# Optimizing Compilers

Efficient mapping of program to machine

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

Don't (usually) improve asymptotic efficiency

- programmer must use proper algorithm
- non-algorithmic differences can still be a 10x difference
- compiler helps programmer and vice versa

Have difficulty overcoming "optimization blockers"

- potential memory aliasing
- potential procedure side-effects

# Writing What You Mean

```
int x = 5, y = 3;
set_add_twice(&x, &y);
```

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

Copy

```
void set_add_twice(int* xp, int* yp) {
  *xp += 2 * *yp;
}
```

Copy

Those two look the same...

# Writing What You Mean

```
int x = 5;
set_add_twice(&x, &x);
```

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

Copy

```
void set_add_twice(int* xp, int* yp) {
  *xp += 2 * *yp;
}
```

Copy

Not the same after all!

# Optimizer Limitations

Optimization must not change the program behavior

... as long as the behavior is defined

Behavior obvious to the programmer can be obfuscated by coding style

e.g., actual range narrower than datatype

Most analysis is performed only within a procedure

... or within a file

Analysis is based only on ***static*** information

i.e., optimizer doesn't know program input

# Generally Useful Optimizations

Performed by you or by compiler:

- Code motion
- Strength reduction
- Sharing common results
  (a.k.a. common subexpression elimination)

# Code Motion

```
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];
}
```

# Code Motion

```
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];
}
```

# Code Motion

```
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];
}
```

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

# Code Motion

```
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];
}
```

gcc -O1

```
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:                              # loop:
  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:                              # done:
  rep ret
```

# Code Motion

```
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];
}
```

gcc -O1

```
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:                              # loop:
  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:                              # done:
  rep ret
```

# Code Motion

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

gcc -O1

```
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:                              # loop:
  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:                              # done:
  rep ret
```

# Strength Reduction

Replace an operation with a simpler one

Example: shift instead of multiply or divide

**`16*x` ⇒ `x<<4`**

```
int sixteen(int v) {
  return 16*v;
}
```

⇒

```
movl  %edi, %eax
sall  $4, %eax
ret
```

**`gcc -O1`**

# Strength Reduction

Replace a sequence of products with additions

```
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;
```

gcc -O2

```
   movslq  %edx, %r9       # r9 = n
   xorl    %r8d, %r8d      # i = 0
   salq    $2, %r9         # %r9 = 4*n
.L6:
   xorl    %eax, %eax      # j = 0
.L4:
   movl    (%rsi,%rax,4), %ecx
   movl    %ecx, (%rdi,%rax,4)
   addq    $1, %rax        # j++
   cmpl    %eax, %edx
   jg      .L4
   addl    $1, %r8d        # i++
   addq    %r9, %rdi       # pa += 4*n
   cmpl    %edx, %r8d
   jne     .L6
```

# Common Subexpressions

```
/* Sum values near i,j */
prev2 = val[i*n + j - 2];
prev  = val[i*n + j - 1];
next  = val[i*n + j + 1];
next2 = val[i*n + j + 2];
sum = (prev2 + prev
       + next + next2);
```

# Common Subexpressions

```
/* Sum values near i,j */
prev2 = val[i*n + j - 2];
prev  = val[i*n + j - 1];
next  = val[i*n + j + 1];
next2 = val[i*n + j + 2];
sum = (prev2 + prev
       + next + next2);
```

⇒

```
long inj = i*n + j;
prev2 = val[inj - 2];
prev  = val[inj - 1];
next  = val[inj + 1];
next2 = val[inj + 2];
sum = (prev2 + prev
       + next + next2);
```

# Common Subexpressions

```
/* 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);
```

# Common Subexpressions

```
/* 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);
```

# Common Subexpressions

```
/* 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);
```

```
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
```

3 multiplications

⇒

```
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
```

1 multiplication

# Optimization Blocker #1: Procedure Calls

```
int g(int v);

int f(int *a, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    accum += g(a[i]);
    accum += a[i];
  }

  return accum;
}
```

Copy

# Optimization Blocker #1: Procedure Calls

```
int g(int v);

int f(int *a, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    accum += g(a[i]);
    accum += a[i];
  }

  return accum;
}
```

Copy

```
.L3:
    movl    (%rbx), %edi
    addq    $4, %rbx
    leal    (%rdi,%rax), %ebp
    call    g
    addl    %ebp, %eax
    addl    -4(%rbx), %eax
    cmpq    %r12, %rbx
    jne     .L3
```

gcc -O2

# Optimization Blocker #1: Procedure Calls

```
int g(int v);

int f(int *a, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    accum += g(a[i]);
    accum += a[i];
  }

  return accum;
}
```

Copy

```
.L3:
    movl    (%rbx), %edi
    addq    $4, %rbx
    leal    (%rdi,%rax), %ebp
    call    g
    addl    %ebp, %eax
    addl    -4(%rbx), %eax
    cmpq    %r12, %rbx
    jne     .L3
```

gcc -O2

```
int g(int v) {
  return v + 1;
}
```

Copy

# Optimization Blocker #1: Procedure Calls

```
int g(int v);

int f(int *a, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    accum += g(a[i]);
    accum += a[i];
  }

  return accum;
}
```

```
.L3:
    movl    (%rbx), %edi
    addq    $4, %rbx
    leal    (%rdi,%rax), %ebp
    call    g
    addl    %ebp, %eax
    addl    -4(%rbx), %eax
    cmpq    %r12, %rbx
    jne     .L3
```

cc -O2

```
char global_a[100] = ....;

int g(int v) {
  global_a[v] = 0;
}

int main() {
  f(global_a, 100);
}
```

# Optimization Blocker #2: Aliasing

```
/* Sum rows of `a` and store in vector `b`  */
void sum_rows1(int *a, int *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];
  }
}
```

# Optimization Blocker #2: Aliasing

```
/* Sum rows of `a` and store in vector `b`  */
void sum_rows1(int *a, int *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];
  }
}
```

```
# inner loop of sum_rows1:
.L4:
   addl         (%rdi), %eax
   addq         $4, %rdi
   cmpq         %rdi, %r8
   movl         %eax, (%rsi,%rcx,4)
   jne          .L4
```

# Optimization Blocker #2: Aliasing

```
/* Sum rows of `a` and store in vector `b`  */
void sum_rows1(int *a, int *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];
  }
}
```

```
int A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
int *B = A+3;

sum_rows1(A, B, 3);
```

```
# inner loop of sum_rows1:
.L4:
   addl         (%rdi), %eax
   addq         $4, %rdi
   cmpq         %rdi, %r8
   movl         %eax, (%rsi,%rcx,4)
   jne          .L4
```

# Optimization Blocker #2: Aliasing

Change the program to avoid question of aliasing:

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

```
# inner loop of sum_rows2:
.L5:
   addl    (%rdi), %eax
   addq    $4, %rdi
   cmpq    %rdi, %rcx
   jne     .L5
```

# Strict Aliasing

```
int sum_and_set(int *a, long *c, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    c[i] = i;
    accum += a[i];
  }

  return accum;
}
```

# Strict Aliasing

```
int sum_and_set(int *a, long *c, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    c[i] = i;
    accum += a[i];
  }

  return accum;
}
```

```
.L3:
    movl    (%rdi,%rcx,4), %r8d
    movq    %rcx, (%rsi,%rcx,8)
    addq    $1, %rcx
    cmpl    %ecx, %edx
    leal    (%rax,%r8,2), %eax
    jg      .L3
```

# Strict Aliasing

```
int sum_and_set(int *a, long *c, int len) {
  int i, accum = 0;

  for (i = 0; i < len; i++) {
    accum += a[i];
    c[i] = i;
    accum += a[i];
  }

  return accum;
}
```

```
.L3:
    movl    (%rdi,%rcx,4), %r8d
    movq    %rcx, (%rsi,%rcx,8)
    addq    $1, %rcx
    cmpl    %ecx, %edx
    leal    (%rax,%r8,2), %eax
    jg      .L3
```

```
int a[6] = {1, 2, 3, 0, 0, 0};
long *c = (long *)a;

sum_and_set(a, c, 3);
```

# Strict Aliasing

```
int sum_and_set(int *a, long *c, int len);
```

***Strict aliasing*** means that a compiler can assume that **`p1`** and **`p2`** have different addresses if they're different pointer types

... but **`void*`** counts as "the same" to any pointer type

... and **`char*`**, too

# Struct Aliasing and Casts

Strict aliasing is why this doesn't work reliably:

```
float f = 1.2;
int i = *(int *)&f;
```

Roughly, this works because **`memcpy`** operates on **`void*`**s:

```
float f = 1.2;
int i;
memcpy(&i, &f, sizeof(float));
```

# Strict Aliasing

Conclusion:

In common cases, you don't have to manually avoid aliasing issues

# The Curious Case of `strlen`

```
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');
}
```

# The Curious Case of `strlen`

```
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');
}
```

Calls **`strlen`** every iteration of the loop

Optimization blocker?

*Algorithmic problem!*

# Looping Until `strlen`

**`strlen`** must walk a string to find the 0 terminator

```
int my_strlen(char *s) {
  int len = 0;
  while (s[len] != 0) len++;
  return len;
}
```

So, don't write this pattern:

```
for (i = 0; i < strlen(s); i++) ....
```

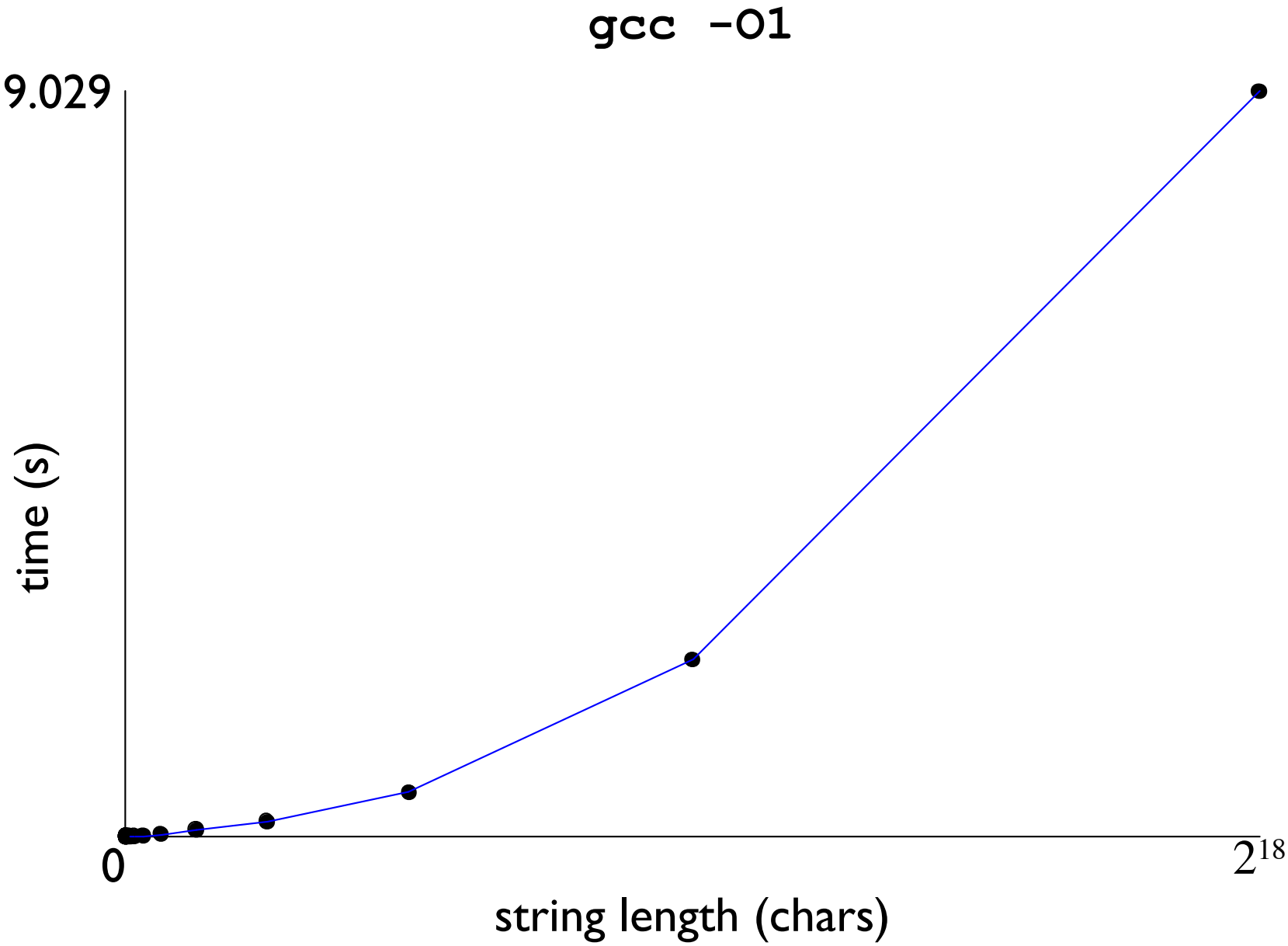
Instead, write

```
int len = strlen(s);
for (i = 0; i < len; i++) ....
```

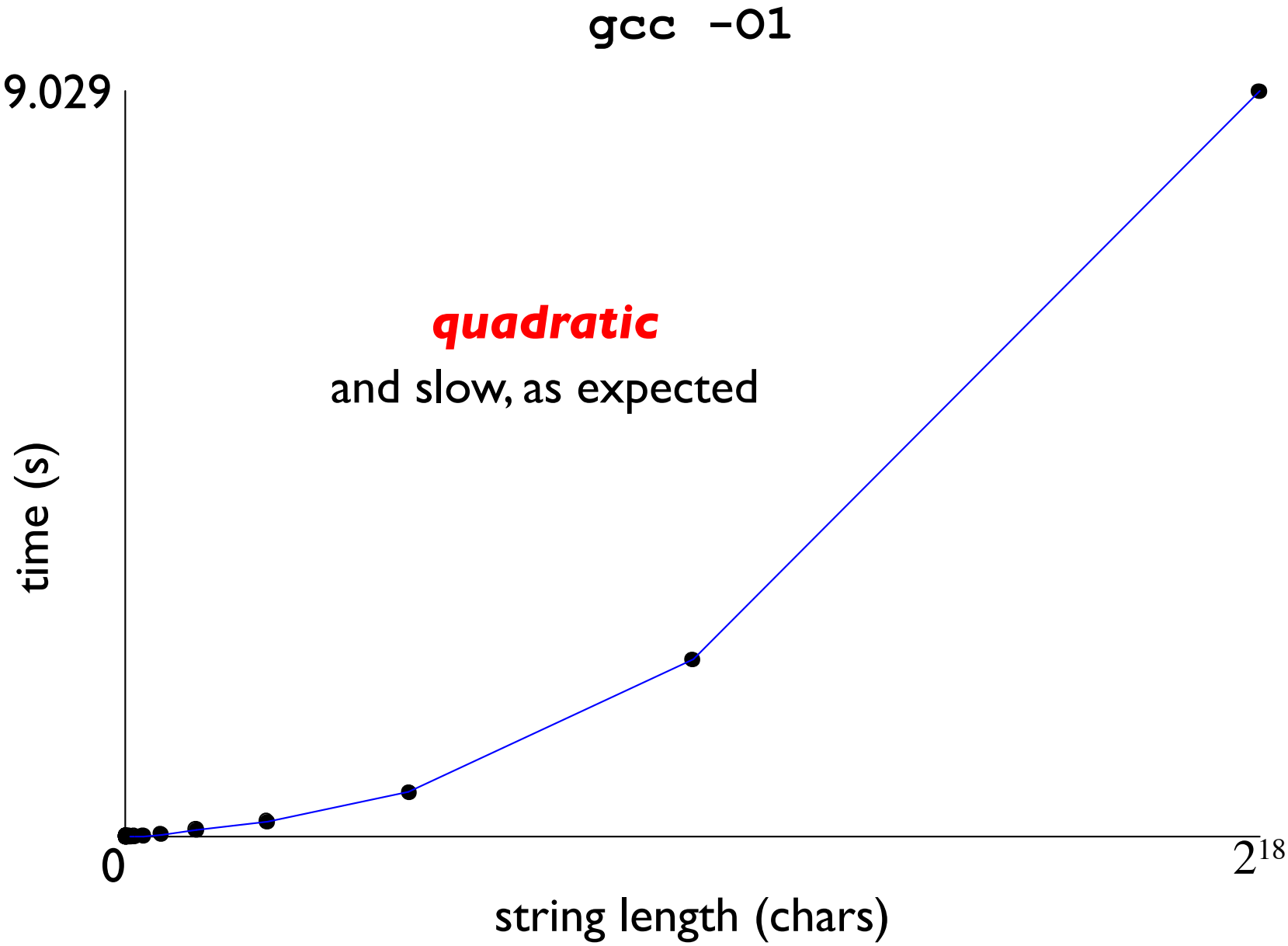
# Benchmarking `lower`

```
memset(s, 'a', MAX_N);

for (i = 1; i <= MAX_N; i *= 2) {
  s[i] = 0; /* set terminator */
  start = get_cpu_time();

  lower(s);

  end = get_cpu_time();
  s[i] = 'a'; /* restore non-terminator */

  printf("%ld %ld\n", i, end - start);
}
```

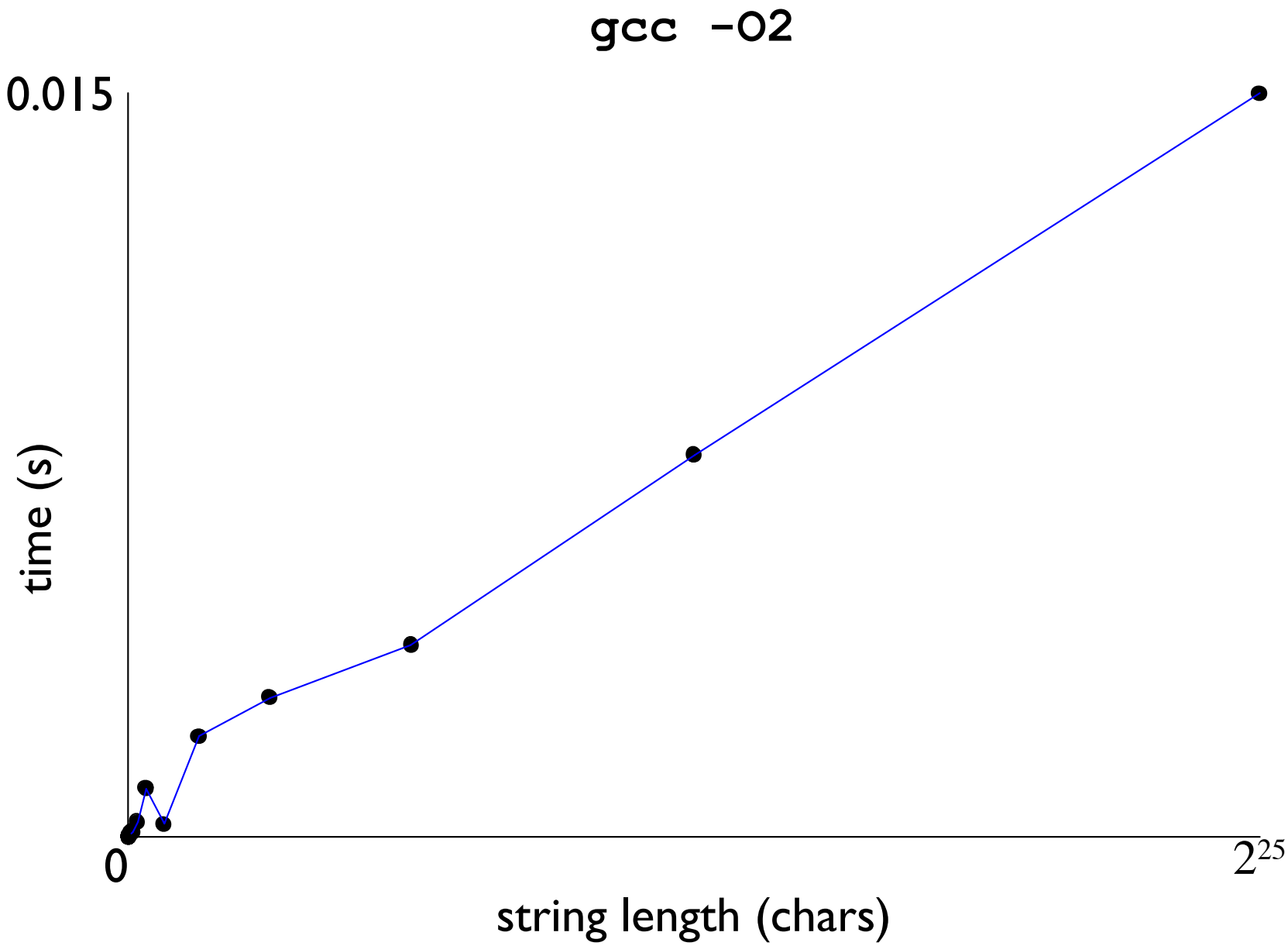
# Benchmark Results



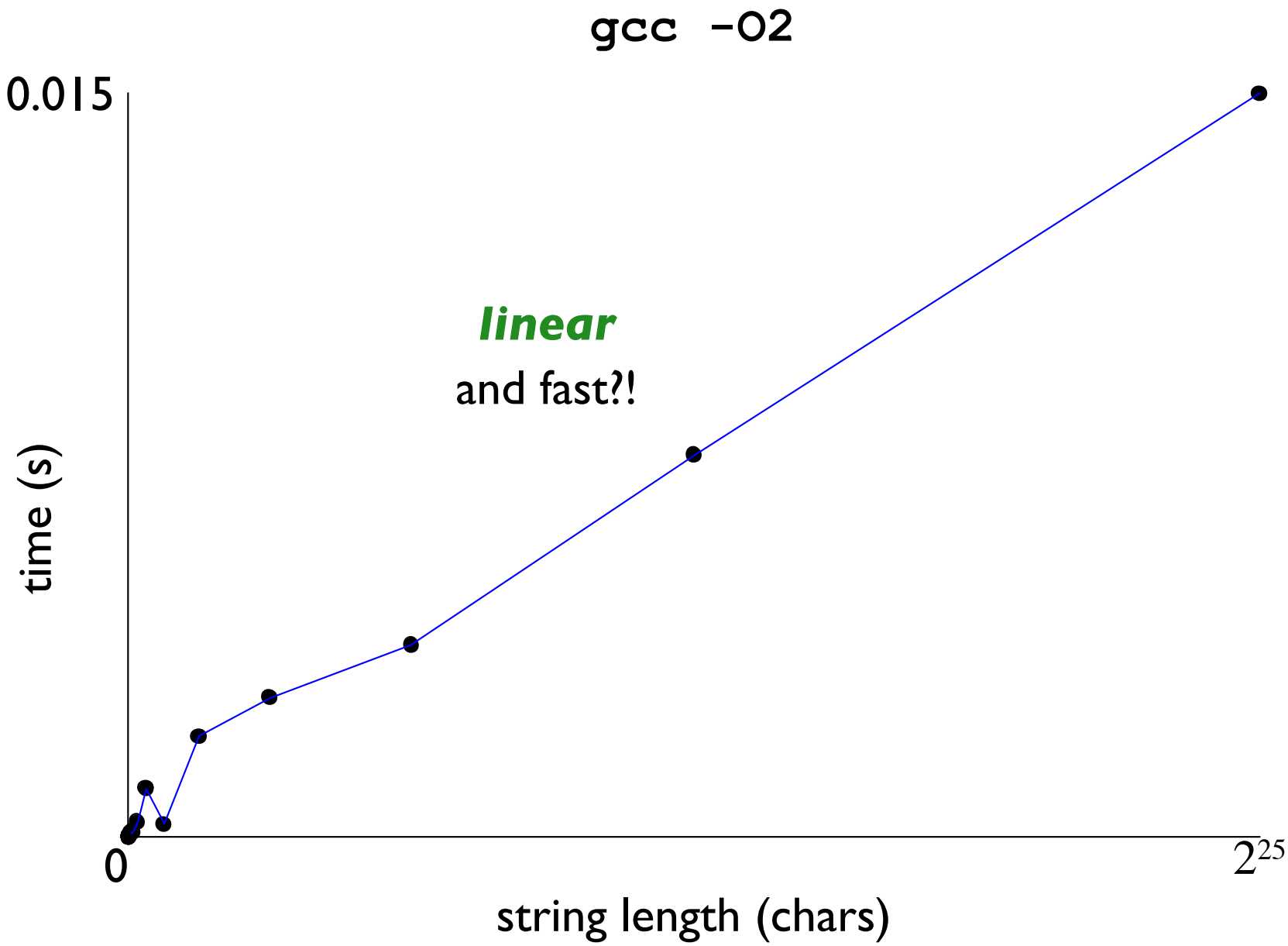
# Benchmark Results



# Benchmark Results



# Benchmark Results



# The Curious Case of `strlen`

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

```
.L11:
   movzbl  0(%rbp,%rbx), %edx
   leal    -65(%rdx), %ecx     # s[i]-'A'
   cmpb    $25, %cl            # result > 'Z'-'A' ?
   ja      .L10                # if so, skip...
   addl    $32, %edx           #  compute lowercase
   movq    %rbp, %rdi
   movb    %dl, 0(%rbp,%rbx)   #  store s[i]
   call    strlen              #  call strlen() again
.L10:
   addq    $1, %rbx
   cmpq    %rax, %rbx
   jb      .L11
```

**`gcc -O2`**

# The Curious Case of `strlen`

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

```
.L11:
   movzbl  0(%rbp,%rbx), %edx
   leal    -65(%rdx), %ecx     # s[i]-'A'
   cmpb    $25, %cl            # result > 'Z'-'A' ?
   ja      .L10                # if so, skip...
   addl    $32, %edx           #  compute lowercase
   movq    %rbp, %rdi
   movb    %dl, 0(%rbp,%rbx)   #  store s[i]
   call    strlen              #  call strlen() again
.L10:
   addq    $1, %rbx
   cmpq    %rax, %r
   jb      .L11
```

**`gcc`** knows that **`strlen`** result is unchanged if **`s`** is unchanged

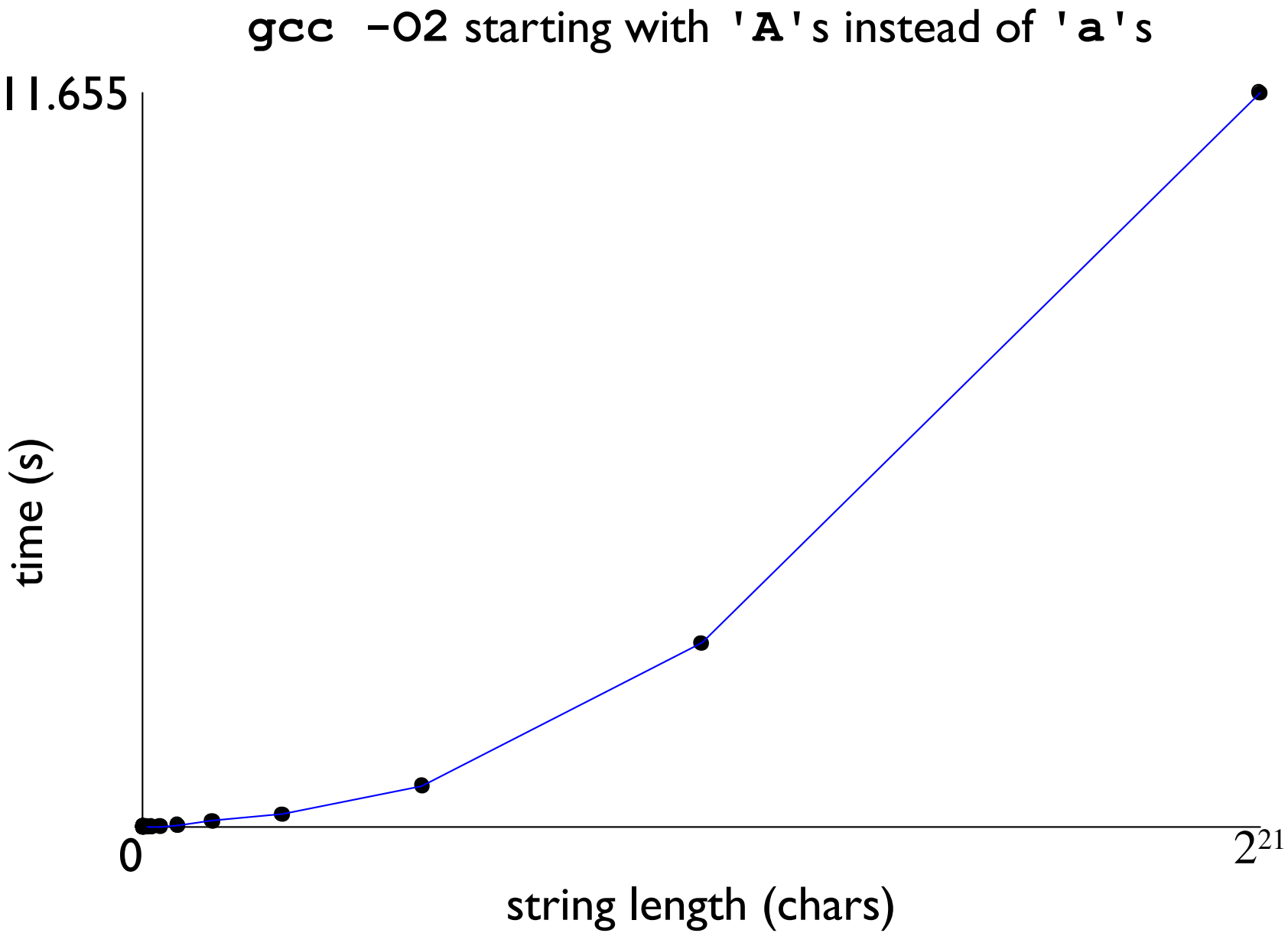
**`gcc -O2`**

# Changing the Benchmark

```
memset(s, 'A', MAX_N);
```

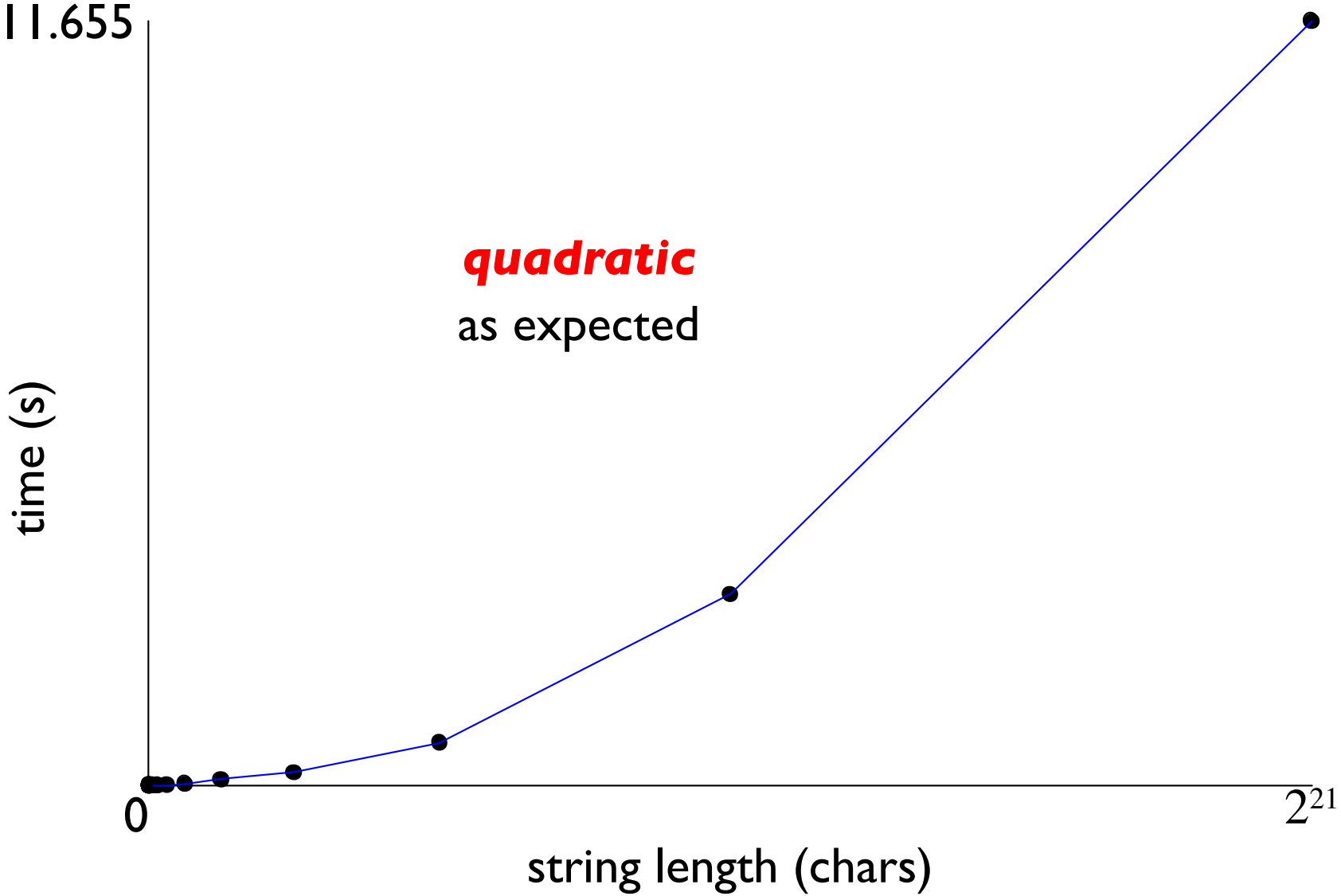
Causes **s** to change on each iteration

# Benchmark Results

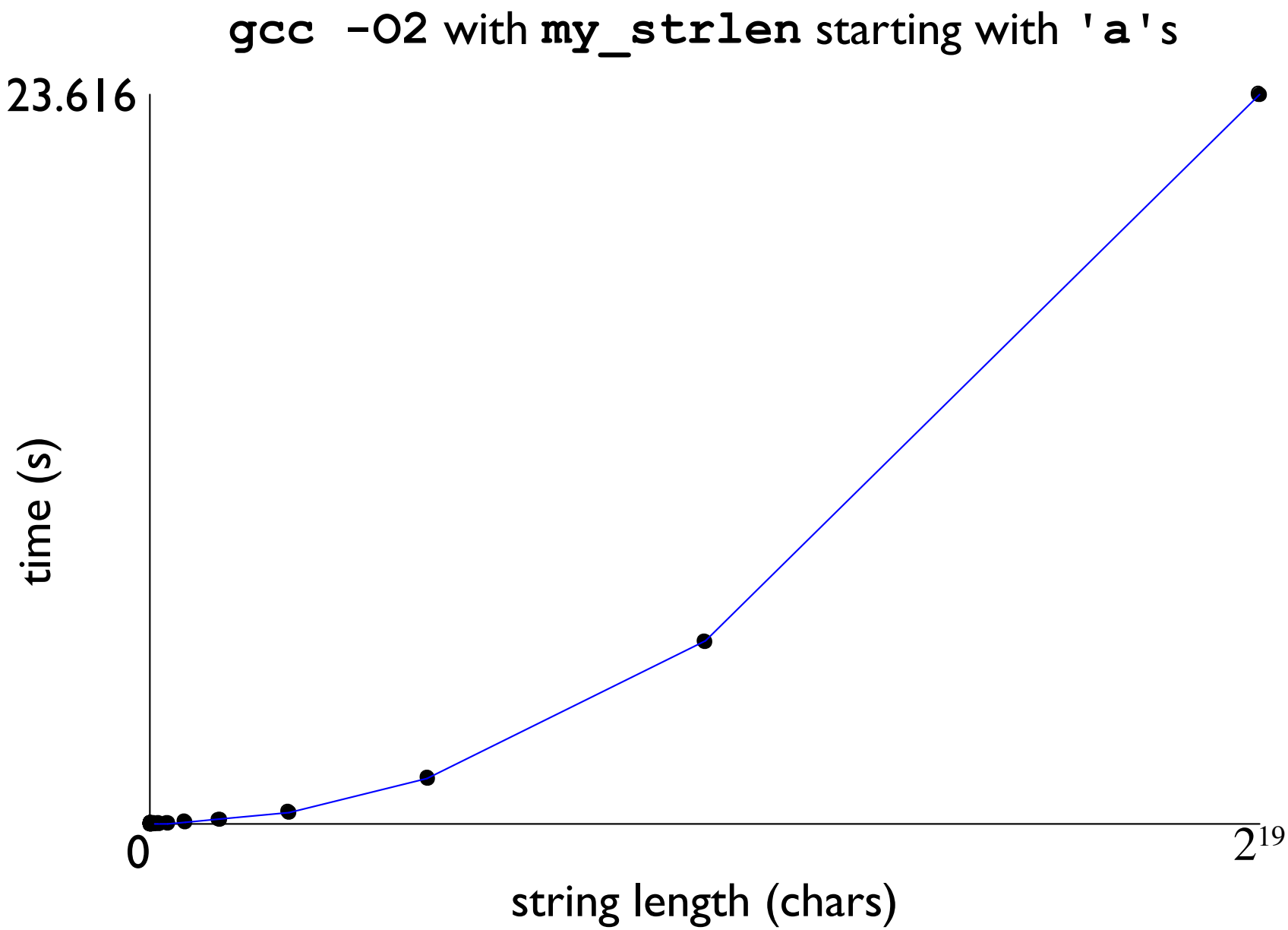


# Benchmark Results

`gcc -O2` starting with `'A'`s instead of `'a'`s



# Benchmark Results



# The Curious Case of `strlen`

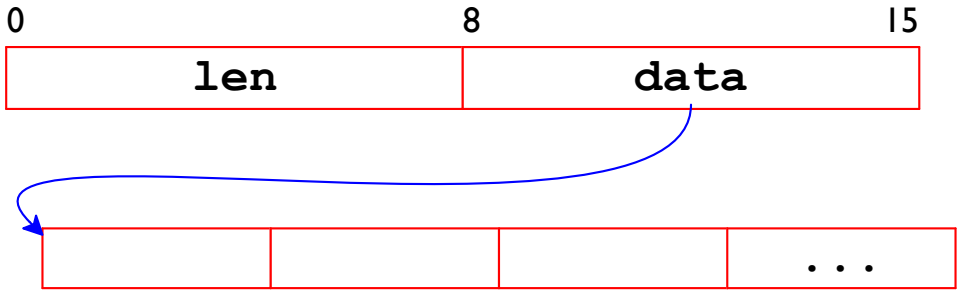
Conclusions:

Don’t use bad algorithms

Compilers can do more with known functions

# Optimization Example: Vectors

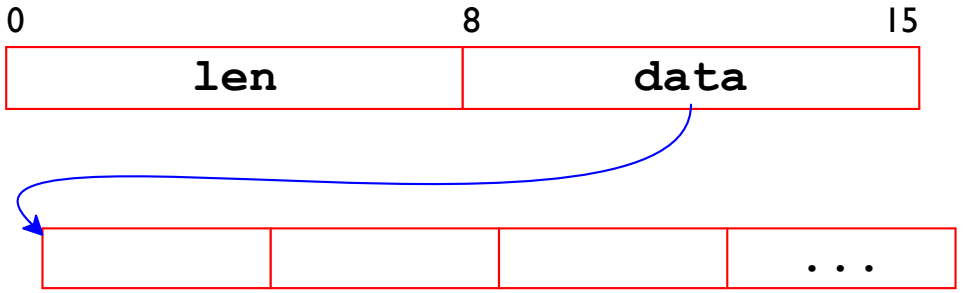
```
typedef .... data_t;

typedef struct {
  size_t len;
  data_t *data;
} vec;
```




```
vec *new_vec(size_t len) {
  vec *result = (vec *) malloc(sizeof(vec));
  result->len = len;
  result->data = calloc(len, sizeof(data_t));
  return result;
}
```

# Optimization Example: Vectors

```
typedef .... data_t;

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;
}
```

# Optimization Example: Vectors

Benchmark computation

```
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_t`** choices:

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

Operation choices:

- **`OP`** = **`+`** and **`IDENT`** = **`0`**
- **`OP`** = **`*`** and **`IDENT`** = **`1`**

# Measuring Performance

Measure loop performance as ***cycles per element*** (CPE) instead of seconds

x86-64 provides a **`rdtsc`** instruction to get approximate cycle counts

... puts result into **`%edx`** and **`%eax`**

```
rdtsc
shlq  $32, %rdx
movl  %eax, %eax
orq   %rdx, %rax
retq
```

# Measuring Performance

Measure loop performance as ***cycles per element*** (CPE) instead of seconds

x86-64 provides a **`rdtsc`** instruction to get approximate cycle counts

... puts result into **`%edx`** and **`%eax`**

```
unsigned long get_ticks() {
  unsigned int lo, hi;
  asm volatile("rdtsc" : "=a" (lo), "=d" (hi));
  return (unsigned long)hi << 32 | lo;
}
```

... but beware of Turbo Boost

# Optimization Example: Vectors

Initial measurements

```
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;
  }
}
```

| | int | | double | |
|---|---|---|---|---|
| | + | * | + | * |
| without -O1 | 22.68 | 20.02 | 19.98 | 20.18 |
| with -O1 | 10.12 | 10.12 | 10.17 | 11.14 |

# Optimization Example: Vectors

Lift **`vec_length`** out of loop

```
void combine2(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);

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

| | int | | double | |
|---|---|---|---|---|
| | + | * | + | * |
| with -O1 | 10.12 | 10.12 | 10.17 | 11.14 |
| lift **`vec_length`** | 7.02 | 9.03 | 9.02 | 11.03 |

# Optimization Example: Vectors

Direct access to vector data

```
void combine3(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);

  *dest = IDENT;
  for (i = 0; i < length; i++)
    *dest = *dest OPER data[i];
}
```

| | int | | double | |
|---|---|---|---|---|
| | + | * | + | * |
| lift `vec_length` | 7.02 | 9.03 | 9.02 | 11.03 |
| direct data | 7.17 | 9.02 | 9.02 | 11.03 |

# Optimization Example: Vectors

Accumulate result in a local variable

```
void combine4(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);
  data_t acc = IDENT;

  for (i = 0; i < length; i++)
    acc = acc OPER data[i];
  *dest = acc;
}
```

| | int | | double | |
|---|---|---|---|---|
| | + | * | + | * |
| direct data | 7.17 | 9.02 | 9.02 | 11.03 |
| accumulate to local | 1.27 | 3.01 | 3.01 | 5.01 |

# Optimization Example: Vectors

A step backwards — revert the direct data access

```
void combineX(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);
  data_t acc = IDENT;

  for (i = 0; i < length; i++) {
    data_t val;
    get_vec_element(v, i, &val);
    acc = acc OPER val;
  }
  *dest = acc;
}
```

| | int | | double | |
|---|---|---|---|---|
| | + | * | + | * |
| direct data | 7.17 | 9.02 | 9.02 | 11.03 |
| lift + accumulate local | 6.04 | 6.04 | 8.98 | 10.97 |
| accumulate to local | 1.27 | 3.01 | 3.01 | 5.01 |

# Optimization Example: Vectors

Summary

| | `int` | | `double` | |
|---|---|---|---|---|
| | `+` | `*` | `+` | `*` |
| without `-O1` | 22.68 | 20.02 | 19.98 | 20.18 |
| with `-O1` | 10.12 | 10.12 | 10.17 | 11.14 |
| lift `vec_length` | 7.02 | 9.03 | 9.02 | 11.03 |
| direct data | 7.17 | 9.02 | 9.02 | 11.03 |
| accumulate to local | 1.27 | 3.01 | 3.01 | 5.01 |