CS420 – Lecture 4

Marc Snir

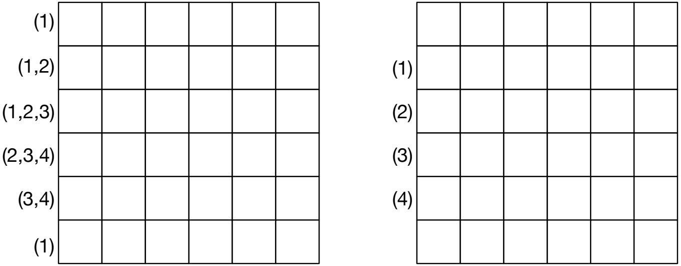
Fall 2018

Jacobi

```
double a[2][M][N];
...

while (!converged){
 for (i=1;i<N-1; i++)
    for(j=1;j<M-1;j++)
       a[1-k][i][j]=0.25*(a[k][i-1][j]
           +a[k][i+1][j]+a[k][i][j-1]
           +a[k][i][j+1]);
k = 1-k;
}
```

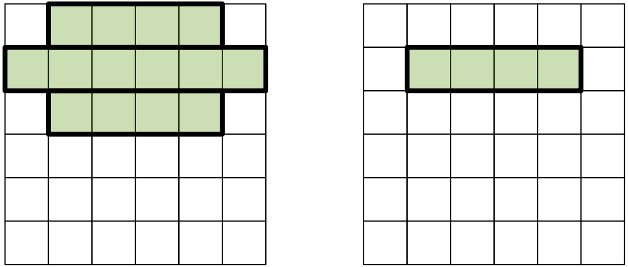
Assume cache can hold more than 4 rows

What row is accessed at each iteration of `i`



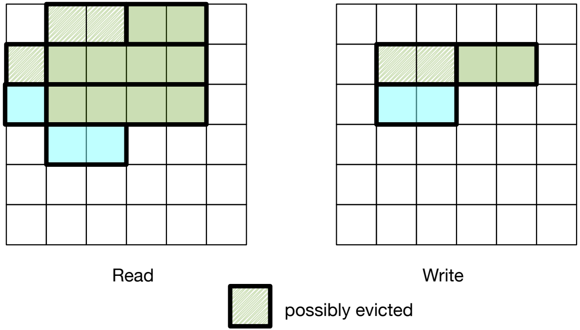
Read Write

Cache content after 1st iteration

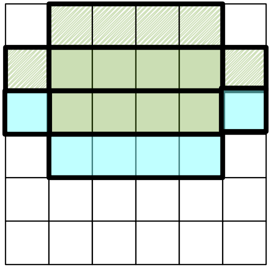


Read Write

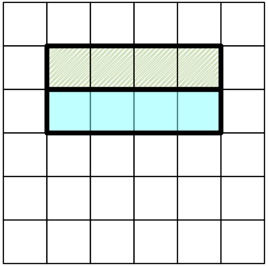
Cache content middle of 2nd iteration



Cache content after 2nd iteration

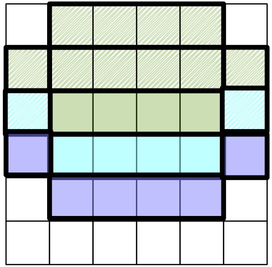


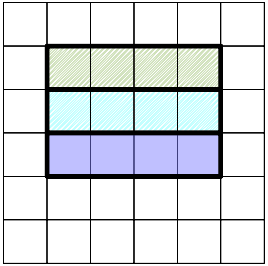
Read



Write

Cache content after 3rd iteration

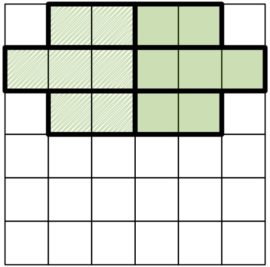

Read


Write

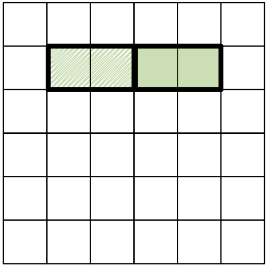
- If cache can hold more than 4 rows than each line is accessed once
- Number of misses is $\sim 2N^2/8 = N^2/4$
- Assume ideal cache – actual performance may vary

Cache cannot hold 4 rows

Cache content middle of 1st iteration



Read



Write

- Each row of left matrix is read from memory again at each iteration.
- $\sim 3N^2/8$ misses on left matrix
- $N^2/8$ misses on right matrix
- Total of $\sim N^2/2$ misses

Case study: transpose

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

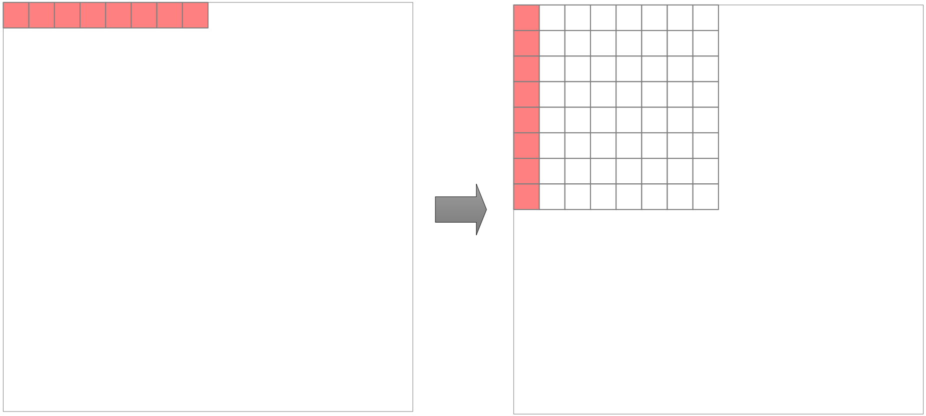
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

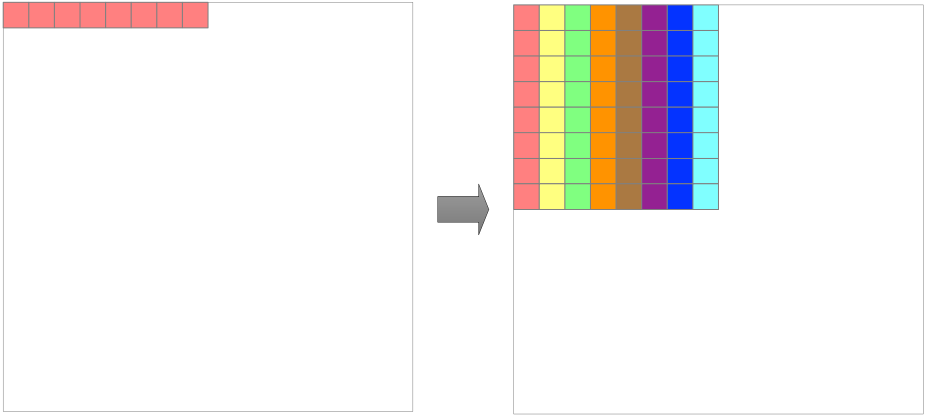
b

- If `a[]` is traversed by rows, then `b[]` is traversed by columns; and vice-versa!
- If matrix is large, have $\sim N^2/16 + N^2 = \frac{17}{16}N^2$ cache misses (assuming 16 words per cache line)

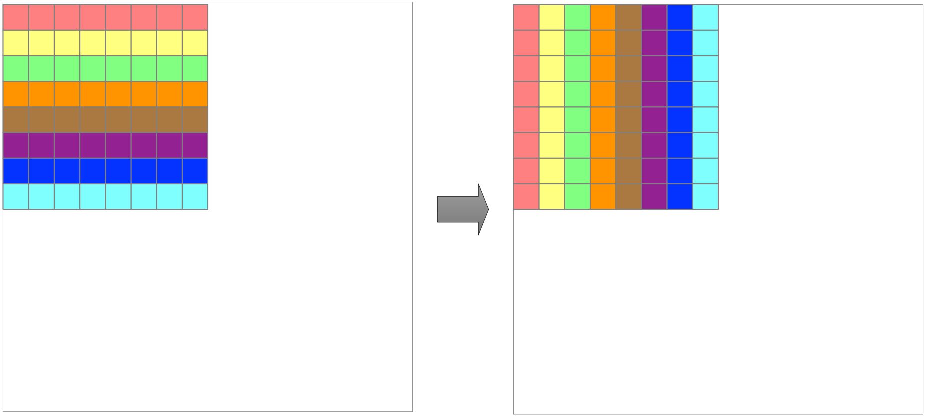


Read cache line (good), write 16 cache lines (bad)

Should fill up the remainder of the 16 cache lines soon after filling the first column

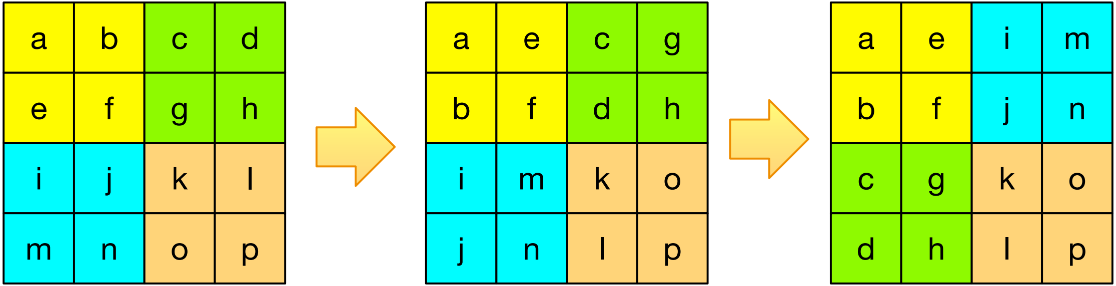


Should read the first cache lines in the next 15 rows soon after reading the first cache line in the first row the first column



Better to read & write larger tile, as long as it fits in cache (prefetch)

In effect, tiles are read to cache, transposed within cache and written back to destination tile



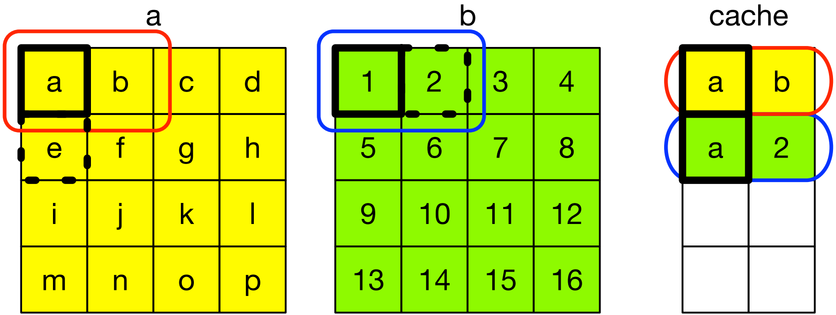
Tiled transpose

```
// L divides N
for(l=0; l<N; l+=L)
 for(m=0;m<N;m+=L)
  for(i=l;i<l+L; i++)
   for(j=m;j<m+L;j++)
    b[i][j] = a[j][i];
```

- 2 outer loops iterate over 2D tiles
- 2 inner loops permute a tile and store it to its final location (one tile read and one tile written)
- Choose tile size so that 2 tiles fit in cache
- Have $\sim 2N^2/16 = N^2/8$ cache misses (close to $\times 8$ improvement)

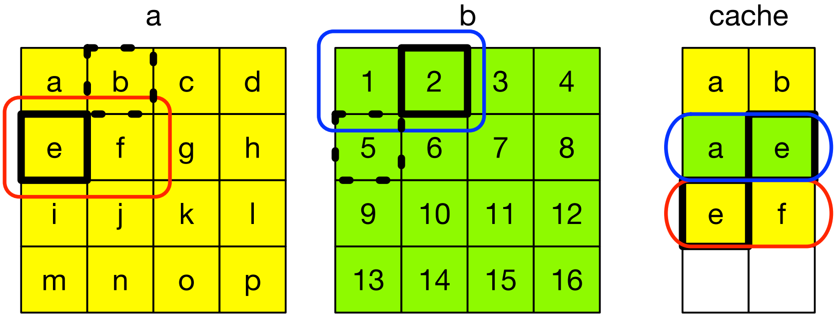
Transpose illustrated, assuming two words per cache line, and 4 lines per cache

Transpose illustrated 1



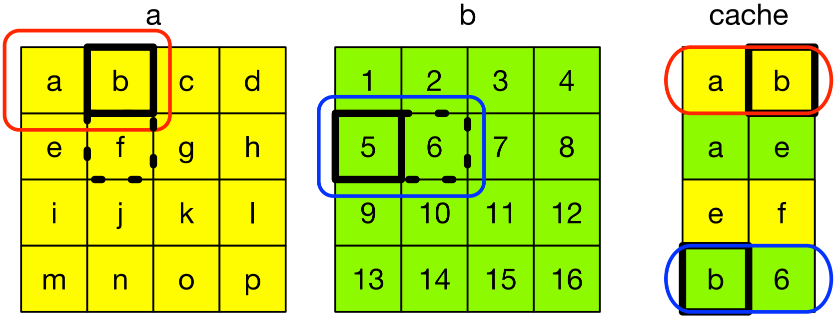
b[0][0]=a[0][0]

Transpose illustrated 2



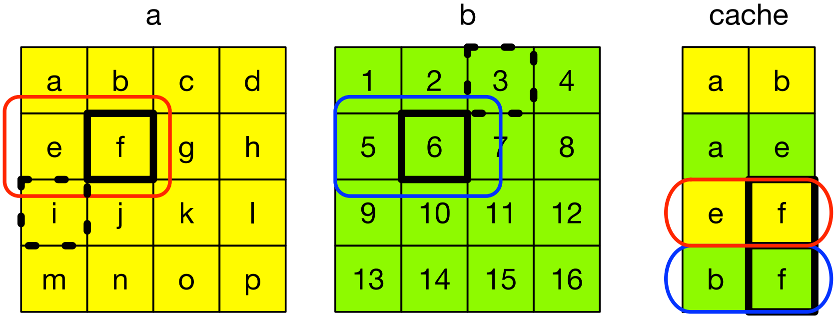
b[0][1]=a[1][0]

Transpose illustrated 3



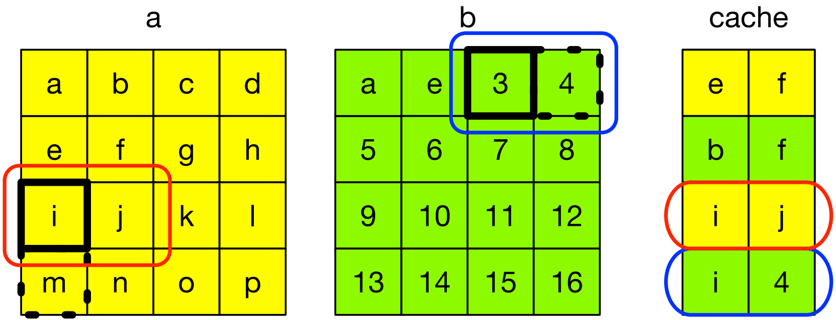
b[1][0]=a[0][1]

Transpose illustrated 4



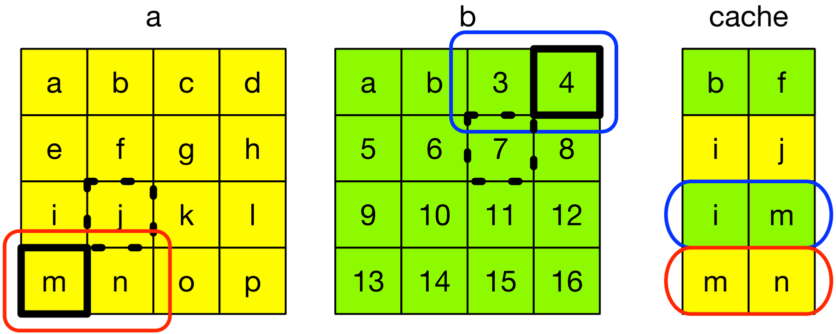
b[1][1]=a[1][1]

Transpose illustrated 5



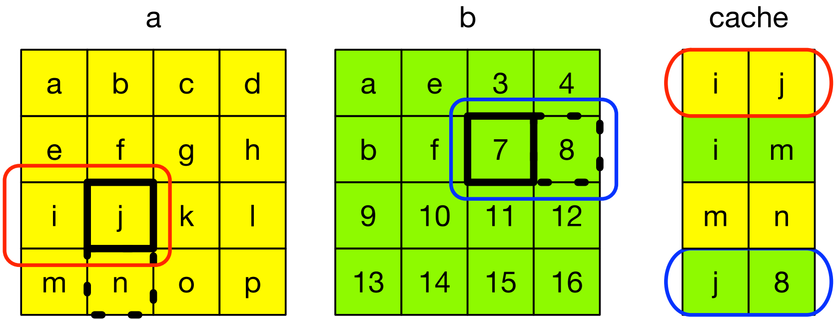
b[2][0]=a[0][2]

Transpose illustrated 6



b[3][0]=a[0][3]

Transpose illustrated 7



b[3][1]=a[1][3]

Experiment

- Transpose of $10{,}000 \times 10{,}000$ matrix, naive code: running time $= 811\text{ms} \pm 17$
- Transpose of $10{,}000 \times 10{,}000$ matrix, tiled code, tile size $= 50$: running time $= 182 \pm 10$
- $\sim \times 15$ improvement – more than expected.
- Transpose of $10{,}000 \times 10{,}000$ matrix. with two levels of tiling: $LL = 250$, $L = 50$: running time $= 162 \pm 2.8$.

Reuse distance – measure of temporal locality

- *Reuse distance* for an access to cache line v: How long since v was last accessed.
- More precisely: how many different lines were accessed since last access to v.

Theorem

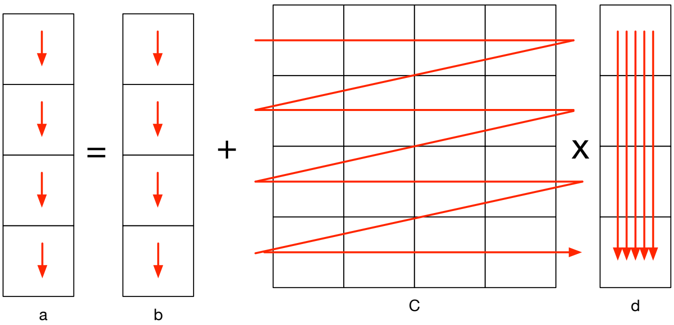
If the cache is fully associative then the access to υ is a miss if and only if the reuse distance of υ is larger than the cache size

More convenient to discuss reuse distance for variable accesses (rather than lines).

Matrix-vector

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

$a = b + C \times d$



Compiler will optimize and keep `temp` in register

```
for(i=0;i<M; i++) {
 temp=b[i];
   for(j=0;j<N; j++)
     temp+=C[i][j]*d[j];
   a[i]=tmp;
 }
```

Compiler will generate Floading Multiply-Add (FMA) operations

Reuse distance

```
for(i=0;i<M; i++) {
  temp=b[i];
  for(j=0;j<N; j++)
      temp +=C[i][j]*d[j];
 a[i]=temp
}
```

- Approximate analysis or reuse distance, ignoring first access
- Focus on inner loop
- `a[i]`, `b[i]` and `C[i][j]` are not reused;
- Each `temp` is accessed $\sim MN$ times with constant reuse distances ($\Theta(1)$)
- Each `d[j]` is accessed M times with reuse distance $\Theta(N)$ (order N)

Tiling improves reuse distance for accesses to d[][]

```
\\ T tile size divides N

for(i=0;i<M;i++)
 a[i]=b[i];
 for(jj=0;jj<N;jj+=T)
  for(i=0;i<M;i++) {
   temp=a[i];
   for(j=jj;j<jj+T;j++)
    temp+=C[i][j]*d[j];
   a[i]=temp;
   }
```

- Consider reuse distance in inner loop

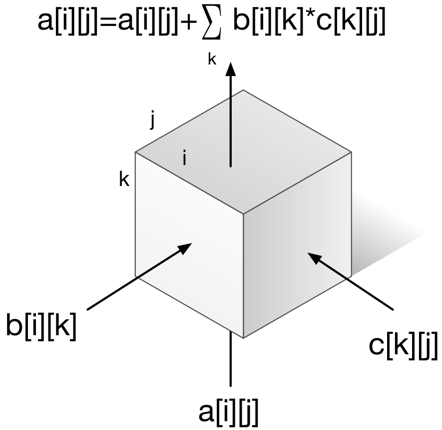
- b[i] is not accessed and C[i][j] is not reused
- temp is accessed $\sim MN$ times with constant reuse distance (and MN/T times with $\Theta(T)$ reuse distance
- d[j] is accessed M times with reuse distance $\Theta(T)$
- Pick T as large as possible while avoiding misses on d[]
- No more capacity misses on d[] ($\Theta(NM)$ gain); more misses on temp and on a[] ($\Theta(NM/T)$ loss). Worthwhile, but gain is small (if at all)

Matrix-matrix

$a = a + b \times c$ – all matrices are $N \times N$

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

- $2N^3$ operations (N^3 FMAs) and only $3N^2$ variables, so good opportunity to achieve higher compute intensity
- Can compute FMAs in any order; which is best?

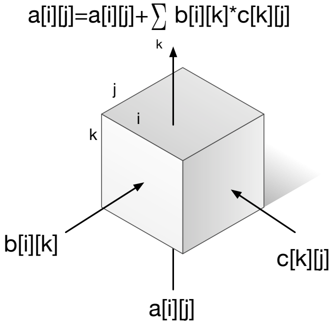


Cube of FMAs

No tiling – We ignore spatial locality in analysis

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

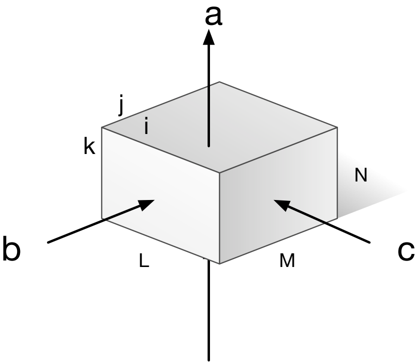
- Each entry of arrray `a`, `b`, `c` is accessed N times.
- `a[i][j]` has mostly reuse distance $\Theta(1)$
- `b[i][k]` has mostly reuse distance $\Theta(N)$
- `c[k][j]` has reuse distance $\Theta(N^2)$
- If cache can hold all of `c`, a column of `b` and an entry of `a` will have $\Theta(N^2)$ misses
- if $N^2 > C$ (cache size), all N^3 accesses to `c` are misses.



What if not cube?

```
for(i=0;i<L;i++)
    for(j=0;j<M;j++)
        for(k=0;k<N;k++)
            a[i][j]=a[i][j]+b[i][k]*c[k][
```

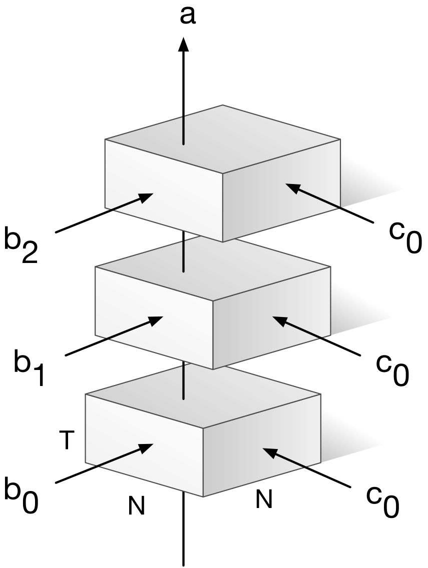
- At least $LM + MN + NL$ cache misses
- Reuse distance 1 for `a[i][j]`, N for `b[i][k]` and MN for `c[k][j]`
- If $C > MN + N + 1$ then have $\Theta(LM + MN + NL)$ misses (optimal)
- Should reorder loops in decreasing range size
- If $C \gg \min(LM, MN, NL)$ then have $\Theta(LM + MN + NL)$ misses



Reduce reuse distance for c [] [] – Tile inner loop

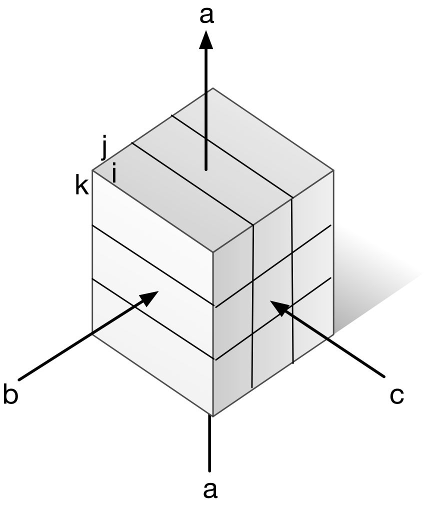
```
\\ T divides N
for(kk=0;kk<N;kk+=T)
 for(i=0;i<N;i++)
  for(j=0;j<N;j++)
   for(k=kk;k<(kk+T;k++)
    a[i][j]=a[i][j]+b[i][k]*c[k][j]
```

- Each iteration of outer loop one computes product of $N \times T$ and $T \times N$ matrices
- $\Rightarrow$ if $C \gg NT$ then have $N^2 + 2NT$ misses for each product, for a total of $\frac{N}{T}(N^2 + 2NT)$
- Pick $T \sim C/N$ so number of misses is $\sim \frac{N}{T}(N^2 + 2NT) = \frac{N^2}{C}(N^2 + 2C) = \frac{N^4}{C} + 2N^2$
- If $N > C$ we still have N^3 misses.



Tile b and c

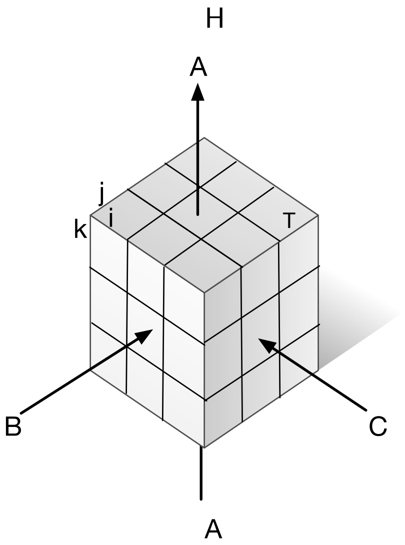
```
for(jj=0;jj<N;jj+=T)
 for(kk=0;kk<N;kk+=T)
  for(i=0;i<N;i++)
   for(j=jj;j<jj+T;j++)
    for(k=kk;k<kk+T;k++)
     a[i][j]=a[i][j]+b[i][k]*c[k][j]
```



- Three inner loops compute product of $N \times T$ by $T \times T$ matrices; there are $(\frac{N}{T})^2$ such products.
- If $C >> T^2$ then have $\sim 2NT + T^2$ cache misses for each product, for a total of $(\frac{N}{T})^2(2NT + T^2)$
- Pick $T \sim \sqrt{C}$ so number of misses is $\sim (\frac{N}{T})^2(2NT + T^2) = \frac{N^2}{C}(2N\sqrt{C} + C) = \frac{2N^3}{\sqrt{C}} + N^2$

Tile a, b and c

- Compute $(\frac{N}{T})^3$ products of $T \times T$ matrices.
- If $C \sim T^2$ then have $\sim 3(\frac{N}{T})^3 T^2 = \frac{3N^3}{T} = \frac{3N^3}{\sqrt{C}}$ cache misses.



Experiment

N=1024, T=32 (L1 cache 32K, L2 cache 256K)

	code	analysis (misses)	measurement (ms)
	Base	$\sim$N^3 $= 2^{40}$	19,371±186
H	Tile k	$\sim N^4/C + 2N^2 \sim 2^{27}$	4353±96
	Tile k & j	$\sim 2N^3/C + N^2 \sim 2^{20}$	4153±121
	Tile k, j & i	$\sim 3N^3/C \sim 2^{18}$	4625±212

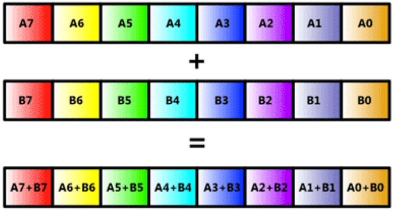
- Analysis did not take into account line size and factors other than misses
- Usually need to experiment in order to find right tiling parameters

- Previous discussion assumed one cache level; how do we deal with multiple cache levels?
- Usually, memory is main bottleneck; tile so as to reduce memory accesses.
- Often sufficient to tile for L2 locality
- But can do hierarchical tiling: E.g., for transpose, can transpose 8×8 in vector registers (load 8 vectors of 8 words; transpose in registres; store 8 vectors)
- For Matrix$\times$Matrix can use recursive algorithm that tiles for successive cache levels

Parallelism

Vector operations

- *Single Instruction, Multiple Data* (SIMD): one instruction operates simultaneously on multiple entries of a vector.
- Uses *vector registers* and vector arithmetic units



Why vector operations?

- Reduce instruction decoding overhead
- Hardware does not need to worry about dependences between operations
    - It is up to the programmer and the compiler to create groups of independent adds
- Runs twice as fast if single precision is used

using single precision (with vector instructions) means twice as many flops – as well as better use of memory capacity and bandwidth and better cache hit ratio Should be done whenever higher precision is not required.

How does on use vector instructions?

- Trust the compiler to *vectorize* loops (but verify)
  - works well for simple loops
- Use vector *intrinsics* (buildin functions recognized by the compiler); they map onto operations on vector registers.
  - Vector intrinsics are machine specific; we illustrate with Intel intrinsics

```
register double a[8],b[8],c[8];
...
for(i=0;i<8; i++)
  a[i]=b[i]+c[i];
```

```
...
__m512d a,b,c; \* 512 bit vector
                  register (8 doubles)
...
a=_mm512_add(b,c);
```

What vector operations exist?

vector op vector apply operation element-wise

masked vector-vector apply operation element-wise where mask is 1

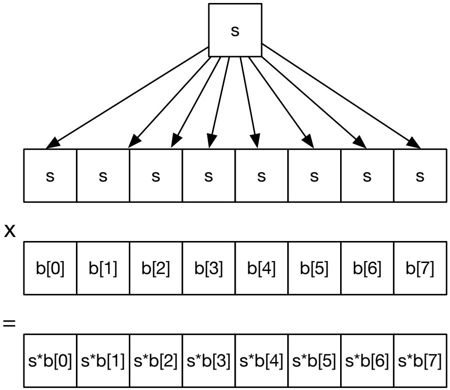
```
  register double a[8],b[8],c[8],d[8];
  register boolean mask[8];
  ...
for(i=0;i<8; i++)
   a[i]=mask[i]?b[i]+c[i]:d[i];


  ...
  __m512d a,b,c,d; __mmask8 mask;
  ...
  a=_mm512_mask_add(d,mask,b,c)
```

- masked vector operations take (at least) as much time as regular vector operations!

vector op scalar done as broadcast, followed by vector-vector operation

```
register double a[4],s
  for(i=0;i<4; i++)
    a[i]=s*b[i];
```

```
__m512d ss,a,b; double s;
...
ss =_mm512_broadcastd_pd(s);
a=_mm512_mul_pd(ss,b)
```



Reduction

sum elements of a vector

```
register double a[8],s;
 for(i=0;i<8; i++)
   s=s+a[i];
```

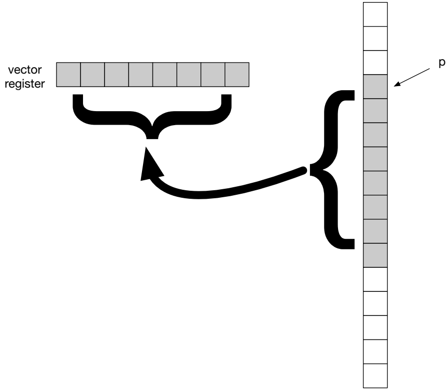
```
...
__m512d a; double s;
s=_mmreduce_add_pd(a);
```

load/store

load/store from/to consecutive memory locations

```
register a[8]; double *p;
...
for(i=0; i<8; i++)
    a[i] = p++
```

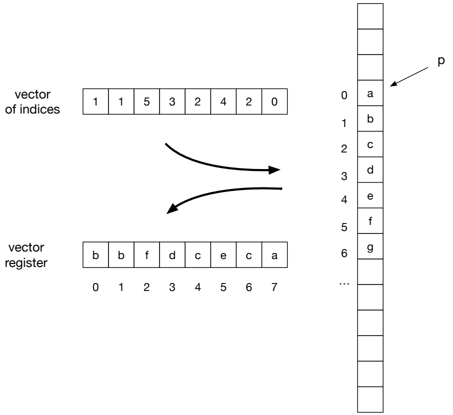
```
...
__m512d a; double *p
a = _mm512_load_pd(p)
```



gather-scatter

- gather nonconsecutive elements in memory to a vector register
- scatter elements in a vector register to nonconsecutive memory location

```
register double a[], b[8]; long p[8
for(i=0;i<8; i++)
  b[i]=a[p[i]];

...
__mmd b; __m512i p; double a[];
b=_mm512_i64gather_pd (_p, a, 8);
```



- Much less efficient than load/store.
- May be required for irregular computations