Fast Tree-Structured Computations and Memory Hierarchies

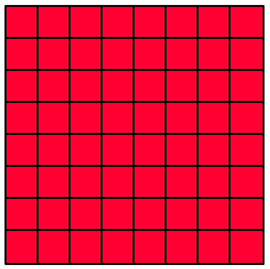
Siddhartha Chatterjee
Department of Computer Science
The University of North Carolina at Chapel Hill
sc@cs.unc.edu

http://www.cs.unc.edu/Research/TUNE/

Collaborators

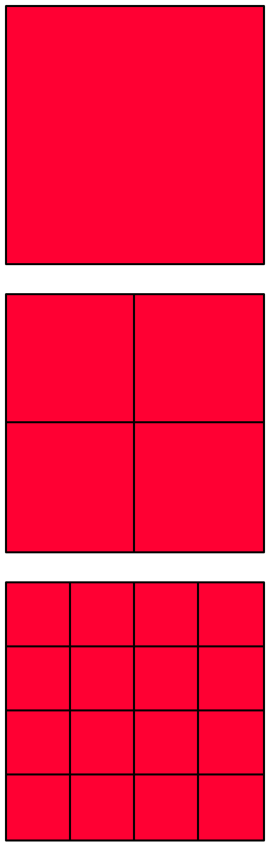
- UNC Chapel Hill
  - Prof. Sandeep Sen
  - Vibhor Jain
  - Shyam Mundhra
  - Sriram Sellappa
  - Erin Parker
  - Tom Bodenheimer
- Duke
  - Prof. Alvy Lebeck
  - Mithuna Thottethodi
- U Michigan (Math)
  - Prof. Phil Hanlon

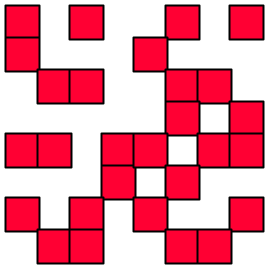
Class of Target Computations



Dense linear algebra

Hierarchical tree-structured (HTS) algorithms

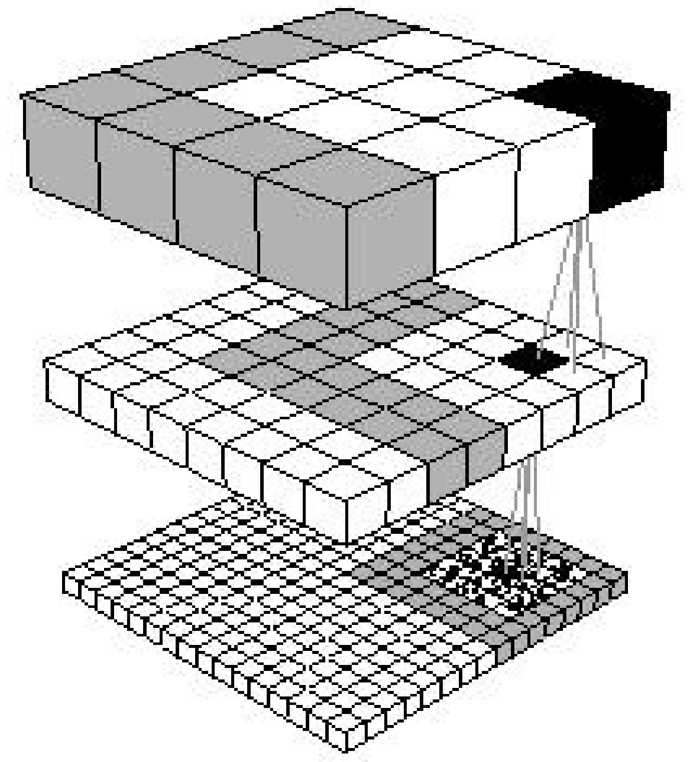




Sparse linear algebra

Application Examples

- N-body simulations
- Linear algebra
  - Matrix products
  - Eigenvalues
  - Fast transforms
  - Iterative methods for PDEs
- Radiosity/occlusion
- Computational geometry
- Others?

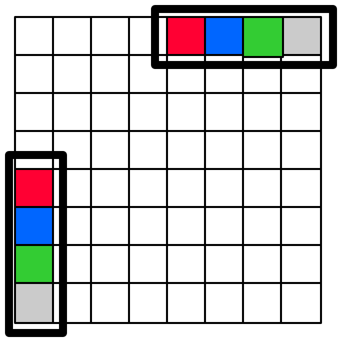


A Simple Example: Matrix Transposition

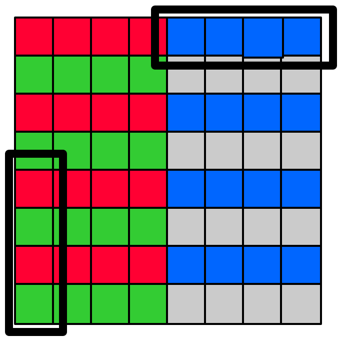
- Transpose an $n \times n$ matrix in-place
  - A structured permutation
  - Occurs as a sub-step of multi-dimensional FFT
- Seems like a simple enough computation
  - Very little spatial locality
  - No temporal locality (reuse)
  - Tricky to implement efficiently for large *n*

Matrix Transposition in More Detail

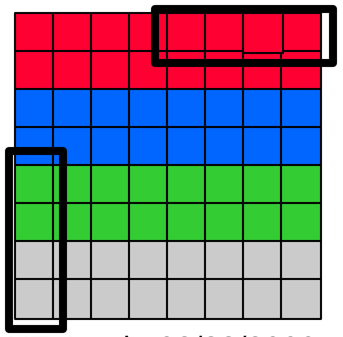
Array index space



Data cache view



TLB view



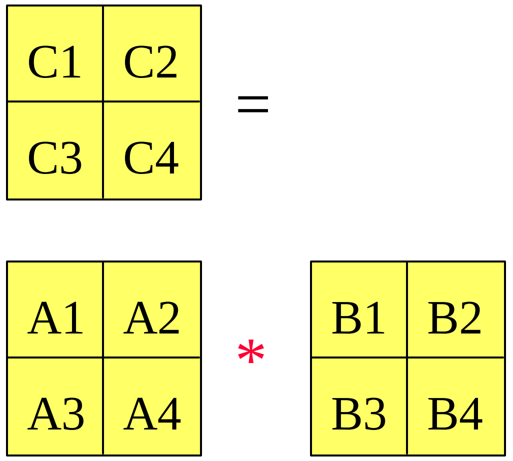
- Pairs of elements of the same color in array index space are exchanged
  - Elements (*i*,*j*) and (*j*,*i*)
- Assuming row-major (or column-major) array layout
  - Contents of memory location *ni*+*j* and *nj*+*i* are exchanged
- No temporal locality
  - Each array element is accessed at most once
- Poor spatial locality
  - Exchanging (*i*,*j*) with (*j*,*i*)
  - Exchanging (*i*,*j*+1) with (*j*+1,*i*)
  - (*i*,*j*) and (*i*,*j*+1) are adjacent in data cache and TLB view
  - How about (*j*,*i*) and (*j*+1,*i*)?
- Transposition is a *Murphy permutation* [Carter-Gatlin 1999]

Fast Matrix Multiplication

Operation Count

P1 = A1*B1
P2 = A2*B3
P3 = A1*B2
P4 = A2*B4
P5 = A3*B1
P6 = A4*B3
P7 = A3*B2
P8 = A4*B4

C1 = P1 + P2
C2 = P3 + P4
C3 = P5 + P6
C4 = P7 + P8



P1 = (A1+A4) *(B1+B4)
P2 = (A2+A4) *(B1)
P3 = (A1) *(B3 - B4)
P4 = (A4) *(B2 - B1)
P5 = (A1+A3) *(B4)
P6 = (A2 - A1)*(B1+B3)
P7 = (A3 - A4)*(B2+B4)

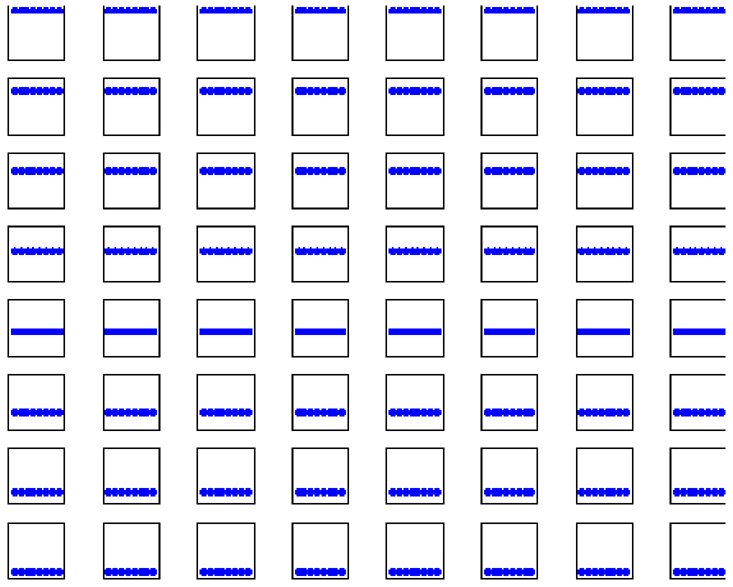
C1 = P1 + P4 - P5 + P7
C2 = P2 + P4
C3 = P3 + P5
C4 = P1 + P3 - P2 + P6

- 7 recursive products vs 8 recursive products
- Smaller operation count ($O(n^{\lg 7})$ vs $O(n^3)$)

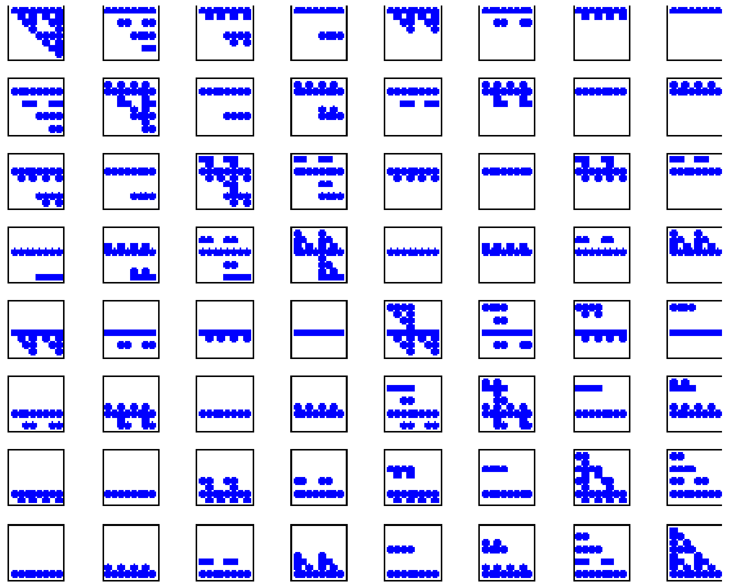
Fast Matrix Multiplication

Memory Access

Standard Algorithm



Strassen's Algorithm



- Dependence of elements of C on elements of A (for 8x8 matrix)
- Standard algorithm: C[i,j] depends on A[i,0:7]
- Strassen's algorithm: C[i,j] depends on A[?,?]

What Comprises an Algorithm?

- The algebraic computation being performed
  - This determines the set of operations to be performed and the partial order (flow dependences) among them
- The schedule of operations
  - This is a linear order (consistent with the dependences) according to which we encode the operations
  - Program transformations such as loop tiling can change this order
- The layout of data structures
- Interactions between schedule and layout
  - Each operation touches data, so the schedule determines the *logical* access sequence
  - The layout applied to the logical access sequence determines the *physical* access sequence, which determines memory system performance

Performance “Pressure Points” (1)

- Algorithm
  - Biggest wins can come from devising a better algorithm
  - Be aware of constants as well as growth rates
- Processor
  - Instruction count
  - Memory reference count
  - Instruction scheduling

Performance “Pressure Points” (2)

- Data cache
  - Limited associativity can significantly degrade running time
  - Miss latency affects crossover point between algorithmic alternatives
- Translation Lookaside Buffer (TLB)
  - Limited number of entries can lead to thrashing
- Registers
  - Register tiling can help re-order the data transfers and ameliorate the effects of limited associativity in data cache
  - Needs compiler assistance
- Array layout function
  - Changing the mapping from array index space to memory address space can benefit TLB and data cache

Three Questions for Today's Talk

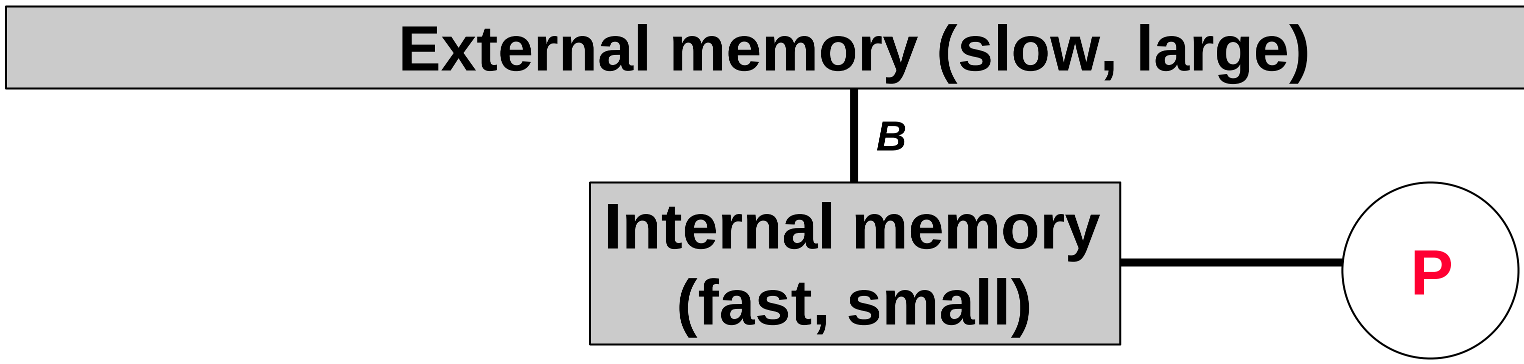
- How can we model a multi-level memory hierarchy in a realistic and predictive manner?
- What alternative data organizations are beneficial for multi-level memory hierarchies?
- How well do such ideas work in practice?

Theoretical Models for Memory Hierarchies

Theoretical Memory Models

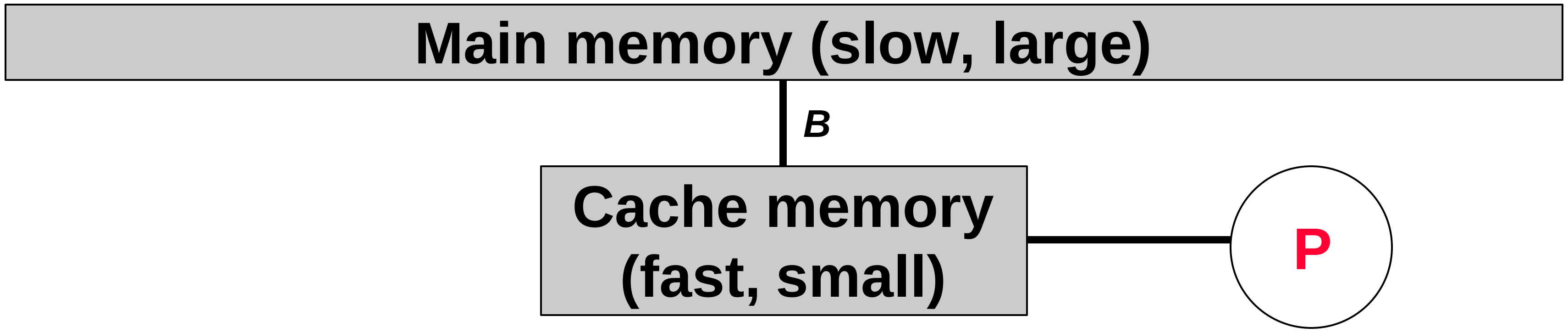
- Model 1: RAM model [Shepherdson-Sturgis 1963]
  - All memory accesses have unit cost
  - Performance metric: Total work
- Model 2: I/O model [Aggarwal-Vitter 1988]
  - Two-level model: slow and fast memory, block transfer
    - Does not model limited associativity
  - Performance metric: Number of transfers between memory levels
- Model 3: Cache-oblivious model [Frigo et al. 1999]
  - Does not use cache parameters in algorithm design, only in analysis
    - Models fully-associative “tall” cache
  - Performance metric: (Total work, memory activity)
- Model 4: Cache model [Sen-Chatterjee 2000]
  - Derived starting from I/O model; models limited associativity
    - Emulation theorem
  - Performance metric: Total work, including memory activity

I/O Model



- Model parameters
  - *M*: size of internal memory
  - *B*: block size for I/O transfers
  - *n*: input size
- Computations can be performed only on elements present in internal memory
- Fully-associative mapping between external and internal memory blocks
- Goal of algorithm design: Minimize number of I/O operations

Cache Model



- Model parameters
  - *M*: size of cache memory
  - *B*: block size for transfers between cache and main memory
  - *n*: input size
  - *L*: cache miss penalty (normalized)
- Computations can be performed only on elements present in cache
- Fixed mapping between cache blocks and memory blocks
- No explicit control on cache locations
- Goal of algorithm design: Minimize (# steps + $L \cdot$#of block transfers)

Emulation Theorem [Sen-Chatterjee 2000]

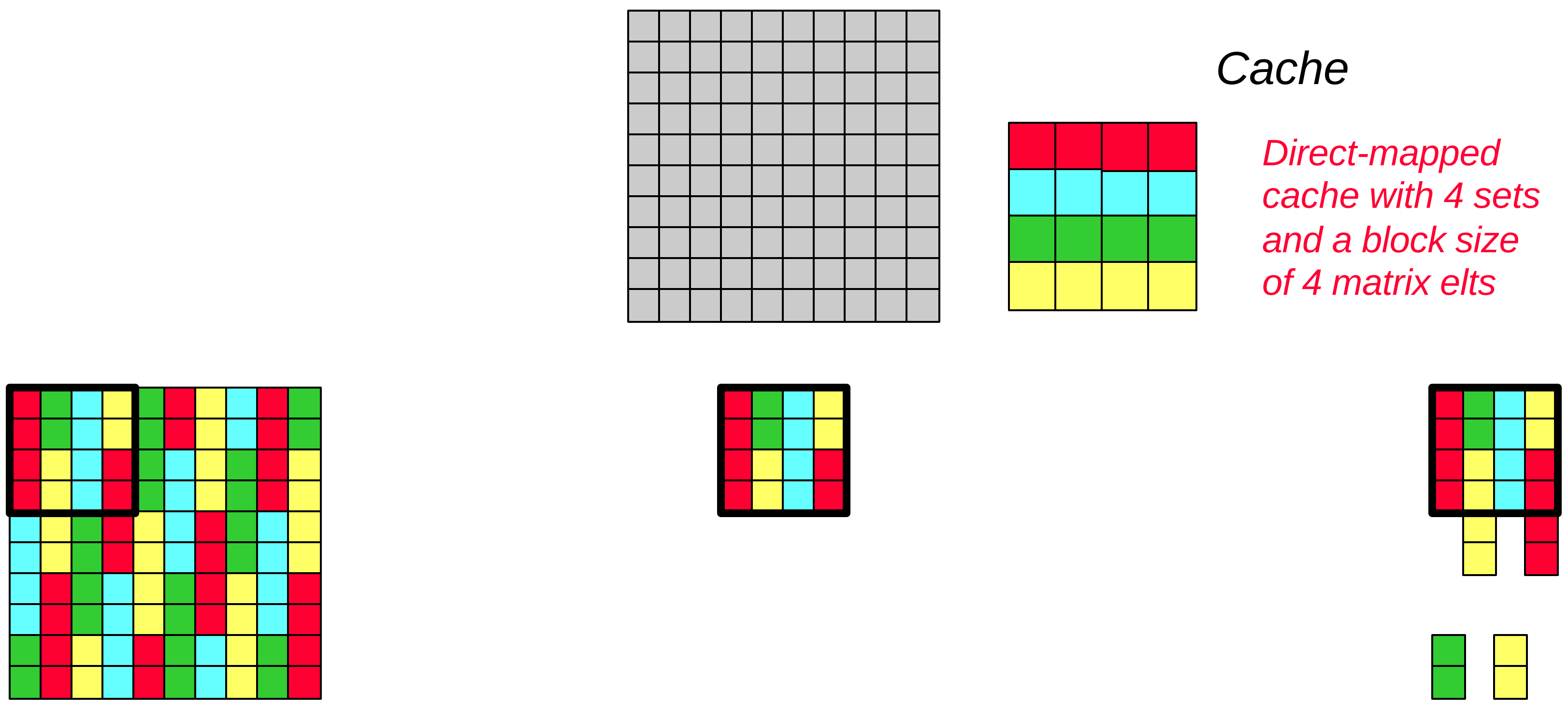
If an algorithm A in the I/O model uses T block transfers and I processing time, then the algorithm can be executed in the cache model in $O(I+(L+B)\cdot T)$ steps. The memory requirement is an additional $M/B+2$ blocks beyond that of the algorithm in the I/O model.

A block-efficient I/O algorithm can be emulated in $O(I+L\cdot T)$ steps.

- Key idea behind emulation: Careful copying of data structures into an additional buffer of size M over which we have explicit control
- Note that this causes extra memory references, but reduces the number of misses
  - In case of fast algorithm, might not be able to amortize copying cost over multiple uses of block

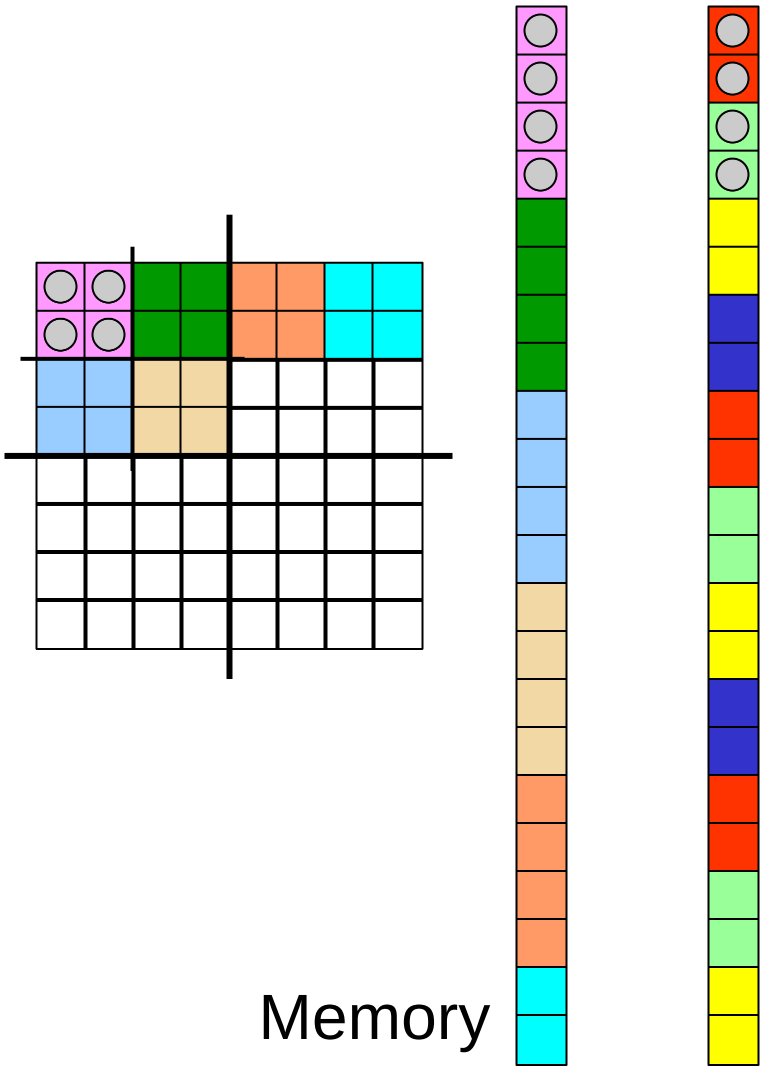
Alternative Data Layouts

Layout and Cache Behavior


Cache
Direct-mapped cache with 4 sets and a block size of 4 matrix elts

- •A tile is not contiguous in memory with row/column-major layout
- •Multi-word line size can cause cache capacity to be exceeded
- •Fixed mapping from memory to cache causes conflict misses

Making Tiles Contiguous



- Elements of a quadrant are contiguous
- Recursive layout
- Elements of a tile are contiguous
- No conflict misses in cache
- Better behavior expected in multi-level hierarchies
  - L2 cache
  - TLB

Array Layout Functions

- Required characteristics
  - One-to-one: Each index point (r,c) should map to a distinct location f
  - Onto/Dense: “Almost each” f should correspond to some (r,c)
- Desired characteristics
  - Cheap address computation
  - Incremental address computation
    - *e.g.*, L(r+1,c) - n = L(r,c) = L(r,c+1) - 1 for row-major layout
- Questions for non-standard layouts
  - How much does it enhance locality/performance?
  - What is the overhead of address computation?
  - Can addresses be computed incrementally?
  - What is the cost of format conversion?

Linear Layout Functions

- Array A with m rows and n columns
- $L_{RM}(r,c;m,n) = n \cdot r+c$
- $L_{CM}(r,c;m,n) = m \cdot c+r$
- These are canonical layouts
  - A d-dimensional array has d! canonical layouts
  - But language designer chooses and fixes one layout
- Canonical layouts favor one array axis, and dilate other axes
  - This can interact badly with caches and TLBs
- Can we do better with non-linear layouts?

Non-linear Layout Functions

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

4-D blocked

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

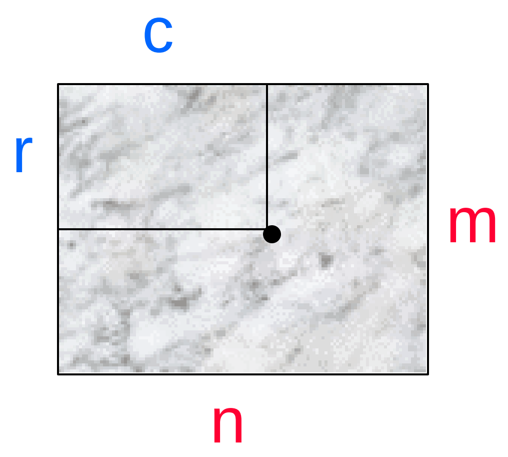
Morton order

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

Hilbert order

- Different locality properties
- Different inclusion properties
- Different addressing costs

4-D Blocked Layout



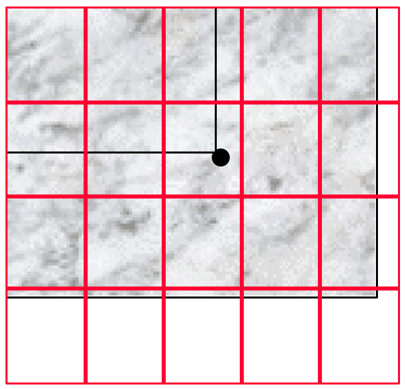
- Point (r,c) in matrix, tiles are p*q

$$(r, c) \xrightarrow{R} (t_r, t_c, o_r, o_c)$$

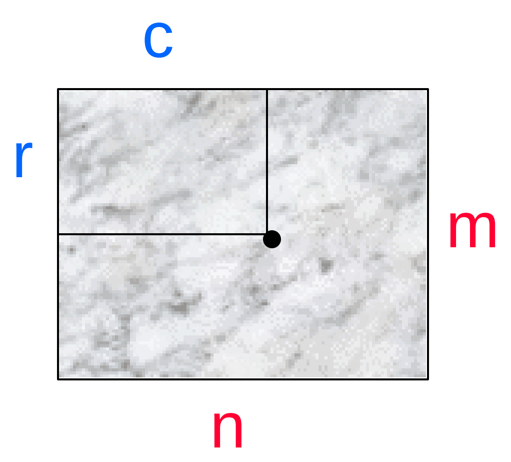
tile row *column offset*

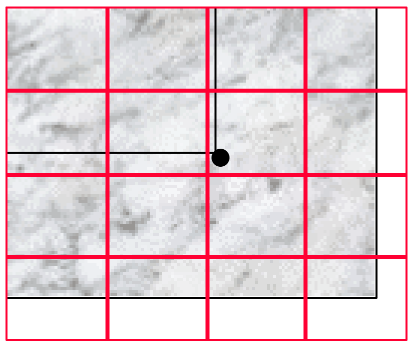
tile column *row offset*

- Mapping R is nonlinear
  - ➢ Integer quotient and remainder
- Lay out each tile contiguously in some canonical order
- Order tiles *lexicographically* by (t_r, t_c) co-ordinates
- Final layout is sum of two components

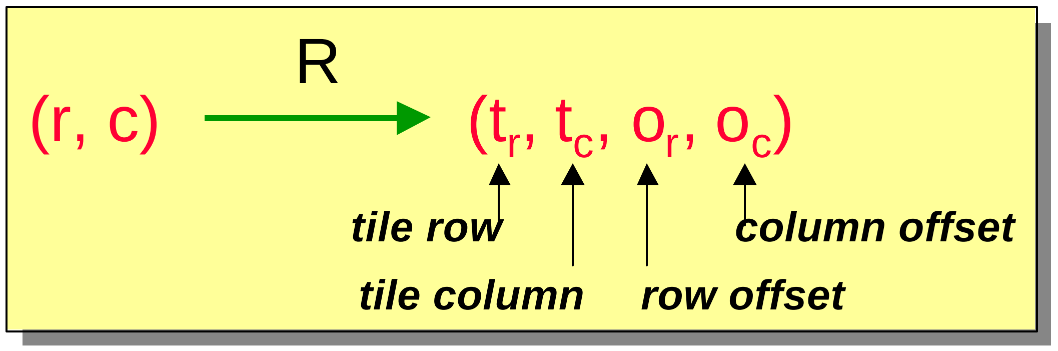


Recursive Layouts





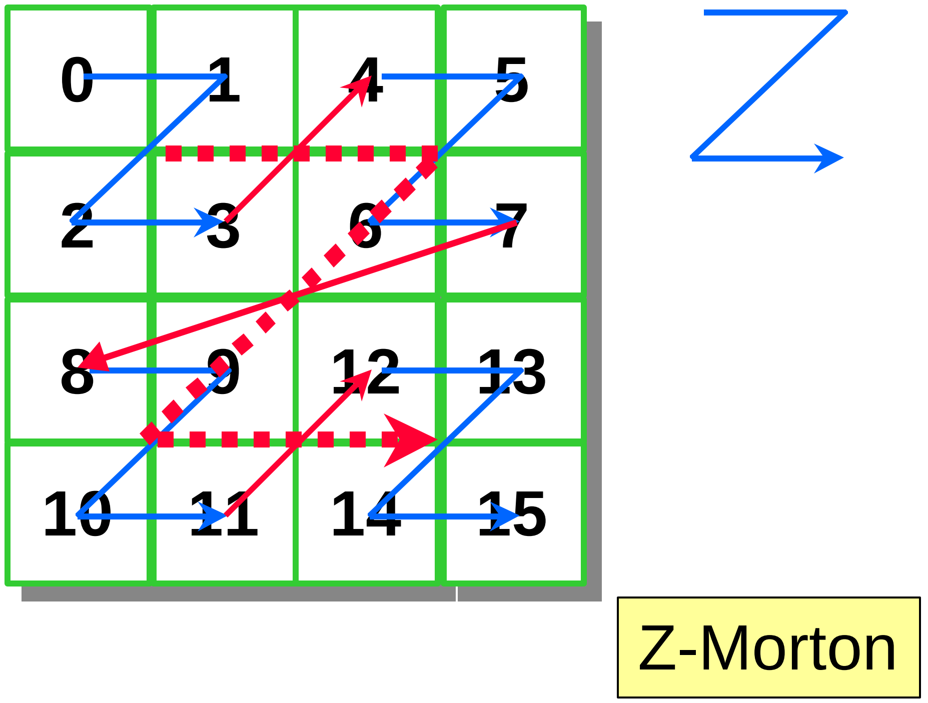
- Point (r,c) in matrix, tiles are p*q

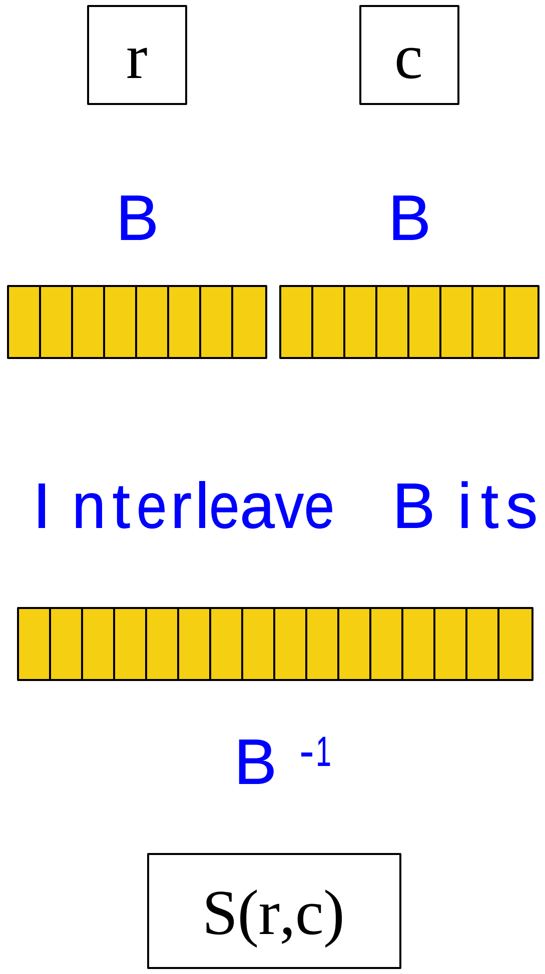


$$(r, c) \xrightarrow{R} (t_r, t_c, o_r, o_c)$$

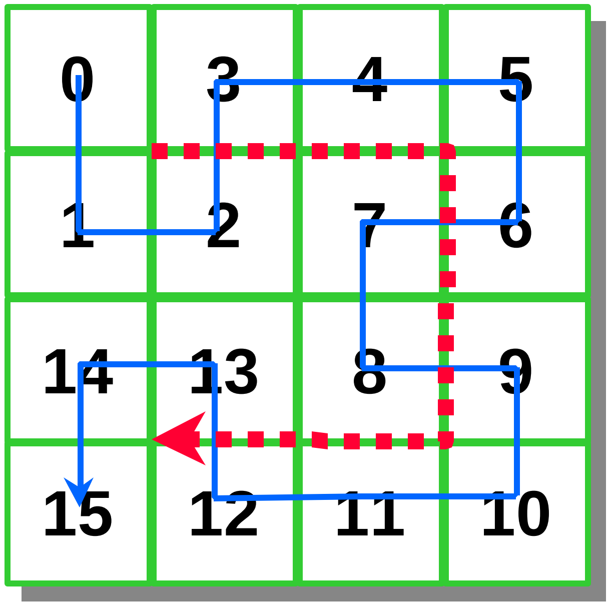
- Mapping R is nonlinear
  - Integer quotient and remainder
- Lay out each tile contiguously in some canonical order
- Order tiles *recursively* by (t_r, t_c) co-ordinates
  - Also called *quadtree* or *space-filling curve* orderings
- Final layout is sum of two components

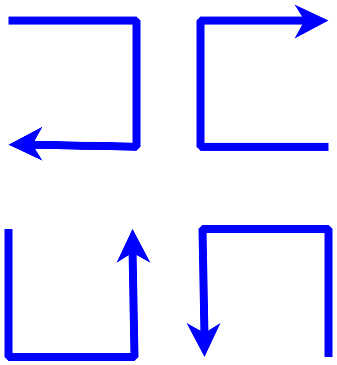
Single-Orientation Layouts



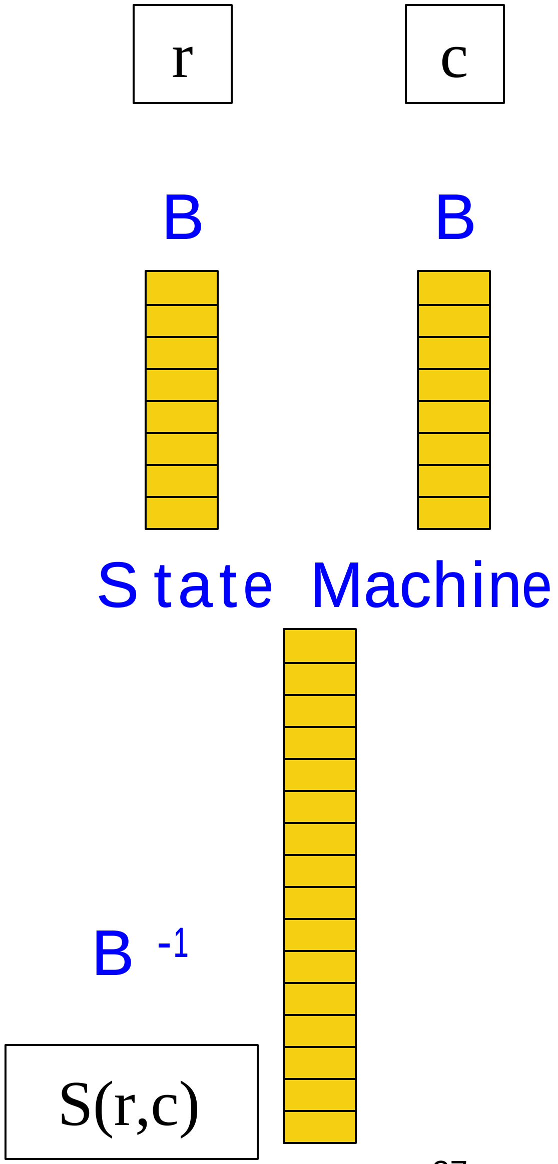


Quad-Orientation Layouts





Hilbert



Morton Order Layout

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

```
int offset(int r, int c, int n, int d, int k) {
    int b = 0;
    while (d > 0) {
        n /= 2;
        d--;
        b = 4*b+(r<n?(c<n?0:1):(c<n?2:3));
        r = r<n?r:r-n;
        c = c<n?c:c-n;
    }
    return (b*k+c)*k+r;
}
```

- Code above computes position of element (r,c) of *n*n* matrix A
  - ➢ *d* levels of subdivision
  - ➢ *k*k* tiles, column-major layout
- This is expensive!
  - ➢ Exploit incremental address calculation capability within single tile
  - ➢ “Embed” address calculation into control structure of algorithm

Appropriate Control Structures

$$C = A \bullet B$$

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

$$= \begin{bmatrix} A_{11} \bullet B_{11} + A_{12} \bullet B_{21} & A_{11} \bullet B_{12} + A_{12} \bullet B_{22} \\ A_{21} \bullet B_{11} + A_{22} \bullet B_{21} & A_{21} \bullet B_{12} + A_{22} \bullet B_{22} \end{bmatrix}$$

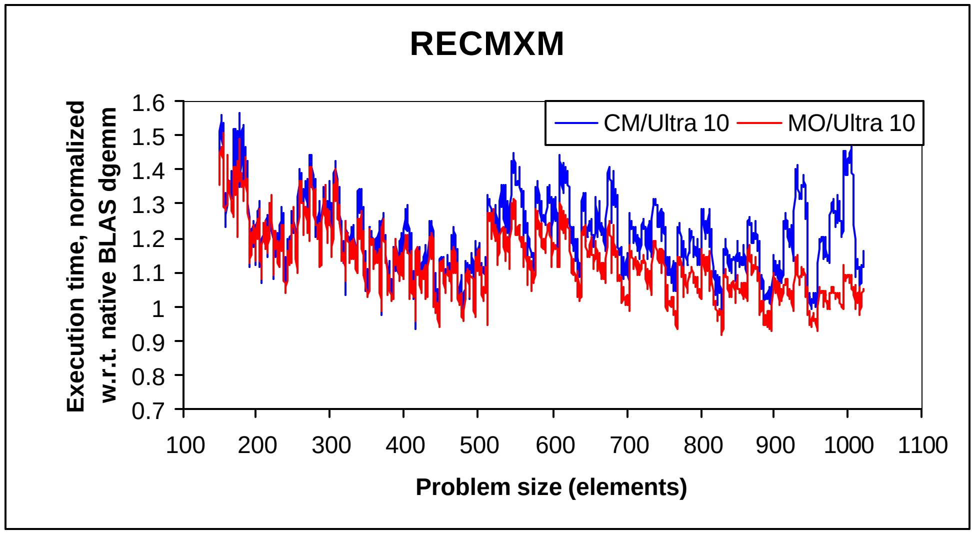
Recursion is the key!

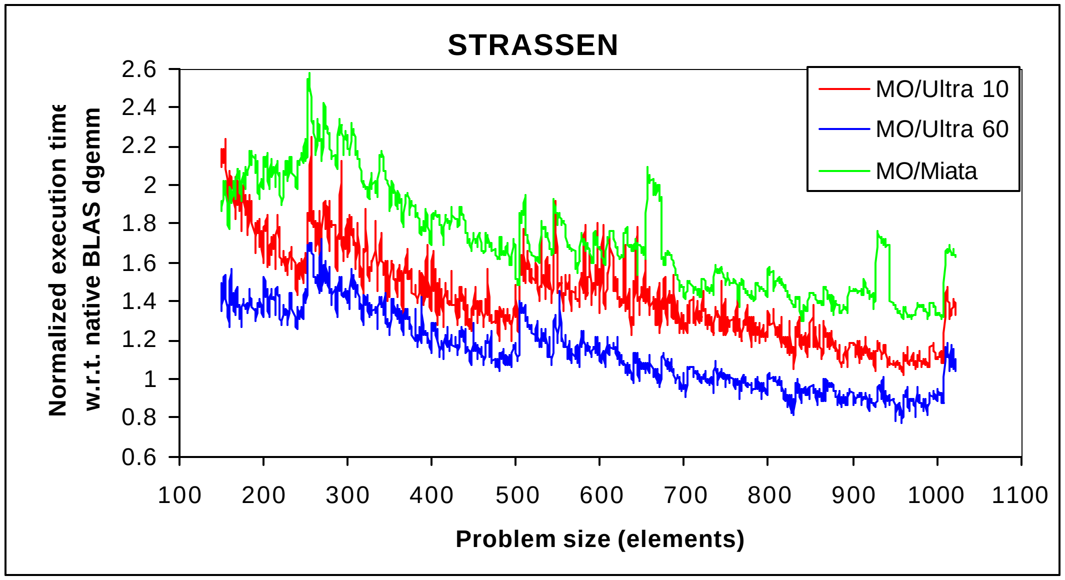
```
 up_mm(A, B, C) {
   if (leaf_level) {
      dgemm(A,B,C);
      return;
   }
   up_mm(A11, B11, C11);
   dn_mm(A11, B12, C12);
   up_mm(A21, B12, C22);
   dn_mm(A21, B11, C21);
   up_mm(A22, B21, C21);
   dn_mm(A22, B22, C22);
   up_mm(A12, B22, C12);
   dn_mm(A12, B21, C11);
}
```

```
 dn_mm(A, B, C) {
   if (leaf_level) {
      dgemm(A, B, C);
      return;
   }
   dn_mm(A12, B21, C11);
   up_mm(A12, B22, C12);
   dn_mm(A22, B22, C22);
   up_mm(A22, B21, C21);
   dn_mm(A21, B11, C21);
   up_mm(A21, B12, C22);
   dn_mm(A11, B12, C12);
   up_mm(A11, B11, C11);
}
```

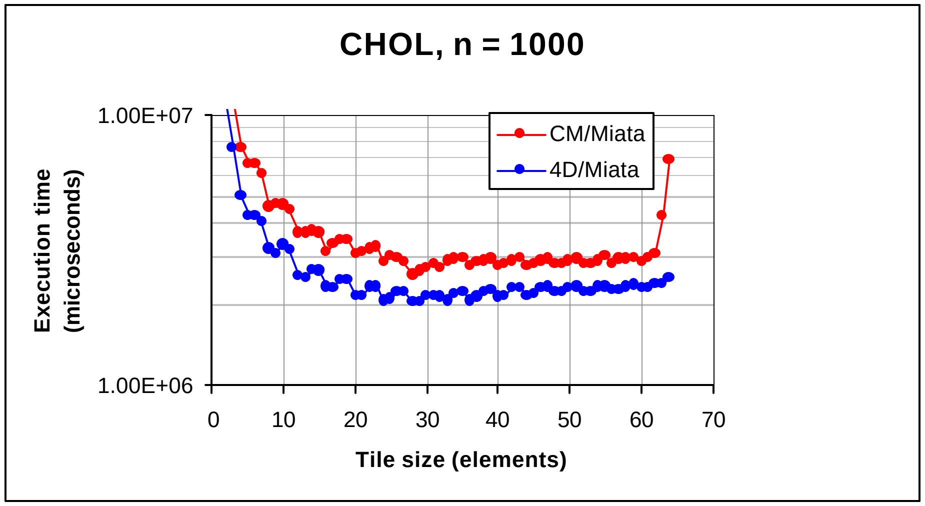
Evaluation

Nonlinear Layouts: Absolute Performance



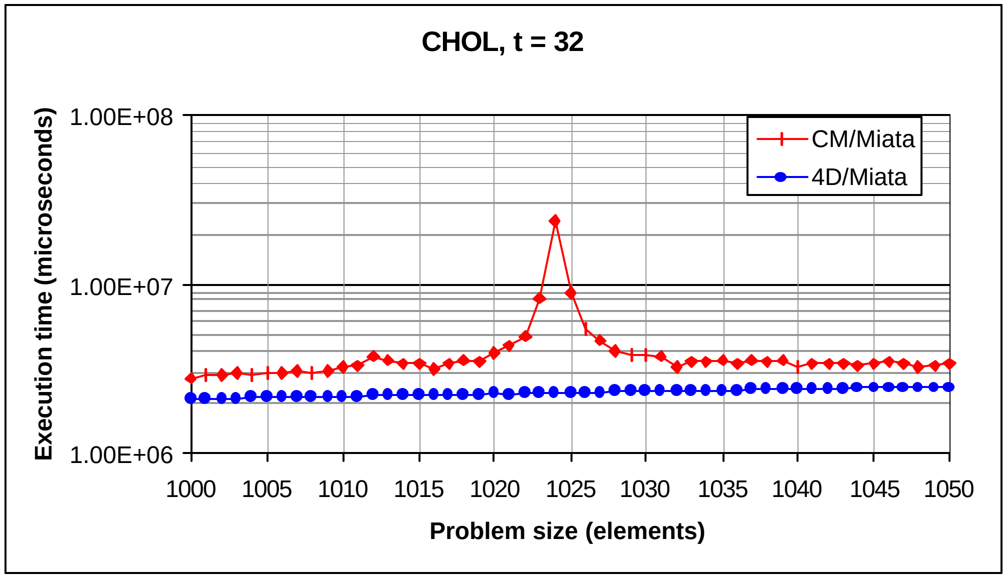


Nonlinear Layouts: Performance Sensitivity



…to choice of tile size for a fixed problem size

…to variations in problem size for a fixed tile size

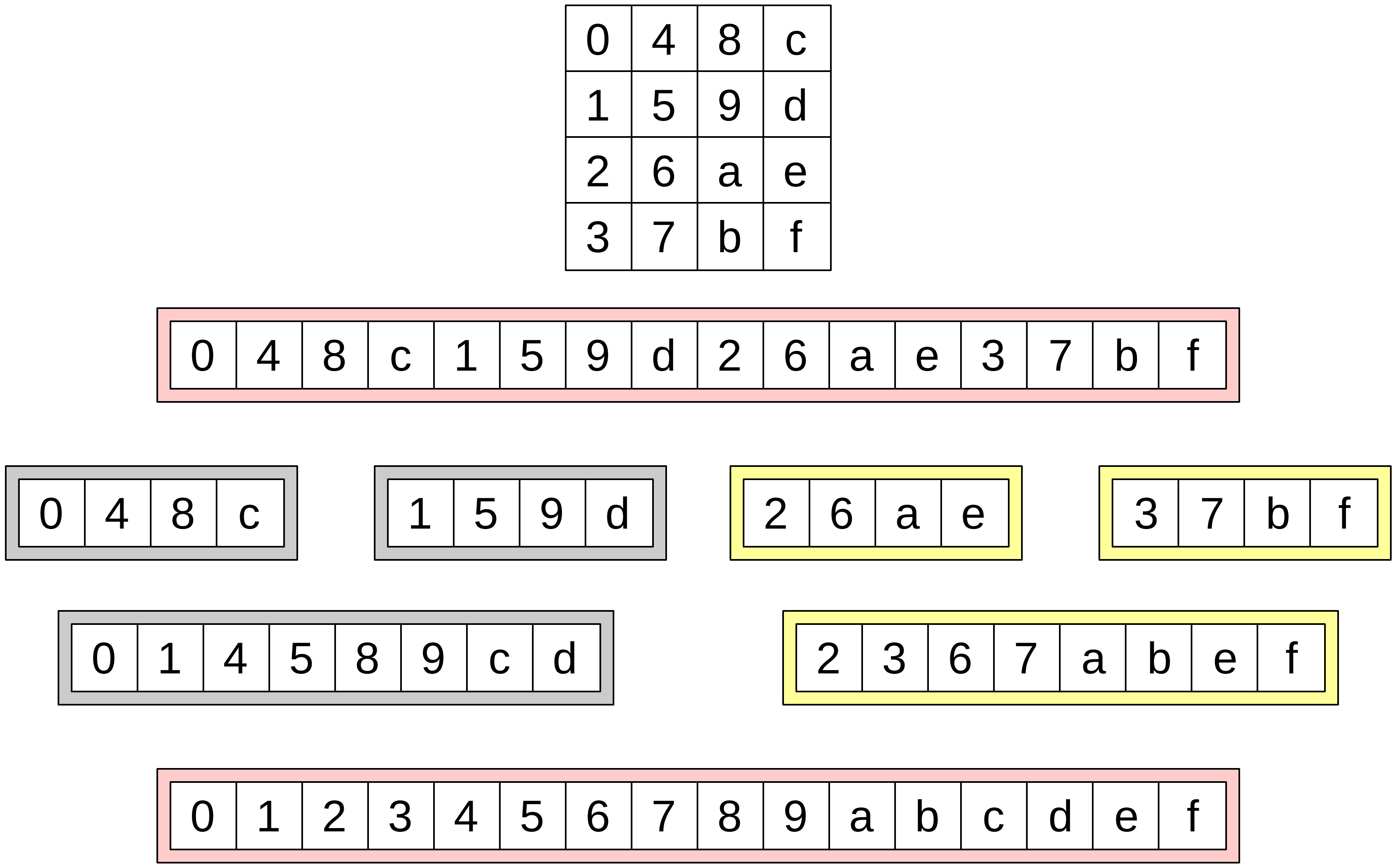


Algorithm 1: RAM Model

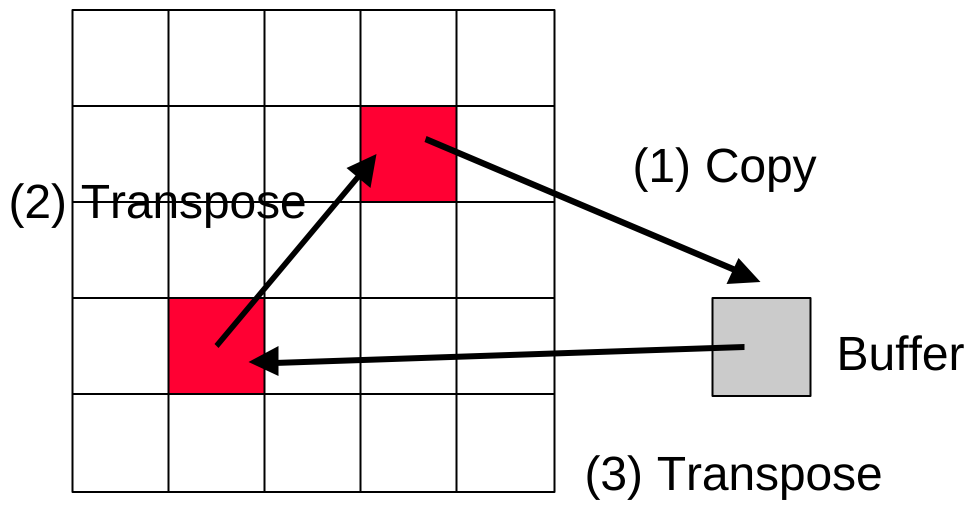
```
 for (i = 0; i < n; i++) {
   for (j = i+1; j < n; j++) {
     tmp = A[i][j];
     A[i][j] = A[j][i];
     A[j][i] = tmp;
   }
}
```

- Optimal algorithm
  - Statements executed $n \cdot (n-1)/2$ times
  - Each statement costs constant number of operations
  - Complexity of $\Theta(n^2)$
  - Optimal up to constant factors
- Problem: Almost every loop iteration has misses
  - Catastrophic conflict misses in data cache
  - Thrashing in TLB

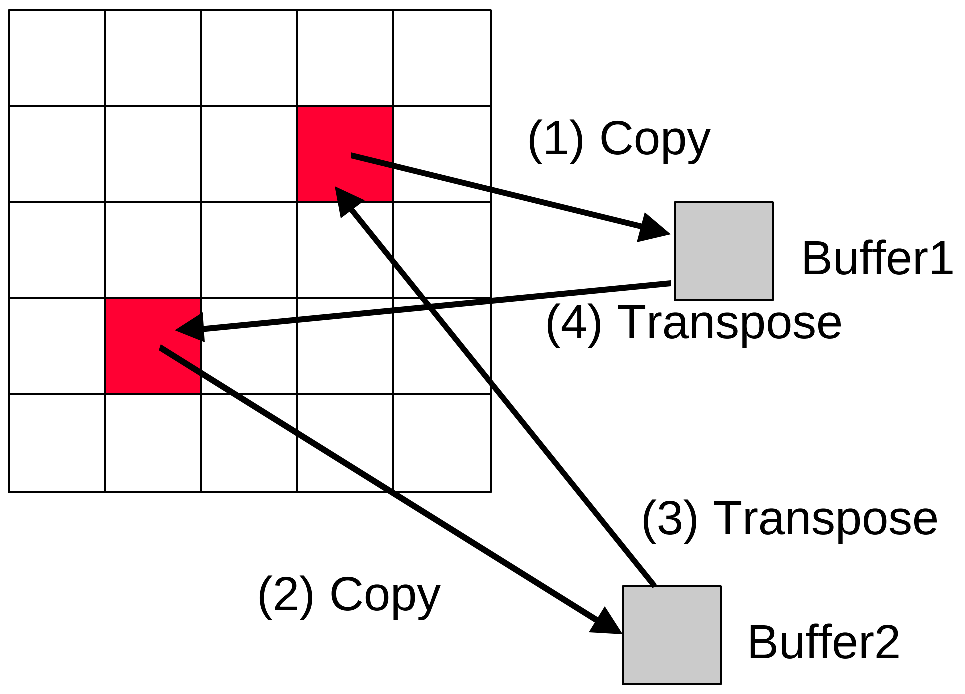
Algorithm 2: Transposing with Merge [Floyd 1972]



Algorithm 3: “Half-Copying”



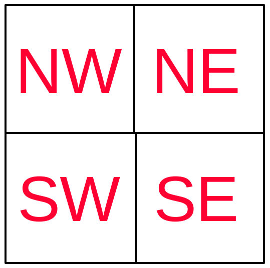
Algorithm 4: “Full-Copying”

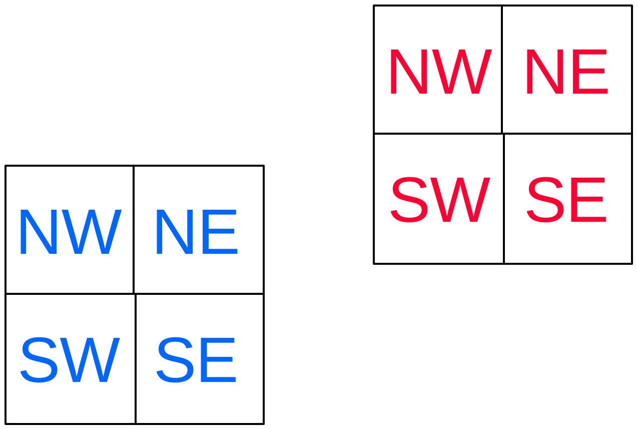


Algorithm 5: Cache-Oblivious

- Key idea
  - Use divide-and-conquer to divide problems into smaller sub-problems
  - The sub-problems will fit in cache once they are small enough
- Cache parameters not required in algorithm design stage, only for algorithm analysis
- Uses a different schedule of operations

Algorithm 6: Recursive Layout





```
tr1(int src, int num)

{
   if (num==1) {
   /* base case */
   }
   else {
     tr1(NW(src),num/4);
     tr2(NE(src),SW(src),num/4);
     tr1(SE(src),num/4);
   }
```

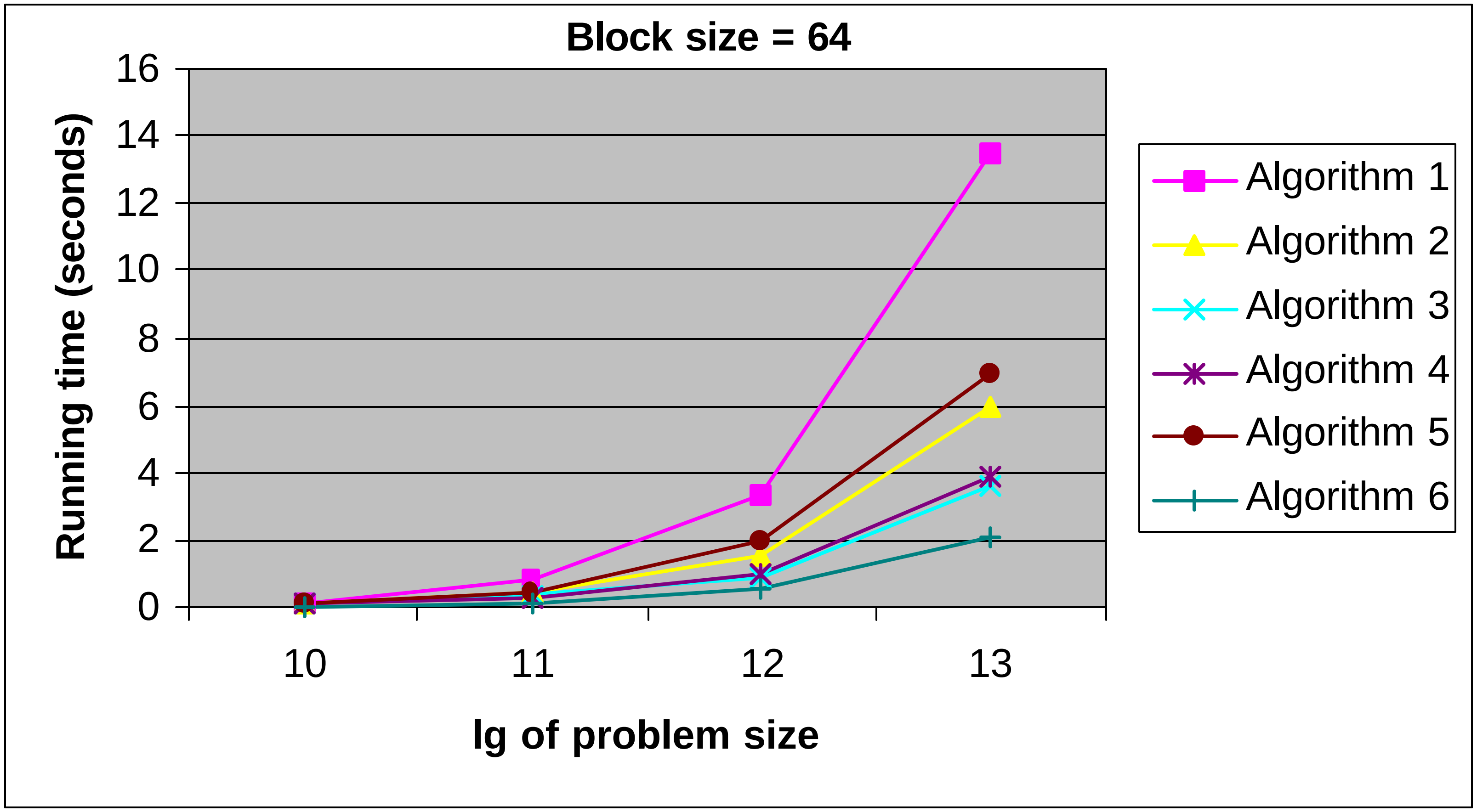
```
tr2(int src, int dst, int num)

{
   if (num==1) {
   /* base case */
   }
   else {
     tr2(NW(src),NW(dst),num/4);
     tr2(NE(src),SW(dst),num/4);
     tr2(SW(src),NE(dst),num/4);
     tr2(SE(src),SE(dst),num/4);
   }
```

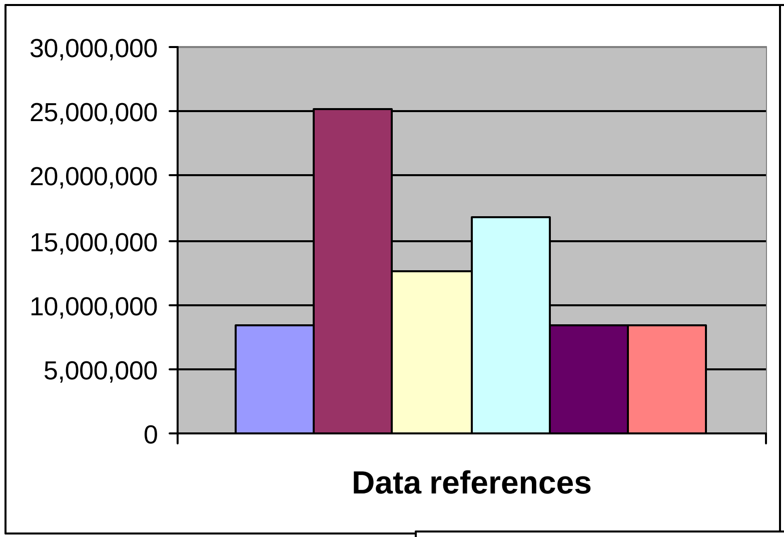
Experimental Platform

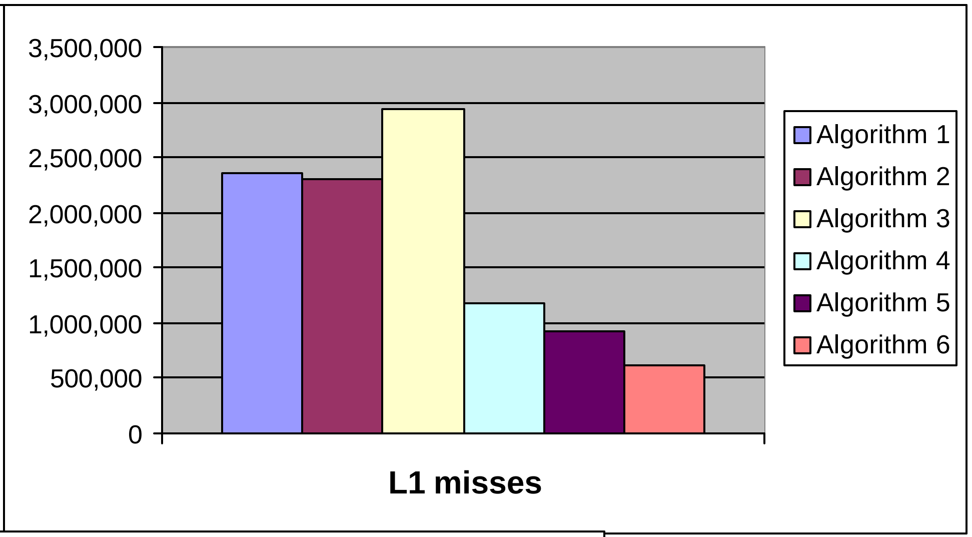
- 300 MHz UltraSPARC-II
- Memory architecture
  - L1 data cache: direct-mapped, 32-byte blocks, 16KB capacity
  - L2 data cache: direct-mapped, 64-byte blocks, 2MB capacity
  - RAM: 512MB
  - VM page size: 8KB
  - Data TLB: fully associative, 64 entries
- Operating system: SunOS 5.6
- Compiler: SUN’s Workshop Compilers 4.2

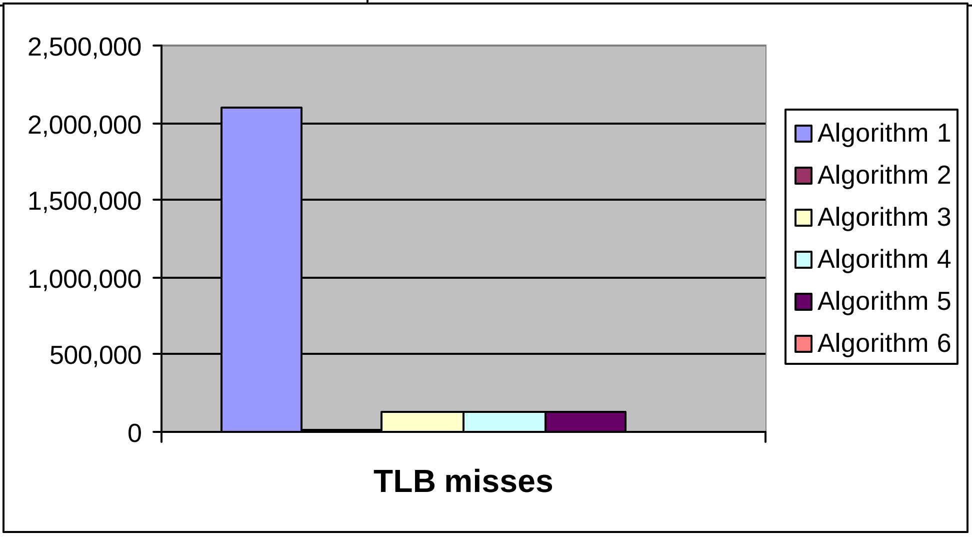
Comparative Performance of Algorithms



Memory System Behavior (n = 2048, b = 64)







Related Work

- Mathematics
  - Peano (1890), Hilbert (1891)
  - Lebesgue
- Libraries
  - PhiPACK (Berkeley)
  - ATLAS (Tennessee)
  - FFTW (MIT)
  - Matrix^{++} (UT Austin)
- Algorithms
  - Frens/Wise (Indiana)
  - I/O algorithms (Duke)
  - Leiserson (MIT)
  - Gustavson (IBM Research)
- Compilers
  - Iteration space tiling
  - Hierarchical tiling (UCSD)
  - Shackling (Cornell)
  - Lam/Rothberg/Wolf (Stanford)
  - Coleman/McKinley (UMass)
  - Rivera/Tseng (Maryland)
  - Cache Miss Equations (Princeton)
  - Cierniak/Li (Rochester)

Future Directions

- Application to more problems
  - Wavelet-based computations
  - JPEG 2000
- Analytical modeling of memory hierarchy behavior for non-linear layouts
  - We have a solution technique
    - Applied to recursive matrix multiplication
    - Also works for set-associative caches

http://www.cs.unc.edu/Research/TUNE/