Boost.SIMD: Generic Programming for portable SIMDization

Pierre Estérie
LRI, Université Paris-Sud XI
Orsay, France
pierre.esterie@lri.fr

Mathias Gaunard
Metascale
Orsay, France
mathias.gaunard@metascale.org

Joel Falcou
LRI, Université Paris-Sud XI
Orsay, France
joel.falcou@lri.fr

Jean-Thierry Lapresté
LASMEA, Université Blaise Pascal
Clermont-Ferrand, France
lapreste@univ-bpclermont.fr

Brigitte Rozoy
LRI, Université Paris-Sud XI
Orsay, France
rozoy@lri.fr

ABSTRACT

SIMD extensions have been a feature of choice for processor manufacturers for a couple of decades. Designed to exploit data parallelism in applications at the instruction level, these extensions still require a high level of expertise or the use of potentially fragile compiler support or vendor-specific libraries. While a large fraction of their theoretical accelerations can be obtained using such tools, exploiting such hardware becomes tedious as soon as application portability across hardware is required. In this paper, we describe Boost.SIMD, a C++ template library that simplifies the exploitation of SIMD hardware within a standard C++ programming model. Boost.SIMD provides a portable way to vectorize computation on Altivec, SSE or AVX while providing a generic way to extend the set of supported functions and hardwares. We introduce a C++ standard compliant interface for the users which increases expressiveness by providing a high-level abstraction to handle SIMD operations, an extension-specific optimization pass and a set of SIMD aware standard compliant algorithms which allow to reuse classical C++ abstractions for SIMD computation. We assess Boost.SIMD performance and applicability by providing an implementation of BLAS and image processing algorithms.

Keywords

SIMD, C++, Generic Programming, Template Metaprogramming

1. INTRODUCTION

Since the late 90's, processor manufacturers provide specialized processing units called multimedia extensions or Single Instruction Multiple Data (SIMD) extensions. The introduction of this feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. With a constantly increasing need for performance in applications, today's processor architectures offer rich SIMD instruction sets working with larger and larger SIMD registers (table 1). For example, the AVX extension introduced in 2011 enhances the x86 instruction set for the Intel Sandy Bridge and AMD Bulldozer micro-architectures by providing a distinct set of 16 256-bit registers. Similary, the forthcoming Intel MIC Architecture will embed 512-bit SIMD registers. Usage of SIMD processing units can also be mandatory for performance on embedded systems as demonstrated by the NEON and NEON2 ARM extensions [7] or the CELL-BE processor by IBM [9] which SPUs were designed as SIMD-only system.

Table 1: SIMD extensions in modern processors

Manufacturer	Extension	Registers size & nbr	Instructions
Intel	SSE	128 bits - 8	70
	SSE2	128 bits - 8/16	214
	SSE3	128 bits - 8/16	227
	SSSE3	128 bits - 8/16	227
	SSE4.1	128 bits - 8/16	274
	SSE4.2	128 bits - 8/16	281
	AVX	256 bits - 8/16	292
AMD	SSE4a	128 bits - 8/16	231
IBM Motorola	VMX	128 - 32	114
	VMX128	128 bits - 128	
	VSX	128 bits - 64	
ARM	NEON	128 bits - 16	100+

However, programming applications that take advantage of the SIMD extension available on the current target remains a complex task. Programmers that use low-level intrinsics have to deal with a verbose programming style due to the fact that SIMD instructions sets cover a few common functionnalities, requiring to bury the initial algorithms in architecture specific implementation details. Furthermore, these efforts have to be repeated for every different extension that one may want to support, making design and maintenance of such applications very time consuming.

Different approaches have been suggested to limit these shortcomings. On the compiler side, **autovectorizers** implement code analysis and transform phases to generate vectorized code as part of the code generation process [15, 18] or as an offline process [13]. By this method compilers are able to detect code fragments that can be vectorized. Autovectorizers find their limits when the classical code is not presenting a clear vectorizable pattern (complex data dependencies, non-contiguous memory accesses, aliasing or control flows). This results in a fragile SIMD code generation where performance and SIMD usage opportunities are uncertain. Other compiler-based systems use code directives to guide the vectorization process by enforcing loop vectorization. The ICC and GCC extension `#pragma simd` is such a system. To use this mechanism, developers explicitly introduce directives in their code, thus having a fine grain control on where to apply SIMDization. However, the generated code quality will greatly depend on the used compiler. Another approach is to use **libraries** like Intel MKL [6] or its AMD equivalent (ACML) [1]. Those libraries offer a set of domain-specific routines (usually linear algebra and/or signal processing) that are optimized for a given architecture. This solution suffers from a lack of flexibility as the proposed routines are optimized for specific use-case that may not fulfill arbitrary code constraints.

In this paper, we present Boost.SIMD, a high-level C++ library to program SIMD architectures. Designed as an *Embedded Domain Specific Language*, Boost.SIMD provides both expresiveness and performance by using generic programming to handle vectorization in a portable way. The paper is organized as follows: Section 2 describes the library API, its essential functionalities and its interaction with standard C++ idioms. Section 3 details the implementation of the code generation engine and its extension systems. In Section 4, experimental results obtained with a representative set of algorithms are shown and commented. Lastly, Section 5 sums up our contributions.

2. BOOST.SIMD API

The main issue of SIMD programming is the lack of proper abstractions over the usage of SIMD registers. This abstraction should not only provide a portable way to use hardware-specific registers but also enable the use of common programming idioms when designing SIMD-aware algorithms. Boost.SIMD solves these problems by providing two components :

- **An abstraction of SIMD registers** for portable algorithm design, coupled with a large set of functions covering the classical set of operators along with a sensible amount (200+) of mathematical functions and utility functions,
- **A support for C++ standard components** like `Iterator` over SIMD `Range`, SIMD-aware allocators and SIMD-aware STL algorithms.

2.1 SIMD register abstraction

The first level of abstraction introduced by Boost.SIMD is the `pack` class. For a given type `T` and a given static integral value `N` (`N` being a power of 2), a `pack` encapsulates the best type able to store a sequence of `N` elements of type `T`. For arbitrary `T` and `N`, this type is simply `std::array<T,N>` but when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used instead. This semantic provides a way to use arbitrarily large SIMD registers on any system and let the library select the best vectorizable type to handle them. By default, if `N` is not provided, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. Moreover, by carrying informations about its underlying scalar type, `pack` enables proper instruction selection even when used on extensions (like SSE2 and above) that map all integral type to a single SIMD type (`__m128i` for SSE2).

`pack` handles these low-level SIMD register type as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value, list of scalar values, iterator or range. In each case, the proper register loading strategy (splat, set, load or gather) will be issued. `pack` also takes care of issues like boolean predicates support and its range and tuple-like interface.

2.1.1 Predicates handling

Comparisons over SIMD register yield a register of boolean results. Problems arise when one wants to use such operations in conjunction with regular, non-SIMD code:

- To use custom types in C++ standard containers or algorithms, comparison operators over these types have to yield a single `bool`,
- Boolean SIMD values are either `0` or `~0` instead of `0` and `1`,
- While most SIMD extensions use the same register type to store booleans, some like Intel MIC have a special register bank for SIMD boolean operations.

Boost.SIMD solves these discrepancies by providing an abstraction over boolean values and a set of associated operations. The `logical` class encapsulates the notion of a boolean value and can be combined with

pack. Thus, for any type T, an instance of pack< logical<T> > encapsulates the proper SIMD register type able to store boolean values resulting from the application of a SIMD predicates over a pack<T>. Moreover, the logical class enables a compile-time detection of functions acting as predicates and allows optimization of SIMD selection operations. With this system, the comparison operators yield a single bool based on the lexicographical order thus making pack usable as an element of standard containers while SIMD comparisons are done through a set of predicate functions like is_equal, is_greater, etc. These functions return pack< logical<T> > and can be directly used in other SIMD function calls.

2.1.2 *Range and Tuple interface*

By providing STL-compliant begin and end member functions, pack can be iterated at runtime as a simple container of N. Similarly, since the size of pack is known at compile-time for any given type and architecture, pack can also be seen as a tuple and used as a compile-time sequence [3].

Another ability of pack is to act as an Array of Structures/Structure of Arrays adaptor. For any given type T adapted as a compile-time sequence, accessing the ith element of a pack will give access to a complete instance of T –acting as an Array of Structures– while iterating over the pack content as a compile-time sequence will yield a tuple of pack and thus making pack acts as a Structure of Arrays.

2.1.3 *Supported functions*

The pack class is completed by a hundred high-level functions:

- **C++ operators:** including support for fused operations whenever possible,
- **Constant generators:** dealing with efficient constant SIMD value generation,
- **Arithmetic functions:** including abs, sqrt, average and various others,
- **IEEE 754 functions:** enabling bit-level manipulation of floating point values, including exponent and mantissa extraction,
- **Reduction functions:** for intra-register operations like sum or product of a register elements.

A fundamental aspect of SIMD programming relies on the effective use of fused operations like multiply-add on VMX extensions or sum of absolute differences on SSE extensions. Unlike simple wrappers around SIMD operations [8], pack relies on *Expression Templates* [2] to capture the Abstract Syntax Tree (AST) of large pack-based expressions and performs compile-time optimization on this AST. These optimizations include the detection of fused operation and replacement or reordering of reductions versus elementwise operations. This compile-time optimization pass ensures that every architecture-specific optimization opportunity is captured and replaced by the superior version of the code. Moreover, the AST-based evaluation process is able to merge multiple function calls into a single inlined one, contrary to solutions like MKL where each function can only be applied on the whole data range at a time. This increases data locality and ensure high performance for any combination of function.

2.1.4 *Sample usage*

Listing 1 showcases the basic usage of pack and various elements of its interface on an architecture with 128 bits wide SIMD register. Note how pack can either be used as a whole SIMD register or as a classical Container of scalar values (including subscript operator), making the integration of SIMD operations straightforward.

```
#include <boost/simd/pack.hpp>
using namespace boost::simd;

int main()
{
  float s, tx[] = {1,2,3,4};

  // Build pack from memory and values
  pack<float> x(tx,tx+4);
  pack<float> a(1.37), b(1,-2,3,-4), r;

  // Operator and function calls
  r += min(a*x+b,b);

  // Array interface
  r[0] = 1.f + r[0];

  // Range interface: using std::accumulate
  s = accumulate(r.begin(), r.end(), 0.f);
  return 0;
}
```

Listing 1: Working with pack

2.2 C++ Standard integration

Writing small functions acting over a few pack has been covered in the previous section and we saw how the API of Boost.SIMD make such functions easy to write by abstracting away the architecture-specific code fragments. Realistic applications usually require such functions to be applied over a large set of data. To support such a use case in a simple way, Boost.SIMD provides a set of classes to integrate SIMD computation inside C++ code relying on the C++ Standard Template Library (STL) components, thus reusing its generic aspect to the fullest.

Based on Generic Programming as defined by [17], the STL is based on the separation between data, stored in various `Containers`, and the way one can traverse these data, thanks to `Iterators` and algorithms. Instead of providing SIMD aware containers, Boost.SIMD reuses existing STL Concepts to adapt STL-based code to SIMD computations. The goal of this integration is to find standard ways to express classical SIMD programming idioms, thus raising expresiveness and still benefiting from the expertise put into these idioms. More specifically, Boost.SIMD provides SIMD-aware allocators, iterators for regular SIMD computations – including interleaved data or sliding window iterators – and hardware-optimized algorithms.

2.2.1 Aligned allocator

The hardware implementation of SIMD processing units introduces constraints related to memory handling. Performance is guaranteed by accessing to the memory through dedicated `load` and `store` intrinsics that perform register-length aligned memory accesses. This constraint requires a special memory allocation strategy via OS and compiler-specific function calls.

Boost.SIMD provides two STL compliant allocators dealing with this kind of alignment. The first one, `simd::allocator`, wraps these OS and compiler functions in a simple STL-compliant allocator. When an existing allocator defines a specific memory allocation strategy, the user can adapt it to handle alignment by wrapping it in `simd::allocator_adaptor`.

2.2.2 SIMD Iterator

Modern C++ programming style based on Generic Programming usually leads to an intensive use of various STL components like Iterators. Boost.SIMD provides iterator adaptors that turn regular random access iterators into iterators suitable for SIMD processing. These adaptors act as free functions taking regular iterators as parameters and return iterators that output `pack` whenever dereferenced. These iterators are then usable directly in usual STL algorithms such as `transform` or `fold` (Listing 2).

```
vector<int,allocator<int> > v(128), r(128);

transform  ( simd::begin(v.begin())
           , simd::end(v.end())
           , simd::begin(r.begin())
           , [](pack<int>& p){ return -p; }
           );
```

Listing 2: SIMD Iterator with STL algorithm

Code written this way keeps a conventional structure and facilitate an usage of template functors for both scalar and SIMD cases also helps maximizing code reuse.

Previous examples of integration within standard algorithm are still limited. Applying `transform` or `fold` algorithm on SIMD aware data requires the size of the data to be an exact multiple of the SIMD register width and, in the case of fold, to potentially perform additional operations at the end of its call. To alleviate this limitation, Boost.SIMD provides its own overload for both `transform` and `fold` that take care of potential trailing data and performs proper completion of `fold`.

2.2.3 Sliding Window Iterator

Some application domains like image or signal processing require specific memory access patterns in which the **neighborhood** of a given value is used in the computation. Digital filtering and convolutions are examples of such algorithms. The efficient techniques for vectorizing such operations is to perform so called shifted loads *i.e.* loads from unaligned memory addresses, so that each value of the neighborhood can be available as a SIMD vector. To limit the number of such loads, a technique called the register rotation technique [14] is often used. This technique allows filter-like algorithms to perform only one load per iteration, swapping neighborhood values as the algorithm goes forward. This idiomatic way of implementing such algorithms usually increases performance by a large factor and is a strong candidate for encapsulation.

In Boost.SIMD, `shifted_iterator` is an iterator adaptor encapsulating such an abstraction. This iterator is constructed from an iterator and a compile-time width `N`. When dereferenced, it returns a static array of `N packs` containing the initial data and its shifted neighbors (Fig. 1). When incremented, the tuple value are internally swapped and the new vector of value is loaded, thus implementing register rotation. With such an iterator, one can simply write an average filter using `std::transform`.

```
struct average
{
  template<class T> typename T::value_type
  operator()(T const& t) const
  {
    typename T::value_type d(1./3);
    return (t[0]+t[1]+t[2])*d;
  }
};

vector<float> in, out;

transform( shifted_iterator<3>(in.begin())
         , shifted_iterator<3>(in.end())
         , begin(out.begin())
         , average()
         );
```

Listing 3: Average filtering

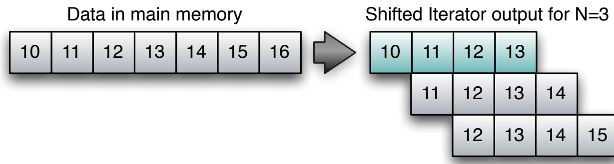

Figure 1: Shifted iterator

2.2.4 *Interleaved Iterator*

Another recurrent use case is the processing of interleaved data which requires proper restructuration into deinterleaved values before processing, either for algorithmic or performance concerns. Some of these cases can be solved by interpreting the loaded data in a SIMD vector as a complete structure instance. Nevertheless, cases like heterogeneous structures where each member has a different type or structures with a number of members which is not a divisor of the SIMD register width make this process unyieldy. Under most current architectures, such deinterleaving requires a series of scalar loads into the vector or data-layout-specific shuffling. Gathering operations as proposed in AVX2 and MIC will provide better performance for these kinds of operations and clearly indicates that this idiom is to be considered as a valid candidate for abstraction.

As for the sliding window case, Boost.SIMD encapsulates this idom in an iterator adaptor – `interleaved_iterator`. This iterator is constructed from an iterator and a type `T` which has been adapted as a tuple through the Boost.Fusion [3] adaptor. When dereferenced, it returns a tuple of `pack` containing a full vector of data for each of the members of `T` (Fig. 2).

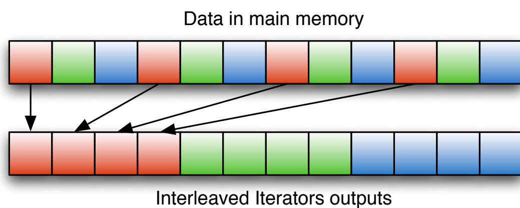

Figure 2: Interleaved iterator

Incrementing such an iterator leads to the proper calculation of the next data to gather by using a gather-like operation (real or emulated) or by using a user-defined shuffle based deinterleaving function called `unwrap` which presence is automatically detected by the system. This allows for the common case of homogeneous structures deinterleaving to be potentially optimized. Listing 4 demonstrates the use of such an iterator to normalize a set of weigthed 2D points.

```
struct point { float x,y; int w; };

struct normalize
{
 template<class T>
 T operator()(T const& t) const
 {
   typedef result_of::at_c<T,2>::type type;
   T that( at_c<0>(t)/to_float(at_c<2>(t))
         , at_c<1>(t)/to_float(at_c<2>(t))
         , One<type>()
         );
   return that;
 }
};

vector<point> in(128), out(128);

vector<point>::iterator b = in.begin();
vector<point>::iterator e = in.end();

transform( interleaved_iterator<point>(b)
         , interleaved_iterator<point>(e)
         , begin(out.begin())
         , normalize()
         );
```

Listing 4: Heterogeneous data processing

3. IMPLEMENTATION

Boost.SIMD's implementation relies on two elements: the use of Boost.Proto [12] to capture and transform expressions at compile time and a generalized tag dispatching system that allows for fine to coarse grain function specialization taking both types and architectures into account.

3.1 AST Manipulation with Boost.Proto

As stated earlier, SIMD instruction sets usually provide DSP-like fused operations which are able to deliver complex computation in a single cycle. Operations like fused multiply-add and sum of absolute differences are available on an increasing sub-range of SIMD extensions set. The main issue with these is that writing portable and efficient code that will use these fused operations whenever available is difficult. It implies handling a large number of variation points in the code and people unaware of their existence will obtain poor performance. To limit the amount of per-architecture expertise required by the developer of SIMD applications, Boost.SIMD is designed as an Embedded Domain Specific Language [16]. Expressions Templates [2] have been a tool of choice for such designs but writing complex EDSLs by hand lead to a hard to maintain code base. To reduce those difficulties, Niebler proposed a compiler construction toolkit for embedded languages called Proto [12]. It allows developers to specify grammar and semantic actions for EDSLs and

provides a semi-automatic generation of all the template structures needed to make it work. Compared to hand-written Expressions Templates-based EDSLs, designing a new embedded language with PROTO is done at a higher level of abstraction. In a way, PROTO supercedes the normal compiler workflow so that domain-specific code transformations can take place as soon as possible. Thanks to PROTO, and contrary to other EDSL-based solutions[5], BOOST.SIMD does not directly evaluate its compile-time AST after its capture. Instead, it relies on a multi-pass system:

- **AST Optimization:** in this pass, the AST is recursively optimized by transforming any potential sequence of functions or operators into their equivalent fused instructions,
- **Code Scheduling:** once optimized, the AST is split depending on the nature of each of its nodes. The main schedule operation takes care of reordering reduction operations so they can be evaluated in a proper temporary. This temporary is then stored into the modified AST as a simple terminal. This pass also splits the AST whenever a scalar emulation is required,
- **Code Generation:** once scheduled, the remaining parts of the AST are recursively evaluated to their hardware specific function calls, eventually by taking care of any remaining scalar emulation.

Each expression involving BOOST.SIMD types is then captured at compile-time as a single entity which is then transformed into the optimized sequence of SIMD instructions to call.

3.2 Function Dispatching

To be able to extend BOOST.SIMD, we need a way to add an arbitrary function overload on any function depending on the argument types, the actual SIMD extensions and the properties of the function itself. Classical generic overloading techniques in C++ include:

- **SFINAE**[1] combined with Traits [11] use the ability of C++ compiler to prune incorrect function overloads due to error in dependent type resolution. Compilation times of SFINAE-based overloading are often poor as every potential overload has to be instantiated before selection.
- **Tag Dispatching** uses class hierarchies, called **tags**, to represent type properties. To resolve an overload, the tag of a specific argument is extracted and used as an argument in a trampoline function. Unlike SFINAE-based solutions, it uses C++ overloading directly and has the advantage of providing best match selection in cases of multiple matches.
- **Concepts** [4] rely on higher level semantic description of type properties. Language support is required to resolve overloads based on the conformance of types to a given Concept. Currently Concept-based overloading is still an experimental language feature under consideration for inclusion in a future C++ standard.

[1]Substitution Failure Is Not An Error

All these techniques also lack a proper way to introduce the architecural information in the dispatching system. BOOST.SIMD proposes to extend Tag Dispatching by using the full set of argument tags for overload selection coupled with a tag representing the current architecture. These tags are computed as follows:

- For each SIMD family, a hierarchy of classes is defined to represent the relationship between each extension variant. For example a SSE3 tag inherits from the SSE2 tag as SSE3 is more refined than SSE2.
- For each argument type, a type called a **hierarchy** is automatically computed. This hierarchy contains information about: the type of register used to store the value (SIMD or scalar), the intrinsic properties of the type (floating point, integer, size in bytes) and the actual type itself. These hierarchies are also ordered from the most fine grained description (for example, `scalar_< int8_<char> >`) to the largest one (for example, `scalar_< arithmetic_<char> >`).

Each function overload is then discriminated by the type list built from the hierarchy of the current architecture and the hierarchies of every argument of the function. This unique set of hierarchies is then used to select a function object to perform the function call.

4. BENCHMARKS

This section displays benchmarks of BOOST.SIMD in several situations. Every benchmark has been tested using the SSE4.2, AVX and Altivec instruction sets to demonstrate the portability of the code. The benchmarks include: an implementation of the `AXPY` kernel and three image processing algorithms featuring the various types of SIMD iterator abstractions. Unless stated otherwise, the tests have been run using g++ 4.6. The SSE2, AVX and Altivec benchmarks have been done on the Nehalem, Sandy Bridge and PowerPC G5 microarchitectures respectively. Table 2 summarizes the frequencies and extensions of the processors. The benchmarks results are reported in GFlop/s and cycles per point (*cpp*) depending on the algorithm.

Table 2: **Processor details**

Architecture	Nehalem	Sandy Bridge	PowerPC G5
Max. Frequency	3.6 GHz	3.8 GHz	1.6 GHz
SIMD Extension	SSE4.2	AVX	Altivec

4.1 AXPY Kernel

The AXPY kernel computes a vector-scalar product and adds the result to another vector. With the x86 processor family, the throughput of the SIMD extension is two floating-point operations per cycle. The SIMD computation of the AXPY kernel fits the hardware specifications and its execution should result to the peak performance of the target. To reach the maximum number of floating-point operations, the source code needs to be tuned with very fine grained architecture optimizations like loop unrolling according to the instructions latency or the number of registers to use. The MKL Library proposes an optimized routine of this algorithm for the x86 processor family. Autovectorizers in compilers are also able to capture this type of kernel and generate optimized code for the targeted architecture. Table 3 shows how Boost.SIMD performs against the autovectorizers. For the particular case of altivec, the SIMD extension does not support the double precision floating-point type.

Table 3: **Boost.SIMD vs Autovectorizers for the AXPY kernel in *GFlop/s***

Type	Size	Version	SSE4.2	AVX	Altivec
float	2^9	gcc	3.56	16.8	0.9
		icc	13.21	17.77	-
		B.SIMD	4.70	7.94	1.07
	2^{14}	gcc	3.73	9.9	0.8
		icc	10.51	14.64	-
		B.SIMD	3.49	6.21	0.89
	2^{19}	gcc	3.59	8.9	0.6
		icc	8.80	11.52	-
		B.SIMD	3.98	5.70	0.35
double	2^9	gcc	3.37	6.6	-
		icc	7.49	11.31	-
		B.SIMD	1.95	3.48	-
	2^{14}	gcc	2.94	4.8	-
		icc	5.56	8.10	-
		B.SIMD	1.80	3.00	-
	2^{19}	gcc	2.85	4.3	-
		icc	4.51	6.07	-
		B.SIMD	1.63	2.45	-

The sizes of the used vectors are chosen according to the cache sizes of their respective targets so that they all fit in cache. The `gcc` and `icc` versions show the autovectorizers work on the AXPY kernel written in C code. First, the memory hierarchy directly impacts the performance of the kernel when the vectors go out of a cache level. The GNU and Intel compilers are able to vectorize and unroll the loop due to the static information available in the C code. The AXPY computation can be catched at the tree SSA (Static Single Assignment) level and gives the opportunity for optimizations to the compilers. Measurements show that the performance of Boost.SIMD is slightly behind that of the two compilers. This is because of the lack of unrolling and fine low-level code tuning, necessary to reach the peak performance of the target. The MKL library goes up to 15 GFlop/s in single precision (7.1 in double precision) and outperforms the previous results. The AXPY kernel of MKL is provided as a user function with high architecture optimizations for the Intel processors and introduces an architecture dependency in user code.

Table 4: **Boost.SIMD vs handwritten SIMD code for the AXPY kernel in *GFlop/s***

Type	Size	Version	SSE4.2
float	2^9	Ref. SIMD	4.03
		Boost.SIMD	4.70
	2^{14}	Ref. SIMD	3.40
		Boost.SIMD	3.49
	2^{19}	Ref. SIMD	3.41
		Boost.SIMD	3.98

Table 4 shows how Boost.SIMD performs against handwritten SIMD code without loop unrolling. The results of the generated code are equivalent to the SSE4.2 code and assess that Boost.SIMD delivers the expected speedup. Loop optimizations and fine load/store scheduling strategies can be added on top of Boost.SIMD to increase performance. The previous results show that Boost.SIMD provides a portable way to access the latent speed-up of the architectures. However, as not being a special-purpose library like MKL, its performance on this very demanding test is satisfactory yet still far from the peak performance. Other optimizations like loop unrolling and jamming are necessary to compete with the library solutions.

4.2 Sigma-Delta Motion Detection

The Sigma-Delta algorithm [10] is a motion detection algorithm used in image processing to discriminate moving objects from the background. Contrary to simple thresholded background substraction algorithms, Sigma-Delta uses a per-pixel variance estimation to filter out outliers due to lighting conditions or contrast variation inside the image. The whole algorithm can be expressed by a series of additions, substractions and various boolean selections. As pointed by Lacassagne in [10], the Sigma-Delta algorithm is mainly limited by memory bandwidth and so no optimizations beside SIMDization is efficient as only point-to-point operations are issued. Table 5 details how Boost.SIMD performs against scalar and handwritten versions of the algorithm; the benchmarks are realized with greyscale

images. To handle this format, the type `unsigned char` is used so each vector of the SSE2 or Altivec extension can carry sixteen elements. On the AVX side, the instruction set is not providing a support for the corresponding type.

Table 5: **Results for Sigma-Delta algorithm in *cpp***

Extension	SSE4.2		Altivec	
Size	256^2	512^2	256^2	512^2
Scalar C++(1)	9.237	9.296	14.312	27.074
Scalar C icc	2.619	2.842	-	-
Scalar C gcc	8.073	7.966	-	-
Ref. SIMD(2)	1.394	1.281	1.380	4.141
Boost.SIMD(3)	1.106	1.125	1.511	5.488
Speedup(1/3)	8.363	8.263	9.469	4.933
Overhead(2/3)	-26%	-13.9%	8.7%	24.5%

The execution time overhead introduced by the use of Boost.SIMD stays below 8.7%. On SSE2, the library outperforms the SSE2 handwritten version while on Altivec, a slow-down appears with images of 512×512 elements. Such a scenario can be explained by the amount of images used by the algorithm and their sizes. Three vectors of type `unsigned char` need to be accessed during the computation which is the critical section of the Sigma-Delta algorithm. The highest cache level of the PowerPC 970FX provide 512 KBytes of memory which is not enough to fit the three images in cache. Cache misses becomes preponderant and the Load/Store unit of the Altivec extension keeps waiting for data from the main memory. This problem goes away on the Nehalem microarchitecture with an additional level of cache memory. The `icc` autovectorizer generates SSE4.2 code with the C version of Sigma-Delta while `gcc` fails. The C++ version keeps its fully scalar properties even with the autovectorizers enabled due to the lack of static information introduced by the Generic Programming Style of the C++ language. Boost.SIMD keeps the high level abstraction provided by the use of STL code and is able to reach the performance of the vectorized reference code. In addition, the portability of the Boost.SIMD code gives access to the previous speedups without rewriting the code.

4.3 RGB2YUV Color space conversion

The RGB and YUV models are both color spaces with three components that encode images or videos. Opposed to RGB, the YUV color space takes into account the human perception of the colors. The Y component encodes the luminance information (brightness) and the U and V components have the chrominance information (color). The YUV model makes artifacts that can happened during transmission or compression less noticeable for the human eye. The conversion from the RGB color space is done as follow:

- Weighted values of R, G and B are summed to obtain Y,
- Scaled differences are computed between Y and the B and R values to produce U and V components.

The RGB2YUV algorithm fits the data parallelism requirement for SIMD computation. The comparison shown in Table 6 aims to measure the performance of Boost.SIMD against scalar C++ code and SIMD reference code. The implementation of the transformation is realized in single precision floating-point.

Table 6: **Results for RGB2YUV algorithm in *cpp***

Size	Version	SSE4.2	AVX	Altivec
128^2	Scalar C++	30.53	23.01	39.07
	Ref. SIMD	5.30	3.99	13.18
	Boost.SIMD	4.32	3.51	12.89
	Speedup	7.07	6.55	3.03
	Overhead	-22.7%	-13.7%	-2.2%
256^2	Scalar C++	29.23	21.46	42.51
	Ref. SIMD	6.48	2.80	29.05
	Boost.SIMD	6.51	2.45	29.01
	Speedup	4.49	8.76	1.47
	Overhead	0.04%	-14.3%	0.1%
512^2	Scalar C++	28.91	22.03	44.93
	Ref. SIMD	6.54	3.01	30.05
	Boost.SIMD	6.53	3.83	30.42
	Speedup	4.42	5.75	1.48
	Overhead	-0.1%	21.4%	1.2%

The speedups obtained with SEE4.2 are superior to the expected $\times 4$ on such an extension and no overhead is added by Boost.SIMD. With AVX, the theoritical speedup is reached for an image of 256×256 elements. A slowdown appears with other sizes due to the memory hierarchy bottleneck. This effect directly impacts the performance on the PowerPC architecture for sizes larger than 128×128 elements. Like introduced in the Sigma-Delta benchmark, the lack of a level 3 cache causes the SIMD unit to wait constantly for data from the main memory. For a size of 64×64 on the PowerPC, Boost.SIMD performs at 4.65 cpp against 36.32 cpp for the scalar version giving a speedup of 7.81 which confirms the memory limitation of the PowerPC architecture. The latent data parallelism in the RGB2YUV algorithm is fully exploited by Boost.SIMD and the benchmarks corroborate the ability of the library to generate effecient SIMD code.

5. CONCLUSION

SIMD instruction sets are a technology present in an ever growing number of architectures. Despite the performance boost that such extensions usually provide, SIMD has been usually underused. However, as new SIMD-enabled hardware gets designed with increasingly large SIMD vector sizes, losing the $\times 4$ to $\times 16$ speed-ups they may provide in HPC applications is starting to be glaring. In this paper, we presented Boost.SIMD, a C++ software library which aims at simplifying the design of SIMD-aware applications while providing a portable high-level API, integrated with the C++ Standard Template Library. Thanks to a meta-programmed optimization opportunity detection and code generation, Boost.SIMD has been shown to be on par with hand written SIMD code while also providing sufficent expressiveness to implement non-trivial applications.In opposition to compilers dependent optimizations and manufacturers libraries, Boost.SIMD solves the problem of portable SIMD code generation.

Future works include providing support for classical numeric types like complex numbers or various pixel encodings, using the AST exploration system to estimate the proper unrolling and blocking factor for any given expression, and expanding the library to SIMD-enabled embedded systems like the ARM Cortex processor family.

6. ACKNOWLEDGMENT

The authors would like to thank Pierre-Louis Caruana for his help on the experiments reported in this paper.

7. REFERENCES

[1] AMD. Amd core math library. http://developer.amd.com/libraries/acml/pages/default.aspx.

[2] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39, 1998.

[3] J. de Guzman, D. Marsden, and T. Schwinger. Boost.fusion library. http://www.boost.org/doc/libs/release/libs/fusion/doc/html.

[4] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in c++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[5] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[6] Intel. Math kernel library. http://developer.intel.com/software/products/mkl/.

[7] M. Jang, K. Kim, and K. Kim. The performance analysis of arm neon technology for mobile platforms. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 104–106, New York, NY, USA, 2011. ACM.

[8] M. Kretz and V. Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 2011.

[9] J. Kurzak, W. Alvaro, and J. Dongarra. Optimizing matrix multiplication for a short-vector simd architecture : Cell processor. *Parallel Computing*, 35(3):138 – 150, 2009.

[10] L. Lacassagne, A. Manzanera, J. Denoulet, and A. Mérigot. High performance motion detection: some trends toward new embedded architectures for vision systems. *Journal of Real-Time Image Processing*, 4:127–146, 2009.

[11] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[12] E. Niebler. Proto : A compiler construction toolkit for DSELs. In *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007.

[13] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor simd: Auto-vectorize once, run everywhere. In *CGO*, pages 151–160, 2011.

[14] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, and S. Bouaziz. Parallelization schemes for memory optimization on the cell processor: a case study on the harris corner detector. In P. Stenström, editor, *Transactions on high-performance embedded architectures and compilers III*, pages 177–200. Springer-Verlag, Berlin, Heidelberg, 2011.

[15] J. Shin. Introducing control flow into vectorized code. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:280–291, 2007.

[16] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91 – 99, 2001.

[17] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, 1995.

[18] H. Wang, H. Andrade, B. Gedik, and K.-L. Wu. A code generation approach for auto-vectorization in the spade compiler. In *LCPC'09*, pages 383–390, 2009.