Specialized Machine Instruction Set for Geometric Algebra

Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz and Andreas Koch

Abstract Geometric algebra covers many mathematical areas such as vector algebra, complex numbers, Pluecker coordinates, quaternions. It is geometrically intuitive and has a lot of potential for optimization and parallelization. In this paper, we develop an approach for the specialized machine instruction set GAPP based on our table-based compilation approach for geometric algebra computing. GAPP can be used as a representation from which consecutive target platform optimizations may be performed. An FPGA platform implementation is even capable of executing this instruction set directly without further transformations, thereby fully exploiting parallelism. An implementation of GAPP as a back end for the Gaalop compiler is already available. This is an important step towards the long term vision of microcomputers specifically designed for Geometric Algebra (Geometric Algebra Computers).

Dietmar Hildenbrand
TU Darmstadt, Germany, e-mail: hildenbrand@cocoon.tu-darmstadt.de

Patrick Charrier
TU Darmstadt, Germany, e-mail: patrick.charrier@stud.tu-darmstadt.de

Christian Steinmetz
TU Darmstadt, Germany, e-mail: christian.steinmetz@stud.tu-darmstadt.de

Andreas Koch
TU Darmstadt, Germany, e-mail: koch@esa.informatik.tu-darmstadt.de

1 Introduction

Conformal Geometric Algebra (CGA) is a new way of expressing many geometry-focused mathematical problems. It deals naturally with intersections and transformations of planes, lines, spheres, circles, points, and point pairs, and is also expressive enough to represent kinematics and dynamics. In Linear Algebra one would have to differentiate a plane-sphere intersection into three distinct cases, namely, circle intersection, point intersection and no intersection. In Conformal Geometric Algebra the intersection itself is formulated as one operation on the plane (P) and the sphere (S), respectively.

$$R = S \wedge P$$

The three different cases of Linear Algebra are implicitly contained in the one result R of Conformal Geometric Algebra, which is compact and better readable. Similar observations can be made in other applications of geometry related mathematics. Therefore, when applied to computer programs, CGA has a high potential for improving code readability and to shorten production cycles. It has also been proven, that if implemented correctly, Geometric Algebra has at least similar performance, but sometimes even better performance, than conventional approaches [8].

This paper works out a table-based compilation approach for Geometric Algebra, as introduced in [5], and extends it in order to achieve higher adaptation to the target computing platforms and better runtime performance.

Most multivector calculations derived with the proposed table-based approach break down to a scalar multiplication (dot product) of two equal-sized n-dimensional vectors. Such a scalar multiplication intrinsically contains a high level of parallelism. This gives motivation to the abstract language defined in the next section.

2 Geometric Algebra Algorithm Parallelism Programs (GAPP)

The instruction set of modern microcomputers contains operations that are performed on vectors in parallel (Intel SSE, AMD 3DNow!). A vector in this context is simply a concatenation of scalar values. Multiplying two vectors means parallely multiplying all elements of vector a to the corresponding elements of vector b. Also, Field Programmable Gate Arrays may be configured to support similar operations.

Unfortunately, finding such fine grained parallelism is usually a hard task and a very common problem in compiler construction. Most interestingly, Geometric Algebra and the presented table-based approach intrinsically expose instruction level-parallelism. In order to support as many of the various target platform instruction-sets as possible, our approach is to define an abstract language, that is subsequently being transformed into the target instruction set. This language is defined in table 1.

As described in the introduction, a scalar multiplication (or dot product) of two equal-sized n-dimensional vectors is a reoccurring pattern. Since a scalar multiplica-

resetMv mv_{dest}**;** Zeros all blades of multivector mv_{dest}.
assignMv $mv_{dest}[sel_0,\ldots,sel_n] = \{const_1,\ldots,const_n\}$**;** Assigns the constants $const_1,\ldots,const_n$ to the multivector mv_{dest} as blade coefficients specified by the selectors $sel_0,\ldots,sel_n$.
setMv $mv_{dest}[dest_0,\ldots,dest_n] = mv_{src}[src_0,\ldots,src_n]$**;** Copies the selected blades from multivector mv_{src} to multivector mv_{dest}. $dest_0$, src_0, $dest_1$, src_1, up to $dest_{31}$ and src_{31}, are blade selectors. Note that it is invalid language syntax to have more than one source multivector specified in this command. To copy elements from several multivectors it is required to use multiple *setMv* commands, one for each multivector. This command is restricted to one source and destination multivector.
setVector $part_{dest} = mv_{src}[sel_0,\ldots,sel_n]$**;** Composes the vector (part of a multivector) $part_{dest}$ from selected elements. sel_0, sel_1, up to sel_{31}, are a blade selectors. Parts and blade selectors are explained below.
dotVectors $mv_{dest}[sel] =< part_1, part_2 >$**;** Performs a scalar multiplication (dot product) on the two vectors (parts of multivectors) $part_1$ and $part_2$. Saves the result in multivector mv_{dest} at the location selected by selector sel.

Table 1 The main commands of the Geometric Algebra Parallelism Programs (GAPP) language; a more detailed list can be found in [10]

tion intrinsically contains a high level of parallelism, in our abstract GAPP language it is defined in the *dotVectors* command.

A vector in this sense is simply the concatenation of selected blades, as in the command *setVector*. It does not contain any information about the blades itself, which differentiates it from a multivector. All this information is removed by *setVector*. A vector is nothing more than an array of signed real numbers and is a preparation of the data for the actual computation performed by *dotVectors*.
For example, if we select blades E_0,$-E_2$, E_3, E_5, and $-E_4$, those blades are stored in a five-dimensional vector in the same order and with the applied sign. Figure 1 visualizes the example.

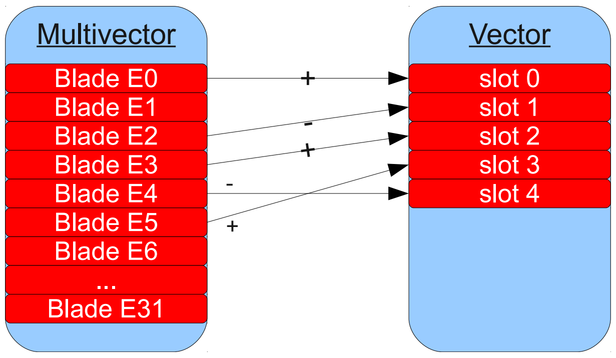

Fig. 1 Selected blades E_0,$-E_2$, E_3, E_5, and $-E_4$ are stored in a five-dimensional vector in the same order and with the sign selected.

Depending on the implementation, vectors may also be higher-dimensional with all remaining elements being set to zero, because zero-elements do not have any impact on the result of the scalar multiplication. An FPGA implementation may, for example, always allocate 32-dimensional vectors for implementational reasons.

The selecting of multivector parts can for example be established by a parallel cascade of multiplexers. The scalar product of multivector parts is decomposed into one parallel multiplication command and $log_2(n)$ parallel additions, with n being the dimensionality of a multivector part. See the example in figure 2.

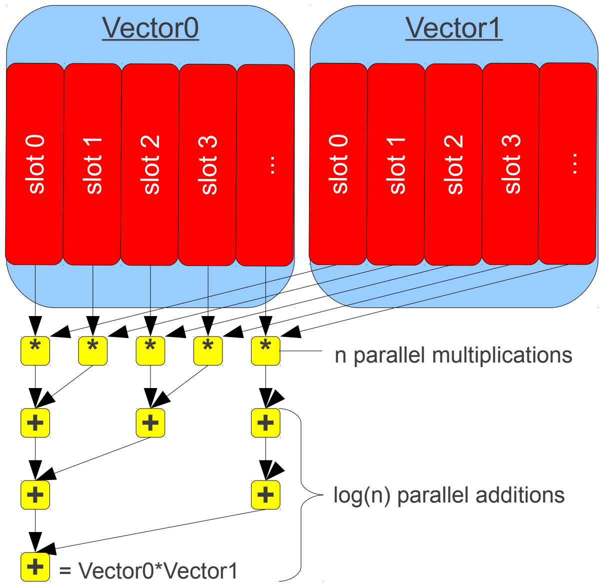

Fig. 2 Parallel scalar product of two vectors, as performed by *dotVectors*.

A blade selector consists of the index of a multivector entry and its sign. The multivector entry index is equal to one of the indices from table 2. To select signed multivector entries we use the indices stated in the table. For example, selecting with index 10 returns the coefficient of blade $e_2 \wedge e_3$. Accordingly, selecting with index -10 returns the negated coefficient of the same blade. This can be efficiently implemented on parallel hardware.

3 Compilation

In order to perform geometric algebra algorithms, the rules for the computation of the products of multivectors must be known. These products can be summa-

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

Table 2 The 32 blades of 5D Conformal Geometric Algebra, that compose a multivector, with corresponding selectors. The first column entry is the corresponding selector. It indexes the positive coefficient of the corresponding blade. The second column is a selector that points to the same coefficient, but brings about a negation of the selected coefficient.

rized (and pre-computed) in multiplication tables describing the product of different blades of the algebra.

The geometric product, the outer product, and the inner product are linear products. They are distributive over addition (see [4] for details). For instance, the geometric product

$$x = ab = \left(\sum_i a_i E_i\right)\left(\sum_j b_j E_j\right) \tag{1}$$

Table 3 Multiplication table for the geometric product of the 2D geometric algebra. This algebra consists of basic algebraic objects of grade (dimension) 0, the scalar, of grade 1, the two basis vectors e_1 and e_2 and of grade 2, the bivector $e_1 \wedge e_2$.

	1	e_1	e_2	$e_1 \wedge e_2$
1	1	e_1	e_2	$e_1 \wedge e_2$
e_1	e_1	1	$e_1 \wedge e_2$	e_2
e_2	e_2	-$e_1 \wedge e_2$	1	-e_1
$e_1 \wedge e_2$	$e_1 \wedge e_2$	-e_2	e_1	-1

Table 4 Multiplication table of the 2D geometric algebra in terms of its basis blades E1, E2, E3 and E4.

		b	**b1**	**b2**	**b3**	**b4**
			E1	E2	E3	E4
a			1	e_1	e_2	$e_1 \wedge e_2$
a1	E1	1	E1	E2	E3	E4
a2	E2	e_1	E2	E1	E4	E3
a3	E3	e_2	E3	-E4	E1	-E2
a4	E4	$e_1 \wedge e_2$	E4	-E3	E2	-E1

of two arbitrary multivectors $a = \sum_i a_i E_i$ and $b = \sum_j b_j E_j$ can be written as

$$x = \sum_i \sum_j a_i b_j (E_i E_j) \tag{2}$$

or as a linear combination of blades $E_{i,j}$

$$x = \sum_i \sum_j a_i b_j (m_{i,j} E_{i,j}) \tag{3}$$

with $m_{i,j}$ being 0, 1 or -1, or as

$$x = \sum_i \sum_j c_{i,j} E_{i,j} \tag{4}$$

with coefficients $c_{i,j} = m_{i,j} a_i b_j$.

This can be rearranged to

$$x = \sum_k c_k E_k \tag{5}$$

with

$$c_k = \sum_{i,j:E_{i,j}=E_k} m_{i,j} a_i b_j. \tag{6}$$

To clarify the concept, we hereby provide some simple algorithmic steps in pseudo-code for the computation of a product of two general multivectors $a = \sum_i a_i E_i$ and $b = \sum_j b_j E_j$.

```
Set all blades of result multivector $c$ to zero;

for each blade coeff $a_i$ of multivector a
    for each blade coeff $b_j$ of multivector b
        Look up target blade $E_{i,j}$ and
        sign $m_{i,j}$ from Multiplication Table;
        ($i$ is the row index and
         $j$ is the column index.)

        Add simple arithmetic product $a_i$ * $b_j$
        with sign $m_{i,j}$ to target blade $E_{i,j}$
        of result multivector $c$:
        $c[E_{i,j}]$ = $c[E_{i,j}]$ + $m_{i,j}$ * ($a_i$ * $b_j$);
    end;
end;
```

For the 2D Euclidean GA this is described in table 4. Each entry $m_{i,j}E_{i,j}$ describes the geometric product of two basis blades E_i and E_j expressed in terms of the basis blades E_k with positive or negative sign. Each coefficient c_k of the product $x = ab$ can be computed by summing up the products $\pm a_i * b_j$ based on the E_k table entries, for instance $c_1 = a_1 * b_1 + a_2 * b_2 + a_3 * b_3 - a_4 * b_4$ for the E_1 table entries.

4 Example

As an example for the compilation of GAPP code we use the example of [5] . Note that this example is based on 3D Euclidean geometric algebra, whereas we have used 5D conformal geometric algebra in Table. 2. In order to stay compatible to [5], we use the multiplication tables of [5], while, in the meantime, these has been changed to multiplication tables starting with an index of 0 (see Table. 2). A consistent description will be published in [6].

Our example uses the following expression

$$f = a \wedge (a + ab) \tag{7}$$

of two 3D vectors a and b. In terms of a CLUCalc script [9], this can be expressed as follows:

```
a=a1*e1+a2*e2+a3*e3;
b=b1*e1+b2*e2+b3*e3;
f=a^(a+a*b);
```

First, the compiler must transform complex expressions to expressions that can easily be handled by multiplication tables. A corresponding CLUCalc script with only simple expressions could look as follows:

```
a=a1*e1+a2*e2+a3*e3;
b=b1*e1+b2*e2+b3*e3;
c=a*b;
d=a+c;
f=a^d;
```

Let us now compile this script step by step into the intermediate language representation explained in section 2.

The first two lines are used for the definitio,n as well as for an automatic specialization of the two multivectors $a = a_1E_1 + a_2E_2 + a_3E_3$. $a1, a2, a3, b1, b2$, and $b3$ are regular scalar variables.

```
resetMv a;
assignMv a[1,2,3] = {a1,a2,a3};
```

and $b = b_1E_1 + b_2E_2 + b_3E_3$

```
resetMv b;
assignMv b[1,2,3] = {b1,b2,b3};
```

For both, only the entries 1, 2 and 3 are needed since they correspond to the three basis vectors e_1, e_2, e_3. Table 3 of the paper [5] shows the corresponding multiplication table for the geometric product of these multivectors. It is derived from the table 1 of the paper [5] with empty rows and columns for multivector entries not needed for a and b. The resulting multivector c only needs the coefficients for the blades E_1, E_5, E_6, E_7 (see table 3 of the paper [5]). Each coefficient $c[k]$ can be computed by summing up the products $\pm a_ib_j$ based on the E_k table entries, for instance, $c_1 = a_1b_1 + a_2b_2 + a_3b_3$.

Expressed in the GAPP language defined in section 2, the code looks as follows.

```
setVector tmp1 = a[1,2,3];
setVector tmp2 = b[1,2,3];
dotVectors c[1] = <tmp1,tmp2>;
setVector tmp1 = a[1,-2];
setVector tmp2 = b[2,1];
dotVectors c[5] = <tmp1,tmp2>;
setVector tmp1 = a[2,-3];
setVector tmp2 = b[3,2];
dotVectors c[6] = <tmp1,tmp2>;
setVector tmp1 = a[1,-3];
setVector tmp2 = b[3,1];
dotVectors c[7] = <tmp1,tmp2>;
```

In the fourth line of the CLUCalc script, the two multivectors a and c are added resulting in the multivector d:

In the GAPP language.

```
setMv d[1,5,6,7] = c[1,5,6,7];
setMv d[2,3,4] = a[2,3,4];
```

This sets the coefficients of blades [1,5,6,7] of multivector c as coefficients of blades [1,5,6,7] of multivector d. Coefficients [2,3,4] of a are set as coefficients [2,3,4] of d with corresponding serial code:

```
d[1]=c[1];
d[2]=a[1];
d[3]=a[2];
d[4]=a[3];
d[5]=c[5];
d[6]=c[6];
d[7]=c[7];
```

The evaluation of the outer product of a with this just computed multivector d leads to the following GAPP language code.

```
setVector tmp1 = a[1];
setVector tmp2 = d[1];
dotVectors f[2] = <tmp1,tmp2>;
setVector tmp1 = a[2];
setVector tmp2 = d[1];
dotVectors f[3] = <tmp1,tmp2>;
setVector tmp1 = a[3];
setVector tmp2 = d[1];
dotVectors f[4] = <tmp1,tmp2>;
setVector tmp1 = a[1,-2];
setVector tmp2 = d[3,2];
dotVectors f[5] = <tmp1,tmp2>;
setVector tmp1 = a[2,-3];
setVector tmp2 = d[4,3];
dotVectors f[6] = <tmp1,tmp2>;
setVector tmp1 = a[1,-3];
setVector tmp2 = d[4,2];
dotVectors f[7] = <tmp1,tmp2>;
setVector tmp1 = a[-2,3,1];
setVector tmp2 = d[7,5,6];
dotVectors f[8] = <tmp1,tmp2>;
```

For this computation, you can use the multiplication table 2 of the paper [5]. Associating the rows with the multivector a and the columns with d we are able to set the rows 1, 5, 6, 7, 8 as well as the column 8 to zero. We recognize that the remaining entries are for the coefficients 2, 3, 4, 5, 6, 7 and 8, E_2 for instance in the second row and the first column associated with the product $a_1 * d[1]$.

5 Target Architectures

GAPP instructions can be mapped to a number of different processor architectures and parallel execution models. This can extend to shared-memory multiprocessor approaches such as OpenMP, short-vector instructions such as SSE4/5 and AVX extending conventional CPUs, the highly-threaded operations on general-purpose GPUs, but also to application-specific computers realized in reconfigurable hardware.

The core arithmetic operation of GAPP is the scalar (dot) product of two vectors, which is a well supported primitive on all of the target architectures given above. However, especially for the smaller vectors (at most a few dozen elements) used in the GAPP primitives, the coarse-grain parallelism in OpenMP for the GPGPU allow only limited scaling with the number of processor cores.

Short-vector instruction sets such as Intel SSE4/5 [3], available on most desktop and server CPUs, also allow the efficient processing of the *smaller* vectors typical for GAPP. However, all of the processor architectures described thus far have only very limited support for efficiently performing the flexible vector permutations (sometimes called *shuffle*) operations required by GAPP.

Such permutations, on the other hand, could be very efficiently implemented on an application-specific compute architecture realized using reconfigurable devices such as FPGAs. For example, an arbitrary permutation of 32 single-precision numbers could be realized within a single clock cycle by using a parallel multiplexer network in hardware. Current-generation FPGAs support floating-point computations. For example., a single-precision architecture, processing 32-element vectors, requires just 30% of the FPGA capacity of a reconfigurable computer such as the Convey HC-1ex [2]. It can perform the 32-element dot product within six clock cycles and has a peak memory bandwidth of 80 GB/s, far exceeding that of a current-model CPU (typically peaking at 25.6 GB).

Given that GAPP itself is machine independent, the high-level optimizations described in prior sections can be performed in early compiler passes. Only the back-end of a compiler using GAPP as an intermediate language must deal with mapping the primitives to the specific target architecture.

In the meantime, the described table-based compilation approach is part of the Gaalop compiler[10] and the GAPP instruction set is used for the OpenCL backend and for the Gaalop precompiler for OpenCL [1], available for download at [7].

6 Conclusion and Future Work

Currently, the presented parallel computing platforms can be seen as approximations to perfect geometric computers. The GAPP language has already been implemented as a back end of the Gaalop compiler (www.Gaalop.de). As our long-term vision, we hope that this research will lead to computing platforms optimally supporting geometric algebra computers in the future.

References

1. Patrick Charrier. Geometric algebra enhanced precompiler for c++ and opencl. Master's thesis, TU Darmstadt, 2012.
2. Convey Computer Corp. Convey HC-1ex Datasheet, 2010.
3. Intel Corp. *Intel SSE4 Programming Reference*. 2007.
4. Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry*. Morgan Kaufman, 2007.
5. Dietmar Hildenbrand. Geometric algebra computers. In *proceedings of the GraVisMa workshop, Plzen*, 2009.
6. Dietmar Hildenbrand. *Foundations of Geometric Algebra Computing*. Springer, to be published in 2012.
7. Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Joachim Pitt. The Gaalop home page. Available at http://www.gaalop.de, 2012.
8. Dietmar Hildenbrand, Daniel Fontijne, Yusheng Wang, Marc Alexa, and Leo Dorst. Competitive runtime performance for inverse kinematics algorithms using conformal geometric algebra. In *Eurographics conference Vienna*, 2006.
9. Christian Perwass. The CLU home page. Available at http://www.clucalc.info, 2010.
10. Christian Steinmetz. Optimizing a geometric algebra compiler for parallel architectures using a table-based. In *Bachelor thesis TU Darmstadt*, 2011.