Accelerating High-Level Engineering Computations by Automatic Compilation of Geometric Algebra to Hardware Accelerators

Jens Huthmann* , Peter Müller*, Florian Stock*, Dietmar Hildenbrand†, and Andreas Koch*
* Embedded Systems and Applications Group
Technische Universität Darmstadt, Germany
Email: {huthmann|mueller|stock|koch}@esa.cs.tu-darmstadt.de
†Computer Science Department
Technische Universität Darmstadt, Germany
Email: dhilden@gris.informatik.tu-darmstadt

***Abstract*—Geometric Algebra (GA), a generalization of quaternions, is a very powerful form for intuitively expressing and manipulating complex geometric relationships common to engineering problems. The actual evaluation of GA expressions, though, is extremely compute intensive due to the high-dimensionality of data being processed. On standard desktop CPUs, GA evaluations take considerably longer than conventional mathematical formulations. GPUs do offer sufficient throughput to make the use of concise GA formulations practical, but require power far exceeding the budgets for most embedded applications. While the suitability of low-power reconfigurable accelerators for evaluating specific GA computations has already been demonstrated, these often required a significant manual design effort. We present a proof-of-concept compile flow combining symbolic and hardware optimization techniques to automatically generate accelerators from the abstract GA descriptions without user intervention that are suitable for high-performance embedded computing.**

I. Introduction

A. Compiling for Reconfigurable Computing

Reconfigurable computers have successfully been used to accelerate a wide spectrum of high-performance embedded applications, while requiring a power budget far below that of Graphics Processing Units (GPUs) with comparable throughput. However, the use of reconfigurable technology often required significant manual implementation effort and knowledge not only of the application, but also of digital design and computer architecture.

As in ASICs, the productivity gap between the HDLs traditionally used for digital design and the ever-increasing FPGA capacities has widened. On one side, this has been addressed by growing the synthesizable subsets of HDLs. Today, some tools can already synthesize variable operand multiplication and division into hardware and infer various kinds of memories directly from the HDL code. On the other side, many attempts have been made to compile from higher-level software programming languages (HLL) into hardware, e.g. [1]–[4].

Despite the progress in that area, translating HLLs into hardware is complex. In many cases, only a limited subset of language constructs can be translated. Restrictions often exist with regard to control flow, data types, and pointer handling. All language features, that software developers expect to be available. Their lack again complicates the use of hardware acceleration by non-experts.

A different approach to compiling to hardware lies in using more abstract domain-specific languages instead of generic HLLs as input. They often pose less difficulty for automatic compilation since, e.g., difficult-to-translate constructs pointers or irregular control flow are not part of the language at all. This has already been done successfully for signal processing applications from MATLAB and Simulink ([5], [6]). Our work also takes this route of compiling from Geometric Algebra, a powerful domain specific language much better suited to hardware mapping than a full HLL.

B. Geometric Algebra

The input language for our compiler are expressions formulated in Geometric Algebra (GA). GA is a very powerful mathematical framework for intuitively expressing and manipulating the complex geometric relationships common to engineering problems. In many cases, GA descriptions require only a fraction of the space of that conventional formulations (e.g., half page instead of dozens of pages).

GA generalizes projective geometry, imaginary numbers, and quaternions to provide a powerful and flexible mathematical framework. It describes the manipulation of multivectors, which are linear combinations of simple vectors (called *blades* in this context). In addition to standard operators such as addition and subtraction, GA also encompasses special operators such as geometric product, inner product, outer product, inverse and division, dual and reverse operators (see [7] for an introduction).

The current form of GA has its roots in work by Grassmann [8] and Clifford [9] from the 19th century. However, its usefulness and wide practical applicability has only recently been discovered. Initially, it became popular in physics to

concisely express complex geometrical relationships [10]–[12].

With the invention of conformal geometric algebra [13] by David Hestenes, this has also been extended to engineering applications such as robotics, computer graphics and computer vision. In conformal geometric algebras, high-level geometric objects such as points, lines, planes and spheres, as well as operations on them (e.g., intersection) can all be concisely expressed using GA operators.

However, due to the significant computation effort necessary to evaluate the multi-dimensional GA expressions, practical adoption has only been limited so far. While modern GPUs do have sufficient compute capacity [14], their long latencies (40 μs for a single computation) and high power requirements (170 W+) make them infeasible for many embedded control scenarios. Most FPGA-based reconfigurable computers do not quite reach the throughput of GPUs, but achieve much shorter latencies (for this example, 2 μs) and a much reduced power draw (here just 7 W). This will be discussed in greater detail in Sec. IV.

II. Related Work

A. Tools

A number of pure software tools exists for working with GA expressions. Some of these, specifically CLUCalc, CLIFFORD, and Gaalop also play a role in our hardware compile flow.

CLUCalc is a software program [7] for developing GA algorithms in CLUCalc-script, a domain specific language. It considerably simplifies development by its ability to graphically visualize the geometric interpretation of GA descriptions in real-time, also extending to accepting user input as geometric data.

CLIFFORD [15] is a library adding GA operations to the symbolic computer algebra system Maple. Now, GA expressions can also be evaluated and simplified (in contrast to CLUCalc, which performs only numerical evaluation). CLIFFORD is limited to multivectors with at most nine dimensions, however, this is sufficient for many practical applications.

Gaalop (Geometric Algebra Algorithms Optimizer) [16] is a plugin-based source-to-source compiler framework. It reads CLUCalc-script programs into a flexible intermediate representation which can then be optimized. Output code can be generated C, LaTeX, dot-graphs or CLUCalc-script (to visualize and verify the output). Gaalop internally uses Maple and CLIFFORD for symbolic transformations. We use it as the base for our hardware compiler.

Gaigen [17], [18] has a similar aim as Gaalop and reaches similar performance when compiling the GA models into C code. However, it requires additional specifications to the GA description (such as identifying zero-coefficient blades). Furthermore, Gaigen is restricted to a C++ code generator.

The main contribution of this work is the extension of Gaalop with numerous hardware-specific optimizations and a code generator for Verilog HDL, thus allowing the compilation of GA models into dedicated hardware accelerators.

B. Hardware Accelerators

With the high computing requirements for actually evaluating a GA model, much effort has been expended on special-purpose processor architectures.

One of the first attempts [19] tried to structurally map GA operators embedded in Prolog descriptions to corresponding hardware units. However, the paper is rather vague and does not give benchmark results for non-trivial examples.

The approach in [20] concentrates on accelerating just the geometric product, using an FPGA-based co-processor running at a clock speed of 20 MHz and attached to the PCI bus of the host computer. It operated on 24b integers and was able to process up to eight-dimensional multi-vectors. Since it operated strictly sequentially (no pipelining was used), lower dimensional multi-vectors could be processed quicker. Performancewise, [20] claims a speed-up of 1.5x over a software implementation in terms of *clock cycles*. When actually considering the clock frequencies of the GA-accelerator and CPU, though, the CPU is roughly 50x faster in terms of *wall-clock time*.

CliffoSor [21] attempts to accelerate more GA operations than the geometric product (e.g., outer product, contraction, etc.). It is restricted to four-dimensional multi-vectors, but does execute the computations in parallel for each component of a multi-vector. Intermediate calculations are performed on 16b integers, with 32b final results. CliffoSor was realized on an FPGA with 50 MHz clock speed, but suffers from large communication overheads with the host machine (49 of the 56 cycles for a geometric product are spent on data transfers). Again, an acceleration over a conventional CPU was demonstrated only on a per-cycle basis. In terms of wall-clock time, the CPU is 9x faster.

[22] is the first co-processor operating on floating-point data. It consists a basic IEEE-754 double precision floating-point unit (FPU) supported by smaller utility units, all executing separate micro-programs for each GA operation. While the FPU itself is pipelined, the micro-programs execute sequentially for each coefficient of the two-dimensional multi-vectors supported. This attempt was actually realized as an ASIC, reaching a clock-speed of 130 MHz. The author's claim a *wall-clock* speed-up of 3x over software [23], but give no details on the CPU used for benchmarking.

S-CliffoSor is another attempt at a general GA co-processor [24] from the same team as CliffoSor. It replaced the GA-operator specific compute units of the latter with so-called *slices* able to execute all GA operators for four-dimensional multivectors, processing 32b integer coefficients. Each slice has an ALU and uses sequential micro-programs to realize the GA operators, pipelining is not performed. The authors argue that multiple slices could be instantiated to achieve greater parallelism, but appear not to have implemented this: The claimed cycle-based speed-up on a 45 MHz FPGA implementation of 3x...4x over a 2 GHz CPU is in reality a wall-clock slowdown of 9x...12x.

The main reason for the poor performance of these prior attempts appears to be the fixation on programmable processor

architectures, even when targeting reconfigurable logic.

As an alternative, [14] evaluates the performance of a fully spatial FPGA realization of a complex inverse kinematics algorithm expressed in GA (see Sec. II-C). The 175 MHz FPGA implementation has a throughput of one result per clock cycle and achieves a wall-clock speed-up of 7x over a very carefully tuned (vector instructions, multi-threaded) C implementation running on a 2.4 GHz quad-core CPU. This success was the main motivation of our work on automatically generating such high-performance compute units.

C. Benchmark Application

As an example for a typical engineering application, we will examine the performance of our proof-of-concept compiler using a inverse kinematic computation expressed in GA: Given a target point and a kinematic chain (e.g., shoulder, upper arm, elbow, forearm, wrist, hand), the algorithm computes the angles of all joints so the target point can be reached.

Such computations occur in practice, e.g., in robotics, or in computer animation (e.g., of human models). In the latter case, the computation speed is actually relevant (in the first, mechanical limits set an upper bound on speed). This specific inverse kinematics algorithm is used in a virtual reality (VR) application [18]. As shown in [25], a formulation in five-dimensional conformal GA was 3x faster in software and much more concise (a page of formulas instead of many pages) than an algorithm using conventional math.

While this specific algorithm would only be used in very specialized embedded systems (e.g., on-board VR/AR visualization systems in vehicles), it is representative both for the expressive power of GA as well as the corresponding computational requirements.

This inverse kinematics algorithm makes for a very interesting benchmark, since we have manually created highly optimized versions for FPGA, GPU and multi-core CPU targets [14], [26]. Each implementation has been carefully hand-tuned for each platform (including, e.g., optimal bit width determination of FPGA operators and multi-threading for the GPU and CPU targets). We can thus judge the performance of the GA-to-hardware compiler using the manual design as a reference.

III. Extending Gaalop

We extend the Gaalop compiler framework with a new back-end to translate its intermediate representation into high-performance pipelined hardware datapaths. In this section, we will give an overview over the entire compile flow.

A. Gaalop Introduction

While CLUCalc is a very productive environment for the interactive development and debugging of GA algorithms, it does not allow the export of the completed models for execution outside of the tool. Gaalop aims to close this gap and export GA models into a variety of external formats (both executable and graphical).

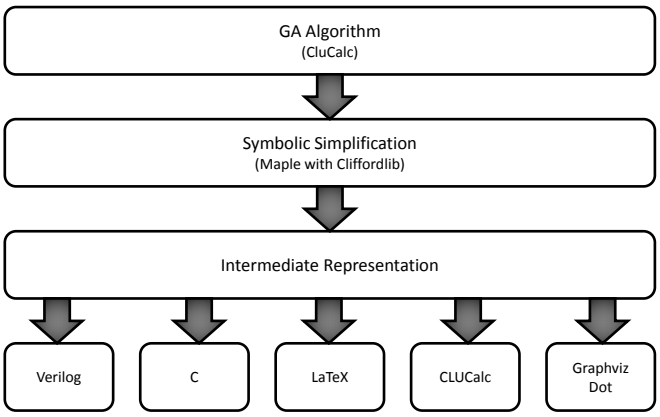


Fig. 1. Compile flow

As shown in Fig. 1, Gaalop reads a description for five-dimensional conformal GA algorithms as developed in CLUCalc. The CLUCalc-script is parsed into an intermediate representation (IR), specifically a control flow graph (CFG) of basic blocks holding the actual GA expression. Each of the blocks is stored as a data flow graph (DFG). The DFG represents the linear combinations of five *blades*. Each blade itself is represented as the outer product of five basis vectors ($e_0, \ldots, e_3, e_\infty$, listed in Tab. I), with the *grade* of the blade being the number of different basis vectors combined. At this stage, the multi-vectors in the DFG may be fed to high-level GA operators. Note that using a CFG in the IR is a forward-looking decision, since CLUCalc-script itself currently does not support control constructs.

From Tab. I, it can be seen that the largest multi-vectors linearly combine at most 32 independent blades. For efficient compilation to a language without GA operators (e.g., C or fully spatial hardware), both the multi-vectors as well as the GA operators combining them have to be translated into their underlying primitive scalar representations and computations.

This is achieved symbolically using the Maple computer algebra system with the CLIFFORD library. With the library, Maple can now symbolically evaluate GA expressions in each DFG, simplifying them. Next, we also symbolically transform the remaining GA operators in the simplified GA expressions into their scalar equivalents, now operating on the individual scalar components of the basis vectors making up the blades. The result are scalar computations, amenable to both parallel as well as pipelined computation. Note that these scalar computations may well include trigonometric and similar functions as operators.

Fig. 2 shows this process of lowering a set of GA expressions into an expression solely consisting of primitive operations.

From this lowered DFG, the various back-ends can then generate code in the desired format. In addition to various textual and graphical formats (for documentation and debugging purposes), we have so far generated executable code in C/++ and CLUCalc-script. In the next Section, we describe the

TABLE I
THE 32 BLADES OF 5D CONFORMAL GA IN GAALOP.

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

flow from the lowered DFG to efficient hardware.

B. IR for Hardware Generation

After performing several standard optimization techniques, i.e. constant folding and common subexpression elimination, the Gaalop-DFG is translated into an expanded form better suited to hardware generation. While also a DFG (now holding only primitive operations acting on scalar data), it also carries additional attributes such as data types (floating or fixed-point), format (bit-widths of integer and fractional parts of fixed-point representations), latencies and scheduling cycles.

C. Word Length Optimization

The area and speed of fully spatial compute units can be improved significantly by matching the width of the hardware operators to the data types processed *at this point* in the calculation. This optimization must be assisted by the developer by specifying the value ranges and precisions of input and output data.

Word length optimization is performed by forward and backward propagation of the desired value ranges and precision. In the forward phase, the incoming value ranges (integer and fractional parts) determine the required width of the operator and its result. In the backward phase, unnecessarily precise (and thus too wide) operators can be narrowed and this truncation also propagated back toward the operator inputs.

```
DefVarsN3();
// Generic example:
// inputs: two points (x1, x2, x3),(p1,p2,p3)
//         two diameters :d1,d2
// two spheres are intersected, and the
// resulting circle is intersected with a plane
// the end result is a pair of points Pp
Pw =x1*e1+x2*e2+x3*e3;
s1 = Pw-0.5*d2*d2*einf;
s2 = e0-0.5*d1*d1*einf;
Ze = s1^s2;

Plane = p1*e1+p2*e2+p3*e3;
?Pp = Ze ^ Plane;
```

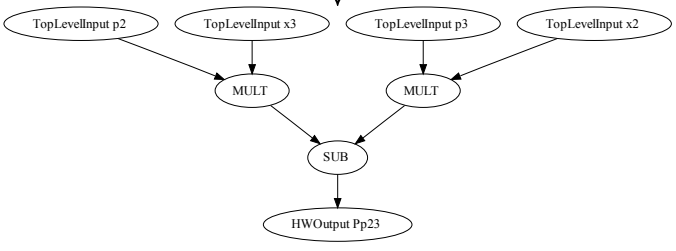
(a)

```
Pw := ((subs(Id=1,(x1 &c e1)) + subs(Id=1,(x2 &c e2))) + subs(Id=1,(x3 &c e3)));
s1 := (Pw - subs(Id=1,(subs(Id=1,(subs(Id=1,(0.5 &c d2)) &c d2)) &c einf)));
s2 := (e0 - subs(Id=1,(subs(Id=1,(subs(Id=1,(0.5 &c d1)) &c d1)) &c einf)));
Ze := (s1 &w s2);
Plane := ((subs(Id=1,(p1 &c e1)) + subs(Id=1,(p2 &c e2))) + subs(Id=1,(p3 &c e3)));
Pp := (Ze &w Plane);
gaalop(Pp);
```

(b)

```
Pw := x1*e1+x2*e2+x3*e3
s1 :=x1*e1+x2*e2+x3*e3-.5*d2^2*e4-.5*d2^2*e5
s2 := -1/2*e4+1/2*e5-.5*d1^2*e4-.5*d1^2*e5
Ze:=x1*(-.5-.5*d1^2)*e14+
    x2*(-.5-.5*d1^2)*e24+
    x3*(-.5-.5*d1^2)*e34+
    .5*d2^2*(-.5-.5*d1^2)*e45+
    x1*(.5-.5*d1^2)*e15+
    x2*(.5-.5*d1^2)*e25+
    x3*(.5-.5*d1^2)*e35-
    .5*d2^2*(.5-.5*d1^2)*e45
Pp :=-.5*x2*(1.+d1^2)*p1*e124-.5*x3*(1.+d1^2)*p1*e134-
     .5*x2*(d1^2-1.)*p1*e125-.5*x3*(d1^2-1.)*p1*e135-
     .5*d2^2*p1*e145-.5*x3*(1.+d1^2)*p2*e234+
     .5*x1*(d1^2-1.)*p2*e125+.5*x1*(1.+d1^2)*p2*e124-
     .5*x3*(d1^2-1.)*p2*e235-.5*d2^2*p2*e245+
     .5*x2*(1.+d1^2)*p3*e234+.5*x1*(d1^2-1.)*p3*e135+
     .5*x1*(1.+d1^2)*p3*e134+.5*x2*(d1^2-1.)*p3*e235-
     .5*d2^2*p3*e345
Plane := p1*e1+p2*e2+p3*e3
gaaloparray(Pp_opt);
Pp_opt[18] := -.5*x2*p1*d1^2+.5*x1*p2*d1^2;
Pp_opt[19] := x2*p1-1.*x1*p2;
Pp_opt[20] := -.5*x3*p1*d1^2+.5*x1*p3*d1^2;
Pp_opt[21] := x3*p1-1.*x1*p3;
Pp_opt[22] := -.5*d2^2*p1;
Pp_opt[23] := -.5*x3*p2*d1^2+.5*x2*p3*d1^2;
Pp_opt[24] := x3*p2-1.*x2*p3;
Pp_opt[25] := -.5*d2^2*p2;
Pp_opt[26] := -.5*d2^2*p3;
```

(c)



(d)

Fig. 2. Converting a geometric algebra expression into primitive scalar operations. (a) GA computation in CLUCalc-script. (b) GA expressions as given to Maple. (c) CliffordLib results in GA, containing only primitive scalar operations. (d) Dataflow graph used for hardware generation. For brevity, we just show the computation of blade 23 of the result *Pp*.

For addition, subtraction, and multiplication the forward propagation is quite simple. However, division or functions such as square root, sine or cosine have more complex behavior. In this proof-of-concept implementation, we currently assume a default value (32b, with 16b fraction) for these functions, but this will be refined in future work. Similarly, we can set a global limit on the width of intermediate values. Note that the operators themselves are not affected by this and compute at the full required precision. Only the result is then clipped to the global limit.

In addition to these established techniques, we can also do word-width optimization based on the original higher-level Gaalop DFG-representation containing GA operators. For the proof-of-concept compiler, e.g., we recognize the normalization of vectors at the GA level, and restrict the output value range of the corresponding scalar operator to $[-1, \ldots, 1]$.

Good examples for operators that profit from backward propagation are inverse trigonometric functions (which will restrict the input value range to $[-1, \ldots, 1]$), or the square root (which limit the input value to be positive). If we cannot determine a narrow value range for an operator analytically, we then perform an automatic Monte-Carlo-Simulation of the entire datapath to achieve a better fit. This Monte-Carlo-Simulation runs in parallel using both floating-point and fixed-point formats to also perform error estimation for all operator nodes.

While we can also directly generate datapaths using single or double-precision floating-point operators, this is currently not practical: The proof-of-concept compiler presented here aims for a fully-spatial implementation (no operator sharing, but higher throughput). Even very simple GA algorithms will quickly lead to hardware exceeding the capacities of even the largest FPGAs. Area optimization of floating-point computations will be one topic for future research (see Sec. V).

D. Scheduling and Balancing

After word-length optimization, the latency of the hardware operations can be determined and the computation actually scheduled. Since we aim for fully spatial operation without operator sharing, we use a simple greedy ASAP (as-soon-as-possible) approach: An operation op_i with latency l_i is scheduled at time $t_i = max_{op_j \in Predecessor(op_i)}\{t_j + l_j\}$, i.e. it is scheduled after the latest predecessor operation has finished its computation.

For maximum pipeline throughput of one result per clock cycle, we then need to balance converging paths with unequal latencies by inserting registers. Also, all paths from all inputs to all outputs need to be brought to equal latency.

Fig. 3 shows the balancing algorithm. The successors j of the current node i in the DFG are sorted by their latency distance. The latter is defined as $dist(op_i, op_j) := t_j - t_i - l_j$, with $op_i \in Predecessor(op_j)$, t being the scheduled start cycles, and l the latency in cycles. If the minimal and maximal and distances are different, a register node is inserted in all paths from the current node that are longer than the shortest path. The register node itself is scheduled at cycle $t_i + D_{\text{first}}$. The algorithm is then restarted on the new register node. The result is a data path with balanced path lengths.

Fig. 3. Balance successors of a node i

let D_{ij} the distance of the current node i to the successor j
sort D_{ij} by ascending distance
if $D_{\text{first}} \neq D_{\text{last}}$ **then**
  create new register node NOP n
  for all successors j of i with $D_k > D_{\text{first}}$ **do**
    remove edge (i, j)
    insert edge (n, j)
  end for
  add edge (i, n)
  execute algorithm for n
end if

E. Hardware Generation

Since the data path is a fully spatial, perfectly balanced pipeline, no additional control logic is required beyond markers indicating if and when results are available in the output (a simple shift register).

We support chaining of some computations within the same clock cycle. At the moment, these are constant shifts, sign/bitwidth extension and bit-select operations that reduce to simple wires.

If the sinks of an operator are scheduled one or more cycles later, the source operator is fitted with a shift register to delay results over that time. Note that the balancing algorithm in Fig. 3 ensured that all sinks (possibly NOP nodes inserted for that very purpose) have the same latency distance from the source operator. Thus, many paths can share the balancing shift register.

Dedicated input registers accept input values for the computation, either as slave-writes from the CPU, or via a streaming mechanism directly from memory. Output registers can also be read from the CPU or streamed back into memory.

The operators themselves are implemented using the flexible module library Modlib [27]. Internally, it expresses simple operators (e.g., addition, etc.) as synthesizable Verilog HDL operators. More complex operators (e.g., multiplication, division, square root, trigonometric functions) are realized internally using the Xilinx CoreGen module generators, using pipelined implementations with maximum throughput. The operators are generated on-the-fly for the specific bit-widths and data types required, caching generated modules for re-use if an operator with the same characteristics occurs again.

IV. Evaluation and Results

As described in Sec. II-C, we use an inverse kinematics application to evaluate the compiler prototype. Specifically, we compare the compiler-generated hardware with an implementation very carefully manually optimized by two experienced designers. In both cases, we target the Xilinx Virtex 5 devices

TABLE II
COMPARISON OF MANUAL DESIGN ([14]) VS. COMPILER-GENERATED DATAPATH (COMPUTE KERNELS ONLY, DISREGARDING COMMUNICATION INTERFACE).

	manual	compiler
# operations	140	258
resources # FFs	49938	71173
resources # LUTs	34912	72664
resources # DSPs	74	817
pipeline length	365	447
max. frequency $[MHz]$	170	180
throughput $[10^6 eval/s]$	170	180
latency $[\mu s]$	2.147	2.483
speed-up to CPU	6.9x	7.3x
average word-length $[bits]$	38	45
average fraction-length $[bits]$	23	41
implementation time [h]	80	$<< 1$

using Synplify Premier for synthesis and Xilinx ISE for mapping.

For a fair comparison of the different platforms, our performance numbers assume that the input and output data is fetched from/stored to memory *local* to the computing device: The CPU has the data in its node-local memory accessed via FSB, the FPGA uses directly attached DRAM, and the GPU processes data in its on-board device memory.

Tab. II compares the area requirements and the performance for both solutions. Obviously, the compiler-generated datapath requires significantly more space than the manually optimized one, specifically a high number of DSP blocks. But with its deeper pipeline, it can be mapped to a Virtex 5 SX 240T device with a slightly higher clock frequency than the manual design.

It is clear that our future work needs to concentrate on area optimization. The human designers exploited a number of high-level algebraic simplifications that are not yet performed automatically using the Maple computer algebra system in the Gaalop flow. This also affects the fixed-point conversion: The manually optimized design contains significantly fewer operators that are infeasible for analytical value range determination. Instead, the compiler has to rely on the Monte-Carlo-Pass to tighten the constraints. That approach, however, suffers from the nature of the Monte-Carlo test data generation: Since we aimed for a general-purpose solution, we generate streams of completely random input vectors. Not all of these will actually be valid inputs for this *specific* problem (e.g., a kinematic chain anchored at the origin can obviously not reach the origin and other points very close to it). Thus, we have to extend operator value ranges to handle values that will actually never appear in practice, leading to wider operators. This explains that the average word-length in the compiler-generated design is 1.2x larger than the one in the manual design.

Performancewise, though, the compiler-generated design performs quite satisfactorily: It slightly exceeds the throughput of the manual design (measured as million function evaluations per second) and has similar latency. It is still significantly better in terms of throughput than a four-threaded software implementation running on a 2.4 GHz Intel Core 2 Quad Q6600 CPU (which would draw 4.6x the power of the FPGA), yielding a real wall-clock speed-up compared to most of the prior approaches outlined in Sec. II. While a GPU under optimum conditions could be even faster (1366M evaluations/s), it also incurs a latency of more that 40 μs on an NVidia GTX 280 card, which also would draw more than 24x the power of the FPGA. [14] gives greater details on these alternate implementations.

Apart from the area and performance issues, however, an automatic tool must be rated by its effect on designer productivity. This is the area where even the proof-of-concept compiler shines: The manual implementation required a total of approx. 80 h of determined effort by two experienced designers, familiar with both digital design/computer architecture as well as the maths underlying GA (which they exploited for the operator-reducing high-level simplifications). The compiler itself takes less than a minute to execute, with the bulk of the total implementation time taken by the Xilinx ISE mapping tools. Now, a *domain expert* proficient in GA can use a familiar notation to describe an algorithm, with no hardware design knowledge required.

V. CONCLUSION AND FUTURE WORK

Even in its proof-of-concept stage, the compiler generates compute pipelines for the GA descriptions with a throughput significantly higher than the carefully tuned software version on a quad-core CPU.

The compiled compute pipeline does not yet reach the performance of the manual reference implementation, but was created in a fraction of the design time (minutes vs. days). Gaalop can already be used to quickly perform experiments with other GA algorithms, something simply not possible if a manual hardware design would be required for each problem.

Ongoing research also tackles going from the fully spatial design presented here to one with a flexible degree of operator sharing. This not only will allow the implementation of even more complex GA applications without using excessive amounts of reconfigurable area, but also the use of smaller reconfigurable devices for less extreme application performance requirements.

The compiler does not yet perform all of the optimizations that were undertaken for the manual design. Specifically, tree height-reduction would have been advantageous. Also, when extending CLUCalc-script with control flow constructs, our very simple word-length optimization has to be replaced with a more precise algorithm, e.g., [28] or [29]. All of these classical techniques will need to be extended to exploit the underlying structure of the high-level GA operators to achieve even tighter word-length fittings. These issues are also the subject of current research in our group.

REFERENCES

[1] M. Budiu, "Spatial computation," Ph.D. dissertation, Carnegie Mellon University, Computer Science Department, December 2003, technical report CMU-CS-03-217. [Online]. Available: http://www.cs.cmu.edu/ mihaib/research/thesis.pdf

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

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

[4] 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.

[5] Xilinx, *MATLAB for Synthesis*, Xilinx, 2008.

[6] ——, *System Generator for DSP*, Xilinx, 2008.

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

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

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

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

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

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

[13] 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.

[14] H. Lange, F. Stock, A. Koch, and D. Hildenbrand, "Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and gpus," in *FCCM*, 2009, pp. 255–258.

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

[16] D. Hildenbrand, J. Pitt, and A. Koch, *Gaalop - High Performance Parallel Computing based on Conformal Geometric Algebra*, ser. American Journal of Mathematics. Springer, 2010, vol. 1, pp. 350–358.

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

[18] D. Hildenbrand, "Geometric computing in computer graphics and robotics using conformal geometric algebra," Ph.D. dissertation, TU Darmstadt, 2006, darmstadt University of Technology.

[19] D. Crookes, K. Alotaibi, B. Bouridane, P. Donachy, and A. Benkrid, "An environment for generating fpga architectures for image algebra-based algorithms," in *Proc. International Conference on Image Processing (ICIP)*, 1998.

[20] C. Perwass, C. Gebken, and G. Sommer, "Implementation of a clifford algebra co-processor design on a field programmable gate array," in *CLIFFORD ALGEBRAS: Application to Mathematics, Physics, and Engineering*, ser. Progress in Mathematical Physics, R. Ablamowicz, Ed., 6th Int. Conf. on Clifford Algebras and Applications, Cookeville, TN. Birkhäuser, Boston, 2003, pp. 561–575.

[21] 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.

[22] B. Mishra and P. Wilson, "Color edge detection hardware based on geometric algebra," in *European Conference on Visual Media Production (CVMP)*, 2006.

[23] B. Mishra and P. R. Wilson, "Vlsi implementation of a geometric algebra parallel processing core," Electronic Systems Design Group, University of Southampton, UK, Tech. Rep., 2006.

[24] 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.

[25] D. Hildenbrand, D. Fontijne, Y. Wang, M. Alexa, and L. Dorst, "Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra," in *Eurographics conference Vienna*, 2006.

[26] D. Hildenbrand, H. Lange, F. Stock, and A. Koch, "Efficient inverse kinematics algorithm based on conformal geometric algebra - using reconfigurable hardware," in *GRAPP*, 2008, pp. 300–307.

[27] H. Gädgke-Lütjens, B. Thielmann, and A. Koch, "A flexible compute and memory infrastructure for high-level language to hardware compilation," submitted to FPL 2010.

[28] M. Budiu and S. C. Goldstein, "Bitvalue inference: Detecting and exploiting narrow bitwidth computations," in *In Proceedings of the EuroPar 2000 European Conference on Parallel Computing*. Springer Verlag, 2000, pp. 969–979.

[29] J. R. C. Patterson, "Accurate static branch prediction by value range propagation," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1995, pp. 67–78.