

# Lecture 11: Optimization

---

CS 105

February 26, 2020

# Under the Abstraction Barrier

```
#include<stdio.h>

int main(int argc,
         char ** argv){

    printf("Hello world!\n");
    return 0;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp
leaq   L_.str(%rip), %rax
movl    $0, -4(%rbp)
movl    %edi, -8(%rbp)
movq    %rsi, -16(%rbp)
movq    %rax, %rdi
movb    $0, %al
callq   _printf
xorl    %ecx, %ecx
movl    %eax, -20(%rbp)
movl    %ecx, %eax
addq    $32, %rsp
popq    %rbp
retq
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```



# Techniques for Improving Performance

- ~~1. Use better algorithms/data structures~~
2. Compile to efficient byte code
3. Write code that compiles to efficient byte code
4. Parallelize your execution

# Optimizing Compilers

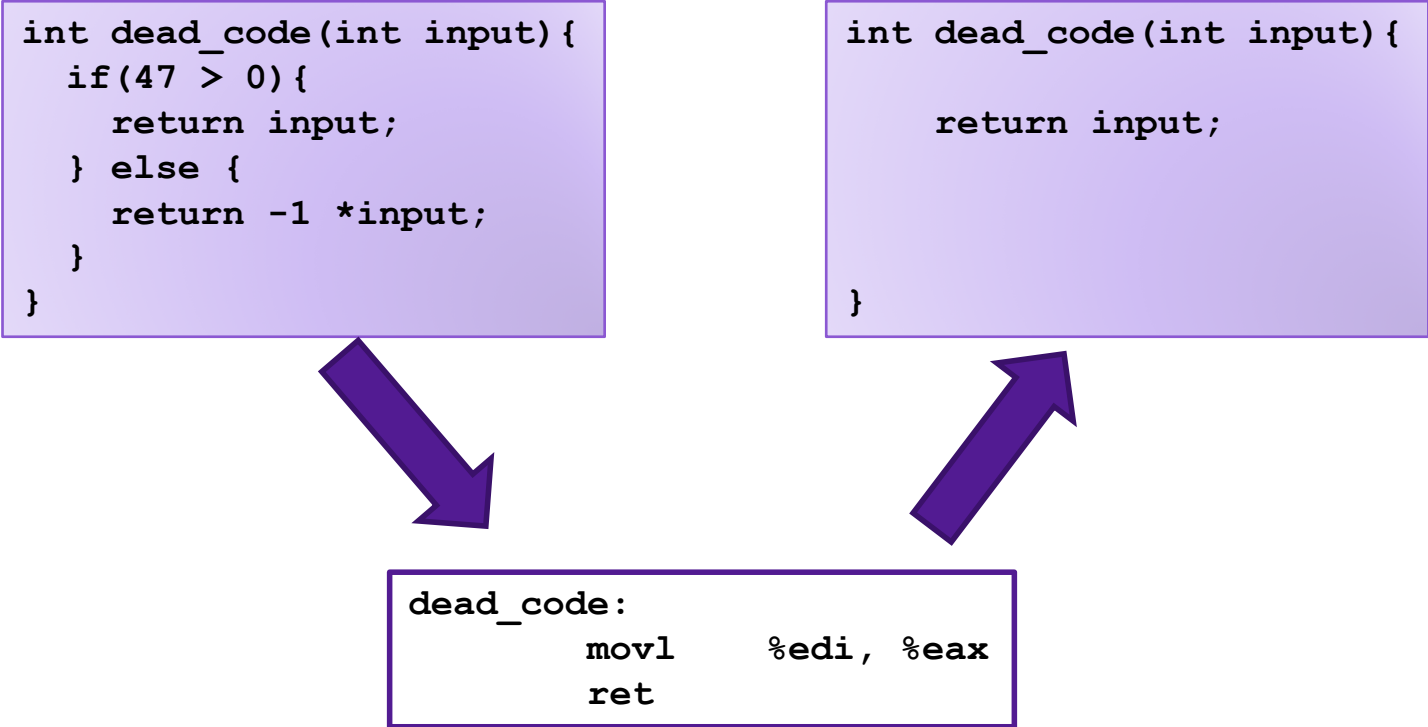
- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - eliminating minor inefficiencies
- Compiler optimization flags
  - `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Og`
- Seldom improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter



# Eliminating Dead Code (-O0)

```
int dead_code(int input){  
    if(47 > 0){  
        return input;  
    } else {  
        return -1 *input;  
    }  
}
```

```
int dead_code(int input){  
    return input;  
}
```



```
dead_code:  
    movl    %edi, %eax  
    ret
```

# Code Motion (-O1)

- Reduce frequency with which computation is performed
- For example, move code out of a loop

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

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

set\_row:

```
testl    %ecx
jle      .L1
imull    %ecx, %edx
addl     %edx, %ecx
```

.L3:

```
movl     (%rsi), %r8d
movslq   %edx, %rax
movl     %r8d, (%rdi,%rax,4)
addl     $1, %edx
addq     $4, %rsi
cmpl     %ecx, %edx
jne      .L3
```

.L1:

```
rep ret
```

# Factoring out Subexpressions (-O1)

- Share common subexpressions
  - Gcc will do this with -O1

```
/* 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;
```

3 multiplications

```
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;
```

1 multiplication

```
imulq    %rcx, %rsi    # i*n  
addq     %rdx, %rsi    # i*n+j  
movq     %rsi, %rax    # i*n+j  
subq     %rcx, %rax    # i*n+j-n  
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

# Loop Elimination (-O1)

```
int loop_while(int a)
{
    int b = 4;
    int i = 0;
    int result = 0;
    while (i < 16) {
        result += a;
        a -= b;
        i += b;
    }
    return result;
}
```

```
int loop_while(int a)
{
    return 4*a-24;
}
```

```
loop_while:
    leal    -24(,%rdi,4), %eax
    ret
```



# Reduction in Strength (-O2)

- Replace costly operation with simpler one
- For example, replace multiplication with shift or addition

```
void set_matrix(long *a, long *b,  
               long n){
```

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

set\_matrix:

```
        xorl    %eax, %eax  
        testq   %rax, %rax  
        leaq   0(%rax), %rdi  
        jle    .L6  
.L3:    movq    (%rax), %rdi  
        movq   %rdi, %rax  
        addq   $1, %rdi  
        cmpq   %rax, %rdx  
        jne    .L3  
        addq   $1, %r8  
        addq   %r9, %rdi  
        cmpq   %r8, %rdx  
        jne    .L6  
.L1:    rep ret
```

```
void set_matrix(long *a, long *b,  
               long n){
```

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



# Loop Unrolling (-O3)

```
int psum1(int a[],int sums[],int n){  
    int i;  
    sums[0] = a[0];  
    for(i = 1; i < n; i++){  
        sums[i] = sums[i-1] + a[i];  
    }  
}
```



```
int psum2(int a[],int p[],int n){  
    int i;  
    sums[0] = a[0];  
    for(i = 1; i < n-1; i+=2){  
        sums[i]   = sums[i-1] + a[i];  
        sums[i+1] = sums[i] + a[i+1];  
    }  
    if (i < n){ // handle odd #iterations  
        sums[i] = sum[i-1] + a[i];  
    }  
}
```

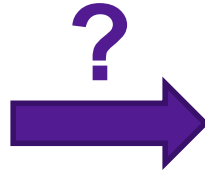


# Limitations of Optimizing Compilers

1. Must not cause any change in program behavior
  - Often prevents optimizations that would only affect behavior under pathological conditions.
    - Data ranges may be more limited than variable type suggests
    - Compiler cannot know run-time inputs
  - When in doubt, the compiler must be conservative

# Limitations of Optimizing Compilers

```
void mystery1(int *xp,  
              int *yp){  
    *xp = *xp + *yp;  
    *yp = *xp - *yp;  
    *xp = *xp - *yp;  
}
```



```
void mystery2(int *xp,  
              int *yp){  
    int temp = *xp;  
    *xp = *yp;  
    *yp = temp;  
}
```



# Case Study 1: Summing Matrix Rows

```
/* Sum rows of nxn matrix a, store
   in vector sums */
void sum_rows1(int *a, ints *sums,
               int n) {
    for (int i = 0; i < n; i++) {
        sums[i] = 0;
        for (long j = 0; j < n; j++){
            sums[i] += a[i*n + j];
        }
    }
}
```



```
# sum_rows1 inner loop
.L4:
    movl    (%rax), %edx
    addl    %edx, (%rsi,%r10,4)
    addq   $4, %rax
    decq   %rcx
    jne    .L3
```

```
/* Sum rows of nxn matrix a, store
   in vector sums */
void sum_rows2(int *a, int *sums,
               int n) {
    for (int i = 0; i < n; i++) {
        int val = 0;
        for (long j = 0; j < n; j++){
            val += a[i*n + j];
        }
        sums[i] = val;
    }
}
```



```
# sum_rows2 inner loop
.L10:
    addl    (%rdx), %eax
    addq   $4, %rdx
    decq   %rcx
    jne    .L3
```

# Optimization Blocker 1

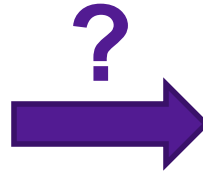
- Aliasing: Two different references to a single location
  - Easy to happen in C
  
- Develop habit of introducing local variables
  - To accumulate within loops, for example
  - Your way of telling the compiler not to check for aliasing

# Limitations of Optimizing Compilers

1. Must not cause any change in program behavior
  - Often prevents optimizations that would only affect behavior under pathological conditions.
    - Data ranges may be more limited than variable type suggests
    - Compiler cannot know run-time inputs
  - When in doubt, the compiler must be conservative
2. Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of `gcc` do interprocedural analysis within files

# Example

```
long f1();  
  
long f2(){  
    return f1() + f1();  
}
```

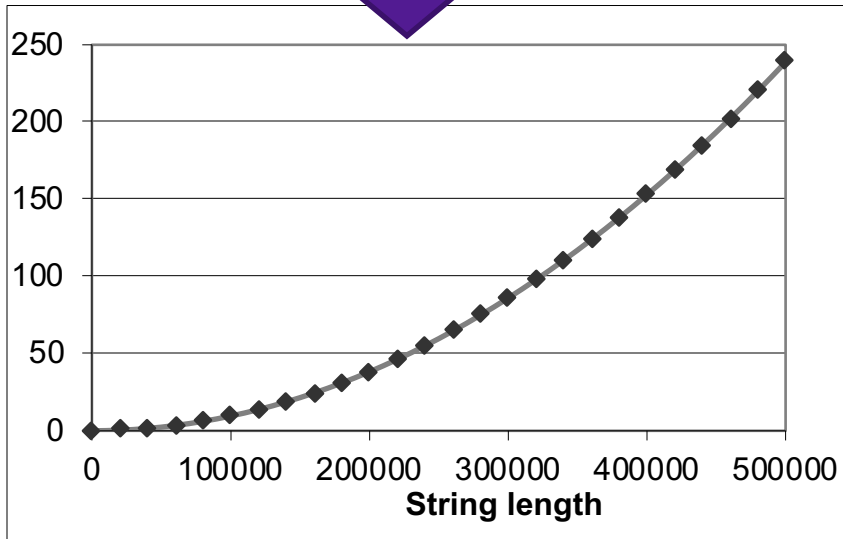


```
long f1();  
  
long f2(){  
    return 2*f1();  
}
```

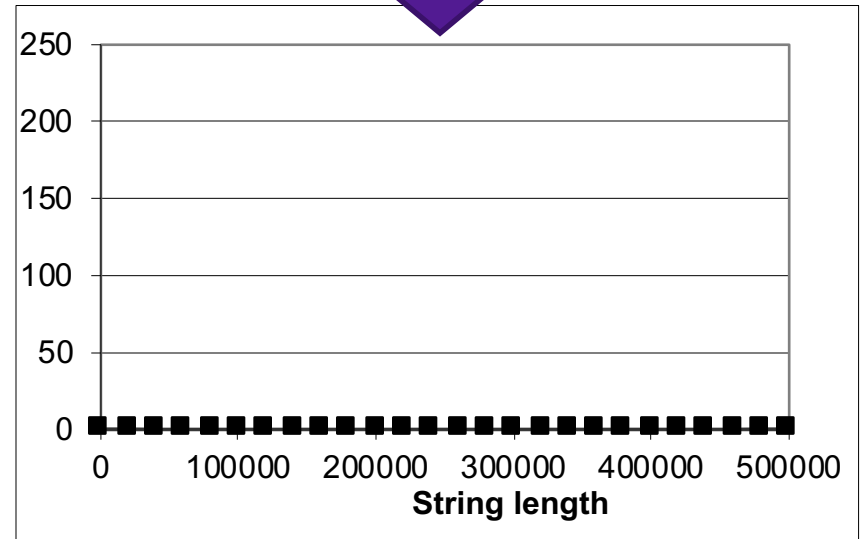
# Case Study 2: Lowering Case

```
void lower(char *s){
    int i;

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



```
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');
        }
    }
}
```



# Optimization Blocker 2

- Compiler treats procedure calls as black boxes
  - Unknown side-effects
  - `strlen` may not always return the same value
- Alternatives:
  - Do your own code motion (necessary here)
  - Use inline keyword when declaring functions
    - `gcc` will optimize within a single file with `-O1`

# Machine Independent Optimization

- Compilers optimize assembly code
  - Dead code elimination
  - Code motion
  - Factoring out common subexpressions
  - Loop elimination
  - Reduction in Strength
  - Loop unwinding
- Optimization blockers:
  - Aliasing
    - Use local variables
  - Procedure calls
    - Move them yourself

# Case Study 3: Vector Data Type

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

data\_t will vary by example

- int
- long
- float
- double

```
/* retrieve vector element and store at val */  
  
data_t * get_vec_element(vec *v, size_t idx) {  
  
    if (idx >= v->len){  
        return NULL;  
    }  
    return &(amp;v->data[idx]);  
}
```



# Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long i;
    *dest = IDENT;

    for (i = 0; i < vec_length(v); i++) {
        data_t * val = get_vec_element(v, i);
        *dest = *dest OP *val;
    }
}
```

Sum or product of vector elements

Metric: CPE, cycles per element

IDENT/OP may be 0/+ or 1/\*

Time = CPE \* n + Overhead

# Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long i;
    *dest = IDENT;

    for (i = 0; i < vec_length(v); i++) {
        data_t * val = get_vec_element(v, i);
        *dest = *dest OP *val;
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O0	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14

Exercise: how could you optimize this code to get even better performance?

# Code-Level Optimizations

```
void combine2(vec_ptr v, data_t *dest)
{
    long i;
    data_t x = IDENT;
    long length = vec_length(v);
    data_t *d = get_vec_element(v,0);
    for (i = 0; i < length; i++){
        x = x OP d[i];
    }
    *dest = x;
}
```

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

# Code-Level Optimizations

```
void combine2(vec_ptr v, data_t *dest)
{
    long i;
    data_t t = IDENT;
    long length = vec_length(v);
    data_t *d = get_vec_element(v,0);
    for (i = 0; i < length; i++){
        t = t OP d[i];
    }
    *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O0	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01

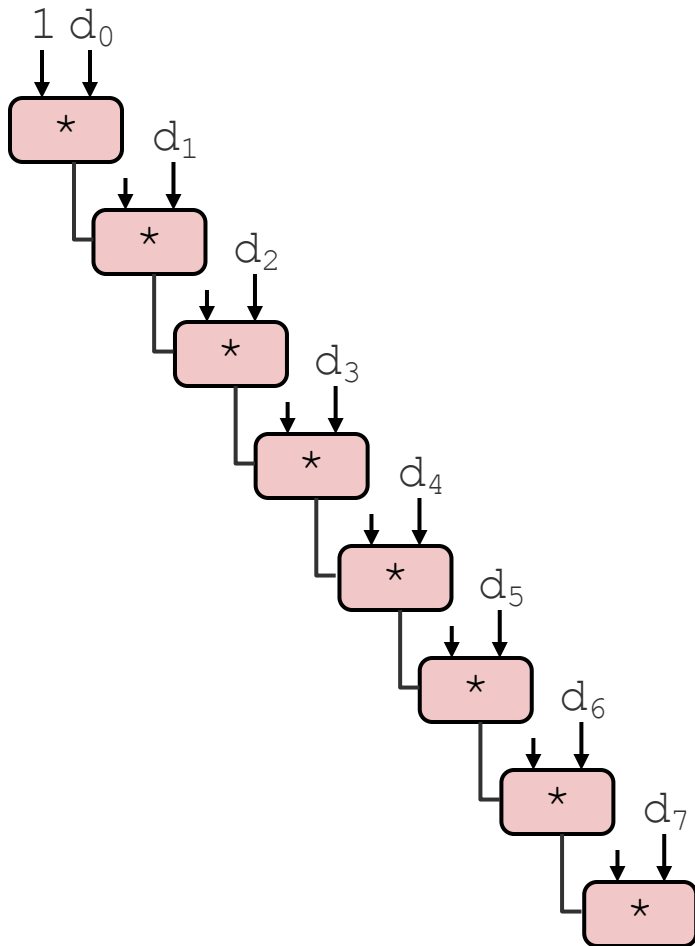
# Combine with Unrolling

```
void unroll2_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_element(v,0);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
}
```

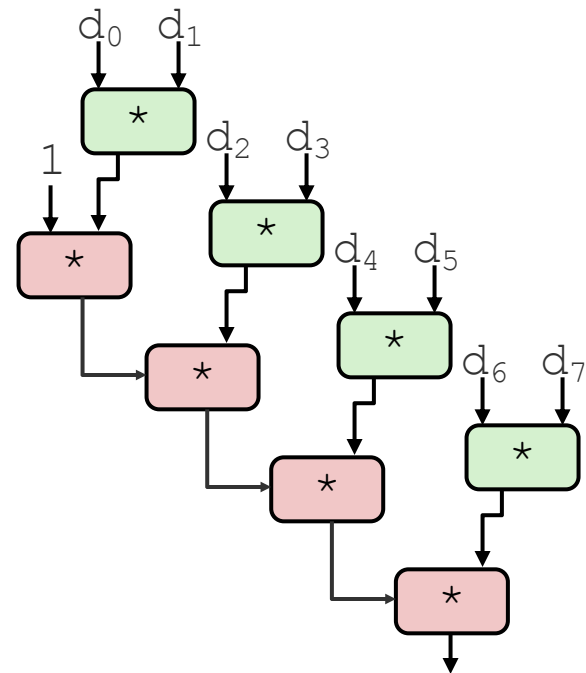
Method	Integer		Double FP	
	Add	Mult	Add	Mult
Combine1 -O0	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

# Reassociation

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



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



# Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O0	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

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

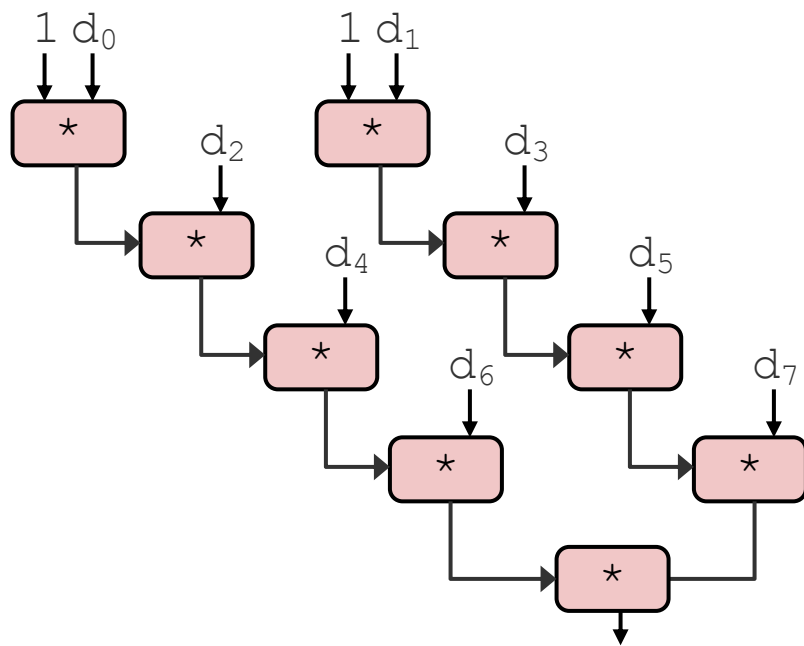
```
x = x OP (d[i] OP d[i+1]);
```

4 func. units for int +  
2 func. units for load

2 func. units for FP \*  
2 func. units for load

# Separate Accumulators

```
void unroll2a_combine(vec_ptr v,
                     data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_element(v, 0);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long 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;
}
```



- Two independent streams of operation



# Effect of Separate Accumulators

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Int + makes use of two load units
- 2x speedup (over unroll2) for Int \*, FP +, FP \*

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

# Machine-Dependent Parallelization

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -O0	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Optimal Unrolling	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Limited only by throughput of hardware
- Up to 42X improvement over original, unoptimized code