VCL
C++ vector class library
manual

Agner Fog

Contents

Chapter 1

Introduction

The VCL vector class library is a tool that helps C++ programmers make their code much faster by handling multiple data in parallel. Modern CPU's have *Single Instruction Multiple Data* (SIMD) instructions for handling vectors of multiple data elements in parallel. The compiler may be able to use SIMD instructions automatically in simple cases, but a human programmer is often able to do it better by organizing data into vectors that fit the SIMD instructions. The VCL library is a tool that makes it easier for the programmer to write vector code without having to use assembly language or intrinsic functions. Let us explain this with an example:

Example 1.1.

```
// Array loop
float a[8], b[8], c[8];          // declare arrays
...                              // put values into arrays
for (int i = 0; i < 8; i++) {    // loop for 8 elements
    c[i] = a[i] + b[i] * 1.5f; // operations on each element
}
```

The vector class library allows you to rewrite example 1.1 using vectors:

Example 1.2.

```
// Array loop using vectors
#include "vectorclass.h"         // use vector class library
float a[8], b[8], c[8];          // declare arrays
...                              // put values into arrays
Vec8f avec, bvec, cvec;          // define vectors of 8 floats each
avec.load(a);                    // load array a into vector
bvec.load(b);                    // load array b into vector
cvec = avec + bvec * 1.5f;       // do operations on vectors
cvec.store(c);                   // save result in array c
```

Example 1.2 does the same as example 1.1, but more efficiently because it utilizes SIMD instructions that do eight additions and/or eight multiplications in a single instruction. Modern microprocessors have these instructions which may give you a throughput of eight floating point additions and eight multiplications per clock cycle. A good optimizing compiler may actually convert example 1.1 automatically to use the SIMD instructions, but in more complicated cases you cannot be sure that the compiler is able to vectorize your code in an optimal way.

1.1 How it works

The type $\mathrm{Vec8f}$ in example 1.2 is a class that encapsulates the intrinsic type $\_\_\mathrm{m256}$ which represents a 256-bit vector register holding 8 floating point numbers of 32 bits each. The overloaded operators + and * represent the SIMD instructions for adding and multiplying vectors. These operators are inlined so that no extra code is generated other than the SIMD instructions. All you have to do to get access to these vector operations is to include "vectorclass.h" in your C++ code and specify the desired instruction set (e.g. SSE2, AVX2, or AVX512) in the compiler options.

The code in example 1.2 can be reduced to just 4 machine instructions if the instruction set AVX or higher is enabled. The SSE2 instruction set will give 8 machine instructions because the maximum vector register size is only half as big for instruction sets prior to AVX. The code in example 1.1 will generate approximately 44 instructions if the compiler does not automatically vectorize the code.

1.2 Features of VCL

- Vectors of 8-, 16-, 32- and 64-bit integers, signed and unsigned
- Vectors of half precision, single precision, and double precision floating point numbers
- Total vector size 128, 256, or 512 bits
- Defines almost all common operators
- Boolean operations and branches on vector elements
- Many arithmetic functions
- Standard mathematical functions
- Permute, blend, gather, scatter, and table look-up functions
- Fast integer division
- Can build code for different instruction set extensions from the same source code
- CPU dispatching to utilize higher instruction sets when available
- Uses metaprogramming to find the optimal implementation for the selected instruction set and parameter values of a given operator or function
- Includes extra add-on packages for special purposes and applications

1.3 Instruction sets supported

Since 1997, every new CPU model has extended the x86 instruction set with more SIMD instructions. The VCL library requires the SSE2 instruction set as a minimum, and supports SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, XOP, FMA3, FMA4, and AVX512F/VL/BW/DQ, as well as the new AVX512VBMI/VBMI2 and AVX512-FP16.

1.4 Platforms supported

VCL has support for Windows, Linux, and Mac, 32-bit and 64-bit, with Intel, AMD, or VIA x86 or x86-64 instruction set processors.

There are no plans to support ARM or other instruction sets in VCL. If you need other platforms than x86 and x86-64 then you may use the cross-platform function wrapper library named Highway. You can find it at https://github.com/google/highway.

A special version of the vector class library for the (now obsolete) Intel Knights Corner coprocessor has been developed at CERN. It is available from https://bitbucket.org/veclibknc/vclknc.git or https://bitbucket.org/edanor/umesimd/

1.5 Compilers supported

The vector class library could not have been made with any other programming language than C++, because only C++ combines all the necessary features: low-level programming such as bit manipulation and intrinsic functions, high-level programming features such as classes and templates, operator overloading, metaprogramming, compiling to machine code without any intermediate byte code, and highly optimizing compilers with support for the many different instruction sets and platforms.

The vector class library works with Gnu, Clang, Microsoft, and Intel C++ compilers. It is recommended to use the newest version of the compiler if the newest instruction sets are used. The best optimization is obtained with the Gnu and Clang compilers. You may use any integrated development environment, make utility, or build system.

The vector class library version 2.xx requires the C++17 or later standard for the C++ language. The vector class library version 1.xx, using standard C++0x, should only be used if it is not possible to use a compiler with C++17 support.

1.6 Intended use

This vector class library is intended for experienced C++ programmers. It is useful for improving code performance where speed is critical and where the compiler is unable to vectorize the code automatically in an optimal way. Combining explicit vectorization by the programmer with other kinds of optimization done by the compiler, it has the potential for generating highly efficient code. This can be useful for optimizing library functions and critical innermost loops (hotspots) in CPU-intensive programs. There is no reason to use it in less critical parts of a program.

1.7 How VCL uses metaprogramming

The vector class library uses metaprogramming extensively to resolve as much work as possible at compile time rather than at run time. Especially, it uses metaprogramming to find the optimal instructions and algorithms, depending on constants in the code and the selected instruction set.

VCL version 1.xx is written for older versions of the C++ language that does not have very good metaprogramming features, but the VCL makes the best use of the available features such as preprocessing directives and templates. Furthermore, it relies extensively on optimizing compilers for doing calculations with constant inputs at compile time and for removing not-taken branches.

VCL version 2.xx is taking advantage of constexpr branches, constexpr functions, and other advanced features in C++14 and C++17 for explicitly telling the compiler what calculations to do at

compile time, and to remove not-taken branches. This makes the code clearer and more efficient. It is recommended to use the latest version of VCL, if possible.

The following cases illustrate the use of metaprogramming in VCL:

- Compiling for different instruction sets. If you are using a bigger vector size than supported by the instruction set, then the VCL code will split the big vector into multiple smaller vectors. If you compile the same code again for a higher instruction set, then you will get a more efficient program with full-size vector registers.
- Permute, blend, and gather functions. There are many different machine instructions that move data between different vector elements. Some of these instructions can only do very specific data permutations. The VCL uses quite a lot of metaprogramming to find the instruction or sequence of instructions that best fits the specified permutation pattern. Often, the higher instruction sets give more efficient results.
- Integer division. Integer division can be done faster by a combination of multiplication and bit-shifting. The VCL can use metaprogramming to find the optimal division method and calculate the multiplication factor and shift count at compile time if the divisor is a known constant. See page 5 for details.
- Raising to a power. Calculating x^8 can be done faster by squaring x three times rather than by a loop that multiplies seven times. The VCL can determine the optimal way of raising floating point vectors to an integer or rational power in the functions $\mathrm{pow\_const}$ and $\mathrm{pow\_rational}$.

1.8 Availability

The newest version of the vector class library is available from github.com/vectorclass

1.9 Support

The vector class library is not a commercial product, but free and open source. You cannot expect the kind of support you would get with a paid product.

A discussion board for software developed by Agner Fog is currently provided at www.agner.org/forum/viewforum.php?f=1. This is intended for general discussion and suggestions, but not for programming support.

Programming questions should preferably be asked at Stackoverflow.com using the tag **vector-class-library**.

1.10 License

I have previously sold commercial licenses to VCL. Now, I have decided for a more permissive license. Instead of selling commercial licenses, I am now suggesting that commercial users make a donation to an open source project of your own choosing or to an organization promoting open source software.

Chapter 2

The basics

2.1 How to compile

Copy the latest version of the header files (*.h) to the same folder as your C++ source files. The header files from any add-on package should be included too if needed. Alternatively, you may put the VCL header files in a separate folder and specify an include path to this folder.

Include the header file vectorclass.h in your C++ source file. Several other header files will be included automatically.

Set your compiler options to the desired instruction set. It is recommended to compile for 64-bit mode. Use C++ version 17 or higher. The instruction set must be at least SSE2. See table 9.2 on page 81 for a list of compiler options.

A command line for the Clang compiler may, for example, look like this:
clang++ -m64 -O2 -std=c++17 -mfma -mavx2 -fabi-version=0 myprogram.cpp

You may compile multiple versions for different instruction sets as explained in chapter 9.9.

The following simple C++ example may help you get started:

Example 2.1.

```
// Simple vector class example C++ file
#include <stdio.h>
#include "vectorclass.h"

int main() {
    // define and initialize integer vectors a and b
    Vec4i a(10,11,12,13);
    Vec4i b(20,21,22,23);

    // add the two vectors
    Vec4i c = a + b;

    // Print the results
    for (int i = 0; i < c.size(); i++) {
        printf(" %5i", c[i]);
    }
    printf("\n");

    return 0;
}
```

2.2 Overview of vector classes

The vector class library supports vectors of 8-bit, 16-bit, 32-bit and 64-bit signed and unsigned integers, 32-bit single precision floating point numbers, and 64-bit double precision floating point numbers. See page 9 for optional support for 16-bit half precision floating point numbers.

A vector contains multiple elements of the same type to a total size of 128, 256 or 512 bits. The vector elements are indexed, starting at 0 for the first element.

The constant MAX_VECTOR_SIZE indicates the maximum vector size. The default maximum vector size is 512 in the current version and possibly larger in future versions. You can disable 512-bit vectors by defining

```
#define MAX_VECTOR_SIZE 256
```

before including the vector class header files.

The vector class library also defines boolean vectors. These are mainly used for conditionally selecting elements from vectors.

The following vector classes are defined:

Table 2.1: Integer vector classes

Vector class	**Integer size bits**	**Signed**	**Elements per vector**	**Total bits**	**Minimum recommended instruction set**
Vec16c	8	signed	16	128	SSE2
Vec16uc	8	unsigned	16	128	SSE2
Vec8s	16	signed	8	128	SSE2
Vec8us	16	unsigned	8	128	SSE2
Vec4i	32	signed	4	128	SSE2
Vec4ui	32	unsigned	4	128	SSE2
Vec2q	64	signed	2	128	SSE2
Vec2uq	64	unsigned	2	128	SSE2
Vec32c	8	signed	32	256	AVX2
Vec32uc	8	unsigned	32	256	AVX2
Vec16s	16	signed	16	256	AVX2
Vec16us	16	unsigned	16	256	AVX2
Vec8i	32	signed	8	256	AVX2
Vec8ui	32	unsigned	8	256	AVX2
Vec4q	64	signed	4	256	AVX2
Vec4uq	64	unsigned	4	256	AVX2
Vec64c	8	signed	64	512	AVX512BW
Vec64uc	8	unsigned	64	512	AVX512BW
Vec32s	16	signed	32	512	AVX512BW
Vec32us	16	unsigned	32	512	AVX512BW
Vec16i	32	signed	16	512	AVX512
Vec16ui	32	unsigned	16	512	AVX512
Vec8q	64	signed	8	512	AVX512
Vec8uq	64	unsigned	8	512	AVX512

Table 2.2: Floating point vector classes

Vector class	Precision	Elements per vector	Total bits	Minimum recommended instruction set
Vec4f	single	4	128	SSE2
Vec2d	double	2	128	SSE2
Vec8f	single	8	256	AVX
Vec4d	double	4	256	AVX
Vec16f	single	16	512	AVX512
Vec8d	double	8	512	AVX512

Table 2.3: Boolean vector classes

Boolean vector class	For use with	Elements per vector	Total size, bits	Minimum recommended instruction set
Vec16cb	Vec16c, Vec16uc	16	16 or 128	SSE2
Vec8sb	Vec8s, Vec8us	8	8 or 128	SSE2
Vec4ib	Vec4i, Vec4ui	4	8 or 128	SSE2
Vec2qb	Vec2q, Vec2uq	2	8 or 128	SSE2
Vec32cb	Vec32c, Vec32uc	32	32 or 256	AVX2
Vec16sb	Vec16s, Vec16us	16	16 or 256	AVX2
Vec8ib	Vec8i, Vec8ui	8	8 or 256	AVX2
Vec4qb	Vec4q, Vec4uq	4	8 or 256	AVX2
Vec64cb	Vec64c, Vec64uc	64	64 or 512	AVX512BW
Vec32sb	Vec32s, Vec32us	32	32 or 512	AVX512BW
Vec16ib	Vec16i, Vec16ui	16	16 or 512	AVX512
Vec8qb	Vec8q, Vec8uq	8	8 or 512	AVX512
Vec4fb	Vec4f	4	8 or 128	SSE2
Vec2db	Vec2d	2	8 or 128	SSE2
Vec8fb	Vec8f	8	8 or 256	SSE2
Vec4db	Vec4d	4	8 or 256	SSE2
Vec16fb	Vec16f	16	16	AVX512
Vec8db	Vec8d	8	8	AVX512

The size of the boolean vectors depends on the instruction set (see page 32).

2.3 Half precision floating point vectors

Half precision floating point numbers are represented by 16 bits (one sign bit, five exponent bits, and ten bits for the mantissa). Half precision is useful for sound, video, and artificial intelligence applications where the low precision is acceptable. A 512-bit vector register can contain 32 half-precision numbers and do 32 arithmetic operations simultaneously with a single instruction.

Microprocessors have various levels of support for half precision. The F16C instruction set extension supports conversion between half precision and single precision, but not arithmetic operations on half precision numbers. F16C has been included in Intel and AMD processors since 2013-2014. The newer AVX512-FP16 instruction set extension implements a full set of arithmetic operations on half precision numbers. The AVX512-FP16 instruction set will be supported by Intel's Sapphire Rapids processor, expected to be introduced in late 2022.

The Vector Class Library supports half precision floating point vectors when the following header file is included:

```
#include "vectorfp16.h"
```

The performance of half-precision vector calculations is highly dependent on the instruction set. Full performance is obtained only when the AVX512-FP16 instruction set is supported by the microprocessor and enabled in the compiler options.

The earlier F16C instruction set allows efficient conversion between single precision and half precision, but not arithmetic operations on half precision vectors. With F16C, arithmetic operations are emulated by converting the operands from half precision to single precision and converting each result back to half precision. This will be inefficient because intermediate results are converted back and forth, as illustrated in this example:

```
#include "vectorfp16.h"
Vec8h a, b, c, d;    // vectors of eight half precision numbers
d = a + b + c;       // calculate sums
```

If this example is compiled with F16C, but not AVX512-FP16, then the code will convert a and b from half precision to single precision, calculate a+b with single precision, convert a+b back to half precision, then convert a+b to single precision again, convert c to single precision, do the next addition with single precision, and convert the final sum a+b+c back to half precision. This is of course not efficient. It is more efficient to do all the intermediate calculations with single precision:

```
#include "vectorfp16.h"
Vec8h a, b, c, d;          // half precision vectors
Vec8f aa = to_float(a);    // convert to single precision
Vec8f bb = to_float(b);
Vec8f cc = to_float(c);
Vec8f dd = aa + bb + cc;   // do the calculations
d = to_float16(dd);        // convert the result to half precision
```

The ability to emulate half precision calculations as illustrated in the first example is useful for verifying half-precision code. This allows you to test whether half precision is sufficient for a particular task even when you do not have access to a computer with AVX512-FP16. If the goal is to get maximum performance then you should use half precision only on microprocessors with AVX512-FP16, but use single precision on microprocessors without AVX512-FP16.

The half precision code can run even on microprocessors without F16C, but this will be extremely slow because every conversion between single and half precision requires a long sequence of instructions. Therefore, it is important to enable F16C in the compiler when it is present. F16C is supported by some AMD and Intel processors that have AVX and all currently known processors that have AVX2 and later instruction sets. It may be useful to make a version of the code that uses conversion between single and half precision for processors that have both AVX2 and F16C, and a more efficient version that does calculations with half precision for processors that have AVX512-FP16. The functions $\mathrm{hasF16C}()$ and $\mathrm{hasAVX512FP16}()$ in instrset_detect.cpp can be used for detecting microprocessor support for these instruction sets.

Compiler support

The AVX512-FP16 instruction set is supported by the following compilers and later versions:
g++ version 12.1.0 with binutils 2.38

clang++ version 14.0.0
Intel c++ compiler version 2021.2

The proper type for half precision scalars is _Float16. This type is supported by the g++ and some Intel compilers. It is supported on the Clang compiler only when AVX512-FP16 is enabled. The vector class library will emulate a type named Float16 if _Float16 is not supported by the compiler. This emulation includes only the most basic operations and operators on half precision floating point scalars, such as $+ - * /$ and conversion to and from float. Other operators and functions on Float16 are not emulated. Float16 is defined as _Float16 whenever _Float16 is supported by the compiler.

Do not use the types __fp16 and __bf16 that are available on some compilers. __fp16 is an interchange format, not an arithmetic format. This means that variables of type __fp16 will be immediately converted to float before any operation. __bf16 is an incompatible format available on some systems. __bf16 has 8 exponent bits and 7 mantissa bits where _Float16 and __fp16 have 5 exponent bits and 10 mantissa bits.

Half precision vector classes

Table 2.4: Half precision floating point vector classes

Vector class	Precision	Elements per vector	Total bits	Minimum recommended instruction set
Vec8h	half	8	128	AVX512-FP16
Vec16h	half	16	256	AVX512-FP16
Vec32h	half	32	512	AVX512-FP16

The corresponding boolean vector classes are Vec8hb, Vec16hb, and Vec32hb.

Subnormal numbers are supported for these vector classes regardless of the floating point control word. The floating point control word (see page 55) has no effect on half precision subnormal numbers.

Functions and operators

The half precision vectors can be used with the same operators and general functions as single and double precision vectors. Some mathematical functions are supported for half precision, including exponential and trigonometric functions.

Complex number algebra and functions with half precision are supported. See complexvec_manual.pdf at github.

The following functions are available for conversion between single precision and half precision:

Table 2.5: Conversion between single and half precision

Function	Conversion	Comment
convert8h_4f	Vec8h -> Vec4f	only lower half is converted
to_float	Vec8h -> Vec8f	
to_float	Vec16h -> Vec16f	
convert4f_8h	Vec4f -> Vec8h	upper half is zero
to_float16	Vec8f -> Vec8h	
to_float16	Vec16f -> Vec16h	

Vec32h cannot be converted directly to and from single precision because there is no Vec32f. Conversion to and from Vec32h can be coded as follows:

```
#include "vectorfp16.h"
Vec16f a, b;                    // single precision vectors
Vec32h h;                       // half precision vector
// conversion from single to half precision:
h = Vec32h(to_float16(a), to_float16(b));
// conversion from half to single precision:
a = to_float(h.get_low());  // lower half
b = to_float(h.get_high()); // upper half
```

2.4 Constructing vectors and loading data into vectors

There are many ways to create vectors and put data into vectors. These methods are listed here.

Method	default constructor
Defined for	all vector classes
Description	the vector is created but not initialized. The value is unpredictable
Efficiency	good

```
// Example:
Vec4i a;     // creates a vector of 4 signed integers
```

Method	constructor with one parameter
Defined for	all vector classes
Description	all elements get the same value
Efficiency	good

```
// Example:
Vec4i a(5);    // all four elements = 5
```

Method	assignment to scalar
Defined for	all vector classes
Description	all elements get the same value
Efficiency	good

```
// Example:
Vec4i a = 6;  // all four elements = 6
```

Method	constructor with one parameter for each vector element
Defined for	all integer and floating point vector classes
Description	each element gets a specified value. The parameter for element number 0 comes first.
Efficiency	good for constant. Medium for variables as parameters

```
// Examples:
Vec4i a(10,11,12,13);           // a = (10,11,12,13)
Vec4i b = Vec4i(20,21,22,23); // b = (20,21,22,23)
```

Method	constructor with one parameter for each half vector
Defined for	all vector classes if a similar class of half the size exists
Description	Concatenates two 128-bit vectors into one 256-bit vector. Concatenates two 256-bit vectors into one 512-bit vector
Efficiency	good

```
// Example:
Vec4i a(10,11,12,13);
Vec4i b(20,21,22,23);
Vec8i c(a, b);      // c = (10,11,12,13,20,21,22,23)
```

Method	insert(index, value)
Defined for	all vector classes
Description	changes the value of element number (index) to (value). The index starts at 0
Efficiency	good if AVX512VL, medium otherwise

```
// Example:
Vec4i a(0);
a.insert(2, 9);   // a = (0,0,9,0)
```

Method	load(const pointer)
Defined for	all integer and floating point vector classes
Description	loads all elements from an array
Efficiency	good, except immediately after inserting elements one by one into the array

```
// Example:
int list[8] = {10,11,12,13,14,15,16,17};
Vec4i a, b;
a.load(list);    // a = (10,11,12,13)
b.load(list+4); // b = (14,15,16,17)
```

Method	load_a(const pointer)
Defined for	all integer and floating point vector classes
Description	loads all elements from an aligned array
Efficiency	good, except immediately after inserting elements separately into the array.

This method does the same as the load method (see above), but requires that the pointer points to an address divisible by 16 for 128-bit vectors, by 32 for 256-bit vectors, or by 64 for 512 bit vectors. If you are not certain that the array is properly aligned then use load instead of load_a. There is hardly any difference in efficiency between load and load_a on newer microprocessors.

Method	load_partial(int n, const pointer)
Defined for	all integer and floating point vector classes
Description	loads n elements from an array into a vector. Sets remaining elements to 0. $0 \leq n \leq$ (vector size).
Efficiency	good if AVX512VL, medium otherwise

```
// Example:
float list[3] = {1.0f, 1.1f, 1.2f};
Vec4f a;
a.load_partial(2, list);   // a = (1.0, 1.1, 0.0, 0.0)
```

Method	cutoff(int n)
Defined for	all integer and floating point vector classes
Description	leaves the first n elements unchanged and sets the remaining elements to zero. $0 \leq n \leq$ (vector size).
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
a.cutoff(2);                    // a = (10, 11, 0, 0)
```

Method	gather<indexes>(array)
Defined for	floating point vector classes and integer vector classes with 32-bit and 64-bit elements
Description	gather non-contiguous data from an array.
Efficiency	medium

```
// Example:
int list[8] = {10,11,12,13,14,15,16,17};
Vec4i a = gather4i<0,2,1,6>(list); // a = (10,12,11,16)
```

2.5 Getting data from vectors

There are many ways to extract elements or parts of a vector. These methods are listed here.

Method	store(pointer)
Defined for	all integer and floating point vector classes
Description	stores all elements into an array
Efficiency	good

```
// Example:
Vec4i a(10,11,12,13);
Vec4i b(20,21,22,23);
int list[8];
a.store(list);
b.store(list+4); // list contains (10,11,12,13,20,21,22,23)
```

Method	store_a(pointer)
Defined for	all integer and floating point vector classes
Description	stores all elements into an aligned array
Efficiency	good

This method does the same as the store method (see above), but requires that the pointer points to an address divisible by 16 for 128-bit vectors, by 32 for 256-bit vectors, or by 64 for 512-bit vectors. If you are not certain that the array is properly aligned then use store instead of store_a. There is hardly any difference in efficiency between store and store_a on newer microprocessors.

Method	store_nt(pointer)
Defined for	all integer and floating point vector classes
Description	stores all elements into an aligned array without caching
Efficiency	recommended only for very large arrays

This method does the same as the store_a method (see above), but without using the cache. This is optimal only for very large arrays when it is unlikely that the data will stay cached until they are read

again. As a rule of thumb, use store_nt for memory blocks bigger than half the size of the last-level cache. You will get a runtime error if the pointer is not properly aligned.

Method	store_partial(int n, pointer)
Defined for	all integer and floating point vector classes
Description	stores the first n elements into an array. The rest of the array is untouched. $0 \leq n \leq$ (vector size)
Efficiency	good if AVX512VL, medium otherwise

```
// Example:
float list[4] = {9.0f, 9.0f, 9.0f, 9.0f};
Vec4f a(1.0f, 1.1f, 1.2f, 1.3f);
a.store_partial(2, list);  // list contains (1.0, 1.1, 9.0, 9.0)
```

Method	extract(index)
Defined for	all integer, floating point and boolean vector classes
Description	gets a single element from a vector
Efficiency	good if AVX512VL, medium otherwise

```
// Example:
Vec4i a(10,11,12,13);
int b = a.extract(2);    // b = 12
```

Method	operator [ ]
Defined for	all integer, floating point and boolean vector classes
Description	gets a single element from a vector
Efficiency	good if AVX512VL, medium otherwise

The operator [ ] does exactly the same as the extract method. Note that you can read a vector element with the [ ] operator, but not write an element.

```
// Example:
Vec4i a(10,11,12,13);
int b = a[2];              // b = 12
a[3] = 5;                  // not allowed!
```

Method	get_low()
Defined for	all vector classes of 256 bits or more
Description	gets the lower half of a 256-bit vector as a 128-bit vector. gets the lower half of a 512-bit vector as a 256-bit vector.
Efficiency	good

```
// Example:
Vec8i a(10,11,12,13,14,15,16,17);
Vec4i b = a.get_low();  // b = (10,11,12,13)
```

Method	get_high()
Defined for	all vector classes of 256 bits or more
Description	gets the upper half of a 256-bit vector as a 128-bit vector. gets the upper half of a 512-bit vector as a 256-bit vector.
Efficiency	good

```
// Example:
Vec8i a(10,11,12,13,14,15,16,17);
Vec4i b = a.get_high();  // b = (14,15,16,17)
```

Method	size()
Defined for	all vector classes
Description	static constant member function indicating the number of elements that the vector can contain
Efficiency	good

```
// Example:
Vec8f a;
int s = a.size();  // s = 8
```

Method	elementtype()
Defined for	all vector classes
Description	static constant member function indicating the type of elements that the vector contains: 1: bits (internal base class) 2: bool (compact) 3: bool (broad) 4: int8_t 5: uint8_t 6: int16_t 7: uint16_t 8: int32_t 9: uint32_t 10: int64_t 11: uint64_t 15: half precision floating point 16: float (single precision floating point) 17: double (double precision floating point)
Efficiency	good

```
// Example:
Vec16s a;
int t = a.elementtype();  // t = 6
```

2.6 Arrays and vectors

Vectors are very useful for array loops with large data sets. The add-on package named 'containers' provides efficient container class templates for implementing arrays with fixed size and dynamic size, as well as matrixes. See containers_manual.pdf for details.

If you are not using the add-on package 'containers' or you are making your own containers then you need to consider the following.

Data arrays may have fixed size or variable size. A fixed size array is particularly efficient if the size is known when the program is compiled, or a reasonable upper limit can be set. For example:

```
int const datasize = 1024;  // size of dataset, constant
float mydata[datasize];     // constant size array
...
Vec8f x;
for (int i = 0; i < datasize; i += 8) {
   x.load(mydata+i);
   x = x * 0.1f + 2.0f;
   x.store(mydata+i);
}
```

If the size of the array is determined at runtime then the most efficient solution is to allocate the array using the operator new:

```
int datasize = 1024;                    // size of dataset, variable
float *mydata = new float[datasize]; // allocate variable size array
...
Vec8f x;
for (int i = 0; i < datasize; i += 8) {
   x.load(mydata+i);
   x = x * 0.1f + 2.0f;
   x.store(mydata+i);
}
...
delete[] mydata; // remember to free the allocated data
```

It is recommended to align an array by the vector size for optimal performance. See page 73 for details.

See page 75 for discussion of the case where the data size is not a multiple of the vector size.

A matrix or multidimensional array can be implemented in various ways. If the length of each row is not more than the vector size, then it is convenient to use one VCL vector for each row. Longer rows can be contained in multiple VCL vectors. If the number of columns is variable then it is recommended to store the rows one after another in a linear array. Use padding space at the end of each row, if necessary, to align the next row by the vector length.

The standard C++ container classes are often inefficient. It is unfortunately common to implement matrixes as nested container classes such as std::vector<std::vector<data_type>>. Such constructs are inefficient and should be avoided.

2.7 Using a namespace

In general, there is no need to put the vector class library into a separate namespace. Therefore, the use of a namespace is optional. You can give the vector class library a namespace, if necessary, by defining VCL_NAMESPACE, for example:

```
#define VCL_NAMESPACE vcl
#include "vectorclass.h"

using namespace vcl;

// your vector code here...
```

Chapter 3

Operators

3.1 Arithmetic operators

Operator	+, ++, +=
Defined for	all integer and floating point vector classes
Description	addition
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = a + b;              // c = (30, 32, 34, 36)
```

Operator	-, –, -=, unary -
Defined for	all integer and floating point vector classes
Description	subtraction
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = a - b;              // c = (-10, -10, -10, -10)
```

Operator	*, *=
Defined for	all integer and floating point vector classes
Description	multiplication
Efficiency	8 bit integers: poor 16 bit integers: good 32 bit integers: good for SSE4.1 and later instruction set, poor otherwise 64 bit integers: good for AVX512DQ instruction set, poor otherwise float: good double: good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = a * b;              // c = (200, 231, 264, 299)
```

Operator	/, /= (floating point)
Defined for	all floating point vector classes
Description	division
Efficiency	medium

```
// Example:
Vec4f a(1.0f, 1.1f, 1.2f, 1.3f);
Vec4f b(2.0f, 2.1f, 2.2f, 2.3f);
Vec4f c = a / b;  // c = (0.500f, 0.524f, 0.545f, 0.565f)
```

Operator	/, /= (integer vector divided by scalar)
Defined for	all classes of 8-bit, 16-bit and 32-bit integers, signed and unsigned. Not available for 64-bit integers
Description	division by scalar. Results are truncated to integer. All elements are divided by the same divisor. See page 22 for explanation
Efficiency	poor

```
// Example:
Vec4i a(10, 11, 12, 13);
int   b = 3;
Vec4i c = a / b;  // c = (3, 3, 4, 4)
```

Operator	/, /= (integer vector divided by constant)
Defined for	all classes of 8-bit, 16-bit and 32-bit integers, signed and unsigned. Not available for 64-bit integers
Description	division by compile-time constant. All elements are divided by the same divisor. See page 22 for explanation
Efficiency	medium (better than division by scalar variable). Good if divisor is a power of 2

```
// Example, signed:
Vec4i  a(10, 11, 12, 13);
Vec4i  b = a / const_int(3);  // b = (3, 3, 4, 4)
// Example, unsigned:
Vec4ui c(10, 11, 12, 13);
Vec4ui d = c / const_uint(3); // d = (3, 3, 4, 4)
```

3.2 Logic operators

Operator	<<, <<=
Defined for	all integer vector classes
Description	bit shift left. All vector elements are shifted by the same amount. Shifting left by n is a fast way of multiplying by 2^n
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b = a << 2;          // b = (40, 44, 48, 52)
```

Operator	>>, >>=
Defined for	all integer vector classes
Description	bit shift right. All vector elements are shifted by the same amount. Unsigned integers use logical shift. Signed integers use arithmetic shift (i.e. the sign bit is copied). Shifting unsigned right by n is a fast way of dividing by 2^n
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b = a >> 2;           // b = (2, 2, 3, 3)
```

Operator	==
Defined for	all vector classes
Description	test if equal. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(10, 11, 12, 13);
Vec4i  b(14, 13, 12, 11);
Vec4ib c = a == b;     // c = (false, false, true, false)
```

Operator	!=
Defined for	all vector classes
Description	test if not equal. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(10, 11, 12, 13);
Vec4i  b(14, 13, 12, 11);
Vec4ib c = a != b;     // c = (true, true, false, true)
```

Operator	>
Defined for	all integer and floating point vector classes
Description	test if bigger. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(10, 11, 12, 13);
Vec4i  b(14, 13, 12, 11);
Vec4ib c = a > b;      // c = (false, false, false, true)
```

Operator	>=
Defined for	all integer and floating point vector classes
Description	test if bigger or equal. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(10, 11, 12, 13);
Vec4i  b(14, 13, 12, 11);
Vec4ib c = a >= b;     // c = (false, false, true, true)
```

Operator	<
Defined for	all integer and floating point vector classes
Description	test if smaller. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(10, 11, 12, 13);
Vec4i  b(14, 13, 12, 11);
Vec4ib c = a < b;        // c = (true, true, false, false)
```

Operator	<=
Defined for	all integer and floating point vector classes
Description	test if smaller or equal. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(10, 11, 12, 13);
Vec4i  b(14, 13, 12, 11);
Vec4ib c = a <= b;       // c = (true, true, true, false)
```

Operator	&, &=
Defined for	all vector classes
Description	bitwise and
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = a & b;         // c = (0, 1, 4, 5)
```

Operator	|, |=
Defined for	all vector classes
Description	bitwise or
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = a | b;         // c = (30, 31, 30, 31)
```

Operator	^
Defined for	all vector classes
Description	bitwise exclusive or
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = a ^ b;         // c = (30, 30, 26, 26)
```

Operator	~
Defined for	all boolean and integer vector classes
Description	bitwise not
Efficiency	good

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b = ~a;                  // b = (-11, -12, -13, -14)
```

Operator	!
Defined for	all vector classes
Description	logical not. Result is a boolean vector
Efficiency	good

```
// Example:
Vec4i  a(-1, 0, 1, 2);
Vec4ib b = !a;                 // b = (false,true,false,false)
```

3.3 Integer division

There are no instructions in the x86 instruction set extensions that are useful for integer vector division, and such instructions might be quite slow if they existed. Therefore, the vector class library is using an algorithm for fast integer division. The basic principle of this algorithm can be expressed in this formula:

$a/b \approx a * (2^n/b) >> n$

This calculation goes through the following steps:

1. find a suitable value for n
2. calculate $2^n/b$
3. calculate necessary corrections for rounding errors
4. do the multiplication and shift-right, and apply corrections for rounding errors

This formula is advantageous if multiple numbers are divided by the same divisor b. Steps 1, 2 and 3 need only be done once while step 4 is repeated for each value of the dividend a. The mathematical details are described in the file vectori128.h. (See also T. Granlund and P. L. Montgomery: Division by Invariant Integers Using Multiplication, Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation)

The implementation in the vector class library uses various variants of this method with appropriate corrections for rounding errors to get the exact result truncated towards zero.
The way to use this in your code depends on whether the divisor b is a variable or constant, and whether the same divisor is applied to multiple vectors. This is illustrated in the following examples:

```
// Division example A:
// A variable divisor is applied to one vector
Vec4i a(10, 11, 12, 13);// dividend is an integer vector
int   b = 3;            // divisor is an integer variable
Vec4i c = a / b;        // result c = (3, 3, 4, 4)

// Division example B:
// The same divisor is applied to multiple vectors
int b = 3;              // divisor
Divisor_i divb(b);      // this object contains the results
                        // of calculation steps 1, 2, and 3
for (...) {             // loop through multiple vectors
    Vec4i a = ...       // get dividend
```

```
    a = a / divb;          // do step 4 of the division
    ...                    // store results
}

// Division example C:
// The divisor is a constant, known at compile time
Vec4i a(10, 11, 12, 13);      // dividend is integer vector
Vec4i c = a / const_int(3);   // result c = (3, 3, 4, 4)
```

Explanation:
The class Divisor_i in example B takes care of the calculation steps 1, 2 and 3 in the algorithm described above. The overloaded / operator takes a vector on the left hand side and an object of class Divisor_i on the right hand side. This object is created before the loop with the divisor as parameter to the constructor. We are saving time by doing this time-consuming calculation only once while step 4 in the calculation is done multiple times inside the loop by a = a / divb;

In example A, we are also creating an object of class Divisor_i, but this is done implicitly. The compiler sees an integer on the right hand side of the / operator where it needs an object of class Divisor_i, and therefore converts the integer b to such an object by calling the constructor Divisor_i(int).

The following divisor classes are available:

Dividend vector type	**Divisor class required**
Vec16c, Vec32c , Vec64c	Divisor_s
Vec16uc, Vec32uc, Vec64uc	Divisor_us
Vec8s, Vec16s , Vec32s	Divisor_s
Vec8us, Vec16us , Vec32us	Divisor_us
Vec4i, Vec8i, Vec16i	Divisor_i
Vec4ui, Vec8ui, Vec16ui	Divisor_ui

If the divisor is a constant and the value is known at compile time, then we can use the method in example C. The implementation here uses macros and templates to do the calculation steps 1, 2 and 3 at compile time rather than at execution time. This makes the code even faster. The expression to put on the right-hand side of the / operator looks as follows:

Dividend vector type	**Divisor expression**
Vec16c, Vec32c, Vec64c	const_int
Vec16uc, Vec32uc, Vec64uc	const_uint
Vec8s, Vec16s, Vec32s	const_int
Vec8us, Vec16us, Vec32us	const_uint
Vec4i, Vec8i, Vec16i	const_int
Vec4ui, Vec8ui, Vec16ui	const_uint

The compiler will generate an error message if the parameter to const_int or const_uint is not a valid compile-time constant. (A valid compile time constant can contain integer literals and operators, as well as macros that are expanded to compile time constants, but not ordinary function calls).

A further advantage of the method in example C is that the code is able to use different methods for different values of the divisor. The division is particularly fast if the divisor is a power of 2. Make sure to use const_int or const_uint on the right hand side of the / operator if you are dividing by 2, 4, 8, 16, etc.

Division is faster for vectors of 16-bit integers than for vectors of 8-bit or 32-bit integers. There is no support for division of vectors of 64-bit integers. Unsigned division is faster than signed division.

Chapter 4

Functions

4.1 Integer functions

Function	horizontal_add
Defined for	all integer vector classes
Description	calculates the sum of all vector elements
Efficiency	medium. For best performance, use normal (vertical) addition where possible.

```
// Example:
Vec4i a(10, 11, 12, 13);
int   b = horizontal_add(a);   // b = 46
```

Function	horizontal_add_x
Defined for	all 8-bit, 16-bit and 32-bit integer vector classes
Description	calculates the sum of all vector elements. The sum is calculated with a higher number of bits to avoid overflow
Efficiency	medium (slower than horizontal_add)

```
// Example:
Vec4i   a(10, 11, 12, 13);
int64_t b = horizontal_add_x(a);   // b = 46
```

Function	horizontal_min, horizontal_max
Defined for	all integer vector classes
Description	Returns the lowest or highest element in a vector.
Efficiency	medium

```
// Example:
Vec4i a(1, 8, -5, 3);
int   b = horizontal_min(a);   // b = -5
int   c = horizontal_max(a);   // c =  8
```

Function	add_saturated
Defined for	all 8-bit, 16-bit and 32-bit integer vector classes
Description	same as operator +. Overflow is handled by saturation rather than wrap-around
Efficiency	fast for 8-bit and 16-bit integers. Medium for 32-bit integers

```
// Example:
Vec4i a(0x10000000, 0x20000000, 0x30000000, 0x40000000);
Vec4i b(0x30000000, 0x40000000, 0x50000000, 0x60000000);
```

```
Vec4i c = add_saturated(a, b);
//  c = (0x40000000,  0x60000000,  0x7FFFFFFF,  0x7FFFFFFF)
Vec4i d = a + b;
//  d = (0x40000000,  0x60000000, -0x80000000, -0x60000000)
```

Function	sub_saturated
Defined for	all 8-bit, 16-bit and 32-bit integer vector classes
Description	same as operator -. Overflow is handled by saturation rather than wrap-around
Efficiency	fast for 8-bit and 16-bit integers. Medium for 32-bit integers

```
// Example:
Vec4i a(-0x10000000,-0x20000000,-0x30000000,-0x40000000);
Vec4i b( 0x30000000, 0x40000000, 0x50000000, 0x60000000);
Vec4i c = sub_saturated(a, b);
//  c = (-0x40000000,-0x60000000,-0x80000000,-0x80000000)
Vec4i d = a - b;
//  d = (-0x40000000,-0x60000000,-0x80000000, 0x60000000)
```

Function	max
Defined for	all integer vector classes
Description	returns the biggest of two values
Efficiency	medium for 64-bit integers with instruction sets lower than SSE4.2. Fast otherwise

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = max(a, b);  // c = (14, 13, 12, 13)
```

Function	min
Defined for	all integer vector classes
Description	returns the smallest of two values
Efficiency	medium for 64-bit integers with instruction sets lower than SSE4.2. Fast otherwise

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = min(a, b);  // c = (10, 11, 12, 11)
```

Function	abs
Defined for	all signed integer vector classes
Description	calculates the absolute value
Efficiency	medium

```
// Example:
Vec4i a(-1, 0, 1, 2);
Vec4i b = abs(a);      // b = (1, 0, 1, 2)
```

Function	abs_saturated
Defined for	all signed integer vector classes
Description	calculates the absolute value. Overflow saturates to make sure the result is never negative when the input is INT_MIN
Efficiency	medium (slower than abs)

```
// Example:
Vec4i a(-0x80000000, -1, 0, 1);
Vec4i b = abs_saturated(a);   // b=( 0x7FFFFFFF,1,0,1)
Vec4i c = abs(a);             // c=(-0x80000000,1,0,1)
```

Function	rotate_left(vector, int)
Defined for	all signed integer vector classes
Description	rotates the bits of each element. Use a negative count to rotate right
Efficiency	8 bit: poor 16 bit: medium 32 and 64 bit: good for AVX512DQ instruction set, medium otherwise.

```
// Example:
Vec4i a(0x12345678, 0x0000FFFF, 0xA000B000, 0x00000001);
Vec4i b = rotate_left(a, 8);
// b = (0x34567812, 0x00FFFF00, 0x00B000A0, 0x00000100)
```

Function	vector shift_bytes_up<n>(vector) vector shift_bytes_down<n>(vector)
Defined for	Vec16c, Vec32c, Vec64c
Description	shifts the bytes of a vector up or down and inserts zeroes at the vacant places
Efficiency	Vec16c: Good for SSSE3, medium otherwise Vec32c: Good for AVX2, medium otherwise Vec64c: Good for AVX512BW, medium otherwise

```
// Example:
Vec16c a(10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25);
Vec16c b = shift_bytes_up<5>(a);
// b = (0,0,0,0,0,10,11,12,13,14,15,16,17,18,19,20)
```

4.2 Floating point simple functions

Function	horizontal_add
Defined for	all floating point vector classes
Description	calculates the sum of all vector elements
Efficiency	medium. For best performance, use normal (vertical) addition where possible.

```
// Example:
Vec4f a(1.0f, 1.1f, 1.2f, 1.3f);
float b = horizontal_add(a);  // b = 4.6
```

Function	max min
Defined for	all floating point vector classes
Description	returns the biggest/smallest of two values
Efficiency	good

max(a,b) is equivalent to a > b ? a : b
min(a,b) is equivalent to a < b ? a : b

These functions will not return a NAN if the first parameter is NAN.
These functions make no distinction between 0 and -0.

```
// Example:
Vec4f a(1.0f, 1.1f, 1.2f, 1.3f);
Vec4f b(1.4f, 1.3f, 1.2f, 1.1f);
Vec4f c = max(a, b);          // c = (1.4f, 1.3f, 1.2f, 1.3f)
```

Function	maximum minimum
Defined for	all floating point vector classes
Description	returns the biggest/smallest of two values
Efficiency	good, but slower than max / min

These functions are similar to max and min, but sure to propagate NAN values.
The sign of zero is ignored unless SIGNED_ZERO is defined.

Function	horizontal_min, horizontal_max
Defined for	all floating point vector classes
Description	Returns the lowest or highest element in a vector. NANs are propagated. The sign of zero is ignored.
Efficiency	medium

```
// Example:
Vec4i a(1, 8, -5, 3);
int   b = horizontal_min(a);  // b = -5
int   c = horizontal_max(a);  // c =  8
```

Function	abs
Defined for	all floating point vector classes
Description	gets the absolute value
Efficiency	good

```
// Example:
Vec4f a(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f b = abs(a);  // b = (1.0f, 0.0f, 1.0f, 2.0f)
```

Function	change_sign<i0, i1, ...>(vector)
Defined for	all floating point vector classes
Description	changes sign of selected vector elements. Each template parameter is 1 for changing sign of the corresponding element, and 0 for no change.
Efficiency	good

```
// Example:
Vec4f a(10.0f, 11.0f, -12.0f, 13.0f);
Vec4f b = change_sign<0,1,1,0>(a); // b = (10.f, -11.f, 12.f, 13.f)
```

Function	sign_combine(vector a, vector b)
Defined for	all floating point vector classes
Description	Returns the value of a, with the sign inverted if b has its sign bit set. Corresponds to select(sign_bit(b), -a, a)
Efficiency	good

```
// Example:
Vec4f  a(-2.0f, -1.0f,  0.0f,  1.0f);
Vec4f  b(-10.f,  0.0f, -20.f,  30.f);
Vec4f  c = sign_combine(a, b);  // c = (2.0f, -1.0f, -0.0f, 1.0f)
```

Function	sign_bit
Defined for	all floating point vector classes
Description	returns a boolean vector with true for elements that have the sign bit set, including -0.0, -INF, and -NAN
Efficiency	medium

```
// Example:
Vec4f  a(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4fb b = sign_bit(a);  // b = (true, false, false, false)
```

Function	sqrt
Defined for	all floating point vector classes
Description	calculates the square root
Efficiency	poor

```
// Example:
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
Vec4f b = sqrt(a);  // b = (0.000f, 1.000f, 1.414f, 1.732f)
```

Function	square
Defined for	all floating point vector classes
Description	calculates the square
Efficiency	good

```
// Example:
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
Vec4f b = square(a);  // b = (0.0f, 1.0f, 4.0f, 9.0f)
```

Function	pow(vector x, int n)
Defined for	all floating point vector classes
Description	raises all vector elements to the same integer power. Will generate a compiler error if n is floating point and vector-math_exp.h is not included, or in general if n is not of type int. See page 59 for pow with floating point exponent.
Precision	slightly imprecise for high values of n due to accumulation of rounding errors
Efficiency	medium

```
// Example:
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
int   b = 3;
Vec4f c = pow(a, b);  // c = (0.0f, 1.0f, 8.0f, 27.0f)
```

Function	pow_const(vector x, const int n)
Defined for	all floating point vector classes
Description	raises all vector elements to the same integer power n, where n is a compile-time constant
Precision	slightly imprecise for high values of n due to accumulation of rounding errors
Efficiency	medium, often better than pow(vector, int)

```
// Example:
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
Vec4f c = pow_const(a, 3);  // c = (0.0f, 1.0f, 8.0f, 27.0f)
```

Function	round
Defined for	all floating point vector classes
Description	round to nearest integer (even value if two values are equally near). The value is returned as a floating point vector. See also roundi and round_to_int32 on page 36.
Efficiency	good if SSE4.1 or higher instruction set

```
// Example:
Vec4f a(1.0f, 1.4f, 1.5f, 1.6f)
Vec4f b = round(a);   // b = (1.0f, 1.0f, 2.0f, 2.0f)
```

Function	truncate
Defined for	all floating point vector classes
Description	truncates number towards zero. The value is returned as a floating point vector. See also truncatei and truncate_to_int32 on page 37.
Efficiency	good if SSE4.1 or higher instruction set
Note	may be slightly inaccurate for x $> 10^7$ if instruction set is less than SSE4.1

```
// Example:
Vec4f a(1.0f, 1.5f, 1.9f, 2.0f)
Vec4f b = truncate(a);   // b = (1.0f, 1.0f, 1.0f, 2.0f)
```

Function	floor
Defined for	all floating point vector classes
Description	rounds number towards $-\infty$. The value is returned as a floating point vector
Efficiency	good if SSE4.1 or higher instruction set
Note	may be slightly inaccurate for x $> 10^7$ if instruction set is less than SSE4.1

```
// Example:
Vec4f a(-0.5f, 1.5f, 1.9f, 2.0f)
Vec4f b = floor(a);   // b = (-1.0f, 1.0f, 1.0f, 2.0f)
```

Function	ceil
Defined for	all floating point vector classes
Description	rounds number towards $+\infty$. The value is returned as a floating point vector
Efficiency	good if SSE4.1 or higher instruction set
Note	may be slightly inaccurate for x $> 10^7$ if instruction set is less than SSE4.1

```
// Example:
Vec4f a(-0.5f, 1.1f, 1.9f, 2.0f)
Vec4f b = ceil(a);   // b = (0.0f, 2.0f, 2.0f, 2.0f)
```

Function	approx_recipr
Defined for	single and half precision floating point vectors
Description	fast approximate calculation of reciprocal
Precision	the relative accuracy depends on the instruction set: Default: 2^{-11} AVX512F: 2^{-14} AVX512ER: full precision
Efficiency	good

```
// Example:
Vec4f a(1.5f, 2.0f, 3.0f, 4.0f)
Vec4f b(0.5f, 1.0f, 0.5f, 1.0f)
Vec4f c = a * approx_recipr(b);  // c approximates a/b
```

Function	approx_rsqrt
Defined for	single and half precision floating point vectors
Description	reciprocal square root. Fast approximate calculation of value to the power of -0.5
Precision	the relative accuracy depends on the instruction set: Default: 2^{-11} AVX512F: 2^{-14} AVX512ER: full precision
Efficiency	good

```
// Example:
Vec4f a(1.0f, 2.0f, 3.0f, 4.0f)
Vec4f b = approx_rsqrt(a) * a;  // b approximates sqrt(a)
```

Chapter 5

Boolean operations and per-element branches

Consider this piece of C++ code:

```
int a[4], b[4], c[4], d[4];
  ...
for (int i = 0; i < 4; i++) {
    d[i] = (a[i] > 0 && a[i] < 10) ? b[i] : c[i];
}
```

We can do this with vectors in the following way:

```
Vec4i a, b, c, d;
  ...
d = select(a > 0 & a < 10, b, c);
```

The select function is similar to the ?: operator. It has three vector parameters: The first parameter is a boolean vector that chooses between the elements of the second and the third vector parameter.

The relational operators >, >=, <, <=, ==, != produce boolean vectors, which accept the boolean operations &, |, ∧, ∼ (and, or, exclusive or, not).

In the above example, the expressions a > 0 and a < 10 are boolean vectors of type Vec4ib. The boolean vectors must have a type that matches the data vectors they are used with. Table 2.3 on page 9 shows which boolean vector class to use for each vector type.

The vector elements that are not selected are calculated anyway because normally all parts of a vector are calculated. For example:

```
Vec4f a(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f b = select(a >= 0.0f, sqrt(a), 0.0f);
```

Here, we will be calculating the square root of -1 even though we are not using it. This will not cause problems if floating point exceptions are masked off, which they normally are. A safe solution that works even if floating point exceptions are enabled would be:

```
Vec4f a(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f b = sqrt(max(a, 0.0f));
```

Likewise, the & and | operators are calculating both input operands, even if the second operand is not needed. The following examples illustrates this:

```
// array version:
float a[4] = {0.0f, 1.0f, 2.0f, 3.0f};
```

```
float b[4];
for (int i = 0; i < 4; i++) {
   if (a[i] > 0.0f && 1.0f/a[i] != 4.0f) {
      b[i] = a[i];
   }
   else {
      b[i] = 1.0f;
   }
}
```

and the vector version of the same:

```
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
Vec4f b = select(a > 0.0f & 1.0f/a != 4.0f, a, 1.0f);
```

In the array version, we will never divide by zero because the && operator does not evaluate the second operand when the first operand is false. But in the vector version, we are indeed dividing by zero because the & operator always evaluates both operands. The vector class library defines the operators && and || as synonyms for & and | for convenience, but they are still doing the bitwise AND or OR operation, so & and | are actually more representative of what these operators really do. This example should be changed to:

```
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
Vec4f b = select(a > 0.0f & a != 0.25f, a, 1.0f);
```

5.1 Internal representation of boolean vectors

The way boolean vectors are stored depends on the instruction set and the Vector Class Library (VCL) version. Older instruction sets have the boolean vectors stored with the same number of bits as the data vectors they are applied to (broad boolean vectors). The later instruction sets AVX512 and AVX512VL allow boolean vectors to be stored with only one bit for each element (compact boolean vectors).

Version 1.xx of the VCL is using the broad boolean vectors for the sake of backwards compatibility, while version 2.xx is prioritizing the more efficient compact boolean vectors when the appropriate instruction set is enabled. The boolean vector sizes are summarized in the following table.

Data vector size and instruction set	**VCL version 1 Boolean vectors**	**VCL version 2 Boolean vectors**
128 bits	broad	broad
128 bits with AVX512VL	broad	compact
256 bits	broad	broad
256 bits with AVX512VL	broad	compact
512 bits	broad	broad
512 bits with AVX512F	compact	compact

The broad boolean vectors are stored as integer vectors with the same number of bits per element as the integer or floating point vectors they are used for. For example, the broad boolean vector class Vec4fb is stored as a vector of four 32-bit integers because it is used with vectors Vec4f of four single precision floating point numbers, using 32 bits each. The broad boolean vector class Vec4db is stored as a vector of four 64-bit integers because it is used with vectors Vec4d of four double precision floating point numbers, using 64 bits each. Note that the integer representation of true in a broad boolean vector element is not 1, but -1. The representation of false is 0. Any other values than 0 and

-1 in broad boolean vectors will produce wrong and inconsistent results that depend on the instruction set.

The compact boolean vectors are stored with one bit per element (at least 8 bits). You should make no assumption about how boolean vectors are stored if your code may be compiled for different instruction sets or different versions of VCL. For example, Vec16ib uses 16 bits of storage when compiling for AVX512, but 512 bits of storage when compiling for AVX2. Do not store boolean vectors directly to binary files, and do not transmit boolean vectors between different functions that may be compiled for different instruction sets or different VCL versions.

Different compact boolean vectors are mutually compatible if they have the same number of elements. Different broad boolean vectors are mutually compatible if they have the same number of elements and the same number of bits. Broad and compact boolean vectors are not compatible with each other. See page 42 for conversion between different types of boolean vectors.

5.2 Functions for use with booleans

Function	vector select(boolean vector s, vector a, vector b)
Defined for	all integer and floating point vector classes
Description	branch per element. result[i] = s[i] ? a[i] : b[i]
Efficiency	good

```
// Example:
Vec4i a(-1, 0, 1, 2);
Vec4i b = select(a>0, a+10, a-10); // b = (-11,-10,11,12)
```

Function	vector if_add(boolean vector f, vector a, vector b)
Defined for	all integer and floating point vector classes
Description	conditional addition result[i] = f[i] ? (a[i] + b[i]) : a[i]
Efficiency	good

```
// Example:
Vec4i a(-1, 0, 1, 2);
Vec4i b = if_add(a < 0, a, 100);  // b = (99,0,1,2)
```

Function	vector if_sub(boolean vector f, vector a, vector b)
Defined for	all integer and floating point vector classes
Description	conditional subtraction result[i] = f[i] ? (a[i] - b[i]) : a[i]
Efficiency	good

Function	vector if_mul(boolean vector f, vector a, vector b)
Defined for	all integer and floating point vector classes
Description	conditional multiplication result[i] = f[i] ? (a[i] * b[i]) : a[i]
Efficiency	good

Function	vector if_div(boolean vector f, vector a, vector b)
Defined for	all floating point vector classes
Description	conditional division result[i] = f[i] ? (a[i] / b[i]) : a[i]
Efficiency	medium

Function	vector andnot(vector, vector)
Defined for	all boolean vector classes
Description	andnot(a,b) = a & ~ b
Efficiency	good

Function	bool horizontal_and(boolean vector)
Defined for	all boolean vector classes
Description	The output is the AND combination of all elements
Efficiency	Medium for broad boolean vectors. Better if SSE4.1 or later. Good for compact boolean vectors

```
// Example:
Vec4i a(-1, 0, 1, 2);
bool  b = horizontal_and(a > 0);  // b = false
```

Function	bool horizontal_or(boolean vector)
Defined for	all boolean vector classes
Description	The output is the OR combination of all elements
Efficiency	Medium for broad boolean vectors. Better if SSE4.1 or later. Good for compact boolean vectors

```
// Example:
Vec4i a(-1, 0, 1, 2);
bool  b = horizontal_or(a > 0);  // b = true
```

Function	int horizontal_find_first(boolean vector)
Defined for	all boolean vector classes
Description	Returns an index to the first element that is true. Returns -1 if all elements are false
Efficiency	medium

```
// Example:
Vec4i  a(1, 2, 3, 4);
Vec4i  b(0, 2, 3, 5);
int c = horizontal_find_first(a == b);  // c = 1
```

Function	unsigned int horizontal_count(boolean vector)
Defined for	all boolean vector classes
Description	counts the number of elements that are true
Efficiency	medium if SSE4.2 or later

```
// Example:
Vec4i  a(1, 2, 3, 4);
Vec4i  b(0, 2, 3, 5);
int c = horizontal_count(a == b);  // c = 2
```

Chapter 6

Conversion between vector types

Below is a list of methods and functions for conversion between different vector types, vector sizes or precisions.

6.1 Conversion between data vector types

Method	conversion between vector class and intrinsic vector type
Defined for	all integer and floating point vector classes
Description	conversion between a vector class and the corresponding intrinsic vector type __m128, __m128d, __m128i, __m256, __m256d, __m256i, __m512, __m512d, __m512i can be done implicitly or explicitly. Boolean vectors can be converted to their internal representation, which is an integer vector for broad boolean vectors, or a single integer for compact boolean vectors.
Efficiency	good

```
// Example:
Vec4i    a(0,1,2,3);
__m128i  b = a;      // b = 0x00000003000000020000000100000000
Vec4i    c = b;      // c = (0,1,2,3)
```

Method	conversion from scalar to vector
Defined for	all integer and floating point vector classes
Description	conversion from a scalar (single value) to a vector can be done explicitly by calling a constructor, or implicitly by putting a scalar where a vector is expected. All vector elements get the same value.
Efficiency	good for constant. Medium for variable as parameter

```
// Example:
Vec4i a, b;
a = Vec4i(5);   // explicit conversion. a = (5,5,5,5)
b = a + 3;      // implicit conversion to Vec4i. b = (8,8,8,8)
```

Method	conversion between signed and unsigned integer vectors
Defined for	all integer vector classes
Description	Conversion between signed and unsigned integer vectors can be done implicitly or explicitly. Overflow and underflow wraps around.
Efficiency	good

```
// Example:
Vec4i   a(-1,0,1,2);    // signed vector
```

```
Vec4ui b = a;          // implicit conversion to unsigned.
                       // b = (0xFFFFFFFF,0,1,2)
Vec4ui c = Vec4ui(a); // same, with explicit conversion
Vec4i  d = c;          // convert back to signed
```

Method	conversion between different integer vector types
Defined for	all integer vector classes
Description	Conversion can be done implicitly or explicitly between all integer vector classes with the same total number of bits. This conversion does not change any bits, just the grouping of bits into elements is changed.
Efficiency	good

```
// Example:
Vec8s a(0,1,2,3,4,5,6,7);
Vec4i b;
b = a;                 // b = (0x1000, 0x3002, 0x5004, 0x7006)
```

Function	reinterpret_d, reinterpret_f, reinterpret_i, reinterpret_h
Defined for	all integer and floating point vector classes
Description	Reinterprets a vector as a different type with the same total number of bits. No bits are changed, only interpreted differently (bit casting). reinterpret_d is used for converting to Vec2d, Vec4d, or Vec8d, reinterpret_f is used for converting to Vec4f, Vec8f, or Vec16f, reinterpret_i is used for converting to any integer vector type, reinterpret_h is used for converting to Vec8h, Vec16h, or Vec32h.
Efficiency	good

```
// Example:
Vec4f a(1.0f, 1.5f, 2.0f, 2.5f);
Vec4i b = reinterpret_i(a);
// b = (0x3F800000, 0x3FC00000, 0x40000000, 0x40200000)
```

Function	Vec8s roundi(Vec8h) Vec16s roundi(Vec16h) Vec32s roundi(Vec32h) Vec4i roundi(Vec4f) Vec8i roundi(Vec8f) Vec16i roundi(Vec16f) Vec2q roundi(Vec2d) Vec4q roundi(Vec4d) Vec8q roundi(Vec8d)
Defined for	all floating point vector classes
Description	Rounds floating point numbers to nearest integer and returns an integer vector of the same size. Where two integers are equally near, the even integer is returned. INF input may give INT_MAX or INT_MIN depending on the implementation and the instruction set.
Efficiency	float types: good double types: good if AVX512DQ instruction set, otherwise poor

```
// Example:
Vec4f a(1.0f, 1.5f, 2.0f, 2.5f);
Vec4i b = round_to_int(a);  // b = (1,2,2,2)
```

Function	Vec4i round_to_int32(Vec2d) Vec4i round_to_int32(Vec2d, Vec2d) Vec4i round_to_int32(Vec4d) Vec8i round_to_int32(Vec8d)
Defined for	Vec2d, Vec4d, Vec8d
Description	rounds double precision floating point numbers and returns vector of 32-bit integers. Where two integers are equally near, the even integer is returned.
Efficiency	good

```
// Example:
Vec4d a(1.0, 1.5, 2.0, 2.5);
Vec4i b = round_to_int32(a);  // b = (1,2,2,2)
```

Function	Vec8s truncatei(Vec8h) Vec16s truncatei(Vec16h) Vec32s truncatei(Vec32h) Vec4i truncatei(Vec4f) Vec8i truncatei(Vec8f) Vec16i truncatei(Vec16f) Vec2q truncatei(Vec2d) Vec4q truncatei(Vec4d) Vec8q truncatei(Vec8d)
Defined for	all floating point vector classes
Description	truncates floating point numbers towards zero and returns signed integer vector of the same size. INF input may give INT_MAX or INT_MIN depending on the implementation and the instruction set.
Efficiency	float types: good double types: good if AVX512DQ instruction set, otherwise poor

```
// Example:
Vec4f a(-1.6f, 1.5f, 2.0f, 2.9f);
Vec4i b = truncate_to_int(a);  // b = (-1,1,2,2)
```

Function	Vec4i truncate_to_int32(Vec2d, Vec2d) Vec4i truncate_to_int32(Vec4d) Vec8i truncate_to_int32(Vec8d)
Defined for	Vec2d, Vec4d, Vec8d
Description	truncates double precision floating point numbers towards zero and returns signed vector of 32-bit integers.
Efficiency	good

```
// Example:
Vec4d a(-1.5, 1.5, 2.0, 2.9);
Vec4i b = truncate_to_int32(a);  // b = (-1,1,2,2)
```

Function	Vec4f to_float(Vec4i) Vec8f to_float(Vec8i) Vec16f to_float(Vec16i)
Defined for	Vec4i, Vec8i, Vec16i
Description	converts signed 32-bit integers to single precision float
Efficiency	good

```
// Example:
```

```
Vec4i a(0, 1, 2, 3);
Vec4f b = to_float(a);  // b = (0.0f, 1.0f, 2.0f, 3.0f)
```

Function	Vec4f to_float(Vec4ui) Vec8f to_float(Vec8ui) Vec16f to_float(Vec16ui)
Defined for	Vec4ui, Vec8ui, Vec16ui
Description	converts unsigned integers to single precision float
Efficiency	good if AVX512VL instruction set. Poor otherwise

```
// Example:
Vec4ui a(0, 1, 2, 3);
Vec4f b = to_float(a);  // b = (0.0f, 1.0f, 2.0f, 3.0f)
```

Function	Vec4f to_float(Vec2d) Vec4f to_float(Vec4d) Vec8f to_float(Vec8d)
Defined for	Vec2d, Vec4d, Vec8d
Description	converts floating point vectors from double precision to single precision.
Efficiency	good

Function	Vec4f convert8h_4f(Vec8h) Vec8f to_float(Vec8h) Vec16f to_float(Vec16h)
Defined for	Vec8h, Vec16h
Description	converts floating point vectors from half precision to single precision.
Efficiency	good if F16C or AVX512-FP16

Function	Vec8h convert4f_8h(Vec4f) Vec8h to_float16(Vec8f) Vec16h to_float16(Vec16f)
Defined for	Vec4f, Vec8f, Vec16f
Description	converts floating point vectors from single precision to half precision.
Efficiency	good if F16C or AVX512-FP16

Function	Vec4d to_double(Vec4i) Vec8d to_double(Vec8i)
Defined for	Vec4i, Vec8i
Description	converts signed 32-bit integers to double precision float. The output vector is larger than the input vector.
Efficiency	medium

```
// Example:
Vec4i a(0, 1, 2, 3);
Vec4d b = to_double(a);  // b = (0.0, 1.0, 2.0, 3.0)
```

Function	Vec2d to_double(Vec2q x) Vec4d to_double(Vec4q x) Vec8d to_double(Vec8q x) Vec2d to_double(Vec2uq x) Vec4d to_double(Vec4uq x) Vec8d to_double(Vec8uq x)
Defined for	Vec2q, Vec4q, Vec8q, Vec2uq, Vec4uq, Vec8uq
Description	converts signed or unsigned 64-bit integers to double precision float
Efficiency	good if AVX512DQ and AVX512VL instruction sets, otherwise poor.

```
// Example:
Vec2q a(0, 1);
Vec2d b = to_double(a);   // b = (0.0, 1.0)
```

Function	Vec4d to_double(Vec4f x) Vec8d to_double(Vec8f x)
Defined for	Vec4f, Vec8f
Description	converts floating point vectors from single precision to double precision. The total number of bits in the vector is doubled
Efficiency	good

Function	Vec2d to_double_low(Vec4i) Vec2d to_double_high(Vec4i)
Defined for	Vec4i
Description	converts signed 32-bit integers to double precision float
Efficiency	medium

```
// Example:
Vec4i a(0, 1, 2, 3);
Vec2d b = to_double_low(a);   // b = (0.0, 1.0)
Vec2d c = to_double_high(a); // c = (2.0, 3.0)
```

Method	concatenating vectors
Defined for	All 128-bit and 256-bit vector classes and corresponding boolean vector classes
Description	Two vectors can be concatenated into one vector of the double size by calling a constructor or the function concatenate2.
Efficiency	good

```
// Example:
Vec4i a(10,11,12,13);
Vec4i b(20,21,22,23);
Vec8i c(a, b);      // c = (10,11,12,13,20,21,22,23)
Vec8i d = concatenate2(a, b); // same as c
```

Method	get_low, get_high
Defined for	all 256-bit and 512-bit vector classes
Description	One big vector can be split into two vectors of half the size by calling the methods get_low and get_high
Efficiency	good

```
// Example:
Vec8i a(10,11,12,13,14,15,16,17);
Vec4i b = a.get_low();   // b = (10,11,12,13)
Vec4i c = a.get_high(); // c = (14,15,16,17)
```

Method	extend_z
Defined for	All 128-bit and 256-bit vector classes and corresponding boolean vector classes
Description	The vector is extended to double size by adding zeroes.
Efficiency	good

```
// Example:
Vec4i a(10,11,12,13);
Vec8i b = extend_z(a);   // b = (10,11,12,13,0,0,0,0)
```

Function	extend
Defined for	Vec16c, Vec16uc, Vec32c, Vec32uc, Vec8s, Vec8us, Vec16s, Vec16us, Vec4i, Vec4ui, Vec8i, Vec8ui,
Description	Extends integers to a larger number of bits per element. The total number of bits in the vector is doubled. Unsigned integers are zero-extended, signed integers are sign-extended.
Efficiency	good for instruction sets that support the highest vector size, medium otherwise.

```
// Example:
Vec8s a(-2, -1, 0, 1, 2, 3, 4, 5);
Vec8i b = extend(a);    // b = (-2, -1, 0, 1, 2, 3, 4, 5)
```

Function	extend_low, extend_high
Defined for	Vec16c, Vec16uc, Vec32c, Vec32uc, Vec64c, Vec64uc, Vec8s, Vec8us, Vec16s, Vec16us, Vec32s, Vec32us, Vec4i, Vec4ui, Vec8i, Vec8ui, Vec16i, Vec16ui
Description	Extends integers to a larger number of bits per element. Only the lower or upper half of the vector is converted. The total number of bits in the vector is unchanged. Unsigned integers are zero-extended, signed integers are sign-extended.
Efficiency	good

```
// Example:
Vec8s a(-2, -1, 0, 1, 2, 3, 4, 5);
Vec4i b = extend_low(a);    // b = (-2, -1, 0, 1)
Vec4i c = extend_high(a);   // c = (2, 3, 4, 5)
```

Function	extend_low, extend_high
Defined for	Vec4f, Vec8f, Vec16f
Description	extends single precision floating point numbers to double precision. Only the lower or upper half of the vector is converted. The total number of bits in the vector is unchanged.
Efficiency	good

```
// Example:
Vec4f a(1.0f, 1.1f, 1.2f, 1.3f);
Vec2d b = extend_low(a);    // b = (1.0, 1.1)
Vec2d c = extend_high(a);   // c = (1.2, 1.3)
```

Function	compress
Defined for	Vec16s, Vec16us, Vec32s, Vec32us, Vec8i, Vec8ui, Vec16i, Vec16ui, Vec4q, Vec4uq, Vec8q, Vec8uq
Description	Reduces integers to a lower number of bits per element. The total number of bits in the vector is halved. There is no overflow check. The upper bits are simply cut off (wrap around).
Efficiency	good for instruction sets that support the highest vector size, medium otherwise .

```
// Example:
Vec8q a(10, 11, 12, 13, 14, 15, 16, 17);
Vec8i b = compress(a); // b = (10, 11, 12, 13, 14, 15, 16, 17)
```

Function	compress (with two vector parameters)
Defined for	Vec8s, Vec8us, Vec16s, Vec16us, Vec32s, Vec32us, Vec4i, Vec4ui, Vec8i, Vec8ui, Vec16i, Vec16ui, Vec2q, Vec2uq, Vec4q, Vec4uq, Vec8q, Vec8uq
Description	Packs two integer vectors into a single vector with the same total number of bits, by reducing each integer to a lower number of bits per element. There is no overflow check. The upper bits are simply cut off (wrap around).
Efficiency	medium

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec8s c = compress(a, b); // c = (10,11,12,13,20,21,22,23)
```

Function	compress (with two vector parameters)
Defined for	Vec2d, Vec4d, Vec8d
Description	reduces double precision floating point numbers to single precision. Two double precision vectors are packed into one single precision vector with the same total number of bits.
Efficiency	medium

```
// Example:
Vec2d a(1.0, 1.1);
Vec2d b(2.0, 2.1);
Vec4f c = compress(a, b); // c = (1.0f, 1.1f, 2.0f, 2.1f)
```

Function	compress_saturated (with one vector parameter)
Defined for	Vec16s, Vec16us, Vec32s, Vec32us, Vec8i, Vec8ui, Vec16i, Vec16ui, Vec4q, Vec4uq, Vec8q, Vec8uq
Description	Packs an integer vector into a vector with the same number of elements and half the number of bits per element. Overflow and underflow saturates
Efficiency	medium (worse than compress in most cases)

Function	compress_saturated (with two vector parameters)
Defined for	Vec8s, Vec8us, Vec16s, Vec16us, Vec32s, Vec32us, Vec4i, Vec4ui, Vec8i, Vec8ui, Vec16i, Vec16ui, Vec2q, Vec2uq, Vec4q, Vec4uq, Vec8q, Vec8uq
Description	Packs two integer vectors into a single vector with the same total number of bits, by reducing each integer to a lower number of bits per element. Overflow and underflow saturates
Efficiency	medium (worse than compress in most cases)

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec8s c = compress_saturated(a, b);
// c = (10,11,12,13,20,21,22,23)
```

6.2 Conversion between boolean vector types

Function	to_bits
Defined for	all boolean vectors
Description	converts a boolean vector to an integer with one bit per element
Efficiency	good for compact boolean vectors. Medium for broad boolean vectors

```
// Example:
Vec4i   a(10, 11, 12, 13);
Vec4i   b(12, 11, 10,  9);
Vec4ib  f = a > b;        // (false, false, true, true)
uint8_t g = to_bits(f);   // = 0b1100
// The order is not reversed, but in the comments above,
// the vector elements are listed in little endian order,
// while the binary number is written in big endian order.
```

Method	load_bits
Defined for	all boolean vectors
Description	converts an integer bit-field to a boolean vector
Efficiency	good for compact boolean vectors. Medium for broad boolean vectors

```
// Example:
uint8_t a = 0b11000010;    // binary number
Vec8fb  b;                 // boolean vector
b.load_bits(a);
// b = (false, true, false, false, false, false, true, true)
// The order is not reversed, but in the comments above,
// the vector elements are listed in little endian order,
// while the binary number is written in big endian order.
```

Method	conversion between boolean vectors of same size and element size
Defined for	Vec4ib ↔ Vec4fb Vec8ib ↔ Vec8fb Vec16ib ↔ Vec16fb Vec2qb ↔ Vec2db Vec4qb ↔ Vec4db Vec8qb ↔ Vec8db
Description	Boolean vectors for use with different types of vectors with the same bit size can be converted to each other.
Efficiency	good

```
// Example:
Vec4i  a(0,1,2,3);
Vec4i  b(4,3,2,1);
Vec4ib f = a > b;      // f = (false,false,false,true)
Vec4fb g = Vec4fb(f); // g = (false,false,false,true)
```

Method	conversion from boolean vectors to integer vectors of the same size and element size
Defined for	broad boolean vectors only.
Description	broad boolean vectors can be converted to integer vectors of the same size and bit size. The result will be -1 for true and 0 for false. Avoid this method if compact boolean vectors may be used. Conversion the other way, e.g. from Vec4i to Vec4ib is possible for broad boolean vectors if the input vector contains -1 for true and 0 for false, but the result is implementation dependent and possibly wrong and inconsistent if the input vector contains any other values than 0 and -1. To prevent errors, it is recommended to use a comparison instead for converting an integer vector to a boolean vector.
Efficiency	good

```
// This example works only for broad boolean vectors
Vec4i  a(0,1,2,3);
Vec4i  b(4,3,2,1);
Vec4ib f = a > b;      // f = (false,false,false,true)
Vec4i  g = Vec4i(f);   // g = (0, 0, 0, -1)
```

Chapter 7

Permute, blend, lookup, gather and scatter functions

7.1 Permute functions

Function	permute..<i0, i1, ...>(vector)
Defined for	all integer and floating point vector classes
Description	permutes vector elements
Efficiency	depends on parameters and instruction set

The permute functions can move any element of a vector into any position, copy the same element to multiple positions, and set any element to zero.

The name of the permute function is "permute" followed by the number of vector elements, for example permute4 for Vec4i. The permute function for a vector of n elements has n indexes, which are entered as template parameters in angle brackets. Each index indicates the desired contents of the corresponding element in the result vector. An index i in the interval $0 \leq i \leq n-1$ indicates that element number i from the input vector should be placed in the corresponding position in the result vector. An index $i = -1$ gives a zero in the corresponding position. An index $i =$ V_DC means don't care. This will give whatever implementation is fastest, regardless of what value it puts in this position. The value you get with "don't care" may be different for different implementations or different instruction sets.

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b = permute4<2,2,3,0>(a);    // b = (12, 12, 13, 10)
Vec4i c = permute4<-1,-1,1,1>(a); // c = ( 0,  0, 11, 11)
```

The indexes in angle brackets must be compile-time constants, they cannot contain variables or function calls. If you need variable indexes then use the lookup functions instead (see page 46).

The permute functions are using advanced metaprogramming techniques in order to find the optimal combination of instructions that fit the given set of indexes and the specified instruction set. The optimization criteria include number of instructions, instruction latency, and data cache use. The metaprogramming may produce extra code when compiling in debug mode, but this extra code is eliminated when compiling for release mode with optimization on. The call to a permute function is reduced to just one or a few machine instructions in favorable cases.

The performance is generally good when the instruction set SSSE3 or higher is enabled. The performance for permuting vectors of 16-bit integers is medium, and the performance for permuting vectors of 8-bit integers is poor for instruction sets lower than SSSE3. You may get the best performance with instruction set AVX2 or AVX512VL.

7.2 Blend functions

Function	blend..<i0, i1, ...>(vector, vector)
Defined for	all integer and floating point vector classes
Description	permutes and blends elements from two vectors
Efficiency	depends on parameters and instruction set

The blend functions are similar to the permute functions, but with two input vectors. The name of the function is "blend" followed by the number of vector elements, for example blend4 for Vec4i. The blend function for a vector of n elements has n indexes, which are entered as template parameters in angle brackets. Each index indicates the desired contents of the corresponding element in the result vector. The indexes must be compile-time constants. An index i in the interval $0 \leq i \leq n-1$ indicates that element number i from the first input vector should be placed in the corresponding position in the result vector. An index i in the interval $n \leq i \leq 2 \cdot n - 1$ indicates that element number $i - n$ from the second input vector should be placed in the corresponding position in the result vector. An index $i = -1$ gives a zero in the corresponding position. An index $i =$ V_DC means don't care.

The blend functions are using metaprogramming in the same way as the permute functions. The performance is similar to the permute functions, or slightly lower.

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = blend4<4,0,6,3>(a, b); // c = (20, 10, 22, 13)
```

There are different methods you can use if you want to blend inputs from more than two vectors:

1. A binary tree of blend calls, where unused values are set to V_DC meaning don't care.

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c(30, 31, 32, 33);
Vec4i d(40, 41, 42, 43);
Vec4i r = blend4<0,5,V_DC,V_DC>(a, b);// r = (10,21,?,?)
Vec4i s = blend4<V_DC,V_DC,2,7>(c, d);// s = (?,?,32,43)
Vec4i t = blend4<0,1,6,7>(r, s);      // t = (10,21,32,43)
```

2. Set unused values to zero, then OR the results.

```
// Example:
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c(30, 31, 32, 33);
Vec4i d(40, 41, 42, 43);
Vec4i r = blend4<0,5,-1,-1>(a, b);// r = (10,21,0,0)
Vec4i s = blend4<-1,-1,2,7>(c, d);// s = (0,0,32,43)
Vec4i t = r | s;                  // t = (10,21,32,43)
```

3. If the input vectors are stored sequentially in memory then use the lookup functions shown below.

7.3 Lookup functions

Function	Vec16c lookup16(Vec16c, Vec16c) Vec32c lookup32(Vec32c, Vec32c) Vec64c lookup64(Vec64c, Vec64c) Vec8s lookup8(Vec8s, Vec8s) Vec16s lookup16(Vec16s, Vec16s) Vec32s lookup32(Vec32s, Vec32s) Vec4i lookup4(Vec4i, Vec4i) Vec8i lookup8(Vec8i, Vec8i) Vec16i lookup16(Vec16i, Vec16i) Vec4q lookup4(Vec4q, Vec4q) Vec8q lookup8(Vec8q, Vec8q)
Defined for	Vec16c, Vec32c, Vec64c, Vec8s, Vec16s, Vec32s, Vec4i, Vec8i, Vec16i, Vec4q, Vec8q
Description	Permutation with variable indexes. The first input vector contains the indexes, the second input vector is the data source. Each index must be in the range $0 \leq i \leq n-1$ where n is the number of elements in a vector.
Efficiency	Vec16i, Vec8q: Good for AVX512F, medium otherwise. Vec64c, Vec32s: Good for AVX512VBMI, medium for AVX512BW, poor otherwise. Vec32c, Vec16s, Vec8i, Vec4i, Vec4q: Good for AVX2, medium otherwise. Vec16c, Vec8s: Good for SSSE3, poor otherwise.

Function	Vec16c lookup32(Vec16c, Vec16c, Vec16c) Vec64c lookup128(Vec64c, Vec64c, Vec64c) Vec8s lookup16(Vec8s, Vec8s, Vec8s) Vec32s lookup64(Vec32s, Vec32s, Vec32s) Vec4i lookup8(Vec4i, Vec4i, Vec4i) Vec16i lookup32(Vec16i, Vec16i, Vec16i)
Defined for	Vec16c, Vec64c, Vec8s, Vec32s, Vec4i, Vec16i
Description	Blend with variable indexes. The first input vector contains the indexes, the following two input vectors contain the data source. Each index must be in the range $0 \leq i \leq 2 \cdot n-1$ where n is the number of elements in each vector.
Efficiency	Vec4i, Vec8s: Good for AVX2, medium or poor otherwise. Vec16i: Good for AVX512, medium or poor otherwise. Vec64c, Vec32s: Good for AVX512VBMI, medium for AVX512BW, poor otherwise. Vec16c, Vec8s: Good for SSSE3, poor otherwise.

Function	Vec4i lookup16(Vec4i, Vec4i, Vec4i, Vec4i, Vec4i) Vec16i lookup64(Vec16i, Vec16i, Vec16i, Vec16i, Vec16i) Vec64c lookup256(Vec64c, Vec64c, Vec64c, Vec64c, Vec64c) Vec32s lookup128(Vec32s, Vec32s, Vec32s, Vec32s, Vec32s)
Defined for	Vec4i, Vec32s, Vec64c
Description	Blend with variable indexes. The first input vector contains the indexes, the following four input vectors contain the data source. Each index must be in the range $0 \leq i \leq 4 \cdot n - 1$ where n is the number of elements in each vector.
Efficiency	Vec4i: Good for AVX2, medium otherwise. Vec16i: Good for AVX512, medium or poor otherwise. Vec64c, Vec32s: Good for AVX512VBMI, medium for AVX512BW, poor otherwise.

Function	Vec8h lookup8(Vec8s, Vec8h) Vec16h lookup16(Vec16s, Vec16h) Vec32h lookup32(Vec32s, Vec32h) Vec4f lookup4(Vec4i, Vec4f) Vec8f lookup8(Vec8i, Vec8f) Vec16f lookup16(Vec16i, Vec16f) Vec2d lookup2(Vec2q, Vec2d) Vec4d lookup4(Vec4q, Vec4d) Vec8d lookup8(Vec8q, Vec8d)
Defined for	all floating point vector classes
Description	Permutation of floating point vectors with integer indexes. Each index must be in the range $0 \leq i \leq n - 1$ where n is the number of elements in a vector.
Efficiency	good for AVX2 and later, medium for lower instruction sets

Function	Vec8h lookup16(Vec8s, Vec8h, Vec8h) Vec4f lookup8(Vec4i, Vec4f, Vec4f) Vec2d lookup4(Vec2q, Vec2d, Vec2d)
Defined for	Vec4f, Vec2d
Description	Blend of floating point vectors with integer indexes. Each index must be in the range $0 \leq i \leq 2 * n - 1$ where n is the number of elements in a vector.
Efficiency	medium

Function	Vec16c lookup<n>(Vec16c index, void const * table) Vec32c lookup<n>(Vec32c index, void const * table) Vec8s lookup<n>(Vec8s index, void const * table) Vec16s lookup<n>(Vec16s index, void const * table) Vec4i lookup<n>(Vec4i index, void const * table) Vec8i lookup<n>(Vec8i index, void const * table) Vec16i lookup<n>(Vec16i index, void const * table) Vec4q lookup<n>(Vec4q index, void const * table) Vec8q lookup<n>(Vec8q index, void const * table) Vec8h lookup<n>(Vec8s index, void const * table) Vec16h lookup<n>(Vec16s index, void const * table) Vec32h lookup<n>(Vec32s index, void const * table) Vec4f lookup<n>(Vec4i index, float const * table) Vec8f lookup<n>(Vec8i const & index, float const * table) Vec16f lookup<n>(Vec16i const & index, float const * table) Vec2d lookup<n>(Vec2q index, double const * table) Vec4d lookup<n>(Vec4q const & i, double const * table) Vec8d lookup<n>(Vec8q const & i, double const * table)
Defined for	all floating point and signed integer vector classes
Description	Permute, blend, table lookup or gather data from array with an integer vector of indexes. Each index must be in the range $0 \leq i \leq n-1$, where n is indicated as a template parameter. n must be a positive compile-time constant.
Efficiency	good for AVX2 and later, medium for lower instruction sets

The lookup functions are similar to the permute and blend functions, but with variable indexes. They cannot be used for setting an element to zero, and there is no "don't care" option. The lookup functions can be used for several purposes:

1. permute with variable indexes
2. blend with variable indexes
3. blend from more than two sources
4. table lookup
5. gather non-contiguous data from an array

The index is always an integer vector. The input can be one or more vectors or an array. The result is a vector of the same type as the input. All elements in the index vector must be in the specified range. The behavior for an index out of range is implementation-dependent and may give any value for the corresponding element. The function may in some cases read up to one vector size past the end of the table for the sake of efficient permutation.

The lookup functions are not defined for unsigned integer vector types, but the corresponding signed versions can be used. You don't have to worry about overflow when converting unsigned integers to signed here, as long as the result vector is converted back to unsigned.

```
// Example of permutation with variable indexes:
Vec4f a(1.0, 1.1, 1.2, 1.3);
Vec4i b(2, 3, 3, 0);
Vec4f c = lookup4(b, a);  // c = (1.2, 1.3, 1.3, 1.0)
```

```
// Example of blending with variable indexes:
Vec4f a(1.0, 1.1, 1.2, 1.3);
Vec4f b(2.0, 2.1, 2.2, 2.3);
Vec4i c(4, 3, 2, 7);
Vec4f d = lookup4(c,a,b); // d = (2.0, 1.3, 1.2, 2.3)

// Example of blending from more than two sources:
float sources[12] = {
1.0,1.1,1.2,1.3,2.0,2.1,2.2,2.3,3.0,3.1,3.2,3.3};
Vec4i i(11, 0, 5, 5);
Vec4f c = lookup<12>(i, sources); // c = (3.3,1.0,2.1,2.1)
```

A function with a limited number of possible input values can be replaced by a lookup table. This is useful if table lookup is faster than calculating the function. The following example has a table of the function $y = x^2 - 1$

```
// Table of the function y = x*x-1
int table[6] = {-1,0,3,8,15,24};
Vec4i x(4,2,0,5);
Vec4i y = lookup<6>(x, table);  // y = (15, 3, -1, 24)

// Example of gathering non-contiguous data from an array:
float x[16] = { ... };
Vec4i i(0,4,8,12);
Vec4f y = lookup<16>(i, x); // y = (x[0],x[4],x[8],x[12])
```

7.4 Gather functions

Function	Vec4i gather4i<indexes>(void const * table) Vec8i gather8i<indexes>(void const * table) Vec16i gather16i<indexes>(void const * table) Vec2q gather2q<indexes>(void const * table) Vec4q gather4q<indexes>(void const * table) Vec8q gather8q<indexes>(void const * table) Vec4f gather4f<indexes>(void const * table) Vec8f gather8f<indexes>(void const * table) Vec16f gather16f<indexes>(void const * table) Vec2d gather2d<indexes>(void const * table) Vec4d gather4d<indexes>(void const * table) Vec8d gather8d<indexes>(void const * table)
Defined for	Vec4i, Vec8i, Vec16i, Vec2q, Vec4q, Vec8q, Vec4f, Vec8f, Vec16f, Vec2d, Vec4d, Vec8d
Description	Load non-contiguous data from a table. Indexes cannot be negative. There is no option for zeroing or don't care. If you need variable indexes, then use the lookup functions instead. The function may read a full vector and permute it if all indexes are smaller than the vector size.
Efficiency	medium

```
// Example:
int tab[8] = {10,11,12,13,14,15,16,17};
Vec4i a = gather4i<6,4,4,0>(tab);
```

```
// a = (16, 14, 14, 10);
```

7.5 Scatter functions

Function	scatter<indexes>(Vec4i data, void * array) scatter<indexes>(Vec8i data, void * array) scatter<indexes>(Vec16i data, void * array) scatter<indexes>(Vec2q data, void * array) scatter<indexes>(Vec4q data, void * array) scatter<indexes>(Vec8q data, void * array) scatter<indexes>(Vec4f data, float * array) scatter<indexes>(Vec8f data, float * array) scatter<indexes>(Vec16f data, float * array) scatter<indexes>(Vec2d data, double * array) scatter<indexes>(Vec4d data, double * array) scatter<indexes>(Vec8d data, double * array)
Defined for	Vec4i, Vec8i, Vec16i, Vec2q, Vec4q, Vec8q, Vec4f, Vec8f, Vec16f, Vec2d, Vec4d, Vec8d
Description	Store vector elements into non-contiguous positions in an array. Each vector element is stored in the array position indicated by the corresponding index. An element is not stored if the corresponding index is negative.
Efficiency	Medium for 512 bit vectors if AVX512F instruction set supported. Medium for 256 bit vectors if AVX512F, or better AVX512VL, supported. Medium for 128 bit vectors if AVX512VL supported. Poor otherwise.

```
// Example:
Vec8i a(10,11,12,13,14,15,16,17);
int array[10] = {0};
scatter<5,4,3,2,-1,-1,7,0>(a, array);
// array = (17,0,13,12,11,10,0,16,0,0)
```

Function	scatter(Vec4i index, uint32_t limit, Vec4i data, void * array) scatter(Vec8i index, uint32_t limit, Vec8i data, void * array) scatter(Vec16i index, uint32_t limit, Vec16i data, void * array) scatter(Vec2q index, uint32_t limit, Vec2q data, void * array) scatter(Vec4i index, uint32_t limit, Vec4q data, void * array) scatter(Vec4q index, uint32_t limit, Vec4q data, void * array) scatter(Vec8i index, uint32_t limit, Vec8q data, void * array) scatter(Vec8q index, uint32_t limit, Vec8q data, void * array) scatter(Vec4i index, uint32_t limit, Vec4f data, float * array) scatter(Vec8i index, uint32_t limit, Vec8f data, float * array) scatter(Vec16i index, uint32_t limit, Vec16f data, float * array) scatter(Vec2q index, uint32_t limit, Vec2d data, double * array) scatter(Vec4i index, uint32_t limit, Vec4d data, double * array) scatter(Vec4q index, uint32_t limit, Vec4d data, double * array) scatter(Vec8i index, uint32_t limit, Vec8d data, double * array) scatter(Vec8q index, uint32_t limit, Vec8d data, double * array)
Defined for	Vec4i, Vec8i, Vec16i, Vec2q, Vec4q, Vec8q, Vec4f, Vec8f, Vec16f, Vec2d, Vec4d, Vec8d
Description	Store vector elements into non-contiguous positions in an array. Each vector element is stored in the array position indicated by the corresponding element of the index vector. An element is not stored if the corresponding index is negative or bigger than or equal to the limit. The limit will typically be the size of the array.
Efficiency	Medium for 512 bit vectors if AVX512F instruction set supported. Medium for 256 bit vectors if AVX512F, or better AVX512VL, supported. Medium for 128 bit vectors if AVX512VL supported. Poor otherwise.

```
// Example:
Vec8i a(10,11,12,13,14,15,16,17);
Vec8i x(5,4,3,2,-1,99,7,0);
int array[10] = {0};
scatter(x, 5, a, array);
// array = (17,0,13,12,11,0,0,0,0,0)
```

The scatter functions are useful for writing sparse arrays. If you have more dense arrays, then it may be more efficient to permute the vector and then store the whole vector into the array.

If you want to permute a dataset that is too big for the permute and blend functions, then it is better to use lookup or gather functions than to use scatter functions.

Chapter 8

Mathematical functions

Function	exponent
Defined for	all floating point vector classes
Description	extracts the exponent part of a floating point number. The result is an integer vector. exponent(a) = floor(log2(abs(a))). The value for a = 0 is implementation dependent. Subnormal numbers are not supported.
Efficiency	medium

```
// Example:
Vec4f a(1.0f, 2.0f, 3.0f, 4.0f);
Vec4i b = exponent(a);  // b = (0, 1, 1, 2)
```

Function	fraction
Defined for	all floating point vector classes
Description	extracts the fraction part of a floating point number. a = pow(2, exponent(a)) * fraction(a) The results for a = 0, subnormal, INF, or NAN are implementation dependent.
Efficiency	medium

```
// Example:
Vec4f a(2.0f, 3.0f, 4.0f, 5.0f);
Vec4f b = fraction(a);  // b = (1.00f, 1.50f, 1.00f, 1.25f)
```

Function	exp2
Defined for	all floating point vector classes
Description	calculates integer powers of 2. The input is an integer vector, the output is a floating point vector. Overflow gives +INF, underflow gives zero. This function will never produce subnormals, and never raise exceptions
Efficiency	medium

```
// Example:
Vec4i a(-1, 0, 1, 2);
Vec4f b = exp2(a);      // b = (0.5f, 1.0f, 2.0f, 4.0f)
```

Function	mul_add nmul_add mul_sub
Defined for	all floating point vector classes
Description	mul_add(a,b,c) = a*b+c nmul_add(a,b,c) = -a*b+c mul_sub(a,b,c) = a*b-c These functions use fused multiply-and-add (FMA) instructions if available. Some compilers use FMA instructions automatically for expressions like a*b+c. Use these functions for optimal performance on all compilers or to specify calculation order, etc.
Precision	The intermediate product a*b is calculated with unlimited precision if the FMA instruction set is enabled.
Efficiency	good

Function	fremainder fmodulo
Defined for	single and double precision floating point vectors
Description	vector fremainder(vector n, double d) vector fmodulo(vector n, double d) n (numerator) is reduced modulo d (denominator). The same denominator is applied to all vector elements. The result is within the following limits: fremainder: -d/2 <= result < d/2 fmodulo: 0 <= result < d Note that fmodulo never gives a negative result even if n is negative, unlike the standard fmod function.
Precision	d is double precision, even if n is single precision. The full double precision of d is utilized. It is recommended to calculate d with double precision, even if n is single precision. Precision and efficiency is best if the FMA instruction set is enabled.
Efficiency	medium

8.1 Floating point categorization functions

Function	is_finite
Defined for	all floating point vector classes
Description	returns a boolean vector with true for elements that are normal, subnormal or zero, false for INF and NAN
Efficiency	medium

```
// Example:
Vec4f  a( 0.0f, 1.0f, 2.0f, 3.0f);
Vec4f  b(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f  c = a / b;
Vec4fb d = is_finite(c);  // d = (true, false, true, true)
```

Function	is_inf
Defined for	all floating point vector classes
Description	returns a boolean vector with true for elements that are +INF or -INF, false for all other values, including NAN
Efficiency	good

```
// Example:
Vec4f  a( 0.0f, 1.0f, 2.0f, 3.0f);
Vec4f  b(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f  c = a / b;
Vec4fb d = is_inf(c);  // d = (false, true, false, false)
```

Function	is_nan
Defined for	all floating point vector classes
Description	returns a boolean vector with true for all types of NAN, false for all other values, including INF
Efficiency	good

```
// Example:
Vec4f  a(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f  b = sqrt(a);
Vec4fb c = is_nan(b);  // c = (true, false, false, false)
```

Function	is_subnormal
Defined for	all floating point vector classes
Description	returns a boolean vector with true for subnormal (denormal) vector elements, false for normal numbers, INF and NAN
Efficiency	medium

```
// Example:
Vec4f  a(1.0f, 1.0E-10f, 1.0E-20f, 1.0E-30f);
Vec4f  b = a * a;             // b = (1.0f, 1.E-20f, 1.E-40f, 0.f)
Vec4fb c = is_subnormal(b); // c = (false,false,true,false)
```

Function	is_zero_or_subnormal
Defined for	all floating point vector classes
Description	returns a boolean vector with true for zero and subnormal (denormal) vector elements, false for nonzero normal numbers, INF and NAN
Efficiency	good

```
// Example:
Vec4f  a(1.0f, 1.0E-10f, 1.0E-20f, 1.0E-30f);
Vec4f  b = a * a;             // b = (1.0f, 1.E-20f, 1.E-40f, 0.f)
Vec4fb c = is_zero_or_subnormal(b); // c = (false,false,true,true)
```

Function	infinite8h, infinite16h, infinite32h, infinite4f, infinite8f, infinite16f, infinite2d, infinite4d, infinite8d
Defined for	all floating point vector classes
Description	returns positive infinity
Efficiency	good

```
// Example:
Vec4f  a = infinite4f(); // a = (INF, INF, INF, INF)
```

Function	nan8h(unsigned int n) nan16h(unsigned int n) nan32h(unsigned int n) nan4f(unsigned int n) nan8f(unsigned int n) nan16f(unsigned int n) nan2d(unsigned int n) nan4d(unsigned int n) nan8d(unsigned int n)
Defined for	all floating point vector classes
Description	returns not-a-number (NAN). The optional parameter n may be used for error tracing. The maximum value of n is 0x003FFFFF for single and double precision, and 0x1FF for half precision. This function generates a quiet NAN in the following way: Half precision: The value n is OR'ed with 0x200 to set the quiet bit, and inserted as a payload. Single precision: The value n is OR'ed with 0x400000 to set the quiet bit, and inserted as a payload. Double precision: The value n is shifted 29 places to the left for the sake of compatibility with single precision. The value is then OR'ed with $1 << 51$ to set the quiet bit. This parameter n (including the quiet bit) can be retrieved later by the function nan_code (page 66).
Efficiency	good

```
// Example:
Vec4f a = nan4f(); // a = (NAN, NAN, NAN, NAN)
```

8.2 Floating point control word manipulation functions

MXCSR is a control word that controls floating point exceptions, rounding mode and subnormal numbers for single and double precision floating point numbers. There is one MXCSR for each thread. The MXCSR has the following bits:

Bit index	meaning
0	Invalid Operation Flag
1	Denormal (subnormal) Flag
2	Divide-by-Zero Flag
3	Overflow Flag
4	Underflow Flag
5	Precision Flag
6	Denormals (subnormals) Are Zeros
7	Invalid Operation Mask
8	Denormal (subnormal) Operation Mask
9	Divide-by-Zero Mask
10	Overflow Mask
11	Underflow Mask
12	Precision Mask
13-14	Rounding control: 00: round to nearest or even 01: round down towards -infinity 10: round up towards +infinity 11: round towards zero (truncate) If the rounding mode is temporarily changed then it must be set back to 00 for the vector class library to work correctly.
15	Flush to Zero

Please see programming manuals from Intel or AMD for further explanation.

Function	get_control_word
Description	reads the MXCSR control word
Efficiency	medium

```
// Example:
int m = get_control_word();  // default value m = 0x1F80
```

Function	set_control_word(n)
Description	writes the MXCSR control word
Efficiency	medium

```
// Example:
// Enable overflow and divide by zero exceptions:
set_control_word(0x1980);
```

Function	reset_control_word
Description	sets the MXCSR control word to the default value
Efficiency	medium

```
// Example:
reset_control_word();
```

Function	no_subnormals
Description	Disables the use of subnormal (denormal) values. Floating point numbers with an absolute value below 1.18E-38 for single precision or 2.22E-308 for double precision are represented by subnormal numbers. The handling of subnormal numbers is extremely time-consuming on many CPUs. The no_subnormals function sets the "denormals are zeros" and "flush to zero" mode to avoid the use of subnormal numbers. It is recommended to call this function at the beginning of each thread in order to improve the speed of mathematical calculations if very low numbers are likely to occur. This function has no effect on half precision numbers.
Efficiency	medium

```
// Example:
no_subnormals();
```

8.3 Standard mathematical functions

Standard mathematical functions such as logarithms, exponential functions, power, trigonometric functions, etc. for vectors are available in two versions: as inline code and as an external function library provided by Intel. These functions all take vectors as input and produce vectors as output.

The use of vector math functions is straightforward:

Example 8.1.

```
#include <stdio.h>
#include "vectorclass.h"
#include "vectormath_trig.h"    // trigonometric functions

int main() {
    Vec4f a(0.0f, 0.5f, 1.0f, 1.5f);// define vector
    Vec4f b = sin(a);               // sine function
    // b = (0.0000f, 0.4794f, 0.8415f, 0.9975f)

    // output results:
    for (int i = 0; i < b.size(); i++) {
        printf("%6.4f ", b[i]);
    }
    printf("\n");
    return 0;
}
```

The inline versions and the external library versions are using different calculation methods. The inline versions may be faster in some cases, while the external library versions may be faster in other cases. Both versions are many times faster than standard (scalar) math function libraries.

The available vector math functions are listed below. The efficiency is listed as poor because mathematical functions take more time to execute than most other functions, but they are still much faster than scalar alternatives. The details listed apply to the inline version. Details for the library version may be sought in the documentation for the Intel compiler.

8.4 Inline mathematical functions

The inline mathematical functions are available by including the appropriate header file, e. g. vectormath_exp.h for powers, logarithms and exponential functions, and vectormath_trig.h for trigonometric functions. An advantage of the inline version is that the compiler can optimize the code across function calls, eliminate common sub-expressions, etc. The disadvantage is that you may get multiple instances of the same function taking up space in the code cache.

The accuracy is good. The calculation error is typically below 2 ULP (Unit in the Last Place = least significant bit) on the output. (The relative value of one ULP is 2^{-52} for double precision and 2^{-23} for single precision). Where a function is steep, the maximum error corresponds to 1 ULP at the input. Cases where the error can exceed 3 ULP are mentioned under the specific function.

The functions do not generate exceptions or set errno when an input is out of range. This would be inefficient and it would be problematic for the error handler to detect which vector element caused the error. Instead, the functions return INF (infinity) or NAN (not a number) in case of error. Generally, an overflow will produce INF. A negative overflow produces -INF. An underflow towards zero returns 0. Other errors produce NAN. An efficient way of detecting errors is to let the INF and NAN codes propagate through the calculations and detect the error at the end of a series of calculations as explained on page 88. It is possible to include an error code in a NAN and detect it with the function nan_code on page 66.

Note that many of the inline math functions do not support subnormal numbers. Subnormal numbers may be treated as zero by the logarithm, exponential, power, and root functions. It is recommended to set the "denormals are zero" and "flush to zero" flags by calling the function $\mathrm{no\_subnormals()}$ first (see page 56). This may speed up some calculations and give more consistent results.

A description of each mathematical function is given below.

8.5 Using an external library for mathematical functions

A function library made by Intel called SVML (Short Vector Math Library) can be used as an alternative to the inline mathematical functions. SVML is a highly optimized function library that calculates mathematical functions on vectors.

The SVML library is part of an Intel compiler installation. The vector class library provides a header file named vectormath_lib.h that makes it possible to use the Intel SVML library with other compilers. The SVML library is optimized for Intel processors, but it works well with AMD processors as well according to my tests, unless you are using the Intel ICC or ICL compiler (named "classic"). Use the newer Intel ICPX compiler instead, or any other compiler. The SVML library is available for all platforms relevant to the vector class library.

The SVML library for Windows can be obtained in the following way:
Install the Intel C++ compiler. You need the files named svml_dispmt.lib and libircmt.lib. These files can be found in the installation directory, for example:
C:\Program Files (x86)\Intel\oneAPI\compiler\2022.1.0\windows\compiler\lib\intel64_win

Note that there is a 32-bit version and a 64-bit version of each library. We generally prefer to compile vector code for 64-bit mode, so you will probably need the 64-bit versions only. You also need the library file svmlpatch.lib which you can find at the VCL Github site under miscellaneous.

svml_dispmt.lib contains the mathematical vector functions. libircmt.lib contains a function dispatcher used by svml_dispmt.lib. The purpose of svmlpatch.lib is to fix a non-standard calling convention in the SVML library. svmlpatch.lib is only needed in 64-bit mode Windows.

Copy the library files svml_dispmt.lib, svml_dispmt.lib, and svmlpatch.lib to a suitable location and

add them to your C++ project.

The SVML library for Linux can be obtained in the following way:
Install the Intel C++ compiler. You need the files named libsvml.a and libirc.a. These files can be found in the installation directory, for example:
~/intel/oneapi/compiler/2022.1.0/linux/compiler/lib/intel64_lin/

Note that there is a 32-bit version and a 64-bit version of each library. We generally prefer to compile vector code for 64-bit mode, so you will probably need the 64-bit version only.

libsvml.a contains the mathematical vector functions, and libircmt.a contains a function dispatcher used by libsvml.a. Copy these two library files to a suitable location and add them to your C++ project.

Using the library functions in vector code:
Include the header file vectormath_lib.h if you want to use the SVML library. Do not include vectormath_exp.h, vectormath_trig.h, or vectormath_hyp.h. It is not possible to mix the two kinds of mathematical functions (inline and library) in the same C++ file. The available vector math functions are listed below.

8.6 Powers, exponential functions and logarithms

Function	pow(vector, vector), pow(vector, scalar)
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	pow(a,b) = a^b See also faster alternatives below for integer and rational powers.
Range	Subnormal numbers are treated as zero. The result is NAN if a is negative and b is not an integer. NAN's are always propagated by the inline version of pow, even in cases where the IEEE 754 standard specifies otherwise. The library version may fail to propagate NANs in the cases pow(NAN,0) and pow(1,NAN).
Precision	better than (0.8*abs(b)+2) ULP
Efficiency	poor

```
// Example:
Vec4f a( 1.0f,  2.0f, 3.0f, 4.0f);
Vec4f b( 0.0f, -1.0f, 0.5f, 2.0f);
Vec4f c = pow(a, b);
// c = (1.0000, 0.5000, 1.7321, 16.0000)
Vec4f d = pow(a, 2.4f);
// d = (1.0000, 5.2780, 13.9666, 27.8576)
```

Function	pow(vector, int)
Defined for	all floating point vector classes
Inline version	no extra header file required
Library version	not available
Description	see page 28
Efficiency	medium

```
// Example:
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
int   b = 3;
```

```
Vec4f c = pow(a, b);  // c = (0.0f, 1.0f, 8.0f, 27.0f)
```

Function	pow_const(vector, const int)
Defined for	all floating point vector classes
Inline version	no extra header file required
Library version	not available
Description	see page 28
Efficiency	medium, often better than pow(vector, int)

```
// Example:
Vec4f a(0.0f, 1.0f, 2.0f, 3.0f);
Vec4f c = pow_const(a, 3);  // c = (0.0f, 1.0f, 8.0f, 27.0f)
```

Function	pow_ratio(vector x, const int a, const int b)
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	not available
Description	Raises all elements of x to the rational power a/b. a and b must be compile-time constant integers.
Range	x may be zero only if a and b are positive. x may be negative only if b is odd. The range is the same as for cbrt (page 62) if b is 3. The result when x is infinite may be NAN in some cases. Subnormal numbers are treated as zero in some cases.
Precision	slightly imprecise for extreme values of a due to accumulating rounding errors. The precision is similar to the cbrt function when b is 3 or 6.
Efficiency	Quite good for b = 1, 2, 4, or 8. Reasonable for b = 3 or 6. No better than pow for other values of b.

```
// Example:
Vec4f a(1.0f, 2.0f, 3.0f, 4.0f);
// Reciprocal square root
Vec4f b = pow_ratio(a, -1, 2);  // c = (1.0, 0.707, 0.577, 0.500)
```

Function	exp
Defined for	all floating point vector classes
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	exponential function e^x
Range	double: abs(x) <708.39. float: abs(x) <87.3
Efficiency	Poor. The performance of the inline version for single precision vectors (Vec16f etc.) is better when the instruction set AVX512ER is supported. The performance can be improved further, at a slight loss of precision, when VCL_FASTEXP is defined in addition to AVX512ER.

```
// Example:
#include "vectormath_exp.h"
Vec16f a, b;
b = exp(a);
```

Function	expm1
Defined for	all floating point vector classes
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	$e^x - 1$. Useful to avoid loss of precision if x is close to 0
Range	double: abs(x) < 708.39. float: abs(x) < 87.3
Efficiency	Poor. (not improved with AVX512ER)

Function	exp2
Defined for	all floating point vector classes
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	2^x
Range	double: abs(x) < 1020. float: abs(x) < 27.
Efficiency	The performance of the inline version is good for single precision vectors if instruction set AVX512ER is supported. (VCL_FASTEXP is not needed). Use pow or pow_const instead if x is an integer.

Function	exp10
Defined for	all floating point vector classes
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	10^x
Range	double: abs(x) < 307.65. float: abs(x) < 37.9.
Efficiency	Poor. The performance of the inline version for single precision vectors (Vec16f etc.) is better when the instruction set AVX512ER is supported. The performance can be improved further, at a slight loss of precision, when VCL_FASTEXP is defined in addition to AVX512ER. Use pow or pow_const instead if x is an integer.

```
// Example:
#include "vectormath_exp.h"
Vec16f a, b;
b = exp10(a);
```

Function	log
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	natural logarithm
Range	The input must be a normal number. Subnormal numbers are treated as zero.
Efficiency	poor

Function	log1p
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	log(1+x) Useful to avoid loss of precision if x is close to 0
Range	x > -1
Efficiency	poor

Function	log2
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	logarithm base 2
Range	The input must be a normal number. Subnormal numbers are treated as zero.
Efficiency	poor

Function	log10
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	logarithm base 10
Range	The input must be a normal number. Subnormal numbers are treated as zero.
Efficiency	poor

Function	cbrt
Defined for	single and double precision floating point vectors
Inline version	vectormath_exp.h
Library version	vectormath_lib.h
Description	cube root
Range	float: 0, $\pm 10^{-28}..10^{28}$ double: 0, $\pm 10^{-200}..10^{200}$ The return value is 0 if abs(x) is too small
Precision	5 ULP
Efficiency	Faster than pow

8.7 Trigonometric functions and inverse trigonometric functions

All angles are in radians.

Function	sin
Defined for	all floating point vector classes
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	sine function
Range	abs(x) is limited to 314, 1.E7, 1.E15 for half, single, and double precision, respectively. This limit is lower if FMA instructions are not supported. The result is 0 for big x. The result may be 0 or NAN when the input is infinity depending on the implementation.
Efficiency	poor

```
// Example:
Vec4f a(0.0f, 0.5f, 1.0f, 1.5f);// define vector
Vec4f b = sin(a);               // sine function
// b = (0.0000f, 0.4794f, 0.8415f, 0.9975f)
```

Function	cos
Defined for	all floating point vector classes
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	cosine function
Range	abs(x) is limited to 314, 1.E7, 1.E15 for half, single, and double precision, respectively. This limit is lower if FMA instructions are not supported. The result is 1 for big x. The result may be 1 or NAN when the input is infinity depending on the implementation
Efficiency	poor

Function	sincos
Defined for	all floating point vector classes
Inline version	vectormath_trig.h
Library version	vectormath_lib.h (not with MS compiler)
Description	sine and cosine computed simultaneously.
Range	abs(x) is limited to 314, 1.E7, 1.E15 for half, single, and double precision, respectively. This limit is lower if FMA instructions are not supported. The result is 0 and 1 for big x. The result may or may not be NAN when the input is infinity.
Efficiency	faster than computing sin and cos separately

```
// Example:
Vec4f a(0.0f, 0.5f, 1.0f, 1.5f);
Vec4f s, c;
s = sincos(&c, a);
// s = (0.0000, 0.4794, 0.8415, 0.9975)
// c = (1.0000, 0.8776, 0.5403, 0.0707)
```

Function	tan
Defined for	all floating point vector classes
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	tangent function
Range	abs(x) is limited to 314, 1.E7, 1.E15 for half, single, and double precision, respectively. This limit is lower if FMA instructions are not supported. The result is 0 for big x. The result may be 0 or NAN when the input is infinity depending on the implementation.
Efficiency	poor

Function	sinpi, cospi, sincospi, tanpi
Defined for	all floating point vector classes
Inline version	vectormath_trig.h
Library version	vectormath_lib.h. Not with Intel compiler. sincospi not available
Description	sinpi(x) = sin(pi*x), etc. The ...pi functions are more accurate than the normal trigonometric functions when x is a multiple or simple fraction of π or x is high. For example, tanpi(0.5) gives INF while tan(pi*0.5) gives a high number less than INF because $\pi/2$ cannot be represented exactly. tanpi(n+0.5) gives INF for n even, and -INF for n odd, in accordance with the IEEE754-2019 standard. The standard for signed zero results is not necessarily followed.
Range	numerically high values of x are interpreted as even integers, giving exact results. The result may or may not be NAN when the input is infinity.
Efficiency	same as normal trigonometric functions, or slightly better

Function	asin
Defined for	single and double precision floating point vectors
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	inverse sine function
Range	$-1 \leq x \leq 1$
Efficiency	poor

Function	acos
Defined for	single and double precision floating point vectors
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	inverse cosine function
Range	$-1 \leq x \leq 1$
Efficiency	poor

Function	atan
Defined for	single and double precision floating point vectors
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	Inverse tangent
Range	Results between $-\pi/2$ and $\pi/2$
Efficiency	poor

Function	atan2
Defined for	single and double precision floating point vectors
Inline version	vectormath_trig.h
Library version	vectormath_lib.h
Description	Inverse tangent with two parameters, x and y, gives the angle to a point in the (x,y) plane
Range	Results between $-\pi$ and π The result of atan2(0,0) is 0 by convention
Efficiency	poor

8.8 Hyperbolic functions and inverse hyperbolic functions

Function	sinh
Defined for	single and double precision floating point vectors
Inline version	vectormath_hyp.h
Library version	vectormath_lib.h
Description	hyperbolic sine
Range	double: abs(x) $<$ 709. float: abs(x) $<$ 88.
Efficiency	poor

Function	cosh
Defined for	single and double precision floating point vectors
Inline version	vectormath_hyp.h
Library version	vectormath_lib.h
Description	hyperbolic cosine
Range	double: abs(x) $<$ 709. float: abs(x) $<$ 88.
Efficiency	poor

Function	tanh
Defined for	single and double precision floating point vectors
Inline version	vectormath_hyp.h
Library version	vectormath_lib.h
Description	hyperbolic tangent
Efficiency	poor

Function	asinh
Defined for	single and double precision floating point vectors
Inline version	vectormath_hyp.h
Library version	vectormath_lib.h
Description	inverse hyperbolic sine
Efficiency	poor

Function	acosh
Defined for	single and double precision floating point vectors
Inline version	vectormath_hyp.h
Library version	vectormath_lib.h
Description	inverse hyperbolic cosine
Efficiency	poor

Function	atanh
Defined for	single and double precision floating point vectors
Inline version	vectormath_hyp.h
Library version	vectormath_lib.h
Description	inverse hyperbolic tangent
Efficiency	poor

8.9 Other mathematical functions

Function	erf
Defined for	single and double precision floating point vectors
Inline version	not available
Library version	vectormath_lib.h
Description	error function
Efficiency	poor

Function	erfc
Defined for	single and double precision floating point vectors
Inline version	not available
Library version	vectormath_lib.h
Description	error function complement
Efficiency	poor

Function	erfinv
Defined for	single and double precision floating point vectors
Inline version	not available
Library version	vectormath_lib.h
Description	inverse error function
Efficiency	poor

Function	cdfnorm
Defined for	single and double precision floating point vectors
Inline version	not available
Library version	vectormath_lib.h
Description	cumulative normal distribution function
Efficiency	poor

Function	cdfnorminv
Defined for	single and double precision floating point vectors
Inline version	not available
Library version	vectormath_lib.h
Description	inverse cumulative normal distribution function
Efficiency	poor

Function	Vec8us nan_code(Vec8h) Vec16us nan_code(Vec16h) Vec32us nan_code(Vec32h) Vec4ui nan_code(Vec4f) Vec8ui nan_code(Vec8f) Vec16ui nan_code(Vec16f) Vec2uq nan_code(Vec2d) Vec4uq nan_code(Vec4d) Vec8uq nan_code(Vec8d)
Defined for	all floating point vector classes
Inline version	vectormath_exp.h
Library version	not available
Description	Extracts an error code hidden as payload in a NAN. This code can be generated with the functions nan4f etc. (page 55) and propagated through a series of calculations. When two NANs are combined (e.g. NAN1+NAN2), current processors propagate the first one. NANs produced by CPU instructions, such as 0./0. or sqrt(-1.) have a code of zero. NANs cannot propagate through integers and booleans. The return value is the payload including the quiet bit. For double precision, the value is shifted 29 places to the right for the sake of compatibility with single precision. The sign bit is ignored. The return value is 0 for inputs that are not NAN.
Efficiency	medium

Chapter 9

Performance considerations

9.1 Comparison of alternative methods for writing SIMD code

The SIMD (Single Instruction Multiple Data) instructions play an important role when software performance has to be optimized. Several different ways of writing SIMD code are discussed below.

Assembly code
Assembly programming is the ultimate way of optimizing code. Almost anything is possible in assembly code, but it is quite tedious and error-prone. There are thousands of different instructions, and it is quite difficult to remember which instruction belongs to which instruction set extension. Assembly code is difficult to document, difficult to debug, and difficult to maintain.

Intrinsic functions
Several compilers support intrinsic functions that are direct representations of machine instructions. A big advantage of using intrinsic functions rather than assembly code is that the compiler takes care of register allocation, function calling conventions, and other details that are difficult to keep track of when writing assembly code. Another advantage is that the compiler can optimize the code further by such methods as scheduling, interprocedural optimization, function inlining, constant propagation, common subexpression elimination, loop invariant code motion, induction variables, etc. Many of these optimization methods are rarely used in assembly code because they make the code unwieldy and unmanageable. Consequently, the combination on intrinsic functions and a good optimizing compiler can often produce more efficient code than what a decent assembly programmer would do.

A disadvantage of intrinsic functions is that these functions have long names that are difficult to remember and they make the code look awkward.

Intel vector classes
Intel has published a number of vector classes in the form of three C++ header files named fvec.h, dvec.h and ivec.h. These are simpler to use than the intrinsic functions, but unfortunately the Intel vector class files are poorly maintained; they provide only the most basic functionality; and Intel has done very little to promote, support, or develop them. The Intel vector classes have no way of converting data between arrays and vectors. This leaves us with no way of putting data into a vector other than specifying each element separately - which pretty much destroys the advantage of using vectors. The Intel vector classes work only with Intel and MS compilers.

The VCL vector class library
The present vector class library has several important features, listed on page 4. It provides the same level of optimization as the intrinsic functions, but it is much easier to use. This makes it possible to make optimal use of the SIMD instructions without the need to remember the thousands of different instructions or intrinsic functions. It also takes away the hassle of remembering which instruction belongs to which instruction set extension and making different code versions for different instruction sets.

Automatic vectorization
A good optimizing compiler is able to automatically transform linear code to vector code in simple cases. Typically, a good compiler will vectorize an algorithm that loops through an array and does some calculations on each array element.

Automatic vectorization is the easiest way of generating SIMD code, and I would very much recommend to use this method when it works. Automatic vectorization may fail or produce suboptimal code in the following cases:

- when the algorithm is too complex.
- when data have to be re-arranged in order to fit into vectors and it is not obvious to the compiler how to do this, or when other parts of the code needs to be changed to handle the re-arranged data.
- when it is not known to the compiler which data sets are bigger or smaller than the vector size.
- when it is not known to the compiler whether the size of a data set is a multiple of the vector size or not.
- when the algorithm involves calls to functions that are defined elsewhere or cannot be inlined and are not readily available in vector versions.
- when the algorithm involves many branches that are not easily vectorized.
- When the compiler cannot rule out that not-taken branches may generate false exceptions or other side effects.
- when floating point operations have to be reordered or transformed and it is not known to the compiler whether these transformations are permissible with respect to precision, overflow, etc.
- when functions are implemented with lookup tables.

The vector class library is intended as a good alternative when automatic vectorization fails to produce optimal code for any of these reasons.

9.2 Choice of compiler and function libraries

It is recommended to compile for 64-bit mode because this gives access to more memory and more registers. The CPU gives you access to only 8 vector registers in 32-bit mode, but 32 vector registers in 64-bit mode if the AVX512 instruction set is enabled. Compiler options are listed in table 9.2.

The vector class library has support for the following compilers:

Gnu C++ compiler
This compiler has produced very good optimizations in my tests. The Gnu compiler (g++) is available for all x86 and x86-64 platforms.

There are several versions of the Gnu compiler for the Windows platform. The version that comes with msys2 is recommended. The Cygwin64 version is not recommended because it is using a less efficient memory model.

Do not use the option -ffast-math or -ffinite-math-only on a Gnu compiler if you want to rely on INF and NAN because these options may disable the detection of INF and NAN.

Clang C++ compiler

This compiler has produced the best optimized code in my tests. The Clang compiler is available for all x86 and x86-64 platforms.

There are different versions of Clang available for the Windows platform. The msys2 version, and the version that comes as a plugin for Visual Studio are both recommended. The Cygwin64 version is not recommended because it is using a less efficient memory model.

Do not use the option -ffast-math or -ffinite-math-only on a Clang compiler if you want to rely on INF and NAN because these options may disable the detection of INF and NAN.

Microsoft Visual Studio
This is a very popular compiler for Windows because it has a good and user friendly IDE (Integrated Development Environment) and debugger. Make sure you are compiling for the "unmanaged" version, i. e. *not* using the .net common language runtime (CLR).

The Microsoft compiler optimizes reasonably well, but not as good as the other compilers. Support for the latest instruction sets is incomplete.

Do not use the option /fp:fast on a Microsoft compiler because this may disable the detection of INF and NAN.

Intel C++ compiler
Version 2021 or later of an Intel C++ compiler is required for compiling VCL version 2.xx.

The Intel C++ compiler currently comes in two versions: "Intel C++ Compiler Classic" (icc for Linux and icl for Windows) and "Intel oneAPI LLVM-based C++ Compiler" (icx). The classic version is a continuation of previous versions and is not recommended for new projects. The LLVM-based version is based on a Clang compiler with additional "Intel proprietary optimizations and code generation".

Note that the Intel compiler "Classic" and some of the function libraries favour Intel CPUs and produce code that runs slower than necessary on CPUs of any other brand than Intel. Do not use the Intel compiler "Classic" for software that may run on non-Intel microprocessors. Code produced by the Intel LLVM-based Compiler will usually give good performance on non-Intel processors. Avoid command line options beginning with -x or /Qx. Code compiled with these options can only run on Intel processors. See table 9.2 for an overview of command line options.

You may use the Intel LLVM-based Compiler if you need Intel-specific features. Otherwise, you may as well prefer the pure Clang compiler which is almost identical.

Conclusion
My recommendation is to use the Clang or Gnu compiler for the release version of a program when performance is important. Microsoft Visual Studio may be a convenient aid in the development phase of a Windows project. Switching to Clang is easy because there is a Clang/LLVM plugin to Visual Studio.

9.3 Choosing the optimal vector size and precision

It takes the same time to make a vector addition with vectors of eight single precision floats ($\mathrm{Vec8f}$) as with vectors of four double precision floats ($\mathrm{Vec4d}$). Likewise, it takes the same time to add two integer vectors whether the vectors have eight 32-bit integers ($\mathrm{Vec8i}$) or sixteen 16-bit integers ($\mathrm{Vec16s}$). Therefore, it is advantageous to use the lowest precision or resolution that fits the data. It may even be worthwhile to modify a floating point algorithm to reduce loss of precision if this allows you to use single precision rather than double precision. Half precision can improve the performance

even further, if supported. However, you should also take into account the time it takes to convert data from one precision to another. Therefore, it is not good to mix different precisions.

The total vector size is 128 bits, 256 or 512 bits, depending on the instruction set. The 256-bit floating point vectors are advantageous when the AVX instruction set is available and enabled. The 256-bit integer vectors are advantageous under the AVX2 instruction set. The 512-bit integer and floating point vectors are available with the AVX512 instruction set. Table 2.1 on page 8 lists the recommended instruction set for each vector class. You can compile multiple versions of your code for different instruction sets as explained in chapter 9.9 below. This makes it possible to code for the largest vector size in order to make your code ready for the newest CPU's. For example, if you are using the vector class Vec16f then you will be using 512-bit vectors when the code is running on a CPU that supports AVX512. The code will use two 256-bit vectors instead of one 512-bit vector when running on a CPU with only AVX2.

Current microprocessors can typically do two full size vector operations per clock cycle in small loops. (See Agner's optimization manuals for details).

9.4 Putting data into vectors

The different ways of putting data into vectors are listed on page 12. If the vector elements are constants known at compile time, then the fastest way is to use a constructor:

Example 9.1.

```
Vec4i a(1);              // a = (1, 1, 1, 1)
Vec4i b(2, 3, 4, 5);     // b = (2, 3, 4, 5)
```

If the vector elements are not constants then the fastest way is to load from an array with the method load. However, it is not good to load data from an array immediately after writing the data elements to the array one by one, because this causes a "store forwarding stall" (see Agner's microarchitecture manual). This is illustrated in the following examples:

Example 9.2.

```
// Make vector using constructor
int MakeMyData(int i); // make whatever data we need
void DoSomething(Vec4i & data); // handle these data
const int datasize = 1000; // total number data elements
  ...
for (int i = 0; i < datasize; i += 4) {
    Vec4i d(MakeMyData(i),   MakeMyData(i+1),
            MakeMyData(i+2), MakeMyData(i+3));
    DoSomething(d);
}
```

Example 9.3.

```
// Load from small array
int MakeMyData(int i); // make whatever data we need
void DoSomething(Vec4i & data); // handle these data
const int datasize = 1000; // total number data elements
  ...
for (int i = 0; i < datasize; i += 4) {
```

```
    int data4[4];
    for (int j = 0; j < 4; j++) {
        data4[j] = MakeMyData(i+j);
    }
    // store forwarding stall for large read after small writes:
    Vec4i d = Vec4i().load(data4);
    DoSomething(d);
}
```

Example 9.4.

```
// Make array a little bigger
int MakeMyData(int i); // make whatever data we need
void DoSomething(Vec4i & data); // handle these data
const int datasize = 1000; // total number data elements
  ...
for (int i = 0; i < datasize; i += 8) {
    int data8[8];
    for (int j = 0; j < 8; j++) {
        data8[j] = MakeMyData(i+j);
    }
    Vec4i d;
    for (int k = 0; k < 8; k += 4) {
        d.load(data8 + k);
        DoSomething(d);
    }
}
```

Example 9.5.

```
// Make array full size
int MakeMyData(int i); // make whatever data we need
void DoSomething(Vec4i & data); // handle these data
const int datasize = 1000; // total number data elements
  ...
int data1000[datasize];
int i;
for (i = 0; i < datasize; i++) {
    data1000[i] = MakeMyData(i);
}
Vec4i d;
for (i = 0; i < datasize; i += 4) {
    d.load(data1000 + i);
    DoSomething(d);
}
```

Example 9.6.

```
// Use insert. No array needed
int MakeMyData(int i); // make whatever data we need
void DoSomething(Vec4i & data); // handle these data
```

```
const int datasize = 1000; // total number data elements
  ...
Vec4i d;  // declare vector
for (int i = 0; i < datasize; i += 4) {
    for (int j = 0; j < 4; j++) {
        d.insert(j, MakeMyData(i+j)); // insert element
    }
    DoSomething(d);
}
```

In example 9.2, we are combining four data elements into vector d by calling a constructor with four parameters. This may not be the most efficient way because it requires several instructions to combine the four numbers into a single vector.

In example 9.3, we are putting the four values into an array and then loading the array into a vector. This is causing the so-called *store forwarding stall*. A store forwarding stall occurs in the CPU hardware when doing a large read (here 128 bits) immediately after a smaller write (here 32 bits) to the same address range. This causes a delay of 10 - 20 clock cycles.

In example 9.4, we are putting eight values into an array and then reading four elements at a time. If we assume that it takes more than 10 - 20 clock cycles to call MakeMyData four times then the first four elements of the array will have sufficient time to make it into the level-1 cache while we are writing the next four elements. This delay is sufficient to avoid the store forwarding stall. A disadvantage of example 9.4 is that we need an extra loop.

In example 9.5, we are putting a thousand elements into an array before loading them. This is certain to avoid the store forwarding stall. A disadvantage of example 9.5 is that the large array takes more cache space.

Example 9.6 avoids any memory intermediate by inserting elements directly into the vector. This method is most efficient when the AVX512VL instruction set is enabled. The compiler is likely to keep often-used vectors in registers without saving them to memory.

9.5 Alignment of arrays and vectors

Reading and writing vectors from or to memory is likely to be slightly faster if the array in memory is aligned to an address divisible by the vector size. The vector size is 16, 32, or 64 bytes for 128, 256, and 512 bits, respectively. The program may not work when compiled for an instruction set less than AVX if vectors are not aligned by at least 16.

Most compilers will align large arrays automatically if they are stored in static memory, but perhaps not if they are stored in local memory or allocated with operator new, etc.

An array can be aligned with the alignas keyword, for example:

```
alignas(64) float mydata[1024];
```

Older compilers use __declspec(align(64)) in Windows, or __attribute__((aligned(64)) in Linux.

It is always recommended to align large arrays for performance reasons if the code uses vectors. C++ version 17 supports alignment with std::aligned_alloc.

A useful method is to align an array of vectors with operator new. The compiler recognizes that a vector class, e.g. Vec8f, needs alignment. An array of such vectors will be aligned correctly, even when allocated with new. This is illustrated in the following example:

```
// size of dataset, as number of floats:
int datasize = 1024;

// variable size array, properly aligned
// (assuming that datasize is divisible by vector size):
Vec8f *mydata = new Vec8f[datasize / Vec8f::size()];

// access array as single elements:
float * mydataf = (float*)mydata;
int i;
for (i = 0; i < datasize; i++) {
    mydataf[i] = (float)i;
}

// access array as vectors:
Vec8f x;
for (i = 0; i < datasize / Vec8f::size(); i++) {
    x = mydata[i];
    x *= 100.f;
    mydata[i] = x;
}

// remember to free the allocated data:
delete[] mydata;
```

The container class template ContainerV is available as an add-on to the vector class library. This is useful for making a properly aligned array of vectors.

Finally, it is possible to do the alignment manually as illustrated in this example:

```
// Example of aligning memory
int arraySize = 1024;      // Required array size
const int alignBy = 64;    // Required alignment (must be a power of 2)
// allocate more than needed
char * unalignedAddress = new char[arraySize*sizeof(float) + alignBy];
// round up the address to nearest multiple of alignBy
char * alignedAddress = (char*)(((size_t)unalignedAddress+alignBy-1) &
   (-alignBy));

// cast aligned pointer to required type
float * mydataf = (float*)alignedAddress;

// use the aligned array
for (int i = 0; i < arraySize; i++) {
    mydataf[i] = (float)i;
}

// remember to free at unalignedAddress, not alignedAddress
delete[] unalignedAddress; // free memory
```

9.6 When the data size is not a multiple of the vector size

It is obviously easier to vectorize a data set when the number of elements in the data set is a multiple of the vector size. Here, we will discuss different ways of handling the situation when the data do not fit into an integral number of vectors. We will use the simple example of adding 134 integers stored in an array. The following examples illustrate different solutions.

Example 9.7.

```
// Handling the remaining data one by one
const int datasize = 134;
const int vectorsize = 8;
const int regularpart = datasize & (-vectorsize); // = 128
// (AND-ing with -vectorsize will round down to the nearest
// lower multiple of vectorsize. This works only if vectorsize
// is a power of 2)

int mydata[datasize];
  ... // initialize mydata

Vec8i sum1(0), temp;
int i;
// loop for 8 numbers at a time
for (i = 0; i < regularpart; i += vectorsize) {
    temp.load(mydata+i);  // load 8 elements
    sum1 += temp;         // add 8 elements
}
int sum = 0;
// loop for the remaining 6 numbers
for (; i < datasize; i++) {
    sum += mydata[i];
}
sum += horizontal_add(sum1); // add the vector sum
```

Example 9.8.

```
// Handling the remaining data with a smaller vector size
const int datasize = 134;
const int vectorsize = 8;
const int regularpart = datasize & (-vectorsize); // = 128
int mydata[datasize];
  ... // initialize mydata

Vec8i sum1(0), temp;
int sum = 0;
int i;
// loop for 8 numbers at a time
for (i = 0; i < regularpart; i += vectorsize) {
    temp.load(mydata+i);  // load 8 elements
    sum1 += temp;         // add 8 elements
}
sum = horizontal_add(sum1); // sum of first 128 numbers
if (datasize - i >= 4) {
```

```
    // get four more numbers
    Vec4i sum2;
    sum2.load(mydata+i);
    i += 4;
    sum += horizontal_add(sum2);
}
// loop for the remaining 2 numbers
for (; i < datasize; i++) {
    sum += mydata[i];
}
```

Example 9.9.

```
// Use partial load for the last vector
const int datasize = 134;
const int vectorsize = 8;
const int regularpart = datasize & (-vectorsize); // = 128

int mydata[datasize];
  ... // initialize mydata

Vec8i sum1(0), temp;
// loop for 8 numbers at a time
for (int i = 0; i < regularpart; i += vectorsize) {
    temp.load(mydata+i);  // load 8 elements
    sum1 += temp;         // add 8 elements
}
// load the last 6 elements
temp.load_partial(datasize-regularpart, mydata+regularpart);
sum1 += temp;             // add last 6 elements

int sum = horizontal_add(sum1); // vector sum
```

Example 9.10.

```
// Read past the end of the array and ignore excess data
const int datasize = 134;
const int vectorsize = 8;
int mydata[datasize];
  ... // initialize mydata

Vec8i sum1(0), temp;
// loop for 8 numbers at a time, reading 136 numbers
for (int i = 0; i < datasize; i += vectorsize) {
    temp.load(mydata+i);  // load 8 elements
    if (datasize - i < vectorsize) {
        // set excess data to zero
        // (this may be faster than load_partial)
        temp.cutoff(datasize - i);
    }
    sum1 += temp;         // add 8 elements
}
int sum = horizontal_add(sum1); // vector sum
```

Example 9.11.

```
// Make array bigger and set excess data to zero
const int datasize = 134;
const int vectorsize = 8;
// round up datasize to nearest higher multiple of vectorsize
const int arraysize =
    (datasize + vectorsize - 1) & (-vectorsize);  // = 136
int mydata[arraysize];
int i;
  ... // initialize mydata

// set excess data to zero
for (i = datasize; i < arraysize; i++) {
    mydata[i] = 0;
}

Vec8i sum1(0), temp;
// loop for 8 numbers at a time, reading 136 numbers
for (i = 0; i < arraysize; i += vectorsize) {
    temp.load(mydata+i);  // load 8 elements
    sum1 += temp;         // add 8 elements
}
int sum = horizontal_add(sum1); // vector sum
```

It is clearly advantageous to increase the array size to a multiple of the vector size, as in example 9.11 above. Likewise, if you are storing vector data to an array, then it is an advantage to make the result array bigger to hold the excess data. If this is not possible then use store_partial to write the last partial vector to the array.

It is usually possible to read past the end of an array, as in example 9.10 above, without causing problems. However, there is a theoretical possibility that the array is placed at the very end of the readable data section so that the program will crash when attempting to read from an illegal address past the end of the valid data area. To consider this problem, we need to look at each possible method of data storage:

- An array declared inside a function, and not static, is stored on the stack. The subsequent addresses on the stack will contain the return address and parameters for the function, followed by local data, parameters, and return address of the next higher function all the way up to main. In this case there is plenty of extra data to read from.
- A static or global array is stored in static data memory. The static data area is often followed by library data, exception handler tables, link tables, etc. These tables can be seen by requesting a map file from the linker.
- Data allocated with the operator new are stored on the heap. I have no information of the size of the end node in a heap.
- If an array is declared inside a class definition then one of the three cases above applies, depending on how the class instance (object) is created.

These problems can be avoided either by making the array bigger or by aligning the array to an address divisible by the vector size, as described on page 73. The memory page size is at least 4 kbytes, and always a power of 2. If the array is aligned by the vector size then the page boundaries are certain to coincide with vector boundaries. This makes sure that there is no memory page

boundary between the end of the array and the next vector-size boundary. Therefore, we can read up to the next vector-size boundary without the risk of crossing a boundary to an invalid memory page.

The add-on package named 'containers' includes efficient container class templates for arrays of fixed size and dynamic size, as well as matrixes. These containers will automatically extend arrays to a multiple of the vector size. Use the container class template ContainerV for making arrays that fit the vector classes. See containers_manual.pdf for details.

9.7 Using multiple accumulators

Consider this function which adds a long list of floating point numbers:

Example 9.12.

```
double add_long_list(double const * p, int n) {
    int n1 = n & (-4);  // round down n to multiple of 4
    Vec4d sum(0.0);
    int i;
    for (i = 0; i < n1; i += 4) {
        sum += Vec4d().load(p + i);  // add 4 numbers
    }
    // add any remaining numbers
    sum += Vec4d().load_partial(n - i, p + i);
    return horizontal_add(sum);
}
```

In this example, we have a loop-carried dependency chain (see Agner's C++ manual). The vector addition inside the loop has a latency of typically 3 - 5 clock cycles. As each addition has to wait for the result of the previous addition, the loop will take 3 - 5 clock cycles per iteration.

However, the throughput of floating point additions is typically one or two vector additions per clock cycle. Therefore, we are far from fully utilizing the capacity of the floating point adder. In this situation, we can double the speed by using two accumulators:

Example 9.13.

```
double add_long_list(double const * p, int n) {
    int n2 = n & (-8);  // round down n to multiple of 8
    Vec4d sum1(0.0), sum2(0.0);
    int i;
    for (i = 0; i < n2; i += 8) {
        sum1 += Vec4d().load(p + i);      // add 4 numbers
        sum2 += Vec4d().load(p + i + 4); // 4 more numbers
    }
    if (n - i >= 4) {
        // add 4 more numbers
        sum1 += Vec4d().load(p + i);
        i += 4;
    }
    // add any remaining numbers
    sum2 += Vec4d().load_partial(n - i, p + i);
    return horizontal_add(sum1 + sum2);
}
```

Here, the addition to sum2 can begin before the addition to sum1 is finished. The loop still takes 3 - 5 clock cycles per iteration, but the number of additions done per loop iteration is doubled. It may even be worthwhile to have three or four accumulators in this case if n is very big.

In general, if we want to predict whether it is advantageous to have more than one accumulator, we first have to see if there is a loop-carried dependency chain. If the performance is not limited by a loop-carried dependency chain then there is no need for multiple accumulators. Next, we have to look at the latency and throughput of the instructions inside the loop. Floating point addition, subtraction and multiplication all have latencies of typically 3 - 5 clock cycles and a throughput of one or two vector additions/subtractions/multiplications per clock cycle. Therefore, if the loop-carried dependency chain involves floating point addition, subtraction or multiplication; and the total number of floating point operations per loop iteration is lower than the maximum throughput, then it may be advantageous to have two accumulators, or perhaps more than two.

There is rarely any reason to have multiple accumulators in integer code, because an integer vector addition has a latency of just 1 or 2 clock cycles.

9.8 Using multiple threads

Performance can be improved by dividing the work between multiple threads running in parallel on processors with multiple CPU cores. It is important to distinguish between coarse-grained parallelism and fine-grained parallelism. Coarse-grained parallelism refers to the situation where a long sequence of operations can be carried out independently of other tasks that are running in parallel. Fine-grained parallelism is the situation where a task is divided into many small subtasks, but it is impossible to work for very long on a particular subtask before coordination with other subtasks is necessary.

Vector operations are useful for fine-grained parallelism, while multithreading is useful only for coarse-grained parallelism. The work should be divided between threads in such as way that communication between the threads is avoided, or at least kept at a minimum.

Modern computers have multiple CPU cores. It is often possible to run two threads simultaneously in each CPU core. This is called simultaneous multithreading (SMT) or hyperthreading. Two threads running in the same CPU core will be competing for the same CPU resources so that each thread is getting only half of the available resources. Therefore, SMT is not advantageous for CPU-intensive code such as heavy mathematical calculations.

The optimal number of threads for CPU-intensive code is equal to the number of CPU cores or physical processors. If the code is not CPU-intensive, i.e. if the performance is limited by something else such as RAM, disk access, or network speed, then you will probably get better performance by setting the number of threads equal to the number of logical processors. This is the number of threads that can run simultaneously without task switching. The number of logical processors is double the number of physical processors if each CPU core can run two threads simultaneously.

The function $\mathrm{physicalProcessors}()$ gives information about the number of physical and logical processors (see page 83).

It is not safe to access the same data from multiple threads simultaneously. For example, it may be uncertain whether one thread is reading a value before or after it has been modified by another thread. Container classes, in particular, are unsafe to access from multiple threads.

The floating point control word (see p. 55) is not shared between threads.

9.9 Instruction sets and CPU dispatching

Historically, almost every new generation of microprocessors has added a new extension to the instruction set. Most of the new instructions relate to vector operations. We can take advantage of these new instructions to make vector code more efficient. The vector class library requires the SSE2 instruction set as a minimum, but it makes more efficient code when a higher instruction set is used. Table 9.1 indicates things that are improved for each successive instruction set extension.

Table 9.1: Instruction set history

Instruction set	**Year introduced**	**VCL functions improved**
SSE2	2001	minimum requirement for vector class library
SSE3	2004	floating point horizontal_add
SSSE3	2006	permute, blend and lookup functions, integer abs
SSE4.1	2007	select, blend, horizontal_and, horizontal_or, integer max/min, integer multiply (32 and 64 bit), integer divide (32 bit), 64-bit integer compare (==, !=), floating point round, truncate, floor, ceil.
SSE4.2	2008	64-bit integer compare (>, >=, <, <=). 64 bit integer max, min
AVX	2011	all operations on 256-bit floating point vectors: Vec8f, Vec4d
XOP AMD only	2011	compare, horizontal_add_x, rotate_left, blend, and lookup on 128-bit integer vectors. Obsolete.
FMA4 AMD only	2011	floating point code containing multiplication followed by addition. Obsolete.
FMA3	2012	floating point code containing multiplication followed by addition
AVX2	2013	All operations on 256-bit integer vectors: Vec32c, Vec32uc, Vec16s, Vec16us, Vec8i, Vec8ui, Vec4q, Vec4uq. Gather.
F16C	2013	Conversion between single precision and half precision floating point numbers.
AVX512F	2016	All operations on 512-bit integer and floating point vectors: Vec16i, Vec16ui, Vec8q, Vec8uq, Vec16f, Vec8d.
AVX512BW	2018	512 bit vectors with 8-bit and 16-bit integer elements
AVX512DQ	2018	Faster multiplication of vectors of 64-bit integers.
AVX512VL	2018	Compact boolean vectors for 128 and 256 bit data. Improved performance of insert, extract, load_partial, store_partial, and several other functions.
AVX512ER	2016	Only on a few processor models have this. Fast exponential functions. Better precision on approx_recipr and approx_rsqrt.
AVX512VBMI	2018	Faster permutation functions etc. for Vec32c and Vec64c
AVX512VBMI2	2019	Faster extract from 8-bit and 16-bit integer vectors
AVX512-FP16	future	Half precision floating point calculations.

The vector class library makes it possible to compile for different instruction sets from the same source code. Different versions are made simply by recompiling the code with different compiler options. The instruction set to use in VCL can be specified on the compiler command line as listed in table 9.2.

The Microsoft compiler does not have command line options for all the instruction sets, but other instruction sets can be specified as defines which are detected in the preprocessing directives of the

Table 9.2: Command line options

Instruction set	Gnu and Clang compiler	Intel compiler Linux	Intel compiler Windows	MS compiler
SSE2	-msse2	-msse2	/arch:sse2	/arch:sse2
SSE3	-msse3	-msse3	/arch:sse3	/arch:sse2 /D__SSE3__
SSSE3	-mssse3	-mssse3	/arch:ssse3	/arch:sse2 /D__SSSE3__
SSE4.1	-msse4.1	-msse4.1	/arch:sse4.1	/arch:sse2 /D__SSE4_1__
SSE4.2	-msse4.2	-msse4.2	/arch:sse4.2	/arch:sse2 /D__SSE4_2__
AVX	-mavx -fabi-version=0	-mavx	/arch:avx	/arch:avx /DINSTRSET=7
FMA3	-mfma	-mfma	-mfma	/DINSTRSET=7
AVX2	-mavx2 -fabi-version=0	-mavx2	/arch:avx2	/arch:avx2 /DINSTRSET=8
F16C	-mf16c	-mf16c	/arch:avx2	/D__F16C__
AVX512F	-mavx512f	-mavx512f	/arch:COMMON-AVX512	not supported without AVX512DQ
AVX512VL/BW/DQ	-mavx512vl -mavx512bw -mavx512dq	-mavx512vl -mavx512bw -mavx512dq	/arch:CORE-AVX512	/arch:avx2 /DINSTRSET=10
AVX512VBMI	-mavx512vbmi	-mavx512vbmi	??	/D__AVX512VBMI__
AVX512VBMI2	-mavx512vbmi2	-mavx512vbmi2	??	/D__AVX512VBMI2__
AVX512ER	-mavx512er	-xMIC-AVX512	/arch:MIC-AVX512	/D__AVX512ER__
AVX512-FP16	-mavx512fp16	-mavx512fp16	/arch:SAPPHIRERAPIDS	/D__AVX512FP16__
C++ standard	-std=c++17	-std=c++17	/Qstd:c++17	-std=c++17

vector class library.

The FMA3 instruction set is not always handled directly by the code in the vector class library, but by the compiler. The compiler may automatically combine a floating point multiplication and a subsequent addition or subtraction into a single instruction, unless you have specified a strict floating point model.

It is recommended to specify compiler options that allow efficient code optimizations. Suitable options on Gnu and Clang compilers are -O2 -fno-trapping-math -fno-math-errno. The option -O3 is sometimes better than -O2, but in some cases it is worse. You may test whether -O2 or -O3 gives the best performance in your specific case.
Suitable options on Microsoft and Intel compilers are /O2 /fp:except-.

There is no advantage in using the biggest vector classes unless the corresponding instruction set is specified, but it can be convenient to use these classes anyway if the same source code is compiled for multiple versions with different instruction sets. Each large vector will simply be split up into two or four smaller vectors when compiling for a lower instruction set.

It is recommended to make an automatic CPU dispatcher that detects at runtime which instruction

sets are supported by the actual CPU, and selects the best version of the code accordingly. For example, you may compile the code three times for three different instruction sets: SSE2, AVX2 and AVX512VL/BW/DQ. The CPU dispatcher will then set a function pointer to point to the appropriate version of the compiled code. You can use the function instrset_detect (see below, page 82) to detect the supported instruction set. Two examples are provided to show how to do the CPU dispatching:

dispatch_example1.cpp: This example is using different function names for the different versions. This is useful for simple cases with only one or a few functions.

dispatch_example2.cpp: This example is using different namespaces for the different versions. This is the preferred method if the code contains multiple functions, classes, objects, etc.

There is an important restriction when you are combining code compiled for different instruction sets: Do not transfer any data as vector objects between different pieces of code that are compiled for different instruction sets because the vectors may be represented differently under the different instruction sets. It is recommended to transfer the data as arrays instead between different parts of the program that are compiled for different instruction sets.

The functions listed below can be used for detecting at runtime which instruction set is supported, and other useful information about the CPU.

The function $\mathrm{instrset\_detect}()$ gives a value representing the instruction set level.

Function	int instrset_detect()
Source	instrset_detect.cpp
Description	returns one of these values: 0 = 80386 instruction set 1 or above = SSE supported by CPU 2 or above = SSE2 3 or above = SSE3 4 or above = Supplementary SSE3 (SSSE3) 5 or above = SSE4.1 6 or above = SSE4.2 7 or above = AVX 8 or above = AVX2 9 or above = AVX512F 10 or above = AVX512VL, AVX512BW, and AVX512DQ
Efficiency	poor

Additional instruction set extensions are not necessarily part of a linear sequence. These extensions can be detected with the following functions.

Function	bool hasFMA3()
Source	instrset_detect.cpp
Description	returns true if FMA3 is supported
Efficiency	poor

Function	bool hasAVX512ER()
Source	instrset_detect.cpp
Description	returns true if AVX512ER is supported
Efficiency	poor

Function	bool hasAVX512VBMI()
Source	instrset_detect.cpp
Description	returns true if AVX512VBMI is supported
Efficiency	poor

Function	bool hasAVX512VBMI2()
Source	instrset_detect.cpp
Description	returns true if AVX512VBMI2 is supported
Efficiency	poor

Function	bool hasF16C()
Source	instrset_detect.cpp
Description	returns true if F16C is supported
Efficiency	poor

Function	bool hasAVX512FP16()
Source	instrset_detect.cpp
Description	returns true if AVX512-FP16 is supported
Efficiency	poor

Function	int physicalProcessors(int * logical_processors = 0)
Source	add-on/physical_processors.cpp
Description	Returns the number of physical processors = the number of CPU cores. The number of logical processors (returned through logical_processors) is double the number of physical processors if the CPU can run two threads simultaneously in each CPU core.
Efficiency	poor

9.10 Function calling convention

Function calls are most efficient when vectors are transferred in registers rather than in memory. This can be achieved in various ways:

- Use inline functions. This is useful for small functions and for functions that are only called in one place. An optimizing compiler may inline functions automatically, even if they are not specified as `inline`. You may declare such functions `static` as well to prevent the compiler from making a non-inline copy of the inlined function.
- Use Linux or MacOS. Vector parameters are transferred in registers by default on these platforms. Vector function returns are transferred in registers in 64-bit mode.
- Use `__vectorcall` in 64-bit Windows. The Clang and Microsoft compilers can transfer vector parameters and vector returns in registers when `__vectorcall` is used on the function declaration. See the example on page 84.
- Use a vector size that fits the instruction set, according to table 2.1 on page 8.

Chapter 10

Examples

This example calculates the polynomial $x^3 + 2 \cdot x^2 - 5 \cdot x + 1$ on a floating point vector. The order of calculation is specified by parentheses in order to make shorter dependency chains.

Example 10.1.

```
Vec4f polynomial (Vec4f x) {
    return (x + 2.0f) * (x * x) + ((-5.0f) * x + 1.0f);
}
```

In 64-bit Windows, you may add __vectorcall and use a Clang or Microsoft compiler. This makes sure that vector parameters are transferred in registers rather than in memory. This is not needed when the function is inlined or when compiling for other platforms than Windows:

Example 10.2.

```
Vec4f __vectorcall polynomial (Vec4f x) {
    return (x + 2.0f) * (x * x) + ((-5.0f) * x + 1.0f);
}
```

The next example transposes a 4x4 matrix, using the AVX2 instruction set.

Example 10.3.

```
void transpose(float matrix[4][4]) {
    Vec8f row01, row23, col01, col23;
    // load first two rows
    row01.load(&matrix[0][0]);
    // load next two rows
    row23.load(&matrix[2][0]);
    // reorder into columns
    col01 = blend8f<0,4, 8,12,1,5, 9,13>(row01, row23);
    col23 = blend8f<2,6,10,14,3,7,11,15>(row01, row23);
    // store columns into rows
    col01.store(&matrix[0][0]);
    col23.store(&matrix[2][0]);
}
```

Same example with AVX512:

Example 10.4.

```
void transpose(float matrix[4][4]) {
    Vec16f rows, columns;
    // load entire matrix as rows
    rows.load(&matrix[0][0]);
    // reorder into columns
    columns = permute16f<0,4,8,12,1,5,9,13,
        2,6,10,14,3,7,11,15>(rows);
    // store columns into rows
    columns.store(&matrix[0][0]);
}
```

The next example makes a matrix multiplication of two 4x4 matrixes.

Example 10.5.

```
void matrixmul(float A[4][4], float B[4][4], float M[4][4]){
    // calculates M = A*B
    Vec4f Brow[4], Mrow[4];
    int i, j;
    // load B as rows
    for (i = 0; i < 4; i++) {
        Brow[i].load(&B[i][0]);
    }
    // loop for A and M rows
    for (i = 0; i < 4; i++) {
        Mrow[i] = Vec4f(0.0f);
        // loop for A columns, B rows
        for (j = 0; j < 4; j++) {
            Mrow[i] += Brow[j] * A[i][j];
        }
    }
    // store M
    for (i = 0; i < 4; i++) {
        Mrow[i].store(&M[i][0]);
    }
}
```

The next example makes a table of the sin function and gets sin(x) and cos(x) by table lookup.

Example 10.6.

```
#include <cmath>

const double pi = 3.14159265358979323846;

// length of table. Must be a power of 2.
#define sin_tablelen 1024
// the accuracy of table lookup is +/- pi/sin_tablelen

class SinTable {
protected:
```

```
    float table[sin_tablelen];
    float resolution;
    float rres;  // 1./resolution
public:
    SinTable();  // constructor
    Vec4f sin(Vec4f x);
    Vec4f cos(Vec4f x);
};

SinTable::SinTable() {  // constructor
    // compute resolution
    resolution = float(2.0 * pi / sin_tablelen);
    rres = 1.0f / resolution;
    // Initialize table (No need to use vectors here because this
    // is calculated only once:)
    for (int i = 0; i < sin_tablelen; i++) {
        table[i] = sinf((float)i * resolution);
    }
}

Vec4f SinTable::sin(Vec4f x) {
    // calculate sin by table lookup
    Vec4i index = roundi(x * rres);
    // modulo tablelen equivalent to modulo 2*pi
    index &= sin_tablelen - 1;
    // look up in table
    return lookup<sin_tablelen>(index, table);
}

Vec4f SinTable::cos(Vec4f x) {
    // calculate cos by table lookup
    Vec4i index = roundi(x * rres) + sin_tablelen/4;
    // modulo tablelen equivalent to modulo 2*pi
    index &= sin_tablelen - 1;
    // look up in table
    return lookup<sin_tablelen>(index, table);
}

int main() {
    SinTable sintab;
    Vec4f a(0.0f, 0.5f, 1.0f, 1.5f);
    Vec4f b = sintab.sin(a);
    // b = (0.0000 0.4768 0.8416 0.9973)
    // accuracy +/- 0.003
    ...
    return 0;
}
```

Chapter 11

Add-on packages

Various extra packages are available with code for special applications. These packages are stored at https://github.com/vectorclass/add-on. Manuals are included with each package. The add-on packages for VCL include:

Container classes. Container class templates for storing arrays of vectors. More efficient than the standard C++ container class templates.

This package also contains a class template for matrices where matrix rows are stored as VCL vectors. Various functions are included for accessing matrix elements and rows and for packing and unpacking matrix data.

Random number generator. A high-quality pseudo random number generator. Capable of generating random integer and floating point vectors. Suitable for large multi-threaded applications.

Decimal string conversion. Converts integer vectors to and from comma-separated lists in human-readable decimal ASCII form. Useful for reading and writing comma-separated files.

3-dimensional vectors. Defines 3-dimensional vectors for use in geometry and physics. Includes operators and functions for addition, multiplication, dot product, cross product, and rotation.

Complex number vectors. Defines complex number vectors for use in mathematics and electronics. Includes operators for add, subtract, multiply, divide, and conjugate, as well as functions such as complex square root, exponential function, and logarithm.

Quaternions. Defines quaternions (hypercomplex numbers) for use in mathematics. Includes operators for add, subtract, multiply, divide, conjugate, etc.

Chapter 12

Technical details

12.1 Error conditions

Runtime errors

The vector class library is generally not producing runtime error messages. An index out of range produces behavior that is implementation-dependent. This means that the output may be different for different instruction sets or for different versions of the vector class library.

For example, an attempt to read a vector element with an index that is out of range may result in various behaviors, such as producing zero, taking the index modulo the vector size, giving the last element, or producing an arbitrary value. Likewise, an attempt to write a vector element with an index that is out of range may variously take the index modulo the vector size, write the last element, or do nothing. This applies to functions such as insert, extract, load_partial, store_partial, cutoff, permute, blend, lookup, and gather. The same applies to a bit-index that is out of range in rotate functions and shift operators («, »).

Boolean vectors in the broad form (see page 32) are stored as integer vectors. The only allowed values for boolean vector elements in this case are 0 (false) and -1 (true). The behavior for other values is implementation-dependent and possibly inconsistent. For example, the behavior of the select function when a boolean selector element is a mixture of 0 and 1 bits depends on the instruction set. For instruction sets prior to SSE4.1, it will select between the operands bit-by-bit. For SSE4.1 and higher it will select integer vectors byte-by-byte, using the leftmost bit of each byte in the selector input. For floating point vectors under SSE4.1 and higher, it will use only the leftmost bit (sign bit) of the selector. Boolean vectors in the compact form have only one bit for each element.

An integer division by a variable that is zero will usually produce a runtime exception.

A program crash may be caused by alignment errors with instruction sets prior to AVX. This can happen if a VCL vector is stored in a dynamic array or a container class template instance that does not have correct alignment. See page 73

Floating point errors

The Vector Class Library produces infinity (INF) or "Not A Number" (NAN) to indicate floating point errors, as discussed on page 90. Floating point overflow will usually produce infinity, floating point underflow produces zero, and an invalid floating point operation produces NAN (Not A Number). The INF and NAN codes will usually propagate to the end result where they can be detected.

There are a few cases where INF and NAN codes do not propagate. For example, dividing a nonzero number by INF produces zero. Error codes cannot propagate through integer and boolean vectors. For example:

```
Vec4d a, b;
...
Vec4db f = a > 1.0;
b = select(f, a, 0.5);
```

The boolean vector elements in f will be either true or false, even if a is NAN, because a boolean can have no other values. In the case that an element of a is NAN, the corresponding element in f will be false, and the element in b will be 0.5. The NAN error is not propagated from a to b. Therefore, you have to check for errors before making a boolean expression. This can be done like this:

```
Vec4d a, b;
...
if ( ! horizontal_and(is_finite(a))) {
    // handle error
    ...
}
Vec4db f = a > 1.0;
b = select(f, a, 0.5);
```

Compile-time errors

The Vector Class Library is making heavy use of metaprogramming features that go to the limit of what modern compilers can do. Occasional problems have been observed with all compilers. Errors that are specific to a particular compiler are listed in separate files at the GIT repository under miscellaneous. Please check these lists of known errors before reporting a problem.

Even small syntax errors may result in very long error messages due to the heavy use of templates and overloading. These error messages may be confusing, but generally indicating the line number of the error.

Integer vector division by a const_int or const_uint can produce a compile-time error message when the divisor is zero or out of range.

"Ambiguous call to overloaded function":
This can happen when parameters have wrong types. Make sure all parameters have the correct type.

Version 1.xx of VCL may produce error messages that are not very informative, such as **"Static_error_check<false>"** due to limitations in template metaprogramming.

Link errors

"unresolved external symbol ___intel_cpu_indicator_x":
This link error occurs when you are using Intel's SVML library without including a CPU dispatcher. Add the library libircmt.lib or libirc.a to use Intel's CPU dispatch function. Make sure to choose the 32-bit or 64-bit of the library, as appropriate. See page 58 for details.

"unresolved external symbol ___svml_sin2@@16, etc.
You need to link the library **svmlpatch.lib**, which you can find at the git repository under miscellaneous.

Implementation-dependent behavior

A big advantage of the VCL library is that you can compile the same source code for different instruction set extensions. A higher instruction set will generally give faster code, but produce the

same results. There may, however, be cases where the same code generates different results with different instruction sets or different compilers. These cases include:

- An index out of range produces implementation-dependent results. Functions such as insert, extract, load_partial, store_partial, cutoff, permute, blend, lookup, gather, and scatter may produce different results for an index out of range depending on the instruction set. No exception or error message is generated, only a meaningless number.
- permute and blend functions allow a "don't care" index (V_DC) to be specified. The result for a don't care element may depend on the instruction set.
- Negative zero. The floating point values of 0.0 and -0.0 are normally regarded as equal. Some functions may return 0.0 or -0.0 depending on the instruction set, e.g. when rounding a negative number. The sign of a zero can be detected by the functions sign_bit and sign_combine. You may #define SIGNED_ZERO to get consistent and pedantic conformance to the specifications of signed zero in the IEEE 754-2019 standard.
- NANs. An error code can be propagated through NAN (not-a-number) values and retrieved by the function nan_code. When two NAN values with different codes are combined, for example by adding them together, the result may be either of the two values, depending on the compiler. The sign of a NAN has no meaning and may vary.
  Use the minimum and maximum functions rather than min and max if you want to propagate NAN values through these functions.

12.2 Floating point behavior details

The Vector Class Library is generally conforming to the new IEEE 754-2019 Standard for Floating-Point Arithmetic, but some compromises have been necessary for the purpose of vector processing and for better performance. The deviations from the standard are discussed below.

Subnormal numbers. Subnormal numbers (also called denormal numbers) are numerically extremely small floating point numbers where the exponent is below the normal range. Some microprocessors are handling subnormal numbers in a very inefficient way that is more than a hundred times slower than for normal floating point numbers. You may call the function no_subnormals() to prevent this and treat subnormal numbers as zero in single and double precision floating point calculations. Calculations in half precision are generally efficient even when values are subnormal. Some of the mathematical functions in VCL always treat subnormal numbers as zero for reasons of performance. This includes logarithm, exponential, and power functions.

Signed zero. Signed zero is a controversial issue. The floating point standard defines two different zeroes: +0.0 and -0.0. The two zeroes are equal, but still distinguishable. Some of the functions may return +0.0 where the standard requires -0.0.
You may #define SIGNED_ZERO if you want the sign of zero to conform to the IEEE 754-2019 standard, though this may slow down performance a little. SIGNED_ZERO may affect several functions, including round, truncate, floor, ceil, maximum, minimum, cbrt, pow_ratio, expm1, log1p.

No exception trapping. Floating point errors are traditionally detected by trapping errors or relying on an errno variable. These methods are not well suited for vector processing and out-of-order processing. This is explained in the document "NAN propagation versus fault trapping in floating point code", Agner Fog, 2019.

The Vector Class Library does not support fault trapping, and it does not indicate exceptions in a variable such as the traditional errno. It is not recommended to turn on floating point exceptions because this can cause inconsistent behavior, such as traps for exceptions in not-taken branches. Do not attempt to trap numerical errors in try/catch blocks.

Instead, the vector class library indicates floating point exceptions by producing INF or NAN codes in the individual vector element that produced the fault. The INF and NAN codes will propagate to the end result of a series of calculations when certain conditions are satisfied. The most efficient way of detecting floating point errors is to look for INF and NAN codes in the result.

Conditions where INF and NAN codes are not propagated are discussed at page 88

Do not use the compiler options -ffast-math, -ffinite-math-only, or /fp:fast because this may disable the detection of INF and NAN.

No signaling NANs. Signaling NANs are special codes that will raise an exception when they are loaded from memory. Signaling NANs are rarely used in modern software. Signaling NANs should not be used in VCL because exception trapping is not supported.

NAN payload operations. A NAN may contain additional information called a payload. This payload can propagate through a series of calculations to the end result. Some of the mathematical functions in VCL can put a payload into the NAN result in case of an error. This makes it possible to identify which function generated the NAN.

The nan.. and nan_code functions make it possible to set and get NAN payloads. The IEEE 754 standard does not specify what happens to the payload when converting between single and double precision, but experiments show that all microprocessors that use the binary floating point format will left-justify the payload. The nan.. and nan_code functions treat the NAN payload as a 22-bit left-justified unsigned integer in order to allow conversions between single and double precision. These functions deviate from the IEEE 754-2019 standard.

NAN propagation in maximum and minimum functions. The max and min functions do not propagate NANs according to the 2008 version of the standard. This unfortunate situation is redressed in the 2019 revision of the standard. VCL offers two different versions of these functions: The max and min functions are equivalent to a > b ? a : b and a < b ? a : b, respectively. These functions return b if a is NAN. The slightly less efficient functions maximum and minimum are sure to propagate NANs, in accordance with the 2019 revision of the standard.

NAN propagation in pow function. The standard specifies that pow(NAN,0) and pow(1,NAN) will give the result 1.0. The VCL implementation deviates from this and produces a NAN output in all cases where an input is NAN, in order to support reliable NAN propagation.

Function parameter range. Some of the mathematical functions have internal overflow for extreme values of the input parameters. These functions have a limited input range because an extra branch to handle the extreme cases would reduce the overall performance. Limitations of the input range are mentioned in the listing of the individual functions.

12.3 Making add-on packages

Anybody can contribute add-on packages for VCL. Contributors must follow the following guidelines:

Purpose
The package must serve a general purpose that is useful for others. The code must rely on the VCL.

Open source
The package must be published under an open source license. The preferred license is the same as for VCL, i.e. Apache 2.0 license or later. Other accepted licenses include GPL 3.0 or later, LGPL 3.0 or later, and revised BSD license.

Documentation
The package must include an instruction manual in English. The manual may be supplied in one of these formats:

- Plain text as a an ASCII .txt file
- Plain text as a comment in the beginning of the code file
- A .pdf file. The source needed for modifying and rebuilding the .pdf file must be included. The file format of the pdf source must be .tex, .odt, or .docx. Closed, proprietary file formats are not allowed.

The documentation must include the name and contact information of at least one person responsible for maintaining the code.

VCL does not use Doxygen or other kinds of metadata for generating documentation. You may use an advanced IDE such as Microsoft Visual Studio for navigating, tracing, browsing, and finding cross-references.

Coding style
The code must be in C++ language, with file format .h and/or .cpp. Names and comments must use English language. Name, date, and version number must be written in a comment at the beginning of each code file.

The file format is plain ASCII. UTF-8 should be avoided if possible. Use Windows-style linefeeds, i.e. \r\n. Indent 4 spaces for every block level. Tabs are not allowed. Remember to set the option in your editor to use spaces instead of tabs.

The purposes of all classes, functions, and variables must be explained in comments unless they are self-explaining.

Use curly brackets for branches and loops. A closing curly bracket must be placed on a separate line. An opening curly bracket does not need a separate line. else-if may be contracted without an extra curly bracket. Example:

```
if (a < 0) {
    // negative
}
else if (a == 0) {
    // zero
}
else {
    // positive
}
```

Optimization
All functions and operators in .h files should be static and inline.

Do not optimize the code for a specific microprocessor, but focus on what is likely to be optimal on future microprocessor models. The most likely bottlenecks to consider are cache use, instruction decoding, and dependency chains. Small loops are usually more efficient than large unrolled loops.

Minimize the use of static constants because they take op memory space even when they are not used. Static constants may be stored in templates that are not instantiated if they are not used.

Preprocessing #define's must have unique names that are unlikely to cause name clashes because they are in the global namespace. It is preferred to use const int etc. instead for defining constants.

Testing
Any code must be thoroughly tested with the latest version of VCL before submission. It should preferably be tested with multiple different compilers and different operating systems. Add-on packages may have their own test bench.

12.4 Contributing to VCL

Bug reports
Bug reports should preferably be filed as issues on the git repository. Please check the list of known bugs at the GIT repository under miscellaneous.

Avoid feature bloat
Do not put new features into the main VCL files unless there is general agreement that they are needed. Special purpose features should instead be placed in add-on packages.

The coding style must follow the guidelines listed above on page 92. Do not insert metadata for Doxygen or similar tools. Follow the optimization guidelines mentioned above.

Any modification to the main VCL files should be tested with different compilers and different operating systems on the test bench described below in chapter 12.5. Avoid any files or features that are specific to a particular CPU, operating system, platform, or development tool.

Copyright is a problem. If different contributions are copyrighted by different contributors than it will be impossible to make any legal decisions regarding VCL if not all contributors can be contacted. There are plans to assign the copyright to a non-profit organization, but no particular organization has been chosen yet.

12.5 Test bench

A test bench has been developed for the purpose of automatic testing of VCL. The test bench includes C++ code and a bash script for automatic testing of operators and functions. The script will run through a list of test cases to test each operator and function with many different combinations of vector classes, instruction sets, compilers, and operating systems. Each test case will be implemented by compiling and running a small test program and comparing the resulting values with the expected values.

The test bench is used in the development of VCL. It is not intended for programmers that use the VCL. All code and documentation for the test bench is provided in the folder named testbench.

12.6 File list

File name	Purpose
manual/vcl_manual.pdf	Instruction manual (this file)

vectorclass.h	Top-level C++ header file. This will include several other header files, according to the indicated instruction set
instrset.h	Detection of which instruction set the code is compiled for, and functions that depend on the instruction set. This file also contains various common definitions and templates. Included by vectorclass.h
vectori128.h	Defines classes, operators and functions for integer vectors with a total size of 128 bits. Included by vectorclass.h
vectori256.h	Defines classes, operators and functions for integer vectors with a total size of 256 bits for the AVX2 instruction set. Included by vectorclass.h if appropriate
vectori256e.h	Defines classes, operators and functions for integer vectors with a total size of 256 bits for instruction sets lower than AVX2. Included by vectorclass.h if appropriate
vectori512.h	Defines classes, operators and functions for vectors of 32-bit and 64-bit integers with a total size of 512 bits for the AVX512F instruction set. Included by vectorclass.h if appropriate
vectori512e.h	Defines classes, operators and functions for vectors of 32-bit and 64-bit integers with a total size of 512 bits for instruction sets lower than AVX512F. Included by vectorclass.h if appropriate
vectori512s.h	Defines classes, operators and functions for vectors of 8-bit and 16-bit integers with a total size of 512 bits for the AVX512BW instruction set. Included by vectorclass.h if appropriate
vectori512se.h	Defines classes, operators and functions for vectors of 8-bit and 16-bit integers with a total size of 512 bits for instruction sets lower than AVX512BW. Included by vectorclass.h if appropriate
vectorf128.h	Defines classes, operators and functions for floating point vectors with a total size of 128 bits. Included by vectorclass.h
vectorf256.h	Defines classes, operators and functions for floating point vectors with a total size of 256 bits for the AVX and later instruction sets. Included by vectorclass.h if appropriate
vectorf256e.h	Defines classes, operators and functions for floating point vectors with a total size of 256 bits for instruction sets lower than AVX. Included by vectorclass.h if appropriate
vectorf512.h	Defines classes, operators and functions for floating point vectors with a total size of 512 bits for the AVX512F and later instruction sets. Included by vectorclass.h if appropriate
vectorf512e.h	Defines classes, operators and functions for floating point vectors with a total size of 512 bits for instruction sets lower than AVX512F. Included by vectorclass.h if appropriate
vectorfp16.h	Defines classes, operators and functions for half precision floating point vectors of all sizes, including mathematical functions, for AVX512-FP16
vectorfp16e.h	Defines emulating classes, operators and functions for half precision floating point vectors of all sizes, including mathematical functions, for processors without AVX512-FP16
vector_convert.h	Defines functions for conversion between different vector sizes, as well as some generic function templates.

vectormath_exp.h	Optional inline mathematical functions: power, logarithms and exponential functions
vectormath_trig.h	Optional inline mathematical functions: trigonometric and inverse trigonometric functions
vectormath_hyp.h	Optional inline mathematical functions: hyperbolic and inverse hyperbolic functions
vectormath_common.h	Common definitions for vectormath_exp.h, vectormath_trig.h and vectormath_hyp.h
vectormath_lib.h	Optional header file for external mathematical vector function library
instrset_detect.cpp	Optional functions for detecting which instruction set is supported at runtime
dispatch_example.cpp	Example of how to make automatic CPU dispatching
LICENSE	Apache 2.0 license
changelog.txt	VCL version history
miscellaneous/svmlpatch	Folder containing the library svmlpatch.lib as well as the source code to build it. Used for fixing a compatibillity issue with Intel SVML library in 64-bit Windows
testbench	Folder containing test bench files for testing the VCL library. This is used in the development of VCL, and is not needed by programmers using the VCL. Includes code and documentation.