

**An-Najah National University
Computer Engineering Department
Parallel Processing - 10636523**

SIMD Vectorization
Programming Assignment #2

1 Introduction

The purpose of this assignment is to give some experience in using SIMD instructions on x86 and getting compiler auto-vectorization to work. We will use matrix- vector and matrix-matrix multiplication to illustrate how SIMD can be used for numerical algorithms.

You will be using GCC in this experiment. 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 are much better documented.

Both Intel¹ and AMD² 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 assignment. You will, however, find them, and the instruction set references, useful as reference literature when using SSE. Another useful reference is the Intel C++ compiler manual³, which documents the SSE intrinsics supported by ICC and GCC.

We will, for various practical reasons, use the Linux lab machines for this lab assignment. 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 lab systems you'll be using are 64-bit machines.

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.

¹<http://www.intel.com/products/processor/manuals/>

²<http://developer.amd.com/documentation/guides/>

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

Header file	Extension name	Abbrev.
<code>xmmintrin.h</code>	Streaming SIMD Extensions	SSE
<code>emmintrin.h</code>	Streaming SIMD Extensions 2	SSE2
<code>pmmmintrin.h</code>	Streaming SIMD Extensions 3	SSE3
<code>tmmintrin.h</code>	Supplemental Streaming SIMD Extensions 3	SSSE3
<code>smmintrin.h</code>	Streaming SIMD Extensions 4 (Vector math)	SSE4.1
<code>nmmintrin.h</code>	Streaming SIMD Extensions 4 (String processing)	SSE4.2
<code>gmmintrin.h</code>	Advanced Vector Extensions Instructions	AVX

Table 1: Header files used for different SSE versions

When reading the manuals, it's important to keep in mind that the size of a *word* in the x86-world is 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 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 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. `-msse3`. 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 assignment, we simply ignore those portability issues and assume that at least SSE3 is present, which is the norm for processors released since 2005.

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. This 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 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

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`.

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);
    }
}
```

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. We will not be using this type in the lab.

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 2.4 for details about how to load constants and how to extract individual elements from a vector.

	Intrinsic	Assembler	Vector Type
Unaligned	<code>_mm_loadu_si128</code>	MOVDQU	<code>__m128i</code>
	<code>_mm_storeu_si128</code>	MOVDQU	<code>__m128i</code>
	<code>_mm_loadu_ps</code>	MOVUPS	<code>__m128</code>
	<code>_mm_storeu_ps</code>	MOVUPS	<code>__m128</code>
	<code>_mm_loadu_pd</code>	MOVUPD	<code>__m128d</code>
	<code>_mm_storeu_pd</code>	MOVUPD	<code>__m128d</code>
	<code>_mm_load1_ps</code>	Multiple	<code>__m128</code>
	<code>_mm_load1_pd</code>	Multiple	<code>__m128d</code>
Aligned	<code>_mm_load_si128</code>	MOVDQA	<code>__m128i</code>
	<code>_mm_store_si128</code>	MOVDQA	<code>__m128i</code>
	<code>_mm_load_ps</code>	MOVAPS	<code>__m128</code>
	<code>_mm_store_ps</code>	MOVAPS	<code>__m128</code>
	<code>_mm_load_pd</code>	MOVAPD	<code>__m128d</code>
	<code>_mm_store_pd</code>	MOVAPD	<code>__m128d</code>
Streaming	<code>_mm_stream_si128</code>	MOVNTDQ	<code>_m128i</code>
	<code>_mm_stream_ps</code>	MOVNTPS	<code>_m128</code>
	<code>_mm_stream_pd</code>	MOVNTPD	<code>_m128d</code>
	<code>_mm_stream_load_si128</code>	MOVNTDQA	<code>_m128i</code>

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

2.3 Arithmetic operations

All of the common arithmetic operations are available in SSE, see Table 4. 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 and Listing 2, 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.

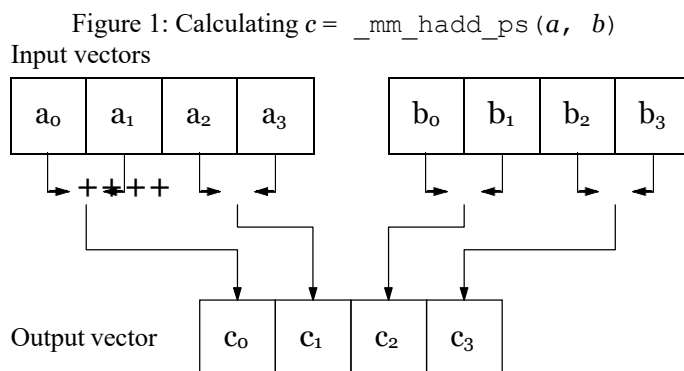
There is an instruction to calculate the scalar product between two vectors. This instruction takes three operands, the two vectors and an 8-bit flag field. The four highest bits in the flag field are used to determine which elements in the vectors to include in the calculation. The lower four bits are 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 `F416`.

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.

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



Listing 2: Sum 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
<code>_mm_set_<type>(p₀, ..., p_n)</code>	$c_i = p_i$
<code>_mm_setzero_(ps pd sil28)()</code>	$c_i = 0$
<code>_mm_set1_<type>(a)</code>	$c_i = a$
<code>_mm_cvtss_f32(a)</code>	Extract the first float from a
<code>_mm_cvtsd_f64(a)</code>	Extract the first double from a

Table 5: Miscellaneous operations. Most of the operations expand into multiple assembler instructions.

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 zero.

```
#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));
    }
}
```

2.4 Loading constants and extracting elements

There are several intrinsics for loading constants into SSE registers, see Table 5. The most general can be used to specify the value of each element in a 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.

2.5 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 6. 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 compiler. See Listing 4 for an example.

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

Table 6: Memory allocation

Listing 4: Aligning static data using attributes

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

3 Getting started

In this homework we will be using a Linux or Windows machine with gcc. If you're using your own computer, be aware that vectorization support and implementation varies completely from computer to computer.

4 The code framework

Open the file `simd.cpp` available from the homework resources. The code in the file contains two versions of the `VectorxVector` dot product. A vectorized version using SSE instructions and a traditional (scalar) version. Try to figure out the correct compilation parameters to use for GCC. Once the code is compiled run it and observe if the vectorized version is running faster or slower than the scalar version. Try experimenting with the size of the input vectors.

5 Multiplying a matrix and a vector

Multiplying a matrix and a vector can be accomplished by the code in Listing 5, this should be familiar if you have taken a linear algebra course. The first step in vectorizing 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. The unrolled code is shown in Listing 6.

Listing 5: Simple matrix-vector multiplication

```
static void
matvec_simple ( size_t n , float vec_c[ n ] ,
                const float mat_a[ n ][ n ] , const float 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 6: Matrix-vector multiplication, unrolled four times

```
static void
matvec_unrolled( size_t n , float vec_c[ n ] ,
                 const float mat_a[ n ][ n ] , const float 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 ] ;
}
```

5.1 Tasks

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

6 Matrix-matrix Multiplication

The simplest way to multiply two matrices is to use the algorithm in Listing 7. 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.

6.1 Tasks

1. Implement a vectorized version of Listing 7 in a function that is called `matmul_sse()`. Run your solution to check that it is correct and measure its speedup compared to the serial version. What is the speedup?

Listing 7: Matrix-matrix multiplication

```
static void
matmat (size_t n, float mat_c[n][n],
        const float mat_a[n][n], const float 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];
            }
        }
    }
}
```

Good Luck

References

Adapted from Prof. Marcus Holm.