Program Optimization II
Achievable Performance so far

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

42X improvement over original, unoptimized code via unrolling and multiple accumulators

Limited only by throughput of functional units

- Each FP multiply unit can issue 1 operation per cycle
- 2 FP multiplication units in Intel Core Haswell CPU (2013)

2 func. units for FP *
2 func. units for load
Modern video resolutions

**8K resolution**

- 7680x4320 at 60 frames per second
- Operations on pixel data requires massive parallelism
- Is our best achievable performance adequate?
Motivation for SIMD

Multimedia, graphics, scientific, and security applications

- Require a single operation across large amounts of data (both integer and floating point)
  - Frame differencing for video encoding
  - Image Fade-in/Fade-out
  - Sprite overlay in game
  - Matrix computations
  - Encryption/decryption

- Algorithm characteristics
  - Access data in a regular pattern
  - Operate on short data types (8-bit, 16-bit, 32-bit)
  - Have an operating paradigm that has data streaming through fixed processing stages
    » Data-flow operation
Natural fit for SIMD instructions

Single Instruction, Multiple Data

- Also known as vector instructions
- Before SIMD
  - One instruction per data location
- With SIMD
  - One instruction over multiple sequential data locations
  - Execution units must support “wide” parallel execution

Examples in many processors

- Intel x86
  - MMX, SSE, AVX
- AMD
  - 3DNow!
Example

\[ R = R + XR \times 1.08327 \]
\[ G = G + XG \times 1.89234 \]
\[ B = B + XB \times 1.29835 \]
Example

for (i=0; i<64; i+=1)
   A[i+0] = A[i+0] + B[i+0]

for (i=0; i<64; i+=4){
   A[i+0] = A[i+0] + B[i+0]
}

for (i=0; i<64; i+=4)
General idea

SIMD (single-instruction, multiple data) vector instructions

- Wide registers
- Parallel operation on vectors of integers or floats
- Example:

```
  +  
```

“4-way”

```
  x  
```
# Intel Architectures and SIMD

## Processors

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1972</td>
<td>8086</td>
</tr>
<tr>
<td>1982</td>
<td>286</td>
</tr>
<tr>
<td>1982</td>
<td>386</td>
</tr>
<tr>
<td>1984</td>
<td>486</td>
</tr>
<tr>
<td>1990</td>
<td>Pentium</td>
</tr>
<tr>
<td>1993</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>1999</td>
<td>Pentium III (1999)</td>
</tr>
<tr>
<td>2007</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>2007</td>
<td>Core 2 Duo (2007)</td>
</tr>
<tr>
<td>2011</td>
<td>Sandy Bridge (2011)</td>
</tr>
<tr>
<td>2013</td>
<td>Haswell (2013)</td>
</tr>
<tr>
<td>2016</td>
<td>Knights Landing (2016)</td>
</tr>
</tbody>
</table>

## Architectures

- **x86-16**
- **x86-32**
- **x86-64 / em64t**

### MultiMedia eXtensions (MMX)
- SSE
- SSE2
- SSE3

### Streaming SIMD Extensions (SSE)
- SSE4
- AVX
- AVX2
- AVX-512

### Advanced Vector eXtensions (AVX)

---

- 9 -
MMX (MultiMedia eXtensions)

Use FPU registers for SIMD execution of integer ops

- 64-bit registers
- Alias FPU registers (st0-st7) as MM0-MM7
- Did not use new registers to avoid adding CPU state (context switching)
- Can’t use FPU and MMX at the same time

8 byte additions (PADDB) +
4 short or word additions (PADDW) +
2 int or dword additions (PADDD) +
SSE (Streaming SIMD Extensions)

Larger, independent registers

- 128-bit data registers separate from FPU
  - 8 new hardware registers (XMM0-XMM7)
  - New status register for flags (MXCSR)
- Vectored floating point operations
- Streaming support (prefetching and cache control for data)
- Permutation operations (shuffling, interleaving)
- Horizontal operations (min, max)
- Video encoding accelerators (sum of absolute differences)
- Graphics building blocks (dot product)

All x86-64 CPUs support SSE
**SSE3 registers**

**Used for all floating point operations**
- Floating point parameter passing (all caller saved)
- Floating point return value (%xmm0)

**Used for vectored integer and floating point operations**

*128 bit*

<table>
<thead>
<tr>
<th>%xmm0</th>
<th>Argument #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>%xmm1</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%xmm2</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%xmm3</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%xmm4</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%xmm5</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%xmm6</td>
<td>Argument #7</td>
</tr>
<tr>
<td>%xmm7</td>
<td>Argument #8</td>
</tr>
<tr>
<td>%xmm8</td>
<td></td>
</tr>
<tr>
<td>%xmm9</td>
<td></td>
</tr>
<tr>
<td>%xmm10</td>
<td></td>
</tr>
<tr>
<td>%xmm11</td>
<td></td>
</tr>
<tr>
<td>%xmm12</td>
<td></td>
</tr>
<tr>
<td>%xmm13</td>
<td></td>
</tr>
<tr>
<td>%xmm14</td>
<td></td>
</tr>
<tr>
<td>%xmm15</td>
<td></td>
</tr>
</tbody>
</table>
# SSE3 Basic Instructions

## Moves

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>movss</td>
<td>movsd</td>
<td>D ← S</td>
</tr>
</tbody>
</table>

- **Move a single value to/from low part of register**
  - Move a single, single-precision float
  - Move a single, double-precision float
## SSE3 Basic Scalar Instructions

### Arithmetic

*Compute inner product of two vectors*

- **addss** = add, scalar, single-precision float
- **adds** = add, scalar, double-precision float

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>addss</td>
<td>addsd</td>
<td>D ← D + S</td>
</tr>
<tr>
<td>subss</td>
<td>subsd</td>
<td>D ← D − S</td>
</tr>
<tr>
<td>mulss</td>
<td>mulsd</td>
<td>D ← D x S</td>
</tr>
<tr>
<td>divss</td>
<td>divsd</td>
<td>D ← D / S</td>
</tr>
<tr>
<td>maxss</td>
<td>maxsd</td>
<td>D ← max(D,S)</td>
</tr>
<tr>
<td>minss</td>
<td>minsd</td>
<td>D ← min(D,S)</td>
</tr>
<tr>
<td>sqrtss</td>
<td>sqrtssd</td>
<td>D ← sqrt(S)</td>
</tr>
</tbody>
</table>
**FP Code Example**

**Compute inner product of two vectors**

- Single precision arithmetic
- Uses SSE3 instructions

```c
float ipf (float x[],
          float y[],
          int n) {
  int i;
  float result = 0.0;
  for (i = 0; i < n; i++)
    result += x[i]*y[i];
  return result;
}
```

**ipf:**

- `xorps %xmm1, %xmm1`  # result = 0.0
- `xorl %ecx, %ecx`  # i = 0
- `jmp .L8`  # goto middle

**.L10:**  # loop:

- `movslq %ecx,%rax`  # icpy = i
- `incl %ecx`  # i++
- `movss (%rsi,%rax,4), %xmm0`  # t = y[icpy]
- `mulss (%rdi,%rax,4), %xmm0`  # t *= x[icpy]
- `addss %xmm0, %xmm1`  # result += t

**.L8:**  # middle:

- `cmpl %edx, %ecx`  # i:n
- `jl .L10`  # if < goto loop
- `movaps %xmm1, %xmm0`  # return result
- `ret`

**FP Code Example**

Compute inner product of two vectors

- Single precision arithmetic
- Uses SSE3 instructions
**SIMD support**

**SSE3 registers**

**Integer vectors:**
- 16-way byte
- 8-way short
- 4-way int

**Floating point vectors:**
- 4-way single (float)
- 2-way double

**Floating point scalars:**
- Single
- Double
SSE3 Basic Instructions revisited

**Moves**

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>movss</td>
<td>movsd</td>
<td>D ← S</td>
</tr>
</tbody>
</table>

- Move a single value to/from low part of register
  - Move a single, single-precision float
  - Move a single, double-precision float

- Packed versions to move a vector (128-bits) to/from register
  - movaps
  - movapd
  - Move aligned vector of single-precision floats
  - Move aligned vector of double-precision floats
SSE3 Basic Instructions revisited

- addss  |  packed (vector)
    |  single slot (scalar)
  addsd  |  single precision  
  addpd  |  double precision

Basic Instructions revisited

- addss
- addsd
- addps
- addpd
SSE3 Basic Instructions revisited

Single precision **scalar add**: `addss %xmm0 %xmm1`

Single precision **4-way vector add**: `addps %xmm0 %xmm1`
SIMD in C via libraries and intrinsics

Call into libraries compiled with SSE support

Rely on compiler intrinsics

- Macros embed in-line assembly into program

```c
// Add two 128-bit vectors containing 4 float
// Microsoft-specific compiler intrinsic function
__m128 _mm_add_ps(__m128 a , __m128 b );
__m128 a, b, c;
// intrinsic function
c = __mm_add_ps(a, b);
```

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
```

```
+ + + +
```

```
|   | 3 | 6 | 9 | 12 |
|---|---|---|---|
| c |   |   |   |   |
```
SIMD in C via gcc

gcc beyond 4.1.1 supports auto-vectorization

- Use –O3 or -ftree-vectorize

Issues

- Memory aliasing
  - Compiler can’t auto-vectorize unless vectors are known not to overlap

- Alignment
  - Vector loads and stores need data aligned on 16-byte boundaries
  - Can not use movaps otherwise
Scalar FP addition example

```c
void add_scalar(double *a, double *b, double *c)
{
    c[0] = a[0] + b[0];
    c[1] = a[1] + b[1];
}
int main() {
    double f[4], g[4], h[4];
    ...  
    add_scalar(f, g, h);
}
```

```
add_scalar:
    movsd   (%rdi), %xmm0
    addsd   (%rsi), %xmm0
    movsd   %xmm0, (%rdx)
    movsd   8(%rdi), %xmm0
    addsd   8(%rsi), %xmm0
    movsd   %xmm0, 8(%rdx)
    movsd   16(%rdi), %xmm0
    addsd   16(%rsi), %xmm0
    movsd   %xmm0, 16(%rdx)
    movsd   24(%rdi), %xmm0
    addsd   24(%rsi), %xmm0
    movsd   %xmm0, 24(%rdx)
    ret
```
SSE FP addition example

“restrict” keyword to denote non-overlapping addresses (Must use C99)

__builtin_assume_aligned to denote memory alignment on 16 byte boundaries

```c
void add_vector(double * restrict a, double * restrict b, double * restrict c) {
    __builtin_assume_aligned(a, 16);
    __builtin_assume_aligned(b, 16);
    __builtin_assume_aligned(c, 16);
    c[0] = a[0] + b[0];
    c[1] = a[1] + b[1];
}
int main() {
    double f[4], g[4], h[4];
    ...
    add_vector(f, g, h);
}
```

Compiled with:
```
gcc -std=gnu99 -O3 -march=corei7
```

add_vector:
```
movapd 16(%rdi), %xmm0          ; a[2:3]
movapd (%rdi), %xmm1            ; a[0:1]
addpd 16(%rsi), %xmm0           ; += b[2:3]
addpd (%rsi), %xmm1             ; += b[0:1]
movapd %xmm0, 16(%rdx)          ; store c[2:3]
movapd %xmm1, (%rdx)            ; store c[0:1]
ret
```
AVX (Advanced Vector Extensions)

256-bit registers

- 16 registers (YMM0-YMM15)
  - 8 32-bit integers or 8 32-bit single-precision floats
    
    ![Diagram of 16 registers with 8 32-bit integers or 8 32-bit single-precision floats]

  - 4 64-bit integers or 4 64-bit double-precision floats
    
    ![Diagram of 4 registers with 4 64-bit integers or 4 64-bit double-precision floats]

- “Gather support” to load data from non-contiguous memory
- 3-operand FMA operations (fused multiply-add operations) at full precision \((a+b\times c)\)
- Dot products, matrix multiply, Horner’s rule
AVX (Advanced Vector Extensions)

VEX (vector extensions) instruction coding scheme to accommodate new instructions using previous opcodes

- Accommodate new instructions using previous opcodes

- packed (128-bits)
  - addps
  - vaddps

- packed (256-bits)
  - addps
  - vaddps

- single precision
  - addps
  - vaddps

- double precision
  - addps
  - vaddps
SSE vs. AVX FP addition example

SSE: 4 doubles need 2 loads
AVX: 4 doubles load as single vector

```c
void add_vector(double * restrict a, double * restrict b, double * restrict c) {
    __builtin_assume_aligned(a,16);
    __builtin_assume_aligned(b,16);
    __builtin_assume_aligned(c,16);
    c[0] = a[0] + b[0];
    c[1] = a[1] + b[1];
}
```

Compiled with
“-march=corei7”

add_vector:
```
    movapd 16(%rdi), %xmm0 ; a[2:3]
    movapd (%rdi), %xmm1 ; a[0:1]
    addpd 16(%rsi), %xmm0 ; += b[2:3]
    addpd (%rsi), %xmm1 ; += b[0:1]
    movapd %xmm0, 16(%rdx) ; store c[2:3]
    movapd %xmm1, (%rdx) ; store c[0:1]
    ret
```

AVX: 4 doubles load as single vector

```c
void add_vector(double * restrict a, double * restrict b, double * restrict c) {
    __builtin_assume_aligned(a,16);
    __builtin_assume_aligned(b,16);
    __builtin_assume_aligned(c,16);
    vmovupd (%rdi), %ymm1 ; a[0:3]
    vmovupd (%rsi), %ymm0 ; b[0:3]
    vaddpd %ymm0, %ymm1, %ymm0 ; add
    vmovupd %ymm0, (%rdx) ; store c[0:3]
    ret
}
```

Compiled with
“-march=core-avx2”

add_vector:
```
    vmovupd (%rdi), %ymm1 ; a[0:3]
    vmovupd (%rsi), %ymm0 ; b[0:3]
    vaddpd %ymm0, %ymm1, %ymm0 ; add
    vmovupd %ymm0, (%rdx) ; store c[0:3]
    ret
```
Using Vector Instructions

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar Best</td>
<td></td>
<td>0.54</td>
<td>1.01</td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>Vector Best</td>
<td></td>
<td>0.06</td>
<td>0.24</td>
<td>0.25</td>
<td>0.16</td>
</tr>
<tr>
<td>Latency Bound</td>
<td></td>
<td>0.50</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td></td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td></td>
<td>0.06</td>
<td>0.12</td>
<td>0.25</td>
<td>0.12</td>
</tr>
</tbody>
</table>

With use of parallel operations on multiple data elements via AVX
AVX-512

512-bit registers

- 16 registers (ZMM0-ZMM15)
  - Capacity for a vector of 8 doubles or 8 long
Detecting if it is supported

```assembly
mov    eax, 1

cpuid          ; supported since Pentium

test   edx, 008000000h ; 008000000h (bit 23) MMX
                ; 020000000h (bit 25) SSE
                ; 040000000h (bit 26) SSE2

jnz    HasMMX
```
Detecting if it is supported

```c
#include <stdio.h>
#include <string.h>

#define cpuid(func,ax,bx,cx,dx)
    __asm__ __volatile__ ("cpuid:\n" "=a" (ax), "=b" (bx), "=c" (cx), "=d" (dx) : "a" (func));

int main(int argc, char* argv[]) {
    int a, b, c, d, i;
    char x[13];
    int* q;
    for (i=0; i < 13; i++)  x[i]=0;
    q=(int *) x;
    /* 12 char string returned in 3 registers */
    cpuid(0,a,q[0],q[2],q[1]);
    printf("str: %s
", x);
    /* Bits returned in all 4 registers */
    cpuid(1,a,b,c,d);
    printf("a: %08x, b: %08x, c: %08x, d: %08x\n",a,b,c,d);
    printf(" bh * 8 = cache line size\n");
    printf(" bit 0 of c = SSE3 supported\n");
    printf(" bit 25 of c = AES supported\n");
    printf(" bit 0 of d = On-board FPU\n");
    printf(" bit 4 of d = Time-stamp counter\n");
    printf(" bit 26 of d = SSE2 supported\n");
    printf(" bit 25 of d = SSE supported\n");
    printf(" bit 23 of d = MMX supported\n");
}
```

http://thefengs.com/wuchang/courses/cs201/class/13/cpuid.c
Detecting if it is supported

mashimaro <~> 10:43AM % cat /proc/cpuinfo
processor : 7
vendor_id : GenuineIntel
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
microcode : 0x17
cpu MHz : 800.000
cache size : 8192 KB
siblings : 8
core id : 3
cpu cores : 4
cpuid level : 13
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtsscp lm constant_tsc arch_perfmon pebs bts rep_good nop1 xtopology
nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx
est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm ida arat epb xsaveopt
pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmid
hle avx2 smep bmi2 erms invpcid rtm
bogomips : 6784.28
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
Next steps

Still not enough parallelism for you?

- GP-GPU
- Deep learning

Google Relies on GPUs for Deep Learning

By: RAID | September 9, 2015

Were you at the GPU Technology Conference in April? One of the featured presentations was called Large-Scale Deep Learning for Building Intelligent Computer Systems, given by Jeff Dean, a Senior Fellow at Google. Since interest has been steadily increasing in our GPU optimized servers and GPU writ large, we are featuring posts this month about how GPUs are being used. Below is a synopsis of Mr. Dean’s presentation.
AES

AES-NI announced 2008

- Added to Intel Westmere processors and beyond (2010)
- Separate from MMX/SSE/AVX
- AESENCE/AESDEC performs one round of an AES encryption/decryption flow
  - One single byte substitution step, one row-wise permutation step, one column-wise mixing step, addition of the round key (order depends on whether one is encrypting or decrypting)
  - Speed up from 28 cycles per byte to 3.5 cycles per byte
  - 10 rounds per block for 128-bit keys, 12 rounds per block for 192-bit keys, 14 rounds per block for 256-bit keys
  - Software support from security vendors widespread

http://software.intel.com/file/24917
Profiling code

Measuring performance

- **Time code execution**
  - Most modern machines have built in cycle counters
  - `rdtsc` from prior lecture
  - Using them to get reliable measurements is tricky

- **Profile procedure calling frequencies**
  - Unix tool `gprof`
Profiling Code Example

Task

- Count word frequencies in text document
- Produce sorted list of words from most frequent to least

Steps

- Convert strings to lowercase
- Apply hash function
- Read words and insert into hash table
  - Mostly list operations
  - Maintain counter for each unique word
- Sort results

Data Set

- Collected works of Shakespeare
- 946,596 total words, 26,596 unique
- Initial implementation: 9.2 seconds

Shakespeare’s most frequent words

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>29,801</td>
</tr>
<tr>
<td>and</td>
<td>27,529</td>
</tr>
<tr>
<td>I</td>
<td>21,029</td>
</tr>
<tr>
<td>to</td>
<td>20,957</td>
</tr>
<tr>
<td>of</td>
<td>18,514</td>
</tr>
<tr>
<td>a</td>
<td>15,370</td>
</tr>
<tr>
<td>you</td>
<td>14010</td>
</tr>
<tr>
<td>my</td>
<td>12,936</td>
</tr>
<tr>
<td>in</td>
<td>11,722</td>
</tr>
<tr>
<td>that</td>
<td>11,519</td>
</tr>
</tbody>
</table>
Augments program with timing functions

- Computes approximate time spent in each function

**Method**
- Periodically (~ every 10ms) interrupt program
- Determine what function is currently executing
- Increment its timer by interval (e.g., 10ms)

**Keep counter per function tracking number of times called**

gcc -O2 -pg prog.c -o prog

`./prog`
- Executes in normal fashion, but also generates file `gmon.out`

gprof prog
- Generates profile information based on `gmon.out`

http://thefengs.com/wuchang/courses/cs201/class/13

make run
Profiling Results

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.60</td>
<td>8.21</td>
<td>8.21</td>
<td>1</td>
<td>8210.00</td>
<td>8210.00</td>
<td>sort_words</td>
</tr>
<tr>
<td>5.80</td>
<td>8.76</td>
<td>0.55</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>lower1</td>
</tr>
<tr>
<td>4.75</td>
<td>9.21</td>
<td>0.45</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>find_ele_rec</td>
</tr>
<tr>
<td>1.27</td>
<td>9.33</td>
<td>0.12</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td>h_add</td>
</tr>
</tbody>
</table>

Call Statistics

- Number of calls and cumulative time for each function

Performance Limiter

- Using inefficient sorting algorithm
- Single call uses 87% of CPU time
Exam practice

Chapter 5 Problems

5.1 Memory aliasing
5.2 Performance ranges
5.4 Memory access and accumulators
5.7 Loop unrolling
5.8 Reassociation
5.9 Conditional move
5.10 Data dependencies
5.11 Reducing memory access
5.12 Reducing memory access
5.14 Unrolling
5.15 Unrolling with parallel accumulation
5.16 Unrolling with reassociation
Extra slides
AVX FMA example

Measuring performance improvement
- FMA (fused-multiple-add) instructions in AVX

```c
double* a, b, c;
c[0] = c[0] + a[0] * b[0];
```
AVX FMA example

```c
void fma_vector(double * restrict a, double * restrict b, double * restrict c) {
    int i;
    __builtin_assume_aligned(a,16);
    __builtin_assume_aligned(b,16);
    __builtin_assume_aligned(c,16);

    c[0] = c[0] + a[0] * b[0];
}
```

Compiled with "-march=corei7"

```
fma_vector:
  movapd 16(%rdi), %xmm0
  movapd (%rdi), %xmm1
  mulpd 16(%rsi), %xmm0
  mulpd (%rsi), %xmm1
  addpd 16(%rdx), %xmm0
  addpd (%rdx), %xmm1
  movapd %xmm0, 16(%rdx)
  movapd %xmm1, (%rdx)
  ret
```

Compiled with "-march=core-avx2"

```
fma_vector:
  vmovupd (%rdx), %ymm1
  vmovupd (%rdi), %ymm0
  vmovupd (%rsi), %ymm2
  vfmadd132pd %ymm2, %ymm1, %ymm0
  vmovupd %ymm0, (%rdx)
  ret
```

http://thefengs.com/wuchang/courses/cs201/class/13
Floating Point history

Floating Point Unit (X87 FPU)
- Hardware to add, multiply, and divide IEEE floating point numbers
- 8 80-bit registers organized as a stack (st0-st7)
- Operands pushed onto stack and operators can pop results off into memory

History
- 8086: first computer to implement IEEE FP
  - separate 8087 FPU (floating point unit)
  - drove the design of floating point instructions
- 486: merged FPU and Integer Unit onto one chip
FPU Data Register Stack

FPU register format (extended precision)

<table>
<thead>
<tr>
<th>79</th>
<th>78</th>
<th>64</th>
<th>63</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>exp</td>
<td>frac</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FPU registers
- 8 registers
- Logically forms shallow stack
- Top called \( \%st(0) \)
- When push too many, bottom values disappear

Stack operation similar to Reverse Polish Notation
- \( a \ b + = \) push \( a \), push \( b \), add (pop \( a \) & \( b \), add, push result)
Example calculation

\[ x = \frac{(a-b)}{(-b+c)} \]

- load c
- load b
- neg
- addp
- load b
- load a
- subp
- divp
- storep x
What About Branches?

Challenge

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

```assembly
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.

Executing

How to continue?
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

...
Branch Prediction

Idea

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

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

...  

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Branch Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029:</td>
<td>vmulsd (%rdx),%xmm0,%xmm0</td>
<td></td>
<td>i = 98</td>
</tr>
<tr>
<td>40102d:</td>
<td>add $0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031:</td>
<td>cmp %rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034:</td>
<td>jne 401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume vector length = 100

Predict Taken (OK)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Branch Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029:</td>
<td>vmulsd (%rdx),%xmm0,%xmm0</td>
<td></td>
<td>i = 99</td>
</tr>
<tr>
<td>40102d:</td>
<td>add $0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031:</td>
<td>cmp %rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034:</td>
<td>jne 401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predict Taken (Oops)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Branch Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029:</td>
<td>vmulsd (%rdx),%xmm0,%xmm0</td>
<td></td>
<td>i = 100</td>
</tr>
<tr>
<td>40102d:</td>
<td>add $0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031:</td>
<td>cmp %rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034:</td>
<td>jne 401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read invalid location

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
<th>Branch Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>401029:</td>
<td>vmulsd (%rdx),%xmm0,%xmm0</td>
<td></td>
<td>i = 101</td>
</tr>
<tr>
<td>40102d:</td>
<td>add $0x8,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401031:</td>
<td>cmp %rax,%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401034:</td>
<td>jne 401029</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Misprediction Invalidation

Assume
vector length = 100

Predict Taken (OK)

i = 98

Predict Taken (Oops)

i = 99

Invalidate

i = 100

i = 101
Branch Misprediction Recovery

\begin{verbatim}
401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp %rax,%rdx
401034: jne 401029
401036: jmp 401040
...
401040: vmovsd %xmm0,(%r12)
\end{verbatim}

Definitely not taken

Reload Pipeline

Performance Cost

- Misprediction on Pentium III wastes \(\sim14\) clock cycles
- That’s a lot of time on a high performance processor