Lecture 12: Optimization

CS 105

Spring 2026

Under the Abstraction Barrier

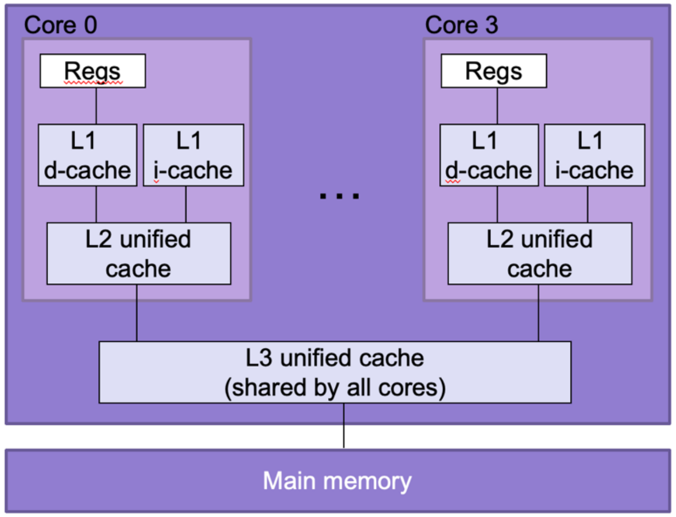
```c
#include<stdio.h>

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 code
  - 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;
  }
}
```

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

```
int dead_code(int input){

    return input;



}
```

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];
  }
}
```

```
set_row:
        testl   %ecx, %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
```

```
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];
  }
}
```

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

Loop Elimination (-O1)

```c
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;
}
```

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

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

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
    }

  }
}
```

```
set_matrix:
         xorl    %
         testq   %
         leaq    0
         jle     .
.L6:     xorl    %
.L3:     movq    (
         movq    %
         addq    $
         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){
  long ni = 0;
  for (long i = 0; i < n; i++) {

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

Machine Independent Optimization

- Compilers optimize assembly code
    - Dead code elimination
    - Code motion
    - Factoring out common subexpressions
    - Loop elimination
    - Reduction in Strength

Case Study: Vector Data Type

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

`data_t` will vary by example

- `int`
- `double`

```
/* get length of vector */
long vec_length(vec* v) {

    return v->len;
}
```

```
/* get address of vector element */
data_t* get_vec_elem(vec* v, long idx) {

    if (idx >= v->len){
        return NULL;
    }
    return &(v->data[idx]);
}
```

Benchmark Computation

```
void combine1(vec* v, data_t* dest){

  *dest = IDENT;

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

Sum or product of vector elements

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

Metric: CPE, cycles per element

Time = CPE * n + Overhead

Benchmark Performance

```
void combine1(vec* v, data_t* dest){

  *dest = IDENT;

  for(long i = 0; i < vec_length(v); i++) {
    data_t* val = get_vec_elem(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

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

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;
}
```

$\xrightarrow{?}$

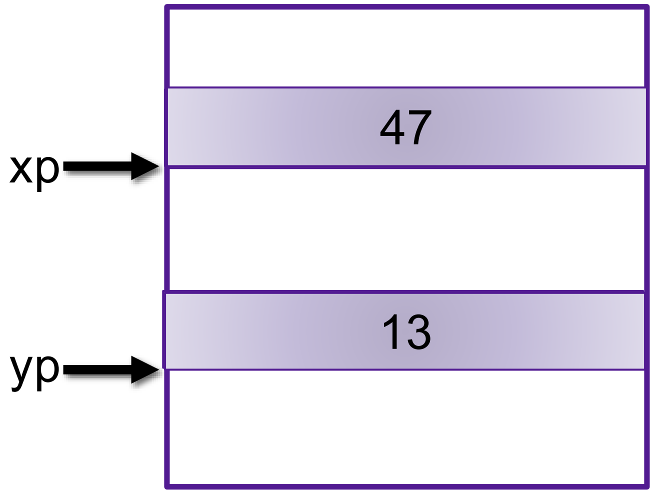
```
void mystery2(int* xp,
              int* yp){
  int temp = *xp;
  *xp = *yp;
  *yp = temp;
}
```

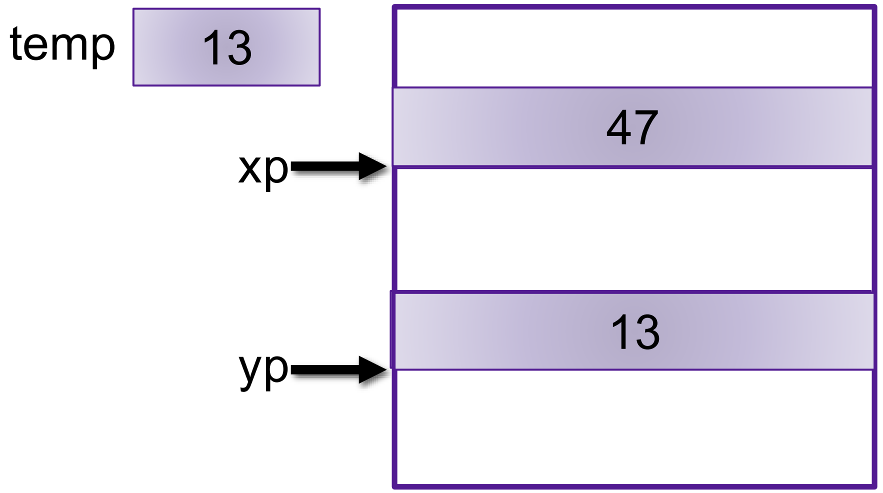
Exercise: What do each of these programs do?
Do they do the same thing?

Comparing Programs

```
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;
}
```

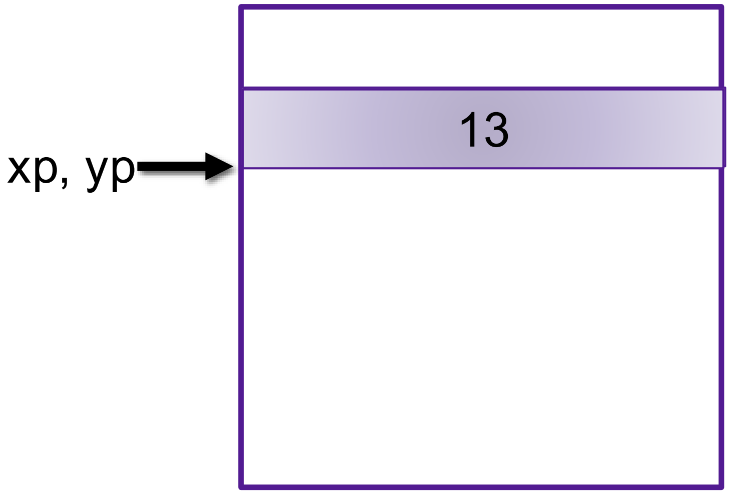


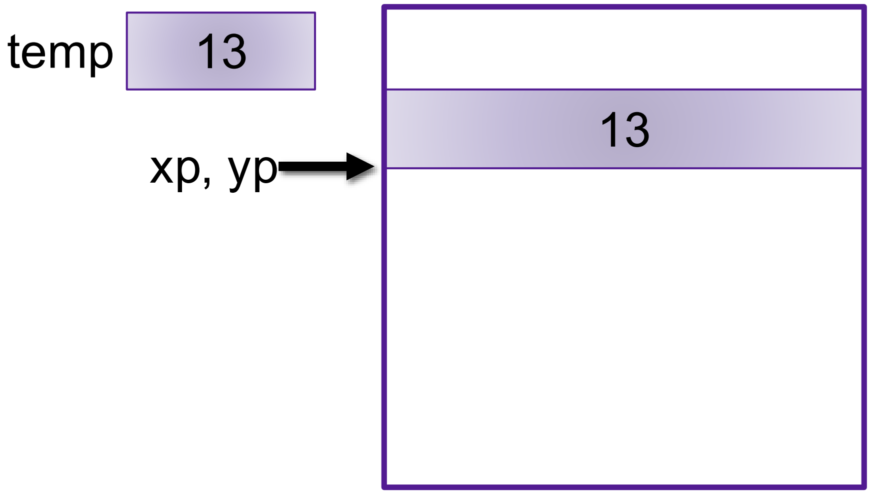


Comparing Programs

```
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;
}
```





Optimization Blocker 1

- Aliasing: Two different references to a single location
  - Easy to happen in C

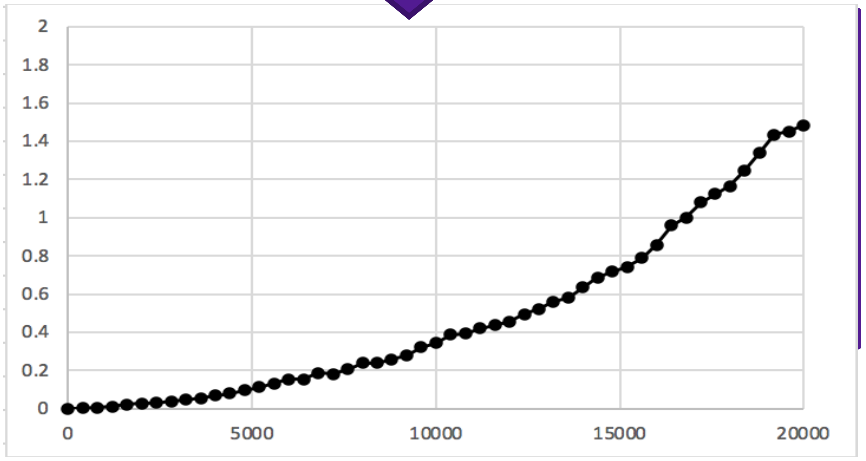
- Develop habit of introducing local variables
  - To accumulate within loops, for example
  - Your way of telling the compiler not to check for aliasing

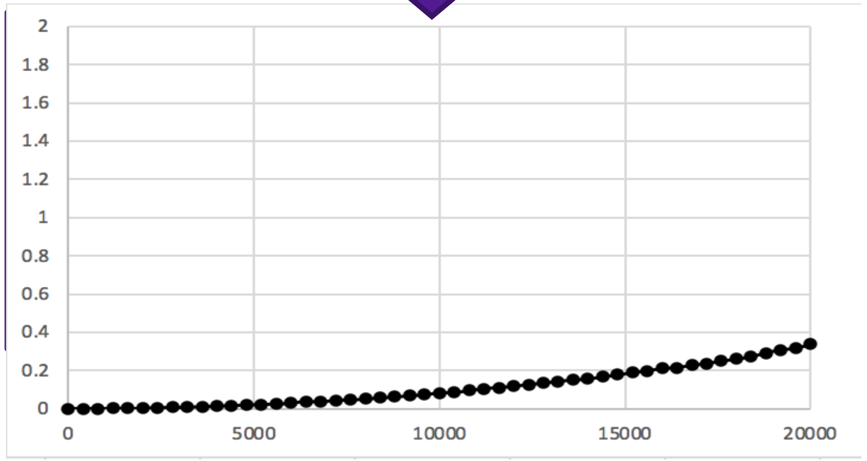
Example: Summing Matrix Rows

```
/* Sum rows of nxn matrix a, store
   in vector sums                   */
void sum_rows1(int* a, int* 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 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;
  }
}
```





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 inter-procedural analysis within files

Exercise 2: Procedure Calls

Consider the following two functions. What do each of these programs do? Do they do the same thing?

```
long f1();

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

?

```
long f1();

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

Optimization Blocker 2

- Compiler treats procedure calls as black boxes
  - Unknown side-effects
  - **`strlen`** may not always return the same value

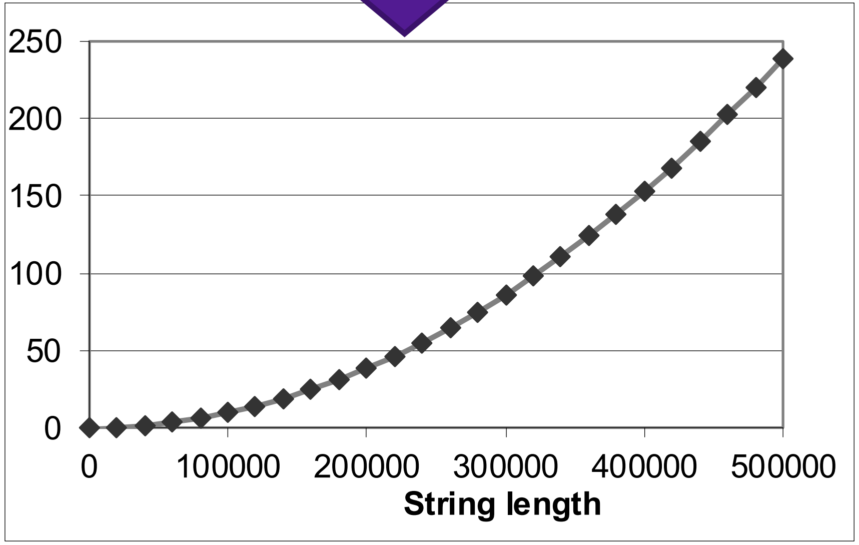
- Alternatives:
  - Do your own code motion (necessary here)
  - Use inline keyword when declaring functions
    - **`gcc`** will optimize within a single file with **`–O1`**

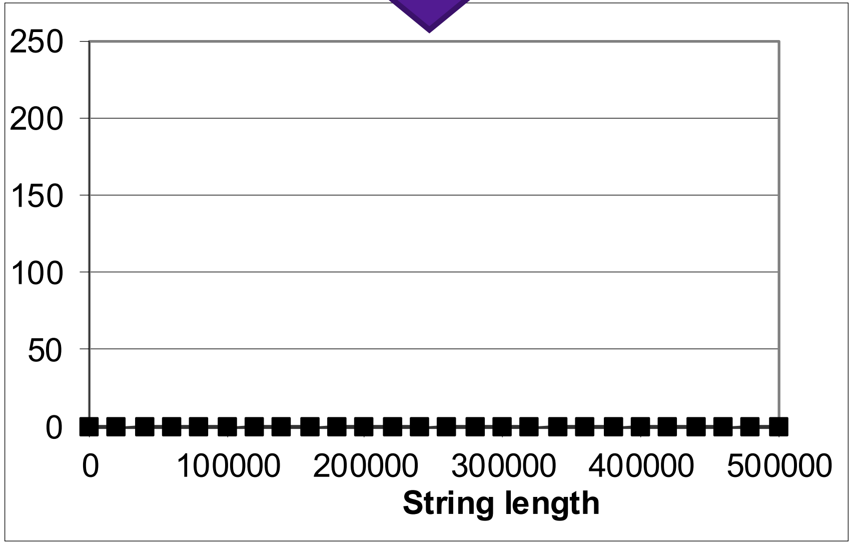
Example: Lowering Case

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





Machine Independent Optimization

- Compilers optimize assembly code
  - Dead code elimination
  - Code motion
  - Factoring out common subexpressions
  - Loop elimination
  - Reduction in Strength

- Optimization blockers:
  - Aliasing
    - Use local variables
  - Procedure calls
    - Move them yourself

Exercise: Code-Level Optimizations

```
void combine1(vec* v, data_t* dest){

  *dest = IDENT;

  for(long i = 0; i < vec_length(v); i++) {
    data_t* val = get_vec_elem(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?

Exercise: Code-Level Optimizations

```
void combine1(vec* v, data_t*
              dest){

  *dest = IDENT;


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

}
```

Code-Level Optimizations

```
void combine2(vec* v, data_t* dest){

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

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

Loop Unrolling

```
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 sums[],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];
  }
}
```

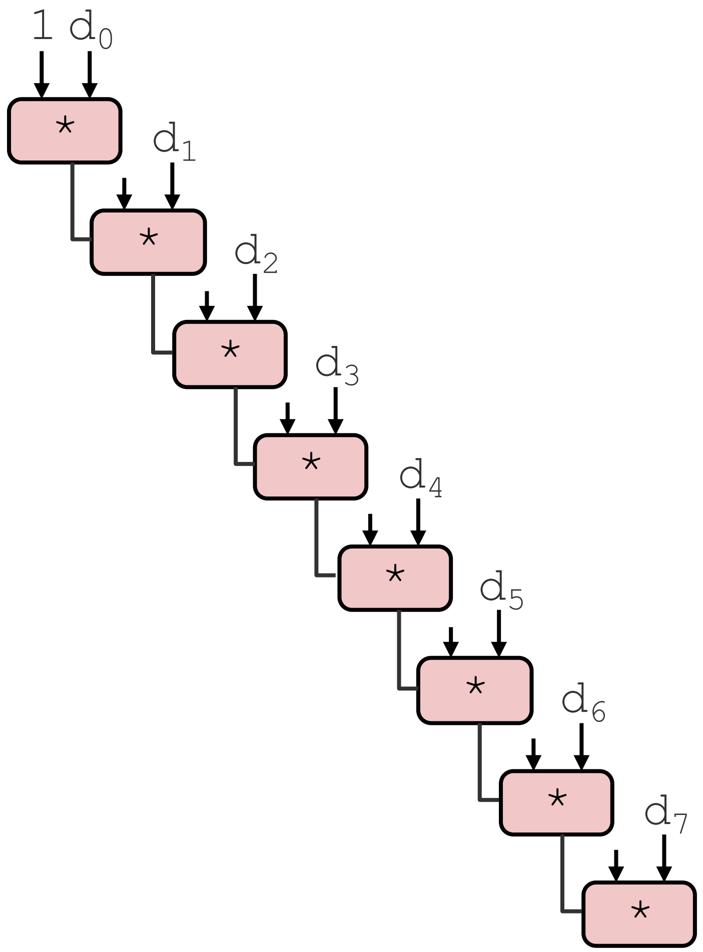
Combine with Unrolling

```
void unroll2_combine(vec* 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;
    /* Combine 2 elements at a time */
    for(long i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
```

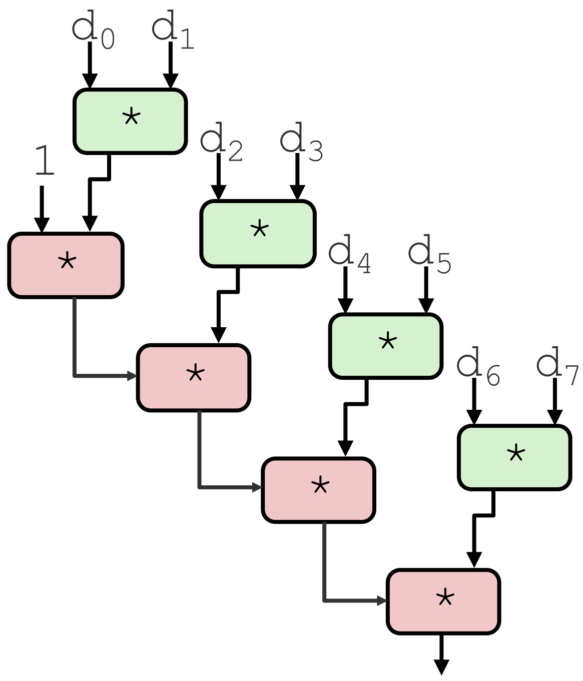
Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

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]);
```

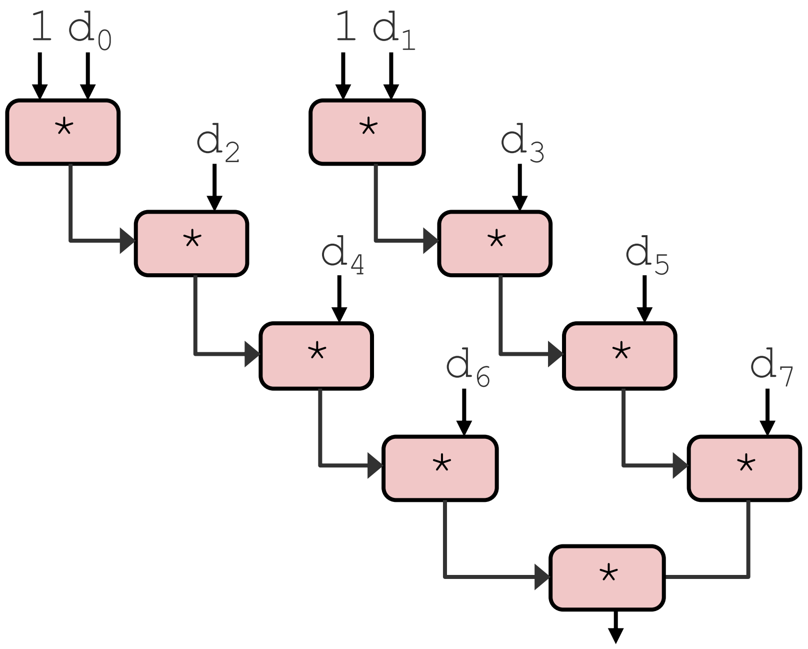
2 func. units for FP *
2 func. units for load

pipelined processor

4 func. units for int +
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

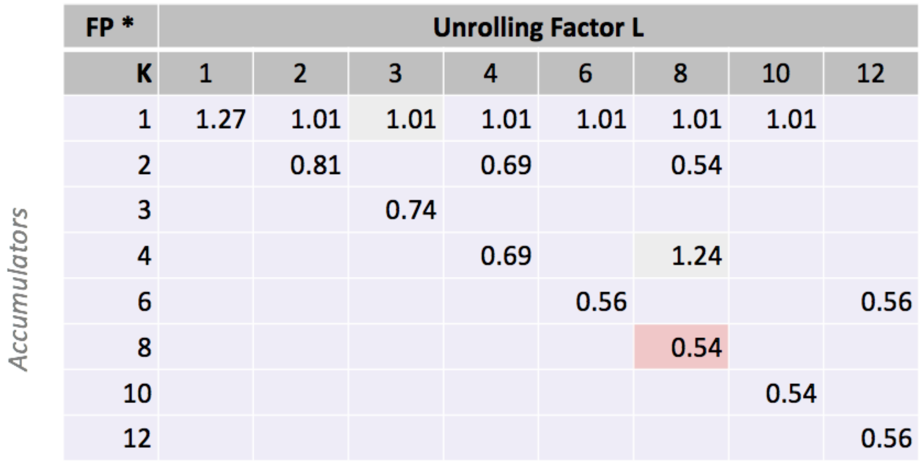
Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput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

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

- Int + makes use of two load units
- for Int *, FP +, FP *, speedup similar to unroll with reassociation

Machine-Dependent Optimization

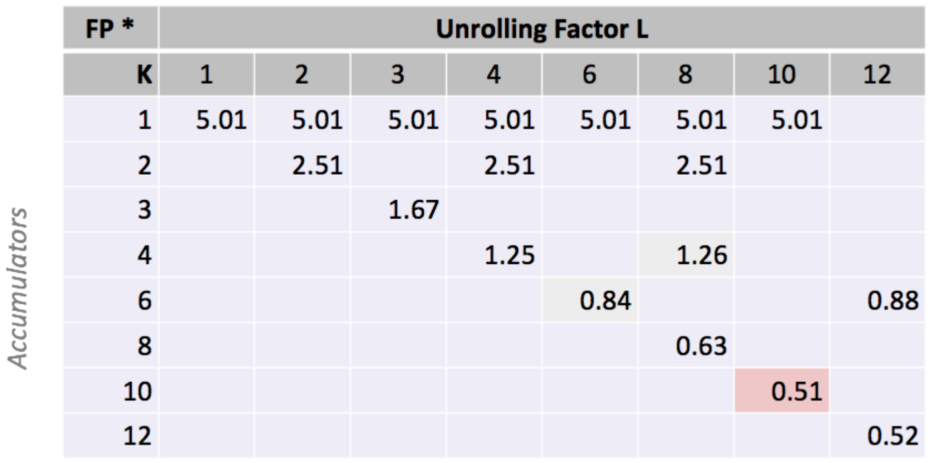
Integer Addition



Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Float Multiplication



Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Machine-Dependent Optimization

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	0.51
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