Introduction to Lab 4

Andreas Sandberg <andreas.sandberg@it.uu.se>
Division of Computer Systems
Dept. of Information Technology
Uppsala University

2011-11-18

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

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
The Kalkyl Cluster

Submitting an interactive job

You won’t measure correct results in your experiments unless you allocate an entire node for your experiments.

- Use `salloc -p node -n 1 -t 15:00 --qos=short -A g2011132 CMD`
  - Runs CMD, or a shell if CMD is omitted
  - `-p node`—Request 1 node
  - `-t 15:00`—Expected runtime for the job
  - `--qos=short`—Use the queue for short jobs
  - `-A g2011132`—Use the course project for accounting
- Jobs running longer than the requested runtime time will be terminated

Loading additional software

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
- doubleword 32 bits
- quadword 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
Loads and Stores

- 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
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
\[ a_0 \quad a_1 \quad a_2 \quad a_3 \quad b_0 \quad b_1 \quad b_2 \quad b_3 \]

Output vector
\[ c_0 \quad c_1 \quad c_2 \quad c_3 \]

Can be used to efficiently summarize 4 vectors
New Instructions

Comparisons

PCMPGTW (Parallel CoMpare Greater Than Word)

Input vectors

\[
\begin{align*}
a_0 & \quad a_1 & \ldots & \quad a_7 \\
& > \\
\end{align*}
\begin{align*}
b_0 & \quad b_1 & \ldots & \quad b_7 \\
\end{align*}
\]

Output vector

\[
\begin{align*}
c_0 & \quad c_1 & \ldots & \quad c_7 \\
\end{align*}
\]

- 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?  \(\Rightarrow\) 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)

Detecting SSE

What we do

This slide is intentionally left blank

C-support

Basics

- Several different interfaces, no standard. Common approaches:
  - Assembler libraries
  - Inline assembler (no standard inline asm syntax)
  - GCC Intrinsic
  - ICC Intrinsic (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

- `xmmintrin.h` SSE
- `emmintrin.h` SSE2
- `pmmintrin.h` SSE3
- `tmmintrin.h` SSSE3
- `smmintrin.h` SSE4.1
- `mmmintrin.h` SSE4.2
- `nmmintrin.h` AVX

- 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 those extensions.

---

### Load/store example using unaligned accesses

```c
#include <pmmintrin.h>

static void
my_memcpy(char *dst, const char *src, size_t len) {
    __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;
}
```
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
int c = 0;
for (int i = 0; i < len; i++)
c += (data[i] == 0) ? 1 : 0;
return c;
```

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
Introduction Uppmax SSE Summary

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 1515, now–17:00
    - A 2011-11-21, Room 1412, 08:15–12:00
    - B 2011-11-21, Room 1412, 13:15–17:00
    - C 2011-11-22, Room 1412, 08:15–12:00
  - Deadline: 2011-11-28 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

And remember…

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

1. http://www.it.uu.se/edu/course/homepage/avdark/ht11