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. 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 such as C++, for instance. For each output multivector the code for the different coefficients is generated, which is finally adjusted to contain only basic arithmetic operations instead. This allows the optimized output to be compiled for 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. Examples for how Geometric Algebra can be applied in engineering or computer graphics are described in [**?**], [**?**], [**?**] or [**?**], for instance. 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 in parallel. Implementing multivectors as a set of coefficients with associated basis blades makes it possible to find algebraic expressions for each coefficient separately. Blades are the basic geometric entities in geometric algebras. Multivectors consist of a linear combination of blades of different grades. Multivector coefficients can actually be calculated simultaneously, e.g. using parallel computing devices such as FPGAs. This paper presents Gaalop (Geometric Algebra Algorithm Optimizer), a compiler that calculates the very expressions for multivector components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Gaalop optimizes Geometric Algebra algorithms written with the help of the CLUCalc software [**?**], using symbolic simplification backed by the Maple [**?**] Computer Algebra System (CAS), and compiles the output to different target languages, currently C/C++, CLUCalc, Verilog, DOT and LaTeX. 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 [**?**], a proof-of-concept implementation for high performance computing based on Conformal Geometric Algebra with a preliminary version of Gaalop.

This paper introduces Gaalop 2.0, a new version which has been completely rewritten in Java in order to extend the feature set of previous versions and to support multiple operating systems. 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 2.0.

This document is organized as follows: Section 2 shows related work in 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.

Section 5 shows the optimization process. Section 6 describes different code generators. Finally, Section 7 concludes this paper and gives an outlook to future work.

2 RELATED WORK

There are a lot of pure hardware or pure software solutions in order to optimize the performance of GA algorithms. The first FPGA (Field Programmable Gate Array) solution was described in [**?**], focusing only on geometric products. In [**?**], the CliffoSor solution was presented, implementing products with a restriction to 4-dimensional Geometric Algebras. [**?**] is the first solution without an integer restriction for coefficients. All of these pure hardware solutions are focusing on the implementation of general Geometric Algebra operations but do not use the high potential of symbolic optimizations. Pure software approaches are especially based on expression templates libraries (especially boost::math::clifford) or the Gaigen approach.

Gaigen [**?**] 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 a simple operation that does not require too much knowledge of Geometric Algebra features. However, implementations do not contain optimizations by default and thus 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 such as spheres which have only 1-dimensional blades (rather than bivectors, trivectors, etc.). Therefore, detailed knowledge is required from the user, 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.

3 INPUT FORMAT

Gaalop 2.0 supports a subset of CLUScript, a script language for the 3D visualization and scientific calculation software CLUCalc/CLUViz [**?**].With CLUCalc it is possible to develop algorithms visually, allowing rapid prototyping. This enables the user to create Geometric Algebra algorithms step by step, supported by visual output. Previous versions of Gaalop supported only sequential algorithms without conditional branches, loop statements or user-defined function calls. Table 1 compares the supported and new features in Gaalop 1.0 and 2.0. Algorithms to be optimized by Gaalop have to use Conformal Geometric Algebra only.

	Version	
CLUScript feature	**1.0**	**2.0**
algebra definition	no	yes
pre-defined algebra functions	yes	yes
macros	no	yes
null-space definition	yes	yes
inner / outer / geometric products	yes	yes
if-statements	no	yes
loops	no	yes
variable lists / scopes / references	no	no
point operators	no	no
drawing / plotting functionality	no	no
LaTeX rendering	no	no

Table 1: Comparsion of CLUScript support in Gaalop 1.0 and 2.0.

Restrictions to the full set of CLUScript features mainly concern visualization features, variable references or scopes. Note that the ? operator is interpreted in a slightly different way than in CLUScript. In Gaalop, this operator is used as a marker for lines of the input code for which the optimized output code should be generated. Therefore, this operator will be referenced to as the optimization marker throughout this document. Variables or lines that are not marked accordingly will be processed by Gaalop to find simplified expressions for multivector coefficients but not explicitly generated as output code. This makes it possible to optimize only crucial parts of an algorithm instead of each line which might not be important in the output.

Example

```
DefVarsN3();
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.

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.

Listing 1 shows a sample input script in Conformal Geometric Algebra. 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 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 modeling the horizon.

Figure 1 shows how the CLUViz visualization window looks like. The earth sphere S is drawn in dark grey, the view point P in shown on the right-hand side. The sphere K around P is indicated in light grey. The intersection circle is finally indicated between both spheres.

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, the ANTLR parser generator tool has been used. 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. Previous versions of Gaalop used a very simple type of control flow graph, which contained only sequential nodes, since real control flow like branches or loops was not supported.

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.

Loops contain the statements from the body, including termination conditions which are usually related to if-statements. Optionally, a number of iterations can be given which is used to unroll the loop.

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 arithmetic expressions which are related to the respective code in the input program. These expressions are modeled by the data flow graph.

Example

Figure 2 shows the control flow graph which corresponds to the input file from Listing 1. For each assignment there is an according node in the CFG. The optimization marker is represented by a dedicated node.

Figure 2: Control 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.

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.

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

Binary operations model 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 of functions, operations and relations. These can be variables, integer or float constants and basis vectors.

Figure 3: Data flow graph corresponding to the assignment to variable S in the input file from section 3.

Other CLUCalc relevant language elements such as 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.

Gaalop supports Conformal Geometric Algebra, defined by the `DefVarsN3` function in CLUScript. This algebra has an associated signature and blade list, which defines the elementary basis blades. 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 6).

Example

Figure 3 shows the data flow graph which corresponds to the assignment to variable S in the input file from Listing 1.

5 OPTIMIZATION

Gaalop 2.0 has been re-designed to have a strictly modular interface. In previous versions, different parts of the program such as parser, optimizer and code generator have been hardwired and were neither exchangeable or extensible. In Gaalop 2.0, these parts are modeled as plugins which can easily be exchanged and extended without modifying the main program. Especially the intermediate representation has been separated to be accessible from all modules that read, modify or write this structure.

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 explained below.

The optimizer is responsible for symbolic simplification and calculation of optimized multivector coefficients for expressions marked with the optimization marker (`?`) in the input code. This compilation pass is implemented using the OpenMaple interface of the Maple Computer Algebra System (CAS) [**?**] with the CLIFFORD library by Rafal Ablamowicz [**?**] 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 (Figures 2, 3). On each optimization step, parts of the control or data flow graph are removed or replaced by expressions representing simplified calculations. After the optimization pass, the CFG and DFG contain only optimized code. Assignments which have not been marked for optimization are not contained in the graph anymore. Gaalop 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. The original assignments from the input CFG are removed and replaced in favor of new assignments, which calculate the individual multivector coefficients using scalar arithmetic operations only.
- Preparation for parallel computing platforms. Multivector components, i.e. coefficients of a multivector's linear combination of basis blades, can be calculated in parallel. This holds potential for parallel execution of instructions calculating the non-zero

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

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

coefficients. For an overview of the basis blades of the Conformal Geometric Algebra, refer to Table 2.

To reach these goals, CFG nodes are translated to Maple syntax and 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 according to the syntax rules of the CLIFFORDLIB package for Maple, e.g. using the `&c` operator for the geometric product.

Optimization markers trigger the calculation of simplified multivector coefficients for the selected variable. Thus, 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 resulting multivector is put into an array of multivector components which is used to return the expression 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 nonzero 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. In both branches, each line is optimized automatically in order to eliminate Geometric Algebra operations. This is necessary for the use of backends which do not support Geometric Algebra.

Macros are inlined in a separate pass. For each macro call, the corresponding code is copied and the call is replaced by the actual return value.

Loops can be processed in two ways: either being unrolled by a fixed number of iterations or similar to if-statements to eliminate Geometric Algebra operations. In each case, further information is required which can be given in the input file using dedicated `#pragma` comments. If the number of iterations for the loop is known, the loop can be unrolled by copying the body n times. In the normal case where only Geometric Algebra operations are removed, control variables have to be given in order to correctly process statements in the body. Due to the generalized CLUCalc syntax for loops, which does not specify the concrete type of loop, Gaalop would otherwise remove assignments to control variables like counter variables, breaking termination conditions in the output.

Listing 2 shows a simple example how a loop can be defined in the input file to be unrolled in Gaalop. The loop body is copied 10 times before the optimizer processes assignments, making the original loop disappear in the output. In the best case, the entire loop can be reduced to a single statement.

```
p = VecN3(x,y,z);
i = 0;
//#pragma unroll 10
loop {
   if (i > 9) {
      break;
   }
   // do something
}
?p;
```

Listing 2: Simple code fragment showing the use of unrolling loops.

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 general structure of the graph after the optimization pass. Remember that only one variable has been marked by `?` 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 6-15, see Table 2). At the end of the new graph there is an output (optimization) node representing the overall multivector C.

6 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 such as CLUCalc, C++, DOT or LaTeX. The more advanced Verilog backend performs additional manipulations to the IR in order to prepare the output for parallel architectures such as an FPGA.

A common property of all backends is that no Geometric Algebra module has to be included. Since the multivector descriptions have been optimized to expressions containing only conventional scalar arithmetic operators, 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 it as backend, too. In this manner, the correctness of the Gaalop compiler can be verified by comparing the original and optimized code visually in the CLUViz software (cf. Figure 1).

```
DefVarsN3();
C_opt = List(32);
C_opt(7) = -(my * px) + mx * py; // e1^e2
C_opt(8) = -(mz * px) + mx * pz; // e1^e3
C_opt(9) = (((mx * pz * mz + mx * py * my) - 0.5 *
     mx^^3.0 + 0.5 * mx^^2.0 * px) - 0.5 * mx * my
     ^^2.0 - 0.5 * mx * mz^^2.0 + 0.5 * mx * r^^2.0)
      - 0.5 * px * my^^2.0 - 0.5 * px * mz^^2.0 +
     0.5 * px * r^^2.0; // e1^einf
C_opt(10) = -(1.0 * px) + mx; // e1^e0
C_opt(11) = -(mz * py) + my * pz; // e2^e3
C_opt(12) = (((my * px * mx + my * pz * mz) - 0.5 *
     my^^3.0 - 0.5 * my * mx^^2.0 + 0.5 * my^^2.0 *
     py) - 0.5 * my * mz^^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; // e2^einf
C_opt(13) = -(1.0 * py) + my; // e2^e0
C_opt(14) = (((mz * px * mx + mz * py * my) - 0.5 *
     mz^^3.0 - 0.5 * mz * mx^^2.0 + 0.5 * mz^^2.0 *
     pz) - 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; // e3^einf
C_opt(15) = -(1.0 * pz) + mz; // e3^e0
C_opt(16) = ((-(1.0 * px * mx) - 1.0 * py * my + my
     ^^2.0 + mz^^2.0) - 1.0 * r^^2.0 + mx^^2.0) -
     1.0 * pz * mz; // einf^e0
?C = C_opt(7) * e1^e2 + C_opt(8) * e1^e3 + C_opt(9)
     * e1^einf + C_opt(10) * e1^e0 + C_opt(11) * e2^
     e3 + C_opt(12) * e2^einf + C_opt(13) * e2^e0 +
     C_opt(14) * e3^einf + C_opt(15) * e3^e0 + C_opt
     (16) * einf^e0;
```

Listing 3: Optimized CLUCalc output for the input file from Listing 1 in section 1. Multivector components and associated blades are indexed according to Table 2 incremented by 1, since counting of list elements in CLUCalc starts with 1.

As a concrete example, listing 3 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, incremented by 1 to match the 1-based counting of list elements in CLUCalc. In the last line, `C` is reassembled using the coefficients and their associated basis blades.

C/C++

The C/C++ backend optionally wraps the generated code in 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. Code generated from this backend can be used in an existing C++ program. Passing the correct parameters to the calculate function, the result can be calculated without knowledge of GA operations.

Listing 4 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.

```
void calculate(float mx, float my, float mz, float
     px, float py, float pz, float r, float C[32]) {
   C[6] = mx * py - my * px; // e1^e2
   C[7] = -(mz * px) + mx * pz; // e1^e3
   C[8] = mx * py * my + mx * pz * mz - 0.5f * pow(
        mx,3.0f) + 0.5f * mx * mx * px - 0.5f * mx *
         my * my - 0.5f * mx * mz * mz + 0.5f * mx *
         r * r - 0.5f * px * my * my - 0.5f * px *
        mz * mz + 0.5f * px * r * r; // e1^einf
   C[9] = -(1.0f * px) + mx; // e1^e0
   C[10] = -(mz * py) + my * pz; // e2^e3
   C[11] = my * px * mx + my * pz * mz - 0.5f * pow(
        my,3.0f) + 0.5f * my * my * py - 0.5f * my *
         mx * mx - 0.5f * my * mz * mz + 0.5f * my *
         r * r - 0.5f * py * mz * mz - 0.5f * py *
        mx * mx + 0.5f * py * r * r; // e2^einf
   C[12] = -(1.0f * py) + my; // e2^e0
   C[13] = mz * py * my + mz * px * mx - 0.5f * pow(
        mz,3.0f) + 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 + 0.5f * pz * r * r; // e3^einf
   C[14] = -(1.0f * pz) + mz; // e3^e0
   C[15] = -(1.0f * px * mx) - 1.0f * py * my - 1.0f
         * pz * mz + mz * mz - 1.0f * r * r + mx *
        mx + my * my; // einf^e0
}
```

Listing 4: 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.

DOT

The DOT backend generates a .dot file for visualization with the Graphviz Graph Visualization Software [**?**]. This is helpful to inspect the intermediate representation as it has been modified by the optimizer. For example, Figures 2 and 3 show parts of the input file's IR which was generated by the DOT code generator with the Maple optimization disabled.

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 equations from Figure 4 have been generated by Gaalop 2.0 according to the example from Listing 1.

$$
\begin{aligned}
C_6 &= mx * py - my * px \\
C_7 &= -mz * px + mx * pz \\
C_8 &= -\frac{1}{2}mx * mz^2 - \frac{1}{2}mx * my^2 + \frac{1}{2}mx * r^2 \\
&\quad + \frac{1}{2}mx^2 * px - \frac{1}{2}px * my^2 - \frac{1}{2}px * mz^2 \\
&\quad + \frac{1}{2}px * r^2 - \frac{1}{2}mx^3 \\
&\quad + mx * py * my + mx * pz * mz \\
C_9 &= -1 * px + mx \\
C_{10} &= -mz * py + my * pz \\
C_{11} &= \frac{1}{2}my^2 * py - \frac{1}{2}my * mz^2 - \frac{1}{2}my * mx^2 \\
&\quad + \frac{1}{2}my * r^2 - \frac{1}{2}py * mz^2 - \frac{1}{2}py * mx^2 \\
&\quad + \frac{1}{2}py * r^2 - \frac{1}{2}my^3 \\
&\quad + my * pz * mz + my * px * mx \\
C_{12} &= -1 * py + my \\
C_{13} &= \frac{1}{2}mz^2 * pz - \frac{1}{2}mz * mx^2 - \frac{1}{2}mz * my^2 \\
&\quad + \frac{1}{2}mz * r^2 - \frac{1}{2}pz * my^2 - \frac{1}{2}pz * mx^2 \\
&\quad + \frac{1}{2}pz * r^2 - \frac{1}{2}mz^3 \\
&\quad + mz * py * my + mz * px * mx \\
C_{14} &= -1 * pz + mz \\
C_{15} &= -1 * py * my - 1 * px * mx - 1 * pz * mz \\
&\quad + mz^2 - 1 * r^2 + mx^2 + my^2
\end{aligned}
$$

Figure 4: LaTeX representation of the optimized output for the input file from Listing 1 in section 1. The actual LaTeX code is wrapped into an align environment.

Verilog

The Verilog Hardware Description Language (HDL) is used to describe hardware circuits. The Verilog backend produces synthesizeable Verilog HDL code, i.e. it can be used for the production of an application specific circuit (ASIC) or it can be mapped on a reconfigurable unit, like a field programmable gate array (FPGA). Similar to the C/C++ backend an independent module is generated, which does not rely on a specific architecture or infrastructure.

As the generated hardware is fully spatial parallel and pipelined, it consumes in each cycle a date and returns a computation result. The latency of the computation depends on the input program and the involved operations.

To generate the hardware, the backend transforms the control flow graph into a pure data flow graph. As more complex control flow like loops is currently not yet supported in this backend, this is always possible. The nodes of the newly generated data flow graph consist only of operators which can be mapped to Verilog via a special library.

Before the final mapping to the operations is done, some optimization is applied to reduce the required silicon area: Constant Propagation, Constant Folding, Common Subexpression Elimination, and in case the user selected not to implement the computation as floating point but as fixed point a Monte-Carlo-Simulation for word length optimization.

Optimization continues while mapping hardware. Multiplications and division by a power of two can be replaced in hardware with a new wiring, and each operator is instantiated with the right size to minimize area requirements.

7 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.

We plan to extend the set of code generators by backends for general-purpose computing platforms. Different standards for multicore architectures like OpenMP (Open Multi-Processing) [**?**] or OpenCL (Open Computing Language) [**?**] 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 computing 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.