Geometric Algebra enhanced Precompiler for C++ and OpenCL

Patrick Charrier and Dietmar Hildenbrand

Abstract The focus of the this work is a simplified integration of algorithms expressed in Geometric Algebra (GA) in modern high level computer languages, namely C++, OpenCL and CUDA. A high runtime performance in terms of GA is achieved using symbolic simplification and code generation by a Precompiler that is directly integrated into CMake-based build toolchains.

1 Introduction

During the last decade, Geometric Algebra (GA) has become increasingly popular in expressing solutions to geometry-related problems in scientific applications of robotics, dynamics, computer graphics, and computer vision. Video game developers are becoming aware of GA, in search for simpler and faster ways to describe their lighting [1] and physics algorithms. Most developers makes use of C-related programming languages, such as C++, OpenCL [8] or CUDA [9], which are performant and abstract enough for most needs.

From a programmer's perspective, the integration of GA directly into C++, OpenCL, and CUDA, and yields a high level of intuitiveness. Coupled with a highly efficient generative software tool like Gaalop [6] in the background, an integration sets new standards for GA-powered software development. An advanced integration itself including other comforts, and to make GA-usage available to a broad audience, is the purpose of this work.

Patrick Charrier
TU Darmstadt, Germany, e-mail: patrick.charrier@stud.tu-darmstadt.de

Dietmar Hildenbrand
TU Darmstadt, Germany, e-mail: hildenbrand@cocoon.tu-darmstadt.de

1.1 Conformal Geometric Algebra

Conformal Geometric Algebra (CGA) is a new way of expressing many geometry focused mathematical problems. It deals naturally with intersections and transformations of planes, lines, spheres, circles, points, and point pairs, but is also good at representing mechanics and dynamics. In Linear Algebra, one would have to differentiate a plane-sphere intersection into three distinct cases, namely, circle intersection, point intersection and no intersection. In Conformal Geometric Algebra the intersection itself is formulated as one operation on the plane (P) and the sphere (S), respectively.

$$R = S \wedge P$$

The three different cases of Linear Algebra are implicitly contained in the one result R of Conformal Geometric Algebra, which is compact and better readable. Similar observations can be made in other applications of geometry related mathematics. Therefore, when applied to computer programs, GA has a high potential for improving code readability and to shortening production cycles. It has also been proven, that if implemented correctly, Geometric Algebra has at least similar performance, but sometimes even better performance, than conventional approaches [7].

2 Related Work

Combining both the aspects of Geometric Algebra and modern programming languages (namely C++, OpenCL and CUDA), promises to have a high potential for scientific work. Unfortunately, GA has such a high level of abstraction that it does not naturally fit into those languages. This section analyses several existing tools attempting to solve this problem in different ways.

Gaigen [4] is implemented by Daniel Fontijne at the University of Amsterdam. At the time of writing, it is in its third major version and has been developed since 2005. All versions work through efficient generation of C++ code, that is later linked to the final application binary. The latest version, Gaigen2, has a very remarkable profiling feedback mechanism, that bases the regeneration of code on the latest application runtime profiling. As [3] notes, Gaigen2 may have some problems with over-fitting that profiling feedback, and also causes some practical programming issues related to the classes and functions required to import into the application, but in general it is ready for practical use.

GMac [3] is developed by Ahmad Eid at Port-Said, Suez Canal University. It is based on C# and the Computer Algebra System Mathematica. GMac is very advanced in terms of stability and concepts; it builds upon the advantages of Gaigen and Gaalop, while trying to avoid their disadvantages. While it succeeds in these goals, it makes itself dependent on the closed source CAS Mathematica and a fixed programming language.

Gaalet [10] is a header-only C++ library that makes heavy use of the expression-template programming-technique of C++ and lazy-evaluation. Its performance is slightly worse than that of Gaalop [11] but with modern C++ compilers such as gcc 4.5 or higher, compile time for expression-templates is significantly reduced. It is perhaps the most suitable implementation in environments where one cannot install a lot of dependencies, such as the dedicated machines of the High Performance Computing Centre (HLRS) in Stuttgart, from where Gaalet originates.

The tools above do not match our general requirements for tools for Geometric Algebra Computing for the following reasons:

- Gaigen2 in its current form requires user interaction to run the code generation and feedback mechanism. This conflicts with the no-user interaction policy of precompilers. Therefore, it can not easily be integrated into a compiler toolchain. It is solely constructed to generate C++ code, leaving the upcoming field of GPU computing languages and other languages completely out of context.
- GMac requires no user-interaction, but is tightly coupled with C#, which is a very modern language, but unfortunately not a widely used one in high performance computing. Existing code bases will therefore most likely not profit from the advantages of GMac. Also, we want to maintain the possibility of choice among a variety of languages, instead of being focused on one language in particular.
- Gaalet has a very mature approach, but is tightly coupled with C++ as well. Like Gaigen, it is too complex to be used in OpenCL or CUDA.

Experience shows that GA is best optimized in connected chunks of code, rather than just simple statements. The tools above do not offer as much support for such functionality as Gaalop does.

But most importantly, all of the above tools are heavily dependent on a specific programming language, that is C++ or C#. Those languages are very complex; therefore, it is reasonable to assume that, those tools will never be able to cope with much simpler, but equally important languages like C, OpenCL, CUDA or Java. Since specialized GPU-Computing languages, such as OpenCL and CUDA promise to have an even more important role in the future, we cannot ignore them for the purpose of Geometric Algebra Computing. This is a major problem with the above approaches.

3 Gaalop Precompiler

In the following we present the Gaalop Precompiler (Gaalop GPC) itself, a technology aimed at solving the problems described in section 2. For that, Gaalop GPC introduces a new concept of integrating CLUScript code into C++, OpenCL, or CUDA code, here called native code.

The fundamental difference between CLUScript code and native code is that the former operates on multivectors, while the latter operates on low level storage containers, such as arrays, OpenCL vectors, or lists.

Clearly, this fact must be considered by separating both CLUScript code and native code explicitly. With Gaalop GPC, this distinction is made by enclosing CLUScript code with so called **#pragma** clucalc-statements, as shown in listing 1.

To transform between the perspective of CLUScript code and the perspective of native code, and vice versa, additional statements are required. We call them Interface Functions, since they interface both perspectives. Again, those statements must be separated from the rest of the code, because they may contain identifiers from both GA code and from native code, making them invalid code in both languages.

For example, the statement **const float** sphere_e1 = mv_get_bladecoeff(sphere,e1); retrieves the blade coefficient e_1 from the multivector sphere. Since neither sphere and e1 are valid C++ identifiers, nor **const float** sphere_e1 and the function call are valid CLUScript code, we call this intermediate code. Another layer of **#pragma** gpc-statements around the **#pragma** clucalc-statements differentiates intermediate code from native code and CLUScript code likewise.

```
// native code
void function(float* R_array,
              const float* S_array,
              const float* P_array) {

#pragma gpc begin

        // intermediate code
        S = mv_from_array(S_array,e1,e2,e3,einf,e0);
        P = mv_from_array(P_array,e1,e2,e3,einf,e0);

#pragma clucalc begin
        ?R = S ^ P // GA code
#pragma clucalc end

        // intermediate code
        R_array = mv_to_array(R,e1,e2,e3,einf,e0);

#pragma gpc end

// native code
}
```

Listing 1 Gaalop GPC code separation scheme example code implementing equation 1.1.

This code makes use of the mv_from_array() and mv_to_array() functions described in subsection 3.3. They are explicitly handling the transformation from arrays to multivectors, and vice versa.

3.1 Multivector Scoping

Also note, that Gaalop GPC enables scoping of multivectors across **#pragma** clucalc-blocks, meaning that multivectors declared for export in a different **#pragma** clucalc-

block, are accessible in succeeding **#pragma** clucalc-blocks, even across **#pragma** gpc-blocks:

```
{
#pragma gpc begin
#pragma clucalc begin // block A
        mv1 = ...;
        mv2 = ...;
        ?a = mv1*mv2;
#pragma clucalc end
#pragma gpc end

        ... // some C++ code

#pragma gpc begin
#pragma clucalc begin // block B
                      // automatically imports
                      // variable a from block A
        ?b = a + 10;
#pragma clucalc end
#pragma gpc end
}
```

Listing 2 Multivectors are accessible from within the same scope.

Accordingly, multivectors are not accessible across different scopes. The following listing will cause a compilation error:

```
{ // scope 1
#pragma gpc begin
#pragma clucalc begin // block A
        mv1 = ...;
        mv2 = ...;
        ?a = mv1*mv2;
#pragma clucalc end
#pragma gpc end
}

... // some code

{ // scope 2
#pragma gpc begin     // block B
#pragma clucalc begin // Compilation will fail,
        ?b = a + 10;  // because the identifier 'a' was declared
                      // in a different scope.
#pragma clucalc end
#pragma gpc end
}
```

Listing 3 Multivectors are not accessible across different scopes.

3.2 Point Triangle Test

To exemplify the very abstract perspective of listing 1, we here provide a real-life example from Computational Cloth Simulation, to clarify the concepts explained above.

The code snippet in listing 4 performs a test for a collision between a triangle t and a point p. The triangle has a thickness h and is a prism mathematically, but for convenience, we call it 'triangle'. Figure 1 illustrates this aspect.

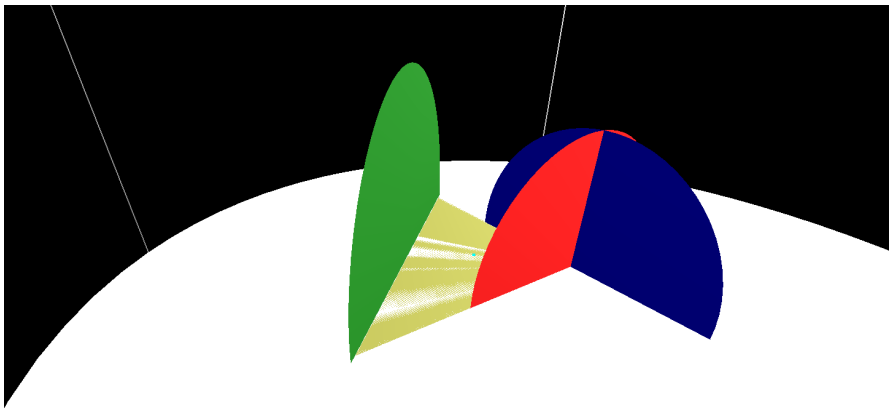

Fig. 1 Point Triangle intersection visualized in CLUCalc. The picture shows the triangle, the plane it is embedded in, and the three planes that define its edges.

```
bool pointTriangleTest(
        const float t1x, const float t1y, const float t1z,
        const float t2x, const float t2y, const float t2z,
        const float t3x, const float t3y, const float t3z,
        const float px, const float py, const float pz,
        const float h) {

#pragma gpc begin
#pragma clucalc begin
        // triangle properties
        TrianglePoint1 = VecN3(t1x, t1y, t1z);
        TrianglePoint2 = VecN3(t2x, t2y, t2z);
        TrianglePoint3 = VecN3(t3x, t3y, t3z);

        // point properties
        TestPoint = VecN3(px, py, pz);

        // construct the base plane
```

```
        plane = *(TrianglePoint1 ^
                  TrianglePoint2 ^
                  TrianglePoint3 ^
                  einf);

        // compute the signed distance of TestPoint to base plane
        ?d = plane . TestPoint;
#pragma clucalc end

        // check if distance to base plane exceeds h
        const float d_SCALAR = mv_get_bladecoeff(d,1);
        if (d_SCALAR * d_SCALAR > h * h)
                return false;

#pragma clucalc begin
        // extract the triangle normal
        normal_ = plane - (plane . e0) ^ einf;

        // construct boundary planes
        side1 = *(TrianglePoint1 ^ TrianglePoint2 ^
                  normal_ ^ einf);
        side2 = *(TrianglePoint2 ^ TrianglePoint3 ^
                  normal_ ^ einf);
        side3 = *(TrianglePoint3 ^ TrianglePoint1 ^
                  normal_ ^ einf);

        // compute distances
        ?d1 = side1 . TestPoint;
        ?d2 = side2 . TestPoint;
        ?d3 = side3 . TestPoint;
#pragma clucalc end

        // get signed distances
        const float d1_S = mv_get_bladecoeff(d1,1);
        const float d2_S = mv_get_bladecoeff(d2,1);
        const float d3_S = mv_get_bladecoeff(d3,1);
#pragma gpc end

        // check if TestPoint is inside boundary planes
        if (d1_S <= 0.0f && d2_S <= 0.0f && d3_S <= 0.0f
         || d1_S >= 0.0f && d2_S >= 0.0f && d3_S >= 0.0f)
                return true;

        return false;
}
```

Listing 4 A simple test, checking the collision of a 'triangle' t with thickness h with a point p, written for Gaalop GPC for C++.

The code executes the following steps.

1. Define the three triangle points and the test point.
2. Construct the base plane out of the triangle points.
3. Compute the signed distance d between the constructed plane and the test point.

4. Compute the normal of the plane.
5. Using this normal and the triangle points, compute the boundary planes, e.g. the planes that are perpendicular to the base plane and pass through every combination of the three points.
6. Compute the signed distances d_1, d_2 and d_3 between the test point and the three boundary planes.
7. Retrieve the scalar multivector-part from d_1, d_2 and d_3 and save them to **float**-constants d1_S, d2_S, and d3_S, by using the mv_get_bladecoeff() function.
8. The condition of a collision is satisfied, if d^2 is smaller than or equal the square of the triangle's thickness h, and all signed distances d_1, d_2 and d_3 share the same sign.

Note that raytracers in computer graphics use algorithms similar to this one. Recent work based on CGA [2] shows promising results with speedups of up to four times.

The example shows all **#pragma**-block-types in action. **#pragma** clucalc-blocks contain pure CLUScript expressing multivector computations. **#pragma** gpc-blocks primarily contain C++ code, but not exclusively. Clearly the statement **const float** d1_S = mv_get_bladecoeff(d1,1); is both C++ and CLUScript, since neither mv_get_bladecoeff(), nor d1 are declared within C++. To further clarify the difference, as an example one could write **const float** d1_E13 = mv_get_bladecoeff(d1,e1^e3); to ask for blade $e_1 \wedge e_3$ (which equals zero in this case).

d1 is a multivector, which is internally reduced to a so called Compressed Multivector Storage-array containing only non-zero multivector blade coefficients. Specifically to ease handling with these arrays, the Interface Function mv_get_bladecoeff() is designed to lookup a particular multivector blade coefficient. We call it an Interface Function specifically, because it is not a real function in terms of C++. Much simplified, Gaalop GPC simply replaces mv_get_bladecoeff() by its corresponding array lookup d1[0], reducing the line to **const float** d1_SCALAR = d1[0];. An internal mechanism makes Gaalop GPC able to tell which blade belongs to which array index.

The separation between **#pragma** gpc-block-contents and pure C++ is mostly a design decision. By definition it would be possible to embed all C++ into one big **#pragma** gpc-block, or to even leave that concept out completely, but the intention is to use **#pragma** gpc-blocks as transformation layers. A secondary, but also well founded consideration is the fact that pure C++-code is guaranteed to preserve relative line numbers throughout the precompiling process, because it is simply copied into the intermediate source-file in its original form. In most **#pragma** gpc-blocks, relative line numbers will almost certainly be altered by the process. Making the separation clear, it is possible to utilize the **#line** compiler directive for plain C++-parts, which enables the native compiler to refer back to the original source-file in the case of errors and warnings. This yields a much better user-friendliness when working in an Integrated Development Environment (IDE). The user may simply click on the listed messages to get directed to the correct positions in the original source-file.

3.3 Gaalop Precompiler Language Specification

Multivectors have a limited number of blades. For example, in Conformal Geometric Algebra, their size is limited to 32 blades. Therefore, a multivector storage for Conformal Geometric Algebra can only save a maximum of 32 blade coefficients. A naive approach may therefore simply save the maximum number of coefficients in an array. The problem with this approach is, that the number of blades grows exponentially with dimensionality. A 9D-Algebra [12] for example, that is proven to be useful in some cases, has exactly 512 blades and 512 blade coefficients, which are too many to save efficiently in an array for each multivector. Since we want to support even higher dimensions, this is not an option.

Fortunately, the simple observation that the majority of multivector blade coefficient of a multivector equals zero, helps us to overcome that. The obvious solution is to save only non-zero blade coefficients (Compressed Multivector Storage). To assist with this approach, several helper functions are defined in table 3.3.

The purpose of these helper functions is the transformation between multivectors and C/C++/OpenCL/CUDA language concepts, such as **float**-variables, arrays, or vectors. For example, mv_get_bladecoeff() is responsible for extracting a blade coefficient from a multivector, whereas mv_from_array() constructs a multivector from a C-like array.

coeff = mv_get_bladecoeff(mv,blade);	Get the coefficient of blade blade of multivector mv.
mv = mv_from_vec(vec);	Construct multivector mv from an OpenCL or CUDA-vector vec.
mv = mv_from_array(array, blades ,..);	Construct multivector mv from array array.
mv = mv_from_stridedarray(array, index, stride, blades ,...);	Construct multivector mv from array array at index index with stride stride. Example mv = mv_from_stridedarray(array ,0, nummvs,e1,e2,e3,e0, einf);.
array = mv_to_array(mv,blades ,...);	Write the blades blades ,... of multivector mv to array array. Example array = mv_to_array(mv,e1,e2,e3,e0, einf);.
array = mv_to_stridedarray (mv,index, stride, blades ,...);	Write the blades blades ,... of multivector mv to array array at index index with stride stride. Example array = mv_to_stridedarray (mv,0,nummvs,e1,e2,e3,e0, einf);.
vec = mv_to_vec(mv);	Write the multivector mv to an OpenCL or CUDA-vector vec.

Table 1 Gaalop GPC helper functions

4 Conclusion

Code simplicity, elegance, and intuitiveness are the major goals of this work. Recalling the code examples shows that these goals were reached to some extent. As Gaalop GPC directly profits from any improvements within Gaalop by invoking it, a high runtime performance is achieved on-the-fly.

Gaalop GPC symbolically optimizes the embedded CLUScript-code in order to improve runtime performance. A longer compile time is a natural consequence of the concept. However, we do not recommend putting much research into this aspect, as the build process can already be parallelized by many build automation tools like GNU Make [5]. It is found, that in reality, using parallel builds, a longer compile time is not a problem.

We would like to conclude, that the Gaalop Precompiler makes it even easier to work with GA inclusions in native code. Instead of separating code generation and code compilation into two distinct processes, it is a single simplified process with tight coupling support between the native and embedded languages.

Gaalop GPC is freely available for download from `www.gaalop.de` including a usage guide and documentation. Since the Gaalop Precompiler eases development, we hope that more scientists, game and software programmers will find their way into the applications of Geometric Algebra.

References

1. The homepage of geomerics ltd. Available at `http://www.geomerics.com`.
2. Michael Burger. Das effiziente raytracen von dreiecksnetzen auf mehrkernprozessoren, gpus und fpgas mittels geometrischer algebra. Master's thesis, TU Darmstadt, 2011.
3. Ahmad Hosney Awad Eid. *Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application*. PhD thesis, Port-Said, 2010.
4. Daniel Fontijne, Tim Bouma, and Leo Dorst. Gaigen 2: A geometric algebra implementation generator. Available at `http://staff.science.uva.nl/~fontijne/gaigen2.html`, 2007.
5. Free Software Foundation. Gnu make. http://www.gnu.org/software/make.
6. Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Joachim Pitt. The Gaalop home page. Available at `http://www.gaalop.de`, 2012.
7. Dietmar Hildenbrand, Daniel Fontijne, Yusheng Wang, Marc Alexa, and Leo Dorst. Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. In *Eurographics conference Vienna*, 2006.
8. Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
9. NVIDIA. The CUDA home page. Available at `http://www.nvidia.com/object/cuda_home.html`, 2010.
10. Florian Seybold. Gaalet - a c++ expression template library for implementing geometric algebra, 2010.
11. Florian Seybold, Patrick Charrier, Dietmar Hildenbrand, M. Bernreuther, and D. Jenz. Runtime performance of a molecular dynamics model using conformal geometric algebra. Slides available at `http://www.science.uva.nl/~leo/agacse2010/talks_world/Seybold.pdf`, 2010.

12. Julio Zamora-Esquivel. G6,3 geometric algebra. In *ICCA9, 7th International Conference on Clifford Algebras and their Applications*, 2011.