

PROGRAM OPTIMIZATION

CS 045

Computer Organization and
Architecture

Prof. Donald J. Patterson

Adapted from Bryant and O'Hallaron,
Computer Systems:
A Programmer's Perspective, Third Edition



PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

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, procedures, and loops
- Must understand system to optimize performance
 - How programs are compiled and executed
 - How modern processors + memory systems operate
 - How to measure program performance and identify bottlenecks
 - How to improve performance without destroying code modularity and generality



PROGRAM OPTIMIZATION

OPTIMIZING COMPILERS

- Provide efficient mapping of program to machine
 - 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
- Have difficulty overcoming “optimization blockers”
 - potential memory aliasing
 - potential procedure side-effects



PROGRAM OPTIMIZATION

LIMITATIONS OF OPTIMIZING COMPILERS

- **Operate under fundamental constraint**
 - Must not cause any change in program behavior
 - Except, possibly when program making use of nonstandard language features
 - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
 - e.g., Data ranges may be more limited than variable types suggest
- **Most analysis is performed only within procedures**
 - Whole-program analysis is too expensive in most cases
 - Newer versions of GCC do interprocedural analysis within individual files
 - But, not between code in different files
- **Most analysis is based only on static information**
 - Compiler has difficulty anticipating run-time inputs
- **When in doubt, the compiler must be conservative**

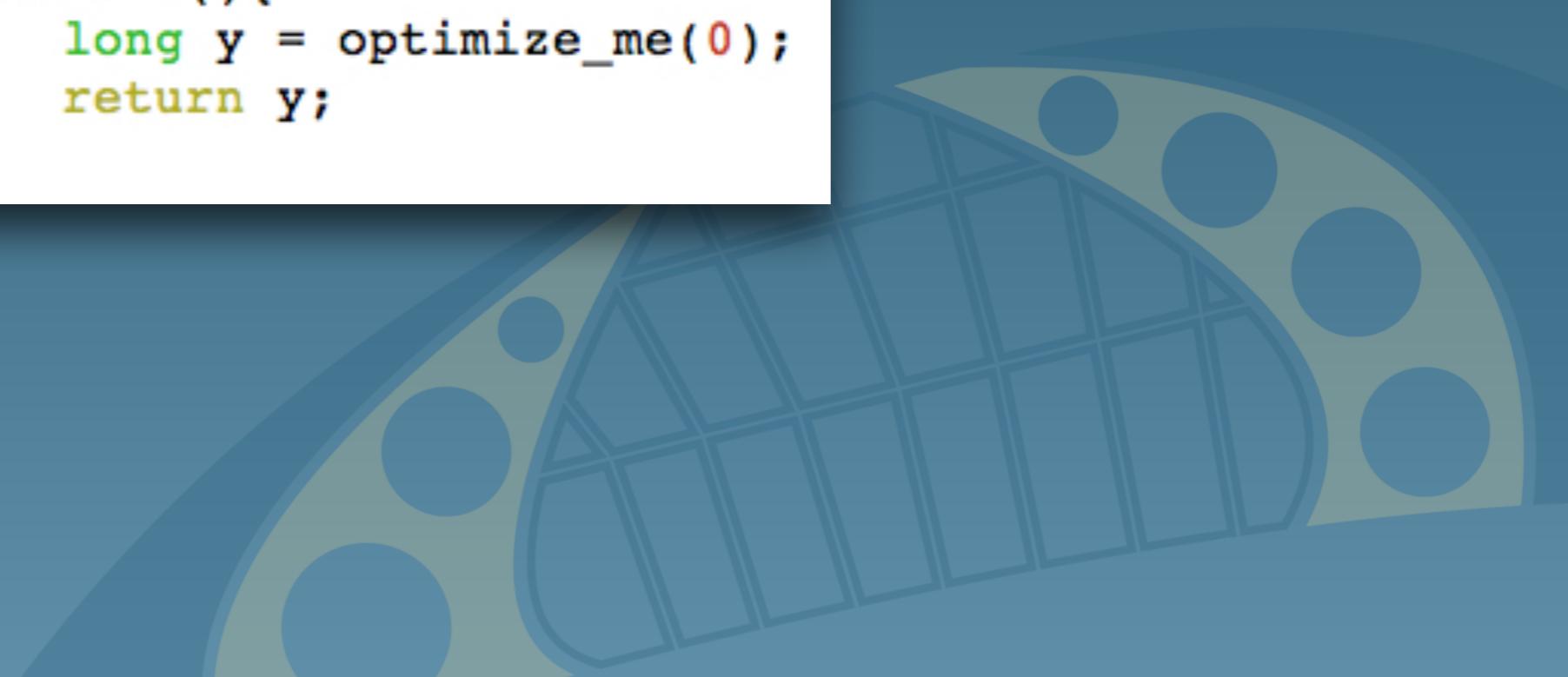


EXAMPLE PROGRAM OPTIMIZATION

```
#include <stdio.h>

long optimize_me(long x){
    while(x < 0x2345){
        x++;
    }
    return x;
}

int main(){
    long y = optimize_me(0);
    return y;
}
```



EXAMPLE PROGRAM OPTIMIZATION

```
#include <stdio.h>

long optimize_me(long x){
    while(x < 0x2345){
        x++;
    }
    return x;
}
```

```
int main(){
    long y = optimize_me(0);
    return y;
}
```

```
gcc -O0 -o optimize_0 optimize.c
objdump -d optimize_0 > optimize_0.dis
```

0000000000400474 <optimize_me>:		
400474:	55	push %rbp
400475:	48 89 e5	mov %rsp,%rbp
400478:	48 89 7d f8	mov %rdi,-0x8(%rbp)
40047c:	eb 05	jmp 400483 <optimize_me+0xf>
40047e:	48 83 45 f8 01	addq \$0x1,-0x8(%rbp)
400483:	48 81 7d f8 44 23 00	cmpq \$0x2344,-0x8(%rbp)
40048a:	00	
40048b:	7e f1	jle 40047e <optimize_me+0xa>
40048d:	48 8b 45 f8	mov -0x8(%rbp),%rax
400491:	c9	leaveq
400492:	c3	retq
0000000000400493 <main>:		
400493:	55	push %rbp
400494:	48 89 e5	mov %rsp,%rbp
400497:	48 83 ec 10	sub \$0x10,%rsp
40049b:	bf 00 00 00 00	mov \$0x0,%edi
4004a0:	e8 cf ff ff ff	callq 400474 <optimize_me>
4004a5:	48 89 45 f8	mov %rax,-0x8(%rbp)
4004a9:	48 8b 45 f8	mov -0x8(%rbp),%rax
4004ad:	c9	leaveq
4004ae:	c3	retq

EXAMPLE PROGRAM OPTIMIZATION

```
#include <stdio.h>

long optimize_me(long x){
    while(x < 0x2345){
        x++;
    }
    return x;
}
```

```
int main(){
    long y = optimize_me(0);
    return y;
}
```

```
push    %rbp
mov     %rsp,%rbp
mov     %rdi,-0x8(%rbp)
jmp     400483 <optimize_me+0xf>
addq    $0x1,-0x8(%rbp)
cmpq    $0x2344,-0x8(%rbp)
jle    40047e <optimize_me+0xa>
mov     -0x8(%rbp),%rax
leaveq 
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub    $0x10,%rsp
mov     $0x0,%edi
callq  400474 <optimize_me>
mov     %rax,-0x8(%rbp)
mov     -0x8(%rbp),%rax
leaveq 
retq
```

```
gcc -O1 -o optimize_1 optimize.c
objdump -d optimize_1 > optimize_1.dis

0000000000400474 <optimize_me>:
    400474:    48 89 f8
    400477:    48 81 ff 44 23 00 00
    40047e:    ba 45 23 00 00
    400483:    48 0f 4e c2
    400487:    c3

0000000000400488 <main>:
    400488:    bf 00 00 00 00
    40048d:    e8 e2 ff ff ff
    400492:    f3 c3

    mov     %rdi,%rax
    cmp     $0x2344,%rdi
    mov     $0x2345,%edx
    cmovle %rdx,%rax
    retq

    mov     $0x0,%edi
    callq  400474 <optimize_me>
    repz   retq
```

EXAMPLE PROGRAM OPTIMIZATION

```
#include <stdio.h>

long optimize_me(long x){
    while(x < 0x2345){
        x++;
    }
    return x;
}

int main(){
    long y = optimize_me(0);
    return y;
}

push    %rbp
mov     %rsp,%rbp
mov     %rdi,-0x8(%rbp)
jmp     400483 <optimize_me+0xf>
addq   $0x1,-0x8(%rbp)
cmpq   $0x2344,-0x8(%rbp)

jle    40047e <optimize_me+0xa>
mov    -0x8(%rbp),%rax
leaveq
retq

push    %rbp
mov     %rsp,%rbp
sub    $0x10,%rsp
mov     $0x0,%edi
callq  400474 <optimize_me>
mov    %rax,-0x8(%rbp)
mov    -0x8(%rbp),%rax
leaveq
retq
```

```
gcc -O2 -o optimize_2 optimize.c
objdump -d optimize_2 > optimize_2.dis

0000000000400480 <optimize_me>:
400480: 48 89 f8
400483: 48 81 ff 44 23 00 00
40048a: ba 45 23 00 00
40048f: 48 0f 4e c2
400493: c3
400494: 66 66 66 2e 0f 1f 84
40049b: 00 00 00 00 00 00

00000000004004a0 <main>:
4004a0: b8 45 23 00 00
4004a5: c3

mov    %rdi,%rax
cmp   $0x2344,%rdi
mov    $0x2345,%edx
cmovle %rdx,%rax
retq

data32 data32 nopw %cs:0x0(%rax)
mov    $0x2345,%eax
retq

mov    %rdi,%rax
cmp   $0x2344,%rdi
mov    $0x2345,%edx
cmovle %rdx,%rax
retq

mov    $0x0,%edi
callq 400474 <optimize_me>
repz retq
```

EXAMPLE PROGRAM OPTIMIZATION

```
#include <stdio.h>

long optimize_me(long x){
    while(x < 0x2345){
        x++;
    }
    return x;
}
```

```
int main(){
    long y = optimize_me(0);
    return y;
}
```

```
push    %rbp
mov     %rsp,%rbp
mov     %rdi,-0x8(%rbp)
jmp    400483 <optimize_me+0xf>
addq    $0x1,-0x8(%rbp)
cmpq    $0x2344,-0x8(%rbp)
jle    40047e <optimize_me+0xa>
mov     -0x8(%rbp),%rax
leaveq
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub    $0x10,%rsp
mov     $0x0,%edi
callq  400474 <optimize_me>
mov     %rax,-0x8(%rbp)
mov     -0x8(%rbp),%rax
leaveq
retq
```

```
o optimize_3 optimize.c
d optimize_3 > optimize_3.dis
0400480 <optimize_me>:
    48 89 f8
    48 81 ff 44 23 00 00
    ba 45 23 00 00
    48 0f 4e c2
    c3
    66 66 66 2e 0f 1f 84
    00 00 00 00 00

    mov    %rdi,%rax
    cmp    $0x2344,%rdi
    mov    $0x2345,%edx
    cmovle %rdx,%rax
    retq
    data32 data32 nopw %cs:0x0(%rax,%rax,1)
```

```
004a0 <main>:
    b8 45 23 00 00
    c3

    mov    $0x2345,%eax
    retq
```

```
mov    %rdi,%rax
cmp    $0x2344,%rdi
mov    $0x2345,%edx
cmovle %rdx,%rax
retq
```

```
mov    $0x0,%edi
callq 400474 <optimize_me>
repz  retq
```

```
mov    %rdi,%rax
cmp    $0x2344,%rdi
mov    $0x2345,%edx
cmovle %rdx,%rax
retq
data32 data32 nopw %cs:0x0(%rax,%rax,
```

```
mov    $0x2345,%eax
retq
```

-O0

-O1

-O2

-O3

EXAMPLE PROGRAM OPTIMIZATION

```
#include <stdio.h>

long optimize_me(long x){
    while(x < 0x2345){
        x++;
    }
    return x;
}

int main(){
    long y = optimize_me(0);
    return y;
}
```

```
push    %rbp
mov     %rsp,%rbp
mov     %rdi,-0x8(%rbp)
jmp     400483 <optimize_me+0xf>
addq    $0x1,-0x8(%rbp)
cmpq    $0x2344,-0x8(%rbp)
jle     40047e <optimize_me+0xa>
mov     -0x8(%rbp),%rax
leaveq 
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub    $0x10,%rsp
mov     $0x0,%edi
callq  400474 <optimize_me>
mov     %rax,-0x8(%rbp)
mov     -0x8(%rbp),%rax
leaveq 
retq
```

-O0

```
mov     %rdi,%rax
cmp    $0x2344,%rdi
mov     $0x2345,%edx
cmovle %rdx,%rax
retq
```



```
mov     $0x0,%edi
callq  400474 <optimize_me>
repz   retq
```

-O1

```
mov     %rdi,%rax
cmp    $0x2344,%rdi
mov     $0x2345,%edx
cmovle %rdx,%rax
retq
data32 data32 nopw %cs:0x0(%rax,%rax,
```

```
mov     $0x2345,%eax
retq
```

-O2-O3

PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

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];
```

PROGRAM OPTIMIZATION

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];
```

PROGRAM OPTIMIZATION

COMPILER-GENERATED CODE MOTION (-O1)

```
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];
}
```



PROGRAM OPTIMIZATION

COMPILER-GENERATED CODE MOTION (-O1)

```
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];
}
```



```
set_row:
    testq    %rcx, %rcx          # Test n
    jle      .L1                # If 0, goto done
    imulq    %rcx, %rdx          # ni = n*i
    leaq     (%rdi,%rdx,8), %rdx # rowp = A + ni*8
    movl    $0, %eax            # j = 0
.L3:
    movsd    (%rsi,%rax,8), %xmm0 # t = b[j]
    movsd    %xmm0, (%rdx,%rax,8) # M[A+ni*8 + j*8] = t
    addq    $1, %rax             # j++
    cmpq    %rcx, %rax           # j:n
    jne     .L3                # if !=, goto loop
.L1:
    rep ; ret                  # done:
```

PROGRAM OPTIMIZATION

COMPILER-GENERATED CODE MOTION (-O1)

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

set_row:

```
    testq    %rcx, %rcx
    jle      .L1
    imulq    %rcx, %rdx
    leaq     (%rdi,%rdx,8), %rdx
    movl     $0, %eax
.L3:
    movsd    (%rsi,%rax,8), %xmm0
    movsd    %xmm0, (%rdx,%rax,8)
    addq    $1, %rax
    cmpq    %rcx, %rax
    jne     .L3
.L1:
    rep ; ret
```

Test n
If 0, goto done
ni = n*i
rowp = A + ni*8
j = 0
loop:
t = b[j]
M[A+ni*8 + j*8] = t
j++
j:n
if !=, goto loop
done:

PROGRAM OPTIMIZATION

REDUCTION IN STRENGTH

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



PROGRAM OPTIMIZATION

REDUCTION IN STRENGTH

- Replace costly operation with simpler one
 - Shift, add instead of multiply or divide
 - $16*x$ becomes $x << 4$
 - 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++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```



PROGRAM OPTIMIZATION

REDUCTION IN STRENGTH

- Replace costly operation with simpler one
 - Shift, add instead of multiply or divide
 - $16*x$ becomes $x << 4$
 - 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++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + 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;  
}
```

PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- GCC will do this with -O1



PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- 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;
```

PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- 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: $i*n$, $(i-1)*n$, $(i+1)*n$

PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- 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: $i*n$, $(i-1)*n$, $(i+1)*n$

```
leaq    1(%rsi), %rax # i+1
leaq    -1(%rsi), %r8  # i-1
imulq   %rcx, %rsi    # i*n
imulq   %rcx, %rax    # (i+1)*n
imulq   %rcx, %r8      # (i-1)*n
addq    %rdx, %rsi     # i*n+j
addq    %rdx, %rax     # (i+1)*n+j
addq    %rdx, %r8      # (i-1)*n+j
```



PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- 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;
```

```
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;
```

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

```
leaq  1(%rsi), %rax # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %rsi    # i*n
imulq %rcx, %rax   # (i+1)*n
imulq %rcx, %r8    # (i-1)*n
addq  %rdx, %rsi    # i*n+j
addq  %rdx, %rax   # (i+1)*n+j
addq  %rdx, %r8    # (i-1)*n+j
```



PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- 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: $i*n$, $(i-1)*n$, $(i+1)*n$

```
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: $i*n$

```
leaq    1(%rsi), %rax # i+1
leaq    -1(%rsi), %r8  # i-1
imulq   %rcx, %rsi    # i*n
imulq   %rcx, %rax    # (i+1)*n
imulq   %rcx, %r8      # (i-1)*n
addq    %rdx, %rsi     # i*n+j
addq    %rdx, %rax     # (i+1)*n+j
addq    %rdx, %r8      # (i-1)*n+j
```



PROGRAM OPTIMIZATION

SHARE COMMON SUBEXPRESSIONS

- Reuse portions of expressions
- 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: $i*n$, $(i-1)*n$, $(i+1)*n$

```
leaq  1(%rsi), %rax # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %rsi   # i*n
imulq %rcx, %rax   # (i+1)*n
imulq %rcx, %r8    # (i-1)*n
addq  %rdx, %rsi   # i*n+j
addq  %rdx, %rax   # (i+1)*n+j
addq  %rdx, %r8    # (i-1)*n+j
```

```
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: $i*n$

```
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
```



PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS

- OPTIMIZATION BLOCKERS

- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS



PROGRAM OPTIMIZATION

OPTIMIZATION BLOCKER #1: PROCEDURE CALLS

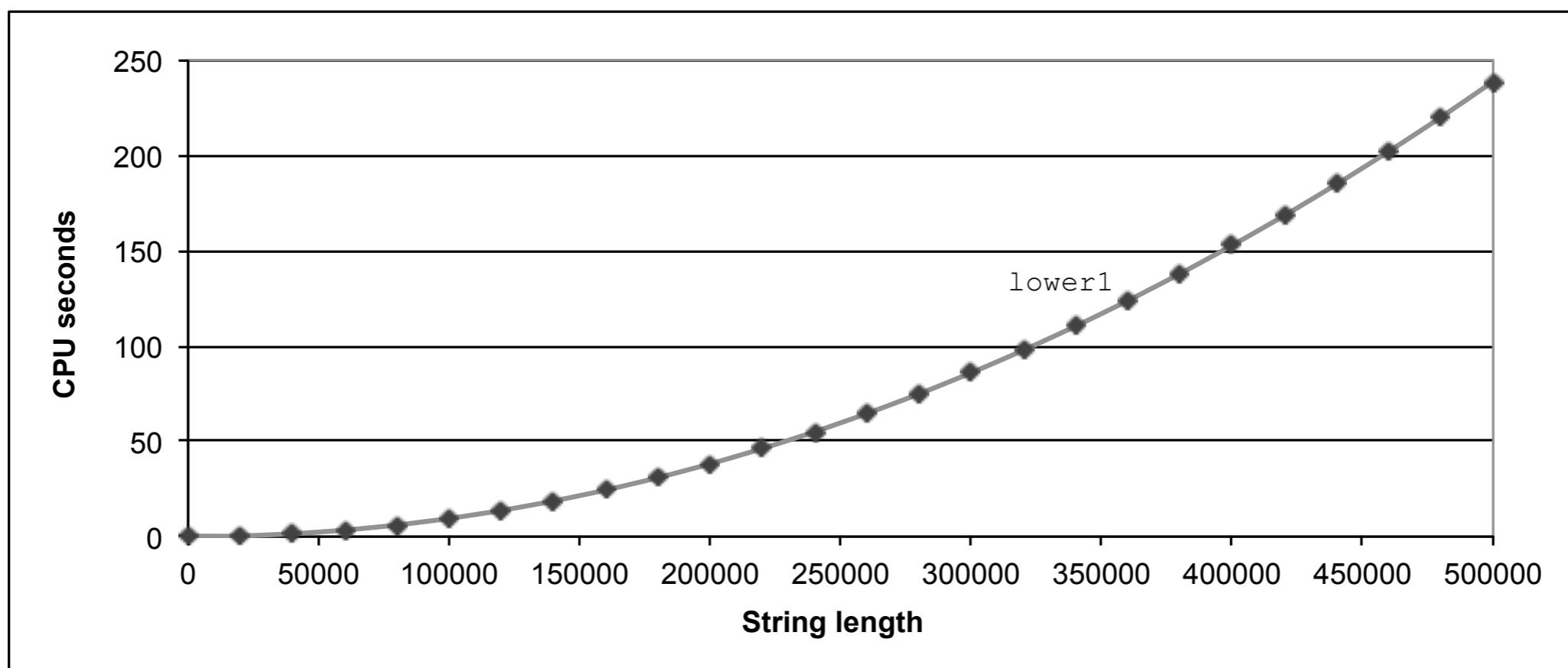
- Procedure to Convert String to Lower Case

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

PROGRAM OPTIMIZATION

LOWER CASE CONVERSION PERFORMANCE

- Time quadruples when double string length
- Quadratic performance



PROGRAM OPTIMIZATION

CONVERT LOOP TO GOTO FORM

- `strlen` executed every iteration

```
void lower(char *s)
{
    size_t 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:
}
```



PROGRAM OPTIMIZATION

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



PROGRAM OPTIMIZATION

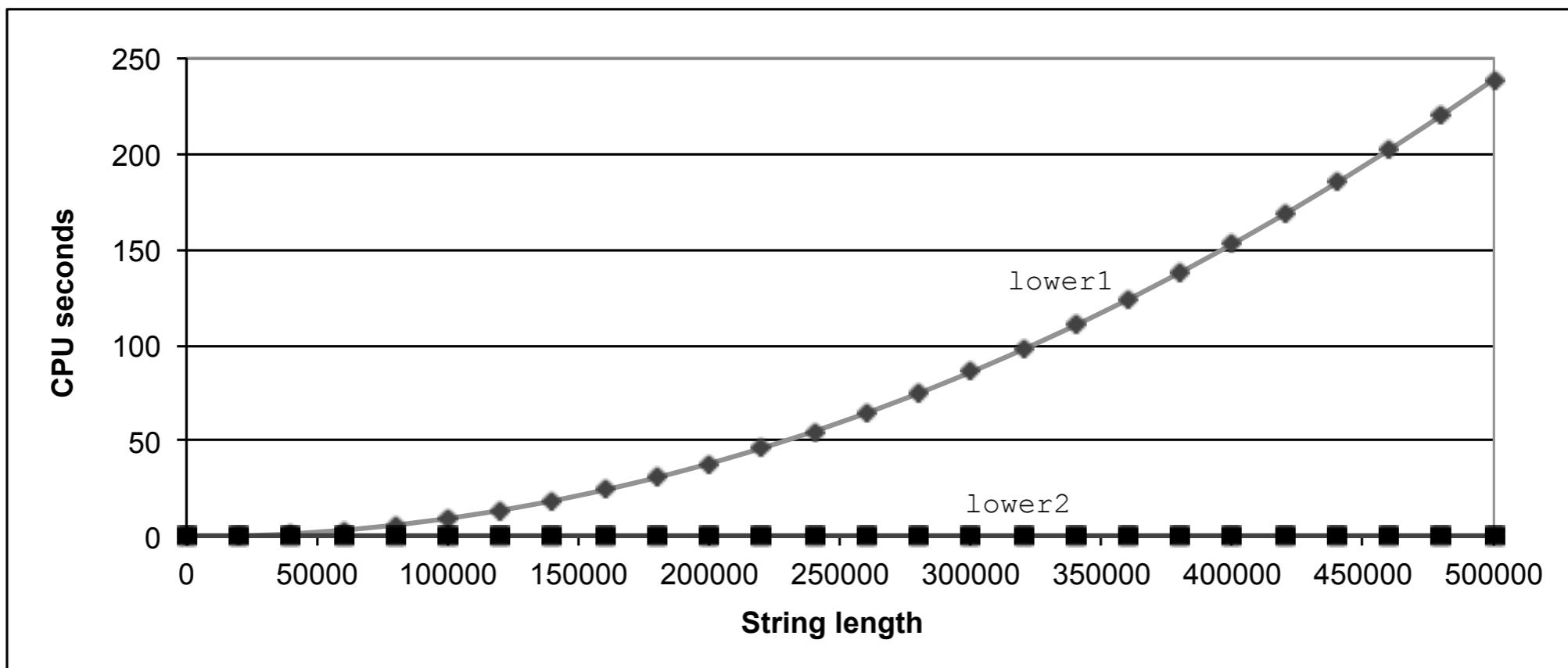
IMPROVING PERFORMANCE

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

PROGRAM OPTIMIZATION

LOWER CASE CONVERSION PERFORMANCE



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

PROGRAM OPTIMIZATION

OPTIMIZATION BLOCKER: PROCEDURE CALLS

- Why couldn't compiler move `strlen` out of inner loop?
 - 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
 - Procedure lower could interact with `strlen`
- **Warning:**
 - Compiler treats procedure call as a black box
 - Weak optimizations near them
- Remedies:
 - Use of inline functions
 - GCC does this with -O1
 - Within single file
 - Do your own code motion

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



PROGRAM OPTIMIZATION

MEMORY MATTERS

```
/* Sum rows is of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long 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 $b[i]$ on every iteration
- Why couldn't compiler optimize this away?



PROGRAM OPTIMIZATION

MEMORY MATTERS

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

sum_rows1 inner loop
.L4:

movsd	(%rsi,%rax,8), %xmm0	# FP load
addsd	(%rdi), %xmm0	# FP add
movsd	%xmm0, (%rsi,%rax,8)	# FP store
addq	\$8, %rdi	
cmpq	%rcx, %rdi	
jne	.L4	

- Code updates b[i] on every iteration
- Why couldn't compiler optimize this away?

PROGRAM OPTIMIZATION

MEMORY ALIASING

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long 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]

- Code updates $b[i]$ on every iteration
- Must consider possibility that these updates will affect program behavior

PROGRAM OPTIMIZATION

REMOVING ALIASING

```
/* Sum rows is of n X n matrix a  
   and store in vector b */  
void sum_rows1(double *a, double *b, long n) {  
    long i, j;  
    for (i = 0; i < n; i++) {  
        b[i] = 0;  
        for (j = 0; j < n; j++)  
            b[i] += a[i*n + j];  
    }  
}
```

- No need to store intermediate results

PROGRAM OPTIMIZATION

REMOVING ALIASING

```
/* Sum rows is of n X n matrix a  
   and store in vector b */  
void sum_rows1(double *a, double *b, long n) {  
    long i, j;  
    for (i = 0; i < n; i++) {  
        b[i] = 0;  
        for (j = 0; j < n; j++)  
            b[i] += a[i*n + j];  
    }  
}
```

```
# sum_rows1 inner loop  
.L4:  
    movsd  (%rsi,%rax,8), %xmm0      # FP load  
    addsd  (%rdi), %xmm0             # FP add  
    movsd  %xmm0, (%rsi,%rax,8)      # FP store  
    addq   $8, %rdi  
    cmpq   %rcx, %rdi  
    jne    .L4
```

- No need to store intermediate results

PROGRAM OPTIMIZATION

REMOVING ALIASING

```
/* Sum rows is of n X n matrix a  
and store in vector b */  
void sum_rows2(double *a, double *b, long n) {  
    long 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;  
    }  
}
```

```
/* Sum rows is of n X n matrix a  
and store in vector b */  
void sum_rows1(double *a, double *b, long n) {  
    long i, j;  
    for (i = 0; i < n; i++) {  
        b[i] = 0;  
        for (j = 0; j < n; j++)  
            b[i] += a[i*n + j];  
    }  
}
```

```
# sum_rows1 inner loop  
.L4:  
    movsd  (%rsi,%rax,8), %xmm0      # FP load  
    addsd  (%rdi), %xmm0             # FP add  
    movsd  %xmm0, (%rsi,%rax,8)      # FP store  
    addq   $8, %rdi  
    cmpq   %rcx, %rdi  
    jne    .L4
```

- No need to store intermediate results

PROGRAM OPTIMIZATION

REMOVING ALIASING

```
/* Sum rows is of n X n matrix a  
and store in vector b */  
void sum_rows2(double *a, double *b, long n) {  
    long 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;  
    }  
}
```

```
/* Sum rows is of n X n matrix a  
and store in vector b */  
void sum_rows1(double *a, double *b, long n) {  
    long i, j;  
    for (i = 0; i < n; i++) {  
        b[i] = 0;  
        for (j = 0; j < n; j++)  
            b[i] += a[i*n + j];  
    }  
}
```

```
# sum_rows2 inner loop  
.L10:  
    addsd  (%rdi), %xmm0      # FP load + add  
    addq   $8, %rdi  
    cmpq   %rax, %rdi  
    jne    .L10
```

```
# sum_rows1 inner loop  
.L4:  
    movsd  (%rsi,%rax,8), %xmm0      # FP load  
    addsd  (%rdi), %xmm0      # FP add  
    movsd  %xmm0, (%rsi,%rax,8)      # FP store  
    addq   $8, %rdi  
    cmpq   %rcx, %rdi  
    jne    .L4
```

- No need to store intermediate results

PROGRAM OPTIMIZATION

OPTIMIZATION BLOCKER: MEMORY ALIASING

- Aliasing
 - Two different memory references specify single location
 - Easy to have happen in C
 - Since allowed to do address arithmetic
 - Direct access to storage structures
 - Get in habit of introducing local variables
 - Accumulating within loops
 - Your way of telling compiler not to check for aliasing



PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

- OVERVIEW
- GENERALLY USEFUL OPTIMIZATIONS
- OPTIMIZATION BLOCKERS
- EXPLOITING INSTRUCTION-LEVEL PARALLELISM
- DEALING WITH CONDITIONALS

PROGRAM OPTIMIZATION

EXPLOITING INSTRUCTION-LEVEL PARALLELISM

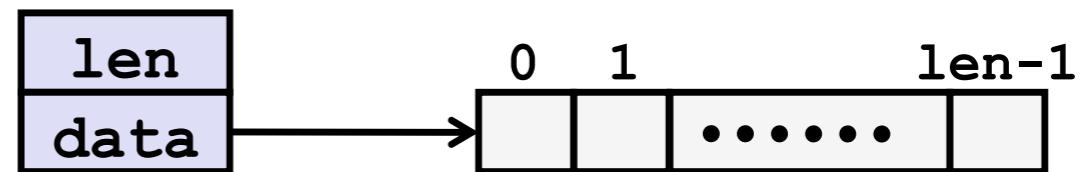
- Need general understanding of modern processor design
 - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
 - Compilers often cannot make these transformations
 - Lack of associativity and distributivity in floating-point arithmetic



PROGRAM OPTIMIZATION

BENCHMARK EXAMPLE: DATA TYPE FOR VECTORS

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



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

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

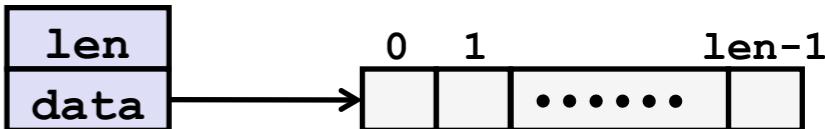
PROGRAM OPTIMIZATION

BENCHMARK COMPUTATION

Compute sum or product of vector elements

```
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;
    }
}
```

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



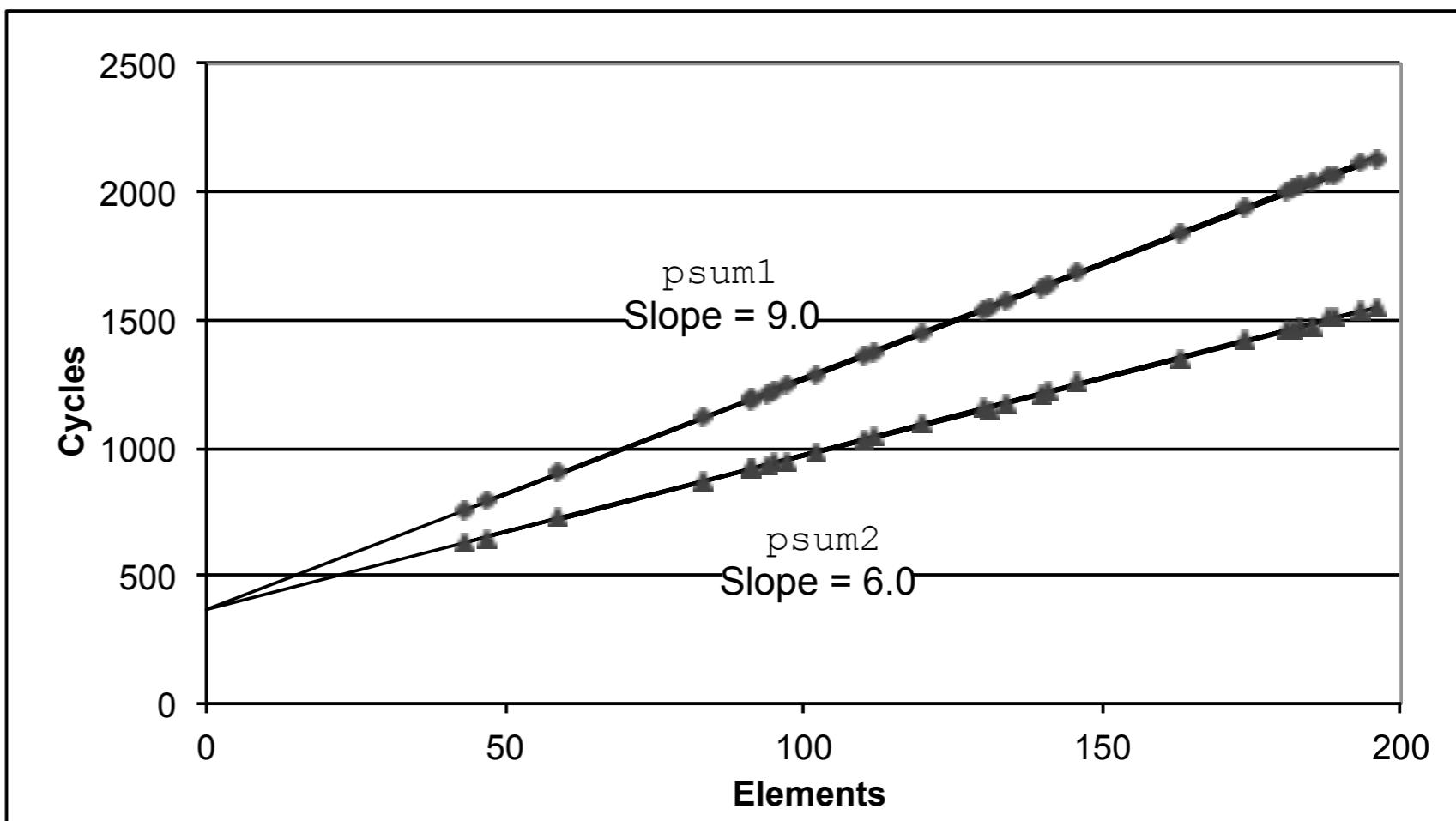
```
/* retrieve vector element
   and store at val */
int get_vec_element
(*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

- Data Types
 - Use different declarations for data_t
 - int
 - long
 - float
 - double
- Operations
 - Use different definitions of OP and IDENT
 - + / 0
 - * / 1

PROGRAM OPTIMIZATION

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**
- $T = \text{CPE} \cdot n + \text{Overhead}$
- CPE is slope of line



PROGRAM OPTIMIZATION

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	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14

PROGRAM OPTIMIZATION

EFFECT OF BASIC OPTIMIZATIONS

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long 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;
}
```

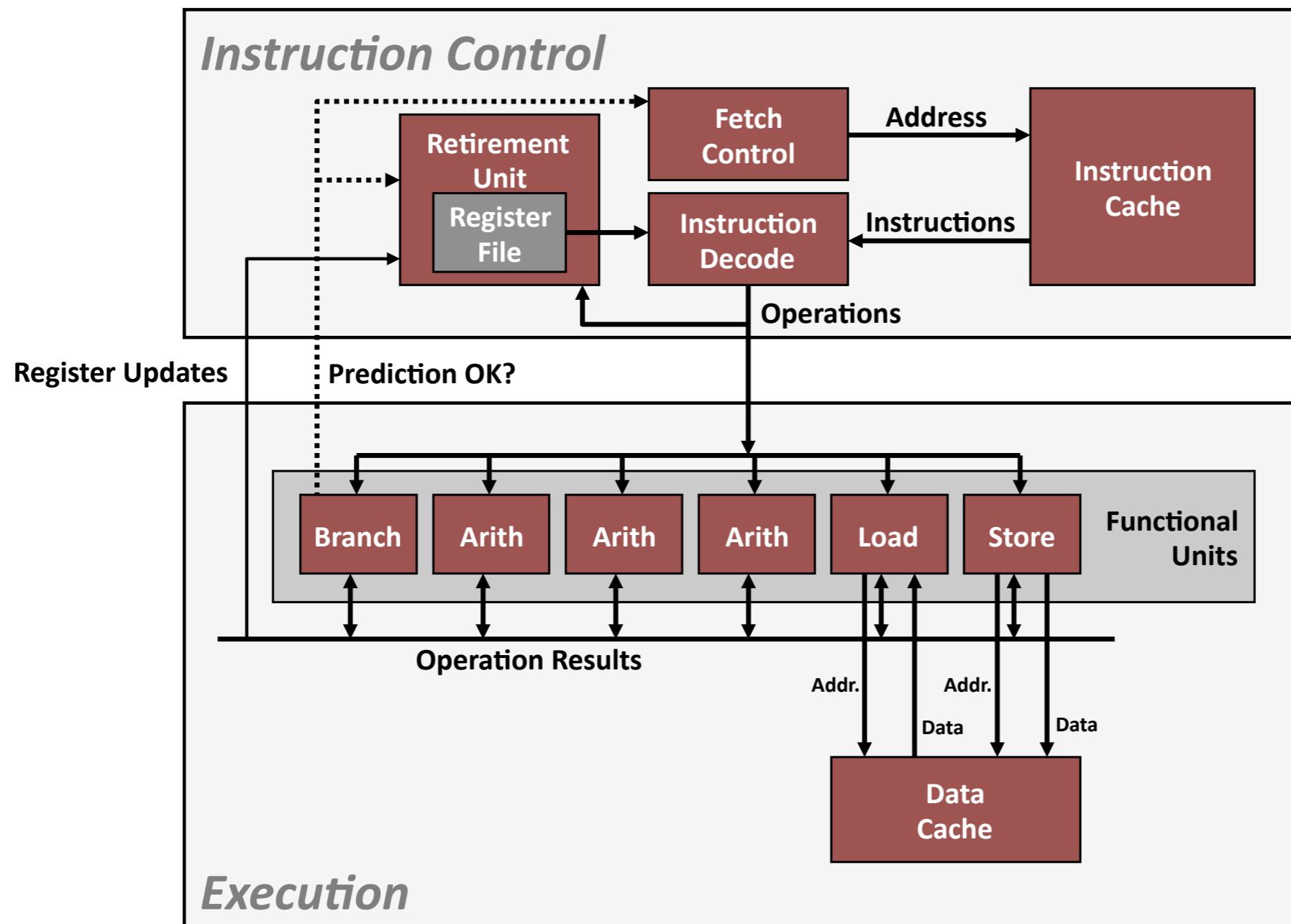
Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 -01	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- Eliminates sources of overhead in loop

PROGRAM OPTIMIZATION

MODERN CPU DESIGN



PROGRAM OPTIMIZATION

SUPERSCALAR PROCESSOR

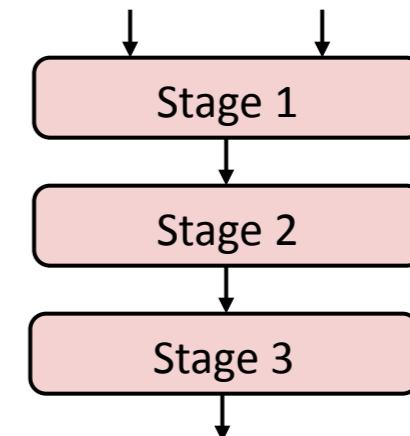
- **Definition:** A superscalar processor can issue and execute **multiple instructions in one cycle**. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- **Benefit:** without programming effort, superscalar processor can take advantage of the **instruction level parallelism** that most programs have
- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)



PROGRAM OPTIMIZATION

PIPELINED FUNCTIONAL UNITS

```
long mult_eg(long a, long b, long c) {  
    long p1 = a*b;  
    long p2 = a*c;  
    long p3 = p1 * p2;  
    return p3;  
}
```



	Time							
	1	2	3	4	5	6	7	
Stage 1	a*b	a*c			p1*p2			
Stage 2		a*b	a*c			p1*p2		
Stage 3			a*b	a*c			p1*p2	

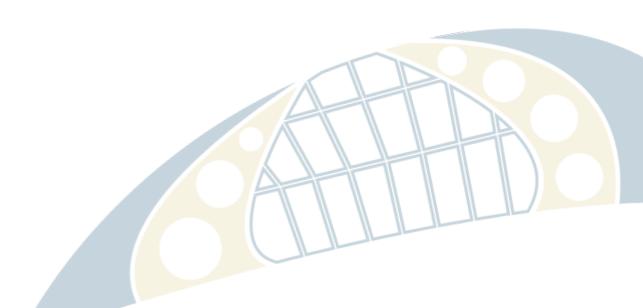
- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to $i+1$
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

PROGRAM OPTIMIZATION

HASWELL CPU

- 8 Total Functional Units
- Multiple instructions can execute in parallel
 - 2 load, with address computation
 - 1 store, with address computation
 - 4 integer
 - 2 FP multiply
 - 1 FP add
 - 1 FP divide
- Some instructions take > 1 cycle, but can be pipelined

Instruction	Latency	Cycles/Issue
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	3-30	3-30
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	3-15	3-15



PROGRAM OPTIMIZATION

X86-64 COMPIRATION OF COMBINE4

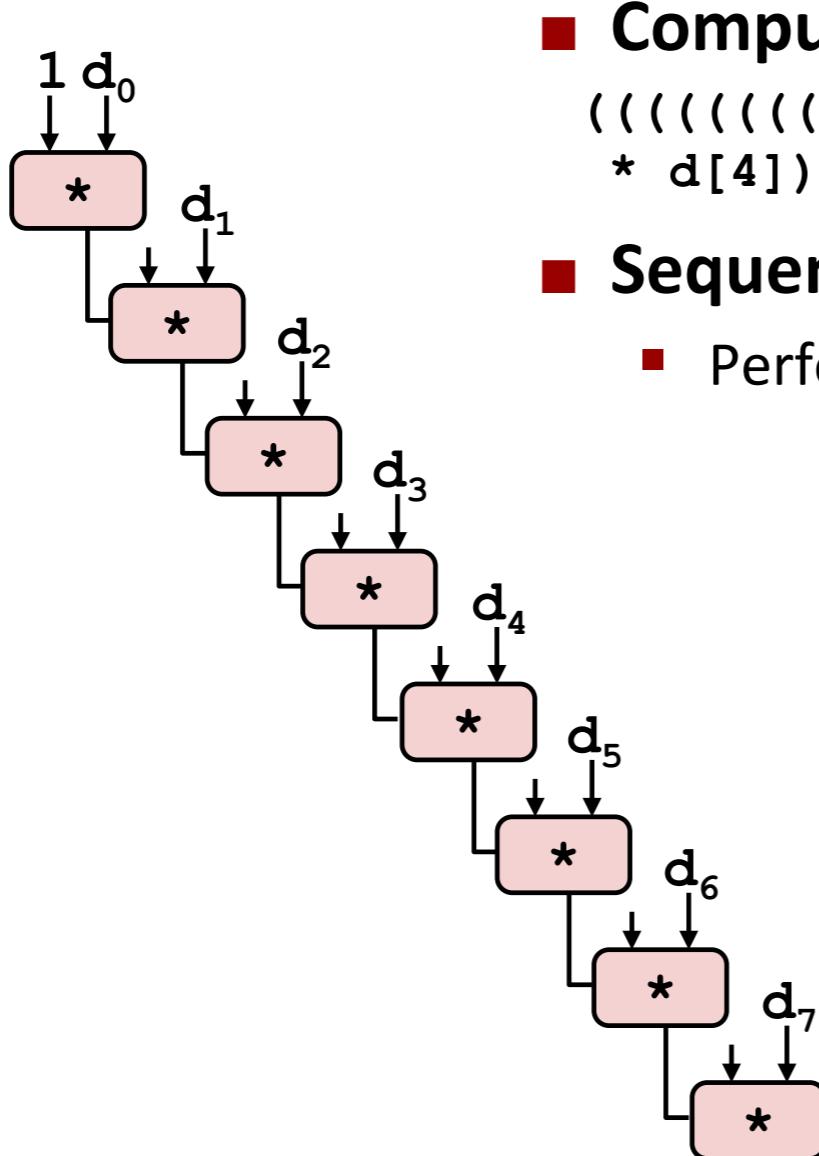
- Inner Loop (Case: Integer Multiply)

```
.L519:                                # Loop:  
    imull (%rax,%rdx,4), %ecx    # t = t * d[i]  
    addq $1, %rdx                # i++  
    cmpq %rdx, %rbp              # Compare length:i  
    jg .L519                     # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

PROGRAM OPTIMIZATION

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

- Performance: determined by latency of OP

PROGRAM OPTIMIZATION

LOOP UNROLLING (2X1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    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];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

PROGRAM OPTIMIZATION

EFFECT OF LOOP UNROLLING

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    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];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Helps integer add
 - Achieves latency bound
- Others don't improve. Why?
 - Still sequential dependency

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

PROGRAM OPTIMIZATION

LOOP UNROLLING WITH REASSOCIATION (2X1A)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    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]);
    }
    /* 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?
- Yes, for FP. Why?

PROGRAM OPTIMIZATION

EFFECT OF REASSOCIATION

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	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

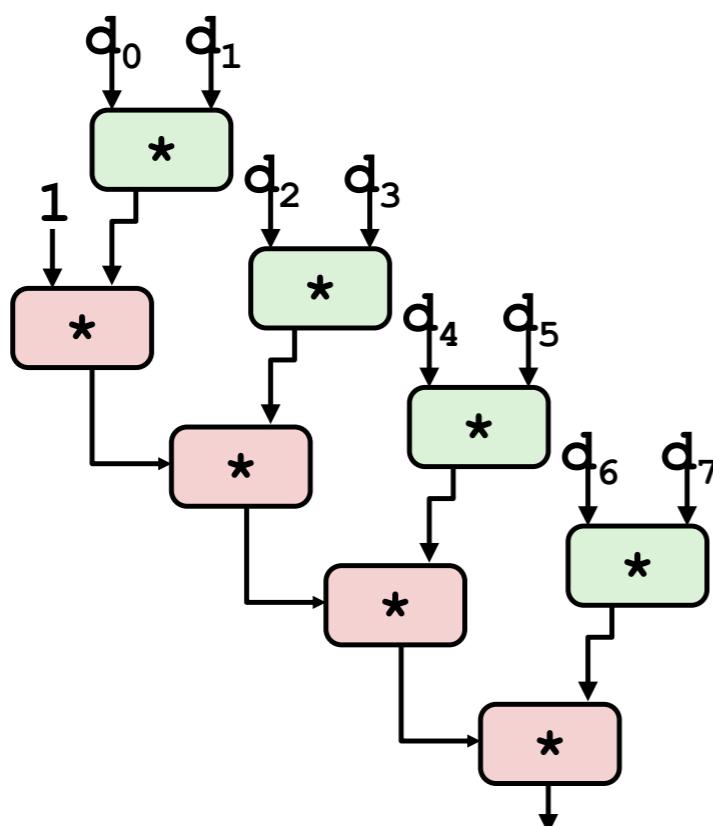
4 func. units for int +
2 func. units for load

- Why is that? (next slide)

PROGRAM OPTIMIZATION

REASSOCIATED COMPUTATION

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

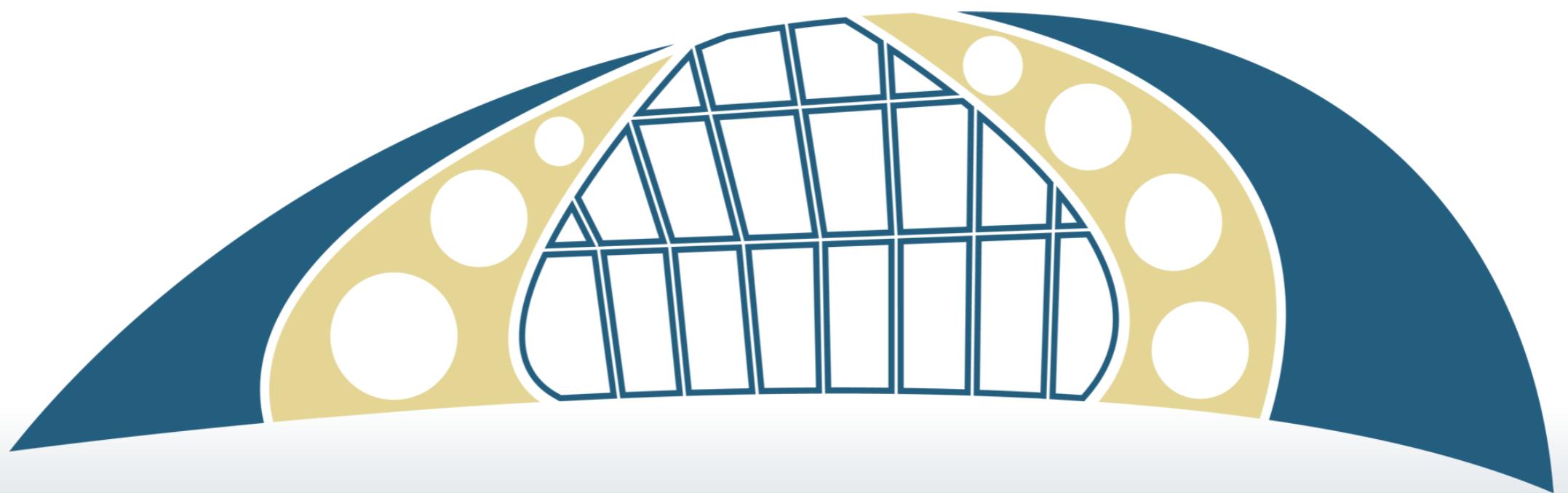


■ What changed:

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

■ Overall Performance

- N elements, D cycles latency/op
- (N/2+1)*D cycles:
CPE = D/2



WESTMONT INSPIRED
— COMPUTING LAB —