Program Optimization

Performance

Previously covered how programs are compiled and executed

Now, how to optimize execution

Optimizing Compilers

Provide basic mapping of program to machine

- Register allocation
- Code selection and ordering
- Eliminating minor inefficiencies

Have difficulty improving asymptotic efficiency

- Programmer must select best overall algorithm
- Big-O savings are often more important than constant factors

Limitations of Optimizing Compilers

Operate under fundamental constraint

- Must not cause any change in program behavior
- Often prevents it from making optimizations that would only affect behavior under pathological conditions.

Most analysis performed within procedures

- Whole-program analysis too expensive in most cases

Most analysis based on static information

- Compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative

Basic Optimizations

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

- Code motion
- Reduction in strength
- Using registers
- Share common sub-expressions

Code motion

Reduce frequency that a computation is performed

- If it will always produce the same result
- Moving code out of inner loop

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

```
for (i = 0; i < n; i++) {
  long 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   -->    x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
- Recognize sequence of products and replace with addition

```
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;
}
```

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

Compiler-generated optimizations

Most compilers do a good job with array code and simple loops

- Code motion and reduction in strength via –O2

```
n = 16;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
foo:
        xorl    %ecx, %ecx
.L2:
        xorl    %eax, %eax
.L5:
        movq    (%rsi,%rax,8), %rdx
        movq    %rdx, (%rdi,%rax,8)
```

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

Using registers

Reading and writing registers much faster than reading/writing memory

Limitation

- Compiler not always able to determine whether variable can be held in register

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

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

What if `a[0]` is an element of b?

- Possibility of Aliasing
  - Variable in memory that can be updated via two different pointers

Share common subexpressions

Reuse computations where possible

- **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;
```

→

```
int 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

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
```

```
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
```

String to lower case example

Procedure to convert string to lower case

```
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

If length of string is n, how does the run-time of this function grow with n?

- Linear, Quadratic, Cubic, Exponential?

String to lower case example

```
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

`strlen` executed every iteration

- `strlen` linear in length of string
- Must scan string until finds '\0'

Loop itself is linear in length of string

Overall performance is quadratic

String to lower case example

```
void lower2(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');
}
```

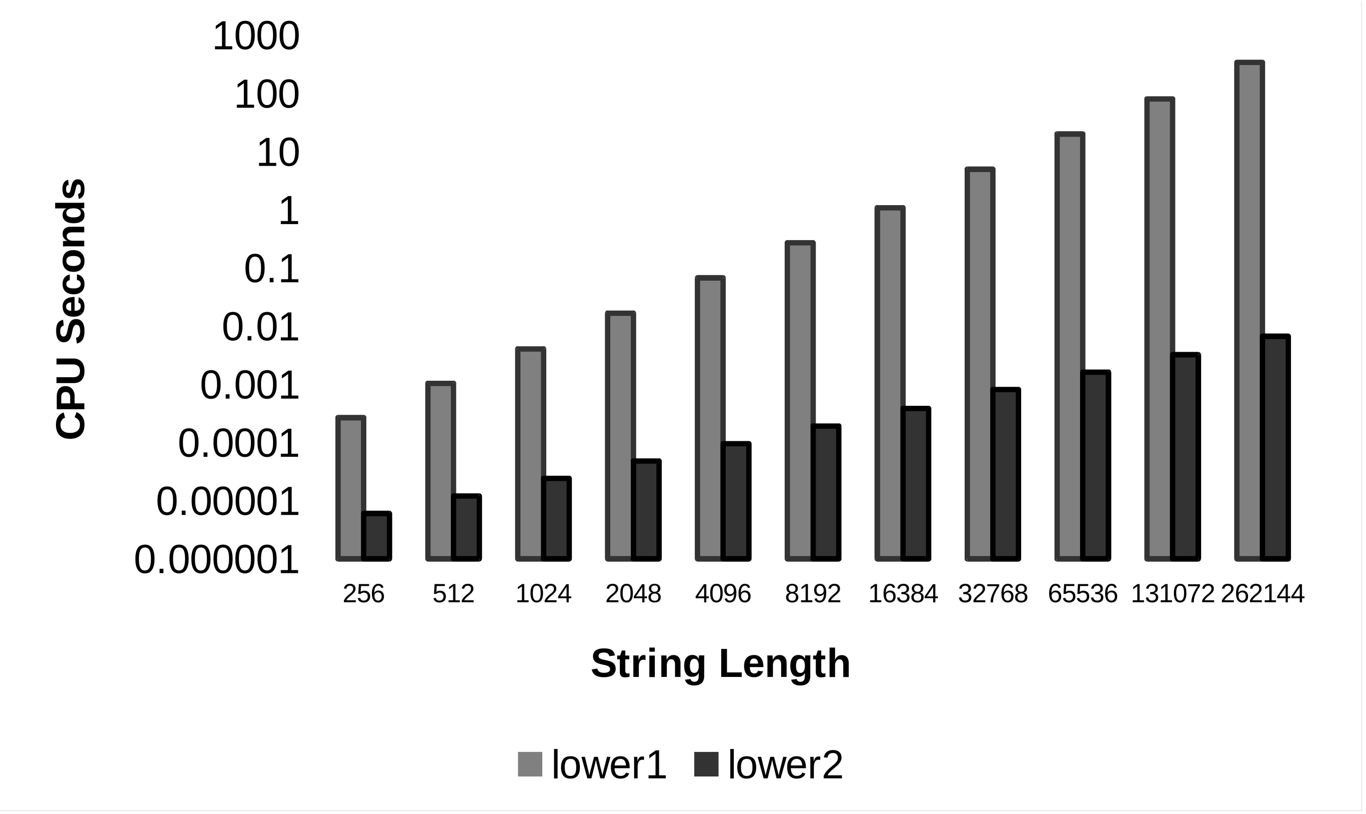
Apply code motion

- Move call to `strlen` outside of loop
- Result does not change from one iteration to another
- Compiler does not know this, though!

String to lower case example

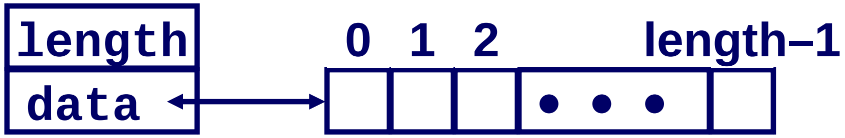
Quadratic performance of lower1

Linear performance of lower2



Vector combine example

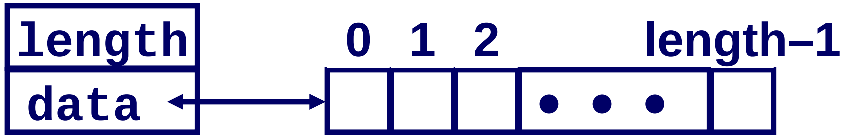
```
/* data structure for vectors */
typedef struct{
        size_t length;
        data_t *data;
} vec;
```



For different data types `data_t`

- `int`
- `long`
- `float`
- `double`

Vector combine example



Functions for vector

`int get_vec_element(vec_ptr v, int idx, data_t *dest)`

- Retrieve vector element at index `idx`, store at `*dest`
- Return 0 if out of bounds, 1 if successful

`int vec_length(vec_ptr v)`

- Returns length of vector (Note that this is O(1) due to length being stored along with vector)

`int *get_vec_start(vec_ptr v)`

- Return pointer to start of vector data

`void combine(vec_ptr v, data_t *dest)`

- Combine vector data, store result at `*dest`

Vector sum combine (`combine1`)

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

Procedure

- Compute sum of all elements of integer vector
- Store result at destination location
- Use code motion to speed up loop

Vector sum combine (`combine1`)

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

Inefficiency

- Procedure `vec_length` called every iteration
- Value does not change from one iteration to next
  - Compiler doesn’t know this, though!

Code motion (combine2)

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Code motion

- Move call to vec_length out of inner loop
  - `vec_length` requires only constant time, but significant overhead

Optimization Blocker: Function calls

Function may have side effects that require its execution on each iteration

- Function can alter global state each time called
- Function may not return same value for given arguments
  - Compiler does not know if inner-loop will change result of vec_length()

Why doesn’t compiler look at code for `vec_length` or `strlen?`

- Cost of interprocedural optimization prohibitive

Result

- Compiler treats procedure call as a black box
- Weak optimizations in and around them

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Explain how this optimization improves performance

Reduction in Strength (combine3)

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Optimization

- Procedure calls are expensive!
- Avoid procedure call to retrieve each vector element
  - Get pointer to start of array before loop
  - Within loop just do array reference
  - Not as clean in terms of data abstraction

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

What does this optimization do?

Using registers (combine4)

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

Optimization

- Memory references are expensive!
- Don’t need to store in destination until end
- Local variable sum held in register
- Avoids 1 memory read, 1 memory write per iteration

Detecting Unneeded Memory Refs.

`combine3`

```
.L18:
        movl (%ecx,%edx,4),%eax
        addl %eax,(%edi)
        incl %edx
        cmpl %esi,%edx
        jl .L18
```

`combine4`

```
.L24:
        addl (%eax,%edx,4),%ecx

        incl %edx
        cmpl %esi,%edx
        jl .L24
```

Performance of inner loop

- Combine3
  - 5 instructions in 6 clock cycles
  - `addl` must read and write memory
- Combine4
  - 4 instructions in 2 clock cycles

Problem with optimization

Compiler can not perform this optimization since combine4 not equivalent

Example

- `v: [3, 2, 17]`
- `combine3(v, get_vec_start(v)+2) -->   ?`
- `combine4(v, get_vec_start(v)+2) -->   ?`

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

Optimization Blocker: Memory Aliasing

`combine4` not equivalent to `combine3` due to aliasing

- Two different memory references specify single location (e.g. `*dest` and `v[2]`)

In example

- `v: [3, 2, 17]`
- `combine3(v, get_vec_start(v)+2) -->   10`
- `combine4(v, get_vec_start(v)+2) -->   22`

Observations

- Aliasing is easy to have happen in C via pointers
- Programmer must introduce local variables to use registers
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing

Practice problem 5.1

What does this procedure do if xp is not the same as yp?

```
void s(int *xp, *yp) {
       *xp = *xp + *yp;   /* x+y       */
       *yp = *xp - *yp;   /* x+y-y = x */
       *xp = *xp - *yp;   /* x+y-x = y */
}
```

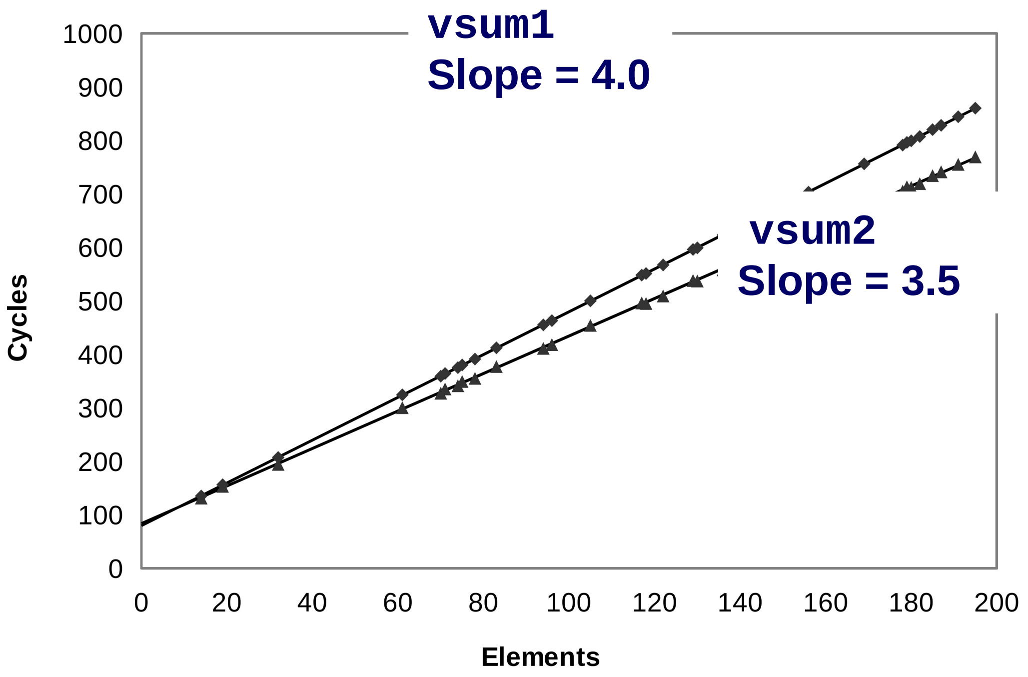
What does this procedure do if xp and yp point to the same location?

```
void s(int *xp, *yp) {
       *xp = *xp +   /* 2x          */
*yp;                   /* 2x – 2x = 0 */
       *yp = *xp -   /* 0 – 0 = 0   */
*yp;
       *xp = *xp -
*yp;
```

Measuring performance

Cycles per element (CPE)

- Express performance of program that operate on vectors
- n = number of elements
- Cycles = CPE*n + Overhead
- CPE = slope of the line



General form of combine

```
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, &val);
            *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Data Types

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

Operations

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

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, &val);
            *dest = *dest OP val;
    }
}
```

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

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;
}
```

Move vec_length out of loop

Avoid bounds check on each cycle

Accumulate in temporary

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;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

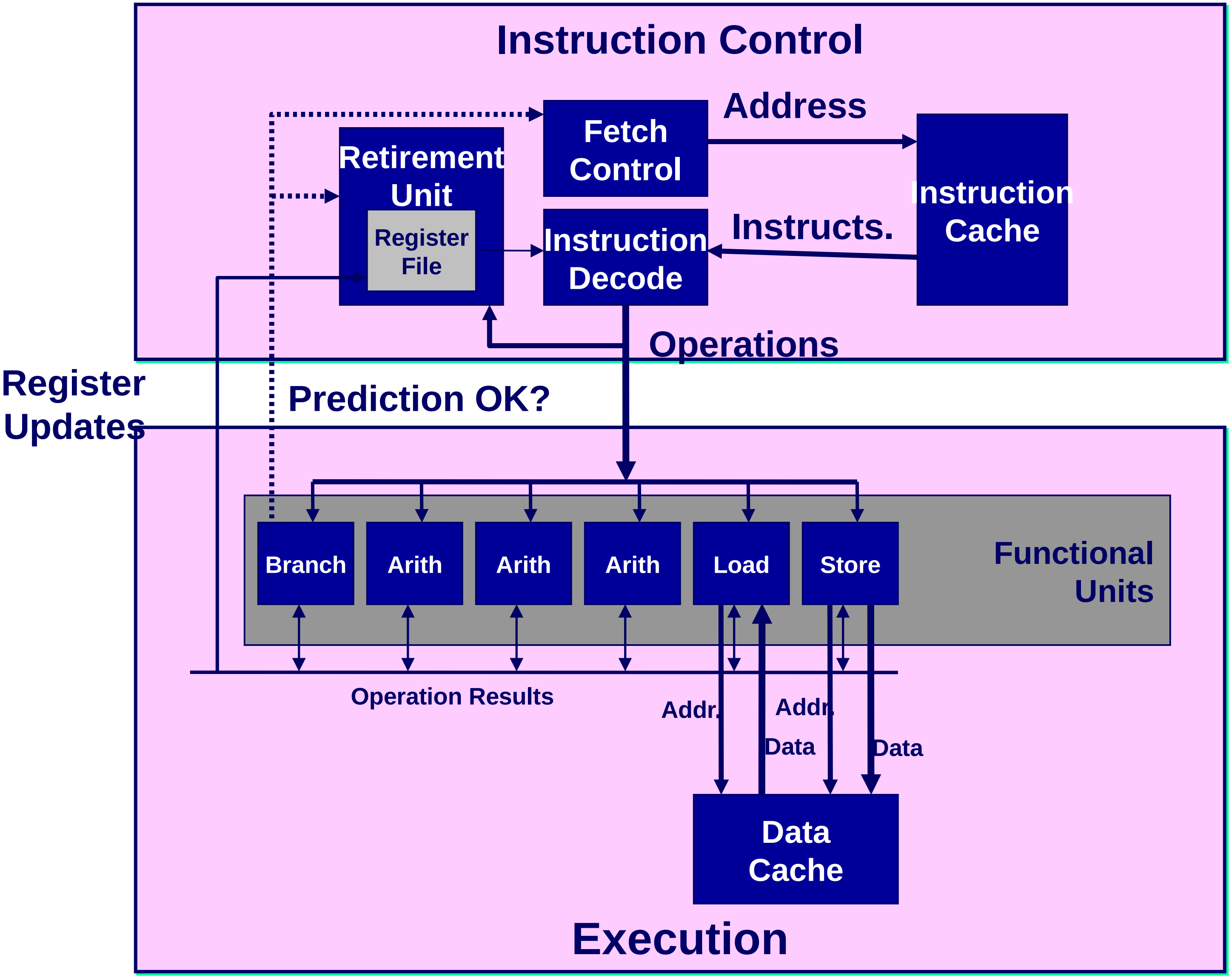
Advanced Optimizations

Modern CPU Design

Superscalar: Can issue and execute multiple instructions in one cycle

- Scheduled dynamically, without programming effort
- Takes advantage of instruction-level parallelism most programs have
- Intel: since Pentium (1993)

Modern CPU Design



Intel Core Haswell CPU (2013)

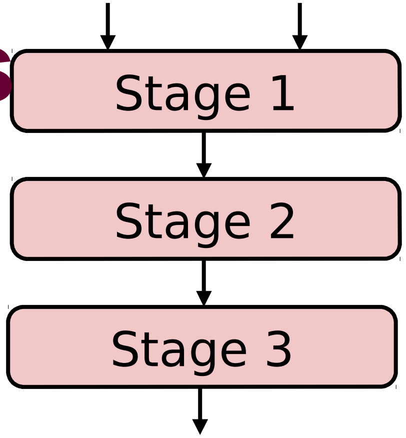
Multiple instructions can execute in parallel

- 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 Divide	3-30	3-30
FP Add	3	1
FP Multiply	5	1
FP Divide	3-15	3-15

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`

Integer multiply

- 3 cycle latency, 1 issue per cycle
- Computation divided into 3 stages
- Partial results passed 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

x86-64 combine4

Is combine4 enough to keep the multiplier busy?

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

```
.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;
```

1 data operation, 3 loop operations per iteration
Is there a way to do more computation and less looping?

Loop Unrolling (2x1)

Perform multiple operations per iteration

- 2x1 (2 operations per loop, 1 accumulator)
- Amortizes loop overhead across multiple iterations
- Finish extras at end

```
void combine5_2x1(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;
}
```

Loop Unrolling example (3x1)

C and assembly

```
void combine5_3x1(vec_ptr v, long *dest)
{
  long length = vec_length(v);
  long limit = length – 2;
  long *d = get_vec_start(v);
  long x = 1;
  long i;

      for (i=0; i<limit; i+=3)
       x *= d[i] * d[i+1] * d[i+2];
  for (; i<length; i++)
       x = x * d[i];
      *dest = x;
}
```

```
.L1:
        imulq   (%rsp,%rax,8),%rdx
        imulq   0x8(%rsp,%rax,8),
%rdx
        imulq   0x10(%rsp,%rax,8),
%rdx
        addq    $0x3,%rax
        cmpq    %rdi,%rax
```

Practice problem

Rewrite the following C code to perform 5-way (5x1) loop unrolling.

```
void combine5_3x1(vec_ptr v, long *dest)
{
  long length = vec_length(v);
  long limit = length – 2;
  long *d = get_vec_start(v);
  long x = 1;
  long i;

      for (i=0; i<limit; i+=3)
       x *= d[i] * d[i+1] * d[i+2];
  for (; i<length; i++)
       x = x * d[i];
      *dest = x;
}
```

Practice problem

What level of unrolling does this assembly implement?

```
.L1:
  imulq  (%rax,%rcx,8),%r8
      imulq  0x8(%rax,%rcx,8),%r8
  imulq  0x10(%rax,%rcx,8),%r8
  imulq  0x18(%rax,%rcx,8),%r8
  imulq  0x20(%rax,%rcx,8),%r8
  imulq  0x28(%rax,%rcx,8),%r8
      imulq  0x30(%rax,%rcx,8),%r8
  imulq  0x38(%rax,%rcx,8),%r8
  addq    %r8,%rdi
  addq    $0x8,%rcx
  cmpq    %rdx,%rcx
  jl     .L1
```

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

Helps integer add

- Achieves latency bound

Others don't improve

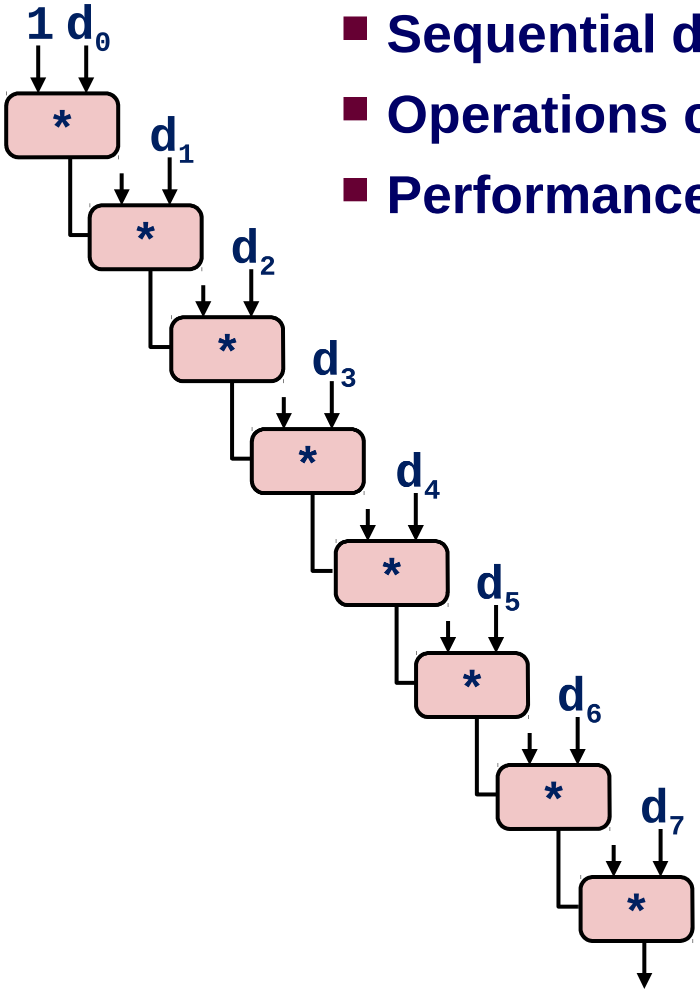
- Why?

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

Revisiting `combine4` (OP = *)

Length=8

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

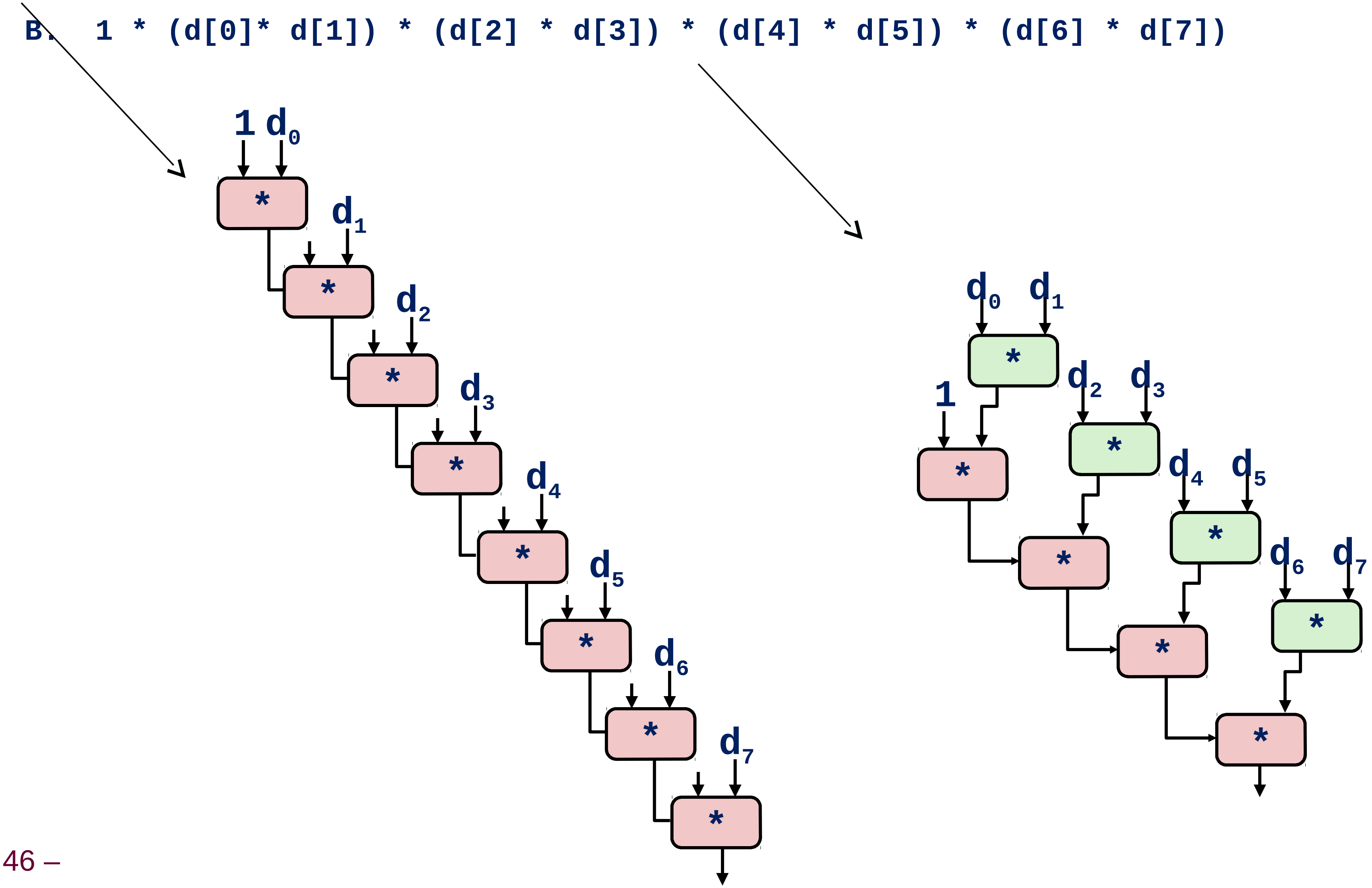


- Sequential dependence
- Operations computed serially
- Performance driven by latency of OP (* = 3 cycles)

Better or worse? Why?

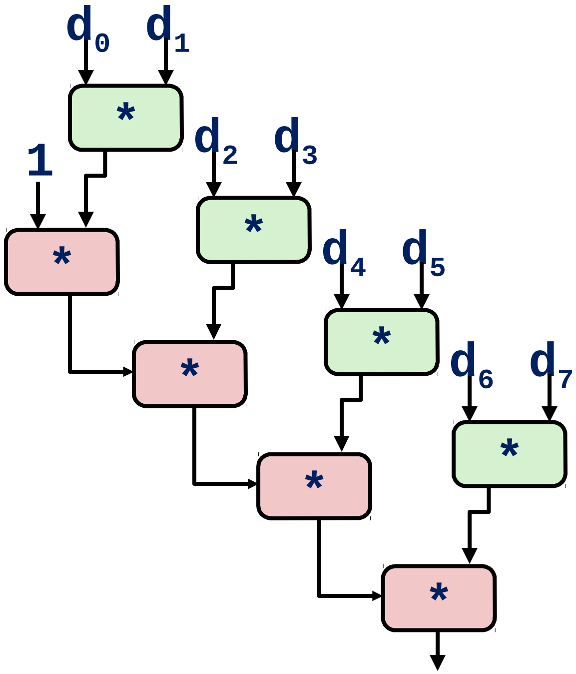
```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])

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



Reassociated Computation

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



What changed?

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

Overall Performance

- N elements, D cycles latency per operation

  (N/2 + 1) * D cycles

  CPE = D/2

Loop Unrolling with Reassociation (2x1a)

```
void combine7_2x1a(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;
}
```

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

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

Breaks sequential dependency

Products of pairs still combined sequentially

- Limits improvement to 2x
- Can reassociate pairs of product pairs, etc.

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

Limits to performance (Haswell)

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

2 func. units for FP
2 func. units for loa

2 load, with address computation
1 store, with address computation
4 integer (only 1 w/ branch and multiply)
2 FP multiply
1 FP add
1 FP divide

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

Loop Unrolling with Separate Accumulators

Idea

- Can unroll to any degree L
- Can accumulate K results in parallel
- L must be multiple of K

Limitations

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

Loop Unrolling with Separate Accumulators (2x2)

```
void combine6_2x2(vec_ptr v, data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    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;
}
```

Effect of Separate Accumulators

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**
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**

Unrolling & Accumulating: Double *

Case

- **Intel Haswell**
- **Double FP multiplication**
- **Latency bound: 5.00. Throughput bound: 0.50**

	FP *	Unrolling Factor L							
	K	1	2	3	4	6	8	10	12
Accumulators	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

Achievable Performance

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Best	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 functional units

Up to 42X improvement over original, unoptimized code

Limitations of Parallel Execution

Need Lots of Registers

- To hold sums/products
  - Also needed for pointers, loop conditions
- Limited integer and FP registers
- When not enough registers, must spill temporaries onto stack
  - Wipes out any performance gains

Advanced Optimizations summary

Speedup based on underlying CPU

- Loop Unrolling
- Reassociation
- Parallel accumulation

Multiple ALU and execution units performing operations in parallel

- Each performs a single operation

Increasing parallelism

- What if each unit could perform multiple simultaneous operations?
- Single-instruction, multiple-data (SIMD)