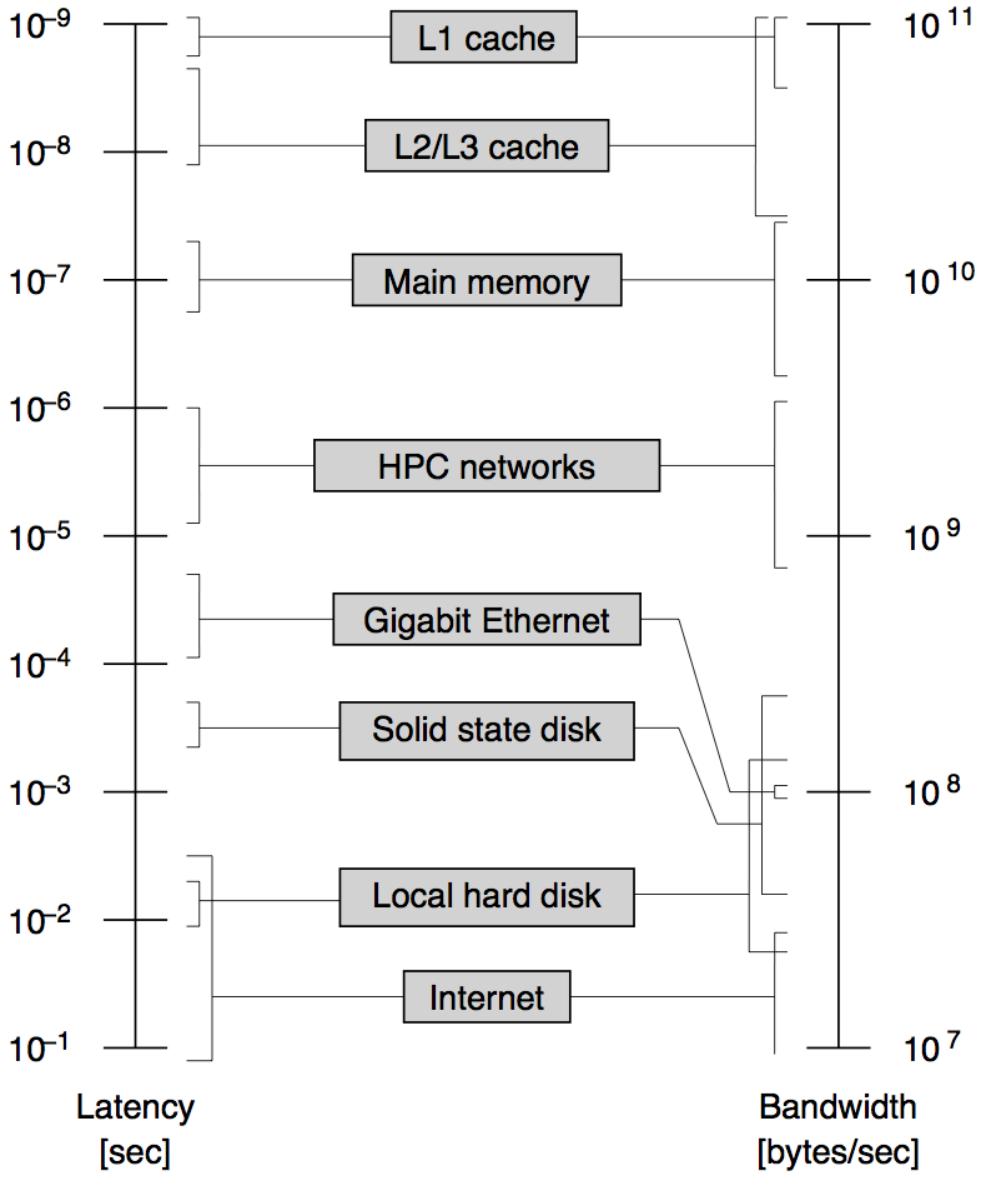
Introduction to High Performance Computing for Scientists and Engineers

Chapter 3: Data Access Optimization

Slides by Mike Heath

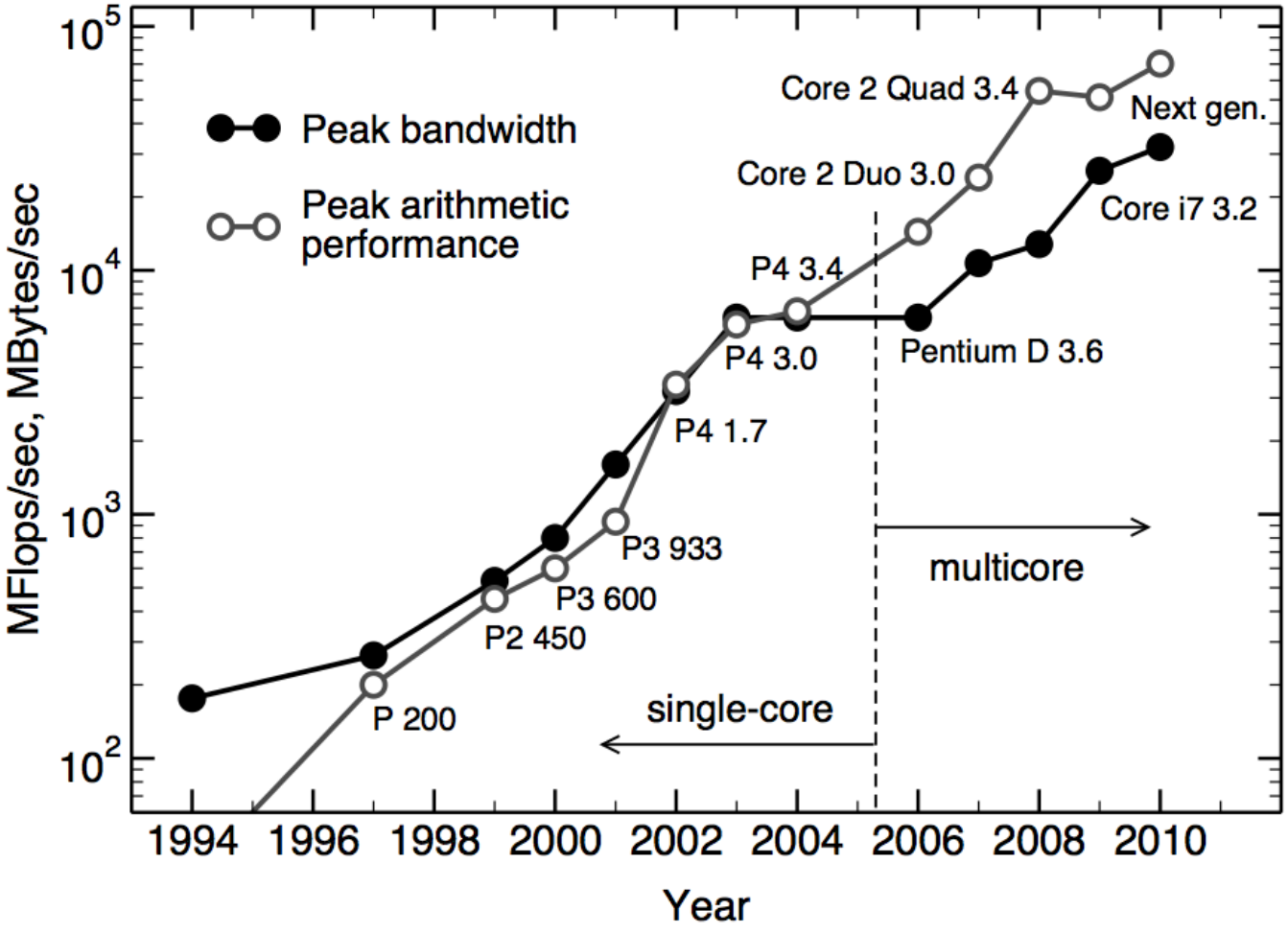
Data Access Latency & Bandwith

- Latency and bandwidth for accessing data span many orders of magnitude across memory hierarchy



Performance & Bandwidth Trends

- Gap between processor performance and memory bandwidth is growing, especially for multicore processors



Machine and Code Balance

- Balance between data access and processing speeds of machine is expressed by ratio $B_m = W/F$, where W and F are measured in words and floating-point operations, respectively, per unit time

data path	**balance** [W/F]
cache	0.5–1.0
machine (memory)	0.03–0.5
interconnect (high speed)	0.001–0.02
interconnect (GBit ethernet)	0.0001–0.0007
disk (or disk subsystem)	0.0001–0.01

- Similarly, balance between loads/stores and flops executed by program is given by $B_c = W/F$, where W and F are measured in words and floating-point operations, respectively

Performance Model

- Ratio of machine balance to code balance gives crude performance model for expected fraction of peak performance, min(1, B_m / B_c)
- For example, vector triad executes three loads, one store, and two flops per iteration, so $B_c = W/F = (3+1)/2 = 2$, and thus expected fraction of peak on processor with $B_m = 0.1$ is 0.05, or 5%

```
do i=1,N
  A(i) = B(i) + C(i) * D(i)
enddo
```

- Reciprocal of code balance, $1/B_c = F/W$, called *computational intensity* of code, provides measure of potential data reuse

STREAM Benchmarks

- STREAM benchmarks are simple loop kernels commonly used to characterize memory performance

type	kernel	DP words	flops	B_c
COPY	`A(:)=B(:)`	2 (3)	0	N/A
SCALE	`A(:)=s*B(:)`	2 (3)	1	2.0 (3.0)
ADD	`A(:)=B(:)+C(:)`	3 (4)	1	3.0 (4.0)
TRIAD	`A(:)=B(:)+s*C(:)`	3 (4)	2	1.5 (2.0)

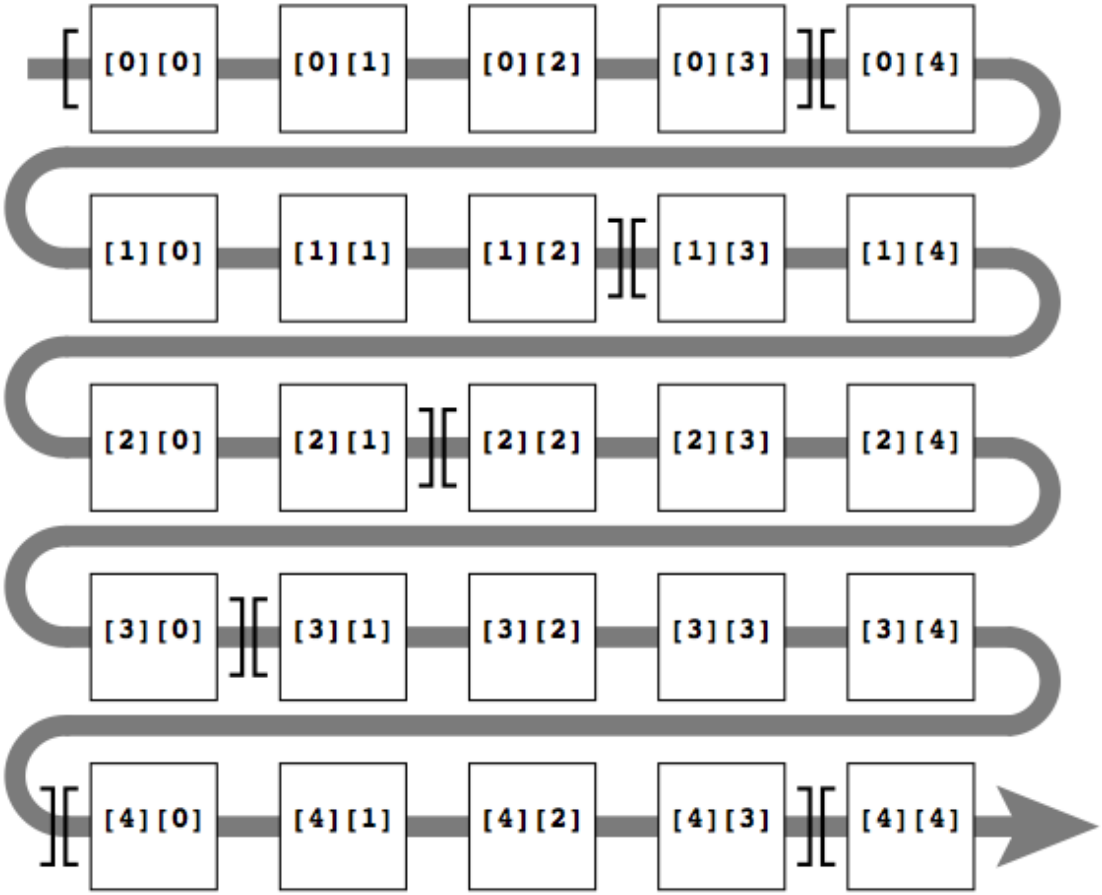
- Unfortunately, even these simple loops often fail to attain substantial fraction of peak performance
- STREAM benchmarks generally provide upper bound on performance expected of more realistic codes

Storage Order

- Multidimensional arrays, especially two-dimensional matrices, are extremely common in scientific codes
- Machine memory layout is inherently one-dimensional, divided into cache lines
- Mapping of multidimensional arrays to one-dimensional memory, as well as order in which array entries are accessed, dramatically affect cache behavior of array-based programs
- For example, *strided* access to one-dimensional array (accessing every *k*th entry rather than consecutive entries) reduces spatial locality and effective utilization of memory bandwidth
- Different programming languages have different conventions for storing multidimensional arrays

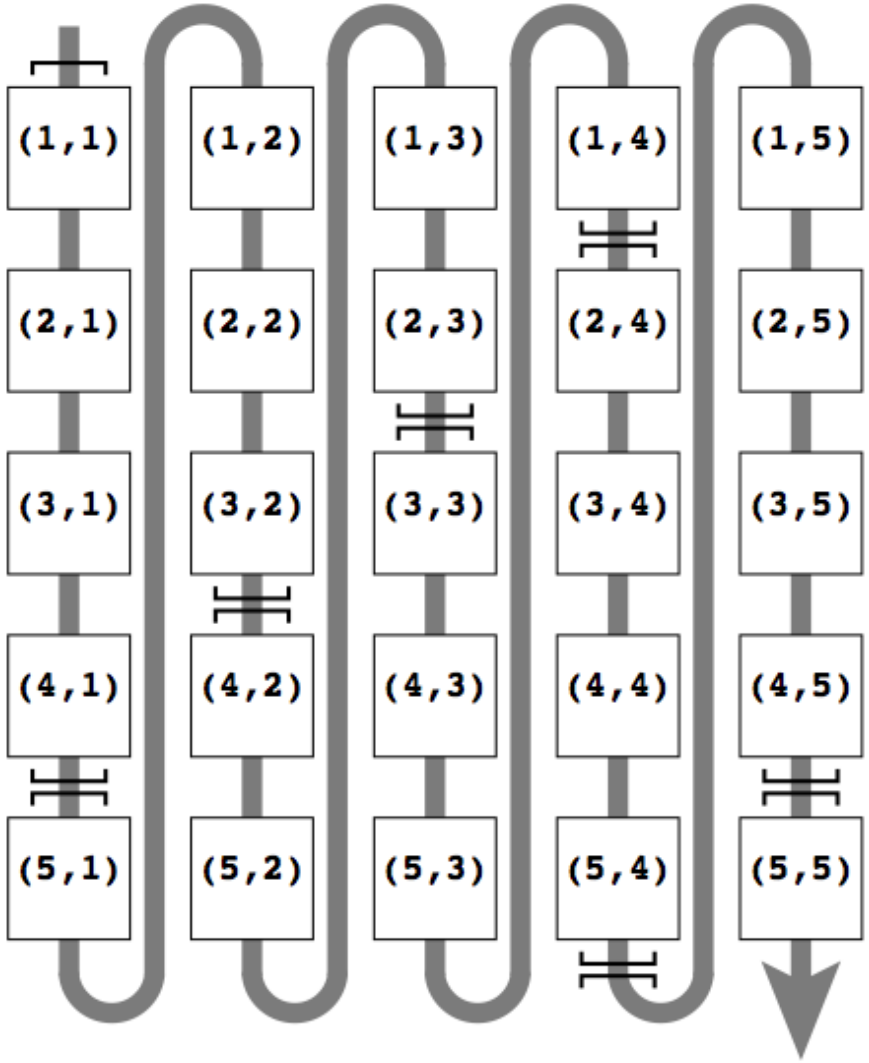
Row Major Order

- C and its variants store arrays in row major order, i.e., last subscript varies most rapidly



Column Major Order

- Fortran stores arrays in column major order, i.e., first subscript varies most rapidly



Strided Memory Access

- Because of different array storage orders, similar codes in different languages may access memory with different strides

Stride-N access

```
1 do i=1,N
2   do j=1,N
3     A(i,j) = i*j
4   enddo
5 enddo
```

Stride-1 access

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

- To optimize memory access, inner loop variable indexing multidimensional array should be chosen to ensure stride-one access (first index in Fortran, last index in C)

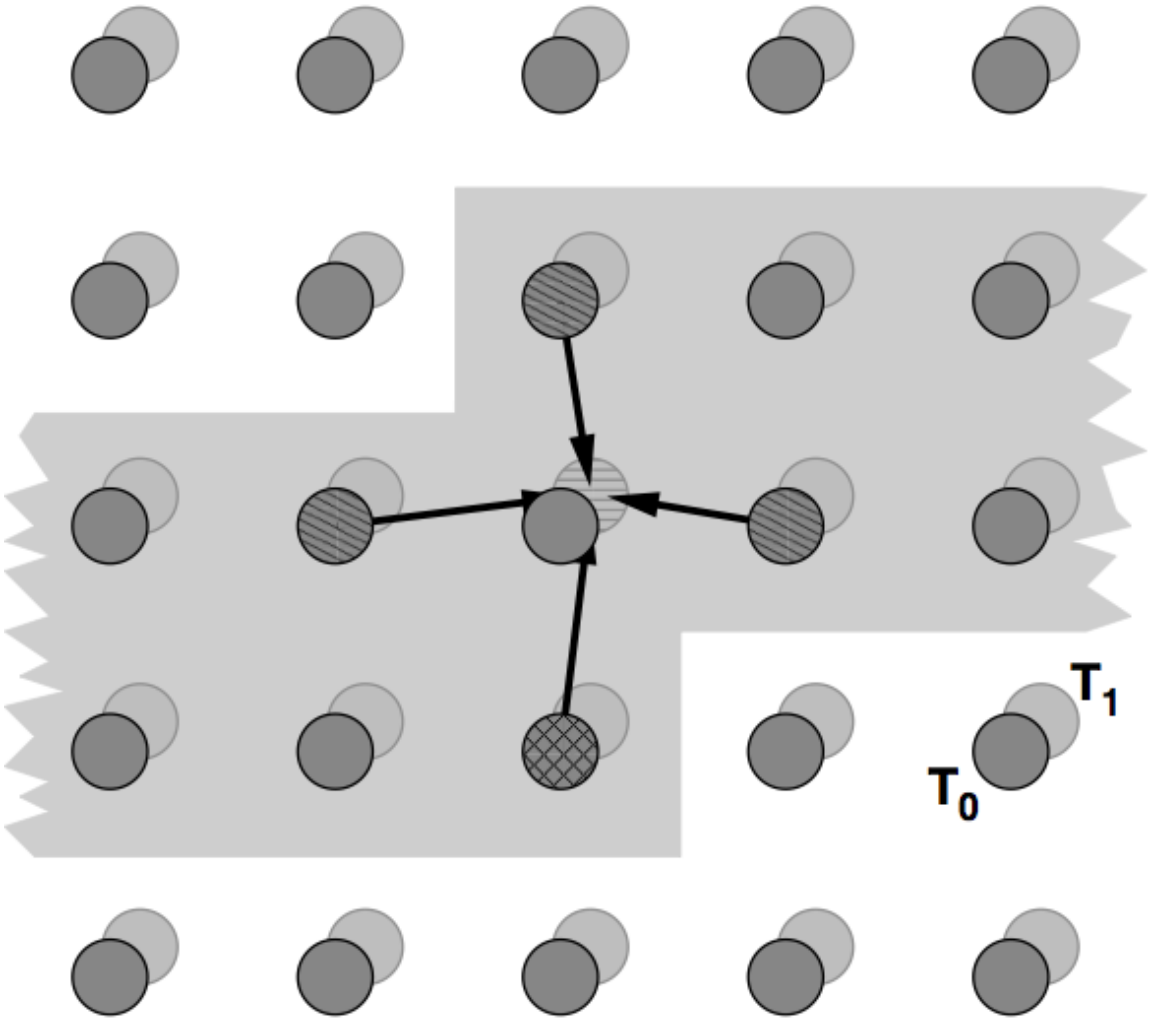
Case Study: Diffusion Equation

- Diffusion equation given by $\frac{\partial \Phi}{\partial t} = \Delta \Phi$
- Jacobi method for solving finite difference discretization

$$\frac{\delta \Phi(x_i, y_i)}{\delta t} = \frac{\Phi(x_{i+1}, y_i) + \Phi(x_{i-1}, y_i) - 2\Phi(x_i, y_i)}{(\delta x)^2} + \frac{\Phi(x_i, y_{i-1}) + \Phi(x_i, y_{i+1}) - 2\Phi(x_i, y_i)}{(\delta y)^2}$$

- Sweep through two-dimensional grid in some order, updating solution at each grid point by contributions from four neighboring grid points
- Requires two copies of solution array, as solution values cannot be overwritten until sweep is complete

Case Study: Diffusion Equation



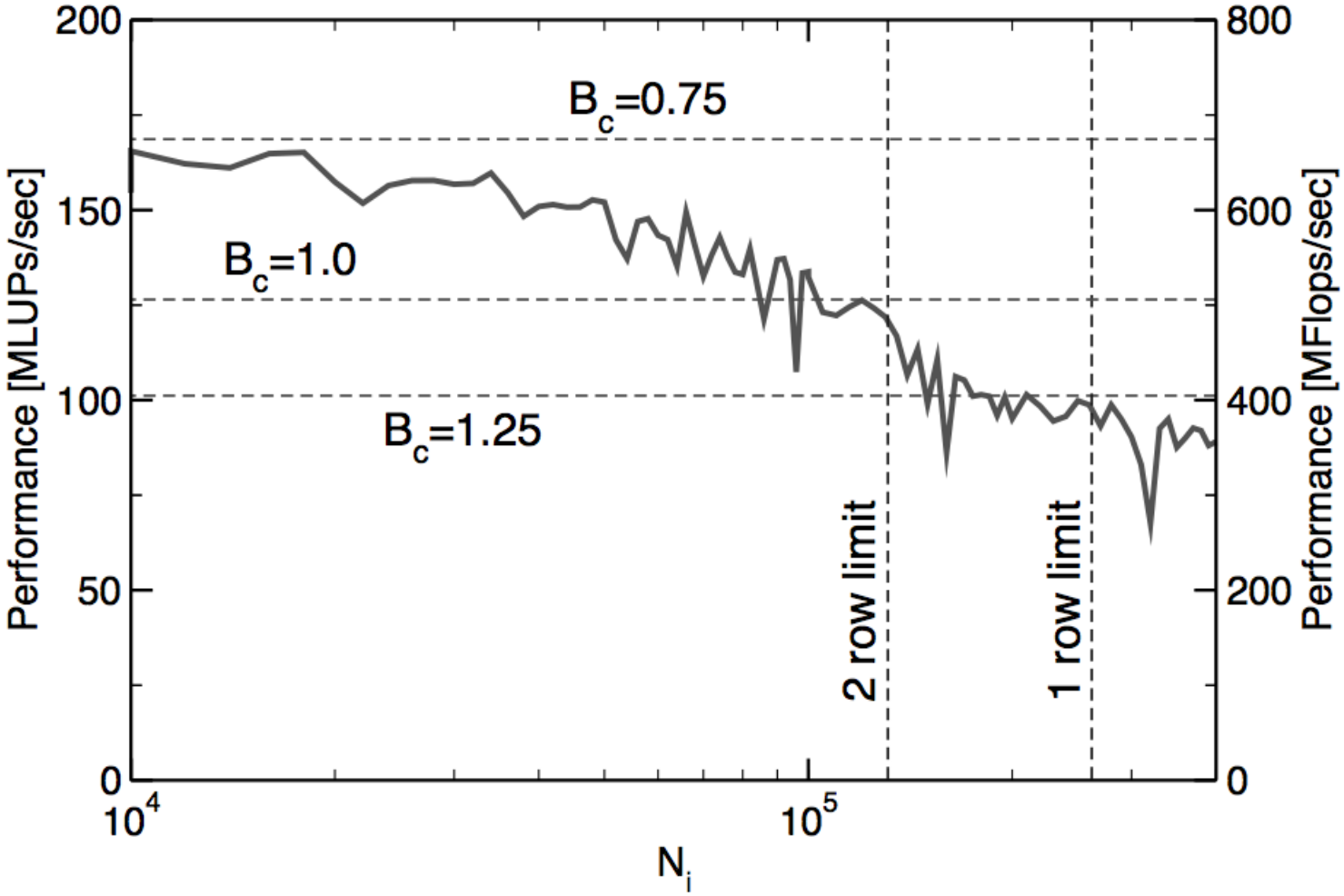
Case Study: Diffusion Equation

```
1  double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
2  integer :: t0,t1
3  t0 = 0 ; t1 = 1
4  do it = 1,itmax    ! choose suitable number of sweeps
5    do k = 1,kmax
6      do i = 1,imax
7         ! four flops, one store, four loads
8         phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
9                        + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
10     enddo
11   enddo
12   ! swap arrays
13   i = t0 ; t0=t1 ; t1=i
14 enddo
```

- Depending on cache line size, problem dimensions, and order of traversal, neighboring points may still be in cache from previous access

Case Study: Diffusion Equation

- Performance graph shows decline in performance when problem size exceeds cache size and code becomes memory bound



Case Study: Matrix Transpose

- Calculating transpose of dense matrix, $A = B^T$, involves no arithmetic operations, but illustrates performance issues in accessing memory
- Access to either A or B must be strided

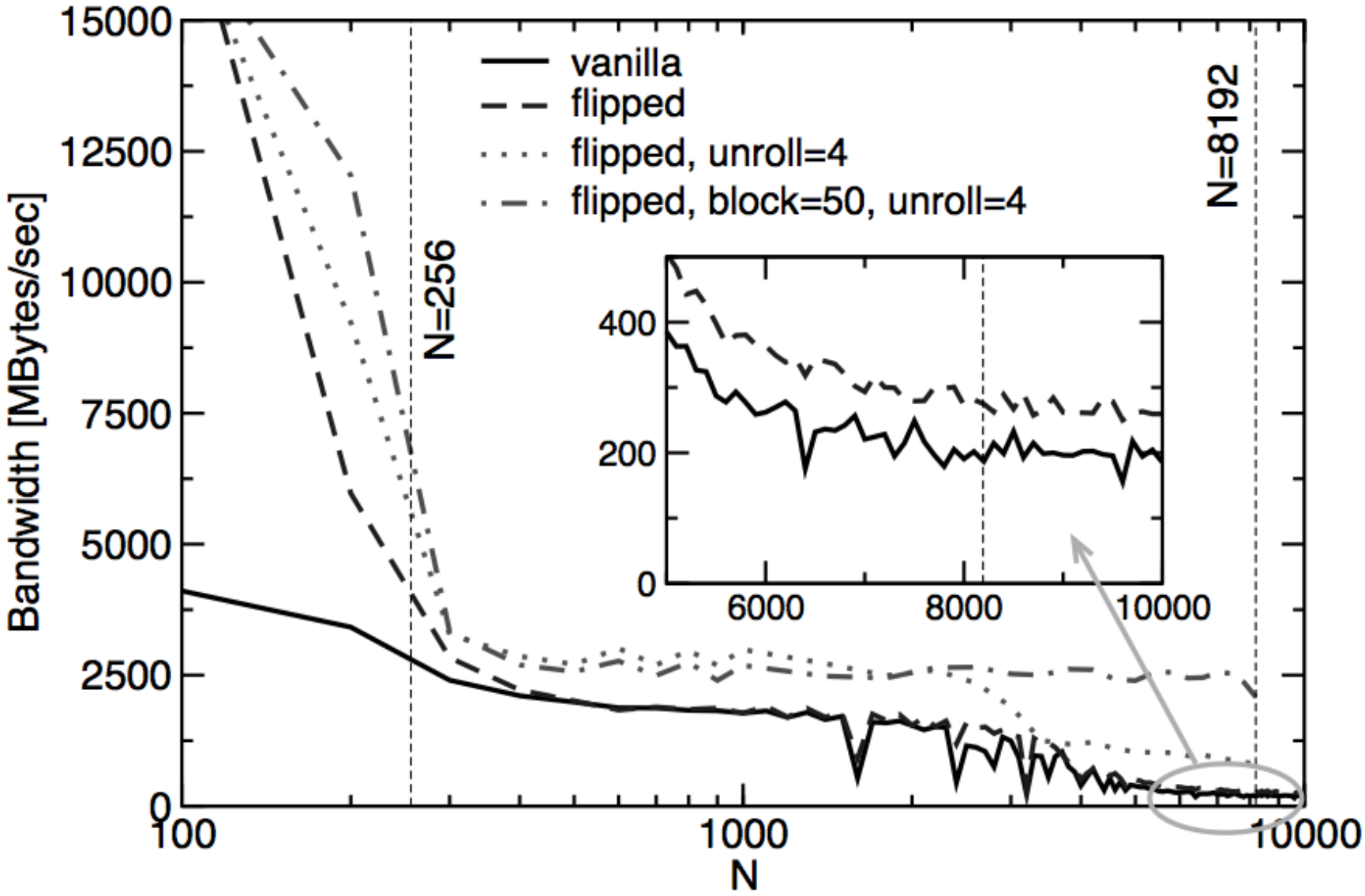
```
1 do i=1,N
2   do j=1,N
3     A(i,j) = B(j,i)
4   enddo
5 enddo
```

- Strided write more expensive than strided read because of write allocate
- Moving index i to inner loop changes access from strided writes to strided reads ("flipped")

Case Study: Matrix Transpose

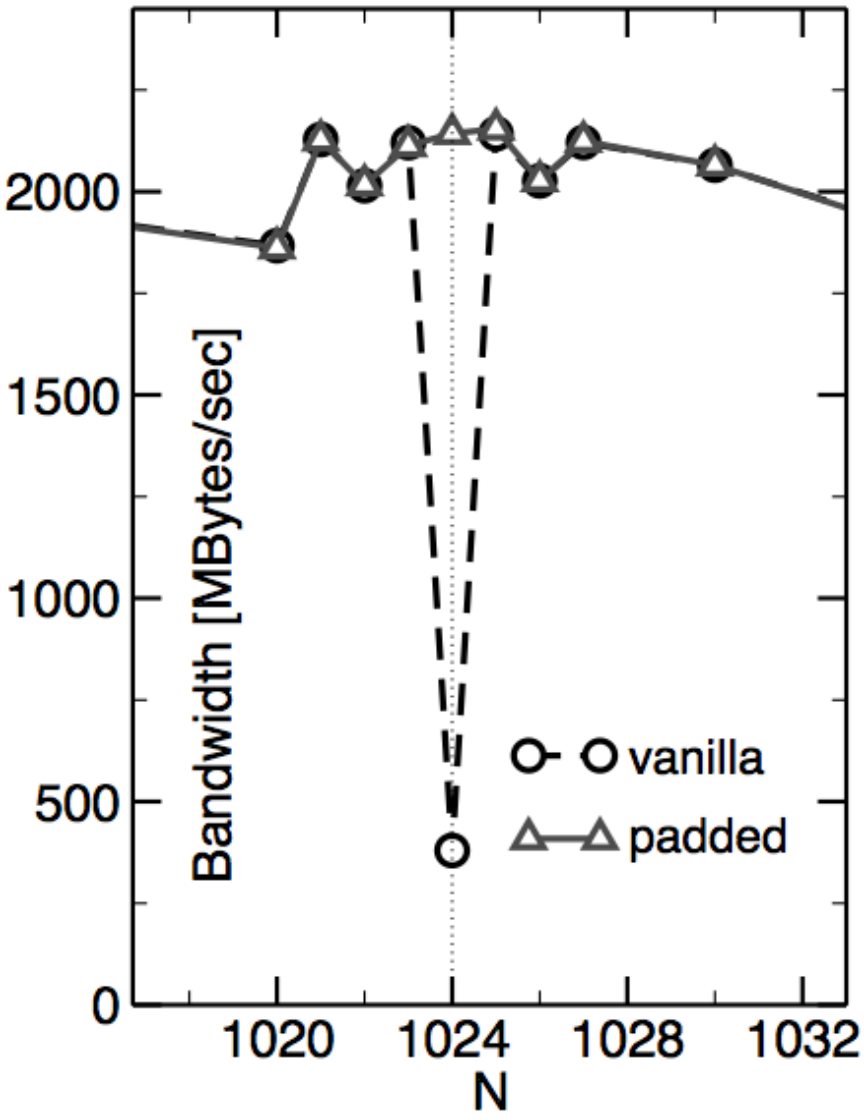
- If both matrices fit in cache ($2\ N^2 \leq C$), code should run at full cache speed despite strided access
- If matrices are too large to fit in cache, but one row or column fits in cache ($N\ L_c \leq C$), then spatial locality may still allow performance at near full cache speed
- If matrices are so large that one row or column does not fit in cache ($N\ L_c > C$), then spatial locality is lost and performance drops
- TLB (translation lookaside buffer) misses can also dramatically affect performance
- TLB caches mapping between logical and physical memory pages

Case Study: Matrix Transpose



Case Study: Matrix Transpose

- If dimension of matrix *N* happens to match cache line size, then strided access can cause cache thrashing
- Padding array can eliminate this effect



Algorithm Classification

- Algorithms can be classified according to ratio of number of arithmetic operations to number of data items involved
- For example,
  - vector addition: $O(N)$ arithmetic operations, $O(N)$ data
  - matrix-vector multiplication: $O(N^2)$ operations, $O(N^2)$ data
  - matrix-matrix multiplication: $O(N^3)$ operations, $O(N^2)$ data
- Opportunities for reusing data already in cache are obviously greater when number of operations greatly exceeds number of data items

O(N)/O(N)

- When number of operations and number of data items are both proportional to problem size, opportunities for data reuse are limited and performance is generally memory bound
- Although loops are not nested, multiple loops can potentially combined to reduce number of loads, as in *loop fusion*

```
1  do i=1,N
2    A(i) = B(i) + C(i)
3  enddo
4  do i=1,N
5    Z(i) = B(i) + E(i)
6  enddo
```

loop fusion →

```
   ! optimized
   do i=1,N
     A(i) = B(i) + C(i)
   ! save a load for B(i)
     Z(i) = B(i) + E(i)
   enddo
```

- Compilers can often apply this optimization

$O(N^2)/O(N^2)$

- This type of algorithm usually involves nested loops with two levels
- Consider code for matrix-vector multiplication

```
1 do i=1,N
2   tmp = C(i)
3   do j=1,N
4     tmp = tmp + A(j,i) * B(j)
5   enddo
6   C(i) = tmp
7 enddo
```

- Matrix ***A*** is loaded once, but vector ***B*** is loaded N times, once for each iteration of outer loop
- We can fuse N inner loops by *loop unrolling*, traversing outer loop with stride m and replicating inner loop m times
- This technique is called *unroll and jam*

Examples: Unroll and Jam

- Matrix-vector multiply

```
1  ! remainder loop ignored
2  do i=1,N,m
3    do j=1,N
4      C(i) = C(i) + A(j,i) * B(j)
5      C(i+1) = C(i+1) + A(j,i+1) * B(j)
6      !  m times
7      ...
8      C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
9    enddo
10 enddo
```

- Matrix transpose

```
1  do j=1,N,m
2    do i=1,N
3      A(i,j)     = B(j,i)
4      A(i,j+1)   = B(j+1,i)
5      ...
6      A(i,j+m-1) = B(j+m-1,i)
7    enddo
8  enddo
```

Example: Loop Blocking

- Loop blocking can achieve optimal cache line use
- Does not reduce loads or stores, but increases cache hit ratio
- Example: matrix transpose

```
1  do jj=1,N,b
2    jstart=jj; jend=jj+b-1
3    do ii=1,N,b
4      istart=ii; iend=ii+b-1
5      do j=jstart,jend,m
6        do i=istart,iend
7          a(i,j) = b(j,i)
8          a(i,j+1) = b(j+1,i)
9          ...
10         a(i,j+m-1) = b(j+m-1,i)
11       enddo
12     enddo
13   enddo
14 enddo
```

$O(N^3)/O(N^2)$

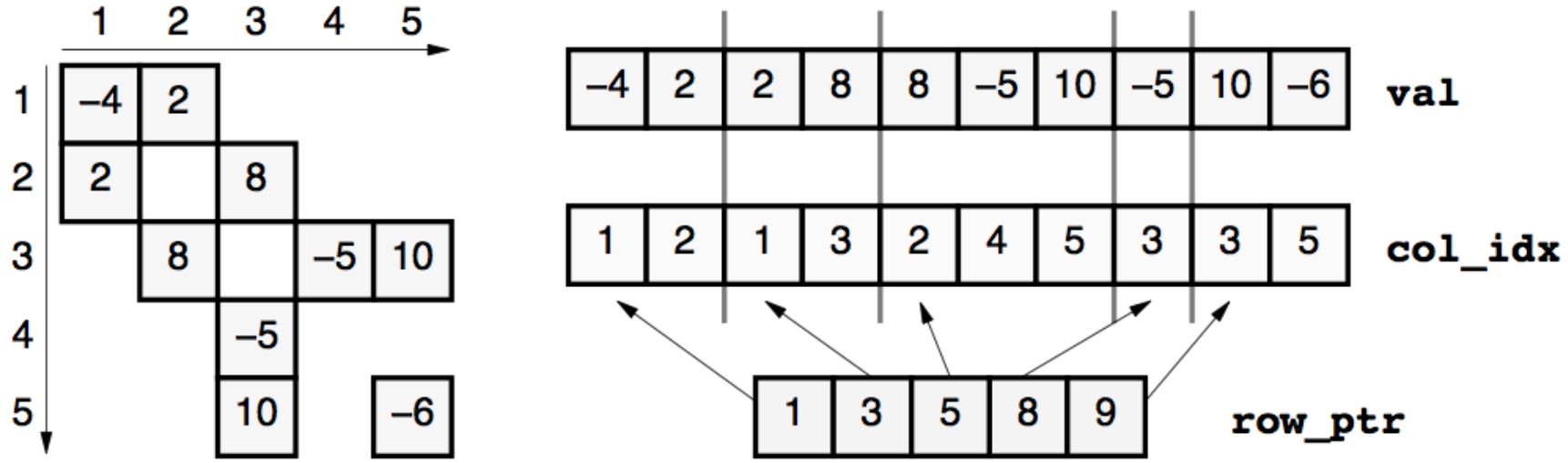
- When number of arithmetic operations exceeds number of data items by factor that grows with problem size, opportunities for reuse of data are greatly enhanced
- This type of algorithm usually involves nested loops with three levels, such as matrix-matrix multiplication
- Carefully chosen blocking and unrolling can often make code cache bound rather than memory bound
- Many vendors provide highly optimized libraries of routines for common operations of linear algebra involving dense vectors and matrices, such as BLAS (Basic Linear Algebra Subprograms), LAPACK, etc.

Sparse Matrix-Vector Multiply

- Many large matrices that arise in practice are *sparse*, with most of their entries being zero
- Sparse matrices typically have constant number of nonzero entries per row or column, and hence have $O(N)$ nonzero entries overall
- To take advantage of sparsity, special data structures must be used that store only nonzero entries and information on their location in matrix
- Two common examples are CRS (Compressed Row Storage) and JDS (Jagged Diagonals Storage)

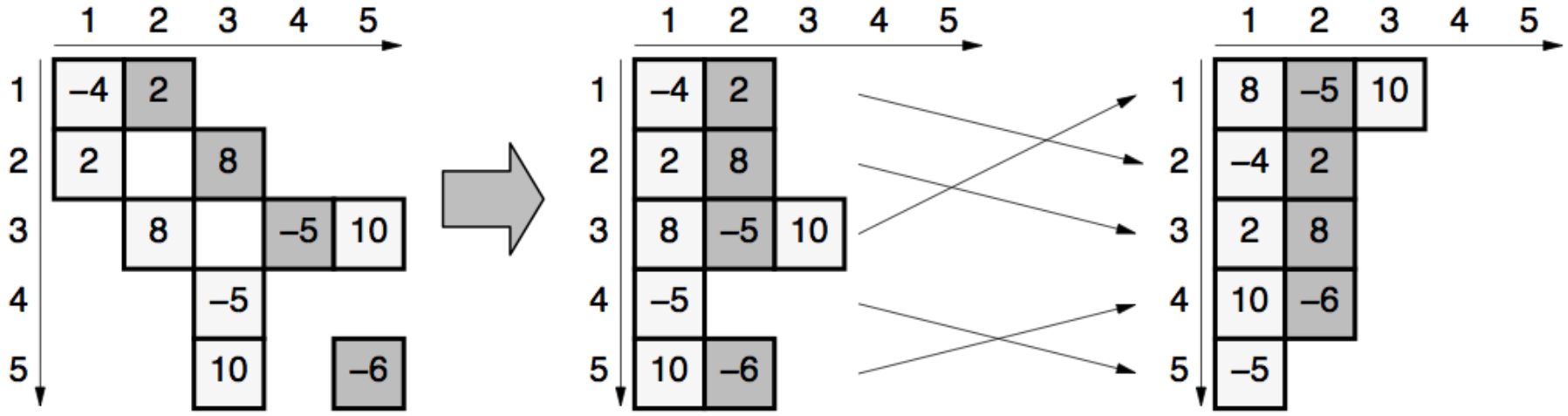
CRS — Compressed Row Storage

- Nonzeros stored row by row in single vector `val` with no gaps
- Corresponding column indices of nonzeros stored in single vector `col_idx`
- Pointers to start of each row stored in single vector `row_ptr`



JDS — Jagged Diagonals Storage

- Nonzeros in each row shifted to left
- Rows sorted by number of nonzeros in descending order
- Columns of resulting array stored consecutively in vector `val`
- Column indices and start of jagged diagonals stored in `col_idx` and `jd_ptr`



Sparse Mat-Vec with CRS and JDS

- CRS

```
1 do i = 1,Nr
2   do j = row_ptr(i), row_ptr(i+1) - 1
3     C(i) = C(i) + val(j) * B(col_idx(j))
4   enddo
5 enddo
```

- JDS

```
1 do diag=1, Nj
2   diagLen = jd_ptr(diag+1) - jd_ptr(diag)
3   offset = jd_ptr(diag) - 1
4   do i=1, diagLen
5     C(i) = C(i) + val(offset+i) * B(col_idx(offset+i))
6   enddo
7 enddo
```

CRS vs. JDS for SpMatVec

- CRS
  - long outer loop, short inner loop
  - result vector loaded only once
  - nonzeros accessed with stride one
  - W/F balance close to one
- JDS
  - short outer loop, long inner loop
  - result vector loaded multiple times
  - nonzeros accessed with stride one
  - W/F balance close to two

Optimizing SpMatVec with JDS

- Unroll and jam for nonuniform lengths of diagonals requires *loop peeling*, leaving partial diagonals to be processed separately

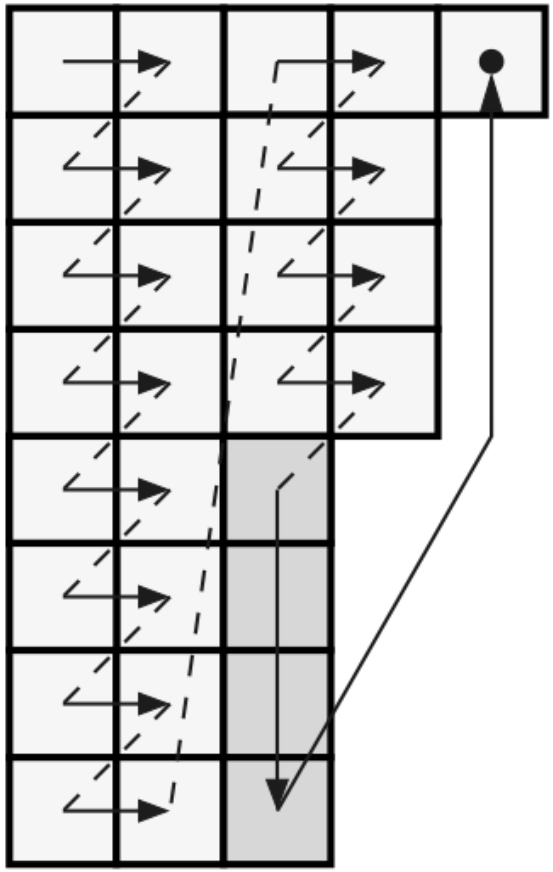
```
1  do diag=1,Nj,2  ! two-way unroll & jam
2    diagLen = min( (jd_ptr(diag+1)-jd_ptr(diag)) ,\
3                   (jd_ptr(diag+2)-jd_ptr(diag+1)) )
4    offset1 = jd_ptr(diag) - 1
5    offset2 = jd_ptr(diag+1) - 1
6    do i=1, diagLen
7      C(i) = C(i)+val(offset1+i)*B(col_idx(offset1+i))
8      C(i) = C(i)+val(offset2+i)*B(col_idx(offset2+i))
9    enddo
10   ! peeled-off iterations
11   offset1 = jd_ptr(diag)
12   do i=(diagLen+1),(jd_ptr(diag+1)-jd_ptr(diag))
13     c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
14   enddo
15 enddo
```

Optimizing SpMatVec with JDS

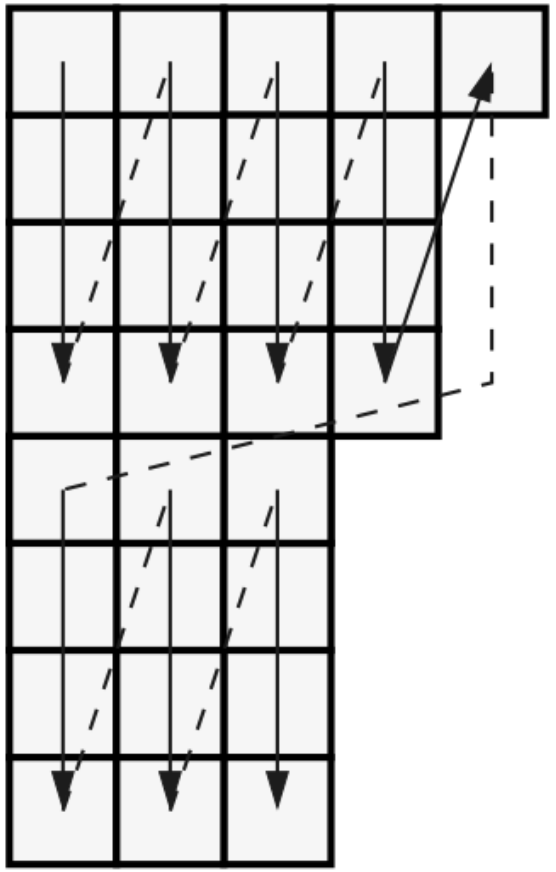
- Can also apply blocking to reduce memory traffic and enhance in-cache performance

```
1  ! loop over blocks
2  do ib=1, N_r, b
3    block_start = ib
4    block_end   = min(ib+b-1, N_r)
5    ! loop over diagonals in one block
6    do diag=1, N_j
7      diagLen = jd_ptr(diag+1)-jd_ptr(diag)
8      offset = jd_ptr(diag) - 1
9      if(diagLen .ge. block_start) then
10       ! standard JDS sMVM kernel
11       do i=block_start, min(block_end,diagLen)
12         B(i) = B(i)+val(offset+i)*B(col_idx(offset+i))
13       enddo
14     endif
15   enddo
16 enddo
```

Unrolling vs. Blocking



unrolling factor 2



blocking factor 4

Performance Comparison