Geometric Algebra Computing for Heterogeneous Systems

D. Hildenbrand, J. Albert, P. Charrier and Chr. Steinmetz

Abstract. Starting from the situation 15 years ago with a great gap between the low symbolic complexity on the one hand and the high numeric complexity of coding in Geometric Algebra on the other hand, this paper reviews some applications showing, that, in the meantime, this gap could be closed, especially for CPUs.

Today, the use of Geometric Algebra in engineering applications relies heavily on the availability of software solutions for the new heterogeneous computing architectures. While most of the Geometric Algebra tools are restricted to CPU focused programming languages, in this paper, we introduce the new Gaalop (**G**eometric Algebra **al**gorithms **op**timizer) Precompiler for heterogeneous systems (CPUs, GPUs, FPGAs, DSPs ...) based on the programming language C++ AMP (Accelerated Massive Parallelism) of the HSA (Heterogeneous System Architecture) Foundation. As a proof-of-concept we present a raytracing application together with some computing details and first performance results.

Mathematics Subject Classification (2010). Primary 99Z99; Secondary 00A00.

Keywords. Geometric Algebra Computing, Ray tracer, C++ AMP, Gaalop.

1. Introduction

Especially since the introduction of CGA (Conformal Geometric Algebra) by David Hestenes et al. [11] [16] there has been an increasing interest in using Geometric Algebra (GA) in engineering. The use of Geometric Algebra in engineering applications relies heavily on the availability of an appropriate computing technology. The main problem of *Geometric Algebra Computing* is the exponential growth of data and computations compared to linear algebra, since the multivector of an n-dimensional Geometric Algebra is 2^n-dimensional. For the 5-dimensional Conformal Geometric Algebra, as used in this article, the multivector is already 32-dimensional. In 2000, Gerald

Sommer stated in the preface of his book [21]: *Today we have to accept a great gap between the low symbolic complexity on the one hand and the high numeric complexity of coding in GA on the other hand. Because available computers cannot even process complex numbers directly, we have to pay a high computational cost at times, when using GA libraries, ... , full profit in real-time applications is only possible with adequate processors.*

What kind of processors do we need for Geometric Algebra 15 years later? How suitable for Geometric Algebra is the current world of parallel heterogenous systems?

Sect. 2 reviews some powerful solutions for Geometric Algebra with a focus on hardware and CPU implementations. In Sect. 3 we focus on the **G**eometric Algebra **al**gorithms **op**timizer Gaalop and some competitive applications in the sense of being faster than conventional solutions.

While the Gaalop Precompiler [14] already supports CPUs, GPUs[1] and FPGAs, in this article, we introduce a solution for an even broader range of heterogeneous computing architectures defined by the HSA Foundation (see Sect. 4). Since the HSA Foundation focuses on heterogeneous computing with the programming language C++ AMP (see Sect. 5), we extended our Gaalop Precompiler accordingly (see Sect. 7), in order to support all the solutions of the companies of this foundation. This Geometric Algebra Computing solution is now part of the ecosystem of the HSA Foundation. As a proof-of-concept we describe a raytracing application implemented by the Gaalop Precompiler for C++ AMP in Sect. 8 together with some computing details in Sect. 9 and first performance results in Sect. 10.

2. Computing with Geometric Algebra

For many engineering applications runtime performance is a big issue. One method to attempt to overcome the limitations of Geometric Algebra has been to look for dedicated hardware architectures for the acceleration of Geometric Algebra algorithms. Integrated circuit technology offers a means to achieve high performance with field-programmable gate arrays (FPGAs). See, for instance, the solutions by Perwass et al. [19], Gentile et al.[9], Franchini et al. [7] and Mishra and Wilson [17]. Recently, Franchini et al. [8] succeeded in a hardware design natively supporting complete Conformal Geometric Algebra geometric operations based on reflections realized in hardware.

Another approach to overcoming the runtime limitations of Geometric Algebra has been through optimized software solutions. Tools have been developed for high-performance implementations of Geometric Algebra algorithms such as the C++ software library generator Gaigen 2 from Daniel Fontijne and Leo Dorst of the University of Amsterdam [5], GMac from Ahmad Hosney Awad Eid of Suez Canal University [4], the Versor library [2] from Pablo Colapinto, the C++ expression template library Gaalet [20] from

[1]this text does not distinguish between the terms GPGPU and GPU, but always uses the term GPU

Florian Seybold of the University of Stuttgart, and our Gaalop compiler [14]. Ten years ago, in 2006 we, together with the Amsterdam group, presented the first Geometric Algebra application that was faster than the standard implementation [15], an inverse kinematics algorithm of a virtual character.

3. Competitive Runtime Performance using Gaalop

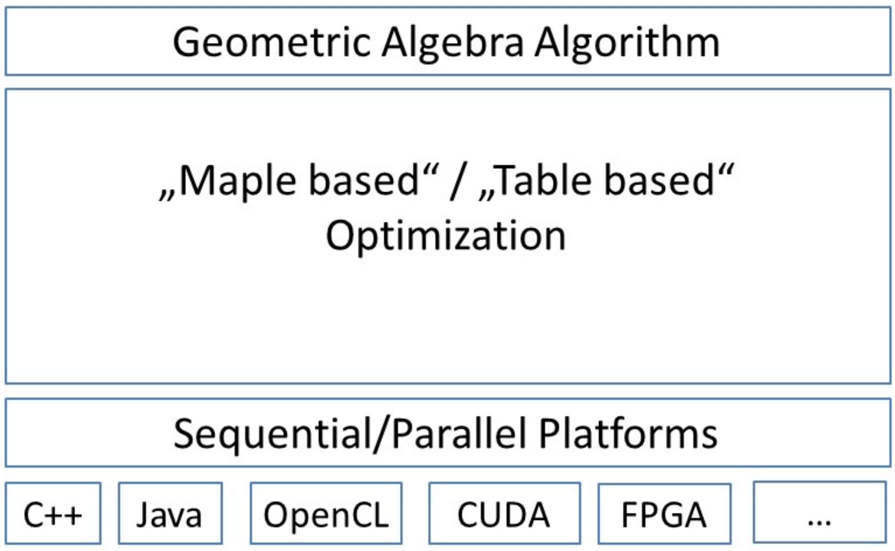
Geometric Algebra Algorithm
„Maple based“ / „Table based“ Optimization
Sequential/Parallel Platforms
C++
Java
OpenCL
CUDA
FPGA
...

Figure 1. Geometric Algebra Computing architecture of Gaalop

A good way of cutting the high complexity of Geometric Algebra before going to the real computing device is to precompute / precompile Geometric Algebra algorithms (see Fig. 1 and [13]). The big potential of optimizations of Geometric Algebra algorithms before runtime can be very good demonstrated with the inverse kinematics algorithm of [15] [12], which was in 2006 the first Geometric Algebra application that was faster than the standard implementation. Naively implemented based on the Gaigen1 library, the runtime was more than 100 times slower than compared to an implementation based on symbolic simplifications with the help of Maple (the Maple based approach of Gaalop). This means, that in this application more than 99% of computing time can be saved and less than 1% is left to the computing at runtime. In this computer animation application, the Geometric Algebra implementation based on symbolic simplification was three times faster than the conventional solution.

In 2009, a very remarkable result in terms of runtime performance could be achieved with the robot grasping algorithm of [23]. Based on optimized

C-code generation with Gaalop, a speedup of 14 could be achieved compared to the conventional mathematics solution.

In the meantime, Gaalop has been extended with an additional (non-commercial) Table based approach (see [13]) and improved in two directions: On one hand, it supports optimized hardware generation [22]. Compared to all the above mentioned hardware solutions, Gaalop precompiles the Geometric Algebra algorithm before generating the hardware description. On the other hand, Gaalop has been extended to a Gaalop Precompiler for many of the newest programming languages, as described in the book [13]. It supports, for instance, GPU programming languages such as CUDA and OpenCL while most of the other software solutions are restricted to CPU focused programming languages such as C++ or C#.

Using the Gaalop Precompiler for OpenCL, the molecular dynamics simulation of Chapt. 13 of [13], achieves very competitive results: the Gaalop Precompiler implementation is faster than the conventional implementation, which is not self-evident for Conformal Geometric Algebra-based implementations of such complexity. Further tests have shown that Gaalop Precompiler also yields a higher numerical stability in terms of energy conservation. This might be due to the fact that the advanced symbolic simplification performed by the Gaalop Precompiler minimizes the number of operations, which otherwise would have been potential sources for numerical errors. For computing details of Gaalop please refer to Sect. 9.

As a solution for the computing architecture of the HSA Foundation (see Sect. 4) we present in Sect. 7 our Gaalop Precompiler for C++ AMP as the main contribution of this article.

4. HSA Foundation

The HSA Foundation [6] is a not-for-profit industry standards body of about 40 companies, founded by AMD, ARM, Imagination, Mediatek, Qualcomm, Samsung and Texas Instruments. It is focused on making it dramatically easier to program heterogeneous computing devices for parallel computation utilizing CPUs, GPUs, DSPs, etc.

Heterogeneous computing is emerging as a requirement for power-efficient system design: modern platforms no longer rely on a single general-purpose processor, but instead benefit from dedicated processors/accelerators tailored for each task. Traditionally these specialized processors have been difficult to program due to separate memory spaces, kernel-driver-level interfaces, and specialized programming models. The Heterogeneous System Architecture (HSA) aims to bridge this gap by providing a common system architecture and a basis for designing higher-level programming models for all devices (including widely used system-on-chip devices, such as tablets, smartphones, and other mobile devices).

5. C++ AMP

C++ AMP is an extension to C++ that enables the acceleration of C++ code on data-parallel hardware (GPUs etc.). The first specification was published by Microsoft in August 2012 as an open specification. The first implementations were available in Visual Studio 2012 and Visual Studio 2013.

With the goal of making it dramatically easier to program heterogeneous computing devices, the HSA foundation released their C++ AMP (Accelerated Massive Parallelism) compiler for Linux in Aug. 2014. C++ AMP version 1.2 enables C++ developers to accelerate applications across a broad set of hardware and software configurations by supporting three outputs:

- Khronos Group **OpenCL**,
  supporting AMD CPU/APU/GPU, Intel CPU/APU, NVIDIA GPU, Apple Mac OS X and other OpenCL compliant platforms;
- Khronos Group **SPIR**,
  supporting AMD CPU/APU/GPU, Intel CPU/APU and SPIR compliant platforms; and
- HSA Foundation **HSAIL**,
  supporting AMD APU and HSA compliant platforms.

As follows, we describe the main extensions of C++ AMP for accelerators:

parallel_for_each describes a computation to be performed by an accelerator accross some N-dimensional execution domain. It expects the number of threads and a lambda function describing the functionality to be executed for each thread.

The ADT (abstract data type) **array_view** $< T, N >$ logically represents an N-dimensional space of type T which resides either on the memory space of the host or of the accelerator, for instance a 2-dimensional pixel array of colours as described in Listing 1.

The ADT **index** $< N >$ represents an N-dimensional point, for instance one point of a 2-dimensional pixel array.

For details please refer to [10].

6. Conformal Geometric Algebra

As shown in [3] for Gaigen and in [4] for GMAC, Conformal Geometric Algebra is very well suitable to realize raytracing applications. This is primarily because of its easy handling of geometric objects such as spheres, planes and lines. In this section we give a brief overview of Conformal Geometric Algebra.

Conformal Geometric Algebra uses the three Euclidean **basis vectors** e_1, e_2, e_3 and two additional basis vectors e_+, e_- with positive and negative signatures, respectively, which means that they square to $+1$ as usual (e_+) and to -1 (e_-).

$$e_+^2 = 1, \qquad e_-^2 = -1, \qquad e_+ \cdot e_- = 0. \tag{6.1}$$

Another basis e_0, e_∞, with the geometric meaning

- e_0 represents the 3D origin,
- e_∞ represents infinity,

can be defined with the relations

$$e_0 = \frac{1}{2}(e_- - e_+), \qquad e_\infty = e_- + e_+. \tag{6.2}$$

Blades are the basic algebraic elements of Geometric Algebra. Conformal Geometric Algebra consists of blades with **grades** 0, 1, 2, ..., 5, where a scalar is a **0-blade** (a blade of grade 0) and the **1-blades** are the basis vectors. The **2-blades** $e_i \wedge e_j$ are blades spanned by two 1-blades, and so on. There exists only one element of the maximum grade 5. It is therefore also called the **pseudoscalar**. A linear combination of k-blades is called a k-vector (or a vector, bivector, trivector. ...). The sum $e_2 \wedge e_3 + e_1 \wedge e_2$, for instance, is a bivector. A linear combination of blades with different grades is called a **multivector**. Multivectors are the general elements of a Geometric Algebra. Table 3 shows the 32 blades of Conformal Geometric Algebra, consisting of the scalar, five (basis) vectors, ten bivectors, ten trivectors, 5 quadvectors and the pseudoscalar.

Conformal Geometric Algebra provides a great variety of basic geometric entities to compute with, namely points, spheres, planes, circles, lines, and point pairs, as listed in Table 1. These entities have two algebraic representations: the IPNS (inner product null space) and the OPNS (outer product null space). These representations are duals of each other (a superscript asterisk denotes the dualization operator). In Table 1, $\mathbf{x}$ and $\mathbf{n}$ are in bold type to indicate that they represent 3D entities obtained by linear combinations of the 3D basis vectors e_1, e_2, and e_3:

$$\mathbf{x} = x_1 e_1 + x_2 e_2 + x_3 e_3. \tag{6.3}$$

The $\{S_i\}$ represent different spheres, and the $\{\pi_i\}$ represent different planes. In the OPNS representation, the outer product "$\wedge$" indicates the construction of a geometric object with the help of points $\{P_i\}$ that lie on it. A sphere, for instance, is defined by four points $(P_1 \wedge P_2 \wedge P_3 \wedge P_4)$ on this sphere. In the IPNS representation, the meaning of the outer product is an intersection of geometric entities. A circle, for instance, is defined by the intersection of two spheres $S_1 \wedge S_2$. Accordingly, the intersection of a line and a sphere can easily be expressed with the help of the outer product of these two geometric entities (Fig. 2).

7. The Gaalop Precompiler for C++ AMP

The first version of Gaalop has been a stand-alone compiler. It was able to generate optimized C code from Geometric Algebra descriptions. In order to simplify the use of Geometric Algebra in engineeering applications, we have developed the Gaalop Precompiler (gpc for short) , which integrates Geometric Algebra into standard programming languages [13]. Figure 3 outlines the

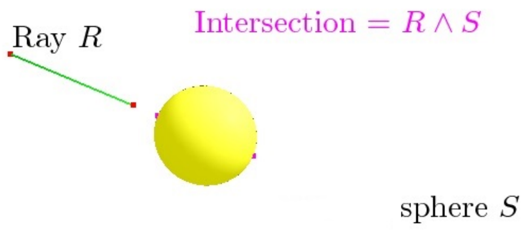

FIGURE 2. Spheres and lines are basic entities of Geometric Algebra that one can compute with. Operations such as the intersection of these objects are easily expressed with the help of their outer product. In our raytracing application, for instance, the result of the intersection of a ray and a sphere is another geometric entity: the point pair consisting of the two points where the line intersects the sphere. The sign of the square of the point pair indicates easily whether there is a real intersection or not.

TABLE 1. The two representations (IPNS and OPNS) of conformal geometric entities. The IPNS and OPNS representations are dual to each other, which is indicated by the asterisk symbol.

Entity	**IPNS representation**	**OPNS representation**
Point	$P = \mathbf{x} + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$	
Sphere	$S = P - \frac{1}{2}r^2 e_\infty$	$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4$
Plane	$\pi = \mathbf{n} + d e_\infty$	$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge e_\infty$
Circle	$Z = S_1 \wedge S_2$	$Z^* = P_1 \wedge P_2 \wedge P_3$
Line	$L = \pi_1 \wedge \pi_2$	$L^* = P_1 \wedge P_2 \wedge e_\infty$
Point pair	$Pp = S_1 \wedge S_2 \wedge S_3$	$Pp^* = P_1 \wedge P_2$

concept for the C++ AMP programming language. With the Gaalop Precompiler, we are able to enhance ordinary C++ AMP code with Geometric Algebra code and automatically generate optimized C++ AMP code.

A precompiler is a way of extending the features of a programming language. For Geometric Algebra Computing, it is of high interest to use both the power of high-performance languages and the elegance of expression of a domain-specific language such as CLUCalc [18]. The Gaalop language is inspired by CLUCalc. We embed this language into C++ AMP code, and compile it by utilizing the precompiler concept and the fast optimizations and code generation features of Gaalop.

The Gaalop Precompiler enhances C++ AMP programs by embedding

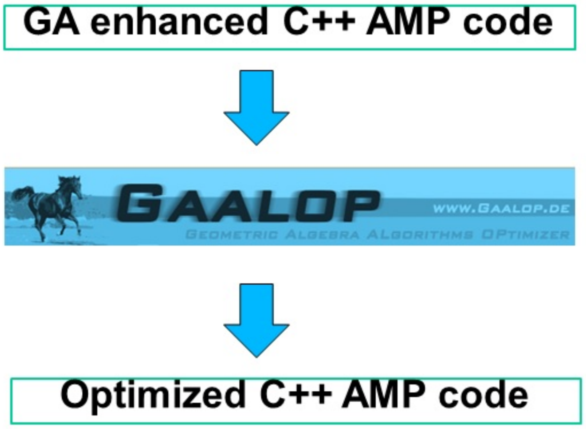

FIGURE 3. Gaalop Precompiler for C++ AMP.

- Geometric Algebra code using multivectors;
- functionality to interact with multivectors.

It transforms these enhanced C++ AMP programs to optimized C++ AMP programs without any explicit Geometric Algebra functionality. More precisely, the Gaalop Precompiler takes C++ AMP programs enhanced with pragmas and translates them to C++ AMP code enhanced with optimized C code which can be understood by the C++ AMP compiler. The embedding of Geometric Algebra code is done based on pragmas with the following structure:

```
#pragma gpc begin
  ...
  Import of multivectors
  ...
  #pragma clucalc begin
    ...
    Geometric Algebra code based on CLUCalc
    ...
  #pragma clucalc end
  ...
  Export of multivectors
  ...
#pragma gpc end
```

Each gpc (Gaalop Precompiler) block includes a clucalc block with the Geometric Algebra functionality. The functions to import/export multivectors are defined in Table 2. The purpose of these functions is the transformation between multivectors and the C++ AMP language concepts of **float** variables

Table 2. Gaalop Precompiler functions for constructing and accessing multivectors

coeff = mv_getbladecoeff(mv,blade);	Get the coefficient of blade **blade** of multivector **mv**.
array = mv_to_array(mv, blades,...);	Write the blades **blades ,...** of multivector **mv** to array **array**. Example **array = mv_to_array (mv,e1,e2,e3,e0, einf);**.
mv = mv_from_array(array,blades,..);	Construct multivector **mv** from array **array**

and arrays. **mv_get_bladecoeff()** is responsible for extracting a blade coefficient from a multivector, whereas **mv_to_array()** constructs an array from a multivector and **mv_from_array()** constructs a multivector from a C-like array.

8. The Raytracer Proof-of-Concept

Here, we present our raytracer application as a proof-of-concept for our Gaalop Precompiler for C++ AMP. Raytracing is a technique for generating a 2D image by tracing the path of rays from a camera through the pixels in an image plane and simulating the light effects at the intersection with objects of a 3D scene (see Fig. 4). Listing 1 describes the main routine of the raytracer. The ImageView object is a type of C++ AMP array_view: a 2-dimensional pixel array (with extention HEIGHT and WIDTH) of colours. With parallel_for_each we describe the raytracing functionality for each pixel. See [1] to download the source files of the proof-of-concept project.

Listing 1. The Scene C++ AMP main routine

```
void Scene::renderOnGPU(std::vector<Colour>& imageData,
                        Camera camera, Light light) {
array_view<Colour, 2> imageView(HEIGHT, WIDTH,
                        &imageData[0]);
array_view<Object, 1> allObjects(objectSize, objects);
int length = objectSize;
Color backgroundColor = background;
parallel_for_each(imageView.get_extent(),
                 [=] (index<2> idx) restrict(amp) {
  const auto y = idx[0]; // inverse order...
  const auto x = idx[1];
  // create a new ray...
```

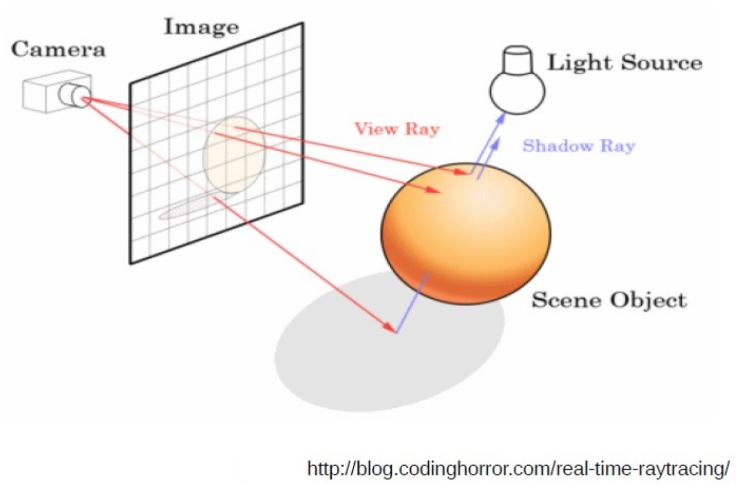

FIGURE 4. Raytracing principle.

```
    Ray ray = createNewRay(camera, x, y);
    imageView[idx] = rayCastAlgorithm(x, y, ray,
                allObjects, light, length, backgroundColor);
  });
}
```

Listing 2 describes the integration of some Geometric Algebra functionality into C++ AMP. It is written in the Gaalop scripting language which is inspired by CLUCalc [18].

In the first gpc block, a sphere S is computed and assigned to the array *sphere*. The predefined function VecN3() computes the conformal point of the Euclidean center point with the coordinates Cx, Cy, Cz (see the first row of Table 1). The sphere S is computed with corresponding radius *radius* according to the second row of Table 1.

LISTING 2. Computations with sphere and ray

```
#pragma gpc begin
#pragma clucalc begin
  ?S = VecN3(Cx, Cy, Cz) - 0.5*radius*radius*einf;
#pragma clucalc end
sphere = mv_to_array(S, e1, e2, e3, einf, e0);
#pragma gpc end
// Ray defined by origin (Ox, Oy, Oz)
// and direction (Lx, Ly, Lz)
#pragma gpc begin
#pragma clucalc begin
```

```
  O = VecN3(Ox, Oy, Oz);
  L = VecN3(Lx, Ly, Lz);
  ?Ray = *(O ^ L ^ einf);
#pragma clucalc end
  newRay.ray = mv_to_array(Ray, e1^e2, e1^e3,
                        e1^einf, e2^e3, e2^einf, e3^einf);
#pragma gpc end
```

The second gpc block computes the ray *Ray* and assigns its relevant coefficients to an array. The ray is computed based on the outer product of two of its points O and L and infinity (see the fifth row of Table 1). Note that the ray has to be dualized (with a leading asterisk) since the standard representation in Gaalop is the IPNS representation.

Listing 3. Intersection of line and sphere with the Gaalop Precompiler for C++ AMP

```
#pragma clucalc begin
?PP = Ray ^ Sphere;
?hasIntersection = PP.PP;
#pragma clucalc end
```

The listing 3 describes the integration of the Geometric Algebra functionality of the intersection of a sphere and a line into C++ AMP as well as the computation of an intersection indicator (see also Figure 2).

9. Computing Details

Here, we investigate some computing details where the high runtime performance and the good adaptability for parallel computing systems come from.

Sect. 3 presents some Geometric Algebra applications being faster than conventional implementations. Also the implementation of the just presented raytracer based on the Gaalop Precompiler for C++ AMP is faster than the conventional mathematics CPU implementation (see Sect. 10). Where do these results come from?

In general, what remains after the Gaalop optimization process via symbolic simplification are only long sums of products, which again can be efficiently parallelized (see Figure 5). Although not designed for computing with Geometric Algebra, most of the current computing devices fortunately support these kind of operations. Additionally, Gaalop has a specific control mechanism for the adaptability to different types of sequential and parallel computing systems: each line of Gaalop code can be indicated whether it should be evaluated explicitely or not. For sequential computing devices it may be better to evaluate all lines while parallel computing devices benefit from a small number of explicit line evaluations even if this can lead to some redundant parallel computations.

The structure of Gaalop results can be best seen based on the Geometric Algebra Parallelism Programs (GAPP). This intermediate language for Gaalop and its instruction set is defined in the book [13]. All implementations will be optimized according to the characteristics of the computing device.

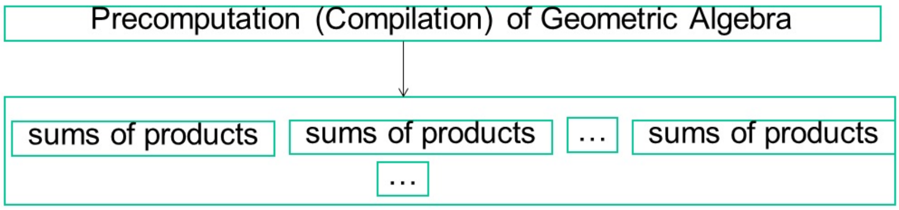

FIGURE 5. Precomputation of Geometric Algebra leads to parallel computations of sums of products

As follows, we investigate in more detail the resulting code of the Gaalop Precompiler based on the Listing 4.

LISTING 4. Gaalop code (inspired by CLUCalc) to be optimized
```
S =  VecN3(Cx, Cy, Cz) - 0.5*r*r*einf;
O = VecN3(Ox, Oy, Oz);
L = VecN3(Lx, Ly, Lz);
R = *(O ^ L ^ einf);
PP = R ^ S;
?hasIntersection = PP.PP;
```

This example computes the sphere S and the ray R through the points O and L (see Table 1, VecN3 computes a conformal point based on its 3D coordinates). Based on the intersection of the ray and the sphere, the intersection indicator *hasIntersection* is computed. Its sign indicates whether the ray and the sphere are really intersecting each other or not.

A question mark at the beginning of a line indicates a multivector variable that has to be explicitly computed by Gaalop. This means that Gaalop is able to optimize not only single statements, but a number of Geometric Algebra statements. In Listing 4, the expressions for S, R, O, L and PP are used only by Gaalop, in order to compute an optimized result for the intersection indicator (see the question mark in the last line of the listing).

Optimizing this listing results in the following C++ AMP code:

LISTING 5. Resulting C++ AMP code of Listing 4.
```
hasIntersection[0] = ((((((-(Oz * Oz)) + 2.0 * Lz * Oz)
- Oy * Oy + 2.0 * Ly * Oy) - Ox * Ox + 2.0 * Lx * Ox)
- Lz * Lz - Ly * Ly - Lx * Lx) * r * r
+ ((Ly * Ly - 2.0 * Cy * Ly + Lx * Lx) - 2.0 * Cx * Lx
+ Cy * Cy + Cx * Cx) * Oz * Oz
+ ((((((2.0 * Cy - 2.0 * Ly) * Lz
+ 2.0 * Cz * Ly) - 2.0 * Cy * Cz) * Oy
+ (((2.0 * Cx - 2.0 * Lx) * Lz + 2.0 * Cz * Lx)
- 2.0 * Cx * Cz) * Ox
+ ((2.0 * Cy * Ly + 2.0 * Cx * Lx)
- 2.0 * Cy * Cy - 2.0 * Cx * Cx) * Lz)
- 2.0 * Cz * Ly * Ly
+ 2.0 * Cy * Cz * Ly) - 2.0 * Cz * Lx * Lx
+ 2.0 * Cx * Cz * Lx)
 * Oz + ((Lz * Lz - 2.0 * Cz * Lz + Lx * Lx)
- 2.0 * Cx * Lx
+ Cz * Cz + Cx * Cx) * Oy * Oy
+ (((((2.0 * Cx - 2.0 * Lx) * Ly
+ 2.0 * Cy * Lx) - 2.0 * Cx * Cy) * Ox
- 2.0 * Cy * Lz * Lz
+ (2.0 * Cz * Ly + 2.0 * Cy * Cz) * Lz
+ (2.0 * Cx * Lx - 2.0 * Cz * Cz - 2.0 * Cx * Cx) * Ly)
- 2.0 * Cy * Lx * Lx + 2.0 * Cx * Cy * Lx) * Oy
+ ((Lz * Lz - 2.0 * Cz * Lz + Ly * Ly)
- 2.0 * Cy * Ly + Cz * Cz + Cy * Cy) * Ox * Ox
+ (((-(2.0 * Cx * Lz * Lz))
+ (2.0 * Cz * Lx + 2.0 * Cx * Cz) * Lz)
- 2.0 * Cx * Ly * Ly + (2.0 * Cy * Lx
+ 2.0 * Cx * Cy) * Ly + ((-(2.0 * Cz * Cz))
- 2.0 * Cy * Cy) * Lx) * Ox
+ (Cy * Cy + Cx * Cx) * Lz * Lz
+ ((-(2.0 * Cy * Cz * Ly)) - 2.0 * Cx * Cz * Lx) * Lz
+ (Cz * Cz + Cx * Cx) * Ly * Ly)
- 2.0 * Cx * Cy * Lx * Ly
+ (Cz * Cz + Cy * Cy) * Lx * Lx; // 1.0
```

What we realize is, that

- only one entry of the 32-dimensional multivector of the intersection indicator is computed. Gaalop recognizes that only the scalar part with index 0 (see Table 3) has to be computed and all the other 31 entries are zero.
- the expression is dependent on the 10 entry values for the three 3D-points (Cx, Cy, Cz), (Ox, Oy, Oz), (Lx, Ly, Lz) and the radius r.

The structure of this C code can be seen based on the GAPP code presented in Listing 6.

Listing 6. Resulting GAPP code of Listing 4.

```
assignInputsVector inputsVector
    = [Cx,Cy,Cz,Lx,Ly,Lz,Ox,Oy,Oz,r];
```

```
resetMv hasIntersection[32];

setVector ve0 = {inputsVector[-8],2.0,inputsVector[-7],
 2.0, inputsVector[-6],2.0,inputsVector[-5,-4,-3,4],
 -2.0,inputsVector[3],-2.0,inputsVector[1,0],-2.0,2.0,
 2.0,-2.0,-2.0,2.0,2.0,-2.0,2.0,2.0,-2.0,-2.0,-2.0,2.0,
 -2.0,2.0,inputsVector[5],-2.0,inputsVector[3],-2.0,
 inputsVector[2,0],-2.0,2.0,2.0,-2.0,-2.0,2.0,2.0,2.0,
 -2.0,-2.0,-2.0,2.0,inputsVector[5],-2.0,inputsVector[4],
 -2.0,inputsVector[2,1],-2.0,2.0,2.0,-2.0,2.0, 2.0,-2.0,
 -2.0,inputsVector[1,0],-2.0,-2.0,inputsVector[2,0],-2.0,
 inputsVector[2,1]};

setVector ve1 = {inputsVector[8,5,7,4,6,3,5,4,3,4,1,3,0,
 1,0,4,1,2,1,3,0,2,0,1,0,1,0,2,1,2,0,5,2,3,0,2,0,3,0,1,0,
 1,2,1,0,2,0,1,0,5,2,4,1,2,1,0,2,0,0,1,0,2,1,1,0,1,0,2,0,
 0,2,1]};

setVector ve2 = {inputsVector[9,8,9,7,9,6,9,9,9,8,4,8,3,
 8,8,5,5,4,2,5,5,3,2,4,3,1,0,4,2,3,2,7,5,7,3,7,7,4,4,3,1,
 5,4,2,3,2,0,3,1,6,5,6,4,6,6,5,3,2,4,3,1,2,1,5,5,2,2,4,
 4,1,3,3]};

setVector ve3 = {inputsVector[9,9,9,9,9,9,9,9,9,8,8,8,8,
 8,8,7,7,7,7,6,6,6,6,5,5,5,5,4,4,3,3,7,7,7,7,7,7,6,6,6,6,
 5,5,5,4,4,4,3,3,6,6,6,6,6,6,5,5,5,4,4,4,3,3,5,5,4,3,
 4,4,3,3,3]};

setVector ve4 = {1.0,inputsVector[9],1.0,
 inputsVector[9],1.0,inputsVector[9],1.0,1.0,1.0,1.0,
 inputsVector[8],1.0,inputsVector[8],1.0,1.0,
 inputsVector[8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8],1.0,
 inputsVector[7],1.0,inputsVector[7],1.0,1.0,
 inputsVector[7,7,7,7,7,7,7,7,7,7,7,7],1.0,
 inputsVector[6],1.0,inputsVector[6],1.0,1.0,
 inputsVector[6,6,6,6,6,6,6,6],1.0,1.0,
 inputsVector[5,5],1.0,1.0,inputsVector[4],1.0,1.0};

dotVectors hasIntersection[0] = <ve0,ve1,ve2,ve3,ve4>;
```

First of all, the 10 entry values for the three 3D-points (Cx, Cy, Cz), (Ox, Oy, Oz), (Lx, Ly, Lz) and the radius r are assigned to the InputsVector.

The result of Listing 4 is the multivector *hasIntersection*, which has to be reset first. If we indicate multivectors with 32 floats means, that this is only the worst case while we internally store only the non-zero coefficients. In this case this is only the coefficient $hasIntersection[0]$.

Next, five vectors (ve1 .. ve4) are set based on entries of the InputsVector as well as of some constants. Counting their entries, we realize, that all these vectors are 72-dimensional. Finally, the dot product of the given vectors is calculated and assigned to the entry 0 of the multivector *hasIntersection*. There is an implicit parallelism in this computation according to Fig. 6. The multiplications of each of the vector coefficients can be done in parallel as well as parts of the additions.

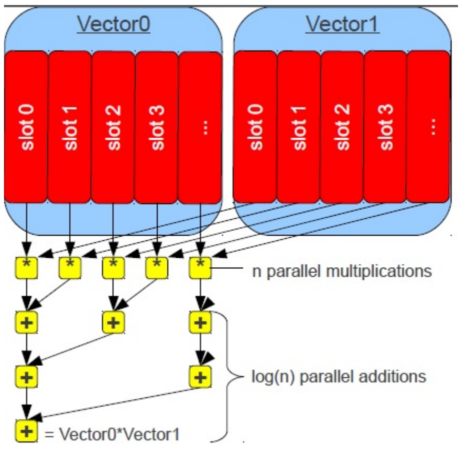

FIGURE 6. Parallel dot product of two n-dimensional vectors Vector0 and Vector1 (n parallel products followed by $\log(n)$ parallel addition steps).

The vectors for the computations of Listing 6 are very long. Let us now look at a second version according to the Gaalop Listing 7. There is an additional question mark, indicating that also the ray R has to be computed explicitly.

LISTING 7. Gaalop code to be optimized (second version)

```
S =  VecN3(Cx, Cy, Cz) - 0.5*r*r*einf;
O = VecN3(Ox, Oy, Oz);
L = VecN3(Lx, Ly, Lz);
?R = *(O ^ L ^ einf);
PP = R ^ S;
?hasIntersection = PP.PP;
```

Looking at the resulting C++ AMP code, we realize that it is more compact. We have to compute six coefficients of the multivector R with the indices 6, 7, 8, 10, 11 and 13. The meaning of each coefficient in terms of its blade meaning is indicated in the comments at the end of each line. Here, we recognize an implicit parallelism of GAPP, since all the coefficients of a multivector can be computed in parallel.

Listing 8. Resulting C++ AMP code of Listing 7.

```
R[6] = Oz - Lz; // e1 ^ e2
R[7] = Ly - Oy; // e1 ^ e3
R[8] = Ly * Oz - Lz * Oy; // e1 ^ einf
R[10] = Ox - Lx; // e2 ^ e3
R[11] = Lz * Ox - Lx * Oz; // e2 ^ einf
R[13] = Lx * Oy - Ly * Ox; // e3 ^ einf
hasIntersection[0] = (R[8] * R[8]
+ ((-(2.0 * Cz * R[7])) - 2.0 * Cy * R[6]) * R[8]
+ ((-(r * r)) + Cz * Cz + Cx * Cx) * R[7] * R[7]
+ (2.0 * Cy * Cz * R[6] + 2.0 * Cx * R[13]
+ 2.0 * Cx * Cy * R[10]) * R[7] + ((-(r * r))
+ Cy * Cy + Cx * Cx) * R[6] * R[6] + (2.0 * Cx * R[11]
- 2.0 * Cx * Cz * R[10]) * R[6] + R[13] * R[13]
+ 2.0 * Cy * R[10] * R[13] + R[11] * R[11])
- 2.0 * Cz * R[10] * R[11] + ((-(r * r))
+ Cz * Cz + Cy * Cy) * R[10] * R[10]; // 1.0
```

The computation for the intersection indicator *hasIntersection* now is shorter, since precomputed values of the multivector R can be re-used. Counting the length of the vectors ve9 .. ve13 for its computations in the GAPP Listing 9, results in a number of 21.

Listing 9. Resulting GAPP code of Listing 7.

```
//R[6] = inputsVector[8] - inputsVector[5]
assignInputsVector inputsVector
 = [Cx,Cy,Cz,Lx,Ly,Lz,Ox,Oy,Oz,r];
resetMv R[32];
setVector ve0 = {inputsVector[8,-5]};
dotVectors R[6] = <ve0>;

//R[7] = inputsVector[4] - inputsVector[7]
setVector ve1 = {inputsVector[4,-7]};
dotVectors R[7] = <ve1>;

//R[8] = (inputsVector[4] * inputsVector[8])
//      - (inputsVector[5] * inputsVector[7])
setVector ve2 = {inputsVector[4,-5]};
setVector ve3 = {inputsVector[8,7]};
dotVectors R[8] = <ve2,ve3>;
```

```
//R[10] = inputsVector[6] - inputsVector[3]
setVector ve4 = {inputsVector[6,-3]};
dotVectors R[10] = <ve4>;

//R[11] = (inputsVector[5] * inputsVector[6])
//       - (inputsVector[3] * inputsVector[8])
setVector ve5 = {inputsVector[5,-3]};
setVector ve6 = {inputsVector[6,8]};
dotVectors R[11] = <ve5,ve6>;

//R[13] = (inputsVector[3] * inputsVector[7])
//       - (inputsVector[4] * inputsVector[6])
setVector ve7 = {inputsVector[3,-4]};
setVector ve8 = {inputsVector[7,6]};
dotVectors R[13] = <ve7,ve8>;

resetMv hasIntersection[32];

setVector ve9 = {R[8],-2.0,-2.0,inputsVector[-9,2,0],
  2.0,2.0,2.0,inputsVector[-9,1,0],2.0,-2.0,R[13],2.0,
 R[11],-2.0,inputsVector[-9,2,1]};

setVector ve10 = {R[8],inputsVector[2,1,9,2,0,1,
  0,0,9,1,0,0,0],R[13],inputsVector[1],R[11],
  inputsVector[2,9,2,1]};

setVector ve11 = {1.0,R[7,6,7,7,7],inputsVector[2],
 R[13],inputsVector[1],R[6,6,6,11],inputsVector[2],
  1.0,R[10],1.0,R[10,10,10,10]};

setVector ve12 = {1.0,R[8,8,7,7,7,6,7,10,6,6,6,6,10],
  1.0,R[13],1.0,R[11,10,10,10]};

setVector ve13 = {1.0,1.0,1.0,1.0,1.0,1.0,R[7],1.0,
 R[7],1.0,1.0,1.0,1.0,R[6],1.0,1.0,1.0,1.0,1.0,1.0,
  1.0};

dotVectors hasIntersection[0]
   = <ve9,ve10,ve11,ve12,ve13>;
```

But, what if we are interested in the point pair because we need it for further computation? Adding a question mark before the computation of the point pair instead of the ray, results in the following Listing 10.

Listing 10. Resulting GAPP code including the computation of the point pair.

```
assignInputsVector inputsVector
  = [Cx,Cy,Cz,Lx,Ly,Lz,Ox,Oy,Oz,r];
resetMv PP[32];
setVector ve0 = {inputsVector[2,1,0,-2,-1,-0]};
setVector ve1 = {inputsVector[8,7,6,5,4,3]};
dotVectors PP[16] = <ve0,ve1>;

setVector ve3 = {inputsVector[8,5,4,3,2,1,0,5,5,2,1,0]};
setVector ve4 = {inputsVector[9,9,8,8,2,1,0,7,6,2,1,0]};
setVector ve5 = {inputsVector[9,9],1.0,1.0,
 inputsVector[8,8,8],1.0,1.0,inputsVector[5,5,5]};
dotVectors PP[17] = <ve2,ve3,ve4,ve5>;

setVector ve6 = {inputsVector[8,-5]};
dotVectors PP[18] = <ve6>;

setVector ve7 = {0.5,-0.5,inputsVector[-2,2,0],
 -0.5,-0.5,-0.5,  inputsVector[-0],0.5,0.5,0.5};
setVector ve8 = {inputsVector[7,4,4,5,3,2,1,0,4,2,1,0]};
setVector ve9 = {inputsVector[9,9,8,7,7,2,1,0,6,2,1,0]};
setVector ve10 = {inputsVector[9,9],1.0,1.0,1.0,
  inputsVector[7,7,7],1.0,inputsVector[4,4,4]};
dotVectors PP[19] = <ve7,ve8,ve9,ve10>;

setVector ve11 = {inputsVector[4,-7]};
dotVectors PP[20] = <ve11>;

setVector ve12 = {inputsVector[4,-5]};
setVector ve13 = {inputsVector[8,7]};
dotVectors PP[21] = <ve12,ve13>;

setVector ve14 = {-0.5,0.5,inputsVector[2,1,-2,-1],
 0.5,0.5,0.5,-0.5,-0.5,-0.5};
setVector ve15 = {inputsVector[6,3,3,3,5,4,2,1,0,2,1,0]};
setVector ve16 = {inputsVector[9,9,8,7,6,6,2,1,0,2,1,0]};
setVector ve17 = {inputsVector[9,9],1.0,1.0,1.0,1.0,
  inputsVector[6,6,6,3,3,3]};
dotVectors PP[22] = <ve14,ve15,ve16,ve17>;

setVector ve18 = {inputsVector[6,-3]};
dotVectors PP[23] = <ve18>;

setVector ve19 = {inputsVector[5,-3]};
setVector ve20 = {inputsVector[6,8]};
dotVectors PP[24] = <ve19,ve20>;

setVector ve21 = {inputsVector[3,-4]};
setVector ve22 = {inputsVector[7,6]};
dotVectors PP[25] = <ve21,ve22>;

resetMv hasIntersection[32];
setVector ve23 = {PP[25,24],2.0,PP[21],2.0,2.0,PP[-16]};
setVector ve24 = {PP[25,24,22,21,19,17,16]};
setVector ve25 = {1.0,1.0,PP[23],1.0,PP[20,18],1.0};
dotVectors hasIntersection[0] = <ve23,ve24,ve25>;
```

Now, we have to compute 10 coefficients of the multivector PP with the indices 16, 17, 18, 19, 20, 21, 22, 23, 24 and 25 (which can be done in parallel). The computations of each coefficient vary from the computations with one vector (means sum-up of the entries of the vector) up to four vectors with lengths between two and twelve. The computation for the intersection indicator *hasIntersection* now is again shorter consisting only of the dot product of three 7-dim. vectors.

10. Results

The main contribution of this paper is to make it possible to use Geometric Algebra for all the heterogeneous computing systems of the HSA Foundation. How easy it is to integrate Geometric Algebra into C++ AMP programs is shown in Sect. 8. But, what about the runtime-performance? Sect. 9 shows some computing details where the good runtime-performance results of Gaalop come from. Can we achieve as good results for heterogeneous systems as with the applications in Sect. 3?

The setup of our first performance tests was a PC with AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ 2 * 2,2 GHz with 4 GB of RAM, an AMD Radeon HD 5450 GPU with 1GB and the operating system Ubuntu 14.04 LTS (64-Bit). The rendering time for the raytracer using linear algebra on the CPU was 5,9714 sec. Using Geometric Algebra on the GPU, the runtime was 5,2469 sec, means the Geometric Algebra implementation on the GPU is about 12% faster than the CPU implementation using standard mathematics. See Sect. 11 for some ideas how to further improve these results.

11. Conclusion and Future Work

Starting from the situation 15 years ago with a great gap between the low symbolic complexity on the one hand and the high numeric complexity of coding in GA on the other hand, this paper shows, that, in the meantime, this gap could be closed, especially for CPUs. Sect 3 presented some competitive Gaalop applications in the sense of being faster than conventional solutions. Also the raytracer proof-of-concept of this article using the Gaalop Precompiler for C++ AMP is faster than the CPU implementation using standard mathematics. There is still some research needed in order to achieve a better result for the GPU solution, since normally there is a better speedup using GPUs. Nevertheless very important is the main contribution of this paper: the Gaalop Precompiler for C++ AMP is the first Geometric Algebra tool supporting a broad range of heterogenous systems, since it is able to support

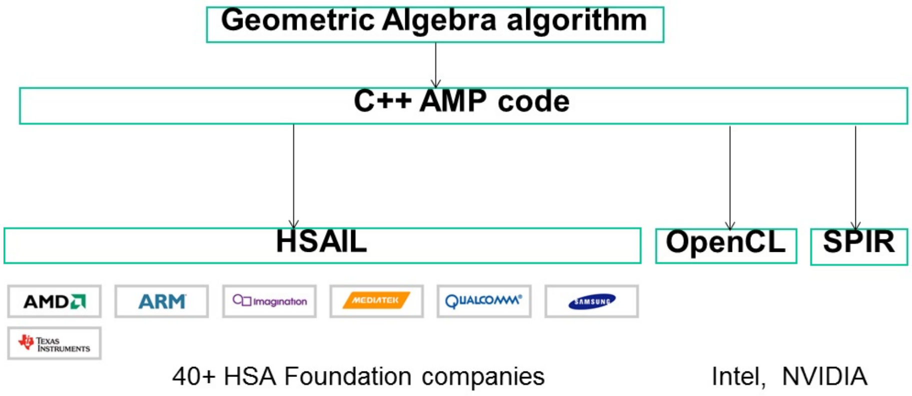

FIGURE 7. Solutions of the companies of the HSA foundation as well as of Intel and NVIDIA can be supported by the Gaalop Precompiler for C++ AMP.

the solutions of the companies of the HSA foundation via the HSAIL output format of their C++ AMP compiler. Since this compiler also supports OpenCL and SPIR, also Intel and NVIDIA solutions are supported. Since Gaalop Precompilers are also available for OpenCL and CUDA, there is also a direct way to support Intel and NVIDIA as diagrammed in Figure 8.

But, even if the current heterogenous systems can be supported by Geometric Algebra, there is still some potential for even better solutions, especially concerning runtime performance. One idea for improving the runtime performance of Geometric Algebra applications on heterogeneous systems is

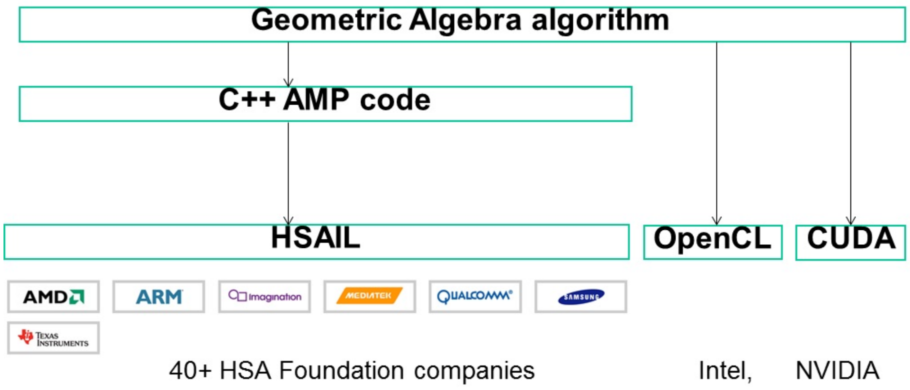

Figure 8. With the Gaalop Precompilers for OpenCL and CUDA, Intel and NVIDIA solutions can be supported directly.

to directly generate optimized HSAIL code from Geometric Algebra algorithms. Another idea with the potential of providing the most benefit for Geometric Algebra Computing applications is a GAPP co-processor directly executing GAPP code in hardware.

TABLE 3. The 32 blades of 5D Conformal Geometric Algebra that compose a multivector. The entry in the first column is the index of the corresponding blade. The negated entries are needed for the selectors of the Geometric Algebra Parallelism Programs (GAPP) indicating that the negated value should be used.

Index	**Negative index**	**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

References

[1] Justin Albert. The download page of the proof-of-concept raytracer based on the gaalop precompiler for C++ AMP. Available at https://bitbucket.org/justinator_92/geometric-algebra-raytracer/wiki/Home, 2015.

[2] Pablo Colapinto. The versor home page. Available at http://versor.mat.ucsb.edu/, 2015.

[3] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry.* Morgan Kaufmann, 2007.

[4] Ahmad Hosney Awad Eid. *Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application.* PhD thesis, Suez Canal University, Port Said, 2010.

[5] Daniel Fontijne, Tim Bouma, and Leo Dorst. Gaigen 2: A geometric algebra implementation generator. Available at `http://staff.science.uva.nl/~fontijne/gaigen2.html`, 2007.

[6] HSA Foundation. The hsa foundation home page. Available at `http://www.hsafoundation.com/`, 2015.

[7] 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.

[8] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. ConformalALU: a conformal geometric algebra coprocessor for medical image processing. In *IEEE Transactions on Computers*, 2015.

[9] Antonio Gentile, Salvatore Segreto, Filippo Sorbello, Giorgio Vassallo, Salvatore Vitabile, and Vincenzo Vullo. Cliffosor, an innovative FPGA-based architecture for geometric algebra. In *ERSA 2005*, pages 211–217, 2005.

[10] Kate Gregory and Ade Miller. *C++AMP.* Microsoft Press, 2012.

[11] David Hestenes. Old wine in new bottles: A new algebraic framework for computational geometry. In Eduardo Bayro-Corrochano and Garret Sobczyk, editors, *Geometric Algebra with Applications in Science and Engineering.* Birkhäuser, 2001.

[12] Dietmar Hildenbrand. *Geometric Computing in Computer Graphics and Robotics using Conformal Geometric Algebra.* PhD thesis, TU Darmstadt, 2006. Darmstadt University of Technology.

[13] Dietmar Hildenbrand. *Foundations of Geometric Algebra Computing.* Springer, 2013.

[14] Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Joachim Pitt. Gaalop home page. Available at `http://www.gaalop.de`, 2015.

[15] 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.

[16] Hongbo Li, David Hestenes, and Alyn Rockwood. Generalized homogeneous coordinates for computational geometry. In G. Sommer, editor, *Geometric Computing with Clifford Algebra*, pages 27–59. Springer, 2001.

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

[18] Christian Perwass. The CLU home page. Available at http://www.clucalc.info, 2010.

[19] Christian Perwass, Christian Gebken, and Gerald Sommer. Implementation of a Clifford algebra co-processor design on a field programmable gate array. In Rafal Ablamowicz, editor, *Clifford Algebras: Application to Mathematics, Physics, and Engineering*, Progress in Mathematical Physics, pages 561–575. 6th International Conference on Clifford Algebras and Applications, Cookeville, TN., Birkhäuser, 2003.

[20] Florian Seybold. Gaalet – a C++ expression template library for implementing geometric algebra, 2010.

[21] Gerald Sommer, editor. *Geometric Computing with Clifford Algebra*. Springer, 2001.

[22] Florian Stock, Dietmar Hildenbrand, and Andreas Koch. Fpga-accelerated color edge detection using a geometric-algebra-to-verilog compiler. In *Symposium on System on Chip (SoC), Tampere, Finland*, 2013.

[23] Florian Woersdoerfer, Florian Stock, Eduardo Bayro-Corrochano, and Dietmar Hildenbrand. Optimization and performance of a robotics grasping algorithm described in geometric algebra. In *Iberoamerican Congress on Pattern Recognition 2009, Guadalajara, Mexico*, 2009.

D. Hildenbrand
Hochschule RheinMain, Ruesselsheim,
Germany
e-mail: `dietmar.hildenbrand@gmail.com`

J. Albert
Hochschule RheinMain, Wiesbaden,
Germany
e-mail: `justin_albert@gmx.de`

P. Charrier
University of Technology, Darmstadt,
Germany
e-mail: `charrier@gsc.tu-darmstadt.de`

Chr. Steinmetz
University of Technology, Darmstadt,
Germany
e-mail: `steinmetz@dik.tu-darmstadt.de`