Geometric Algebra enhanced Precompiler for C++ and OpenCL

Master-Thesis von Patrick Charrier aus Darmstadt
März 2012

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Geometric Algebra Computing
Dr. Ing. Dietmar Hildenbrand

Geometric Algebra enhanced Precompiler for C++ and OpenCL

Vorgelegte Master-Thesis von Patrick Charrier aus Darmstadt

Gutachter: Dr. Ing. Dietmar Hildenbrand

Tag der Einreichung: 29. März 2012

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 29. März 2012

(P. Charrier)

Table of Contents

List of Figures

List of Tables

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 [3] and physics algorithms. The majority of developers makes use of C-related programming languages like C++, OpenCL [20] or CUDA [22], 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 [17] in the background, an integration sets new standards to 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 Quick Start

To quickly start programming with Gaalop Precompiler (Gaalop GPC) , the precompiler developed in this thesis, it is a good idea to start by reading the Language Specification in chapter 5, and to continue with the Start Guide in appendix A. A basic knowledge of GA and CLUScript is however required to understand this thesis in full detail. Refer to [24] for an introduction to both topics.

1.2 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, being more compact and better readable. Similar observations can be made in other applications of geometry related mathematics. Applied to computer programs, GA 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 [18].

Multivectors of Conformal Geometric Algebra

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 and their grading are listed in table 1.2.

1.3 Higher Dimensional Algebras

The paper [30] introduced a new algebra called the G6,3 Algebra that supports features far beyond of what is possible with Conformal Geometric Algebra. Mathematical objects such as ellipsoids, cylinders, quadrics, and 1D-quadratic strings are represented as multivectors, much like spheres, planes, circles, points and point-pairs are represented in CGA. All the geometric objects contained in Conformal Geometric Algebra and all operations on multivectors, such as translation, rotation, reflection, scaling and even intersection using the outer-product $\wedge$ are also representable in G6,3 Algebra, along with some additional operations like non-uniform scaling. We strongly believe that this algebra and even higher dimensional algebras, not subject to research yet, have even more potential than the well-known Conformal Geometric Algebra and lesser dimensional algebras. This is why support for the G6,3 Algebra has been integrated into Gaalop by [27], and the reason that this functionality is also a substantial factor in the design of

blade	grade
1	0
e_1	1
e_2	1
e_3	1
e_∞	1
e_0	1
$e_1 \wedge e_2$	2
$e_1 \wedge e_3$	2
$e_1 \wedge e_\infty$	2
$e_1 \wedge e_0$	2
$e_2 \wedge e_3$	2
$e_2 \wedge e_\infty$	2
$e_2 \wedge e_0$	2
$e_3 \wedge e_\infty$	2
$e_3 \wedge e_0$	2
$e_\infty \wedge e_0$	2

blade	grade
$e_1 \wedge e_2 \wedge e_3$	3
$e_1 \wedge e_2 \wedge e_\infty$	3
$e_1 \wedge e_2 \wedge e_0$	3
$e_1 \wedge e_3 \wedge e_\infty$	3
$e_1 \wedge e_3 \wedge e_0$	3
$e_1 \wedge e_\infty \wedge e_0$	3
$e_2 \wedge e_3 \wedge e_\infty$	3
$e_2 \wedge e_3 \wedge e_0$	3
$e_2 \wedge e_\infty \wedge e_0$	3
$e_3 \wedge e_\infty \wedge e_0$	3
$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$	4
$e_1 \wedge e_2 \wedge e_3 \wedge e_0$	4
$e_1 \wedge e_2 \wedge e_\infty \wedge e_0$	4
$e_1 \wedge e_3 \wedge e_\infty \wedge e_0$	4
$e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	4
$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5

Tabelle 1.1.: The 32 blades of 5D Conformal Geometric Algebra, that a multivector is composed of.

Gaalop Precompiler. Gaalop and Gaalop Precompiler support higher algebras without major performance decreases, only at the cost of longer compile times.

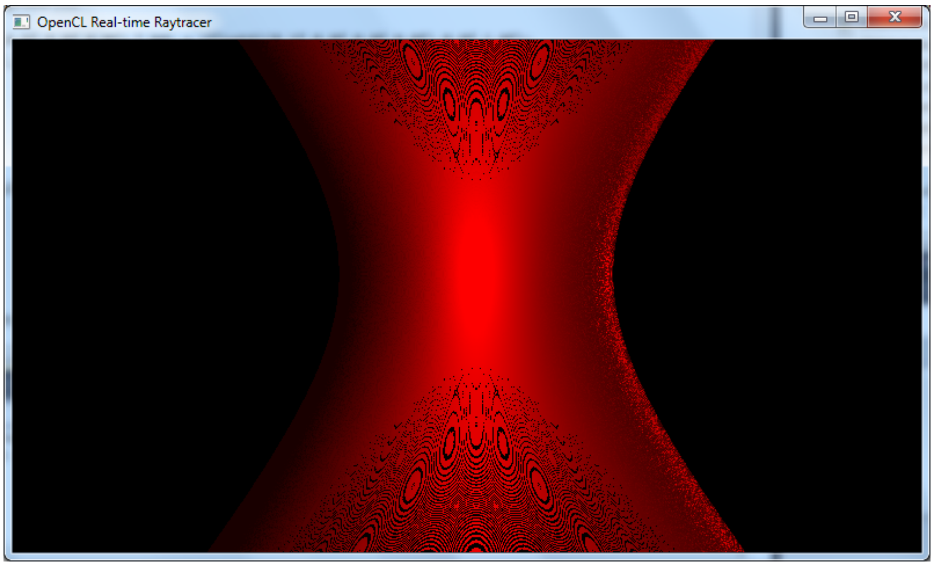


Abbildung 1.1.: A hyperboloid in G6,3 Algebra

1.4 High Level Programming Languages

Modern very high level software development tools, like Java [28], 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, lead by C and C++.

Recently NVIDIA's Compute Unified Device Language (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.

The new computing language standard named OpenCL, created by the independent Khronos Group, is being supported by a broad range of devices and offers the most promising features.

We strongly believe, that the applications of 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 GA-code into C/C++/OpenCL/CUDA has the most advantage.

1.5 Contributions of this work

The foundations of this work were made in the paper [9] and the thesis [8]. The project was called Gaalop Compiler Driver (Gaalop GCD) at the time. While the concept reached its goal of integrating GA directly into high performance applications, it was much more a proof-of-concept than a final solution and it was far less than complete from a user friendliness point of view. The primary goal of this work is therefore the improvement of usability and stability. By rethinking a former method, runtime performance and efficiency was also slightly improved (section 7.8 and section 9.7). Other influences such as higher dimensional algebras (see paper [30] and section 1.3) and the new GAPP language have also been taken into account. CMake functionality has been further extended by making the most important internal Gaalop settings controllable through CMake, and by improving the build logic in general.

Work on the Geometric Algebra Parallelism Programs language compiler middle-end started with [27], but no backend that took advantage of this work was established yet. A secondary goal of this work was therefore the implementation of such a backend based on OpenCL and testing its performance.

The molecular dynamics simulation [26], described in detail in chapter 9, has been a useful testing and comparison environment between conventional Linear Algebra and Geometric Algebra implementations in the past and is reused for the same purpose by this work. It is a good indicator for performance and numerical stability issues, and bugs in general. It is very complex and several implementations in C++, OpenCL, and CUDA exist. It was required to rewrite and test huge amounts of code to adapt it to the new language, and to make full use of the new Interface Feature defined in chapter 7. The OpenCL implementation also offers the possibility to evaluate both GAPP and non-GAPP compiled code.

A realtime OpenCL raytracer (chapter 10) was implemented as a use-case for the evaluation of Gaalop GPC with higher dimensional algebras and the new GAPP OpenCL backend.

2 Related Work

Combining both the aspects of Conformal Geometric Algebra and Modern Programming Languages (namely C++, OpenCL 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++, OpenCL or CUDA programs. In order to solve this problem and to make GA-based implementations faster, recent approaches try to wrap GA into template multivector classes (Gaalet [25]) or make use of a Domain Specific Language (DSL) as input language for a code generator (Gaalop [17], Gaigen2 [12] and GMac [11]). All software tools are very well suited in their domain and produce good results.

2.1 Performance of Geometric Algebra in comparison to conventional approaches

The 2006 paper Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. [18] 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 shrink significantly while the stability constantly improved.

The 2010 molecular dynamics simulation, described in chapter 9 and in the initial project documentation [26], showed slightly better performance of a Geometric Algebra solver compared to a conventional solver.

2.2 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 environments for CLUScript are called CLUCalc (old) and CLUViz (new), and are freely available at [24]. In words of the author Dr. Christian Perwass [23, 24]:

> 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 Windows platforms. Unfortunately, the support for Linux and Macintosh platforms is very limited, but it may run with some effort.

2.3 OpenCL

Shortly after General Purpose Computing on Graphic Cards (GPGPU) became increasingly popular, the need for an open standard arose. One could of course use pre-existing technologies like NVIDIA CUDA or ATI Close To Metal, but this would mean, that every Compute Application had to be specialized for every possible device and its respective software library. Therefore, the proposal for the open computing standard OpenCL [20] was brought to Khronos Group by Apple Computers. Khronos Group founded a working group, in which industry leaders like AMD, Intel, NVIDIA and Qualcomm negotiated a specification. The specification was finished November 18, 2008 and reviewed internally by Khronos members until December 8, 2008, when it was released to the public. It took another few months until the industry finished their implementations, released Software Development Kits and fixed the initial bugs, but at the moment of writing, it is time to actively start using OpenCL in compute intensive applications. So far, AMD, NVIDIA, Apple and Intel have released OpenCL SDKs.

2.4 Gaalop as the foundation of this work

The Geometric Algebra Algorithms Optimizer (Gaalop) [17] 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 a GA-specific language 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 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.

With recent work [27], Gaalop is no longer dependent on Maple. It can optimize CLUScript code 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.

2.5 Alternatives to Gaalop GPC

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

2.5.1 Gaigen

Gaigen [12] is being implemented by Daniel Fontijne at the University of Amsterdam. At the time of writing, it is in its third major version and is being 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 [11] 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.

2.5.2 GMac

GMac [11] is being 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, for such 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.

2.5.3 Gaalet

Gaalet [25] 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 less than that of Gaalop [26] and 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 can not install a lot of dependencies, such as the dedicated machines of the High Performance Computing Centre (HLRS) in Stuttgart, where Gaalet originates from.

2.5.4 Requirements Evaluation

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

- Gaigen2 in its current form requires user interaction to get the code generation and feedback mechanism running. This collides with the no-user interaction policy of precompilers entirely. It can therefore 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 between a variety of languages, instead of being focused on one language in particular.
- Gaalet is 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, it is therefore very likely to assume, those tools will ever be able to cope with much simpler languages like C, OpenCL, CUDA or Java. Since especially GPU-Computing languages like OpenCL and CUDA promise to have an even more important role in the future, we can not ignore them for the purpose of Geometric Algebra Computing. This is a major problem with all approaches above.

3 Previous Work

Some functionality used in Gaalop Precompiler is not entirely new. Gaalop Compiler Driver already defined some functionality that is being extended by Gaalop GPC. We briefly explain this functionality, and show how it had to be adapted to fit into the advanced concepts of Gaalop GPC.

3.1 The state of Gaalop Compiler Driver

The previous work on Gaalop Compiler Driver (Gaalop GCD) showed the successful integration of Geometric Algebra code into modern high level programming languages. However, there was still potential to make development easier and friendlier.

As a demonstration, consider the following example.

```
void SolverMolecule::convertFrom(const Molecule& molecule) {
        mass = molecule.mass;
        I_1 = molecule.I1;
        I_2 = molecule.I2;
        I_3 = molecule.I3;
        atomIndices = molecule.atomIndices;

        //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];

#pragma gcd begin
        rotor = arw + arx * e2 ^ e3 + ary * e3 ^ e1 + arz * e1 ^ e2;
        translator = 1 - (0.5 * lpx * e1 ^ einf + 0.5 * lpy * e2 ^ einf + 0.5 * lpz
                     * e3 ^ einf);
        ?Din = translator*rotor;

        lv = lvx * e1 + lvy * e2 + lvz * e3;
        av = avx * e1 + avy * e2 + avz * e3;

        ?Vin = einf*lv - e1^e2^e3*av;
#pragma gcd end

        // map gaalop data to molecule data
        D0[0] = Din_SCALAR;
        D0[1] = Din_E12;
        D0[2] = Din_E13;
        D0[3] = Din_E1INF;
        D0[4] = Din_E23;
        D0[5] = Din_E2INF;
        D0[6] = Din_E3INF;
        D0[7] = Din_E123INF;

        V0[0] = Vin_E12;
        V0[1] = Vin_E13;
        V0[2] = Vin_E1INF;
        V0[3] = Vin_E23;
        V0[4] = Vin_E2INF;
        V0[5] = Vin_E3INF;
}
```
Listing 3.1: Gaalop Compiler Driver for C++ input code.

The above code snippet is taken from the Molecular Dynamics simulation described in deep detail in chapter 9. Firstly, it constructs a versor Din out of the previously declared rotor with the name rotor and a translator, name of translator. The components of rotor and translator are taken from the C/C++-arrays molecule.lpos, molecule.arot respectively. A versor is a multivector in Conformal Geometric Algebra that describes a transformation in three-dimensional space. A versor is similar to a matrix, for that it is able to translate, rotate, or scale a geometric object. Secondly, the velocity screw Vin is constructed out of the multivector components contained in molecule.lvel, and molecule.avel. A velocity screw is a multivector that defines linear and angular velocity of a geometric object. The integral of a velocity screw is a versor.

Notice the large assignment blocks before #pragma gcd begin and after #pragma gcd end. Their purpose is the explicit transfer of data between multivectors and arrays. Much of this code is purely mechanical and increases the code size. The automation of this purely mechanical work is a part of this work and is described in full detail in chapter 5.

The Scoping feature allows multivectors from one #pragma gpc-block to be accessed in other #pragma gpc-blocks.

The following code is valid Gaalop GPC-syntax:

```
#pragma gpc 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
#pragma gpc end
```
Listing 3.2: Simplified way of reusing multivectors from previous #pragma-blocks.

The solution guarantees correct scoping. For example, the following listing will cause a compilation error:

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

... // some code

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

Listing 3.3: Multivectors are not available in different scopes.

Whereas outer scopes are imported into inner scopes as usual (listing 3.4). The scoping rules work in a way a programmer would expect them to work, without any knowledge of the underlying concept.

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

                ... // some C++ code

                { // begin inner scope
#pragma clucalc begin // block B
                        ?b = a + 10; // Will work as expected.
#pragma clucalc end
                } // end inner scope
        } // end outer scope
#pragma gpc end
```

Listing 3.4: Outer scope multivectors are handled as expected.

3.3 Compressed Multivector Storage

The naive approach to multivector storage is to save all multivector blade coefficients in one array sequentially, including the ones equal to zero. This leads to a non-optimal memory and cache efficiency, ultimately with higher dimensional algebras. A higher efficiency is achieved by only storing the non-zero entries of a multivector in one array sequentially.

An example output, including all meta-info, may then look like listing 3.5. Note that these aspects are internal details. It is not necessary to understand this listing in order to start programming with Gaalop GPC. The listing is intended for those who wish to understand the internal works of Gaalop GPC.

```
//#pragma gpc multivector V0_t_dt
float V0_t_dt[6];
//#pragma gpc multivector V1_t_dt
float V1_t_dt[6];

//#pragma gpc multivector_component V1_t_dt e1^e2 V1_t_dt[0]
V1_t_dt[0] = ((I_2 - I_1) * V013 * V023 - am12) / I_3;
//#pragma gpc multivector_component V1_t_dt e1^e3 V1_t_dt[1]
V1_t_dt[1] = ((I_3 - I_1) * V012 * V023 + am13) / I_2;
//#pragma gpc multivector_component V1_t_dt e1^einf V1_t_dt[2]
V1_t_dt[2] = (-(array_lmom[0] / mass));
//#pragma gpc multivector_component V1_t_dt e2^e3 V1_t_dt[3]
V1_t_dt[3] = ((I_3 - I_2) * V012 * V013 - am23) / I_1;
//#pragma gpc multivector_component V1_t_dt e2^einf V1_t_dt[4]
V1_t_dt[4] = (-(array_lmom[1] / mass));
//#pragma gpc multivector_component V1_t_dt e3^einf V1_t_dt[5]
V1_t_dt[5] = (-(array_lmom[2] / mass));
//#pragma gpc multivector_component V0_t_dt e1^e2 V0_t_dt[0]
V0_t_dt[0] = dt / 2.0 * V1_t_dt[0] + array_V0[(index) + 0 * (numMolecules)];
//#pragma gpc multivector_component V0_t_dt e1^e3 V0_t_dt[1]
V0_t_dt[1] = dt / 2.0 * V1_t_dt[1] + array_V0[(index) + 1 * (numMolecules)];
//#pragma gpc multivector_component V0_t_dt e1^einf V0_t_dt[2]
V0_t_dt[2] = dt / 2.0 * V1_t_dt[2] + array_V0[(index) + 2 * (numMolecules)];
//#pragma gpc multivector_component V0_t_dt e2^e3 V0_t_dt[3]
V0_t_dt[3] = dt / 2.0 * V1_t_dt[3] + array_V0[(index) + 3 * (numMolecules)];
//#pragma gpc multivector_component V0_t_dt e2^einf V0_t_dt[4]
V0_t_dt[4] = dt / 2.0 * V1_t_dt[4] + array_V0[(index) + 4 * (numMolecules)];
```

```
//#pragma gpc multivector_component V0_t_dt e3^einf V0_t_dt[5]
V0_t_dt[5] = dt / 2.0 * V1_t_dt[5] + array_V0[(index) + 5 * (numMolecules)];
```

Listing 3.5: An example output of codegen−compressed.

In listing 3.5, the pragmas //#pragma gpc multivector and //#pragma gpc multivector_component are the Gaalop Precompiler versions of #pragma gcd multivector and #pragma gcd multivector_component. They identify which array element belongs to which multivector blade coefficient. This information is important for further operations on multivector in following #pragma-blocks.

The exact specification of these meta-information #pragma-statements is as follows:

```
//#pragma gpc multivector_component mvName mvBlade arrayEntry
```

Listing 3.6: Meta-information #pragma-statement specification with mvName being the name of the multivector mvBlade being the name of the blade and arrayEntry being the storage location.

Note that this concept is also valid for OpenCL vectors, which is a very important fact for the GAPP OpenCL backend, described in chapter 6.

3.4 History, Architecture and Concepts with focus on Implementation

This section briefly describes the history of versions of Gaalop Precompiler (formerly Gaalop Compiler Driver) with focus on implementation.

3.4.1 The first version

The first version of Gaalop Precompiler, then called Gaalop Compiler Driver was based on C++. It was itself compiled to a native executable that called Gaalop during runtime. At the time this was the most flexible possibility of implementing the tool. The basic idea was to completely hide the complex inner implementation details and mimic the interface of the standard compiler of a platform. On Linux, this would for example mean that Gaalop GCD could be called just like the GNU Compiler Collection (gcc) , for example gcd−cxx −o test.o test.cpp. On Windows Gaalop GCD would mimic the Microsoft Visual C++ Compiler (cl.exe), such that it could be called like gcd−cxx /o test.obj test.cpp. After the transformation process, Gaalop GCD would simply call gcc to actually compile the file. In general this approach was very useful and it guaranteed a lot of flexibility.

The downside however, was a very hard maintainability. Since Gaalop GCD was calling Gaalop internally, it had to know its path and it also had to create and delete intermediate files.

3.4.2 The second version

This version tried to overcome the maintainability problems of the first version by completely implementing Gaalop Compiler Driver in Java. This had a very positive impact on code size and maintainability, but lost some of the flexibility of the first approach. The build was now completely reliant on CMake to handle generation of code files using the Gaalop GCD Java version, and secondly compilation by a compiler of your choice.

3.4.3 The third version

With the start of this work Gaalop Compiler Driver was renamed to Gaalop Precompiler (Gaalop GPC), emphasizing its advanced functionality compared to Gaalop Compiler Driver. Introducing a new set of commands (see chapter 5) drastically increased the code size of Gaalop GPC. Restructuring was required to keep the code maintainable. The current version of Gaalop GPC therefore has three major logical parts, described in section 7.1.

4 Concept Phase

The final language specification described in chapter 5 was not our first concept of the language syntax. Initially a much more integrated version was defined. At the time, a tight coupling between the embedded Geometric Algebra code and the native language code made sense.

4.1 Requirements

From a high level perspective, we have several requirements on the integration of Geometric Algebra code into native language code.

- The integration has to be as smoothly as possible. Programming with Gaalop GPC should feel natural and intuitive.
- The concept should support as many programming languages as possible.
- The embedded Geometric Algebra code should be conform to the CLUScript language, as much as possible.
- The user should have to know as little as possible about the tool itself and the underlying optimization techniques. The user should be able to be productive with a minimum learning curve.

From the implementation perspective Gaalop GPC should

- be implemented as simple as possible,
- reuse as much Gaalop functionality as admissible,
- and utilize a maximum of modern parsing technology, e.g. ANTLR.

4.2 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 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, OpenCL, CUDA and Java. An obvious solution to bridge the gap between the two paradigms stated above, is therefore to generate arrays from multivectors.

Since Gaalop is perfect in doing so, it is a reasonable choice as the foundation of this work, especially because one of the requirements stated in section 4.1 is to support as many programming languages as possible.

Even if a language does not support arrays, like the functional programming language Haskell [2], 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.

The Gaalop GAPP OpenCL backend, described in chapter 6, also supports the usage of OpenCL vectors instead of arrays, enabling the power of Single Instruction Multiple Data (SIMD) vector operations on GPUs.

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.

4.3 The first concept

Based on Gaalop as the foundation of this work, we now lay focus on defining a concept to embed Geometric Algebra code into native language code, sustaining the requirements stated above. Listing 4.1 shows an example conforming to the very first concept of Gaalop Precompiler. As you can see, Geometric Algebra code is very smoothly embedded into the surrounding C++ code. The code contains two kinds of multivectors, temporary and array-mapped multivectors. Temporary multivectors are marked with const tempmv and array-mapped multivectors are declared with const floatmv.

```
void SolverMolecule::convertFrom(const Molecule& molecule) {
        mass = molecule.mass;
        I_1 = molecule.I1;
        I_2 = molecule.I2;
        I_3 = molecule.I3;
        atomIndices = molecule.atomIndices;

        const tempmv lp = mv_from_array(molecule.lpos,e1,e2,e3);
        const tempmv rotor = mv_from_array(molecule.arot,1,-e2^e3,-e3^e1,-e1^e2);
        const tempmv lv = mv_from_array(molecule.lvel,e1,e2,e3);
        const tempmv av = mv_from_array(molecule.avel,e1,e2,e3);

        const tempmv translator = 1 - 0.5*lp^einf;
        const floatmv Din = translator*rotor;
        const floatmv Vin = einf*lv - e1^e2^e3*av;

  D0 = mv_to_array(Din,1,e1^e2,e1^e3,
                   e1^einf,e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
  V0 = mv_to_array(Vin,e1^e2,e1^e3,
                   e1^einf,e2^e3,e2^einf,e3^einf);
}
```

Listing 4.1: Example for the first concept.

Array-mapped multivectors are optimized and transformed into C float-arrays by Gaalop GPC and may be accessed from native code later on. Temporary multivectors on the other hand may only be used as source multivectors for computations resulting in array-mapped multivectors.

The const qualifier is required on purpose, because multivectors may by definition only be assigned once in Gaalop.

Multivectors may be constructed from arrays by mv_from_* commands and may be saved to arrays by using mv_to_* commands.

4.3.1 Variable Masking

In listing 4.1 line 8, molecule.lpos is used as the first parameter of mv_from_array(). According to the specification of mv_from_array() (later defined in chapter 5), molecule.lpos is required to be an array, which is in fact the case. In previous versions of Gaalop GPC it was only allowed to use parameters that fulfill the standard requirements for identifiers meaning that their name may only contain letters and numbers and have to start with a letter. With this new version of Gaalop GPC that need is eliminated.

4.3.2 Pragma Output

The thesis [27] defined //#pragma output mv mvBlades, a new #pragma-statement allowing to specify exactly which particular multivector blade coefficients of a multivector get compiled into an output array, skipping all the others.

This is useful for the case, that a user only wants to use certain multivector blade coefficients, but leave out others. Without using this #pragma-statement, unused coefficients, not equalling zero, would still be computed and would still affect runtime performance in many ways. Using //#pragma output completely removes unnecessary code.

Its syntax is the following:

```
//#pragma output mv mvBlades
```

Whereas mv is the target multivector and mvBlades are the blades that the final output should contain. Not specifying //#pragma output for a particular multivector will lead to all multivector blade coefficient being output.

Pragma output could have been easily integrated into the first concept. The method, actually implemented for the second concept is documented in section 7.6. It is mostly equivalent to how it would have been implemented in the first concept.

4.3.3 Evaluation Only Multivectors

The problem of Evaluation Only Multivectors is described in full detail in section 7.7. This functionality was being discussed at the time primarily with focus on Gaalop, but the mapping of this functionality from Gaalop to Gaalop GPC was an obvious question to ask. The most likely answer would have been that integration is established through a new type const evalmv.

4.3.4 Advantages

The main advantage of this definition is a borderless integration of Geometric Algebra code code into native code. It would feel very naturally for most programmers to write code this way.

4.3.5 Disadvantages

There are two major disadvantages of the first concept.

Firstly, and most importantly, from a technical point of view it is a problem to construct a compiler realizing this concept. Defining two different grammars for two different languages is a straightforward task, but defining one grammar for both languages is not. Most modern parsers and parser generation tools like ANTLR [29] are not designed for such a job.

Secondly, choosing this definition would include making a major design decision against code-compatibility with CLU-Script. The code would not be directly be compatible with CLUScript, meaning that one could not just test the same code in CLUCalc as well as Gaalop GPC without major code restructuring.

Also, the approach to Evaluation Only Multivectors as designed for the first concept, described in subsection 4.3.3, has the problem that is not easy to explain the user the three types of multivectors that would have been involved in order to optimally write Gaalop GPC code. This problem was elegantly avoided in the final definition of the specification, and the solution is explained in detail in section 7.7.

4.3.6 Conclusion

Primarily the first disadvantage led us to conclude, that this concept of the language specification, although it has a certain feel of completeness, the grammar is just too complex. Context-specific grammar is a major problem in compiler construction, and in practice is usually avoided by design. Designing a language containing context-specific grammar is a choice against almost all language implementation tools, like ANTLR [29], LLVM, etc., to our knowledge. This would not necessarily mean it is not possible to use any of these tools, but dominating problems would certainly arise. The first concept may still be realized in a future version of Gaalop Precompiler, but at this time our goals do not meet up with the consequences of this concept.

4.4 The second and final concept

Listing 4.2 shows an example of the second concept, that was specifically designed to address the problems of the first concept.

```
void SolverMolecule::convertFrom(const Molecule& molecule) {
        mass = molecule.mass;
        I_1 = molecule.I1;
        I_2 = molecule.I2;
        I_3 = molecule.I3;
        atomIndices = molecule.atomIndices;

#pragma gpc begin
        lp = mv_from_array(molecule.lpos,e1,e2,e3);
        rotor = mv_from_array(molecule.arot,1,-e2^e3,-e3^e1,-e1^e2);
        lv = mv_from_array(molecule.lvel,e1,e2,e3);
        av = mv_from_array(molecule.avel,e1,e2,e3);

#pragma clucalc begin
        translator = 1 - 0.5*lp^einf;
        ?Din = translator*rotor;
        ?Vin = einf*lv - e1^e2^e3*av;
#pragma clucalc end

  D0 = mv_to_array(Din,1,e1^e2,e1^e3,
                   e1^einf,e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
  V0 = mv_to_array(Vin,e1^e2,e1^e3,
                   e1^einf,e2^e3,e2^einf,e3^einf);
#pragma gpc end
}
```

Listing 4.2: Gaalop Precompiler for C++ input code.

4.4.1 The two different types of blocks

Deviating from Gaalop Compiler Driver and the first definition, there are now two different types of #pragma-blocks:

- #pragma gpc-blocks marked by #pragma gpc begin and #pragma gpc end.
- and #pragma clucalc-blocks marked by #pragma clucalc begin and #pragma clucalc end.

While the latter may solely contain pure Geometric Algebra code, the former may contain a mix of both Geometric Algebra code and native language code, more specifically the commands of chapter 7.

This solves the problem of parsing in an elegant way, since no longer context-specific grammar is required to parse the Geometric Algebra code, which is now exclusively contained within the #pragma clucalc-block.

The second problem of CLUScript code compatibility is also intrinsically solved, since the #pragma clucalc-block now contains pure CLUScript code.

5 Gaalop Precompiler Language Specification

Multivectors have a limited number of blades. For example in Conformal Geometric Algebra their size is limited to 32 blades. A multivector storage for CGA therefore has to 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 [30] 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 them 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 this problem. The obvious solution is to save only non-zero blade coefficients. This technique has been previously used in Gaalop Compiler Driver and is explained in full detail in section 3.3. To assist with this approach, several helper functions are defined in table 5.

The purpose of these helper functions, listed in table 5, is the transformation between multivectors and C/C++/OpenCL/CUDA language concepts like 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_getbladecoeff(mv,blade);	Get the coefficient of blade blade of multivector mv.
mv = mv_from_vec(vec);	Construct multivector mv from OpenCL-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 OpenCL vector vec.

Tabelle 5.1.: Gaalop GPC helper functions

6 Geometric Algebra Parallelism Programs OpenCL Backend

The paper [16] introduced a new approach to make advantage of the parallel instruction processing power available on many modern computing devices named Geometric Algebra Parallelism Programs, specifically for GA. Its main purpose is the best possible utilization of Instruction Level Parallelism (ILP) by harvesting the intrinsic parallelism of Geometric Algebra exposed by the Table Based Approach (TBA) algorithm, described in the same paper. At its core, this new technology is a new intermediate representation language, on which a whole variety of platform backends, like SSE [5], AVX [4], OpenCL, CUDA, or even a fully dedicated FPGA soft-core processor could build upon.

While the GAPP technology itself was completely realized in Gaalop, no backend was implemented yet. A secondary task of this thesis was therefore the fulfilment of a GAPP backend targeting the Open Computing Language (OpenCL) . The choice of OpenCL as the first target language was specifically made, because it is the computing language running on the highest number of platforms with a theoretical support of operations on vectors of up to 16 entries (float16).

The rest of this chapter explains in detail how the corresponding GAPP instructions are translated into OpenCL code by the backend and gives a short explanation what their purpose is in the first place.

Further focus is laid on the evaluation of GAPP code versus non-GAPP code.

6.1 Instruction Set

assignInputsVector $inputsVector = [val_0, \dots, val_n]$; Assigns a set of input scalar variables or constants to a vector. This vector is further used as a source for values for the following commands.
resetMv mv_{dest}; Zeros all blades of multivector mv_{dest}.
assignMv $mv_{dest}[sel_0, \dots, sel_n] = \{const_1, \dots, const_n\}$; Assigns the constants $const_1, \dots, const_n$ to the multivector mv_{dest} as blade coefficients specified by the selectors $sel_0, \dots, sel_n$.
setMv $mv_{dest}[dest_0, \dots, dest_n] = var_{src}[src_0, \dots, src_n]$; Copies the selected blades from multivector or vector var_{src} to multivector mv_{dest}. $dest_0$, src_0, $dest_1$, src_1, up to $dest_{31}$ and src_{31}, are blade selectors. Note that it is invalid language syntax to have more than one source multivector specified in this command. To copy elements from several multivectors it is required to use multiple setMv commands, one for each multivector. This command is restricted to one source and one destination multivector.
setVector $part_{dest} = \{var1_{src}[sel_0, \dots, sel_n], var2_{src}[sel_0, \dots, sel_n], \dots\}$; Composes the vector (part of a multivector) $part_{dest}$ from selected elements of multiple source multivectors or vectors $varn$. sel_0, sel_1, up to sel_{31}, are a blade selectors. Parts and blade selectors are explained in detail in [16].
dotVectors $mv_{dest}[sel] = < part_1, part_2 >$; Performs a scalar multiplication (dot product) on the two vectors (parts of multivectors) $part_1$ and $part_2$. Saves the result in multivector mv_{dest} at the location selected by selector sel.

Tabelle 6.1.: Geometric Algebra Parallelism Programs (GAPP) language

6.2 Preliminary steps and considerations

Code Generation requires certain knowledge that is not available during the Code Generation process itself. This knowledge has therefore to be gained by some preliminary preprocessing steps.

6.2.1 Multiple blocks

Gaalop GPC code is usually made up by multiple #pragma gpc-blocks containing one or more #pragma clucalc-blocks. Sharing of information is allowed between the blocks in the same scope for all variables.

The only exception to this rule is the specially declared helper vector named inputsVector. Its name cannot be the same across blocks, because two blocks in the same scope would then come into conflict with each other, if they declared inputsVector without any further considerations.

The obvious solution here is to rename inputsVector for each block. This can be achieved by a static counter n, counting the current block number, and by then suffixing inputsVector with n.

The specially declared temporary variables dot with suffix m, introduced in subsection 6.3.5 by the dotVectors command, also have to be taken into account. This is simply done by globally counting m, similar to n.

6.2.2 Compressed multivector sizes

More specifically, the Code Generator needs the exact size of compressed multivectors beforehand. This is equal to the number of blades with non-zero coefficients. To determine this number for each multivector, a pre-pass of all GAPP commands is executed. This pre-pass simply counts the number of set commands on each multivector, totalling to the number of non-zero coefficients.

6.2.3 Internal multivector blade map

To maintain the correct handling of Compressed Multivector Storage in correlation with GAPP an internal mapping of blades to OpenCL array/vector entries for each multivector is required. This is implemented as a cascade of map data-structures, an outer map of type HashMap<String,HashMap<Integer,String>> and an inner map of type HashMap<Integer,String>>. The inner map stores a multivector index of type integer as key and the full storage location as value of type string. The outer map simply has the multivector name as a key of type string and the full inner map as value.

An example for this is the component e1 of multivector point, which is stored in the corresponding array `float point[3]`.

```
// pragma gpc multivector point
float point[5];

// pragma gpc multivector_component point e1 point[0]
point[0] = x;
// pragma gpc multivector_component point e2 point[1]
point[1] = y;
// pragma gpc multivector_component point e3 point[2]
point[2] = z;
...
```

Here, the following mapping would occur inside the internal multivector blade map.

```
point -> 1 (e1) -> point[0]
point -> 2 (e2) -> point[1]
point -> 3 (e3) -> point[2]
```

If point was a OpenCL vector instead of an array, then the mapping would be the following:

```
point -> 1(e1) -> point.x
point -> 2(e2) -> point.y
point -> 3(e3) -> point.z
```

With this internal abstraction, we can now store multivectors in arrays, vectors or even arrays of vectors, depending on our needs. The abstraction mechanism will always return the true storage location of the multivector blade coefficient given a multivector component.

6.2.4 OpenCL vector size logic

This number may then be used to determine the required sizes of OpenCL vectors. This scheme is described in table 6.2.

6.3 Code Generation

This section shows by example, to which OpenCL code a particular GAPP command is evaluated.

compressed multivector size = 1	float
compressed multivector size = 2	float2
compressed multivector size >= 3	float4
compressed multivector size >= 5	float8
compressed multivector size >= 9	float16

Tabelle 6.2.: OpenCL vector size logic

6.3.1 resetMv

resetMv simply declares an OpenCL vector of the size determined in subsection 6.2.2 and subsection 6.2.4.

Source Code

```
resetMv D1_t;
```

Listing 6.1: Example of the GAPP command resetMv.

Generated Code

```
//#pragma gpc multivector D1_t
float8 D1_t;
```

Listing 6.2: Example of OpenCL code generated from the GAPP command resetMv.

6.3.2 setMv

setMv firstly determines the current set index. This is simply the total number of sets on the target multivector minus one. The number of sets can be easily determined as the size of the internal multivector blade map described in subsection 6.2.3. Then sets the new entries counting from that position.

Source Code

```
setMv D1_t[3] = inputsVector_0[0];
```

Listing 6.3: Example of the GAPP command setMv.

Generated Code

```
D1_t.s3 = inputsVector_0[0];
```

Listing 6.4: Example of OpenCL code generated from the GAPP command setMv.

6.3.3 assignMv

Like setMv, except that it retrieves the source data from constants instead of multivectors or vectors.

Source Code

```
assignMv D1_t[4] = 5.0;
```

Listing 6.5: Example of the GAPP command assignMv.

Generated Code

```
D1_t.s4 = 5.0f;
```

Listing 6.6: Example of OpenCL code generated from the GAPP command assignMv.

6.3.4 setVector

setVector firstly declares a multivector of a size determined by the OpenCL vector size logic. The input parameter for this vector size logic is the number of arguments supplied to the setVector command.
Secondly it assigns its arguments as initial values of the newly declared multivectors. Since the number of arguments might not be equal to the size determined by the OpenCL vector size logic, it is required to fill possible unfilled space with zero so it does not affect further calculations.

Source Code

```
setVector ve1 = {inputsVector_0[15,11,20]};
```

Listing 6.7: Example of the GAPP command setVector.

Generated Code

```
float4 ve1 = (float4)(inputsVector_0[15],inputsVector_0[11],inputsVector_0[20],0);
```

Listing 6.8: Example of OpenCL code generated from the GAPP command setVector.

6.3.5 dotVectors

dotVectors is responsible for the computation of dot products of one or more vectors declared by setVector. It performs this in two steps. Firstly the parallel multiplication of up to 16 floats, and secondly a tree-like sum reduction of the result of the first step.

Source Code

```
dotVectors D2_t[3] =<ve33,ve34,ve35>;
```

Listing 6.9: Example of the GAPP command dotVectors.

Generated Code

```
//#pragma gpc multivector_component D2_t e1^einf D2_t.s3
float16 dot29 = ve33 * ve34 * ve35;
float8 dot30 = dot29.lo + dot29.hi;
float4 dot31 = dot30.lo + dot30.hi;
float2 dot32 = dot31.lo + dot31.hi;
D2_t.s3 = dot32.lo + dot32.hi;
```

Listing 6.10: Example of OpenCL code generated from the GAPP command dotVectors.

The second line in this example performs two element-wise parallel multiplications of float16 OpenCL vectors. The result is then sum-reduced and finally stored in the vector entry D2_t.s3 in the last line.

6.3.6 assignInputsVector

assignInputsVector is a special command to declare input variables supplied to the GAPP program.

Source Code

```
assignInputsVector inputsVector = [dt, array_V0[(index) + 2 * (numMolecules)], ...];
```

Listing 6.11: Example of the GAPP command assignInputsVector.

Generated Code

```
inputsVector_0[0] = dt;
inputsVector_0[1] = array_V0[(index) + 2 * (numMolecules)];
inputsVector_0[2] = array_D0[(index) + 6 * (numMolecules)];
inputsVector_0[3] = array_D0[(index) + 5 * (numMolecules)];
inputsVector_0[4] = array_V0[(index) + 5 * (numMolecules)];
```

```
inputsVector_0[5] = array_V1[(index) + 0 * (numMolecules)];
inputsVector_0[6] = array_V1[(index) + 2 * (numMolecules)];
inputsVector_0[7] = array_D0[(index) + 3 * (numMolecules)];
inputsVector_0[8] = array_V0[(index) + 4 * (numMolecules)];
inputsVector_0[9] = array_D0[(index) + 0 * (numMolecules)];
...
```

Listing 6.12: Example of OpenCL code generated from the GAPP command assignInputsVector.

6.4 Comparison between GAPP and non-GAPP code

The tables 6.3 and 6.4 show analyses of GAPP and non-GAPP code against multiple AMD-GPUs produced by the AMD APP KernelAnalyzer tool.

An analysis of the data in the tables shows slightly better performance of GAPP over the non-GAPP implementation on older devices, and equalling performance on newer ones, for all six tested kernels of the Molecular Dynamics application of chapter 9. The benchmarks were performed over 15 AMD devices, both legacy and state-of-the-art, over low-end, mainstream, performance and high-end price segments. The order of devices in the table is by their age, starting from legacy FireStream devices, ending with state-of-the-art Radeon HD 6000 series devices. In the following, 4000 series devices are referred to as old devices, whereas 5000 and 6000 series devices are considered as new, since there seems to have been a major change in architecture and compiler technology between beginning with 5000 series devices. In the tables, the two different groups of devices, old and new, are separated by two horizontal lines.

The most important parts of both tables are the throughput columns. They depict how many million threads of the first sample kernel may approximately be processed per second per device.

If you compare both tables the picture becomes very clear. GAPP almost doubles the performance for old devices in terms of throughput (last column), reversely correlating with the of the average execution time in column 4, the number of arithmetic logic operations in column 5, and the number of estimated cycles in column 7. For new devices however, the performance equals exactly, for the same measured units for both versions of the code, with the exception of the first two models of the Radeon HD 5000 series.

Further research has to show, how exactly it is possible to further improve performance for OpenCL code generated from GAPP, especially on new devices. Experience shows, that sometimes small tweaks can change the performance in significant ways.

Conventions

In the tables below the following conventions are used.
GPR $\widehat{=}$ General Purpose Registers used.
SReg $\widehat{=}$ If General Purpose Registers are used register entries are stored in local memory and called Scratch Registers (lower is better, zero is preferred).
ALU $\widehat{=}$ Arithmetic Logic Unit operations used (lower is better.)
Fetch $\widehat{=}$ Number of fetch operations (lower is better).
ALU:Fetch $\widehat{=}$ Ratio between Arithmetic Logic Unit and fetch operations (ALU ops are cheaper than fetch ops.).
Bottleneck $\widehat{=}$ The most probable bottleneck.
Thread/Clock $\widehat{=}$ Average thread throughput per clock cycle.
Throughput $\widehat{=}$ The expected throughput in thread executions per second.

Name	GPR	SReg	Avg	ALU	Fetch	Est Cycles	ALU:Fetch	Bottleneck	Throughput
FireStream 9250	39	0	12.56	172	20	12.56	2.27	ALU Ops	796 M Threads/Sec
FireStream 9270	39	0	12.56	172	20	12.56	2.27	ALU Ops	955 M Threads/Sec
Radeon HD 4550	39	0	57.02	174	20	57.02	4.69	ALU Ops	84 M Threads/Sec
Radeon HD 4670	30	12	21.71	205	63	19.16	0.87	Global Fetch	313 M Threads/Sec
Radeon HD 4770	39	0	15.88	174	20	15.88	2.29	ALU Ops	756 M Threads/Sec
Radeon HD 4870	39	0	12.56	172	20	12.56	2.27	ALU Ops	955 M Threads/Sec
Radeon HD 4890	39	0	12.56	172	20	12.56	2.27	ALU Ops	1083 M Threads/Sec
Radeon HD 5450	35	0	40.12	140	20	40.12	7.20	ALU Ops	65 M Threads/Sec
Radeon HD 5670	35	0	17.42	140	20	17.42	3.58	ALU Ops	356 M Threads/Sec
Radeon HD 5770	35	0	9.85	140	20	9.85	1.78	ALU Ops	1380 M Threads/Sec
Radeon HD 5870	35	0	9.70	140	20	9.70	0.89	Global Write	1402 M Threads/Sec
Radeon HD 6450	35	0	49.27	140	20	49.27	7.11	ALU Ops	244 M Threads/Sec
Radeon HD 6670	35	0	19.40	140	20	19.40	3.56	Global Write	660 M Threads/Sec
Radeon HD 6870	35	0	9.70	140	20	9.70	0.89	Global Write	1485 M Threads/Sec
Radeon HD 6970	39	0	9.70	175	20	9.70	1.11	Global Write	1452 M Threads/Sec

Tabelle 6.3.: GAPP code analysis results. The two horizontal lines separate old devices from new devices, with old above and new below the two lines.

Name	GPR	SReg	Avg	ALU	Fetch	Est Cycles	ALU:Fetch	Bottleneck	Throughput
FireStream 9250	44	0	23.37	328	20	23.37	4.22	ALU Ops	428 M Threads/Sec
FireStream 9270	44	0	23.37	328	20	23.37	4.22	ALU Ops	513 M Threads/Sec
Radeon HD 4550	44	0	103.85	328	20	103.85	8.54	ALU Ops	46 M Threads/Sec
Radeon HD 4670	30	17	36.18	382	105	31.93	0.94	Global Fetch	188 M Threads/Sec
Radeon HD 4770	44	0	29.21	328	20	29.21	4.22	ALU Ops	411 M Threads/Sec
Radeon HD 4870	44	0	23.37	328	20	23.37	4.22	ALU Ops	513 M Threads/Sec
Radeon HD 4890	44	0	23.37	328	20	23.37	4.22	ALU Ops	582 M Threads/Sec
Radeon HD 5450	25	0	24.24	83	20	24.24	4.35	ALU Ops	107 M Threads/Sec
Radeon HD 5670	25	0	10.49	83	20	10.49	2.16	ALU Ops	591 M Threads/Sec
Radeon HD 5770	25	0	9.70	83	20	9.70	1.07	Global Write	1402 M Threads/Sec
Radeon HD 5870	25	0	9.70	83	20	9.70	0.53	Global Write	1402 M Threads/Sec
Radeon HD 6450	25	0	38.80	83	20	38.80	4.26	Global Write	309 M Threads/Sec
Radeon HD 6670	25	0	19.40	83	20	19.40	2.13	Global Write	660 M Threads/Sec
Radeon HD 6870	25	0	9.70	83	20	9.70	0.53	Global Write	1485 M Threads/Sec
Radeon HD 6970	20	0	9.70	93	20	9.70	0.60	Global Write	1452 M Threads/Sec

Tabelle 6.4.: Non-GAPP (TBA) code analysis results. The two horizontal lines separate old devices from new devices, with old above and new below the two lines.

7 Implementation Details

This chapter provides some insights into the inner workings of Gaalop Precompiler. It may be used as a useful documentation for further projects based on Gaalop GPC.

The specific contributions are

- a new set of Interface Features for simplified communication between Geometric Algebra code and native language code,
- reimplementing existing Interface Features with focus on higher algebras and GAPP,
- automatic utilization of #pragma output and #pragma onlyEvaluate features of Gaalop,
- internal performance optimizations reducing storage redundancy for better cache efficiency,
- a more robust and purely Java-based implementation utilizing the ANTLR parser generator,
- advanced algebra configuration features through the Gaalop Precompiler CMake settings,
- and the integration of the Visualization Code Inserter stage (by Christian Steinmetz) for future work on raytracing and automatic visualization code generation.

7.1 Software Architecture

Internally, Gaalop Precompiler code is divided into three logical sub-modules; the Input Files Composer, the Block Transformer, and the Output File Composer. This approach is much better to understand than the purely monolithic design of Gaalop Compiler Driver, and enhances maintainability a lot.

The diagram in figure 7.1 depicts the Software Architecture of Gaalop GPC from a high level perspective. Gaalop GPC performs the following actions on each run, with numbers applying to the numbers of the diagram figure 7.1.

1. The Main unit is executed on startup and controls all other units. Its first action is to call the Input Files Composer.
2. The Input Files Composer unit loads the source file from disk, parses through it, does some preprocessing for mv_* commands, extracts #pragma clucalc-blocks and gives a list of them back to the Main unit.
3. The Main unit passes the list of #pragma clucalc-blocks to the Block Transformer.
4. The Block Transformer unit sends each #pragma clucalc-block through Gaalop piece-by-piece.
5. Gaalop optimizes each block, returns the results as C code through the Compressed Multivector Storage section 3.3 back-end and sends it back to the Block Transformer.
6. The Block Transformer handles Multivector Scoping section 3.2 on blocks and gives control back to the Main unit.
7. The Main unit transfers everything to the Output File Composer.
8. The Output File Composer parses through the original source file and replaces the original #pragma clucalc-blocks with their optimized version. Also, it generates import statements

The following subsections describe the three important subunits with more detail.

7.1.1 Input Files Composer

The purpose of this part is

1. to read in the source file,
2. to extract the blocks containing the Geometric Algebra code (in between #pragma-blocks),
3. to generate multivector import handling code from mv_from_* statements and to prepend it to the existing block-code,

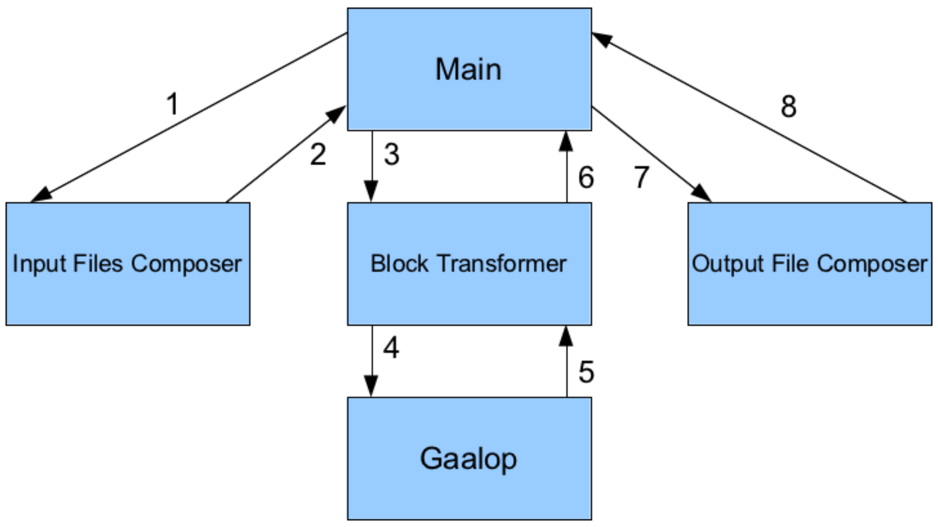


Abbildung 7.1.: Software Architecture

4. and finally, to save the blocks into an internal list.

7.1.2 Block Transformer

This part simply transforms the extracted blocks piece-by-piece using the regular Gaalop-routines internally. It also handles masking and de-masking of variables, detailed in section 4.3.1. Most importantly, it handles Multivector Scoping (section 3.2).

7.1.3 Output File Composer

The purpose of this part is

1. to read in the original source file,
2. to replace all blocks of original Geometric Algebra code code with their optimized native language version,
3. to parse mv_to_* statements and to replace them with generated multivector-to-array export handling code,
4. and finally, to save everything into an intermediate file, ready for compilation by regular compilers.

7.2 Variable Masking Implementation

As described in section 4.3.1, Gaalop GPC does no longer require input variables to be valid CLUScript identifiers. This section lays out, how this is achieved.

If Gaalop GPC were to use the qualifier molecule.lpos exactly as it is internally, this would most likely result in undefined behaviour for that it would be interpreted as the inner product . of molecule and lpos.

Gaalop GPC therefore substitutes molecule.lpos and all other incoming values by a hash string of the form hash*. After optimization by Gaalop it will replace all occurring hash* terms back to their original form.

For the purpose of replacing and its reverse, a global hash map is used internally.

7.3 Command-wise parsing

Since we are embedding a domain specific language into another language, the standard techniques of parsing do not apply. Therefore, a much simpler method has to be used to parse the contents of a #pragma gpc-block. Being inside a #pragma gpc-block Gaalop GPC reads the contents of the block, until it

- either finds the end of the block,
- or finds an embedded #pragma clucalc-block (see section 4.4.1),

- or finds a semicolon.

In case it finds a semicolon, it assumes a complete command has been found and is now contained in an internal buffer, called the command buffer.

The next step is now to search for occurrences of the mv_getbladecoeff() since this command may be embedded in any native code. If found, Gaalop GPC simply replaces the command with the referenced multivector component. For example,

```
if(mv_get_bladecoeff(indi1,1) < 0 ||
   mv_get_bladecoeff(indi2,1) < 0 ||
   mv_get_bladecoeff(indi3,1) < 0)
        return;
```

, gets replaced to:

```
if(indi1[0] < 0 ||
   indi2[0] < 0 ||
   indi3[0] < 0)
        return;
```

In case no mv_get_bladecoeff() occurrence is found, it is further searched for occurrences of mv_from_*() or mv_to_*() commands. If any of those are found, the whole command is then parsed by a special parser generated by the ANTLR Parser Generator for improved robustness. After parsing, a specialized function handles code generation for each particular command, as described in section 7.5.

If none of the above is found, the command is assumed as a native command and directly written to the output buffer.

7.4 ANother Tool for Language Recognition (ANTLR)

Gaalop Precompiler is heavily based on the ANother Tool for Language Recognition Parser Generator tool. The tool generates Java parser source code from an abstract language definition. Especially all the Gaalop GPC helper functions from chapter 5 are parsed this way. ANTLR greatly increases the robustness of Gaalop GPC, while most of the existing Gaalop parser descriptions could be reused.

7.5 Code Generation

This section shows in examples, how the new Interface Feature methods get transformed into code internally.

coeff = mv_getbladecoeff(mv,blade);

Get the coefficient of blade blade of multivector mv.

The code

```
if(mv_get_bladecoeff(indi1,1) < 0)
        return;
```

gets transformed into

```
if(indi1[0] < 0)
        return;
```

mv = mv_from_vec(vec);

Construct multivector mv from OpenCL-vector vec.

The code

```
ray_dir_mv = mv_from_vec(ray_dir_vec,e1,e2,e3);
```

gets transformed into

```
ray_dir_mv = ray_dir_vec.s0 * e1 + ray_dir_vec.s1 * e2 + ray_dir_vec.s2 * e3;
```

mv = mv_from_array(array,blades,..);

Construct multivector mv from array array.

The code

```
ray_dir = mv_from_array(ray_dir_arr,e1,e2,e3);
```

gets transformed into

```
ray_dir_mv = ray_dir_arr[0] * e1 + ray_dir_arr[1] * e2 + ray_dir_arr[2] * e3;
```

mv = mv_from_stridedarray(array,stride,blades,...);

Construct multivector mv from array array with stride stride.

The code

```
ray_dir_mv = mv_from_stridedarray(ray_dir_arr,index,stride,e1,e2,e3);
```

gets transformed into

```
ray_dir_mv = ray_dir_arr[index + 0*stride] * e1
                   + ray_dir_arr[index + 1*stride] * e2
                   + ray_dir_arr[index + 2*stride] * e3;
```

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);.

The code

```
ray_dir_arr = mv_to_array(ray_dir_mv,e1,e2,e3);
```

gets transformed into

```
ray_dir_arr[0] = ray_dir_mv[0];
ray_dir_arr[1] = ray_dir_mv[1];
ray_dir_arr[2] = ray_dir_mv[2];
```

array = mv_to_stridedarray(mv,stride,blades,...);

Write the blades blades ,... of multivector mv to array array with stride stride.
Example: array = mv_to_array(mv,nummvs,e1,e2,e3,e0,einf);.

The code

```
ray_dir_arr = mv_to_stridedarray(ray_dir_mv,index,stride,e1,e2,e3);
```

gets transformed into

```
ray_dir_arr[index + 0*stride] = ray_dir_mv[0];
ray_dir_arr[index + 1*stride] = ray_dir_mv[1];
ray_dir_arr[index + 2*stride] = ray_dir_mv[2];
```

vec = mv_to_vec(mv);

Write the multivector mv to OpenCL vector vec.

The code

```
ray_dir_vec = mv_to_vec(ray_dir_mv,e1,e2,e3);
```

gets transformed into

```
ray_dir_vec.s0 = ray_dir_mv[0];
ray_dir_vec.s1 = ray_dir_mv[1];
ray_dir_vec.s2 = ray_dir_mv[2];
```

7.6 Integration of Pragma Output into Gaalop Precompiler

The fact that Gaalop GPC knows exactly which multivector blade coefficient are actually needed through the mv_to_* statements, can be made advantage of in order to generate //#pragma output statements automatically. This is done through an internal hash map of lists. Each time a mv_to_* command is parsed by the Input Files Composer (subsection 7.1.1) it inserts an entry into the list assigned to a particular multivector.

At the end of each #pragma gpc-block, the map and the containing lists are iterated through and //#pragma output statements get generated and prepended to all #pragma clucalc-blocks inside the #pragma gpc-block.

7.7 Pragma Only Evaluate

Christian Steinmetz came up with the idea, that we might need another type of multivector in Gaalop. This addresses the specific case, that a user wishes to define a multivector that is to be used in further calculations, but not subject to explicit output as array-mapped multivector. Such a multivector may in theory just be declared as temporary multivector.

The following example listing depicts the problem.

```
?p = VecN3(px,py,pz);
?s = p - 0.5*r*r*einf;
```

Listing 7.1: Original Geometric Algebra code without //#pragma onlyEvaluate usage.

In practice, declaring it as temporary might not be suitable, because it might result in a very long optimization and compile time. The usual approach is therefore, to mark this multivector as array-mapped even though it is not used in native code.

The problem with this approach is, that all multivector component will be computed, but most likely not all of them will be used in the following computations. Some amount of them will therefore waste processing time, cache and memory.

The code above is for example transformed into the following C code by Gaalop.

```
p[0] = px; // e1
p[1] = py; // e2
p[2] = pz; // e3
p[3] = (px * px + py * py + pz * pz) / 2.0f; // einf
p[4] = 1.0f; // e0
s[0] = p[0]; // e1
s[1] = p[1]; // e2
s[2] = p[2]; // e3
s[3] = p[3] - r / 2.0f * r; // einf
s[4] = 1.0f; // e0
```

Listing 7.2: Generated C code without //#pragma onlyEvaluate usage.

If you look closely, you will notice, that even though p[4], is not used in s, it is still declared in line 5.

The solution we came up with for this problem is to mark multivectors, for which this problem applies for evaluation only. This is specifically done by an internal statement //#pragma evaluateOnly mvName. Multivectors marked by this are referred to as Evaluation Only Multivector.

The following listing depicts this solution.

```
//#pragma onlyEvaluate p s
p = VecN3(px,py,pz);
s = p - 0.5*r*r*einf;
```

Listing 7.3: Original Geometric Algebra code with //#pragma onlyEvaluate usage.

In contrary to listing 7.2, p[4] is now removed from the generated code.

```
p[0] = px; // e1
p[1] = py; // e2
p[2] = pz; // e3
p[3] = (px * px + py * py + pz * pz) / 2.0f; // einf
s[0] = p[0]; // e1
s[1] = p[1]; // e2
s[2] = p[2]; // e3
s[3] = p[3] - r / 2.0f * r; // einf
s[4] = 1.0f; // e0
```

Listing 7.4: Generated C code with //#pragma onlyEvaluate usage.

7.7.1 Integration into Gaalop Precompiler

Integration of this feature into Gaalop GPC is easy. Subsection 7.6 describes an internal hash map of lists that track the multivector blade coefficients actually needed in the code. The simple, but reasonable assumption that multivector blade coefficients are only accessed through mv_to_*() or mv_get_bladecoeff() Interface Features, lets us follow that multivectors not contained in the hash map are meant for evaluation only. In this particular case, we will simply generate a //#pragma evaluateOnly statement, and prepended it to all #pragma clucalc-blocks inside the #pragma gpc-block.

7.8 Resolving of the const float and array entry redundancy problem

Gaalop Compiler Driver allowed access of multivector blade coefficient either through the getMvComp() function with direct access to the underlying array (Compressed Multivector Storage, see section 3.3), or through predefined float constants.

Using these float constants it was for example possible to access the blade e_1, of the multivector s defined above, through the statement s_E1.

This was possible, since Gaalop GCD internally generated code similar to the following example code, with compressed array and float constants redundantly.

```
float p[4];
const float p_E1 = p[0] = px; // e1
const float p_E2 = p[1] = py; // e2
const float p_E3 = p[2] = pz; // e3
const float p_EINF = p[3] = (px * px + py * py + pz * pz) / 2.0f; // einf
float s[5]
const float s_E1 = s[0] = p[0]; // e1
const float s_E2 = s[1] = p[1]; // e2
const float s_E3 = s[2] = p[2]; // e3
const float s_EINF = s[3] = p[3] - r / 2.0f * r; // einf
const float s_E0 = s[4] = 1.0f; // e0
```

Note that this is inefficient because each multivector blade coefficient is stored twice, but it was necessary at the time because there was the need for an easy access method.

The introduction of the new Interface Feature (chapter 5) solves this problem elegantly. It is now very easy to access the blade coefficient e_1 of s through mv_get_bladecoeff(s,e1), for example.

8 The Horizon example

The horizon example is a simple application that has very often been used to demonstrate the usage of Gaalop. It is therefore a must to test this same example with Gaalop Precompiler.

Mathematical background

Consider an observer standing on a planet. Given a description of the particular planet and the viewpoint of the observer, we try to find an algebraic expression of the horizon as seen by the observer, provided there is no occlusion of any sort other than the planet itself, in the scene.

We define P as the viewpoint of the observer, S as a sphere describing the planet with center point M and radius r. Let m_x, m_y, m_z be the 3D coordinates of the planet's center and p_x, p_y, p_z be the ones of the viewpoint, then M, P and S have the following definition in 5D conformal space.

$$M = m_x e_1 + m_y e_2 + m_z e_3 + \frac{1}{2}(m_x^2 + m_y^2 + m_z^2)e_\infty + e_0$$

$$P = p_x e_1 + p_y e_2 + p_z e_3 + \frac{1}{2}(p_x^2 + p_y^2 + p_z^2)e_\infty + e_0$$

$$S = M - \frac{1}{2}r^2 e_\infty$$

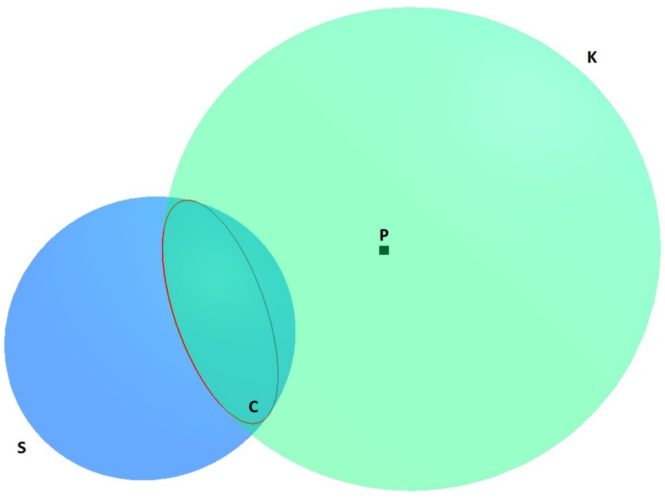


Abbildung 8.1.: Calculation of the intersection circle (horizon)

Given these definitions, we may construct another sphere K around P. The radius for this second sphere is computed by the inner product $S \cdot P$. For a mathematical background on this, see [10] for example. The circle presenting the horizon may then be calculated by the outer product of both spheres. Figure 8.1 illustrates this calculation.

$$K = P + (S \cdot P)e_\infty$$

$$C = S \wedge K$$

CLUScript implementation

Listing 8.1 shows the CLUScript equivalent of the equations above. The goal is to find an algebraic expression for the horizon on the earth as seen from a view point P.

```
P = VecN3(px,py,pz);      // view point
M = VecN3(mx,my,mz);      // center point of earth
S = M-0.5*r*r*einf;       // sphere representing earth
K = P+(P.S)*einf;         // sphere around P
?C=S^K;                   // intersection circle
```

Listing 8.1: The Horizon example in CLUScript.
Variables px, py, pz and mx, my, mz are free variables, that will be handled symbolically.

OpenCL implementation

Finally, the CLUScript code may be embedded into an OpenCL kernel, resulting in the code of listing 8.2.

```
__kernel void horizonKernel(__global float* circleCenters,__global const float* points)
{
  const int id = get_global_id(0);
  #pragma gpc begin
    P = VecN3(points[id],
              points[id]+num_points,
              points[id]+2*num_points);
    #pragma clucalc begin
      r = 1;
      S = e0-0.5*r*r*einf;
      C = S^(P+(P.S)*einf);

      ?homogeneousCenter = C*einf*C;
      ?scale = -homogeneousCenter.einf;
      ?EuclideanCenter = homogeneousCenter / scale;
    #pragma clucalc end
    circleCenters = mv_to_stridedarray(EuclideanCenter,
                                       id, num_points, e1,e2,e3);
  #pragma gpc end
}
```

Listing 8.2: The Horizon example in OpenCL.

9 Molecular Dynamics using Gaalop GPC for OpenCL

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. This chapter presents a molecular dynamics simulation based on Gaalop GPC for OpenCL.

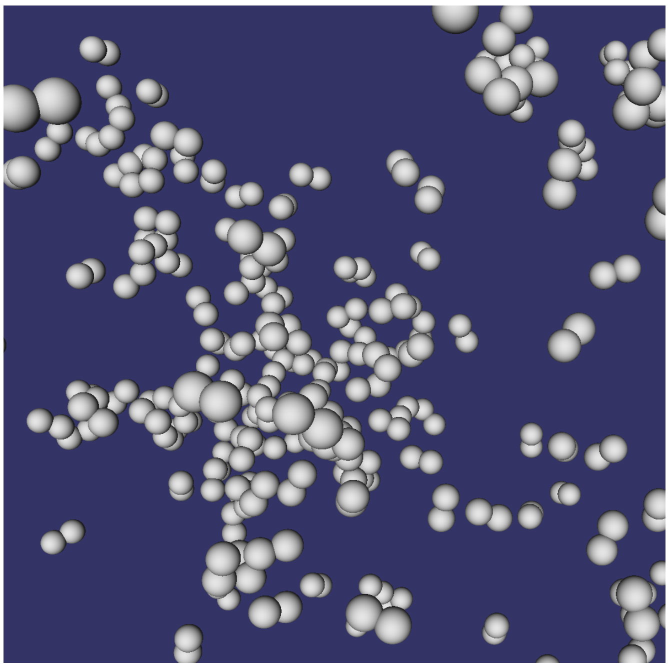


Abbildung 9.1.: Screenshot of the molecular dynamics simulation using CGA.

The Gaalop GPC implementation is faster compared to the conventional implementation, which is not self-evident for CGA-based implementations of such complexity. Further tests have shown, that Gaalop GPC yields also a higher numerical stability in terms of energy conservation. This might be due to the fact, that the advanced symbolic simplification by Gaalop GPC minimizes the amount of operations, which otherwise would have been potential sources for numerical errors.

9.1 Molecular Dynamics in a Nutshell

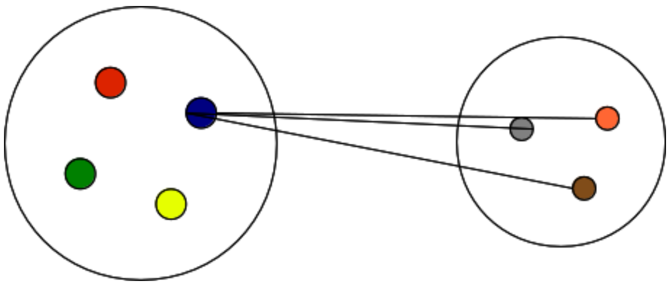


Abbildung 9.2.: The forces between all the atoms of the molecules result in a movement of the molecules.

In the following, we describe very briefly, how molecular dynamics is modelled in our simulation:

- A molecule is a compound of several atoms, which are assumed to be static inside the molecule.
- Every atom sends out attraction or repulsion forces to every other atom.

- These forces then result in a movement of the molecules according to Newton's and Euler's laws. This is simulated for 1000s of molecules in parallel.

The Lennard Jones potential

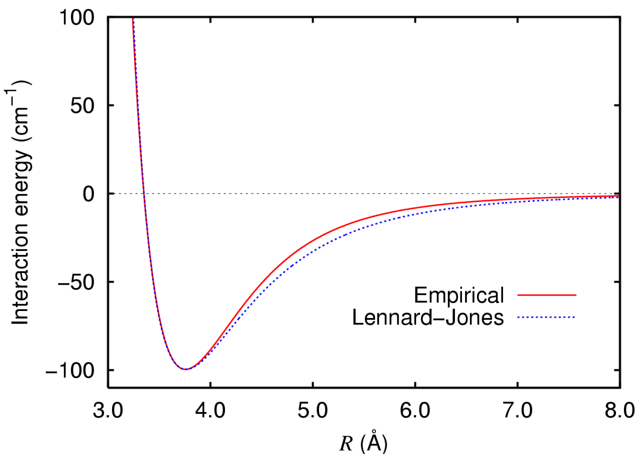


Abbildung 9.3.: The Lennard Jones potential describes the energy level between two atoms dependent on the distance between them (Image source: [6]).

A potential function describes the energy level between two atoms dependent on the distance between them. A popular approximation of the potential between real physical atoms, is the so called Lennard Jones potential, which we make strong use of in this application.

The usual method here is to derive the forces of each atom from the potentials. Mathematically and in the context of molecular dynamics, a force pointing to the direction of lowest local energy is defined as the negative gradient $-\nabla\Phi(\vec{d})$, where $\vec{d} = \vec{p_{ji}}$ is the distance between the two atom positions and Φ is the Lennard Jones potential function

$$\Phi(r) = 4\epsilon \left\{ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right\}, \tag{9.1}$$

where ϵ is a scale factor and σ is the distance at which the repulsive part overweights the attractive part of the potential.

9.2 Software Architecture

An initialization on the host (CPU) is needed prior to running the simulation itself (see section 9.3). This consists of loading a so called MOLD file, which is a snapshot of a group of real physical state of molecules including their definition.

The actual simulation is done by the OpenCL solver for the molecular dynamics computations initiated by some kernel calls. The OpenCL solver is separated into the following three parts:

1. Molecule verlet time integration step 1
   The kernel in section 9.4 updates the molecule's position and orientation. N computations are required for N molecules.

2. Computation of potential forces
   This updates each molecule's force and torque. n x (n-1) computations are required for n atoms. The kernel of section 9.5 is responsible for this step.

3. Molecule verlet time integration step 2
   The kernel in section 9.6 updates the molecule's linear and angular velocity. N computations are required for N molecules.

The downloaded data can be used for the visualization of the motion of the molecules.

The following listings show code extracted from the OpenCL versions of the solver implemented in Gaalop GPC for OpenCL.

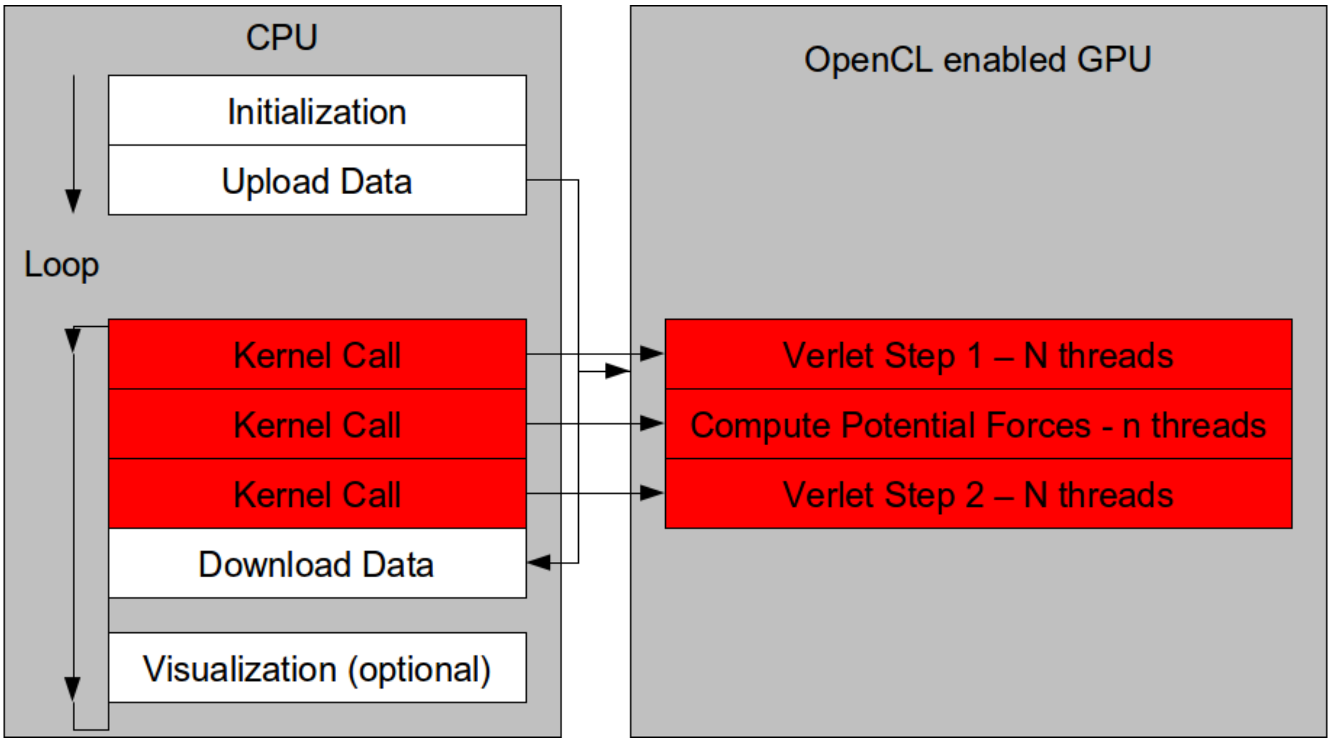


Abbildung 9.4.: Code architecture of the molecular dynamics application.

9.3 Initialization

The listing 9.1 shows host/CPU code for the initialization of the molecular dynamics simulation.

```
void convertStandardModelToSolverModel(const BaseModel& model) {
 const MoleculeVector& molecules = model.molecules;
 const AtomVector& atoms = model.atoms;
 const int numMolecules = molecules.size();
 const size_t numAtomPositions = atoms.size();
 for (int index = 0; index < numMolecules; ++index) {
   // get molecule
   const Molecule& molecule = molecules[index];

   #pragma gpc begin
     lp = mv_from_array(molecule.lpos,e1,e2,e3);
     rotor = mv_from_array(molecule.arot,1,
                          e2^e3,e3^e1,e1^e2);
     lv = mv_from_array(molecule.lvel,e1,e2,e3);
     av = mv_from_array(molecule.avel,e1,e2,e3);

     #pragma clucalc begin
       // compute start values
       translator = 1 - 0.5 * lp^einf;
       ?D_in = translator*rotor;
       ?V_in = einf*lv - e1^e2^e3*av;
     #pragma clucalc end

     host_mol_D0 = mv_to_stridedarray(D_in,index,numMolecules,
       1,e1^e2,e1^e3,
       e1^einf,e2^e3,
       e2^einf,e3^einf,
       e1^e2^e3^einf);
     host_mol_V0 = mv_to_stridedarray(V_in,index,numMolecules,
       e1^e2,e1^e3,
        e1^einf,e2^e3,
        e2^einf,e3^einf);
    #pragma gpc end

  }

  // fill device buffer, copy host buffer to device buffers
  commandQueue.enqueueWriteBuffer(dev_mol_D0.getBuffer(), CL_TRUE, 0,
        host_mol_D0.size() * sizeof(float), &host_mol_D0.front() );
  commandQueue.enqueueWriteBuffer(dev_mol_V0.getBuffer(), CL_TRUE, 0,
        host_mol_V0.size() * sizeof(float), &host_mol_V0.front() );
  std::fill(host_mol_V1.begin(), host_mol_V1.end(), 0.0f);
  commandQueue.enqueueWriteBuffer(dev_mol_V1.getBuffer(), CL_TRUE, 0,
        host_mol_V1.size() * sizeof(float), &host_mol_V1.front() );
  std::fill(host_mol_lmom.begin(), host_mol_lmom.end(), 0.0f);
  commandQueue.enqueueWriteBuffer(dev_mol_lmom.getBuffer(), CL_TRUE, 0,
        host_mol_lmom.size() * sizeof(float), &host_mol_lmom.front() );
  std::fill(host_mol_amom.begin(), host_mol_amom.end(), 0.0f);
  commandQueue.enqueueWriteBuffer(dev_mol_amom.getBuffer(), CL_TRUE, 0,
        host_mol_amom.size() * sizeof(float), &host_mol_amom.front() );

}
```
Listing 9.1: Gaalop GPC code for the conversion of the properties (position, orientation, linear and angular velocity) of the molecules into the CGA-representation as versor Din and velocity screw Vin.

The goal is to convert the data of all the molecules into their GA representation and to prepare this data for the GPU. Location and orientation of all the molecules are transformed to their displacement versor Din. Linear and angular velocity are defined through the molecule's velocity screw Vin, an expression of combined linear and angular velocity (refer to [14] and [15]).

The consecutive steps are as follows:

1. Take the position lp from the molecule position array lpos and the orientation rotor from the quaternion arot.
2. Take the linear velocity lv from the molecule velocity array lvel and the angular velocity av from the molecule array avel.
3. Define a translator from the Euclidean translation vector lp.
4. The displacement versor Din is simply the geometric product of translator and rotor.
5. The velocity screw Vin is defined as the difference of the geometric product of e_∞ with the euclidean linear velocity vector lv, and the geometric product of $e_1 \wedge e_2 \wedge e_3$ with the euclidean angular velocity vector av.
6. Move the Din and Vin data of each molecule to the strided arrays host_mol_D0 and host_mol_V0. They are versor type multivectors consisting of the scalar, six 2-blades and one 4-blade. Note that versors are even multivectors with blades of even grade.
7. Copy all the host buffers to the corresponding device buffers

Wisely chosen OpenCL kernels are called one or many times per frame, as the one in listing 9.2. This kernel computes the first half of an implicit velocity verlet integration on a molecule. A velocity verlet numerical integrator propagates the position and velocity of a mass point for a given time step.

```
__kernel void verletStep1(__global float* array_D0,
                          __global float* array_V0,
                          __global const float* array_V1,
                          const float dt,
                          const unsigned int numMolecules) {
    // compute index
    const unsigned int index = get_global_id(0);
    // clamp
    if(index >= numMolecules)
```

```
        return;

    #pragma gpc begin
      D0_t = mv_from_stridedarray(array_D0,index,numMolecules,
                          1,e1^e2,e1^e3,e1^einf,
                          e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
      V0_t = mv_from_stridedarray(array_V0,index,numMolecules,
                          e1^e2,e1^e3,e1^einf,
                          e2^e3,e2^einf,e3^einf);
      V1_t = mv_from_stridedarray(array_V1,index,numMolecules,
                          e1^e2,e1^e3,e1^einf,
                          e2^e3,e2^einf,e3^einf);
      #pragma clucalc begin
        // calculation
        const floatmv D1_t = 0.5 * D0_t * V0_t;
        const floatmv D2_t = 0.5 * D1_t * V0_t + 0.5 * D0_t * V1_t;

        const floatmv D0_t_dt = D0_t + D1_t * dt + 0.5 * D2_t * dt * dt;
        const floatmv V0_t_05dt = V0_t + 0.5 * V1_t * dt;
      #pragma clucalc end
        array_D0 = mv_to_stridedarray(D0_t_dt,index,numMolecules,
                              1,e1^e2,e1^e3,e1^einf,
                              e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
        array_V0 = mv_to_stridedarray(V0_t_dt,index,numMolecules,
                              e1^e2,e1^e3,e1^einf,
                              e2^e3,e2^einf,e3^einf);
    #pragma gpc end
}
```

Listing 9.2: Compute intensive Gaalop GPC for OpenCL code for the first step of the velocity verlet numerical integration of a molecule displacement versor and velocity screw.

The code performs the displacement propagation and computes the midpoint velocity, as described in the following equations:

Displacement propagation:	$D(t+\Delta t)$	$= D(t) + \dot{D}(t)\Delta t + \frac{1}{2}\ddot{D}(t)(\Delta t)^2$
Midpoint velocity:	$V_b\left(t+\frac{\Delta t}{2}\right)$	$= V_b(t) + \dot{V}_b(t)\frac{\Delta t}{2}$
Acceleration:	$\dot{V}_b(t+\Delta t)$	$= e_\infty \dot{v}_b(t+\Delta t) - e_{123}\dot{\omega}_b(t+\Delta t)$
Velocity propagation:	$V_b(t+\Delta t)$	$= V_b\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\dot{V}_b(t+\Delta t)\Delta t$

with

$$\dot{D} = \frac{1}{2}DV_b \left(= \frac{1}{2}V_bD = \frac{1}{2}DV_bD^{-1}D\right)$$

$$\ddot{D} = \frac{1}{2}\dot{D}V_b + \frac{1}{2}D\dot{V}_b = \frac{1}{4}DV_b^2 + \frac{1}{2}D\dot{V}_b$$

and with the Euclidean pseudoscalar

$$e_{123} = e_1 \wedge e_2 \wedge e_3$$

where the state of a molecule is described by
$D(t)$ as its displacement versor in the inertial frame,
$V_b(t)$ as its velocity screw in body frame ($V = DV_b(\tilde{D})$ in inertia frame),
$\dot{v}_b(t)$ its translational acceleration,
$\dot{\omega}_b(t)$ its rotational acceleration.

The convention for the variable names is as follows: Displacement versors and velocity screws are indicated by the letters D and V, followed by the number of differentiations (D1, for instance, means the first differentiation of the displacement versor).

A more detailed description of this approach can be found in [26].

After executing the first part of the velocity verlet algorithm, it is now required to update forces and torques acting upon all molecules.

The net force acting upon a molecule $\vec{f_m}$ is equal to the sum of forces $\sum_i \vec{f_i}$ acting upon its atoms. The net torque $\vec{t_i}$ acting upon a molecule is equal to the sum of cross products $\sum_i \vec{f_i} \times \vec{r_i} = \sum_i -(\vec{f_i} \wedge \vec{r_i})e_{123}$ of the molecule's atoms, where $\vec{r_i}$ is the position of atom i. It can be proven that the outer-product $-(u \wedge v)e_{123}$ is equal to the cross product $u \times v$.

__kernel void accumulateForcesPerAtom() computes $\vec{f_i}$ and $\vec{f_i} \wedge \vec{r_i}$ on a per atom basis. The summation is then consecutively performed in void computeMoleculeForceAndTorque(), as explained below.

```
__kernel void accumulateForcesPerAtom(
__global float* array_mol_lmom_temp,
__global float* array_mol_amom_temp,
__global const float* array_mol_D0,
__global const float* array_atom_pos,
__global const unsigned int* array_atom_pos_ind,
__global const unsigned int* array_atom_mol_ind,
const float epsilon,const float sigma,
const unsigned int numMolecules,
const unsigned int numAtoms,
const unsigned int numAtomPositions) {
   // compute index
   const unsigned int atom_index1 = get_global_id(0);
   // clamp
   if(atom_index1 >= numAtoms)
       return;

   // get atom data
   const unsigned int atom_mol_ind1
       = array_atom_mol_ind[atom_index1];

   /*
    * Precache a number of BLOCK_SIZE versors into fast local memory.
    * BLOCK_SIZE is a preprocessor definition supplied by the host
    * at compilation time.
    * Let every thread load from the strided global memory array into
    * non-strided local memory array in parallel.
    */
   __local float* versor1
       = &array_versor_cache_block[8 * get_local_id(0)];
   __local float array_versor_cache_block[8 * BLOCK_SIZE];
   {
       unsigned int shiftedIndex;
       versor1[0] = array_mol_D0[shiftedIndex
                                = atom_mol_ind1];
       versor1[1] = array_mol_D0[shiftedIndex
                                += numMolecules];
       versor1[2] = array_mol_D0[shiftedIndex
                                += numMolecules];
       versor1[3] = array_mol_D0[shiftedIndex
                                += numMolecules];
       versor1[4] = array_mol_D0[shiftedIndex
                                += numMolecules];
       versor1[5] = array_mol_D0[shiftedIndex
                                += numMolecules];
       versor1[6] = array_mol_D0[shiftedIndex
                                += numMolecules];
       versor1[7] = array_mol_D0[shiftedIndex
                                += numMolecules];
   }
   __local unsigned int array_atom_mol_ind2_block[BLOCK_SIZE];
```

```
    __local float4 array_gpos2_block[BLOCK_SIZE];

    float4 pos1,gpos1;
    {
        const unsigned int atom_pos_ind1
            = array_atom_pos_ind[atom_index1];
        pos1 = (float4)(array_atom_pos[atom_pos_ind1],
                        array_atom_pos[numAtomPositions
                        + atom_pos_ind1],
                        array_atom_pos[(numAtomPositions << 1)
                        + atom_pos_ind1],0.0f);

        #pragma gpc begin
          p1 = VecN3(pos1);
          D1 = mv_from_array(versor1,
                             1,e1^e2,e1^e3,e1^einf,
                             e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
          #pragma clucalc begin
            // calculate
            :gp1 = D1*p1*(~D1);
          #pragma clucalc end
          gpos1 = mv_to_vector(gp1,e1,e2,e3);
        #pragma gpc end
    }

    // accumulate forces
    float4 accumulated_forces = (float4)(0.0f);
    {
        for(unsigned int atom_index2_block = 0;
            atom_index2_block < numAtoms;
            atom_index2_block += BLOCK_SIZE)
        computeLennardJonesForce(&accumulated_forces,gpos1,
                array_mol_D0,array_atom_pos,
                array_atom_mol_ind,array_atom_pos_ind,
                array_atom_mol_ind2_block,array_gpos2_block,
                epsilon,sigma,
                atom_mol_ind1,atom_index2_block,
                numMolecules,numAtoms,numAtomPositions);
    }

    // transform and save molecule's force and torque
    computeMoleculeForceAndTorque(array_mol_lmom_temp,
                                  array_mol_amom_temp,
                                  versor1,pos1,
                                  accumulated_forces,
                                  atom_index1,numAtoms);
}
```

The following two utility device functions

- void computeLennardJonesForce()
- void computeMoleculeForceAndTorque()

are invoked by
__kernel void accumulateForcesPerAtom(). Their purpose is the computation of the forces acting between two atoms of different molecules.

The catch here is, that the atom positions are expressed in their molecule's frame. A transformation to inertia frame coordinates is therefore required prior to computing their forces. The inertia frame transformation $D_j \vec{p_{ji}} \tilde{D_j}$ with molecule versor D_j and atom position $\vec{p_{ji}}$ performs this job for atom 2, whereas the position of atom 1 is already pre-transformed by __kernel void accumulateForcesPerAtom() prior to calling the device function void computeLennardJonesForce().

Once both atoms are transformed, the computation of Lennard Jones potential forces is performed easily by void computeLennardJonesForceSimple(). Mathematically, a force pointing to the direction of lowest local energy is defined as the negative gradient $-\nabla\Phi(\vec{d})$, where $\vec{d} = \vec{p_{ji}}$ is the distance between the two atom positions.

```
void computeLennardJonesForce(
float4* accumulated_forces,
const float4 gpos1,
__global const float* array_mol_D0,
__global const float* array_atom_pos,
__global const unsigned int* array_atom_mol_ind,
__global const unsigned int* array_atom_pos_ind,
__local unsigned int* array_atom_mol_ind2_block,
__local float4* array_gpos2_block,
const float epsilon,const float sigma,
const unsigned int atom_mol_ind1,
const unsigned int atom_index2_block,
const unsigned int numMolecules,
const unsigned int numAtoms,
const unsigned int numAtomPositions) {

   if(atom_index2_block + get_local_id(0) < numAtoms) {
       const unsigned int atom_mol_ind2
           = array_atom_mol_ind[atom_index2_block
           + get_local_id(0)];
       array_atom_mol_ind2_block[get_local_id(0)]
           = atom_mol_ind2;
       const unsigned int atom_pos_ind2
           = array_atom_pos_ind[atom_index2_block
           + get_local_id(0)];

     #pragma gpc begin
       D2 = mv_from_array(array_mol_D0, 1,e1^e2,e1^e3,e1^einf,
                      e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
       p2 = mv_from_stridedarray(array_atom_pos,
                    atom_pos_ind2,numAtomPositions,e1,e2,e3);
       #pragma clucalc begin
         // calculate
         :gp2 = D2*p2*(~D2);
       #pragma clucalc end
       array_gpos2_block[get_local_id(0)]=mv_to_vector(gp2,e1,e2,e3);
     #pragma gpc begin
   }

   // sync for shared memory consistency
   barrier(CLK_LOCAL_MEM_FENCE);

   // compute lennard jones force using transformed positions
   for(unsigned int index_block = 0;
       index_block < BLOCK_SIZE;
       ++index_block)
        if(atom_index2_block + index_block < numAtoms
          && atom_mol_ind1
          != array_atom_mol_ind2_block[index_block])
       computeLennardJonesForceSimple(accumulated_forces,
       gpos1,array_gpos2_block[index_block],epsilon,sigma);
}

void computeLennardJonesForceSimple(float4* accumulated_forces,
                                    const float4 pos1,
                                    const float4 pos2,
                                    const float epsilon,
                                    const float sigma) {
   // compute lennard jones potential force
   const float4 distVec = pos1 - pos2;
```

```
    const float distPow2 = distVec.x * distVec.x
                         + distVec.y * distVec.y
                         + distVec.z * distVec.z;
    const float distPow6 = distPow2 * distPow2 * distPow2;
    const float distPow8 = distPow6 * distPow2;
    const float distPow14 = distPow8 * distPow6;
    const float sigmaPow6 = sigma * sigma * sigma
                          * sigma * sigma * sigma;
    const float sigmaPow12 = sigmaPow6 * sigmaPow6;
    const float factor = (24.0f * epsilon)
                       * (sigmaPow12 / distPow14
                       - sigmaPow6 / distPow8);

    accumulated_forces[0].x += distVec.x * factor;
    accumulated_forces[0].y += distVec.y * factor;
    accumulated_forces[0].z += distVec.z * factor;
}
```

Listing 9.3 is responsible for transforming a force into a molecule's local coordinate system, computing the torque by multiplying the transformed force with the position it acts upon, and for saving both in memory.

The explicit steps are as follows:

1. Transform the force into molecule body frame using the operation $\tilde{D} f D$.
   f is the force and D is the molecule's versor.
2. The torque is computed by a simple outer product $\wedge$ of the force application point and the force itself. The operands and the result are expressed in terms of the body frame.

```
void computeMoleculeForceAndTorque(__global float* atom_lmom_temp,
                                   __global float* atom_amom_temp,
                                   __local const float* versor1,
                                   const float4 localPos,
                                   const float4 globalForce,
                                   const unsigned int atom_index,
                                   const unsigned int numAtoms)
{
  #pragma gpc begin
    moleculeVersor = mv_from_array(versor1,
                1,e1^e2,e1^e3,e1^einf,
                e2^e3,e2^einf,e3^einf,e1^e2^e3^einf);
    #pragma clucalc begin
            posLocal = VecN3(localPos);
            forceGlobal = VecN3(globalForce);

        // calculate
        :local_force = ~moleculeVersor
                     * forceGlobal
                     * moleculeVersor;
        :local_torque = posLocal ^ local_force;
    #pragma clucalc end
        atom_lmom_temp = mv_to_stridedarray(local_force,atom_index,
            numAtoms,e1,e2,e3);
        atom_amom_temp = mv_to_stridedarray(local_torque,atom_index,
            numAtoms,e1,e2,e3);
  #pragma gpc end
}
```

Listing 9.3: Gaalop GPC for OpenCL code for the computation of force and torque

9.6 Velocity Verlet Step 2

This kernel performs the second step of the velocity verlet implicit integration. Mathematically, it integrates acceleration and velocity propagation as defined below:

Acceleration:	$\dot{V}_b(t+\Delta t)$	$= e_\infty \dot{v}_b(t+\Delta t) - e_{123}\dot{\omega}_b(t+\Delta t)$
Velocity propagation:	$V_b(t+\Delta t)$	$= V_b\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\dot{V}_b(t+\Delta t)\Delta t$

```
__kernel void verletStep2(__global float* array_V0,
                          __global float* array_V1,
                          __global float* array_lmom,
                          __global float* array_amom,
                          __global const float* array_masses,
                          __global const float* array_inertia,
                          const float dt,
                          const unsigned int numMolecules) {
   // compute index
   const unsigned int index = get_global_id(0);
   // clamp
   if(index >= numMolecules)
       return;

   #pragma gpc begin
     lmom = mv_from_stridedarray(array_lmom,index,
          numMolecules,e1,e2,e3);
     amom = mv_from_stridedvec(array_amom,index,
          numMolecules,e1,e2,e3);
     V0 = mv_from_stridedarray(array_V0,index,numMolecules,
                           e1^e2,e1^e3,e1^einf,
                           e2^e3,e2^einf,e3^einf);
     const float mass = array_masses[index];
     const float I_1 = array_inertia[shiftedIndex
                                     = index];
     const float I_2 = array_inertia[shiftedIndex
                                    += numMolecules];
     const float I_3 = array_inertia[shiftedIndex
                                    += numMolecules];
     #pragma clucalc begin
       avel1_t = V023;
       avel2_t = V013;
       avel3_t = V012;
       v1_t_dt = lmom / mass;

       // temporary values
       w1_t_dt_1 = (am23 - (I_3 - I_2) * avel2_t * avel3_t) / I_1;
       w1_t_dt_2 = (am13 - (I_1 - I_3) * avel3_t * avel1_t) / I_2;
       w1_t_dt_3 = (am12 - (I_2 - I_1) * avel1_t * avel2_t) / I_3;
       w1_t_dt = e1 * w1_t_dt_1 + e2 * w1_t_dt_2 + e3 * w1_t_dt_3;

       // calculate
       :V1_t_dt = einf * v1_t_dt - e1^e2^e3 * w1_t_dt;
       :V0_t_dt = V0_t_05dt + 0.5 * V1_t_dt * dt;
     #pragma clucalc end
     array_V0 = mv_to_stridedarray(V0_t_dt,index,numMolecules,
                                   e1^e2,e1^e3,e1^einf,
                                   e2^e3,e2^einf,e3^einf);
     array_V1 = mv_to_stridedarray(V1_t_dt,index,numMolecules,
                                   e1^e2,e1^e3,e1^einf,
                                   e2^e3,e2^einf,e3^einf);
   #pragma gpc end
}
```

9.7 Evaluation of Gaalop GPC on the Molecular Dynamics simulation

Evaluating the performance of Gaalop Precompiler in terms of code quality in the case of the Molecular Dynamics simulation showed that Gaalop GPC meets up with the initial requirements defined in section 4.1. The code size has shrunk by about 40 percent, while readability has improved. Runtime performance has slightly improved due to the optimization in section 7.8.

10 Realtime OpenCL Raytracer

The Realtime OpenCL Raytracer is a project adopted from the original sources by the paper [21]. Our first goal was to adapt the original code in such a way that it can be used with Gaalop Precompiler and to visualize all triangles using Geometric Algebra-based ray-triangle tests. This goal was achieved in a way similar to [7]. The application is primarily a proof-of-concept for the applicability of higher dimensional algebras in Gaalop GPC, but is also a possible basis for the integration of visualization into Gaalop GPC as a language feature, outlined in section 11.

10.1 Visualizing the G6,3 Algebra

Our second interest is in visualizing the nine-dimensional G6,3 Algebra, described in section 1.3, primarily as a proof-of-concept for the successful integration of higher dimensional algebras into Gaalop GPC. This algebra, like any other Geometric Algebra, defines its geometric objects on the so called Inner Product Null Space (IPNS) .

The IPNS of a multivector mv is defined as all points p in G6,3 Algebra space, which satisfy the equation $IPS(p) = 0$, where IPS is the so called Inner Product Space $IPS(p) = mv.p$. Since mv might have a different grade than p, it follows that IPS is a field of multivectors spanned over 3D-space. The only exception is the special case that mv is a grade-one vector, for which IPS will be a scalar field since p is itself a grade-one vector.

To satisfy the IPNS equation $mv.p = 0$ for a point p, the Inner Product Space IPS must have a value of 0 for all blades IPS_i at the given point. One could reformulate this to $f(p) = \sum_i IPS_i(p)^2$, which transforms the problem to a minimization (optimization) task.

This may be further transformed into a one-dimensional problem by substituting p with $p(t) = b + t * d$ with b and d being vectors. The full equation now looks as follows:

$f(t) = \sum_i IPS_i(p(t))^2$

If we now interpret b as the origin and d as the direction, we have a raytracing application on an implicit surface. The task is now to find the smallest t for which $f(t) < \epsilon$ is satisfied, with ϵ being small.

The line search itself is implemented using a fairly simple method. Starting at the ray origin, we iterate along the ray by incrementing t as long as $f(t)$ is decreasing.

Putting the concept into code, it looks like figure 10.1 for an ellipsoid.

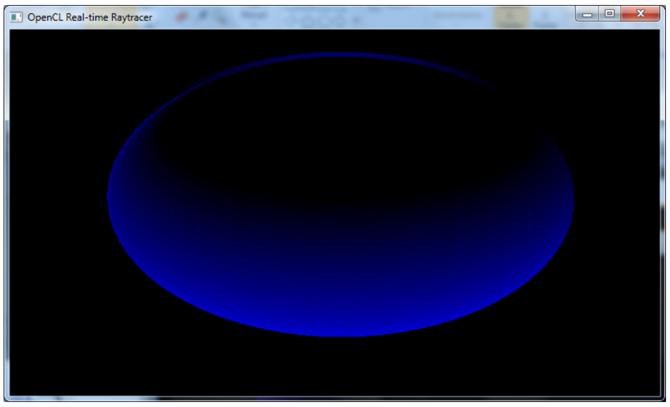


Abbildung 10.1.: Screenshot of Realtime OpenCL Raytracer, visualizing an ellipsoid in nine-dimensional G6,3 Algebra.

Note that this method is not exclusive to the G6,3 Algebra. It should work out-of-the-box for any other Geometric Algebra for which the IPNS is defined.

10.2 Source Code

The following subsections show and explain the most important parts of the source code of the raytracer.

10.2.1 Inner Product Space

Each point inside the three dimensional Inner Product Space scalar field is computed using the following OpenCL code.

```
float computeIPS(const float4 p_vec)
{
#pragma gpc begin
const float px = p_vec.x;
const float py = p_vec.y;
const float pz = p_vec.z;
#pragma clucalc begin
mv = createEllipsoid(3,1,1,0,0,0);
p = createPoint(px,py,pz);
?I = mv.p;
#pragma clucalc end
#pragma gpc end

float sum = 0;
for(unsigned int i = 0; i < sizeof(I) / sizeof(float); ++i) {
        const float coeff = I[i];
        sum += coeff*coeff;
}

return sum;
}
```

createEllipsoid() creates the ellipsoid shown in figure 10.1, with radii 1 in y and z dimensions, and radius 3 in x-dimension.

The point p is constructed at the coordinates px, py, and pz, taken from the OpenCL vector p_vec.

The inner product . of multivector mv and point p is assigned to the multivector I, which is optimized to an array.

Since the result of the inner product is not necessarily a scalar, it is then iterated through, each element is squared and added to sum.

10.2.2 One-dimensional search

The one-dimensional search along each ray is performed using the code below:

```
void intersectRayMultivector(const RTRay* ray,
                             __constant RTPrimitive* primitive,
                             RTIntersection* result)
{
const float4 ray_dir = ray->direction;

float stepsize = 0.01f;
float t = 0.0f;
float4 x_last;
float4 x = ray->origin;
float y_last;
float y = computeIPS(x);
do {
        x_last = x;
        y_last = y;
        const float4 s = stepsize * ray_dir;
        x = x_last + s;
        y = computeIPS(x);
        t += stepsize;
} while(y - y_last < 0.0f);

        if(y > 0.001f)
                return;

        result->t = t;
        result->p = 0;
}
```

For each pixel in camera-space, a line search is now performed along the viewing direction ray−>direction, starting at the pixel's position in 3D-space ray−>origin.

For this, we will save the last position x_last, the current position x, the value y_last at the last position, and the value at the current position y.

The stepsize is predefined now, but should be automatically adapted by future algorithms for faster convergence. The step vector s is simply the 3D-distance between two iteration positions, such that the sum of old position x_last and s equals the current position x.

t is the current one-dimensional coordinate along the ray, such that x = ray−>origin + t∗s holds.

The function computeIPS(), defined in subsection 10.2.1, is used to compute the value at the current position in each step.

The algorithm now computes the current position x and the value y at that position in each step, and computes the difference of y to the last value y_last. The do while() loop condition checks in each step, that this difference is negative, meaning that the values are getting smaller each step.

If the difference is positive, it is assumed that the first minimum on the line has just been passed, and the loop is exited. Even though a minimum was found, the object might not have been found, and it is therefore required to check for y being zero with a tolerance.

If the above is true, we save the found line distance t in the structure result of type RTIntersection.

10.2.3 Computing normals

Computing the normals comes down to computing the gradient using finite differences on the Inner Product Space and normalizing it:

```
float4 computeNormalFromIPS(float4 p)
{
        const float stepsize = 0.001f;

        const float ip = computeIPS(p);
        const float ipx = computeIPS(p + (float4)(stepsize,0.0f,0.0f,0.0f));
        const float ipy = computeIPS(p + (float4)(0.0f,stepsize,0.0f,0.0f));
        const float ipz = computeIPS(p + (float4)(0.0f,0.0f,stepsize,0.0f));

        const float4 normal = (float4)(ipx - ip,ipy - ip,ipz - ip,0.0f);
        return normal / length(normal);
}
```

10.2.4 Shading

Finally, the shading is done using the following code:

```
float4 position = ray.origin + ray.direction * result.t;
float4 normal = computeNormalFromIPNS(position);

float4 lightDir = ray.direction;
float NdL = max(0.0f, dot(normal, lightDir));

color = (float4)(0.0f,0.0f,1.0f,0.0f) * NdL;
```

This code snippet shades the object of colour blue with exactly one light source, that is always pointing along the camera direction.

The basic math here is to compute the dot product between normal and the light direction lightDir. The dot product gives a measure of how much a point on a surface in space is lit, given its normal and the light direction. This measure is exactly one at the brightest level, zero if not lit, and negative if the light is behind the surface. By clamping at zero and storing in NdL, we assure that if the light is behind the surface, we just render the point as not lit.

The final color is simply computed by multiplying a diffuse color, in this case blue (float4)(0.0f,0.0f,1.0f,0.0f) with NdL.

11 Future Work

This chapter is meant to discuss some ideas on how to further improve Gaalop Precompiler in the future.

Gaalop GPC for Java and other languages

Apart from C++, OpenCL and CUDA, other languages Java, Microsoft DirectCompute and shading languages (CG, HLSL) are interesting target languages for Gaalop GPC and promising topics for further research. Development of Gaalop GPC for Java has already been started, but needs to be continued. The basic functionality is implemented, but some organizational matters still need to be decided.

Headers

More thought should be put into the way Gaalop GPC handles header files. To this point, it does not perform any optimizations in them, but would doing so make sense at all? Some general questions need to be answered, some concepts need to be worked out.

OpenCL-Extension for Geometric Algebra

The concept of the direct integration of Geometric Algebra code into OpenCL-code carries significant advantages for OpenCL developers, as they would not have to install additional software to integrate GA-support into their toolchains. It would lower the initial barriers, while moderating the learning curve, because the connection of both languages could be established even smoother and the number of dependencies would be reduced.

Algebraic multivector rendering for Conformal Geometric Algebra as a language feature

Gaalop GPC will be expanded in future work to support algebraic rendering of multivectors similar to CLUCalc (see figure 11.1 for example). 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 [19] or other rendering APIs. Work on this has been started, but is far from complete yet.

Direct multivector raytracing for higher algebras as a language feature

The raytracer application in section 10 showed ways to render the objects of the nine-dimensional G6,3 Algebra. Future applications could profit a direct visualization method embedded in Gaalop GPC, that would allow rendering G6,3 Algebra multivectors by prefixing them with a colon, similar to CLUCalc.

11.1 Applications

This section is intended to show promising applications, which would profit from an implementation based on Gaalop GPC.

11.1.1 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. Work on this has been started by Christian Steinmetz with promising results and will be published in his Master thesis.

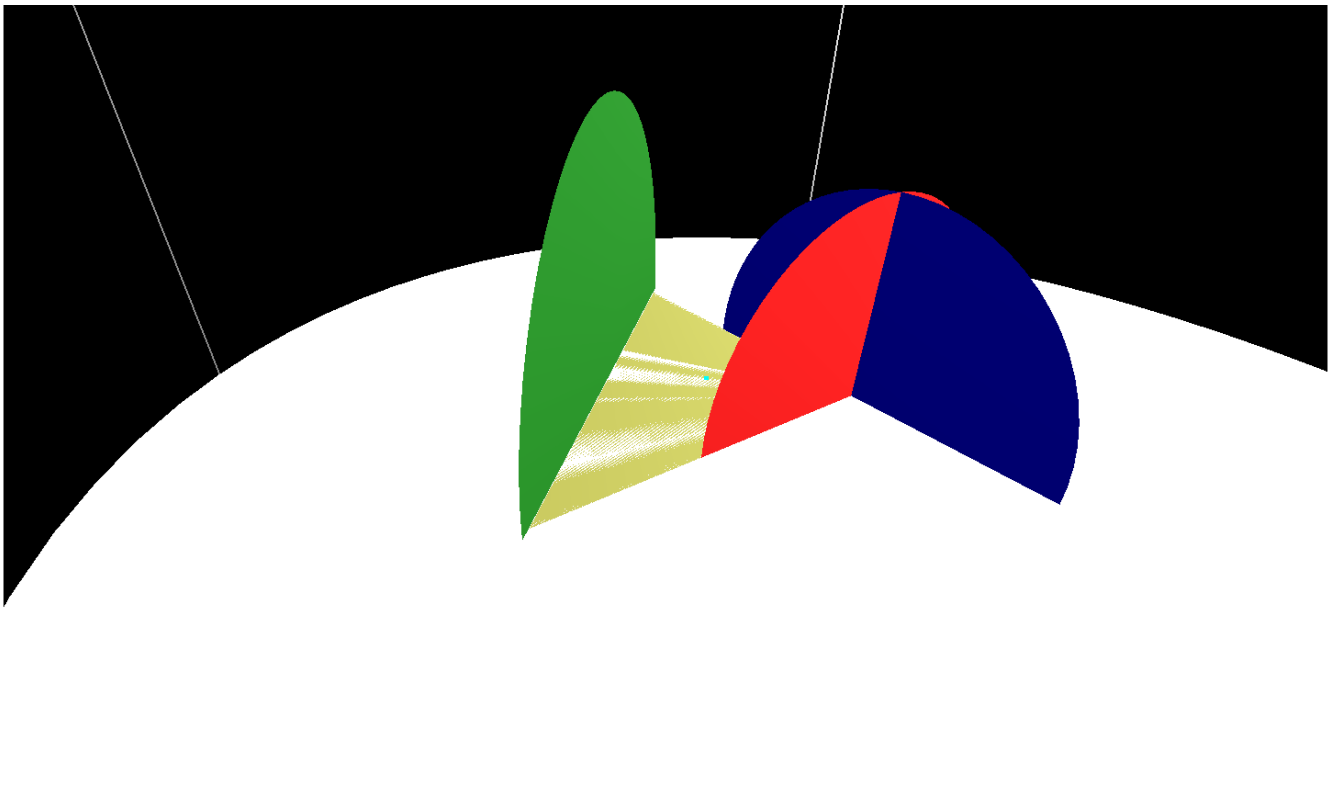


Abbildung 11.1.: An example of CLUCalc generated graphics

11.1.2 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 [14].

11.1.3 Computer Graphics and Games

Eric Lengyel presented his research on Grassmann Algebra at the Game Developers Conference 2012 in San Francisco [1]. 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.

11.1.4 Molecular dynamics with focus on polymer-chains

Chapter 9 showed the applicability of Geometric Algebra in molecular dynamics, 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 reducing the number of force calculations, since force computations between strongly-bonded atoms within the groups are no longer required.

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

12 Conclusion

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

Gaalop GPC symbolically optimizes the embedded Geometric Algebra 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 in many build automation tools like GNU Make [13]. It is found, that in reality, using parallel builds, longer compile time is not a problem.

We would like to conclude, that Gaalop Precompiler goes far beyond the frontiers set by Gaalop Compiler Driver and 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 language.

Especially the combination of GA with OpenCL or CUDA enables new methods of research, while GAPP language helps to utilize the power of advanced SIMD GPU-architectures.

Higher algebras open the door to a completely new field of mathematics. 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 Geometric Algebra.

A A Guide to Gaalop GPC

This section shows the particular steps that need to be performed, to create, setup and build an application using Gaalop GPC for C++ and OpenCL.

A.1 How To Use Gaalop GPC for C++

This manual shows the particular steps that need to be performed, to create, setup and build an application using Gaalop GPC for C++.

1. Create the file horizon.cpg with the contents below. For a mathematical description of the code, you may refer to chapter 8.

```
#include <iostream>
#include <gpc.h>

int main() {
  #pragma gpc begin
    #pragma clucalc begin
      P = VecN3(1,1,0);
      r = 1;
      S = e0-0.5*r*r*einf;
      C = S^(P+(P.S)*einf);
      ?homogeneousCenter = C*einf*C;
      ?scale = -homogeneousCenter.einf;
      ?EuclideanCenter = homogeneousCenter / scale;
    #pragma clucalc end
    std::cout << mv_get_bladecoeff(homogeneousCenter,e0)<<std::endl;
    std::cout << mv_get_bladecoeff(EuclideanCenter,e0)<<std::endl;
    std::cout << mv_get_bladecoeff(EuclideanCenter,e1)
    << "," << mv_get_bladecoeff(EuclideanCenter,e2)
    << "," << mv_get_bladecoeff(EuclideanCenter,e3);
  #pragma gpc end
  return 0;
}
```

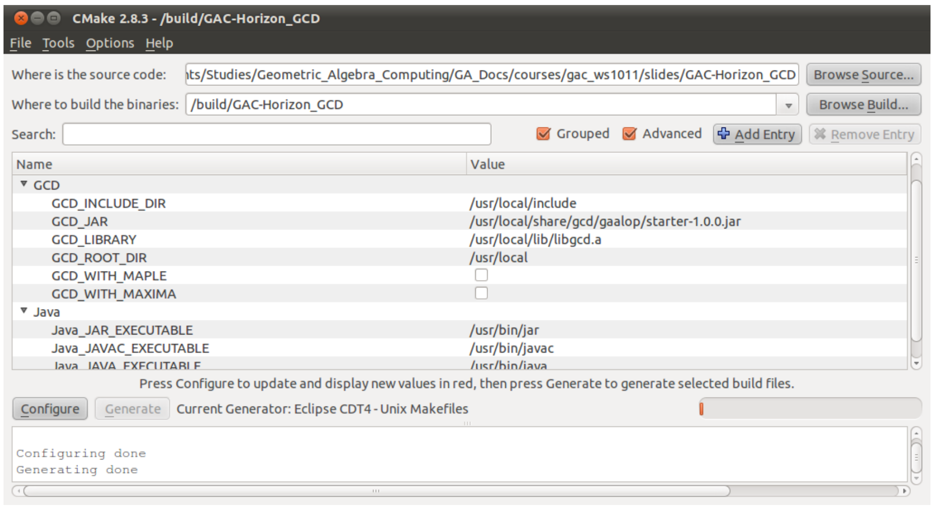
Listing A.1: horizon.cpg source file

2. Create the CMakeLists.txt build script:

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
PROJECT(horizon)
SET(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})
FIND_PACKAGE(GPC)
GPC_CXX_ADD_EXECUTABLE(horizon "horizon.cpg")
```

Listing A.2: CMakeLists.txt build script

3. Start CMake.
4. Fill in the source directory. (first input field). Fill in the destination directory. (second input field).
5. In the window opening, choose GNU make generator.
6. Click Configure.
7. Fill in the root path to Gaalop GPC in the GPC_ROOT_DIR field. On Linux this should be automatically discovered.
8. Click Configure again. All other Gaalop GPC-related paths should now be discovered.
9. Click Generate.
   Figure A.1 shows how CMake may look like after Configuration and Generation.



Where is the source code: ts/Studies/Geometric_Algebra_Computing/GA_Docs/courses/gac_ws1011/slides/GAC-Horizon_GCD
Where to build the binaries: /build/GAC-Horizon_GCD
Search:

Name	Value
GCD	
GCD_INCLUDE_DIR	/usr/local/include
GCD_JAR	/usr/local/share/gcd/gaalop/starter-1.0.0.jar
GCD_LIBRARY	/usr/local/lib/libgcd.a
GCD_ROOT_DIR	/usr/local
GCD_WITH_MAPLE	☐
GCD_WITH_MAXIMA	☐
Java	
Java_JAR_EXECUTABLE	/usr/bin/jar
Java_JAVAC_EXECUTABLE	/usr/bin/javac
Java_JAVA_EXECUTABLE	/usr/bin/java

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Eclipse CDT4 - Unix Makefiles

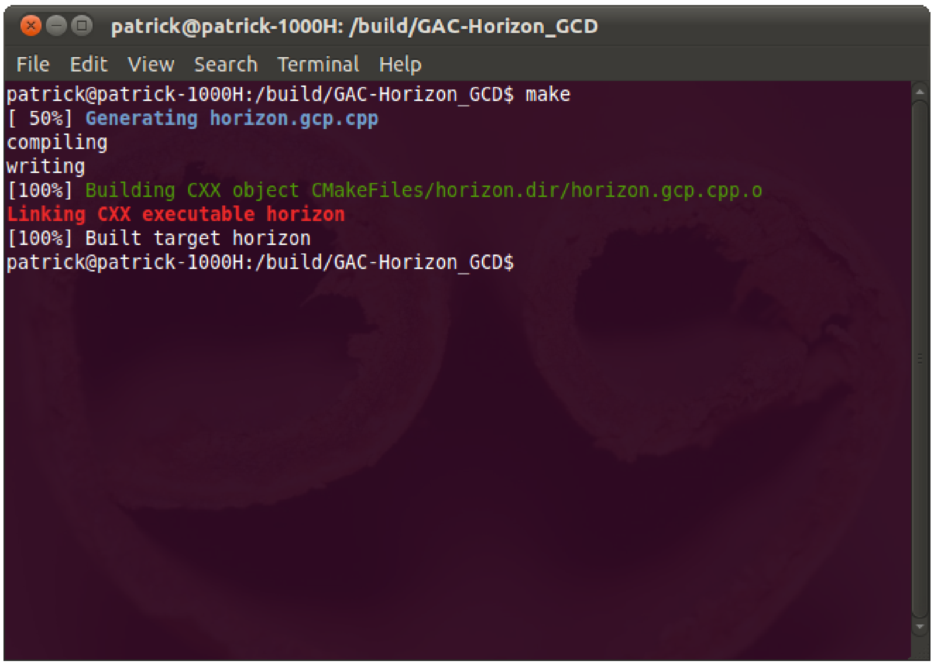
```
Configuring done
Generating done
```

Abbildung A.1.: CMake configuration for Gaalop GPC for C++

10. Open the CMake generated destination directory in your terminal or console.
    Enter make (Unix) or MinGW (Windows) and confirm with the Enter key.
    Wait until the build processes finishes.

    Hint: Using CMake and Gaalop GPC, builds are also easily possible with Visual Studio, Borland Builder or any other build tool of your choice.

11. Start the compiled application (figure A.2).
    Unix: ./horizon
    Windows: horizon.exe


patrick@patrick-1000H: /build/GAC-Horizon_GCD
File Edit View Search Terminal Help

```
patrick@patrick-1000H:/build/GAC-Horizon_GCD$ make
[ 50%] Generating horizon.gcp.cpp
compiling
writing
[100%] Building CXX object CMakeFiles/horizon.dir/horizon.gcp.cpp.o
Linking CXX executable horizon
[100%] Built target horizon
patrick@patrick-1000H:/build/GAC-Horizon_GCD$
```

Abbildung A.2.: Screenshot of Gaalop GPC for C++ build process.

A.2 How To Use Gaalop GPC for OpenCL

The Horizon example is not perfectly suited as an OpenCL application, but here is how the horizon could be computed for thousands of observers at once.

1. Create the file horizon.clg with the following contents:

```
__kernel void horizonKernel(__global float* circleCenters,
                            __global const float* points,
                            const unsigned int num_points) {
  const int id = get_global_id(0);
  #pragma gpc begin
    P = VecN3(points[id],
              points[id]+num_points,
              points[id]+2*num_points);
    #pragma clucalc begin
      r = 1;
      S = e0-0.5*r*r*einf;
      C = S^(P+(P.S)*einf);

      ?homogeneousCenter = C*einf*C;
      ?scale = -homogeneousCenter.einf;
      ?EuclideanCenter = homogeneousCenter / scale;
    #pragma clucalc end
    circleCenters = mv_to_stridedarray(EuclideanCenter,
                                        id, num_points, e1,e2,e3);
  #pragma gpc end
}
```

Listing A.3: horizon.clg OpenCL kernel source file

2. The fastest way to carry out the following steps is to copy one of the examples that come with your OpenCL distribution and modify it according to your needs. Only the most important parts of the code are pointed out in the following listings. The code resides in the horizon.cpp source-file.

3.
```
// list platforms
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
std::cout << "listings␣platforms\n";
for (std::vector<cl::Platform>::const_iterator it =
        platforms.begin(); it != platforms.end(); ++it)
std::cout << it->getInfo<CL_PLATFORM_NAME>() << std::endl;

// create context
cl_context_properties properties[] = { CL_CONTEXT_PLATFORM,
        (cl_context_properties)(platforms[0])(), 0 };
cl::Context context(CL_DEVICE_TYPE_ALL, properties);
std::vector<cl::Device> devices = context.getInfo<
        CL_CONTEXT_DEVICES>();
cl::Device& device = devices.front();

// create command queue
cl::CommandQueue commandQueue(context, device);
```

Listing A.4: List platforms and create context and command queue.

4.
```
// settings
const size_t numPoints = 10000;
cl_float circleCenters[3*numPoints];
cl_float points[3*numPoints];
```

Listing A.5: Create a host buffer.

5.
```
// Allocate the OpenCL buffer memory objects for source
// and result on the device GMEM
clDeviceVector<cl_float> dev_circle_centers(context,
                commandQueue,numPoints * 3,CL_MEM_READ_ONLY);
clDeviceVector<cl_float> dev_points(context,
                commandQueue,numPoints * 3,CL_MEM_READ_ONLY);
```

Listing A.6: Create a device buffer with the same size.

6.
   ```
   // Asynchronous write of data to GPGPU device
   dev_points = points;
   ```

Listing A.7: Copy the host buffer to the device buffer.

7.
   ```
   // read the OpenCL program from source file
   std::string sourceString;
   readFile(sourceString, "horizon.gcl.cl");
   cl::Program::Sources clsource(1, std::make_pair(
           sourceString.c_str(), sourceString.length()));
   cl::Program program(context, clsource);

   // build
   program.build(devices);
   std::cout
           << program.getBuildInfo<CL_PROGRAM_BUILD_LOG> (device)
           << std::endl;

   // create kernel and functor
   cl::Kernel horizonKernel(program, "horizonKernel");
   cl::KernelFunctor horizonFunctor =
           horizonKernel.bind(commandQueue,
                   cl::NDRange(numPoints),cl::NullRange);
   ```

Listing A.8: Load the OpenCL kernel.

8.
   ```
   // Launch kernel
   horizonFunctor(dev_circle_centers.getBuffer(),
           dev_points.getBuffer());

   // Synchronous/blocking read of results, and check accumulated errors
   dev_circle_centers.copyTo(circleCenters);
   ```

Listing A.9: Set the device buffers as kernel arguments and start the kernel by using the functor.

9.
   ```
   // Synchronous/blocking read of results,
   // and check accumulated errors
   dev_circle_centers.copyTo(circleCenters);
   ```

Listing A.10: Read back the results from device to host.

10.
    ```
    // print first circle center
    std::cout << circleCenters[0] << "," << circleCenters[1]
            << "," << circleCenters[2] << std::endl;
    ```

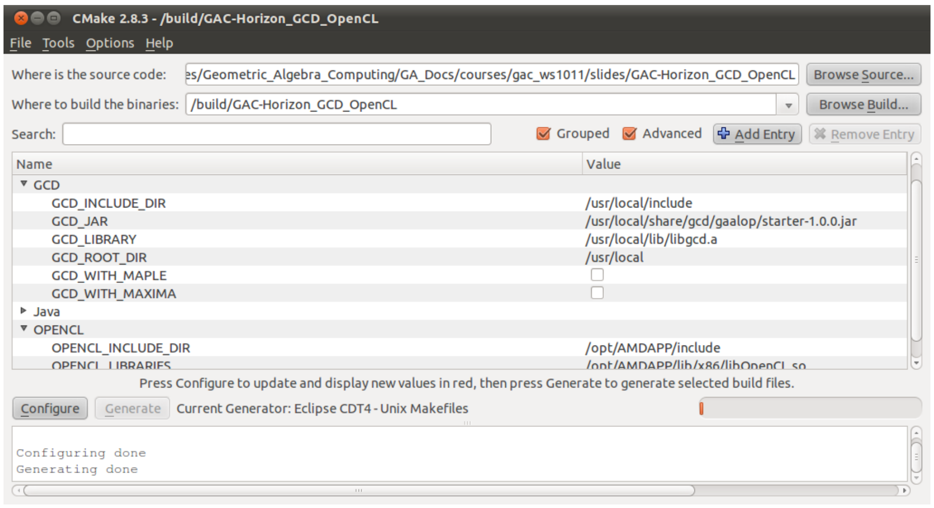
Listing A.11: Print the center of the first circle.

11.
    ```
    CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
    PROJECT(horizon)
    SET(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})
    FIND_PACKAGE(OpenCL REQUIRED)
    FIND_PACKAGE(GPC REQUIRED)
    GPC_OPENCL_ADD_EXECUTABLE(horizon "horizon.cpp" "horizon.clg")
    ```

Listing A.12: Create the file CMakeLists.txt with the following contents.

12. Start CMake.
13. Fill in the source directory (first input field). Fill in the destination directory (second input field).
14. In the window opening, choose GNU make as generator.
15. Click Configure.
16. Fill in the path to Gaalop GPC in the GPC_ROOT_DIR field.

17. Set OPENCL_INCLUDE_DIR to the include directory of your OpenCL distribution and OPENCL_LIBRARIES to the corresponding library. (Hint: ATI Stream SDK OpenCL library is located in /lib/x86/*OpenCL.lib.)
18. Click Configure again.
19. Click Generate.
    Figure A.3 shows how CMake may look like after Configuration and Generation.



CMake 2.8.3 - /build/GAC-Horizon_GCD_OpenCL

Where is the source code: es/Geometric_Algebra_Computing/GA_Docs/courses/gac_ws1011/slides/GAC-Horizon_GCD_OpenCL

Where to build the binaries: /build/GAC-Horizon_GCD_OpenCL

Search:

✓ Grouped ✓ Advanced

Name	Value
GCD	
GCD_INCLUDE_DIR	/usr/local/include
GCD_JAR	/usr/local/share/gcd/gaalop/starter-1.0.0.jar
GCD_LIBRARY	/usr/local/lib/libgcd.a
GCD_ROOT_DIR	/usr/local
GCD_WITH_MAPLE	☐
GCD_WITH_MAXIMA	☐
Java	
OPENCL	
OPENCL_INCLUDE_DIR	/opt/AMDAPP/include
OPENCL_LIBRARIES	/opt/AMDAPP/lib/x86/libOpenCL.so

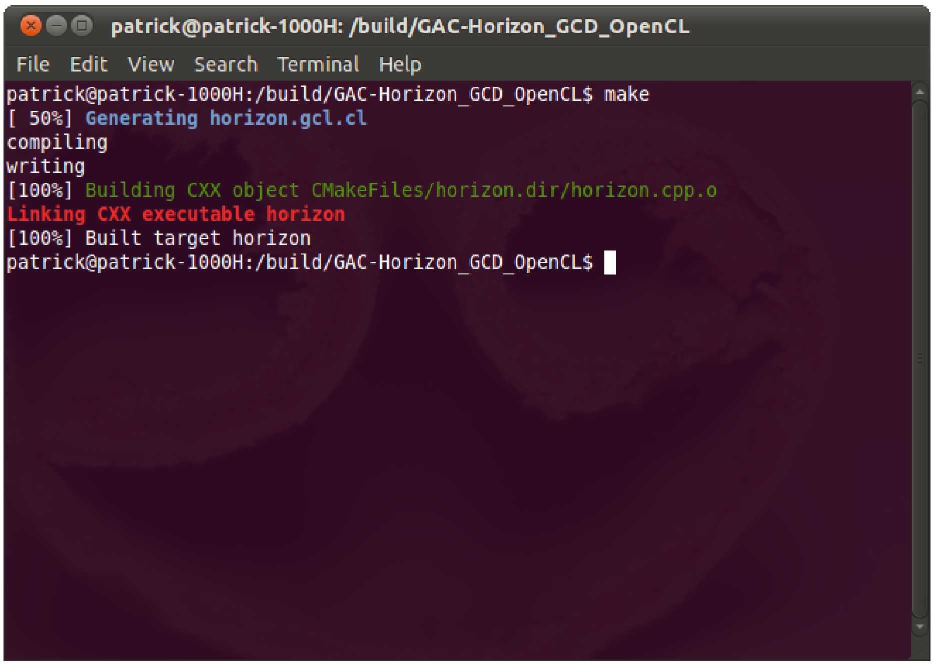
Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Eclipse CDT4 - Unix Makefiles

```
Configuring done
Generating done
```

Abbildung A.3.: CMake configuration for Gaalop GPC for OpenCL.

20. Open the CMake generated destination directory in your terminal or console.
    Enter make (Unix) or MinGW (Windows) and confirm with the Enter key.
    Wait until the build processes finishes.
21. Start the compiled application (figure A.4).
    Unix: ./horizon
    Windows: horizon.exe



patrick@patrick-1000H: /build/GAC-Horizon_GCD_OpenCL

```
patrick@patrick-1000H:/build/GAC-Horizon_GCD_OpenCL$ make
[ 50%] Generating horizon.gcl.cl
compiling
writing
[100%] Building CXX object CMakeFiles/horizon.dir/horizon.cpp.o
Linking CXX executable horizon
[100%] Built target horizon
patrick@patrick-1000H:/build/GAC-Horizon_GCD_OpenCL$
```

Abbildung A.4.: Screenshot of a Gaalop GPC for OpenCL build process.

Literaturverzeichnis

[1] Bullet continuous collision detection and physics library.

[2] Haskell.

[3] The homepage of geomerics ltd. Available at `http://www.geomerics.com`.

[4] Wikipedia entry on advanced vector extensions.

[5] Wikipedia entry on streaming simd extensions.

[6] Wikipedia entry on the lennard jones potential.

[7] Michael Burger. Das effiziente raytracen von dreiecksnetzen auf mehrkernprozessoren, gpus und fpgas mittels geometrischer algebra. Master's thesis, TU Darmstadt, 2011.

[8] Patrick Charrier. Gaalop compiler driver. In Bachelor thesis TU Darmstadt, 2011.

[9] Patrick Charrier and Dietmar Hildenbrand. Gaalop compiler driver. In proceedings of the GraVisMa workshop, Brno, 2010.

[10] Leo Dorst, Daniel Fontijne, and Stephen Mann. Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry. Morgan Kaufman, 2007.

[11] Ahmad Hosney Awad Eid. Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application. PhD thesis, Port-Said, 2010.

[12] 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.

[13] Free Software Foundation. Gnu make. http://www.gnu.org/software/make.

[14] 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.

[15] David Hestenes. New tools for computational geometry and rejuvenation of screw theory. In Eduardo Bayro-Corrochano and Gerik Scheuermann, editors, Geometric Algebra Computing in Engineering and Computer Science, volume 1, pages 3–33. Springer, May 2010.

[16] Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Andreas Koch. Specialized machine instruction set for geometric algebra. In submitted to AGACSE conference La Rochelle, 2012.

[17] Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Joachim Pitt. The Gaalop home page. Available at `http://www.gaalop.de`, 2012.

[18] 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.

[19] Khronos. OpenGL Specifications, 2010. http://www.opengl.org/documentation/specs/.

[20] Khronos OpenCL Working Group. The OpenCL Specification, version 1.0.29, 8 December 2008.

[21] Kanit Mekritthikrai. Opencl real-time raytracer, 2010. Available at `http://www.assembla.com/spaces/clrtrt/wiki`.

[22] NVIDIA. The CUDA home page. Available at `http://www.nvidia.com/object/cuda_home.html`, 2010.

[23] Christian Perwass. Geometric Algebra with Applications in Engineering. Springer, 2009.

[24] Christian Perwass. The CLU home page. Available at http://www.clucalc.info, 2010.

[25] Florian Seybold. Gaalet - a c++ expression template library for implementing geometric algebra, 2010.

[26] 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.

[27] Christian Steinmetz. Optimizing a geometric algebra compiler for parallel architectures using a table-based. In Bachelor thesis TU Darmstadt, 2011.

[28] Sun. Java. http://java.com/en/.

[29] ANTLR Development Team. Another tool for language recognition. Available at http://www.antlr.org.

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