6.035

Memory Optimization

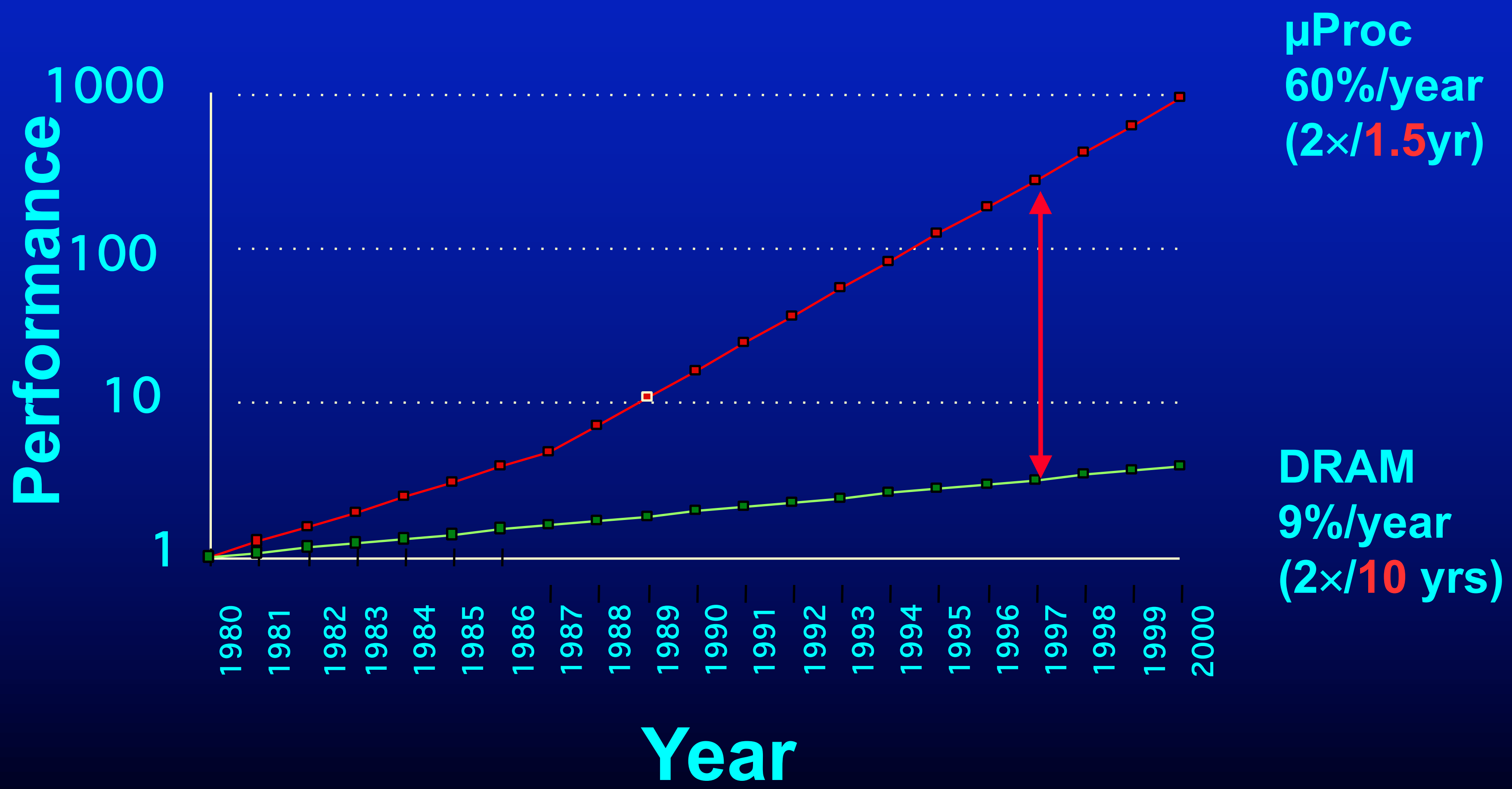
Outline

- Issues with the Memory System
- Loop Transformations
- Data Transformations
- Prefetching
- Alias Analysis

Memory Hierarchy

1 - 2 ns	Registers	32 – 512 B
3 - 10 ns	L1 Private Cache	16 – 128 KB
8 - 30 ns	L2/L3 Shared Cache	1 – 16 MB
60 - 250 ns	Main Memory (DRAM)	1 GB – 128 GB
5 - 20 ms	Permanent Storage (Hard Disk)	250 GB – 4 TB

Processor-Memory Gap



Cache Architecture

		Pentium D	Core Duo	Core 2 Duo	Athlon 64
L1 code (per core)	size	12 K uops	32 KB	32 KB	64 KB
	associativity	8 way	8 way	8 way	2 way
	Line size	64 bytes	64 bytes	64 bytes	64 bytes
L1 data (per core)	size	16 KB	32 KB	32 KB	64 KB
	associativity	8 way	8 way	8 way	8 way
	Line size	64 bytes	64 bytes	64 bytes	64 bytes
L1 to L2	Latency	4 cycles	3 cycles	3 cycles	3 cycles
L2 shared	size	4 MB	4 MB	4 MB	1 MB
	associativity	8 way	8 way	16 way	16 way
	Line size	64 bytes	64 bytes	64 bytes	64 bytes
L2 to L3(off)	Latency	31 cycles	14 cycles	14 cycles	20 cycles

Cache Misses

- Cold misses
  - First time a data is accessed
- Capacity misses
  - Data got evicted between accesses because a lot of other data (more than the cache size) was accessed
- Conflict misses
  - Data got evicted because a subsequent access fell on the same cache line (due to associativity)
- True sharing misses (multicores)
  - Another processor accessed the data between the accesses
- False sharing misses (multicores)
  - Another processor accessed different data in the same cache line between the accesses

Data Reuse

- Temporal Reuse
  - A given reference accesses the same location in multiple iterations

```
for i = 0 to N
  for j = 0 to N
    A[j] =
```

- Spatial Reuse
  - Accesses to different locations within the same cache line

```
for i = 0 to N
  for j = 0 to N
    B[i, j] =
```

- Group Reuse
  - Multiple references access the same location

```
for i = 0 to N
    A[i] = A[i-1] + 1
```

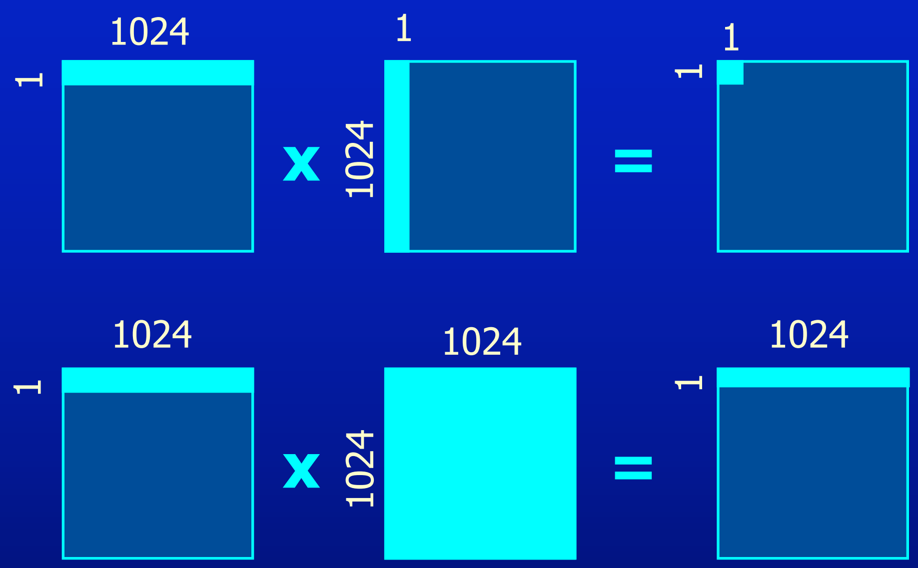
Outline

- Issues with the Memory System
- **Loop Transformations**
- Data Transformations
- Prefetching
- Alias Analysis

Matrix Multiply

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            c[i,j] += a[i,k]*b[k,j]
```

Example: Matrix Multiply



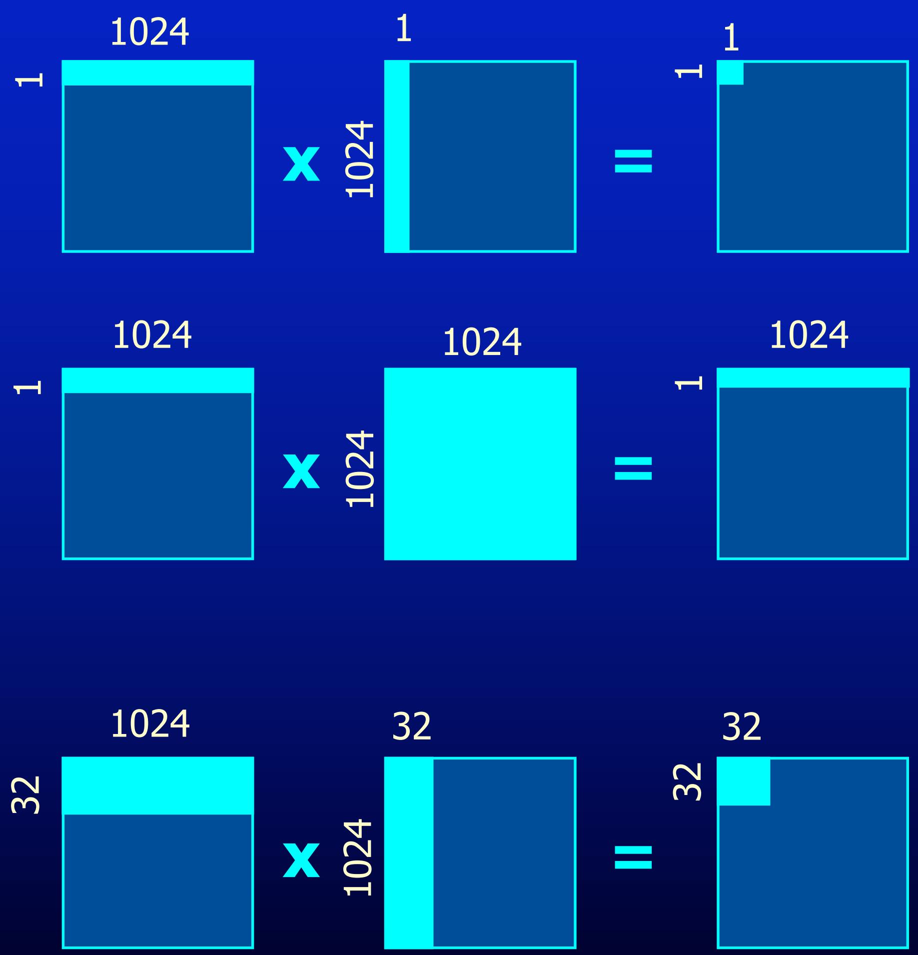
Data Accessed

1,050,624

Matrix Multiply

```
for i0 = 1 to n step b
    for j0 = 1 to n step b
        for k0 = 1 to n step b
            for 1 = i0 to min(i0+b-1, n)
                for j = j0 to min(j0+b-1, n)
                    for k = k0 to min(k0+b-1, n)
                        c[i,j] += a[i,k]*b[k,j]
```

Example: Matrix Multiply



Data Accessed

1,050,624

66,560

Loop Transformations

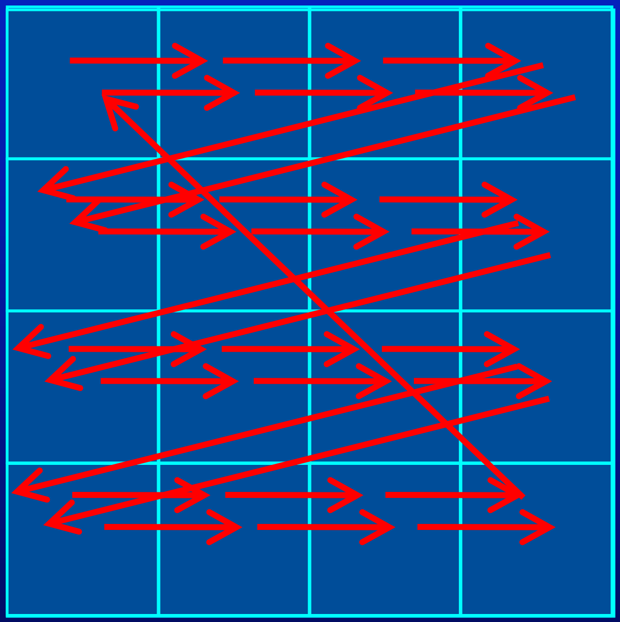
- Transform the iteration space to reduce the number of misses
- Reuse distance – For a given access, number of other data items accessed before that data is accessed again
- Reuse distance > cache size
  - Data is spilled between accesses

Divide and Conquer Matrix Multiply

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

Loop Transformations

```
for i = 0 to N
    for j = 0 to N
        for k = 0 to N
            A[k,j]
```



Reuse distance = N^2

If Cache size < 16 doubles?

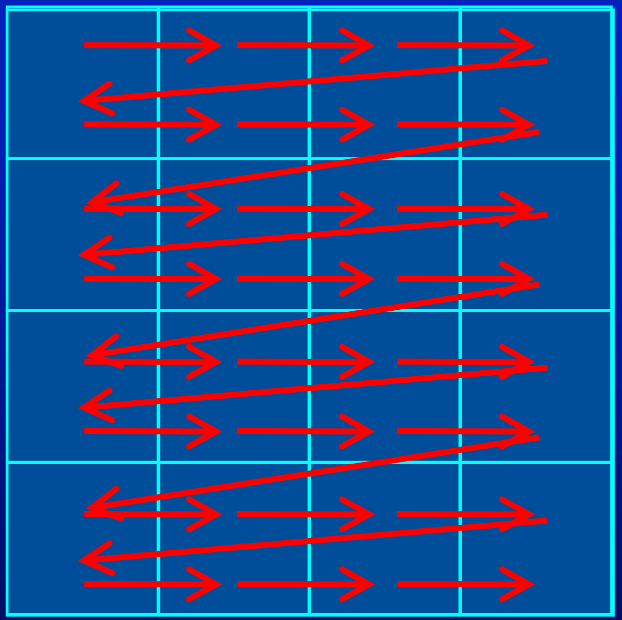
A lot of capacity misses

Loop Transformations

```
for i = 0 to N
    for j = 0 to N
        for k = 0 to N
            A[k,j]
```

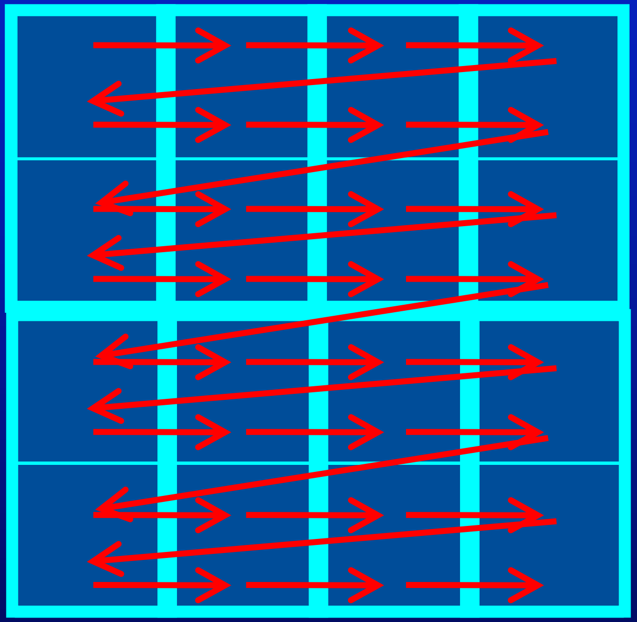
Loop Interchange

```
for j = 0 to N
    for i = 0 to N
        for k = 0 to N
            A[k,j]
```



Loop Transformations

for j = 0 to N
    for i = 0 to N
        for k = 0 to N
            A[k,j]



Cache line size > data size
Cache line size = L
Reuse distance = LN

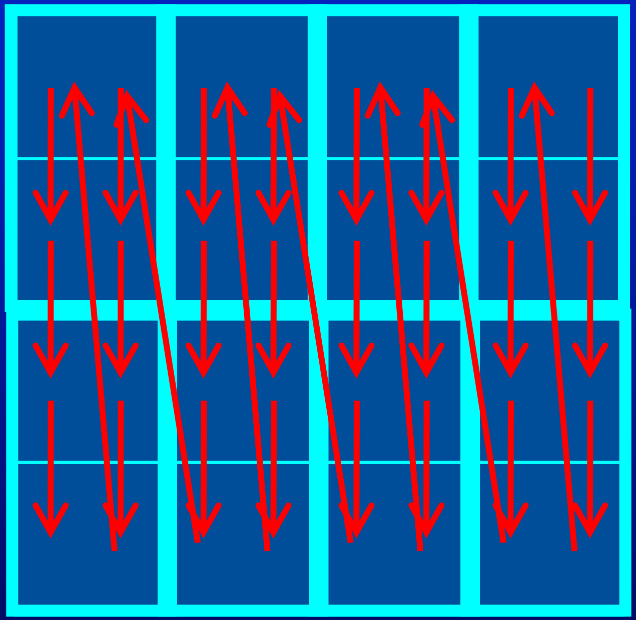
If cache size < 8 doubles?
    Again a lot of capacity misses

Loop Transformations

```
for j = 0 to N
    for i = 0 to N
        for k = 0 to N
            A[k,j]
```

Loop Interchange

```
for k = 0 to N
    for i = 0 to N
        for j = 0 to N
            A[k,j]
```



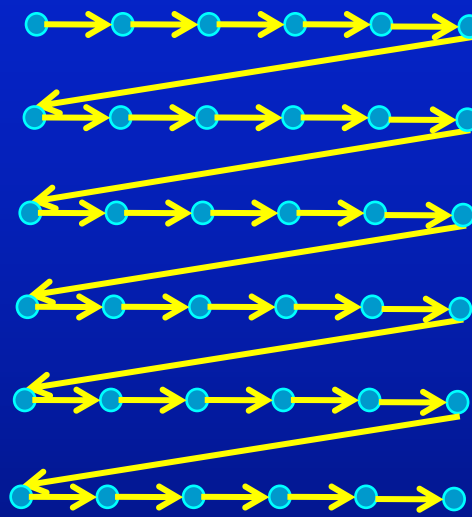
Loop Transformations

```
for i = 0 to N
    for j = 0 to N
        for k = 0 to N
            A[i,j]= A[i,j]+ B[i,k]+ C[k,j]
```

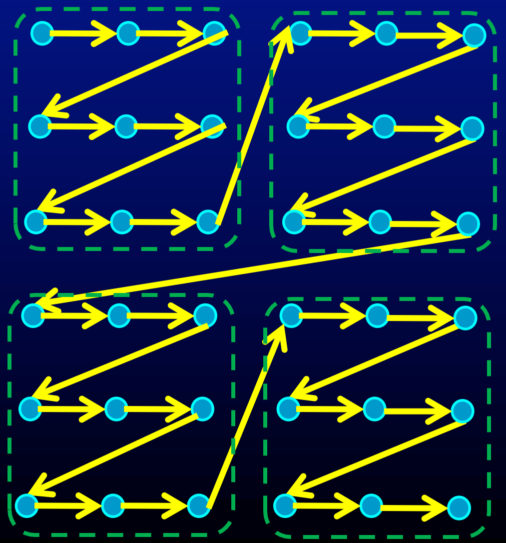
- No matter what loop transformation you do one array access has to traverse the full array multiple times

Loop Tiling

for i = 0 to N
    for j = 0 to N



for ii = 0 to ceil(N/b)
  for jj = 0 to ceil(N/b)
    for i = b*ii to min(b*ii+b-1, N)
      for j = b*jj to min(b*jj+b-1, N)

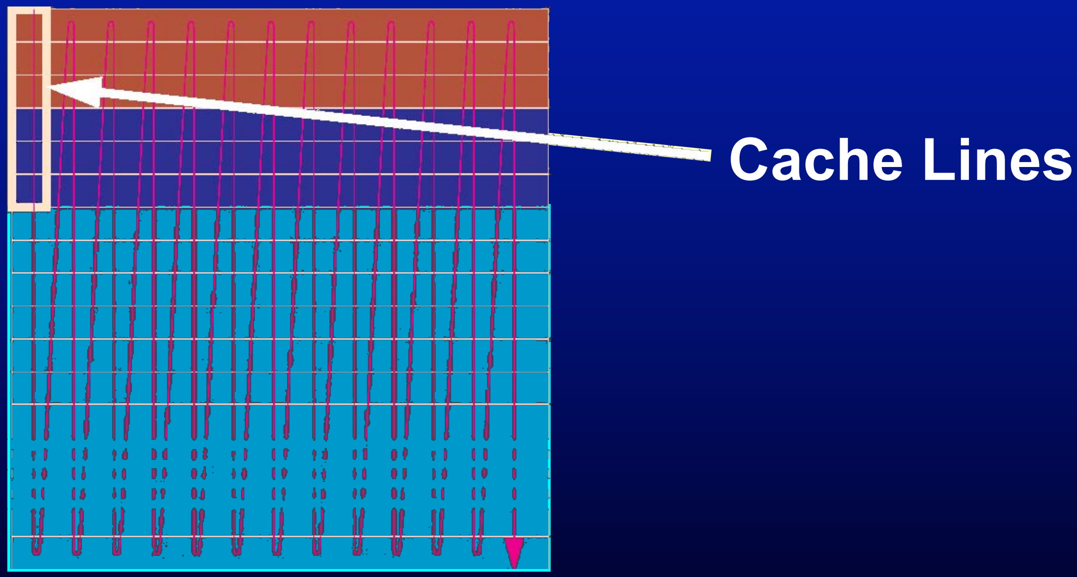


Outline

- Issues with the Memory System
- Loop Transformations
- **Data Transformations**
- Prefetching
- Alias Analysis

False Sharing Misses

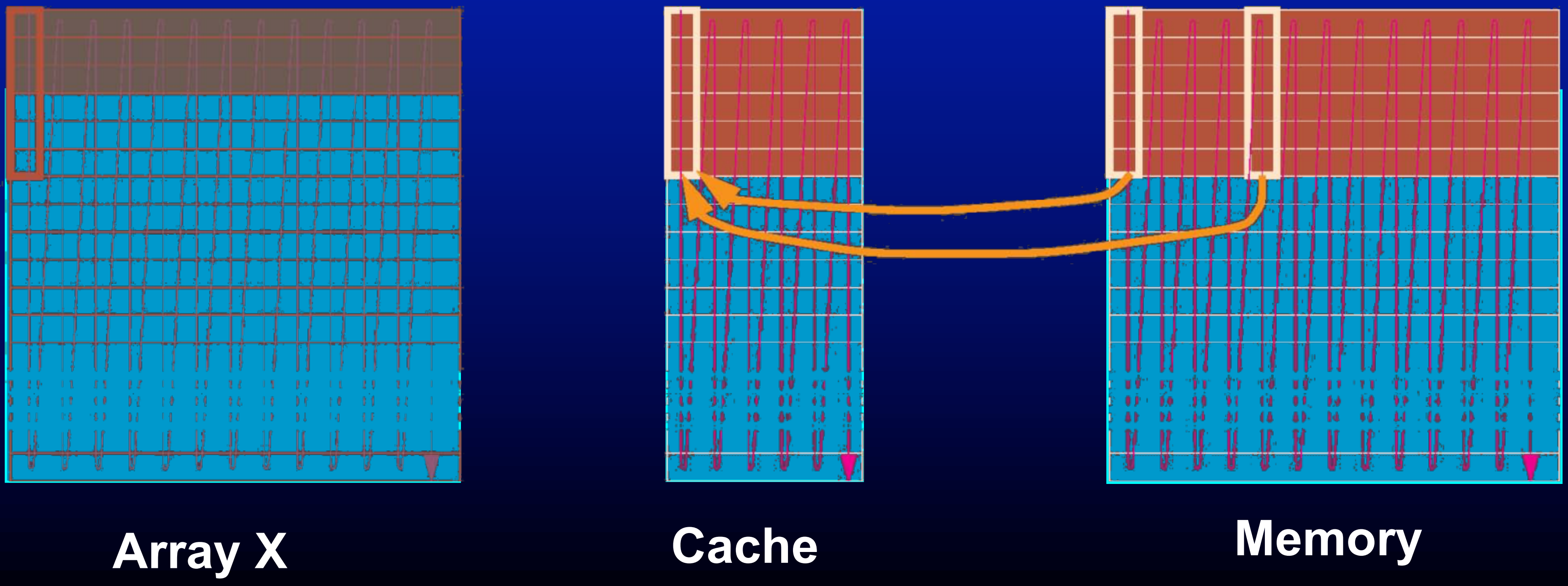
```
for J =
    forall I =
        X(I, J) = ...
```



Array X

Conflict Misses

```
for J =
    forall I =
        X(I, J) = …
```



Data Transformations

- Similar to loop transformations
- All the accesses have to be updated
  - Whole program analysis is required

Strip-Mining

Create two dims from one

With blocksize=4

Storage Declaration

$$[N] \rightarrow \begin{bmatrix} 4 \\ \left\lceil \dfrac{N}{4} \right\rceil \end{bmatrix}$$

Array Access

$$[i] \rightarrow \begin{bmatrix} i \bmod 4 \\ i/4 \end{bmatrix}$$

Memory Layout

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	1	2	3	4	5	6	7

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0,0	1,0	2,0	3,0	0,1	1,1	2,1	3,1

	Strip-Minding Create two dims from one	**Permutation** Change memory layout
	With blocksize=4	With permutation matrix $\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$
Storage Declaration	$\left[N\right] \rightarrow \left[\begin{matrix}4\\ \left\lceil \frac{N}{4} \right\rceil\end{matrix}\right]$	$\begin{bmatrix}N_1\\ N_2\end{bmatrix} \rightarrow \begin{bmatrix}N_2\\ N_1\end{bmatrix}$
Array Access	$\left[i\right] \rightarrow \left[\begin{matrix}i \bmod 4\\ i/4\end{matrix}\right]$	$\begin{bmatrix}i_1\\ i_2\end{bmatrix} \rightarrow \begin{bmatrix}i_2\\ i_1\end{bmatrix}$
Memory Layout	see below	see below

Memory Layout — Strip-Minding

0 0	*1* 1	*2* 2	*3* 3	*4* 4	*5* 5	*6* 6	*7* 7

0 0,0	*1* 1,0	*2* 2,0	*3* 3,0	*4* 0,1	*5* 1,1	*6* 2,1	*7* 3,1

Memory Layout — Permutation

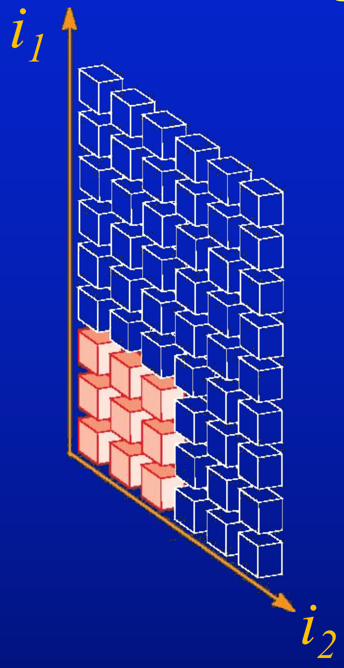
0 0,0	*4* 0,1	*8* 0,2
1 1,0	*5* 1,1	*9* 1,2
2 2,0	*6* 2,1	*10* 2,2
3 3,0	*7* 3,1	*11* 3,2

0 0,0	*1* 1,0	*2* 2,0
3 0,1	*4* 1,1	*5* 2,1
6 0,2	*7* 1,2	*8* 2,2
9 0,3	*10* 1,3	*11* 2,3

Data Transformation Algorithm

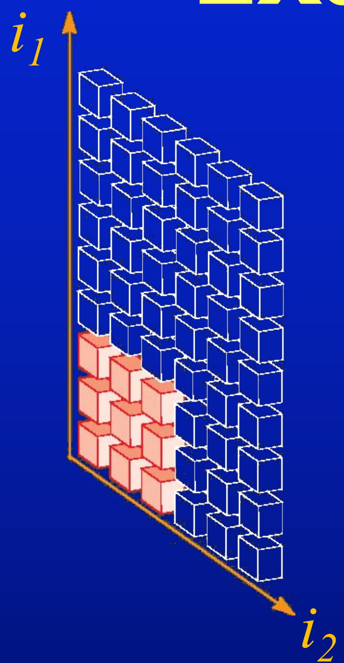
- Rearrange data: Each processor's data is contiguous
- Use data decomposition
  - *, block, cyclic, block-cyclic
- Transform each dimension according to the decomposition
- Use a combination of strip-mining and permutation primitives

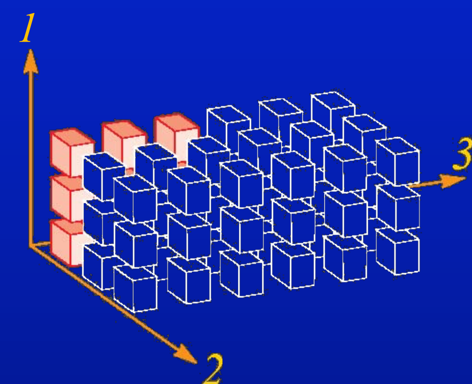
Example I: (Block, Block)



$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix}$$

Example I: (Block, Block)

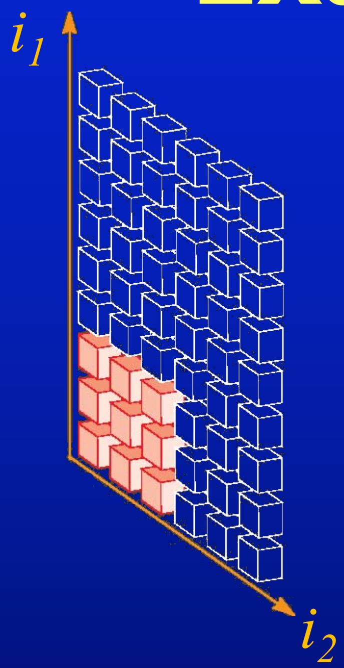


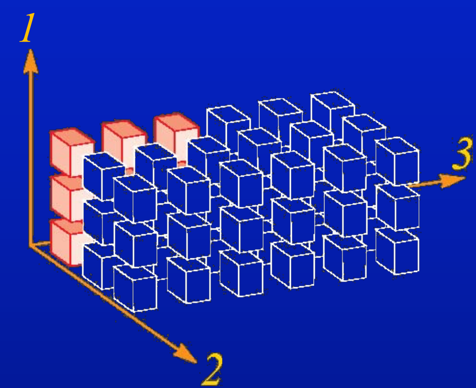


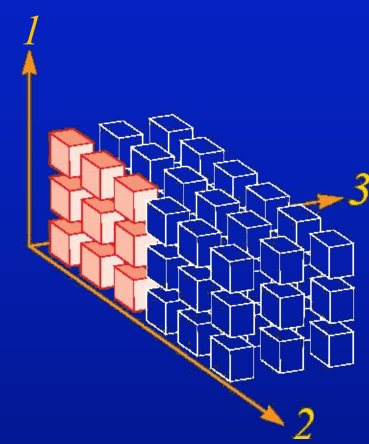
Strip-Mine

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \rightarrow \begin{pmatrix} i_1 \bmod \left\lceil \frac{d_1}{P} \right\rceil \\ i_1 / \left\lceil \frac{d_1}{P} \right\rceil \\ i_2 \end{pmatrix}$$

Example I: (Block, Block)

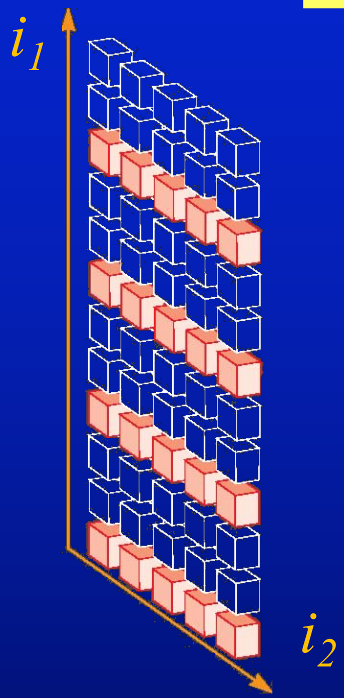






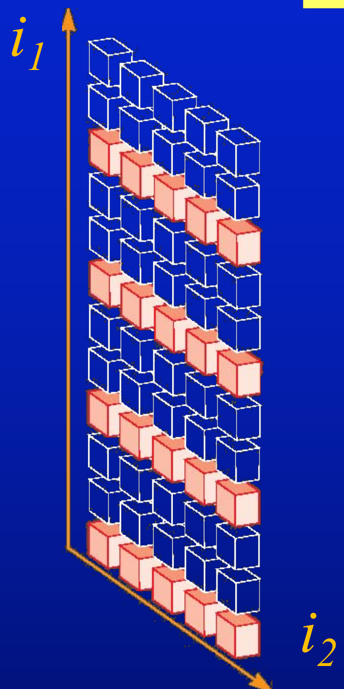
$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \xrightarrow{\text{Strip-Mine}} \begin{pmatrix} i_1 \bmod \left\lceil \frac{d_1}{P} \right\rceil \\ i_1 / \left\lceil \frac{d_1}{P} \right\rceil \\ i_2 \end{pmatrix} \xrightarrow{\text{Permute}} \begin{pmatrix} i_1 \bmod \left\lceil \frac{d_1}{P} \right\rceil \\ i_2 \\ i_1 / \left\lceil \frac{d_1}{P} \right\rceil \end{pmatrix}$$

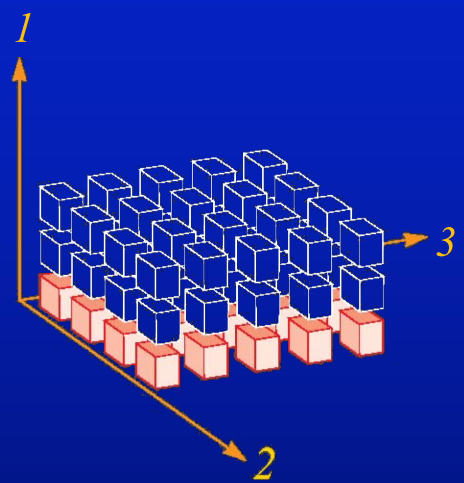
Example I: (Cyclic, *)



$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix}$$

Example I: (Cyclic, *)

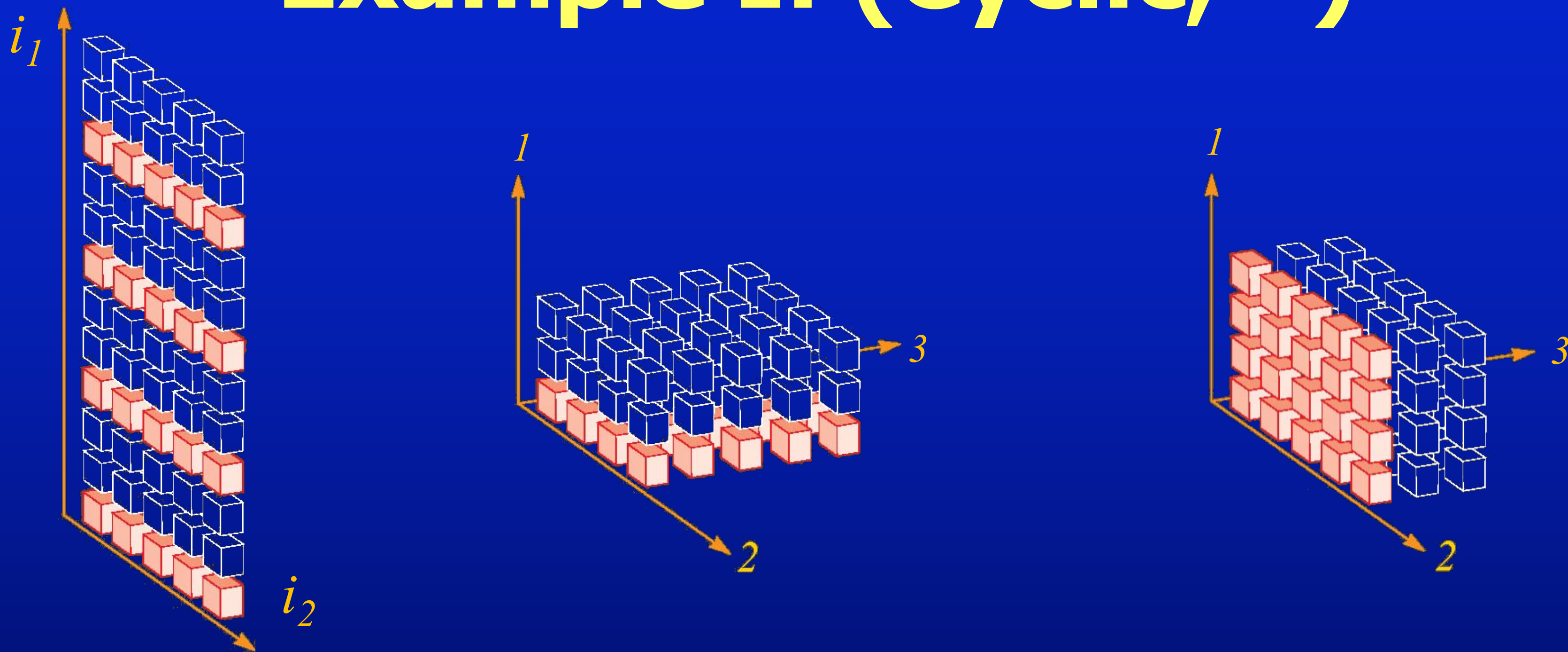




Strip-Mine

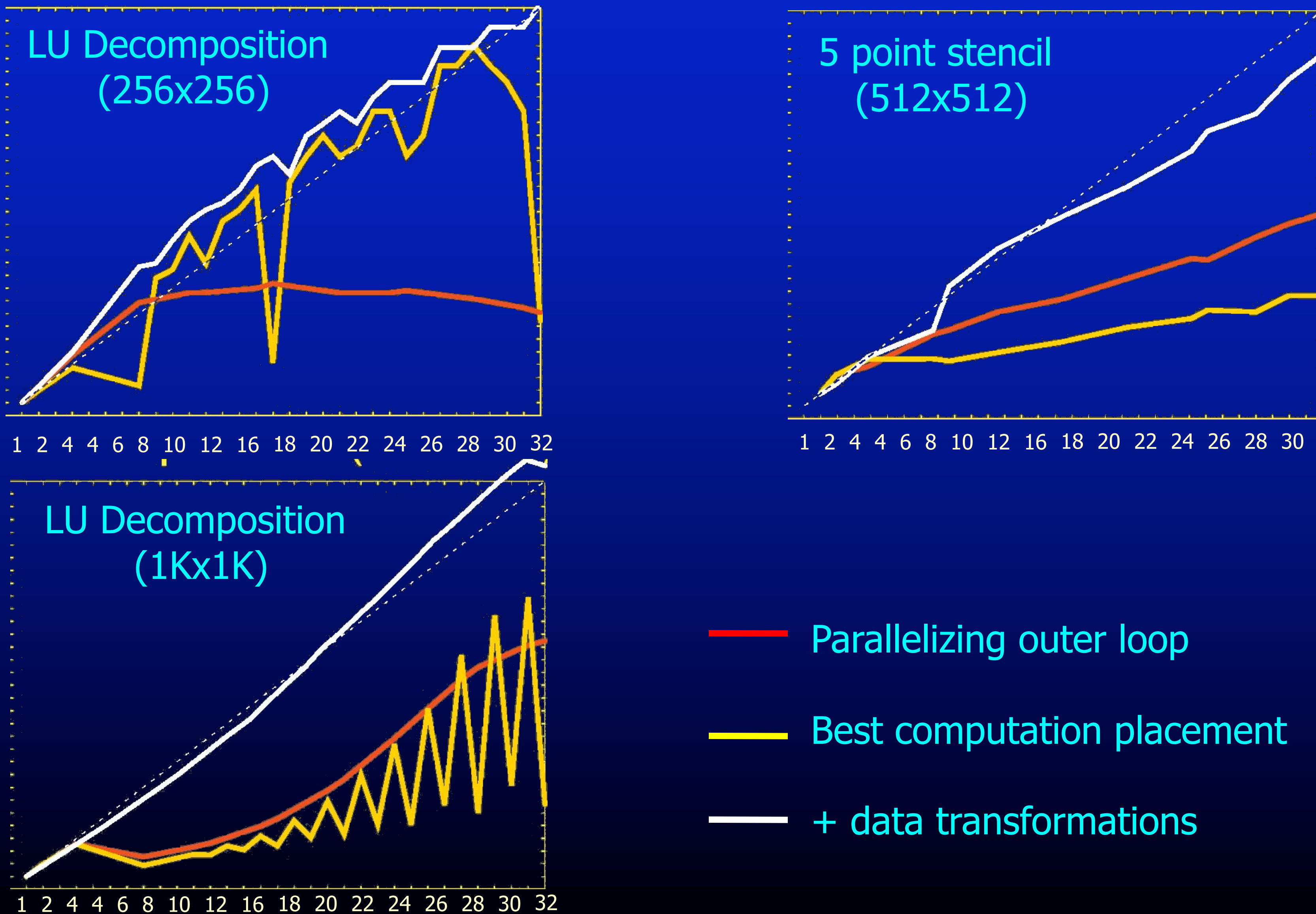
$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \bmod P \\ i_1 / P \\ i_2 \end{bmatrix}$$

Example I: (Cyclic, *)



$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \xrightarrow{\text{Strip-Mine}} \begin{bmatrix} i_1 \bmod P \\ i_1 / P \\ i_2 \end{bmatrix} \xrightarrow{\text{Permute}} \begin{bmatrix} i_1 \bmod P \\ i_2 \\ i_1 / P \end{bmatrix}$$

Performance



Optimizations

- Modulo and division operations in the index calculation
  - Very high overhead

- Use standard techniques
  - Loop invariant removal, CSE
  - Strength reduction exploiting properties of modulo and division
  - Use knowledge about the program

Outline

- Issues with the Memory System
- Loop Transformations
- Data Transformations
- **Prefetching**
- Alias Analysis

Prefetching

- Cache miss stalls the processor for hundreds of cycles
  - Start fetching the data early so it'll be available when needed

- Pros
  - Reduction of cache misses → increased performance
- Cons
  - Prefetch contents for fetch bandwidth
    - Solution: Hardware only issue prefetches on unused bandwidth
  - Evicts a data item that may be used
    - Solution: Don't prefetch too early
  - Pretech is still pending when the memory is accessed
    - Solution: Don't prefetch too late
  - Prefetch data is never used
    - Solution: Prefetch only data guaranteed to be used
  - Too many prefetch instructions
    - Prefetch only if access is going to miss in the cache

Prefetching

- Compiler inserted
  - Use reuse analysis to identify misses
  - Partition the program and insert prefetches
- Run ahead thread (helper threads)
  - Create a separate thread that runs ahead of the main thread
  - Runahead only does computation needed for control-flow and address calculations
  - Runahead performs data (pre)fetches

Outline

- Issues with the Memory System
- Loop Transformations
- Data Transformations
- Prefetching
- **Alias Analysis**

Alias Analysis

- Aliases destroy local reasoning
  - Simple, local transformations require global reasoning in the presence of aliases
  - A critical issue in pointer-heavy code
  - This problem is even worse for multithreaded programs

- Two solutions
  - Alias analysis
    - Tools to tell us the potential aliases
  - Change the programming language
    - Languages have no facilities for talking about aliases
    - Want to make local reasoning possible

Aliases

- Definition

Two pointers that point to the same location are ***aliases***

- Example

```
Y = &Z
X = Y
*X = 3     /* changes the value of *Y */
```

Example

```
foo(int * A, int * B, int * C, int N)
    for i = 0 to N-1
        A[i]= A[i]+ B[i] + C[i]
```

- Is this loop parallel?

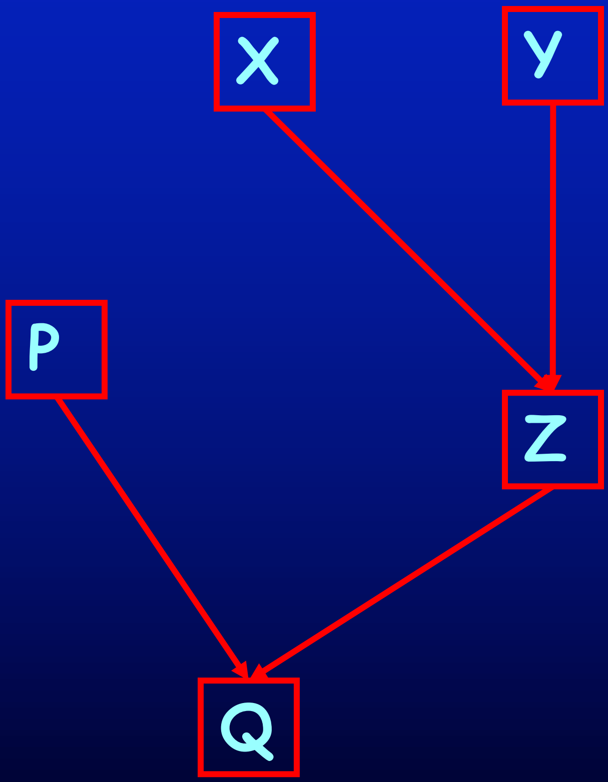
- Depends

```
int X[1000];
int Y[1000];
int Z[1000]
foo(X, Y, Z, 1000);
```

```
int X[1000];
foo(&X[1], &X[0], &X[2], 998);
```

Points-To Analysis

- Consider:
  - P = &Q
  - Y = &Z
  - X = Y
  - *X = P
- Informally:
  - P can point to Q
  - Y can point to Z
  - X can point to Z
  - Z can point to Q



Points-To Relations

- A graph
  - Nodes are program *names*
  - Edge (x,y) says x may point to y
- Finite set of names
  - Implies each name represents many heap cells
  - Correctness: If *x = y in any state of any execution, then (x,y) is an edge in the points-to graph

Sensitivity

- *Context sensitivity*
  - Separate different uses of functions
  - Different is the key – if the analysis think the input is the same, reuse the old results
- *Flow sensitivity*
  - For insensitivity makes any permutation of program statements gives same result
  - Flow sensitive is similar to data-flow analysis

Conclusion

- Memory systems are designed to give a huge performance boost for “normal” operations
- The performance gap between good and bad memory usage is huge
- Programs analyses and transformations are needed
- Can off-load this task to the compiler