Introduction to Lab 4

Andreas Sembrant <andreas.sembrant@it.uu.se>
Division of Computer Systems
Dept. of Information Technology
Uppsala University
2012-12-04

What is lab 4?
Vectors? Who needs vectors anyway?

The purpose of this assignment is to give insights into:
▶ How vector instructions can be used for floating point code
▶ How integer operations can be performed using vector instructions
▶ How memory alignment affects performance and correctness

x86 Terminology

Integers
- Byte 8 bits
- Word 16 bits
- DWord 32 bits
- QWord 64 bits

Floating Point
- Single 32 bits
- Double 64 bits
- Extended 80 bits (only available in the x87)

SSE

Registers
- 16 new 128-bit registers (8 registers in 32-bit mode)
- Registers can hold either FP or integer values
- Number of elements depends on element type
SSE
Compared to classical x86

Classical x86/x87
- Stack based FP math
- Uses extended 80-bit FP precision internally
- Some instructions have fixed operands
- Memory operations can generally be unaligned

SSE
- Register based FP math
- Uses standard 32-bit or 64-bit FP precision
- All registers are general purpose
- Memory operations must generally be aligned

New instructions
- Several new MOV instructions
  - Most of them can act as both loads and stores
- Behavior with respect to memory system:
  - Aligned Requires aligned memory operands
  - Unaligned Allows unaligned memory operands
  - Non-temporal Accesses optimized for streaming data
- Different versions depending on data type
  - Can be used to optimize data placement inside the CPU

New instructions
- Loads and Stores

New instructions
- All common arithmetic operations are available
  - Operate on individual elements
  - At least one version per data type (8 versions of add!)
- Binary logic operators are available
  - Operate on entire 128-bit registers
  - Different versions for integer and FP
- Vector specific instructions
  - Dot-products
  - Horizontal add
  - …
- Hordes esoteric instructions

New Instructions
Horizontal Add

HADDPS (Horizontal ADD Packed Single fp)

Input vectors
\[
\begin{align*}
a_0 & \quad a_1 \\
b_0 & \quad b_1 \\
\vdots
\end{align*}
\]

Output vector
\[
\begin{align*}
c_0 & \quad c_1 \\
c_2 & \quad c_3
\end{align*}
\]

- Can be used to efficiently summarize 4 vectors
New Instructions

Comparisons

PCMPGTW (Parallel CoMpare Greater Than Word)

Input vectors

\[ a_0 \ a_1 \ldots a_7 > b_0 \ b_1 \ldots b_7 \]

\[ c_i := a_i > b_i \ ? \ FFFF_{16} : 0000_{16} \]

Output vector

\[ c_0 \ c_1 \ldots c_7 \]

- Compares element-wise
- An element is binary all 1 if the predicate is true, 0 otherwise
- Can be used to generate bit masks

Detecting SSE

How it should be done

1. Can bit 21 in EFLAGS be toggled? ⇒ CPUID is present
2. Execute CPUID\_eax\_0. Check manufacturer and maximum CPUID function #.
3. Execute CPUID\_eax\_1. Check the following bits:
   - EDX:25 SSE
   - EDX:25 SSE2
   - ECX:0 SSE3
   - ECX:9 SSSE3
   - ECX:19 SSE4.1
   - ECX:20 SSE4.2
4. Check for optional instructions (use CPUID)

C-support

Basics

- Several different interfaces, no standard. Common approaches:
  - Assembler libraries
  - Inline assembler (no standard inline asm syntax)
  - GCC Intrinsics
  - ICC Intrinsics (supported by GCC)
- Intrinsic names for ICC are documented in Intel’s CPU manuals
- GCC’s native instructions are “documented” in the GCC manual

C-support

Headers

- Some header files include earlier versions headers from earlier SSE versions.
- GCC requires that SSE extensions are enabled through command line switches.
  Warning: This normally allows GCC to automatically generate code for the those extensions.

\[ \text{xmmintrin.h} \quad \text{SSE} \]
\[ \text{emmintrin.h} \quad \text{SSE2} \]
\[ \text{pmmintrin.h} \quad \text{SSE3} \]
\[ \text{tmmintrin.h} \quad \text{SSSE3} \]
\[ \text{smmintrin.h} \quad \text{SSE4.1} \]
\[ \text{nmmintrin.h} \quad \text{SSE4.2} \]
\[ \text{gmmintrin.h} \quad \text{AVX} \]
### C-support

**Instruction and Type Naming**

<table>
<thead>
<tr>
<th>suffix</th>
<th>Vector type</th>
<th>Element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>epi8</td>
<td>_m128i</td>
<td>int8_t</td>
</tr>
<tr>
<td>epi16</td>
<td>_m128i</td>
<td>int16_t</td>
</tr>
<tr>
<td>epi32</td>
<td>_m128i</td>
<td>int32_t</td>
</tr>
<tr>
<td>epi64</td>
<td>_m128i</td>
<td>int64_t</td>
</tr>
<tr>
<td>ps</td>
<td>_m128</td>
<td>float</td>
</tr>
<tr>
<td>pd</td>
<td>_m128d</td>
<td>double</td>
</tr>
</tbody>
</table>

### Example

**Loading and Storing**

Load/store example using *unaligned* accesses

```c
#include <pmmintrin.h>

static void
my_memcpy(char *dst, const char *src, size_t len)
{
    /* Assume that length is an even multiple of the vector size */
    assert((len & 0xF) == 0);
    for (int i = 0; i < len; i += 16) {
        _m128i v = _mm_loadu_si128((__m128i *) (src + i));
        _mm_storeu_si128((__m128i *) (dst + i), v);
    }
}
```

### A Small Vectorization Tutorial

1. Start with a simple serial version of your algorithm
2. Remove conditional control flow
3. Unroll loops
4. Vectorize!

```c
static int
count(const uint32_t *data, size_t len)
{
    int c = 0;
    for (int i = 0; i < len; i++)
        if (data[i] == 0)
            c++;
    return c;
}
```
1. Start with a simple serial version of your algorithm
2. Remove conditional control flow
3. Unroll loops
4. Vectorize!

```c
int c = 0;
assert(!(len & 0x3));
for (int i = 0; i < len; i += 4)
  c += ((data[i + 0] == 0) ? 1 : 0) + ((data[i + 1] == 0) ? 1 : 0)
      + ((data[i + 2] == 0) ? 1 : 0) + ((data[i + 3] == 0) ? 1 : 0);
return c;
```

```c
__m128i c = _mm_setzero_si128();
const __m128i one = _mm_set1_epi32(1);
const __m128i zero = _mm_setzero_si128();
for (int i = 0; i < len; i += 4) {
  __m128i v = _mm_loadu_si128((__m128i *) (data + i));
  const __m128i cond = _mm_cmpeq_epi32(v, zero);
  c = _mm_add_epi32(c, _mm_and_si128(cond, one));
}
return _mm_extract_epi32(c, 0) + _mm_extract_epi32(c, 1)
     + _mm_extract_epi32(c, 2) + _mm_extract_epi32(c, 3);
```

Common Error Sources

- Unaligned memory accesses
  - Causes a Segmentation fault
  - May be due to an unintentional memory operand
  - Can be hard to spot in memory debuggers

- Unsupported SSE instructions
  - Causes an Illegal instruction error
  - GCC may automatically emit SSE instructions if SSE has been enabled on the command line

Where to go from here

- Intel
  - C++ Compiler Manual
  - Optimization Reference Manual
  - Intel 64 and IA-32 Architectures Software Developer’s Manual (vol. 1 & 2)

- AMD
  - Software Optimization Guide for AMD Family 10h
  - AMD64 Architecture Programmer’s Manual (vol. 1 & 3)

- The GCC manual
Important dates

Groups:
- **Prep.** Room 1412, now–17:00
- A 2012-12-06, Room 1412, 08:15–12:00
- B 2012-12-06, Room 1412, 13:15–17:00
- C 2012-12-07, Room 1412, 13:15–17:00
- Deadline: **2012-12-14 15:14**

Summary

You will:
- Implement an algorithm to convert text to lower case
- Measure how memory alignment and memory access types affect performance
- Implement a simple matrix-vector multiplication
- Implement a simple matrix-matrix multiplication

**Bonus** Implement an optimized matrix-matrix multiplication
- Complete lab manual on the course homepage

\[\text{http://www.it.uu.se/edu/course/homepage/avdark/ht12}\]