The full computational power of modern CPUs can only be harnessed by using their SIMD vector units. Vector instructions can conveniently be accessed from high-level languages like C or C++ by means of vector intrinsics. However, these intrinsics are specific for the chosen vector extension and for the data type of the vector elements. Porting to a different vector extension or changing the data type requires modifications of many intrinsics and results in the duplication of large portions of code. The low-level C++ template SIMD library described in this report wraps vector intrinsics in C++ templates or overloaded functions for which the vector extension and the data type can be changed with minimal effort for entire portions of the code. In addition, C++ template meta-programming can be exploited to produce generic code for arbitrary vector extensions and data types.


This document relates to release CODE12 of WarpingSIMDStandAlone.¹

¹Available from www.ti.uni-bielefeld.de/html/people/moeller/tsimd_warpingsimd.html
Contents

1 Introduction .............................................. 1

2 Element Data Types ..................................... 3

3 Vector Templates Classes ................................. 4

4 Vector Template Functions and Overloaded Functions .... 5
4.1 Introduction ............................................ 5
4.1.1 Template Specialization vs. Overloading ............ 5
4.1.2 Overloaded Functions .............................. 6
4.1.3 Template specializations ............................ 8
4.1.4 Implementation Details ......................... 9
4.2 Workarounds .......................................... 9
4.3 Template Functions for Standard Intrinsics .......... 11
4.4 Convenience Template Functions .................... 13
4.5 Type-Specific Template Functions .................... 14
4.6 Generalized Template Functions ..................... 14
4.6.1 Generalized \texttt{unpack()} ...................... 14
4.6.2 Generalized type packing: \texttt{packs()} .......... 16
4.6.3 Generalized type extension: \texttt{extend()} ...... 17
4.7 Template Functions for Data Swizzling and Unswizzling ... 19
4.7.1 Single Set of Input Vectors ..................... 19
4.7.2 Double Set of Input Vectors .................... 22
4.8 Implementation of AVX Functions ................... 22
4.9 Implementation Details .............................. 25

5 Generic Vector Template Functions .................... 26
5.1 Type Conversion ...................................... 26
5.2 Float Operations on Arbitrary Input and Output Types ... 28
5.3 Multi-Vector Load and Store .......................... 30
5.4 Memory Copy ......................................... 30
5.5 Various Functions .................................... 30

6 Template Meta-Programming ............................ 31
6.1 Horizontal Addition .................................. 31
6.2 Register Transpose .................................. 33
6.3 Inter-Lane Swizzling .................................. 38

7 Multi-Vector Template Functions ....................... 39

8 Application Examples .................................... 41
8.1 Horizontal Binomial Filter ......................... 41
8.2 Vertical Edge Filter .................................. 42
8.3  Visual Compass ......................................................... 43
8.4  Minimum of all Pixels of an Image ............................... 44
8.5  Average of Two Arrays Considering Invalid Entries ........... 44

9  Open Problems  .......................................................... 45
  9.1  Not Yet Implemented ................................................. 45
  9.2  Masked Operations ................................................... 45
  9.3  Pointer Arguments and Results ................................... 46

References ................................................................. 46
1 Introduction

Modern CPUs come equipped with SIMD vector extensions, such as Intel’s SSE* or AVX* extensions (the asterisk being a place-holder for several extension steps) or the NEON extension of ARM CPUs. Broadly speaking, such extensions comprise wide vector registers (e.g. 16 byte in SSE* and NEON, 32 byte in AVX*) on which vector instructions operate. These simultaneously apply the same operation to multiple vector elements of the same type (SIMD: single instruction, multiple data). Many programs can be noticeably accelerated by using vector instructions instead of the traditional sequential instructions. C/C++ compilers are getting better at automatically generating these vector instructions from unmodified code (Lockless Inc., accessed 2015), but their full power can only be exploited by using SIMD vector data types and so-called vector intrinsics provided as language extensions by several compilers. Vector intrinsics look like function calls in C/C++ but are directly mapped onto vector machine instructions (in most cases onto a single vector instruction).

Intrinsics provide an easy-to-use C/C++ interface to vector instructions without assembly programming and with all advantages of high-level languages (automatic register handling, optimization, better readability). On the downside, the fixed relationship between intrinsic and instruction binds code using intrinsics to a specific vector instruction set and to a specific element data type. For example, the instruction paddw behind the intrinsic _mm_add_epi16() operates on SSE vector registers of 16 byte width (_mm prefix) and adds signed 16-bit words (epi16 suffix). When switching to another vector extension (e.g. from SSE* to AVX*) or to another element data type (e.g. using integer double words instead of words), other vector intrinsics have to be used (e.g. _mm256_add_epi16() or _mm_add_epi32()), and thus extensive changes in the source code are required.

Additional complexity is added since Intel vector instructions were introduced in several extension steps (SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2; AVX, AVX2) and therefore some CPU models provide only a subset of the full instruction sets. Other manufacturers provide completely different instruction sets (e.g. NEON). Moreover, even the full instruction set can have “holes”, i.e. some vector instructions are not available for some data types. In these cases it is necessary to provide workarounds for missing operations.

The goal of the low-level C++ template SIMD library presented here is to, first, simplify code modifications when the vector extension or the element data type changes, and, second, to hide the complex details of the instruction set from the user. C++ templates in combination with function overloading are the methods to specifically accomplish the first goal, while the second goal is accomplished as a by-product.

With respect to the first goal, vector data types are wrapped in C++ templates classes and vector intrinsics are wrapped in C++ template functions or defined as overloaded

---

2The template SIMD library compiles under C++98 and C++11.
functions. Both template classes and template functions have the vector register width (in bytes) and the element data type as template parameters, and specific values for width (e.g. 16, 32) and types (e.g. SIMDInt, SIMDFloat) can be provided as template arguments when the templates are instantiated. Template specialization is used to implement the vector template class and vector template function for different width and element data types. Overloaded functions are defined for template class arguments of different vector types and vector width and selected by the compiler according to these arguments.

Flexibility and portability is achieved by using preprocessor definitions (#define), type declarations (typedef), or templated code to select the width and type arguments for entire portions of code. The advantages of this are obvious: Minimal changes in the application code are sufficient to switch to another vector extension and thus the same application code runs on different CPUs. Switching to another data type is easy as well, which is useful if the chosen integer type turns out to be too small, or if prototype code uses a floating point type or integer double words (such that overflow is unlikely) whereas the production code uses smaller integer types to optimally exploit the parallelism of the vector instructions (e.g. 16 bytes instead of only 4 floats can reside in a SSE register).

Another aspect influenced the design of the SIMD template library: Switching from existing code using intrinsics to template classes and template/overloaded functions should be possible with minimal structural changes of the code. Therefore, the vector template classes only provide a minimal set of methods while non-member functions are used to implement the vector operations. In this way, vector data types can be replaced by template class instantiations (e.g. __m128 x, y becomes SIMDVec<SIMDFloat, 16> x, y) and vector intrinsics by vector template function instantiations or overloaded functions (e.g. _mm_add_ps(x, y) becomes add(x, y) where the arguments x, y of type SIMDVec<SIMDFloat, 16> determine which of the overloaded functions is selected). When rewriting the code, just the extension prefix _mm_ and the type suffix _ps have to be removed. However, while straight-forward porting is possible for SSE intrinsics, it can be more complex for AVX code since some AVX instructions operate separately on two 128-bit lanes rather than on the entire 256-bit register (see section 4.8). Moreover, some operations may require more porting effort for the generalization to arbitrary vector width (e.g. shuffling).

Since the basic data unit is a single SIMD vector (which is mapped onto a single vector register) and the template/overloaded functions closely correspond to intrinsics, the template SIMD library provides a low-level interface. Higher-level data types like vectors or matrices in the mathematical sense and the corresponding operations on arbitrary numbers of elements are not supported, but this functionality could easily be built on top of the template SIMD library. Operator overloading is only provided for convenience.

---

Currently, SSE* and AVX* vector extensions are supported for Intel CPUs, and the vector width (16, 32) is used to switch between them. The NEON vector extension of ARM CPUs is supported as well (width 16). To implement vector instruction sets with the same width, different header files with template specializations or overloaded functions are included, e.g. 16 for both SSE* and NEON. This is not a problem as Intel and ARM code cannot be used in the same program anyhow.
on top of this functionality as its value is probably limited: Addition, subtraction, multiplication etc. come in different flavors in vector instructions (saturated/non-saturated, high/low result part), some vector intrinsics cannot be mapped on a single operator (e.g. _mm_andnot_si128()), and the bulk of intrinsics has no relation to operators (e.g. data re-organization intrinsics like _mm_unpacklo_epi16()). Moreover, as pointed out above, using the core names of intrinsics simplifies the porting of intrinsics-based code to the vector template library.

Specifically with respect to Intel vector intrinsics, the SIMD template library has another advantage: Since the same vector data type is used for all integer element types, and as not all vector intrinsics reveal what element type is currently handled (e.g. _mm_unpacklo_epi16() could operate on words or on pairs of bytes), the intention behind some intrinsics-based code may be difficult to discern. Here the SIMD template library improves the clarity of the code as the type of a vector object is strictly specified, and reveals type errors (e.g. when passing a vector of bytes to a function expecting a vector of words). The price to pay for this is the necessity to explicitly cast between different integer types.

2 Element Data Types

In the present version of the C++ template SIMD library (“T-SIMD” for short), six element data types are defined: SIMDByte (8 bit unsigned), SIMDSignedByte (8 bit signed), SIMDWord (16 bit unsigned), SIMDSHORT (16 bit signed), SIMDInt (32 bit signed), and SIMDFloat (IEEE-754 single precision floating point, 32 bit). These type names are used in the specialization of the templates classes and functions. They can also be used to define the data structures processed by vector template functions.

A template class SIMDTypeInfo<typename T> is provided with specializations for all supported element data types. It offers the functions name() (returning a string with the name of the type) and format() (returning a fprintf() format specifier for the type), the bool constants isSigned, isFloatingPoint, and isInteger, and the functions min() and max() of type T (for SIMDFloat, the constant min is the most negative number representable, in contrast to numeric_limits where it is the smallest positive number). A type NextLargerType is defined depending on the template parameter T. It performs the following type mapping:

4 Other integer types are only sparingly supported by Intel CPUs and were therefore not included. Vector operations on double precision floating-point numbers are provided by Intel CPUs, but are currently not supported by T-SIMD as the level of parallelism is small (e.g. only 2 elements in SSE); moreover, T-SIMD aims more at applications like image processing or neural networks where numerical precision is often less important.
T-SIMD introduces the following primary template class:

```
1 template <typename T, int SIMD_WIDTH>
2 class SIMDVec;
```

This template is specialized for the different element types and vector extensions. For SSE* and AVX*, only two instantiations each are required as all integer types can be mapped onto the same vector data type. Thus, for SSE*, a *partial* specialization is provided which covers all integer types (see vector data type member in line 5):

```
1 template <typename T>
2 class SIMDVec<T, 16>
3 {
4     public:
5         __m128i xmm;
6     enum { elements = 16 / sizeof(T), bytes = 16 };  
7     SIMDVec() {}
8     SIMDVec(const __m128i &x) { xmm = x; }
9     SIMDVec& operator=(const __m128i &x) { xmm = x; return *this; }
10    operator __m128i() const { return xmm; }
11  
```

while the only floating-point type supported is covered by a *full* specialization (vector data type member in line 5):

```
1 template <>
2 class SIMDVec<SIMDFloat, 16>
3 {
4     public:
5         __m128 xmm;
6     enum { elements = 16 / sizeof(SIMDFloat), bytes = 16 };  
7     SIMDVec() {}
8     SIMDVec(const __m128 &x) { xmm = x; }
```

---

<table>
<thead>
<tr>
<th>T</th>
<th>NextLargerType</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMDByte</td>
<td>SIMDWord</td>
</tr>
<tr>
<td>SIMDSignedByte</td>
<td>SIMDShort</td>
</tr>
<tr>
<td>SIMDWord</td>
<td>SIMDInt</td>
</tr>
<tr>
<td>SIMDShort</td>
<td>SIMDInt</td>
</tr>
<tr>
<td>SIMDInt</td>
<td>SIMDInt</td>
</tr>
<tr>
<td>SIMDFloat</td>
<td>SIMDFloat</td>
</tr>
</tbody>
</table>
The enum identifier `elements` holds the number of elements in the vector, `bytes` the number of bytes in the vector.

An additional constructor and two operators were defined as in Agner Fog’s Vector Class Library (Fog, accessed 2016). These considerably simplify the coding of the template functions: `SIMDVec` arguments can be passed to intrinsics expecting vector data types, results of intrinsics (vector data types) can be returned in functions with return type `SIMDVec` and can be assigned to variables of type `SIMDVec`.

The corresponding AVX* code uses the width 32 and `__m256i`, `__m256` instead of 16 and `__m128i`, `__m128`, respectively. No other methods are defined in the template classes.

NEON intrinsics use a strict type concept. Therefore only a primary `SIMDVec` template class is defined. The vector type member is determined via a type template which in turn is specialized using preprocessor macros for different element types.

4 Vector Template Functions and Overloaded Functions

4.1 Introduction

4.1.1 Template Specialization vs. Overloading

Vector intrinsics are either wrapped in template functions or in overloaded functions which operate on vector template classes. The vector template functions, as the template classes, have the element data type and the vector width of the arguments and/or of the return type as template parameters; in same cases, additional integer parameters are provided. Overloaded functions can either be pure functions or can be template functions with their own template parameters (such as immediate integer arguments). Pure overloaded functions are defined for a specific combination of vector element type and vector with of the arguments (typically the same as the result type); templated overloaded functions can be defined for immediate integer arguments or vector element types (if the same vector intrinsic can by used for several vector element types).

The mixed design with vector template functions and overloaded functions (introduced in release CODE7) results from two restrictions imposed by the vector hardware and by the C++ language standard:
**Hardware restriction** In Intel vector units (SSE*, AVX*), integer arguments like shift lengths are encoded as *immediate* arguments directly in the instruction, and can therefore only be constants, not variables or expressions involving variables in C++. Depending on the chosen optimization level, the C++ compiler may even fail to compute constant expressions at compile time. It is therefore necessary to provide immediate arguments as template parameters.

**C++ language restriction** While template classes can be partially specialized (i.e. some parameters are fixed, some are left free), the C++ standard unfortunately doesn’t permit the partial specialization of template functions. Therefore, a vector template function using immediate arguments (which need to remain free) cannot be specialized for fixed vector element type and vector width. In addition, it is not possible to fix the vector width and write a specialized function with a free vector element type in cases where the same vector intrinsic can be used for different vector element types of the same vector width.

The solution is to use overloaded functions (which can themselves depend on template parameters) instead of function template specialization in most cases. However, overload resolution only operates on the argument types of the functions, not on the return types. In cases where multiple functions with the same argument types but different return types exist (as in conversion functions) or for functions without arguments (as in load functions), template specialization is used.⁵

### 4.1.2 Overloaded Functions

An example for an overloaded function is `add()`. Since addition intrinsics are specific for the vector element type and vector width, overloaded definitions of `add()` have to be provided for each combination, e.g. for `SIMDWord` and 16 byte vector width (SSE*)

```cpp
static SIMD_INLINE SIMDVec<SIMDWord,16> add(const SIMDVec<SIMDWord,16> &a,
                                           const SIMDVec<SIMDWord,16> &b)
{
    return _mm_add_epi16(a, b);
}
```

or for `SIMDFloat` and 32 byte vector width (AVX*)

```cpp
static SIMD_INLINE SIMDVec<SIMDFloat,32> add(const SIMDVec<SIMDFloat,32> &a,
                                           const SIMDVec<SIMDFloat,32> &b)
{
    return _mm256_add_ps(a, b);
}
```

⁵Note that the value of specialization of function templates is debated, see Sutter (accessed 2016).
An example where the same vector intrinsic can be used for all integer element types is a bit-wise (not element-wise) operation like `or()`:

```cpp
// all integer versions
template <typename T>
static SIMD_INLINE SIMDVec<T,16> or(const SIMDVec<T,16> &a, const SIMDVec<T,16> &b) {
   return _mm_or_si128(a, b);
}
```

Only for the element type `SIMDFloat`, a separate overloaded and non-templated version of `or()` has to be provided:

```cpp
// float version
static SIMD_INLINE SIMDVec<SIMDFloat,16> or(const SIMDVec<SIMDFloat,16> &a, const SIMDVec<SIMDFloat,16> &b) {
   return _mm_or_ps(a, b);
}
```

According to the standard, pure functions are preferred over templated functions during overload resolution such that the non-templated version is used for `SIMDFloat` but the templated version for all other types (in this case all integer types).

In vector intrinsics expecting an immediate argument, this can be provided as an integer template argument to a templated overloaded function, as in `srai()`:

```cpp
template <int IMM>
static SIMD_INLINE SIMDVec<SIMDInt,16> srai(const SIMDVec<SIMDInt,16> &a) {
   return _mm_srai_epi32(a, IMM);
}
```

Using overloaded functions, constant expressions to compute immediate arguments can be resolved at compile time, such as in `alignre()` which defines an element-wise alignment using an intrinsic `_mm_alignr_epi8()` which expects a byte-wise alignment:

```cpp
template <int IMM>
static SIMD_INLINE __m128i x_mm_alignr_epi8(__m128i h, __m128i l) {
```
5     return _mm_alignr_epi8(h, l, IMM);
6 }
7
8 // all integer versions
9 template <int IMM, typename T>
10 static SIMD_INLINE SIMDVec<T, 16>
11 alignre(const SIMDVec<T, 16> &h,
12         const SIMDVec<T, 16> &l)
13 {
14     return x_mm_alignr_epi8<IMM * sizeof(T)>(h, l);
15 }
16

Overload resolution is used for the following functions: store(), storeu(),
stream_store(), extract(), add(), adds(), sub(), subs(), neg(),
mul(), div(), ceil(), floor(), round(), truncate(), rcp(),
rsqrt(), sqrt(), min(), max(), abs(), extend(), srai(), srli(),
slli(), hadd(), hadds(), hsub(), hsubs(), srle(), slle(), elem0(),
alignre(), ifelse(), cmp*(), and(), or(), andnot(), xor(), not(),
div2r0(), div2rd(), avg(), test_all_zeros(), test_all_ones().

4.1.3 Template specializations

Template specialization is used for the remaining functions. An example is a function
without arguments such as setzero(). The primary template is defined as

1 template <typename T, int SIMD_WIDTH>
2 static SIMD_INLINE SIMDVec<T, SIMD_WIDTH>
3 setzero();

where the template parameters have to be explicitly supplied:

1 SIMDVec<SIMDByte,16> a;
2 a = setzero<SIMDByte,16>();

In some cases, it proved to be helpful to reverse the order of the template parameters, e.g.
for load():

1 template <int SIMD_WIDTH, typename T>
2 static SIMD_INLINE SIMDVec<T, SIMD_WIDTH>
3 load(const T *const p);
template arguments and deduce others, but only if the former come at the beginning of
the template parameter list while the latter come at the end. In the example, it is sufficient
to write

1. SIMDVec<SIMDFloat,32> a;
2. SIMDFloat *p;
3. a = load<32>(p);

The following vector functions are defined as template specializations:
reinterpret(), setzero(), set1(), load(), loadu(), packs(), cvts(),
swizzle(). All generic functions which do not directly relate to vector intrinsics
(section 5) are defined in this way as well.

4.1.4 Implementation Details

All primary template functions and all overloaded functions are defined as static and
SIMD_INLINE, the latter being defined with the g++ attribute\(^5\)

1. #define SIMD_INLINE inline __attribute__((always_inline))

This prevents the compiler from producing versions of the functions that can be called
from outside the module, and forces the compiler to inline the template functions instead
of using function calls.

In all functions, the methods of the template class allow to directly pass SIMDVec
arguments to intrinsics (expecting builtin vector data types) and to return a builtin vector data
type as SIMDVec.

4.2 Workarounds

Intel SSE* instructions were introduced in several steps (section 1). If processors don’t
have the full set of SSE* instructions (everything up to SSE4.2) — which is, for exam-
ple, the case for some Intel Atom CPUs where SSE4.1 and SSE4.2 are not available —,
workarounds are required. These are hidden inside the T-SIMD template functions. The
user can just use the template functions without having to deal with these details, but has
to be aware that workarounds are more costly.

Examples are the following workaround (taken from Intel (accessed 2016b)) for the over-
loaded function extract() for SIMDByte arguments:

\(^*\)Attributes are compiler-specific and have to be defined for each compiler. Currently, only g++ and icc
is supported.
template <int IMM>
static SIMD_INLINE int
x_mm_extract_epi8(__m128i a)
{
#ifdef __SSE4_1__
    return _mm_extract_epi8(a, IMM);
#else
    return (((IMM & 0x1) == 0) ?
        x_mm_extract_epi16<(IMM>>1)>(a) & 0xff :
        x_mm_extract_epi16<(IMM>>1)>(_mm_srli_epi16(a, 8)));
#endif
}

template <int IMM>
static SIMD_INLINE SIMDByte
extract(const SIMDVec<SIMDByte,16> &a)
{
    return x_mm_extract_epi8<IMM>(a);
}

and the workaround (from VCL (Fog, accessed 2016); also other workarounds in T-SIMD were taken from VCL) for min() for element type SIMDWord

static SIMD_INLINE SIMDVec<SIMDWord,16>
min(const SIMDVec<SIMDWord,16> &a, const SIMDVec<SIMDWord,16> &b)
{
#ifdef __SSE4_1__
    return _mm_min_epu16(a, b);
#else
    __m128i signbit = _mm_set1_epi32(0x80008000);
    __m128i a1 = _mm_xor_si128(a, signbit); // add 0x8000
    __m128i b1 = _mm_xor_si128(b, signbit); // add 0x8000
    __m128i m1 = _mm_min_epi16(a1, b1); // signed min
    return _mm_xor_si128(m1, signbit); // sub 0x8000
#endif
}

where the input vectors are transformed from unsigned to signed, the minimum is determined with an intrinsic for signed numbers, and the result is transformed back from signed to unsigned. Note that if the workaround is used inside a loop, the compiler probably moves the initialization of signbit (line 9) out of the loop.

On Intel CPUs, the T-SIMD library requires at least SSE2 (then the preprocessor symbol _SIMD_VEC_16_AVAIL_ is defined). If SSE3 or SSSE3 is missing, the corresponding intrinsics are implemented sequentially such that the code at least compiles (but will be slow). If SSSE3 or better is available, all template functions use vector intrinsics, but workarounds are used for some depending on the extension steps available.
AVX* extensions have been introduced in only two steps (AVX, AVX2), with the bulk of integer instructions being introduced with AVX2. If at least AVX is available, _SIMD_VEC_32_AVAIL_ is defined. If only AVX is available, integer instructions are implemented via SSE workarounds (then _SIMD_VEC_32_FULL_AVAIL_ is undefined). If also AVX2 is available, _SIMD_VEC_32_FULL_AVAIL_ is defined. Therefore, workarounds are only required for “holes” in the full (AVX2) instruction set.

4.3 Template Functions for Standard Intrinsics

Many T-SIMD template functions are directly mapped onto the corresponding intrinsics and share the center portion of the Intel intrinsic name (see Intel Intrinsics Guide, Intel, accessed 2016a): setzero(), set1(), load(), loadu(), store(), storeu(), extract(), add(), adds(), sub(), subs(), min(), max(), abs(), hadd(), hadds(), hsub(), hsubs(), cmp*() (where the asterisk indicates conditions lt, le, eq, neq, ge, gt), srai(), srl(), slli(), and(), or(), xor(), andnot(), not().\(^7\)^\(^8\)

An exception of the naming convention is stream_store() where the intrinsic core name is just stream (which is more consistent as the corresponding load intrinsic is stream_load()). Currently not implemented are insert() (for which often specific solutions are more efficient), set()\(^9\), and reverse load/store. The overloaded and templated functions slre() and slle() perform element-wise shift (which leads to clearer code than with the byte-wise shift of the underlying intrinsics). The same holds for the overloaded and templated function alignre() which performs element-wise (rather than byte-wise) alignment. The overloaded functions test_all_ones() and test_all_zeros() both receive only a single argument (whereas the _mm_test_all_zeros() expects an additional mask) and return an int indicating whether the argument contains only 1 bits or only 0 bits, respectively. Their names relate to the corresponding SSE intrinsics (AVX intrinsics are called testz and testc).

The above-mentioned template and overloaded functions are typically provided for all 6 element data types (section 2). Exceptions are abs() and neg() which are only available for signed types,\(^10\) hadd() and hsub() which are not available for SIMDByte and SIMDSignedByte, hadds() and hsubs() which are not available

\(^7\)For Intel CPUs, the unary not() is always implemented by workarounds since, surprisingly, not is not supported by intrinsics.

\(^8\)The compiler option -fno-operator-names is required to avoid name conflicts for and, or, xor, not.

\(^9\)The functionality of set() may be difficult to implement as the number of parameter varies with vector width and element data type.

\(^10\)For abs() it would be straight-forward to provide specializations for unsigned integers, but code applying abs() should probably not be executed for unsigned types. For neg() it would be unclear what the result should be on unsigned input types.
for SIMDByte, SIMDSignedByte, and SIMDWord,\(^{11}\) srli() and slli() which are not supported for SIMDFloat, and srai() which is not supported for SIMDByte, SIMDSignedByte, and SIMDFloat.

Saturated addition and subtraction is a special case. For SIMDInt and SIMDFloat, for which these operations are not supported by SSE* and AVX*, the overloaded functions adds() and subs() are mapped onto non-saturated addition and subtraction intrinsics. The motivation for this deviation from the one-to-one mapping is that workarounds would be costly and overflow in these data types can be more easily avoided that in the other element data types. The user has to be aware of this limitation and use the appropriate scaling of the processed data to avoid overflow.

As most other T-SIMD template functions, the cmp*() intrinsics have the same element data type of input and output. This is not an obvious choice for comparisons as the output vector is a mask where all bits of elements where the comparison yields false are 0 and all bits of elements where the comparison yields true are 1. These masks can be used for a selection operation in the style of the ?: operator of C (template function ifelse(), see below), but only if the element size (number of bytes) is the same in the mask and in the arguments provided for selection. Thus only masks for SIMDByte and SIMDSignedByte, for SIMDWord and SIMDShort, and for SIMDInt and SIMDFloat would be interchangeable (e.g. a mask from a comparison of SIMDInt could be used for a selection of SIMDFloat). It seemed to be easier to not introduce three additional types for condition masks but to ask the user to clarify the intention by using a reinterpretation cast operation (reinterpret(), see below). For convenience, second-level overloaded template functions for interchangeable masks are provided.

The ifelse() function is mapped onto blendv intrinsics or onto workarounds if these are not available. It receives three vector arguments cond, trueVal, and falseVal and selects either the element from trueVal if the condition mask cond is true (1 bits) for this element, or the element from falseVal if the mask is false (0 bits). This operation most closely resembles the ?: operator of C but with the above-mentioned type restrictions. An example is given in section 8.5.

The template function reinterpret() is either mapped to a C++ reinterpret_cast() cast template (for reinterpretation between integer element types which for Intel CPUs share the same specialization of SIMDVec) or to cast vector intrinsics (for reinterpretation between SIMDFloat and all integer element types):

\[
\begin{align*}
1 & \quad \text{// primary template} \\
2 & \quad \text{template <typename Tdst, typename Tsrc, int W> } \\
3 & \quad \text{static SIMD_INLINE SIMDVec<Tdst,W>} \\
4 & \quad \text{reinterpret(const SIMDVec<Tsrc,W>& vec)}
\end{align*}
\]

\(^{11}\)For horizontal addition and subtraction, complex workarounds would be required, and it might be better if the user is aware of the missing intrinsics for these types.
return reinterpret_cast<const SIMDVec<Tdest,W>&>(vec);

// example of specialization

// example of specialization

template <>
SIMD_INLINE SIMDVec<SIMDFloat,16>
reinterpret(const SIMDVec<SIMDByte,16>& vec)
{
    return _mm_castsi128_ps(vec);
}

4.4 Convenience Template Functions

Some operations are useful but not available as vector intrinsics, and therefore are provided as template functions for convenience. The unary neg() overloaded function performs a sign change, the unary not() performs a bit-wise not, and the unary elem0() extracts the element with index 0 from the vector. The overloaded function div2r0() divides the input by 2 and rounds the result towards zero for integer types; div2rd() does the same but rounds down (no rounding is performed for SIMDFloat in both functions). A synonym avgru() is provided for avg() which indicates that for integer types the average is rounded up (for SIMDFloat, the result is not rounded). A function avgrd() where for integer types the average is rounded down (no rounding for SIMDFloat) is implemented using tag dispatching (see section 5.1):

template <typename T, int SIMD_WIDTH>
SIMD_INLINE SIMDVec<T, SIMD_WIDTH>
avgrd(IsFloatingPoint<true>,
    const SIMDVec<T, SIMD_WIDTH> &a,
    const SIMDVec<T, SIMD_WIDTH> &b)
{
    SIMDVec<T, SIMD_WIDTH> one = set1<T, SIMD_WIDTH>(1), as, bs, lsb;
    as = div2rd(a);
    bs = div2rd(b);
    return add(lsb, add(as, bs));
}

template <typename T, int SIMD_WIDTH>
SIMD_INLINE SIMDVec<T, SIMD_WIDTH>
avgrd(IsFloatingPoint<false>,
    const SIMDVec<T, SIMD_WIDTH> &a,
    const SIMDVec<T, SIMD_WIDTH> &b)
{
    SIMDVec<T, SIMD_WIDTH> one = set1<T, SIMD_WIDTH>(1), as, bs, lsb;
    lsb = and(and(a, b), one);
    as = div2rd(a);
    bs = div2rd(b);
    return add(lsb, add(as, bs));
}
4.5 Type-Specific Template Functions

A fundamental idea of T-SIMD is that element types can be changed easily and the code still works after such a change. This requires that all template and overloaded functions are available for all 6 types or at least for the majority of these. However, some functions break with this rule and are available only for few types or type combinations.\(^{12}\)

One group are the overloaded functions \texttt{mul()}, \texttt{div()}, \texttt{ceil()}, \texttt{floor()}, \texttt{round()}, \texttt{truncate()}, \texttt{rcp()}, \texttt{rsqrt()}, and \texttt{sqrt()} which are only defined for \texttt{SIMDFloat}. Integer multiplication works differently from floating-point multiplication, integer division is not available in the Intel vector extensions, and code using the other three operations is probably not portable to integer element types.

The template function \texttt{cvts()} is only provided for the (saturated) conversion from \texttt{SIMDInt} to \texttt{SIMDFloat} and vice versa. These are used by other template functions such as \texttt{packs()}. For SSE* and AVX*, saturated conversion \texttt{cvts()} from \texttt{SIMDFloat} to \texttt{SIMDInt} deviates from the underlying intrinsic: It avoids overflow as this leads to the “invalid int” result 0x80000000 which unfortunately encodes a negative number. The result is therefore clamped at the maximal positive \texttt{SIMDFloat} which is convertible to \texttt{SIMDInt} without triggering overflow (2147483520.0f). The template function \texttt{cvts()} is also defined for the opposite direction (\texttt{SIMDInt} to \texttt{SIMDFloat}), but no saturation is necessary in this case.

4.6 Generalized Template Functions

Generalization is necessary to support template meta-programming (section 4.6.1) or where type conversion operations have to be supported for arbitrary type combinations (sections 4.6.2 and 4.6.3).

4.6.1 Generalized \texttt{unpack()}\(^1\)

\texttt{unpack()} intrinsics come in two versions: those unpacking the low and those unpacking the high half of the input vectors. Since \texttt{unpack()} is the basis of transpose (and possible also of data swizzling) operations (see section 6), they were generalized. For SSE* types, the following “hub” function is defined:

\begin{verbatim}
  template <int PART, int NUM_ELEMS, typename T>

12\(^{12}\)In some applications, the element data types may be fixed, thus even type-specific functions are useful.
static SIMD_INLINE SIMDVec<T, 16> unpack(const SIMDVec<T, 16> &a, const SIMDVec<T, 16> &b) {
    return unpack(a, b, Part<PART>(), Bytes<NUM_ELEMS * sizeof(T)>());
}

The template parameter PART can either be 0 or 1 for the version operating on the low or high half. The template parameter NUM_ELEMS indicates how many elements of the element data type are transported as a unit. The hub function call is redirected to specific functions by tag dispatching (see section 5.1) relating to the part and number of bytes that are handled as a block, e.g.

template <typename T>
static SIMD_INLINE SIMDVec<T, 16> unpack(const SIMDVec<T, 16> &a, const SIMDVec<T, 16> &b, Part<0>, Bytes<2>) {
    return _mm_unpacklo_epi16(a, b);
}

unpacks the lower half, transporting 2 bytes as a unit, and

template <typename T>
static SIMD_INLINE SIMDVec<T, 16> unpack(const SIMDVec<T, 16> &a, const SIMDVec<T, 16> &b, Part<1>, Bytes<4>) {
    return _mm_unpackhi_epi32(a, b);
}

unpacks the higher half, transporting 4 bytes as a unit.

This generalization paves the way for generic code using template meta-programming (see section 6).

Since often both the lower and the higher half of the same input data are unpacked, the command zip() is provided. Using zip() in these situations is advantageous for compilation on ARM NEON, since this function is performed in a single machine instruction. The inverse of zip() is called unzip(). This function uses a single machine instruction on ARM NEON, but requires multiple instructions on Intel CPUs. Both functions have the number of elements which are transported as a block as template argument.
4.6.2 Generalized type packing: packs ()

The packs() function converts vectors of element types with larger width and packs them into vectors of element types with smaller width (with the s indicating saturation). Packing is only possible from signed to signed or from signed to unsigned types (as only these operations are supported by Intel intrinsics). The template functions come in two flavors: In the first (non-generalized) one, packing is done from an element type to one of the next smaller element type, e.g. from SIMDInt (4 byte) to SIMDWord (2 byte) or for SIMDShort (2 byte) to SIMDSignedByte (1 byte), but not e.g. from 4 byte to 1 byte:

```cpp
1 template <typename Tout, typename Tin, int SIMD_WIDTH>
2 static SIMD_INLINE SIMDVec<Tout, SIMD_WIDTH>
3 packs(const SIMDVec<Tin, SIMD_WIDTH> &a,
4       const SIMDVec<Tin, SIMD_WIDTH> &b);
```

Workarounds are hidden inside these functions which are used for functions of the second flavor.

In the second flavor, packing is generalized, i.e. arbitrary types can be packed, the only limitation being that the packed type has less-or-equal bytes than the input type. The “hub” function is defined as:

```cpp
1 template <typename Tout, typename Tin, int SIMD_WIDTH>
2 static SIMD_INLINE SIMDVec<Tout, SIMD_WIDTH>
3 packs(const SIMDVec<Tin, SIMD_WIDTH> *const a)
4 {
5   return packs(a,
6       OutputType<Tout>(),
7       Compression<sizeof(Tin)/sizeof(Tout)>());
8 }
```

Here the input is an array of vectors, the size of which depends on the input and output types (Tin, Tout). Tag dispatching (see section 5.1) is used to redistribute the calls to specific implementations. Packing can involve zero stages (compression 1) as in

```cpp
1 template <typename T, int SIMD_WIDTH>
2 static SIMD_INLINE SIMDVec<T, SIMD_WIDTH>
3 packs(const SIMDVec<T, SIMD_WIDTH> *const a,
4       OutputType<T>, Compression<1>)
5 {
6   return *a;
7 }
```

or as in the following case where conversion is involved
template <int SIMD_WIDTH>
static SIMD_INLINE SIMDVec<SIMDFloat, SIMD_WIDTH> packs(const SIMDVec<SIMDInt, SIMD_WIDTH> *const a, 
    OutputType<SIMDFloat>, Compression<1>)
{
    return cvts<SIMDFloat>(*a);
}

or can involve a single stage as in

template <typename Tout, typename Tin, int SIMD_WIDTH>
static SIMD_INLINE SIMDVec<Tout, SIMD_WIDTH> packs(const SIMDVec<Tin, SIMD_WIDTH> *const a, 
    OutputType<Tout>, Compression<2>)
{
    return packs<Tout>(a[0], a[1]);
}

or two stages as in

template <typename Tout, typename Tin, int SIMD_WIDTH>
static SIMD_INLINE SIMDVec<Tout, SIMD_WIDTH> packs(const SIMDVec<Tin, SIMD_WIDTH> *const a, 
    OutputType<Tout>, Compression<4>)
{
    // always via SIMDShort
    return packs<Tout>(packs<SIMDShort>(a[0], a[1]),
                        packs<SIMDShort>(a[2], a[3]));
}

To determine the number of input vectors, either the following macro or the template function can be used:

#define NUM_INPUT_SIMDVECS(TOUT,TIN) ((sizeof(TOUT) < sizeof(TIN)) ? (sizeof(TIN) / sizeof(TOUT)) : 1)

template <typename Tout, typename Tin>
static SIMD_INLINE int numInputSIMDVecs()
{
    return NUM_INPUT_SIMDVECS(Tout,Tin);
}

4.6.3 Generalized type extension: extend()

The opposite direction — extending a type to a larger-or-equal one — is made possible by conversion intrinsics such as _mm_cvtepu8_epi16(). Type extension is implemented by the overloaded function extend(). Supported is the sign extension from
signed to signed, and the zero extension from unsigned to unsigned and from unsigned to signed types (with the exception of zero-stage extensions, see below). Multiple output vectors (type Tout) are produced from a single input vector (Tin).

In the implementation, the extension can involve zero stages as in

```c
// all types
template <typename T>
static SIMD_INLINE void
extend(const SIMDVec<T, 16> &vIn,
      SIMDVec<T, 16> *const vOut)
{
    *vOut = vIn;
}
```

with the restriction that no conversion between signed and unsigned types of the same size and vice versa is possible (as this may destroy the content). It can involve a single stage as in

```c
static SIMD_INLINE void
extend(const SIMDVec<SIMDShort, 16> &vIn,
      SIMDVec<SIMDInt, 16> *const vOut)
{
    #ifdef __SSE4_1__
    vOut[0] = _mm_cvtepi16_epi32(vIn);
    vOut[1] = _mm_cvtepi16_epi32(_mm_srli_si128(vIn, 8));
    #else
    vOut[0] = _mm_srai_epi32(_mm_unpacklo_epi16(vIn, vIn), 16);
    vOut[1] = _mm_srai_epi32(_mm_unpackhi_epi16(vIn, vIn), 16);
    #endif
}
```

or two stages as in

```c
static SIMD_INLINE void
extend(const SIMDVec<SIMDByte, 16> &vIn,
      SIMDVec<SIMDFloat, 16> *const vOut)
{
    #ifdef __SSE4_1__
    vOut[0] = _mm_cvtepi32_ps(_mm_cvtepu8_epi32(vIn));
    vOut[1] = _mm_cvtepi32_ps(_mm_cvtepu8_epi32(_mm_srli_si128(vIn, 4)));
    vOut[2] = _mm_cvtepi32_ps(_mm_cvtepu8_epi32(_mm_srli_si128(vIn, 8)));
    vOut[3] = _mm_cvtepi32_ps(_mm_cvtepu8_epi32(_mm_srli_si128(vIn, 12)));
    #else
    __m128i zero = _mm_setzero_si128();
    __m128i lo8 = _mm_unpacklo_epi8(vIn, zero);
    vOut[0] = _mm_cvtepi32_ps(_mm_unpacklo_epi16(lo8, zero));
    vOut[1] = _mm_cvtepi32_ps(_mm_unpackhi_epi16(lo8, zero));
    __m128i hi8 = _mm_unpackhi_epi8(vIn, zero);
    vOut[2] = _mm_cvtepi32_ps(_mm_unpacklo_epi16(hi8, zero));
    vOut[3] = _mm_cvtepi32_ps(_mm_unpackhi_epi16(hi8, zero));
    #endif
}
```
The number of output vectors can be determined by either the following macro or the template function:

```c
#define NUM_OUTPUT_SIMDVECS(TOUT,TIN) 
((sizeof(TOUT) > sizeof(TIN)) ? (sizeof(TOUT) / sizeof(TIN)) : 1)

template <typename Tout, typename Tin>
static SIMD_INLINE int
numOutputSIMDVecs()
{
  return NUM_OUTPUT_SIMDVECS(Tout,Tin);
}
```

### 4.7 Template Functions for Data Swizzling and Unswizzling

Data swizzling and unswizzling are required to convert data streams from “array-of-structures” (AoS) to “structure-of-arrays” (SoA) format and back, respectively. The SoA format is much better suitable for SIMD processing than AoS. T-SIMD currently implements data swizzling in two different versions (unswizzling only in one). Only the special case of structures with \( n \) members of the same type is considered. In the following example, \( n = 3 \) SSE* registers with SIMDWord elements are swizzled:

```plaintext
input stream (structures indicated by curly brackets):
{0 1 2} {3 4 5} {6 7 8} {9 10 11} ... {21 22 23}
input vectors:
v[0] = 0 1 2 3 4 5 6 7
v[1] = 8 9 10 11 12 13 14 15
output vectors:
v[0] = 0 3 6 9 12 15 18 21
v[1] = 1 4 7 10 13 16 19 22
v[2] = 2 5 8 11 14 17 20 23
```

An example would be the swizzling of RGB data into separate R, G, and B arrays.

Swizzling is currently implemented in two different ways. The function `swizzle()` expects a single set of \( n \) vectors as input (the inverse `unswizzle()` operation is not implemented). The function `swizzle2()` expects a double set of \( 2 \times n \) vectors as input (the inverse `unswizzle2()` operation also exists).

#### 4.7.1 Single Set of Input Vectors

Swizzling for a single set of \( n \) vectors as input is a relatively complex operation for which T-SIMD can currently offer no generalized solution but only specific solutions for dif-
different values of \( n \) and for all element data types. As several of these solutions require precomputed lookup tables or masks, a generic template class

\[
\text{template <int N, typename T, int SIMD_WIDTH>}
\]

\[
\text{struct SwizzleTable;}
\]

is defined which is passed as an argument to the \text{swizzle()} “hub” function

\[
\text{template <int N, typename T, int SIMD_WIDTH>}
\]

\[
\text{static SIMD_INLINE void}
\]

\[
\text{swizzle(const SwizzleTable\text{<N, T, SIMD_WIDTH}> &t,}
\]

\[
\text{SIMDVec\text{<T, SIMD_WIDTH>} *const v)}
\]

\[
\{
\]

\[
\text{swizzle(t, v,}
\]

\[
\text{Int\text{<N>},}
\]

\[
\text{TypeIsIntSize\text{<T>}{}};
\]

\[
\}
\]

where \( N \) is the template argument corresponding to \( n \) (swizzling is currently implemented for values \( n = 1, 2, 3, 4, 5 \); unswizzling is not implemented in for a single set of input vectors). Tag dispatching (see section 5.1) is used to redistribute the calls to specific implementations. Swizzling is implemented as an in-place operation, so the array \( v \) is both input and output of the template function.

An example SSE* implementation for \( n = 3 \) and element data type SIMDWord or SIMDShort is

\[
\text{template <typename T>}
\]

\[
\text{static SIMD_INLINE void}
\]

\[
\text{swizzle(const SwizzleTable<3, T, 16> &t,}
\]

\[
\text{SIMDVec\text{<T, 16>} *const v,}
\]

\[
\text{Int\text{<3>},}
\]

\[
\text{IsIntSize\text{<true,2>}{}};
\]

\[
\{
\]

\[
\text{__m128i s0 = align\_shuffle\_word\_128<0>(v[0], v[1], t.mask);}
\]

\[
\text{__m128i s1 = align\_shuffle\_word\_128<6>(v[0], v[1], t.mask);}
\]

\[
\text{__m128i s2 = align\_shuffle\_word\_128<4>(v[1], v[2], t.mask);}
\]

\[
\text{__m128i s3 = align\_shuffle\_word\_128<2>(v[2], v[0], t.mask);}
\]

\[
// s3: v[0] is a dummy
\]

\[
\text{__m128i 101 = _mm\_unpacklo\_epi32(s0, s1);}
\]

\[
\text{__m128i h01 = _mm\_unpackhi\_epi32(s0, s1);}
\]

\[
\text{__m128i 123 = _mm\_unpacklo\_epi32(s2, s3);}
\]

\[
\text{__m128i h23 = _mm\_unpackhi\_epi32(s2, s3);}
\]

\[
\text{v[0] = _mm\_unpacklo\_epi64(101, 123);}
\]

\[
\text{v[1] = _mm\_unpackhi\_epi64(101, 123);}
\]

\[
\text{v[2] = _mm\_unpacklo\_epi64(h01, h23);}
\]

\]

20
where a mask from the SwizzleTable is used in the first processing steps, a combination of an alignment and a shuffle operation. The rest of the function resembles code for a matrix transposition (see section 6). Clever solutions (Melax, 2010) have been suggested for SIMDFloat swizzling with $n = 3$:

```c
1 static SIMD_INLINE void
2 swizzle(const SwizzleTable<3, SIMDFloat, 16> &,
3 SIMDVec<SIMDFloat, 16> *const v,
4 Int<3>,
5 IsIntSize<false,4>)
6 {
7 // x0y0z0x1 = v[0]
8 // y1z1x2y2 = v[1]
9 // z2x3y3z3 = v[2]
10 __m128 x2y2x3y3 = _mm_shuffle_ps(v[1], v[2], _MM_SHUFFLE(2,1,3,2));
11 __m128 y0z0y1z1 = _mm_shuffle_ps(v[0], v[1], _MM_SHUFFLE(1,0,2,1));
12 // x0x1x2x3
13 v[0] = _mm_shuffle_ps(v[0], x2y2x3y3, _MM_SHUFFLE(2,0,3,0));
14 // y0y1y2y3
15 v[1] = _mm_shuffle_ps(y0z0y1z1, x2y2x3y3, _MM_SHUFFLE(3,1,2,0));
16 // z0z1z2z3
17 v[2] = _mm_shuffle_ps(y0z0y1z1, v[2], _MM_SHUFFLE(3,0,3,1));
18 }
```

Note that this solution doesn’t use the SwizzleTable argument.

The following is an example for an image processing function performing cyclic swizzling on all rows of an image (only portions of code shown). It uses a SwizzleTable as local variable (as the effort for constructing a SwizzleTable should always be negligible compared to swizzling the entire image):

```c
1 template <int N, typename T, int SIMD_WIDTH, int SIMD_ALIGN>
2 void
3 swizzleCyclic(const SIMDImage<T,SIMD_WIDTH,SIMD_ALIGN> &inImg,
4 int nAuxVecCols,
5 SIMDImage<T,SIMD_WIDTH,SIMD_ALIGN> &outImg)
6 {
7 SwizzleTable<N, T, SIMD_WIDTH> swizzleTable;
8 ...
9 const int n = N ...;
10 ...
11 SIMDVec<T,SIMD_WIDTH> vecs[n];
12 ...
13 for (...) {
14 ...
15 for (...) {
16 swizzle(swizzleTable, vecs);
17 ...
18 }
19 }
20 ...
21 }
22 }
```

If $n$ is variable, the following function can be used:
template <typename T, int SIMD_WIDTH, int SIMD_ALIGN>
void swizzleCyclic(const SIMDImage<T,SIMD_WIDTH,SIMD_ALIGN> &inImg,
int n, int nAuxVecCols,
SIMDImage<T,SIMD_WIDTH,SIMD_ALIGN> &outImg)
{
switch (n) {
    case 1:
        swizzleCyclic<1>(inImg, nAuxVecCols, outImg);
        break;
    case 2:
        swizzleCyclic<2>(inImg, nAuxVecCols, outImg);
        break;
    case 3:
        swizzleCyclic<3>(inImg, nAuxVecCols, outImg);
        break;
    ...
}
}

4.7.2 Double Set of Input Vectors

If a double set of input vectors is passed, swizzling and unswizzling becomes a very regular operation. The implementation of swizzle2() in T-SIMD generalizes the RGB-deinterleaving technique suggested by Dukhan (2012). It uses zip() which for Intel CPUs comprises two unpack() operations. The inverse unswizzle2() function in T-SIMD is based on unzip() (which is not very efficient on Intel CPUs as it requires multiple intrinsics; see section 4.6.1). A swizzle table is not required in this case. As in swizzle(), both functions operate in-place.

4.8 Implementation of AVX Functions

Difficulties in the implementation of AVX* functions arise from the fact that most AVX* instructions where data crosses the 128-bit lane boundary are only operating separately on the two 128-bit lanes. A general way of thinking about AVX* is that SSE* operations are applied individually two both halves of the 256-bit register — “vertical” operations (like addition of corresponding vector elements) appear to work on the entire 256-bit width, but inter-lane operations behave like two SSE* operations. For the implementation of T-SIMD, two possible ways could have been chosen: Either full 256-bit operations could be emulated, or all operations could be restricted to 128-bit lanes. The emulation way has the advantage that existing T-SIMD code (e.g. obtained from porting code based on SSE* intrinsics) runs without any changes on both SSE* and AVX* platforms. The disadvantage of emulation is that each template function contains additional intrinsics for

\[^{13}\text{See also https://stackoverflow.com/a/15377386/3852630.}\]
\[^{14}\text{This concerns pack, unpack, permute, shuffle, horizontal arithmetic, alignr, and byte-wise shift. The only exception are conversion operations.}\]
data rearrangement. The lane-based way would have the advantage that template func-
tions implemented for AVX* would not contain additional intrinsics. The disadvantage
of the lane-based solution is that each program would have to be written with the lane
concept in mind.

For T-SIMD, the emulation way was considered to be the better solution. Therefore, some
template functions use permute operations to rearrange inputs before the lane-oriented
AVX* intrinsic is applied, e.g.\textsuperscript{15}

```
template <typename T>
static SIMD_INLINE SIMDVec<T, 32> unpack(const SIMDVec<T, 32> &a,
const SIMDVec<T, 32> &b,
  Part<0>,
  Bytes<4>)
{
  return x_mm256_unpacklo_epi32
  (x_mm256_transpose4x64_epi64(a),
   x_mm256_transpose4x64_epi64(b));
}
```

others use permute operations to rearrange outputs after the lane-oriented AVX* intrinsic
was applied, e.g.

```
template <>
SIMD_INLINE SIMDVec<SIMDSignedByte,32> packs(const SIMDVec<SIMDShort,32> &a,
const SIMDVec<SIMDShort,32> &b)
{
  return x_mm256_transpose4x64_epi64
  (x_mm256_packs_epi16(a, b));
}
```

or

```
static SIMD_INLINE SIMDVec<SIMDInt,32> hadd(const SIMDVec<SIMDInt,32> &a,
const SIMDVec<SIMDInt,32> &b)
{
  return x_mm256_transpose4x64_epi64
  (x_mm256_hadd_epi32(a, b));
}
```

\textsuperscript{15}See stackoverflow.com/questions/25622745/transpose-an-8x8-float-using-
avs-avx2
Functions for element-wise shift (srle(), slle()) use workarounds for 256-bit byte-wise shift, e.g.:

```cpp
// IMM = 0
template <int IMM>
static SIMD_INLINE __m256i
x_mm256_srli256_si256(__m256i a,
 Range<true,0,16>)
{
    return a;
}

// IMM = 1..15
template <int IMM>
static SIMD_INLINE __m256i
x_mm256_srli256_si256(__m256i a,
 Range<false,0,16>)
{
    __m256i _0h = x_mm256_permute2x128_si256<_MM_SHUFFLE(2,0, 0,1)>(a, a);
    return x_mm256_alignr_epi8<IMM>(_0h, a);
}
```

Here tag dispatching (see section 5.1) is used to switch to implementations for different values of the immediate parameter. A similar solution is used for the implementation of alignre().

For data swizzling (see section 4.7), the AVX* implementation follows the suggestions by Melax (2010): The elements of the input vectors are rearranged and then processed using lane-oriented intrinsics (in the same way as in the SSE* implementation). For the same example as given in section 4.7, the implementation for n = 3 and element data type SIMDWord is

```cpp
template <typename T>
static SIMD_INLINE void
swizzle(const SwizzleTable<3, T, 32> &t,
    SIMDVec<T, 32> *const v,
    Int<3>,
    IsIntSize<true,2>)
{
    SIMDVec<T, 32> vs[3];
    swizzle_32_16<3>(v, vs);
    __m256i s0 = align_shuffle_word_256<0>(vs[0], vs[1], t.mask);
    __m256i s1 = align_shuffle_word_256<6>(vs[0], vs[1], t.mask);
    __m256i s2 = align_shuffle_word_256<4>(vs[1], vs[2], t.mask);
    __m256i s3 = align_shuffle_word_256<2>(vs[2], vs[0], t.mask);
    // s3: v[0] is a dummy
    __m256i t01 = x_mm256_unpacklo_epi32(s0, s1);
    __m256i t01 = x_mm256_unpackhi_epi32(s0, s1);
    __m256i t123 = x_mm256_unpacklo_epi32(s2, s3);
```

See stackoverflow.com/questions/25248766/emulating-shifts-on-32-bytes-with-avx
The `swizzle_32_16()` template function in line 9 (explained in section 6) rearranges the input elements such that the lane-oriented swizzling produces the right result for the entire 256-bit vector. For the example above, `swizzle_32_16()` accomplishes the following rearrangement:

\[
\begin{align*}
\text{vIn}[0] &= 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 \\
\text{vIn}[1] &= 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 \\
\text{vIn}[2] &= 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 \\

\text{vOut}[0] &= 0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 3 \\
\text{vOut}[1] &= 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 \\
\text{vOut}[2] &= 2 2 2 2 2 2 2 2 5 5 5 5 5 5 5 5
\end{align*}
\]

### 4.9 Implementation Details

T-SIMD offers support for debugging by introducing a sandbox mode and a mode where the alignment of data addresses is checked in aligned load and store template functions. If SIMDEVC_SANDBOX is defined, all SIMD template functions print their names, the template parameters, and some of the function parameters to stdout instead of generating vector code. If SIMD_ALIGN_CHK is defined, aligned load and store template functions assert that data addresses are actually aligned (on 16-byte boundaries for SSE*, on 32-byte boundaries for AVX*).

Fence intrinsics (`lfence()`, `sfence()`, `mfence()`) are not defined as template functions but as normal functions since they do not depend on the vector width and on any data type.

Compiler-specific definitions are bundled in the header file SIMDDefs.H, among them SIMD_INLINE (see section 4.1) and SIMD_ATTR_ALIGNED(ALIGN). The latter is defined for g++ as:

\[
#define SIMD_ATTR_ALIGNED(ALIGN) __attribute__ ((aligned(ALIGN)))
\]

This macro allows to define aligned data structures in a compiler-independent way, e.g.

\[
\text{ImgType rowBuf[w] SIMD_ATTR_ALIGNED(SIMD_WIDTH);}
\]
5 Generic Vector Template Functions

The template functions described in this section are called “generic” since they are not directly mapped onto intrinsics but are second-level template functions which are based based on the first-level vector template functions / overloaded functions described above.\(^{17}\)

5.1 Type Conversion

In addition to the template functions numInputSIMDVecs<Tout,Tin>() and numOutputSIMDVecs<Tout,Tin>() (see section 4.6.2 and 4.6.3) which for template functions with type conversion determine the number of input or output SIMDVec vectors, respectively, the following macro and template functions are provided:

- Macro NUM_SIMDVECS_ELEMENTS(TOUT,TIN,,SIMD_WIDTH) and template function numSIMDVecsElements<Tout,Tin,SIMD_WIDTH>() return the number of elements in all input / all output vectors (this number coincides for input and output), and

- macro NUM_SIMDVEC_ELEMENTS(T,SIMD_WIDTH) and template function numSIMDVecElements<T,SIMD_WIDTH>() return the number of elements in a vector respectively (and can be used to determine the number of elements in each input or output vector).

In addition to the conversion template/overloaded functions packs() (larger to smaller or equal types) and extend() (smaller to larger or equal types), a generic template function convert() is provided

\[
\begin{align*}
\text{template } & \text{<typename Tout, typename Tin, int SIMD_WIDTH> } \\
\text{static } & \text{SIMD_INLINE void } \\
\text{convert(} & \text{const SIMDVec<Tin,SIMD_WIDTH> *const inVecs, } \\
& \text{SIMDVec<Tout,SIMD_WIDTH> *const outVecs);} \\
\end{align*}
\]

which converts in both directions. This function uses either packs() or extend(), depending on the output and input types (Tout and Tin). The implementation requires a C++ trick called “tag dispatching”.\(^{18}\) In tag dispatching, a “dispatcher” template function dispatches a task to several overloaded “worker” template functions. This is accomplished by providing an additional argument to each worker function which is not used inside the function (and therefore hopefully optimized away by the compiler). However, the type of

\(^{17}\)Note that the level on which a function is defined may change in later code revisions.

this function argument is used to distinguish between the different overloaded functions. The dispatcher template function determines the type according to some properties of types provided as its template arguments. Tag dispatching is required in this case since the different cases can not be handled by putting the code in different branches of the same function: Even if unused branches would be optimized away, the compiler checks their syntax, and there simply is no packs() which increases the type size and no extend() which decreases the type size.

The dispatcher template function uses the int2type trick described by Alexandrescu (2001); a somewhat simpler example is presented in section 4.4. It is based on the fact that each combination of integer template arguments declares a unique type, in our case the type of an empty struct

```cpp
1 template <bool less, bool equal, bool greater>
2 struct Compare {}
```

Type definitions are used to defines the “tags” for the worker template functions:

```cpp
1 typedef Compare<true,false,false> CompareLess;
2 typedef Compare<false,true,false> CompareEqual;
3 typedef Compare<false,false,true> CompareGreater;
```

For the special case of type size comparisons, the following template class was derived

```cpp
1 template <typename Tout, typename Tin>
2 struct CompareTypes :
3     Compare<(sizeof(Tout) < sizeof(Tin)),
4             (sizeof(Tout) == sizeof(Tin)),
5             (sizeof(Tout) > sizeof(Tin))> {};
```

which is used to generate the tag in the dispatcher template function:

```cpp
1 template <typename Tout, typename Tin, int SIMD_WIDTH>
2 static SIMD_INLINE void
3     convert(const SIMDVec<Tin,SIMD_WIDTH> *const inVecs,
4              SIMDVec<Tout,SIMD_WIDTH> *const outVecs)
5     {
6         convert(CompareTypes<Tout,Tin>(), inVecs, outVecs);
7     }
```

The “packing” worker template function is tagged by CompareLess
template <typename Tout, typename Tin, int SIMD_WIDTH>
static SIMD_INLINE void
convert(CompareLess,
    const SIMDVec<Tin,SIMD_WIDTH> *const inVecs,
    SIMDVec<Tout,SIMD_WIDTH> *const outVecs)
{
    *outVecs = packs<Tout>(inVecs);
}

whereas the “extending” worker template function is tagged by CompareGreater

template <typename Tout, typename Tin, int SIMD_WIDTH>
static SIMD_INLINE void
convert(CompareGreater,
    const SIMDVec<Tin,SIMD_WIDTH> *const inVecs,
    SIMDVec<Tout,SIMD_WIDTH> *const outVecs)
{
    extend(*inVecs, outVecs);
}

The CompareEqual case is also handled by extend().

5.2 Float Operations on Arbitrary Input and Output Types

Multiplication of integer types is awkward with SSE* and AVX* intrinsics (especially with respect to the question of how many bits of the result are kept), and integer division is not provided at all. We therefore defined a group of template functions which take input arguments of an arbitrary type, convert the arguments to SIMDFloat, perform some operation (like division and subsequent multiplication) using floating-point arithmetic, and convert back to some arbitrary output type: fdivmul() (divide, then multiply), fmul() (multiply), faddmul() (add then multiply), fmuladd() (multiply then add), fwaddmul() (weighed addition then multiplication). Note that, as in convert, these functions distinguish between different cases depending on the size of input and output types. Tag dispatching (rather than branching) is necessary in these functions as well since otherwise zero-length arrays would occur in unused branches (violating the standard19). The following code gives an example:

template <typename Tout, typename Tin, int SIMD_WIDTH>
static SIMD_INLINE void
fmul(CompareLess,
    const SIMDVec<Tin, SIMD_WIDTH> *const vecsIn,
    double fac,
    SIMDVec<Tout, SIMD_WIDTH> *const vecsOut)

19 Except of this problem, the syntax would be correct even in the inactive branches as conversion is always upward or equal to SIMDFloat and downward or equal from SIMDFloat.
we assume that sizeof(Tout), sizeof(Tin) <= sizeof(SIMDFloat)

```cpp
typeSizeLEQ<Tin, SIMDFloat>();
typeSizeLEQ<Tout, SIMDFloat>();
SIMDVec<SIMDFloat, SIMD_WIDTH> facF = set1<SIMDFloat, SIMD_WIDTH>({fac});
const int nIn = sizeof(Tin) / sizeof(Tout);
const int fanIn = sizeof(SIMDFloat) / sizeof(Tin);
SIMDVec<SIMDFloat, SIMD_WIDTH> inF[fanIn];
SIMDVec<SIMDFloat, SIMD_WIDTH> resF[nIn * fanIn];
for (int i = 0, k = 0; i < nIn; i++) {
    extend(vecsIn[i], inF);
    for (int j = 0; j < fanIn; j++, k++)
        resF[k] = mul(inF[j], facF);
}
*vecsOut = packs<Tout>({resF});
```
5.3 Multi-Vector Load and Store

Versions of load(), loadu(), store(), storeu() are provided which load and store multiple input vectors, e.g.

```cpp
template <typename T, int SIMD_WIDTH>
static SIMD_INLINE void
loadu(const T *const p,
      SIMDVec<T, SIMD_WIDTH> *inVecs, int numInVecs)
{
  const int vecElemsIn = SIMD_WIDTH / sizeof(T);
  for (int vi = 0, off = 0; vi < numInVecs; vi++, off += vecElemsIn)
    inVecs[vi] = loadu<SIMD_WIDTH>(p + off);
}
```

5.4 Memory Copy

For transfer of a single vector from one memory location to another, the template function load_store() is provided:

```cpp
template <int SIMD_WIDTH, typename T>
static SIMD_INLINE void
load_store(const T *const src, T *const dst)
{
  SIMDVec<T, SIMD_WIDTH> copy = load<SIMD_WIDTH>(src);
  store(dst, copy);
}
```

This function loads and stores / from to aligned addresses. Three more functions are provided for unaligned load and / or store: loadu_store(), load_storeu(), loadu_storeu().

5.5 Various Functions

The template function setones() sets all bits to 1. The template functions setmin(), setmax(), and setunity() return the minimal, maximal, or unity value of the element data type. The function setnegunity() returns the value $-1$ in signed data types. For integer element types, this is done efficiently by some bit magic; for
SIMDFloat, the `set1()` function is used. The template function `avgru()` is a synonym for `avg()` (indicating that it rounds up for integer types). The template function `avgrd()` computes the average but rounds down for integer types.

Binary arithmetic Operators `+ - * / += -= *- /=` and the unary `-` are defined, with addition and subtraction mapped to the saturated versions. Binary bitwise operators `& | ^ &= |= ^=` and the unary `~` are defined, but no logical operators (e.g. `&&`). Comparison operators `> >= == != <= <` are defined as well. These operators can only be used if the underlying functions are defined; currently, multiplication and division only work for SIMDFloat, and negation (`~`) only for signed types.

### 6 Template Meta-Programming

C++ templates define a functional language and thus allows to execute programs at compile time (Vandervoorde and Josuttis, 2003; Veldhuizen, accessed 2016). Recursive instantiation of templates can be used to implement the language constructs recursion and loops, and recursion termination as well as branching can be accomplished by template specialization. Note that since partial specialization of template functions is not possible, template classes need to be used.

In T-SIMD, template meta-programming is exploited for two purposes: First, generic second-level template functions (see section 5) can be written which generate entire code sequences that otherwise would have to be explicitly written down for each vector width and data type (sections 6.1, 6.2). Second, all SSE* and AVX* vector instructions with scalar integer parameters (for element selection, shift range etc.) encode the corresponding arguments as immediate: The integer (typically only few bits wide) becomes part of the instruction. Therefore, only constants can be used as parameters. Template meta-programming allows to repeatedly use an intrinsic with different constant parameters, similar to loop unrolling (section 6.3).

### 6.1 Horizontal Addition

One example of template meta-programming is the implementation of a generic horizontal add. If an SSE* algorithm computed the vector sum over 4 rows of a float matrix using vertical instructions (lines 3-9), the elements can subsequently be added in each of the resulting 4 vectors, yielding a vector of 4 scalar sums (lines 10-12):

```cpp
float matrix[4][LEN] __attribute__((aligned(16)));
__m128 rowsum[4];
for (i = 0; i < 4; i++) {
    rowsum[i] = _mm_setzero_ps();
    for (j = 0; j < LEN; j += 4) {
```
For \( n \) elements in each vector, \( n - 1 \) recursive invocations of the horizontal add intrinsic are required. This code can be automatically written by using template metaprogramming: A primary template class declaration implements the recursive invocation of the first-level \texttt{hadd()} template function

\begin{verbatim}
// num: number of elements processed
// i0, i1: indices of lowest elements of block
template <typename T, int SIMD_WIDTH, int num, int i0, int i1>
class Horizontal
{
    static SIMD_INLINE SIMDVec<T, SIMD_WIDTH> _hadd(const SIMDVec<T, SIMD_WIDTH> *const v)
    {
        return hadd(Horizontal<T, SIMD_WIDTH, num/2, i0, i0 + num/4>::_hadd(v),
                    Horizontal<T, SIMD_WIDTH, num/2, i1, i1 + num/4>::_hadd(v));
    }
};

and a partial specialization terminates the recursion (num==2)

template <typename T, int SIMD_WIDTH, int i0, int i1>
class Horizontal<T, SIMD_WIDTH, 2, i0, i1>
{
    static SIMD_INLINE SIMDVec<T, SIMD_WIDTH> _hadd(const SIMDVec<T, SIMD_WIDTH> *const v)
    {
        return hadd(v[i0], v[i1]);
    }
};

The template function \texttt{hadd()} is defined as

\begin{verbatim}
template <typename T, int SIMD_WIDTH>
static SIMD_INLINE SIMDVec<T, SIMD_WIDTH> hadd(const SIMDVec<T, SIMD_WIDTH> *const v)
{
    return Horizontal<T, SIMD_WIDTH, SIMD_WIDTH / sizeof(T), 0,
                     (SIMD_WIDTH / sizeof(T)) / 2>::_hadd(v);
}
\end{verbatim}
Horizontal subtraction (hsub()) and the saturated versions of horizontal addition (hadds()) and subtraction (hsubs()) of multiple vectors are defined in the same way.

A second form of hadd() receives a single vector argument and returns a the scalar horizontal sum over all vector elements:

```cpp
1 template <typename T, int SIMD_WIDTH>
2 static SIMD_INLINE T
3     hadd(const SIMDVec<T,SIMD_WIDTH> &v)
```

Note that even though horizontal addition in this form is implemented with vector instructions, it is not fully parallel. Again, hsub(), hadds(), and hsubs() are defined in the same way. In addition, horizontal minimum (hmin()) and maximum (hmax()) are available for this form; an example application is shown in section 8.4.

### 6.2 Register Transpose

Matrix transpose operations are another example where template meta-programming proves to be useful to write generic code. Register transpose can be seen as the elementary matrix transpose operation from which transposition of matrices of arbitrary size can be constructed. If a register holds \( n \) elements, a register transpose operates on \( n \) registers that store the \( n \) rows of a sub-matrix (here \( n = 8 \))

\[
\begin{align*}
\text{inRows}[0] &= 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7  \\
\text{inRows}[1] &= 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15  \\
\text{inRows}[2] &= 16 \ 17 \ 18 \ 19 \ 20 \ 21 \ 22 \ 23  \\
\text{inRows}[3] &= 24 \ 25 \ 26 \ 27 \ 28 \ 29 \ 30 \ 31  \\
\text{inRows}[4] &= 32 \ 33 \ 34 \ 35 \ 36 \ 37 \ 38 \ 39  \\
\text{inRows}[5] &= 40 \ 41 \ 42 \ 43 \ 44 \ 45 \ 46 \ 47  \\
\text{inRows}[6] &= 48 \ 49 \ 50 \ 51 \ 52 \ 53 \ 54 \ 55  \\
\text{inRows}[7] &= 56 \ 57 \ 58 \ 59 \ 60 \ 61 \ 62 \ 63
\end{align*}
\]

and produces the transposed sub-matrix in \( n \) output registers:

\[
\begin{align*}
\text{outRows}[0] &= 0 \ 8 \ 16 \ 24 \ 32 \ 40 \ 48 \ 56  \\
\text{outRows}[1] &= 1 \ 9 \ 17 \ 25 \ 33 \ 41 \ 49 \ 57  \\
\text{outRows}[2] &= 2 \ 10 \ 18 \ 26 \ 34 \ 42 \ 50 \ 58  \\
\text{outRows}[3] &= 3 \ 11 \ 19 \ 27 \ 35 \ 43 \ 51 \ 59  \\
\text{outRows}[4] &= 4 \ 12 \ 20 \ 28 \ 36 \ 44 \ 52 \ 60  \\
\text{outRows}[5] &= 5 \ 13 \ 21 \ 29 \ 37 \ 45 \ 53 \ 61  \\
\text{outRows}[6] &= 6 \ 14 \ 22 \ 30 \ 38 \ 46 \ 54 \ 62  \\
\text{outRows}[7] &= 7 \ 15 \ 23 \ 31 \ 39 \ 47 \ 55 \ 63
\end{align*}
\]
Matrix transpose uses unpack intrinsics. Here is the SSE* code for the case of $n = 8$ words:

```c
  __m128i a03b03 = _mm_unpacklo_epi16(a, b);
  __m128i c03d03 = _mm_unpacklo_epi16(c, d);
  __m128i e03f03 = _mm_unpacklo_epi16(e, f);
  __m128i g03h03 = _mm_unpacklo_epi16(g, h);
  __m128i a47b47 = _mm_unpackhi_epi16(a, b);
  __m128i c47d47 = _mm_unpackhi_epi16(c, d);
  __m128i e47f47 = _mm_unpackhi_epi16(e, f);
  __m128i g47h47 = _mm_unpackhi_epi16(g, h);
  __m128i a01b01c01d01 = _mm_unpacklo_epi32(a03b03, c03d03);
  __m128i a23b23c23d23 = _mm_unpackhi_epi32(a03b03, c03d03);
  __m128i e01f01g01h01 = _mm_unpacklo_epi32(e03f03, g03h03);
  __m128i e23f23g23h23 = _mm_unpackhi_epi32(e03f03, g03h03);
  __m128i a45b45c45d45 = _mm_unpacklo_epi32(a47b45, c47d45);
  __m128i a67b67c67d67 = _mm_unpackhi_epi32(a47b45, c47d45);
  __m128i e45f45g45h45 = _mm_unpacklo_epi32(e47f45, g47h45);
  __m128i e67f67g67h67 = _mm_unpackhi_epi32(e47f45, g47h45);

  __m128i a0b0c0d0e0f0g0h0 = _mm_unpacklo_epi64(a01b01c01d01, e01f01g01h01);
  __m128i a1b1c1d1e1f1g1h1 = _mm_unpackhi_epi64(a01b01c01d01, e01f01g01h01);
  __m128i a2b2c2d2e2f2g2h2 = _mm_unpacklo_epi64(a23b23c23d23, e23f23g23h23);
  __m128i a3b3c3d3e3f3g3h3 = _mm_unpackhi_epi64(a23b23c23d23, e23f23g23h23);
  __m128i a4b4c4d4e4f4g4h4 = _mm_unpacklo_epi64(a45b45c45d45, e45f45g45h45);
  __m128i a5b5c5d5e5f5g5h5 = _mm_unpackhi_epi64(a45b45c45d45, e45f45g45h45);
  __m128i a6b6c6d6e6f6g6h6 = _mm_unpacklo_epi64(a67b67c67d67, e67f67g67h67);
  __m128i a7b7c7d7e7f7g7h7 = _mm_unpackhi_epi64(a67b67c67d67, e67f67g67h67);
```

Vector registers $a$ to $h$ hold the input vectors, registers $a0b0c0d0e0f0g0h0$ to $a7b7c7d7e7f7g7h7$ hold the transposed sub-matrix.

Since recursion is the basic language construct of template meta-programming, the T-SIMD implementation starts from a recursive version of this code. By inserting the expressions for the temporary registers (line 1-8, 10-17) in the computations of line 19-26, we obtain the following pseudo code:

```c
  r0 = L64(L32(L16(0,1),L16(2,3)), L32(L16(4,5),L16(6,7)));  
  r1 = H64(L32(L16(0,1),L16(2,3)), L32(L16(4,5),L16(6,7)));  
  r2 = L64(H32(L16(0,1),L16(2,3)), H32(L16(4,5),L16(6,7)));  
  r3 = H64(H32(L16(0,1),L16(2,3)), H32(L16(4,5),L16(6,7)));  
  r4 = L64(L32(H16(0,1),H16(2,3)), L32(H16(4,5),H16(6,7)));  
  r5 = H64(L32(H16(0,1),H16(2,3)), L32(H16(4,5),H16(6,7)));  
  r6 = L64(H32(H16(0,1),H16(2,3)), H32(H16(4,5),H16(6,7)));  
  r7 = H64(H32(H16(0,1),H16(2,3)), H32(H16(4,5),H16(6,7)));  
```

In this pseudo code, input registers only appear with their number (from 0 for $a$ to 7 for $h$). Unpack operations are encoded by L for unpacklo and H for unpackhi, followed by the bit number $(16, 32, 64)$. The registers r0 to r7 are the output registers.

---

20See stackoverflow.com/questions/2517584/transpose-for-8-registers-of-16-bit-elements-on-sse2-sse3
This code is of course inefficient since the same expressions occur multiple times, but it turns out that the common sub-expression elimination of the compiler removes all duplicate expressions, generating only the $n \log_2 n$ unpack operations of the original code ($g++$ 4.6.3, 4.8.4). We can therefore derive a recursive implementation. Its basic idea is revealed when only the specifications for the unpack operations used in each code line are listed:

```
1 r0: L16 L32 L64
2 r1: L16 L32 H64
3 r2: L16 H32 L64
4 r3: L16 H32 H64
5 r4: H16 L32 L64
6 r5: H16 L32 H64
7 r6: H16 H32 L64
8 r7: H16 H32 H64
```

Obviously the $L$ and $H$ specifications form the binary code of the register index, thus the meta code performs a conversion of this index into binary code.

In the implementation, the `unpack()` template function is wrapped into a template class such that it can be exchanged (e.g. if lane-oriented unpacking is required instead):

```
1 template <int PART, int NUM_ELEMS, typename T, int SIMD_WIDTH>
2 class Unpack
3 {
4   public:
5     static SIMD_INLINE SIMDVec<T,SIMD_WIDTH> _unpack(const SIMDVec<T,SIMD_WIDTH> &a,
6               const SIMDVec<T,SIMD_WIDTH> &b)
7     {
8         return unpack<PART,NUM_ELEMS>(a, b);
9     }
10  });
```

The recursive code for the computation of a single output row is found in the primary template definition

```
1 template <template <int, int, typename, int> class Unpack,
2           typename T, int SIMD_WIDTH,
3           // INDEX: index of first input element to unpack
4           // NLOHI: low/high unpack selector index
5           // ELEMS: number of elements to unpack in this stage
6           int INDEX, int NLOHI, int ELEMS>
7   class Transpose
8 {
9   enum { PART = (NLOHI & 0x01), NEXT = (NLOHI >> 1) };
10  enum { LIDX = INDEX, RIDX = INDEX + ELEMS };
11  enum { HALF = ELEMS / 2 };
12 };
```
public:

static SIMD_INLINE SIMDVec<T, SIMD_WIDTH> _transpose1(const SIMDVec<T, SIMD_WIDTH> *const inRows)
{
    return Unpack<PART, ELEMS, T, SIMD_WIDTH>::_unpack
    (Transpose1<Unpack, T, SIMD_WIDTH, LIDX, NEXT, HALF>::_transpose1(inRows),
     Transpose1<Unpack, T, SIMD_WIDTH, RIDX, NEXT, HALF>::_transpose1(inRows));
}

Line 9 contains the computation of the binary code. Line 10 determines the indices of the input registers. Line 11 determines how many elements are transported by unpack().

The recursion is terminated by the partial template specialization for ELEMS == 1.

template <template <int, int, typename, int> class Unpack,
    typename T, int SIMD_WIDTH, int INDEX, int NLOHI>
class Transpose1<Unpack, T, SIMD_WIDTH, INDEX, NLOHI, 1>
{

    enum { PART = (NLOHI & 0x01) };

    public:

    static SIMD_INLINE SIMDVec<T, SIMD_WIDTH> _transpose1(const SIMDVec<T, SIMD_WIDTH> *const inRows)
    {
        return Unpack<PART, 1, T, SIMD_WIDTH>::_unpack(inRows[INDEX],
            inRows[INDEX+1]);
    }

};

To compute all transposed rows, recursion is used to implement a loop from output register index ROW to ROW + NUM_TRANSPOSE_ROWS-1 (to also allow for partial transposition, possibly used by a generic swizzle operation), again comprising a primary template

template <template <int, int, typename, int> class Unpack,
    typename T, int SIMD_WIDTH,
    // NUMROWS: total number of rows
    // NUM_TRANSPOSE_ROWS: number of rows to transpose
    // ROW: index of row to transpose
    int NUMROWS, int NUM_TRANSPOSE_ROWS, int ROW>
class Transpose
{

    public:

    static SIMD_INLINE void _transpose(const SIMDVec<T, SIMD_WIDTH> *const inRows,
        SIMDVec<T, SIMD_WIDTH> *const outRows)
    {
        outRows[ROW] =
            // INDEX=0, NLOWHI=ROW, ELEMS=NUMROWS/2
            Transpose1<Unpack, T, SIMD_WIDTH,};

36
0, ROW, NUMROWS/2>::_transpose1(inRows);

// transpose next row
// NUMROWS=NUMROWS, ROW=ROW+1
TransposedUnpack<T, SIMD_WIDTH,
NUMROWS, NUM_TRANSPOSE_ROWS,
ROW+1>::_transpose(inRows, outRows);
}
};

and a partial specialization (which in this case is empty) to terminate the recursion if ROW == NUM_TRANSPOSE_ROWS:

template <template <int, int, typename, int> class Unpack,
typename T, int SIMD_WIDTH, int NUMROWS,
int NUM_TRANSPOSE_ROWS>
class Transpose<Unpack, T, SIMD_WIDTH,
NUMROWS, NUM_TRANSPOSE_ROWS, NUM_TRANSPOSE_ROWS>
{
public:
  static SIMD_INLINE void
  _transpose(const SIMDVec<T, SIMD_WIDTH> *const inRows,
             SIMDVec<T, SIMD_WIDTH> *const outRows)
  {
  }
};

The transpose template functions for partial and full transpose are defined as

// function template: partial transpose
template <int NUM_TRANSPOSE_ROWS, typename T, int SIMD_WIDTH>
static SIMD_INLINE void
transpose(const SIMDVec<T, SIMD_WIDTH> *const inRows,
          SIMDVec<T, SIMD_WIDTH> *const outRows)
{
  TransposedUnpack<T, SIMD_WIDTH,
  // NUMROWS, NUM_TRANSPOSE_ROWS, ROW
  SIMD_WIDTH / sizeof(T),
  NUM_TRANSPOSE_ROWS, 0>::_transpose(inRows, outRows);
}

// function template: full transpose
template <typename T, int SIMD_WIDTH>
static SIMD_INLINE void
transpose(const SIMDVec<T, SIMD_WIDTH> *const inRows,
          SIMDVec<T, SIMD_WIDTH> *const outRows)
{
  transpose<SIMD_WIDTH/sizeof(T)>(inRows, outRows);
}
Note that an alternative solution `transpose2()` was added later which implements transposition as a special case of `swizzle2()` (section 4.7.2). This solution is based on loops and not on template meta-programming.

### 6.3 Inter-Lane Swizzling

Inter-lane swizzling is required to prepare the input vectors in the implementation of AVX* swizzle operations (see section 4.7). In this application of template meta-programming, the main purpose is to generate immediate operands for a permutation intrinsic. If `l / h` denotes the lower / higher 128-bit lane, respectively, and the numbers specify the indices of each of `n` vectors, examples for the inter-lane swizzling are

- **n=3:** `l0 h0 l1 h1 l2 h2` → `l0 h1 h0 l2 l1 h2`
- **n=4:** `l0 h0 l1 h1 l2 h2 l3 h3` → `l0 l2 h0 h2 l1 l3 h1 h3`

For the permutation itself, the intrinsic `_mm256_permute2x128_si256()` is wrapped into overloaded functions

```cpp
// all integer versions
template <int N, typename T>
static SIMD_INLINE SIMDVec<T, 32>
x_mm256_perm16(const SIMDVec<T, 32> &a,
               const SIMDVec<T, 32> &b)
{
    return _mm256_permute2x128_si256(a, b, N);
}

// float version
template <int N>
static SIMD_INLINE SIMDVec<SIMDFloat, 32>
x_mm256_perm16(const SIMDVec<SIMDFloat, 32> &a,
               const SIMDVec<SIMDFloat, 32> &b)
{
    return _mm256_permute2f128_ps(a, b, N);
}
```

In the template function above, `N` is an immediate parameter for which different constant values are produced in the recursive loop:

```cpp
// primary template
template <int N, typename T, int I>
class Swizzle_32_16
{
    public:
```
static SIMD_INLINE void
chwizzle_32_16(const SIMDVec<T,32> *const vIn,
SIMDVec<T,32> *const vOut)
{
  // example: n=3
  // I=0: x_mm256_perm16(vIn[0], vIn[1], _MM_SHUFFLE(0, 3, 0, 0));
  // I=1: x_mm256_perm16(vIn[0], vIn[2], _MM_SHUFFLE(0, 2, 0, 1));
  // I=2: x_mm256_perm16(vIn[1], vIn[2], _MM_SHUFFLE(0, 3, 0, 0));
  vOut[I] =
      x_mm256_perm16<_MM_SHUFFLE(0, (2+(I+N)%2), 0, (I%2))>
          (vIn[I/2], vIn[(I+N)/2]);
  Swizzle_32_16<N,T,I+1>::chwizzle_32_16(vIn, vOut);
}
};

// termination
template <int N, typename T>
class Swizzle_32_16<N, T, N>
{
  public:
    static SIMD_INLINE void
chwizzle_32_16(const SIMDVec<T,32> *const,
SIMDVec<T,32> *const)
    {
    }
};

The template function starts the recursive loop at I == 0:

template <int N, typename T>
static SIMD_INLINE void
chwizzle_32_16(const SIMDVec<T,32> *const vIn,
SIMDVec<T,32> *const vOut)
{
  Swizzle_32_16<N,T,0>::chwizzle_32_16(vIn, vOut);
}

7 Multi-Vector Template Functions

Multi-vector template functions use plain C arrays of SIMDVec vectors as parameters (see section 4.6.2, 4.6.3, 5.1, 5.2, 5.3, 6.1, 6.2, 6.3). It will have disastrous consequences if the length of the arrays passed to a template function is smaller than the length expected by the function, or if the vector index is out of range. While it is probably not advisable to use STL’s vector instead (as this might produce overhead), a special template class containing multiple vectors (with the vector number as template parameter) can be used to make access to the individual vector elements safer – functions would accept only arrays
of certain lengths as arguments, or the operations inside the function would depend on the length of the array:

```cpp
1 template <int NUM, typename T, int SIMD_WIDTH>
2 class SIMDVecs
3 {
4   public:
5     enum {
6       vectors = NUM,
7       elements = NUM * SIMDVec<T,SIMD_WIDTH>::elements,
8       bytes = NUM * SIMDVec<T,SIMD_WIDTH>::bytes
9     };
10    SIMDVec<T,SIMD_WIDTH> vec[NUM];
11  }
```

For template functions where the element type of input and output vectors differs, the following template class can be used to compute the number of input and output vectors, also in template parameters (using a function such as `numInputSIMDVecs()` instead would require `constexpr` in the functions’s definition which is only available in C++11):

```cpp
1 template <typename Tout, typename Tin>
2 class NumSIMDVecs
3 {
4   public:
5     enum {
6       in = (sizeof(Tout) < sizeof(Tin)) ?
7           (sizeof(Tin) / sizeof(Tout)) : 1,
8       out = (sizeof(Tout) > sizeof(Tin)) ?
9           (sizeof(Tout) / sizeof(Tin)) : 1
10     };
11 }
```

All template functions operating on plain C arrays (via pointers to `SIMDVec`) are wrapped in template functions of the same name operating on `SIMDVecs`. In the first example, the number of vectors in the `SIMDVecs` parameters is computed using `NumSIMDVecs`:

```cpp
1 template <typename Tout, typename Tin, int SIMD_WIDTH>
2 static SIMD_INLINE void
3 convert(const SIMDVecs<NumSIMDVecs<Tout,Tin>::out,
4    Tout,SIMD_WIDTH> &inVecs,
5    SIMDVecs<NumSIMDVecs<Tout,Tin>::in,
6    Tin,SIMD_WIDTH> &outVecs)
7 {
8    convert(inVecs.vec, outVecs.vec);
9 }
```
In the second example, the number of vectors in the SIMDVecs parameter is computed from the number of elements in each SIMDVec:

```cpp
1 template <typename T, int SIMD_WIDTH>
2 static SIMD_INLINE void
3 transpose(const SIMDVecs<(SIMD_WIDTH/sizeof(T)),T,SIMD_WIDTH> &inRows,
4        SIMDVecs<(SIMD_WIDTH/sizeof(T)),T,SIMD_WIDTH> &outRows)
5 {
6        transpose(inRows.vec, outRows.vec);
7    }
```

In the third example, the number of vectors in the SIMDVecs argument determines the number of SIMDVec elements that are loaded:

```cpp
1 template <int NUM, typename T, int SIMD_WIDTH>
2 static SIMD_INLINE void
3 load(const T *const p, SIMDVecs<NUM,T,SIMD_WIDTH> &inVecs)
4 {
5        load(p, inVecs.vec, inVecs.vectors);
6    }
```

The following functions using SIMDVecs are currently implemented: convert(), fdivmul(), fmul(), faddmul(), fmuladd(), fwaddmul(), load(), loadu(), store(), storeu(), packs(), extend(), swizzle(), transpose(), transposePartial(), hadd(), hadds(), hsub(), hsubs(), add(), adds(), sub(), subs(), min(), max(), setzero(), set1().

However, it is currently not clear whether the use of the wrappers using the SIMDVecs template class entails any overhead.

### 8 Application Examples

In the following, example code is provided which uses T-SIMD. The code portions were taken from the implementation of MinWarping (Möller, 2016). Some parts of the code have been removed as they are not relevant to demonstrate the application of T-SIMD.

#### 8.1 Horizontal Binomial Filter

The first example is a horizontal binomial filter (cyclic in horizontal direction) based on the alignre() and avg() template functions. It operates on a template class SIMDImage with the template parameters element data type (ImgType), vector width (SIMD_WIDTH), and alignment of the data (SIMD_ALIGN).
Vector objects of type SIMDVec are defined in line 13. The generic load_store() template function is used in line 17 to copy a vector. The first-level load() is used in lines 19 and 22 to load data. The core of the processing is done in lines 23-25. Note the computation of the number of elements (elems) in line 9. This value is used to advance index and pointer in the loop in line 20.

8.2 Vertical Edge Filter

A vertical edge filter is applied to an image. SIMDVec objects are defined in line 8. Data is loaded from subsequent rows in line 17 and 20, and stored in line 22. The edge filter is a subtraction (line 21). The constant simd_elems (line 9) is used to advance index (line 14) and pointers (line 27 and 28).
The following function is used to compute a “visual compass” in MinWarping. This function operates on a stack of images (“scale-plane stack”). Over selected image rows, the minimum is computed over all images of the stack (line 34-38). Then the minima are added over all selected rows (line 40-41).

```cpp
8.3 Visual Compass

The following function is used to compute a “visual compass” in MinWarping. This function operates on a stack of images (“scale-plane stack”). Over selected image rows, the minimum is computed over all images of the stack (line 34-38). Then the minima are added over all selected rows (line 40-41).

```
This example uses `convert()` to convert from the type of the images (SPSType) to the type of the result (CompassType). This is necessary since the summation is typically done with a larger data type than the summed terms. Template functions are used to compute the number of input and output vectors and the number of vector elements (lines 17-19). Before `convert()` can be applied in line 39, an array of input vectors (`minv`, line 21) is computed (lines 32-38, in this case a minimum operation). After `convert()`, the resulting array of output vectors (`minvC`, line 22) is processed further (lines 40-41, in this case a summation). This code would also allow to use a smaller data type for summation than the type of the summed terms. Note also how the data types and the vector width are handed down from the application code to the template class WarpingSPS and from there to the template member function `compassEstimate`.

### 8.4 Minimum of all Pixels of an Image

This example code computes the minimum of all pixels (with type `T`) of an image. It uses a vertical minimum operation (line 7-8, also note the initialization with the maximal value of the type in line 5) and computes the scalar minimum of the elements of the resulting vector `minv` using a horizontal minimum function (line 9):

```cpp
template <typename T, int SIMD_WIDTH, int SIMD_ALIGN>
T min(const SIMDImage<T, SIMD_WIDTH, SIMD_ALIGN> &img)
{
    SIMDVec<T,SIMD_WIDTH> minv
    = set1<T,SIMD_WIDTH>(SIMDTypeInfo<T>::max());
    const int nElems = SIMD_WIDTH / sizeof(T);
    for (T *d = img.data; d < img.data + img.size; d += nElems)
        minv = min(minv, load<SIMD_WIDTH>(d));
    return hmin(minv);
}
```

### 8.5 Average of Two Arrays Considering Invalid Entries

In the following method, entries in two arrays are averaged. If at least one of the entries is invalid, the result is invalid as well. This provides an example of the use of `ifelse()`
and \texttt{cmp*()}. 

\begin{verbatim}
template <typename MatchType, int SIMD_WIDTH, int SIMD_ALIGN>
class MinWarpingMatch 
{
public:
    const MatchType invalid;
    SIMDImage<MatchType,SIMD_WIDTH,SIMD_ALIGN> match;

    void averageOf(const MinWarpingMatch &match1,
                   const MinWarpingMatch &match2)
    {
        SIMDVec<MatchType,SIMD_WIDTH> vec1, vec2,
        result, resultInvalid,
        invalidVec = set1<MatchType,SIMD_WIDTH>(invalid);
        for (size_t i = 0;
             i < match.size;
             i += SIMDVec<MatchType,SIMD_WIDTH>::elements) 
        {
            vec1 = load<SIMD_WIDTH>(match1.match.data + i);
            vec2 = load<SIMD_WIDTH>(match2.match.data + i);
            resultInvalid = or(cmpeq(vec1, invalidVec),
                                 cmpeq(vec2, invalidVec));
            result = ifelse(resultInvalid,
                             invalidVec, avgrd(vec1, vec2));
            store(match.data + i, result);
        }
    }
};
\end{verbatim}

9 Open Problems

9.1 Not Yet Implemented

The following template functions (corresponding to intrinsics with the same core name) have not yet been implemented (according to the list of SSE* intrinsics): \texttt{addsub()}, \texttt{string comparison, cmpord(), cmpunord()}, \texttt{single element intrinsics, CRC intrinsics, insert()}, \texttt{instructions operating on the machine status word, lddqu(), loadr(), mask move instructions, madd(), maddubs(), minpos(), mpsaddbw(), integer multiplication, sad(), set(), setr(), shuffle*(), sign(), sll(), srl(), sra(), stream_load()}. 

9.2 Masked Operations

The next Intel vector extension, AVX-512, introduces masked operations where additional mask registers determine which elements of vectors are accessed. As masked instructions
are not available in the SSE* and AVX* instruction set, all masked operations would have to be emulated in SSE* and AVX* if masked operations are introduced as template functions in T-SIMD. However, this is not trivial as e.g. memory access is also masked — an emulation reading or writing masked elements may violate memory segmentation. This points to a more general problem: If later extensions introduce instructions not present in earlier extensions, they have to be emulated (if possible) in the template functions implementing the earlier extensions, which probably leads to inefficient code.

9.3 Pointer Arguments and Results

Using intrinsics, it is possible to access memory data directly as argument or result of an intrinsic by using pointers to vector data types, as it is done for the pointers \( b \) and \( c \) in this example:

```c
float *a, *b, *c;
__m128 a4 = _mm_load_ps(a);
*(__m128*)c = _mm_add_ps(a4, *(__m128*)b);
```

The compiler automatically generates the necessary load and store intrinsics. Whether similar constructs are possible with SIMDVec arguments or results is currently not clear. It is recommended to explicitly call load() and store().

References


Changes

June 15, 2017: Added setnegunity().

December 2, 2017: Added zip() and unzip().

December 29, 2017: Added swizzle2() and unswizzle2() in section 4.7.2. Added transpose2() in section 6.2.