Code Optimization

Abhishek Bichhawat

Goals of compiler optimization

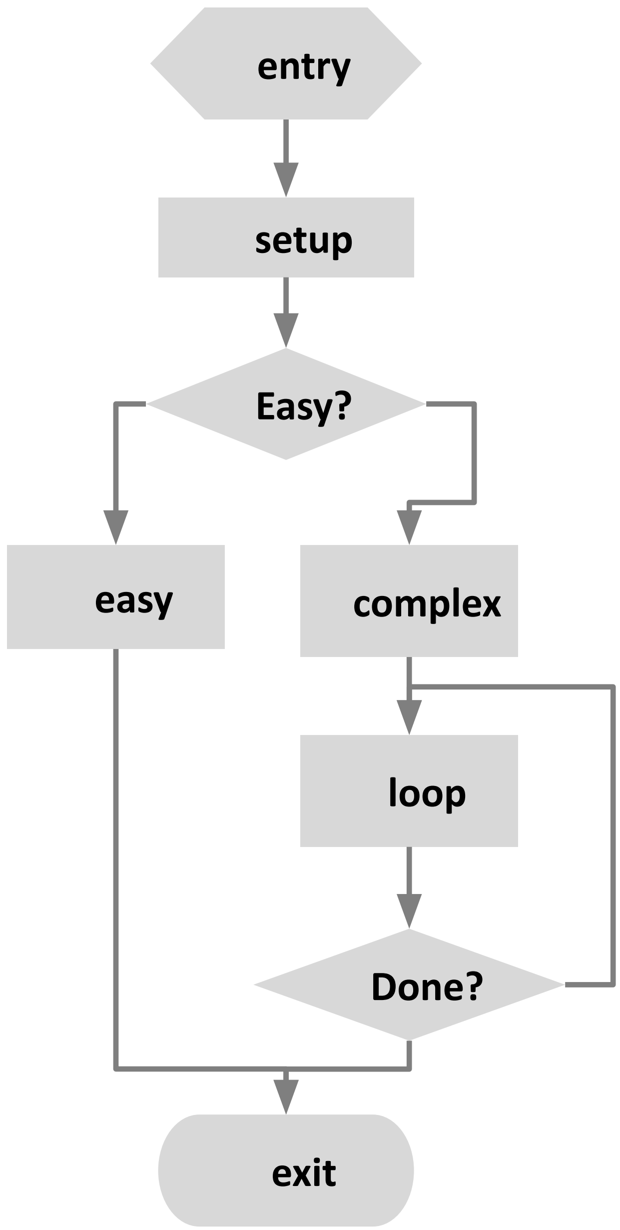
- **Minimize number of instructions**
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once
- **Avoid branching**
  - Don’t make unnecessary decisions at all
  - Make it easier for the CPU to predict branch destinations
  - “Unroll” loops to spread cost of branches over more instructions

Limits to compiler optimization

- **Generally cannot improve algorithmic complexity**
  - Only constant factors, but those can be worth 10x or more…
- **Must not cause *any* change in program behavior**
  - Programmer may not care about "edge case" behavior, but compiler does not know that
  - Exception: language may declare some changes acceptable
- **Often only analyze one function at a time**
  - Whole-program analysis ("LTO") expensive but gaining popularity
  - Exception: *inlining* merges many functions into one
- **Tricky to anticipate run-time inputs**
  - Profile-guided optimization can help with common case, but…
  - "Worst case" performance can be just as important as "normal"
  - Especially for code exposed to *malicious* input (e.g. network servers)

Two kinds of optimizations

- **Local optimizations work inside a single *basic block***
  - Constant folding, strength reduction, dead code elimination, (local) CSE, …
- **Global optimizations process the entire *control flow graph* of a function**
  - Loop transformations, code motion, (global) CSE, …



Constant folding

- **Do arithmetic in the compiler**

```
long mask = 0xFF << 8; →

long mask = 0xFF00
```

- **Any expression with constant inputs can be folded**
- **Might even be able to remove library calls…**

```
size_t namelen = strlen("Harry Bovik");

size_t namelen = 11;
```

Dead code elimination

- **Don't emit code that will never be executed**

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

- **Don't emit code whose result is overwritten**

```
x = 23;
x = 42;
```

- **These may look silly, but...**
  - Can be produced by other optimizations
  - Assignments to x might be far apart

Common subexpression elimination

- **Factor out repeated calculations, only do them once**

```
norm[i] = v[i].x*v[i].x + v[i].y*v[i].y;
```

$\longrightarrow$

```
elt = &v[i];
x = elt->x;
y = elt->y;
norm[i] = x*x + y*y;
```

Code motion

- Move calculations out of a loop
- Only valid if every iteration would produce same result

```
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];
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size; i-cache)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true

Does nothing

Can constant fold

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

Memory Aliasing

```
/* Sum rows 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];
    }
}
```

```
        movq    $0, (%rsi)
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%rdi), %xmm0
        movsd   %xmm0, (%rsi)
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

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

Memory Aliasing

```
/* Sum rows 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);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

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

Avoiding Aliasing Penalties

```
/* Sum rows 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;
    }
}
```

```
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%rdi), %xmm0
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L4
        movsd   %xmm0, (%rsi)
```

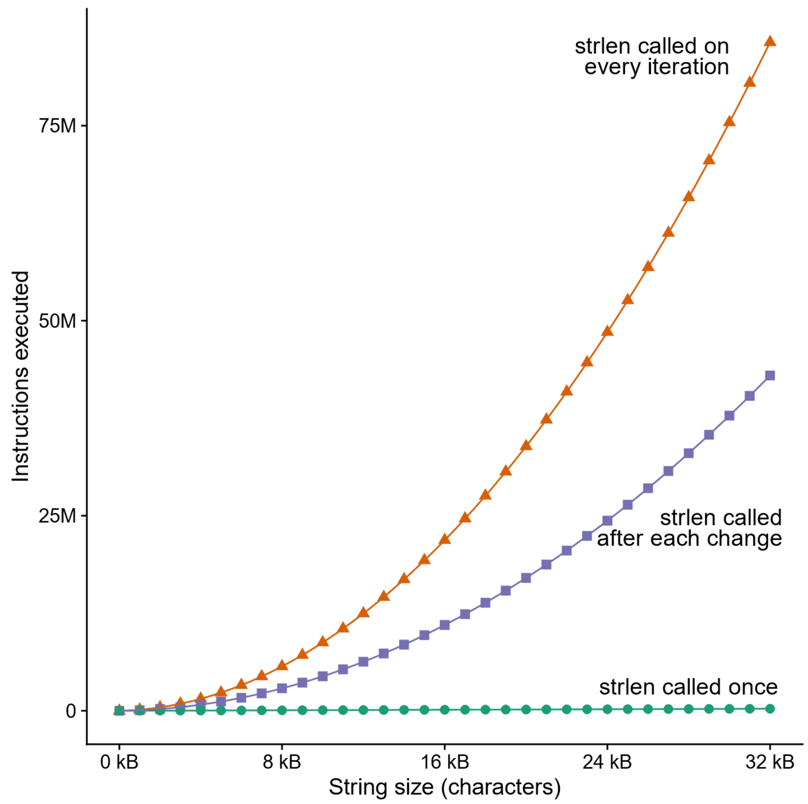
- Use a local variable for intermediate results

Can’t move function calls out of loops

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

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

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



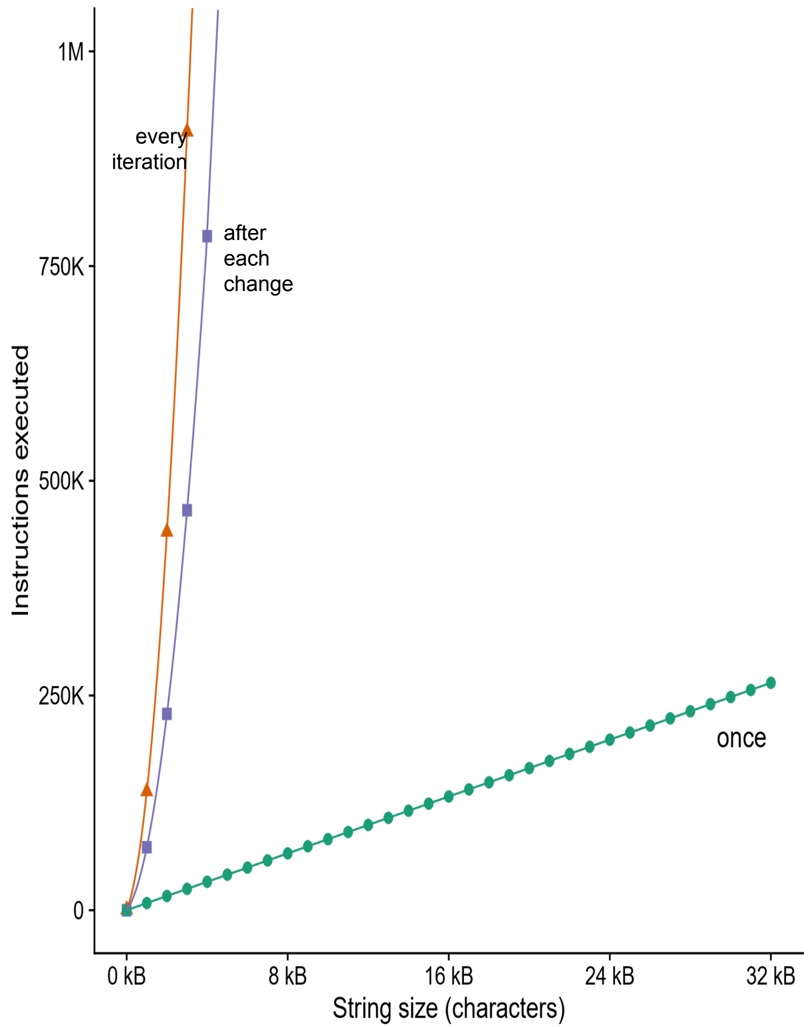
Lots more examples of this kind of bug: accidentallyquadratic.tumblr.com

Can’t move function calls out of loops

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

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

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



Strength Reduction

- X = I * 4 → x = I << 2
- Replace expensive operations with cheaper ones

Optimizing for Branch Prediction

- **Reduce # of branches**
  - Transform loops
  - Unroll loops
  - Use conditional moves
    - Not always a good idea
- **Make branches predictable**
  - Sort data
    https://stackoverflow.com/questions/11227809
  - Avoid indirect branches
    - function pointers
    - virtual methods

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    leal   -65(%rdx), %ecx
    cmpb   $25, %cl
    ja     .Lskip
    addl   $32, %edx
    movb   %dl, 0(%rbp,%rbx)
.Lskip:
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    movl   %edx, %esi
    leal   -65(%rdx), %ecx
    addl   $32, %edx
    cmpb   $25, %cl
    cmova  %esi, %edx
    movb   %dl, 0(%rbp,%rbx)
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

Memory write now unconditional!

Loop Unrolling

- **Amortize cost of loop condition by duplicating body**
- **Creates opportunities for CSE, code motion, scheduling**
- **Prepares code for vectorization**
- **Can hurt performance by increasing code size**

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

When would this change be incorrect?

Scheduling

- **Rearrange instructions to make it easier for the CPU to keep all functional units busy**
- **For instance, move all the loads to the top of an unrolled loop**
  - Now maybe it's more obvious why we need lots of registers

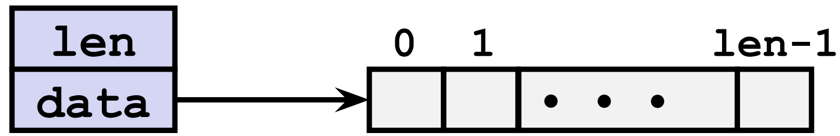
```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

When would *this* change be incorrect?

Benchmark Example: Data Type for Vectors

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




```
/* 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;
}
```

Benchmark Computation

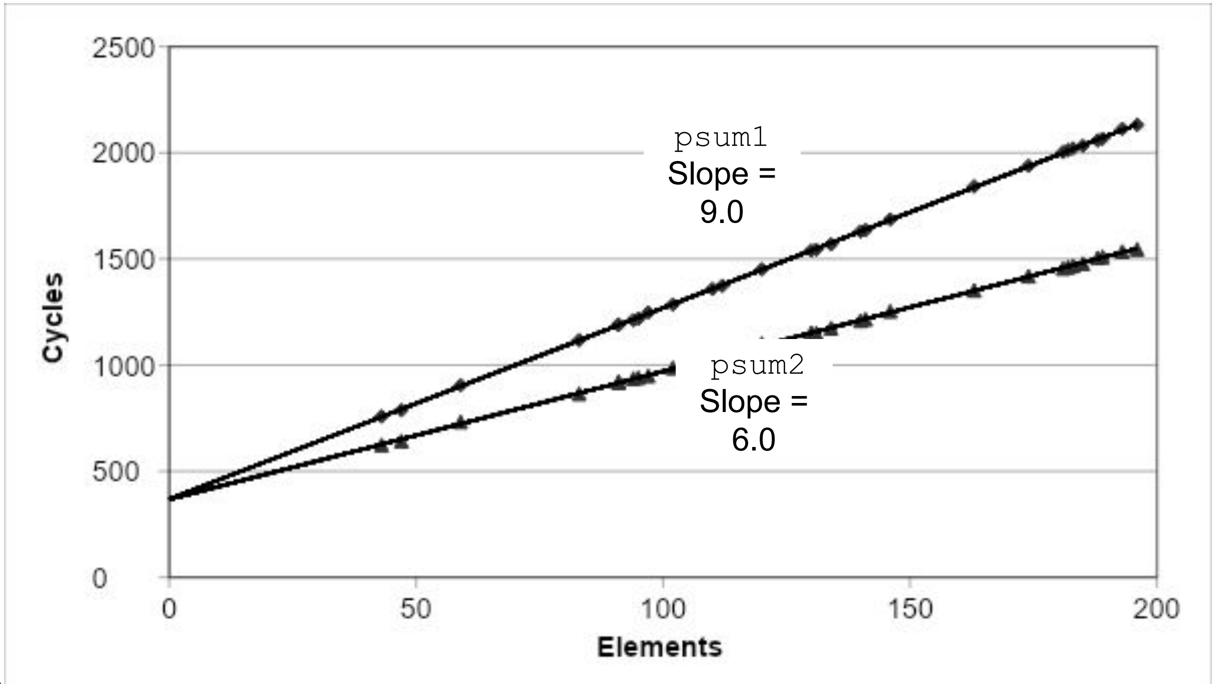
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

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

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
- Cycles = CPE*n + Overhead
  - CPE is slope of line



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
Combine1 –O3	4.5	4.5	6	7.8

Results in CPE (cycles per element)

Basic Optimizations

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

```
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;
}
```

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;
}
```

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
Combine1 –O3	4.5	4.5	6	7.8
Combine4	1.27	3.01	3.01	5.01

Loop Unrolling

```
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 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 Loop Unrolling

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
Combine1 –O3	4.5	4.5	6	7.8
Combine4	1.27	3.01	3.01	5.01
Unroll	0.81	1.51	1.51	2.51

Multiple instructions every cycle!

Going Further

- **Compiler optimizations are an easy gain**
  - 20 CPE down to 3-5 CPE
- **With careful hand tuning and computer architecture knowledge**
  - 4-16 elements per cycle
  - Newest compilers are closing this gap

Summary: Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly