

Program Optimization

Slides adapted from the CMU version of the course (thanks to Randal E. Bryant and David R. O'Hallaron)

Today

■ Overview

■ Generally Useful Optimizations

- Code motion/precomputation
- Strength reduction
- Common subexpressions elimination
- Removing unnecessary procedure calls

■ Optimization Blockers

- Procedure calls
- Memory aliasing

■ Parallelization

■ Exploiting Instruction-Level Parallelism

Performance Realities

■ There's more to performance than asymptotic complexity

■ Constant factors matter too!

- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels
 - algorithm, data representations, and loops

■ Must understand system to optimize performance

- How programs are compiled and executed
- How to measure program performance and identify bottlenecks
- How to improve performance without destroying code modularity and generality

Optimizing Compilers

■ Optimizing compilers have made progress!

- register allocation
- code selection and ordering (scheduling)
- dead code elimination
- eliminating minor inefficiencies

■ Don't (usually) improve asymptotic efficiency

- up to programmer to select best overall algorithm
- big-O savings are (often) more important than constant factors
 - but constant factors also matter

Limitations of Optimizing Compilers

■ When in doubt, the compiler must be conservative

- Must not cause any change in program behavior
- Don't have enough information (e.g., runtime information such as data sizes)
- Whole-program analysis is too expensive in most cases

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor/compiler

- Code Motion

- Reduce frequency with which computation performed
 - If it will always produce same result
 - Especially moving code out of loop

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



```
long j;  
int ni = n*i;  
for (j = 0; j < n; j++)  
    a[ni+j] = b[j];
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
 - $16 * x \rightarrow x \ll 4$
 - Utility machine dependent
 - Depends on cost of multiply or divide instruction
 - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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



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

Common Subexpressions Elimination

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n      + j-1];
right = val[i*n      + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

```
/* Sum neighbors of i,j */
up =    val[i*n  -n + j  ];
down =  val[i*n  +n + j  ];
left =  val[i*n      + j-1];
right = val[i*n      + j+1];
sum = up + down + left + right;
```

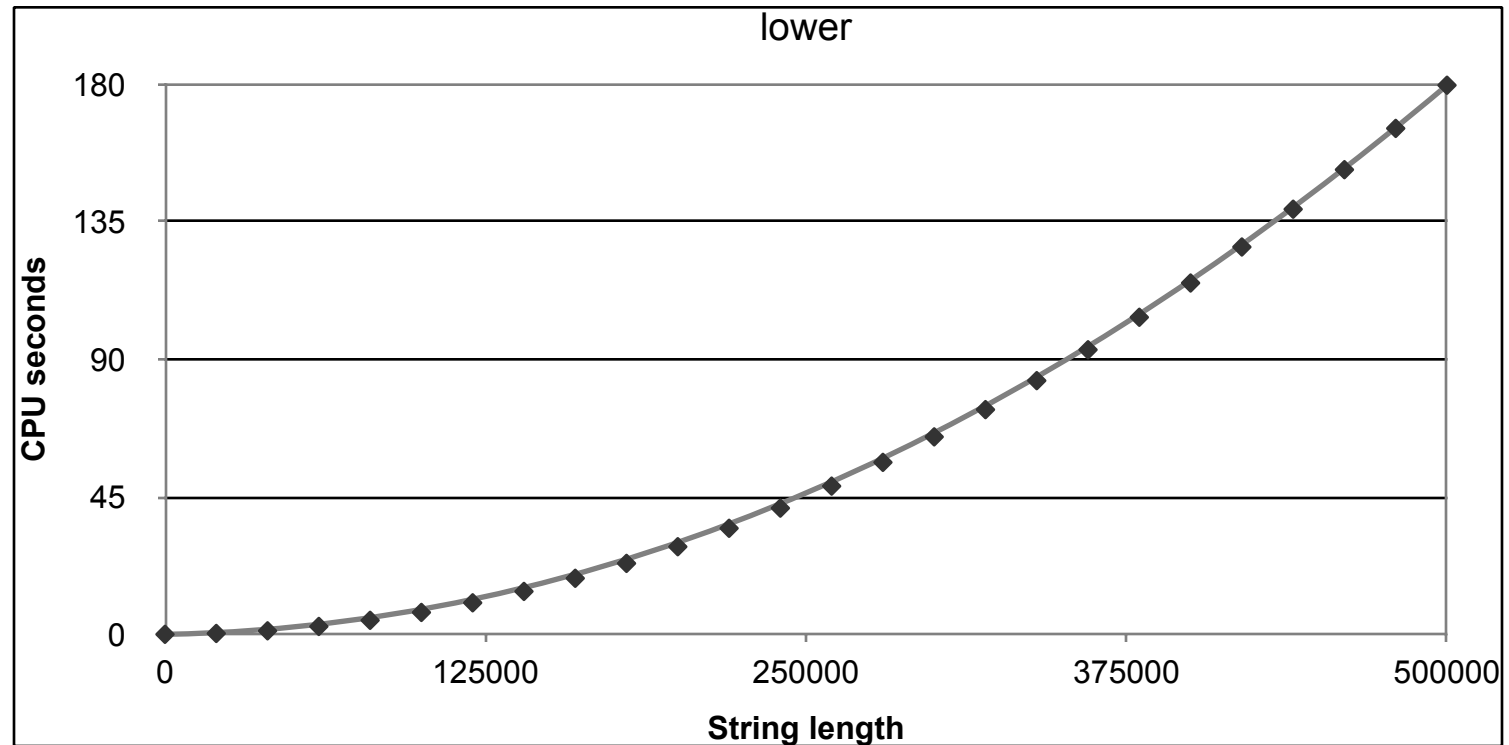

Optimization Blocker #1: Procedure Calls

■ Procedure to Convert String to Lower Case

```
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
    int i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```

- `strlen` executed every iteration

Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

■ Strlen performance

- Only way to determine length of string is to scan its entire length, looking for null character.

■ Overall performance, string of length N

- N calls to strlen
- Require times N, N-1, N-2, ..., 1
- Overall $O(N^2)$ performance

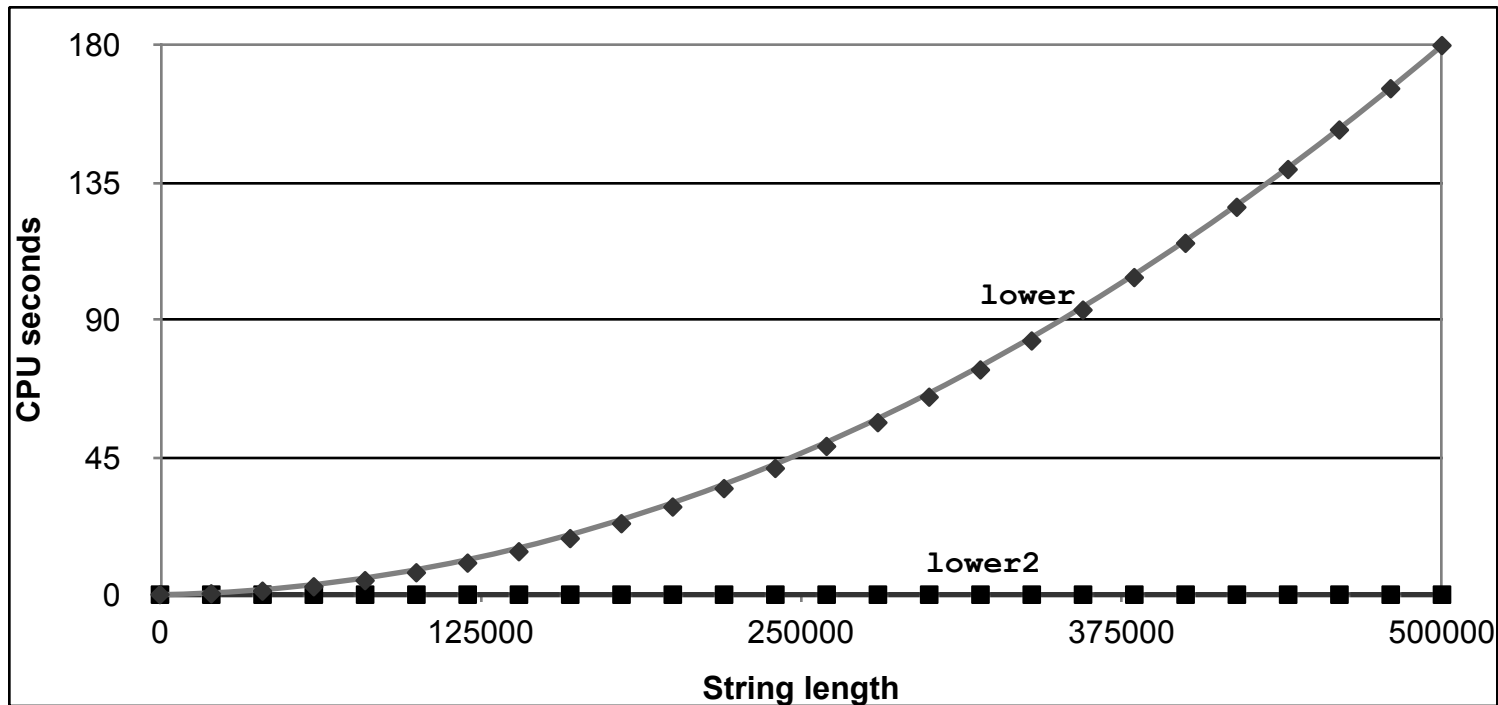
Improving Performance

```
void lower(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker: Procedure Calls

■ *Why couldn't compiler move str1en out of inner loop?*

- Compiler treats procedure call as a black box
 - It may not have the source code of the function
 - Whole-program analysis takes a lot of time
- Procedure may have side effects
 - Alters global state each time called
- Function may not return same value for given arguments
 - Depends on other parts of global state

■ Remedies

- Do your own code motion

Loop Parallelization Improve Performance

```
void assign(double *a) {  
    long i;  
  
    for (i = 0; i < 100; i++)  
        a[i] = 0;  
}
```

- Running a loop in parallel
 - Dividing the loop iterations into groups
 - Running each group by a CPU cores (parallel/simultaneous execution)

```
// Execute by CPU core 0  
for (i = 0; i < 50; i++)  
    a[i] = 0;
```

```
// Execute by CPU core 1  
for (i = 50; i < 100; i++)  
    a[i] = 0;
```


Loop Parallelization: MultiThreading Example

- Loop parallelization: run the loop iterations in parallel

Original

```
for x in 0 ... N
  for y in 0 ... N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

Run the loop by 2 threads

Thread 0

```
for x in 0 ... N/2
  for y in 0 ... N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

Thread 1

```
for x in N/2 ... N
  for y in 0 ... N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

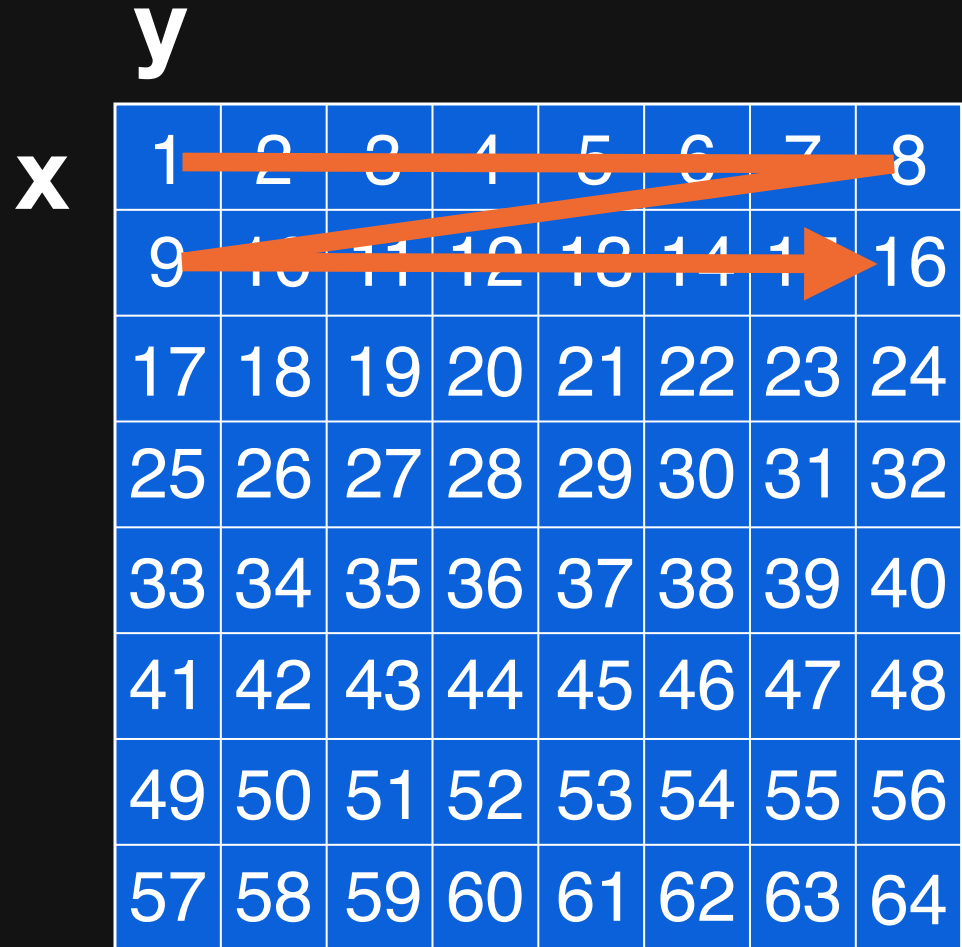
Serial Execution

Serial y
Serial x

y

x

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



Parallel Execution

Serial y

Parallel x

y

x

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Loop Optimizations

- Loop parallelization: run the loop iterations in parallel

OpenMP Parallelization

```
#pragma omp parallel for
for x in 0 ... N
  for y in 0 ... N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

Exploiting Instruction-Level Parallelism

■ Main idea

- Hardware can execute multiple instructions in parallel
- Instructions can be run in parallel if there is no dependency between them (i.e., instructions do not need values calculated by other instructions)

■ Dependence ==> no instruction parallelism

- $B = A + 1;$
- $C = B + 2;$
- $D = C * 3;$

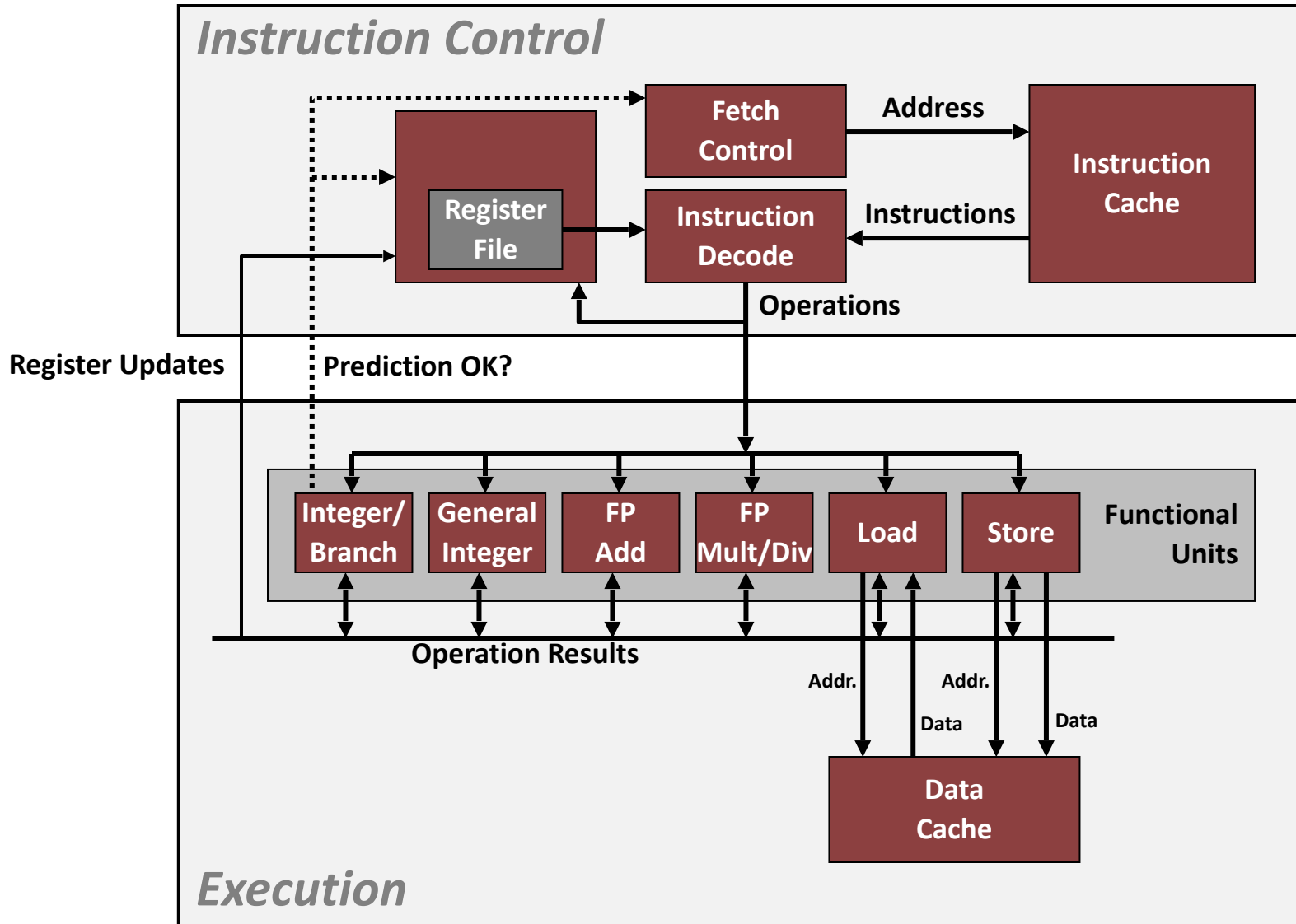
■ No dependence ==> instruction parallelism

- $B = A + 1;$
- $D = C + 2;$
- $E = 0;$

Superscalar Processor

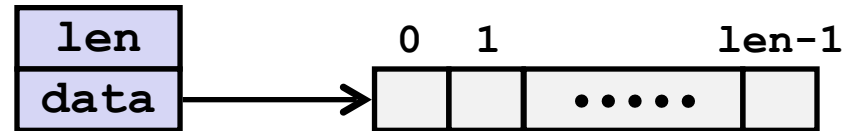
- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have
- **Most CPUs** since about 1998 are superscalar.
- **Intel:** since Pentium Pro

Modern CPU Design



Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */  
typedef struct{  
    int len;  
    data_t *data;  
} vec;
```



```
/* retrieve vector element and store at val */  
double get_vec_element(vec *v, int idx, data_t *val)  
{  
    if (idx < 0 || idx >= v->len)  
        return 0;  
    *val = v->data[idx];  
    return 1;  
}
```


Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or
product of vector
elements

■ Data Types

- Use different declarations for `data_t`
 - `int`
 - `double`

■ Operations

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

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: **CPE = cycles per OP**
- Time = $\text{CPE} * n + \text{Overhead}$

Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or
product of vector
elements

Method	Integer		Double FP (Floating Point)	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	29.0	29.2	27.4	27.9
Combine1 -O1	12.0	12.0	12.0	13.0

Basic Optimizations

```
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;
}
```

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Effect of Basic Optimizations

```
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;
}
```

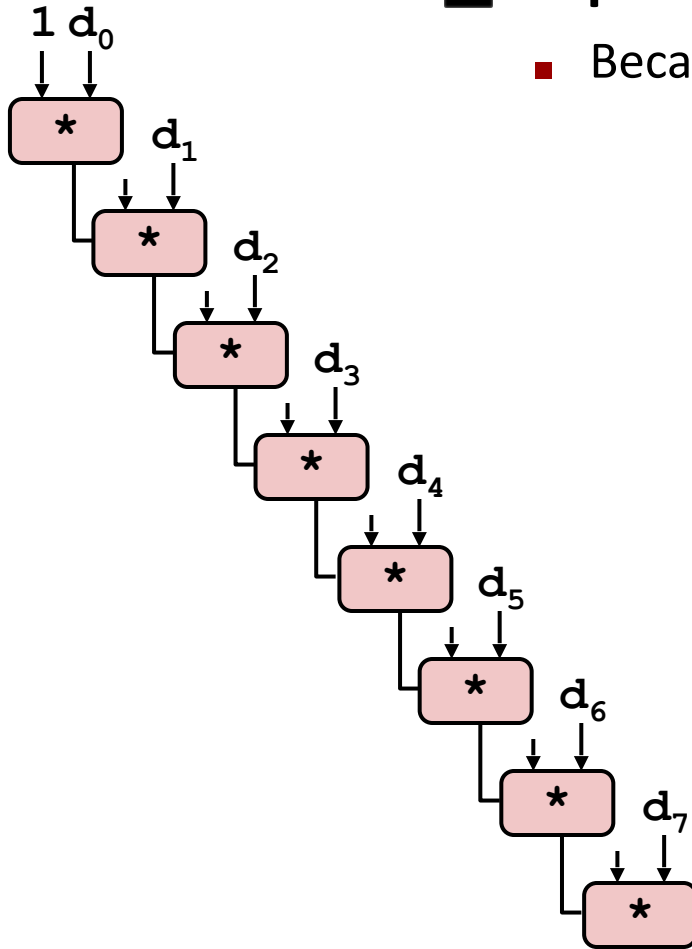
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O1	12.0	12.0	12.0	13.0
Combine4	2.0	3.0	3.0	5.0

■ Eliminates sources of overhead in loop

Combine4 = Serial Computation (OP = *)

■ Sequential Execution

- Because of the dependences



Loop Unrolling

Original

```
void foo(int *A, int limit)
{
    for (int i = 0; i < limit; i+=1) {
        A[i] = 0;
    }
}
```

Unrolled

```
void foo(int *A, int limit)
{
    for (int i = 0; i < limit; i+=4) {
        A[i] = 0;
        A[i+1] = 0;
        A[i+2] = 0;
        A[i+3] = 0;
    }
}
```

Loop Unrolling

```
void unroll2a_combine(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	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0

■ It does not help a lot !

■ *Why?*

- Still sequential dependency

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

Loop Unrolling with Reassociation

```
void unroll2aa_combine(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;
}
```

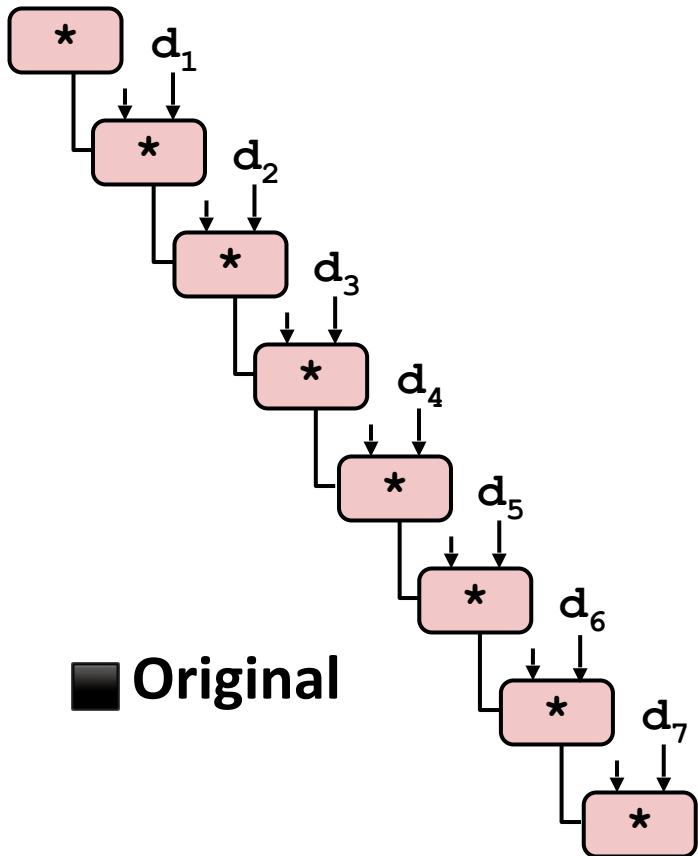
Compare to before

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

■ Can this change the result of the computation?

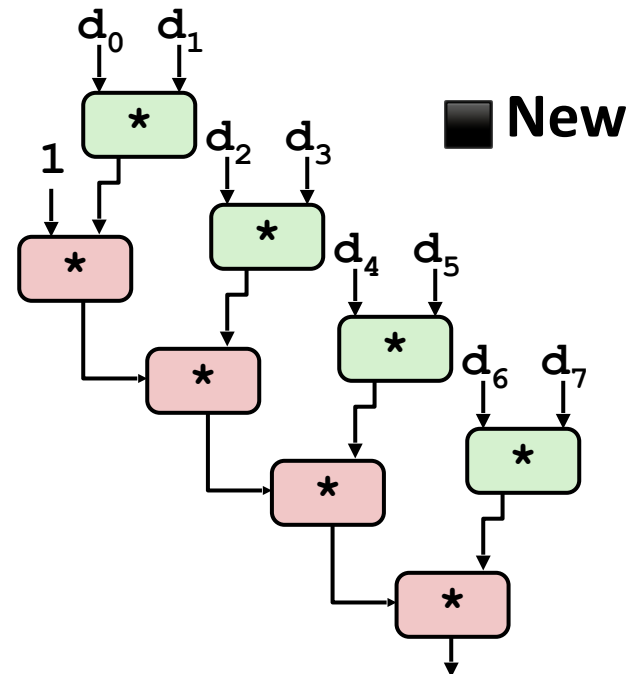
Reassociated Computation

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



What changed:

- Ops in the next iteration can be started early (no dependency)



Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0

■ Nearly 2x speedup

- Reason: Breaks sequential dependency

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