A high-performance portable abstract interface for explicit SIMD vectorization.

P. Karpiński
CERN, the European Organization for Nuclear Research
CH-1211 Geneva 23
przemyslaw.karpinski@cern.ch

J. McDonald
National University of Ireland
Maynooth, Ireland
johnmcd@cs.nuim.ie

Abstract

This work establishes a scalable, easy to use and efficient approach for exploiting SIMD capabilities of modern CPUs, without the need for extensive knowledge of architecture specific instruction sets. We provide a description of a new API, known as UME::SIMD, which provides a flexible, portable, type-oriented abstraction for SIMD instruction set architectures. Requirements for such libraries are analysed based on existing, as well as proposed future solutions. A software architecture that achieves these requirements is explained, and its performance evaluated. Finally we discuss how the API fits into the existing, and future software ecosystem.

CCS Concepts •**Theory of computation → Parallel computing models;** •**Computer systems organization → Single instruction, multiple data;** •**Software and its engineering → Concurrent programming structures; Software libraries and repositories;**

Keywords SIMD, C++, Vectorization, Portability, Abstract Interface

ACM Reference format:
P. Karpiński and J. McDonald. 2017. A high-performance portable abstract interface for explicit SIMD vectorization.. In *Proceedings of PMAM'17, Austin, TX, USA, February 04-08 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3026937.3026939

1 Introduction

This paper provides details of the Unified Multi/Many-Core Environment (UME), an experimental framework to establish a scalable and easy to use system to assist developers in exploiting upcoming multi- and manycore CPU architectures. The final framework will consist of a number of separate modules, each dealing with a separate aspect of modern CPU architectures, however in this paper we focus on UME::SIMD, which specifically addresses the issue of SIMD vectorization. UME::SIMD allows the programmer to access the SIMD capabilities of modern CPU's, without the need for extensive knowledge of architecture specific instruction sets. This is achieved by defining a set of abstract vector types together with a wide set of architecture independent operations to allow portability, decrease the learning curve and to provide a concise and complete model for vectorization.

The internal performance component of the library is implemented using vendor specific extensions to the C++ programming language, so called compiler intrinsic functions. Given that C++ is a language with a mature compiler infrastructure, we demonstrate that generic compiler optimizations ensure that a negligible overhead is introduced by this abstraction. For non-SIMD CPU's the library provides a default implementation of the interface that is developed using standard C++ language only.

PMAM'17, Austin, TX, USA

978-1-4503-4883-6/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3026937.3026939

2 Background

2.1 Vectorization in Commodity CPU's

Vectorization in computing is not a new topic with the technology already applied for more than 50 years. The basic idea behind vectorization is to allow operations not only on scalars (single-value variables), but also on vectors (1-D arrays of scalars), matrices (2-D arrays of scalars) and even higher order primitives. Vectorization has two basic advantages: simplified mathematical notation and increased performance. The first is related to the representation of algebraic formulas, where formulas use vectors and matrices as entities for simplifying expressions and providing a canonical abstract notation. Secondly, since there are no flow dependencies between elements of vector primitives, there exists a hypothetical possibility of executing certain operations (e.g. vector-vector addition) in parallel on all scalar elements of such a primitive. Furthermore because of data locality this will result in a decrease of the overall latency of the computational process [6]. An interesting mathematical formulation of vector processing primitives has been presented by Iverson [14].

In order to achieve higher performance, vector operations require additional hardware resources. Until mid-90's vectorization machinery was limited to supercomputers howeverm relatively recently, a similar although limited, set of capabilities became a part of consumer level commodity CPU's. The most advanced CPU vectorization technology is a set of SIMD extensions for the x86 instruction set architecture (ISA) developed by Intel. These instruction sets started as the MMX instruction set [20], a small number of highly specialized instructions allowing operation on multiple scalar elements using 64b registers. Due to a very restricted instruction set the range of potential applications was limited to multimedia or highly specialized primitive math functions (e.g. implementations of standard libraries or compression algorithms). The second family of extensions called SSE (Streaming SIMD Extensions) has been introduced, with vector registers extended to 128b, and additional instructions. In recent years a third extension called Advanced Vector Extensions (AVX) was presented, broadening further the vector registers to 256b (for floating point operations). It was then followed-up with AVX2 which added, 256b integer operations and gather operations. The AVX-512 extension, currently the state-of-the-art, doubles the size of vector registers to 512b, introduces independent mask registers and adds conflict detection instructions, gather/scatter instructions, prefetching instructions, reciprocal/exponential approximation instructions, and others [5].

The biggest issue of the new extension set is that it is fragmented into specific sub-extensions (AVX512-Foundation, AVX512-Conflict Detection, AVX512-Vector Length, etc.) making portability more difficult even when dealing with different CPUs manufactured by the same company. Apart from multiple extensions developed by Intel, there are also other SIMD technologies in CPUs from other manufacturers, such as ARM - NEON instruction set [2], Cell - SPU [10] or IBM - AltiVec [11].

While each of these SIMD instruction set extensions promise performance gain (e.g. up to 16x against scalar code for AVX-512), programming for a particular instruction set heavily limits application portability. Various gaps in the available instruction sets, such as AVX512 fragmentation, lead to poor efficiency or very time consuming workarounds. Finally any potential user requires a detailed knowledge of the behaviour of all instruction set that the developed software requires, which prolongs the learning process and increases software complexity.

Programming languages are an interface between a conceptual, mathematical notation and underlying computer hardware architecture. On one hand they constrain the formal mathematical specification so that solutions are expressed unambiguously, on the other, they provide, portability over hardware architectures. A broad range of array languages such as APL[14], Fortran[12], or Cilk Plus[3] already provide a rich syntax for expressing algorithms using array notations or vector primitives. However, such support is not present in more mainstream languages such as C++ [22]. Support for vectorization within C++ has already been proposed multiple times, by independent contributors [7] [19] [17], however none of these proposals have been accepted by the standardisation committee thus far.

As explained in Section 1, the aim of this work is to create an abstraction that would allow programmers to increase their productivity, but without significant sacrifices in performance or portability. We selected C++ as our language of choice, because of its popularity, typeset programming capabilities and a mature tool-chains. At the same time we also believe that the proposed typeset model can be implemented at compilers level to enable similar capabilities in other languages. We thus present a typeset model, a C++ template library and evaluation of its' performance on state-of-art microprocessors. We also present an intuitive and easy to learn API aiming for software development productivity.

2.2 Prior Work

C++ compilers, such as: GNU Compiler Collection, Intel C++ Compiler, Visual Studio Compiler and Clang, provide multiple different ways of generating vector instructions:

- **Auto-vectorization** - as described in [18] using this method the performance of the resulting output is highly variable.
- **Explicit vectorization using compiler directives** - an example can be OpenMP [4] which offers a set of portable standardised clauses for SIMD vectorization.
- **Explicit vectorization using compiler intrinsic functions** - this approach is very similar to assembly programming, yet it allows some compiler optimizations.

Work done by Kretz and Lindenstruth on the VC library is the closest work to that presented in this paper. It has already been discussed why explicit vectorization using compiler intrinsics is an appealing approach, that it leads to significant speedups over auto-vectorization, addresses code readability and positively affects learning curve [18]. In VC library any vector type derives its' rank (number of elements in binary representation) and available interface from its base scalar type. However mask types are not universal, but rather depend on the vector type used. The vector rank is implied by the instruction set used and cannot be controlled by the user. The library also provides implementation for basic math functions such as trigonometric, exponential and FMA (*Fused Multiply and Add*) operations.

Similar work has been done also by A. Fog on VCL library [8]. As in VC, VCL defines a set of vector types, and mask types which create an abstract interface allowing the users to write C++ standard code. This hides intrinsic functions and exposes vector operations in a user-friendly manner. The major difference with VC is a model of vectorization using three parameters: scalar type, scalar precision rank and length of a vector to determine unambiguously a vector type. While the library provides higher ease of use, it also explicitly prohibits user from using inefficient operations, which leads to non-portable code.

A portable, virtual vector instruction set was also presented by Bocchino and Adve on LLVA[21]. This and similar representations are suitable for intermediate compiler representations, as well as for virtual execution environments. For languages such as C++ where no vectorization support is provided, creating a mapping between available language constructs and such virtual ISA, is possible only with some compiler dependant extensions or by relying on auto-vectorization. As a result modification to existing compiler infrastructure might be required. In addition to infrastructural problems, also the set of operations for such vector abstraction should be extended to be used not only in media applications, but for any future software.

3 UME::SIMD Interface

In this section we present details of the UME::SIMD interface. The interface is a C++ 11 compatible programming API and consists of two parts:

- A set of SIMD data types abstracting vector register types (e.g. XMM, YMM, ZMM for AVXx extensions), and,
- A set of operations permitted on these data types.

The library uses 3-parametric model to unambiguously distinguish data types:

- **scalar category** one of boolean, unsigned integer, signed integer, floating point and index.
- **scalar rank** number of bits of precision used to represent a scalar value, that is: 8b, 16b, 32b, 64b.
- **vector rank (length)** number of scalar elements packed in a SIMD vector.

3.1 Primitive types

The simplest and most basic category of types distinguished in the library is a **mask** ($\mathrm{SIMDMask}[RANK]$). A mask type, similarly as in VC or VCL is a vector of boolean elements, with one major difference: in UME::SIMD the mask types are defined as independent of the vector types that they interact with. The mask is defined purely with regard to the vector length. The reason for that is that in instruction sets preceding AVX512 there was no concept of a mask register, and all mask operations were only permitted with another vector register used as the masking operand. Types used

for representing masks in SSEx and AVX/AVX2 are not uniform and depend on the instruction used. To avoid additional mask conversions when mixing both integer/floating point operations, both VC and VCL define separate mask types for every vector type. When considering AVX512 this behavior is unnecessary and unintuitive.

The second basic data type, an **index vector** (SIMDSwizzle[*RANK*]), is an experimental data type for representing permutations and element reordering. In simplest implementation, a swizzle index scalar can be an unsigned integer value ranging from zero to vector length (*RANK*). A swizzle vector could be then represented using an unsigned integer vector. However since the permutation operands change for both instruction sets, vector types and even single CPU instructions, separate representations for index vectors are required.

The third class of data types are **arithmetic vectors** (SIMDVec[*RANK*]_[*BASE_SCALAR*]). which are used for majority of data manipulation operations. The library defines types necessary for representing up to 1024b arithmetic vectors. Supported vector ranks (for both arithmetic, swizzle and mask vectors) are natural numbers in range <1, 128 >, which allows representation of both SIMDVec1_x (i.e. SIMD types containing only one element) and SIMDVec128_x (i.e. since 128 is the number of 8b elements that can be packed in 1024b vector). While no currently available ISA extension supports 1024b vector registers, the value was chosen to prove scalability of the interface. Summarizing, the interface defines: 8 mask types, 8 swizzle types and 63 arithmetic vector types.

3.2 Primitive operations

The second part of the interface is the definition of operations permitted on different SIMD types. Each operation has been defined formally in a way that makes its result compatible with equivalent C++ 11 code. **Table 2** shows an example of the formal definition for the MADDV operation.

The complete interface has been divided into specific classes depending on the category of permitted vector types: (mask, swizzle, signed/unsigned integer or float) and vector length (PACK/UNPACK operations). In total the interfaces define more than 300 operations including:

- operation between mask types (e.g. boolean AND/OR),
- basic arithmetic operations (e.g. addition, multiplication),
- reduction operations (e.g. HADD - add all vector elements and return a scalar value),
- bitwise operations between vectors (e.g. binary AND, binary OR),
- swizzle operations (e.g. SWIZZLE - permute elements of a vector, BLEND - mix content of two vectors),
- pack operations (e.g. PACK - assemble vector with a vector of half-length),
- additional math functions (e.g. SIN - calculate sine, MIN - select minimum of two values),
- fused arithmetic (e.g. FMADD - fused multiply and add).

The implementation groups specific types of operations into sub-interfaces which can then be inherited by vector types. A full diagram of the interface hierarchy is shown in **Figure 1**. The full list of operations is available online at project website [16].

The majority of operations, permitted on arithmetic vector types, are also available in masked versions. Additionally, some operations are also implemented in 'in-place' assignment form to mimic the behaviour of assignment operators, such as: '+=', '*=', etc.

Operator overloading for vector types is a very important feature, with both VC and VCL defining overloaded operators, to expose functionality of intrinsics in an intuitive manner via C++. Vector types require an additional operand for representing the operation mask. Unfortunately the language does not allow overloading of a ternary operator, and thus, another approach to representing an operation has to be used. Consider function definition at **Figure 2**. The function takes 3 vector arguments of type SIMD4_64f (vectors of four 64b floating point scalars), and returns a fused multiply and add result. This function uses prefix notation in the context of the member function interface (MFI). Member functions, here *mul* and *add* take one explicit vector argument, and one implicit argument (a *this* object). With overloaded operators, the same function can be represented as on **Figure 3**. Clearly this notation is easier to comprehend. Unfortunately, the masked version can only be represented correctly with additional argument, which makes it impossible to use, even overloaded, operator syntax (**Figure 4**).

Seemingly a syntactic detail, this notation is necessary for mask propagation through whole chain of operations, without the need to rely on code generation level compiler optimizations. A simplified syntax using write masks used in VC and available in UME::SIMD allows masking only the last operation evaluated that is an assignment (**Figure 5**). The syntax of this construct is similar to one present in *std::valarray* [13].

Regardless of syntax problems, the MFI provides a flexible methodology for exposing a wide set of functionality of the underlying hardware, without exposing the implementation details to the user. Usage of operators is still allowed in cases where it is possible, to facilitate software development.

Besides MFI and operator syntax, it is also possible to use a namespace scoped non-member functions to access required operations, as presented on **Figure 6**.

3.3 Adaptability

A number of features allowing scalability such as: uniform mask/swizzle types, extensibility of vector lengths and definition of MFI have already been discussed. From the long-term perspective, it is important to allow extensibility of the interface in terms of new instruction sets and new MFI operations. The biggest issue in API design is that it is not possible to predict what types of operations might be available in future instruction sets, and what operations might be necessary for performance optimizations. This concern is important in terms of long-running scientific projects that have to undergo periodic hardware changes, without extensive software modifications, and for which optimization is a life-long process.

To provide maximum capability, the supported interface must be a superposition of interfaces exposed by all supported instruction sets. As a consequence the full implementation of the existing interface for a new instruction set can take significant amount of time. While an interface that is too extensive might be a burden to both develop and maintain, a comprehensive interface allow problems such as the efficient implementation for both RISK and CISC architectures to be addressed.

For maximum portability and flexibility, a *scalar emulation engine* was designed. The advantage of this approach is that it provides a set of default fall back routines and creates a compatibility verification mechanism. The scalar emulation engine uses static polymorphism

Table 1. Results of UME::SIMD microbenchmarks on Xeon E3-1280 processor. The values show speedup against a reference configuration: best result obtained with optimized (also auto-vectorized) scalar code. Results limited by specialized code availability.

	ICC 17.0			**Clang 3.9**			**GCC 5.3**		
Microbenchmark	scalar	AVX2 intrinsics	UME::SIMD	scalar	AVX2 intrinsics	UME::SIMD	scalar	AVX2 intrinsics	UME::SIMD
Single floating-point precision (32b)									
average	1.00	6.90	8.01	0.97	7.07	7.92	1.64	6.84	7.95
exp	1.00	-	1.23	2.81	-	1.64	0.78	-	0.49
log	1.00	-	1.03	0.49	-	0.15	0.10	-	0.10
histogram2	1.00	0.89	1.01	1.23	1.10	1.36	1.49	1.52	1.49
mandelbrot2	1.00	5.71	6.48	1.00	5.72	6.57	0.98	5.76	6.35
polynomial	1.00	4.35	4.34	0.59	3.14	3.18	0.62	3.23	3.19
QuadraticSolver	1.00	5.13	5.26	1.00	5.23	5.26	0.89	-	4.63
sincos	1.00	-	0.79	1.15	-	0.35	0.10	-	0.11
Double floating-point precision (64b)									
average	1.00	3.62	4.16	0.97	3.67	4.42	1.67	3.62	4.42
exp	1.00	-	0.95	1.84	-	1.11	0.61	-	0.60
log	1.00	-	2.15	1.86	-	0.91	0.24	-	0.23
histogram2	1.00	0.82	1.01	1.11	0.88	1.11	1.46	1.32	1.44
mandelbrot2	1.00	-	2.94	0.98	-	3.21	0.98	-	2.23
polynomial	1.00	3.58	3.64	0.56	1.97	1.97	0.58	2.01	2.02
QuadraticSolver	1.00	-	1.83	0.94	-	1.60	0.96	-	5.16
sincos	1.00	-	2.63	2.77	-	2.19	0.46	-	0.46

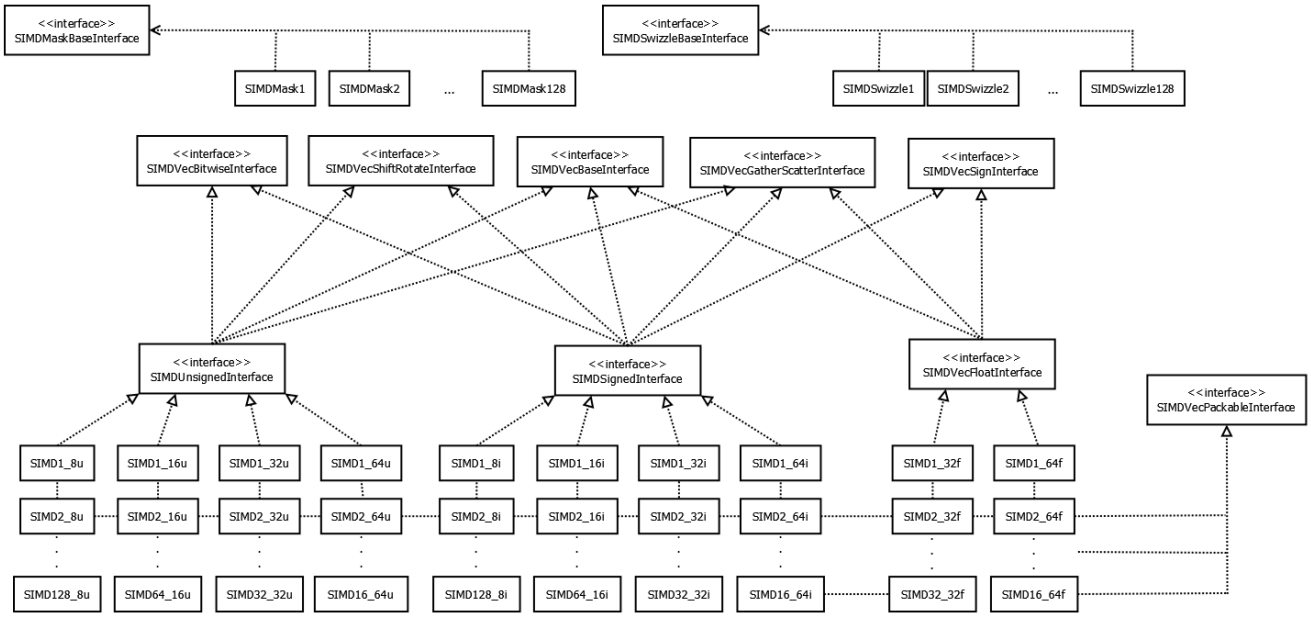

Figure 1. Taxonomy of types in UME::SIMD. Different operations are permitted on different data types.

to feed a final SIMD type with a full definition of MFI which is executed in the case where no specialized version of a given function is explicitly provided for the given vector type. Scalar emulation provides the following benefits to the library design:

- allows testing design choices without writing difficult and error prone intrinsic code
- provides an extensible software architecture for extending interfaces with new operations
- provides an easy alternative version for ISA's not supporting specific operations
- creates a reference for testing results generated for any vector ISA

Table 2. Formal definition of MADDV operation. Because of member function abstract interface, one of the operands is implicit.

MADDV - Masked addition between two vectors	
Signature:	VEC_T add (MASK_T const & mask, VEC_T const & vectorOp)
Parameters:	mask: used for element selection vectorOp: explicit vector operand
Returns:	Temporary vector object (temp)

Behaviour:	This method performs SIMD addition of operands from caller (*this) and from vector specified as vectorOp.
	The following holds true for all index values in the range<0; VEC_LEN-1>:
	
	IF mask[i] == true THEN
	temp[i] <- vec[i] + op[i]
	ELSE
	temp[i] <- vec[i]
	ENDIF

Figure 2. FMA primitive using Member-function (prefix syntax).

```
SIMD4_64f fma(
  SIMD4_64f & a,
  SIMD4_64f & b,
  SIMD4_64f & c)
{
  return a.add(b.mul(c));
}
```

Figure 3. FMA primitive using overloaded operator (postfix syntax).

```
SIMD4_64f fma(
  SIMD4_64f & a,
  SIMD4_64f & b,
  SIMD4_64f & c)
{
  return a + b * c;
}
```

Figure 4. FMA primitive using masking (prefix syntax).

```
SIMD4_64f fma(
  SIMDMask4 & mask,
  SIMD4_64f & a,
  SIMD4_64f & b,
  SIMD4_64f & c)
{
  return a.add(mask, b.mul(mask, c));
}
```

- is implemented using C++ 11 standard language and hence should port easily to new targets, even if supported ISA are not present.

Figure 5. FMA primitive using masking (mask-assignment syntax).

```
SIMD4_64f fma(
  SIMDMask4 & mask,
  SIMD4_64f & a,
  SIMD4_64f & b,
  SIMD4_64f & c)
{
  SIMD4_64f temp;
  temp[mask] = a + b * c;
  return temp;
}
```

Figure 6. FMA primitive using masking (C-like function syntax).

```
SIMD4_64f fma(SIMDMask4 & mask,
  SIMD4_64f & a,
  SIMD4_64f & b,
  SIMD4_64f & c)
{
  SIMD4_64f temp = add(mask, a, mul(b, c));
  return temp;
}
```

3.4 SIMD-1

The majority of basic, well defined operations (in terms of the core standard language) are permissible on vector types. However, some operations, such as mask operations or reduction operations, never had any purpose in scalar codes (except for boolean predicates and array indices, which again are limited to scalar entities). For that reason it is not possible to write generic template code using either scalar or SIMD vector types. SIMD-1 is a special SIMD rank that in practice creates an encapsulation of scalar elements, but which also supports the MFI interface. With SIMD-1 types it is possible to write generic code that can be then compiled to either scalar or vectorized code. Because there is no need to use specialized intrinsic functions and since compilers are already very good at optimizing trivial scalar code, usually the result of SIMD-1 operations is equivalent to regular scalar code executed. Hence the use of SIMD-1 types removes the need for specialized scalar versions of the code. It also permits the generic optimization rules to be also applied for code executed on scalar machines, or in cases when specific workloads would degrade the performance when higher SIMD ranks are used.

3.5 Type info

SIMD data types can be used with template metaprogramming techniques. Such techniques usually require additional type information. An example here would be information about other correlated types that might be necessary in algorithm code. UME::SIMD defines a set of trait template classes (SIMDTraits<*T*>) that allow accessing, at compile time, additional type information. This feature is necessary to limit the number of template parameters in templated code and to make type dependencies easier to comprehend.

3.6 Type conversions

Creating an unambiguous, consistent and closed type system requires precise definition of relations between different types of vectors. In the UME::SIMD typeset there are three types of conversions available:

- concatenating and splitting
- precision promotion
- fundamental type change

Concatenating (**PACK**) or splitting (**UNPACK**) operations are necessary for modifying vector rank that is used during vector operations. This type of operations might be necessary for solving problems with flow dependencies between vector elements. Precision promotions (**PROMOTE** and **DEMOTE**) are operations that do not change the rank of a SIMD vector, but change the precision of packed scalars. Fundamental type changing conversions (**UTOI**, **ITOU**, **UTOF**, **FTOU**, **ITOF**, **FTOI**) are the conversions that change between: signed/unsigned integer and floating point types. When performing this type of conversions both scalar precision and vector rank are preserved. **Figure 7** illustrates the possible type conversions for SIMD4_32i (a) and SIMD1_32i (b) types. **NullType** is a special termination type used for conversions that don't have a correct semantic meaning (e.g. unpacking rank 1 vectors). Null types can be used in templated code for defining boundary scenarios.

3.7 Programming Model

UME::SIMD does not impose any programming model on the user code, other than one imposed by vector arithmetic. The library does not use any non-static context, does not require runtime initialization and does not require any additional linking. It is provided purely as a set of header files defining a namespace (UME::SIMD::) which contains all SIMD types and operations. The types can be then used in standard user code in exactly the same manner as regular scalar variables.

A suggested usage consists of writing kernels (functions) of vectorized code with data layouts managed by the calling code. In other words, UME::SIMD can act as an instruction-level generation mechanism to obtain fine-grained instruction generation, to replace mechanisms provided by current C++ compilers.

4 Performance

UME::SIMD is a library intended for achieving highly efficient code kernels. The evaluation we are presenting here consists of two parts: synthetic microbenchmarks and a real-world usage scenario. The microbenchmarks are used primarily for performance monitoring at a granularity of specific interface operations, such as: trigonometric functions, fused multiply-add operations or conflict detection. For the real-world usage scenario, the library has been integrated into the GeantV project [1] at CERN. GeantV is a new version of highly parallel detector simulation framework, with a variety of potential applications in areas including: High Energy Physics (HEP), medicine, aeronautics and others. Integration of UME::SIMD as a critical software component in a project of this scale creates a unique possibility to direct the development of the library based on user requirements and feedback.

When performing such evaluation, multiple factors have to be taken into consideration. First of all, the quality of code generated depends heavily on the compiler. Monitoring different toolchains as they evolve is necessary for ensuring that portability is not impacted negatively. Second, we are interested in finding sets of compilation flags to make sure that compilers don't break the performance by unwanted optimizations. Third, the codes heavily dependent on runtime information might not be efficiently implemented with the existing interface, and therefore might require interface extension. Finally, we would like to compare our new solution to existing implementation technologies, which requires additional code infrastructure and also highly efficient, specialized implementations.

In order to make the data presentation clear, we present aggregation of selected measurements for specific static and runtime configurations. We also present the results only for representatives of Xeon and Xeon Phi processor families. It is not our intention to perform a comparison between these platforms. Such comparison shouldn't be performed at kernel level but rather at the level of the actual working application. Since both UME::SIMD and GeantV projects are in development phase, such comparison is not yet possible. We are therefore interested in showing performance comparison between different implementation methodologies, such as auto-vectorized scalar code, vector intrinsic functions and VC library.

Tables 1 and 3 aggregate the results for Xeon E3-1280 and for Xeon Phi 7120P respectively. For both platforms we performed set of measurements testing different configurations of optimization flags which we observed to affect vectorization efficiency, such as: optimization level, vector instructions extension and inlining parameterization. Over 50 different configurations have been tested, each of them with proper task pinning, multiple replications on both task and process levels, and results averaging to reach statistically reliable results. Full measurements for the most recent version are available at the project website [15].

Eight microbenchmarks, selected based on frequent usage scenarios, are presented with values obtained using different compilers. For Xeon Phi, current instabilities in Clang compiler tool-chain forced us to disable a range of affected configurations. For higher mathematical functions we are not presenting results for the intrinsic codes due to the lack of availability of proper implementations. We are hoping to fill these gaps in future microbenchmarking efforts. For different implementations we also selected only the best possible result achieved with scalar code, vector intrinsics and with UME::SIMD specific code. The results presented do not necessarily belong to the same static configuration.

A number of conclusions can be drawn from the performance values presented in the tables . First of all there is a significant variability between different compilers, and compiler versions. The quality of code generated with UME::SIMD in the majority of cases matches or even exceeds the intrinsic code. In some configurations, mostly related to 64b precision, we expect even better results as both the library and compilers development progress. A second observation is that scalar code can actually be faster in some scenarios, for instance when gather/scatter conflicts appear. The biggest issues are that some of these scenarios heavily depend on the workload. For that reason it is not possible to tell whether vectorization will improve or decrease performance. The implications of this point for projects using vectorization, is that proper performance monitoring should be a process performed continuously during the project development. A final observation is that in certain cases UME::SIMD reaches performance higher than theoretically expected due to putting more pressure on registers and microprocessor ports. This approach creates a potential for parameterized application tuning, without the need for code re-writes.

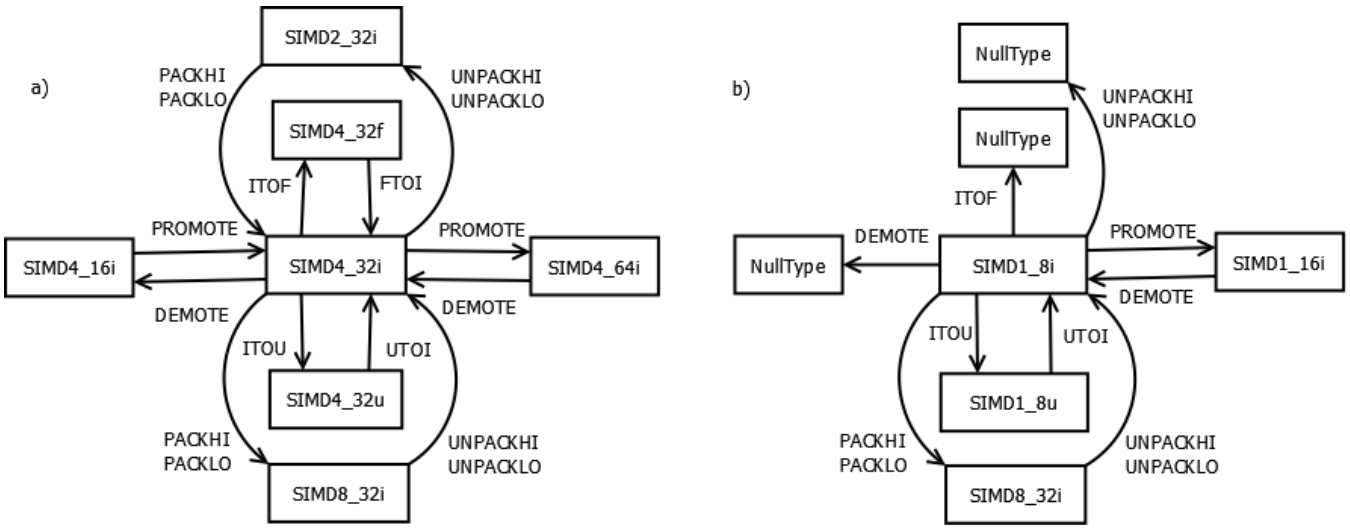

Figure 7. Diagrams of type conversions for selected types: a) SIMD4_32i has all conversions well defined; b) SIMD1_8i supports only semantically valid type conversions.

Table 3. Results of UME::SIMD microbenchmarks on Xeon Phi 7120P processor. The values show speedup against a reference configuration: best result obtained with optimized (also auto-vectorized) scalar code. Results limited by specialized code availability.

	ICC 17.0			**GCC 6.2**		
Microbenchmark	scalar	AVX512 intrinsics	UME::SIMD	scalar	AVX512 intrinsics	UME::SIMD
Single floating-point precision (32b)						
average	1.00	8.67	9.78	1.00	10.03	10.56
exp	1.00	-	1.01	0.01	-	0.01
log	1.00	-	1.01	0.02	-	0.02
histogram2	1.00	0.53	1.03	0.19	0.54	0.61
mandelbrot2	1.00	13.90	18.35	1.02	13.29	18.93
polynomial	1.00	15.79	15.86	0.67	14.97	15.46
QuadraticSolver	1.00	14.50	15.43	0.85	14.37	14.80
sincos	1.00	-	1.00	0.04	-	0.04
Double floating-point precision (64b)						
average	1.00	5.84	4.76	1.00	6.62	5.08
exp	1.00	-	44.39	1.25	-	1.26
log	1.00	-	9.25	0.16	-	0.16
histogram2	1.00	0.55	0.38	0.24	0.52	0.24
mandelbrot2	1.00	6.76	10.76	1.03	6.50	10.89
polynomial	1.00	7.57	8.21	0.65	7.52	7.86
QuadraticSolver	1.00	9.07	9.12	0.89	8.95	8.92
sincos	1.00	-	11.41	0.18	-	0.18

Further to the above it is also worth considering the code size requirements. While it is not always a good metric to be used when evaluating programmer performance, it is in this case as we are dealing with assembly-like intrinsics kernels, heavily dependent on ISA and CPU model. For intrinsic implementations, due to name C-style name mangling, separate implementations are required for both different instruction sets, as well as different scalar precision. For AVX/AVX2/AVX512 and pure x86 codes only, implemented for both single and double precision, the UME::SIMD code was roughly 8 times smaller than equivalent intrinsic code. This is due to the fact that the UME::SIMD typeset uses both template metaprogramming for precision selection and concise type-oriented design to ensure a uniform interface. This allows parameterization which in turn reduces effective code size. We hope that the existing set of microbenchmarks can be used in near future for evaluation of vectorization efficiency also on ARM and Power architecture based processors, without much porting efforts.

For the realistic benchmarking scenarios, we used a set of existing real-world benchmarks designed to test basic volume shapes usable as building blocks for high-energy physics (HEP) detector simulation. In practical cases the size and geometry of a full detector depends on a wide range of parameters, such as: particle types,

energy deposition, density of particles in a specific volume etc. This forces the frameworks such as GeantV to offer high freedom in parameterization of the shapes. At the same time, there exists a need for continuous monitoring of the performance. The shape benchmarks have been defined so that each of them exposes an interface consisting of 6 functions to be used in higher level code: **Inside**, **Contains**, **DistanceToIn**, **DistanceToOut**, **SafetyToIn** and **SafetyToOut**. Each of these functions evaluates a relation between a given particle (e.g. an electron) and a specific volume primitive placed in detector space. For example **DistanceToIn** gives a positive value describing absolute distance of the particle to the surface of given shape. These functions have been chosen for benchmarking, as they correspond to the interface exposed by the previous generation of simulators.

The GeantV project uses an explicit vector programming approach, and defines an abstraction layer (**VecCore**) mapping to different explicit vectorization backends. At the moment three backends are supported: scalar, VC and UME::SIMD. Each backend defines a set of primitive operations, such as MaskedAssign for blending operations or Exp for exponent. These backend agnostic operations then map to specific functionality exposed by different libraries, or to a generic scalar code. At the same time further room for performance improvement is left so that specialized implementations can be provided without the VecCore abstraction. This design choice had been made to allow future incremental performance improvements.

GeantV defines a set of 18 shape benchmarks, each of them measuring the time necessary for evaluating all six functions over a number of particles where the representation exceeds the processor cache size. The particles are generated randomly but with constraints on the distribution of particle position and momentum. The constraints have been selected by framework developers to a resemble the configuration of a known real simulation environment.

We are primarily interested in generating a comparison between UME::SIMD and the VC library, which is considered the state of art solution. We are also interested in knowing what is the vectorization efficiency of shape implementations. The full benchmark set we evaluated consisted of measuring all possible combinations over the following settings:

- Backend selection: scalar, VC, UME::SIMD
- compilation flags: various selections of -O2/O3, disabled vectorization (Intel compiler only)/ AVX2/ AVX512, -finline-limit, -fno-streaming-stores
- c++ 11 compatible compilers: GCC(5.3), ICC(17.0), Clang(3.9)
- all 6 functions of 18 shape benchmarks

All measurements were performed with multiple replications on task and process level and with proper platform configuration. We ran the same set of configurations and benchmarks on a Xeon E3-1280 platform and a Xeon Phi 7120 platform (see Tables 4 and 5, respectively). A single build configuration consists of 108 independent measurement values. For Xeon the list of configurations consists of 31 build configurations, whilst for Xeon Phi the configurations with Clang are disabled due to broken tool-chain. To make a presentation of the results more readable, we calculated a set of derived metrics which allow us to further limit the comparison to only few selected configurations. The selection was made based on the highest geometric mean of speedups reached for each of the framework backends. Given early stage of implementation it was necessary to disable certain inefficient kernels from our comparison.

For an AVX2 based platform (see Table 4) both VC and UME::SIMD score high on the geometric mean metric. It is important to understand, that while the expected vectorization speedup would be 4x for double precision measurements, the reference configuration is already well auto-vectorized using SSEx instructions, which cannot be fully disabled. This suggests that real vectorization efficiency for both libraries might lie in proximity of theoretical peak. We would like to state that both VC and UME::SIMD are under on-going development and still have room for performance improvement. For that reason the results presented should get better over the time for both libraries.

For AVX512 platform (see Table 5) the measurements again show similar level of performance between both libraries. In this case however the speedup, even compensated with the knowledge about partial auto-vectorization of reference configuration, suggests that there is still room for quality improvement. We also observed a number of very small performance degradations when switching between scalar and SIMD instruction generation schemes.

It is worth pointing out that the generic nature of GeantV backend system uses only a fraction of the UME::SIMD interface, with SIMD lengths selected based on target architecture. In contrast almost all of the interface exposed by VC is used, making this comparison only partial. The possibility for vertical vectorization is thus not exploited in this benchmarking comparison. Furthermore, from the functional perspective the quality of UME::SIMD interface and its robustness is already on a par with VC (in the part that is possible to be compared), whilst at the same time UME::SIMD provides a simplified programming model.

5 Supported Instruction Sets

Supported instruction sets in UME::SIMD currently include AVX, AVX2, AVX512 and IMCI. Proof of concept implementations are already present for NEON and AltiVec instruction sets. The library provides also an implementation using OpenMP 4.0, which might serve in the future as a methodology for performance portability on unsupported, yet standard compliant platforms. We hope to present similar benchmarking also for these in a follow-up publications.

6 Licensing

Ensuring scalable performance on both existing and future CPU's is widely achievable requires API's that not only provide open intuitive interfaces. Such API's should also ensure that the interface exposed provides an abstraction to the user that is relatively simple, even though the underlying implementation can grow very complex as the variety of SIMD capable CPU's increase. Maintaining a unified interface will require incorporating new architectures and vectorization extensions which we believe is best achieved by allowing users to use and extend the API under a permissive licensing structure. For this reason we have published the library [16] under an MIT license. This permits its use by any project, for academic, industrial or amateur applications, without any further obligations from or towards the project owners.

Table 4. Results of GeantV shape benchmarks on Xeon E3-1280 processor. The values show aggregated speedups against a reference configuration: scalar backend, intel compiler, '-no-vec -O2' compilation flags. All calculations using double precision.

Derived metric	ICC 17.0 scalar backend -no-novec -O2	Clang 3.9 scalar backend -march=avx2 -O2	Clang 3.9 VC backend -march=avx2 -O2	Clang 3.9 UME::SIMD backend -march=avx2 -O3
# of best results	0	15	**45**	5
# of results in top 5%	0	19	**53**	9
# of results in top 20%	1	22	**59**	43
# of results <1x	**0**	24	0	0
Geomean of speedup (65 kernels)	1.00	1.38	**2.24**	1.92

Table 5. Results of GeantV shape benchmarks on Xeon Phi 7120 processor. The values show aggregated speedups against a reference configuration: scalar backend, intel compiler, '-no-vec -O2' compilation flags. All calculations using double precision.

Derived metric	ICC 17.0 scalar backend -no-novec -O2	GCC 6.2 scalar backend -march=knl -O3	GCC 6.2 VC backend -march=knl -O3	GCC 6.2 UME::SIMD backend -march=knl -O2
# of best results	0	13	16	**37**
# of results in top 5%	0	13	26	**38**
# of results in top 20%	0	14	38	**43**
# of results <1x	**0**	34	**0**	6
Geomean of speedups (65 kernels)	1.00	1.30	2.29	**2.32**

7 Future work

Multiple directions can be taken in order to build upon this work. One approach, would be to extended the interface with additional, domain specific operations. By forking the implementation and developing domain algorithms as MFI functions, and next specializing them for all necessary architectures the users can obtain additional simplifications for dedicated applications and/or computing environments.

Another approach would be do develop specialized support for other architectures. Additional support for both processors and coprocessors would result in developing uniform code basis which could work in heterogeneous environments, without need for multiple separate implementations.

The specialized implementation can be also provided by means of already existing, standard technologies such as OpenCL (vector types) [9] or OpenMP (**#pragma simd**) [4]. Both extensions are very appealing due to the fact that they offer code generation schemes without the need for architecture specific code development. Both standards are also widely recognized and for that reason the users can expect them to be supported by hardware vendors even if non-standard extensions (for instance intrinsics) become obsolete. While the longevity of these approaches can be ensured, the code generation quality relies on compilers and for that reason the control over kernels' performance cannot be guaranteed.

8 Summary

This paper describes a new API, known as UME::SIMD, which offers a flexible, portable, type-oriented abstraction for SIMD instruction set architectures. By designing a flexible interface it is possible to quickly add specialized operations, tuned for specific target architectures. Scalar emulation of all vector types creates a C++ compatible reference for comparing results obtained using non-standard intrinsic operations on different architectures. It also allows incremental implementation of SIMD types for new instruction sets, without breaking existing codes. Extending the set of types available in C++ with abstract vector types shows promising performance improvements for both scalar and vector microprocessors. Benchmarking results show that improvements can be achieved even for SIMD ranks other than native rank of underlying vector registers. This creates an opportunity for additional parametric tuning of applications.

By ensuring compatibility with a standard API across a variety of architectures, it is possible to achieve both compiler and hardware portability. Addition of SIMD-1 types gives the unique opportunity for comparison with efficiency of equivalent scalar code without the need for code duplication. Furthermore, by implicitly meeting vectorization requirements, SIMD-1 aids compiler auto-vectorization capabilities. For non-vectorizable workloads, SIMD-1 provides a necessary mechanism which allows reverting from SIMD execution to scalar execution mode.

By maintaining a uniform interface over all SIMD types, vector types can be used in generative programming techniques. These techniques can then be used for selection of optimal vector rank and precision in dedicated kernels of code. Such techniques can be used for fine tuning of code targeting different platforms. Such generative programming techniques are not easily implemented using intrinsic functions directly due to incoherent call conventions between different instruction sets.

We also presented extensive set of measurements showing the performance quality of the library in comparison to state-of-art VC library, and intrinsic kernels. In both cases the library offers simplification over the programming model without performance losses, and with increase in programmer productivity.

Acknowledgments

The authors would like to thank all the users for feedback without which no software could never reach its proper quality.

We would also like to thank all the reviewers for constructive feedback we could use to improve both the quality of this paper and the software presented here.

We would like to thank Intel Corporation for contiguous support in technical aspects required during preparation of our software, and for its involvement in the ICE-DIP project.

ICE-DIP is a European Industrial Doctorate project funded by the European Community's 7th Framework Programme Marie-Curie Actions under grant PITN-GA-2012-316596.

References

[1] J. Apostolakis, M. Bandieremonte, G. Bitzes, R. Brun, P. Canal, F. Carminatio, G. Cosmo, J. C. De Fine Licht, L. Duchem, V. D. Elviera, A. Gheata, S. Y. Jun, G. Lima, T. Nikitina, M. Novak, R. Sehgal, O. Shadura, and S. Wenzel. 2015. Towards a high performance geometry library for particle-detector simulations. In *Journal of Physics: Conference Series (Volume: 608, Number:1)*. IOP Publishing.

[2] ARM. 2007-2015. ARM Compiler toolchain, Version 5.0: Assembler Reference. (2007-2015). http://infocenter.arm.com

[3] R. Asai and A. Vladimirov. 2015. Intel Cilk Plus for complex parallel algorithms. In *Parallel Computing (Volume: 48, Issue: C)*. Elsevier, Amsterdam, The Netherlands.

[4] OpenMP Architecture Review Board. 2013. OpenMP Application Programming Interface. (2013). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[5] Intel Corp. 2010. A guide to Vectorization with Intel® C++ Compilers. (2010). https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

[6] P. Carini Dz-Ching R. Ju, Chuan-Lin Wu. 1994. The Classification, Fusion, and Parallelization of Array Language Primitives. In *IEEE Transcations on parallel and distributed systems, vol. 5, no. 10,*.

[7] P. Esterie, M. Gaunard, and J. Falcou. 2013. N3571: A Proposal to add Single Instruction Multiple Data Computation to the Standard Library. (2013). https://isocpp.org/files/papers/n3571.pdf

[8] A. Fog. 2016. VCL: C++ vector class library. (2016). http://www.agner.org/optimize/vectorclass.pdf

[9] Khronos OpenCL Working Group. 2008. The OpenCL Specification. (2008). https://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[10] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and Y. Yamazaki. 2006. Synergistic Processing in Cell's Multicore Architecture. In *Micro, IEEE (Volume: 26 Issue: 2)*. Los Alamitos, CA, USA.

[11] IBM. 2013. IBM Power ISA, Version 2.07. (2013). www.power.org

[12] ISO/IEC. 2010. ISO/IEC WD1539-1 Programming Languages - Fortran. (2010). http://www.j3-fortran.org/doc/year/10/10-007.pdf

[13] ISO/IEX. 2014. Information technology - Programming languages - C++ (ISO/IEC 14882:2014). International Standard Specification. (Dec. 2014).

[14] Kenneth E. Iverson. 1962. *A Programming Language*. Wiley.

[15] P. Karpiński. 2016. UME::SIMD benchmark results. (2016). https://bitbucket.org/edanor/umesimd/wiki/Microbenchmark%20results

[16] P. Karpiński. 2016. UME::SIMD library overview. (2016). https://bitbucket.org/edanor/umesimd

[17] M. Kretz. 2013. N3759: SIMD Vector Types, A proposal to C++ standardization Committee. (2013). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3774.pdf

[18] M. Kretz and V. Lindenstruth. 2012. VC: A C++ library for explicit vectorization. In *Software-Practice I& Experience (Volume: 42, Issue: 11)*. Wiley, New York, USA.

[19] A. Naumann and S. Wenzel. 2013. N3774: C++ Needs Language Support For Vectorization. (2013). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3774.pdf

[20] A. Peleg and U. Weiser. 1996. MMX Technology Extension to the Intel Architecture. In *Microm, IEEE (Volume: 16 Issue: 4)*. Los Alamitos, CA, USA.

[21] V. S. Adve R.L.Bocchino Jr. 2006. Vector LLVA: A Virtual Vector Instruction Set for Media Processing. In *VEE '06 Proceedings of the 2nd international conference on virtual execution environments*. ACM, New York, USA.

[22] B. Stroustrup. *The C++ Programming Language* (4th ed.). Addison-Wesley.