High Performance Programming
Uppsala University
Spring 2018
Lab 8 — SIMD and Vectorization

February 8, 2018

1 Introduction

Abbreviations used below:
Single **I**nstruction, **M**ultiple **D**ata (SIMD)
Streaming **SIMD** **E**xtensions (SSE)

The purpose of this lab is to give some experience in using SIMD instructions on x86 architectures and getting compiler auto-vectorization to work.

You will be using GCC in this lab. GCC supports two sets of intrinsics, or built-in functions, 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 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 may, however, find them, and the instruction set references, useful as reference literature when using SSE.

The **Intel Intrinsics Guide** is an interactive reference tool for Intel intrinsic instructions and we recommend to use it in this lab:
https://software.intel.com/sites/landingpage/IntrinsicsGuide/[3]
It can be very useful for looking up which intrinsic functions exist, and precisely what each function does.

2 Getting started

In this lab **we will mainly be using the Linux lab machines**. If you're using your own computer, be aware that vectorization support and implementation varies depending on the CPU model, so if you have an older CPU model the lab instructions may not work at all. Check the flags in `/proc/cpuinfo` (or use the `lscpu` command) to

[1] `http://www.intel.com/products/processor/manuals/`
[2] `https://developer.amd.com/resources/developer-guides-manuals/`
[3] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

see which instruction sets are available on the computer you are using (more about that below).

Download the source package `Lab08_SIMD_and_Vec.tar.gz` from the Student Portal and extract the files.

3 Introduction to SSE

The SSE extension to the x86 architecture 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.

Each vector register can contain several numbers; how many depends on the datatype of the elements. For example, a 128-bit vector register can contain two 64-bit numbers or four 32-bit numbers. The elements in the vector are sometimes referred to as being "packed" since they sit right next to each other in the vector register.

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 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.

3.1 Using SSE in C-code

Using SSE with a modern C-compiler is fairly straightforward. In general, no assembler coding is needed. Most modern compilers expose a set of vector types and intrinsic functions (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 lab, 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 intrinsic functions. 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 the rightmost column in Table 2.

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
`immintrin.h`	Advanced Vector Extensions Instructions	AVX

Table 1: Header files used for different SSE versions (different instruction set extensions). The more recent instruction set extensions AVX2 and AVX-512 also use the `immintrin.h` header file; the same one as AVX.

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 "Elements/Reg." column gives the number of elements that fit into one 128-bit vector register. The fixed-length C-types requires the inclusion of `stdint.h`.

3.2 Using AVX in C-code

In the AVX instruction set the 128-bit registers are extended to 256-bit registers. The AVX instruction set uses a similar naming convention as SSE. The intrinsic vector functions have names that begin with `_mm256`.

For example, a vector of integers denoted by `__m256i` and the function to store 256-bits of integer data into the memory is `_mm256_store_si256`. To compile AVX code, pass the `-mavx` option to the compiler.

The following sections will present some useful instructions and examples to get you started with SSE and AVX. 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.

3.3 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 access classes are available:

Unaligned A "normal" memory access. Does not require any special alignment, but

	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

Table 3: Load and store operations. The suffixes `ps` and `pd` here refer to single and double precision floating-point numbers, respectively (`ps`: "packed single", `pd`: "packed double"). The `load1` operation is used to load one value into all elements in a vector.

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.

Note that constants should usually not be loaded using these instructions, see subsection 3.5 for details about how to load constants and how to extract individual elements from a vector.

Task-1

1. Issue the command "`cat /proc/cpuinfo`" and examine the `flags` part of the output. Look for the abbreviations for the different SIMD extensions in Table 1 in lowercase, e.g. `sse`, `sse2`, `ssse3`, etc. This shows you which instruction sets are available on tge computer you are using. (If you see `ssse3` in the `/proc/cpuinfo` info then that means that you have both SSE3 and SSSE3, for some reason `sse3` is not shown separately.)

2. In the `Task-1` directory you can find a load-store example using unaligned accesses. Here is given example for the `char` type. In the code we use SSE3

instructions. Since registers are 128 bits and the `char` type is 8 bits, then the length of the vector is 16 elements. Assume that the length of the array is a multiple of the vector size. Study and run the code.

The use of vector operations on smaller data elements is more advantageous. Try to run code for different data types (char, short int, long int, long long) and measure the time. Note that you should also change the size of the vector.

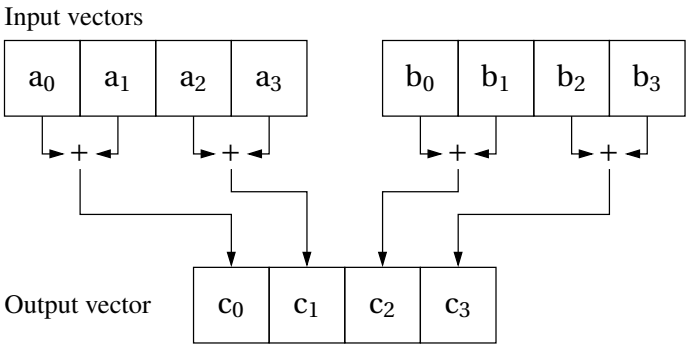
3. Rewrite the `copy_vect` function using AVX intrinsic load and store functions. Note: the lab computers do not support the AVX instruction set extension, so you will not be able to run a program using AVX there. What happens if you try to run the program? To test it, you can copy the program to e.g. `vitsippa.it.uu.se` or `tussilago.it.uu.se` and run it there (using scp and ssh). Does it work?

3.4 Arithmetic operations

All of the common arithmetic operations are available in SSE, see Table 4. Addition and subtraction are available for all vector types. The vector multiplication and division are available for both single and double precision floating-point types, SSE3 implements multiplication for signed 32-bits integers. Since SSE4.1 the multiplication of unsigned 32-bit integers is possible. There are a few instructions which operate with 16 and 32-bit integers, but store just the lower or upper part of the result (check for example the function `_mm_mullo_epi16`). There are no instructions for integer division available.

A special *horizontal add* operation is available to add pairs of values in its input vectors, see Figure 1. 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.

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



Task-2

In Task-2 you need to sum the elements of four vectors with single-precision (32-bit) floating-point elements and store each vector's sum as one element in a destination vector. It can be done using three calls of the horisontal add function `_mm_hadd_ps`.

There is an instruction to calculate the scalar product (dot product) between two vectors. This instruction takes three operands, the two vectors and an 8-bit flag field.

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|epi32|epu32)(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.

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 $\mathrm{F4}_{16}$ (conveniently written as `0xF4` in C code).

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.

3.5 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.

Task-3
In the Task-3 you need to write a simple threshold function. Values larger than the threshold value (4242) should be set to -1 and values smaller than or equal to the threshold should be set to 0. To achieve this, first load data into vector registers, then do the comparison, then store the result back.

Extracting one element: 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

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)	$c_i = a$

Table 5: Intrinsic functions for loading constants into SSE registers. Most of the operations expand into multiple assembler instructions.

operations. Check for example the function _mm_cvtss_f32.

3.6 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 6. Remember that data allocated using _mm_malloc must be freed using _mm_free.

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 6: Memory allocation

Listing 1: Aligning static data using attributes

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

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 1 for an example.

The preferable alignment for 128-bit vectors is 16 and for 256-bit vectors is 32. In general, use AVX or later instruction set if possible. The AVX have very few restrictions on alignment.

4 Multiplying a matrix and a vector

Multiplying a matrix and a vector can be accomplished by the code in Listing 2, 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 3.

Task-4
Implement your version of the matrix-vector multiplication using intrinsics in the matvec_sse() function. Run your code and make sure that it produces the correct result. Is it faster than the traditional version of the code?

Listing 2: 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 3: Matrix-vector multiplication, unrolled by four

```
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 Auto-vectorization using gcc

Modern compilers can try to automatically apply vector instructions where possible. For gcc, the flag to enable auto-vectorization is `-ftree-vectorize`. However, this process is often hindered by the way code is written. The first step in ensuring that auto-vectorization is doing what is possible is to ask the compiler to tell us about what it is trying to do. This is done with by giving gcc the flag `-ftree-vectorizer-verbose=2`. You can set this flag up to 7, with more information being displayed for each level, see "`man gcc`" for details.

For more recent gcc versions, you may need to use the option `-fopt-info-vec-missed` instead to get information about missed vectorization optimization opportunities. The `-fopt-info-vec` option gives information about vectorization optimizations that were done.

See also the GCC online documentation about auto-vectorization in GCC, in particular the part "Using the Vectorizer":

https://gcc.gnu.org/projects/tree-ssa/vectorization.html[4]

Once you see which loops that gcc can or cannot auto-vectorize, you can begin to make changes in the program to try to improve auto-vectorization.

Task-5

In the Task-5 you will again work with matrix-vector multiplication. Edit `Makefile`

[4] `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`

to enable auto-vectorization and output vectorization results for the matrix-vector multiplication program. Does the compiler autovectorize the code?

If the compiler does not autovectorize the code, try to edit the function `matvec_autovec` such that compiler will be able to vectorize the code. Does it work? What is the speedup?

Hint 1: for autovectorization is preferable to have *independent* loop iterations. When the inner loop (over `j`) contains `vec_c[i] += ...` that may be an obstacle since values are added to `vec_c[i]` in each iteration (all inner loop iterations add to the same `vec_c[i]`).

Hint 2: the compiler can use different instruction set extensions for autovectorization, the effect is likely to be larger if a more advanced instruction set extension is used. If the CPU you are running on supports AVX or even AVX2, try the `-march=native` or `-mavx` or `-mavx2` compiler flags. Does that improve performance?

Hint 3: since using vector operations often requires reordering the computations somewhat, the compiler may be able to do a better job if strict adherence to floating-point standards is not required. Therefore, autovectorization often works better when the `-ffast-math` option is used. Note however that the result will probably be slightly different then due to rounding errors. (The `-ffast-math` option means that `-funsafe-math-optimizations` is set, see "`man gcc`" for details.)

Task-6

In the Task-6 you will auto-vectorize the matrix-matrix multiplication. Edit `Makefile` to enable auto-vectorization and output vectorization results for the matrix-matrix multiplication program. What is the speedup?

(The hints from the previous task apply here also.)

Extra part: in general we can expect greater speedup from vectorization the more expensive the performed operations are. Since floating-point division is expensive, there should in principle be much to gain by vectorizing such operations. Write a small program that demonstrates how vectorization can be used for floating-point division. If you want, you can use code from the previous task(s) here as a starting point. What speedup can you achieve?