Gaalet - a C++ expression template library for implementing geometric algebra

Florian Seybold
Uwe Wössner

High Performance Computing Center Stuttgart (HLRS),
University of Stuttgart, Germany
seybold@hlrs.de, woessner@hlrs.de

Abstract

Geometric Algebra is a universal mathematical language, used in many scientific areas. Its graded structure inherits different mathematical concepts. Gaalet (Geometric Algebra ALgorithms Expression Templates) enables the implementation of Geometric Algebra in an elegant and concise way in the programming language C++. By using expression templates techniques and accomplishing grading operations at compile time, algorithms and expressions implemented with Gaalet yield good runtime performance.

1 Introduction and Previous Work

1.1 Geometric Algebra

1.1.1 Brief Introduction

Geometric Algebra can be defined as a non-degenerate Clifford Algebra over the reals, although this definition may vary in publications from different authors. The term "Geometric Algebra" was coined by David Hestenes, the "godfather" of Geometric Algebra. He fostered the usage of Geometric Algebra as a common mathematical language, as Geometric Algebra does inherit a lot of different concepts, for example the complex numbers, quaternions and the exterior (Grassmann) algebra, the latter being a foundation of Clifford and Geometric Algebra. David Hestenes [Hestenes & Sobczyk, 1984], [Hestenes, 1999] published some important books on this subject.
Following the definition above, we define a Geometric Algebra

$$\mathcal{G}(p,q) \equiv C\ell_{p,q}(\mathbb{R}) \tag{1}$$

with non-degenerate metric tensor signature (p,q) (square of basis vector ${e_{1..p}}^2 = 1$, ${e_{p+1..q}}^2 = -1$), Clifford Algebra $C\ell_{p,q}(\mathbb{R})$ over the real numbers $\mathbb{R}$.

This definition may also be extended to a degenerate metric tensor, resulting in more freedom. On the other side, some theorems David Hestenes formulated for Geometric Algebra hold only for the non-degenerate signature case.

A brief introduction to Geometric Algebra might be given by discussing the geometric product. The geometric product

$$e_i e_j = \begin{cases} e_i \wedge e_j & , i \neq j \\ e_i e_j = \pm 1 & , i = j \end{cases} \tag{2}$$

of two basis vectors e_i, e_j either generates another graded element of the algebra or it produces a scalar, depending on the algebra's signature. The graded element generated by the geometric product of two different basis vectors is called a basis 2-blade, or basis bivector, of three different basis vectors a basis 3-blade, or basis trivector, and so on. A scalar is a 0-blade. All these basis blades can be scaled by a coefficient and summed up, which produces a general multivector of the algebra (e.g. $A = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 \wedge e_2 \in \mathcal{G}(2,0)$).
By using specialised multivectors and developing theorems on their operations, expressions can be handled in a coordinate free manner. Important types of multivectors are k-vectors, which contain basis blades of the same grade k.

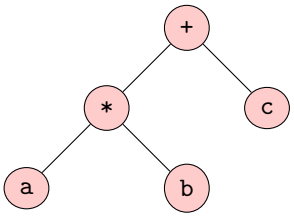
For a more sophisticated introduction to Geometric Algebra, we would like to refer the reader to literature of David Hestenes [Hestenes & Sobczyk, 1984], [Hestenes, 1999], Chris Doran and Anthony Lasenby [Doran & Lasenby, 2003], as well as Leo Dorst et al. [Dorst et al., 2007].

1.1.2 Implementation

It is a common notion that algorithms developed or described in Geometric Algebra take an elegant and compact form, compared to conventional algebras. On the other side, Geometric Algebra algorithms often tend to yield a worse runtime performance if implemented naively. (For some examples, see [Skala & Hildenbrand, 2009].)
There has been research going on about the implementation of Geometric Algebra expressions and algorithms, mainly concerning the problem to maintain the elegant algorithm description of Geometric Algebra in the implementation, while overcoming common performance issues.
In the next lines, common implementation frameworks for geometric algebra will shortly be described. It will be distinguished between external frameworks, i.e. code generators, which are not embedded into a specific programming language, as well as internal frameworks, i.e. libraries for a specific language.
External frameworks tend to offer high runtime performance by exploiting the circumstance that grading in Geometric Algebra expressions are fixed, thus grading can be handled beforehand and only multivector coefficients are subject to runtime computations.
Gaalop [Hildenbrand et al., 2010] uses an Geometric Algebra specific language

(CLUScript) for the description of expressions and algorithms and compiles these descriptions for certain target languages and platforms. It additionally applies heavy simplifications to expressions on the multivector coordinate basis and can examine whole algorithms for optimisations purposes.
Gaigen [Fontijne, 2006] can produce optimised implementations of Geometric Algebra elements and operations for certain target languages. It specialises the implementation of Geometric Algebra for a certain metric and generates an implementation in form of a library the programmer can use.
There are libraries available for specific programming languages, modelling the Clifford algebra or Geometric Algebra. To our knowledge, these libraries deal with the grading of Geometric Algebra elements at runtime, thus producing a certain overhead. In the following a short assortment of C++ libraries are listed.
GluCat [Leopardi, 2007] is a templated library and models Clifford algebra over the reals of arbitrary dimension and arbitrary signature in accordance with its author. It has been originally designed to be used with other generic C++ libraries.
C++ MV [Bell, 2004] models Clifford algebra over the reals of up to 63 dimensions. Its grading implementation is quite sophisticated.

1.2 Expression Templates



(a) Expression tree

```
typedef
addition<
  product<
    a_type,
    b_type>,
  c_type> expr;
```

(b) Expression type in C++

```
a_type a;
b_type b;
c_type c;
auto expr = a*b+c;
```

(c) Definition via operator overloading

Figure 1: Expression $a*b+c$: Modelled with expression tree (left), implementation type in C++ (middle) and concise definition of an expression, enabled by operator overloading (right). (Note upcoming standard C++0x keyword `auto`.)

Expressions Templates is a technique to describe expressions as types in the programming language C++. Templates form the generic part of C++, which enables the programmer to define a C++ class or functions using template arguments. Latter can be types or integer constants, thus the programmer does not know these types or constants when defining the class or function. Not until the templated class is used with template arguments to declare an object, or the templated function is called with template arguments.
Figure 1.2 shows an expression tree, modelling an example expression and how the C++ templated class type might look like. It is defined by nesting templated,

unary or binary operator classes as template arguments into parent operator classes. These templated operator classes are the nodes of the expression tree, with the special case of an end node, which might be an algebra element, e.g. a real number or a vector, as well as another predefined expression. Using the expression templates technique in conjunction with operator overloading allows for writing expressions in C++ in a concise, domain-specific form.
Expression Templates model the concept of lazy evaluation, thus an expression must not be evaluated when it is defined. This enables the compiler to handle the whole expression at compile time, inlining function calls and avoiding creation of temporary values. Vector-valued expressions can be evaluated on a coordinate basis, without creating temporary vectors. Using expression templates result in potentially faster code compared to a naive implementation, as Todd Veldhuizen [Veldhuizen, 1995] or David Vandevoorde [Vandevoorde & Josuttis, 2003] show. Both developed the expression template technique independently.
Recently Jochen Härdtlein et al. [Härdtlein et al., 2009] published advanced expression templates techniques, which, along others, reduce the complexity of expression templates, both on the implementation side and in the depth of nested template classes, which reduces compile time. This is done by applying the *curiously recurring template pattern* (CRTP) mechanism to the definition of expression tree nodes.

1.3 Applications in Visualisation

Important applications of Geometric Algebra algorithms are in Computer Graphics, Physics and Engineering. Especially the domain of visualisation takes advantage of Geometric Algebra, with a lot of potential for future work in this domain. See for example proceedings in [Skala & Hildenbrand, 2009].

As an example for an application in visualisation, we would like to refer to the problem of camera navigation in a visualisation framework. Werner Benger et al. [Benger et al., 2009] describe a Geometric Algebra algorithm used for camera navigation in the VISH [Benger et al., 2007] visualisation framework. The algorithm rotates a camera located at a point defined by position vector $P \in \mathbb{R}^3$, looking at a point defined by position vector $L \in \mathbb{R}^3$, around the view direction vector $t = L - P$ about an angle $\varphi \in \mathbb{R}$. This is simply described by the rotor

$$R_t = e^{-\frac{1}{2}\varphi\left(\frac{t}{|t|}\right)^\star}, \tag{3}$$

denoting the rotation of the camera.
The magnitude $|t|$ is used to normalise the view direction vector t. The star in the expression $(\frac{t}{|t|})^\star$ denotes the dual operator, yielding the plane normal the normalised view direction vector $\frac{t}{|t|}$ in form of a bivector. Multiplying the bivector by the negative half rotation angle and operating the exponential function on it yields the camera rotation about angle φ, denoted by rotor R_t. Benger et al. claim that this formulation is considered to be very simple compared to respective formulations using matrix and linear algebra.

A rotor R in Geometric Algebra is similar to a quaternion and may describe the rotation of a rigid object. For example, in order to rotate vector a using rotor R, the sandwich product RaR^{-1} can be applied.

2 The Motivation

2.1 Geometric Algebra Implementation

Implementations of Geometric Algebra algorithms created with help of the external frameworks Gaalop and Gaigen have shown to yield good runtime performance. Nevertheless, we think that their programming language independence can have negative impact on the development process of software using Geometric Algebra algorithms, although this opinion might be well countered by others. To make the reasons for this opinion clear, the implementation process of a Geometric Algebra algorithm and its integration into software is shortly discussed, both when using Gaalop and Gaigen.
As input, Gaalop explicitly uses a Geometric Algebra specific language, called CLUScript, for the description of Geometric Algebra algorithms and expressions. Compiling such algorithms or expressions written in CLUScript, Gaalop generates optimised code snippets for supported target languages. These code snippets don't represent the underlying coordinate-free description of a Geometric Algebra algorithm or expression anymore, but come in form of optimised code of a coordinate based evaluation. Furthermore, these code snippets have to be manually integrated into the actual software implementation. Beside the additional manual work of connecting generated code snippets with the application code, debugging of algorithms might get very hard. Note that there is already work going on to overcome these problems in form of a compiler driver [Charrier & Hildenbrand, 2010], similar to the NVIDIA CUDA Compiler Driver.
Gaigen generates specialised libraries for a specific Geometric Algebra in terms of its metric. As far as we know, Gaigen doesn't use operator overloading for C++, so the algorithm implementation lack a concise way of writing. There is still runtime overhead due to grading operations, which can be optimised out by profiling (analysis at runtime), with the cost of additional development work.
The C++ libraries GluCat and MV C++ must in general deal with grading of Geometric Algebra elements at runtime, thus keep a disadvantage concerning runtime performance.

3 Our Approach

3.1 Overview

Our approach is to combine Geometric Algebra and Expression Templates. It is presumed that the types of the input multivectors to an algorithm is known when implementing the algorithm. Thus grading of Geometric Algebra elements doesn't have to be handled at runtime, but can be done beforehand. The idea is

to use the metaprogramming capabilities of C++ templates in conjunction with the Expression Templates technique in order to handle grading at compile time. What remains at runtime are the computations with multivector coefficients, in an efficient way which the expression templates technique provides.

3.2 Multivector implementation

A multivector class consists of an array of arbitrary size for storing coefficients of a multivector, and a templated list of integer constants, mapping the coefficients to basis blades, as well as member functions for accessing coefficients. The integer list, mapping coefficients to basis blades, allows for storing only the needed coefficients in a multivector. This is sometimes referred to as compressed multivector. Note that at runtime, only the coefficients are accessed, and the constant map is normally only used for grading operations at compile time.
A bitmap representation for basis blades, as described amongst others by Daniel Fontijne [Fontijne, 2007], is used in order to store them in a templated constant integer list. A bit denotes a specific basis vector, the combination of bits denote the outer products of basis vectors, thus basis blades, for example:

$$001_b \equiv e_1, 010_b \equiv e_2, 100_b \equiv e_3, 101_b \equiv e_1 \wedge e_3 = e_{13}$$

Note the strict order of outer product operands. For example bitmap 101_b represents $e_1 \wedge e_3$, not $e_3 \wedge e_1$. Because the outer product of vectors anticommutes, the latter case can be represented with a sign flip in the coefficient ($a_{13}e_1 \wedge e_3 = -a_{13}e_3 \wedge e_1$).
So a definition of a multivector object in Gaalet, for example an rotor $R = a_0 + a_{12}e_1 \wedge e_2 + a_{23}e_2 \wedge e_3 + a_{31}e_3 \wedge e_1 \in \mathcal{G}(3,0)$, looks like this:

```
typedef gaalet::algebra<gaalet::signature<3,0> > em;
em::mv<0,3,5,6>::type R = {a0, a12, -a31, a23};
```

3.3 Operation implementation

Unary and binary operations are implemented like shown by Härdtlein et al. [Härdtlein et al., 2009], with additional routines for return type determination. This means that Gaalet determines the type of the resulting multivector object, that is the basis blades it contains, depending on the multivector operand types. For example the geometric product $ab \in \mathcal{G}(3,0)$ of two vectors $a = \langle a \rangle_1$, $b = \langle b \rangle_1$ results in a multivector containing the basis blades of a rotor: $ab = \langle ab \rangle_0 + \langle ab \rangle_2$. In Gaalet, this would look like:

```
typedef gaalet::algebra<gaalet::signature<3,0> > em;
em::mv<1,2,4>::type a = {1,2,3};
em::mv<1,2,4>::type b = {3,4,5};
std::cout << "ab: " << a*b << std::endl;
```

The output of this example would be:

```
ab: [ 26 -2 -4 -2 ] { 0 3 5 6 }
```

The integers in the curly brackets denote the basis blades contained in the resulting multivector.

3.4 Evaluation implementation

An important aspect of expression templates is their lazy evaluation capabilities. This is illustrated by implementing the camera rotation algorithm described in section 1.3:

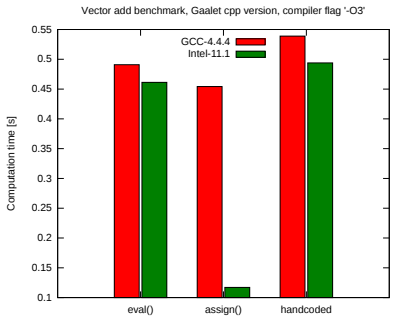
```
//Pseudoscalar for G(3,0):
em::mv<7>::type I = {1.0};
//Position vector declaration of camera and look-at point:
em::mv<1,2,4>::type L, P;
//Camera rotation angle:
double phi;
...
//View direction vector expression defined and evaluated:
em::mv<1,2,4>::type t = L - P;
//Expression defined, not evaluated:
auto S_t = t * !magnitude(t) * I;
//Expression defined, including another expression:
auto R_t = exp(-0.5*phi*S_t);
...
//Rotating vector definition:
em::mv<1,2,4>::type a = {0.0, 0.0, -1.0};
//Expression defined and evaluated into multivector:
auto b = eval(grade<1>(R_t*a*(~R_t)));
```

Without using the `grade<1>()`-operation, the resulting multivector type of `b` would be { 1 2 4 7 }, thus including the pseudoscalar type. Because a sandwich product like `R_t*a*(~R_t)` returns a pseudoscalar coefficient of zero, we can filter it out by using the `grade<1>()`-operation. This results in a multivector type { 1 2 4 }, and a pseudoscalar coefficient is never computed.

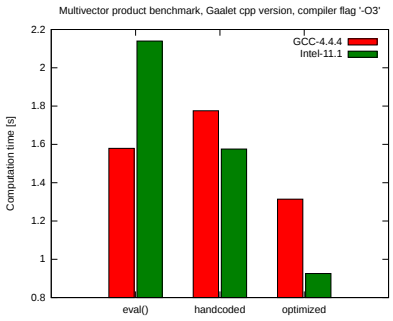
In general, expressions can depend on the multivector the result is stored in, for example in an iteration. Because the coefficients of that multivector might be cross accessed, the result of an evaluated expression must first be stored into a temporary multivector, the content of which has then to be moved or copied into the storing multivector. This procedure is implemented in the multivector member function `operator=()` and the global function `eval()`, latter if used with an existing multivector the result is stored in.
If the global function `eval()` is used for initialising a newly constructed multivector, the compiler should generate code that stores the resulting coefficients

directly into the newly constructed multivector. The constructor of a multivector, taking an expression for evaluation as argument, also stores the evaluated coefficients directly into the newly allocated multivector. Both procedures are used in the example shown above.
The programmer might know that it is safe to store coefficients directly to an existing multivector. He might then use the function `assign()` as well.

4 Results



(a) Vector addition benchmark: $c = c + a + b - a - b + a + b - a - b - c \in \mathbb{R}^{12}$, 10^7 times



(b) Multivector product benchmark: $a = ba\widetilde{b}$, $b = \langle b \rangle_0 + \langle b \rangle_2$, $a, b \in \mathcal{G}(3,0)$, 10^8 times

Figure 2: Comparing benchmarks of different implementation methods using different compilers: GCC 4.4.4 and Intel C++ Compiler 11.1

While elegance of implemented code might be now and then in the eye of the beholder, we can do performance measures of implementations. In figure 4 we show comparing benchmarks of different implementation methods of two artificial problems. Compared are the two different procedures of expression evaluation in Gaalet, either storing resulting coefficients directly to an existing multivector or firstly to a temporary multivector, and a respective handcoded implementation on a coordinate basis. Two different compilers are used for compiling and optimising code: GNU Compiler Collection 4.4.4 and Intel C++ Compiler 11.1.
The benchmarked problem shown in figure 2a is a simple vector addition of the form $c = c + a + b - a - b + a + b - a - b - c$ with $a, b, c \in \mathbb{R}^{12}$, conducted 10^7 times. Of interest here is the optimisation capabilities of the Intel compiler if using the function `assign()` for expression evaluation. In other respects, the Gaalet implementation and the handwritten implementation yield roughly the same performance.
In figure 2b, a benchmark of the problem $a = ba\widetilde{b}$ (the tilde denotes a reverse operation) with $a = \langle a \rangle_1 + \langle a \rangle_3$, $b = \langle b \rangle_0 + \langle b \rangle_2$ and $a, b \in \mathcal{G}(3,0)$ is shown, conducted 10^8 times. Here the function `assign()` is not considered, but two hand-

written implementations: One without a symbolic optimisation on a coordinate basis ("handcoded") and one with ("optimised"). One aspect this benchmarks shows, is the dependency of implementation performance, whether handwritten or with Gaalet, on compiler optimisations: The Intel Compiler yields more variance in the performance of different implementations then the GNU Compiler.

5 Summary

Geometric Algebra is a universal mathematical language with a graded structure based on the exterior algebra. For the implementation of Geometric Algebra algorithms or expressions, the grading operations produce a certain overhead when done at runtime. In general the grading operations can be accomplished beforehand. Gaalet does the grading operations at compile time by extending the concept of expression templates to grade type determination of a resulting multivector. This is done by metaprogramming using C++ templates, so no external tool is needed besides a C++ compiler (this includes the CUDA device code compiler), resulting in good performance of the Geometric Algebra implementations, while preserving its compact form.

6 Acknowledgements

While researching for literature on the Internet, we found that Jaap Suter (`http://www.jaapsuter.com`) had the idea of implementing Geometric Algebra with expression templates as well, although no publication or software could be found. One author learnt in private conversation with Robert Valkenburg (Industrial Research Limited, Auckland, NZ), that he also works on a Geometric Algebra implementation using expression templates.

References

[Bell, 2004] Bell, I. (2004). Multivector programming. URL: `http://home.clara.net/iancgbell/maths/geoprg.htm`.

[Benger et al., 2009] Benger, W., Hamilton, A., Folk, M., Koziol, Q., Su, S., Schnetter, E., Ritter, M., & Ritter, G. (2009). Using geometric algebra for navigation in riemannian and hard disc space. In *Proceedings of the International Workshop on Computer Graphics, Computer Vision and Mathematics, 2009*: Plzen, Czech Republic, Vaclav Skala - Union Agency. URL: `http://gravisma.zcu.cz/GraVisMa-2009`.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The Concepts of VISH. In 4^{th} *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Charrier & Hildenbrand, 2010] Charrier, P. & Hildenbrand, D. (2010). Gaalop compiler driver. URL: http://gravisma.zcu.cz/GraVisMa-2010.

[Doran & Lasenby, 2003] Doran, C. & Lasenby, A. (2003). *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. Cambridge University Press.

[Dorst et al., 2007] Dorst, L., Fontijne, D., & Mann, S. (2007). *Geometric Algebra for Physicists*. Burlington, MA, USA, Morgan Kaufmann Publishers, Elsevier Inc.

[Fontijne, 2006] Fontijne, D. (2006). Gaigen 2: a geometric algebra implementation generator. In *Proceedings of the 5th international conference on Generative programming and component engineering*: ACM. URL: http://staff.science.uva.nl/~fontijne/gaigen2.html.

[Fontijne, 2007] Fontijne, D. (2007). *Efficient Implementation of Geometric Algebra*. PhD thesis.

[Härdtlein et al., 2009] Härdtlein, J., Pflaum, C., Linke, A., & Wolters, C. H. (2009). Advanced expression templates programming. *Computing and Visualization in Science, Vol. 13*.

[Hestenes, 1999] Hestenes, D. (1999). *New foundations for classical mechanics: Fundamental Theories of Physics*. Dordrecht, Kluwer Academic Publishers.

[Hestenes & Sobczyk, 1984] Hestenes, D. & Sobczyk, G. (1984). *Clifford algebra to geometric calculus, a unified language for mathematics and physics*. D. Reidel Publishing Company, Dordrecht, Holland.

[Hildenbrand et al., 2010] Hildenbrand, D., Pitt, J., & Koch, A. (2010). *Gaalop - High Performance Parallel Computing based on Conformal Geometric Algebra*. Springer. URL: http://www.gaalop.de.

[Leopardi, 2007] Leopardi, P. (2007). Glucat: Generic library of universal clifford algebra templates. URL: http://glucat.sourceforge.net.

[Skala & Hildenbrand, 2009] Skala, V. & Hildenbrand, D., Eds. (2009). *Proceedings of the International Workshop on Computer Graphics, Computer Vision and Mathematics, 2009*. Plzen, Czech Republic, Vaclav Skala - Union Agency. URL: http://gravisma.zcu.cz/GraVisMa-2009.

[Vandevoorde & Josuttis, 2003] Vandevoorde, D. & Josuttis, N. M. (2003). *C++ templates: The Complete Guide*. Pearson Education, Boston, MA, USA.

[Veldhuizen, 1995] Veldhuizen, T. (1995). Expression templates. *C++ Report, Vol. 7*.