Optimizations and Performance of a Robotics Grasping Algorithm described in Geometric Algebra

Florian Wörsdörfer[1], Florian Stock[2],
Eduardo Bayro-Corrochano[3], and Dietmar Hildenbrand[1]

[1] Technische Universität Darmstadt (Germany),
Graphical Interactive Systems Group
[2] Technische Universität Darmstadt (Germany),
Embedded Systems and Applications Group
[3] CINVESTAV Guadalajara (Mexico)

Abstract. The usage of Conformal Geometric Algebra leads to algorithms that can be formulated in a very clear and easy to grasp way. But it can also increase the performance of an implementation because of its capabilities to be computed in parallel. In this paper we show how a grasping algorithm for a robotic arm is accelerated using a Conformal Geometric Algebra formulation. The optimized C code is produced by the CGA framework Gaalop automatically. We compare this implementation with a CUDA implementation and an implementation that uses standard vector algebra.

Key words: Conformal Geometric Algebra, Robot Grasping, CUDA, Runtime Performance

1 Introduction

While points and vectors are normally used as basic geometric entities, in the 5D conformal geometric algebra we have a wider variety of basic objects. For example, spheres and circles are simply represented by algebraic objects. To represent a circle you only have to intersect two spheres, which can be done with a basic algebraic operation. Alternatively you can simply combine three points to obtain the circle through these three points. Similarly, transformations like rotations and translations can be expressed in an easy way. For more details please refer for instance to the book [3] as well as to the tutorials [6] and [4].

In a nutshell, geometric algebra offers a lot of expressive power to describe algorithms geometrically intuitive and compact. However, runtime performance of these algorithms was often a problem. In this paper, we investigate a geometric algebra algorithm of the grasping process of a robot [5] from the runtime performance point-of-view.

At first we present an alternative solution of the grasping algorithm using conventional mathematics and use its implementation as a reference for two

optimization approaches. These approaches are based on Gaalop [7], a tool for the automatic optimization of geometric algebra algorithms. We use the optimized C code of Gaalop in order to compare it with our reference implementation. In the next step we implement this C code also on the new parallel CUDA platform [11] and compare the runtime performance with the other two implementations.

2 The Grasping Algorithm with Conventional Mathematics

In this chapter we give a description of the algorithm described below using standard vector algebra and matrix calculations. To keep this version comparable to the one using geometric algebra the same amount of work and time has been spend for both. We assume that all necessary data has been extracted from the stereo images as explained in Section 4.1.

The circle z_b is the circumscribed circle of the triangle Δ_b which is formed by the three base points. To compute its center two perpendicular bisectors have to be constructed. The intersection of them is the center point p_b of z_b. First the middle points $m_{12} = \frac{1}{2}(x_{b_1} + x_{b_2})$ and $m_{13} = \frac{1}{2}(x_{b_1} + x_{b_3})$ of two sides of Δ_b are computed.

Next the direction vectors $d_{12} = (x_{b_2} - x_{b_1}) \times n_b$ and $d_{13} = (x_{b_3} - x_{b_1}) \times n_b$ are needed to construct the perpendicular bisectors. For this the normal vector $n_b = (x_{b_2} - x_{b_1}) \times (x_{b_3} - x_{b_1})$ of the plane defined by the base points has to be constructed.

Now the perpendicular bisectors pb_{12} and pb_{13} and their intersection p_b can be computed:

$$p_b = m_{12} + \lambda_{12_S} \cdot d_{12} = m_{13} + \lambda_{13_S} \cdot d_{13} \tag{1}$$

The circle z_b has to be translated in the direction of the normal vector n_b of the plane π_b in which z_b lies in. The distance z_b has to be translated is half the distance $d = \frac{n_b}{|n_b|}(x_a - p_b)$ between the point x_a and the plane π_b. So the translation vector is $T_b = \frac{1}{2}d \cdot \frac{n_b}{|n_b|}$. The normal vector of the plane in which z_t lies in equals the one of z_b, so $n_t = n_b$.

To be able to compute the necessary rotation the normal vector of the plane in which the gripper lies in has to be constructed. The robot is able to extract the center p_h of the gripper circle and two additional points g_1 and g_2 on it from the stereo pictures. With that the normal vector $n_h = (g_1 - p_h) \times (g_2 - p_h)$ of the gripper plane can be computed.

Because the needed rotation axes is perpendicular to the plane that is spanned by n_h and n_t its normal vector $n_{th} = n_h \times n_t$ has to be computed. With the vector n_{th} the rotation axes $l_R = l_R = p_h + \lambda \cdot n_{th}$ can be described.

The translation vector $l_T = p_t - p_h$ is just the difference of the two circle centers.

The angle between the two planes in which the circles lie in is equal to the angle between their normal vectors, so $\phi = acos\left(\frac{n_h \cdot n_t}{|n_h||n_t|}\right)$.

To perform the final rotation the following steps are necessary: compute the normalized rotation vector $n = \frac{n_{th}}{|n_{th}|}$, translate the rotation axes into the origin using RT_{orig}, compute the rotation using R and finally translate the axes back with RT_{back}.

$$RT_{orig} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_{h_1} & -p_{h_2} & -p_{h_3} & 1 \end{bmatrix} RT_{back} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p_{h_1} & p_{h_2} & p_{h_3} & 1 \end{bmatrix} \quad (2)$$

$$c = cos(\phi), \; s = sin(\phi), \; m = 1 - cos(\phi) \quad (3)$$

$$R = \begin{bmatrix} n_1^2 m + c & n_1 n_2 m + n_3 s & n_1 n_3 m - n_2 s & 0 \\ n_1 n_2 m - n_3 s & n_2^2 m + c & n_2 n_3 m + n_1 s & 0 \\ n_1 n_3 m + n_2 s & n_2 n_3 m - n_1 s & n_3^2 m + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

Finally the transformation can be computed by translating and rotating the points g_1 and g_2 from which the new position of the gripper circle can be derived.

3 Gaalop

The main goal of Gaalop is the combination of the elegance of algorithms using geometric algebra with the generation of efficient implementations.

Gaalop uses the symbolic computation functionality of Maple (together with a library for geometric algebras [1]) in order to optimize the geometric algebra algorithm developed visually with CLUCalc [12]. Gaalop computes the coefficients of the desired variable symbolically, returning an efficient implementation.

3.1 The Main Data Structure of Gaalop

The main data structure of Gaalop is an array of all the basic algebraic entities. They are called **blades** and are the basic computational elements and the basic geometric entities of geometric algebras. The 5D conformal geometric algebra consists of blades with **grades** 0, 1, 2, 3, 4 and 5, whereby a scalar is a **0-blade** (blade of grade 0). The element of grade five is called the pseudoscalar. A linear combination of blades is called a **k-vector**. So a bivector is a linear combination of blades with grade 2. Other k-vectors are vectors (grade 1), trivectors (grade 3) and quadvectors (grade 4). Furthermore, a linear combination of blades of different grades is called a **multivector**. Multivectors are the general elements of a geometric algebra. Table 1 lists all the 32 blades of conformal geometric algebra with all the indices as used by Gaalop. The indices indicate 1: scalar, 2...6: vector, 7...16: bivector, 17...26: trivector, 27...31: quadvector, 32: pseudoscalar.

A point $P = x_1 e_1 + x_2 e_2 + x_3 e_3 + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$ for instance can be written in terms of a multivector as the following linear combination of blades

$$P = x_1 * blade[2] + x_2 * blade[3] + x_3 * blade[4] + \frac{1}{2}\mathbf{x}^2 * blade[5] + blade[6] \quad (5)$$

For more details please refer for instance to the book [3] as well as to the tutorials [6] and [4].

Table 1. The 32 blades of the 5D conformal geometric algebra with the corresponding indices used by Gaalop.

index	blade	grade
1	1	0
2	e_1	1
3	e_2	1
4	e_3	1
5	e_∞	1
6	e_0	1
7	$e_1 \wedge e_2$	2
8	$e_1 \wedge e_3$	2
9	$e_1 \wedge e_\infty$	2
10	$e_1 \wedge e_0$	2
11	$e_2 \wedge e_3$	2
12	$e_2 \wedge e_\infty$	2
13	$e_2 \wedge e_0$	2
14	$e_3 \wedge e_\infty$	2
15	$e_3 \wedge e_0$	2
16	$e_\infty \wedge e_0$	2

index	blade	grade
17	$e_1 \wedge e_2 \wedge e_3$	3
18	$e_1 \wedge e_2 \wedge e_\infty$	3
19	$e_1 \wedge e_2 \wedge e_0$	3
20	$e_1 \wedge e_3 \wedge e_\infty$	3
21	$e_1 \wedge e_3 \wedge e_0$	3
22	$e_1 \wedge e_\infty \wedge e_0$	3
23	$e_2 \wedge e_3 \wedge e_\infty$	3
24	$e_2 \wedge e_3 \wedge e_0$	3
25	$e_2 \wedge e_\infty \wedge e_0$	3
26	$e_3 \wedge e_\infty \wedge e_0$	3
27	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$	4
28	$e_1 \wedge e_2 \wedge e_3 \wedge e_0$	4
29	$e_1 \wedge e_2 \wedge e_\infty \wedge e_0$	4
30	$e_1 \wedge e_3 \wedge e_\infty \wedge e_0$	4
31	$e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	4
32	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5

3.2 Use of Gaalop

For our application Gaalop is used in the following way: At first our algorithm is described using CLUCalc. One big advantage of CLUCalc is that the algorithm can be developed in an interactive and visual way. Another advantage is that the algorithm can be already verified and tested within CLUCalc. Gaalop uses this CLUCalc code in the next step in order to compute optimized multivectors using its symbolic optimization functionality. Leading question marks in the CLUCalc code indicate the variables that are to be optimized. Gaalop automatically generates C code for the computation of all the coefficients of the resulting multivectors. This C-Code is used as a basis for our CPU as well as our CUDA implementation.

4 The Algorithm in Geometric Algebra and its Optimization

The algorithm used here is based on the robotics algorithm described in the paper [5]. It is used by the robot "‘Geometer"’ and can be downloaded as a CLUScript from [8].

4.1 Interface and Input Data

The algorithm needs four points that identify the object to be grasped. The robot acquires these points by taking a calibrated stereo pair of images of the object and extracting four non-coplanar points from these images.

After the points are gathered the orientation of the object has to be determined. For this the distance from one point to the plane spanned by the other three points is calculated. The point with the greatest distance d_a is called apex point x_a while the others are called base points $x_{b_1}, x_{b_2}, x_{b_3}$.

We assume that all the steps described above are already performed. So the starting point for our algorithm is the situation shown in Figure 1. The aim of the algorithm is now to compute the necessary translation and rotation for the gripper of the robot arm so that it moves to the middle of the base points and the apex point. This is done by first computing the grasping circle z_t as a translation of the base circle z_b. Then the necessary translator and rotor are computed to move the gripper circle z_h. The position of the gripper is also extracted from the stereo pictures by tracking its center and two screws on its rim.

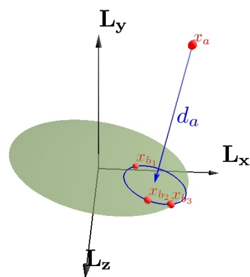

Fig. 1. The four points identifying the object to be grasped.

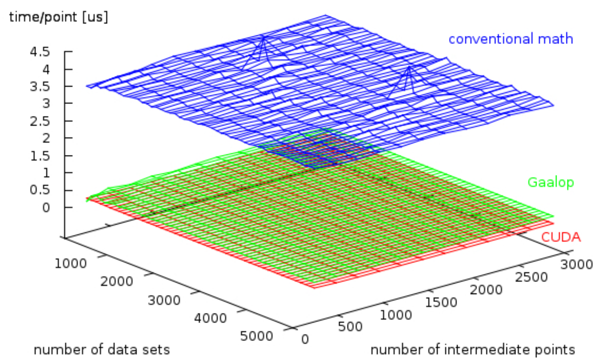

Fig. 2. Comparison of performance: The Gaalop code is up to fourteen times faster than the conventional math. The CUDA implementation gives a further performance improvement.

4.2 Short Description in Geometric Algebra

In this section we give a short description of the grasping algorithm using geometric algebra. The code of the listing in Figure 3 can be directly pasted into a CLUScript to visualize the results. The same code is used with Gaalop to produce the optimized C code. In our case we noticed that the `BasePlane` is needed as an intermediate expression to avoid the construction of extreme large expressions causing long computation times or sometimes even abnormal program termination. For the same reason the products `T*R` and `~R*~T` in the last line also were generated as intermediate results. Also Gaalop can only optimize

```
zb_d = xb1 ^ xb2 ^ xb3;
?zb = *zb_d;
?BasePlane = *(zb_d ^ einf);
NVector = (BasePlane * einf).e0;
NLength = abs(NVector);
NVector = NVector/NLength;
Plane = BasePlane/NLength;
d_a=(xa.Plane)*NVector;
?T = 1 + 0.25 * d_a * einf;
?z_t = T * zb * ~T;
S_h = VecN3(g1,g2,g3)
      - 0.5*(g4*g4)*einf;
Pi_h = -e2;
?z_h = S_h ^ Pi_h;
s_h = -0.5*S_h*einf*S_h;
S_t = *z_t / ((*z_t) ^ einf);
s_t = -0.5*S_t*einf*S_t;
?l_T = s_h ^ s_t ^ einf;
?d = abs(l_T);
?l_T = l_T / d;
l_h = z_h ^ einf;
l_t = z_t ^ einf;
?Pi_th = l_t ^ (l_h*(einf^e0));
l_r_direct = s_h ^ (*Pi_th) ^ einf;
?l_r = *l_r_direct / abs(l_r_direct);
?phi = acos((l_t.l_h)
       / (abs(l_t)*abs(l_h)));
?R = cos(0.5*phi) - l_r*sin(0.5*phi);
?T = 1 + 0.5*d*l_T*einf;
?z_h_new = T*R*z_h*~R*~T;
```

Fig. 3. This is a CLUScript implementing the complete grasping algorithm. Only concrete values for the points of the object and the position of the gripper are missing.

the arguments of function calls like abs or acos. In order to compute the target circle first the base circle $z_b = x_{b_1} \wedge x_{b_2} \wedge x_{b_3}$ (the blue one depicted in Figure 1) has to be compute and translated in the direction and magnitude of $\frac{d_a}{2}$ with the translator $T_b = 1 + \frac{1}{4} d_a e_\infty$. This gives the grasping circle $z_t = T_b z_b \widetilde{T_b}$.

Now that the final position the gripper has to reach is known the necessary translation and rotation can be computed. To get the translator T the translation axes $l_T^* = p_h \wedge p_t \wedge e_\infty$ is needed. It is defined by the center points $p_t = z_t e_\infty z_t$ and $p_h = z_h e_\infty z_h$ of the two circles z_t and z_h. Also the distance between p_t and p_h has to be computed. It is given by $d = |l_T^*|$. Finally the translation is $T = 1 + \frac{1}{2} \Delta d \, l_T \, e_\infty$.

To compute the rotor R the axes of the circles z_t and z_h have to be used. They are $l_t^* = z_t \wedge e_\infty$ and $l_h^* = z_h \wedge e_\infty$ and are needed to calculate the plane $\pi_{th}^* = l_t^* \wedge (l_h^* e_0 \wedge e_\infty)$. This plane is used to get the rotation axes $l_r^* = p_h \wedge \pi_{th} \wedge e_\infty$. The angle between the two circles can be computed with the help of the inner product of their axes which gives $cos(\phi) = \frac{l_t^* \cdot l_h^*}{|l_t^*||l_h^*|}$. Finally the rotor is $R = e^{-\frac{1}{2} \Delta \, \phi \, l_r} = cos(\frac{1}{2} \Delta \phi) - l_r sin(\frac{1}{2} \Delta \phi)$.

4.3 C Code Generated by Gaalop

Because the C code Gaalop generates is quite large only a small portion of it will be given here. The following listing shows an excerpt of the C code for the first question mark from the listing above. For brevity zb[8] to zb[16] have been omitted.

```
float zb[32] = {0.0};
zb[7]=-xb3[4]*xb1[5]*xb2[6]+xb3[4]*xb1[6]*xb2[5]+xb2[4]*xb1[5]*xb3[6]
      -xb2[4]*xb1[6]*xb3[5]+xb1[4]*xb2[6]*xb3[5]-xb1[4]*xb2[5]*xb3[6];
```

```
zb[8]=xb2[3]*xb1[6]*xb3[5]-xb2[3]*xb1[5]*xb3[6]-xb3[3]*xb1[6]*xb2[5]
      -xb1[3]*xb2[6]*xb3[5]+xb1[3]*xb2[5]*xb3[6]+xb3[3]*xb1[5]*xb2[6];
...
```

5 The CUDA Implementation

General Purpose Graphics Processing Unit (GPGPU) are gaining much attention as cheap high performance computing platforms. They developed from specialized hardware, where general problems could only be computed by mapping the problem to the graphics domain, to versatile many-core processors. These offer, depending on the manufacturer, different programming models and interfaces. As our hardware is a NVIDIA based GTX 280 board, we use CUDA [11]. Other possibilities would be vendor independent RapidMind [10] or Brook++ [2], or the new OpenCL standard from the Khronos Group [9].

The GTX 280 consists of 30 multiprocessors, with each containing 8 parallel floating point data paths operating in a SIMD-like manner, i.e. they basically execute the same operation with different data. Are more threads scheduled the hardware automatically uses this to hide memory latencies.

As the computation for the grasping algorithm contains no control flow and access of the input pure sequential, further architectural particularities (i.e. memory organisation, control flow in parallel task) can be ignored. For a more detailed information about the architecture see the documentation [11].

The implementation of the grasping algorithm uses the same code for the computation as for the CPU. The hundreds of parallel data paths are utilized by computing in each thread a separate point.

In the following benchmarking, the host system was the benchmarked CPU system, and the measured times include the time to transfer the input onto the onboard memory and the time to transfer the result back to the host system.

6 Performance Measurements

Table 2 shows that the Gaalop code without any further manual improvements is up to fourteen times faster than the code derived from the conventional math algorithm although it seems to be sensitive to the quality of the input data. The reason for that is subject to further studies. Two third of the time needed by the implementation using conventional math is used for the calculation of the rotation matrices. The CUDA implementation is more than three times faster than the Gaalop code.

In Figure 2 the time needed to compute one single intermediate step is plotted against the used number of data sets and the number of computed intermediate steps. It shows that the time needed to calculate one intermediate result is independent of the total number of intermediate steps and the number of used data sets for all implementations. Only the CUDA implementation gains a slight benefit from a higher number of intermediate steps because of the additional memory transfers.

The performance measurements were conducted on a Pentium Core2Duo clocked at 3.06 GHz by taking the time to calculate a certain number of intermediate steps with a various number of random data sets. One data set consists of 3D-Cartesian coordinates for seven points. The algorithm is capable of calculating an arbitrary number of intermediate results to represent the motion of the gripper.

Table 2. Time needed to compute a single intermediate point using a total of 1000 data sets with 240 intermediate points and the according speedup factor.

Implementation	**Time [μs]**	**Speedup**
conventional math CPU	3.50	1
CGA CPU	0.25	14
CGA CUDA	0.08	44

7 Conclusion

In this paper, we compared three implementations of an algorithm for the grasping process of a robot from the performance point of view. The basic algorithm was developed using the geometrically very intuitive mathematical language of geometric algebra. It turned out that the geometric algebra algorithm when optimized with the Gaalop tool can be up to fourteen times faster than the conventional solution based on vector algebra. Another improvement can be achieved when implementing this optimized code on the parallel CUDA platform.

References

1. R. Ablamowicz and B. Fauser. The homepage of the package Cliffordlib. HTML document http://math.tntech.edu/rafal/cliff9/, 2005. Last revised: September 17, 2005.
2. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786, 2004.
3. L. Dorst, D. Fontijne, and S. Mann. *Geometric Algebra for Computer Science, An Object-Oriented Approach to Geometry.* Morgan Kaufman, 2007.
4. D. Hildenbrand. Geometric computing in computer graphics using conformal geometric algebra. *Computers & Graphics*, 29(5):802–810, 2005.
5. D. Hildenbrand, E. Bayro-Corrochano, and J. Zamora. Inverse kinematics computation in computer graphics and robotics using conformal geometric algebra. In *Advances in Applied Clifford Algebras.* Birkhuser, 2008.
6. D. Hildenbrand, D. Fontijne, C. Perwass, and L. Dorst. Tutorial geometric algebra and its application to computer graphics. In *Eurographics conference Grenoble*, 2004.

7. D. Hildenbrand and Joachim Pitt. The Gaalop home page. HTML document http://www.gaalop.de, 2008.
8. Dietmar Hildenbrand. Home page. HTML document http://www.gris.informatik.tu-darmstadt.de/ dhilden/, 2009.
9. Khronos Group. *OpenCL Specification 1.0*, June 2008.
10. Michael D. McCool. *Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform*. Rapidmind, 2006.
11. NVIDIA Corp. *NVIDIA CUDA Compute Unified Device Architecture – Programming Guide*, June 2007.
12. C. Perwass. The CLU home page. HTML document http://www.clucalc.info, 2008.