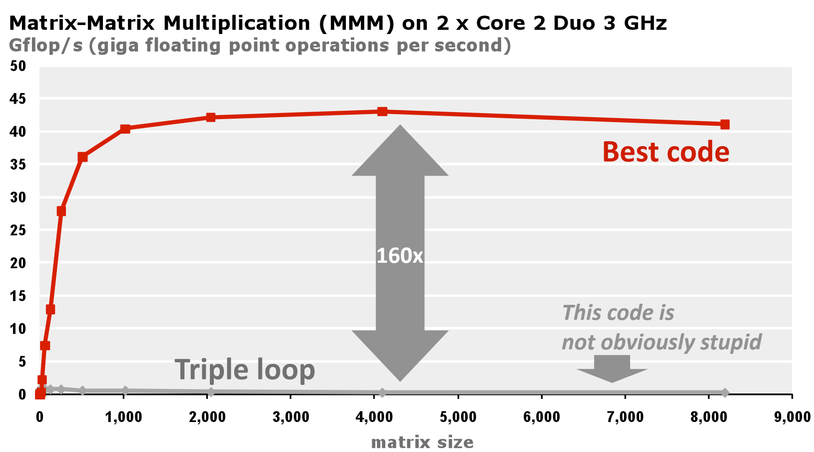
Today

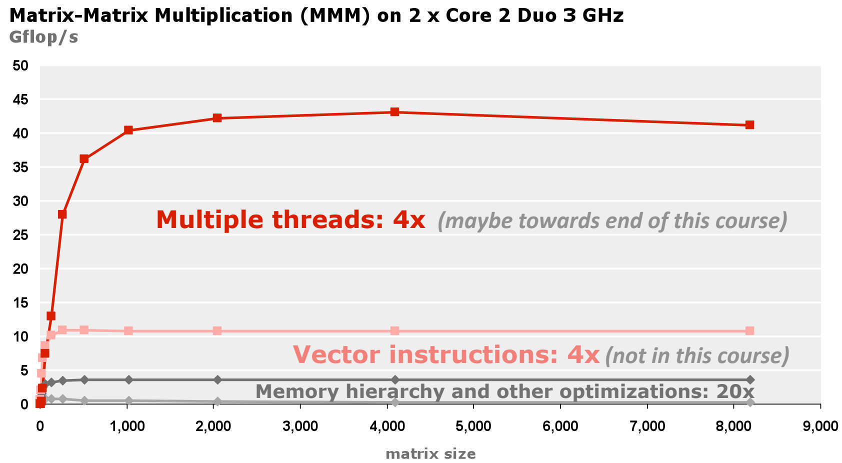
- **Program optimization**
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

Example Matrix Multiplication



- **Standard desktop computer, compiler, using optimization flags**
- **Both implementations have exactly the same operations count ($2n^3$)**
- ***What is going on?***

MMM Plot: Analysis



- **Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice**
- ***Effect: more instruction level parallelism, better register use, less L1/L2 cache misses, less TLB misses***

Harsh Reality

- ***There's more to runtime performance than asymptotic complexity***
- ***One can easily loose 10x, 100x in runtime or even more***
- **What matters:**
  - Constants (100n and 5n is both O(n), but ….)
  - Coding style (unnecessary procedure calls, unrolling, reordering, …)
  - Algorithm structure (locality, instruction level parallelism, …)
  - Data representation (complicated structs or simple arrays)

Harsh Reality

- **Must optimize at multiple levels:**
  - Algorithm
  - Data representations
  - Procedures
  - Loops

- **Must understand system to optimize performance**
  - How programs are compiled and executed
    - Execution units, memory hierarchy
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Optimizing Compilers

-O

- **Use optimization flags, default is no optimization (-O0)!**
- **Good choices for gcc: -O2, -O3, -march=xxx, -m64**
- **Try different flags and maybe different compilers**

Example

```
double a[4][4];
double b[4][4];
double c[4][4]; # set to zero

/* Multiply 4 x 4 matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}
```

- **Compiled without flags:**
  ~1300 cycles
- **Compiled with –O3 –m64 -march=… –fno-tree-vectorize**
  ~150 cycles
- Core 2 Duo, 2.66 GHz

Optimizing Compilers

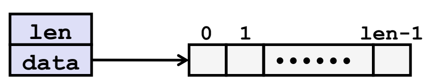
- **Compilers are good at: mapping program to machine**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **Compilers are not good at: improving asymptotic efficiency**
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- **Compilers are not good at: overcoming "optimization blockers"**
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- *If in doubt, the compiler is conservative*
- **Operate under fundamental constraints**
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - e.g., data ranges may be more limited than variable types suggest
- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs

Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
        int len;
        double *data;
} vec;
```




```
/* retrieve vector element and store at val */
int get_vec_element(vec *v, int idx, double *val)
{
        if (idx < 0 || idx >= v->len)
                return 0;
        *val = v->data[idx];
        return 1;
}
```

Example: Summing Vector Elements

```
double get_vec_element(vec *v, int idx,
                                  double *val)
{
  if (idx < 0 || idx >= v->len)
        return 0;
  *val = v->data[idx];
  return 1;
}
```

Bound check unnecessary in sum_elements
Why?

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = v->len;
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
         *res += val;
  }
  return res;
}
```

Overhead for every fp +:
- One fct call
- One <
- One >=
- One ||
- One memory variable access

Slowdown:
probably 10x or more

Removing Procedure Call

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = v->len;
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
         *res += val;
  }
  return res;
}
```

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = v->len;
  *res = 0.0;
  double *data = get_vec_start(v);

  for (i = 0; i < n; i++)
         *res += data[i];
  return res;
}
```

Removing Procedure Calls

- **Procedure calls can be very expensive**
- **Bounds checking can be very expensive**
- **Abstract data types can easily lead to inefficiencies**
  - Usually avoided in superfast numerical library functions

- **Watch your innermost loop!**

- **Get a feel for overhead versus actual computation being performed**

Code Motion

- **Reduce frequency with which computation is performed**
  - If it will always produce same result
  - Especially moving code out of loop
- **Sometimes also called pre-computation**

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

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

Compiler-Generated Code Motion

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

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

```
copy_row:
        xorl    %r8d, %r8d           #  j = 0
        cmpq    %rcx, %r8            #  j:n
        jge     .L7                  #  if >= goto done
        movq    %rcx, %rax           #  n
        imulq   %rdx, %rax           #  n*i outside of inner loop
        leaq    (%rdi,%rax,8), %rdx  #  rowp = A + n*i*8
.L5:                                 # loop:
        movq    (%rsi,%r8,8), %rax   #  t = b[j]
        incq    %r8                  #  j++
        movq    %rax, (%rdx)         #  *rowp = t
        addq    $8, %rdx             #  rowp++
        cmpq    %rcx, %r8            #  j:n
        jl      .L5                  #  if < goto loop
.L7:                                 # done:
        rep ; ret                    #  return
```


Strength Reduction

- **Replace costly operation with simpler one**
- **Example: Shift/add instead of multiply or divide**

  `16*x  →  x << 4`

  - Depends on cost of multiply or divide instruction
  - On Pentium IV, integer multiply requires 10 CPU cycles
- **Example: Recognize sequence of products**

```
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;
}
```

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

*3 mults: i*n, (i–1)*n, (i+1)*n*

```
/* 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;
```

```
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
```

*1 mult: i*n*

```
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;
```

```
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
```

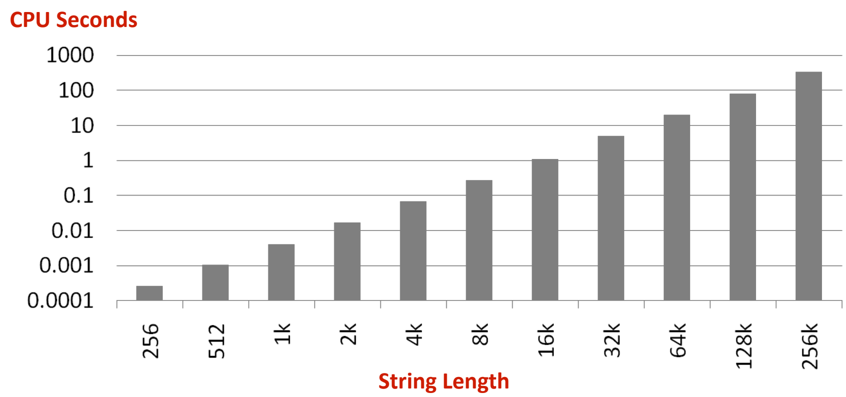

Optimization Blocker: Procedure Calls

- Procedure to convert string to lower 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');
}
```

Performance

- Time quadruples when double string length
- Quadratic performance



Why is That?

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

- **String length is called in every iteration!**
  - And **`strlen`** is O(n), so **`lower`** is $O(n^2)$

```
/* A version of strlen */
size_t strlen(char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

Improving Performance

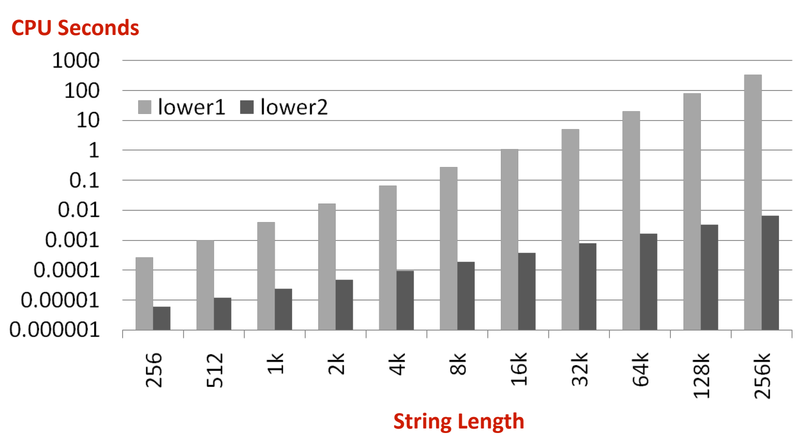
```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
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');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion/precomputation

Performance

- Lower2: Time doubles when double string length
- Linear performance



Optimization Blocker: Procedure Calls

- **Why couldn't compiler move `strlen` out of inner loop?**
  - Procedure may have side effects
  - Function may not return same value for given arguments
    - Could depend on other parts of global state
    - Procedure `lower` could interact with `strlen`
- **Compiler usually treats procedure call as a black box that cannot be analyzed**
  - Consequence: conservative in optimizations
- **Remedies:**
  - Inline the function if possible
  - Do your own code motion

```
int lencnt = 0;
size_t strlen(char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```


Optimization Blocker: Memory Aliasing

```
// add twice the value stored at yp to the value stored at xp

void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2*(*yp);
}
```

- **twiddle1 appears to be less efficient**
  - 6 memory references: two reads each of *yp and *xp, two writes of *xp
- **twiddle2 appears to be more efficient**
  - 3 memory references: read *yp, read *xp, write *xp
- **Can a compiler come up with twiddle2 if given twiddle1?**

Optimization Blocker: Memory Aliasing

```
// add twice the value stored at yp to the value stored at xp
// *xp = *xp + 2 * *yp;

void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2*(*yp);
}
```

- **But what if xp == yp?**
  - twiddle1 quadruples value at xp
  - twiddle2 triples value at xp
- **Because of this 'aliasing', compiler does not optimize twiddle1**
  - Would lead to different result
  - Assume twiddle1 is programmer's intent

University of Washington

Optimization Blocker: Memory Aliasing

```
x = 1000;
y = 3000;
*q = y;
*p = x;
return *q;
```

- **What is the return value?**
- **Two cases:**
  - q and p are different addresses
  - q and p are aliases for the same address

Optimization Blocker: Memory Aliasing

- **Memory aliasing: Two different memory references write to the same location**
- **Can happen easily in C**
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- **Hard to analyze = compiler cannot figure it out**
  - Hence the compiler is conservative

A Solution to Aliasing

- **Apply a programming style consistently**
  - Copy values for memory variables into local variables
  - Then assign local variables to final destinations

```
x = 1000;
y = 3000;
*q = y;
*p = x;
return *q;
```

```
x = 1000;
y = 3000;
temp1 = y;
temp2 = x;
*q = temp1;
*p = temp2;
return temp1;
```