VCL
C++ vector class library

Version 1.14. www.agner.org/optimize

Table of Contents

Introduction

The VCL vector class library is a tool that allows C++ programmers to 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 1a. Calculations on arrays
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 write this code as vectors:

```
// Example 1b. Same code 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
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 1b does the same as example 1a, 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 1a 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.

How it works

The type `Vec8f` in example 1b is a class that encapsulates the intrinsic type `__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 or AVX) in your compiler options.

The code in example 1b 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 half as big for instruction sets prior to AVX. The code in example 1a will generate approximately 44 instructions if the compiler does not automatically vectorize the code.

Features

- vectors of 8-, 16-, 32- and 64-bit integers, signed and unsigned
- vectors of single 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 and table look-up functions
- fast integer division
- can build code for different instruction sets from the same source code
- CPU dispatching to utilize higher instruction sets when available
- uses metaprogramming to find the best implementation for the selected instruction set and parameter values of a given operator or function
- includes several extra header files for special purposes and applications

Instruction sets supported

Since 1997, each 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,

AVX512, XOP, FMA3 and FMA4.

Platforms supported

Windows, Linux and Mac, 32-bit and 64-bit, with Intel, AMD or VIA x86 or x86-64 instruction set processor. The AVX and later instruction sets require one of the following operating system versions or later: Windows 7 SP1, Windows Server 2008 R2 SP1, Linux kernel version 2.6.30, Apple OS X Snow Leopard 10.6.8.

Compilers supported

The vector class library works with Microsoft, Intel, Gnu and Clang C++ compilers. It is recommended to use the newest version of the compiler if the newest instruction sets are used.
The forthcoming AVX512 instruction set requires at least Gnu compiler version 4.10 with binutils version 2.24, or Intel compiler version 14.0. Microsoft and Clang compilers do not yet have sufficient support for AVX512 (July 2014).

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.

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.

The C++ language does not have very good metaprogramming features, but the VCL makes the best use of the available features. Three methods are used for metaprogramming:

1. preprocessing directives.
2. templates.
3. `if` and `switch` statements with constant conditions to be optimized away.

The if and switch statements are actually seen when debugging the code, but the compiler will optimize them away in the optimized version of the code.

All three metaprogramming methods rely on expressions that can be computed at compile time. The C++ syntax requires that expressions such as template parameters can be recognized as *compile-time constants*. A compile-time constant is typically just a number, such as 5, but it may also contain operators, preprocessing macros and template parameters. It may not contain function calls, loops, etc. It may contain `?:` branches, but not `if-else` branches. An optimizing compiler may be able to resolve more calculations than these at compile time, but the syntax requires that the above rules be obeyed where a compile-time constant is required. Several of the functions in VCL require compile-time constants for certain parameters, especially where templates are used.

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 are multiplying vectors of 32-bit integers with an instruction set that supports only 16-bit multiplication then the VCL code will calculate the 32-bit product using multiple 16-bit multiplications. If you compile the same code again for a higher instruction set then you will get a more efficient program.
- 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 specific permutation pattern specified. 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 21 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 `pow_const` and `pow_rational`.

Availability

The newest version of the vector class library is available from
http://www.agner.org/optimize/vectorclass.zip

There is a discussion board for the vector class library at
http://www.agner.org/optimize/vectorclass/

License

The VCL vector class library has a dual license system. You can use it for free in open source software, or pay for using it in proprietary software.

You are free to copy, use, redistribute and modify this software under the terms of the GNU General Public License as published by the Free Software Foundation, version 3 or any later version. See the file license.txt.

Commercial licenses are available on request.

The basics

How to compile

Copy the header files (*.h) from vectorclass.zip to the same folder as your C++ source files. The header files in the sub-archive named special.zip should only be included if needed.

Include the header file vectorclass.h in your C++ source file:

```
include "vectorclass.h"
```

Several other header files will be included automatically.

Set your compiler options to the desired instruction set. The instruction set must be at least SSE2. See page 85 for a list of compiler options. You may compile multiple versions for different instruction sets as explained in the chapter starting at page 84.

If you are using the Gnu compiler version 3.x then you must set the ABI version to 4 or more, or 0 for a reasonable default. For example:

g++ -mavx -fabi-version=0 -O3 myfile.cpp

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

```
// 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;
}
```

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. 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 256 in the current version and possibly larger in future versions. You can get access to 512-bit vectors by defining

```
#define MAX_VECTOR_SIZE 512
```

before including the vector class header files.

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

The following vector classes are defined:

Integer vector classes:

vector class	integer size, bits	signed	elements per vector	total bits	recommen-ded 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
Vec2q	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
Vec16i	32	signed	16	512	AVX512
Vec16ui	32	unsigned	16	512	AVX512
Vec8q	64	signed	8	512	AVX512
Vec8uq	64	unsigned	8	512	AVX512

Note that vectors of 8-bit and 16-bit integers are not available in 512 bit vectors because of a limitation in the AVX512 instruction set.

Floating point vector classes:

vector class	precision	elements per vector	total bits	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

Boolean vector classes:

Boolean vector	for use with	elements per vector	total bits	recommended instruction set
Vec128b		128	128	SSE2
Vec16cb	Vec16c, Vec16uc	16	128	SSE2
Vec8sb	Vec8s, Vec8us	8	128	SSE2
Vec4ib	Vec4i, Vec4ui	4	128	SSE2
Vec2qb	Vec2q, Vec2uq	2	128	SSE2
Vec256b		256	256	AVX2
Vec32cb	Vec32c, Vec32uc	32	256	AVX2
Vec16sb	Vec16s, Vec16us	16	256	AVX2
Vec8ib	Vec8i, Vec8ui	8	256	AVX2
Vec4qb	Vec4q, Vec4uq	4	256	AVX2
Vec512b		512	512	AVX512
Vec16ib	Vec16i, Vec16ui	16	16 or 512	AVX512
Vec8qb	Vec8q, Vec8uq	8	8 or 512	AVX512
Vec4fb	Vec4f	4	128	SSE2
Vec2db	Vec2d	2	128	SSE2
Vec8fb	Vec8f	8	256	AVX
Vec4db	Vec4d	4	256	AVX
Vec16fb	Vec16f	16	16 or 512	AVX512
Vec8db	Vec8d	8	8 or 512	AVX512

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 integer and floating point vector classes
description	all elements get the same value
efficiency	good

Examples:

```
Vec4i a(7);   // all four elements = 7
Vec4i b = 8;  // all four elements = 8
```

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 bigger than 128 bits.
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 integer and floating point vector classes
description	changes the value of element number (index) to (value). The index starts at 0.
efficiency	medium, depending on instruction set

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 separately into the array.

This is the preferred way of putting values into a vector, except immediately after values have been put into the array one by one (see page 90).

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 very little 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 ≤ n ≤ (vector size).
efficiency	medium

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 ≤ n ≤ (vector size).
efficiency	good

Example:

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

method	set_bit(index, value)
defined for	all integer vector classes smaller than 512 bits
description	changes a single bit to 0 or 1. index starts at bit 0 of element 0 and ends with the last bit of the last element. value = 0 or 1.
efficiency	medium

Example:

```
Vec4i a(10);
a.set_bit(34, 1);   // a = (10,14,10,10)
```

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)
```

Reading 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

This is the preferred way of getting the individual elements of a vector.

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`.

method	store_partial(int n, pointer)
defined for	all integer and floating point vector classes

description	stores the first n elements into an array. 0 ≤ n ≤ (vector size).
efficiency	medium

Example:

```
float list[3] = {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)
```

method	extract(index)
defined for	all integer, floating point and boolean vector classes
description	gets a single element from a vector
efficiency	medium

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	medium

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_bit(index)
defined for	all integer vector classes smaller than 512 bits
description	reads a single bit. index starts at bit 0 of element 0 and ends with the last bit of the last element.
efficiency	medium

Example:

```
Vec4i a(10);
int b = a.get_bit(34);   // b = 0
```

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 member function indicating the number of elements that the vector can contain
efficiency	good

Example:

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

Operators

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	good for vectors of float, double, and 16-bit integers, poor for vectors of 8-bit integers and 64-bit integers, good for vectors of 32-bit integers if SSE4.1 or higher instruction set

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	poor

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. All elements are divided by the same divisor. See page 21 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 21 for explanation
efficiency	poor, but 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)
// unsigned
Vec4ui c(10, 11, 12, 13);
Vec4ui d = c / const_uint(3); // d = (3, 3, 4, 4)
```

Logic operators

operator	<<, <<=
defined for	all integer vector classes
description	logical 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	shift right. All vector elements are shifted by the same amount. Unsigned integers use logical shift, signed integers use arithmetic shift (i. e. sign bit is copied)
efficiency	good

Example:

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

operator	==
defined for	all integer and floating point 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 integer and floating point 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 integer and boolean 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 integer and floating point 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)
```

Integer division

There are no instructions in the x86 instruction set and its extensions that are useful for integer vector division, and such instructions would 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	Divisor_s
Vec16uc, Vec32uc	Divisor_us
Vec8s, Vec16s	Divisor_s
Vec8us, Vec16us	Divisor_us
Vec4i, Vec8i	Divisor_i
Vec4ui, Vec8ui	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	const_int
Vec16uc, Vec32uc	const_uint
Vec8s, Vec16s	const_int
Vec8us, Vec16us	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 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.

Functions

Integer functions

function	horizontal_add
defined for	all integer and floating point vector classes
description	calculates the sum of all vector elements
efficiency	medium

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	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	fast for Vec16uc, Vec32uc, Vec8s, Vec16s, medium for other integer vector classes

Example:

```
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	fast for Vec16uc, Vec32uc, Vec8s, Vec16s, medium for other integer vector classes

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	vector = rotate_left(vector, int)
defined for	all integer vector classes
description	rotates the bits of each element. Use a negative count to rotate right
efficiency	medium

Example:

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

function	vector shift_bytes_up(vector, int) vector shift_bytes_down(vector, int)
defined for	Vec16c, Vec32c
description	shifts the bytes of a vector up or down and inserts zeroes at the vacant places
efficiency	medium. (you may use permute functions instead if the shift count is a compile-time constant)

Example:

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

Floating point simple mathematical functions

function	horizontal_add
defined for	all floating point vector classes
description	calculates the sum of all vector elements
efficiency	medium

Example:

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

function	max
defined for	all floating point vector classes
description	returns the biggest of two vallues
efficiency	good

Example:

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

function	min
defined for	all floating point vector classes
description	returns the smallest of two vallues
efficiency	good

Example:

```
Vec4f a(1.0, 1.1, 1.2, 1.3);
Vec4f b(1.4, 1.3, 1.2, 1.1);
Vec4f c = min(a, b);          // c = (1.0, 1.1, 1.2, 1.1)
```

function	abs
defined for	all floating point vector classes
description	gets the absolute value
efficiency	good

Example:

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

function	change_sign<i0, i1, ...>(vector)
defined for	all floating point vector classes
description	changes sign of vector elements
efficiency	good

Each template parameter is 1 for changing sign of the corresponding element, and 0 for no change. Example:

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

function	sqrt
defined for	all floating point vector classes
description	calculates the square root
efficiency	poor

Example:

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

function	square
defined for	all floating point vector classes
description	calculates the square
efficiency	good

Example:

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

function	pow(vector x, int n)
defined for	all floating point vector classes
description	raises all vector elements to the same integer power. See also page 39 for pow with floating point exponent.
precision	slightly imprecise for extreme values of n due to accumulation of rounding errors
efficiency	medium

Example:

```
Vec4f a(0.0, 1.0, 2.0, 3.0);
int   b = 3;
```

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

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 extreme values of n due to accumulation of rounding errors
efficiency	medium, often better than pow(vector, int)

Example:

```
Vec4f a(0.0, 1.0, 2.0, 3.0);
Vec4f b = pow_const(a, 3);
// b = (0.0, 1.0, 8.0, 27.0)
```

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 round_to_int and round_to_int64 on page 66.
efficiency	good if SSE4.1 instruction set

Example:

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

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 truncate_to_int and truncate_to_int64 on page 67.
efficiency	good if SSE4.1 instruction set

Example:

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

function	floor

defined for	all floating point vector classes
description	rounds number towards -∞. The value is returned as a floating point vector
efficiency	good if SSE4.1 instruction set

Example:

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

function	ceil
defined for	all floating point vector classes
description	rounds number towards +∞. The value is returned as a floating point vector
efficiency	good if SSE4.1 instruction set

Example:

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

function	approx_recipr
defined for	Vec4f, Vec8f, Vec16f
description	fast approximate calculation of reciprocal. Relative accuracy better than 2^{-11}
efficiency	good

Example:

```
Vec4f a(0.5, 1.0, 2.0, 3.0)
Vec4f b = approx_recipr(a);  // b = (2.0, 1.0, 0.5, 0.333)
```

function	approx_rsqrt
defined for	Vec4f, Vec8f, Vec16f
description	fast approximate calculation of value to the power of -0.5. Relative accuracy better than 2^{-11}
efficiency	good

Example:

```
Vec4f a(1.0, 2.0, 3.0, 4.0)
Vec4f b = approx_rsqrt(a);  // b = (1.0,0.707,0.577,0.500)
```

function	exponent
defined for	all floating point vector classes
description	extracts the exponent part of a floating point number. Result is an integer vector. exponent(a) = floor(log2(abs(a))), except for a = 0
efficiency	medium

Example:

```
// single precision:
Vec4f a(1.0, 2.0, 3.0, 4.0);
Vec4i b = exponent(a);  // b = (0, 1, 1, 2)
// double precision:
Vec2d a(1.0, 2.0);
Vec2q b = exponent(a);  // b = (0, 1)
```

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), except for a = 0
efficiency	medium

Example:

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

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:

```
// single precision:
Vec4i a(-1, 0, 1, 2);
Vec4f b = exp2(a);     // b = (0.5, 1.0, 2.0, 4.0)
// double precision:
Vec2q a(-1, 0);
Vec2d b = exp2(a);     // b = (0.5, 1.0)
```

Floating point categorization functions

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	good

Example:

```
// single precision:
Vec4f  a(-1.0, 0.0, 1.0, 2.0);
Vec4fb b = sign_bit(a);  // b = (true, false, false, false)
// double precision:
Vec2d  a(-1.0, 0.0);
Vec2db b = sign_bit(a);  // b = (true, false)
```

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

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( 0.0, 1.0, 2.0, 3.0);
Vec4f  b(-1.0, 0.0, 1.0, 2.0);
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( 0.0, 1.0, 2.0, 3.0);
Vec4f  b(-1.0, 0.0, 1.0, 2.0);
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	medium

Example:

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

function	is_subnormal is_zero_or_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. is_zero_or_subnormal also returns true for elements that are zero.
efficiency	is_subnormal: medium is_zero_or_subnormal: good

Example:

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

function	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	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 precision, unlimited for double precision. This parameter can be retrieved later by the function nan_code (page 50).
efficiency	good

Example:

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

Floating point control word manipulation functions

MXCSR is a control word that controls floating point exceptions, rounding mode and subnormal 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
description	writes the MXCSR control word
efficiency	medium

Example:

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

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 $1.18 \cdot 10^{-38}$ for single precision or $2.22 \cdot 10^{-308}$ 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.
efficiency	medium

Example:

```
no_subnormals();
```

Floating point mathematical functions

Mathematical functions such as logarithms, exponential functions, power, trigonometric functions, etc. are available either as inline code or through external function libraries. These functions all take vectors as input and produce vectors as output.

The use of vector math functions is straightforward. Example:

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

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

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

```
    return 0;
}
```

Inline mathematical functions

The inline mathematical functions are made 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 speed of the inline functions is similar to or better than external vector function libraries in most cases and many times faster than standard (scalar) math function libraries.

The precision is good. The calculation error is typically below 1 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). Cases where the error can exceed 2 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 just 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. It is possible to include an error code in a NAN and detect it with the function nan_code on page 50.

There are a few cases, though, 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;
...
b = select(a > 1.0, a, 0.5);
```

Now, if an element of a is NAN then the boolean vector element in a > 1.0 will be either true or false because a boolean can have no other values. Whether it is true of false is implementation-dependent. If it happens to be false then the corresponding element in b will be 0.5, and the error is not propagated 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
   ...
}
b = select(a > 1.0, a, 0.5);
```

It is not recommended to unmask floating point exceptions. It is not guaranteed that this will generate exceptions in case of errors in these functions.

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 no_subnormals() first (see above). This will speed up some calculations and give more consistent results.

A description of each mathematical function is given below.

Using an external library for mathematical functions

As an alternative to the inline mathematical functions, you can use an external function library. Include the header file `vectormath_lib.h` for this. Set the define `VECTORMATH` to one of the following values to specify which external library you are using:

VECTORMATH value	Function library
0	Uses the standard math library that is included with the compiler. You do not have to include any extra libraries. The library function is called once for each vector element. This is slow (especially for the Gnu library). Use this option for testing purposes or where performance is not critical.
1	AMD LIBM library. The LIBM library is available for 64-bit Linux and 64-bit Windows, but not for 32-bit systems. File name: amdlibm.lib or libamdlibm.a. Performance is good for AMD processors with FMA4, but inferior for processors without FMA4. Currently, the FMA4 instruction set is supported only in AMD processors.
2	Use Intel SVML library (Short Vector Math Library) with any compiler. The SVML library is available for all platforms relevant to the vector class library. It is included with Intel C++ compilers but can be used with other compilers as well. File

	name: svml_dispmt.lib or libsvml.a. Be sure to choose the 32-bit version or 64-bit version according to the platform you are compiling for. Performance is good on Intel processors. Performance is inferior on other brands of processors unless you replace Intel's own CPU dispatch function. Link in the library libircmt.lib to use Intel's own CPU dispatch function for Intel processors, or use an object file from the asmlib library under "inteldispatchpatch" for best performance on all brands of processors. See my blog and my C++ manual for details.
3	Use Intel SVML library with an Intel compiler. You do not have to link in any extra libraries. The Intel compiler gives access to different versions with different precision. Performance is good on Intel processors, but inferior on other brands of processors unless you link in the dispatch patch from the asmlib library as described above.

The value of `VECTORMATH` can be defined on the compiler command line or by a define statement:

```
#define VECTORMATH 2
#include "vectormath_lib.h"
```

The chosen function library must be linked into your project if the value of `VECTORMATH` is 1 or 2.

Details about range, error handling and precision must be sought in the documentation for the specific library.

If you want to use both inline and library math functions then you have to include the header files for the inline functions *before* `vectormath_lib.h`. This will give you the library version for functions that are not available as inline versions.

List of mathematical functions

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 faster than most alternatives. The details listed apply to the inline version. Details for the library versions may be sought in the documentation for the specific library.

Powers, exponential functions and logarithms:

function	pow(vector, vector) pow(vector, scalar)

defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH have pow(vector, vector)
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. pow(0,0) = 1. pow(NAN,0) is NAN or 1, depending on the implementation.
precision	better than (0.1*abs(b)+1) ULP
efficiency	poor

Example:

```
Vec4f a( 1.0,  2.0, 3.0, 4.0);
Vec4f b( 0.0, -1.0, 0.5, 2.0);
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 versions	not available
description	see page 28.
efficiency	medium

function	pow_const(vector x, const int n)
defined for	all floating point vector classes
inline version	no extra header file required
library versions	not available
description	see page 29.
efficiency	medium, often better than pow(vector, int)

function	pow_ratio(vector x, const int a, const int b)
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	not available
description	Raises all elements of x to the rational power a/b. a and b must be compile-time constant integers. For example pow_ratio(x, -1, 2) gives the reciprocal square root of x. x may be zero only if a and b are positive. x may be negative only if b is odd. The result when x is infinite is implementation dependent.
range	subnormal numbers are treated as zero in some cases
precision	slightly imprecise for extreme values of a due to accumulation of rounding errors. the precision is similar to the pow function when b is not 1, 2, 3, 4, 6 or 8.
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.

function	exp
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH
description	exponential function e^x
range	double: abs(x) < 708.39. float: abs(x) < 87.3
efficiency	poor

function	expm1
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH, except 0 for some libraries
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

function	exp2
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH, except 0 for some libraries
description	2^x
range	double: abs(x) < 1022. float: abs(x) < 126.
efficiency	poor

function	exp10
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH
description	10^x
range	double: abs(x) < 307.65. float: abs(x) < 37.9.
efficiency	poor

function	log
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH
description	natural logarithm
range	The input must be a normal number. Subnormal numbers are treated as zero.
efficiency	poor

function	log1p
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH, except 0 for some libraries
description	log(1+x)

	Useful to avoid loss of precision if x is close to 0
efficiency	poor

function	log2
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH
description	logarithm base 2
efficiency	poor

function	log10
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	all values of VECTORMATH
description	logarithm base 10
efficiency	poor

function	cbrt
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	VECTORMATH = 1, 2, 3
description	cube root
range	input must be a normal number. Subnormal numbers are treated as zero
efficiency	faster than pow

function	cexp
defined for	all floating point vector classes
inline version	not available
library versions	VECTORMATH = 0, 2, 3

description	complex exponential function. Even-numbered vector elements are real part, odd-numbered vector elements are imaginary part.
efficiency	poor

Trigonometric functions (angles in radians):

function	sin(x)
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	all values of VECTORMATH
description	sine function.
range	Defined for abs(x) < $1.7 \cdot 10^9$. 0 otherwise
efficiency	poor

function	cos(x)
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	all values of VECTORMATH
description	cosine function.
range	Defined for abs(x) < $1.7 \cdot 10^9$. 1 otherwise
efficiency	poor

function	sincos(x)
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	all values of VECTORMATH, may not be available for Vec16f and Vec8d
description	sine and cosine computed simultaneously.
range	Defined for abs(x) < $1.7 \cdot 10^9$. 0 and 1 otherwise
efficiency	poor

Example:

```
Vec4f a(0.0, 0.5, 1.0, 1.5);
```

```
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(x)
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	all values of VECTORMATH
description	tangent function. tan(π/2) will not produce infinity because the value of π/2 cannot be represented exactly as a floating point number. The output will be big, though, when the input is as close to π/2 as possible.
range	Defined for abs(x) < $1.7 \cdot 10^9$. 0 otherwise
efficiency	poor

Inverse trigonometric functions (angles in radians)

function	asin
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	VECTORMATH = 0, 2, 3
description	inverse sine function
efficiency	poor

function	acos
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	VECTORMATH = 0, 2, 3
description	inverse cosine function
efficiency	poor

function	atan
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	VECTORMATH = 0, 2, 3
description	Inverse tangent. Results between $-\pi/2$ and $\pi/2$.
efficiency	poor

function	atan2
defined for	all floating point vector classes
inline version	vectormath_trig.h
library versions	VECTORMATH = 0, 2, 3
description	Inverse tangent with two parameters, x and y, gives the angle to a point in the (x,y) plane. Results between $-\pi$ and π. The result of atan2(0,0) is 0 by convention.
efficiency	poor

Hyperbolic functions and inverse hyperbolic functions:

function	sinh
defined for	all floating point vector classes
inline version	vectormath_hyp.h
library versions	VECTORMATH = 0, 2, 3
description	hyperbolic sine
range	double: abs(x) < 709.7. float: abs(x) < 89.
efficiency	poor

function	cosh
defined for	all floating point vector classes
inline version	vectormath_hyp.h
library versions	VECTORMATH = 0, 2, 3
description	hyperbolic cosine
range	double: abs(x) < 709.7. float: abs(x) < 89.
efficiency	poor

function	tanh
defined for	all floating point vector classes
inline version	vectormath_hyp.h
library versions	VECTORMATH = 0, 2, 3
description	hyperbolic tangent
efficiency	poor

function	asinh
defined for	all floating point vector classes
inline version	vectormath_hyp.h
library versions	VECTORMATH = 2, 3
description	inverse hyperbolic sine
precision	The error is less than 3 ULP for double and 4 ULP for float
efficiency	poor

function	acosh
defined for	all floating point vector classes
inline version	vectormath_hyp.h
library versions	VECTORMATH = 2, 3
description	inverse hyperbolic cosine
precision	The error is less than 5 ULP for double and 4 ULP for float
efficiency	poor

function	atanh
defined for	all floating point vector classes
inline version	vectormath_hyp.h
library versions	VECTORMATH = 2, 3
description	inverse hyperbolic tangent
efficiency	poor

Error function, etc.:

function	erf
defined for	all floating point vector classes
inline version	not available
library versions	VECTORMATH = 2, 3, and some libraries VECTORMATH = 0
description	error function
efficiency	poor

function	erfc
defined for	all floating point vector classes
inline version	not available
library versions	VECTORMATH = 2, 3, and some libraries VECTORMATH = 0
description	error function complement
efficiency	poor

function	erfinv
defined for	all floating point vector classes
inline version	not available
library versions	VECTORMATH = 2, 3

description	inverse error function
efficiency	poor

function	cdfnorm
defined for	all floating point vector classes
inline version	not available
library versions	VECTORMATH = 2, 3
description	cumulative normal distribution function
efficiency	poor

function	cdfnorminv
defined for	all floating point vector classes
inline version	not available
library versions	VECTORMATH = 2, 3
description	inverse cumulative normal distribution function
efficiency	poor

Miscellaneous:

function	mul_add mul_sub mul_sub_x
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	not available
description	mul_add(a,b,c) = a*b+c mul_sub(a,b,c) = a*b-c mul_sub_x(a,b,c) = a*b-c These functions use fused multiply-and-add (FMA) instructions if available. The compiler may use FMA anyway for expressions like a*b+c, but these functions are useful for specifying calculation order in cases like a*b+c*d
precision	The intermediate product a*b is calculated with infinite

	precision if the FMA or FMA4 instruction set is enabled. mul_sub_x has extra precision even if FMA or FMA4 is not available, just slightly less precise.
efficiency	mul_add, mul_sub: good. mul_sub_x: good if FMA or FMA4 enabled, medium otherwise

function	Vec4i nan_code(Vec4f) Vec8i nan_code(Vec8f) Vec16i nan_code(Vec16f) Vec2q nan_code(Vec2d) Vec4q nan_code(Vec4d) Vec8q nan_code(Vec8d)
defined for	all floating point vector classes
inline version	vectormath_exp.h
library versions	not available
description	Extracts an error code hidden in a NAN. This code can be generated with the functions nan4f etc. (page 34) and propagated through a series of calculations. When two NANs are combined (e. g. NAN+NAN), current processors propagate the first one. It has been suggested that future processors should OR the two values. NANs produced by CPU instructions, such as 0./0. or sqrt(-1.) do not have a code. NANs cannot propagate through integer and boolean vectors. The return value is (0x00400000 + code) for single precision and (0x0008000000000000 + code) for double. The sign bit is ignored. The return value is 0 for inputs that are not NAN.
efficiency	medium

Permute, blend, lookup and gather functions

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" + the vector type suffix, for example permute4i 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 ≤ *i* ≤ *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* = -256 means don't care (i. e. use 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 = permute4i<2,2,3,0>(a);   // b = (12, 12, 13, 10)
Vec4i c = permute4i<-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 53).

The permute functions contain a lot of metaprogramming code which is used for finding the best implementation for the given set of indexes and the specified instruction set. This metaprogramming produces a lot of extra code when compiling in debug mode, but it is reduced out 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. But in unfavorable cases where the selected instruction set has no machine instruction that matches the desired permutation pattern, it may produce many machine instructions.

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.

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. An index *i* in the interval 0 ≤ *i* ≤ *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 ≤ *i* ≤ 2**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* = -256 means don't care.

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(20, 21, 22, 23);
Vec4i c = blend4i<4,0,4,3>(a, b); // c = (20, 10, 20, 13)
```

If you want to blend input from more than two vectors, there are three different methods you can use:

1. A binary tree of blend calls, where unused values are set to don't care (-256). 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 = blend4i<0,5,-256,-256>(a, b);// r = (10,21,?,?)
Vec4i s = blend4i<-256,-256,2,7>(c, d);// s = (?,?,32,43)
Vec4i t = blend4i<0,1,6,7>(r, s);      // t = (10,21,32,43)
```

2. Set unused values to zero, and 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 = blend4i<0,5,-1,-1>(a, b);// r = (10,21,0,0)
Vec4i s = blend4i<-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.

Lookup functions:

function	Vec16c lookup16(Vec16c, Vec16c) Vec32c lookup32(Vec32c, Vec32c) Vec8s lookup8(Vec8s, Vec8s) Vec16s lookup16(Vec16s, Vec16s) Vec4i lookup4(Vec4i, Vec4i) Vec8i lookup8(Vec8i, Vec8i) Vec16i lookup16(Vec16i, Vec16i) Vec4q lookup4(Vec4q, Vec4q) Vec8q lookup8(Vec8q, Vec8q)
defined for	Vec16c, Vec32c, Vec8s, Vec16s, 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 $0 \le i \le n-1$ where n is the number of elements in a vector.
efficiency	good for AVX2, medium for lower instruction sets

function	Vec16c lookup32(Vec16c, Vec16c, Vec16c) Vec8s lookup16(Vec8s, Vec8s, Vec8s) Vec4i lookup8(Vec4i, Vec4i, Vec4i) Vec4i lookup16(Vec4i, Vec4i, Vec4i, Vec4i, Vec4i)
defined for	Vec16c, Vec8s, Vec4i
description	blend with variable indexes. The first input vector contains the indexes, the following two or four input vectors contain the data source. Each index must be in the range $0 \le i \le n-1$ where n is the number indicated by the name.
efficiency	good for AVX2, medium for lower instruction sets

function	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 $0 \le i \le n-1$ where n is the number of elements in a vector.
efficiency	good for AVX2, medium for lower instruction sets

function	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 $0 \le i \le 2*n-1$ 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) 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 $0 \le i \le n-1$, where n is

	indicated as a template parameter (n must be a positive compile-time constant).
efficiency	good for AVX2, 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. This example has a table of the function $y = x^2 - 1$

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

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 table. Indexes cannot be negative. There is no option for zeroing or don't care. If you need variable indexes, use the lookup functions instead. (If all indexes are smaller than the vector size, the function may read a full vector and permute it)
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);
```

Number ↔ string conversion functions

These functions require the header file "decimal.h" from the sub-archive named "special.zip".

Binary to binary-coded-decimal (BCD) conversion:

function	vector bin2bcd(vector)
defined for	All unsigned integer vector types
description	Each vector element is converted to BCD code. The behavior in case of overflow is implementation dependent.
efficiency	medium.

Example:

```
#include "decimal.h"
 ...
Vec4ui a(100,101,102,103);
Vec4ui b = bin2bcd(a);  // b = (0x100, 0x101, 0x102, 0x103)
// (maximum value without overflow = 99999999)
```

Binary to decimal ASCII string conversion:

function	int bin2ascii (vector a, char * string, int fieldlen, int numdat, bool signd, char ovfl, char separator, bool term)	
defined for	Vec16c, Vec32c, Vec8s, Vec16s, Vec4i, Vec8i	
description	Makes an ASCII string of numbers, where each vector element is converted to a human-readable decimal ASCII representation, right-justified in a field of specified length.	
parameters	a	Vector of signed or unsigned integers to convert
	string	Character array that will receive the string. Must be big enough to contains the worst-case string length, including separators and terminating zero.
	fieldlen	Desired length of each field in the output string. (default = 2, 4, or 8 depending on vector type)
	numdat	Number of vector elements to convert. (default = number of elements in a)
	signd	Each number will be interpreted as signed if signd = true, unsigned if false. (default = true)
	ovfl	Specifies how to handle cases where a number is too big to fit into a field of length fieldlen. ovfl = 0: the size of the field will be made big enough to hold the number (max 11 characters). ovfl = ASCII character: the field will be filled with this character if the number is too big to fit into the field. (default = '*')
	separator	Specifies an ASCII character to insert between

		fields (but not after the last field). 0 for no separator. (default = ',')
	term	Writes a zero-terminated ASCII string if term is true. The string has no terminator if term is false. (default = true)
	return value	The returned value is the length of the string written. The terminating zero is not included in the count.
efficiency	poor, but better than alternatives. Improved by instruction sets SSSE3, SSE4.1, AVX2.	

Example:

```
#include "decimal.h"
 ...
Vec4ui a(123, 123456, 0, -78);
char text[50];
bin2ascii(a, text, 5, 4, true, '*', ',', true);
// text = "  123,*****,    0,  -78"
```

Binary to hexadecimal ASCII string conversion:

function	int bin2hex_ascii (vector a, char * string, int numdat, char separator, bool term)	
defined for	All signed integer vector types	
description	Makes an ASCII string of hexadecimal numbers, where each vector element is converted to an unsigned hexadecimal ASCII representation in a field of 8, 4 or 2 characters, depending on the vector type.	
parameters	a	Vector of integers to convert
	string	Character array that will receive the string. Must be big enough to contains the string length, including separators and terminating zero.
	numdat	Number of vector elements to convert. (default = number of elements in a)
	separator	Specifies an ASCII character to insert between fields (but not after the last field). 0 for no separator. (default = ',')
	term	Writes a zero-terminated ASCII string if term is true. The string has no terminator if term is false. (default = true)
	return value	The returned value is the length of the string written. The terminating zero is not included in

		the count.
efficiency	Medium. Improved by instruction sets SSSE3, AVX2.	

Example:

```
#include "decimal.h"
 ...
Vec4ui a(256, 0x1234abcd, 0, -1);
char text[50];
bin2hex_ascii(a, text, 4, ',', true);
// text = "00000100,1234ABCD,00000000,FFFFFFFF"
```

Decimal ASCII string to binary number conversion:

function	Vec4i ascii2bin(Vec32c string)
defined for	Vec32c
description	The input vector contains an ASCII string, organized as four fields of 8 characters each. Each field contains a decimal number. There are no separator or terminator characters. Each number must be right-justified in its field. Spaces and a minus sign are allowed to the left of each number. No other characters are allowed. The function returns a vector of four signed integers. A syntax error is indicated by the value 0x80000000, which cannot occur otherwise.
efficiency	medium.

Each field must be exactly 8 characters wide. The number must have one or more digits '0' - '9'. Spaces and one minus sign are allowed to the left of the number. Nothing is allowed to the right of the number. No other characters than digits, spaces and minus sign are allowed. The syntax of the input string can be defined with the following EBNF description:

```
<string>    ::= <field> <field> <field> <field>
<field>     ::= { <space> } [ <minus> ] { <space> } <digit> { <digit> }
<space>     ::= ' '
<minus>     ::= '-'
<digit>     ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

A syntax error in a field will set the corresponding number to INT_MIN = 0x80000000. This will not affect the other numbers. It is OK to input a string where only part of the string contains valid numbers and ignore the rest because there is no performance penalty for syntax errors.

The error-value 0x80000000 = -2147483648 cannot occur with a correct input because it requires more than eight digits to represent. The numbers cannot be

bigger than 99999999 or smaller than -9999999 because they have to fit into eight characters.

The following example has a syntax error in the last field because there are spaces to the right of the number:

```
#include "decimal.h"
 ...
char str[] = "     123  -45678  - 0004   5    ";
Vec32c string = Vec32c().load(str);
Vec4i a = ascii2bin(string);
// a = (123, -45678, -4, 0x80000000)
```

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 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 the same number of elements as the vectors they are used with. There is a table on page 9 showing 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 squareroot of -1 even though we are not using it. This could possibly generate an exception if floating point exceptions are not masked. A better solution would therefore 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 used. 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:

```
// vector version:
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 to `&` and `|` for convenience, but they are still doing a bitwise AND or OR operation, so `&` and `|` are actually more representative of what these operators really do. This example should, of course, 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);
```

Internal representation of boolean vectors

The way boolean vectors are stored depends on the instruction set. For all data vectors of 128 bit and 256 bit size, the boolean vectors are stored as integer vectors with the same element size as the integer or floating point vectors they are used for. For example, the 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 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 boolean vector element is not 1, but -1. The representation of false is 0. Any other values than 0 and -1 will

most likely produce wrong and inconsistent results that depend on the instruction set.

The AVX512 instruction set allows boolean vectors to be stored internally as compact bitfields with a single bit for each vector element. This method is used in boolean vectors for use with 512 bit data vectors when compiling for AVX512 or higher instruction sets. The old method is used when compiling for AVX2 and lower instruction sets, even for 512 bit vectors.

If you want your code to be compatible with multiple instruction sets, then you should make no assumption about how a boolean vector is stored. For example, the boolean vector Vec16ib uses 16 bits of storage when compiling for AVX512, but 512 bits of storage when compiling for AVX2.

Boolean vectors for use with floating point and integer vectors are in principle identical when they have the same number of bits per element. For example, the boolean vector types `Vec8fb` and `Vec8ib` are both vectors of 8 boolean elements, using 32 bits each. These types can easily be converted to each other, but it is still recommended to choose `Vec8fb` for use with `Vec8f` and `Vec8ib` for use with `Vec8i` because this helps the compiler select the fastest implementation in each case. See page 71 for conversion of boolean vectors.

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_mul(boolean vector f, vector a, vector b)
defined for	all floating point vector classes
description	conditional multiplication. result[i] = f[i] ? a[i] * b[i] : a[i]
efficiency	good

function	vector andnot(vector, vector)
defined for	all boolean vector classes
description	andnot(a,b) = a & ~ b
efficiency	good (better than a & ~ b)

function	bool horizontal_and(boolean vector)
defined for	all boolean vector classes
description	The output is the AND combination of all elements
efficiency	medium. Better if SSE4.1 enabled

```
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. Better if SSE4.1 enabled

```
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, except Vec128b, Vec256b, Vec512b
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, 0);
Vec4ib c = a == b;
int d = horizontal_find_first(c);  // d = 1
```

function	unsigned int horizontal_count(boolean vector)
defined for	all boolean vector classes, except Vec128b, Vec256b, Vec512b
description	counts the number of elements that are true
efficiency	medium if SSE4.2 enabled

```
Example:
Vec4i  a(1, 2, 3, 4);
Vec4i  b(0, 2, 3, 0);
Vec4ib c = a == b;
int d = horizontal_count(c);  // d = 2
```

Conversion between vector types

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

method	conversion between vector class and intrinsic vector type
defined for	all 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 either an integer vector or a single integer, depending on the size and instruction set.
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)
```

Implicit conversion is convenient in the example b = a + 3, which adds 3 to all elements of the vector. Use explicit conversion where there is ambiguity about the desired vector type.

method	conversion between signed and unsigned integer vectors
defined for	all integer vector classes
description	signed ↔ unsigned conversion 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 = Vec4i(a); // b = (0x1000, 0x3002, 0x5004, 0x7006)
```

method	reinterpret_d, reinterpret_f, reinterpret_i
defined for	all integer and floating point vector classes
description	reinterprets a vector as a different type without changing any bits (bit casting). reinterpret_d is used for converting to Vec2d or Vec4d, reinterpret_f is used for converting to Vec4f or Vec8f, reinterpret_i is used for converting to any integer vector type
efficiency	good

Example

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

method	Vec4i round_to_int(Vec4f) Vec4i round_to_int(Vec2d) Vec4i round_to_int(Vec2d, Vec2d) Vec8i round_to_int(Vec8f) Vec16i round_to_int(Vec16f) Vec4i round_to_int(Vec4d) Vec8i round_to_int(Vec8d)
defined for	all floating point vector classes
description	rounds floating point numbers to nearest integer and returns integer vector. (where two integers are equally near, the even integer is returned)
efficiency	medium

Example:

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

method	Vec2q round_to_int64(Vec2d x) Vec4q round_to_int64(Vec4d x) Vec8q round_to_int64(Vec8d x) Vec2q round_to_int64_limited(Vec2d x) Vec4q round_to_int64_limited(Vec4d x) Vec8q round_to_int64_limited(Vec8d x)
defined for	Vec2d, Vec4d, Vec8d
description	rounds floating point numbers to nearest integer and

	returns integer vector. (where two integers are equally near, the even integer is returned). The _limited versions are limited to abs(x) < 2^{31}. Outside of this range, the result is implementation dependent.
efficiency	round_to_int64: poor round_to_int64_limited: medium

Example:

```
Vec4d a(1.0, 1.5, 2.0, 2.5);
Vec4q b = round_to_int64(a);  // b = (1,2,2,2)
```

method	Vec4i truncate_to_int(Vec4f) Vec4i truncate_to_int(Vec2d, Vec2d) Vec8i truncate_to_int(Vec8f) Vec16i truncate_to_int(Vec16f) Vec4i truncate_to_int(Vec4d) Vec8i truncate_to_int(Vec8d)
defined for	all floating point vector classes
description	truncates floating point numbers towards zero and returns signed integer vector.
efficiency	medium

Example:

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

method	Vec2q truncate_to_int64(Vec2d x) Vec4q truncate_to_int64(Vec4d x) Vec8q truncate_to_int64(Vec8d x) Vec2q truncate_to_int64_limited(Vec2d x) Vec4q truncate_to_int64_limited(Vec4d x) Vec8q truncate_to_int64_limited(Vec8d x)
defined for	Vec2d, Vec4d, Vec8d
description	truncates floating point numbers towards zero and returns signed integer vector. The _limited versions are limited to abs(x) < 2^{31}. Outside of this range, the result is implementation dependent.
efficiency	truncate_to_int64: poor truncate_to_int64_limited: medium

Example:

```
Vec4d a(1.0, 1.5, 2.0, 2.5);
```

```
Vec4q b = truncate_to_int64(a);  // b = (1,2,2,2)
```

method	Vec4f to_float(Vec4i) Vec8f to_float(Vec8i) Vec16f to_float(Vec16i)
defined for	Vec4i, Vec8i, Vec16i
description	converts signed integers to single precision float
efficiency	medium

Example:

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

method	Vec4d to_double(Vec4i) Vec8d to_double(Vec8i)
defined for	Vec4i, Vec8i
description	converts signed 32-bit integers to double precision float
efficiency	medium

Example:

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

method	Vec2d to_double(Vec2q x) Vec4d to_double(Vec4q x) Vec8d to_double(Vec8q x) Vec2d to_double_limited(Vec2q x) Vec4d to_double_limited(Vec4q x) Vec8d to_double_limited(Vec8q x)
defined for	Vec2q, Vec4q, Vec8q
description	converts signed 64-bit integers to double precision float. The _limited versions are limited to abs(x) < 2^{31}. Outside of this range, the result is implementation dependent.
efficiency	to_double: poor to_double_limited: medium

Example:

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

method	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 128-bit vectors can be concatenated into one 256-bit vector of the corresponding type by calling a constructor. two 256-bit vectors can be concatenated into one 512-bit vector of the corresponding type by calling a constructor.
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	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_low, extend_high
defined for	Vec16c, Vec16uc, Vec8s, Vec8us, Vec4i, Vec4ui,

	Vec32c, Vec32uc, Vec16s, Vec16us, Vec8i, Vec8ui, Vec16i, Vec16ui
description	extends integers to a larger number of bits per element. 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)
```

method	extend_low, extend_high
defined for	Vec4f, Vec8f, Vec16f
description	extends single precision floating point numbers to double precision
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)
```

method	compress
defined for	Vec8s, Vec8us, Vec4i, Vec4ui, Vec2q, Vec2uq Vec16s, Vec16us, Vec8i, Vec8ui, Vec4q, Vec4uq, Vec16i, Vec16ui, Vec8q, Vec8uq
description	reduces integers to a lower number of bits per element. Overflow and underflow wraps 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)
```

method	compress
defined for	Vec2d, Vec4d, Vec8d

description	reduces double precision floating point numbers to single precision
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)
```

method	compress_saturated
defined for	Vec8s, Vec8us, Vec4i, Vec4ui, Vec2q, Vec2uq Vec16s, Vec16us, Vec8i, Vec8ui, Vec4q, Vec4uq, Vec16i, Vec16ui, Vec8q, Vec8uq
description	reduces integers 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)
```

Conversion between boolean vector types

method	to_bits
defined for	all boolean vectors, except Vec128b, Vec256b, Vec512b
description	converts a boolean vector to an integer with one bit per element.
efficiency	good for Vec8qb, Vec16ib, Vec8db, Vec16fb when AVX512 used. Medium otherwise

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); // = 0x0C (1100 binary)
```

(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	to_Vec4ib, to_Vec8ib, to_Vec16ib, to_Vec2qb, to_Vec4qb, to_Vec8qb, to_Vec4fb, to_Vec8fb, to_Vec16fb, to_Vec2db, to_Vec4db, to_Vec8db
defined for	all boolean vectors for 32-bit and 64-bit integers, float and double
description	converts an integer bit-field to a boolean vector
efficiency	good for Vec8qb, Vec16ib, Vec8db, Vec16fb when AVX512 used. Medium or poor otherwise

Example:

```
uint8_t a = 0xC2;  // 11000010 binary
Vec8qb  b = to_vec8qb(a);
// = 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	Vec16cb → Vec16c Vec32cb → Vec32c Vec8sb → Vec8s Vec16sb → Vec16s Vec4ib, Vec4fb → Vec4i Vec8ib, Vec8fb → Vec8i Vec2qb, Vec2db → Vec2q Vec4qb, Vec4db → Vec4q Not defined for vectors bigger than 256 bits.
description	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.
efficiency	good

Example:

```
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)
```

Conversion the other way, e.g. from `Vec4i` to `Vec4ib` is possible for vector types smaller than 512 bits 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. For example:

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

Special applications

3-dimensional vectors

The header file "vector3d.h" in the sub-archive named "special.zip" defines 3-dimensional vectors for use in geometry and physics.

Vector classes defined in vector3d.h:

vector class	precision	elements per vector	total bits	recommended instruction set
Vec3f	single	3	128	SSE3
Vec3d	double	3	256	AVX

These vector classes are actually using vector registers that can hold 4 floats or 4 doubles, respectively. The last element in the vector register is not used.

Most operators and functions are similar to those of Vec4f and Vec4d. A constructor with the three coordinates is defined:

method	constructor with 3 elements as parameter
defined for	Vec3f, Vec3d
description	contents is initialized with x, y, z coordinates

Note that some operators and functions inherited from Vec4f and Vec4d make little or no sense. For example, the > operator will make a not very useful element-by-element comparison rather than comparing vector lengths:

```
Vec3f a(10,11,12);
Vec3f b(12,11,10);
Vec4fb c = a > b;    // c = (false,false,true,false)
bool d = vector_length(a) > vector_length(b); // d = false
```

Member functions:

member function	get_x(), get_y(), get_z()
defined for	Vec3f, Vec3d
description	extract a single coordinate

member function	extract(index)
defined for	Vec3f, Vec3d
description	extracts coordinate x, y or z for index = 0, 1 or 2, respectively

Arithmetic operators:

operators	+, -, *, /
defined for	Vec3f, Vec3d
description	element-by-element operation

Comparison operators:

operators	==, !=
defined for	Vec3f, Vec3d
description	returns a boolean telling if vectors are equal or not equal. The unused fourth element is ignored.

There are several different ways to multiply 3-dimensional vectors:

operator	vector * vector
defined for	Vec3f, Vec3d
description	element-by-element multiplication

operator	vector * scalar, scalar * vector
defined for	Vec3f, Vec3d
description	all elements are multiplied by the scalar

function	dot_product(vector, vector)
defined for	Vec3f, Vec3d
description	returns the dot-product as a scalar

function	cross_product(vector, vector)
defined for	Vec3f, Vec3d
description	returns the cross-product as a vector perpendicular to the two input vectors

Other functions:

function	vector_length(vector)
defined for	Vec3f, Vec3d
description	returns the length as a scalar

function	normalize_vector(vector)
defined for	Vec3f, Vec3d
description	returns a vector with unit length and same direction as the input vector

function	rotate(vector c0, vector c1, vector c2, vector a)
defined for	Vec3f, Vec3d
description	rotates vector a by multiplying with the matrix defined by the columns (c0,c1,c2). (If the rotation matrix is defined by rows then it must first be transposed to get the column vectors, see page 102 for an example).

Example:

```
Vec3f a(11,22,33);
Vec3f c0(0,1,0), c1(0,0,1), c2(1,0,0);
Vec3f b = rotate(c0, c1, c2, a); // b = (22,33,11)
```

function	to_single
defined for	Vec3d
description	converts to Vec3f

function	to_double
defined for	Vec3f
description	converts to Vec3d

Complex number vectors

The header file "complexvec.h" in the sub-archive named "special.zip" defines classes for complex numbers and complex vectors for use in mathematics and electronics.

Classes defined in complexvec.h:

vector class	precision	complex numbers per vector	total bits	recommended instruction set
Complex2f	single	1	128	SSE2
Complex4f	single	2	128	SSE2
Complex8f	single	4	256	AVX
Complex2d	double	1	128	SSE2
Complex4d	double	2	256	AVX

The class Complex2f uses the lower half of a 128-bit register, while the upper half of the register is unused. The other complex classes use a full 128-bit or 256-bit register.

The minimum instruction set is SSE2. The performance of multiplication is improved by compiling for the SSE3 instruction set. The performance of multiplication and division is improved by compiling for the FMA3 or FMA4 instruction set.

Constructors:

method	default constructor
defined for	all complex classes
description	contents is not initialized

method	constructor with real and imaginary parts
defined for	all complex classes
description	all elements are initialized with real and imaginary parts

Example:

```
Complex4f a(1.0f, 2.0f, 3.0f, 4.0f);
// a = (1+2i, 3+4i)
```

method	constructor with one real and one imaginary part
defined for	all complex classes
description	all elements are initialized with the same (real,imaginary) pair

method	constructor with one real part only
defined for	all complex classes
description	all elements are initialized with the same real number. The imaginary parts are set to zero

method	constructor with one Complex2f or Complex2d
defined for	Complex4f, Complex8f, Complex4d
description	all elements are initialized with the same (real,imaginary) pair

method	constructor with two Complex4f or four Complex2f
defined for	Complex8f
description	vectors are concatenated

Member functions:

method	load
defined for	all complex classes
description	all elements are initialized from a float or double array containing alternating real and imaginary parts

Example:

```
double x[4] = {1.0, 2.0, 3.0, 4.0};
Vec4d a;
a.load(x); // a = (1+2i, 3+4i)
```

method	get_low, get_high
defined for	Complex4f, Complex8f, Complex4d
description	get lower or upper half or the vector as a Complex2f, Complex4f, Complex2d, respectively

method	extract(index)
defined for	all complex classes
description	extract a single real or imaginary part. index = 0 gives real part of first element, index = 1 gives imaginary part of first element, etc.

Operators:

operators	+, +=, -, -=, unary minus, *, *=, /, /=
defined for	all complex classes

description	arithmetic functions between two complex numbers:
	`(a+i*b) + (c+i*d) = ((a+c) + i*(b+d))`
	`(a+i*b) - (c+i*d) = ((a-c) + i*(b-d))`
	`(a+i*b) * (c+i*d) = ((a*c-b*d) + i*(a*d+b*c))`
	`(a+i*b) / (c+i*d) = ((a*c+b*d)+i*(b*c-a*d))/(c²+d²)`

Operators combining complex and real

operators	+, +=, -, -=, *, *=, /, /=
defined for	all complex classes

description	arithmetic functions between a complex number and a real:
	`(a+ib) + c = ((a+c) + i*b)`
	`(a+ib) - c = ((a-c) + i*b)`
	`(a+ib) * c = ((a*c) + i*(b*c))`
	`(a+ib) / c = ((a/c) + i*(b/c))`

	c / (a+ib) = ((a*c) - i*(b*c))/(a^2+b^2)

Complex conjugate:

operators	~
defined for	all complex classes
description	complex conjugate of all vector elements: `~(a+i*b) = (a-i*b)`

Comparison operators:

operators	==, !=
defined for	all complex classes
description	returns a boolean for Complex2f and Complex2d. returns a boolean vector for Complex4f, Complex8f, Complex4d. The output can be used in the select function

Functions:

function	abs
defined for	all complex classes
description	abs(a+i*b) = sqrt(a*a+b*b)

function	sqrt
defined for	all complex classes
description	square root of complex number

function	select
defined for	all complex classes
description	selects between the elements of two vectors

Example:

```
Complex4f a(1,2,3,4);
Complex4f b(1,2,5,6);
Complex4f c = select(a == b, Complex4f(0), b);
// c = (0+i*0, 5+i*6)
```

function	to_single
defined for	Complex2d, Complex4d
description	converts to Complex2f, Complex4f

function	to_double
defined for	Complex2f, Complex4f
description	converts to Complex2d, Complex4d

function	cexp
defined for	all complex classes
description	complex exponential function: `cexp(a+i*b) = exp(a)*(cos(b)+i*sin(b))` For best performance, include vectormath.h before complexvec.h and use Intel SVML library as explained on page 38.

Quaternions

The header file "quaternion.h" in the sub-archive named "special.zip" defines classes for quaternions (hypercomplex numbers) for use in mathematics and geometry.

Classes defined in quaternion.h:

vector class	precision	quaternions per vector	total bits	recommended instruction set
Quaternion4f	single	1	128	SSE2
Quaternion4d	double	1	256	AVX

Constructors:

method	default constructor
defined for	Quaternion4f, Quaternion4d
description	contents is not initialized

method	constructor with real and imaginary parts
defined for	Quaternion4f, Quaternion4d
description	initialized with real and imaginary parts

Example:

```
Quaternion4f a(1.0f, 2.0f, 3.0f, 4.0f);
```

```
// a = (1 + 2*i + 3*j + 4*k)
```

method	constructor with one real part only
defined for	Quaternion4f, Quaternion4d
description	initialized with the real number. The imaginary parts are set to zero

method	constructor with two Complex2f or two Complex2d
defined for	Quaternion4f, Quaternion4d
description	The quaternion is constructed from two complex numbers: Quaternion((a+b*i),(c+d*i)) = (a+b*i) + (c+d*i)*j = a+b*i+c*j+d*k

method	constructor with vector
defined for	Quaternion4f(Vec4f), Quaternion4d(Vec4d)
description	The four vector elements go into the real part and the three imagniary parts

method	constructor from 3-dimensional vector
defined for	Quaternion4f(Vec3f), Quaternion4d(Vec3d)
description	(x,y,z) is converted to (x*i+y*j+z*k). Conversion from quaternion to 3-dimensional vector is also possible. The cross_product function for Vec3f and Vec3d corresponds to the operator * for Quaternion4f and Quaternion4d. Note that these conversions are only available if vector3d.h is included before quaternion.h

Member functions:

method	load(pointer)
defined for	Quaternion4f, Quaternion4d
description	The quaternion is read from a float or double array containing the real part followed by the imaginary parts

method	store(pointer)
defined for	Quaternion4f, Quaternion4d
description	The quaternion is stored as four values in a float or double array

method	get_low(), get_high()
defined for	Quaternion4f, Quaternion4d
description	Split the quaternion into two complex numbers. q = q.get_low() + q.get_high()*j

method	real()
defined for	Quaternion4f, Quaternion4d
description	Get the real part as a float or double

method	imag()
defined for	Quaternion4f, Quaternion4d
description	Get the imaginary parts, with the real part set to zero

method	extract(index)
defined for	Quaternion4f, Quaternion4d
description	extract a single real or imaginary part. index = 0 gives real part of first element, index = 1 gives first imaginary part, etc.

method	to_vector()
defined for	Quaternion4f, Quaternion4d
description	Convert to a vector Vec4f or Vec4d containing the real part and the imaginary parts.

Operators:

operators	+, +=, -, -=, unary minus, *, *=, /, /=
defined for	Quaternion4f, Quaternion4d
description	Arithmetic functions between two quaternions. Multiplication is not commutative. Division of quaternions is ambiguous. Here, devision is defined as

	q / r = q * reciprocal(r).

Operators combining quaternion and real

operators	+, +=, -, -=, *, *=, /, /=
defined for	Quaternion4f, Quaternion4d
description	Arithmetic functions between a quaternion and a real

Complex conjugate:

operators	~
defined for	Quaternion4f, Quaternion4d
description	The conjugate is defined as `~(a+b*i+c*j+d*k) = (a-b*i-c*j-d*k)`

Comparison operators:

operators	==, !=
defined for	Quaternion4f, Quaternion4d
description	returns a boolean

Functions:

function	abs
defined for	Quaternion4f, Quaternion4d
description	abs(a + b*i + c*j + d*k) = sqrt(a*a + b*b + c*c + d*d)

function	select
defined for	Quaternion4f, Quaternion4d
description	selects between two quaternions

function	to_single
defined for	Quaternion4d
description	converts Quaternion4d to Quaternion4f

function	to_double
defined for	Quaternion4f
description	converts Quaternion4f to Quaternion4d

Instruction sets and CPU dispatching

Almost every new generation of microprocessors has 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. The following table indicates things that are improved for each successive instruction set extension.

Instruction set	**Year introduced**	**Functions that are improved**
SSE2	2001	minimum requirement for vector class library
SSE3	2004	floating point horizontal_add
SSSE3	2006	permute, blend and lookup functions, integer horizontal_add, 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	on 128-bit integer vectors: compare, horizontal_add_x, rotate_left, blend, lookup
FMA4 AMD only	2011	floating point code containing multiplication followed by addition
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.
AVX512f	expected 2016	All operations on 512-bit integer and floating point vectors: Vec16i, Vec16ui, Vec8q, Vec8uq

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 desired instruction set can be specified on the compiler command line as follows:

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
XOP	-mavx -mxop -fabi-version=0	not available	not available	/arch:avx -D__XOP__
FMA4	-mfma4	not available	not available	not available
FMA3	-mfma	-mfma	/Qfma	not available
AVX2	-mavx2 -fabi-version=0	-mavx2	/arch:avx2	/arch:avx -D__AVX2__
AVX512F	-mavx512f	-xMIC-AVX512	/arch:MIC-AVX512	not available

The Microsoft compiler supports only a few of the instruction sets, but other instruction sets can be specified as defines which are detected in the preprocessing directives of the vector class library.

The FMA3 and FMA4 instruction sets are not handled directly by the code in the vector class library, but by the compiler. The compiler will 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.

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 run time which instruction sets are supported by the actual CPU and operating system, and selects the best version of the code accordingly. For example, you may compile the code four times for four different instruction sets: SSE2, SSE4.1, AVX2 and AVX512. The CPU dispatcher will then set a function pointer to point to the appropriate version. You can use the function `instrset_detect` (see below, page 86) to detect the supported instruction set. The file `dispatch_example.cpp` shows an example of how to make a CPU dispatcher that selects the appropriate code version. The critical part of the program is called through a function pointer. This function pointer initially points to the CPU dispatcher, which is activated the first time the function is called. The CPU dispatcher changes the function pointer to point to the best version of the code, and then continues in the selected code. The next time the function is called, the call goes directly to the right version of the code without calling the CPU dispatcher first. It is probably not necessary to make a branch for instruction sets prior to SSE2 because old computers without SSE2 are rarely in use today, and certainly not for demanding applications.

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 following functions, defined in the file `instrset_detect.cpp`, can be used for detecting at run time which instruction set is supported.

function	int instrset_detect(void)
description	returns one of these values: 0: 80386 instruction set 1: or above = SSE supported by CPU (not testing for O.S. support) 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 supported by CPU and O.S. 8: or above = AVX2 9: or above = AVX512F
efficiency	poor

function	bool hasFMA3(void)
description	returns true if FMA3 is supported
efficiency	poor

function	bool hasFMA4(void)
description	returns true if FMA4 is supported
efficiency	poor

function	bool hasXOP(void)
description	returns true if XOP is supported
efficiency	poor

Performance considerations

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 everything is possible in assembly code, but it is quite tedious and error-prone. There are far more than a thousand 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 which are often 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 optimizations 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 which 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 have not been updated to support the AVX and later instruction sets, 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 3. 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 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 which are not readily available in vector versions.
- when the algorithm involves many branches that are not easily vectorized.
- 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 present vector class library is intended as a good alternative when automatic vectorization fails to produce optimal code for any of these reasons.

Choice of compiler and function libraries

The vector class library has support for the following four compilers:

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

The Microsoft compiler optimizes reasonably well, but not as good as the other compilers, and it does not support all instruction sets.

Intel Studio / Intel Composer
This compiler optimizes very well. Intel also provides some of the best optimized function libraries for mathematical and other purposes. Unfortunately, the Intel compilers and some of the function libraries favor Intel CPUs, and often produce code that runs slower than necessary on CPUs of any other brand than Intel. It is possible to work around this limitation for the Intel function libraries and in some cases also for the compiler. See my blog and my C++ manual for details. Intel's compilers are available for Windows, Linux and Mac platforms.

Gnu C++ compiler
This compiler produced the best optimizations in my tests. The g++ compiler is available for all x86 and x86-64 platforms. The math functions in the glibc library are currently not fully optimized.

Clang C++ compiler
This compiler has recently been developed to a stage where it is feasible for our purpose. Thorough testing of the compiled code is recommended as the combination of the vector class library and the Clang compiler is quite new

(September 2013). The performance is similar to the Gnu compiler and it supports the same platforms.

Choosing the optimal vector size and precision

The time it takes to make a vector operation such as addition or multiplication typically depends on the total number of bits in the vector rather than the number of elements. For example, it takes the same time to make a vector addition with vectors of eight single precision floats (`Vec8f`) as with vectors of four double precision floats (`Vec4d`). Likewise, it takes the same time to add two integer vectors whether the vectors have eight 32-bit integers (`Vec8i`) or sixteen 16-bit integers (`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. 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 8-bit and 16-bit integers can not be used with vectors bigger than 256 bits.

The total vector size is 128 bits, 256 or 512 bits. Whether it is advantageous to use the biggest vector size depends on the instruction set. The 256-bit floating point vectors (`Vec8f` and `Vec4d`) are only advantageous when the AVX instruction set is available and enabled. The 256-bit integer vectors (`Vec32c`, `Vec16s`, `Vec8i`, `Vec4q`, etc.) are only advantageous under the AVX2 instruction set. The 512-bit integer and floating point vectors (`Vec16f`, `Vec8d`, `Vec16i`, `Vec8q`, etc.) will be available with the future AVX512F instruction set, expected in 2016.

Putting data into vectors

The different ways of putting data into vectors are listed on page 10. If the vector elements are constants known at compile time, then the fastest way is to use a constructor:

```
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` or `load_a`. 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 my microarchitecture manual). This is illustrated in the following examples:

```
// Example 1. Make vector with 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 2. 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 here!
    Vec4i d = Vec4i().load(data4);
    DoSomething(d);
}

// Example 3. 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 4. 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);
}
```

In example 1, 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 2, 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 3, 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.

In example 4, we are putting a thousand elements into an array before loading them. This is certain to avoid the store forwarding stall.

Example 3 and 4 are likely to be the best solutions. A disadvantage of example 3 is that we need an extra loop. A disadvantage of example 4 is that the large array takes more cache space.

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 way 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.

1. 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 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
```

2. 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];
}
```

3. use partial load for the last vector

```
const int datasize = 134;
const int vectorsize = 8;
int mydata[datasize];
  ... // initialize mydata
```

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

4. 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 is faster than load_partial)
        temp.cutoff(datasize - i);
    }
    sum1 += temp;          // add 8 elements
}
int sum = horizontal_add(sum1); // vector sum
```

5. make array bigger and set excess data to zero

```
const int datasize = 134;
const int vectorsize = 8;
// round up datasize to 136
const int arraysize =
    (datasize + vectorsize - 1) & (-vectorsize);
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 case 5 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 case 4 above, without causing problems. However, there is a theoretical possibility that the array is placed at the very end of the readable data area 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:

a) 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.
b) 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.
c) 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.
d) If an array is declared inside a class definition then case (a), (b) or (c) 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 16 for 128-bit vectors or divisible by 32 for 256-bit vectors. The memory page size is at least 4 kbytes, and always a power of 2. If the array is aligned by the vector size (16 or 32) 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.

A further advantage of aligning the array by 16, 32 or 64 is that reading and writing vectors from an aligned array may be faster. To align an array by 16 in Windows, write:

```
__declspec(align(16)) int mydata[1000];
```

In Unix-like systems, write:

```
int mydata[1000] __attribute__((aligned(16)));
```

It is always recommended to align large arrays for performance reasons if the code uses vectors. Unfortunately, it may be more complicated to align arrays created with operator `new`.

Using multiple accumulators

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

```
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 my 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 vector addition 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:

```
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 vector addition or subtraction plus one vector multiplication 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.

Using multiple threads

Performance can be improved by dividing the work between multiple threads on processors with multiple CPU cores. This technique is outside the scope of the present manual. The vector class library is thread-safe as long as the same vector is not accessed from multiple threads simultaneously. The floating point control word (see p. 34) is not shared between threads.

Error conditions

Runtime errors

The vector class library is generally not producing runtime error messages. An index that is 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 functions like `set_bit`, `get_bit`, `rotate`, and shift operators (`<<`, `>>`).

Boolean vectors for instruction sets lower than AVX512 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 the boolean selector input 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 for the biggest vector size compiled under the AVX512 instruction set have only one bit for each element.

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

A floating point overflow will usually produce infinity, floating point underflow produces zero, and an invalid floating point operation may produce not-a-number (NAN). Floating point exceptions can occur only if exceptions are unmasked. Unmasking floating point exceptions does not guarantee that VCL floating point functions will generate exceptions in case of error.

Mathematical functions will signal an error by producing INF or NAN, not by raising exceptions or setting an errno variable.

Compile-time errors

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. The error message may not be as informative as we could wish, due to the limitations of template metaprogramming. The error message may possibly contain the text "Static_error_check<false>".

Combination of incompatible vector classes, or other syntax errors produce compile-time error messages. These error messages may be quite long and confusing due to overloading and templates, but generally indicating the line number of the error.

"error C2719: formal parameter with __declspec(align('16')) won't be aligned". The Microsoft compiler cannot handle vectors as function parameters. The easiest solution is to change the parameter to a const reference, e. g.:

```
Vec4f my_function(Vec4f const & x) {
    ... }
```

"ambiguous call to overloaded function". Make sure all parameters have the correct type, e.g.:

```
Vec4f a, b;
b = pow(a,0.8);
// this should be:
b = pow(a,0.8f);
```

The same can happen with operators, e.g.

```
Vec4ui a;
a >>= 2;
// this should be:
a >>= 2u;
```

Link errors

"unresolved external symbol __intel_cpu_indicator". This link error occurs when you are using Intel's SVML library without including a CPU dispatcher. Link in the library libircmt.lib to use Intel's own CPU dispatch function for Intel processors, or use an object file from the asmlib library under "inteldispatchpatch" for best performance on all brands of processors. See my blog and my C++ manual for details.

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 special 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` and `gather` 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 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 should be 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`.

- 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, or an OR-combination of the two, depending on the compiler and the CPU. The sign of a NAN has no meaning and may vary.

File list

file name	purpose
VectorClass.pdf	instructions (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 various common definitions. 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 integer vectors with a total size of 512 bits for the AVX512 instruction set. Included by vectorclass.h if appropriate
vectori512e.h	defines classes, operators and functions for integer vectors with a total size of 512 bits for instruction sets lower than AVX512. 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 AVX512 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 AVX512. Included by vectorclass.h if appropriate
vectormath_lib.h	optional header file for external mathematical vector function libraries
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
special.zip/decimal.h	optional header file for conversion of integer vectors to decimal and hexadecimal ASCII number strings and vice versa
special.zip/vector3d.h	optional header file for 3-dimensional vectors
special.zip/complexvec.h	optional header file for complex numbers and complex vectors
special.zip/quaternion.h	optional header file for quaternions
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.txt	Gnu general public license
changelog.txt	change log

Examples

This example calculates the polynomial $x^3 + 2 \cdot x^2 - 5 \cdot x + 1$ on a floating point vector. The function parameter `x` is declared as a `const` reference in order to avoid problems in the Microsoft compiler. The constants `a`, `b` and `c` are declared `static` so that they don't need to be initialized at every function call. The order of calculation is specified by parentheses in order to make shorter dependency chains.

```
Vec4f polynomial (Vec4f const & x) {
    static const Vec4f a(2.0f), b(-5.0f), c(1.0f);
    return (a + x) * (x * x) + (b * x + c);
}
```

The next example transposes a 4x4 matrix.

```
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]);
}
```

or with AVX512:

```
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.

```
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.

```
#include <math.h>

#ifndef M_PI  // define pi if not defined
#define M_PI  3.14159265358979323846
#endif

// 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 const & x);
    Vec4f cos(Vec4f const & x);
};

SinTable::SinTable() {  // constructor
    // compute resolution
    resolution = float(2.0 * M_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 const & x) {
    // calculate sin by table lookup
    Vec4i index = round_to_int(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 const & x) {
    // calculate cos by table lookup
    Vec4i index = round_to_int(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;
}
```