FPGA-Accelerated Color Edge Detection Using a Geometric-Algebra-To-Verilog Compiler

Florian Stock
and Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
Darmstadt, Germany
email: stock|koch@esa.informatik.tu-darmstadt.de

Dietmar Hildenbrand
LOEWE Priority Program Cocoon
TU Darmstadt
Darmstadt, Germany
email: hildenbrand@cocoon.tu-darmstadt.de

Abstract— **Geometric Algebra (GA) is a branch of mathematics that generalizes complex numbers and quaternions. One of the advantages of the framework is, that it allows intuitive description and manipulation of geometric objects. While even complex operations can be described concisely, the actual evaluation of these GA expressions is extremely compute intensive. However, it has significant fine-grained parallelism, which makes it a profitable target for hardware implementation. In this paper, we present the automatic acceleration of a color edge-detection algorithm from a GA description. Using our Gaalop GA compiler with its Verilog back-end, we can show speed-ups of over 1000x even compared to a recent GA processor ASIC.**

I. Introduction

Geometric Algebra (GA) is a powerful mathematical framework that generalizes projective geometry, imaginary numbers, and quaternions. As complex geometric relationships and transformations can be intuitively expressed, GA allows very concise formulation of many engineering and scientific problems. In many cases, these descriptions require only a fraction of the space of conventional formulations.

The roots of GA go back to the work of Grassmann [1] and Clifford [2] from the 19th century. Similar to the development of other mathematical frameworks, its usefulness and wide practical applicability was not immediately recognized and was only later rediscovered.

Initially, GA became popular in physics to concisely express complex geometrical relationships [3], [4], [5]. Later, with the invention of *conformal* geometric algebra [6] by David Hestenes, the use of GA has also been extended to engineering application domains such as robotics, computer graphics, and computer vision.

Conformal geometric algebra not only allow the flexible modeling of geometric objects and transformations, but also describe *intuitively* applicable concepts such as points, lines, planes and spheres, as well as operations on them (e.g., intersection and rotations). However, the evaluation of the multi-dimensional GA expressions requires significant computational effort. This drawback has slowed its adoption as a practical algorithmic tool. While this situation has improved with faster processors, the core computations do not profit from many-core architectures due to the very fine-grained (operator-level) nature of their parallelism. On the other hand, they are highly amenable to acceleration by specialized architectures.

Today, this idea is often associated with computation on Graphics Processing Units (GPUs). This is not the best choice for GA calculations, however, since the expressions generally do not have much *SIMD* (Single Instruction Multiple Data) parallelism. For this reason, other approaches, such as processors with ISAs (Instruction Set Accelerator) specialized for GA, or the use of reconfigurable computing have been considered by the research community. As we will demonstrate, especially the use of the latter is highly effective, even compared to dedicated GA processor ASICs. But as usual, there exists a large gap between the highly abstract GA operations and the actual hardware architecture of a reconfigurable accelerator. To close this gap, we have developed a tool-flow capable of translating a Domain-Specific Language (DSL) for GA to highly optimized FPGA implementations. Here, the high abstraction level of GA data and operators actually works to our advantage, since attempts to translate less abstract languages such as C into hardware [7], [8], [9], [10] are often inefficient or fraught with multiple limitations (e.g., no dynamic pointers, only unrollable loops etc.).

The key contributions of this work over prior publications [11], [12] are optimizations in our compile flow and the examination of color edge-detection as an application for concise GA description. We will show performance improvements of over 1000x over a recent specialized GA processor ASIC for the same algorithm.

II. Related Work

To actually make use of GA as a tools to solve practical problems, appropriate software tools are required. Roughly half a dozen of specialized GA tools are currently in wider use.

They include libraries for existing computer algebra systems (e.g., CLIFFORD [13] for Maple, MSTA for Mathematica, or GABLE [14] for Matlab), as well as for high-level programming languages (e.g., Gaalet [15], a C++ template library).

Other software tools directly operate on DSLs specialized for GA. A common one is CLUCalc [16], which presents an interactive environment for developing GA algorithms in the CLUScript DSL.

While allowing highly productive algorithm development with its powerful visualization features, the fact that CLUCalc relies on an interpreter for execution limits performance.

Higher performance can be achieved by generating native code from the GA descriptions. Gaigen [17] compiles GA algorithms expressed in XML to a number of high-level languages such as C/C++ and Java. Our own Gaalop 2.0 [18] system supports a number of front-ends, including CLUScript as well as an embedded DSL for GA operations inserted in C++ source code. Its multiple compiled-code back-ends cover C++, OpenCL, and CUDA to support a number of software-programmable target processors.

Due to the inefficiency of even compiled-code algorithms executing on CPUs and GPUs (as sketched in Section I), significant research effort has been expended on GA-optimized hardware architectures. Initial attempts include Cliffosor [19] and its successor S-Cliffosor ([20]. However, these systems suffered from a number of architectural bottlenecks or only supporting limited GA operations (e.g., low dimensionality, restricted operator subset).

A more powerful architecture that supports higher-dimensional algebras is described by Mishra and Wilson in [21]. They realized a processor with an ISA supporting a full set of GA operations, which are then executed on a Geometrics Algebra Micro Architecture (GAMA) unit. The ASIC implementation in an ST 120 nm process has a maximum clock frequency of 130 MHz (though the authors only clock it at 125 MHz for their benchmarks).

All of these previous hardware solutions had hardwired, internally parallel operators for the GA primitives, but implemented the actual GA algorithm as a sequential instruction sequence of these primitives.

The approach we use in Gaalop 2.0 [18] is different: For each specific GA input program, our hardware back-end synthesizes a custom micro-architecture that embodies the entire computation. The architecture has a finely parallel, pure dataflow structure, completely avoiding the sequentiality of a serial instruction stream.

Our initial experiments (described in [11], [12]) concentrated on inverse kinematics computations for high-frame rate computer animation. In this work, we will address the same problem tackled by Mishra and Wilson, namely a rotor-based edge detector following the scheme proposed by Bayro-Corrochano and Flores [22]. This will allow a comparison between the ISA- and direct hardware synthesis-based acceleration approaches for GA algorithms.

III. Geometric Algebra

A. Basics

For space reasons, this section can present just a very basic introduction to GA. A more comprehensive description is given, e.g., in [23].

Primitive GA operations are performed on elements called *multivectors*. These multivectors are linear combinations of a limited number of so-called *blades*. The number of blades depends on the dimensionality of the GA: An n-dimensional GA has n basis vectors (usually called $e_1, \dots e_n$), with the outer products across multiple basis vectors being the the blades. The number of basis vectors in each blade are defined as the blade's *grade*. Including the extreme cases (no basis vectors, grade=0, and all basis vectors, grade=n, entering into the outer product), 2^n possible combinations of basis vectors are possible. Table I shows the blades of a 3-dimensional GA. We will limit our discussion to such a 3-D GA, since that dimensionality is sufficient for the GA-based edge-detection algorithm. For comparison, in the model of Hamilton quaternions, the equivalent of 2-blades would be the commonly used basis vectors i, j, and k.

TABLE I: The 8 blades of a 3-dimensional GA.

index	**blade**	**grade**	**name**
1	1	0	scalar, 0-blade
2	e_1	1	vector, 1-blade
3	e_2	1	vector, 1-blade
4	e_3	1	vector, 1-blade
5	$e_1 \wedge e_2$	2	bivector, 2-blade
6	$e_1 \wedge e_3$	2	bivector, 2-blade
7	$e_2 \wedge e_3$	2	bivector, 2-blade
8	$e_1 \wedge e_2 \wedge e_3$	3	trivector (I), pseudoscalar, 3-blade

GA relies on the addition of multivectors and various kinds of multiplications as primitive operations. From a purely mathematical perspective, not all of the multiplication operations need to be defined separately, since some can be expressed in terms of the others, but more concise and intuitive forms of algorithmic descriptions are often enabled by explicitly providing all of them. Thus, the three multiplications operations on two multivectors a and b are the inner, outer, and geometric products:

Inner Product:
: For 3-D Euclidean space, the inner product of two vectors $a \cdot b$ is the same as the Euclidean scalar product of two vectors (the actual GA definition is more general). This implies, e.g., that for perpendicular vectors, the inner product will be 0, a relation that also applies in higher dimensional algebras.

Outer Product:
: For parallel vectors, the outer product $a \wedge b$ is always 0, making it very useful to express parallelity relations even in higher dimensionality.

Geometric Product:
: The geometric product ab is GA-specific and defined for vectors as $ab := a \cdot b + a \wedge b$. In GA, it is a powerful tool for expressing transformations. Subsection III-B describes its use in the edge detection algorithm.The relationship

$$e_i e_j = \begin{cases} -e_j e_i & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \tag{1}$$

holds between all of our basis vectors.

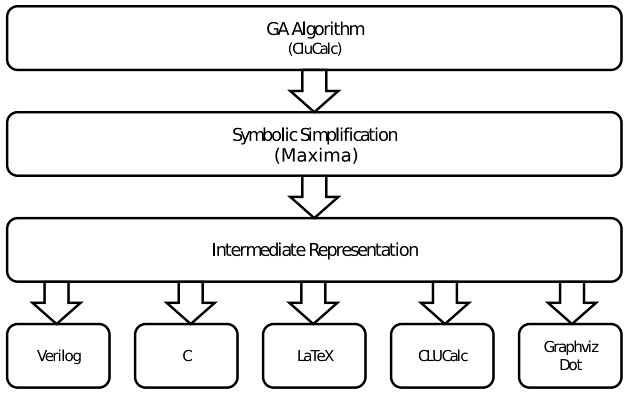

Fig. 1: Gaalop 2.0 compile flow.

B. Rotation

A rotor R is defined as the operator

$$R = e^{\frac{\phi}{2}L} = \cos(\frac{\phi}{2}) - L\sin(\frac{\phi}{2}) \tag{2}$$

Geometrically, it describes a rotation around axis L (represented by a normalized bivector), with the rotation angle given by ϕ.

The rotation of a geometric object o is performed by the applying the rotor R as

$$o_{rot} = Ro\tilde{R} \tag{3}$$

where $\tilde{R}$ is the conjugate of R.

IV. Edge Detection Algorithm in 3-D GA

Similar to the operation of the human eye, a holistic edge-detection algorithm does not process separate RGB planes, but considers the entire image as a single entity. This multi-dimensionality is easily described in GA using multivectors.

Color information can be expressed as bivectors. Let $r_{m,n}$,$g_{m,n}$, and $b_{m,n}$ be the separate RGB color channels for an image of the dimension $M \times N$ at row m and column n. We can then define a *single* multivector encompassing all image color information $c_{m,n}$ as:

$$c_{m,n} = r_{m,n}e_2e_3 + g_{m,n}e_3e_1 + b_{m,n}e_1e_2 \tag{4}$$

Bayro-Corrochano and Flores [22] developed their GA edge detection algorithm as a convolution, using two masks m_L and m_R (for left and right, given below). The convolution is applied as a rotor, formulated by the geometric product (for masks of the size $2X+1$ and $2Y+1$):

$$\hat{c}(m,n) = \sum_{x=-X}^{X}\sum_{y=-Y}^{Y} m_L(x,y)\cdot \\ c_{(m-x \mod M),(n-y \mod N)}m_R(x,y) \tag{5}$$

For color edge detection, the two masks used are described by

$$m_L = \begin{bmatrix} R & R & R \\ 0 & 0 & 0 \\ \tilde{R} & \tilde{R} & \tilde{R} \end{bmatrix} \tag{6}$$

$$m_R = \begin{bmatrix} \tilde{R} & \tilde{R} & \tilde{R} \\ 0 & 0 & 0 \\ R & R & R \end{bmatrix}, \tag{7}$$

with the individual rotors being

$$R = se^{\frac{L\pi}{4}} = s(\cos(\frac{\pi}{4}) + L\sin(\frac{\pi}{4})) = s(\frac{\sqrt{2}}{2} + L\frac{\sqrt{2}}{2}). \tag{8}$$

Here, L is the unit vector given by

$$L = \frac{(e_2e_3 + e_3e_1 + e_1e_2)}{\sqrt{3}} \tag{9}$$

and s is the scale factor $s = \frac{1}{\sqrt{6}}$. For brevity, we give only the masks m_L and m_R for the convolution in the horizontal direction. The vertical computation proceeds analogously, but uses instead of the masks m_L and m_R their transposed.

Applying these masks to the color pixels described by c yields this expression:

$$\begin{aligned}\hat{c}(m,n) =& \tilde{R}(c_{m-1,n-1} + c_{m+1,n}c_{m-1,n+1})R \\ &+ R(c_{m+1,n-1} + c_{m+1,n} + c_{m+1,n+1})\tilde{R} \\ =& \tilde{R}c_uR + Rc_l\tilde{R}\end{aligned} \tag{10}$$

, where c_u and c_l are the upper and lower row of the convolution mask.

Equation 10 concisely describes the *entire* holistic edge detection algorithm. It operates as follows: A color vector c is split into two components, $\vec{c}_\perp$ and $\vec{c}_\|$. These two components represent the vector components perpendicular and parallel to the rotation axis. Intuitively, in a 3-D color cube, the unit vector (which is used as the rotation axis L) describes the gray colors.

If the colors in the upper and lower row are homogeneous (do not contain an edge), the rotation of $Rc_\perp\tilde{R}$ would rotate the color vector by the same amount as the rotation $\tilde{R}c_\perp R$, with the rotations canceling out. The pixel thus computed would fall on the gray axis in the result image (indicating that no edge is present).

On the other hand, if the two colors in the upper and lower row are not homogeneous (do contain an edge), the rotations will *not* cancel each other out. Thus, the result will lie somewhere else in the color cube, off the gray axis, indicating the presence of an edge.

This kind of convolution belongs to a class of linear vector filters, and could be also applied to signals different from images, e.g., speech signals.

V. Gaalop Compiler Architecture

Gaalop, the Geometric Algebra Algorithms Optimizer, is our plugin-based source-to-source GA compiler framework. It accepts the CLUCalc-script DSL [16], and supports a number of different output formats (see Fig. 1). Compared to the initial version of the Verilog back-end presented in [11], [12], the current GA optimization engine was completely replaced. The rewritten engine has the following changes:

First, the dependency on the Cliffordlib library was removed. In the first version the library, which was running on top of

```
// r,g,b are arrays with indices for
// the conv mask as follows
// 1,2,3 = upper left, middle, right
// 4,5,6 = left, self/center, right
// 7,8,9 = lower left, middle, right

// converting rgb into bivector
c1 = r1*e2^e3 + g1*e3^e1 + b1*e1^e2;
c2 = r2*e2^e3 + g2*e3^e1 + b2*e1^e2;
c3 = r3*e2^e3 + g3*e3^e1 + b3*e1^e2;
c7 = r7*e2^e3 + g7*e3^e1 + b7*e1^e2;
c8 = r8*e2^e3 + g8*e3^e1 + b8*e1^e2;
c9 = r9*e2^e3 + g9*e3^e1 + b9*e1^e2;

s = 1/sqrt(6);
n = (e2 * e3 + e3*e1 + e1*e2)/sqrt(3);
R = s * ((sqrt(2)/2) + n * (sqrt(2)/2));
?p = ~R*(c1+c2+c3)*R + R*(c7+c8+c9)*~R;
```

Fig. 2: CluCalc code for the GA-based edge detection.

the commercial Maple Computer Algebra System (CAS), took the GA expression, optimized them, and transformed them into scalar operations. The new version uses a table driven approach to map the GA operations itself to scalar operations. Further optimization of the expression be optionally be done using the open-source Maxima [24] CAS. This not only removes the dependency from an commercial tool, but also removes a limitation imposed by Cliffordlib: The new engine can not only handle conformal 3D algebras, but also higher dimensional algebras.

Second, the new hardware-synthesis back-end can now flexibly handle multiple floating- and fixed-point numerical representations. The latter can be determined automatically using developer-provided precision and value range constraints on input and result values, which are then bi-directionally propagated across the dataflow graph (DFG) using Monte Carlo analysis. Hereby random data from input range is send through the DFG, to analyze the impact on the value ranges of the inputs and outputs at the operators within the graph.

Internally, Gaalop first parses the DSL, then transforms it to a DFG-based intermediate form. This graph is then subjected to multiple optimizations, which include the fixed-point transformation as well as algebraic simplifications and reductions of the scalar computations underlying the GA primitives. More details about some of these steps are given in [12].

The optimized DFG is then mapped to a datapath and fitted with a simple controller that provides ASAP scheduling. For debugging and testing, the system also generates an automated testbench for the hardware accelerator, testing user supplied input and result values. The generated hardware is fully spatial and pipelined, both of which result in a very high degree of fine-grained parallelism.

VI. Experimental Results

The entire edge detection was abstractly formulated in CluCalc, requiring very few lines of code (Listing 2). For

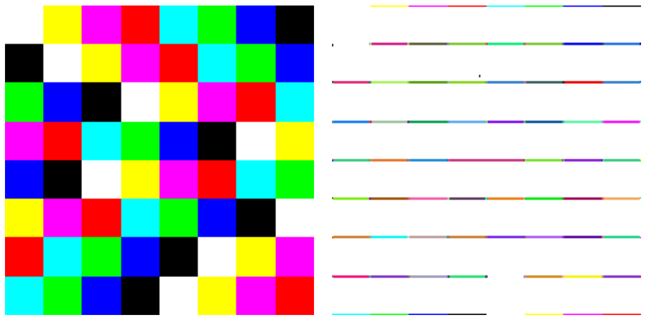

Fig. 3: Example pictures with color blocks before (a) and after (b) edge detection transformation (non-edge gray pixels removed).

Implementation	Clock [MHz]	Runtime [ms] for image size 128 × 128	256 × 256	512 × 512
ASIC [21]	125	46	184	735
Fixed Point	125	0.131	0.524	2.097
Float, Single	125	0.132	0.525	2.098
Float, Double	125	0.132	0.525	2.098
Float, Single	*400*	0.041	0.165	0.656
C Back-End on i7	3200	0.472	5.857	7.764

TABLE II: Comparison with previous results.

comparison with [21], we followed Mishra and Wilson's lead and implemented just the horizontal pass. We also used the same pictures as input (Lena and color blocks) with sizes of 128 × 128, 256 × 256, and 512 × 512. Fig. 3a shows an input picture before the color edge detection, Fig. 3b shows the result after applying the convolution. For clarity in monochrome printing, we have manually erased all non-edge pixels, which would normally show up as gray points in the result image (see Section IV for the underlying explanation).

The generated Verilog HDL code was synthesized, placed, and routed with Xilinx Vivado 2012.3, targeting a Xilinx Virtex 7 XC7VX690T-2, power values are estimated post-route with Vivado Power Analysis. Table IV shows the required FPGA resources when directing Gaalop to use floating-point (single/double precision) and automatically optimized fixed-point computations. For a fair comparison with Mishra and Wilson's work, and disregarding any performance gains due to chip-fabrication technology-induced, the clock speed increases. So we present an initial set of performance measurements the FPGA also executes just at the 125 MHz used in their GA ASIC. Furthermore, we disregard communication costs between the CPU and the accelerator (as was also done by Mishra and Wilson). For the FPGA, this is actually justified: We assume an adaptive computing system architecture combining a software-programmable main processor with the reconfigurable computing unit using shared (virtual) memory, supporting high bandwidth and low signaling latency. We have already demonstrated the practical feasibility of such a machine in [25].

Note that, due to the highly pipelined nature of the generated

Precision	LUTs		FFs		DSPs		BRAMs		Power [mW]
Fixed-Point, 32b	3618	0.83 %	6602	0.76 %	132	3.66 %	0	0 %	574
Float, Single	66529	15.35 %	79211	9.14 %	88	2.44 %	0	0 %	1656
Float, Double	152268	35.15 %	176210	20.34 %	480	13.33 %	0	0 %	3371

TABLE IV: Used resources and utilization on an Virtex 7 XC7VX690T-2 for different number formats (Total / % of Device).

Implementation	Op.	Image size		
		128×128	256×256	512×512
ASIC [21]	$+$	901120	3604480	14417920
	$*$	917504	3670016	14680064
Verilog Back-End	$+$	1507328	6029312	24117248
	$*$	786432	3145728	12582912
C Back-End	$+$	884736	3538944	14155776
	$*$	884736	3538944	14155776

TABLE III: Comparison with previous results. The table gives the number of additions and multiplications for different image sizes.

datapath (between 12 and 96 stages for the fixed- and floating-point designs), the throughput for all of the generated numerical representations is identical (one result pixel per clock cycle), with the latency differences due to varying pipeline lengths being almost negligible relative to the total number of pixels to process. At the same clock frequency, we achieve a performance improvement of roughly 350x across all of the image sizes relative to Mishra and Wilson's GA processor ASIC. When running the FPGA at 400 MHz, the highest clock speed achievable for the synthesized designs on the target FPGA, the performance gain increases to over 1120x.

One can argue, that Mishra and Wilson's ASIC accelerator is more versatile - which is correct to a certain degree: When algorithms change, the GA ASIC only needs rapid reprogramming instead of full hardware synthesis, place, and route. But GA-based applications will generally run a fixed set of GA compute kernels, which can easily be mapped to the FPGA in advance. During execution, the hardware kernels can be rapidly loaded onto the FPGA using techniques such as dynamic partial reconfiguration, which requires time only on the order of milliseconds for smaller designs (such as our fixed-point implementation).

On the other hand, a commonly used measure for quantifying the performance of GA systems (both hard- and software), namely the number of Geometric Algebra Operations Per Second (GOPS), is not applicable to our approach: Since we deconstruct the GA primitives down to their scalar components and optimize (e.g., merge, transform, move, eliminate, etc.) those, we can no longer determine the number of individual GA operators actually present in the generated hardware datapath.

Finally, for completeness, we compare the performance of our automatically compiled edge-detector accelerator with an optimized C implementation, also generated by Gaalop from the same input program. Since the C back-end also takes full advantage of the high-level transformations enabled by using a CAS in the compile flow, as well as the deconstruction and optimization of GA operations down to the scalar level, this is a fair comparison. When executing the gcc-compiled C output with the options -ftree-vectorizer, -fwhole-program, -O3, and executing it on a recent Intel Core i7 i7-3930K CPU clocked at 3.20 GHz, the performance lead of the FPGA solution drops, but is still greater than 10x. Note that for embedded applications, the advantage of the FPGA-based solution would be even greater due to the significantly better energy efficiency compared to the CPU (which has a maximum power draw of 130W).

VII. Conclusion

After introducing the fundamentals of GA and sketching the color edge-detection algorithm using rotors, we gave on overview over our multi-target GA compiler system Gaalop.

Using the hardware back-end, Gaalop synthesizes an algorithm specific accelerator architecture from the CLUScript DSL. It easily allows experimentation with different number formats, including support for the automated optimization of fixed-point representations.

From a very concise GA description of the color edge-detection algorithm, the synthesized accelerator achieved not only a speed-up of 11.8x against highly optimized software running on a current-generation processor, but also beat a programmable GA accelerator ASIC by a speed-up of 350x even when constraining the FPGA to run at the same clock rate.

Our research shows the great potential of combining application-specific microarchitectures, implemented on reconfigurable devices, with automated compile flows accepting expressive domain-specific languages.

Future work will focus on further internal optimizations of the intermediate representation of the GA (specifically, inducing the CAS to optimize for fewer operators than for better readability of expressions), and better support for control flow in the input DSL.

References

[1] W. K. Clifford, "Applications of grassmann's extensive algebra," in *Mathematical Papers*, R. Tucker, Ed. Macmillian, London, 1882, pp. 266–276.

[2] ——, "On the classification of geometric algebras," in *Mathematical Papers*, R. Tucker, Ed. Macmillian, London, 1882, pp. 397–401.

[3] D. Hestenes and G. Sobczyk, *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics*. Dordrecht, 1984.

[4] D. Hestenes, *New Foundations for Classical Mechanics*. Dordrecht, 1986.

[5] D. Hestenes and R. Ziegler, "Projective Geometry with Clifford Algebra," *Acta Applicandae Mathematicae*, vol. 23, pp. 25–63, 1991.

[6] D. Hestenes, "Old wine in new bottles : A new algebraic framework for computational geometry," in *Geometric Algebra with Applications in Science and Engineering*, E. Bayro-Corrochano and G. Sobczyk, Eds. Birkhäuser, 2001.

[7] L. Séméria, K. Sato, and G. D. Micheli, “Synthesis of hardware models in c with pointers and complex data structures,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 6, pp. 743–756, 2001.

[8] N. Kasprzyk and A. Koch, “High-level-language compilation for reconfigurable computers,” in *Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC)*, 2005.

[9] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, “Optimized generation of data-path from c codes for fpgas,” in *Design Automation Conference*, 2005.

[10] M. Budiu, “Spatial computation,” Ph.D. dissertation, Carnegie Mellon University, Computer Science Department, December 2003, technical report CMU-CS-03-217.

[11] J. Huthmann, P. Muller, F. Stock, D. Hildenbrand, and A. Koch, “Accelerating high-level engineering computations by automatic compilation of geometric algebra to hardware accelerators,” in *ICSAMOS*, F. J. Kurdahi and J. Takala, Eds. IEEE, 2010, pp. 216–222.

[12] J. Huthmann, P. Müller, F. Stock, D. Hildenbrand, and A. Koch, “Compiling Geometric Algebra Computations into Reconfigurable Hardware Accelerators,” in *Dynamically Reconfigurable Architectures*, P. M. Athanas, J. Becker, J. Teich, and I. Verbauwhede, Eds., Dagstuhl, Germany, 2010.

[13] R. Ablamowicz and B. Fauser, “Mathematics of clifford - a maple package for clifford and graßmann algebras,” in *Advances in Applied Clifford Algebras*. Birkhäuser, 2005.

[14] L. Dorst, S. Mann, and T. Bouma, “Gable: A geometric algebra learning environment,” 2002. [Online]. Available: http://www.science.uva.nl/leo/GABLE

[15] F. Seybold and U. Wössner, “Gaalet - a c++ expression template library for implementing geometric algebra,” in *6th High-End Visualization Workshop*, 2010.

[16] C. Perwass, *Geometric Algebra with Applications in Engineering*. Springer, 2009.

[17] D. Fontijne, “Efficient implementation of geometric algebra,” Ph.D. dissertation, University of Amsterdam, 2007.

[18] C. Schwinn, D. Hildenbrand, F. Stock, and A. Koch, “Gaalop 2.0 - a geometric algebra algorithm compiler,” in *GraVisMa 2010 Proceedings*, V. Skala and E. M., Eds., 2010.

[19] A. Gentile, S. Segreto, F. Sorbello, G. Vassallo, S. Vitabile, and V. Vullo, “Cliffosor, an innovative fpga-based architecture for geometric algebra,” in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2005, pp. 211–217.

[20] S. Franchini, A. Gentile, M. Grimaudo, C. 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.

[21] B. Mishra and P. Wilson, “Color edge detection hardware based on geometric algebra,” in *Visual Media Production, 2006. CVMP 2006. 3rd European Conference on*. IET, 2006, pp. 115–121.

[22] E. Bayro-Corrochano and S. Flores, “Color edge detection using rotors,” in *Applications of Geometric Algebra in Computer Science and Engineering*, L. Dorst, C. Doran, and J. Lasenby, Eds. Birkhäuser Boston, 2002, pp. 333–339.

[23] D. Hildenbrand, *Foundations of Geometric Algebra Computing*. Springer, 2013.

[24] Maxima. (2011) Maxima, a computer algebra system. version 5.25.1. http://maxima.sourceforge.net/. [Online]. Available: http://maxima.sourceforge.net/

[25] H. Lange and A. Koch, “Architectures and execution models for hardware/software compilation and their system-level realization,” *IEEE Transactions on Computers*, vol. 59, no. 10, pp. 1363–1377, 2010.