Code Optimization

15-213/18-213/14-513/15-513: Introduction to Computer Systems
10th Lecture, February 19, 2019
Today

- **Overview**

- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions

- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing

- **Exploiting Instruction-Level Parallelism**

- **Dealing with Conditionals**
Performance Realities

- **There’s more to performance than asymptotic complexity**
- **Constant factors matter too!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Optimizing Compilers

- **Provide efficient mapping of program to machine**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- **Don’t (usually) 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

- **Have difficulty overcoming “optimization blockers”**
  - potential memory aliasing
  - potential procedure side-effects
Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```c
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];
}
```
Compiler-Generated Code Motion (-O1)

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

set_row:
    testq %rcx, %rcx
    jle .L1
    imulq %rcx, %rdx
    leaq (%rdi,%rdx,8), %rdx
    movl $0, %eax
    .L3:
    movsd (%rsi,%rax,8), %xmm0
    movsd %xmm0, (%rdx,%rax,8)
    addq $1, %rax
    cmpq %rcx, %rax
    jne .L3
    .L1:
    rep ; ret
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \quad \rightarrow \quad x << 4 \]
  - Utility is machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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

```c
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
- GCC will do this with \(-O1\)

```c
/* 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: \(i*n\), \((i-1)*n\), \((i+1)*n\)

```asm
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
...
```

```asm
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: \(i*n\)

```asm
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
...
```
Today

- Overview

- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions

- Optimization Blockers
  - Procedure calls
  - Memory aliasing

- Exploiting Instruction-Level Parallelism

- Dealing with Conditionals
Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that 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
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files

- **Most analysis is based only on static information**
  - Compiler has difficulty anticipating run-time inputs

- **When in doubt, the compiler must be conservative**
Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```c
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');
}
```

- Extracted from 213 lab submissions, Fall, 1998
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
Convert Loop To Goto Form

```c
void lower(char *s) {
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
    done:
}
```

- `strlen` executed every iteration
Calling Strlen

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

- Strlen performance
  - Only way to determine length of string is to scan its entire length, looking for null character.

- Overall performance, string of length N
  - N calls to strlen
  - Require times N, N-1, N-2, ..., 1
  - Overall O(N^2) performance
Improving Performance

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');
}

- Move call to `strlen` outside of loop
- Legal since result does not change from one iteration to another
- Form of code motion
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2
Optimization Blocker: Procedure Calls

**Why couldn’t compiler move strlen out of inner loop?**
- Procedure may have side effects
  - Alters global state each time called
- Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`

**Warning:**
- Compiler may treat procedure call as a black box
- Weak optimizations near them

**Remedies:**
- Use of inline functions
  - GCC does this with –O1
    - Within single file
- Do your own code motion

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

/* 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];
    }
}

# sum_rows1 inner loop
.L4:
    movsd (%rsi,%rax,8), %xmm0  # FP load
    addsd (%rdi), %xmm0          # FP add
    movsd %xmm0, (%rsi,%rax,8)   # FP store
    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 is 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 = A+3;
sum_rows1(A, B, 3);

- Code updates \( b[i] \) on every iteration
- Must consider possibility that these updates will affect program behavior

Value of B:

- init: [4, 8, 16]
- i = 0: [3, 8, 16]
- i = 1: [3, 22, 16]
- i = 2: [3, 22, 224]
Removing Aliasing

/* Sum rows is 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;
    }
}

# sum_rows2 inner loop
.L10:
    addsd (%rdi), %xmm0      # FP load + add
    addq $8, %rdi
    cmpq %rax, %rdi
    jne   .L10

- No need to store intermediate results
Optimization Blocker: Memory Aliasing

**Aliasing**

- Two different memory references specify single location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Accumulating within loops
  - *Your way of telling compiler not to check for aliasing*
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/8555
Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Example: Bubblesort
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Conservative compilers often cannot make these transformations
  - E.g., Lack of associativity and distributivity in FP arithmetic
Benchmark Example: Data Type for Vectors

```c
/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;
```

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

```c
/* 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

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

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

\[ T = \text{CPE} \times n + \text{Overhead} \]
Benchmark Performance

```c
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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine1 –O3</td>
<td>4.5</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Results in CPE (cycles per element)
Basic Optimizations

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

```c
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

```c
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;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop
Modern CPU Design

**Instruction Control**

- **Instruction Cache**
  - Instructions
  - Address

- **Instruction Decode**
  - Operations

- **Fetch Control**
  - Instructions

- **Retirement Unit**
  - Register File
  - Prediction OK?
  - Register Updates

**Execution**

- **Data Cache**
  - Data
  - Addr.

- **Functional Units**
  - Branch
  - Arith
  - Arith
  - Arith
  - Load
  - Store

  - Operation Results
  - Addr.
  - Data

Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*.

- The processor fetches instructions sequentially and executes *out of order (OoO)* via *dynamic scheduling*.

- **Benefit:** without programming effort, OoO superscalar execution can take advantage of the *instruction level parallelism (ILP)* that most programs have.

- **Many modern CPUs are superscalar.**
  - Intel: Pentium (1993)
  - Pentium Pro (1995) [Architect: Bob Colwell, CMU PhD Alum]
  - Some new low-power chips are reverting to in-order to save power
### Pipelined Functional Units

```c
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles
Haswell CPU

- 8 Total Functional Units

Multiple instructions can execute in parallel
- 2 load, with address computation
- 1 store, with address computation
- 4 integer
- 2 FP multiply
- 1 FP add
- 1 FP divide

Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

**Inner Loop (Case: Integer Multiply)**

```
.L519:       # Loop:
    imull (%rax,%rdx,4), %ecx    # t = t * d[i]
    addq $1, %rdx               # i++
    cmpq %rdx, %rbp             # Compare length:i
    jg .L519                   # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td><strong>Latency Bound</strong></td>
<td><strong>1.00</strong></td>
<td><strong>3.00</strong></td>
</tr>
</tbody>
</table>
Combine4 = Serial Computation (OP = *)

- Computation (length=8)
  \[ (((((1 \times d[0]) \times d[1]) \times d[2]) \times d[3]) \times d[4]) \times d[5]) \times d[6]) \times d[7]) \]

- Sequential dependence
  - Performance: determined by latency of OP
Loop Unrolling (2x1)

```c
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 x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration
Effect of Loop Unrolling

- Helps integer add
  - Achieves latency bound
  - Eliminates overheads: e.g., index computation

- Others don’t improve. Why?
  - Still sequential dependence
    \[ x = (x \text{ OP } d[i]) \text{ OP } d[i+1]; \]

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Loop Unrolling with Reassociation (2x1a)

void unroll2aa_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 x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}

- Can this change the result of the computation?
- Yes, for FP. *Why?*
### Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td><em>Latency Bound</em></td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td><em>Throughput Bound</em></td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency
    
    \[
    x = x \text{ OP} (d[i] \text{ OP} d[i+1]);
    \]
  - Why is that? (next slide)

- Not .25? Limited by loads
- 4 func. units for int +,
  2 func. units for load
- 1 func. unit for FP +
  3-stage pipelined FP +
- 2 func. units for FP *,
  2 func. units for load
  5-stage pipelined FP *
Reassociated Computation

What changed:
- Ops in the next iteration can be started early (no dependency)

Overall Performance
- N elements, D cycles latency/op
- \((N/2+1)\)*D cycles:
  \[ \text{CPE} = \frac{D}{2} \]

Is this transformation correct?

\[ x = x \text{ OP} (d[i] \text{ OP} d[i+1]); \]
Loop Unrolling with Separate Accumulators (2x2)

```c
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;
}
```

- Different form of reassociation
### Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>

- **Int +** makes use of two load units

```plaintext
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- **2x speedup (over unroll2) for Int *, FP +, FP ***
Separate Accumulators

\[ x_0 = x_0 \text{ OP } d[i]; \]
\[ x_1 = x_1 \text{ OP } d[i+1]; \]

- **What changed:**
  - Two independent “streams” of operations

- **Overall Performance**
  - \( N \) elements, \( D \) cycles latency/op
  - Should be \((N/2+1)*D\) cycles:
    \[ \text{CPE} = D/2 \]
  - CPE matches prediction!

*What Now?*
Unrolling & Accumulating

■ Idea
  ▪ Can unroll to any degree L
  ▪ Can accumulate K results in parallel
  ▪ L must be multiple of K

■ Limitations
  ▪ Diminishing returns
    ▪ Cannot go beyond throughput limitations of execution units
  ▪ Large overhead for short lengths
    ▪ Finish off iterations sequentially
Unrolling & Accumulating: Double *

Case

- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Accumulators
Achievable Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td>0.54</td>
<td>1.01</td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code
Programming with AVX2

YMM Registers

- 16 total, each 32 bytes
- 32 single-byte integers
- 16 16-bit integers
- 8 32-bit integers
- 8 single-precision floats
- 4 double-precision floats
- 1 single-precision float
- 1 double-precision float
SIMD Operations

- SIMD Operations: Single Precision
  
  \texttt{vaddps} \hspace{1em} \texttt{%ymm0}, \hspace{1em} \texttt{%ymm1}, \hspace{1em} \texttt{%ymm1}

- SIMD Operations: Double Precision
  
  \texttt{vaddpd} \hspace{1em} \texttt{%ymm0}, \hspace{1em} \texttt{%ymm1}, \hspace{1em} \texttt{%ymm1}
Using Vector Instructions

Make use of AVX Instructions
- Parallel operations on multiple data elements
- See Web Aside OPT:SIMD on CS:APP web page

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>
What About Branches?

**Challenge**

- **Instruction Control Unit** must work well ahead of **Execution Unit** to generate enough operations to keep EU busy

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

... 

404685:  repz retq
```

- When encounters conditional branch, cannot reliably determine where to continue fetching
Modern CPU Design

Instruction Control

- Fetch Control
- Instruction Decode
- Instruction Cache
- Operations
- Prediction OK?
- Register File
- Instruction Control

Functional Units

- Branch
- Arith
- Arith
- Arith
- Load
- Store

Data Cache

Operation Results

Register Updates

Addr.

Data
Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```assembly
404663:  mov       $0x0,%eax
404668:  cmp       (%rdi),%rsi
40466b:  jge       404685  ; Branch Not-Taken
40466d:  mov       0x8(%rdi),%rax  ; Branch Taken
          ...  ; Branch Not-Taken
404685:  repz retq
```
Branch Prediction

- Idea
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don’t actually modify register or memory data *(How?)*

```
404663: mov $0x0,%eax
404668: cmp (%rdi),%rsi
40466b: jge 404685
40466d: mov 0x8(%rdi),%rax

...  
```

Predict Taken

```
404685: repz retq
```

Begin Execution
### Branch Prediction Through Loop

<table>
<thead>
<tr>
<th>Line</th>
<th>Opcode</th>
<th>Arguments</th>
<th>Condition</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029</td>
<td>vmulsd</td>
<td>(%rdx),%xmm0,%xmm0</td>
<td>i = 98</td>
<td>401029</td>
</tr>
<tr>
<td>40102d</td>
<td>add</td>
<td>$0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031</td>
<td>cmp</td>
<td>%rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034</td>
<td>jne</td>
<td>401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Assume**
  - Vector length = 100

<table>
<thead>
<tr>
<th>Line</th>
<th>Opcode</th>
<th>Arguments</th>
<th>Condition</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029</td>
<td>vmulsd</td>
<td>(%rdx),%xmm0,%xmm0</td>
<td>i = 99</td>
<td>401029</td>
</tr>
<tr>
<td>40102d</td>
<td>add</td>
<td>$0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031</td>
<td>cmp</td>
<td>%rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034</td>
<td>jne</td>
<td>401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Predict Taken (OK)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Opcode</th>
<th>Arguments</th>
<th>Condition</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029</td>
<td>vmulsd</td>
<td>(%rdx),%xmm0,%xmm0</td>
<td>i = 100</td>
<td>401029</td>
</tr>
<tr>
<td>40102d</td>
<td>add</td>
<td>$0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031</td>
<td>cmp</td>
<td>%rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034</td>
<td>jne</td>
<td>401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Predict Taken (Oops)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Opcode</th>
<th>Arguments</th>
<th>Condition</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029</td>
<td>vmulsd</td>
<td>(%rdx),%xmm0,%xmm0</td>
<td>i = 101</td>
<td>401029</td>
</tr>
<tr>
<td>40102d</td>
<td>add</td>
<td>$0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031</td>
<td>cmp</td>
<td>%rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034</td>
<td>jne</td>
<td>401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Read invalid location**

**Executed**

**Fetched**
Branch Misprediction Invalidation

Assume
vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Invalidate

\[
\begin{align*}
401029: & \text{ vmulsd } (%rdx),%xmm0,%xmm0 \\
40102d: & \text{ add } $0x8,%rdx \\
401031: & \text{ cmp } %rax,%rdx \\
401034: & \text{ jne } 401029 \\
\end{align*}
\]

\[
\begin{align*}
i = 98
\end{align*}
\]

\[
\begin{align*}
401029: & \text{ vmulsd } (%rdx),%xmm0,%xmm0 \\
40102d: & \text{ add } $0x8,%rdx \\
401031: & \text{ cmp } %rax,%rdx \\
401034: & \text{ jne } 401029 \\
\end{align*}
\]

\[
\begin{align*}
i = 99
\end{align*}
\]

\[
\begin{align*}
401029: & \text{ vmulsd } (%rdx),%xmm0,%xmm0 \\
40102d: & \text{ add } $0x8,%rdx \\
401031: & \text{ cmp } %rax,%rdx \\
401034: & \text{ jne } 401029 \\
\end{align*}
\]

\[
\begin{align*}
i = 100
\end{align*}
\]

\[
\begin{align*}
401029: & \text{ vmulsd } (%rdx),%xmm0,%xmm0 \\
40102d: & \text{ add } $0x8,%rdx \\
401031: & \text{ cmp } %rax,%rdx \\
401034: & \text{ jne } 401029 \\
\end{align*}
\]

\[
\begin{align*}
i = 101
\end{align*}
\]
Branch Misprediction Recovery

- Performance Cost
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

```
401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp %rax,%rdx
401034: jne 401029
401036: jmp 401040
    ... i = 99
401040: vmovsd %xmm0,(%r12)
```
Branch Prediction Numbers

- Default behavior:
  - Backwards branches are often loops so predict taken
  - Forwards branches are often if so predict not taken

- Predictors average better than 95% accuracy
  - Most branches are already predictable.

- Annual branch predictor contests at top Computer Architecture conferences
  - [https://www.jilp.org/jwac-2-program/JWAC-2-program.htm](https://www.jilp.org/jwac-2-program/JWAC-2-program.htm)
  - Winner: 34.1 mispredictions per kilo-instruction (!)
Getting High Performance

- **Good compiler and flags**
- **Think about the constant factors in your machine!**
  - 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 (Covered later in course)
Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Example: Bubblesort
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals