

# Lab 4: SIMD

**Due** May 7 by 12pm      **Points** 1


## Introduction

The purpose of this lab assignment 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 assignment. 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](http://www.intel.com/products/processor/manuals/) [\\_](http://www.intel.com/products/processor/manuals/)(<http://www.intel.com/products/processor/manuals/>) and [AMD](https://developer.amd.com/resources/developer-guides-manuals/) [\\_](https://developer.amd.com/resources/developer-guides-manuals/)(<https://developer.amd.com/resources/developer-guides-manuals/>) 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. A more useful reference for this lab is the [Intel Intrinsics Guide](https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=MMX,SSE,SSE2,SSE3,SSSE3,SSE4_1) [\\_](https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=MMX,SSE,SSE2,SSE3,SSSE3,SSE4_1)([https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=MMX,SSE,SSE2,SSE3,SSSE3,SSE4\\_1](https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=MMX,SSE,SSE2,SSE3,SSSE3,SSE4_1)), which documents all the intrinsics available.

You are highly encouraged to solve this assignment in groups of two students (using the same groups as for previous labs). Contact the teaching assistants if you have any problems with your current group. During the examination, you will be asked to demonstrate and explain your solutions.

 Lab has three scheduled sessions: Lab 4 PREP (where you can start the lab and ask questions) and Lab 4A and Lab 4B (where you can ask questions and be examined). During the PREP-lab we may be available in the Zoom lab (link below) otherwise the best way to reach us is by email (address below). You are only required to attend one of the A and B sessions. Please sign up for which slot you want to attend under [People -> Groups -> Lab 4](#). Please try to show up at the lab slot that you have registered for. In case of long queues, priority will be given to people that have registered for the slot.

During the the A and B session, please use the lab queue (link below) to indicate when you need help or is ready to be examined. We will attend to you as soon a possible. If you finish the lab before the session you signed up for, you can pre-book an examination slot [here](https://forms.gle/8igKhCko3JoAhdqe8) [\\_](https://forms.gle/8igKhCko3JoAhdqe8)(<https://forms.gle/8igKhCko3JoAhdqe8>) [\\_](https://forms.gle/8igKhCko3JoAhdqe8)(<https://forms.gle/8igKhCko3JoAhdqe8>). A schedule of pre-booked slots will be published here around 17.00 the day before each session (and it will be updated again right before the lab starts).

Remember that it is mandatory to pass the lab in order to pass the course, with the exception of the bonus questions, which give you one bonus point.

Zoom link: <https://uu-se.zoom.us/j/69544346205> [\\_](https://uu-se.zoom.us/j/69544346205)(<https://uu-se.zoom.us/j/69544346205>)

Lab queue (available during the lab): <https://forms.gle/1cFCgfvNLgQwXMi69> [\\_](https://forms.gle/1cFCgfvNLgQwXMi69)(<https://forms.gle/1cFCgfvNLgQwXMi69>)

TA email: [it-avdark-ta@lists.uu.se](mailto:it-avdark-ta@lists.uu.se) [\\_](mailto:it-avdark-ta@lists.uu.se)(<mailto:it-avdark-ta@lists.uu.se>)

## Pre-booked lab schedule

Following is a schedule of pre-booked lab slots. If you are on the list, you do not need to sign up to the lab queue durign the lab to be examined. Please join the Zoom meeting and enter your breakout room and setup 10 minutes ahead of your slot if possible. I will try to adhere to the schedule as closely as possible but please be patient if there is a delay for a few minutes.

Time (Friday)	Group
08:15 - 08:30	Group 6 (Jonathan & Guru & Christos)
08:30 - 08:45	Group 9 (Bruno & Bispor)

# 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 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.

## 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. 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 assignment, we simple ignore those portability issues and assume that at least SSE 4.1 is present, which is the case for the 45nm Core 2 and newer.

 SSE intrinsics add a set of new data types to the language, these are summarized in the table that follows. 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`), but there are 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 here:

Intel Name	Number of elements in each vector	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

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 been left out. If you are interested, you should refer to the optimization manuals from the CPU manufacturers for a more thorough introduction.

## 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, especially on older processors.
- **Aligned:** Memory access type that requires data to be aligned. Might perform slightly better than unaligned memory accesses, especially on older processors. 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 can be 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.

You can find a non-exhaustive list of load and store intrinsics and their corresponding assembler instructions in the table that follows:

	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

Here you can also see a usage example. Constants should usually not be loaded using these instructions, instead we will see later how to correctly handle them.

```
#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) {
```

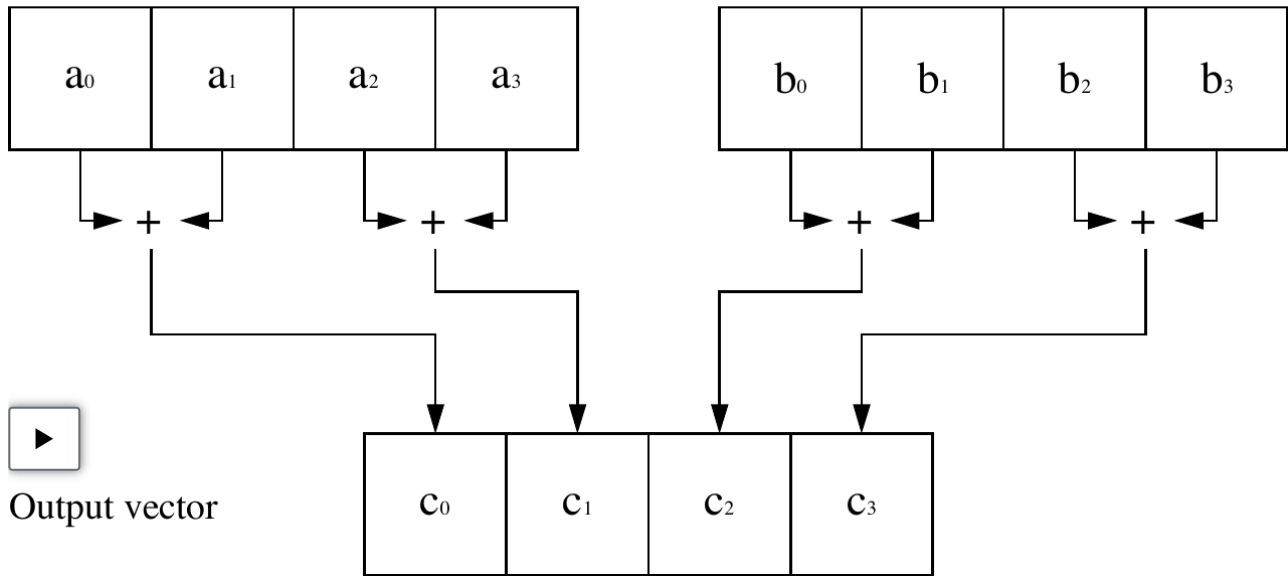
```
__m128i v = _mm_loadu_si128((__m128i*)(src + i));
_mm_storeu_si128((__m128i*)(dst + i), v);
}
```

## 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 in its input vectors. This operation can be used to implement efficient reductions. For example, `c = _mm_hadd_ps(a, b)` produces the following result:

Input vectors



Output vector

Using this instruction to create a vector of sums of four vectors with four floating point numbers can be done using only three instructions:

```
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));
}
```

There is also 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  $F4_{16}$ .


A transpose macro is available to transpose  $4 \times 4$  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. The following listing contains an example of how to use the SSE compare instructions:

```
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));
}
```

You can find a non-exhaustive list of arithmetic intrinsics and their corresponding operations in the table that follows:

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)	$c_i = a_i b_i$ ( <b>element-wise multiplication</b> )
_mm_div_(ps pd)(a , b)	$c_i = a_i / b_i$
_mm_hadd_(ps pd)(a, b)	Performs a horizontal add
_mm_dp_(ps pd)(a , b, FLAGS)	$c_i = a_i \cdot b_i$ ( <b>dot product</b> )
_MM_TRANSPOSE4_PS(a , ..., d)	Transpose the matrix ( $a^t \dots d^t$ ) in place
_mm_cmpeq_<type>(a, b)	Set all bits of $c_i$ to 1 if $a_i = b_i$ , 0 otherwise
_mm_cmpgt_<type>(a, b)	Set all bits of $c_i$ to 1 if $a_i > b_i$ , 0 otherwise
 _mm_cmplt_<type>(a, b)	Set all bits of $c_i$ to 1 if $a_i < b_i$ , 0 otherwise

## 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 (per bit)	Assembler
_mm_and_si128(a, b)	$c = a \& b$	PAND
_mm_andnot_si128(a, b)	$c = (!a) \& b$	PANDA
_mm_or_si128(a, b)	$c = a   b$	POR
_mm_xor_si128(a, b)	$c = a \wedge b$	PXOR
_mm_and_(ps pd)(a, )	$c = a \& b$	AND(PS PD)
_mm_andnot_(ps pd)(a, b)	$c = (!a) \& b$	ANDN(PS PD)
_mm_or_(ps pd)(a, b)	$c = a   b$	OR(PS PD)
_mm_xor_(ps pd)(a, b)	$c = a \wedge b$	XOR(PS PD)

## 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 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, the `_mm_setzero_si128` intrinsic emits a *single* `PXOR` instruction to generate a register with all bits set to 0. Meanwhile, the `_mm_set1_epi64` and `_mm_set_epi64` instructions could also be used but will generate *multiple* instructions to load 0 into the two 64-bit integer positions in the vector.

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.

Intrinsic	Operation
<code>_mm_set_&lt;type&gt;(p<sub>0</sub>, ..., p<sub>n</sub>)</code>	$c_i = p_i$
<code>_mm_setzero_(ps pd si128)()</code>	$c_i = 0$
<code>_mm_set1_&lt;type&gt;(a)</code>	$c_i = a$
<code>_mm_cvtss_f32(a)</code>	Extract the first (least significant) float from a
<code>_mm_cvtsd_f64(a)</code>	Extract the first (least significant) double from a

## Data alignment

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

Intrinsic	Operation
<code>_mm_malloc(s, a)</code>	Allocate <i>s</i> bytes of memory aligned at <i>a</i> bytes.
<code>_mm_free(*p)</code>	Free memory previously allocated by <code>_mm_malloc</code> .

## IT Department's Linux servers

Much like we did for the other labs, we will be using the department's Linux servers. Any of the [Ubuntu 18.04 x86\\_64 machines that the department provides](https://www.it.uu.se/datordrift/maskinpark/linux) [\\_](https://www.it.uu.se/datordrift/maskinpark/linux) (<https://www.it.uu.se/datordrift/maskinpark/linux>) are suitable for this lab. You are also free to use your own computer, but make sure that it supports the SSE instructions we will be using (most recent Intel and AMD processors do). Also, be aware that certain compiler and optimisation combinations might make it hard to write SSE code that is faster than the compiler-generated code. The provided `Makefile` uses `gcc` with `-O2` optimisation which should make it possible to see improved behaviour.

If you are running on one of the servers at the department, keep in mind that other people might also be using them at the same time. If you get unintuitive results, check that nobody else is stressing the system, affecting your results as well.

## The code for the lab

You can find the skeleton code for the lab here: [Assignments 4 -> Lab 4 Files](#)

Below is a brief description of the files.

- `Makefile`: Automates the compilation. You can use the usual `make all`, `make clean`, and `make test` commands.

- `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.
- `matvec.c`: Skeleton code for multiplying a matrix with a vector. Also contains reference code for testing and timing.
- `matmul.c`: Skeleton code for multiplying matrices. 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.

## Part 1: Converting text into lower-case

We will assume that all characters we need to handle can be represented in the [ASCII](http://en.wikipedia.org/wiki/ASCII) character set. This means that we only care about A through Z (although, feel free to suggest an alteration to support special characters as well).

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  $0x20_{16}$ :

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

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 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 ( $0x41_{16}$ - $0x5A_{16}$ ), this can easily be accomplished in C by the following code:

```
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;
    }
}
```

Converting this into SIMD is not as straight forward as converting the previous code. 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.

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 (sound familiar?). 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 here:

```
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);
    }
}
```



## 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` again.
3. Is your SSE version faster than the reference version? Why?
4. Compare the performance of `lcase_MOVDQU`, `lcase_MOVDQA`, and `lcase_MOVNTDQ` (you can use `make test_lcase` to run all these variants). What is the difference between the variants? Do you expect differences in performance? Why? Are there differences in performance? Why?

## Part 2: Multiplying a matrix and a vector

If you have ever taken a linear algebra course, you should be familiar with how to multiply a matrix with a vector:

```
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];
}
```

To vectorize this code using SSE, you should first unroll it. Since we are working with 32-bit floating point elements and 128-bit vectors, we can process four elements in parallel, so we have to unroll the code 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];
}
```

## Tasks

1. Implement your version of the matrix-vector multiplication using SSE 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? Why?
2. Think of some optimizations that may make the code faster and discuss them with the TAs. *You don't need to implement them.*

## Part 3: Multiplying two matrices

The simplest way of multiplying two matrices is with the following code:

```
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];
}
```

Much like we had to do for the matrix-vector multiplication, the first step to vectorizing this code is to unroll it, and then proceed from there.

## Tasks

1. Implement an **efficient** vectorized version of the matrix-matrix multiplication algorithm in the `matmul_sse()` function. Run your solution to check that it is correct and measure the speedup compared to the serial version. How much is the speedup and why?
2. Bonus: [Please see the bonus task.](#)

