Gaalop 2.0 - A Geometric Algebra Algorithm Compiler

Christian Schwinn
TU Darmstadt, Germany
Department of Computer Science
schwinn@rbg.informatik.tu-darmstadt.de

Dietmar Hildenbrand
TU Darmstadt, Germany
Department of Computer Science
Interactive Graphics Systems Group
dhilden@gris.informatik.tu-darmstadt.de

Florian Stock
TU Darmstadt, Germany
Department of Computer Science
Embedded Systems and Applications Group
stock@esa.informatik.tu-darmstadt.de

Andreas Koch
TU Darmstadt, Germany
Department of Computer Science
Embedded Systems and Applications Group
koch@esa.informatik.tu-darmstadt.de

ABSTRACT

In recent years, Geometric Algebra (GA) has become more and more popular in fields of science and engineering due to its potential for compact algorithms. However, the execution of GA algorithms and the related need for high computational power is still the limiting factor for these algorithms to be used in practice. Multivectors in 5D Conformal Geometric Algebra consist of 32 elements, each of which have to be calculated per basic operation. A nice property of Geometric Algebra is that elements of a multivector can usually be calculated independently. Therefore, it would be desirable to automatically detect parts that can be calculated in parallel by a software tool. In this paper, we present Gaalop 2.0, a Geometric Algebra Algorithm Compiler, which takes as input the description of a GA algorithm, symbolically optimizes the output multivectors and compiles the optimized code into a target language source file like C++, for instance. For each output multivector the code for non-zero coefficients is generated, which is finally adjusted to contain only basic arithmetic operations instead. This allows the optimized output to be compiled for different parallel computing platforms like FPGAs, for instance.

Keywords: Geometric Algebra, Geometric Computing, Compiler, Optimization, FPGA, Parallel Computing.

1 INTRODUCTION

Geometric Algebra is a mathematical framework that facilitates the development of algorithms in different fields of engineering and research. Algorithms in GA are geometrically intuitive and very compact compared to conventional approaches. Though, a major drawback is the increased runtime, which is an implication of high dimensions in multivectors of geometric algebras (2^n for dimension n). In order to make GA algorithms comparable to standard implementations, it is necessary to find optimizations that lead to a better runtime performance. Fortunately, the components of a multivector can be calculated independently of each other. Implementing multivectors as a set of coefficients with associated blades[1] makes it possible to find algebraic expressions for each coefficient separately. These coefficients can actually be calculated simultaneously, e.g. using parallel computing devices such as FPGAs. In this paper, we present Gaalop (Geometric Algebra Algorithm Optimizer), a compiler that calculates the very expressions for multivector components.

Gaalop optimizes Geometric Algebra algorithms written with the help of the CLUCalc software by Christian Perwass [7], using symbolic simplification backed by a Computer Algebra System (CAS), and compiles the output to different target languages. The optimized code has no more Geometric Algebra operations and is ready to be run efficiently on various platforms, particularly on parallel computing architectures. This software is based on [4], a proof-of-concept implementation for high performance computing based on Conformal Geometric Algebra with a preliminary version Gaalop.

This paper introduces Gaalop 2.0, a new version which has been completely rewritten in order to support multiple operating systems[2]. Gaalop 2.0 incorporates multiple new features, primarily the support for control flow instructions and code generation for parallel computing platforms. We describe the optimization and compilation process performed by Gaalop.

This document is organized as follows: Section 2 shows related work to the field of GA implementations, Section 3 describes the input format based on CLUCalc. Section 4 gives an overview of the intermediate representation that is used between the compilation steps. In Section 5 we show the optimization process. Section 6 concludes this paper and gives an outlook to future work.

[1] Blades are the basic geometric entities in geometric algebras. Multivectors consist of a linear combination of blades of different grades.

[2] Note that the first version of Gaalop has been Windows-only.

2 RELATED WORK

Gaigen [3] is a Geometric Algebra implementation generator that focuses on the generation of C++ code for specified algebra definitions. Given a signature and metric of a Geometric Algebra, Gaigen generates code that can be embedded into C++ applications directly. Generating code is quite a simple operation that does not require too much knowledge about Geometric Algebra features. However, implementations do not contain optimizations by default and thereby suffer from performance lacks that can only be eliminated by applying optimizations manually. This requires to identify special cases in which multivectors can be reduced in size (called specializations), for instance geometric entities like spheres which have only 1-dimensional blades (rather than bivectors, trivectors, etc.). Therefore, detailed knowledge is required by the use, since specializations cannot be deduced automatically. Gaalop does not require the step of manual optimization because multivectors are decomposed into their coefficients of basis blades and only non-zero coefficients are relevant to the generated output.

3 INPUT FORMAT

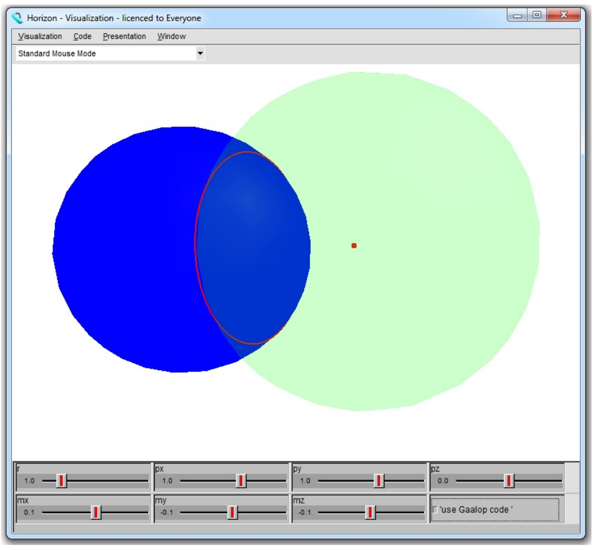
Gaalop 2.0 supports a subset of CLUScript, a script language for the 3D visualization and scientific calculation software CLUCalc/CLUViz [7]. Previous versions of Gaalop supported only sequential algorithms without conditional branches, loop statements or user-defined function calls. These are new features introduced in Gaalop 2.0. Table 1 shows the supported features.

CLUScript feature	supported
algebra definition	yes
pre-defined algebra functions	yes
macros	yes
null-space definition	yes
inner / outer / geometric products	yes
if-statements	yes
loops	not yet
variable lists / scopes / references	no
point operators	no
drawing / plotting functionality	no
LaTeX rendering	no

Table 1: CLUScript support in Gaalop 2.0

Restrictions to the full set of CLUScript features mainly concern visualization features or syntactic sugar like variable references or scopes.[3]

[3] Note that the ? operator is interpreted in a slightly different way than in CLUScript. In terms of Gaalop, this operator is used as a marker for lines of the input code for which the optimized output code should be generated. Variables or lines that are not marked accordingly will be processed by Gaalop to find simplified expressions for multivector coefficients but not generated as output code explicitly.


Horizon - Visualization - licenced to Everyone
Visualization Code Presentation Window
Standard Mouse Mode

r 1.0 | px 1.0 | py 1.0 | pz 0.0
mx 0.1 | my -0.1 | mz -0.1 | 'use Gaalop code'

Figure 1: Screenshot of the CLUViz visualization window. Sliders can be used to modify input parameters. The check box on the bottom right allows to switch between original code and Gaalop optimized code. Errors in the optimized code would immediately become visible when switching modes.

Example

```
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 1: Sample input code showing a CLUScript file. Variables px, py, pz and mx, my, mz are free variables that will be handled symbolically.

Listing 1 shows a sample input script. To illustrate the compilation process step by step, this code is taken as an example throughout this document. The task is to calculate an algebraic expression of the horizon on the earth viewed from an arbitrary point P. The earth is represented by a sphere S with center M and radius r (line 3). Line 4 defines a sphere K around view point P. The radius of this sphere is defined by the inner product of P and S, which corresponds to the squared distance between P and any point on S that touches a tangent through P. Thus, K has exactly the radius that matches the distance of the horizon to P. Finally, line 5 calculates the intersection circle C, which models the horizon.

Figure 1 shows how the CLUViz visualization window looks like. The earth sphere S is drawn in blue color, the view point P in red. The sphere K around P is indicated in light green. The intersection circle is finally drawn in red color.

4 INTERMEDIATE REPRESENTATION

Gaalop parses input files and transforms the input into an intermediate representation, on which different compilation steps operate. For the parser implementation, we used the ANTLR parser generator tool [6]. Gaalop 2.0 uses two kinds of intermediate representations (IR), a control flow graph (CFG) and a data flow graph (DFG) to represent the algorithmic structure of the input program and related arithmetic operations such as assignments or application of mathematic functions, respectively. In fact, Gaalop builds a *control dataflow graph*, representing arithmetic expressions in terms of a DFG whose nodes themselves are referenced in CFG nodes. Hence, the DFG is not a graph of its own but rather implicitly contained in CFG nodes.

4.1 Control Flow Graph

The control flow graph represents the overall structure of the input program. It distinguishes sequential statements such as assignments or procedure calls and control flow elements like if-statements or macros. As opposed to the data flow graph, the CFG does not represent details of arithmetic operations (e.g. additions, geometric products, etc.). Concrete types of CFG nodes are outlined below.

Assignments represent a variable to which an arbitrary expression is assigned. Both variable and expression are represented by an appropriate DFG node.

Optimization markers encapsulate a variable for which the optimized output code will be generated.

If-statements consist of a condition, a positive branch and a negative branch. Conditions are modeled via a DFG expression, branches as an (implicit) list of other CFG nodes. This type of node is self-contained, so nested statements are possible.

Macros are represented by a name and associated list of statements. The name is further used to identify usages of this macro in the input code. This can be used to inline the use of macros in order to augment the range of optimization by replacing the call of a macro by its actual code.

Some CFG nodes contain references to an arithmetic expression which is related to the respective code in the input program. These expressions are modeled by the data flow graph.

4.2 Data Flow Graph

The data flow graph represents the arithmetic parts of the input code. Elements of an assignment, such as variable and value, are modeled via DFG nodes. For each mathematical operation supported by Gaalop there exists a corresponding type of nodes, outlined below. The common basis of DFG nodes is an *expression* type. Concrete nodes can be placed on any location where an expression is expected.

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

Table 2: Blades of the 5D conformal geometric algebra and their ordering.

Unary operations are operations like negation or dualization, that take only one expression as argument.

Binary operations are operations like addition, subtraction or inner / outer / geometric products, which take two expressions as argument. Each binary operation has a left and right operand.

Language elements consist of pre-defined function calls (e.g. `VecN3` to define a conformal point) or mathematic functions like sin, cos or sqrt. These functions take different numbers and types of arguments, each of which is another expression.

Identifiers are the actual parameters to functions, operations and relations. This can be variables, integer or float constants and basis vectors.

Other CLUCalc relevant language elements like null space definitions or algebra selection are not modeled as dedicated DFG nodes. These are handled by a separate type which will be referenced to as *algebra signature*. This signature is directly referenced by the control flow graph, since contained properties are global to the input code and not related to single CFG or DFG nodes.

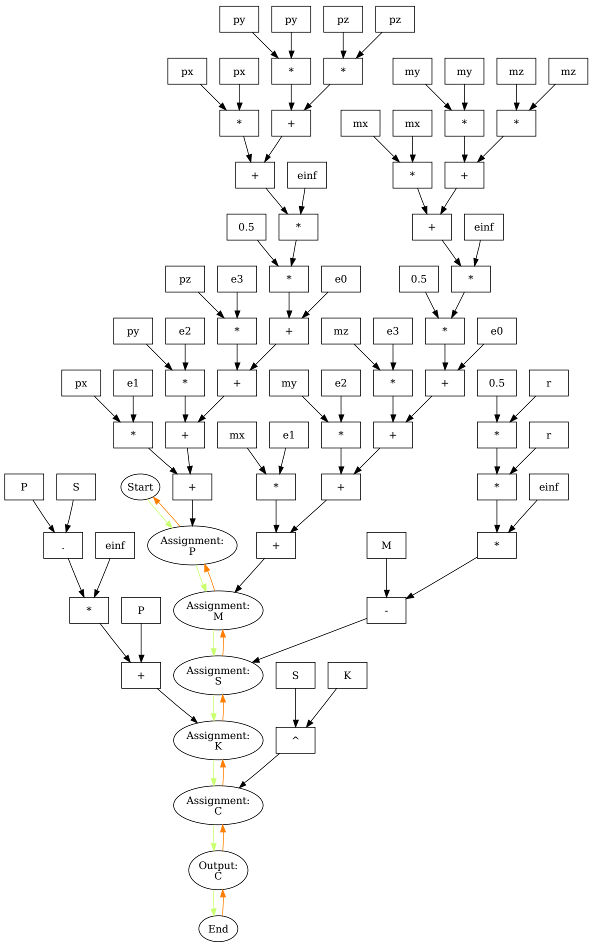


Figure 2: Control and data flow graph corresponding to the input file from section 3. The algebra definition from the input code is handled separately. Start and end node are special marker nodes. Control flow nodes are connected by colored arrows, other nodes correspond to the data flow graph.

CLUScript supports multiple algebra types, where the Conformal Geometric Algebra is the most important one, defined by the `DefVarsN3` function. Each type has an associated signature and blade list, which define the elementary basis blades of the underlying algebra. Table 2 lists the 32 basis blades of the Conformal Geometric Algebra in the canonical ordering. This table can also be seen as a lookup table for the association between multivector coefficient and the related blade, as it is used by the code generators (Section 5.3).

Example

Figure 2 shows the entire control dataflow graph which corresponds to the input file from Listing 1. The CFG part consists of only eight nodes which are connected by colored arrows. The rest of the graph corresponds to the DFG parts of the five assignments to variables P, M, S, K and C.

5 OPTIMIZATION

The compilation process from input file to the optimized output code consists of three major passes. Starting with the input file, Gaalop parses this file to produce the intermediate representation (CFG / DFG), optimizes the input by symbolic manipulation and generates the output code depending on the selected target language. These passes are illustrated below.

5.1 Parser

The code parser is responsible for parsing the CLUCalc input code and transforming it into an IR. Supported by the ANTLR parser generator [6], the input file is lexically analyzed, syntax checked and finally transformed into a control dataflow graph, as outlined in section 4. A lexer grammar defines lexical rules for keywords, identifiers, procedure calls and mathematical operations. The resulting token stream is processed by a parser grammar that actually checks the syntax of the input code and builds an abstract syntax tree (AST), which can be processed to build the control and data flow graphs. This transformation process is implemented by a transformer grammar, which traverses the AST and builds CFG / DFG nodes that are appended to the control dataflow graph. As Gaalop is implemented in Java, this grammar is the interface that connects the parsing process from the automatically generated parser code to the optimizer module in Gaalop.

5.2 Optimizer

The optimizer is responsible for symbolic simplification and calculation of optimized multivector coefficients for expressions marked in the input code. This compilation pass is implemented using the OpenMaple interface of the Maple CAS [5] with the CLIFFORD library by Rafal Ablamowicz [1] for Geometric Algebra calculations. This allows to use Maple for symbolic calculations from Java directly. The Maple optimizer traverses the input control dataflow graph as it has been set up by the parser in the previous compilation pass (cf. Figure 2).

Macro Inlining

Before the actual optimization pass is executed, another traversal is necessary to inline macros that have been defined in the input script. Therefore, the CFG is traversed in order to find references to self-defined macros. On each occurrence, the call to the macro along with its parameters is replaced by the actual code from the macro. This step is performed to increase the potential for symbolic optimization. Without macro inlining, only the code from the macro itself could be optimized, since no assumptions about input variables can be made. By inlining the macro code and implicitly gaining access to the actual input parameters it is possible to find situations where the macro code can be reduced to a small number of instructions, for instance when one of the parameter happens to be zero.

Special treatment is necessary for local variables and formal parameters in macros. In order not to override variables in the scope of the caller by simply copying the macro code to the place of the macro call, "macro-local" variables must be renamed in the case a variable with same name exists on the caller side. Consider a macro with local variable a and a code snippet calling the macro where another local variable a is defined. Replacing the macro call by its code would override the definition of a in the calling code. Therefore, the macro's local variable has to be renamed to another unused name.

Formal parameters in CLUCalc macros are represented as variable lists, referenced by `_P(i)`, where `i` specifes the i-th parameter. Thus, it is necessary to replace occurrences of `_P(i)` by the actual parameters from the macro call.

Optimization

Actual optimization of the input code is performed in this step. Gaalop 2.0 optimizes the input trying to achieve two objectives:

- Symbolic simplification. Each assignment of the input script is translated to Maple syntax and sent to the Maple engine. Maple keeps track of commands executed by the engine and symbolically simplifies assignments where appropriate.
- Preparation for parellel computing platforms. Multivector components, i.e. coefficients of a multivector's linear combination of basis blades, can be calculated independently. This holds potential for parallel execution of instructions calculating the non-zero coefficients. For an overview of the basis blades of the Conformal Geometric Algebra, please refer to Table 2.

To reach these goals, CFG nodes are processed to be sent to the Maple engine. The concrete representation is a string value describing the command to be executed by Maple. Therefore, control flow nodes are processed in the following way.

Assignments are translated to have the same variable name and value as specified in the CFG node. The assignment operator is translated to its Maple pendant (`:=`). The right-hand side of an assignment is handled according to the syntax rules defined by Maple. Standard arithmetics can be translated 1:1, whereas GA related operators have to be matched to the syntax specified in the CLIFFORDLIB package for Maple. The geometric product, for instance, has to be translated to use the `&c` operator, surrounded by a substitution of the clifford library symbol `Id` to 1. Consequently, an input line `a=b*c;` would be translated to `a:=subs(Id=1,b &c c);`.

Optimization markers trigger the calculation of simplified multivector coefficients for the selected variable. Therefore, a self-defined Maple procedure decomposes the multivector to its 2^n components. For each component, the relevant part of the linear combination of basis blades is selected. Finally, the related coefficient is evaluated and symbolically simplified. The result multivector is put into an array of multivector components which is used to get the outcome back into the control flow graph. Before returning control to Gaalop, another procedure tries to replace variable references that have previously been optimized, so results that have already been calculated can be reused in further calculations. Afterwards, Gaalop removes assignments to the old multivector and inserts new assignments for each non-zero multivector component. Each of these assignments represents the simplified expression that calculates the coefficient of the respective basis blade (Table 2).

If-statements are processed recursively by traversing the instructions of their positive and negative branches.

Macros have been removed in the previous pass and do not have to be considered anymore.

After this transformation, assignment nodes are replaced by the simplified expressions that have been calculated by Maple and inserted into the CFG. Hence, the CFG has been modified to contain the optimized code instead.

Example

The modified control dataflow graph corresponding to the example from Listing 1 is too large to be illustrated here. Nevertheless, it is easy to imagine the qualitative structure of the graph after the optimization pass. Remember that only one variable has been marked to be printed in the optimized output (the intersection circle C). Hence, the assignments to other variables have been removed without replacement (their contribution to the result is implicitly contained in the value of C). As the assignment to C has been replaced by the assignments to its multivector components, there are 10 new nodes now, representing the assignments to the bivector parts of the multivector (indices 7-16, see Table 2). At the end of the new graph there is an output (optimization) node representing the overall multivector C.

5.3 Code Generator

After the optimization pass, the intermediate representation contains the simplified expressions calculating the result multivectors that were marked in the input file. The IR is now ready to be processed by code generators, also called backends of the compiler. Each backend is implemented as a plugin to Gaalop which can be selected before the compilation process starts.

Code generators traverse the IR to translate the optimized code to the syntax rules of the target platform. For backends that do not support parallel computing, no modifications to the IR have to be performed. This is the case for most backends implemented in Gaalop 2.0 like CLUCalc, C++, DOT or LaTeX. Advanced backends supporting parallel computing platforms perform additional manipulations to the IR in order to prepare the output for parallel architectures like FPGAs. This is the case for the Verilog backend, for instance.

A common property of all backends is that no Geometric Algebra module has to be included. Since multivector descriptions have been optimized to expressions containing only standard arithmetics, no time-consuming operations, such as the geometric product, have to be executed. Generated output always operates on lists or arrays of coefficients, which are directly related to the blade indices in the canonical ordering of Table 2, rather than on the geometric entities (blades) themselves.

CLUCalc

Even if CLUCalc is the input language, it is also sensible to offer this as backend, too. Thereby, the correctness of the Gaalop compiler can be verified by comparing the original and optimized code visually in the CLUViz software (cf. Figure 1).

As a concrete example, listing 2 shows the resulting CLUCalc output code for the input file from section 1. `C` is defined as a list of 32 entries with coefficients 7 to 16 set to the optimized expressions as calculated from the optimizer, while other coefficients are zero. The associated blade indices correspond directly to the ones defined in Table 2. In the last line, `C` is reassembled using the coefficients and their associated basis blades.

C/C++

The C/C++ backend wraps the generated code into a `calculate` method that takes the unknown input variables and a reference to the output multivector as parameters. Multivectors are handled as float arrays whose indices correspond to the ones from Table 2 minus one, since counting in C++ starts with 0. Code generated from this backend can directly be copied into an existing C++ program without further modification. Passing the correct parameters to the calculate function, the result can be calculated without knowledge of GA operations.

Listing 3 shows the resulting C/C++ output code for the input file from section 1. `C` is defined as an array of float with 32 entries.

```
C=List(32);
C(7)= -(my*px)+mx*py; // e1^e2
C(8)= mx*pz-mz*px; // e1^e3
C(9)= ((-(0.5*mx*mz^^2.0)-0.5*mx*my^^2.0+0.5*mx*r
     ^^2.0+0.5*mx^^2.0*px)-0.5*px*my^^2.0-0.5*px*mz
     ^^2.0+0.5*px*r^^2.0)-0.5*mx^^3.0+mx*py*my+mx*pz
     *mz; // e1^einf
C(10)= -(1.0*px)+mx; // e1^e0
C(11)= -(mz*py)+my*pz; // e2^e3
C(12)= ((0.5*my^^2.0*py-0.5*my*mz^^2.0-0.5*my*mx
     ^^2.0+0.5*my*r^^2.0)-0.5*py*mz^^2.0-0.5*py*mx
     ^^2.0+0.5*py*r^^2.0)-0.5*my^^3.0+my*pz*mz+my*px
     *mx; // e2^einf
C(13)= -(1.0*py)+my; // e2^e0
C(14)= ((0.5*mz^^2.0*pz-0.5*mz*mx^^2.0-0.5*mz*my
     ^^2.0+0.5*mz*r^^2.0)-0.5*pz*my^^2.0-0.5*pz*mx
     ^^2.0+0.5*pz*r^^2.0)-0.5*mz^^3.0+mz*py*my+mz*px
     *mx; // e3^einf
C(15)= -(1.0*pz)+mz; // e3^e0
C(16)= (-(1.0*pz*mz)-1.0*px*mx-1.0*py*my+mz^^2.0)
     -1.0*r^^2.0+mx^^2.0+my^^2.0; // einf^e0
?C=C(7)*e1^e2+C(8)*e1^e3+C(9)*e1^einf+C(10)*e1^e0+C
     (11)*e2^e3+C(12)*e2^einf+C(13)*e2^e0+C(14)*e3^
     einf+C(15)*e3^e0+C(16)*einf^e0;
```

Listing 2: Optimized CLUCalc output for the input file from Listing 1 in section 1. Multivector components and associated blades are numbered according to Table 2, since counting of list elements in CLUCalc starts with 1.

```
void calculate(float mx, float my, float mz, float
     px, float py, float pz, float r, float **C_out)
{
   float C[32];
   C[6]=mx*py-my*px;
   C[7]=mx*pz-mz*px;
   C[8]=-(0.5f*mx*mz*mz)-0.5f*mx*my*my += 0.5f*mx*r*
        r*0.5f*mx*mx*px-0.5f*px*my*my-0.5f*px*mz*mz
        *0.5f*px*r*r-0.5f*pow(mx,3.0f)*mx*py*my*mx*
        pz*mz;
   C[9]=-(1.0f*px)*mx;
   C[10]=-(mz*py)*my*pz;
   C[11]=0.5f*my*my*py-0.5f*my*mz*mz-0.5f*my*mx*mx
        *0.5f*my*r*r-0.5f*py*mz*mz-0.5f*py*mx*mx*0.5
        f*py*r*r-0.5f*pow(my,3.0f)*my*pz*mz*my*px*mx
        ;
   C[12]=-(1.0f*py)*my;
   C[13]=0.5f*mz*mz*pz-0.5f*mz*mx*mx-0.5f*mz*my*my
        *0.5f*mz*r*r-0.5f*pz*my*my-0.5f*pz*mx*mx*0.5
        f*pz*r*r-0.5f*pow(mz,3.0f)*mz*py*my*mz*px*mx
        ;
   C[14]=-(1.0f*pz)*mz;
   C[15]=-(1.0f*py*my)-1.0f*pz*mz*my*my*mz*mz-1.0f*r
        *r*mx*mx-1.0f*px*mx;
   memcpy(C_out, C, sizeof(C));
}
```

Listing 3: Optimized C/C++ output for the input file from Listing 1 in section 1. Multivector components and associated blades are numbered according to Table 2 with indices decreased by 1 in order to match counting of array elements in C/C++.

DOT

The DOT backend generates a .dot file for visualization with the Graphviz Graph Vizalization Software [2]. This is helpful to inspect the intermediate representation as it has been modified by the optimizer. For example, Figure 2 shows the IR visualization of the input file without optimizations.

LaTeX

For scientific reports about algorithms optimized with Gaalop, it has been necessary to manually transform the output code to a human-readable text. The LaTeX backend automates this step by generating a description of the output code in form of math formulae that can be embedded in a .tex document. The following equations have been generated by Gaalop 2.0 according to the example from Listing 1 (excerpt):

$$
\begin{aligned}
C_6 &= mx * py - my * px \\
C_7 &= -mz * px + mx * pz \\
C_8 &= -\frac{1}{2}mx * mz^2 - \frac{1}{2}mx * my^2 + \frac{1}{2}mx * r^2 + ... \\
C_9 &= -1 * px + mx \\
C_{10} &= -mz * py + my * pz \\
C_{11} &= \frac{1}{2}my^2 * py - \frac{1}{2}my * mz^2 - \frac{1}{2}my * mx^2 + ... \\
C_{12} &= -1 * py + my \\
C_{13} &= \frac{1}{2}mz^2 * pz - \frac{1}{2}mz * mx^2 - \frac{1}{2}mz * my^2 + ... \\
C_{14} &= -1 * pz + mz \\
C_{15} &= -1 * py * my - 1 * pz * mz + my^2 + ...
\end{aligned}
$$

Verilog

TODO: somebody should describe this, as it is a very advanced backend with lots of further optimization. Maybe reference recent publications on this backend.

6 CONCLUSION & FUTURE WORK

We presented Gaalop 2.0, an advanced version of the Gaalop compiler. Written in Java, Gaalop 2.0 can be used on any platform where Maple is installed. We have introduced CLUScript as the input language for algorithms to be optimized with Gaalop, giving an overview of supported language features. After introducing the intermediate representation for the internal handling of the input code, we focused on the optimization process, giving details about the input parser, Maple simplifier and different code generators. We showed how Gaalop 2.0 transforms the input code to an optimized representation that goes without explicit Geometric Algebra operations. Exploiting the fact that multivector components can be calculated simultaneously, we have implemented a Verilog code generator which produces code that compiles the input algorithm to an FPGA hardware description.

For future releases of Gaalop we plan several extensions. Currently, we are working on an extended support for control flow instructions in input algorithms, particularly loops. Loops are another source of potential for parallelism: Unrolling loops of an input program where possible can lead to stronger optimizations through symbolic simplification while identifying parts of loops that operate on disjoint data makes it possible to execute these parts on different cores of a parallel computing platform.

Furthermore, we plan to extend the set of code generators by backends for general-purpose computing platforms. Different standards for multicore architectures like OpenMP or OpenCL offer the opportunity to make use of the huge processing power of modern computers. This is where two worlds come together: High-dimensional Geometric Algebras offering an elegant and intuitive way of describing algorithms, requiring considerable compute performance, and multicore architectures taking advantage of the increasing parallelism in integrated circuits as compensation of lacking performance of GA algorithms. The combination of these approaches advances to the vision of a “Geometric Algebra Computer” that accelerates standard implementations while keeping algorithms compact and intuitive.

REFERENCES

[1] Rafal Ablamowicz and Bertfried Fauser. CLIFFORD - A Maple Package for Clifford Algebra Computations. http://math.tntech.edu/rafal/.

[2] AT&T and Bell-Labs. Graphviz - Graph Visualization Software. http://www.graphviz.org/.

[3] Daniel Fontijne. Gaigen 2: A Geometric Algebra Implementation Generator. In *GPCE'06*. ACM, 2006.

[4] Dietmar Hildenbrand and Andreas Koch. Gaalop - High Performance Computing based on Conformal Geometric Algebra. http://www.gaalop.de/download.php?f=fe223b8c9c01b866293e9622e21f0307, 2008.

[5] Maplesoft. OpenMaple - an API into Maple. http://www.maplesoft.com/applications/view.aspx?SID=4383.

[6] Terence Parr. ANTLR Parser Generator. http://www.antlr.org.

[7] Christian Perwass. CLUCalc / CLUViz Interactive Visualization. http://www.clucalc.info, 2010.