C++ vector class library

Version 1.00 β. www.agner.org/optimize

Table of Contents

Introduction

This vector class library is a tool that makes it simpler to utilize Single-Instruction-Multiple-Data (SIMD) instruction sets such as SSE2 or AVX in C++ programs. This is best explained with an example:

```
// Example 1a. Adding list of numbers
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. Adding list of numbers as 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
avec = bvec + cvec * 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 automatically.

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.

Platforms supported

Windows, Linux and Mac, 32-bit and 64-bit, with Intel, AMD or VIA processor.

Instruction sets supported

x86 and x86-64 with SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, XOP, FMA3, FMA4. The AVX and later instruction sets require newer operating system versions (Windows 7 SP1, Windows Server 2008 R2 SP1, Linux kernel version 2.6.30, Apple OS X Snow Leopard 10.6.8).

Compilers supported

Microsoft, Intel and Gnu C++ compilers. It is recommended to use the newest version of the compiler if the newest instruction sets are used. Older compiler versions can be used up to the SSE4.2 instruction set.

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 or 256 bits
- defines almost all common operators
- boolean operations and branches on vector elements
- defines many arithmetic functions
- permute, blend and table-lookup functions
- many mathematical functions (requires external library)
- can build code for different instruction sets from the same source code
- CPU dispatching to utilize higher instruction sets when available
- uses metaprogramming (including preprocessing directives and templates) to find the best implementation for the selected instruction set and parameter values of a given operator or function

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 the program.

License

See the file license.txt.

Commercial licenses are available on request.

The basics

Overview of vector classes

Integer vector classes:

vector class	integer size, bits	signed	elements per vector	total bits	recommended instruction set
Vec16c	8	signed	16	128	SSE2
Vec16uc	8	unsigned	16	128	SSE2
Vec8s	16	signed	8	128	SSE2
Vec8us	16	unsigned	8	128	SSE2
Vec4i	32	signed	4	128	SSE2
Vec4ui	32	unsigned	4	128	SSE2
Vec2q	64	signed	2	128	SSE2
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

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

Vector classes that can be used for Boolean operations:

vector class	for use with	elements per vector	total bits	recommended instruction set
Vec128b	Vec128b	128	128	SSE2
Vec16c	Vec16c, Vec16uc	16	128	SSE2
Vec8s	Vec8s, Vec8us	8	128	SSE2
Vec4i	Vec4i, Vec4ui	4	128	SSE2
Vec2q	Vec2q, Vec2uq	2	128	SSE2
Vec256b	Vec256b	256	256	AVX2
Vec32c	Vec32c, Vec32uc	32	256	AVX2
Vec16s	Vec16s, Vec16us	16	256	AVX2
Vec8i	Vec8i, Vec8ui	8	256	AVX2
Vec4q	Vec4q, Vec4uq	4	256	AVX2
Vec4fb	Vec4f	4	128	SSE2
Vec2db	Vec2d	2	128	SSE2
Vec8fb	Vec8f	8	256	AVX
Vec4db	Vec4d	4	256	AVX

Constructing vectors and loading data into vectors

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

method	default constructor
defined for	all vector classes
description	the vector is created but not initialized. The value is unpredictable
efficiency	good

Example:

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

method	constructor with one parameter
defined for	all vector classes
description	all elements get the same value
efficiency	good for constant. Medium for variable as parameter

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 vector classes, except Vec128b, Vec256b
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 256-bit vector classes
description	concatenates two 128-bit vectors into one 256-bit vector
efficiency	good

Example:

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

method	insert(index, value)
defined for	all vector classes, except Vec128b, Vec256b
description	changes the value of element number (index) to (value). The index starts at 0.
efficiency	medium to poor, depending on instruction set

Example:

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

method	load(const pointer)
defined for	all vector classes, except Vec4fb, Vec8fb, Vec2db, Vec4db

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 57).

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 vector classes, except Vec4fb, Vec8fb, Vec2db, Vec4db
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, or divisible by 32 for 256-bit vectors. If you are not certain that the array is properly aligned then use `load` instead of `load_a`. `load_a` is more efficient than `load` on Intel Atom processor.

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
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)
```

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 vector classes, except Vec4fb, Vec8fb, Vec2db, Vec4db
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 vector classes, except Vec4fb, Vec8fb, Vec2db, Vec4db
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, or divisible by 32 for 256-bit vectors. If you are not certain that the array is properly aligned then use `store` instead of `store_a`. `store_a` is more efficient than `store` on Intel Atom processor.

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 vector classes, except Vec128b, Vec256b
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 vector classes, except Vec128b, Vec256b
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
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 256-bit vector classes
description	gets the lower half of a 256-bit vector as a 128-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 256-bit vector classes
description	gets the upper half of a 256-bit vector as a 128-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)
```

Operators

Arithmetic operators

operator	+, ++, +=
defined for	all vector classes except Booleans
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 vector classes except Booleans
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 vector classes except Booleans
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	Vec4f, Vec8f, Vec2d, Vec4d
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 integer vector classes, except 64-bit integers
description	division by scalar. All elements are divided by the same divisor. See page 15 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 integer vector classes, except 64-bit integers
description	division by compile-time constant. All elements are divided

	by the same divisor. See page 15 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. Alle 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. Alle 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 (true is represented by an element where all bits are 1)
efficiency	good

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = a == b;         // c = (0, 0, -1, 0)
```

operator	!=
defined for	all integer and floating point vector classes
description	test if not equal. Result is a Boolean vector (true is represented by an element where all bits are 1)
efficiency	good

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = a != b;         // c = (-1, -1, 0, -1)
```

operator	>
defined for	all integer and floating point vector classes
description	test if bigger. Result is a Boolean vector (true is represented by an element where all bits are 1)
efficiency	good

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = a > b;          // c = (0, 0, 0, -1)
```

operator	>=
defined for	all integer and floating point vector classes
description	test if bigger or equal. Result is a Boolean vector (true is represented by an element where all bits are 1)
efficiency	good

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = a >= b;          // c = (0, 0, -1, -1)
```

operator	<
defined for	all integer and floating point vector classes

description	test if smaller. Result is a Boolean vector (true is represented by an element where all bits are 1)
efficiency	good

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = a < b;          // c = (-1, -1, 0, 0)
```

operator	<=
defined for	all integer and floating point vector classes
description	test if smaller or equal. Result is a Boolean vector (true is represented by an element where all bits are 1)
efficiency	good

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b(14, 13, 12, 11);
Vec4i c = a <= b;         // c = (-1, -1, -1, 0)
```

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
efficiency	good

Example:

```
Vec4i a(-1, 0, 1, 2);
Vec4i b = !a;             // b = (0, -1, 0, 0)
```

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	const_int
Vec4ui, Vec8ui	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 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)
```

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	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, int)
defined for	all floating point vector classes

description	raises all vector elements to the same integer power
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(vector, const_int)
defined for	all floating point vector classes
description	raises all vector elements to the same integer power, where the integer is a compile-time constant
efficiency	medium, often better than pow(vector,int)

Example:

```
Vec4f a(0.0, 1.0, 2.0, 3.0);
Vec4f b = pow(a, const_int(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
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	round_to_int
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 an integer vector
efficiency	good

Example:

```
// single precision:
Vec4f a(1.0, 1.4, 1.5, 1.6)
Vec4i b = round_to_int(a);    // b = (1, 1, 2, 2)
// double precision:
Vec2d a(1.0, 1.4);
Vec2d b(1.5, 1.6)
Vec4i c = round_to_int(a, b); // c = (1, 1, 2, 2)
```

function	truncate
defined for	all floating point vector classes
description	truncates number towards zero. The value is returned as a floating point vector
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	truncate_to_int
defined for	all floating point vector classes
description	truncates number towards zero. The value is returned as an integer vector
efficiency	good if SSE4.1 instruction set

Example:

```
// single precision:
Vec4f a(1.0, 1.5, 1.9, 2.0)
Vec4i b = truncate_to_int(a);    // b = (1, 1, 1, 2)
// double precision:
Vec2d a(1.0, 1.4);
Vec2d b(1.5, 1.6)
Vec4i c = truncate_to_int(a, b); // c = (1, 1, 1, 2)
```

function	truncate_to_int64
defined for	Vec2d, Vec4d
description	truncates number towards zero. The value is returned as an integer vector
efficiency	poor

Example:

```
Vec2d a(1.5, 1.9)
Vec2q b = truncate_to_int64(a);    // b = (1, 1)
```

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
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
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 denormals, 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 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	is_finite
defined for	all floating point vector classes
description	returns true for elements that are normal, denormal 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 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 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_denormal
defined for	all floating point vector classes
description	returns true for denormal numbers, false for normal

	numbers, zero, INF and NAN
efficiency	medium

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_denormal(b);  // c = (false,false,true,false)
```

function	infinite4f, infinite8f, infinite2d, infinite4d
defined for	all floating point vector classes
description	returns positive infinity
efficiency	good

Example:

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

function	nan4f, nan8f, nan2d, nan4d
defined for	all floating point vector classes
description	returns positive not-a-number
efficiency	good

Example:

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

function	snan4f, snan8f, snan2d, snan4d
defined for	all floating point vector classes
description	returns a signalling NAN. (Note: you cannot always rely on a signalling NAN causing an exception)
efficiency	good

Example:

```
Vec4f  a = snan4f(); // a = (NAN, NAN, NAN, NAN)
```

Floating point control word manipulation functions

MXCSR is a control word that controls floating point exceptions, rounding mode and denormal numbers. The MXCSR has the following bits:

bit index	meaning
0	Invalid Operation Flag
1	Denormal Flag

2	Divide-by-Zero Flag
3	Overflow Flag
4	Underflow Flag
5	Precision Flag
6	Denormals Are Zeros
7	Invalid Operation Mask
8	Denormal 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_denormals
description	Disables the use of 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 denormal numbers. The handling of denormal numbers is extremely time-consuming on many CPUs. The no_denormals function sets the "denormals are zeros" and "flush to zero" mode to avoid the use of denormal numbers. It is recommended to call this function at the beginning of a program or thread if extremely low numbers are likely to occur and it is acceptable to replace these numbers by zero.
efficiency	medium

Example:

```
no_denormals();
```

Floating point mathematical library functions

Mathematical functions such as logarithms, exponential functions, trigonometric functions, etc. are available through external function libraries. You get access to the vector math functions by including the header file "vectormath.h".

You can choose between the following mathematical function libraries and indicate your choice through the define `VECTORMATH`:

VECTORMATH value	Function library
0	Uses the standard math library that is included with the compiler. You don't 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. Filename: 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. Filename: svml_dispmt.lib or libsvml.a. Be sure to choose the 32-bit version or 64-bit version according 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 dispatching function. Link in the library libircmt.lib to use Intel's CPU dispatching function, or use the example in the file inteldispatchpatch.cpp for better performance on non-Intel processors. See my blog for details.
3	Use Intel SVML library with an Intel compiler. You don't 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.

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

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

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

The use of a vector math function is straightforward. Example:

```
#include <stdio.h>
#define  VECTORMATH  0
#include "vectorclass.h"
#include "vectormath.h"

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

    for (int i = 0; i < 4; i++) {
        printf("%6.4f ", b[i]);   // output results
    }
    printf("\n");
    return 0;
}
```

The available vector math functions are listed below. The efficiency is listed as

poor because these functions take longer time to execute than simple arithmetic functions, but the vector math libraries are nevertheless much faster than alternatives.

Powers, exponential functions and logarithms:

function	exp
defined for	all floating point vector classes, all values of VECTORMATH
description	exponential function
efficiency	poor

function	expm1
defined for	all floating point vector classes, all values of VECTORMATH, except 0 for some libraries
description	exp(x) - 1. Useful to avoid loss of precision if x is close to 0
efficiency	poor

function	exp2
defined for	all floating point vector classes, all values of VECTORMATH, except 0 for some libraries
description	2^x
efficiency	poor

function	exp10
defined for	all floating point vector classes, all values of VECTORMATH
description	10^x
efficiency	poor

function	pow
defined for	all floating point vector classes, all values of VECTORMATH
description	pow(a,b) = a^b where a and b are both vectors.

	See also pow function page 21.
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)
```

function	log
defined for	all floating point vector classes, all values of VECTORMATH
description	natural logarithm
efficiency	poor

function	log1p
defined for	all floating point vector classes, 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, all values of VECTORMATH
description	logarithm base 2
efficiency	poor

function	log10
defined for	all floating point vector classes, all values of VECTORMATH
description	logarithm base 10
efficiency	poor

function	cubic_root

defined for	all floating point vector classes, VECTORMATH = 1, 2, 3
description	cubic root = pow(x, 1./3.)
efficiency	poor

function	recipr_sqrt
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	reciprocal squareroot = pow(x, -0,5)
efficiency	poor

function	cexp
defined for	all floating point vector classes, VECTORMATH = 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
defined for	all floating point vector classes, all values of VECTORMATH
description	sine function
efficiency	poor

function	cos
defined for	all floating point vector classes, all values of VECTORMATH
description	cosine function
efficiency	poor

function	sincos
defined for	all floating point vector classes, all values of VECTORMATH
description	sine and cosine computed simultaneously
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
defined for	all floating point vector classes, all values of VECTORMATH
description	tangent function
efficiency	poor

Inverse trigonometric functions

function	asin
defined for	all floating point vector classes, VECTORMATH = 0, 2, 3
description	inverse sine function
efficiency	poor

function	acos
defined for	all floating point vector classes, VECTORMATH = 0, 2, 3
description	inverse cosine function
efficiency	poor

function	atan
defined for	all floating point vector classes,

	VECTORMATH = 0, 2, 3
description	inverse tangent function. atan(a) = inverse tangent(a) atan(a, b) = inverse tangent(a / b)
efficiency	poor

Hyperbolic functions and inverse hyperbolic functions:

function	sinh
defined for	all floating point vector classes, VECTORMATH = 0, 2, 3
description	hyperbolic sine
efficiency	poor

function	cosh
defined for	all floating point vector classes, VECTORMATH = 0, 2, 3
description	hyperbolic cosine
efficiency	poor

function	tanh
defined for	all floating point vector classes, VECTORMATH = 0, 2, 3
description	hyperbolic tangent
efficiency	poor

function	asinh
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	inverse hyperbolic sine
efficiency	poor

function	acosh
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	inverse hyperbolic cosine
efficiency	poor

function	atanh
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	inverse hyperbolic tangent
efficiency	poor

Error function, etc.:

function	erf
defined for	all floating point vector classes, VECTORMATH = 2, 3, and some libraries VECTORMATH = 0
description	error function
efficiency	poor

function	erfc
defined for	all floating point vector classes, VECTORMATH = 2, 3, and some libraries VECTORMATH = 0
description	error function complement
efficiency	poor

function	erfinv
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	inverse error function
efficiency	poor

function	cdfnorm
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	cumulative normal distribution function
efficiency	poor

function	cdfnorminv
defined for	all floating point vector classes, VECTORMATH = 2, 3
description	inverse cumulative normal distribution function
efficiency	poor

Permute, blend and lookup 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 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 which value it puts in this position).

Example:

```
Vec4i a(10, 11, 12, 13);
Vec4i b = permute<2,2,3,0>(a);   // b = (12, 12, 13, 10)
Vec4i c = permute<-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 (see page 39).

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 = permute<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 = blend<0,5,-256,-256>(a, b); // r = (10,21,?,?)
Vec4i s = blend<-256,-256,2,7>(c, d); // s = (?,?,32,43)
Vec4i t = blend<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 = blend<0,5,-1,-1>(a, b); // r = (10,21,0,0)
Vec4i s = blend<-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) Vec4q lookup4(Vec4q, Vec4q)
defined for	Vec16c, Vec32c, Vec8s, Vec16s, Vec4i, Vec8i, Vec4q
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) Vec2d lookup2(Vec2q, Vec2d) Vec4d lookup4(Vec4q, Vec4d)
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 $0 \le i \le 2n-1$ 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) Vec4q lookup<n>(Vec4q index, void const * table) Vec4f lookup<n>(Vec4i index, float const * table) Vec8f lookup<n>(Vec8i const & index, float const * table) Vec2d lookup<n>(Vec2q index, double const * table) Vec4d lookup<n>(Vec4q const & index, 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.

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])
```

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 interpreted as a Boolean vector, which chooses between the elements of the second and third vector parameter. The relational operators `>`, `>=`, `<`, `<=`, `==`, `!=` produce Boolean vectors, which accept the Boolean operations `&`, `|`, `^`, `~`. There is a table on page 5 showing which vector classes can be used as Boolean vectors for these operators.

In the above example, the Boolean vectors must have the same number of elements per vector as `a`, `b`, `c` and `d`. A value of true is represented by -1, i.e. all bits in the vector element are 1. False is represented by 0. It is important to note that the `select` function and the Boolean operators will not work correctly here if a vector element that is supposed to represent a Boolean value has any other value than 0 or -1. The behavior for other values than 0 and -1 is implementation dependent and different for different instruction sets.

The vector elements that are not selected are calculated anyway because you cannot calculate a part of a vector. 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] = {-1.0f, 0.0f, 1.0f, 2.0f};
float b[4];
for (int i = 0; i < 4; i++) {
    if (a > 0.0f && sqrt(a) < 8.0f) b = a; else b = 1.0f;
}
```

and the vector version of the same:

```
// vector version:
Vec4f a(-1.0f, 0.0f, 1.0f, 2.0f);
Vec4f b = select((a > 0.0f) & (sqrt(a) < 8.0f), a, 1.0f);
```

In the array version, we will never call `sqrt(-1)` because the `&&` operator doesn't evaluate the second operand when the first operand is false. But in the vector version we are indeed calculating `sqrt(-1)` 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.

The vector class library defines separate vector classes for use as Boolean vectors in connection with floating point vectors. For example, in the above example where `a` is an object of class `Vec4f`, the expression `(a > 0.0f)` is an object of class `Vec4fb`. The class `Vec4fb` does exactly the same as the class `Vec4i` when used as Boolean. Both are Boolean vectors with 4 elements of 32 bits each. The reason why we have defined a separate Boolean vector class for use with floating point vectors is that it enables us to produce faster code. (Many modern CPU's have separate execution units for integer vectors and floating point vectors. It is sometimes possible to do the Boolean operations in the floating point unit and thereby avoid the delay from moving data between the two units). The following table shows the Boolean vector classes that are intended for use with floating point vectors:

Boolean vector for float	Use with floating point vector class	Equivalent Boolean vector for integers
Vec4fb	Vec4f	Vec4i
Vec8fb	Vec8f	Vec8i
Vec2db	Vec2d	Vec2q
Vec4db	Vec4d	Vec4q

The equivalent Boolean vectors can be converted to each other, for example:

```
Vec4f  a(-1.0f, 0.0f,  1.0f,  2.0f);
Vec4f  b( 1.0f, 0.0f, -1.0f, -2.0f);
Vec4fb c = a > b;      // c = (false, false, true, true)
Vec4i  d = c;          // d = (0, 0, -1, -1)
```

However, you should remember when converting an integer vector to Boolean, that the only allowed values are 0 and -1. For example:

```
Vec4i  a(-1, 0, 1, 2);
Vec4fb b = a;    // will not work: 1 and 2 are wrong
```

It is possible to use the bitwise operators `&`, `|`, `^`, `~` with any data vectors. For example, if `b` is a Boolean vector and `a` is some other kind of data, then `b & a` will be equal to `a` if `b` is true, and zero if `b` is false:

```
Vec4f  a(-1.0f, 0.0f,  1.0f,  2.0f);
Vec4f  c( 1.0f, 0.0f, -1.0f, -2.0f);
Vec4fb b = a > c;
Vec4f  d = Vec4f(b) & a;   // d = (0.0, 0.0, 1.0, 2.0);
Vec4f  e = select(b, a, 0.0f);  // same, but slower
```

The vector classes `Vec128b` and `Vec256b` contain 128 and 256 Booleans of 1 bit each, respectively. These classes can be used with the operators `&`, `|`, `^`, `~` but they are not useful for selecting data from the other vector classes.

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, a + 10);  // b = (9,10,1,2)
```

function	bool horizontal_and(vector)
defined for	all integer vector classes, Vec4fb, Vec8fb, Vec2db, Vec4db
description	the input is a vector used as boolean. The output is the AND combination of all elements
efficiency	medium

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

function	bool horizontal_or(vector)
defined for	all integer vector classes, Vec4fb, Vec8fb, Vec2db, Vec4db

description	the input is a vector used as boolean. The output is the OR combination of all elements
efficiency	medium

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

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 can be done implicitly or explicitly.
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 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 vector classes
description	reinterprets a vector as a different type without changing any bits. 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, Vec2d)

	Vec8i round_to_int(Vec8f) Vec4i round_to_int(Vec4d)
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) Vec4q round_to_int64(Vec4d)
defined for	Vec2d, Vec4d
description	rounds floating point numbers to nearest integer and returns integer vector. (where two integers are equally near, the even integer is returned)
efficiency	poor

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) Vec4i truncate_to_int(Vec4d)
defined for	all floating point vector classes
description	truncates floating point numbers towards zero and returns 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)

	Vec4q truncate_to_int64(Vec4d)
defined for	Vec2d, Vec4d
description	truncates floating point numbers towards zero and returns integer vector.
efficiency	poor

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)
defined for	Vec4i, Vec8i
description	converts 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)
defined for	Vec4i
description	converts 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) Vec4d to_double(Vec4q)
defined for	Vec2q, Vec4q
description	converts 64-bit integers to double precision float
efficiency	poor

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 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 vector classes
description	two 128-bit vectors can be concatenated into one 256-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 vector classes
description	one 256-bit vector can be split into two 128-bit vectors 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,
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
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
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
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
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)
```

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	same as FMA4

AVX2	2013	All operations on 256-bit integer vectors: Vec32c, Vec32uc, Vec16s, Vec16us, Vec8i, Vec8ui, Vec4q, Vec4uq

The vector class library makes it possible to compile for different instruction sets from the same source code by using preprocessing branches. 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 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	-mavx	/arch:avx	/arch:avx
XOP	-mxop	not available	not available	/arch:avx -D__XOP__
FMA4	-mfma4	not available	not available	not available
FMA3	-mfma	-mfma	/Qfma	not available
AVX2	-mavx2	-mavx2	/arch:avx2	/arch:avx -D__AVX2__

The Microsoft compiler supports only a few of the instruction sets, but the remaining 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 any 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.

There is no advantage in using the 256-bit floating point vector classes (Vec8f, Vec4d) unless the AVX instruction set is specified, but it can be convenient to use these classes anyway if the same source code is used with and without AVX. Each 256-bit vector will simply be split up into two 128-bit vectors when compiling without AVX. Likewise, a 256-bit integer vector (e.g. Vec8i) will be split up into

two 128-bit vectors when compiling without AVX2.

It is recommended to make an automatic CPU dispatcher that detects at runtime 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 three times for the three different instruction sets: SSE2, SSE4.1 and AVX. The CPU dispatcher should then set a function pointer to point to the appropriate version. You can use the function `instrset_detect` (see below, page 53) 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.

The AVX2 instruction set is not yet available in any CPU (May 2012), but it is supported in the compilers. It is not recommended to make automatic CPU dispatching for AVX2 until it can be properly tested, because the AVX2 compiler support is not fully stable yet. (The AVX2 support in the vector class library has been tested with Intel's emulator).

There is an important restriction when you are combing code compiled for different instruction sets: Do not transfer any data *as vectors* between different pieces of code that are compiled for different instruction sets, because the vectors may be represented differently under the different instruction sets. More specifically, 256-bit floating point vectors are represented differently when compiled with and without AVX, and 256-bit integer vectors are represented differently when compiled with and without AVX2. 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 runtime 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
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. Such optimizations are not always used in assembly code because they make the code unwieldy and unmanageable. Consequently, the combination on intrinsic code 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 classes provide only the most basic functionality, and Intel has done very little to promote, support or develop it. 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.

This 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 1000+ 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 transform an algorithm that loops through an array and does some calculations on each array element to vector code.

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 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.

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

Choise of compiler and function libraries

The vector class library has support for the following three 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 Intel and Gnu compilers.

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 favorize 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 is come 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 currently available for all x86 and x86-64 platforms, except 64-bit Windows.

The math functions in the glibc library are not fully optimized.

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 4 single precision floats (`Vec4f`) as with vectors of two double precision floats (`Vec2d`). Likewise, it takes the same time to add two integer vectors whether the vectors have four 32-bit integers (`Vec4i`) or eight 16-bit integers (`Vec8s`). 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 vectors rather than double precision vectors. However, you should also take into account the time it takes to convert data from one precision to another. Therefore, it is not good to mix different precisions.

The total vector size is either 128 bits or 256 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.

Putting data into vectors

The different ways of putting data into vectors are listed on page 5. 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 a 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 a whole 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 or 32 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[134];
```

In unix-like systems, write:

```
int mydata[134] __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. 27) 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 behavior 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` and `lookup`. The same applies to a bit-index that is out of range in functions like `set_bit`, `get_bit`, `rotate`, and shift operators (`<<`, `>>`).

The only allowed values for a Boolean vector element 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.

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.

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) {
    ... }
```

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. Use the Intel library libirc.lib if you want to use Intel's CPU dispatcher, or use the example in the file inteldispatchpatch.cpp for optimal performance on all brands of CPUs.

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 set lower than AVX2. 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
vectormath.h	optional header file for mathematical vector function libraries
instrset_detect.cpp	functions for detecting which instruction set is supported at runtime. Must be included in project if needed
dispatch_example.cpp	example of how to make automatic CPU dispatching
inteldispatchpatch.cpp	example of alternative CPU dispatcher for Intel SVML library for better performance on other brands of CPUs than Intel
license.txt	Gnu general public license

Examples

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

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

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 = 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;
}
```