Introduction to Computer Systems
15-213/18-243, spring 2009
10th Lecture, Oct. 1st

Instructors:
Roger B. Dannenberg and Greg Ganger

Example Matrix Multiplication
Matrix-Matrix Multiplication (MVM) on 2 x Core 2 Duo 3 GHz
Gflops/4 (giga-floating point operations per second)

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

Harsh Reality
- There’s more to runtime performance than asymptotic complexity
- One can easily lose 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)

Today
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism

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

- 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

```c
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

Example: Data Type for Vectors

```c
typedef struct{
    int len;
    double *data;
} vec;
```

Example

```
/* 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

Today

- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
Example: Summing Vector Elements

```c
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *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

```c
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;
    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}
```

Today

- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism

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 precomputation

```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];
}
```

Removing Procedure Calls

- Procedure calls can be very expensive
- Bound checking can be very expensive
- Abstract data types can easily lead to inefficiencies
  - Usually avoided for in superfast numerical library functions
- Watch your innermost loop!
- Get a feel for overhead versus actual computation being performed

Compiler-Generated Code Motion
Today
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism

Strength Reduction
- Replace costly operation with simpler one
  - Example: Shift/add instead of multiply or divide
    - \( 16 \times x \rightarrow x \ll 4 \)
    - Benefits are machine dependent
    - Depends on cost of multiply or divide instruction
    - On Pentium IV, integer multiply requires 10 CPU cycles
  - Example: Recognize sequence of products

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

Optimization Blocker #1: Procedure Calls
- Procedure to convert string to lower case
  ```c
  void lower(char *s)
  {
    int 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
Performance

- Time quadruples when double string length
- Quadratic performance

CPU Seconds

<table>
<thead>
<tr>
<th>String Length</th>
<th>256</th>
<th>512</th>
<th>2k</th>
<th>4k</th>
<th>8k</th>
<th>16k</th>
<th>32k</th>
<th>64k</th>
<th>128k</th>
<th>256k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>0.1</td>
<td>0.01</td>
<td>0.001</td>
<td>0.0001</td>
<td>0.00001</td>
<td>0.000001</td>
<td>0.0000001</td>
<td>0.00000001</td>
</tr>
</tbody>
</table>

Why is That?

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

```
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');
}
```

Improving Performance

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

```
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');
}
```

Performance

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

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

Optimization Blocker: Procedure Calls

- Why couldn’t compiler move a 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 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(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        lencnt += length;
    }
    return length;
}
```

Today

- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing
  - Out of order processing: Instruction level parallelism
Optimization Blocker: Memory Aliasing

Reason
- If memory is accessed, compiler assumes the possibility of side effects
- Example:

```c
/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows1(double *a, double *b, long n) {
    for (i = 0; i < n; i++)
        b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
}
```

Value of B:
- init: [4, 6, 14]
- i = 0: [9, 6, 14]
- i = 1: [14, 22, 14]
- i = 2: [17, 22, 21]

Unaliased Version When Aliasing Happens

```c
/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows2(double *a, double *b, long n) {
    double val = 0;
    for (j = 0; j < n; j++)
        val += a[i*n + j];
    b[i] = val;
}
```

Value of B:
- init: [4, 6, 14]
- i = 0: [9, 6, 14]
- i = 1: [14, 22, 14]
- i = 2: [17, 22, 21]

More Difficult Example
- Matrix multiplication: \( C = A^T \times B + C \)

```c
// Matrix multiplication: C = A^T * B + C
m = double (*) malloc(nrows(double), ncols(double));
/* Multiply m x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    for (i = 0; i < n; i++)
        c[i*n] = a[i*n] * b[i*n] + c[i*n];
}
```

Which array elements are reused?
- All of them! But how to take advantage?

Removing Aliasing

Reason
- If memory is accessed, compiler assumes the possibility of side effects
- Example:

```c
/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows1(double *a, double *b, long n) {
    for (i = 0; i < n; i++)
        b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
}
```

Value of B:
- init: [4, 6, 14]
- i = 0: [9, 6, 14]
- i = 1: [14, 22, 14]
- i = 2: [17, 22, 21]

Unaliased Version When Aliasing Happens

```c
/* Sums rows of n x n matrix a and stores in vector b */
void sum_rows2(double *a, double *b, long n) {
    double val = 0;
    for (j = 0; j < n; j++)
        val += a[i*n + j];
    b[i] = val;
}
```

Value of B:
- init: [4, 6, 14]
- i = 0: [9, 6, 14]
- i = 1: [14, 22, 14]
- i = 2: [17, 22, 21]

More Difficult Example
- Matrix multiplication: \( C = A^T \times B + C \)

```c
// Matrix multiplication: C = A^T * B + C
m = double (*) malloc(nrows(double), ncols(double));
/* Multiply m x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    for (i = 0; i < n; i++)
        c[i*n] = a[i*n] * b[i*n] + c[i*n];
}
```

Which array elements are reused?
- All of them! But how to take advantage?
Step 1: Blocking (Here: 2 x 2)
- Blocking, also called tiling = partial unrolling + loop exchange
  - Assumes associativity (= compiler will never do it)

### Example: Compute Factorials

#### Machines
- Intel Pentium 4 Nocona, 3.2 GHz
- Fish Machines
- Intel Core 2, 2.7 GHz

#### Compiler Versions
- GCC 3.4.2 (current on Fish machines)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3a</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact_u3b</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3c</td>
<td>15.5</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Something changed from Pentium 4 to Core: Details later
Modern CPU Design

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 CPUs since about 1998 are superscalar.

Intel: since Pentium Pro

Pentium 4 Nocona CPU

Multiple instructions can execute in parallel
1 load, with address computation
2 simple integer (one may be branch)
1 complex integer (multiply/divide)
1 FP/SSE3 unit
1 FP move (does all conversions)

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>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

Latency versus Throughput

Last slide: latency cycles/issue

<table>
<thead>
<tr>
<th>Integer Multiply</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>1 cycle</td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>1 cycle</td>
<td></td>
</tr>
<tr>
<td>Step 10</td>
<td>1 cycle</td>
<td></td>
</tr>
</tbody>
</table>

Consequence:

How fast can 10 independent int multis be executed?
\[ t_1 = t_2 \times t_3; \quad t_4 = t_5 \times t_6; \ldots \]

How fast can 10 sequentially dependent int multis be executed?
\[ t_1 = t_2 \times t_3; \quad t_4 = t_5 \times t_6; \quad t_6 = t_7 \times t_4; \ldots \]

Major problem for fast execution: Keep pipelines filled

Hard Bounds

Latency and throughput of instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

How many cycles at least if

Function requires n int multis?
Function requires n float adds?
Function requires n float ops (adds and multis)?

Performance in Numerical Computing

Numerical computing = computing dominated by floating point operations

Example: Matrix multiplication

Performance measure: Floating point operations per second (flop/s)

- Counting only floating point adds and multiplies
- Higher is better
- Like inverse runtime

Theoretical scalar (no vector SSE) peak performance on fish machines?

- 3.2 GFlop/s = 3200 MFlop/s. Why?
Nocona vs. Core 2

- **Nocona (3.2 GHz) (Saltwater fish machines)**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Load / Store</th>
<th>Integer Multiply</th>
<th>Integer/Long Divide</th>
<th>Single/Double FP Multiply</th>
<th>Single/Double FP Add</th>
<th>Single/Double FP Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>5</td>
<td>10</td>
<td>18/106</td>
<td>7</td>
<td>5</td>
<td>32/46</td>
</tr>
<tr>
<td>Cycles/Issue</td>
<td>1</td>
<td>1</td>
<td>3/106</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Core 2 (2.7 GHz) (Recent Intel microprocessors)**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Load / Store</th>
<th>Integer Multiply</th>
<th>Integer/Long Divide</th>
<th>Single/Double FP Multiply</th>
<th>Single/Double FP Add</th>
<th>Single/Double FP Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>5</td>
<td>3</td>
<td>18/50</td>
<td>4/5</td>
<td>3</td>
<td>18/32</td>
</tr>
<tr>
<td>Cycles/Issue</td>
<td>1</td>
<td>1</td>
<td>36/106</td>
<td>36/106</td>
<td>1</td>
<td>32/46</td>
</tr>
</tbody>
</table>

**Instruction Control**

- Grabs instruction bytes from memory
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- Translates instructions into micro-operations (for CISC style CPUs)
  - Micro-op = primitive step required to perform instruction
  - Typical instruction requires 1–3 operations
- Converts register references into tags
  - Abstract identifier linking destination of one operation with sources of later operations

**Translating into Micro-Operations**

- **Goal**: Each operation utilizes single functional unit
  - Requires: Load, integer arithmetic, store

- **Example**:
  ```
  imulq trax, 8(%rbx, krdx, 4)
  ```

**Traditional View of Instruction Execution**

- **Imperative View**
  - Registers are fixed storage locations
  - Individual instructions read & write them
  - Instructions must be executed in specified sequence to guarantee proper program behavior

**Dataflow View of Instruction Execution**

- **Functional View**
  - View each write as creating new instance of value
  - Operations can be performed as soon as operands available
  - No need to execute in original sequence

**Example Computation**

```c
void combine(void *vec_ptr v, data_t *dest)
{  
  int i;
  int length = vec_length(v);
  data_t t = GET_VEC_START(v);
  for (i = 0; i < length; ++i)
  {
    t = t OP d[i];
  }
  *dest = t;
}
```

- **Data Types**
  - Use different declarations for `data_t`
    - `int`
    - `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 (gives hard lower bound)
- T = CPE*n + Overhead

x86-64 Compilation of Combine4
```c
void combine4(vec_ptr v, data_t *dest) {
  int length = vec_length(v);
  data_t x = get_vec_start(v);
  for (i = 0; i < length; i++)
    dest = x;
}
```

Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Method</th>
<th>Int [add/mult]</th>
<th>Float [add/mult]</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer sum
- Others don’t improve. Why?
  - Still sequential dependency
    ```c
    x = (x OP d[i]) OP d[i+1];
    ```
Effect of Reassociation

**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)

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-va</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Reassociated Computation

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

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be \((N/2 + 1)^2 \times D\) cycles: \(CPE = D/2\)
  - Measured CPE slightly worse for FP

\[ x = x \text{ OP } (d[i] \text{ OP } d[i+1]); \]

Loop Unrolling with Separate Accumulators

```c
void unroll2a_combine(vec_ptr v, data_t *dest) {
  int length = vec_length(v);
  int limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x0 = IDENT;
  data_t x1 = IDENT;
  int 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

- **Almost exact 2x speedup (over unroll2) for Int *, FP +, FP ***

- Breaks sequential dependency in a “cleaner,” more obvious way

\[ x0 = x0 \text{ OP } d[i]; \]
\[ x1 = x1 \text{ OP } d[i+1]; \]

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-va</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
## Unrolling & Accumulating: Intel FP *

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - FP Multiplication
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>1</td>
<td>7.00 7.00 7.01 7.00</td>
</tr>
<tr>
<td>2</td>
<td>3.50 3.50 3.50</td>
</tr>
<tr>
<td>3</td>
<td>2.34</td>
</tr>
<tr>
<td>4</td>
<td>2.01 2.00</td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accumulators</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

## Unrolling & Accumulating: Intel FP +

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - FP Addition
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>1</td>
<td>5.00 5.00 5.02 5.00</td>
</tr>
<tr>
<td>2</td>
<td>2.50 2.51 2.51</td>
</tr>
<tr>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>4</td>
<td>2.01 2.00</td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

## Unrolling & Accumulating: Intel Int *

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - Integer Multiplication
  - Theoretical Limit: 1.00

<table>
<thead>
<tr>
<th>Int *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>1</td>
<td>10.00 10.00 10.01 10.01</td>
</tr>
<tr>
<td>2</td>
<td>5.00 5.01 5.00</td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td>2.50 2.51</td>
</tr>
<tr>
<td>6</td>
<td>1.67</td>
</tr>
<tr>
<td>8</td>
<td>1.25</td>
</tr>
<tr>
<td>10</td>
<td>1.09</td>
</tr>
<tr>
<td>12</td>
<td>1.14</td>
</tr>
</tbody>
</table>

## Unrolling & Accumulating: Intel Int +

- **Case**
  - Intel Nocona (Saltwater fish machines)
  - Integer addition
  - Theoretical Limit: 1.00 (unrolling enough)

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>1</td>
<td>2.20 1.50 1.10 1.03</td>
</tr>
<tr>
<td>2</td>
<td>1.50 1.10 1.03</td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
</tr>
<tr>
<td>4</td>
<td>1.09 1.03</td>
</tr>
<tr>
<td>6</td>
<td>1.01</td>
</tr>
<tr>
<td>8</td>
<td>1.04</td>
</tr>
<tr>
<td>10</td>
<td>1.11</td>
</tr>
<tr>
<td>12</td>
<td>1.11</td>
</tr>
</tbody>
</table>

## FP *: Nocona versus Core 2

- **Machines**
  - Intel Nocona
  - 3.2 GHz
  - Intel Core 2
  - 2.7 GHz

- **Performance**
  - Core 2 lower latency & fully pipelined (1 cycle/issue)

## Nocona vs. Core 2 Int *

### Performance
- Newer version of GCC does reassociation
- Why for int’s and not for float’s?
Intel vs. AMD
FP

- Machines
  - Intel Nocona
    - 3.2 GHz
  - AMD Opteron
    - 2.0 GHz

- Performance
  - AMD lower latency & better pipelining
  - But slower clock rate

Int
vs.
AMD
Int
+

- Performance
  - AMD gets below 1.0
  - Even just with unrolling

- Explanation
  - Both Intel & AMD can "double pump" integer units
  - Only AMD can load two elements / cycle

Can We Go Faster?

- Yes, SSE!
  - But not in this class

Summary

- Optimization comes from many directions:
  - Algorithm design: huge potential
  - Optimizing compilers: effective but conservative
  - Manual tuning: many techniques
  - Parallel computation: we’ll talk about this later
- Understanding processors, memory, and compilers