Gaalop - High Performance Computing based on Conformal Geometric Algebra

Dietmar Hildenbrand, Andreas Koch

ABSTRACT We present Gaalop (Geometric algebra algorithms optimizer), our tool for high performance computing based on Conformal Geometric Algebra (GA). The main goal of Gaalop is to realize implementations that are most likely faster than conventional solutions. We describe the concepts, the state-of-the-art as well as the future perspectives of Gaalop dealing with optimized software implementations, hardware implementations as well as mixed solutions.

Keywords: Clifford algebras, runtime performance.

1 Introduction

In recent years, Geometric Algebra, and especially the 5D Conformal Geometric Algebra, 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 paper, we present our approach for the automatic generation of high performance implementations. In the chapters 2 and 3, we present some related work as well as the basics of Conformal Geometric Algebra.

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 chapter 4. The corresponding architecture of Gaalop is described in chapter 5. Its current state-of-the-art as well as its future perspectives can be found in the chapter 6.

AMS Subject Classification: 15A66, 17B37, 20C30, 81R25.

2 Related Work

Despite the tremendous expressive power of the GA, it has only seen limited use in practical applications. One of the reasons for this might be that the actual processing of GA algorithms requires significant computational effort. Related tools with the intention of optimizing GA 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 [3]). You can find some benchmarks comparing Gaigen with other software implementations in [3].

2.2 Hardware Implementations

To resolve the above mentioned quandary, it is promising to look at dedicated hardware architectures for the acceleration of GA algorithms. Current integrated circuit technology offers a means to achieve this in the form of field-programmable gate arrays (FPGAs). These so-called 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.

Prior Attempts

The first serious approach is described in [15]. 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 clockcycles 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 in [5]: 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 in [4]: 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 GA. 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 GA 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 in [12], 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 GA 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 [13] shows a real wall-clock *speed-up* of 3x over a software programmable processor. However, the authors do not state which processor they used as a reference.

2.3 Our Proof-of-Concept Approach

In [8] we could show that an approach with symbolic simplification of GA algorithms is able to lead to an implementation which is three times faster than a conventional solution. In a second stage we implemented this algorithm also on hardware and got an additional speedup of more than 100 times (see [9]). Please find some detailed information about our hardware approach as follows:

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 [9] for a specific

GA 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 GA operators, we decomposed the GA 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, 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 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 pipeline 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 GA.

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 1.1 lists the two representations of the geometric entities in Conformal Geometric Algebra. In this table $\mathbf{x}$ and $\mathbf{n}$ 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.

$$\mathbf{x} = x_1 e_1 + x_2 e_2 + x_3 e_3 \tag{3.1}$$

TABLE 1.1. 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	$\pi = \mathbf{n} + de_\infty$	$\pi^* = x_1 \wedge x_2 \wedge x_3 \wedge e_\infty$
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 additional two base vectors are indicated by

- $\mathbf{e}_0$ representing the 3D origin
- $\mathbf{e}_\infty$ representing the point at infinity

The $\{s_i\}$ represent different spheres and the $\{\pi_i\}$ different planes. 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 the geometric algebra. 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). There exists only one element of grade five in the Conformal Geometric Algebra. It is therefore also 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 1.2 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 $P = x_1 e_1 + x_2 e_2 + x_3 e_3 + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$ (see table 1.1 and equation (3.1)) for instance can be written in terms of a multivector as the following linear combination of blades

$$P = x_1 * blade[2] + x_2 * blade[3] + x_3 * blade[4] + \frac{1}{2}\mathbf{x}^2 * blade[5] + blade[6] \tag{3.2}$$

TABLE 1.2. The 32 blades of the 5D Conformal Geometric Algebra

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

For more details please refer for instance to the book [2] as well as to the tutorials [7] and [6].

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.

4.1 Symbolic Optimization

We use the symbolic computation functionality of Maple (together with a library for geometric algebras [1]) in order to optimize parts of a GA algorithm in a way as described in the following example:

The following code

```
px:= x1*e1 + x2*e2 + x3*e3;
P1 := conformal(px);

py:= y1*e1 + y2*e2 + y3*e3;
P2 := conformal(py);

S1 := P1 - 0.5*r1*r1*einf;
S2 := P2 - 0.5*r2*r2*einf;

C_e := S1 &w S2;

gaalop(C_e);
```

describes the intersection of the sphere $S1$ (center at $P1$ and radius $r1$) with another sphere $S2$ (center at $P2$ and radius $r2$). The resulting intersection circle C_e is optimized with the help of symbolic computations and simplifications by executing `gaalop(C_e);`.

The resulting C-code generated by Gaalop is as follows:

```
float C_e [32];

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

C_e[9] = -.5*y1*x1*x1-.5*y1*x2*x2-.5*y1*x3*x3+.5*y1*r1*r1
+.5*x1*y1*y1+.5*x1*y2*y2+.5*x1*y3*y3-.5*x1*r2*r2;

C_e[10] = -1.*y1+x1;
C_e[11] = -x3*y2+x2*y3;

C_e[12] =-.5*y2*x1*x1-.5*y2*x2*x2-.5*y2*x3*x3+.5*y2*r1*r1
+.5*x2*y1*y1+.5*x2*y2*y2+.5*x2*y3*y3-.5*x2*r2*r2;

C_e[13] = -1.*y2+x2;

C_e[14] = -.5*y3*x1*x1-.5*y3*x2*x2-.5*y3*x3*x3+.5*y3*r1*r1
+.5*x3*y1*y1+.5*x3*y2*y2+.5*x3*y3*y3-.5*x3*r2*r2;

C_e[15] = -1.*y3+x3;

C_e[16] = -.5*y3*y3+.5*x3*x3+.5*x2*x2+.5*r2*r2
-.5*y1*y1-.5*y2*y2+.5*x1*x1-.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 1.1) are used. As you can see, all the corresponding coefficients of this multivector are very simple expressions with basic arithmetic operations.

4.2 Use of Inherent Fine-Grained Parallel Structure

With the help of symbolic optimization the GA 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 detail in section 2.3 and in [9].

5 The Architecture of Gaalop

Figure 1 shows an overview over the architecture of Gaalop. Its input is a GA algorithm. Via symbolic simplification it is transformed into an generic intermediate representation (IR) that can be used for the generation of different output formats such as C-code, FPGA descriptions (as a structural hardware description, currently written in the Verilog language, CLUCalc-code in order to visualize the results (see [14]).

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 [11]. 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 GA 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.

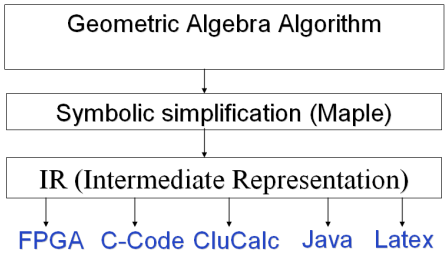

FIGURE 1. Architecture of Gaalop

6 State-of-the-art and Future Perspectives

Gaalop is currently able to handle sequential GA algorithms. It optimizes parts of the algorithm at locations where this is explicitly indicated. The algorithm is currently transformed into C-code as well as CLUCalc-code. Please find always the newest information on the Gaalop homepage ([10]).

In the future we will be able to handle not only sequential algorithms but also loops, conditions etc. We will generate additional output formats like Java and Latex.

One focus will lie on the automatic compilation of hardware descriptions as well as on mixed solutions handling reasonable combinations of software and hardware implementations.

From the runtime performance point-of-view, Gaalop is just able to automatically generate software implementations comparable to the software implementation of our proof-of-concept application (see section 2.3). In this application we are three times faster than the conventional algorithm (see [8]). In the future, we expect an additional speedup of more than 100 times based on the automatic generation of hardware implementations as described in [9].

7 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] Daniel Fontijne. *Efficient Implementation of Geometric Algebra.* PhD thesis, University of Amsterdam, 2007.

[4] 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.

[5] 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.

[6] D. Hildenbrand. Geometric computing in computer graphics using conformal geometric algebra. *Computers & Graphics*, 29(5):802–810, 2005.

[7] D. Hildenbrand, D. Fontijne, C. Perwass, and L. Dorst. Tutorial geometric algebra and its application to computer graphics. In *Eurographics conference Grenoble*, 2004.

[8] 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.

[9] 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.

[10] D. Hildenbrand and Joachim Pitt. The Gaalop home page. HTML document http://www.gaalop.de, 2008.

[11] Nico Kasprzyk and Andreas Koch. High-level-language compilation for reconfigurable computers. In *Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC)*, 2005.

[12] Biswajit Mishra and Peter Wilson. Color edge detection hardware based on geometric algebra. In *European Conference on Visual Media Production (CVMP)*, 2006.

[13] 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.

[14] C. Perwass. The CLU home page. HTML document http://www.clucalc.info, 2005.

[15] 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.

7.1 Information about the Authors

Dietmar Hildenbrand
Interactive Graphics Systems Group
University of Technology Darmstadt, Germany

E-mail: dhilden@gris.informatik.tu-darmstadt.de

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

Submitted: July 29, 2008; Revised: TBA.