HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2019
LAB 4: MEMORY SPACE, COMPILER OPTIMIZATION, SIMD AND VECTORIZATION

In the first part of this lab we are going to work on conserving the memory footprint of programs and examine compiler optimization in a bit more detail. We will look at the size (memory space used) for structs and union datatypes in C (extra task), and what can be done to reduce the size.

There is one task showing usage of the `gprof` profiler. Profiling is a process of looking for slow parts of the program and optimizing them.

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

There is an extra task in the lab. Look at it if you are done with other tasks and have more time. If you need all your time for the non-extra tasks, then don't worry about the extra task.

Note. You are not required to write makefiles in order to compile your code.

Log into a system and download the lab tar-ball `Lab04_MemSpace_SIMD_Vec.tar.gz` from the Student Portal. Save it and unpack.

Part 1. Memory space, compiler optimizations and profiling

1. Memory space and compiler optimizations

Task 1:

The code for this task is in the `Task-1` directory.

This task is about *structure packing*, or minimizing the size of structs. When variables are stored in memory, they are always stored at an address that is a multiple of the size of the variable. This is called *natural alignment*. For example, a pointer on a 64-bit machine will be 8-byte aligned and can be stored in addresses ending in `0x0` or `0x8`.

Structs are stored in a similar way. Struct members are naturally aligned, and invisible "padding" bytes are inserted into the struct by the compiler to ensure this alignment. This means that the size of a struct is not necessarily equal to the sum of the size of each member; the struct size may be larger due to padding bytes. Moreover, the natural alignment of a struct is the same as the largest natural

Date: February 14, 2019.

alignment of its members. For example, a struct containing a pointer and a char (9 bytes of data) will be 8-byte aligned.

You'll find four structs declared in `structs.c`. Without running anything, write down the size of each struct. Then execute the code to check your answers.

Gcc provides a way of removing the padding automatically. Add `__attribute__((__packed__))` to the structure declarations to see the packed size. Pay extra attention to `foo3`!

The compiler expects the attribute to be placed after the closing bracket of the struct. Example:

```
struct xy {
  char x;
  int y;
};
struct xy_packed {
  char x;
  int y;
} __attribute__((__packed__)) ;
```

The downside of using the packed attribute is that accessing the data will be slower (requiring extra instructions). Reordering the members of a struct so that large members come first will remove padding between members, although any padding in the end will remain.

Adjust the order of members in `foo3` and `foo4` and remove the packed attribute.

The `Task-1` directory also contains a small test program called `more_structs.c`. Look at that code and try to understand what it is doing.

The idea with `more_structs.c` is to explicitly look at how data in a struct is ordered and how padding bytes are added. A char buffer is filled with zeros, then that memory is used as a struct of the type C. The struct is filled with some data.

Run the code and look at the output, and make sure you understand what is happening and why the output looks the way it does.

Task 2:

As you already know, the compiler is the best optimisation tool you have. This task will explore the GCC optimization options in more detail.

The main optimization flag is `-O` or (equivalently) `-O1`. With this setting, the compiler tries to reduce code size and execution time without performing any optimizations that take a great deal of compilation time.

With `-O2`, every optimization is turned on that does not result in a bigger executable. This option exists because the size of the executable affects the efficiency of the instruction cache.

Option `-O3` turns on additional optimization options that may increase the executable size or can take a very long time to compile.

Option `-Os` is like `-O2` but includes additional options that can shrink the executable size.

The `-Ofast` option is like `-O3`, but disregards strict IEEE standards for floating point math (it turns on `-ffast-math`).

(On some systems, `-O4` performs optimization at link time, enabling optimization across compilation units.)

This is only the beginning, however, because GCC will by default assume very little about the processor architecture and even the instruction set. By specifying the CPU architecture with the `-march` option, you let GCC use a wider, architecture-dependent instruction set. You can compile to any one of a number of target architectures, but you usually want to use `-march=native`, which automatically selects the architecture to match your system. The `-mtune` option tells GCC to just tune the compilation to suit a particular machine, while still using a smaller and more generic instruction set. `-march` implies the same tuning steps as `-mtune`.

Important: if you specify `-march`, the binary output will be less portable to other machines. If you want to use an executable or library on another machine, use `-mtune` instead.

In the `Task-2` directory, you'll find the code for a multigrid solver, a makefile, and a data file. The makefile will attempt to compile ten different versions of the solver as well as output the assembly for each. Your task is to evaluate the performance of each executable. Be sure to note the size as well.

Task 3: (EXTRA TASK)

The code for this task is in the `Task-3` directory.

Bitfields provide a way to declare structure fields smaller than one byte. In C, this is done with the following syntax:

```
struct {
   type member_name : width;
};
```

The `type` of the field determines how the bitfield's value is interpreted. You'll usually want `unsigned int`, but `int` is also allowed. `member_name` is the name of the field, as usual. The `width` is the length of the bitfield in bits and can be in the range `[1,sizeof(type)*8]`. You can combine bitfields with ordinary types in a struct.

In the `Task-3` directory, you will find a program that "reads in" information about a series of button presses and stores them in an array. Then the information is copied to another array, using a different kind of struct.

Look at the two struct types that are declared in `buttonread.h` and write down the sizes you expect them to have. Then run the program and look at the sizes printed, to check if you were correct.

In this case, we know that the numbers stored are small, in the range 0-3 for "leftright" and "updown", and in the range 0-7 for "held". Therefore it should be possible to use the datatype "unsigned char" instead of "unsigned int". Change this in both structs, and then run the program again. What are the sizes of the

structs now? Except for the sizes, the program should work in the same way as before.

Look in `main.c` where there is a loop copying values from `presses_org` to `presses_2`. As you can see, there is an outer loop repeating this 100000 times, just to give us a larger time to measure. Inside the inner loop, values are copied into the `presses_2` array. The bitfield accesses there may be a little slow. It can be faster to compose a bitfield by using `<<` and `|` (left shift and bitwise OR) than writing to the field members individually.

C will not let us assign a char directly to a `buttonpress_2` type, so we create a `char*` pointer that points to the same address as `presses_2[i]` — as you can see, a line setting such a pointer `p` already exists in the code. Now, your job is to do some left shift and bitwise OR operations to achieve the same effect as using the bitfield accesses. Remember to check that the result is still correct! Are you able to make it work? Does the code run faster after this change?

If you need a hint, consult Section 7.27 in the book by Agner Fog.

Unions

A *union* is a structure where members share the same memory space. This can be used to access the same data in different ways without pointer casting, but it can also be used to save memory space. *Caveats:* Because the use of unions prevents the use of register variables, it is not recommended to put simple variables into a union. Beware also that the compiler does not know if you use the union improperly (e.g. accidentally corrupt the data in one member by using the other).

A union is declared simply in a way that is analogous to a struct:

```
typedef union {
   float lookThisIsAFloat;
   int lookThisIsAnInt;
} myUnionType;

myUnionType foo;
foo.lookThisIsAFloat = 1.2345;  // valid!
foo.lookThisIsAnInt = 300;      // works!
```

It is possible to create a union of a struct and e.g. an int, for example:

```
 typedef union {
   struct {
     int a:23;
     int b:5;
   };
   int ab;
 } abType;
```

Note that the struct with members `a` and `b` then shares the same memory space as the int `ab`, so modifying `ab` can change `a` and/or `b`, and vice versa.

One example of how a union datatype can be used is given in the `Task-3/union_test.c` program. There, a union is used as a convenient way to extract and/or manipulate

the individual bits of a `char`. Look at the code to figure out what it is doing, then try running it and modifying it to see the bit values for different `char` values.

Some other examples of using bitfields and unions can be found in section 14.9 in the book by Agner Fog. There, the sign bit, exponent, and mantissa parts of a floating-point number are extracted using bitfields.

2. Profiling using gprof

*Note:*If you're using your own Mac, it may happen that `gprof` is not available. It should be possible to install them, but it may be easier for you to use the Linux lab computers. Alternatively, you can use ssh to login to one of the university's Linux servers, see the list of available Linux hosts here: http://www.it.uu.se/datordrift/maskinpark/linux.

Task 4:

The code for this task is in the `Task-4` directory.

This task is about profiling using `gprof`.

Note about `gprof` and compiler optimization flags: Unfortunately, when some more advanced compiler optimization are turned on that often leads to less information from `gprof`. For this reason, first do this task using only the `-O1` compiler optimization flag, to see what information `gprof` gives then. Then you can try also with other optimization flags.

For a description of `gprof` and its options type

```
man gprof
```

In order to profile with `gprof` add the flag `-pg` when compiling:

```
gcc -O1 -g -pg matmul.c
```

Run the program for matrix with size 900

```
./a.out
```

Then list all the files in the current directory using `ls -l`:

```
ls -l
```

Note the new file gmon.out where profiling data has been created. Note that the profiler shows results corresponding to the actual run of the program. If in this run some function was not called, then this function will not be included in the analysis. To include all the functions, even those that were never called, use the `-z` option.

Now run the profiler and examine the output:

```
gprof a.out gmon.out
```

The output information is divided into two parts. The *flat profile* shows how much time your program spent in each function, and how many times that function was called. The *call graph* shows relations between functions. Which functions called

the current function, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function.

Rerun the program and compare results of gprof for different matrix sizes.

- What are the most time consuming operations?
- Which loop ordering is more efficient?
- What is the most called function?
- Does the profiler output time include time while waiting for the user input?
- Which metric is used to order function in the call graph?
- What are the self and children columns representing?

Note that the `-pg` compiler option that is needed for `gprof` profiling means that extra instrumentation code is added in the program, and the performance may be affected by this; your program will probably be slower when compiled with the `-pg` option.

Part 2. SIMD and Vectorization

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 http://www.intel.com/products/processor/manuals/Intel and https://developer.amd.com/resources/developer-guides-manuals/AMD 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/
It can be very useful for looking up which intrinsic functions exist, and precisely what each function does.

3. 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.

4. 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.

4.1. 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 4.1. 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 4.2. 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 4.2.

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 4.1. 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 4.2. 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`.

4.2. **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.

4.3. **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 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

	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 4.3. 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.

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 4.3 for a list of load and store intrinsics and their corresponding assembler instructions.

Note that constants should usually not be loaded using these instructions.

Task 5:

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 4.1 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.)

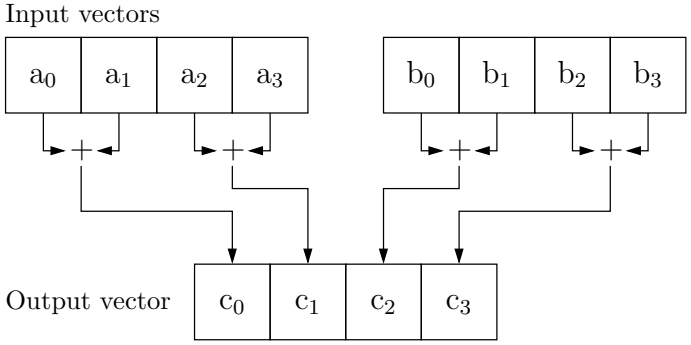
Task 6:

In the `Task-6` 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.

4.4. **Arithmetic operations.** All of the common arithmetic operations are available in SSE, see Table 4.4. 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 4.1. 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 4.1. Calculating c = `_mm_hadd_ps(`a`,` b`)`



Task 7:

In `Task-7` 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. 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 $F4_{16}$ (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

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 4.1
`_mm_dp_(ps|pd)(`a`, `b`, FLAGS)`	$\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ (dot product)
`_MM_TRANSPOSE4_PS(`a`, ..., `d`)`	Transpose the matrix $(a^{\mathrm{t}} \dots d^{\mathrm{t}})$ 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 4.4. Arithmetic operations available in SSE. The transpose operation is a macro that expands to several SSE instructions to efficiently transpose a matrix.

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.

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

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 8:

In the `Task-8` you will work with matrix-vector multiplication. Edit `Makefile` 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.)

Task 9:

In the `Task-9` 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.)