Geometric Algebra enhanced Precompiler for C++, OpenCL and Mathematica's OpenCLLink

Patrick Charrier, Mariusz Klimek, Christian Steinmetz and Dietmar Hildenbrand

Abstract The focus of this work is a simplified integration of algorithms expressed in Geometric Algebra (GA) into 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. Finally, we demonstrate how to interface our technology with Mathematica's OpenCLLink.

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 [2] and physics algorithms. Most developers makes use of C-related programming languages, such as C++, OpenCL [15] or CUDA [16], which are performant and abstract enough for most needs.

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

Mariusz Klimek
TU Darmstadt, Germany, e-mail: klimek@gsc.tu-darmstadt.de

Christian Steinmetz
TU Darmstadt, Germany, e-mail: cssteinmetz@gmx.de

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

From a programmer's perspective, the integration of GA directly into these languages yields a high level of intuitiveness. Coupled with a highly efficient generative software tool like Gaalop [9] 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.

1.1 Geometric Algebra by example

Geometric Algebra (GA) 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 at all. In the so called Conformal Geometric Algebra the whole intersection may be formulated as one operation on the plane (P) and the sphere (S), respectively:

$$R = S \wedge P \tag{1}$$

Note 1. Here, the symbol $\wedge$ denotes the so called outer product (see Subsubsection 1.2.1), as used in CLUScript [18] and [9].

The three different cases of Linear Algebra are implicitly contained in the one result R of Conformal Geometric Algebra in Equation 1, which is more 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 [12].

The example above shall serve as motivation towards Geometric Algebra in general and is formulated in the so called Conformal Geometric Algebra. However the concept of Geometric Algebra is much broader, as there exist several other algebras, with the most commonly known ones being the Euclidean-, Projective- and Conformal Geometric Algebras. Subsection 1.2 explains the three most basic operations defined on all Geometric Algebras.

1.2 The products of Geometric Algebra

Table 1 Notations of Conformal Geometric Algebra

notation	**meaning**	**alternative**
AB	geometric product of A and B	
$A \wedge B$	outer product of A and B	A^B
$A \cdot B$	inner product of A and B	A.B
A^*	dual of A	dual(A)
A^{-1}	inverse of A	1/A
$\tilde{A}$	reverse of A	
e_0	conformal origin	e_0
e_∞	conformal infinity	e_{inf}

The three most often used products of Geometric Algebra are the **outer**, the **inner** and the **geometric** product. In table 1 the notations of these products are listed. We will use the outer product mainly for the construction and intersection of geometric objects while the inner product will be used for the computation of angles and distances. The geometric product will be used mainly for the description of transformations.

Note that the three products do not only apply to the Conformal Geometric Algebra explained in Subsection 1.1. Many different types of algebras with various dimensions may be defined, such as the three-dimensional Euclidean Geometric Algebra. All of these are subject to the three products explained in the following.

1.2.1 The outer product

Geometric Algebra provides an outer product $\wedge$ with the following properties

	Property	Meaning
1.	anti-symmetry	$u \wedge v = -(v \wedge u)$
2.	linearity	$u \wedge (v + w) = u \wedge v + u \wedge w$
3.	associativity	$u \wedge (v \wedge w) = (u \wedge v) \wedge w$

Property (1.) applies only for vectors, the other ones are generally valid (so also for multivectors).

The outer product of parallel vectors is 0.

$$a \wedge a = -(a \wedge a) = 0 \tag{2}$$

This is the reason why the outer product can be used as a measure for parallelness.

1.2.2 The inner product

For the 3D Euclidean space, the inner product of 2 vectors is the same as the well known Euclidean scalar product of 2 vectors. For perpendicular vectors the inner product is 0, for instance $e_1 \cdot e_2 = 0$. In Geometric Algebra, the inner product is not only defined for vectors. The inner product is grade decreasing, e. g. the result of the inner product of an element with grade 2 and grade 1 is an element of grade 2-1 =1. Please refer to [19] and [11] for a mathematical treatment.

Note that in literature you will find different versions of the inner product, e.g. the left contraction or the Hestenes inner product. The one we use (as well as CLUCalc) is sometimes also called the "dot product" - because we *only* use this one, we call it "the" inner product.

1.2.3 The geometric product

The geometric product is an amazingly powerful operation. It has a lot of geometric meaning whereby the easy handling of transformations is the most important one. The geometric product is a combination of the outer product and the inner product. The geometric product of u and v is denoted by uv. *For vectors* u and v the geometric product uv can be defined as

$$uv = u \wedge v + u \cdot v \tag{3}$$

We derive for the inner and the outer product

$$u \cdot v = \frac{1}{2}(uv + vu) \tag{4}$$

$$u \wedge v = \frac{1}{2}(uv - vu), \tag{5}$$

but as noted before: In this form these formulas only apply for vectors.

Divisions by algebraic objects are possible due to the fact that the geometric product is invertible. The **dual** of an algebraic object is calculated with the help of its division by the pseudoscalar. The **reverse** is an operator simply reversing the order of vectors in a blade. The notations of these operations are listed in table 1 on page 3.

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 these languages. A number of software packages exist to overcome this problem. They are listed in the following.

2.1 CLUScript

Conformal Geometric Algebra can not be expressed in terms of regular mathematical syntax. CGA-specific operators like the outer product $\wedge$, inner product . and geometric product $*$ require special treatment in regular programming languages or the definition of a completely new Domain Specific Language (DSL).

The DSL that powers this work is CLUScript. The especially designed integrated development environment for CLUScript is called CLUViz (new), and is freely available at [18]. In words of the author Dr. Christian Perwass [18, 17]:

> CLUCalc/CLUViz is a freely (for non-commercial use) available software tool for 3D visualizations and scientific calculations that was conceived and written by Dr. Christian Perwass. CLUCalc interprets a script language called CLUScript, which has been designed to make mathematical calculations and visualizations very intuitive.

Indeed, CLUScript is a very intuitive language, and we have found CLUCalc to be an advanced tool for developing and testing Geometric Algebra algorithms. It is easy to use, installs and runs smoothly on Microsoft Windows platforms. Unfortunately, the support for Linux and Macintosh platforms is very limited.

2.2 Gaalop as the foundation of this work

The Geometric Algebra Algorithms Optimizer (Gaalop) [13] was developed by TU Darmstadt (Germany) and is a powerful tool for optimizing algorithms, expressed in Geometric Algebra. It generates non GA-specific code from code defined in CLUScript and symbolically optimizes the algorithm on-the-fly, optionally invoking a Computer Algebra System (CAS). In this context, GA can be seen as a higher level mathematical language that is being transformed into simple arithmetics by Gaalop. Philosophically spoken, Gaalop could be defined as a math compiler.

CLUScript as an input language and C/C++ as output language has proven to be an extremely powerful combination. It is also possible to generate LaTeX, CLUScript,

and Verilog representations, the latter being suitable for Field Programmable Gate Arrays (FPGA). For evaluation purposes it is often helpful to choose CLUScript output, then replace the original CLUScript with the optimized code and test the result for the same functionality as the original code.

With [22], Gaalop is no longer dependent on Maple. It can optimize CLUScript with its internal Table Based Approach and several other internal optimization mechanisms. It may also invoke the Open-Source CAS Maxima on-the-fly, but this feature is completely optional. Gaalop is especially good at optimizing larger connected chunks of code, where other tools mainly focus on single statements.

Support for arbitrary Geometric Algebras was added with [23]. This enables the usage of customizable Geometric Algebra-definition files, for example for the Euclidean- (3d), Projective- (4d), Conformal- (5d) and Space-time (st4d) Geometric Algebras. Self-defined Geometric Algebras may be added with ease, giving the user the freedom to experiment with new and yet undocumented Geometric Algebras while taking full advantage of Gaalop's symbolic optimization engine.

2.3 Contributions of this work

The foundations of this work were made with [13] with the tool Gaalop. While Gaalop provides a graphical user interface (GUI) that enables conversion from pure CLUScript [18] to C++, Verilog and many more, this work goes one step further and places this functionality inside a modern programming toolchain, namely all toolchains targeted by the build manager CMake. This provides a much better workflow, since it eliminates the tedious process of copying code snippets by hand.

A plain and simple embedding of CLUScript into a native language would not be sufficient, because some communication is required between both languages. This work proposes so called Interface Features to close this gap between languages, while also carefully taking into account memory access performance on GPU-hardware. A way to use Gaalop GPC and arbitrary Geometric Algebras in collaboration with Mathematica's OpenCLLink is demonstrated in the space-time Geometric Algebra example in Section 4.

2.4 Alternatives to Gaalop GPC

Several other tools exist as alternatives to Gaalop GPC. This section motivates why those do not match our general requirements on tools for Geometric Algebra Com-

puting. Firstly, we analyze several existing tools attempting to solve this problem in different ways.

Gaigen [6] was 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 [5] 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 [5] 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, for example C#.

Gaalet [20] 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 [21] 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.

Several libraries similar to Gaalet exist, for example GluCat or EBgal. We do not further investigate them, because of the similarities of their approaches to Gaalet's approach. It is noted that expression templates seem to be the most common GA-implementation method in C++.

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

- Gaigen2 in its current form requires user interaction to run the code generation and feedback mechanism. The tool has to be manually controlled in order to get the best results. This conflicts with the no-user interaction policy of precompilers. Therefore, it can not easily be integrated into a compiler tool-chain. It is solely constructed to generate C++-code, leaving the upcoming field of GPU computing-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++. 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.

2.5 Performance of Geometric Algebra in comparison to conventional approaches

The 2006 Paper *Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra* [12] compares Gaalop, Gaigen and a conventional approach. It concludes that Gaalop and Gaigen excel the performance of the conventional approach by a factor of three. On the other hand Gaalop and Gaigen both required a lot more implementation effort. Since then, a lot of work has been put into both tools, and the effort required to implement applications was shrunken significantly while the stability constantly improved.

2.6 Arrays, vectors and lists as the common denominator of programming languages

Geometric Algebra is based on multivectors. Those have to be expressed in the best possible form available in the target programming language.

Multivectors are consisted of multiple blades and their coefficients. From a programming perspective, a multivector could be seen as a plain collection of coefficients, without interpreting it in any mathematical way. The simplest possible notion of such a collection is an array in most programming languages, especially in C/C++, OpenCL, CUDA and Java. An obvious solution to bridge the gap between the two paradigms stated above, is therefore to generate an array for each multivector.

Since Gaalop is perfect in doing so, it is a reasonable choice as the foundation of this work, especially because we would like to support as many programming languages as possible.

Even if a language does not support arrays, like the functional programming language Racket [1], it will most likely have some similar low end storage container like lists, so that Gaalop could still target it through the implementation of a new backend.

We conclude, that Gaalop as the foundation of this work has the most potential for bringing Geometric Algebra to most programming languages, primarily because of its focus on low level storage containers which are available in all programming languages.

3 Gaalop Precompiler

In the following we present the Gaalop Precompiler (Gaalop GPC) [3], a technology aimed at solving the problems described in Section 2. For that, Gaalop GPC introduces a new concept of integrating CLUScript into C++, OpenCL, or CUDA-code, here called native code.

The fundamental difference between CLUScript 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 and native code explicitly. With Gaalop GPC, this distinction is made by enclosing CLUScript with so called **#pragma** clucalc-statements, as shown in Listing 1.

To transform between the perspective of CLUScript and the one of native code, and vice-versa, additional statements are required. We call them Interface Features, 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 Interface-code. Another layer of **#pragma** gpc-statements around the **#pragma** clucalc-statements differentiates Interface-code from native-code and CLUScript likewise.

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

[1] http://racket-lang.org

```
#pragma gpc begin

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

        // Interface 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.

This code makes use of the mv_from_array() and mv_to_array() functions. They are explicitly handling the transformation from arrays to multivectors, and vice versa. The functions are part of a set of so called Interface Feature defined in [3] and on the Gaalop-homepage [10].

In this example, the blades $e_1, e_2, e_3, e_{inf}, e_0$ are read from S_array and P_array into the multivectors S (sphere) and P (plane) using the mv_from_array() Interface Feature. The intersection R of both is then computed by ?R = S ^ P, where the question-mark ? marks the multivector to be visible outside the **#pragma** clucalc-statements. The Interface Feature mv_to_array() then copies the stated blade-coefficients of R into the array R_array.

Line numberings

Please note that pure C++-code is guaranteed to have preserved relative line numbers throughout the precompiling-process, because it is simply copied into an 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.

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:

```
{ // scope begin

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

        ... // some C++ code

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

} // scope end
```

Listing 2 Multivectors are accessible from within the same scope.

Note that if no Interface code is required, the **#pragma** gpc-statements can be omitted. Also, multivectors are not accessible across different scopes. The following listing will cause a compilation error:

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

... // some code

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

Listing 3 Multivectors are not accessible across different scopes.

4 A Faraday example for Mathematica's OpenCLLink

In this example we demonstrate how to compute the electromagnetic field generated by a moving charged point particle. In order to capture relativistic effects, four-dimensional space-time is employed. An analytic treatment of this example is explained in section 7.3 of the book [4].

The basis of space-time algebra consists of a time-like vector e_4 and three space-like vectors e_1, e_2, e_3. Our sign convention is chosen as follows

$$e_4^2 = +1 \text{ and } e_1^2 = e_2^2 = e_3^2 = -1\,.$$

To translate three-dimensional vector, e.g., electric field, to its space-time form we proceed as follows:

$$\mathbf{E} = E^x\sigma_x + E^y\sigma_y + E^z\sigma_z = E^x e_1 e_4 + E^y e_2 e_4 + E^z e_3 e_4\,.$$

The extraction of, e.g., E^y, from Faraday multivector $F = \mathbf{E} + I\mathbf{B}$ is obtained by calculating

$$E^y = (e_2 e_4)\cdot F\,.$$

The vector $r_0 = [x_0, y_0, z_0, t_0]$ is the position of our moving particle in space-time, where x,y,z denote the spatial coordinates and t the time coordinate. Its trajectory $r_0(\tau) = [x_0(\tau), y_0(\tau), z_0(\tau), t_0(\tau)]$ is parameterized through the so called proper time τ.

The null vector

$$X = x - x_0(\tau) \tag{6}$$

connects an arbitrary point x in space-time with the trajectory $x_0(\tau)$. Since this vector is null by definition, we use

$$X^2 = 0 \tag{7}$$

to obtain $\tau(x)$. From the two possible solutions we only consider the one which is in the past. This solution is called retarded and has the property of $t_0(\tau) < t$.

With the knowledge of $\tau(x)$, one may calculate the electromagnetic field tensor

$$F(\tau) = \frac{X \wedge v + \frac{1}{2} X\dot{v} \wedge vX}{(X\cdot v)^3} \tag{8}$$

where $X = X(\tau)$,$v = v(\tau)$,$\dot{v} = \dot{v}(\tau)$ are all functions of τ, with

$$v = v(\tau) = \frac{dx_0}{d\tau} \tag{9}$$

being the velocity of the particle and

$$\dot{v} = \dot{v}(\tau) = \frac{d^2 x_0}{d\tau^2} \tag{10}$$

being its acceleration in space-time.

Having F it is easy to implement the following OpenCL-kernel:

```
code = "
__kernel void faraday_kernel(__global float* toMathematica,
                             __global float* fromMathematica,
                             const int length) {

        const size_t index = get_global_id(0);
        if(index >= length)
                return;

#pragma gpc begin
        float Xx = fromMathematica[index*12 + 0];
        float Xy = fromMathematica[index*12 + 1];
        float Xz = fromMathematica[index*12 + 2];
        float Xt = fromMathematica[index*12 + 3];
        float Vx = fromMathematica[index*12 + 4];
        float Vy = fromMathematica[index*12 + 5];
        float Vz = fromMathematica[index*12 + 6];
        float Vt = fromMathematica[index*12 + 7];
        float Vdotx = fromMathematica[index*12 + 8];
        float Vdoty = fromMathematica[index*12 + 9];
        float Vdotz = fromMathematica[index*12 + 10];
        float Vdott = fromMathematica[index*12 + 11];
 #pragma clucalc begin
        X = Xx*e1+Xy*e2+Xz*e3+Xt*e4;
        V = Vx*e1+Vy*e2+Vz*e3+Vt*e4;
        Vdot = Vdotx*e1+Vdoty*e2+Vdotz*e3+Vdott*e4;

        dot = X.V;
        ?F = (X^V+0.5f*X*Vdot^V*X)/(dot*dot*dot);
#pragma clucalc end
        (toMathematica+index) = mv_to_array(F,
                1,e1,e2,e3,e4,e1^e2,e1^e3,e1^e4,e2^e3,e2^e4,e3^e4,
                e1^(e2^e3),e1^(e2^e4),e1^(e3^e4),e2^(e3^e4),
                e1^(e2^(e3^e4)));
#pragma gpc end
}"
```

This reads all values from an array called fromMathematica. It then calculates F and saves it to a second array called toMathematica.

The code may be compiled and loaded using the following Mathematica-commands:

```
(*import OpenCLLink*)
Needs["OpenCLLink`"]

(*system command for precompilation with Gaalop GPC*)
command =
"java -jar starter-1.0.0.jar -algebraName st4d"
```

```
"-m usr/bin/maxima -optimizer de.gaalop.tba.Plugin"
"-generator de.gaalop.compressed.Plugin"
"-o \"out.cl\" -i \"in.clg\""

(*export code to file, precompile and import*)
Export["in.clg", code]
Run[command]
code = Import["out.cl"];

(*compile and link the OpenCL-kernel to the function faraday*)
faraday = OpenCLFunctionLoad[code, "faraday_kernel",
        {{_Real}, {_Real}, {_Integer}, {16},
        "ShellOutputFunction" -> Print]
```

As you can see, with the statement −algebraName st4d, the code is compiled using the four dimensional space-time geometric algebra. This example shows that Gaalop can handle geometric algebras of arbitrary dimension and signature. This was a contribution of [23]: Gaalop needs only two definition files for using a Geometric Algebra. The well-known Conformal Geometric Algebra, the Euclidean and Projective Geometric Algebra and some others are already included in Gaalop besides of the four dimensional space-time Geometric Algebra.

They are specified respectively by putting −algebra name 5d, −algebraName 3d and −algebraName 4d in the Gaalop command. Refer to [23] and the Gaalop-manual at [10] for more information on this topic.

Several other settings such as the Maxima-path, the optimization-plugin and the generator-plugin are specified as well, but are not in the focus of this document.

The OpenCLLink-function faraday may be used to compute F over a range and to plot it subsequently.

```
(*constants*)
alpha = 1;
Omega = Pi;
startTime = -20;

(*define the space-time range to be evaluated*)
t0[tau_] = tau*Cosh[alpha];
x0[tau_] = (1/Omega)*Cos[Omega*tau]*Sinh[alpha];
y0[tau_] = (1/Omega)*Sin[Omega*tau]*Sinh[alpha];
z0[tau_] = 0;

(*compute tau table*)
range = 10;
tab = ParallelTable[temp[{t, x, y, z},
    tau /.
    FindRoot[(t - t0[tau])^2 - (x - x0[tau])^2 -
             (y - y0[tau])^2 - (z - z0[tau])^2,
             {tau, startTime}]],
             {t, -range, range, .5}, {x, -range, range, .5},
             {y, -range, range, .5}, {z, -range, range, 2}];
             // AbsoluteTiming
```

```
tab = Flatten[tab];
tab = tab /. temp -> List
tau = Interpolation[tab]

(*define the input arrays*)
r = {t, x, y, z} // RotateLeft[#, 1] &
r0 = {taus*Cosh[alpha],(1/Omega)*Cos[Omega*taus]*Sinh[alpha],
   (1/Omega)*Sin[Omega*taus]*Sinh[alpha], 0}
   // RotateLeft[#, 1] &
v = D[r0, taus]
vdot = D[v, taus]
X[t_, x_, y_, z_] = N[r - r0] /. {taus -> tau[t, x, y, z]}
V[t_, x_, y_, z_] = N[v] /. {taus -> tau[t, x, y, z]}
Vdot[t_, x_, y_, z_] = N[vdot] /. {taus -> tau[t, x, y, z]}

(*run faraday OpenCL-kernel and save results to F*)
F[t_, x_, y_, z_] =
 faraday[Sequence @@
   Join[X[t, x, y, z], V[t, x, y, z], Vdot[t, x, y, z]]]

(*extract electric field E from F*)
ExEy[x_, y_] = (F[3.1, x, y, 0.]);

(*plot E*)
Block[{t = 0, z = 0},
 DensityPlot[Norm[ExEy[x, y]], {x, -10, 10}, {y, -10, 10},
  PlotPoints -> 150, ImageSize -> 500]]
```

The OpenCLLink-kernel produces the data for Figure 1 showing the magnitude of the electric field $|\mathbf{E}| = \sqrt{E_x^2 + E_y^2}$ extracted from F.

5 Arbitrary Geometric Algebras

To exemplify arbitrary Geometric Algebras in Gaalop the *definition.csv* file of the four dimensional space-time Geometric Algebra is listed below:

```
1,e1,e2,e3,e4
// empty line
1,e1,e2,e3,e4
e1=1,e2=-1,e3=-1,e4=-1
// empty line
```

The lines have the following meaning:

- Set of all base vectors, starting with the base vector 1 and separated mas. This is named baseVectorSet1.
- The basis transformation from baseVectorSet1 to baseVectorSet2 for each element of baseVectorSet2.

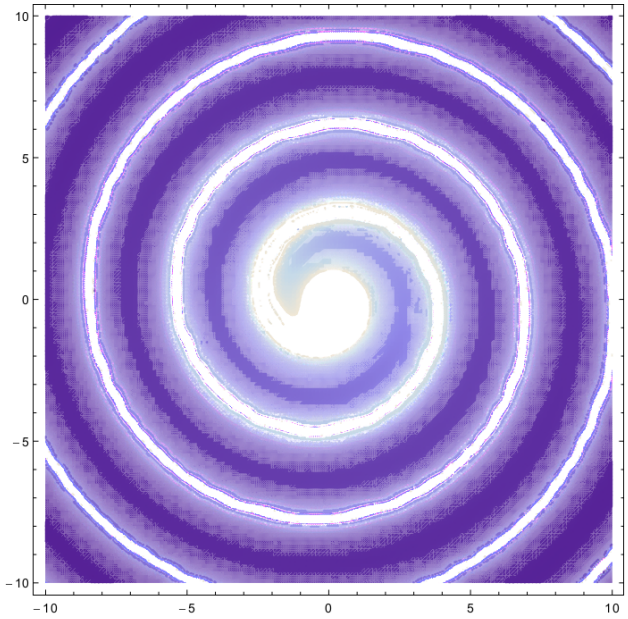

Fig. 1 Magnitude of the electric field $|\mathbf{E}| = \sqrt{E_x^2 + E_y^2}$ of a rotating charge at relativistic velocity. Note that $E_z = 0$ due to the symmetry of the field.

- Set of all base vectors, starting with the base vector 1 and separated mas. It is named baseVectorSet2.
- Signature of the algebra in baseVectorSet1.
- The basis transformation from baseVectorSet2 to baseVectorSet1 for each element of baseVectorSet1.

In contrast see the *definition.csv* file of the well-known Conformal Geometric Algebra:

```
1,e1,e2,e3,einf,e0
ep=0.5*einf-e0,em=0.5*einf+e0
1,e1,e2,e3,ep,em
e1=1,e2=1,e3=1,ep=1,em=-1
e0=0.5*em-0.5*ep,einf=em+ep
```

More information on Geometric Algebra definitions can be found in [24].

6 Future Work

This section discusses some ideas on how to further improve Gaalop Precompiler in the future.

Towards a deeper integration

In future work, a much more integrated language without the need for precompiler-directives could be defined, but with experimental work on this subject we found that this is not an idea worth achieving from neither a technical nor a transparency-point-of-view. The technical problems were mainly caused by the need for context-specific grammar and the lack of support thereof in parser tools like ANTLR. We also found that this goes far beyond the scope of a precompiler and would require the definition of a completely new compiler, which would contradict with our goal to support a variety of programming languages. Lack of transparency means that users (or programmers) of this future language, that are relatively new to the concept, would most likely have trouble to understand the intermix of multivectors and native variables resulting from this concept. Hence, we settled on the clear separation of CLUScript and native code with the language definition in Section 3.

Algebraic multivector rendering for Conformal Geometric Algebra as a language feature

Gaalop GPC is currently being enhanced to support algebraic rendering of multivectors similar to CLUCalc. That is, given a particular multivector m, which is marked for visualization using the colon-prefix : in Geometric Algebra code, equal to the functionality in CLUCalc, Gaalop GPC will firstly determine its representation in three-dimensional space (e.g. sphere, plane, circle, line, point-pair or point). Given the representation and its parameters, Gaalop GPC will render the appropriate object with OpenGL [14] or other rendering APIs, similar to the method used in CLUCalc. The rendering interface will be defined in an abstract way, so that new rendering APIs may be supported in the future. The user may create its own window or have it created automatically by our software.

7 Future Applications

There are some promising applications, which would profit from an implementation based on Gaalop GPC.

A moving least-squares approach to rendering surfaces using higher-dimensional Geometric Algebras

The moving least squares or weighted least-squares approach is a useful method of fitting arbitrary point clouds with a set of weighted continuous functions. More specific, multiple subsets of a point-cloud are fitted by independent continuous functions and the surface is then defined through weighted interpolations of these functions. These continuous functions could now be defined through the fitting of higher-dimensional multivectors to the subsets of the point-cloud.

Physics Libraries

Modern physics-libraries like Bullet [1] could profit from GA-based collision detection. Versors and velocity screws are an interesting basis upon which to define dynamics in physics-libraries. Theoretical background on this has been laid by [8].

Computer Graphics and Games

Eric Lengyel presented his research on Grassmann Algebra at the Game Developers Conference 2012 in San Francisco [2]. Although he is not considering algebras with dimensionality higher than four, his proposals could also be implemented using Gaalop GPC. A full set of Gaalop GPC-based computer gaming oriented libraries could be a promising topic for Geometric Algebra.

[2] `http://www.terathon.com/gdc12_lengyel.pdf`

Molecular dynamics with focus on polymer-chains

The paper [21] showed the applicability of Geometric Algebra in molecular dynamics simulations using local coordinates, for molecules with a limited number of atoms.

Special interest for further work lies on simulating so called semi-rigid polymer-chains. Polymers are macromolecules that have a chain-like composition. The chain itself consists of a large number of atoms, most of which are strongly-bonded.

The idea is now to define these strongly-bonded groups of atoms as rigid, giving multiple semi-rigid groups of atoms, that interact with each other. These semi-rigid groups of atoms can now be simulated according to Newton's and Euler's laws of motion.

We expect a computational speed-up by less and more intelligent memory accesses caused primarily by the mv_from_stridedarray () and mv_to_stridedarray () Interface Features. Furthermore, it will feature lesser space requirements of GA-versors and higher numerical stability compared to Matrices and/or approaches without internal coordinates.

8 Conclusion

Code simplicity, elegance, and intuitiveness are the major goals of this work. Recalling the code examples shows that these goals were reached to a large 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 researching this topic, as the build process can already be parallelized by many build automation tools like GNU Make [7]. 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 native and embedded languages.

The combination of GA with OpenCL or CUDA especially enables new methods of research. Mathematica's OpenCLLink provides ways to make use of our technology inside the well-known Computer Algebra System.

References

1. Bullet continuous collision detection and physics library.
2. The homepage of geomerics ltd. Available at `http://www.geomerics.com`.
3. Patrick Charrier. Geometric algebra enhanced precompiler for c++ and opencl. Master's thesis, TU Darmstadt, 2012.
4. Chris Doran and Anthony Lasenby. *Geometric Algebra for Physicists*. Cambridge University Press, 2003.
5. Ahmad Hosney Awad Eid. *Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application*. PhD thesis, Port-Said, 2010.
6. 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.
7. Free Software Foundation. Gnu make. http://www.gnu.org/software/make.
8. David Hestenes. Old wine in new bottles : A new algebraic framework for computational geometry. In Eduardo Bayro-Corrochano and Garret Sobczyk, editors, *Geometric Algebra with Applications in Science and Engineering*. Birkhäuser, 2001.
9. Dietmar Hildenbrand. *Foundations of Geometric Algebra Computing*. Springer, 2013.
10. Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Joachim Pitt. The Gaalop home page. Available at `http://www.gaalop.de`, 2012.
11. Dietmar Hildenbrand, Daniel Fontijne, Christian Perwass, and Leo Dorst. Tutorial geometric algebra and its application to computer graphics. In *Eurographics conference Grenoble*, 2004.
12. 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.
13. Dietmar Hildenbrand, Joachim Pitt, and Andreas Koch. Gaalop - high performance parallel computing based on conformal geometric algebra. In Eduardo Bayro-Corrochano and Gerik Scheuermann, editors, *Geometric Algebra Computing in Engineering and Computer Science*. Springer, May 2010.
14. Khronos. *OpenGL Specifications*, 2010. http://www.opengl.org/documentation/specs/.
15. Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
16. NVIDIA. The CUDA home page. Available at `http://www.nvidia.com/object/cuda_home.html`, 2010.
17. Christian Perwass. *Geometric Algebra with Applications in Engineering*. Springer, 2009.
18. Christian Perwass. The CLU home page. Available at http://www.clucalc.info, 2010.
19. Christian Perwass and Dietmar Hildenbrand. Aspects of geometric algebra in euclidean, projective and conformal space. Technical report, University of Kiel, 2004.
20. Florian Seybold. Gaalet - a c++ expression template library for implementing geometric algebra, 2010.
21. 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.
22. Christian Steinmetz. Optimizing a geometric algebra compiler for parallel architectures using a table-based approach. In *Bachelor thesis TU Darmstadt*, 2011.
23. Christian Steinmetz. Examination of new geometric algebras including a visualization and integration in a geometric algebra compiler. In *Master thesis TU Darmstadt*, 2013.
24. Christian Steinmetz. The gaalop guide. `http://www.gaalop.de/documentation/gaalopguide/`, 2013.