
Lecture 6

Software Optimizations for Memory Performance

Department of Electrical Engineering
Stanford University

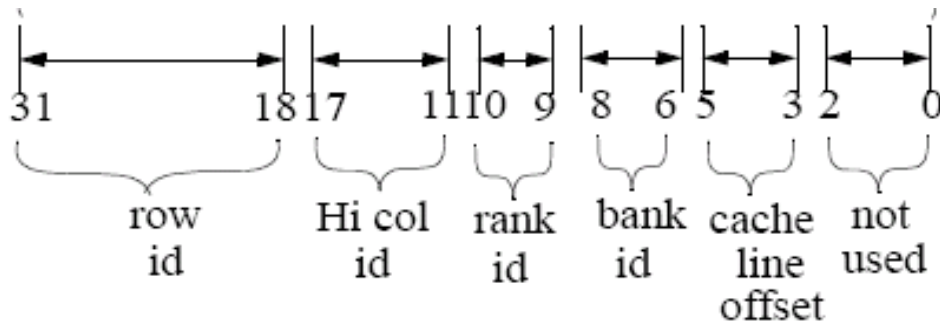
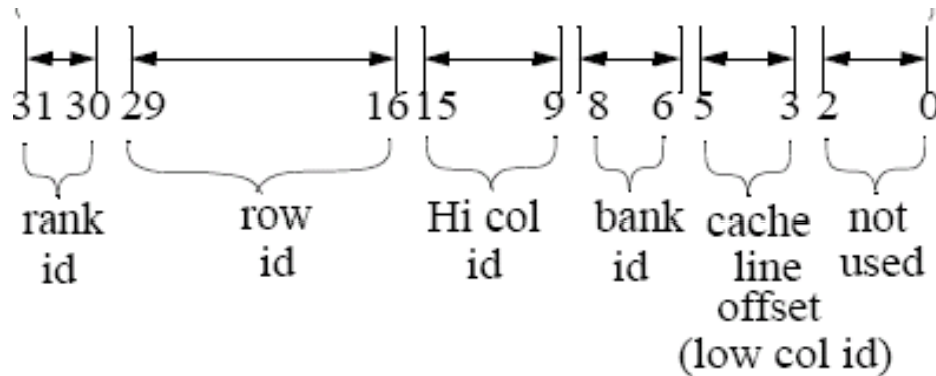
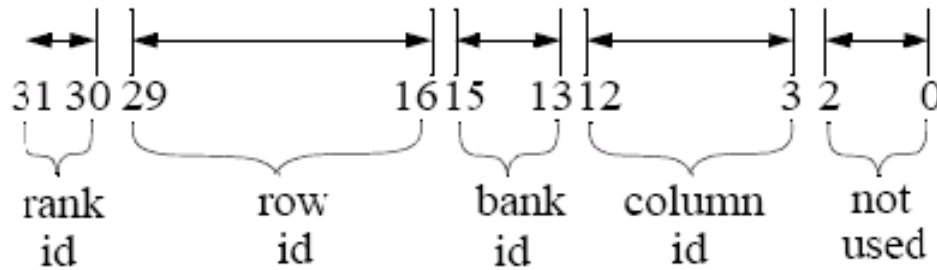
<http://eeclass.stanford.edu/ee282>

Announcements

Review: Concurrency in DRAM Systems

- Banks
 - Blocks within each chip that can be accessed independently
- Ranks
 - Groups of chips that can be accessed independently
 - Parallelism across DIMMs
- Channels
 - DIMMs connected on independent buses
- Banks, ranks, and channels allow access overlapping
 - Assuming accesses go to different resources
 - Assuming all timing constraints are satisfied

Review: Address Mapping Examples (aka Address Interleaving)

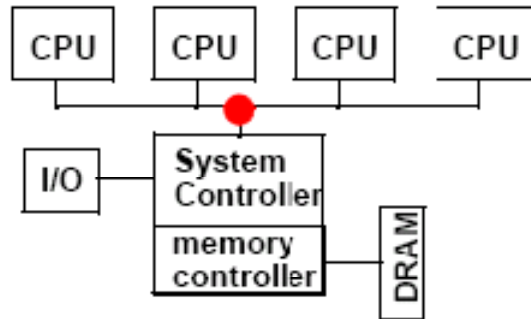


- What are the tradeoffs?
 - Think about sequential patterns initially...
- What is fast and what is slow in memory accesses?
- What about non-sequential accesses?
- Other issues?

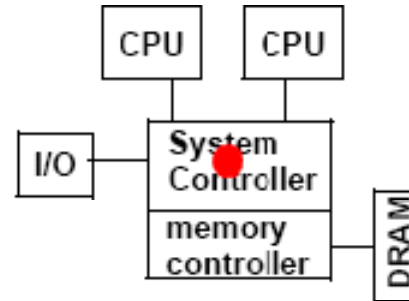
DRAM Controllers

- Their role
 - Generate proper controls for DRAM DIMMs for each access
 - Schedule across banks & potentially reorder DRAM accesses
 - Involves queuing & buffering
- Their location
 - In the chipset/memory controller/north bridge
 - In the processor chip
 - Reduces latency & improve BW between CPU & controller
- What makes them complicated
 - Variability of timings across different systems/DRAM chips
 - Ordering requirements
 - Trade-off between latency and bandwidth

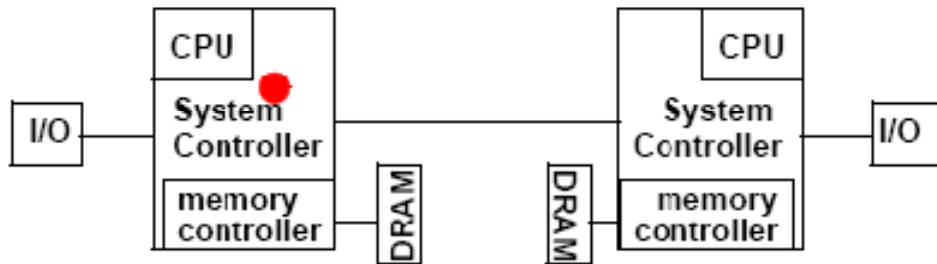
DRAM Controller Topologies



Classic small system topology
(Lots of systems)



Point-to-point processor-controller
system topology
(AMD Athlon/Alpha EV6/PPC 970)



Integrated system controller system topology
(AMD Opteron/Alpha EV7 etc.)

● represents point of synchronization*. (for local access)

- Tradeoffs?
- See optional paper for examples
 - Available on-line...

DRAM Controller Scheduling Policies

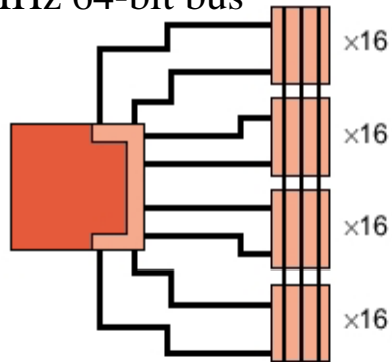
- Bank precharging: open or closed
 - Open: leave row open until new row request
 - Closed: precharge bitlines as soon as current burst satisfied
- Power mode
 - Active, stand-by, self-refresh, power-down
- Basic ordering:
 - In-order, load-over-store, bank-ready, age-threshold, ...
 - Remember that ordering matters across banks as well
 - All banks share same IO pins
- Advanced ordering:
 - Open row first, row with most pending, row with less pending, ...

DRAM Evolution: SDRAM & DDR

- SDRAM: 1st synchronous DRAM
 - 66 to 133MHz with multiplexed address bus
 - 4 banks
 - Programmable burst (1 to 8)
- DDR SDRAM: double data rate (both clock edges)
 - 100 to 266MHz with multiplexed address bus
 - 4 banks
 - Programmable burst (2 to 8)
- DDR2
 - 200 to 333MHz, 4 banks, 4-8 burst, ...
- Over time:
 - Clock ↑, minimum burst ↑, banks ↑, ...

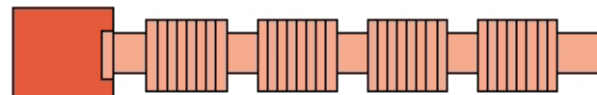
DDR Vs. Rambus

200MHz 64-bit bus



(a)

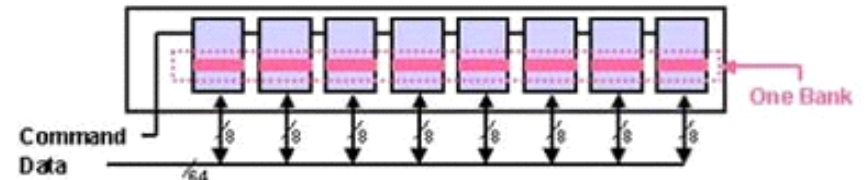
800MHz 16-bit bus



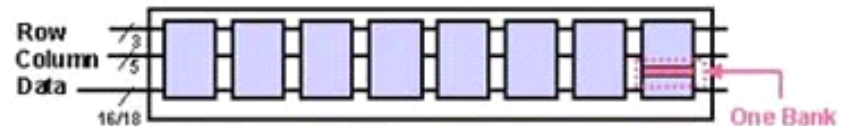
(b)

Figure 8. Bank counts: a 32-Mbyte, 64M SDRAM system with four large banks (a) versus a 32-Mbyte, 64M Direct RDRAM system with 32 small banks (b).

DIMM Modules



RIMM Modules



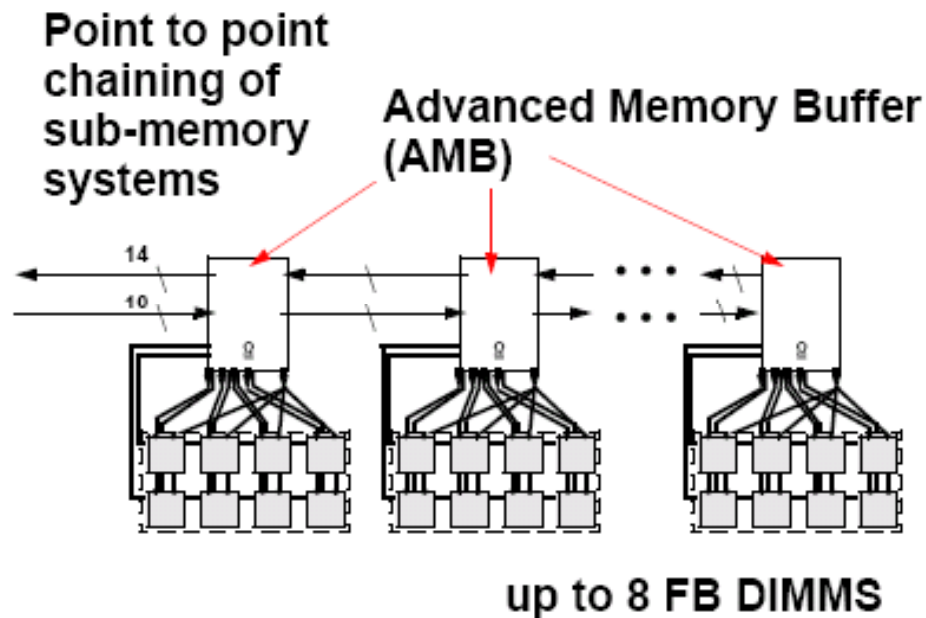
- Many banks/chip (4-32)
- Narrow fast interconnect (pipelined)
- High bandwidth
- Latency & area penalty

Other DRAM Options

- GDDRx: DRAM specialized for graphics
 - Unidirectional signaling, higher clock rate, lower tRC, ...
- RLDRAM/FCDRAM: reduced latency / fast cycle DRAM
 - Mostly targeted toward L3 caches & telecom gear
 - Wider bus, low tRC/tRAC, non multiplexed address bus, small bursts
- ESDRAM: 1T SRAM (SRAM replacement)
 - 16 banks, hidden refresh, 4-6 cycle latency, large bursts
- VCDRAM: virtual channel DRAM
 - Includes a small SRAM cache
- Mobile DRAM
 - Low cost and low power design, hidden refresh

Fully Buffered DIMM (FB-DIMM)

- The DDR problem
 - Higher capacity \Rightarrow more DIMMs \Rightarrow lower data-rate (multidrop bus)
- FBDIMM approach: use point-to-point links
 - While still using commodity DRAM chips
 - Network with 12-beat packages, separate up/downstream wires



Fully Buffered DIMM (FB-DIMM)

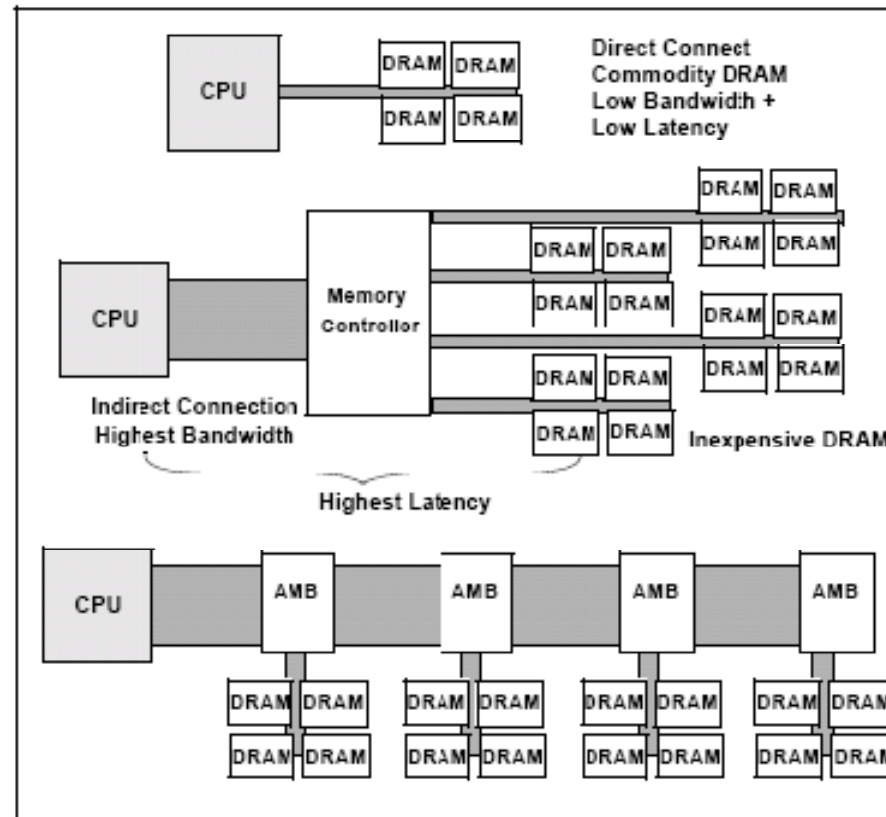
	DDR2	FBD
Datarate (Mbps)	667	4000
Pin Count (Data Bus)	108	-
Channel pin count (without pwr and gnd)	141	59
Channel pin count (with pwr and gnd)	~200	~70
Peak bandwidth (GB/s)	5.3	8.0
Theoretical Efficiency (bandwidth/pin) (Mbps)	213	914

72 data pins + 18 diff pairs of DQS strobes

~ 4.5x increase in pin-bandwidth (counting power and ground)

- Watch out for:
 - Asymmetric upstream/downstream
 - Requires deep channel for maximum bandwidth efficiency
 - Power overhead of current generation AMBs

System Level Choices for DRAM



How to Select a DRAM Architecture

Don't just make a decision based on specs!

- Bandwidth: measure for your own workload
 - Mix of reads/writes, bursts, locality, strides, ...
 - Different architectures/chips are optimized for different cases
- Latency: typically not critical but...
 - Don't forget other latency contributors (e.g. DRAM controller)
- Cost:
 - pins (board traces), signaling, cost/DRAM bit
- Power:
 - Voltage, power modes, ...
- Risk:
 - Number of suppliers

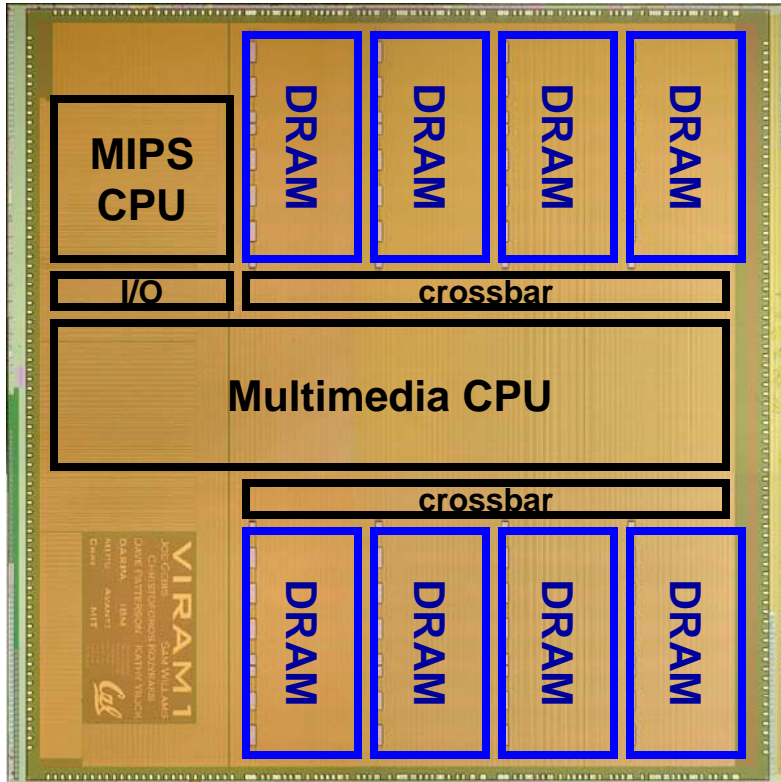
DRAM Trends to Keep in Mind

- DRAMs: capacity +60%, cost –30% per year
 - 2.5x cells/area, 1.5x die size in -3 years
- '98 DRAM fabrication line costs \$2B
 - DRAM only: density, leakage v. speed
- Rely on increasing number of computers & memory per computer (60% market)
 - DIMM is replaceable unit → computers use any generation DRAM
- Commodity, second source industry → high volume, low profit, conservative
 - Little organization innovation in 20 years
- Order of importance: 1) Cost/bit 2) Capacity
 - First Rambus: 10x BW, +30% cost → little impact

Embedded DRAM

- The inevitable: CPU & DRAM integration
 - Embedded DRAM, Merged-DRAM-logic, intelligent RAM, ...
 - Allows for high bandwidth
 - Multiple wide busses, switched interconnect
 - Allows for low latency
 - Current set of problems
 - Cost and capacity of single chip
- Alternatives
 - MCM packaging
 - 3D packaging

Embedded DRAM Example



- VIRAM media processor
 - 125M transistors
 - 200MHz, 2 Watt
- Embedded DRAM
 - 13 Mbytes
 - 8 banks
 - 6.4GB/sec per bank (peak)
- Processor
 - 4-lane vector processor
 - 6.4 Gop/sec
 - 64-bit MIPS core

Non-volatile Memory (Flash)

- Storage technology
 - Charge trapped in a floating gate
 - Retains information even without power supply
- Two design alternatives
 - NOR: used primarily for code
 - better E/W endurance (100K vs 10K), fast reads (100ns), slow writes (10usec)
 - NAND: used primarily for data
 - Smaller cell (~40%), reads and writes are 1usec
- Applications
 - MP3 players, cameras, ...
 - Hard disk replacement
 - Main memory replacement or assist?

Optimizing Programs for Cache Hierarchies

- Goal: eliminate cache misses
 - Enhance temporal locality (reuse)
 - Enhance spatial locality (sequential accesses)
 - Eliminate conflicts
- What can software control?
 - Memory layout (where & how an object is placed)
 - Access order (when an object is placed)
- Software means:
 - You, the programmer OR
 - An optimizing compiler (sometimes :)

Steps to Produce Efficient Code

1. Select a good algorithm
 - Pitfall: asymptotic analysis does not account for constants
 - Problem size may dictate best algorithm
 2. Use efficient libraries
 - Vendor libraries often use good algorithms and machine-specific optimizations
 - Other optimized libraries: BLAS, ATLAS, NAG, ...
 - Check netlib for pointers
 3. Select right compiler and compile options
 - Get the most out of optimizing compilers
 - Typically, use the compiler vendor uses for SPEC benchmarks
 - Useful flags: -arch=XYZ, -On, -fast, -unroll, ...
 4. Optimize your code manually
 - Typically optimize for locality
 - Compilers can optimize for your pipeline (most often)
- Repeat if needed

The General Approach

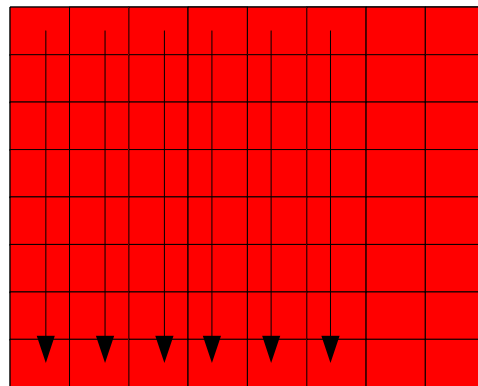
- Most cache optimizations done for loops & arrays (why?)
- How to go about optimizing locality for loops
 1. Consider the order in which elements are accessed
 2. Check when misses should occur for each array
 3. Change loop order/structure or data layout to reduce misses
- Watch out for the following
 - Is your new code correct?
 - Does your new code introduce new misses?

Loop Interchange

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

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

- Assume row-major layout (C)
 - Assume N is very large
 - Read squares are misses
- Spatial locality improved



Loop Fusion

```
for (i=0; i<N; i++)  
    a[i] = b[i] + 1;  
for (i=0; i<N; i++)  
    c[i] = 3*b[i];
```

- Assume N is very large
- No temporal locality across loops

```
for (i=0; i<N; i++) {  
    a[i] = b[i] + 1;  
    c[i] = 3*b[i];  
}
```

- Temporal locality improved for array b[]

Loop Reversal

- Initial code

```
for (i=0; i<N; i++){  
    a[i] = b[i] + 1;  
    c[i] = 3*a[i]  
}  
for (i=0; i<N; i++){  
    d[i] = c[i+1] + 1;
```

- Can we fuse these loops?
- Why?

- After reversal

```
for (i=N-1; i>=; i--){  
    a[i] = b[i] + 1;  
    c[i] = 3*a[i]  
}  
for (i=N-1; i>=; i--){  
    d[i] = c[i+1] + 1;
```

- Can we fuse these loops?
- Is the code correct?

Loop Fission

```
for (i=0; i<N; i++){  
    a[i] = b[i] + 1;  
    c[i] = 3*a[i];  
    f[i] = g[i] + h[i];  
}
```

- Assume N is very large
- Assume 4-way cache
- When do we get misses?

```
for (i=0; i<N; i++){  
    a[i] = b[i] + 1;  
    c[i] = 3*a[i];  
}  
for (i=0; i<N; i++){  
    f[i] = g[i] + h[i];  
}
```

- Any better now?

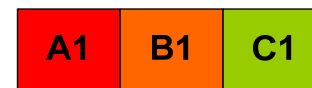
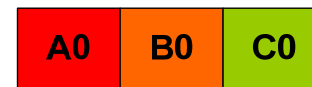
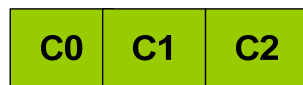
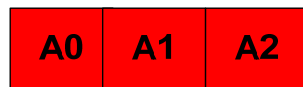
Array Merging

```
int a[N], b[N], c[N];  
for (i=0; i<N; i++)  
    a[i] = b[i] + c[i];
```

- Assume N is large
- Assume 2-way cache
- How many misses?

```
struct ALL {int a,b,c};  
struct ALL M[N]  
for (i=0; i<N; i++)  
    M[i].A = M[i].b + M[i].c;
```

- Any better now?



Array Padding

```
int a[N], b[N];  
for (i=0; i<N; i++)  
    sum += a[i]*b[i];
```

```
int A[N], pad[8], B[N];  
for (i=0; i<N; i++)  
    sum += a[i]*b[i];
```

- Assume
 - N is large
 - N multiple of cache size
 - Direct mapped cache
 - Arrays allocated sequentially
 - Cache line is 32 bytes
- How many misses?
- Any better now?
- What is the cost?
- Can also do intra-array padding
 - Pad rows in a 2D array
 - Can reduce conflicts for banks/associativity

Review: Optimizing SW for Memory Hierarchies

- Goal: eliminate cache misses
 - Enhance temporal locality (reuse) & spatial locality (sequential accesses)
 - Eliminate conflicts
- What can software control?
 - Memory layout (where & how an object is placed)
 - Access order (when an object is placed)
 - Examples: loop interchange, loop fusion & fission, array merging & padding, ...
- Steps for code optimization
 1. Select a good algorithm
 - Pitfalls: ignore the constants, ignore problem size
 2. Use efficient libraries
 - Pitfall: library may not address all the problem cases you care about
 3. Select right compiler and compile options
 - Pitfall: overspecialize or underspecialize your executable; caching vs other overheads
 4. Optimize your code manually for locality

Matrix Multiplication Example

- A simple problem dominated by cache performance
 - Total cache size
 - Exploit temporal locality and keep the working set small using blocking
 - Block size
 - Exploit spatial locality

- Description:

- Multiply $N \times N$ matrices
- $O(N^3)$ total operations
- Accesses

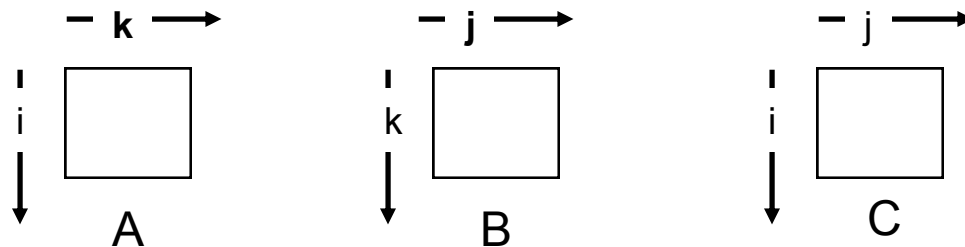
- N reads per source element
- N values summed per destination
 - but may be able to hold in register

```
/* ijk */
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;
  }
}
```

Variable sum held in register

Miss Rate Analysis for MxM

- Assume:
 - Line size = 32B (big enough for 4 64-bit words)
 - Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
 - Cache is not even big enough to hold multiple rows
- Analysis Method:
 - Look at access pattern of inner loop

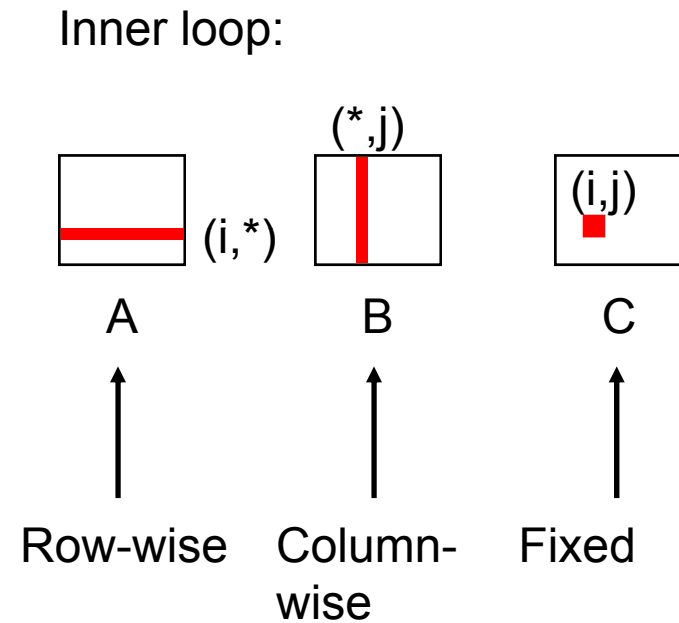


Array Layout (for C) and Miss Patterns

- C arrays allocated in row-major order
 - each row in contiguous memory locations
- Stepping through columns in one row:
 - `for (i = 0; i < N; i++)`
`sum += a[0][i];`
 - Accesses successive elements
 - if block size (B) > 4 bytes, exploit spatial locality
 - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
 - `for (i = 0; i < n; i++)`
`sum += a[i][0];`
 - accesses distant elements
 - no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
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;
  }
}
```

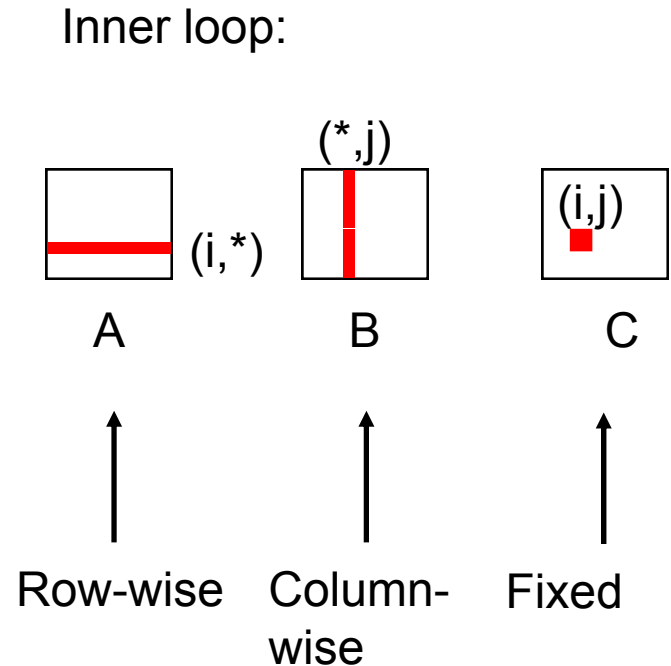


- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (jik)

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

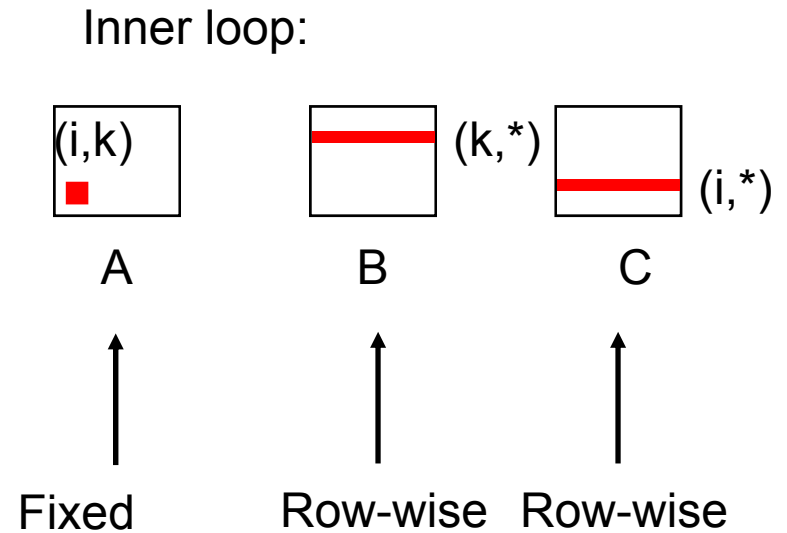


- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
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];
  }
}
```



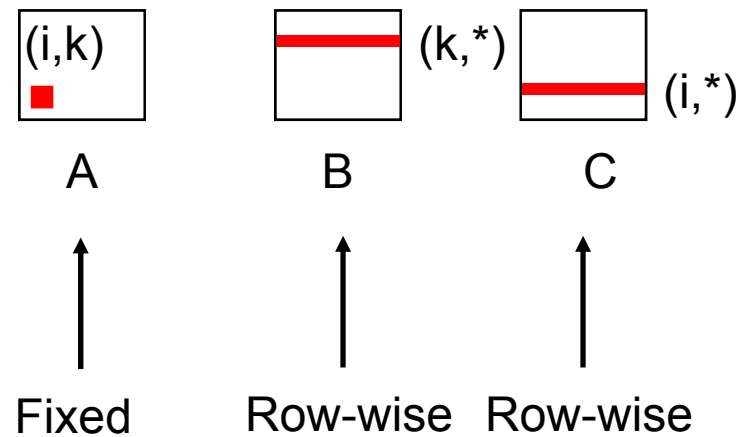
- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Matrix Multiplication (ikj)

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

Inner loop:



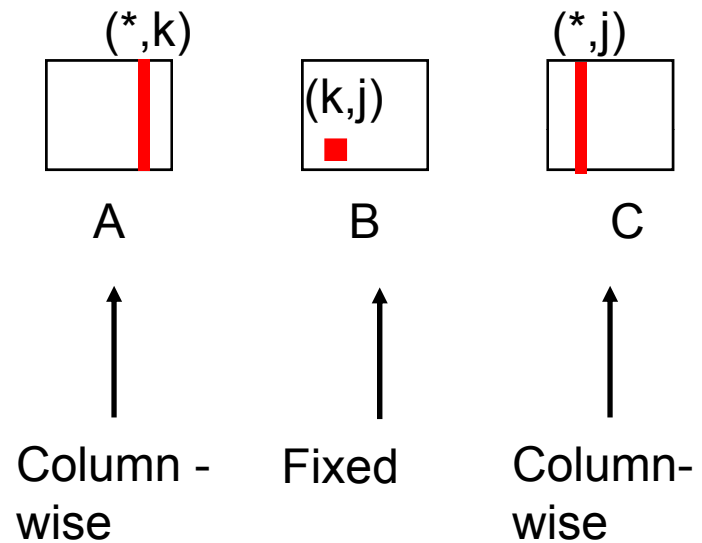
- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */  
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;  
  }  
}
```

Inner loop:

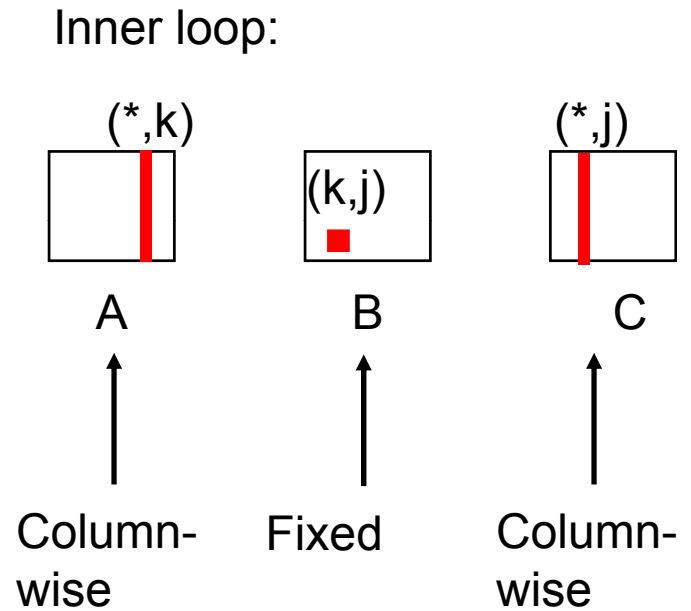


- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Matrix Multiplication (kji)

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



- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
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;  
  }  
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

```
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];  
  }  
}
```

jki (& kji):

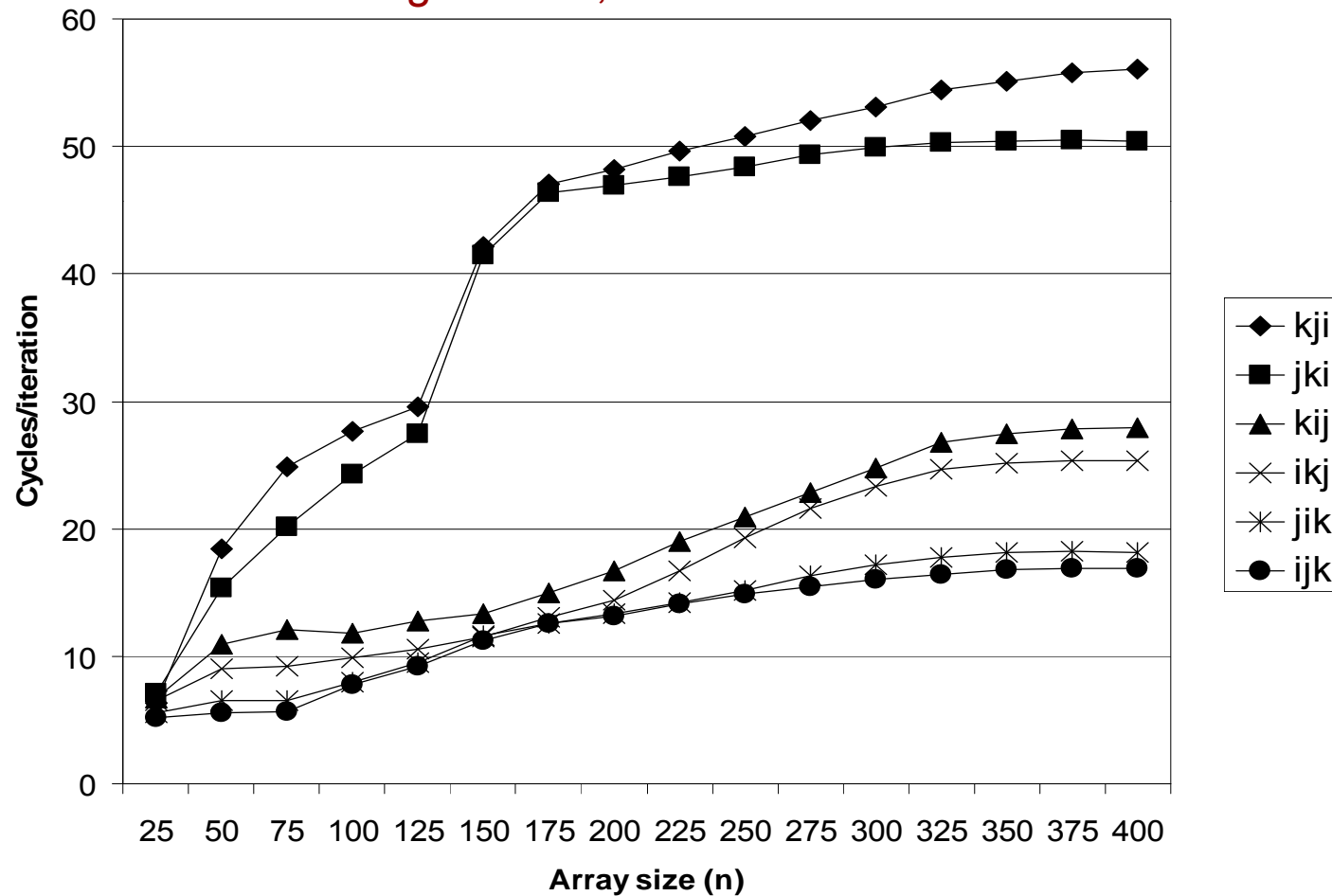
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;  
  }  
}
```

Matrix Multiply Performance

- Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



Improving Temporal Locality by Blocking

- Blocked matrix multiplication
 - “block” (in this context) does not mean “cache block”.
 - Instead, it mean a sub-block within the 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., A_{xy}) can be treated just like scalars.

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} & C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} & C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Blocked Matrix Multiply (bijk)

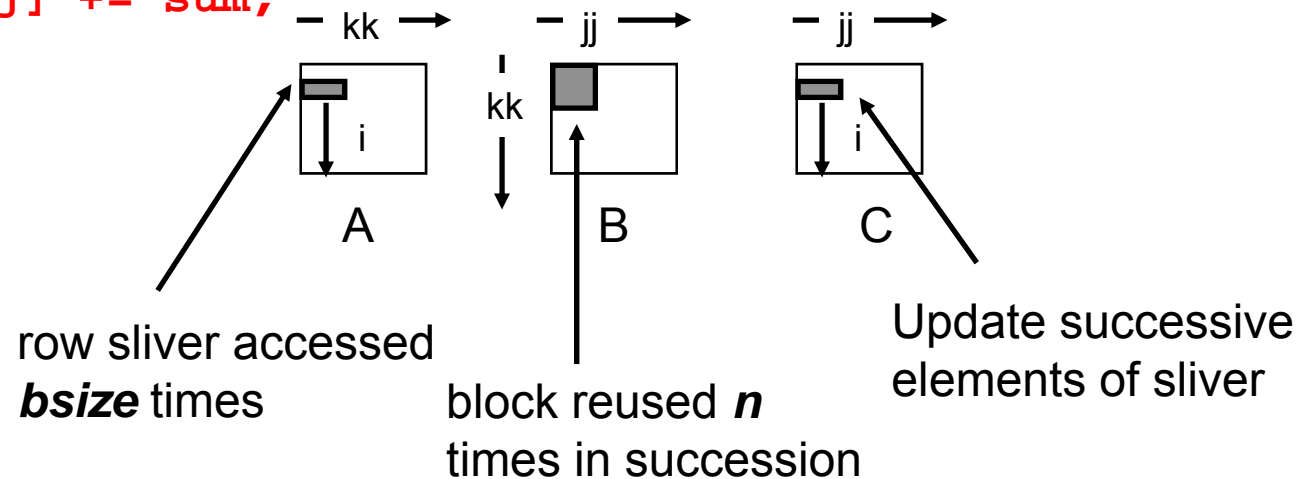
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

Blocked Matrix Multiply Analysis

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

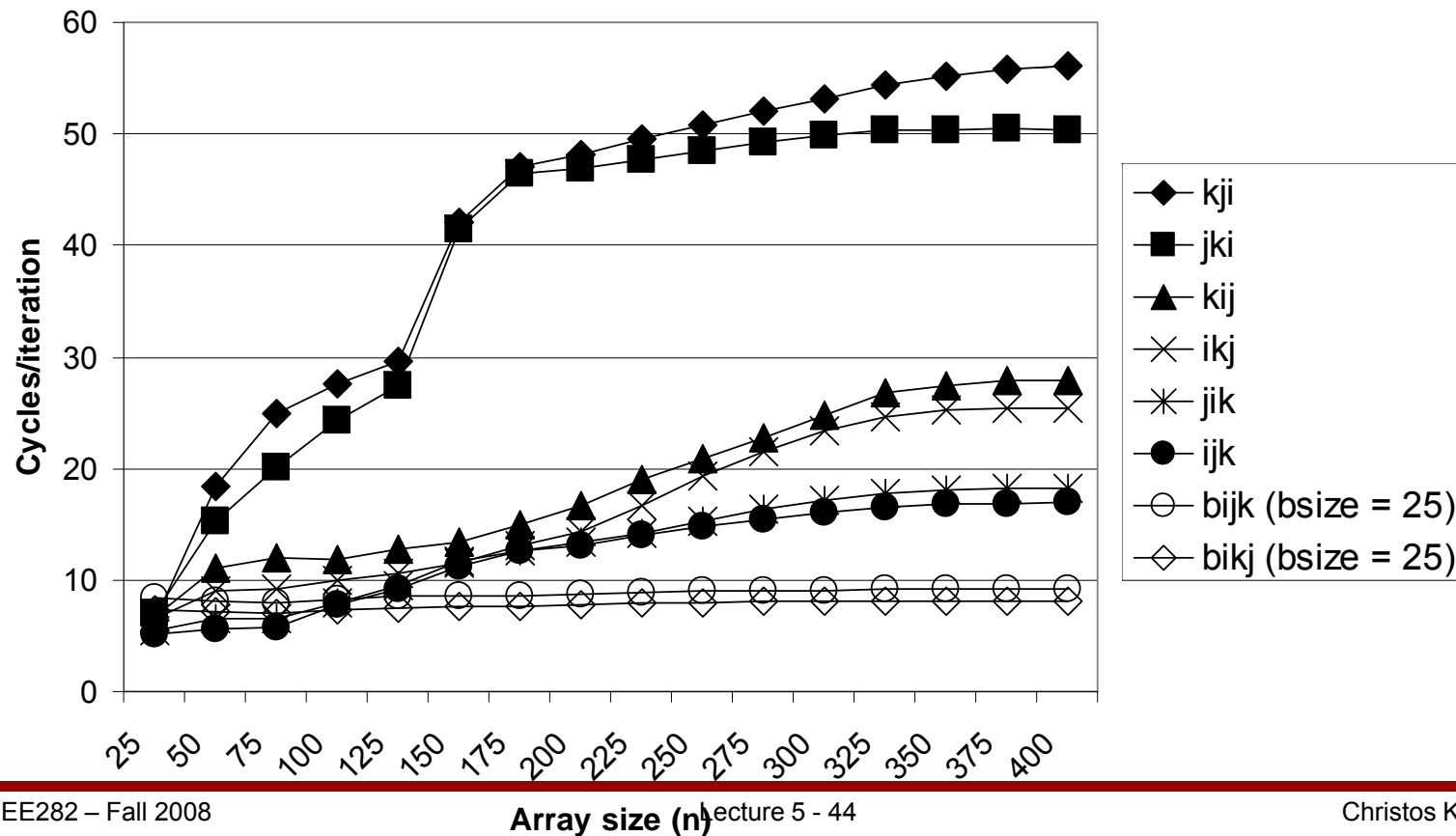
```
for (i=0; i<n; i++) {  
    for (j=jj; j < min(jj+bsize,n); j++) {  
        sum = 0.0  
        for (k=kk; k < min(kk+bsize,n); k++) {  
            sum += a[i][k] * b[k][j];  
        }  
        c[i][j] += sum;  
    }  
}
```

Innermost
Loop Pair



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.



Blocking choices

- Dimensions
 - 1D (aka strip-mining, easy...)
 - 2D (most common case)
 - 3D, ...
- Block size
 - Any ideas?
- Blocking levels
 - Any ideas?

Other Loop Optimizations

- For locality
 - Loop skewing
- To reduce loop overhead
 - Loop unrolling
 - Do twice or more work per loop iteration
 - Reduces branch overhead & exposes more ILP
- Miscellaneous
 - Inline function calls, eliminate if-statements, ...
- Always watch out for
 - Code size effects
 - Overheads
 - Branch predictability

Loop Unrolling

- Consider the following loop:

```
for (i=0; i<100; i++) a[i] = b[i] + c;
```

- Loop unrolled twice:

```
for (i=0; i<100; i+=2) {  
    a[i] = b[i] + c;  
    a[i+1] = b[i+1] + c; }  
}
```

- How to unroll K times a loop with N iterations
 - Start with original loop for (N mod K) iterations
 - Continue with unrolled loop for [N/K] iterations
 - This is called “strip mining”

Loop Unrolling

- Advantages
 - Reduces loop overhead (branches)
 - Exposes independent instructions (more ILP)
 - Very useful with wide issue machines
- Disadvantages & limitations
 - Needs more registers for renaming
 - Increased code size
 - Loop-carried dependencies
 - Limited ILP despite unrolling
 - Conditional statements (if) within each iteration

The Compiler is Your Friend

- Choose an optimizing compiler and go beyond `-O3`
 - Read your compiler's optimization manual
- Optimizations the compiler should always do for you
 - Code motion, strength reduction, common subexpression elimination, register allocation, instruction scheduling, inlining,...
- But it can often do more
 - Unrolling & software pipelining, loop optimizations, optimized build-in functions, inter-procedural analysis, prefetching, padding & alignment
- But be careful...
 - Is the optimization safe for your code (e.g. FP rounding issues)
 - Optimizations can reveal bugs in your code

Keep in Mind: Limitations of Optimizing Compilers

- Operate Under Fundamental Constraint
 - Must not cause any change in program behavior under any possible condition
- Most analysis is performed only within procedures
 - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
 - Compiler has difficulty anticipating run-time inputs
 - Profile driven compilation becoming more prevalent (JIT)
- When in doubt, the compiler must be conservative