Lab 4 — SIMD

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

1 Introduction

The purpose of this lab is to give some experience in using SIMD instructions on x86. We will use a matrix-vector multiplication to illustrate how SIMD can be used for numerical algorithms and a simple algorithm to convert text into lower-case to demonstrate how SIMD can be used for integer code.

You will be using GCC in this lab. GCC supports two sets of intrinsics, or built-ins, for SIMD. One is native to GCC and the other one is defined by Intel for their C++ compiler. We will use the intrinsics defined by Intel since these much better documented.

Both Intel[1] and AMD[2] provide excellent optimization manuals that discuss the use of SIMD instructions and software optimizations. These are good sources for information if you are serious about optimizing your software, but they are not mandatory reading for this lab. You will, however, find them, and the instruction set references, useful as reference literature when using the different SSE. Another useful reference is the Intel C++ compiler manual[3], which documents the SSE intrinsics supported by ICC and GCC.

We will, for various practical reasons, use Uppmax throughout the lab. section 3 introduces the tools and commands you need to know to get started.

2 Introduction to SSE

The SSE extension to the x86 consists of a set of 128-bit vector registers and a large number of instructions to operate on them. The number of available registers depends on the mode of the processor, only 8 registers are available in 32-bit mode, while 16 registers are available in 64-bit mode.

The data type of the packed elements in the 128-bit vector is decided by the specific instruction. For example, there are separate addition instructions for adding vectors of single and double precision floating point numbers. Some operations that are normally independent of the operand types (integer or floating point), e.g. bit-wise operations, have separate instructions for different types for performance reasons.

When reading the manuals, it's important to keep in mind that the size of a *word* in the x86-world is not really the native word size, i.e. 32-bits or 64-bits. Instead, it's 16-bits, which was the word size of the original microprocessor which the entire x86-line descends from. Whenever the manual talks about a *word*, it's really 16-bits. A 64-bit

[1] http://www.intel.com/products/processor/manuals/

[2] http://developer.amd.com/documentation/guides/

[3] http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/index.htm

Header file	Extension name	Abbrev.
`xmmintrin.h`	Streaming SIMD Extensions	SSE
`emmintrin.h`	Streaming SIMD Extensions 2	SSE2
`pmmintrin.h`	Streaming SIMD Extensions 3	SSE3
`tmmintrin.h`	Supplemental Streaming SIMD Extensions 3	SSSE3
`smmintrin.h`	Streaming SIMD Extensions 4 (Vector math)	SSE4.1
`nmmintrin.h`	Streaming SIMD Extensions 4 (String processing)	SSE4.2

Table 1: Header files used for different SSE versions

Intel Name	Elements/Reg.	Element type	Vector type	Type
Bytes	16	`int8_t`	`__m128i`	`epi8`
Words	8	`int16_t`	`__m128i`	`epi16`
Doublewords	4	`int32_t`	`__m128i`	`epi32`
Quadwords	2	`int64_t`	`__m128i`	`epi64`
Single Precision Floats	4	`float`	`__m128`	`ps`
Double Precision Floats	2	`double`	`__m128d`	`pd`

Table 2: Packed data types supported by the SSE instructions. The fixed-length C-types requires the inclusion of `stdint.h`.

integer, i.e. the register size of a modern x86, is known as a quadword. Consequently, a 32-bit integer is known as a doubleword.

2.1 Using SSE in C-code

Using SSE in a modern C-compiler is fairly straightforward. In general, no assembler coding is needed. Most modern compilers expose a set of vector types and intrinsics to manipulate them. We will assume that the compiler supports the same SSE intrinsics as the Intel C-compiler. The intrinsics are enabled by including a the correct header file. The name of the header file depends on the SSE version you are targeting, see Table 1. You may also need to pass an option to the compiler to allow it to generate SSE code, e.g. `-msse4.1`. A portable application would normally try to detect which SSE extensions are present by running the `CPUID` instruction and use a fallback algorithm if the expected SSE extensions are not present. For the purpose of this lab, we simple ignore those portability issues and assume that at least SSE 4.1 is present, which is the case for the 45 nm Core 2 and newer.

The SSE intrinsics add a set of new data types to the language, these are summarized in Table 2. In general, the data types provided to support SSE provide little protection against programmer errors. Vectors of integers of different size all use the same vector type (`__m128i`), there are however separate types for vectors of single and double precision floating point numbers.

The vector types do not support the native C operators, instead they require explicit use of special intrinsics. All SSE intrinsics have a name on the form `_mm_<op>_<type>`, where `<op>` is the operation to perform and `<type>` specifies the data type. The most common types are listed in Table 2.

The following sections will present some useful instructions and examples to get you started with SSE. It is not intended to be an exhaustive list of available instructions or intrinsics. In particular, most of the instructions that rearrange data within vectors (shuffling), various data-packing instructions and generally esoteric instructions have

	Intrinsic	Assembler	Vector Type
Unaligned	_mm_loadu_si128	MOVDQU	__m128i
	_mm_storeu_si128	MOVDQU	__m128i
	_mm_loadu_ps	MOVUPS	__m128
	_mm_storeu_ps	MOVUPS	__m128
	_mm_loadu_pd	MOVUPD	__m128d
	_mm_storeu_pd	MOVUPD	__m128d
	_mm_load1_ps	Multiple	__m128
	_mm_load1_pd	Multiple	__m128d
Aligned	_mm_load_si128	MOVDQA	__m128i
	_mm_store_si128	MOVDQA	__m128i
	_mm_load_ps	MOVAPS	__m128
	_mm_store_ps	MOVAPS	__m128
	_mm_load_pd	MOVAPD	__m128d
	_mm_store_pd	MOVAPD	__m128d
Streaming	_mm_stream_si128	MOVNTDQ	__m128i
	_mm_stream_ps	MOVNTPS	__m128
	_mm_stream_pd	MOVNTPD	__m128d
	_mm_stream_load_si128	MOVNTDQA	__m128i

Table 3: Load and store operations. The `load1` operation is used to load one value into all elements in a vector.

been left out. Interested readers should refer to the optimization manuals from the CPU manufacturers for a more thorough introduction.

2.2 Loads and stores

There are three classes of load and store instructions for SSE. They differ in how they behave with respect to the memory system. Two of the classes require their memory operands to be naturally aligned, i.e. the operand has to be aligned to its own size. For example, a 64-bit integer is naturally aligned if it is aligned to 64-bits. The following memory accesses classes are available:

Unaligned A "normal" memory access. Does not require any special alignment, but may perform better if data is naturally aligned.

Aligned Memory access type that requires data to be aligned. Might perform slightly better than unaligned memory accesses. Raises an exception if the memory operand is not naturally aligned.

Streaming Memory accesses that are optimized for data that is streaming, also known as non-temporal, and is not likely to be reused soon. Requires operands to be naturally aligned. Streaming stores are generally much faster than normal stores since they can avoid reading data before the writing. However, they require data to be written sequentially and, preferably, in entire cache line units.

See Table 3 for a list of load and store intrinsics and their corresponding assembler instructions. A usage example is provided in Listing 1. Constants should usually not be loaded using these instructions, see section 2.5 for details about how to load constants and how to extract individual elements from a vector.

Listing 1: Load store example using *unaligned* accesses

```
#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);
    }
}
```

2.3 Arithmetic operations

All of the common arithmetic operations are available in SSE. Addition, subtraction and multiplication is available for all vector types, while division is only available for floating point vectors.

A special *horizontal add* operation is available to add pairs of values, see Figure 1, in its input vectors. This operation can be used to implement efficient reductions. Using this instruction to create a vector of sums of four vectors with four floating point numbers can be done using only three instructions.

There is an instruction to calculate the scalar product between two vectors. This instruction takes three operands, the two vectors and an 8-bit flag field. The four highest bits in the flag field are used to determine which elements in the vectors to include in the calculation. The lower four bits are used as a mask to determine which elements in the destination are updated with the result, the other elements are set to 0. For example, to include all elements in the input vectors and store the result to the third element in the destination vector, set flags to $\mathrm{F4}_{16}$.

A transpose macro is available to transpose 4×4 matrices represented by four vectors of packed floats. The transpose macro expands into several assembler instructions that perform the in-place matrix transpose.

Individual elements in a vector can be compared to another vector using compare intrinsics. These operations compare two vectors; if the comparison is true for an element, that element is set to all binary 1 and 0 otherwise. Only two compare instructions, equality and greater than, working on integers are provided by the hardware. The less than operation is synthesized by swapping the operands and using the greater than comparison. See Listing 3 for an example of how to use the SSE compare instructions.

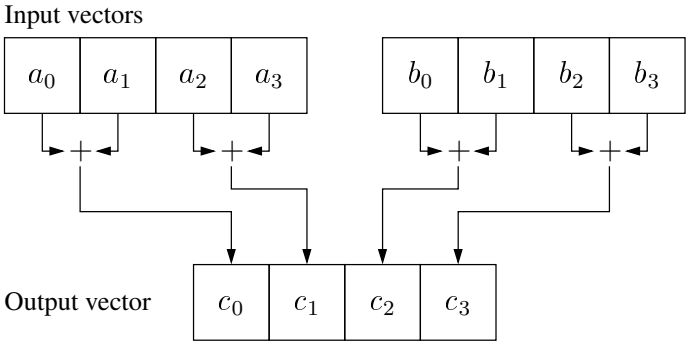
2.4 Bitwise operations

Bitwise SSE operations behave exactly like their non-SSE counter parts, the only difference is the size of the operands. All operations work on the entire register. Note that there is a different set of operations for integers and floating point types, even though the instructions are functionally identical. The CPU uses the information about the data type to eliminate a potential stall due to data dependencies in the vector pipelines.

Intrinsic	Operation
`_mm_add_<type>(a, b)`	$c_i = a_i + b_i$
`_mm_sub_<type>(a, b)`	$c_i = a_i + b_i$
`_mm_mul_(ps|pd)(a, b)`	$c_i = a_i b_i$
`_mm_div_(ps|pd)(a, b)`	$c_i = a_i / b_i$
`_mm_hadd_(ps|pd)(a, b)`	Performs a horizontal add, see Figure 1
`_mm_dp_(ps|pd)(a, b, FLAGS)`	$\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ (dot product)
`_MM_TRANSPOSE4_PS(a, ..., d)`	Transpose the matrix $(a^{\mathrm{t}} \ldots d^{\mathrm{t}})$ in place
`_mm_cmpeq_<type>(a, b)`	Set c_i to -1 if $a_i = b_i$, 0 otherwise
`_mm_cmpgt_<type>(a, b)`	Set c_i to -1 if $a_i > b_i$, 0 otherwise
`_mm_cmplt_<type>(a, b)`	Set c_i to -1 if $a_i < b_i$, 0 otherwise

Table 4: Arithmetic operations available in SSE. The transpose operation is a macro that expands to several SSE instructions to efficiently transpose a matrix.

Figure 1: Calculating c = `_mm_hadd_ps(a, b)`



Listing 2: Summarize the elements of four vectors and store each vectors sum as one element in a destination vector

```
#include <pmmintrin.h>

static __m128
vec_sum(const __m128 v0, const __m128 v1,
        const __m128 v2, const __m128 v3)
{
    return _mm_hadd_ps(
        _mm_hadd_ps(v0, v1),
        _mm_hadd_ps(v2, v3));
}
```

Intrinsic	Operation (per bit)	Assembler
_mm_and_si128(a, b)	$c = a \wedge b$	PAND
_mm_andnot_si128(a, b)	$c = \neg a \wedge b$	PANDA
_mm_or_si128(a, b)	$c = a \vee b$	POR
_mm_xor_si128(a, b)	$c = a \oplus b$	PXOR
_mm_and_(ps|pd)(a, b)	$c = a \wedge b$	AND(PS|PD)
_mm_andnot_(ps|pd)(a, b)	$c = \neg a \wedge b$	ANDN(PS|PD)
_mm_or_(ps|pd)(a, b)	$c = a \vee b$	OR(PS|PD)
_mm_xor_(ps|pd)(a, b)	$c = a \oplus b$	XOR(PS|PD)

Table 5: Bitwise operations available in SSE. All operations are performed bitwise on entire 128-bit vector registers.

Intrinsic	Operation
_mm_set_<type>(p_0, ..., p_n)	$c_i = p_i$
_mm_setzero_(ps|pd|si128)()	$c_i = 0$
_mm_set1_<type>(a)	$c_i = a$
_mm_cvtss_f32(a)	Extract the first float from a
_mm_cvtsd_f64(a)	Extract the first double from a

Table 6: Miscellaneous operations. Most of the operations expand into multiple assembler instructions.

2.5 Loading constants and extracting elements

There are several intrinsics for loading constants into SSE registers. The most general can be used to specify the value of each element in the vector. In general, try to use the most specific intrinsic for your needs. For example, to load 0 into all elements in a vector, _mm_set_epi64>, _mm_set1_epi64> or _mm_setzero_si128> could be used. The two first will generate a number of instructions to load 0 into the two 64-bit integer positions in the vector. The _mm_setzero_si128> intrinsic uses a shortcut and emits a PXOR instruction to generate a register with all bits set to 0.

There are a couple of intrinsics to extract the first element from a vector. They can be useful to extract results from reductions and similar operations.

2.6 Data alignment

Aligned memory accesses are usually required to get the best possible performance. There are several ways to allocate aligned memory. One would be to use the POSIX API, but posix_memalign has an awkward syntax and is unavailable on many platforms. A more convenient way is to use the intrinsics in Table 7. Remember that data allocated using _mm_malloc must be freed using _mm_free.

It is also possible to request a specific alignment of static data allocations. The preferred way to do this is using GCC attributes, which is also supported by the Intel

Intrinsic	Operation
_mm_malloc(s, a)	Allocate s B of memory with a B alignment
_mm_free(*p)	Free data previously allocated by _mm_malloc(s, a)

Table 7: Memory allocation

Listing 3: Transform an array of 16-bit integers using a threshold function. Values larger than the threshold (4242) are set to FFFF_{16} and values smaller than the threshold are set to 0

```
#include <stdint.h>
#include <pmmintrin.h>

static void
threshold(uint16_t *dst, const uint16_t *src, size_t len)
{
    const __m128i t = _mm_set1_epi16(4242);
    for (int i = 0; i < len; i += 8) {
        const __m128i v = _mm_loadu_si128((__m128i *)(src + i));
        _mm_storeu_si128((__m128i *)(dst + i),
                         _mm_cmpgt_epi16(v, t));
    }
}
```

Listing 4: Aligning static data using attributes

```
float foo[SIZE] __attribute__((aligned (16)));
```

compiler. See Listing 4 for an example.

3 Using Uppmax

In this lab we will be using the Kalkyl cluster[4] at Uppmax. To login on the cluster, connect with *SSH* to *kalkyl.uppmax.uu.se*, e.g. using:
host$ ssh -Y *user-name*@kalkyl.uppmax.uu.se.

You are now connected to one of the login nodes of the cluster. These nodes are only used to test code and submit jobs to the cluster.

Before you start working on the lab, you need to load the GCC module to ensure that you have a recent GCC version:
host$ module load gcc

To setup the working environment, either download the source files from the course homepage or execute the following command:
host$ ~ansan501/avdark/2010/init_lab4.sh

You should develop and test your solutions on the login node. However, you should run your performance experiments in the cluster. The easiest way to do this is to schedule an interactive shell with a short run time. Use the following command:
host$ salloc -p node -n 1 -t 15:00 --qos=short -A g2010003

The command above allocates an entire node for 15 minutes. Restricting the runtime to 15 minutes or less allows the job to be enqueued in a special high-priority queue for short jobs. If you need to run a longer job, just remove the `-qos=short` option and increase the estimated runtime.

[4] `http://www.uppmax.uu.se/systems/kalkyl`

Listing 5: Conditional lower case conversion algorithm

```
static void
lcase_simple(char *dst, const char *src, size_t len)
{
        const char *cur = src;
        while (cur != src + len)
                *(dst++) = *(cur++) | 0x20;
}
```

4 The lab framework

Makefile Automates the compilation. You can use **`make clean`** to remove automatically generated files from the working directory.

lcase.c Skeleton code for the text conversion part of the lab. Contains a testing and timing harness that tests that your vectorized version is correct and computes the speedup compared to the serial reference version.

matmul.c Skeleton code for multiplying matrices. Also contains reference code for testing and timing.

matvec.c Skeleton code for multiplying a matrix and a vector. Also contains reference code for testing and timing.

util.(c|h) Utility functions for printing vectors and measuring time. See the header file for more information.

5 Converting text into lower-case

We will assume that all characters we need to handle can be represented in the ASCII[5] character set. This means that we only care about A through Z and disregard silly inventions like umlauts.

It turns out that the ASCII character set is ordered so that the case of a character is determined by one bit. To convert a character into lower-case, simply logically OR it with 20_{16}. Listing 5 shows how this is done in plain C. Converting this code into SSE is fairly straight forward, just unroll the loop 16-times (SSE registers are 128 bits, which means that they can hold 16-bytes) and replace the serial operations with SIMD operations. There are, however, a few gotchas. Depending on what kind of memory access you use, you may have to make sure that data is aligned. You also have to make sure that data sizes that are not even multiples of the SIMD register length are correctly handled.

In order to preserve symbols that are not letters, we need to check that the character code is within the range of the upper-case letters (41_{16}–$5A_{16}$), this can easily be accomplished in C by the code in Listing 6. Converting this into SIMD is not as straight forward as converting the code in Listing 5. The intuitive way to handle conditionals in serial code is to change the control flow, this is usually undesirable in SIMD code. Instead, you have to resort to bit manipulation.

[5] `http://en.wikipedia.org/wiki/ASCII`

Listing 6: Conditional lower case conversion algorithm

```
static void
lcase_cond(char *dst, const char *src, size_t len)
{
        const char *cur = src;
        while (cur != src + len) {
                const char c = *(cur++);
                *(dst++) = (c >= 'A' && c <= 'Z') ?
                    c | 0x20 : c;
        }
}
```

Listing 7: Conditional lower case conversion using bit manipulation

```
static void
lcase_cond2(char *dst, const char *src, size_t len)
{
        const char *cur = src;
        while (cur != src + len) {
                const char c = *(cur++);
                *(dst++) = c | (cmpgt(c, 'A' - 1)
                                & cmpgt('Z' + 1, c)
                                & 0x20);
        }
}
```

Imagine that you have a function, `cmpgt(a, b)`, that evaluates the expression $a > b$ and returns a bit pattern that is all ones if the result is true and zero otherwise. This would allow us to remove the conditions in the control flow and replace it with a logical expression. The result of this transformation is shown in Listing 7.

5.1 Tasks

1. Implement the simple algorithm in the function `lcase_sse_simple()` using SSE. Test that your implementation is correct by running `lcase_MOVDQU`.

2. Implement the full algorithm using SSE in `lcase_sse_cond()`. Test that your implementation is correct by running `lcase_MOVDQU`.

3. Run `lcase_MOVDQU`. Why is the SIMD version faster than the reference version?

4. Compare the performance of `lcase_MOVDQU`, `lcase_MOVDQA` and `lcase_MOVNTDQ`. Explain the differences in performance.

6 Multiplying a matrix and a vector

Multiplying a matrix and a vector can be accomplished by the code in Listing 8, this should be familiar if you have taken a linear algebra course. The first step in vectorizing

Listing 8: Simple matrix-vector multiplication

```
static void
matvec_simple(size_t n, double vec_c[n],
              const double mat_a[n][n], const double vec_b[n])
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            vec_c[i] += mat_a[i][j] * vec_b[j];
}
```

Listing 9: Matrix-vector multiplication, unrolled four times

```
static void
matvec_unrolled(size_t n, double vec_c[n],
                const double mat_a[n][n], const double vec_b[n])
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j += 4)
            vec_c[i] += mat_a[i][j + 0] * vec_b[j + 0]
                      + mat_a[i][j + 1] * vec_b[j + 1]
                      + mat_a[i][j + 2] * vec_b[j + 2]
                      + mat_a[i][j + 3] * vec_b[j + 3];
}
```

this code is to unroll it four times. Since we are working on 32-bit floating point elements, this allows us to process 4 elements in parallel using the 128-bit SIMD registers in x86. The unrolled code is shown in Listing 9.

6.1 Tasks

1. Implement your version of the matrix-vector multiplication in the `matvec_sse()` function. Run your code and make sure that it produces the correct result. Is it faster than the traditional serial version?

2. Can you think of any optimizations that may make this code faster? *You don't need to implement them.*

7 Matrix-matrix multiplication

The simplest way to multiply two matrices is to use the algorithm in Listing 10. Again, the first step in converting this algorithm to SSE is to unroll some of the loops. The simplest vectorization of this code is to unroll the inner loop 4 times, remember that we can fit four single precision floating point numbers in a vector, and use vector instructions to compute the results of the inner loop.

Listing 10: Matrix-matrix multiplication

```
static void
matmat(size_t n, double mat_c[n][n],
       const double mat_a[n][n], const double mat_b[n][n])
{
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                mat_c[i][j] += mat_a[i][k] * mat_b[k][j];
            }
        }
    }
}
```

7.1 Tasks

1. Implement a vectorized version of Listing 10 in the `matmul_sse()` function that belongs to the SSE mode. (Search for the *TASK:* comment). Run your solution to check that it is correct and measure its speedup compared to the serial version. What is the speedup?

2. **Bonus:** Finish the blocked version of the matrix multiplication by implementing an efficient vectorized version of `matmul_block_sse()`. Run your solution to check that it is correct and measure its speedup compared to the serial version. What is the speedup?