Core 2:
Instruction Decoding and Execution Units

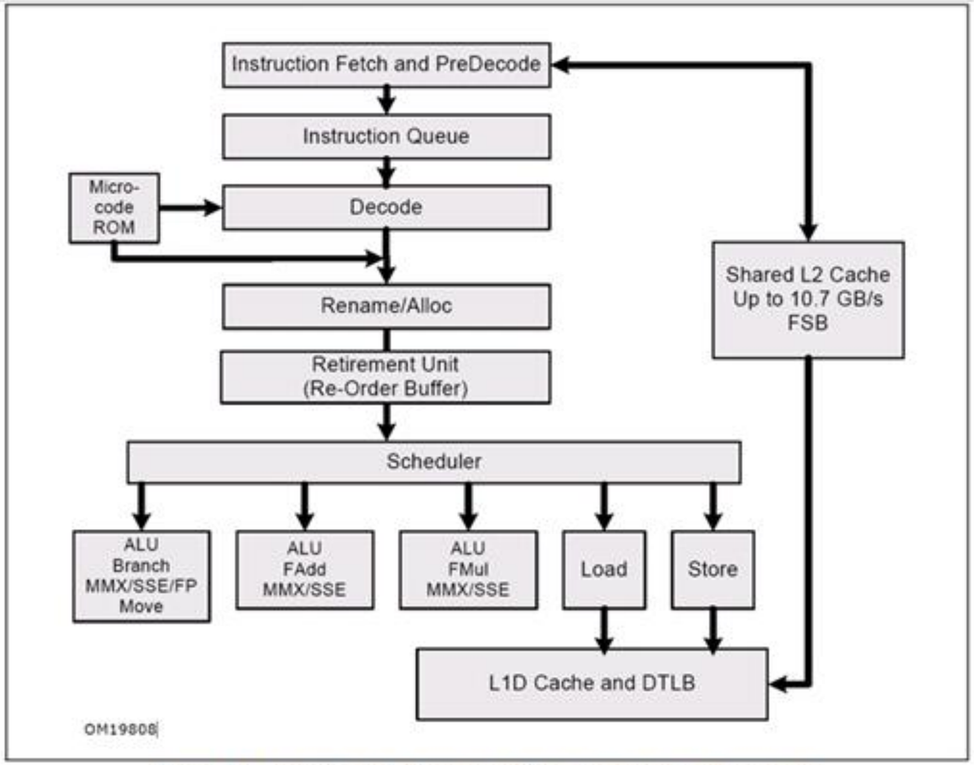


Figure 2-1. Intel Core Microarchitecture Pipeline Functionality

Latency/throughput (double)

FP Add: 3, 1

FP Mult: 5, 1

Hard Bounds: Pentium 4 vs. Core 2

- **Pentium 4 (Nocona)**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	**7**	**2**
Single/Double FP Add	**5**	**2**
Single/Double FP Divide	32/46	32/46

- **Core 2**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	3	1
Integer/Long Divide	18/50	18/50
Single/Double FP Multiply	**4/5**	**1**
Single/Double FP Add	**3**	**1**
Single/Double FP Divide	18/32	18/32

Hard Bounds (cont'd)

- **How many cycles at least if**
  - Function requires n float adds?
  - Function requires n float ops (adds and mults)?
  - Function requires n int mults?

Example Computation (on Pentium 4)

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

d[0] OP d[1] OP d[2] OP … OP d[length-1]

- **Data Types**
  - Use different declarations for **data_t**
  - **int**
  - **float**
  - **double**

- **Operations**
  - Use different definitions of **OP** and **IDENT**
  - **+ / 0**
  - *** / 1**

Runtime of Combine4 (Pentium 4)

- Use cycles/OP

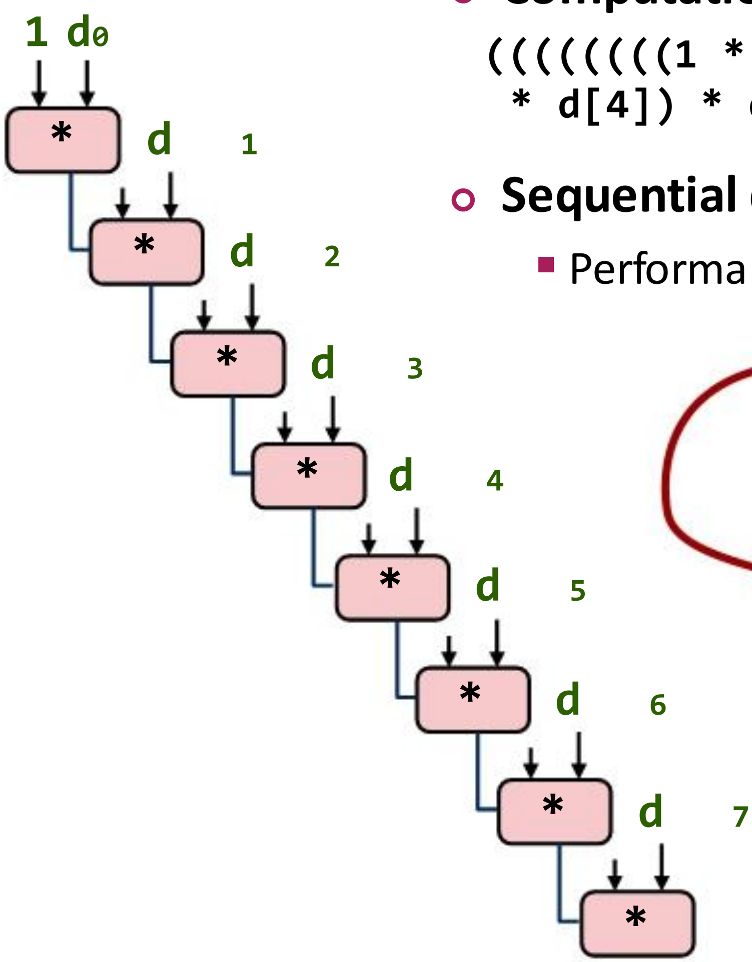
```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Questions:
  - Explain red row
  - Explain gray row

Cycles per OP

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

Combine4 = Serial Computation (OP = *)



- **Computation (length=8)**

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- **Sequential dependence = no ILP! Hence,**
  - Performance: determined by latency of OP!

Cycles per element (or per OP)

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

Loop Unrolling

```
void unroll2(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit  = length-1;
    data_t *d  = get_vec_start(v);
    data_t x   = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2)
        x = (x OP d[i]) OP d[i+1];
    /* Finish any remaining elements */
    for (; i < length; i++)
        x = x OP d[i];
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

- **Helps integer sum**
- **Others don't improve. *Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

Loop Unrolling with Reassociation

```
void unroll2_ra(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit  = length-1;
    data_t *d  = get_vec_start(v);
    data_t x   = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2)
        x = x OP (d[i] OP d[i+1]);
    /* Finish any remaining elements */
    for (; i < length; i++)
        x = x OP d[i];
    *dest = x;
}
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Effect of Reassociation

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
bound	1.0	1.0	2.0	2.0

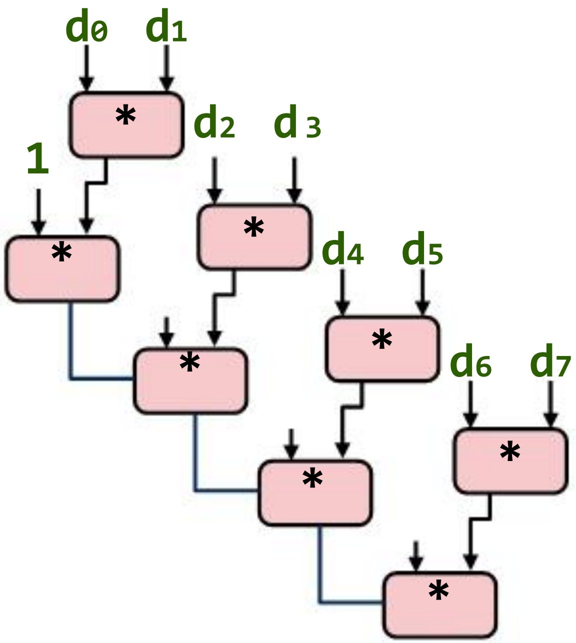
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

  - Why is that? (next slide)

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



- **What changed:**
  - Ops in the next iteration can be started early (no dependency)

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    cycle per OP ≈ D/2
  - Measured is slightly worse for FP

Loop Unrolling with Separate Accumulators

```
void unroll2_sa(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit  = length-1;
    data_t *d  = get_vec_start(v);
    data_t x0  = IDENT;
    data_t x1  = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2)
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    /* Finish any remaining elements */
    for (; i < length; i++)
        x0 = x0 OP d[i];
    *dest = x0 OP x1;
}
```

- **Different form of reassociation**

Effect of Separate Accumulators

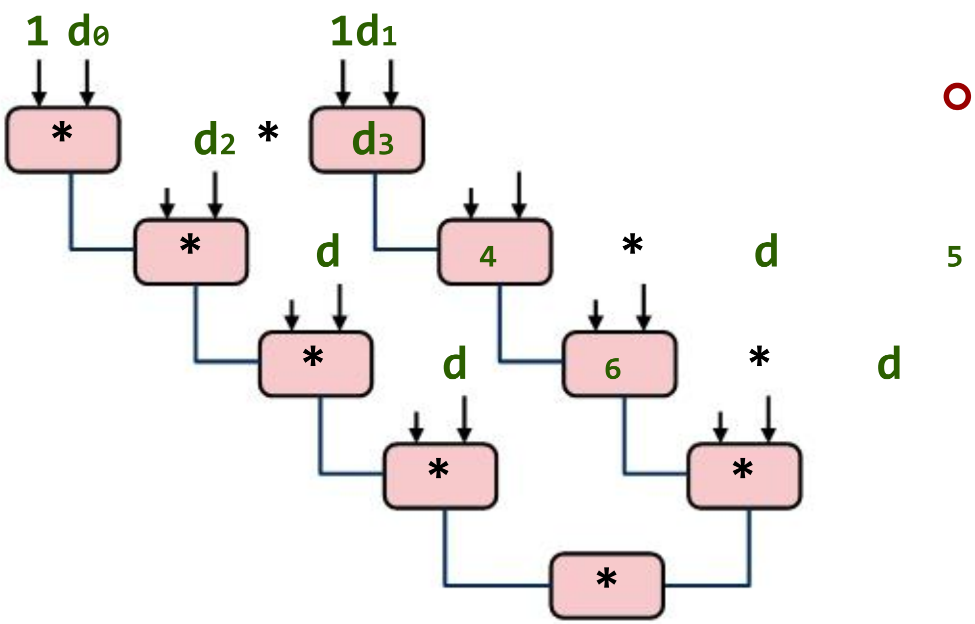
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

- **Almost exact 2x speedup (over unroll2) for Int *, FP +, FP ***
  - Breaks sequential dependency in a "cleaner," more obvious way

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent “streams” of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    cycles per OP ≈ D/2

What Now?

Unrolling & Accumulating

- **Idea**
  - Use K accumulators
  - Increase K until best performance reached
  - Need to unroll by L, K divides L

- **Limitations**
  - Diminishing returns:
    Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths: Finish off iterations sequentially

Unrolling & Accumulating: Intel FP *

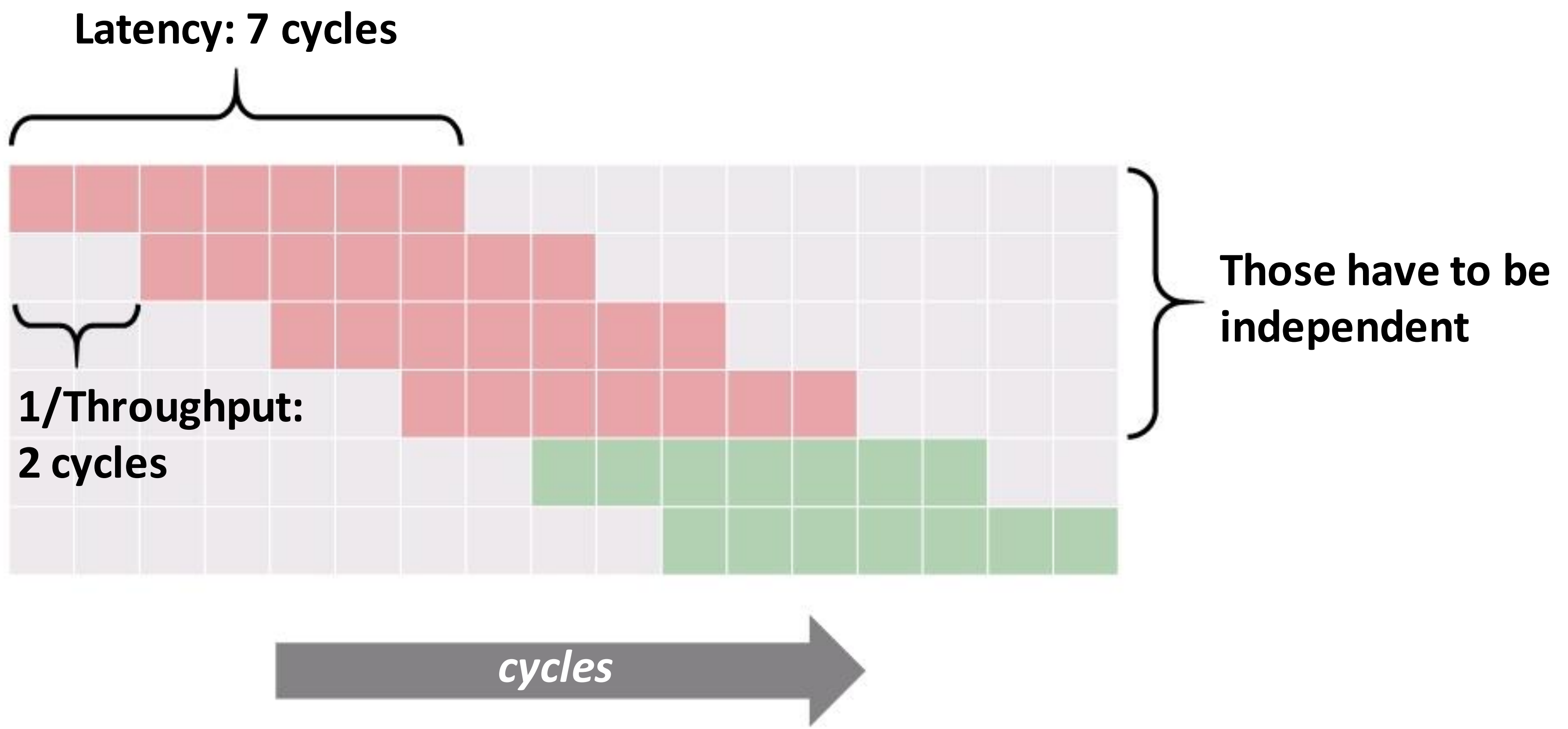
- **Case**
  - Pentium 4
  - FP Multiplication
  - Theoretical Limit: 2.00

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

Accumulators

Why 4?

Why 4?



Based on this insight: **K = #accumulators = ceil(latency/cycles per issue)**

Unrolling & Accumulating: Intel FP +

- **Case**
  - Pentium 4
  - FP Addition
  - Theoretical Limit: 2.00

FP +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.00	5.00		5.02		5.00		
2		2.50		2.51		2.51		
3			2.00					
4				2.01		2.00		
6					2.00			1.99
8						2.01		
10							2.00	
12								2.00

Unrolling & Accumulating: Intel Int *

- **Case**
  - Pentium 4
  - Integer Multiplication
  - Theoretical Limit: 1.00

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.00	10.00		10.00		10.01		
2		5.00		5.01		5.00		
3			3.33					
4				2.50		2.51		
6					1.67			1.67
8						1.25		
10							1.09	
12								1.14

Unrolling & Accumulating: Intel Int +

- **Case**
  - Pentium 4
  - Integer addition
  - Theoretical Limit: 1.00

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.20	1.50		1.10		1.03		
2		1.50		1.10		1.03		
3			1.34					
4				1.09		1.03		
6					1.01			1.01
8						1.03		
10							1.04	
12								1.11

Pentium 4

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

Core 2

*FP * is fully pipelined*

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.00	4.00		4.00		4.01		
2		2.00		2.00		2.00		
3		1.34						
4				1.00		1.00		
6					1.00			1.00
8						1.00		
10							1.00	
12								1.00

Summary (ILP)

- **Instruction level parallelism may have to be made explicit in program**
- **Potential blockers for compilers**
  - Reassociation changes result (FP)
  - Too many choices, no good way of deciding
- **Unrolling**
  - By itself does often nothing (branch prediction works usually well)
  - But may be needed to enable additional transformations (here: reassociation)
- **How to program this example?**
  - Solution 1: program generator generates alternatives and picks best
  - Solution 2: use model based on latency and throughput

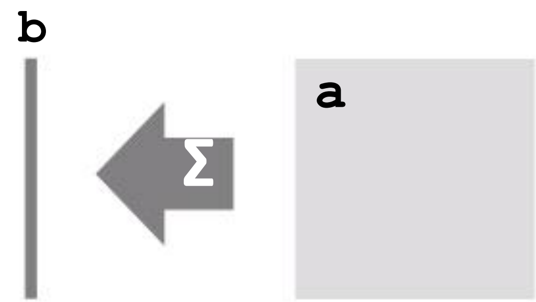
Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
  - Overview
  - Removing unnecessary procedure calls
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - *Optimization blocker: Memory aliasing*
  - Summary

Optimization Blocker: Memory Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, int n) {
  int i, j;

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

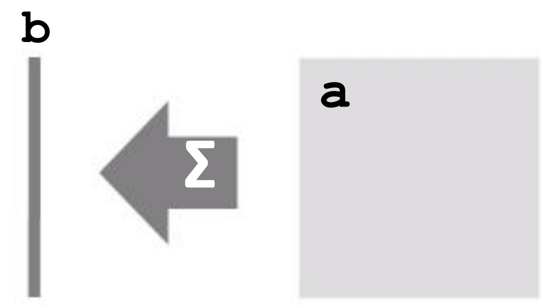


- Code updates `b[i]` (= memory access) on every iteration

Optimization Blocker: Memory Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, int n) {
  int     i, j;

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




```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows2(double *a, double *b, int n) {
  int     i, j;

  for (i = 0; i < n; i++) {
    double  val = 0;
    for (j = 0; j < n; j++)
      val     += a[i*n + j];
    b[i] = val;
  }
}
```

Does compiler optimize this?

No!

Why?

Reason: Possible Memory Aliasing

- If memory is accessed, compiler assumes the possibility of side effects
- Example:

```
/* Sums rows of n x n matrix a
   and stores in vector b*/
void sum_rows1(double *a, double *b, int n) {
  int     i, j;

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

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:   [4,  8,  16]

i = 0: [3,  8,  16]

i = 1: [3, 22,  16]

i = 2: [3, 22, 224]
```

Removing Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b*/
void sum_rows2(double *a, double *b, int n) {
  int     i, j;

  for (i = 0; i < n; i++) {
    double  val = 0;
    for (j = 0; j < n; j++)
      val      += a[i*n + j];
    b[i] = val;
  }
}
```

- **Scalar replacement:**
  - Copy array elements ***that are reused*** into temporary variables
  - Perform computation on those variables
  - Enables register allocation and instruction scheduling
  - Assumes no memory aliasing (otherwise possibly incorrect)

Optimization Blocker: Memory Aliasing

- **Memory aliasing:**
  Two different memory references write to the same location
- **Easy to have happen in C**
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- **Hard to analyze = compiler cannot figure it out**
  - Hence is conservative
- **Solution: Scalar replacement in innermost loop**
  - Copy memory variables that are reused into local variables
  - Basic scheme:

    Load: *t1 = a[i], t2 = b[i+1], ….*

    Compute: *t4 = t1 * t2; ….*

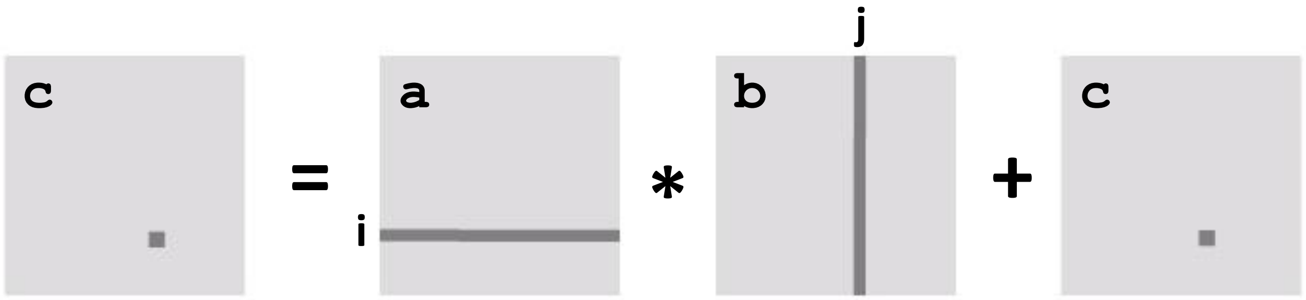
    Store: *a[i] = t12, b[i+1] = t7, …*

More Difficult Example

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices c = a*b + c  */
void mmm(double *a, double *b, double *c, int n) {
  int     i, j, k;

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



- Which array elements are reused?
- All of them! *But how to take advantage?*

Step 1: Blocking (Here: 2 x 2)

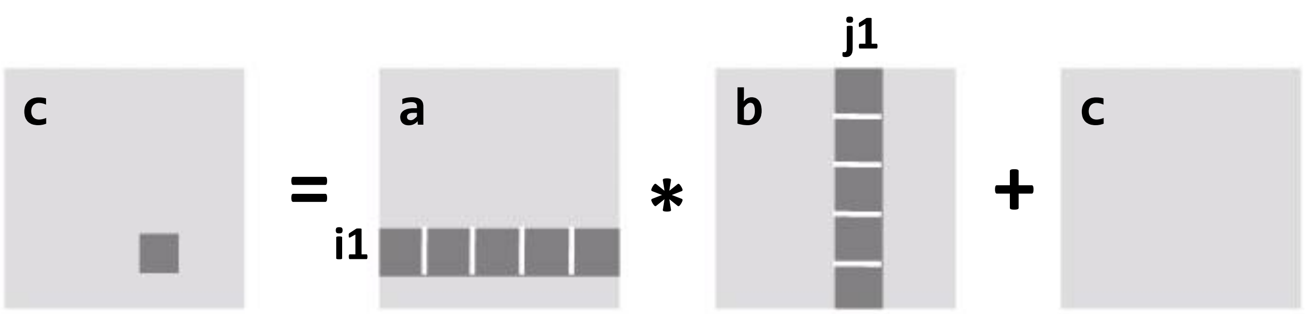
Blocking, also called tiling = partial unrolling + loop exchange

Assumes associativity (= compiler will not do it)

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices c = a*b + c   */
void mmm(double *a, double *b, double *c, int n) {
  int     i, j, k;

  for (i = 0; i < n; i+=2)
    for (j = 0; j < n; j+=2)
      for (k = 0; k < n; k+=2)
        for (i1 = i; i1 < i+2; i1++)
          for (j1 = j; j1 < j+2; j1++)
            for (k1 = k; k1 < k+2; k1++)
              c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Step 2: Unrolling Inner Loops

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices c = a*b + c  */
void mmm(double *a, double *b, double *c, int n) {
  int     i, j, k;

  for (i = 0; i < n; i+=2)
    for (j = 0; j < n; j+=2)
      for (k = 0; k < n; k+=2)
        <body>
}
```

```
<body>:
c[i*n + j]         = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j]
                     + c[i*n + j]
c[(i+1)*n + j]     = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j]
                     + c[(i+1)*n + j]
c[i*n + (j+1)]     = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)]
                     + c[i*n + (j+1)]
c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)]
                     + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)]
```

- **Every array element a[…], b[…], c[…] used twice**
- **Now scalar replacement can be applied**
  (so again: loop unrolling is done with a purpose)

Can Compiler Remove Aliasing?

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

Potential aliasing: Can compiler do something about it?

Compiler can insert runtime check:

```
if (a + n < b || b + n < a)
  /* further optimizations may be possible now */

else
  /* aliased case */
```

Removing Aliasing With Compiler

- **Globally with compiler flag:**
  - `-fno-alias, /Oa`
  - `-fargument-noalias, /Qalias-args-` (function arguments only)
- **For one loop: pragma**

```
void add(float *a, float *b, int n) {
  #pragma ivdep
  for (i = 0; i < n; i++)
    a[i] = a[i] + b[i];
}
```

- **For specific arrays: restrict (needs compiler flag** `-restrict, /Qrestrict`**)**

```
void add(float *restrict a, float *restrict b, int n) {
  for    (i = 0; i < n; i++)
    a[i] = a[i] + b[i];
}
```