Gaalop Compiler Driver

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

Dietmar Hildenbrand
TU Darmstadt, Germany
dhilden@gris.informatik.tu-darmstadt.de

ABSTRACT

The focus of the this work is on the better integration of algorithms expressed in Conformal Geometric Algebra (CGA) in modern high level computer languages, namely C++ and NVIDIA's Compute Unified Device Architecture (CUDA). A high runtime performance in terms of CGA is achieved using symbolic optimizing through the invocation of Gaalop.

Keywords. Conformal Geometric Algebra, Compiler Driver, Runtime Performance

1 INTRODUCTION

During the last decade Conformal Geometric Algebra (CGA) has become increasingly popular in expressing solutions to geometry related problems in scientific applications of robotics, dynamics, rendering and computer vision. Video game developers are becoming aware of CGA, in search for simpler and faster ways to describe their lighting [**?**] and physics algorithms. The majority of developers makes use of C-related programming languages like C++ or CUDA [**?**], which are performant and abstract enough for most needs.

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

1.1 Conformal Geometric Algebra

Conformal Geometric Algebra (CGA) is a new way of expressing most 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, being more compact and better readable. Similar observations can be made in other applications of geometry related mathematics. Applied to computer programs, CGA therefore has a high potential for improving code readability and to shorten production cycles. It has also been proven, that if implemented right, Geometric Algebra has at least similar performance, but sometimes even better performance, than conventional approaches [**?**].

An element of Conformal Geometric Algebra is referred to as multivector. A multivector consists of a linear combination of so called blades. Blades define the basis of CGA and are combinations of the vectors e_1, e_2, e_3, e_0 and e_∞. All possible blades are listed in table 1. Keep this in mind for section 2.2.

1.2 High Level Programming Languages

Modern very high level software development tools, like Java [**?**], define a very abstract language, on which programmers and scientists can work in a natural way and with results of moderate performance. Machine level languages on the other hand, like Assembler, tend to produce very fast results, but with less intuition, which often leads to longer and more costly development cycles.

In order to shorten development time and to produce fast code at the same time, the solution lies somewhere in between. Object oriented programming languages like C++, C#, Objective Pascal and Smalltalk provide a good level of abstraction, but also excellent performance. They seem to be a good choice for most modern scientific and business projects and are therefore the most common languages, leaded by C and C++. Recently NVIDIA's Compute Unified Architecture (CUDA) programming language enabled users to utilize the very high computing power of modern graphics chips. CUDA device code is a subset of the common C language with some extensions added to it.

index	blade	grade
1	1	0
2	e_1	1
3	e_2	1
4	e_3	1
5	e_∞	1
6	e_0	1
7	$e_1 \wedge e_2$	2
8	$e_1 \wedge e_3$	2
9	$e_1 \wedge e_\infty$	2
10	$e_1 \wedge e_0$	2
11	$e_2 \wedge e_3$	2
12	$e_2 \wedge e_\infty$	2
13	$e_2 \wedge e_0$	2
14	$e_3 \wedge e_\infty$	2
15	$e_3 \wedge e_0$	2
16	$e_\infty \wedge e_0$	2
17	$e_1 \wedge e_2 \wedge e_3$	3
18	$e_1 \wedge e_2 \wedge e_\infty$	3
19	$e_1 \wedge e_2 \wedge e_0$	3
20	$e_1 \wedge e_3 \wedge e_\infty$	3
21	$e_1 \wedge e_3 \wedge e_0$	3
22	$e_1 \wedge e_\infty \wedge e_0$	3
23	$e_2 \wedge e_3 \wedge e_\infty$	3
24	$e_2 \wedge e_3 \wedge e_0$	3
25	$e_2 \wedge e_\infty \wedge e_0$	3
26	$e_3 \wedge e_\infty \wedge e_0$	3
27	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$	4
28	$e_1 \wedge e_2 \wedge e_3 \wedge e_0$	4
29	$e_1 \wedge e_2 \wedge e_\infty \wedge e_0$	4
30	$e_1 \wedge e_3 \wedge e_\infty \wedge e_0$	4
31	$e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	4
32	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5

Table 1: The 32 blades of 5D Conformal Geometric Algebra, that compose a multivector.

We strongly believe, that the applications of Conformal Geometric Algebra are most likely to be found in high performance applications, such as games, industrial or scientific software. Since, as described above, C-like languages are leading the field of high performance computing, possible integration of CGA code into C/C++/CUDA has the most advantage.

2 RELATED WORK

Combining both the aspects of Conformal Geometric Algebra and Modern Programming Languages (namely C++ and CUDA), promises to have a high potential for scientific work. Unfortunately CGA has such a high level of abstraction, that it does not naturally fit into C++ and CUDA programs. In order to solve this problem and to make CGA run fast, recent approaches try to wrap CGA into templated multivector classes (Gaalet [**?**]) or make use of a domain specific language (DSL) as input language for a code generator (Gaigen2 [**?**], GMac [**?**] and Gaalop [**?**]). All software tools are very well suited in their domain and produce good results. In the following, we will present the CLUScript language, Gaalop and the Compiler Driver concept.

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 a completely new domain specific language (DSL).

The DSL that powers this work is CLUScript. The especially designed integrated development environments for CLUScript are called CLUCalc (old) and CLUViz (new). In words of the author Dr. Christian Perwass [**?**, **?**].

> 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 or Macintosh platforms is very limited, but it may run with some effort.

2.2 Gaalop

The Geometric Algebra Algorithms Optimizer (Gaalop [**?**]) was developed by TU Darmstadt (Germany) and is a powerful tool for optimizing algorithms, expressed in Conformal Geometric Algebra. It generates non CGA-specific code from code defined in a CGA-specific language and symbolically optimizes the algorithm on-the-fly, invoking a Computer Algebra System (CAS). In this context, CGA can be seen as a higher level mathematical language that is being transformed into simple arithmetic mathematical language 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 Field Programmable Array (FPGA), LaTex and CLUScript representations. For evaluation purposes it is often helpful

to choose CLUScript output, then replace the original CLUScript code with the optimized code, and test the result for the same functionality as the original code. Generated Gaalop C/C++ code has to be pasted into the final code and requires additional handwork.

2.3 Compiler Drivers and CUDA

The Compiler Driver concept is a very simple, but powerful approach to extend the features of a programming language. It has recently been used by NVIDIA's CUDA [**?**]. We will use CUDA as an example to explain the usage of compiler drivers. A traditional C++ program is separated into several source files. The source files will then be converted to an intermediate form by the C++ compiler. This intermediate form is called an object file. All the object files are then linked together by the linker, resulting in the final executable file. For instance, see the following example in figure 1.

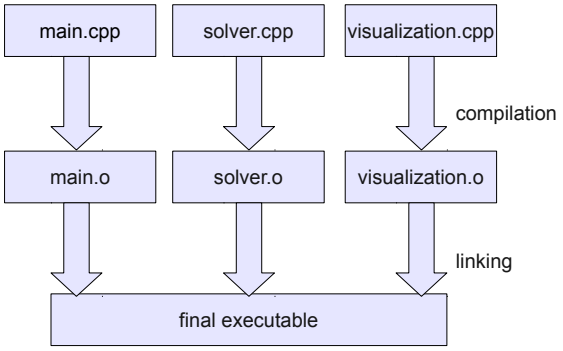

Figure 1: C++ compilation process.

The language syntax of CUDA is fully compatible with C++. One may compile any C++ code with NVIDIA's NVCC compiler without any modifications to the original code. The produced machine code, however, will still run on the host.

In order to make use of the graphics chip's high number of streaming processors, we will have to include CUDA-specific language extensions like the *__global__* or *__device__* keywords and kernel call statements, which are not part of the original C++ language. The compiled object file will seamlessly link with the code compiled with the regular C++ compiler. To make this clear, see the simplified diagram in figure 2. The solver.cu source file is CUDA code, all other files are C++ code.

Notice how the CUDA compiler seamlessly integrates with the C++ compiler. With the *__global__* and *__device__* and other keywords NVIDIA has extended the C++ language syntax. But instead of creating their own C++ compiler from scratch, they came up with the approach of reusing the existing C++ compiler and linker. This is called the Compiler Driver concept.

From the NVIDIA CUDA Programming Guide 3.0 [**?**].

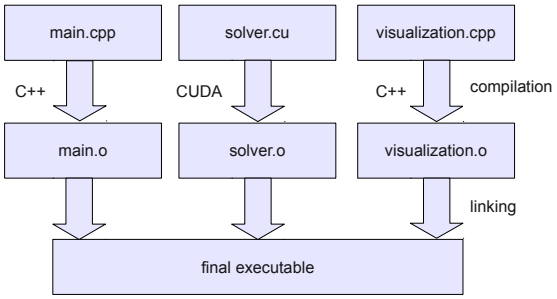

Figure 2: Simplified CUDA compilation process.

Source files compiled with nvcc can include a mix of host code (i.e. code that executes on the host) and device code (i.e. code that executes on the device). nvcc's basic workflow consists in separating device code from host code and compiling the device code into an assembly form (PTX code) and/or binary form (cubin object). The generated host code is output either as C code that is left to be compiled using another tool or as object code directly by letting nvcc invoke the host compiler during the last compilation stage.

This way CUDA makes full usage of the existing C++ compiler and linker features, extends them, but separates both compilers at the same time. This enables lower complexity and better maintenance of NVCC. The full diagram can be seen in figure 3.

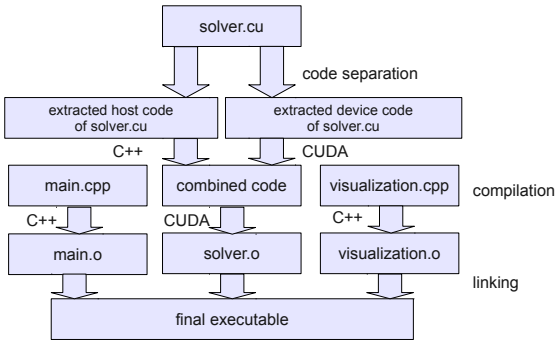

Figure 3: Full CUDA compilation process.

3 GAALOP COMPILER DRIVER

We have presented Conformal Geometric Algebra, CLUScript, Gaalop and C++. All these aspects are somehow related to each other, but simply not linked yet. This is where the Compiler Driver concept comes in and elegantly connects all parts. The diagram in figure **??** shows, how this will be achieved in particular.

We now use the file extension .gcp for CLUScript-extended C++ source files. The particular steps, that occur when compiling .gcp-files, are the following.

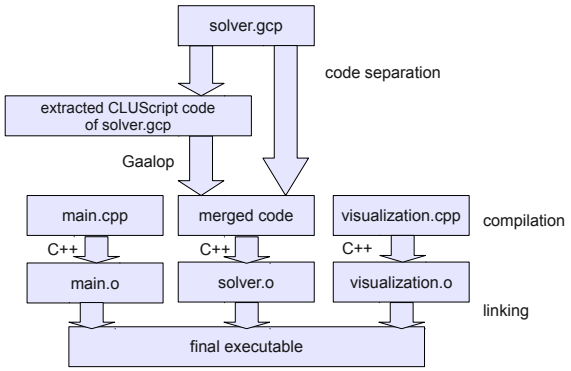

Figure 4: GCD C++ compilation process.

1. The user issues the build command. This can happen in an integrated development environment (IDE) using custom build rules, as well as using *GNU make* [**?**] or other build automation tools.

2. The build tool passes .gcp-files to Gaalop Compiler driver (GCD) over the command line.

3. GCD extracts the CLUScript parts of the .gcp-file and writes them into separate files.

4. GCD invokes Gaalop over its new command line interface, passing the extracted code files, one at a time.

5. Gaalop symbolically simplifies the extracted code files.

6. GCD merges the returned code into with the original code, exactly where the *pragmas* are.

7. GCD invokes the regular C++ compiler passing the merged C++ file.

8. Finally, the C++ compiler produces an object file, which seamlessly integrates into the linking process.

3.1 Gaalop Compiler Driver for CUDA

Note that the concept is not restricted to C++. It can be applied to CUDA or other programming languages in the same manner. The resulting diagram in figure **??** for CUDA is slightly more complex, as GCD passes its data to NVCC, which itself is a compiler driver.

We choose the file extension .gcu for CLUScript-extended CUDA programs. The compilation process steps remain the same as with C++ GCD, with the exception of passing the generated file to NVCC instead of the C++ compiler in steps 7 and 8.

4 A GUIDE TO GAALOP GCD

The following section shows, how to make use of GCD in real world-applications. It is intended as a quick start-guide, described by three example code snippets.

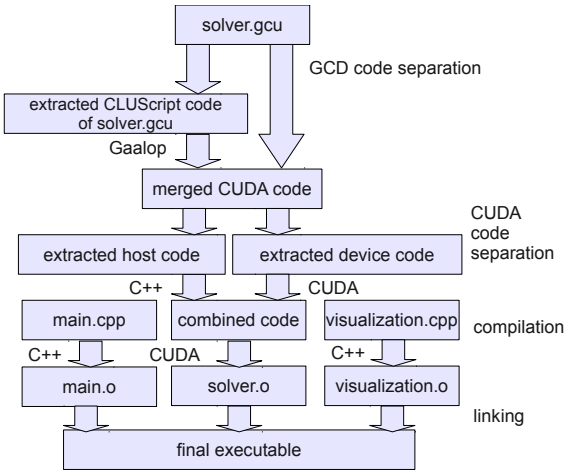

Figure 5: GCD CUDA compilation process.

4.1 The Test Case

The code in the following listings was extracted from a Molecular Dynamics Simulation currently under development by us and the High Performance Computing Center Stuttgart (HLRS) and is partially optimized with Gaalop and GCD. A molecular dynamics simulation models the point-pair interactions of a system of molecules, each one consisting of several atoms, and numerically solves Newton's and Euler's equations of motion for each molecule.

The aim of the project is a runtime comparison between a conventional solver and several implementations of a new formalism based on Hestenes' work on screw mechanics described in CGA [**?**], including a compact formulation of combined translational and rotational dynamics within a velocity verlet algorithm. Details of the Molecular Dynamics formalism used as a test case in the present work is intended to be part of a future publication. The CGA solvers were implemented with Gaalet, Gaalop and GCD running on the CPU, as well as Gaalop and GCD running on CUDA.The listings show code extracted from both the CPU and CUDA versions of the Gaalop and GCD solvers.

4.2 Code examples for C++

The whole simulation was firstly implemented in CLUCalc using the CLUScript language and later ported to C++ using Gaalop. Listing 1 shows the initialization of a particular molecule, taken from the original CLUScript. Location and orientation are defined through the molecule's versor *D_result* (refer to [**?**]). Linear and angular velocity are defined through the molecule's velocity screw *V_result*. The consecutive simulation steps start with the values computed in this initialization code.

```
// create versor from input values
rotor = arw + arx*e2^e3 + ary * e3^e1 + arz * e1^e2;
translator = 1 - 0.5*(lpx*e1 + lpy*e2 + lpz*e3)^einf;
?D_result = translator*rotor;
```

```
// create velocity screw from input values
lv = lvx*e1+lvy*e2+lvz*e3;
av = avx*e1+avy*e2+avz*e3;
?V_result = einf*lv - e1^e2^e3*av;
```

Listing 1: Original CLUScript input for Gaalop.

Note that *D_result* and *V_result* are already declared for export in Gaalop indicated by the question marks. The resulting Gaalop output code is then directly pasted into the target C++ file, as can be seen in listing 2.

```
// map molecule data to gaalop data
const float lpx = molecule.lpos[0];
const float lpy = molecule.lpos[1];
const float lpz = molecule.lpos[2];
const float arw = molecule.arot[0];
const float arx = -molecule.arot[1];
const float ary = -molecule.arot[2];
const float arz = -molecule.arot[3];
const float lvx = molecule.lvel[0];
const float lvy = molecule.lvel[1];
const float lvz = molecule.lvel[2];
const float avx = molecule.avel[0];
const float avy = molecule.avel[1];
const float avz = molecule.avel[2];

// gaalop generated code
float D_result_opt[32] = {0.0f};
D_result_opt[1]=arw;
D_result_opt[7]=arz;
D_result_opt[8]=-ary;
D_result_opt[9]=0.5*lpy*arz-0.5*lpx*arw-0.5*lpz*ary;
D_result_opt[11]=arx;
D_result_opt[12]=0.5*lpz*arx-0.5*lpy*arw-0.5*lpx*arz;
D_result_opt[14]=-0.5*lpz*arw+0.5*lpx*ary-0.5*lpy*arx;
D_result_opt[27]=-0.5*lpy*ary-0.5*lpz*arz-0.5*lpx*arx;

// gaalop generated code
float V_result_opt[32] = {0.0f};
V_result_opt[7]=-avz;
V_result_opt[8]=avy;
V_result_opt[9]=-lvx;
V_result_opt[11]=-avx;
V_result_opt[12]=-lvy;
V_result_opt[14]=-lvz;

// map gaalop data to molecule data
GaalopMapVersor(D, D_result_opt);
GaalopMapVelocityScrew(V, V_result_opt);
```

Listing 2: Merged Gaalop and C++ code.

Notice that the result code not only contains the generated Gaalop code, but also several variable and array assignments. Those assignments are data mappings between the original molecule data structure and the generated Gaalop code. They are quite common for most Gaalop-powered applications.

The function *GaalopMapVersor* assigns the elements 1,7,8,9,11,12,14,27 of array *D_result_opt* to the elements 0,1,2,3,4,5,6,7 of array *D*, that was previously declared with size 8. The function *GaalopMapVelocityScrew* assigns the elements 7,8,9,11,12,14 of array *V_result_opt* to the elements 0,1,2,3,4,5 of array *V*, that was previously declared with size 6. The code in between the two mapping blocks is the actual Gaalop output code.

One may modify the array indices and variable names by hand to improve speed and to avoid data mappings. For reasons of transparency this is usually not a good choice, meaning that if we might discover a bug or we would like to include a new feature in one of our CLUScript files, we again have to modify the generated code by hand. As there might be a large number of Gaalop-generated code snippets in a GGA-powered C++ program, this process will increase the probability of bugs and development time. Using data-mappings enables us to re-paste modified code at any time.

Also notice that even without the data mappings, the generated code is hardly interpretable by human means. Keeping this in mind, review the following code.

```
    // map molecule data to gaalop data
    ...

#pragma gcd begin
  // create versor from input values
  rotor = arw + arx*e2^e3 + ary * e3^e1 + arz * e1^e2;
  translator = 1 - 0.5*(lpx*e1 + lpy*e2 + lpz*e3)^einf;
  ?D_result = translator*rotor;

  // create velocity screw from input values
  lv = lvx*e1+lvy*e2+lvz*e3;
  av = avx*e1+avy*e2+avz*e3;
  ?V_result = einf*lv - e1^e2^e3*av;
#pragma gcd end

    // map gaalop data to molecule data
    GaalopMapVersor(D, D_result);
    GaalopMapVelocityScrew(V, V_result);
```

Listing 3: Gaalop Compiler Driver for C++ input code.

Note: The preceding variable mappings were removed in order to keep the size of this document small. Please consider them to be in place when evaluating the code.

The CLUScript code is now directly embedded in the C++ code in between the *gcd pragmas*, instead of the pasted Gaalop code. As result, this reduces the source code size and makes it much better readable.

Since the code now contains CLUScript statements, which are apparently not part of the C++ standard, we will not be able to compile it with a regular C++ compiler. To be specific, the C++ standard is being extended using the Compiler Driver concept, as stated in section 3.

4.3 Code example for CUDA

The example above was chosen, because it includes a lot of CGA-statements and is easy to understand. It is also possible to include this code into a CUDA-Kernel, but not meaningful here. The code shown is only called once for each molecule before the simulation, and never called again. Wisely chosen CUDA-Kernels are called one or many times per frame, as the one in listing 4. Again, it is taken from the original simulation, with some parts removed.

```
__device__ void addMoleculeForceAndTorque(
        float* mol_lmom,
        float* mol_amom,
        const float* versor1,
        const float3& localPos,
        const float3& globalForce,
        ...)
{
// map molecule data to gaalop data
```

```
const float Di = versor1[0];
const float D12 = versor1[1];
const float D13 = versor1[2];
const float D1x = versor1[3];
const float D23 = versor1[4];
const float D2x = versor1[5];
const float D3x = versor1[6];
const float D123x = versor1[7];

const float px = localPos.x;
const float py = localPos.y;
const float pz = localPos.z;

const float fgx = globalForce.x;
const float fgy = globalForce.y;
const float fgz = globalForce.z;

#pragma gcd begin
  // input values
  moleculeVersor = Di + D23*e2^e3 + D13*e1^e3
                      + D12*e1^e2 + D1x*e1^einf
                      + D2x*e2^einf + D3x*e3^einf
                      + D123x*e1^e2^e3^einf;
  posLocal = px*e1 + py*e2 + pz*e3;
  forceGlobal = fgx*e1 + fgy*e2 + fgz*e3;

  // final values
  ?result_force = ~moleculeVersor
                      * forceGlobal
                      * moleculeVersor;
  ?result_torque = posLocal ^ result_force;
#pragma gcd end

// add resulting force and torque to molecule's data
...
}
```

Listing 4: Gaalop Compiler Driver for CUDA input code.

5 RESULTS

5.1 Performance and Compile Time

Figure 6 shows the performance of the GCD solver versus the Conventional solver for the Test Case in section **??** on a quadcore machine. The GCD solver is slightly faster or equally fast compared to the Conventional solver, which is not self-evident for CGA-based implementations of such complexity. Additionally, the GCD solver has a more compact code, as described in section 4.1. Notice, that section 6 shows ways to achieve more than two times faster results, by removing unused memory and caching artifacts (figure 7).

Compilation time is 1.920s for the Conventional versus 10.426s for the GCD solver. The prolonged compile time is due to the Gaalop-performed symbolic optimizations in the background.

5.2 CMake Support

Development with most programming languages, especially C++, is highly dependent on specifying build logic. Build logic explicitly defines which source files need to be compiled with which tool, and how the resulting intermediate files get linked together into the final executable or library file. Integrated development environments (IDE) like *Microsoft Visual Studio* [**?**] or *Code::Blocks* [**?**] automatically manage the default parts of the build logic.

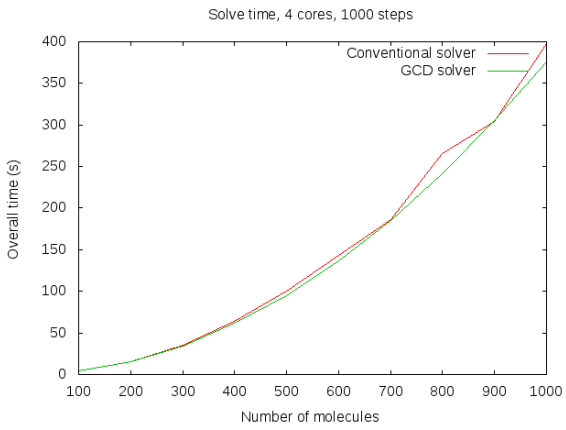

Figure 6: Performance Results - GCD solver is slightly faster or equally fast compared to Conventional solver. Much higher Performance is achieved in figure 7, Future Work section 6.

However, with a rising number of operating systems, compilers and build tools, it has become very difficult to maintain the build logic for every possible combination of operating system and compiler. *CMake* [**?**] elegantly solves this problem by acting as a build logic generator. More detailed, *CMake* defines a script language, that is independent of the build platform. This script language, is then transformed into the target platform definition, e.g. **.sln* project files for *Visual Studio* or *Makefiles* for *GNU make* [**?**].

CMake is rapidly becoming the de facto standard for cross platform build tools. It also supports automatic unit testing (*CTest*), install and deploy mechanisms (*CPack*), and web-based error reporting (*CDash*).

CMake support for GCD is provided by a CMake-script named *FindGCD.cmake*. If the script is installed in CMake's *Modules* subdirectory, it can be invoked by CMake's *FIND_PACKAGE(GCD)* command. Libraries and executables containing GCD code may be built using *GCD_CPP_ADD_LIBRARY* and *GCD_CPP_ADD_EXECUTABLE* commands. GCD for CUDA builds use *GCD_CUDA_ADD_LIBRARY* and *GCD_CUDA_ADD_EXECUTABLE*.

An example *CMakeLists.txt* build script is shown in listing 5.

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
FIND_PACKAGE(GCD)
GCD_CPP_ADD_EXECUTABLE(test1 "Test1_PointTriangle.gcp")
ADD_TEST(NAME "Test1_PointTriangle" COMMAND test1)
```

Listing 5: Example CMake build script using GCD.

Given this definition, *CMake* compiles and links the GCD source file *"Test1_PointTriangle.gcp"* into an application *test1*, and specifies it as a runtime test. *CTest* runs the executable and reports it as PASSED, if its return value is zero, or FAILED, if it is non-zero. Tests like this one are an important part of software quality assurance.

5.3 GCD helper library

The goal behind GCD helper library is to assist users of GCD by providing essential functions, that simplify development of CGA-powered applications. It is intended as a multi-purpose library, adaptable to a broad range of applications working with Conformal Geometric Algebra. Reoccurring tasks, like the mapping of versors (*GaalopMapVersor* from section **??**), are implemented within the library. It also provides C++-macros, that state the position of a particular multivector entry, e.g. $e1 \wedge e2$ is listed as *"#define E12 6"*. For example, *D[E12]* returns the blade $e1 \wedge e2$ of a multivector *D* (see table 1 for a refresh of the 32 blades of CGA).

The GCD helper library is automatically linked when using *CMake*, as described in section 5.2. However, the library's header files have to be included with *"#include <gcd.h>"* in **.gcp* (GCD for C++) or **.gcu* (GCD for CUDA) source files.

6 FUTURE WORK

The presented approach still has a lot of potential to improve on. Further work has to be put into simplifying data mappings and advancing memory usage. Gaalop automatically allocates arrays of 32 floating point numbers to make space for all possible multivector entries, which results in a memory usage of 128 bytes per multivector (see section **??**). Most multivectors usually contain about up to 8 entries, which results in about 24 unused multivector entries and 96 bytes of unneeded memory. It is not trivial to reduce the required space, because we must deal with the theoretical assumption, that all multivector entries could be assigned.

Using data mappings, the effect can be hidden, and no useless data will be saved and read from RAM, but it still occupies register space and cache, which has an effect on performance. This turned out to to be a major bottleneck in our Molecular Dynamics simulation.

Listing 6 shows an outlook of how future GCD code may look like.

```
    // map molecule data to gaalop data
    ...

#pragma gcd begin
  // create versor from input values
  rotor = arw + arx*e2^e3 + ary * e3^e1 + arz * e1^e2;
  translator = 1 - 0.5*(lpx*e1 + lpy*e2 + lpz*e3)^einf;
  ?D = translator*rotor;

  // create velocity screw from input values
  lv = lvx*e1+lvy*e2+lvz*e3;
  av = avx*e1+avy*e2+avz*e3;
  ?V = einf*lv - e1^e2^e3*av;
#pragma gcd end
```

Listing 6: Future GCD code without subsequent data mappings.

Notice that *D* and *V* do not need to be saved into temporary arrays *D_result* and *V_result* anymore. They are directly stored into the final arrays, saving additional copy time, register and cache usage. Preliminary tests on this subject reduced the runtime of our test case down to 36 percent of its original value and are very promising (figure 7).

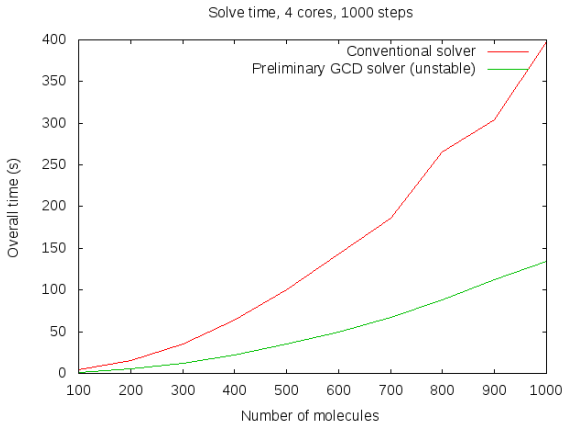

Figure 7: Future Performance Results

Ongoing work is being put into symbolically optimizing larger parts of the CLUScript-syntax with Gaalop. For example, *while-loops* can be unrolled and symbolically treated in the same way as other code.

Apart from C++ and CUDA, other languages like OpenCL [**?**], Microsoft DirectCompute and shading languages (CG, HLSL) are interesting target languages for GCD and promising topics for further research.

The GCD helper library will be expanded in future work to support direct rendering of multivectors similar to CLUCalc (see figure 8 for example). That is, given a particular multivector *m*, the helper library 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, the library will render the appropriate object with OpenGL [**?**] or other rendering APIs.

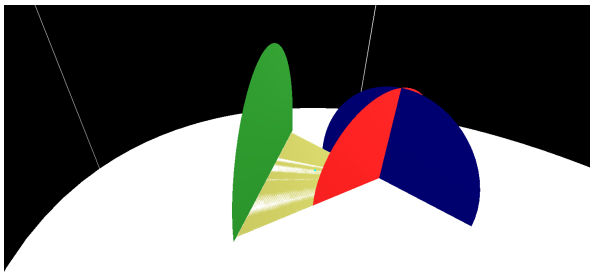

Figure 8: An example of CLUCalc generated graphics

7 CONCLUSION

Code simplicity, elegance and intuitiveness are the major goals of this work. Recalling the code examples shows that these goals were reached. As GCD directly profits from any improvements within Gaalop through

its invocation, a high runtime performance is achieved on-the-fly.

Gaalop GCD symbolically optimizes the embedded CLUScript code in order to improve runtime. 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 in many build automation tools like *GNU make* [**?**]. We found, that in reality, using parallelized builds, longer compile time is not a problem.

We would like to conclude, that Gaalop Compiler Driver has the potential to change the way programmers work with Conformal Geometric Algebra inclusions in their code. Instead of separating code generation and code compilation into two distinct processes, it is now a single simplified process. Especially the combination of CGA and CUDA enables new methods for research. As it is now easier to develop with, we hope that more scientists, game and software programmers will find their way into the applications of Conformal Geometric Algebra.