Geometric Algebra Computing Technology for Accelerated Processing Units

Patrick Charrier and Dietmar Hildenbrand

January 21, 2013

1 Motivation

Development on embedded devices, even on today's hardware, limits us to a minimum of third party-library dependencies due to hardware memory and power restrictions. In setups requiring intense geometric operations on limited hardware, such as in robotics, this problem can often lead to a tedious reimplementation of matrix, vector, and quaternion operations. Furthermore, certain unnecessary floating point operations are hard to avoid, because C++-features like expression template libraries such as eigen [2] can possibly not be used, because of strict C enforcement. Memory accesses are often the most limiting factor in today's applications due to high memory latency. Yet traditional programming techniques unfortunately steer into the wrong direction by not easing data-oriented programming, which is often cumbersome to implement in C or C++.

Many of the restrictions above are in a similar form the case on modern heterogeneous architectures such as AMD's embedded Accelerated Processing Units or in GPGPU written in OpenCL/CUDA. Our technology based on Geometric Algebra and a Domain Specific language called CLUCalc will especially excel under these conditions.

The focus of this work is Gaalop Precompiler, a new technology combining the advanced processing power of Accelerated Processing Units (APU) with the geometric intuitiveness of a new mathematical concept named Geometric Algebra [6]. The combination of both not only promises a more compact and maintainable code for graphics, vision, robotics and other scientific and engineering applications, but also automatically exploits parallelism on GPU or combined computing unit (APU) through OpenCL [8] or CUDA [9]. C/C++ CPU targeting is also supported. It is presented in the following, after a short introduction on Geometric Algebra.

2 Advantages of Geometric Algebra Computing Technology

Our technology has the following advantages in geometry-related fields compared to conventional approaches:

- A Domain Specific language for Geometric Algebra
  - Increased geometric intuitivity.
  - Compactness of algorithms.
- Easy Integration into standard programming languages.
  - Full C/C++/OpenCL/CUDA compatibility.
  - May be easily integrated into your toolchain through provided CMake build scripts.
  - Does not require linking to external libraries.
  - Completely free and downloadable from `www.gaalop.de`.
- Better Runtime Performance
  - Eases data-oriented programming and coalescing for maximum cache performance and memory throughput through built-in language features. Especially important for GPGPU computing.
  - Autovectorization features for the OpenCL target language.
    Full support for SSEx and AVXx, and the next-generation Intel MIC instruction sets with vector widths of up to 512 bits (float16 type).

3 Geometric Algebra by example

As a motivation towards Geometric Algebra (GA) we would like to introduce the following aspect of a Raytracer, which calculates the intersection of a sphere (S) and a ray (R).

$$R = S \wedge P$$

As we know from Linear Algebra, the intersection of a line and a sphere conforms to one of three cases, namely no intersection, a point intersection, or a point-pair intersection. In Geometric Algebra, all these cases are contained in the statement above, yielding a higher level of intuitiveness and compactness. Similar observations can be made in other applications of geometry related mathematics. Applied to computer programs, GA therefore has a high potential for improving code readability and to shorten production cycles. In our experience and using our compiler technology, Geometric Algebra often offers better performance than conventional approaches [6].

4 Gaalop Precompiler

Gaalop Precompiler (Gaalop GPC) [4] makes it possible to directly embed Geometric Algebra Computing code into C,C++,CUDA, or OpenCL code. The complete build process is automated. The tool will be further explained through an example in the next section.

5 Test case: Collision Detection for Computational Cloth Simulation

The following section describes a Collision Detection application as a test case for Gaalop GPC. A cloth in computer graphics might be simulated using a large number of triangles, making the connection between many points (vertices) in three-dimensional space. All these triangles may collide with other triangles on the same cloth, which is called self-collision, with other cloth, or even rigid bodies. Theoretically, to check for collisions between the triangles, we must assume, that they are all potential colliders and must therefore check for collision between every triangle and all other triangles and with every other object. Without any further information, this test would not be computationally manageable for larger scenes. The common method used to solve this problem are primarily hierarchical methods to break down the number of tests. Such a method might for example be a hierarchy of spheres that can be traversed with the following rules.

1. Perform a sphere-sphere test, starting with the root level.
2. If we have no collision, then stop the test.
   If we do have a collision, then recursively traverse the underlying spheres (broad phase testing).
3. In case we hit the leafs, perform tests on the underlying geometry, namely triangles. Those tests come down to two individual cases (narrow phase testing).
   (a) A point versus triangle test. See Figure 1.
   (b) An edge versus edge test.

So far, some experimental methods based on the so called Conformal Geometric Algebra (CGA) were tried on the narrow phase tests. We strongly believe that CGA will come to its fullest potential applied within the broad phase tests and will investigate this aspect in further research.

5.1 Point Triangle Test

The code snippet in Listing 1 performs a test for a collision between a triangle t and a point p, as described in subtest 3a above. The triangle has a thickness h

and is actually a prism mathematically, but for convenience, we call it 'triangle'. Figure 1 illustrates this aspect.

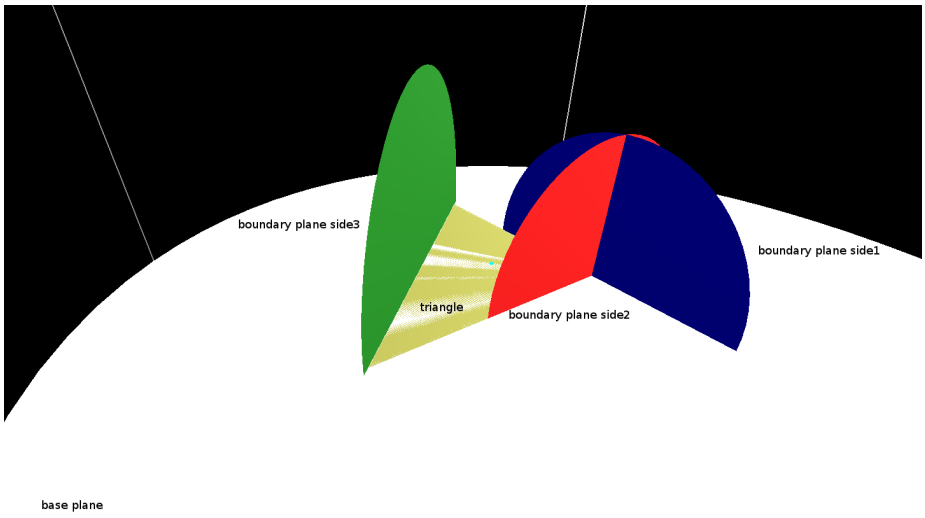

Figure 1: Point Triangle intersection visualized in CLUCalc. The picture shows the triangle, the plane it is embedded in, and the three planes that define its edges.

As an example for programming with Gaalop Precompiler in OpenCL kernels now consider the following code, performing the point triangle test explained above:

```
__kernel void pointTriangleTest(
        __global bool *collision,
        __global const float *triangles,
        __global const float *point,
        float h,
        unsigned int stride,
        unsigned int numTriangles)
{
        const size_t id = get_global_id(0);

        const float t1x = triangles[0 * stride + id];
        const float t1y = triangles[1 * stride + id];
        const float t1z = triangles[2 * stride + id];
        const float t2x = triangles[3 * stride + id];
        const float t2y = triangles[4 * stride + id];
        const float t2z = triangles[5 * stride + id];
        const float t3x = triangles[6 * stride + id];
        const float t3y = triangles[7 * stride + id];
```

```
        const float t3z = triangles[8 * stride + id];
        const float px = point[0];
        const float py = point[1];
        const float pz = point[2];

#pragma gpc begin
#pragma clucalc begin
        // triangle properties
        TrianglePoint1 = VecN3(t1x,t1y,t1z);
        TrianglePoint2 = VecN3(t2x,t2y,t2z);
        TrianglePoint3 = VecN3(t3x,t3y,t3z);

        // point properties
        TestPoint = VecN3(px,py,pz);

        // construct plane
        plane = *(TrianglePoint1 ^ TrianglePoint2 ^
            TrianglePoint3 ^ einf);

        // compute distance to plane
        ?d = plane . TestPoint;

        // extract triangle normal
        ?normal_ = plane - (plane . e0) ^ einf;

        // construct boundary planes
        side1 = *(TrianglePoint1 ^ TrianglePoint2 ^
            normal_ ^ einf);
        side2 = *(TrianglePoint2 ^ TrianglePoint3 ^
            normal_ ^ einf);
        side3 = *(TrianglePoint3 ^ TrianglePoint1 ^
            normal_ ^ einf);

        // compute distances
        ?d1 = side1 . TestPoint;
        ?d2 = side2 . TestPoint;
        ?d3 = side3 . TestPoint;
#pragma clucalc end

        const float d_  = mv_get_bladecoeff(d,1);
        const float d1_ = mv_get_bladecoeff(d1,1);
        const float d2_ = mv_get_bladecoeff(d2,1);
        const float d3_ = mv_get_bladecoeff(d3,1);
        if(d_*d_ > h*h ||
                d1_ <= 0.0f && d2_ <= 0.0f && d3_ <= 0.0f
                    ||
```

```
                d1_ >= 0.0f && d2_ >= 0.0f && d3_ >= 0.0f
                    )
                collision[id] = true;
        else
                collision[id] = false;
#pragma gpc end
}
```

Listing 1: A simple test, checking the collision of a 'triangle' t with thickness h with a point p, written for Gaalop GPC for OpenCL.

#pragma clucalc begin and **#pragma** clucalc end mark the boundaries between two languages, here namely OpenCL and a Domain Specific language called CLUCalc [10], specifically made to express Geometric Algebra and commonly used in the field. Another language layer embedded inside **#pragma** gpc begin and **#pragma** gpc end adds so called Interface Functions (see Subsection 5.3) to transform between CLUCalc and host language. This layer is encapsuling the CLUCalc layer.

An entity in Geometric Algebra is called multivector and is not directly compatible with normal C/C++/OpenCL language types like scalars, arrays or structures. The reader may refer to the references and [6] for a deeper background on GA.

An intelligent transformation/generation precompilation process handles the transformation between CLUCalc to C/C++/OpenCL and vice-versa based on the information provided through the Interface Functions. This may be integrated with any toolchain of your choice (like gcc, or Microsoft Visual C++). Automatic generation of toolchain project files using CMake [1] is supported through a CMake module.

5.2 The Code explained

The code executes the following steps:

1. Define the three triangle points and the test point.
2. Construct the base plane out of the triangle points.
3. Compute the signed distance d between the constructed plane and the test point.
4. Compute the normal of the plane.
5. Using this normal and the triangle points, compute the boundary planes, e.g. the planes that are perpendicular to the base plane and pass through every combination of the three points.
6. Compute the signed distances d_1, d_2 and d_3 between the test point and the three boundary planes.

7. Retrieve the scalar multivector-part from d_1, d_2 and d_3 and save them to **float**-constants d1_S, d2_S, and d3_S, by using the mv_get_bladecoeff() function.

8. The condition of a collision is satisfied, if d^2 is smaller than or equal the square of the triangle's thickness h, and all signed distances d_1, d_2 and d_3 share the same sign.

Note that raytracers in computer graphics use algorithms similar to this one. Recent work based on GA [3] shows promising results with speedups of up to four times.

5.3 Interface Functions

The example uses several **#pragma**-block-types. **#pragma** clucalc-blocks contain pure CLUCalc code expressing multivector computations. **#pragma** gpc-blocks primarily contain code, but not exclusively. Clearly the statement **const float** d1_S = mv_get_bladecoeff(d1,1); is both OpenCL and CLUCalc code, since neither mv_get_bladecoeff(), nor d1 are declared within the OpenCL parts of the function.

d1 is a multivector, which is internally reduced to a so called Compressed Multivector Storage-array containing only non-zero multivector blade coefficients. Specifically to ease handling with these arrays, the Interface Function mv_get_bladecoeff() is designed to lookup a particular multivector blade coefficient. We call it an Interface Function specifically, because it is not a real function in terms of C/C++/OpenCL. Much simplified, Gaalop GPC simply replaces mv_get_bladecoeff() by its corresponding array lookup d1[0], reducing the line to **const float** d1_SCALAR = d1[0];. An internal mechanism makes Gaalop GPC able to tell which blade belongs to which array index. This and other mechanisms are used to define more complicated Interface Functions, further detailed in [4].

The separation between **#pragma** gpc-block-contents and pure C/C++/OpenCL is mostly a design decision. By definition it would be possible to embed all C/C++/OpenCL code into one big **#pragma** gpc-block, or to even leave that concept out completely, but the intention is to use **#pragma** gpc-blocks as transformation layers. A secondary, but also well founded consideration is the fact that pure C/C++/OpenCL code is guaranteed to preserve relative line numbers throughout the precompiling process, because it is simply copied into the intermediate source-file in its original form. In most **#pragma** gpc -blocks, relative line numbers will almost certainly be altered by the process. Making the separation clear, it is possible to utilize the **#line** compiler directive for plain C/C++/OpenCL parts, which enables the native compiler to refer back to the original source-file in the case of errors and warnings. This yields a much better user-friendliness when working in an Integrated Development Environment (IDE). The user may simply click on the listed messages to get directed to the correct positions in the original source-file.

5.4 Code Generation

The following code is generated from the source-code in Listing 1. Note that this code will be automatically precompiled by the toolchain. Programmers do not have to deal with it normally, though they do have the option to look at it, it is merely shown for clarification:

```
__kernel void pointTriangleTest(
        __global bool* collision,
        __global const float* triangles,
        __global const float* point,
        const float h,
        const unsigned int stride,
        const unsigned int numTriangles)
{
        const size_t id = get_global_id(0);

        const float t1x = triangles[0 * stride + id];
        const float t1y = triangles[1 * stride + id];
        const float t1z = triangles[2 * stride + id];
        const float t2x = triangles[3 * stride + id];
        const float t2y = triangles[4 * stride + id];
        const float t2z = triangles[5 * stride + id];
        const float t3x = triangles[6 * stride + id];
        const float t3y = triangles[7 * stride + id];
        const float t3z = triangles[8 * stride + id];
        const float px = point[0];
        const float py = point[1];
        const float pz = point[2];

//#pragma gpc multivector d
float d;
//#pragma gpc multivector d1
float d1;
//#pragma gpc multivector d2
float d2;
//#pragma gpc multivector d3
float d3;
//#pragma gpc multivector normal_
float normal_[4];
//#pragma gpc multivector plane
float plane[4];
//#pragma gpc multivector side1
float side1[4];
//#pragma gpc multivector side2
float side2[4];
//#pragma gpc multivector side3
```

```
float side3[4];

//#pragma gpc multivector_component plane e1 plane[0]
plane[0] = (-(t1y * t2z + (-(t1z * t2y)) + (-((t1y + (-
    t2y)) * t3z)) + (t1z + (-t2z)) * t3y));
//#pragma gpc multivector_component plane e2 plane[1]
plane[1] = t1x * t2z + (-(t1z * t2x)) + (-((t1x + (-t2x))
     * t3z)) + (t1z + (-t2z)) * t3x;
//#pragma gpc multivector_component plane e3 plane[2]
plane[2] = (-(t1x * t2y + (-(t1y * t2x)) + (-((t1x + (-
    t2x)) * t3y)) + (t1y + (-t2y)) * t3x));
//#pragma gpc multivector_component plane einf plane[3]
plane[3] = (-((t1x * t2y + (-(t1y * t2x))) * t3z + (-((
    t1x * t2z + (-(t1z * t2x))) * t3y)) + (t1y * t2z + (-(
    t1z * t2y))) * t3x));
//#pragma gpc multivector_component d 1 d
d = plane[0] * px + plane[1] * py + plane[2] * pz + (-
    plane[3]);

... // code has been cut here

        const float d_ = d;
        const float d1_ = d1;
        const float d2_ = d2;
        const float d3_ = d3;
        if(d_*d_ > h*h ||
           d1_ <= 0.0f && d2_ <= 0.0f && d3_ <= 0.0f ||
           d1_ >= 0.0f && d2_ >= 0.0f && d3_ >= 0.0f)
                collision[id] = true;
        else
                collision[id] = false;
}
```

Listing 2: Code generated from the code in Listing 1

6 Streaming processor utilization

Gaalop GPC breaks Geometric Algebra code down into simple arithmetic expressions, as seen in Listing 1. These expressions are (typically) composed of a sum-of-products structure, which most compilers are able to exploit for an improved parallel efficiency.

The sum-of-products structure can be further utilized for generating SIMD-code directly, providing SIMD code even for compilers with weak autovectorization support. With an additional translation step called Geometric Algebra Parallelism Programs (GAPP) [7], which is essentially a further internal interme-

diate representation, the resulting code consists of instruction level-parallelism operations on vectors, yielding an even higher performance. In contrast to typical hands-on optimization approaches, the original code stays untouched. Complex optimization details performed by Gaalop GPC are not visible in the development code but only in the generated code, which improves the general maintainability in comparison to a hands-on-approach.

We will not provide further details on GAPP itself here, because it is a very extensive language aimed at a lot of platforms. However, the sum-of-products computation scheme is a key performance-concept of GAPP and Figure 2 pictures its OpenCL-backend implementation.

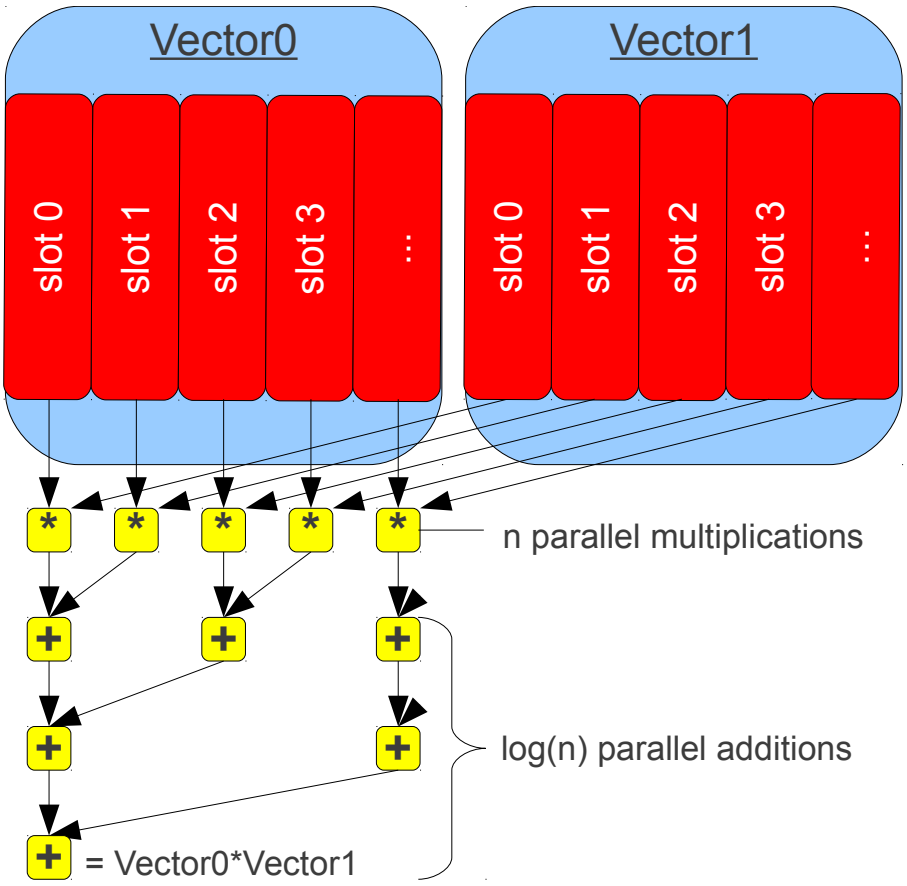

Figure 2: Sum-of-products computation scheme used in our Geometric Algebra Parallelism Programs OpenCL backend

As an example, please reconsider the computation of the variable d (distance of the test point to the base plane) from Listing 2. Its computation can be

further improved automatically by our compiler through GAPP if you choose to use it through the compiler CMake options:

```
//#pragma gpc multivector d
float d;
float16 ve0_0 = (float16)(t1x,-px,-t1y,py,px,-py,-t1x,px,
    t1z,-pz,-px,pz,t1y,-py,-t1z,pz);
float8 ve0_1 = (float8)(py,-pz,-px,py,px,-pz,-py,pz);
float16 ve1_0 = (float16)(t2y,t2y,t2x,t2x,t1y,t1x,t2z,t2z
    ,t2x,t2x,t1z,t1x,t2z,t2z,t2y,t2y);
float8 ve1_1 = (float8)(t1z,t1y,t1y,t1x,t1z,t1x,t1z,t1y);
float16 ve2_0 = (float16)(t3z,t3z,t3z,t3z,t3z,t3z,t3y,t3y
    ,t3y,t3y,t3y,t3y,t3x,t3x,t3x,t3x);
float8 ve2_1 = (float8)(t3x,t3x,t2z,t2z,t2y,t2y,t2x,t2x);

// 4 element-wise parallel multiplications
float16 dot0_0 = ve0_0 * ve1_0 * ve2_0;
float8 dot0_1 = ve0_1 * ve1_1 * ve2_1;

// parallel pyramid sum reduction (5 element-wise
    parallel adds)
float8 dot1_0 = dot0_0.lo + dot0_0.hi + dot0_1;
float4 dot2_0 = dot1_0.lo + dot1_0.hi;
float2 dot3_0 = dot2_0.lo + dot2_0.hi;
//#pragma gpc multivector_component d 1 d
d = dot3_0.lo + dot3_0.hi;
```

Listing 3: SIMD-optimized Code generated from Listing 1 through GAPP

Here, all multiplications are performed in parallel, given the target architecture supports the vector width. The pyramid sum reduction operation is not as efficient as the multiplication, but still provides a lot of parallelism.

Note that vectors with a size of more than 24 had to be split into subvectors of up to size 16 because OpenCL does not support larger vectors. That explains the subindices 0 and 1 for dot0_. It may be of interest that Intel's upcoming MIC architecture will actually support 512 bit-wide SIMD instructions, hence OpenCL vectors of size 16. With GAPP there is no theoretical limit to the size of vectors supported.

7 Coalesced memory access

We speak of a coalesced memory access if all neighboring work items (threads) in an OpenCL or CUDA compute unit access neighboring data and the accessed block as a whole is aligned to a certain number of bytes.

Coalescing enables devices to do efficient caching because the memory controller can answer all memory requests from all compute unit threads with one cache line. As it reads whole cache lines anyway, it can save multiple reads from

memory. As memory is the most limiting factor in today's hardware, coalescing can make a huge difference in terms of performance.

Gaalop Precompiler tries to ease coalescing by providing two Interface Functions defined in Table 7 especially focused on this aspect. The particular method used is called strided memory access and is a form of data-oriented programming. Gaalop GPC can handle most details of this internally.

mv = mv_from_stridedarray(array, index, stride , blades ,...) ;	Construct multivector mv from array array at index index with stride stride assigning to blade coefficients blades ,... . Example mv = mv_from_stridedarray(array ,0, nummvs,e1,e2,e3,e0,einf) ;.
array = mv_to_stridedarray(mv,index, stride , blades ,...) ;	Write the coefficients of blades blades ,... of multivector mv to array array at index index with stride stride . Example array = mv_to_stridedarray(mv ,0,nummvs,e1,e2,e3,e0,einf);.

Table 1: Gaalop GPC Interface Functions for coalesced (strided) memory access

Strided memory access lays out sets of structured data in a very different way than the conventional approach. For a set of n-dimensional vectors v_i stored in memory, the naive (traditional) approach would be to store one full vector, followed by the next one, and so forth. With data-oriented design in mind, it is better to store all v_{i_1} entries of all vectors i first, followed by all v_{i_2}, followed by all v_{i_3}, and so on, sequentially. The resulting speedup can be enormous depending on the size of the stride between accesses in the original implementation. Note that the set of n-dimensional vectors have been chosen as an academic example for structured data. Accesses on n-dimensional vectors will most likely be efficient if the whole vector is completely used for a computation in some form, like for example adding it to another vector. But if for example only the first vector entry is used, then all other entries will most likely be cached for no reason, resulting in many cache misses. For further information refer [5].

8 Gaalop GPC in practice: A Molecular Dynamics Simulation

A Molecular Dynamics Simulation [11] based on the presented technology was able to yield a much higher performance (see Figure 3) than a conventional approach based on Linear Algebra. In addition to the fact, that the proposed Geometric Algebra Computing Technology had about 80-times faster results than a conventional approach using the CPU and Linear Algebra, it also yields

better simulation results in terms of numerical stability. This might be due to the observation, that the generated code contains less floating point operations and therefore less potential sources for numerical errors.

Figure 3: Runtime performance of a Molecular Dynamics simulation featuring a Linear Algebra-based conventional solver on the CPU (red), a GA-based solver on the CPU (green), and a GA-based solver on the GPU (blue).

9 Conclusion

We have shown and reasoned the advantages described in Section 2 in more detail. With systems having an intense focus on instruction-level and SIMD-performance Geometric Algebra can make a difference. GA has been used in robotics [12] and many other fields mostly resulting in increased performance and sometimes better numerical stability. The technology does not require further libraries for linkage and greatly eases geometry-focused development.

Our tool provides a solid foundation to integrate GA into your C,C++,OpenCL or CUDA toolchain, making Geometric Algebra-based development as comfortable and fast as possible for embedded and non-embedded applications. Gaalop Precompiler may be downloaded freely from `www.gaalop.de`.

References

[1] Cmake - cross platform make. URL http://www.cmake.org/.

[2] Eigen. URL http://eigen.tuxfamily.org/index.php?title=Main_Page#Documentation.

[3] Michael Burger. Das effiziente raytracen von dreiecksnetzen auf mehrkernprozessoren, gpus und fpgas mittels geometrischer algebra. Master's thesis, TU Darmstadt, 2011.

[4] Patrick Charrier and Dietmar Hildenbrand. Gaalop precompiler. In *Special Issue AGACSE 2012*, Advances in Applied Clifford Algebras. Springer, 2012.

[5] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 3.0*, 2010. URL www.nvidia.com.

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

[7] Dietmar Hildenbrand, Patrick Charrier, Christian Steinmetz, and Andreas Koch. Specialized machine instruction set for geometric algebra. In *Special Issue AGACSE 2012*, Advances in Applied Clifford Algebras, 2012.

[8] Khronos-Group. The OpenCL home page, 2009. URL http://www.khronos.org/opencl/.

[9] NVIDIA. The CUDA home page, 2010. URL http://www.nvidia.com/object/cuda_home.html.

[10] Christian Perwass. The CLU home page, 2010. URL http://www.clucalc.info.

[11] Florian Seybold, Patrick Charrier, Dietmar Hildenbrand, M. Bernreuther, and D. Jenz. Runtime performance of a molecular dynamics model using conformal geometric algebra. 2010. URL http://www.science.uva.nl/~leo/agacse2010/talks_world/Seybold.pdf.

[12] 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.