Lecture 12: Optimization

CS 105

Fall 2026

Review: Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
  - Typically 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typically 4 clock cycles for L1, 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Review: Matrix Multiplication

- `a` $= \begin{bmatrix} 2.0 & 1.5 \\ 1.0 & 0.5 \end{bmatrix}$
- `b` $= \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$

- What is `a` x `b`?

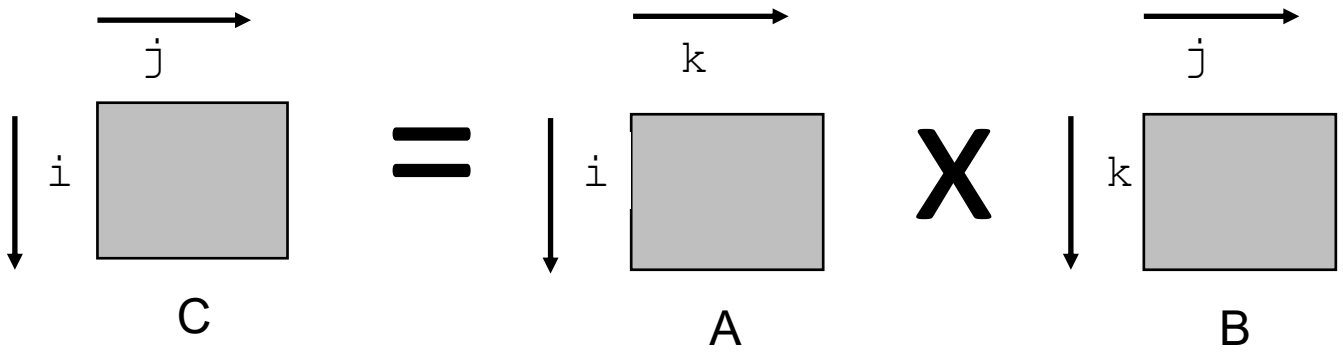
Example: Matrix Multiplication

- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination

```
/* ijk */
for(int i=0; i<n; i++)  {
  for(int j=0; j<n; j++) {
    sum = 0.0;
    for(int k=0; k<n; k++){
      sum += a[i][k] * b[k][j];
    }
    c[i][j] = sum;
  }
}
```

Miss Rate Analysis for Matrix Multiply

- Assume:
  - datablock size = 32 bytes (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop

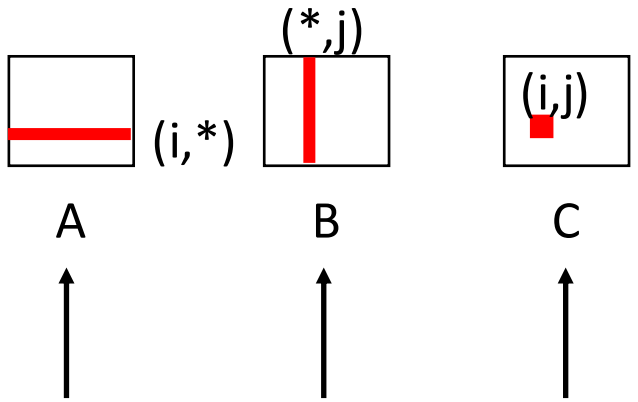


Matrix Multiplication (ijk)

(jik is similar)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



Average Misses per inner loop iteration:

A	B	C	Total
.25	1.0	0.0	1.25

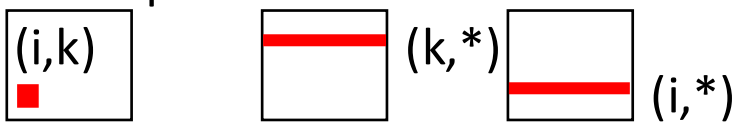
2 reads, 0 writes per inner loop iteration

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

2 reads, 1 write per inner loop iteration

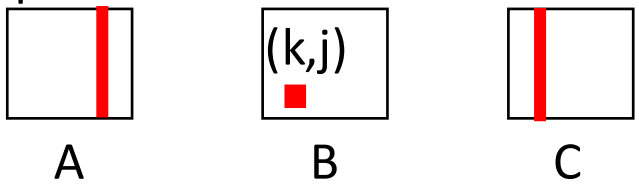
Inner loop:



Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Inner loop: (*,k)



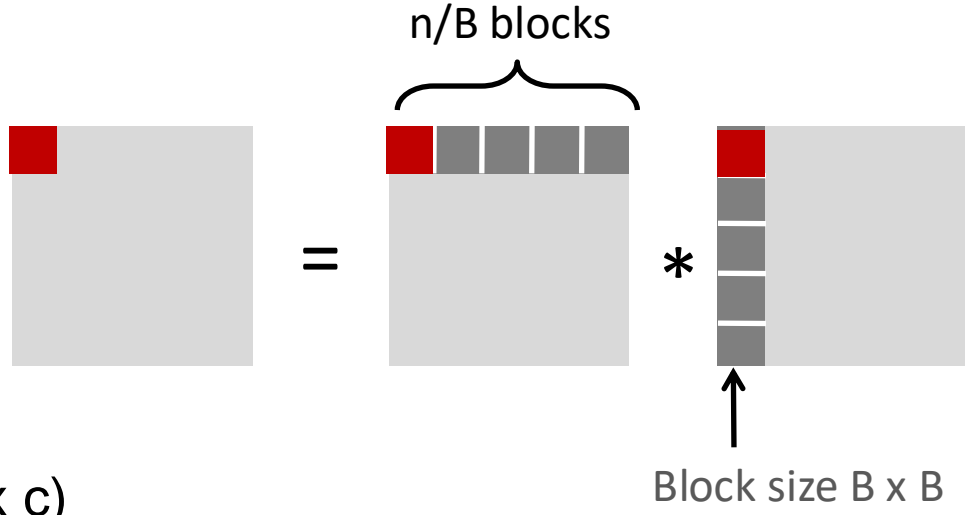
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Cache Miss Analysis

- Assume:
  - Cache data block = 4 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- First (block) iteration:
  - B^2 elements in each block, so $B^2/4$ misses for each block
  - n/B blocks in each row/col, so $2 * n/B * B^2/4 = nB/2$ misses in first iteration(omitting matrix c)



- Total misses:
  - $nB/2 * (n/B)^2 = n^3/(2B)$

Writing Cache-Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (**temporal locality**)
  - Stride-1 reference patterns are good (**spatial locality**)

COMPILER OPTIMIZATIONS

Under the Abstraction Barrier

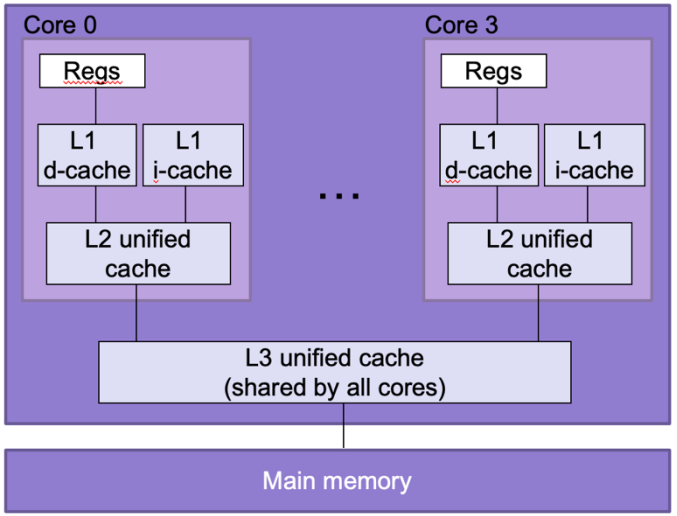
```
#include<stdio.h>

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

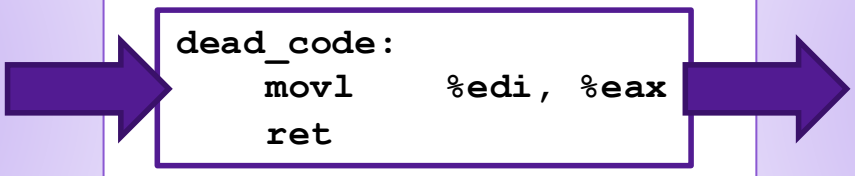
- Compiler optimization flags
  - `-O0, -O1, -O2, -O3, -Os, -Og`

- 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

Compiling with -O0

- Produced assembly instructions most similarly resemble the source code
  - Minimal compilation complexity
  - Useful for debugging
- Optimizations:
  - Dead Code Elimination

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




```
int dead_code(int input){

  return input;



}
```

Compiling with -O1

- Produce assembly instructions with minimal optimization on top of source code
- Optimizations:
  - Code Motion: moving assembly instructions to reduce the frequency of redundant computations

```
void set_row(int* a, int* b,
             int i,  int n) {

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



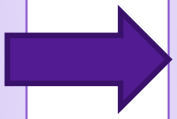
```
void set_row(int* a, int* b,
             int i,  int n) {
  int ni = n*i;
  for (int j = 0; j < n; j++){
    a[ni+j] = b[j];
  }
}
```

Compiling with -O1

- Produce assembly instructions with minimal optimization on top of source code
- Optimizations:
  - Code Motion: moving assembly instructions to reduce the frequency of redundant computations
  - Factoring out Subexpressions: produce intermediate helper variables to achieve similar computation

```
/* 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

Compiling with -O1

- Produce assembly instructions with minimal optimization on top of source code
- Optimizations:
  - Code Motion: moving assembly instructions to reduce the frequency of redundant computations
  - Factoring out Subexpressions: produce intermediate helper variables to achieve similar computation
  - Loop Elimination: compute equivalent functionality without a loop

```
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;
}
```

Compiling with -O1

- Produce assembly instructions with minimal optimization on top of source code
- Optimizations:
  - Code Motion: moving assembly instructions to reduce the frequency of redundant computations
  - Factoring out Subexpressions: produce intermediate helper variables to achieve similar computation
  - Loop Elimination: compute equivalent functionality without a loop
- Takeaway: the compiler can transform the C source into *functionally equivalent* assembly code

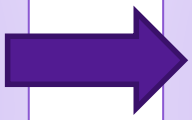
Compiling with -O2

- Optimize even more! Produce a much faster version of the program in assembly as compared to the original C source
- Optimizations:
  - All of the optimizations described in -O1
  - Reduction in Strength: replace costly assembly instructions with simpler ones

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

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

  }
```



```
void set_matrix(long* a, long* b,
                long n){
  long ni = 0;
  for (long i = 0; i < n; i++) {

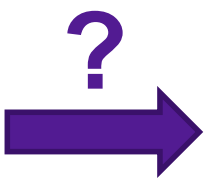
    for (long j = 0; j < n; j++){
      a[ni + j] = b[j];
    }
    ni += n;
  }
```

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;
}
```



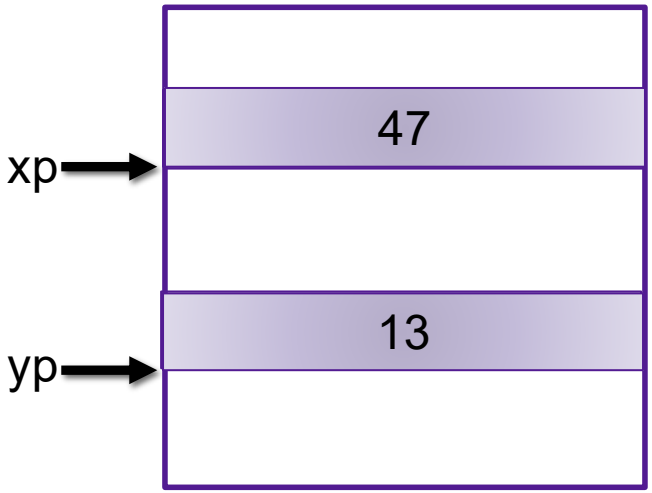

```
void mystery2(int* xp,
              int* yp){
  int temp = *xp;
  *xp = *yp;
  *yp = temp;
}
```

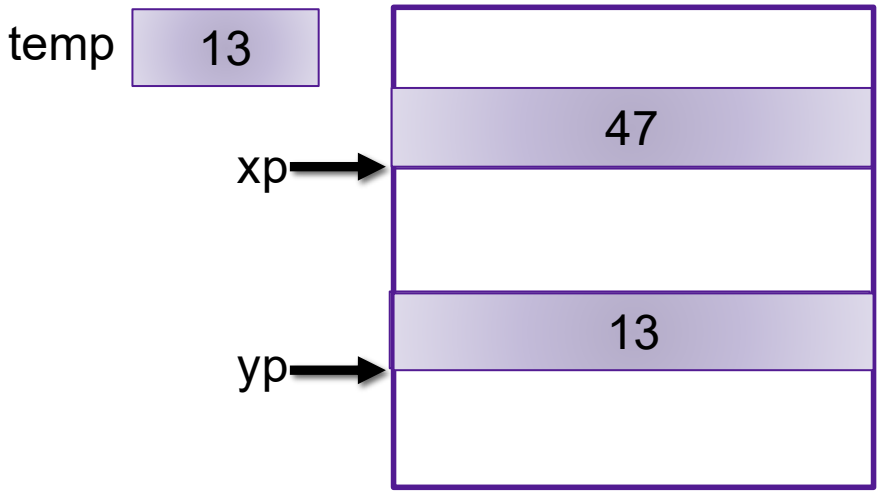
Exercise: What do each of these programs do?
Do they do the same thing?

Comparing Programs

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

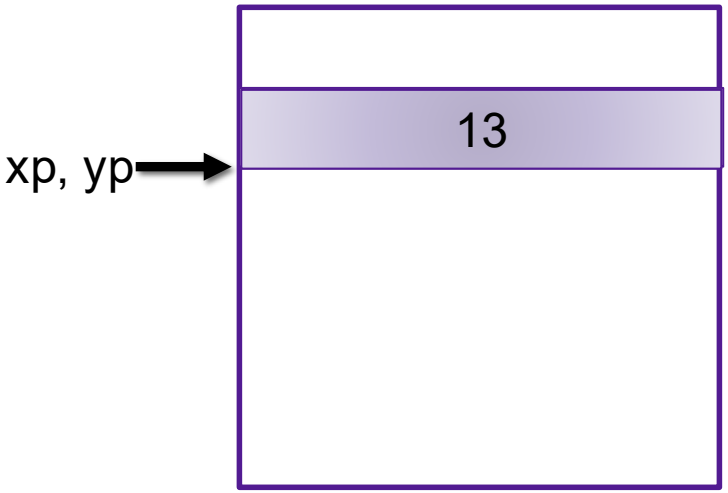
```
void mystery2(int* xp,
              int* yp){
  int temp = *xp;
  *xp = *yp;
  *yp = temp;
}
```



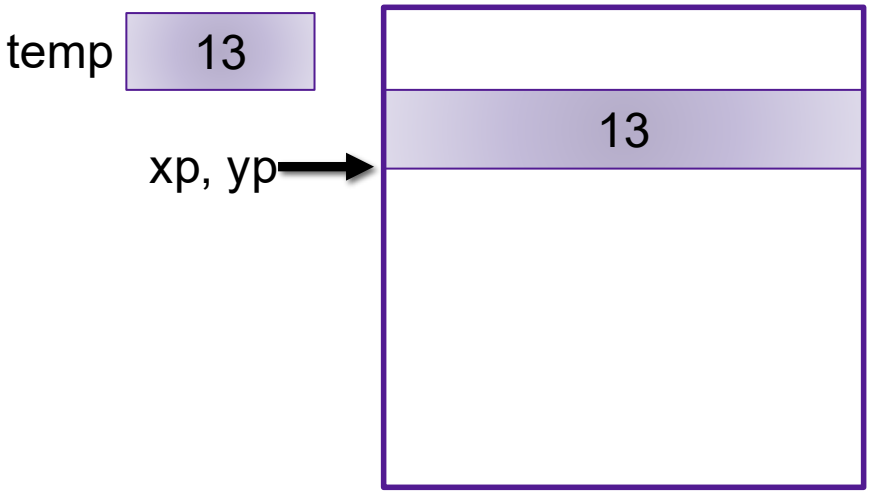


Comparing Programs

```
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

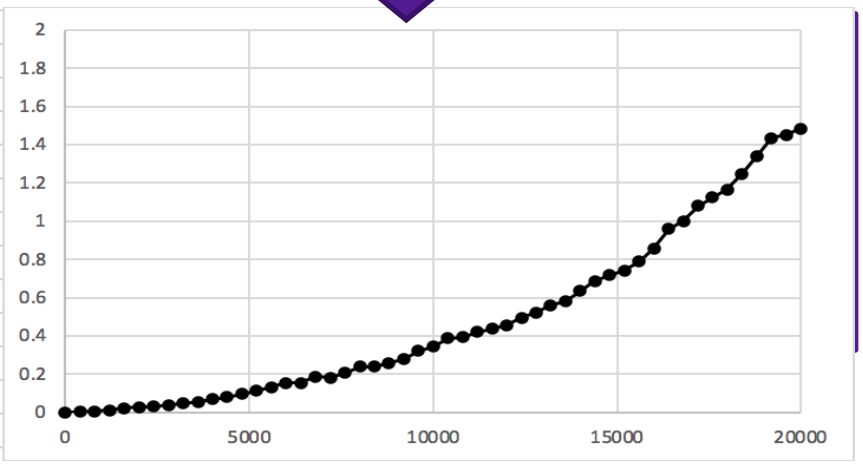
- Aliasing: Two different references to a single location
  - Easy to happen in C

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

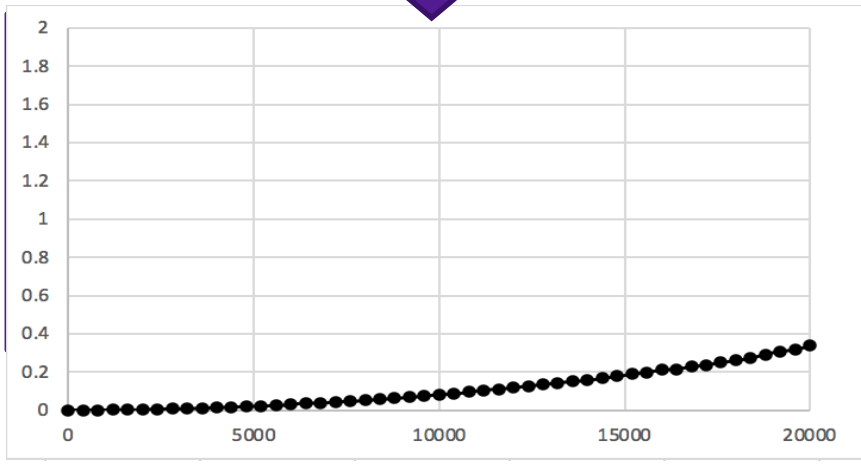
Example: Summing Matrix Rows

```
/* 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

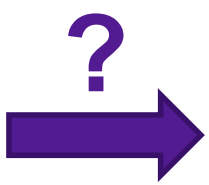
2. Most analysis is performed only within procedures
   - Whole-program analysis is too expensive in most cases
   - Newer versions of `gcc` do inter-procedural analysis within files

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

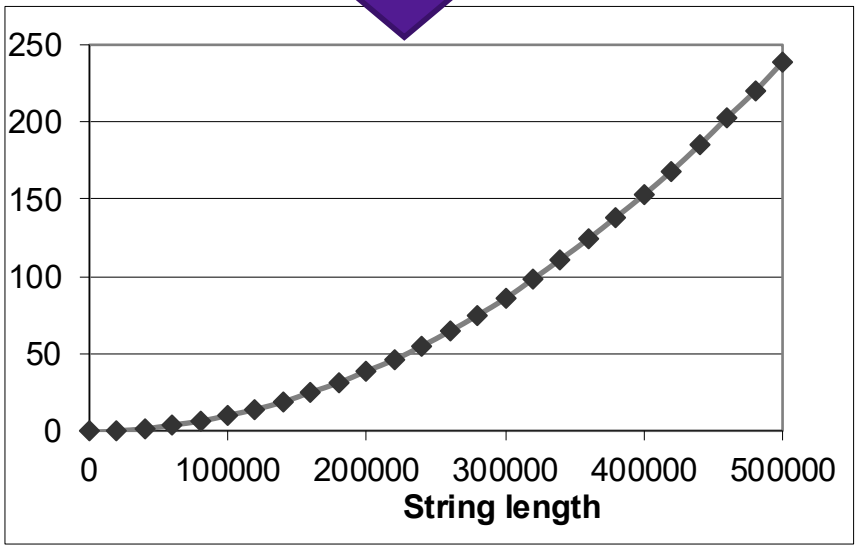
- Compiler treats procedure calls as black boxes
  - Unknown side-effects
  - **`strlen`** may not always return the same value

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

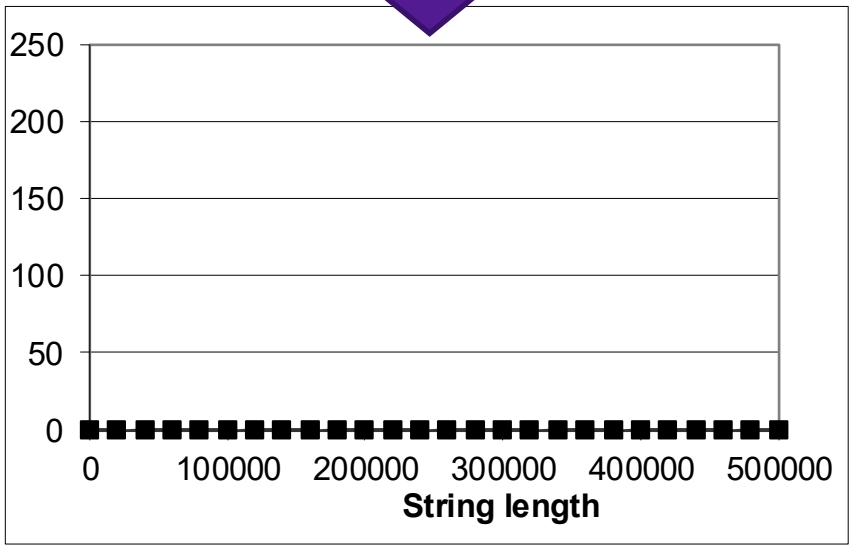
Example: Lowering Case

```
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];
  }
}
```



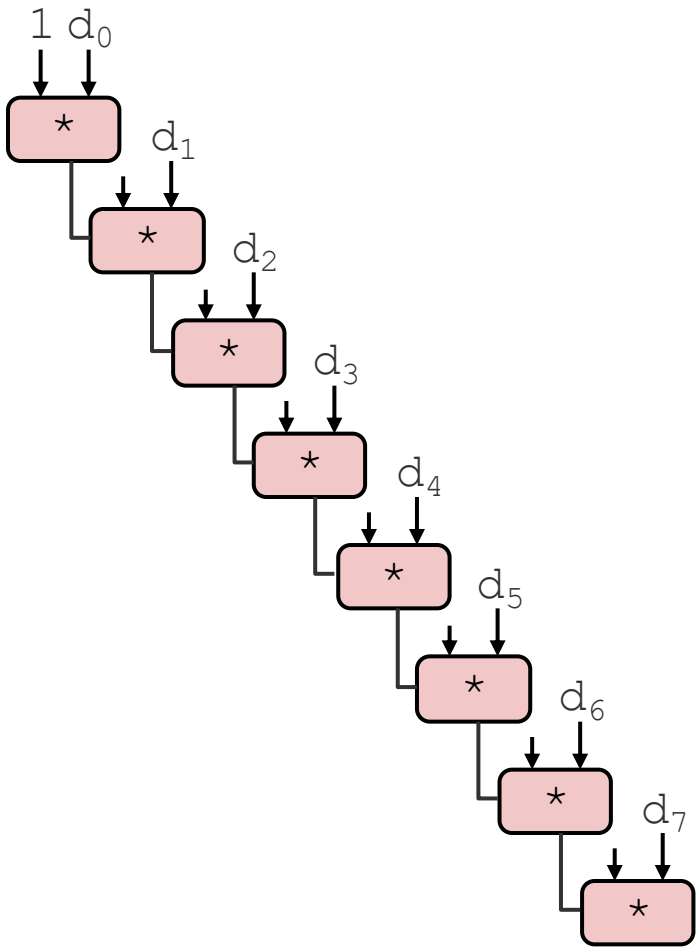
Combine with Unrolling

```
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 */
```

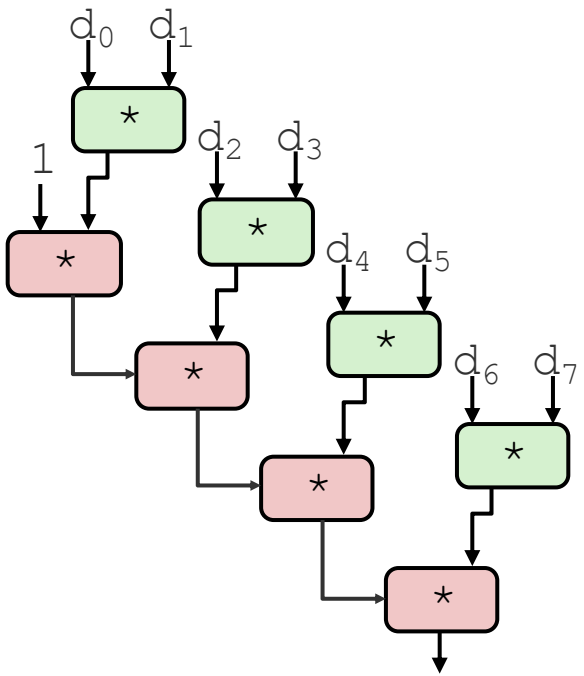
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

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]);
```

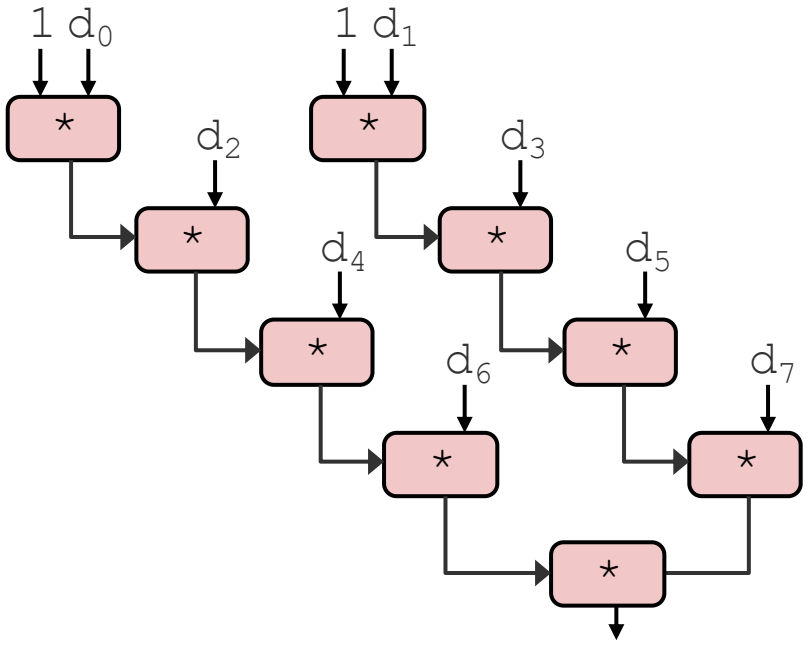
2 func. units for FP *
2 func. units for load

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

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

Accumulators

Float Multiplication

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

Accumulators

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