Gaalop - High Performance Parallel Computing based on Conformal Geometric Algebra

Dietmar Hildenbrand, Joachim Pitt, Andreas Koch

Abstract We present Gaalop (Geometric algebra algorithms optimizer), our tool for high performance computing based on conformal geometric algebra. The main goal of Gaalop is to realize implementations that are most likely faster than conventional solutions. In order to achieve this goal, our focus is on parallel target platforms like FPGA (field-programmable gate arrays) or the CUDA technology from NVIDIA. We describe the concepts, the current status, as well as the future perspectives of Gaalop dealing with optimized software implementations, hardware implementations as well as mixed solutions. An inverse kinematics algorithm of a humanoid robot is described as an example.

1 Introduction

In recent years, geometric algebra, and especially the 5D Conformal geometric algebra, has proved to be a powerful tool for the development of geometrically intuitive algorithms in a lot of engineering areas like robotics, computer vision and computer graphics. However, runtime performance of these algorithms was often a problem.

In this chapter, we present our approach for the automatic generation of high performance implementations with a focus on parallel target platforms like FPGA or CUDA. In the sections 2 and 3, we present some related work as well as the basics of conformal geometric algebra.

Dietmar Hildenbrand and Joachim Pitt
University of Technology Darmstadt, Germany
Interactive Graphics Systems Group
e-mail: dhilden@gris.informatik.tu-darmstadt.de

Andreas Koch
University of Technology Darmstadt, Germany
Embedded Systems and Applications Group
e-mail: koch@esa.informatik.tu-darmstadt.de

Our main goal with Gaalop is to realize implementations that are most likely faster than conventional solutions. The main concepts combining both approaches for the optimization of software and of hardware implementations are presented in section 4. The corresponding architecture of Gaalop is described in section 5. An inverse kinematics algorithm for the leg of a humanoid robot is presented in section 6 as a complex example for the use of Gaalop. The current status of Gaalop as well as its future perspectives can be found in section 7.

2 Related Work

Despite the tremendous expressive power of geometric algebra, it has only seen limited use in practical applications. One of the reasons for this might be that the actual processing of geometric algebra algorithms requires significant computational effort. Related tools with the intention of optimizing geometric algebra implementations focus either on pure software or pure hardware solutions.

2.1 Software Implementations

The most advanced pure software solution is Gaigen developed at the university of Amsterdam (see [2] and [4]). You can find some benchmarks comparing Gaigen with other software implementations in [4].

2.2 Hardware Implementations

To resolve the above mentioned quandary, it is promising to look at dedicated hardware architectures for the acceleration of geometric algebra algorithms. Current integrated circuit technology offers a means to achieve this in the form of field-programmable gate arrays (FPGAs). These *reconfigurable* devices allow the implementation of a wide variety of digital logic circuits without the need for a very expensive photochemical circuit fabrication. Furthermore, the same device is able to realize different logic circuits by reconfiguring them onto the same silicon area.

2.2.1 Prior Attempts

The first serious approach is described by Perwass et al. [17]. That accelerator realizes the geometric product implemented on a 20 MHz FPGA connected via the PCI bus to the host computer. Due to the limited capacity of the FPGA employed, techniques such as wide parallel or pipelined processing, and the use of fast on-chip

memories, were not exploited. Similarly, subspace coefficients consist only of 24 bit integers, other fixed or floating point formats are not supported. The architecture is able to process multivectors of up to eight dimensions, with smaller vectors being processed faster. While the resulting accelerator does achieve a speedup over a conventional software programmable processor when counting clock cycles, comparisons with actual clock cycles lead to a practical *slow-down* when using the FPGA-based solution over simple software running on a conventional computer.

A different approach was presented by Gentile et al. [6]: This accelerator supports functions beyond the geometric product, namely, the outer product, contractions etc., each being implemented on a dedicated hardware unit. The architecture is limited to multivectors of three to four dimensions. As before, the coefficients are limited to integers, in this case 16 bit wide. The FPGA implementation requires a lot of communication with the host computer over the PCI bus. And additionally, when taking the different clock frequencies into account to compute the real world execution times, this approach does not lead to a speedup compared to a software implementation.

An update of this work is given by Franchini et al. [5]: the operation-specific hardware units have now been replaced by a variable number of so-called slices. Each slice is able to compute all operations of the four-dimensional geometric algebra. The coefficients have now been extended to 32 bit integers. In terms of hardware, a slice consists of a 32 bit wide arithmetic logic unit capable of addition, subtraction, multiplication, and logical computations. The geometric algebra operations are decomposed into these primitive calculations, with their execution being orchestrated step-by-step by on-chip software (microcode). The FPGA implementation achieves a clock frequency of 45 MHz and runs by a factor 3x to 4x faster than a software programmable processor when counting cycles. When actually considering the 2 GHz clock frequency of the reference processor, the actual execution time again slows down by a factor of 9x to 12x versus software.

The first coprocessor to lift the integer limitation on coefficients is the custom-fabricated integrated circuit (ASIC) implementation introduced by Mishra and Wilson [13], which allows two-dimensional multivectors with double precision floating-point coefficients. At its core, it consists of a floating point adder and multiplier each, supported by smaller hardware units to compute the product of basis blades. While pipeline-parallel execution is employed within these compute units, actual geometric algebra operations (geometric product, rotor, etc) are again computed sequentially by decomposing them into primitive calculations controlled by microcode. The experimental evaluation of the system in [14] shows a real *wall-clock speed-up* of 3x over a software programmable processor.

2.3 Our Proof-of-Concept Approach

In [9] we could show that an approach with symbolic simplification of geometric algebra algorithms is able to lead to an implementation which is three times faster

than a conventional solution. In a second stage we implemented this inverse kinematics algorithm also on hardware and got an additional speedup of more than 100 times (see [10]).

When studying all of the prior hardware attempts, it is obvious that most of them lead to an application slowdown instead of the hoped-for acceleration. The major reason for this disappointing result is due to the architectural choices made. The discrepancy in achievable clock frequencies of conventional processors (which are now into multiple gigahertz), and that of FPGAs (which currently top out at 500-600 MHz), implies the need for massive parallelism in the FPGA to achieve better performance.

As a proof-of-concept, we implemented an accelerator [10] for a specific geometric algebra algorithm, namely the inverse kinematics of the arm of a virtual human. It is a completely different architectural approach compared to the approaches described above. Instead of coarse granular computation units capable of handling entire geometric algebra operators, we decomposed the geometric algebra description into the underlying scalar equations. These equations are optimized symbolically and employ only basic arithmetic operators. The resulting set of equations was then implemented one arithmetic operator at a time. For each of these arithmetic operators, we carefully examined the range of values to be processed for the specific problem. With this data, and external requirements on computational precision (in this case, the positional accuracy of the hand), we determined for each operator the optimal numerical representation (e.g., values in the range of 0 to 100 with 1/16mm of accuracy would be represented as 11 bit unsigned fixpoint numbers). The circuits of the operators were then optimally matched to their representation as well as to one of their operands being the constant.

The resulting accelerator, which exploits parallelism between multivector components, between fine-grained arithmetic operators, and in a pipelined fashion over the entire computation, achieves currently a real-world speedup in execution time of 185x over a conventional processor with a 1.5 GHz clock frequency. The compute pipeline consists of 363 stages with an average of 12 arithmetic operators per stage. This extreme degree of parallelism allows the real-world acceleration even though the FPGA device (which is by now two generations out of date) only runs at 100 MHz clock frequency.

One of the aims of the Gaalop project is to develop a tool flow for automatically executing the optimization and hardware generation which we had to perform manually for our reference design. Before giving an overview of the planned flow, we will first give a brief introduction into geometric algebra.

3 Conformal geometric algebra

While points and vectors are normally used as basic geometric entities, in the 5D conformal geometric algebra we have a wider variety of basic objects. For example, spheres and circles are simply represented by algebraic objects. To represent a circle

you only have to intersect two spheres, which can be done with a basic algebraic operation. Alternatively you can simply combine three points to obtain the circle through these three points.

Table 1 lists the two representations of the geometric entities in conformal geometric algebra. In this table **x** and **n** are marked bold to indicate that they represent 3D entities as linear combination of the 3D base vectors e_1, e_2 and e_3.

$$\mathbf{x} = x_1 e_1 + x_2 e_2 + x_3 e_3 \tag{1}$$

The additional two base vectors are indicated by

- e_0 representing the 3D origin
- e_∞ representing the point at infinity

The $\{s_i\}$ represent different spheres and the $\{\pi_i\}$ different planes.

Table 1 Representations of the conformal geometric entities

entity	**standard representation**	**direct representation**
Point	$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$	
Sphere	$s = P - \frac{1}{2}r^2 e_\infty$	$s^* = x_1 \wedge x_2 \wedge x_3 \wedge x_4$
Plane	$\pi = \mathbf{n} + d e_\infty$	$\pi^* = x_1 \wedge x_2 \wedge x_3 \wedge e_\infty$
Circle	$z = s_1 \wedge s_2$	$z^* = x_1 \wedge x_2 \wedge x_3$
Line	$l = \pi_1 \wedge \pi_1$	$l^* = x_1 \wedge x_2 \wedge e_\infty$
Point Pair	$P_p = s_1 \wedge s_2 \wedge s_3$	$P_p^* = x_1 \wedge x_2$

The two representations are dual to each other. In order to switch between the two representations, the dual operator which is indicated by '*', can be used. For example in the standard representation a sphere is represented with the help of its center point P and its radius r, while in the direct representation it is constructed by the outer product '$\wedge$' of four points x_i that lie on the surface of the sphere ($x_1 \wedge x_2 \wedge x_3 \wedge x_4$). In standard representation the dual meaning of the outer product is the intersection of geometric entities. For example a circle is defined by the intersection of two spheres ($s_1 \wedge s_2$).

Blades are the basic computational elements and the basic geometric entities of geometric algebras. The 5D conformal geometric algebra consists of blades with **grades** 0, 1, 2, 3, 4 and 5, whereby a scalar is a **0-blade** (blade of grade 0). The element of grade five is called the pseudoscalar. A linear combination of blades is called a **k-vector**. So a bivector is a linear combination of blades with grade 2. Other k-vectors are vectors (grade 1), trivectors (grade 3) and quadvectors (grade 4). Furthermore, a linear combination of blades of different grades is called a **multivector**. Multivectors are the general elements of a geometric algebra. Table 2 lists all the 32 blades of conformal geometric algebra. The indices indicate 1: scalar, 2...6: vector, 7...16: bivector, 17...26: trivector, 27...31: quadvector, 32: pseudoscalar.

A point $P = x_1e_1 + x_2e_2 + x_3e_3 + \frac{1}{2}\mathbf{x}^2e_\infty + e_0$ (see table 1 and equation (1)) for instance can be written in terms of a multivector as the following linear combination of blades

$$P = x_1 * blade[2] + x_2 * blade[3] + x_3 * blade[4] + \frac{1}{2}\mathbf{x}^2 * blade[5] + blade[6] \quad (2)$$

For more details please refer for instance to the book [2] as well as to the tutorials [8] and [7].

Table 2 The 32 blades of the 5D conformal geometric algebra

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

4 Concepts

The main goal of Gaalop is the combination of the elegance of algorithms using geometric algebra with the generation of implementations that are most likely faster than conventional implementations. Depending on the application these can be either optimized software implementations or optimized hardware implementations or a mixture between them.

For that purpose we propose a two-stage approach with

- symbolic optimization
- use of the inherent fine-grained parallel structure

of geometric algebra algorithms. Algorithms can vary from just a set of formulas to complex control flows.

4.1 Symbolic Optimization

We use the symbolic computation functionality of Maple (together with a library for geometric algebras [1]) in order to optimize the geometric algebra algorithm. Algorithms can be developed visually with CLUCalc [16] and afterwards be optimized with Gaalop. Gaalop parses and translates the CLUCalc code to Maple code. A small Maple library provided with Gaalop implements correspondent CLUCalc functions in Maple. Maple computes the coefficients of the desired variable symbolically, returning an efficient implementation depending just on the input variables.

As an example the following CLUCalc code computes the intersection circle C of two spheres $S1$ and $S2$. While CLUCalc requires the definition of the variables $x1$, $x2$, $x3$, $y1$, $y2$, $y3$, $r1$ and $r2$, we don't want to compute with fixed values for these variables. So just the second part is needed for Gaalop.

```
DefVarsN3();
:IPNS;

x1 = 0.2; x2 = 0.3; x3 = 0.5; r1 = 0.7;
y1 = 0.7; y2 = 1.1; y3 = 1.3; r2 = 0.9;

// Gaalop uses the code below

P1 = x1*e1 +x2*e2 +x3*e3 +1/2*(x1*x1+x2*x2+x3*x3)*einf +e_0;
P2 = y1*e1 +y2*e2 +y3*e3 +1/2*(y1*y1+y2*y2+y3*y3)*einf +e_0;

S1 = P1 - 1/2 * r1*r1 * einf;
S2 = P2 - 1/2 * r2*r2 * einf;

?C = S1 ^ S2;
```

A question mark in CLUCalc at the beginning of a line prints the result after evaluation of the corresponding line in the output window. Gaalop interprets these question marks almost the same, as it computes and prints out the coefficients of the following variable symbolically, depending on the previous input.

The computation of the conformal points $P1$ and $P2$ and the spheres $S1$ and $S2$ correspond to table 1.

The resulting C code generated by Gaalop for the intersection circle C is as follows and only depends on the variables $x1$, $x2$, $x3$, $y1$, $y2$, $y3$, $r1$ and $r2$:

```
float C [32] = {0.0};

C[7] = x1*y2-x2*y1;
C[8] = x1*y3-x3*y1;

C[9] = -0.5*y1*x1*x1-0.5*y1*x2*x2-0.5*y1*x3*x3+0.5*y1*r1*r1
       +0.5*x1*y1*y1+0.5*x1*y2*y2+0.5*x1*y3*y3-0.5*x1*r2*r2;

C[10] = -y1+x1;
C[11] = -x3*y2+x2*y3;
```

```
C[12] = -0.5*y2*x1*x1-0.5*y2*x2*x2-0.5*y2*x3*x3+0.5*y2*r1*r1
        +0.5*x2*y1*y1+0.5*x2*y2*y2+0.5*x2*y3*y3-0.5*x2*r2*r2;

C[13] = -y2+x2;

C[14] = -0.5*y3*x1*x1-0.5*y3*x2*x2-0.5*y3*x3*x3+0.5*y3*r1*r1
        +0.5*x3*y1*y1+0.5*x3*y2*y2+0.5*x3*y3*y3-0.5*x3*r2*r2;

C[15] = -y3+x3;

C[16] = -0.5*y3*y3+0.5*x3*x3+0.5*x2*x2+0.5*r2*r2
        -0.5*y1*y1-0.5*y2*y2+0.5*x1*x1-0.5*r1*r1;
```

Gaalop always computes optimized 32-dimensional multivectors. Since a circle is described with the help of a bivector, only the blades 7 to 16 (see table 2) are used. As you can see, all the corresponding coefficients of this multivector are very simple expressions with basic arithmetic operations. A more complex example is described in section 6.

4.2 Use of Inherent Fine-Grained Parallel Structure

With the help of symbolic optimization the geometric algebra algorithm is transformed into an algorithm computing the coefficients of 32D multivectors using only basic arithmetical operations. This can be implemented very efficiently in digital logic on silicon devices such as FPGAs using parallel computation of coefficients of multivectors, deeply pipelined processing, and the exploitation of constant values by propagating them directly into the circuit. These techniques are described in section 2.3 and in more detail in [10] for an inverse kinematics example.

5 The Architecture of Gaalop

Figure 1 shows an overview over the architecture of Gaalop. Its input is a geometric algebra algorithm written in CLUCalc (see [16]). Via symbolic simplification it is transformed into a generic intermediate representation (IR) that can be used for the generation of different output formats. Gaalop supports sequential platforms such as C and Java as well as parallel platforms such as CUDA [15] or FPGA descriptions (as a structural hardware description, currently written in the Verilog language). CLUCalc can also be used as an output format in order to visualize the optimized results (see [16]).

The basis of the mapping the IR, which is expressed on an abstract mathematical/behavioral level, to a hardware accelerator is the technology already used in the COMRADE compiler [12]. COMRADE is designed to translate from ANSI

C (complete language, no additional user annotations required) into hybrid hardware/software applications, with the hardware parts being executed on an FPGA. Since geometric algebra algorithms are far more abstract than C (which contains, e.g. pointers and gotos), they are considerably easier to optimize and translate efficiently to an FPGA-based accelerator.

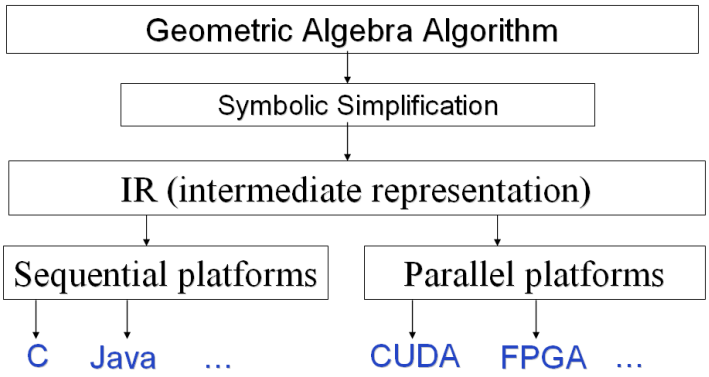

Fig. 1 Architecture of Gaalop

6 Inverse Kinematics of the Leg of a Humanoid Robot

In this section we present an inverse kinematics algorithm for the leg of the humanoid robot 'Mr. DD Junior 2' (see figure 2). We use CLUCalc code as an example for the input language of Gaalop. Parts of the generated C code are shown as an example for a target implementation of Gaalop.

'Mr. DD Junior 2' is a humanoid robot of about 38 cm total height and was used for the 2005 RoboCup competition [3] where robots play soccer completely autonomously. Its legs have six degrees of freedom each: from hip to foot, the first joint rotates about the forward oriented axis, the next three joints about the sidewards oriented axis, and the last two joints about the forward and upward oriented axis. This is different from the standard configuration of humanoid robot legs, where the joint that rotates about the upward oriented axis usually is located in the hip. The robot is equipped with a camera for vision, a pocket PC for computation and servo motors for actuation. The robot must localize itself on the field which has color-coded landmarks, identify other players, the location of the ball and the goal. For walking, 'Mr. DD Junior 2' uses the following inverse kinematics approach: The motion of the hips and feet are given by smooth trajectories, that are described by several parameters and the joint angle trajectories of the legs are computed from the hips and feet trajectories by inverse kinematics. This computation is done online on the pocket PC, which also is used for image processing.

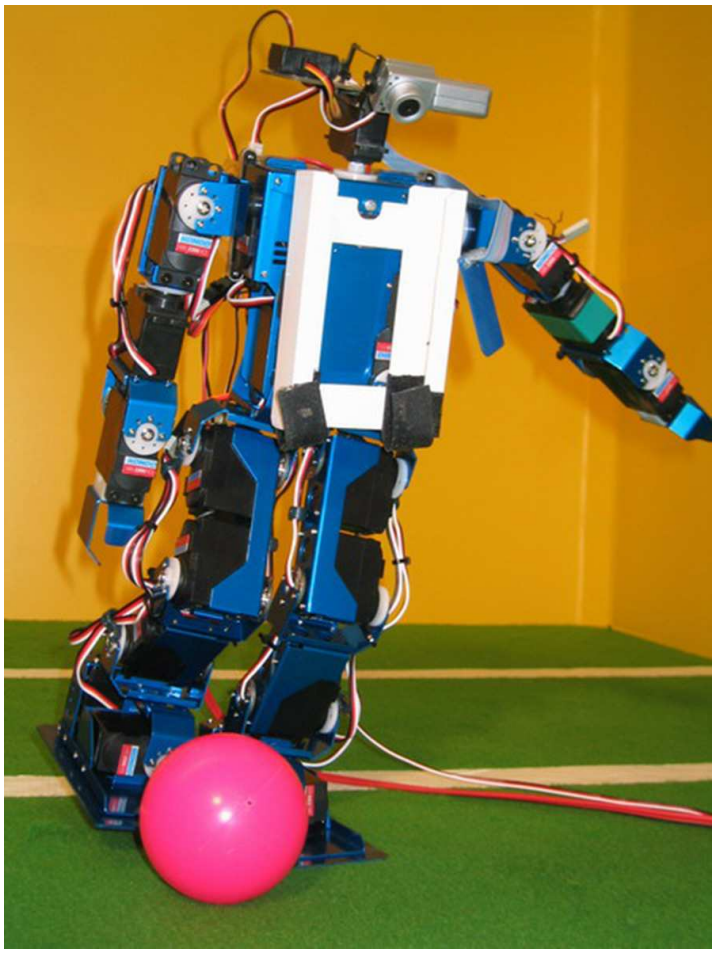

Fig. 2 The robot 'Mr. DD Junior 2' of the RoboCup team, the Darmstadt Dribbling Dackels (DDD)

6.1 Solving the Inverse Kinematics Algorithm

The following inverse kinematics algorithm has been developed using conformal geometric algebra to solve the 6 DOF kinematic chain for the leg of the humanoid robot 'Mr. DD Junior 2' (see figure 3). The leg consists of joints with one degree of freedom each. The hip (P_1) defines the first joint and lies in the origin, rotating about the x-axis. Three joints rotating about the y-axis, one rotating about the x-axis and a final one rotating about the z-axis follow, leading to the foot-point (P_7). The coordinates of the foot (P_x, P_y, P_z), the normal of the foot (n) and the length of the links ($l_1, l_2, l_3, l_4, l_5, l_6$) are needed to solve the inverse kinematics chain. Please refer to table 3 for a list of the input and output parameters of the algorithm.

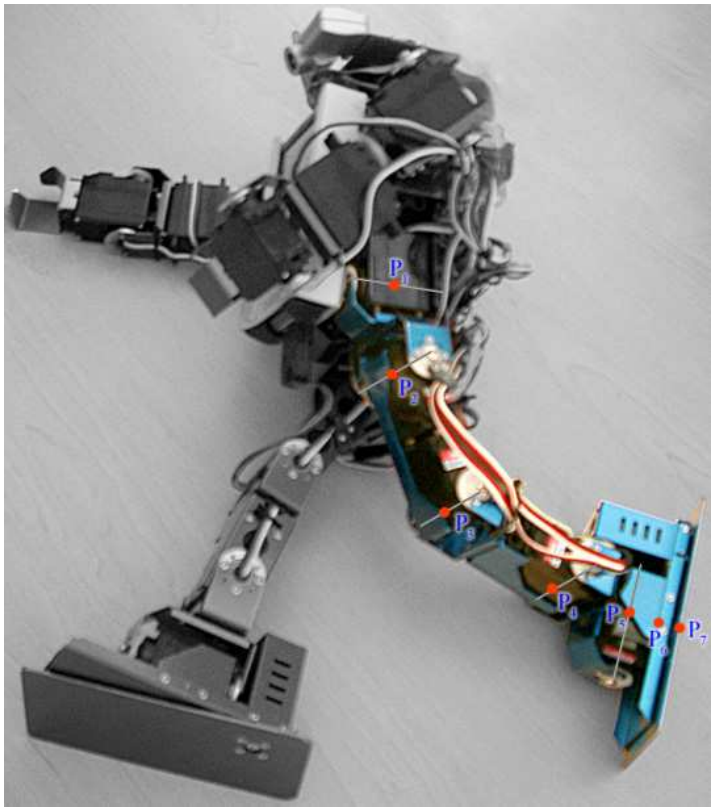

Fig. 3 The leg of the robot 'Mr. DD Junior 2' with the indication of the points P_1 to P_7

Table 3 Input/Output of the inverse kinematics algorithm

Input	
var	**description**
P_x	foot x-value
P_y	foot y-value
P_z	foot z-value
n	normal of foot x-value normal of foot y-value normal of foot z-value
l_1	length of 1^{st} link
l_2	length of 2^{nd} link
l_3	length of 3^{rd} link
l_4	length of 4^{th} link
l_5	length of 5^{th} link
l_6	length of 6^{th} link

Output	
var	**description**
ang_1	angle at P_1
ang_2	angle at P_2
ang_3	angle at P_3
ang_4	angle at P_4
ang_5	angle at P_5
ang_6	angle at P_6

6.1.1 Computation of the positions of link 5 and 6 in the kinematics chain

Since there is only a rotation about the z-axis in P_6, link 5 and 6 (in P_5 and in P_6) are on the normal (n) of the foot. The translator T_1 is needed to translate P_7 about l_6 in direction of n.

$$\begin{aligned} T_1 &= 1 - \left(\frac{1}{2}\, n\, l_6\right) e_\infty \\ P_6 &= T_1\, P_7\, \tilde{T}_1 \end{aligned} \tag{3}$$

Please notice that a translator is defined by the expression $T = 1 - \frac{1}{2}\, t_{vec}\, e_\infty$ with t_{vec} being the 3D translation vector and that a translation is defined by a multiplication of the translator from the left and of its reverse from the right. Another translator (T_2) is necessary to compute P_5, where the distance to P_7 is $l_5 + l_6$.

$$\begin{aligned} T_2 &= 1 - \left(\frac{1}{2}\, n\, (l_5 + l_6)\right) e_\infty \\ P_5 &= T_2\, P_7\, \tilde{T}_2 \end{aligned} \tag{4}$$

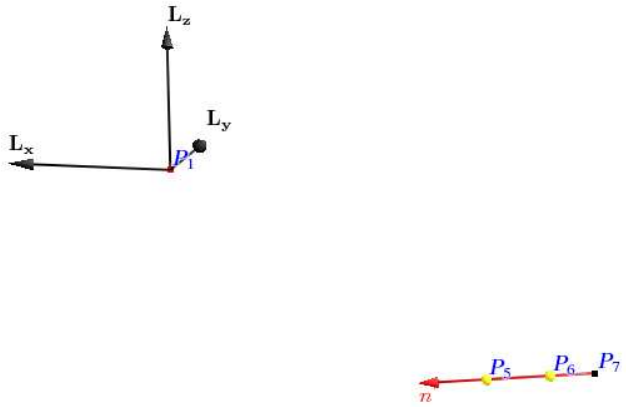

Fig. 4 Step A. - translating P_7 by l_6 and l_5 in direction of n to get P_6 and P_5

6.1.2 Computation of the position of link 4

By taking a closer look at the kinematic chain, one will notice, that P_1, P_2, P_3, P_4, P_5 define a plane π_3, which includes the x-axis. Since the joints in P_2, P_3 and P_4 all

rotate about the y-axis, these and the joints directly connected to them (P_1 and P_5), are all in one plane. Every plane can be defined by 3 points, which here are P_1, P_5 and the auxiliary point on the x-axis P_{H2}. Another plane, defined by the points P_4, P_5, P_6, P_7, is orthogonal to plane π_3. So the projection of the line through P_5 and P_7 onto the plane π_3 yields L_{Proj}, which intersects the sphere S_5 (center: P_5, radius: l_4), resulting in a point pair. Function `pp_get2nd` selects link 4 (P_4) from it.

$$\begin{aligned} S_5 &= \text{Sphere}(P_5, l_4)) \\ \pi_3 &= (P_{H2} \wedge P_1 \wedge P_5 \wedge e_\infty)^* \\ \pi_3 &= \frac{\pi_3}{|\pi_3|} \\ L_{P5P7} &= (P_5 \wedge P_7 \wedge e_\infty)^* \\ L_{Proj} &= \frac{(\pi_3 \cdot L_{P5P7})}{\pi_3} \\ P_4 &= \text{pp\_get2nd}((S_5 \wedge L_{Proj})^*) \end{aligned} \tag{5}$$

$\text{Sphere}(x,r)$ generates a sphere around x with the radius r.

$$\text{Sphere}(x,r) = x - \frac{1}{2}\, r^2\, e_\infty \tag{6}$$

The functions pp_get1st and pp_get2nd each pick one point out of a point pair.

$$\text{pp\_get1st}(x) = \frac{\sqrt{|x \cdot x|} - x}{e_\infty \cdot x} \tag{7}$$

$$\text{pp\_get2nd}(x) = \frac{-\sqrt{|x \cdot x|} + x}{e_\infty \cdot x} \tag{8}$$

6.1.3 Computation of the position of link 2

Link 1 and 2 are located on the yz-plane π_1 respectively, so the intersection of planes π_1 and π_3 results in a line, with P_1 and P_2 on it. The distance between P_2 and P_1 is l_1, hence the intersection of the sphere S_1 around P_1 with radius l_1 results in a point pair, from which P_2 can be selected.

$$\begin{aligned} \pi_1 &= e_1 \\ S_1 &= \text{Sphere}(P_1, l_1) \\ L_1 &= \pi_1 \wedge \pi_3 \\ P_2 &= \text{pp\_get1st}((L_1 \wedge S_1)^*) \end{aligned} \tag{9}$$

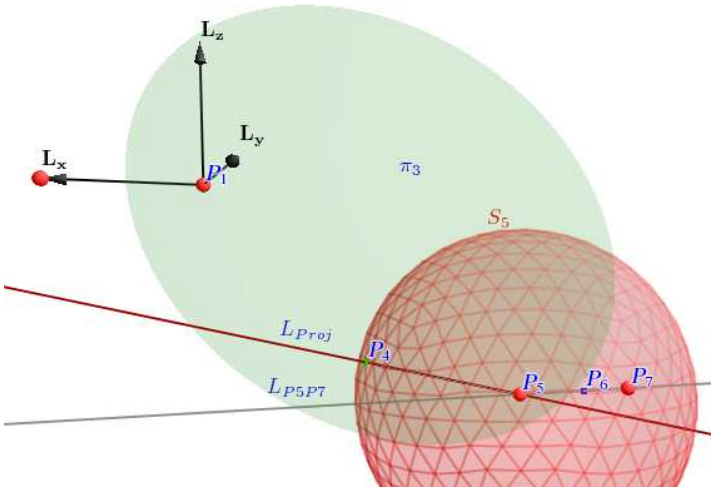

Fig. 5 Step B. - Projection of the line through P_7 and P_5 onto the green plane defined by P_1, P_5 and an auxiliary point on the x-axis. Intersection of the Sphere around P_5 with radius l_4 and the projected line returns P_4

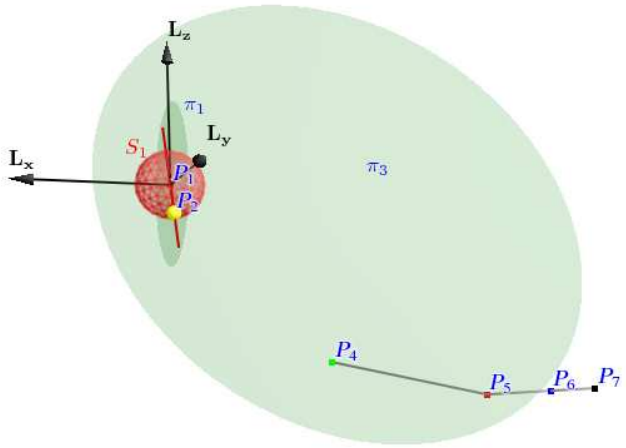

Fig. 6 Step C. - Intersecting the sphere around P_1 with radius l_1 with the intersection of the plane π_3 and the yz-plane returns P_2

6.1.4 Computation of the position of link 3

The intersection of the two spheres S_2 and S_4 results in a circle Z_3. P_3 must be located on circle Z_3 and on plane π_3 as well, thus the intersection of Z_3 and π_3 results in a point pair again, from which P_3 can be selected.

$$
\begin{aligned}
S_2 &= \text{Sphere}(P_2, l_2) \\
S_4 &= \text{Sphere}(P_4, l_3) \\
Z_3 &= S_2 \wedge S_4 \\
P_3 &= \text{pp\_get1st}((Z_3 \wedge \pi_3)^*)
\end{aligned} \tag{10}
$$

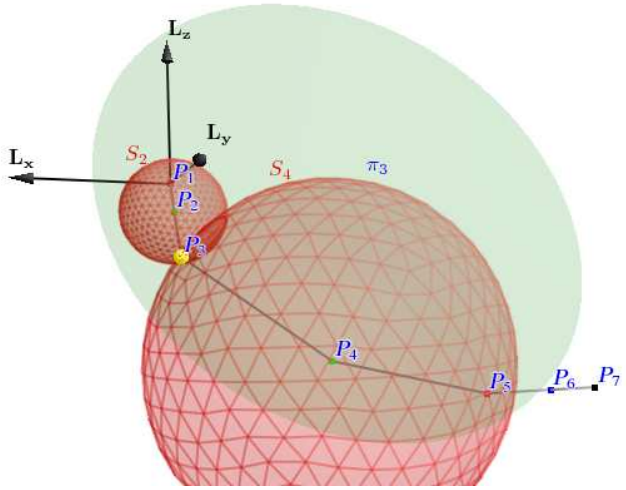

Fig. 7 Step D. - The intersection of the spheres around P_2 with radius l_2 and around P_4 with radius l_3 results in the red circle. Intersecting the circle with the plane π_3 returns P_3

6.1.5 Compute the angles of the links

Since a line is defined by 2 points, 3 points are necessary to generate 2 intersecting lines and to compute the angle in between. To compute the angle of the first link, a point above the origin (0,0,1) is used as a parameter.

$$
\text{angle}(x,y,z) = \pi - \arccos\left(\frac{(x \wedge y \wedge e_\infty) \cdot (z \wedge y \wedge e_\infty)}{|x \wedge y \wedge e_\infty| \; |z \wedge y \wedge e_\infty|}\right) \tag{11}
$$

6.2 Symbolic optimization of the kinematic chain

In this section, we present the CLUCalc code for the just described inverse kinematics algorithm as an example for the input language of Gaalop. Parts of the generated C code are presented as an example for a target implementation of Gaalop.
CluCalc models the first part of the inverse kinematics algorithms as follows:

```
PH3 = VecN3(hx-1,hy,hz);
:P1 = VecN3(hx,hy,hz);
:P7 = VecN3(-px,py,pz);

norm1 = FNorm[1]*e1 - FNorm[2]*e2 - FNorm[3]*e3;
?norm2 = 1/abs(norm1);
norm3 = norm1 * norm2;

//generate translator and compute P6
tvec = (norm3*(len[6]))/2;
T1 = 1 -  tvec * einf;
?P6 = T1 * P7 * ~T1;

tvec = (norm3*(len[5]+len[6]))/2;
T2 = 1 -  tvec * einf;
?P5 = T2 * P7 * ~T2;

S5 = Sphere(P5,len[4]);
Pi3 = *(PH3 ^ e0 ^ P5 ^ einf);
?Pi3a = 1/abs(Pi3);
?Pi3b = Pi3 * Pi3a; // Pi3/abs(Pi3)
```

Gaalop optimizes the above CLUCalc code (see section 4.1 for details about the optimization approach) and generates the following C code:

```
float norm2_opt[32] = {0.0};
norm2_opt[1]=1/sqrt(FNorm[1]*FNorm[1]+FNorm[2]*FNorm[2]
                    +FNorm[3]*FNorm[3]);

float P6_opt[32] = {0.0};
P6_opt[2]=-px+FNorm[1]*norm2_opt[1]*len[6];
P6_opt[3]=py-FNorm[2]*norm2_opt[1]*len[6];
P6_opt[4]=pz-FNorm[3]*norm2_opt[1]*len[6];
P6_opt[5]=0.5*FNorm[1]*FNorm[1]*norm2_opt[1]*norm2_opt[1]
          *len[6]*len[6]+0.5*py*py+0.5*pz*pz-FNorm[1]
          *norm2_opt[1]*len[6]*px+0.5*FNorm[3]*FNorm[3]
          *norm2_opt[1]*norm2_opt[1]*len[6]*len[6]+0.5
          *px*px+0.5*FNorm[2]*FNorm[2]*norm2_opt[1]
          *norm2_opt[1]*len[6]*len[6]-FNorm[2]*norm2_opt[1]
          *len[6]*py-FNorm[3]*norm2_opt[1]*len[6]*pz;
P6_opt[6]=1;


float P5_opt[32] = {0.0};
P5_opt[2]=-px+FNorm[1]*norm2_opt[1]*len[5]
          +FNorm[1]*norm2_opt[1]*len[6];
P5_opt[3]=-FNorm[2]*norm2_opt[1]*len[5]
```

```
            -FNorm[2]*norm2_opt[1]*len[6]+py;
    P5_opt[4]=-FNorm[3]*norm2_opt[1]*len[5]
            -FNorm[3]*norm2_opt[1]*len[6]+pz;
    P5_opt[5]=0.5*FNorm[2]*FNorm[2]*norm2_opt[1]*norm2_opt[1]
            *len[6]*len[6]+0.5*FNorm[3]*FNorm[3]*norm2_opt[1]
            *norm2_opt[1]*len[6]*len[6]+0.5*FNorm[1]*FNorm[1]
            *norm2_opt[1]*norm2_opt[1]*len[6]*len[6]-FNorm[3]
            *norm2_opt[1]*pz*len[5]-FNorm[1]*norm2_opt[1]*px
            *len[5]-FNorm[2]*norm2_opt[1]*py*len[5]+0.5*py*py
            +0.5*pz*pz+FNorm[2]*FNorm[2]*norm2_opt[1]
            *norm2_opt[1]*len[5]*len[6]+FNorm[1]*FNorm[1]
            *norm2_opt[1]*norm2_opt[1]*len[5]*len[6]+FNorm[3]
            *FNorm[3]*norm2_opt[1]*norm2_opt[1]*len[5]*len[6]
            +0.5*px*px-FNorm[1]*norm2_opt[1]*len[6]*px
            -FNorm[3]*norm2_opt[1]*len[6]*pz-FNorm[2]
            *norm2_opt[1]*len[6]*py+0.5*FNorm[1]*FNorm[1]
            *norm2_opt[1]*norm2_opt[1]*len[5]*len[5]
            +0.5*FNorm[2]*FNorm[2]*norm2_opt[1]*norm2_opt[1]
            *len[5]*len[5]+0.5*FNorm3*FNorm3*norm2_opt[1]
            *norm2_opt[1]*len[5]*len[5];
    P5_opt[6]=1;

    float Pi3a_opt[32] = {0.0};
    Pi3a_opt[1]=1/sqrt(hy*hy*P5_opt[4]*P5_opt[4]-2*hy*P5_opt[4]
                *hz*P5_opt[3]+hz*hz*P5_opt[3]*P5_opt[3]
                +P5_opt[4]*P5_opt[4]*hx*hx-2*P5_opt[4]*hx*hz
                *P5_opt[2]-2*P5_opt[4]*P5_opt[4]*hx+hz*hz
                *P5_opt[2]*P5_opt[2]+2*hz*P5_opt[2]*P5_opt[4]
                +P5_opt[4]*P5_opt[4]+P5_opt[3]*P5_opt[3]*hx*hx
                -2*P5_opt[3]*hx*hy*P5_opt[2]-2*P5_opt[3]
                *P5_opt[3]*hx+hy*hy*P5_opt[2]*P5_opt[2]+2*hy
                *P5_opt[2]*P5_opt[3]+P5_opt[3]*P5_opt[3]);


    float Pi3b_opt[32] = {0.0};
    Pi3b_opt[2]=-Pi3a_opt[1]*(-P5_opt[4]*hy+hz*P5_opt[3]);
    Pi3b_opt[3]=Pi3a_opt[1]*(-P5_opt[4]*hx+hz*P5_opt[2]
                +P5_opt[4]);
    Pi3b_opt[4]=-Pi3a_opt[1]*(-P5_opt[3]*hx+hy*P5_opt[2]
                +P5_opt[3]);
```

As you can see the optimized code is very complex in terms of length. Therefore we only list the CLUCalc code for the second part of the algorithm below.

```
LP5P7 = *(P5 ^ P7 ^ einf);
LProj = (Pi3b.LP5P7)/Pi3b;
:P4 = pick2nd( *(S5 ^ LProj) );

Pi1 = e1;
S1 = Sphere(P1,len(1));
L1 = Pi1 ^ Pi3:Red;
:P2 = pick1st(*(L1 ^ S1));

S2 = Sphere(P2,len(2));
```

```
S4 = Sphere(P4,len(3));
C3 = S2 ^ S4;
?P3 = pick1st(*(C3 ^ Pi3));

?angle(VecN3(0,1,0),P1,P2);
?angle(P1,P2,P3);
?angle(P2,P3,P4);
?angle(P3,P4,P5);
?angle(P4,P5,P6);
```

From the runtime performance point-of-view, our optimized C code achieved results comparable to the conventional algorithm. This is why the actual speedup that Gaalop can provide for this inverse kinematics application will result from future implementations on parallel platforms.

Because of the similarity of this inverse kinematics algorithms to our proof-of-concept application (see section 2.3 and [9]) we expect for a FPGA implementation a comparable hardware speedup of about 100 times.

7 Current Status and Future Perspectives

Gaalop is currently able to handle sequential conformal geometric algebra algorithms. The algorithm is currently transformed into C code as well as CLUCalc code and simple LaTeXformulas. Please find always the latest news on the Gaalop homepage [11].

We are just extending Gaalop in order to handle control flow with loops, conditions etc. We are also developing generators for additional output formats like Java code, CUDA [15] and FPGA descriptions.

One focus will lie on on mixed solutions handling reasonable combinations of software and hardware implementations.

8 Conclusion

Geometric algebra is applicable in many different engineering scenarios and provides a straightforward and intuitive problem solving approach. With the help of our Gaalop tool these algorithms can be automatically transformed into high runtime performance implementations. With these results, we are convinced that conformal geometric algebra will be able to become more and more fruitful in a great variety of engineering applications.

References

1. R. Ablamowicz and B. Fauser. The homepage of the package Cliffordlib. HTML document http://math.tntech.edu/rafal/cliff9/, 2005. Last revised: September 17, 2005.
2. L. Dorst, D. Fontijne, and S. Mann. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry*. Morgan Kaufman, 2007.
3. The RoboCup Federation. Robocup official site. http://www.robocup.org.
4. Daniel Fontijne. *Efficient Implementation of Geometric Algebra*. PhD thesis, University of Amsterdam, 2007.
5. S. Franchini, A. Gentile, M. Grimaudo, C.A. Hung, S. Impastato, F. Sorbello, G. Vassallo, and S. Vitabile. A sliced coprocessor for native Clifford algebra operations. In *Euromico Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2007.
6. A. Gentile, S. Segreto, F. Sorbello, G. Vassallo, S. Vitabile, and V. Vullo. Cliffosor, an innovative FPGA-based architecture for geometric algebra. In *ERSA 2005*, pages 211–217, 2005.
7. D. Hildenbrand. Geometric computing in computer graphics using conformal geometric algebra. *Computers & Graphics*, 29(5):802–810, 2005.
8. D. Hildenbrand, D. Fontijne, C. Perwass, and L. Dorst. Tutorial geometric algebra and its application to computer graphics. In *Eurographics conference Grenoble*, 2004.
9. D. Hildenbrand, D. Fontijne, Yusheng Wang, M. Alexa, and L. Dorst. Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. In *Eurographics conference Vienna*, 2006.
10. D. Hildenbrand, H. Lange, Florian Stock, and Andreas Koch. Efficient inverse kinematics algorithm based on conformal geometric algebra using reconfigurable hardware. In *GRAPP conference Madeira*, 2008.
11. D. Hildenbrand and Joachim Pitt. The Gaalop home page. HTML document http://www.gaalop.de, 2008.
12. Nico Kasprzyk and Andreas Koch. High-level-language compilation for reconfigurable computers. In *Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC)*, 2005.
13. Biswajit Mishra and Peter Wilson. Color edge detection hardware based on geometric algebra. In *European Conference on Visual Media Production (CVMP)*, 2006.
14. Biswajit Mishra and Peter R. Wilson. VLSI implementation of a geometric algebra parallel processing core. Technical report, Electronic Systems Design Group, University of Southampton, UK, 2006.
15. NVIDIA. The CUDA home page. HTML document http://www.nvidia.com/object/cuda_home.html, 2009.
16. C. Perwass. The CLU home page. HTML document http://www.clucalc.info, 2005.
17. C. Perwass, C. Gebken, and G. Sommer. Implementation of a Clifford algebra co-processor design on a field programmable gate array. In R. Ablamowicz, editor, *CLIFFORD ALGEBRAS: Application to Mathematics, Physics, and Engineering*, Progress in Mathematical Physics, pages 561–575. 6th Int. Conf. on Clifford Algebras and Applications, Cookeville, TN, Birkhäuser, Boston, 2003.