Program Optimization

CENG331 - Computer Organization

Instructor:
Murat Manguoglu (Sections 1-2)
Fall 2019
Today

- Overview

- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls

- 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
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**
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];
}
```

```c
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];
}
```
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
  \[ 16\times x \quad \rightarrow \quad x \ll 4 \]
  - Utility 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];
}
```
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: \(i\cdot n\), \((i-1)\cdot n\), \((i+1)\cdot n\)

```assembly
leaq  1(%rsi), %rax  # i+1
leaq  -1(%rsi), %r8  # i-1
imulq %rcx, %rsi  # i\cdot n
imulq %rcx, %rax  # (i+1)\cdot n
imulq %rcx, %r8   # (i-1)\cdot n
addq %rdx, %rsi  # i\cdot n+j
addq %rdx, %rax  # (i+1)\cdot n+j
addq %rdx, %r8   # (i-1)\cdot n+j
```

### 1 multiplication: \(i\cdot n\)

```assembly
imulq %rcx, %rsi  # i\cdot n
addq %rdx, %rsi  # i\cdot n+j
movq %rsi, %rax  # i\cdot n+j
subq %rcx, %rax  # i\cdot n+j-n
leaq (%rsi,%rcx), %rcx  # i\cdot n+j+n
```
Procedure to Convert String to Lower Case

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');
}
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
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:
}
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

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

```c
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');
}
```
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 treats 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

- Code updates \( b[i] \) on every iteration
- Why couldn’t compiler optimize this away?
Memory Aliasing

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

```c
/* 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];
    }
}
```

```c
double A[9] =
{ 0,   1,   2,
  4,   8,  16},
32,  64, 128};


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

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

```c
/* 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;
   }
}
```

- 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
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
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Data Type for Vectors

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

```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;
}
```

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

Compute sum or product of vector elements
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**
- \( T = CPE \times n + \text{Overhead} \)
  - CPE is slope of line

![Graph showing cycles per element](image-url)
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;
    }
}

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Combine1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unoptimized</td>
<td>22.68</td>
<td>20.02</td>
<td>19.98</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
</tr>
</tbody>
</table>

Compute sum or product of vector elements
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></th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop
Modern CPU Design

**Instruction Control**
- Fetch Control
- Instruction Decode
- Instruction Cache

**Execution**
- Branch
- Arith
- Arith
- Arith
- Load
- Store
- Data Cache

**Functional Units**
- Operation Results
- Addr.
- Data
- Addr.

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

**Data Flow**
- Instructions
- Address
- Operations
Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)
Pipelined Functional Units

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><strong>Integer/Long Divide</strong></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><strong>Single/Double FP Divide</strong></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 1.27</td>
<td>Add 3.01</td>
</tr>
<tr>
<td>Combine4</td>
<td>Mult 3.01</td>
<td>Mult 3.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Combine4 = Serial Computation (OP = *)

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

- **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

- Others don’t improve. *Why?*
  - Still sequential dependency

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Mult</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.01</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>

\[ x = (x \text{ OP } d[i]) \text{ OP } d[i+1]; \]
Loop Unrolling with Reassociation (2x1a)

```c
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></td>
<td></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>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</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)
Reassociated Computation

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

Overall Performance
- N elements, D cycles latency/op
- $(N/2+1) \times D$ cycles:
  \[ CPE = \frac{D}{2} \]
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>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

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

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

- **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 \((N/2+1) \times D\) cycles:
  \[CPE = D/2\]
- 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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unrolling Factor L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
<td>2.51</td>
<td>2.51</td>
<td>2.51</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
<td>1.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0.84</td>
<td></td>
<td></td>
<td></td>
<td>0.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.52</td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Int +

Case

- Intel Haswell
- Integer addition
- Latency bound: 1.00. Throughput bound: 1.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.27</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

Accumulators
Achievable Performance

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

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

\[ \text{vaddsd} \ %\text{ymm0}, \ %\text{ymm1}, \ %\text{ymm1} \]

- SIMD Operations: Double Precision

\[ \text{vaddpd} \ %\text{ymm0}, \ %\text{ymm1}, \ %\text{ymm1} \]
Using Vector Instructions

<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</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>

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

- Instruction Cache
- Fetch Control
- Instruction Decode
- Retirement Unit
  - Register File

Operations
- Address
- Instructions
- Prediction OK?
- Register Updates

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

Operation Results
- Addr.
- Data

Data Cache
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

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

...  
404685:  repz retq
```

Branch Not-Taken

Branch Taken
Branch Prediction

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

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

...  
404685:  repz retq
```
Branch Prediction Through Loop

Assume vector length = 100

Predict Taken (OK)
i = 99

Predict Taken (Oops)
i = 100

Read invalid location

Executed

Fetched
Branch Misprediction Invalidation

Assume
vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Invalidate
Branch Misprediction Recovery

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

- **Good compiler and flags**
- **Don’t do anything stupid**
  - 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)