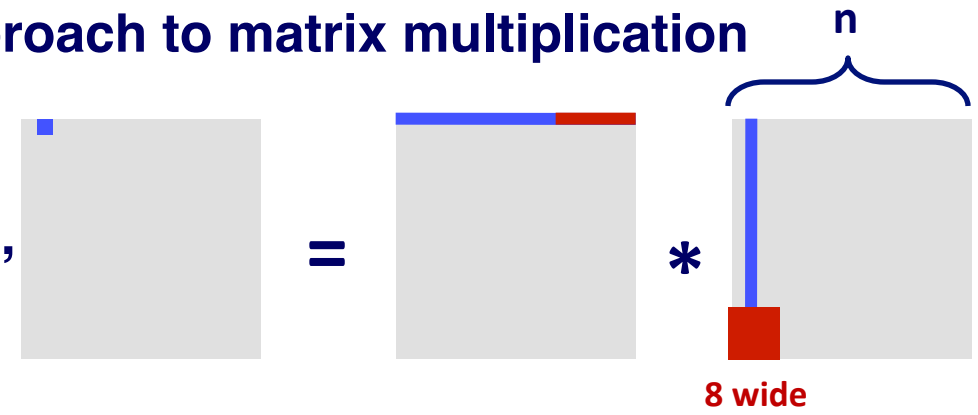
Total misses:

- $9n/8 * n^2 = (9/8) * n^3$ -- can we do better?

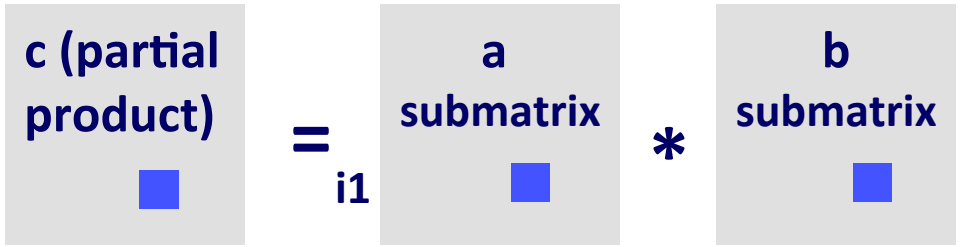
Improving Temporal Locality by Blocking

Organize your application data into chunks that fit in L1 cache, and are read/written together, then load the next related chunks, etc. – exploit temporal locality

- Example: Row*Column approach to matrix multiplication has poor temporal locality
- Working set size of row (and column) >> cache size, so keep getting misses



- Instead, divide matrix into small chunks/submatrices, cache them, multiply them block by block, then discard



  - Rows & columns of submatrices reused – temporal locality

Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- "block" (in this context) does not mean "cache block".
- Instead, it mean a sub-block within the application's data matrix.
- Example: N = 8; sub-block size = 4

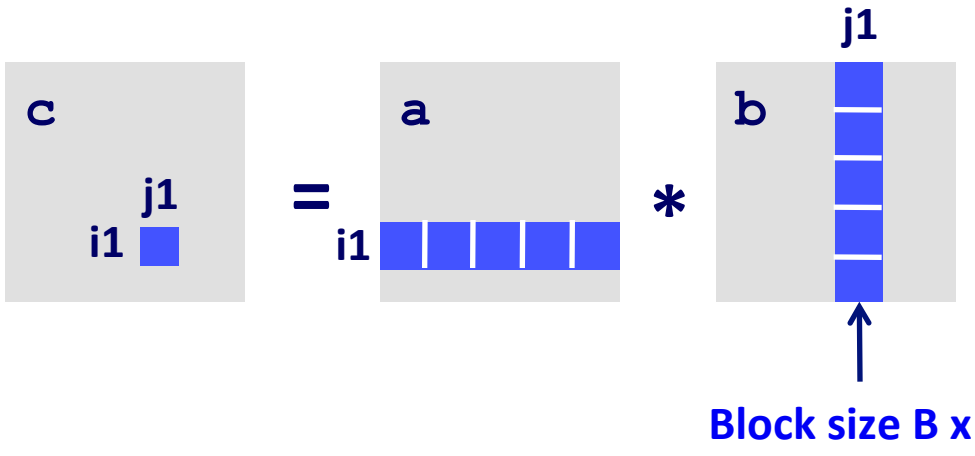
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., $\mathbf{A_{xy}}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Blocked Matrix Multiplication



Repeat for all i1,j1

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)        // i'th block row
        for (j = 0; j < n; j+=B)      // j'th block column
            for (k = 0; k < n; k+=B)    // k'th block of "vector" multiply
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

Total Cache Miss Analysis

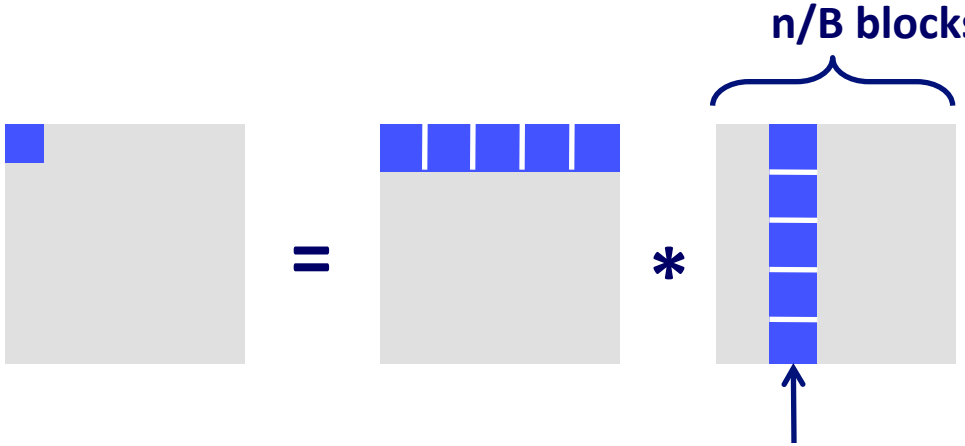
Assume:

- Cache block = 8 doubles
- Cache size C << n (much smaller than n)
- Three blocks ■ fit into cache: $3B^2 < C$
  - Three submatrices of a, b and c

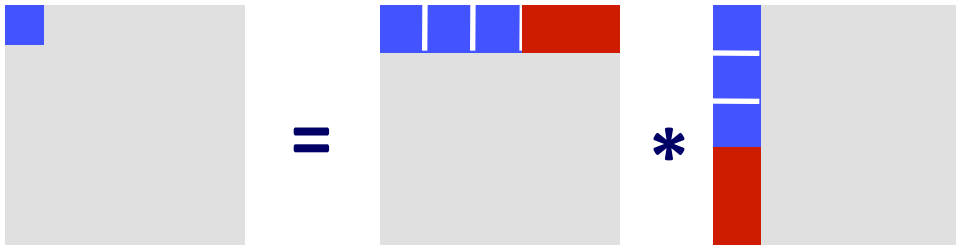
First (block) iteration:

- $B^2/8$ misses for each block =

  B/8 misses/row * B rows
- 2n/B * $B^2/8$ = nB/4 (omitting matrix c)



- Afterwards in cache (schematic)



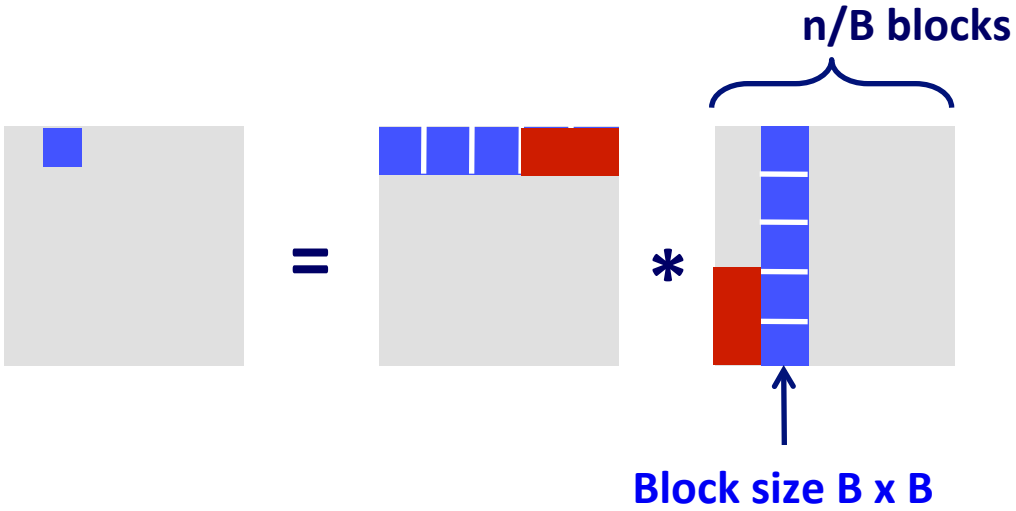
Total Cache Miss Analysis

Assume:

- Cache block = 8 doubles
- Cache size C << n (much smaller than n)
- Three blocks ■ fit into cache: $3B^2 < C$

Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$

Summary

No blocking: (9/8) * n^3

Blocking: 1/(4B) * n^3

Suggest largest possible block size B, but limit $3B^2 < C$!

(can possibly be relaxed a bit, but there is a limit for B)

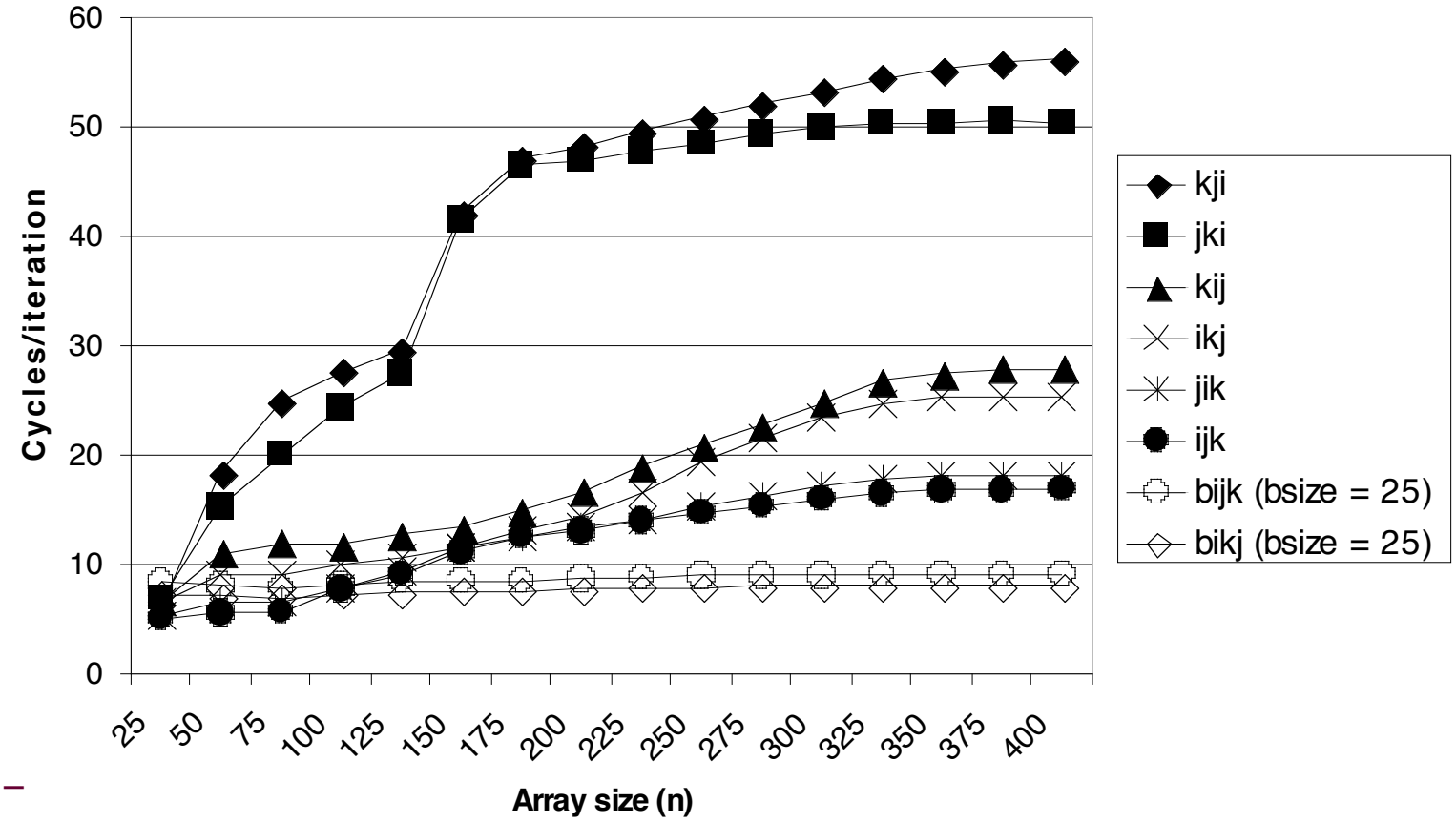
Reason for dramatic difference:

- **Matrix multiplication has inherent temporal locality:**
  - **Input data: $3n^2$, computation $2n^3$**
  - **Every array elements used O(n) times!**
- **But program has to be written properly**

Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- **relatively insensitive to array size.**



Concluding Observations

Programmer can optimize for cache performance

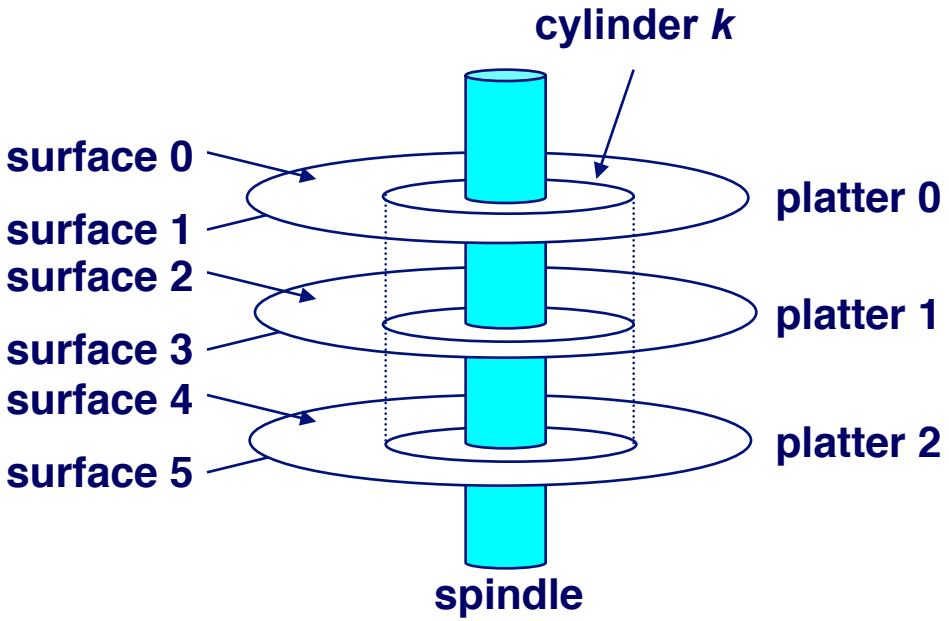
- How data structures are organized
- How data are accessed
  - Nested loop structure
  - Blocking is a general technique

All systems favor “cache friendly code”

- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small, e.g. use blocking to exploit temporal locality!
  - Use small strides to exploit spatial locality!

Magnetic Disk Geometry

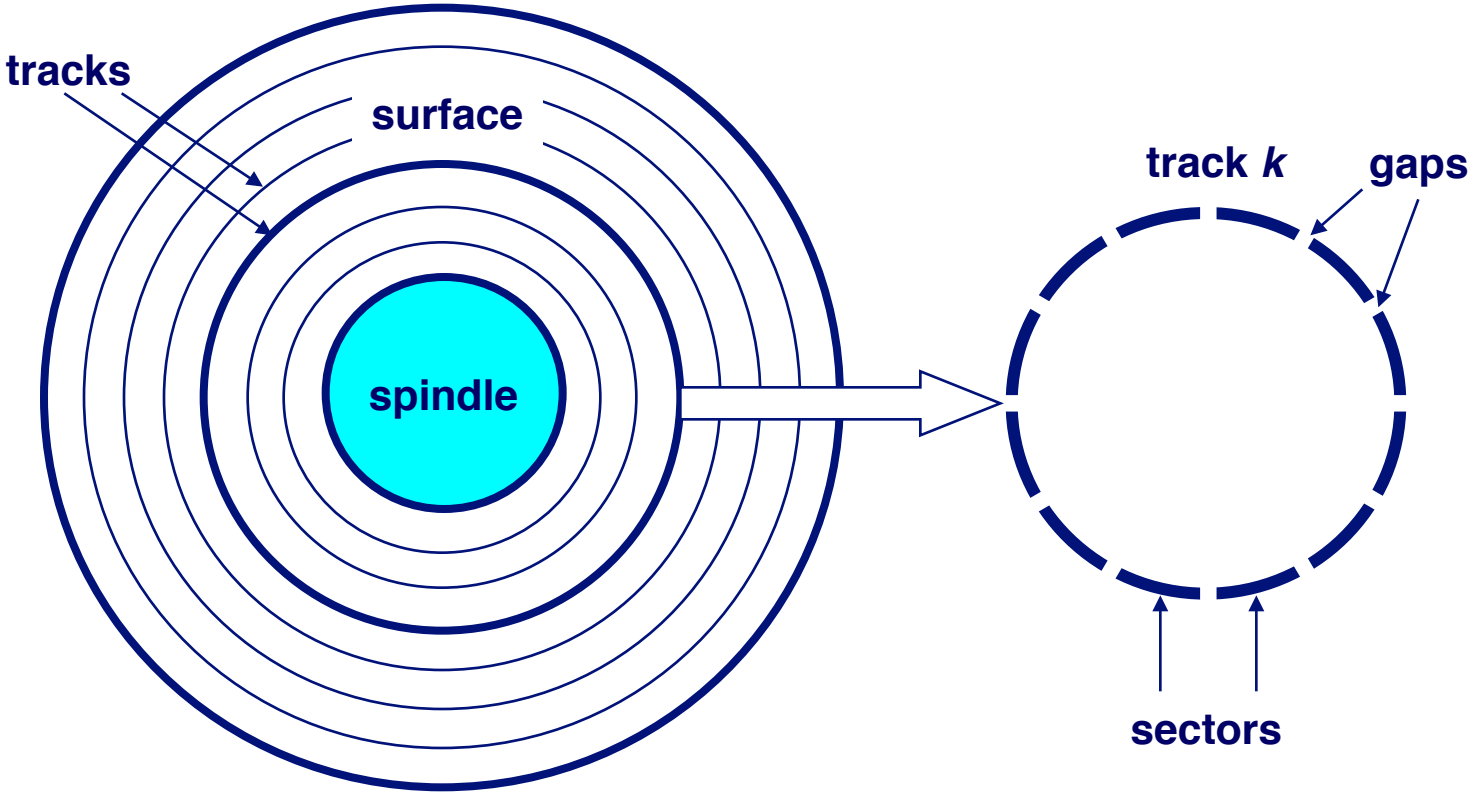
Disks consist of platters, each with two surfaces.



Magnetic Disk Geometry

Each surface consists of concentric rings called tracks.

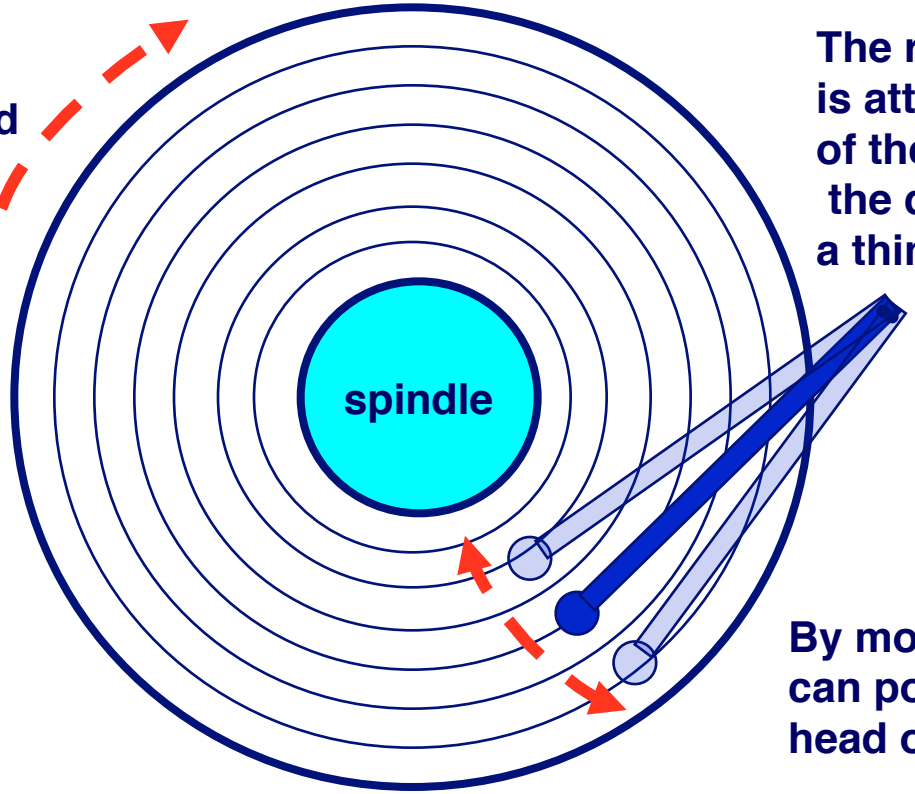
Each track consists of sectors separated by gaps.



Total disk capacity = # platters* # surfaces/platter * # tracks/surface * # sectors/track * # bits/sector

Disk Operation (Single-Platter View)

The disk surface spins at a fixed rotational rate



The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Access Time

Average time to access some target sector approximated by :

- **Taccess = Tavg seek + Tavg rotation + Tavg transfer**

Seek time (Tavg seek)

- **Time to position heads over cylinder containing target sector.**
- **Typical Tavg seek = 9 ms**

Rotational latency (Tavg rotation)

- **Time waiting for first bit of target sector to pass under r/w head.**
- **Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min (~ ms as well)**

Transfer time (Tavg transfer)

- **Time to read the bits in the target sector.**
- **Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min. (typically small)**

Hence magnetic disks are slow (~ ms)!

Solid State Disks (SSDs)

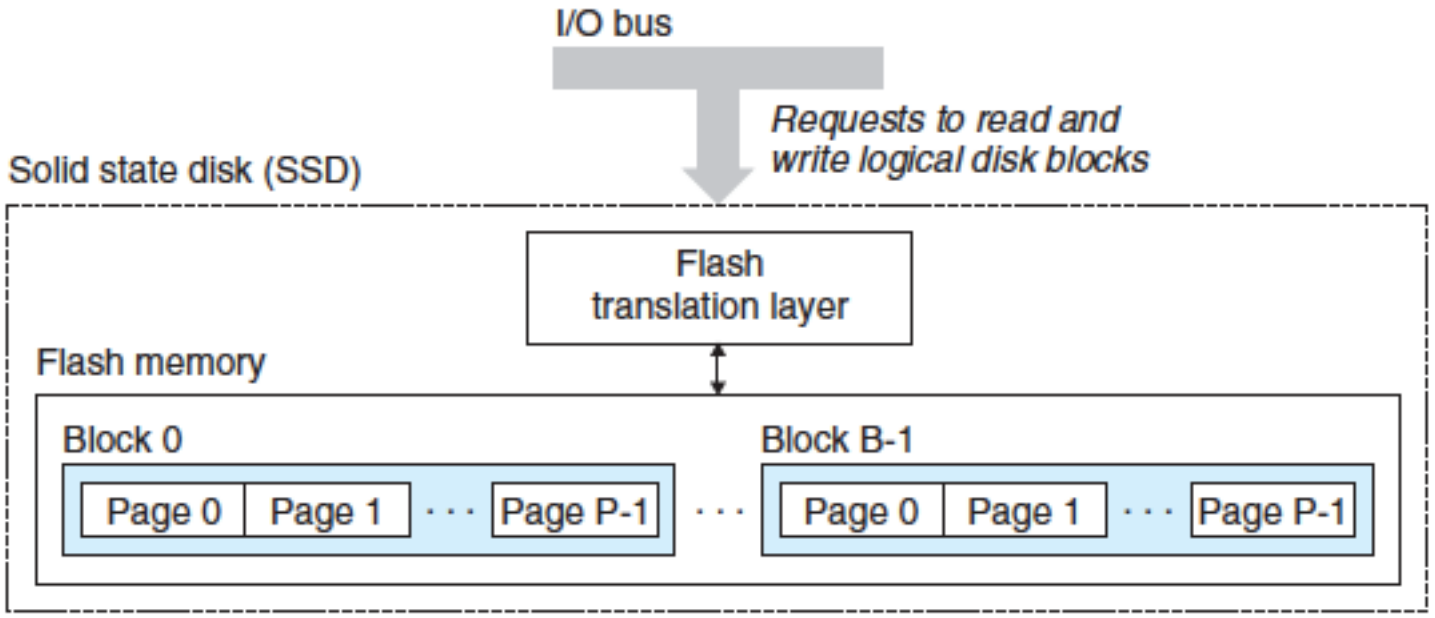


Figure 6.15 **Solid state disk (SSD).**

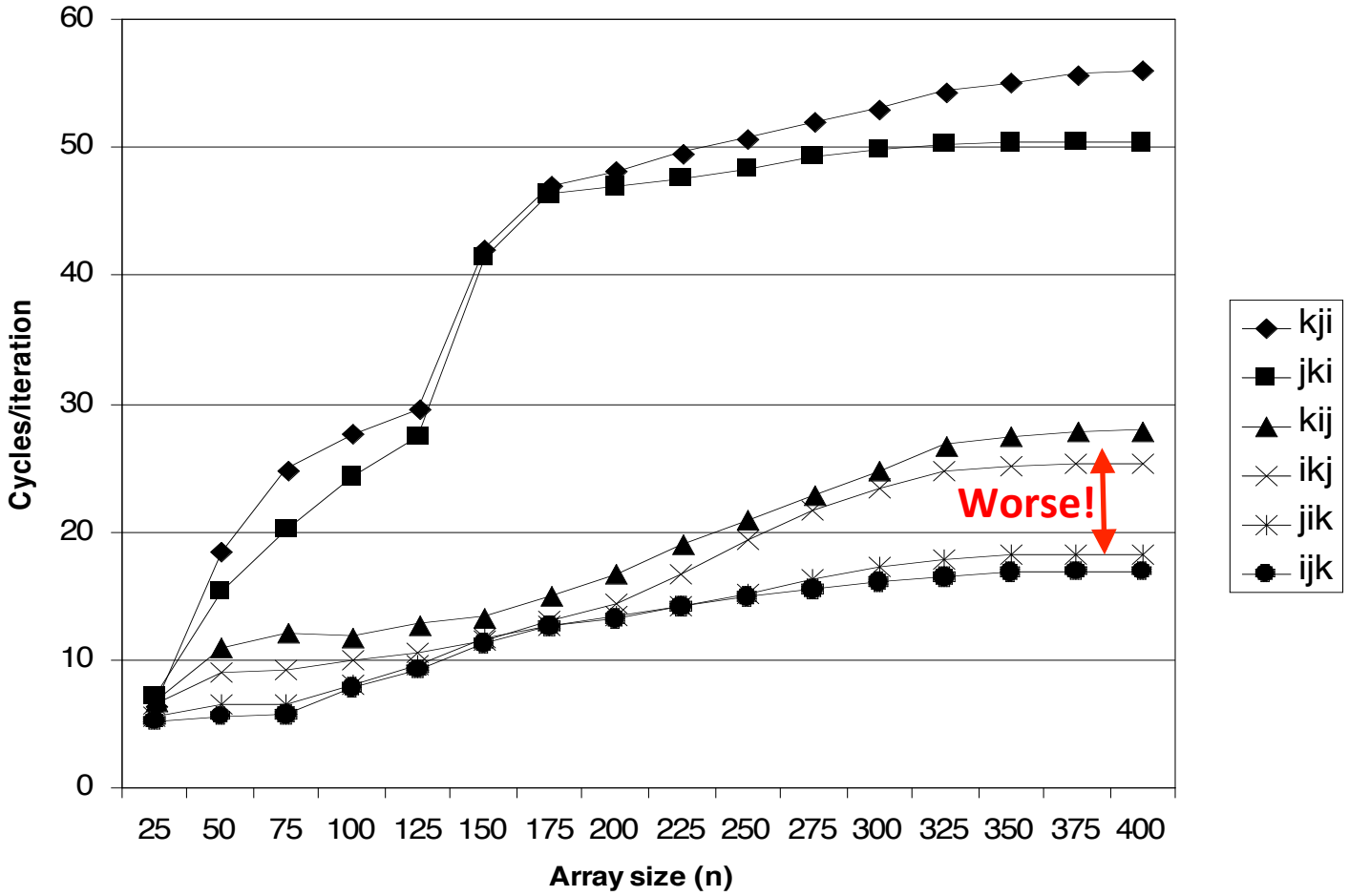
- **Based on flash memory – all electronic, no mechanical parts**
- **Divided into B blocks, with P pages/block**
- **Reads are straightforward (~30 µs), but writes are more complicated (~300 µs)**
- **Flash blocks can wear out => do wear leveling**

Supplementary Slides

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- **Code scheduling matters, too.**



Blocked Matrix Multiply (bijk)

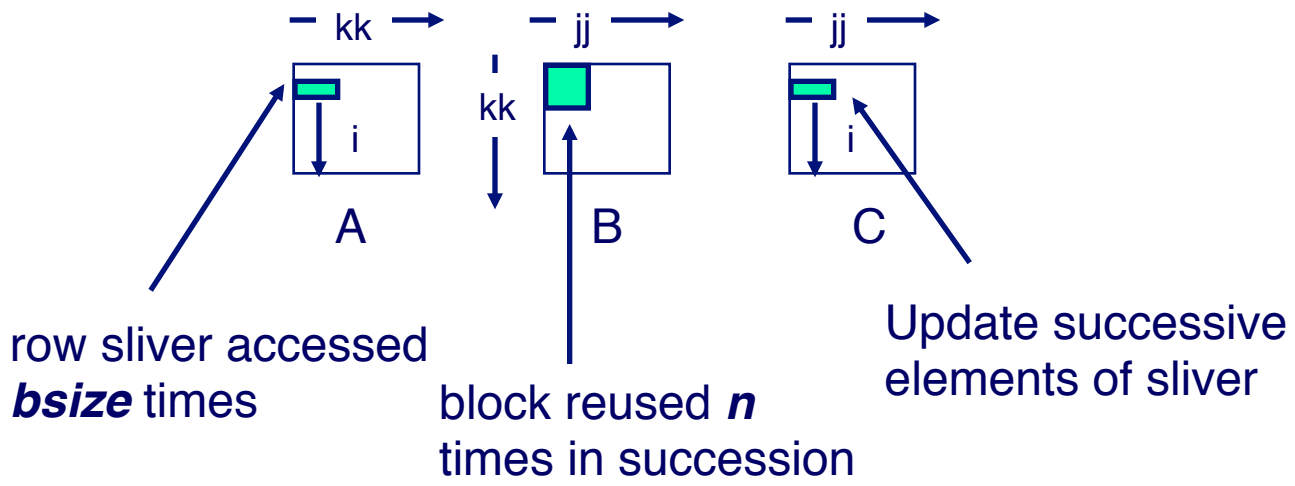
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
      }
    }
  }
}
```

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

```
for (i=0; i<n; i++) {
  for (j=jj; j < min(jj+bsize,n); j++) {
    sum = 0.0
    for (k=kk; k < min(kk+bsize,n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
```

Innermost Loop Pair



Optimizations for the Memory Hierarchy

- Write code that has locality
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time
- How to achieve?
  - Proper choice of algorithm
  - Loop transformations
- Cache versus register level optimization:
  - In both cases locality desirable
  - Register space much smaller + requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

Recap…

Writing through vs Write back caching

Cache-friendly code

- sumarraycols miss rate = 100%!
- sumarrayrows miss rate = 25%

Memory mountain

- Ridges of Temporal locality
- Slopes of Spatial locality

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```