An Embedded Domain Specific Language for General Purpose Vectorization

Przemysław Karpiński[1,2(✉)] and John McDonald[2]

[1] CERN, The European Organization for Nuclear Research, 1211 Geneva 23, Switzerland
przemyslaw.karpinski@cern.ch
[2] Maynooth University, Maynooth, Co Kildare, Ireland

Abstract. Portable SIMD code generation is an open problem in modern High Performance Computing systems. Performance portability can already be achieved, however it might fail when user-framework interaction is required.

Of all portable vectorization techniques, explicit vectorization, using wrapper-class libraries, is proven to achieve the fastest performance, however it does not exploit optimization opportunities outside the simplest algebraic primitives. A more advanced language is therefore required, but the design of a new independent language is not feasible due to its high costs.

This work describes an Embedded Domain Specific Language for solving generalized 1-D vectorization problems. The language is implemented using C++ as a host language and published as a lightweight library. By decoupling expression creation from evaluation a wider range of problems can be solved, without sacrificing runtime efficiency.

In this paper we discuss design patterns necessary, but not limited, to efficient EDSL implementation. We also study specific scenarios in which a language-based interface can surpass procedural interfaces in both efficiency, portability and ease of use. In particular we demonstrate higher performance when compared with equivalent BLAS Level 1 routines.

Keywords: Vectorization · SIMD · EDSL · Performance · Portability · Programmability

1 Introduction

In this paper we present an Embedded Domain Specific Language (EDSL) for explicit vectorization. This work extends Unified Multi/Many-Core Environment (UME) [13] framework with the expression template based mechanism to provide an additional level of abstraction over different SIMD and SIMT architectures.

We start our discussion with an overview of current state-of-art, focusing on selected techniques and their usage. We also discuss problems arising from routine-based vectorization interfaces.

© Springer International Publishing AG 2017
J.M. Kunkel et al. (Eds.): ISC High Performance Workshops 2017, LNCS 10524, pp. 515–537, 2017.
https://doi.org/10.1007/978-3-319-67630-2_37

Next we present our abstract vector language and show how this type of abstraction can reduce overall code complexity, as well as improve code readability and make it easier to comprehend.

Then we discuss how such a language can be implemented for CPU-based computations, without the need for a costly, custom compilation toolchain. We also discuss specific problems of portable SIMD code generation.

The discussion is continued with a presentation of selected C++ techniques useful for solving issues of performance bottlenecks arising between frameworks and application codes. Specifically we present a technique for compile-time coalescence between user, and framework defined kernels.

We then present a concept of custom evaluation schemes, capable of handling more complex statement classes. This concept is necessary for providing high language expressibility without losses in performance.

The discussions described above are followed by a performance comparison of vector code kernels and their equivalent implementation using an EDSL approach.

Finally we discuss scenarios in which this approach might fail, as well as some of practical limitations flowing from the current C++ specification and compliant compilers.

The main contribution of this paper is a concept of decoupling expression graph creation and machine code generation scheme. We show that more than one evaluator class is required in order to handle arbitrary vector statements. We also present selected evaluation schemes for handling non-trivial expressions. A secondary contribution is a discussion of design patterns useful for both evaluator creation and for interaction between framework and end-user code. In particular we show that user defined expressions can be coalesced with external solvers thus improving machine code quality. We also investigate scenarios in which expression-based vector processing can be more efficient than routine based approaches, with simultaneous improvement in code readability and portability. All our considerations are demonstrated by UME::VECTOR, an open source library that provides an existing implementation of the vector language [11].

1.1 Prior Work

Evaluation of SIMD programming models performed by [17] showed that explicit vectorization gives the best performance when compared to compiler auto-vectorization. We presented a design for this approach [13] that makes the concept of an abstract SIMD vector more portable, and using masking as a primary mechanism for control flow. As we discuss further this approach for SIMD code generation has multiple drawbacks. In this paper, we explain how to overcome these difficulties, with minimal losses in expressibility.

The approach presented here is strongly based on the *expression templates* (ET) technique [18,20]. Work presented by Härdtlein et. al. [7] demonstrated an approach in which the expression templates can be either made easier to implement or faster in terms of runtime performance. In this work we simplify this idea

by implementing an ET generator which allows us to propagate design changes in ET without the need for manual code changes. This approach allows simplifications in ET design, without relying on complex template-metaprogramming techniques needed otherwise, and exploited by libraries such as boost::proto [15] and NT2 [2].

Creating *Embedded Domain Specific Languages* has been explained already in [8] but without considerations for performance. In [2] the authors deal with parallelization schemes for ET graphs, but the topic of efficient SIMD code generation is not explained there in detail. We discuss this topic and show what trade-offs are required between expressibility and ease of use when fine granularity of code generation is required.

Using expression templates for EDSL design for linear algebra package design, has been already demonstrated multiple times for instance in [6,21]. They both explore in detail the topic of user-interface for matrix-based computations but focus only on matrix computations, with more object-oriented approach for data storage and compute flow. We present a generic set of vector primitives suited to a wider class of array processing problems and discuss further situations where this programming model can improve portability and performance, and reduce software efforts.

1.2 Selected Problems

A basic motivation for this work comes from a practical situation that we observed in GeantV, a particle detector simulator developed at CERN [1]. The main goals of the project are to improve performance of simulations by exploiting multi-threading and vectorization capabilities of modern HPC systems. Since the High Energy Physics(HEP) community is largely fragmented in terms of the type of computational resources available, the framework has to retain very high portability. As a framework it is also expected to provide components that can be re-used for wide variety of fields including HEP, medical imaging, aeronautics and others, meaning that the interface flexibility is an important design issue.

For the GeantV project, a decision was made to use an explicit SIMD library for efficient machine code generation. The feasibility of this approach was already discussed in [13,17]. A problem arising from this solution is, that the framework code has to implement the iterative structure around the data sets, as presented in Listing 1.1. In the simplest case, the framework developers need to write both SIMD and scalar versions of the same kernel. An alternative for loop peeling is to only use data buffers of lengths that are multiples of hardware supported SIMD strides. Both approaches require additional effort from developers to either duplicate the functionality by providing both scalar and SIMD versions of the same kernel, or to make sure that data sets are padded properly, so that only the SIMD version of the kernel is required. As we explained before in [13] the problem of code duplication can be solved by extending the scalar typeset with support for a vector interface, where certain vector operations become identities for 1-element vectors. Code Listing 1.2 shows both a peel loop and a remainder loop implemented using a templated version of such a kernel. Thanks to compiler

optimizations, the codes generated by compilers are the same in both situations, with the latter one requiring only a single framework kernel implementation.

Listing 1.1. Loop peeling for correct explicit SIMD-ization. Peel loop and remainder loop require different kernels.

```
template<int SIMD_LENGTH>
void framework_func(float *input0,*input1,float *output,int LENGTH){
 int REMAINDER_OFF = (LENGTH/SIMD_LENGTH)*SIMD_LENGTH;

 for(int i=0;i<LENGTH;i+=SIMD_LENGTH) {
     SIMD_kernel<SIMD_LENGTH>(&input0[i],&input1[i],&output[i]); // Execute peel loop
 }
 for(int i=REMAINDER_OFF;i<LENGTH; i++) {
     scalar_kernel(input0[i],input1[i],&output[i]); // Execute remainder loop
 }
}
```

Listing 1.2. Loop peeling with SIMD-1. Both peel and remainder loops use the same kernel definition.

```
template<int SIMD_LENGTH>
void framework_func(float *input0,*input1,float *output,int LENGTH)
{
 int REMAINDER_OFF=(LENGTH/SIMD_LENGTH)*SIMD_LENGTH;

 for(int i=0;i<LENGTH;i+=SIMD_LENGTH) {
     // Execute peel loop
     SIMD_kernel<SIMD_LENGTH>(&input0[i],&input1[i],&output[i]);
 }
 for(int i=REMAINDER_OFF;i<LENGTH;i++) {
     // Execute remainder loop
     SIMD_kernel<1>(input0[i],input1[i],&output[i]);
 }
}
```

Even with the scheme described above, the framework has to provide a set of **SIMD_kernel** implementations, as well as as set of wrapper functions **framework_func** to expose a SIMD-agnostic interface to the end user. The user code would then make a series of invocations similar to one presented at Listing 1.3. In this kind of a situation, the user requests execution of specific fast kernels, developed as part of a domain specific framework. A potential performance problems arise in this situation. If the data buffers are big enough to exceed the cache size, the temporary data resulting from call to **framework_func_1** might be pushed out from cache towards slower memory, before a call to **framework_func_2** happens. In that case, data locality is not preserved, and therefore computational resources might not be utilised efficiently.

Listing 1.3. User and framework code interaction.

```
void user_func(float *input0, *input1, *input2, float *output, int LENGTH){
  float* tmp = new float[LENGTH]; // allocate a temporary buffer for intermediate

  framework_func_1(input0,intput1,tmp,LENGTH);
  framework_func_2(tmp,input2,output,LENGTH);

  delete[] tmp;
}
```

Similar scenarios can lead either to significant performance losses or to users developing custom kernels of code and effectively replicating work already performed by framework developers. In very optimistic scenarios, framework developers can design custom functions for instance to merge the functionality of functions **framework_func_1** and **framework_func_2**. Unfortunately this will

only happen when there is enough direct feedback from users to framework developers, when there is an existing business need to do so, and if it doesn't explode the size of framework code. In most situations such an approach cannot be used.

Listing 1.4. Scalar solver for 4-th order Runge-Kutta method.

```
// User-defined function to be passed
// to RK-4 method as the 'func' parameter
float user_func_scalar(float x, float y){
    return 5.0f*x*x/exp(x+y);
}
//...
float framework_RK4_solver_scalar(float x, float y, float dx, USER_FUNC &func){
  float halfdx=dx*0.5f;
  float k1=dx*func(x,y);
  float k2=dx*func(x+halfdx,y+k1*halfdx);
  float k3=dx*func(x+halfdx,y+k2*halfdx);
  float k4=dx*func(x+dx,y+k3*dx);
  return y+(1.0f/6.0f)*(k1+2.0f*k2+2.0f*k3+k4);
}
```

Another kind of problem can be visualised by an example shown in Listing 1.4. In this example, the framework implements a domain specific algorithm for calculating Runge-Kutta method. The problem that we can identify quickly, is that the user defined function **func** is not known at the time of framework development. For this reason the framework cannot assure the users that this function will be properly inlined to avoid excessive function calls, nor that it will be properly SIMD-ized, as the function defined by the user might not be subject to vectorization. An alternative would be to force the users to write their functions using an explicit SIMD library already exploited by the framework. An example of such interaction is presented in Listing 1.5. In this example the users need to be fully aware of the concept of SIMD computations. The direct benefit of this approach is that there is no performance penalty from SIMD under-utilisation, however the function might still not be inlined properly. In addition to that, the users still might need to implement a scalar duplicate of their function, to be used with the rest of their code. Also there is no guarantee that current explicit SIMD approaches will retain their portability over future SIMD hardware, forcing the users to write possibly less-portable code.

Listing 1.5. Explicit SIMD solver for 4-th order Runge-Kutta method.

```
// User-defined function to be passed to RK-4 method as the 'func' parameter
SIMD<float,8> user_func_SIMD(SIMD<float,8> &x, SIMD<float,8> &y){
  return 5.0f*x*x / (x+y).exp();
}
...
SIMD<float,8> framework_RK4_solver_scalar(SIMD<float,8> &x, SIMD<float,8> &y,
        float dx, USER_FUNC &func){
  SIMD<float,8> dx_vec(dx);
  SIMD<float,8> halfdx_vec(dx*0.5f);
  SIMD<float,8> k1=dx*func(x,y);
  SIMD<float,8> k2=dx*func(x+halfdx,y+k1*halfdx);
  SIMD<float,8> k3=dx*func(x+halfdx,y+k2*halfdx);
  SIMD<float,8> k4=dx*func(x+dx,y+k3*dx);
  return y+(1.0f/6.0f)*(k1+2.0f*k2+2.0f*k3+k4);
}
```

2 Vector EDSL Overview

Given the issues detailed in the previous section we would argue that there is a clear need for a more expressive way to communicate between user and

framework codes. We could imagine such communication happening by a user expressing an intent for a more complex aggregation of framework primitives (routines), and by frameworks making decisions as late as possible about the final machine code to be executed. This concept of *Lazy evaluation* is already being explored for higher level parallelism, for example in [10], however it cannot be applied for efficient code generation at instruction level. The main problem is the requirement for the higher-level code to be presented in a form that is *statically deductible*. That is, the decision about the specific instruction to be generated, must be made at the compile time. Hence, this *lazy code generation* applies currently at the level of low-level programming languages and is handled by compilers.

Development of a new language, and a corresponding compiler, is not a feasible solution for a lazy SIMD code generation, as it would require replication of work already done at the level of compiler toolchains and core libraries. An equivalent effort put into extension of C++ language and compliant compilers could bring more benefits, than re-designing a new language just to exploit this specific hardware feature. Recent developments in C++ language standard made it more feasible to use Expression Templates as a way to provide a library with compiler-like capabilities [19]. Specific meta-programming features, such as automatic type deduction, variadic templates, move semantics and constant expressions allow providing more static (compile-time) information to the compiler, enabling it to generate more efficient machine code. By operator overloading expression templates also allow creation of more intuitive interfaces.

In this paper we present UME::VECTOR which provides a C++ based implementation of EDSL dedicated for handling 1-dimensional vectors, focusing on efficient SIMD code generation. The language provides a set of types representing scalars and vectors of scalar elements, and a set of basic operations applicable to these types.

2.1 Typeset

Listing 1.6 shows basic declarations for terminal types. The basic requirement is made that the size of a vector needs to be passed at the latest moment of vector declaration. The rationale behind this requirement is, that the operations between vectors are possible only if specific requirements on vector lengths are correct in terms of the arithmetic operations to be executed. The fundamental type of packed elements is passed as a template parameter. This requirement is in line with standard C++ conventions, and it is driven by static deductibility requirement.

In the given example, vectors **a** and **b** are initialized using external memory locations owned and managed by user code. By *binding* the memory region to vector primitives it is possible to decrease both the memory footprint and execution time. Since the vector primitive does not own any memory location, no additional allocations have to be performed. If such an allocation would have to be handled, the data from the original location would still have to be copied

to new location, requiring significant amount of time if a specific computational kernel has to be executed repetitively.

In some cases the user might want to have a dedicated memory region, used for the storage of vectors, e.g. for temporaries. It is possible for the user to pass a specific allocator type to be used to handle specific memory region allocations. Since the language cannot make extensive assumptions about specific execution environment and target platform, the possibility to allocate memory in the specific memory regions, such as high bandwidth memory, is required. Since the method of allocation, or specific external tools required might differ depending on the user platform, it is up to the user to choose a proper allocator. The information about specific vector storage locations can be used by an *expression evaluator* to perform additional memory-based optimizations. At the same time a default allocation mechanism is provided to facilitate ease of use in the simplest scenarios.

Listing 1.6. Declaration of terminals.

```
float raw_a[1000];
int   raw_b[123];
bool raw_mask[1000];

UME::VECTOR::Vector<float> a(1000, raw_a);
UME::VECTOR::Vector<int> b(123, raw_b);
UME::VECTOR::Vector<float, userAllocator> c(1000);
                                        // Vector is responsible for memory management.
UME::VECTOR::Scalar<float> pi(3.14);
UME::VECTOR::Mask mask(1000, raw_mask);
```

Listing 1.6 presents also a declaration of a **Scalar<float>** type. Since C++ already provides a mechanism, for scalar declarations, the standard scalar types can be used in user defined formulas instead. Any C++ scalar variable and constant will be automatically converted to a corresponding **Scalar<>** type when used in such a formula. The main reason behind this wrapping of scalar types is that different semantic rules apply for C++ fundamental and non-fundamental types. Creating a scalar wrapper allows more uniform handling of 1-D vectors and scalars within a language implementation. The second reason is, that a wrapped scalar type is derived from the same interface as the *Vector* types and composite expressions. As a result the same invocation conventions and interfaces can be used to handle both scalar variables and vector variables within the library implementation. Awareness of the Scalar type might be important for handling some minor corner-cases in user code, and is critical for the situations when a custom, user-defined evaluator is developed. We will discuss the topic of evaluators in Sect. 3.

The last type in Listing 1.6 is a *Mask* type. A mask, or a predicate vector, is a vector of elements responsible for conditional evaluation of an expression. The concept has been already discussed in [13,14] with the rationale of masks already being an integral part of existing instruction sets [9,16]. The vector EDSL does not provide any block level control flow, such as **if-else** or **for** statements. The only way of providing an efficient handling of conditional executions is by the means of mask types. For the purpose of C++ compatibility, a mask vector should be considered as a vector of packed boolean variables used for selective execution of specified operations.

2.2 Syntax

The most natural way of providing language extensions in C++ can be done using the operator overloading mechanism. Overloaded operators offer the capability of changing the default meaning of supported unary and binary operators, and provide a custom evaluation scheme for a new operation. There are couple of issues that have to be overcome when dealing with operator overloading in terms of performance and expressibility.

First of all, an operator is essentially a function. For efficiency reasons, excessive function calls have to be avoided. In the case of the vector language each overloaded function relates roughly to a single CPU instruction, therefore function calls have to be completely avoided. C++ offers the **inline** keyword to inform the compiler that a given function (or an operator) should not generate a corresponding stack frame, however due to the fact that **inline** is only a hint, there is no guarantee on compiler behaviour. Luckily most of the compilers, including open-sourced GNU GCC and Clang++, support additional function attribute to force inlining. We use this non-standard keyword wrapped as a portable macro to pass our stronger intent to compilers.

The second problem is, that at the moment C++ only permits a limited number of operators to be overloaded. As we already pointed out in [13] the number of available operators is not sufficient for expressive SIMD vectorization. There is also no possibility to overload the ternary operator (<*mask* >**?** <*true-exp* >**:** <*false-exp* >), required for binary operations using the optional mask operand. This made it necessary to develop an alternative interface. We therefore use a *Member Function Interface* (MFI) to provide the user with a mechanism to express all operations with a uniform interface. At the same time, we also allow users to use the classical operator form to facilitate easier expressibility for operations for which it is possible. Listing 1.7 shows few examples of how the user can write down specific expressions. In case of MFI operations, the operand on the left of **.** operator is treated as an implicit operand.

As can be observed, some of the operations do not have a corresponding C++ operator. MFI offers a wider and more uniform interface. As we pointed out already in [13], the MFI function calls can be easily mapped to C-like functions without further losses in portability and performance. We reserve this type of language syntax for future library releases.

Listing 1.7. Syntactic conventions of vector language.

```
\\ Operator syntax
a=b+c;
\\ Masked syntax with MFI
a=b.add(mask,c);
\\ Ternary operation with MFI
a=b.fmuladd(c,d);
\\ Binary destructive addition (+=)
a.adda(b);
\\ Operations can be nested if necessary
c=b.add(a > 0, d * e);
```

Element-Wise Operations. The set of *arithmetic operations* consists of the ones already defined by the C++ standard but generalized for vectors of packed

scalars. As already mentioned most of the arithmetic operations accept required **Vector** and arithmetic expression operands, and return an arithmetic expression or, in case of comparison operations, a logical expression. Similarly all *logical operations* accept **Mask** and logical expressions and return a logical expression. Except for comparison instructions, all arithmetic operations accept an optional mask operand.

A subset of arithmetic operations called *destructive* operations allow the operation to modify one of the operands. A C++ equivalent of such operations would be to use assignment operators, such as '+=' or '\='. Since use of an assignment operator with a left hand vector is considered to be an *evaluation trigger*, that is an operation forcing expression evaluation, its use would prohibit nesting of destructive operations within expressions. From the performance perspective however, nesting these operation within composite expressions allow us to improve data locality. Therefore destructive operations need to be accessed using MFI interface, if they are meant to be used as parts of an expression. An example can be reviewed in Listing 1.8. In the second case of that listing, the destructive operation is performed on operand 'c' before its value is passed for evaluation of the rest of the expression. Since a destructive operation can only be applied to a proper l-value, a compile-time error will occur when the operation is applied on a r-value type.

Listing 1.8. Using destructive operations.

```
\\ basic destructive operation (/=)
a/=b;
\\ Nested destructive operation (+=)
\\ Both'a' and'c' are modified.
a = b + c.adda(d);
```

Control Flow. As mentioned before, masking is the only way to perform control flow in this language. An example of a masking operation has already been presented in Listing 1.7. For MFI functions, the optional mask operator is always the first parameter.

A mask can be either loaded by the user in the process of binding with a **bool** array, or obtained as a result of one of the arithmetic comparison instructions. The comparison operations can be expressed using either one of the relational operators, or an equivalent MFI function. The class of logical expressions does not accept an optional mask parameter, as it accepts and returns a mask parameter only. Masking of a logical operation can therefore be performed using an additional **.land** (Logical AND, or &&) operation.

When a masked operation is executed, its effect is applied only for the elements where the mask value was equivalent to 'true'. We don't specify how an implementation should treat the masks within an expression, as such assumptions might impact the performance on specific platforms. We only make a requirement on the final persistent result of the operation. In that sense, a masked operation has to operate 'as if' it was propagated towards the evaluation destination (left hand side of the"="operator) and through specific destructive

operations. This soft requirement offers optimization opportunities for platform specific evaluators' implementations.

Reduction Operations. A set of operations converting a vector type into a fundamental-castable type is called *a reduction operation*. At the same time applying a reduction operation on a **Scalar<T>** will be considered an identity operation. Reductions are an important class of basic problems, as they already have their reflection in existing instruction sets. On the other hand, reduction operations are not as trivial to parallelize as element-wise operations. Since a reduction operation requires traversal of all elements of a vector, or evaluation of sub-expression forming such vector, it might create a performance bottleneck. For that reason reductions might require a specialized implementation. A classical way of providing a serial reduction operation in C/C++ consists of iterating over an array of elements, and performing a partial reduction in each iteration. Implemented as such, reductions might require a small number of additional lines of code to be expressed.

By making basic reduction operations accessible using the MFI interface, it is possible to make the user code more compact, and easier to read. At the same time writing more complex reduction operations can be implemented using existing horizontal operations, and basic reduction operations. Listing 1.9 shows the example implementation of an infinity norm applied between two vectors.

A complete list of operations available as part of the language is subject to frequent changes, therefore we refer the reader to the implementation website [11] for further reading.

Listing 1.9. Using max-reduction to calculate infinity norm between two vectors.

```
// Inifinity norm calculation using vector EDSL:
err = ((a-b).abs()).hmax();
...
// The same intent expressed using scalar C++ code:
err = 0.0f;
for(int i=0;i<LEN;i++) {
  float diff=abs(a[i]-b[i]);
  if(diff>err) err=diff;
}
```

3 EDSL Implementation

While we don't limit the possibility of implementing our vectorization EDSL to any type of interpreted or compiled languages, it was designed primarily to be implemented using a library approach. As the implementation required has to be able to reach very high performance without sacrificing usability, we find it important to discuss specific design patterns and techniques used. Most of these techniques can be adopted to user codes to reach more flexible and efficient designs.

We find two existing design patterns to be critical for our design: *Expression Templates* (ET) and *Curiously Recurring Template Pattern* (CRTP). Both patterns are already well established and can be referred to in [18]. In our case

ET pattern is important, as it gives the ability to construct expression graphs with minimal overhead, and handle them using a lazy evaluation approach. The CRTP technique is used as a basis not only for ET creation, but also as a core technique for advanced patterns and for expression evaluator creation. Its biggest advantage is that it allows generation of machine codes specialized for specific expressions.

3.1 Additional Design Patterns

We would like to present few additional design patterns that show flexibility of the embedded language, and its compiler-like nature. We discuss these patterns on simple examples, however we would like to point out that the their applicability is not limited to such.

Static Expression Visitor Pattern. A visitor pattern, such as described in [4] is useful for recursive traversal of a tree-like graph. The visitor pattern has the advantage of being separate from the graph structure definition and allows both introspection and modification of graphs. Since the ET pattern creates a static graph, there is no need for virtual function dispatch. Instead, the visitor class takes the form of a template class with the type of expression treated as a specialization parameter. Such a functor might still need to perform certain operations at runtime as some information, such as exact memory locations, is not available at compilation time. Because the graph traversal order is known at compile time **visit** methods can be inlined, possibly decreasing the runtime overhead.

Listing 1.10 shows an example of *Static Expression Visitor* pattern with the purpose of printing a specific instance of an expression. We found this technique particularly useful when debugging EDSL code, as mangled names for nested types are difficult to analyse.

Static Transformation Pattern. The Static Expression Visitor pattern, can be further used to implement *Static Transformations* of expressions. We don't provide a detailed exploration of the requirements here, or an effective complex implementation of this pattern, but only show that a basic variation can be constructed and applied easily.

In the example given in Listing 1.11, an expression **A*B** is being transformed into an expression **A+B**. As the traversal happens using type recursion, it is possible to apply this pattern for a complex expressions, to replace all occurrences of a given expression structure with a different one. The transformation happens at compilation time so no runtime overhead is introduced.

Listing 1.10. Expression printing is a simple way to debug ET code.

```
template<typename EXP>
class ExpressionPrinter{
public:
  // Construct the visitor from
  // specific expression instance
  ExpressionPrinter(EXP exp){ visit(exp); }
  ...
  // Visit a terminal
  template<typename SCALAR_T>
  FORCE_INLINE void visit(FloatVector<SCALAR_T> exp){
    std::cout<<"Vector("<<exp::LENGTH()<<")"<<&exp.elements[0]<<"\n";
  }
  ...
  // Recursively print ADD expression
  template<typename SCALAR_T, typename E1, typename E2>
  FORCE_INLINE void visit(ArithmeticADDExpression<SCALAR_T,E1,E2> exp){
    std::cout <<"ADD:\n";
    visit(exp._e1); // Visit children
    visit(exp._e2);
  }
  ...
};
...
// Print expression
ExpressionPrinter printer(myExpression);
...
```

Listing 1.11. An example on how to transform one expression into another.

```
//Replace a MUL(E1,E2) node
//with an ADD(E1,E2) node
template< typename SCALAR_TYPE, typename E1, typename E2>
FORCE_INLINE ArithmeticADDExpression<SCALAR_TYPE,E1,E2>
  transform(ArithmeticMULExpression<SCALAR_TYPE,E1,E2> exp)
{
  // Construct a replacement expression using sub-expression nodes of 'exp'
  return ArithmeticADDExpression<SCALAR_TYPE,E1,E2>(exp._e1,exp._e2);
}
...
// Call transformation on
float a[10],b[10];
Vector<float> A(10,a), B(10,b);
auto t0=A*B;
auto t1=transform(t0); // t1 is now 'A+B'
...
```

Static Expression Coalescence Pattern. Certain scenarios of interaction between user and framework codes such as Runge-Kutta method described in Sect. 1.2 can now be solved effectively using vector EDSL. By using the *Static Expression Coalescence* pattern, a generic solver provided by a framework can be specialized for a specific user defined function.

In Listing 1.12 we show an implementation together with an invocation of a RK-4 solver. The **auto** keyword used on input parameters of the solver makes it possible to pass either specific scalar or vector expression types. In the case of the former, the behaviour would be the same as if the solver was defined using scalar code similar to one from Listing 1.4.

If the parameters passed as **x**, **y** are of the EDSL types then instead of carrying in-place computations, such as calls to the function **func**, a static graph is created. This graph treats the user function as a structure to be merged into a full computational graph, meaning that both the framework code, and user code become *coalesced* into a single vector EDSL expression. As the language can then apply lazy code generation for the fully coalesced expression, the resulting code can be vectorized and inlined more effectively.

Two minor drawbacks of this design pattern exist at present. First of all, the constructions used require *generalized return type deduction* features available

as of C++14. This might delay the introduction of this design pattern into popular frameworks relying on older language standards. Second, a contractual agreement needs to exist between framework and user code to use vector EDSL, or its specific dialect. While this can be easily achieved for framework codes, additional user education might be required.

Listing 1.12. Static Expression Coalescence pattern merges user function written using Vector EDSL with framework-defined solver.

```
template<typename USER_FUNC_T>
void rk4_framework_solver(auto & result, auto x, auto y, float dx,
                          USER_FUNC_T& func) {
  float halfdx=dx*0.5f;
  auto k1=dx*func(x,y);
  auto k2=dx*func(x+halfdx,y+k1*halfdx);
  auto k3=dx*func(x+halfdx,y+k2*halfdx);
  auto k4=dx*func(x+dx,y+k3*dx);
  result=y+(1.0f/6.0f)*(k1+2.0f*k2+2.0f*k3+k4);
                                            // Evaluation starts with this statement
}
...
// User defined function has to be defined using the same Vector EDSL dialect.
auto userFunction=[](auto X, auto Y){
  return X.sin()*Y.exp();
};
...
// User passes her function to solver
rk4_framework_solver(result_vec, x_exp, y_exp, timestep, userFunction);
...
```

An obvious benefit of this approach is that it greatly simplifies complexity of both user and framework code. A specific solver is described as a hardware-agnostic kernel which can be treated differently by the language depending on target architecture. The same observation applies for user codes as the user is no longer required to write architecture specific SIMD code, using for instance an explicit vectorization approach. The same user-defined function can be used for graph coalescing, as well as directly within the user code, meaning that no unnecessary code replication happens.

3.2 Evaluators

As we have explained, the vector EDSL is used to construct a static graph of vector operations. This graph stores the relation between nodes representing specific vector operations, and vector terminals. The construction of a graph is a process taking place at compile-time. At the same time we want to create a kernel of code, preferably using SIMD instructions, and responsible for evaluation of a given expression depending on specific run-time terminals. For performance reasons, construction of such kernel should follow the lazy code generation principle, and for that reason has to be also carried at a compile-time.

Default Evaluators. We described previously, that the evaluation of a specific expression is triggered when an assignment operator = is used with a LHS expression being either of **Vector** or **Scalar** type, and with RHS being a valid vector EDSL expression. We call this evaluation method a *default evaluator*. The default evaluator is an integral part of current implementation and is provided together with ET classes. The evaluation is triggered by **Vector::operator=** implementation, as presented in Listing 1.13. This scheme splits the execution of

a vector expression into two loops similar to ones from Listing 1.1. In each loop a recursive evaluation of the expression, for a given dataset offset is triggered, and the result is written to the data array representing LHS vector.

Each expression class defines **evaluate_SIMD** method (Listing 1.14), responsible for generating instructions corresponding to the specific expression semantics. The evaluation method is forced to be inlined as, in most cases, the actual code is limited to only a few machine instructions. Depending on the number of arguments of the expression and its additional semantic meaning, the method calls evaluation methods of sub-expressions.

Listing 1.13. Default evaluator uses very straightforward evaluation scheme. Instead of traversing a vector in the data direction (horizontally), depth-first (vertical) traversal of the full expression is performed. The 'elements' pointer refers to the memory location represented by an instance of 'FloatVector' type.

```
template<typename E>
UME_FORCE_INLINE FloatVector<SCALAR_TYPE>&
                              operator= (ArithmeticExpression<SCALAR_TYPE,E>& vec){
  E & reinterpret_vec=static_cast<E&>(vec);

  // SIMD_STRIDE - a target specific library macro
  for(int i=0;i<LOOP_PEEL_OFFSET();i+=SIMD_STRIDE){
    auto t0=reinterpret_vec.evaluate_SIMD<SIMD_STRIDE>(i);
    t0.store(&this->elements[i]);
                              // t0 needs to be a type respecting UME::SIMD interface.
  }
  for(int i=LOOP_PEEL_OFFSET();i<mLength;i++){
    auto t1=reinterpret_vec.evaluate_SIMD<1>(i);
                              // Evaluate remainder part using SIMD-1 (scalar) mode.
    t1.store(&this->elements[i]);
  }
  return *this;
}
```

Listing 1.14. Evaluation method can use a depth-first approach to calculate dependencies.

```
template<int SIMD_STRIDE>
UME_FORCE_INLINE SIMDVec<SCALAR_T,SIMD_STRIDE> evaluate_SIMD(int index){
  SIMDVec<SCALAR_T,SIMD_STRIDE> t0=_e1.evaluate_SIMD(index);
                                                    // Evaluate subexpressions
  SIMDVec<SCALAR_T,SIMD_STRIDE> t1=_e2.evaluate_SIMD(index);
  return t0.add(t1); // Evaluate current expression node
}
```

Custom Evaluators. The scheme just described is useful only in basic cases, when the left-hand destination is an explicit terminal. When the destination is an implicit operand, for instance when the last operation is a destructive operation, an alternative trigger mechanism must be provided. The **operator** = trigger can be generalized by providing an external class with a specific evaluation scheme, dedicated to handling a specific statement form. Because of that there is no explicit LHS operand to be used to trigger the evaluation. A similar situation will also happen when the last operation is a reduction operation.

Listing 1.15. Monadic evaluator definition. Stores are removed, as they will be carried as side-effects of **evaluate_SIMD** calls.

```
class MonadicEvaluator {
 ...
 // Evaluate expression with an implicit destination
 template<typename SCALAR_TYPE, typename EXP_T>
 FORCE_INLINE MonadicEvaluator(ArithmeticExpression<SCALAR_TYPE,EXP_T>& exp){
   EXP_T& r_exp=static_cast<EXP_T&>(exp);

   for(int i=0;i<r_exp.LOOP_PEEL_OFFSET(); i+=SIMD_STRIDE){
    r_exp.evaluate_SIMD<SIMD_STRIDE>(i); // implicit operand is updated automatically
   }
   for(int i=r_exp.LOOP_PEEL_OFFSET();i<r_exp.LENGTH();i++) {
     r_exp.evaluate_SIMD<1>(i);
   }
 }
 ...
};
...
// user code uses destructive operation:
auto t0=a.adda(b);
// user triggers evaluation manually
MonadicEvaluator eval(t0);
```

The example of a *generalized monadic evaluator* is presented in Listing 1.15. A monadic evaluator is responsible for evaluating an expression with only one, possibly implicit, destination operand. In the scheme presented, no explicit **store** operations occur, as they are carried out as a side-effect of the destructive operation evaluation.

Listing 1.16. Expression divergence happens when two expressions share a common sub-expression. This problem can cause memory locality issues, but can be solved with a very simple evaluation scheme.

```
auto t0=A+B;
auto t1=C+D;
auto t2=t0*t1;
E=t2*F;
G=t2*H;
```

Listing 1.17. Dyadic evaluator calculates both expressions before updating destination values. This way data hazards are avoided.

```
class DyadicEvaluator {
public:
 ...
 // Evaluate a pair of expressions simultaneously
  template<typename SCALAR_T_1,typename DST_T_1,typename EXP_T_1,
           typename SCALAR_T_2,typename DST_T_2,typename EXP_T_2>
  DyadicEvaluator(
    DST_T_1& dst1, ArithmeticExpression<SCALAR_T_1,EXP_T_1>& exp1,
    DST_T_2& dst2, ArithmeticExpression<SCALAR_T_2,EXP_T_2>& exp2)
  {
    EXP_T_1& r_exp1=static_cast<EXP_T_1&>(exp1);
    EXP_T_2& r_exp2=static_cast<EXP_T_2&>(exp2);

    for(int i=0;i<dst1.LOOP_PEEL_OFFSET(); i+=SIMD_STRIDE){
      auto t0= r_exp1.evaluate_SIMD<SIMD_STRIDE>(i);
                                                      // evaluate multiple results at a time
      auto t1= r_exp2.evaluate_SIMD<SIMD_STRIDE>(i);
      dst1.update_SIMD(t0, i);
      dst2.update_SIMD(t1, i);
    }
    for(int i=dst1.LOOP_PEEL_OFFSET();i<dst1.LENGTH();i++){
      auto t0= r_exp1.evaluate_SIMD<1>(i); // evaluate single result at a time
      auto t1= r_exp2.evaluate_SIMD<1>(i);
      dst1.update_scalar(t0, i);
      dst2.update_scalar(t1, i);
    }
  }
};
...
auto t0=A+B;
auto t1=C+D;
auto t2=t0*t1;

DyadicEvaluator eval(E,t2*F,G,t2*H); // Evaluation trigger
```

Non-monadic Evaluators. A more complicated scenario, when the default evaluator cannot be used is when *expression divergence* occurs. In the example in Listing 1.16 the sub-expression **t2** is calculated twice: once for statement **E=t2*F** and once for statement **F=t2*H**. In both cases both sub-expressions **t0** and **t1** require accessing all data fields of **A**, **B**, **C** and **D**. This might have a serious performance impact when operating on long vectors, as data locality will not be preserved.

Listing 1.18. Main loop of DyadicEvaluator generated by Clang++. The assembly code is very close to expected.

```
.LBB019:
  vmovups ymm0,ymmword ptr[rbx+4*rdx]        # A
  vaddps  ymm0,ymm0,ymmword ptr[rsi+4*rdx]   # t0=A+B
  vmovups ymm1,ymmword ptr[rdi+4*rdx]        # C
  vaddps  ymm1,ymm1,ymmword ptr[rcx+4*rdx]   # t1=C+D
  vmulps  ymm0,ymm0,ymm1                     # t2=t0*t1
  vmulps  ymm1,ymm0,ymmword ptr[r14+4*rdx]   # t3=t2*E
  vmulps  ymm0,ymm0,ymmword ptr[r12+4*rdx]   # t4=t2*F
  vmovups ymmword ptr[rbp+4*rdx],ymm1        # G=t3
  vmovups ymmword ptr[r15+4*rdx],ymm0        # H=t4
  add rdx,8
  cmp rdx,rax
  jl  .LBB019
```

By defining a *Dyadic Evaluator*, such as presented in Listing 1.17, we can improve the data locality of such divergent expressions by a mechanism that triggers evaluation of both of them simultaneously. With such an evaluation scheme, any data reads on input vectors are local, as the expression evaluation is carrying the same index localization to both expressions. In addition a capable compiler, such as Clang, is able to remove recursive function calls, and reorder operations in such a way that common dependencies are executed only once. Listing 1.18 shows the optimized loop for the dyadic evaluator compiled for an AVX2 instruction set. Because a specific instance of evaluator is specialized for a specific expression, generated code can be highly specialized.

3.3 Language Extensibility

As with every language, there are certain limitations for both expressibility and performance. By its nature, a DSL should offer users the ability to adopt it for specific scenarios required within the computational domain.

By making the language embedded, it is possible to extend it with user defined operations, without the need to re-design the language from scratch. The process of extension can be achieved in two ways. The first approach is to design a functor composed with basic operations and provide more compact notation for the user code. An example can be viewed in Listing 1.19. This mode of extension is the advised mode, as it is similar to already known paradigms of functional programming and, except for a few syntactic differences, is as easy to work with, as regular C++ functions. The second method of extending the functionality of the EDSL is to provide custom expressions and specific evaluation schemes for these expressions. The drawback of this method is that it might require modifications to all evaluators used by the user code. A most obvious benefit of this solution is that the user can express precisely the meaning of such scheme and reach potentially higher performance for specific usage scenarios.

Listing 1.19. Infinity L_∞ norm functor.

```
auto inf_norm(auto a, auto b) {
    return ((a-b).abs()).hmax();
}
...
auto c = inf_norm(a, b);
```

The default evaluation scheme works only for platforms that can be supported under the UME::SIMD typeset. For other targets, a separate implementation would have to be provided to carry out computations using specific language extensions or techniques. Providing additional evaluators does not require modification to either EDSL-based expressions, nor to the expression-based code. The only modification required might be the evaluation trigger invocation.

At the same time a number of particular cases, which cannot be predicted at the moment of language design, might appear for specific domains or even expression groups. The users are given the ability to design additional evaluation schemes that might accelerate the evaluation of their codes, without the need to re-write user or framework based algorithms.

4 Performance Evaluation

Performance evaluation of vectorization techniques poses multiple issues. Various compiler optimizations such as auto-vectorization, inlining and constant folding/propagation can affect the results obtained. As compilers evolve, we can also expect performance improvements on the same benchmarking target and configuration. Selection of compiler flags can also affect the results, as some unsafe optimizations, such as *fast-math* [3] offer significant speedups with the cost of decreased accuracy. As different compilers offer different sets of compiler flags, improper selection of flag configuration might result in an unfair comparison.

At the same time incorrect benchmarking methodology can lead to results which do not reflect the actual computational problem. A simple, yet not uncommon, example is when the results of computation is not used in any way within the benchmarking application. In such case compilers can generate code that carries incorrect or incomplete computations.

Each computational kernel might depend on specific compile-time and runtime parameters, as well as on data with or without specific distribution. Different algorithms/implementations can perform differently based on given parametrization. It is therefore required to verify specific implementations for a whole range of input parameter values.

4.1 Benchmarking Methodology

To follow the spirit of scientific method, we developed a set of benchmarks that allow easier comparison of different approaches in both the performance, and expressibility. All benchmarks are available as a part of the UME framework and can be accessed online [12]. Due to large number of possible combinations,

we only present a few selected benchmarks here, and discuss both qualitative and quantitative aspects of the EDSL approach. As it is difficult to find universal metrics to assess expressibility of a language, we reserve additional space for discussion of why an EDSL approach is easier to operate with the context of such benchmarks.

We limit the discussion to a single platform (Intel Xeon E3-1280v3, Haswell architecture, 16GB of DDRAM, running SLC6 operating system), as the qualitative differences would only be an effect of the different system software stack and the underlying explicit SIMD implementation. The platform we used is dedicated for benchmarking purposes and was not used for any other purpose during each benchmarks' execution. We used linux *top* command to determine least used core and pinned each benchmark execution to that core using the *taskset* command.

We built each benchmark using selected toolchains and equivalent compile-time configuration ($-$O2, AVX2 enabled, no fast-math). In each execution of a benchmark, we ran each implementation of the benchmark multiple times, calculating average of all runs. We also ran each benchmarking application multiple times, averaging results from each run. The reason for this approach is that interlacing of different implementation executions distributes noise uniformly over all configurations. The specific number of repetitions is defined separately for each benchmark, as the memory and execution time requirements vary.

Each benchmarking code is written carefully, so that compilers couldn't remove or reorder measurement-sensitive fragments of the code. A specific technique we used was to place fragments of benchmarked code within a function marked with the **never-inline** attribute (actual mapping varies for different compilers). Such a function is placed between two calls to a stopwatch using *std::chrono*. The time is measured with nanosecond precision.

For all benchmarks, a mandatory verification step is performed which serves two purposes. Firstly, the numerical correctness of an implementation is verified. We don't impose any limit on how accurate a specific implementation should be, but we rather treat this as a measure of performance orthogonal to execution time. Secondly, the verification steps disallow compilers to generate code carrying incorrect computations and possibly generating fake time measurements.

4.2 Runge-Kutta Solver

In each case the user function is available as a lambda function, defined within the benchmarked routine as: $x^2 + y$. The solver is defined outside the benchmarking code, as a templated function with the approach defined previously.

Results of the Runge-Kutta benchmark can be reviewed in Table 1. All values are shown as speedup versus scalar code compiled with GCC. Numbers separated with '/' are for single and double floating point precision, respectively. There are two important points to note here. Firstly, out of the three compilers used, only Clang was able to auto-vectorize the code efficiently. This suggests that this code could be, but is not, auto-vectorized by other compilers. For Clang the performance obtained with an explicit SIMD approach is almost the same as for

the scalar code. Secondly, for all configurations the majority of the best results, are reached using vector EDSL, with the explicit approach being second-best. Furthermore, the best performance obtained with GCC and ICPC is, in general, higher than for Clang.

Table 1. Speedup of different implementations of RK4 solver vs. reference. Values given for single/double precision. Only Clang gives comparable results with auto-vectorization. Highest performance obtained with explicit SIMD and EDSL in all cases.

Problem size	1	10	10^2	10^3	10^4	10^5	10^6	10^7	Geomean
GCC 5.2									
Scalar	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00
UME::SIMD	0.97/0.98	2.66/2.43	5.72/3.72	7.33/3.09	6.39/3.26	6.13/3.17	6.12/2.68	5.78/2.99	4.43/2.63
UME:: VECTOR	0.96/0.98	2.75/3.50	6.37/4.47	7.39/3.44	7.38/3.76	7.27/3.61	7.01/2.93	6.43/3.34	**4.84/3.02**
ICPC 17.0									
Scalar	1.29/1.78	1.58/1.98	1.58/1.53	1.58/1.77	1.65/1.76	1.65/1.76	1.66/1.90	1.63/1.60	1.57/1.75
UME::SIMD	1.17/1.60	2.85/3.61	8.27/5.74	10.89/4.67	8.85/5.04	8.73/4.80	8.68/3.84	7.07/3.69	5.88/3.9
UME:: VECTOR	1.56/2.16	2.70/4.17	8.21/6.11	9.56/4.66	8.95/4.94	8.77/4.72	8.66/3.66	6.87/3.60	**5.94/4.1**
Clang 3.9									
Scalar	1.01/1.21	2.66/2.54	6.01/2.66	7.40/1.92	6.83/3.20	6.76/3.25	6.55/3.78	6.18/3.32	4.66/2.6
UME::SIMD	0.97/1.19	2.69/2.63	6.06/4.40	7.18/2.58	6.88/3.15	6.68/3.22	6.53/3.52	5.89/3.06	4.6/2.81
UME:: VECTOR	0.98/1.15	2.76/3.15	6.13/4.41	7.21/2.88	7.15/3.20	6.90/3.29	6.70/3.28	5.86/3.09	**4.68/2.89**

4.3 BLAS Kernels

We will now present a comparison of three selected BLAS-based kernels, with a very straightforward implementations obtained using vector EDSL. As the nature of the EDSL presented is to operate on vector primitives and not matrices, we only compare vector-vector operations.

Some works such as Eigen [5] show comparison for kernels consisting of a single invocation of a BLAS primitive. A similar comparison for BLAS AXPY kernel performance is presented in Fig. 1 (a) & (b). Performance of vector EDSL is similar to that of the BLAS implementation, and compiler-optimized scalar code.

Rarely a single kernel is all we need to execute in a complex algorithm. We therefore defined a second benchmark, which consists of a chained execution of 10 AXPY kernels, with each of them being dependant on results of the previous one. Results for this variant are presented in Fig. 1 (c) & (d). The second variant shows an interesting property of kernel-based computations: operation atomicity breaks potential for data-locality based optimization. For high problem sizes UME configurations are up to 2x faster than the BLAS implementation. At the same time this potential does not seem to be exploited by the compilers when dealing with scalar codes. This performance gap can be only exploited with an expression-based interface, as it requires information about a broader computational context.

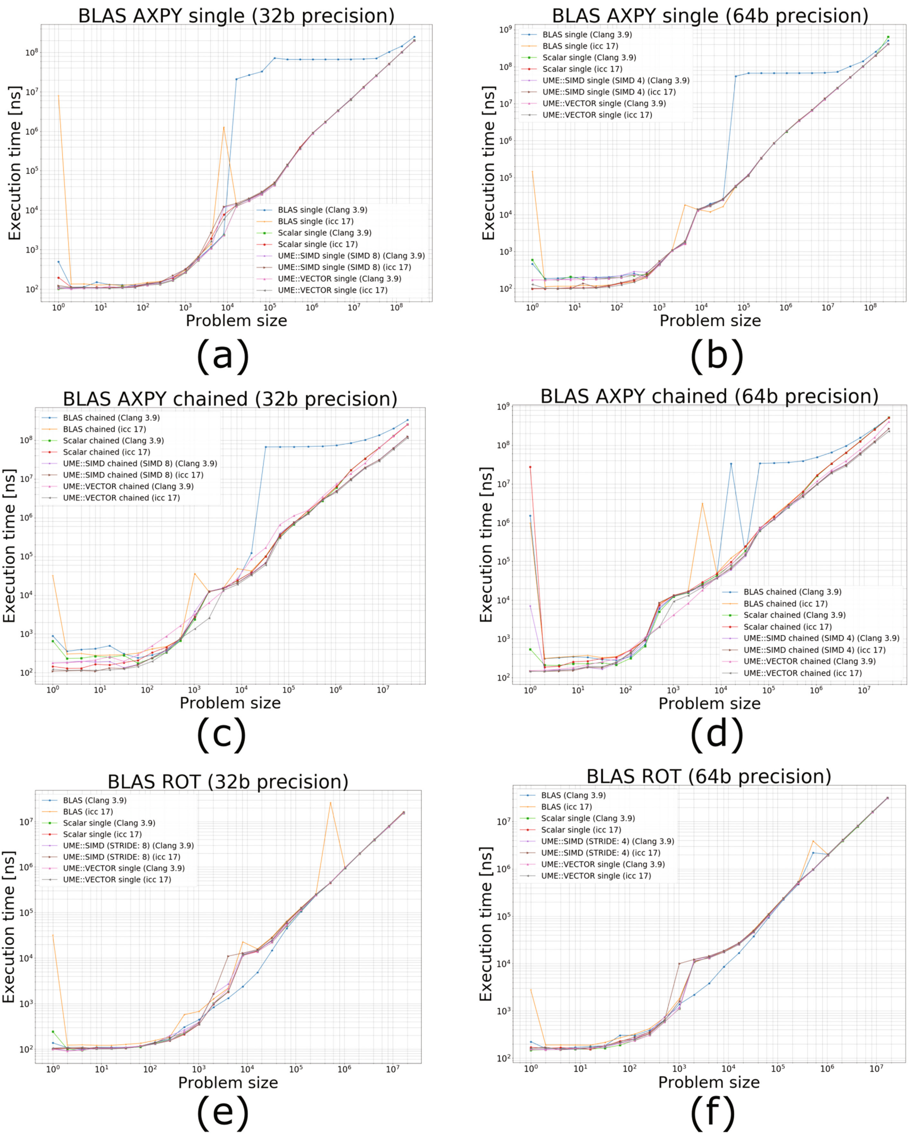

Fig. 1. Blas comparison benchmarks. Clang configuration uses OpenBlas, ICC uses MKL. (a) and (b) show results of a single AXPY kernel execution for 32b and 64b precision. Vector EDSL (UME::VECTOR) does not differ significantly from current technologies. (c) and (d) show that both UME::SIMD and EDSL are faster when solving complicated vector expressions. In (e) and (f) we show that there is no performance penalty when evaluating multi-statement expressions.

Given that AXPY is a very straightforward kernel and might put into doubt the actual expressibility of the language, we also present results for BLAS ROT kernel (Fig. 1 (e) & (f)). In this kernel a pair of variables is updated simultaneously, and depend on previous values of each variable. This makes it impossible to evaluate such a scheme as a single expression. It is also not possible to serialize both expressions, as the results would be invalid. We therefore construct two expressions and then use a dyadic evaluator to perform simultaneous evaluation. Results show that there is no performance degradation and no losses in language expressibility.

5 Practical Limitations

While we already showed, that with modern C++ techniques EDSLs can be a very powerful mechanism however, we would like to briefly point at certain limitations of this technique. Identification of these limitations is necessary for future developments of both EDSL, and its host language.

One of the most important limitations is the fact, that type-based expressions cannot be manipulated during program runtime. This limitation comes from the fact that usual machine code generation cannot happen at runtime.

Another limitation is the possibility of inefficient code generation in cases when a specific vector is used more than once in expression evaluation. Pointer aliasing might not be recognized and as a result some amount of repetition of code might appear, leading to suboptimal performance.

Last but not least, an optimal evaluator for a given class of statements might be difficult to create. The same expression can have more than one optimal evaluation scheme, depending on specific runtime-data and target platforms. This limitation might prohibit creation of very complex expressions and in turn lead users to revert to non-portable coding techniques.

6 Conclusions and Future Work

We have presented an EDSL for explicit vectorization. The language allows high-performance operations to be carried on 1-D vectors and scalars. We have shown that the SIMD programming model can be simplified, compared to an explicit SIMD approach, without a need for any compiler toolchain extensions. Furthermore we showed that in certain situations an expression-based approach can make better use of memory locality, leading to performance improvements over kernel-based interfaces such as BLAS.

The construction of an EDSL can be difficult, especially when performance is of highest importance. We presented a study of specific design patterns required for an effective EDSL implementation, as well as discussion of selected problems related to user-code. We presented a concept of separation between expression graph creation and evaluation, which allows solving more general classes of computational problems.

For future work we predict two directions: investigation of possible performance improvements for matrix expressions, and generalization of the concept of evaluators, so that arbitrary classes of vector statements could be handled. We hope to also investigate the possibility of JIT compilation given that it might allow building a dynamic language representation, further improving performance of expression evaluators.

References

1. Apostolakis, J., Bandieremonte, M., Bitzes, G., Brun, R., Canal, P., Carminati, F., Cosmo, G., De Fine Licht, J.C., Duchem, L., Elviera, V., Gheatea, A., Jun, S.Y., Lima, G., Nikitina, T., Novak, M., Sehgal, R., Shadura, O., Wenzel, S.: Towards a high performance geometry library for particle-detector simulations. J. Phys. Conf. Ser. 608(1) (2015). IOP Publishing
2. Falcou, J., Sérot, J., Pech, L., Lapresté, J.-T.: Meta-programming applied to automatic SMP parallelization of linear algebra code. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 729–738. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85451-7_78
3. Free Software Foundation: GNU GCC reference: Semantics of Floating Point Math in GCC. https://gcc.gnu.org/wiki/FloatingPointMath. Accessed 27 Mar 2016
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Reading (1995). ISBN:0-201-63361-2
5. Gunnabaus, G., Jacob, B., et al.: Eigen benchmarks website: http://eigen.tuxfamily.org/index.php?title=Benchmark. Accessed 27 Mar 2016
6. Gunnabaus, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org. Accessed 27 Mar 2016
7. Härdtlein, J., Pflaum, C., Linke, A., Wolters, C.H.: Advanced expression templates programming. Comput. Vis. Sci. **13**, 59–68 (2010). ISBN:1432-9360
8. Hudak, P.: Building Domain-Specific Embedded Languages. ACM Comput. Surv. 28 (1996)
9. Intel Corporation: Intel®64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Accessed 27 Mar 2016
10. Kaiser, H., et al.: HPX V0.9.99: A general purpose C++ runtime system for parallel and distributed applications of any scale, July 2016. https://zenodo.org/record/58027
11. Karpiński, P.: UME:: VECTOR: Vectorization EDSL library. https://github.com/edanor/umevector. Accessed 27 Mar 2016
12. Karpiński, P.: UME: Unified Multi/Many-Core Environment. https://github.com/edanor/ume. Accessed 27 Mar 2016
13. Karpinski, P., McDonald, J.: A high-performance portable abstract interface for explicit SIMD vectorization. In: PMAM 2017 (2017). ISBN: 978-1-4503-4883-6
14. Kretz, M., Lindenstruth, V.: VC: A C++ library for explicit vectorization. Softw. Pract. Experience **42**(11), 1409–1430 (2012). Wiley
15. Niebler, E.: Proto: A compiler Construction Toolkit for DSELs. In: LCSD 2007. ACM, October 2007. ISBN 978-1-60558-086-9
16. Petrogalli, F.: A sneak peak into SVE and VLA programming. https://developer.arm.com/hpc/a-sneak-peek-into-sve-and-vla-programming. Accessed 27 Mar 2016

17. Pohl, A., Cosenza, B., Mesa, M., Chi, C., Juurlink, B.: An evaluation of current SIMD programming models for C++. In: WPMVP 2016. ACM, March 2016. ISBN 978-1-4503-4060-1
18. Vandevoorde, D., Josuttis, N.: C++ Templates: The Complete Guide. Addison-Wesley, Boston (2002). ISBN:0-201-73484-2
19. Veldhuizen, T.: Blitz++: The library that thinks it is a compiler. In: Langtangen, H.P., Bruaset, A.M., Quak, E. (eds.) Advances in Software Tools for Scientific Computing. Lecture Notes in Computational Science and Engineering, vol. 10, pp. 57–87. Springer, Heidelberg (2000)
20. Veldhuizen, T.: Expression Templates. C++ Mag., June 1995. ISSN:1040–6042
21. Veldhuizen, T., Ponnambalam, K.: Linear algebra with C++ template metaprograms. Dr. Dobb's J. Softw. Tools (1996)