Introduction to Lab 4

Andreas Sandberg <andreas.sandberg@it.uu.se>

Division of Computer Systems
Dept. of Information Technology
Uppsala University

2010-11-10

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

The Kalkyl Cluster
Specifications

Cluster specifications
- 348 Nodes interconnected with Infiniband
- 2784 CPU Cores
- 9404 GB RAM
- 113 TB disk

Node specifications
- Runs Scientific Linux (RedHat Enterprise Linux customized for scientific applications)
- 2x Quad-Core Intel Xeon 5520 (Nehalem based)
- At least 24 GB RAM

Logging in transferring files

Connecting with SSH
- Always connect to kalkyl.uppmax.uu.se
- ssh -Y username@kalkyl.uppmax.uu.se
  - -Y – Enables X-forwarding

Transferring files
- Transfer files using the scp command
- Use the same server as for normal SSH logins
- scp ./foo username@kalkyl.uppmax.uu.se:bar/
  - Transfers the file ./foo to the directory bar in your home directory on Uppmax

Introduction Uppmax SSE Summary

Introduction Uppmax SSE Summary

Introduction Uppmax SSE Summary

Introduction Uppmax SSE Summary
You won’t measure correct results in your experiments unless you allocate an exclusive node for your experiments.

- Use `salloc -p node -n 1 -t 15:00 --qos=short -A g2010003 CMD` - Runs CMD, or a shell if CMD is omitted
- `-p node -n 1`—Request 1 node
- `-t 15:00`—Expected runtime for the job
- `--qos=short`—Use the queue for short jobs
- `-A g2010003`—Use the course project for accounting

Jobs running longer than the requested runtime time will be terminated.

Uppmax provides optional software in modules that can be easily loaded an unloaded.

- `module load gcc`—Load the latest version of the GCC compiler
- `module unload gcc`—Unload the currently loaded GCC module
- `module list`—List loaded modules
- `module whatis`—List available modules

### 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

<table>
<thead>
<tr>
<th>Classical x86/x87</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack based FP math</td>
<td>Register based FP math</td>
</tr>
<tr>
<td>Uses extended 80-bit FP precision internally</td>
<td>Uses standard 32-bit or 64-bit FP precision</td>
</tr>
<tr>
<td>Some instructions have fixed operands</td>
<td>All registers are general purpose</td>
</tr>
<tr>
<td>Memory operations can generally be unaligned</td>
<td>Memory operations must generally be aligned</td>
</tr>
</tbody>
</table>

### 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

- Horizontal Add

HADDPS (Horizontal ADD Packed Single fp)

Input vectors:

```
| a0 | a1 | a2 | a3 |
+---+---+---+---+
| b0 | b1 | b2 | b3 |
```

Output vector:

```
| c0 | c1 | c2 | c3 |
+---+---+---+---+
```

- Can be used to efficiently summarize 4 vectors

**Summary**

- All common arithmetic operations are available
  - Operate 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**

**Comparisons**

**PCMPGTW (Parallel Comapre Greater Than Word)**

- **Input vectors**
  
  \[
  \begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
  \end{array} >
  \begin{array}{cccc}
  b_0 & b_1 & b_2 & b_3 \\
  \end{array}
  \]

- **Output vector**
  
  \[
  \begin{array}{cccc}
  c_0 & c_1 & c_2 & c_3 \\
  \end{array}
  \]

- \(c_i := a_i > b_i \iff \text{FFFF}_{16} : \text{0000}_{16}\)

- 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? \(\implies\) CPUID is present
2. Execute CPUID \(\text{eax}=0\). Check manufacturer and maximum CPUID function #.
3. Execute CPUID \(\text{eax}=1\). Check the following bits:
   - EDX: 25 - SSE
   - EDX: 26 - SSE2
   - EDX: 29 - SSE3
   - EDX: 19 - SSSE3
   - 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
  - Intrinsics (GCC native)
  - Intrinsics (ICC native, supported by GCC)
- Intrinsic names for ICC are documented in Intel’s CPU manuals
- GCC’s native instructions are “documented” in the GCC manual
Introduction

Uppmax

SSE

Summary

C-support

Headers

- xmmINTRIN.h
- emmintrin.h
- pmmintrin.h
- tmmintrin.h
- mmmintrin.h
- gmmintrin.h
- AVX

Choose the header file for the extension you are targeting
- Some header files include earlier versions
- GCC requires that SSE extensions are enabled through a command line switch

Instruction and Type Naming

_\text{mm}_<\text{op}>_<\text{suffix}>

<table>
<thead>
<tr>
<th>&lt;suffix&gt;</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 \textit{unaligned} accesses

#include <pmmintrin.h>

\texttt{static void mymemcpy(char *dst, const char *src, size_t len)}
{
  \texttt{// Assume that length is an even multiple of the}
  \texttt{// vector size} /\texttt{
  assert((len & 0xF) == 0);

  \texttt{for (int i = 0; i < len; i += 16) { \texttt{
  _mm128i v = _mm_loadu_si128((\texttt{__m128i}) (src + i));}

  \texttt{_mm_storeu_si128((\texttt{__m128i}) (dst + i)), v);}

  \texttt{}}}

A Small Vectorization Tutorial

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

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

```
int c = 0;
for (int i = 0; i < len; i++)
c += (data[i] == 0) ? 1 : 0;
return c;
```

```
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;
```

```
__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. 2010-11-10, Room 1549, now–12:00
  - A 2010-11-11, Room 1549, 08:15–12:00
  - B 2010-11-11, Room 1549, 13:15–17:00
  - C 2010-11-12, Room 1549, 08:15–12:00
- Deadline: See course homepage

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

And remember...

Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest “foo” someone someday shall type “supercallfragilisticexpialidocious”.  

1. http://www.it.uu.se/edu/course/homepage/avdark/ht10