Optimization

There's more to performance than asymptotic complexity(time complexity).
But all the instructions are not consume the same amount of time. Constant factors matter too! So we need to understand system to optimize performance.

- How programs are compiled and executed
- How modern processors and memory system operate
- How to measure performance and identify bottlenecks
- How to improve performance without destroying code modularity and generality

Provide efficent mapping of program to machine code

- Register allocation
- Code selection and ordering (scheduling)
- Dead code elimination
- Elimininating minor inefficiencies

Don't improve asymptotic efficiency.

Generally Useful Optimizations

Code Motion(Hoisting)

Reduce frequecy where computation performed. If it will always produce the same result, then move it to a place where it is computed once and reused.
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[i * n + j] = b[j];
    }
}
```

Default	Optimized

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

`4_1.o:  file format elf64-x86-64`
 	
`Disassembly of section .text:`
 	
`0000000000000000 <set_row>:`
`0: f3 0f 1e fa endbr64`
`4: b8 00 00 00 00 mov $0x0,%eax`
`9: eb 19 jmp 24 <set_row+0x24>`
`b: 49 89 c8 mov %rcx,%r8`
`e: 4c 0f af c2 imul %rdx,%r8`
`12: 49 01 c0 add %rax,%r8`
`15: f2 0f 10 04 c6 movsd (%rsi,%rax,8),%xmm0`
`1a: f2 42 0f 11 04 c7 movsd %xmm0,(%rdi,%r8,8)`
`20: 48 83 c0 01 add $0x1,%rax`
`24: 48 39 c8 cmp %rcx,%rax`
`27: 7c e2 jl b <set_row+0xb>`
`29: c3 ret`
	`4_2.o:  file format elf64-x86-64`
	
	`Disassembly of section .text:`
	
	`0000000000000000 <set_row>:`
	`0: f3 0f 1e fa endbr64`
	`4: 48 85 c9 test %rcx,%rcx`
	`7: 7e 20 jle 29 <set_row+0x29>`
	`9: 48 0f af d1 imul %rcx,%rdx`
	`d: 48 8d 14 d7 lea (%rdi,%rdx,8),%rdx`
	`11: b8 00 00 00 00 mov $0x0,%eax`
	`16: f2 0f 10 04 c6 movsd (%rsi,%rax,8),%xmm0`
	`1b: f2 0f 11 04 c2 movsd %xmm0,(%rdx,%rax,8)`
	`20: 48 83 c0 01 add $0x1,%rax`
	`24: 48 39 c1 cmp %rax,%rcx`
	`27: 75 ed jne 16 <set_row+0x16>`
	`29: c3 ret`

`imul` is located in the loop.	can see that `imul` is located out of the loop.

Above two codes have same number of instructions. But optimized version has **fewer executed instructions**.

GCC will do this with `-O1` options

Reduction in Strength

Replace costly operation with simpler one.

for example: power of 2 multiply to shift operation. normally, multiply and divide are expensive exmaple. on Intel Nehalem, `imul` requires 3 CPU cylcles on the other hand, `add` requires 1 cycle.

Default

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

Optimized

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

Share Common Subexpressions

Reuse portations of expressions

GCC will do this with `-O1`

Default

```
double test_scs(double* val, long i, long j, long n) {
    double up, down, left, right;

    up = val[(i - 1) * n + j];
    down = val[(i + 1) * n + j];
    left = val[i * n + (j - 1)];
    right = val[i * n + (j + 1)];
    return up + down + left + right;
}
```

Optimized

```
double test_scs_opt(double *a, double *b, long i, long n) {
    double up, down, left, right;

    long inj = i * n + j;

    up = a[inj - n];
    down = a[inj + n];
    left = b[inj - 1];
    right = b[inj + 1];
    return up + down + left + right;
}
```

```
4_3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <test_scs>:
   0:   f3 0f 1e fa             endbr64
   4:   48 89 f8                mov    %rdi,%rax
   7:   48 83 ee 01             sub    $0x1,%rsi
   b:   48 0f af f1             imul   %rcx,%rsi
   f:   48 8d 3c 4e             lea    (%rsi,%rcx,2),%rdi
  13:   48 8d 0c 0e             lea    (%rsi,%rcx,1),%rcx
  17:   48 01 d6                add    %rdx,%rsi
  1a:   48 01 d7                add    %rdx,%rdi
  1d:   f2 0f 10 04 f0          movsd  (%rax,%rsi,8),%xmm0
  22:   f2 0f 58 04 f8          addsd  (%rax,%rdi,8),%xmm0
  27:   48 01 ca                add    %rcx,%rdx
  2a:   f2 0f 58 44 d0 f8       addsd  -0x8(%rax,%rdx,8),%xmm0
  30:   f2 0f 58 44 d0 08       addsd  0x8(%rax,%rdx,8),%xmm0
  36:   c3                      ret
```

Above dump shows only one `imul`, which shows that share common subexpressions are applied.

Remove Unnecessary Procedure

Think with your intuition.

Optimization Blockers

Compilers cannot always optimize your code.

```
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');
        }
    }
}
```

Above code's performance is bad. time quadruples when double string length.
Because `strlen` is executed on every loop. so `strlen` is $O(n)$, therefore overall performance of `lower` is $O(n^2)$
Therefore we optimized by Code Motion by moving the calculation length parts to out of the loop.

```
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');
        }
    }
}
```

#1 Procedure Calls

Procedure may have side effects. and Function may not return same value for given arguments.
So compiler treats procedure call as a black box. Weak optimizations near them. Therefore strong optimizations like **Code Motion** are not applied.
In order to apply strong optimizations, First, use of inline function with `-O1` option, or **do your self**.

Memory Aliasing

```
void sum_rows(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];
        }
    }
}
```

```
4_4.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <sum_rows>:
   0:   f3 0f 1e fa             endbr64
   4:   48 85 d2                testq  %rdx,%rdx
   7:   7e 4d                   jle    56 <sum_rows+0x56>
   9:   49 89 f0                movq   %rsi,%r8
   c:   4c 8d 0c d5 00 00 00    leaq   0x0(,%rdx,8),%r9
  13:   00
  14:   4a 8d 0c 0f             leaq   (%rdi,%r9,1),%rcx
  18:   4a 8d 3c 0e             leaq   (%rsi,%r9,1),%rdi
  1c:   48 f7 da                negq   %rdx
  1f:   48 8d 34 d5 00 00 00    leaq   0x0(,%rdx,8),%rsi
  26:   00
  27:   4c 89 c2                movq   %r8,%rdx
  2a:   49 c7 00 00 00 00 00    movq   $0x0,(%r8)
  31:   48 8d 04 31             leaq   (%rcx,%rsi,1),%rax
  35:   f2 0f 10 02             movsd  (%rdx),%xmm0
  39:   f2 0f 58 00             addsd  (%rax),%xmm0
  3d:   f2 0f 11 02             movsd  %xmm0,(%rdx)
  41:   48 83 c0 08             addq   $0x8,%rax
  45:   48 39 c8                cmpq   %rcx,%rax
  48:   75 eb                   jne    35 <sum_rows+0x35>
  4a:   49 83 c0 08             addq   $0x8,%r8
  4e:   4c 01 c9                addq   %r9,%rcx
  51:   49 39 f8                cmpq   %rdi,%r8
  54:   75 d1                   jne    27 <sum_rows+0x27>
  56:   c3                      retq
```

Compiler leave `b[i]` on every iteration. Because compiler must consider possibility that the updates will affect program behavior. (`b` and `a` is shared, memory aliasing)

Memory aliasing means two different memory references specify single location.

in C, it is easy to have happen. because address arithmetic and direct access to storage structures.

```
void sum_rows(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;
    }
}
```

```
4_5.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <sum_rows>:
   0:   f3 0f 1e fa             endbr64
   4:   48 85 d2                testq  %rdx,%rdx
   7:   7e 3f                   jle    48 <sum_rows+0x48>
   9:   49 89 f0                movq   %rsi,%r8
   c:   4c 8d 0c d5 00 00 00    leaq   0x0(,%rdx,8),%r9
  13:   00
  14:   4a 8d 0c 0f             leaq   (%rdi,%r9,1),%rcx
  18:   4c 01 ce                addq   %r9,%rsi
  1b:   48 f7 da                negq   %rdx
  1e:   48 c1 e2 03             shlq   $0x3,%rdx
  22:   48 8d 04 11             leaq   (%rcx,%rdx,1),%rax
  26:   66 0f ef c0             pxor   %xmm0,%xmm0
  2a:   f2 0f 58 00             addsd  (%rax),%xmm0
  2e:   48 83 c0 08             addq   $0x8,%rax
  32:   48 39 c8                cmpq   %rcx,%rax
  35:   75 f3                   jne    2a <sum_rows+0x2a>
  37:   f2 41 0f 11 00          movsd  %xmm0,(%r8)
  3c:   49 83 c0 08             addq   $0x8,%r8
  40:   4c 01 c9                addq   %r9,%rcx
  43:   49 39 f0                cmpq   %rsi,%r8
  46:   75 da                   jne    22 <sum_rows+0x22>
  48:   c3                      retq
```

By introducing local local variables, we can easy to get optimized code.

Exploiting Instruction-Level Parallelism(ILP)

Execute multiple instructions at the same time. it can reduce average instruction cycle, which needs general understanding of modern processor design: HW can execute many operations in parallel.

- performance limited by data dependency

simple transformations can yield dramatic performance improvement.

Superscalar Processors

Issue and Execute Multiple Instructions in one cycle.
pipelining -> data dependency.
for example Haswell CPU Functional Units

- 2 load
- 1 store
- 4 integer
- 2 FP mult
- 1 FP add
- 1 FP div
- 1 int mult

Programming with AVX2

YMM register: 256bit, total 16 registers.

SIMD Operations

for single precision
`vaddps %ymm0, %ymm1, %ymm1`:
for double precision
`vaddpd %ymm0, %ymm1, %ymm1`