Gaalet Tutorial

Florian Seybold
High Performance Computing Center
University of Stuttgart, Germany
seybold@hlrs.de

ABSTRACT

*G*eometric *a*lgebra *a*lgorithm *e*xpression *t*emplates (*Gaalet*) is a C++ Expression Templates Library allowing for an easy implementation of Geometric Algebra (GA) expressions in C++, offering reasonable speed for the numerical expression evaluation. In this tutorial the usage of Gaalet is exemplarily shown, for both microprocessor and CUDA architectures. An insight into Gaalet's functioning is delivered, and some evaluation speed comparisons between Gaalet and handwritten implementations are given.

Keywords: tutorial, gaalet, geometric algebra, expression templates, C++, library.

```
auto S = (e1^e2)*(0.5*M_PI);
auto R = exp(-0.5*S);

auto a = 1*e1 + 2*e2 + 3*e3;
auto b = eval(grade<1>(R*a*(~R)));
```

Figure 1: Example code for Gaalet usage. Vector a is rotated using rotor R resulting in vector b. Rotor R is determined by bivector S, defining a rotation in the $e_1 \wedge e_2$-plane with an angle of $\frac{1}{2}\pi$ (M_PI denotes π).

1 GAALET

1.1 Compact expressions - compact implementation

Due to the possibility of overloading operators in C++, Gaalet offers a concise way of writing GA expressions in an implementation. Figure 1 shows a simple example of code which a vector a is rotated in using a rotor R, resulting into a vector b.
Rotor R is generated with a bivector S defining a rotation in the $e_1 \wedge e_2$-plane with an angle of $\frac{1}{2}\pi$. Vector a is initialized with coefficients scaling the three basis vectors e_1, e_2 and e_3. Vector b is determined using a sandwich operation, multiplying rotor R, vector a and the reverse of rotor R, the latter expressed by the overloaded, unary C++ $\sim$-operator in Gaalet. Another example for an operator overload is the geometric product, expressed in Gaalet with the binary C++ $*$-operator, often used in the code snippet of figure 1.

Note that the implementation in this example makes usage of the new type inference functionality (keyword `auto`) of the upcoming C++0x standard. With latter the C++ language will be enhanced to important new features, especially concerning the usage of templates in C++. For this reason Gaalet internally maintains two versions, one for the upcoming C++0x standard and a restricted version in terms of implementation comfort for the current C++ standard. While the C++0x version of Gaalet at this time is only compilable using the GNU Compiler Collection from version 4.5 upwards, the Gaalet version for the current C++ standard can be compiled using any modern C++-compiler (e.g. GNU Compiler Collection, Microsoft Visual C++, Intel C++, NVIDIA CUDA Compiler Driver).
Gaalet supports any algebra with a nondegenerate metric tensor signature of up to 64 dimensions (including Conformal Geometric Algebra).

1.2 Runtime performance

Expression templates build up an expression tree at compile time. This enables Gaalet to determine the resulting multivector type of an expression, if the multivector types of the operands are known. With this knowledge at compile time, no code is generated evaluating multivector elements which aren't part of the resulting multivector, thus reducing the basis operations of an expression evaluation notedly. Because the types of multivectors are supposed to be known, Gaalet uses compressed multivectors in general, thus reducing also the amount of memory needed to store multivectors.
Another important aspect of Gaalet as an expression templates library is the lazy evaluations of expressions. Expressions are evaluated when needed, but not necessarily when defined. Looking at the example of figure 1, in every line an expression is defined, but only the expression in the last line is evaluated to a multivector (vector b) by using the `eval()` function. A benefit is the possible reduction of temporary multivectors

in expressions defined over several lines. In general, expression templates can avoid computations of temporary results when evaluating an expression, in order to reduce expansive store- and fetch-operations, hence enabling optimized evaluations with respect to the processor cache.
Note that no symbolic optimizations are conducted on expressions by Gaalet, unlike Gaalop (Geometric Algebra Algorithms Optimizer) [Hildenbrand et al., 2008, 2010] does.

2 TUTORIAL PRESENTATION

As an introduction Gaalet's purpose and general functioning is overviewed, including aspects of metaprogramming in C++, expression templates and lazy evaluation. For clarification simple examples are discussed and the basic usage of Gaalet is explained.
A comparison of the evaluation speed of Gaalet examples and handwritten implementations of corresponding coordinate-based expressions is given, with fully and symbolic simplified coordinate-based expressions. Comparison results are discussed, especially concerning the lazy evaluation principle.
Limits of the C++ version in comparison to the C++0x version with regard to the implementation aspects are shown.
GPU programming with CUDA and Gaalet is discussed and an example is given.
If time allows, a larger example might be presented. The tutorial takes about 45 minutes.

3 PRESENTER

Dipl.-Ing. Florian Seybold is the author of Gaalet and works at the High Performance Computing Center Stuttgart (HLRS), University of Stuttgart. The visualization department operates a driving simulation, for which Florian Seybold developed a vehicle dynamics simulation using Geometric Algebra and implemented it with Gaalet. In collaboration with Dr. Dietmar Hildenbrand, TU Darmstadt, research on classical molecular dynamics simulation algorithms using conformal geometric algebra is underway, and respective implementations are done with Gaalet and Gaalop.

4 USAGE OVERVIEW

as of Gaalet release 0.1.
Sources at http://sourceforge.net/projects/gaalet

Definition of algebra $\mathscr{G}(p,q)$:

```
typedef
gaalet::algebra<gaalet::signature<p,q> >
algebra;
```

Definition of a multivector A:

```
algebra::mv<b1, b2, ...>::type A;
```

following Daniel Fontijne's bitmap representation:
(b_1, b_2, ..., are bitmaps denoting basis blades.)

1	$b = 00_2 = 0$
e_1	$b = 01_2 = 1$
e_2	$b = 10_2 = 2$
$e_1 \wedge e_2$	$b = 11_2 = 3$
...	...

Accessing multivector elements:

by index i	`A[i]`
by bitmap b	`A.element<b>()`

Unary operators:

grade: $\langle E \rangle_n$	`grade<n>(E)`
part: $b_1, b_2, \ldots \in E$	`part<b1, b2, ...>(E)`
reverse: $\widetilde{E}$	`~E`
inverse of E^{-1}	`!E`
exponential: e^E	`exp(E)`

Binary operators: (Inner product after Hestenes)

addition: $L+R$	`L+R`
subtraction: $L-R$	`L-R`
geometric product: LR	`L*R`
inner product: $L \cdot R$	`(L&R)`
outer product: $L \wedge R$	`(L^R)`

Expression evaluation:

evaluate E and return multivector	`eval(E)`
evaluate E and copy result into multivector A	`A = E;`
assign evaluated basis blades of expression E directly to multivector A	`A.assign(E);`