Geometric Algebra Computing on the CUDA Platform

Christian Schwinn
TU Darmstadt, Germany
Department of Computer Science
schwinn@rbg.informatik.tu-darmstadt.de

Andreas Görlitz
TU Darmstadt, Germany
Department of Computer Science
A.Goerlitz@stud.tu-darmstadt.de

Dietmar Hildenbrand
TU Darmstadt, Germany
Department of Computer Science
dhilden@gris.informatik.tu-darmstadt.de

ABSTRACT

Geometric Algebra (GA) is a mathematical framework that allows a compact, geometrically intuitive description of geometric relationships and algorithms. These algorithms require significant computational power because of the intrinsically high dimensionality of geometric algebras. Algorithms in an n-dimensional GA require 2^n elements to be computed for each multivector. GA is not restricted to a maximum of dimensions, so arbitrary geometric algebras can be constructed over a vector space $\mathcal{V}_n$. Since computations in GA can be highly parallelized, the benefits of a parallel computing architecture can lead to a significant speed-up compared to standard CPU implementations, where elements of the algebra have to be calculated sequentially. An upcoming approach of coping with parallel computing is to use general-purpose computation on graphics processing units (GPGPU). In this paper, we focus on the Compute Unified Device Architecture (CUDA) from NVIDIA [9]. We present a code generator that takes as input the description of an arbitrary geometric algebra and produces an implementation of geometric products for the underlying algebra on the CUDA platform.

Keywords: Geometric Algebra, Geometric Computing, GPU, CUDA.

1 INTRODUCTION

Geometric Algebra (GA) has become more and more popular in different fields of research. Using GA makes it possible to develop very compact algorithms while keeping them geometrically intuitive. One major drawback is the reduced performance when executing GA algorithms without further processing. But recent research has shown that it is possible to speed up GA algorithms drastically by means of static code optimization and switching to parallel computing architectures like field-programmable gate arrays (FPGA), for instance. Moreover, this can lead to performance improvements compared to standard implementations [8].

Applications written in GA require a very large number of calculations to be processed, e.g. feature extraction algorithms [11]. In many cases it is necessary to define highly customized non-standard algebras in order to fit the problem statement. What these problems have in common is a remarkable amount of parallelization required to fulfill the constraints of reduction. In theory, it is possible to decrease the order of time complexity for certain applications which, in turn, requires virtually infinite operations to be executed in parallel.

In this paper, we investigate the potential of executing GA operations on parallel architectures. As a very first approach we focus on the implementation of arbitrary geometric products on the CUDA platform as a means for evaluating the performance of parallel computing in GA compared to standard implementations. We implement the calculation of the geometric product without any restrictions to the underlying algebra and associated metric and signature. We exploit the property that elements of the result multivector can be easily computed in parallel, e.g. each one in a separate (parallel) thread on a CUDA-enabled GPU. Therefore, we implement a code generator producing parallel CUDA code calculating the geometric product of the related algebra. This code can be used to speed up algorithms.

Our approach takes as input the description of an n-dimensional algebra in terms of a metric or signature and calculates a data structure describing the elementary product of all possible combinations of basis blades. This can be seen as a lookup table that is first optimized according to GA simplifications and then used to generate expressions for the individual result multivector components that only depend on the coefficients of the input multivectors to be multiplied. This corresponds to a *table based* compilation process as described in [5]. Finally, these expressions are translated into parallel CUDA code that calculates the result multivector and can be used for efficient calculation of the geometric product.

As a result, we evaluate the performance of the parallelized geometric product to get an estimation on the impact of parallel computing on problems in GA.

2 CONTRIBUTION

We present a code generator which translates the description of an arbitrary geometric algebra with associated signature and metric into CUDA code that implements the geometric product for the specified algebra. This code is a building block that can be used in GA algorithms to calculate the geometric product with the

help of a NVIDIA GPU. The code generator is written in Java, allowing the integration into a future versions of the Gaalop [6] optimizer.

Our approach consists of the steps outlined below:

1. Input description of signature and metric for the geometric algebra to be optimized.

2. Computation and optimization of a lookup data structure representing elementary products of basis blades for the given algebra.

3. Calculation of expressions for each component of the result multivector.

4. Code generation for optimized output code implementing the geometric product.

2.1 Specification of Algebra

We support arbitrary geometric products to be optimized by our compiler. Therefore, it must be possible to specify an arbitrary n-dimensional geometric algebra by means of a signature and metric description. From this information we derive a $n \times n$ matrix describing the geometric product of basis vectors with $e_{ij} = e_i e_j$.

2.2 Computation of Data Structure

Using the information from the previous step, we calculate a lookup data structure that describes the geometric product of all possible basis blades that can be constructed over the n basis vectors from the algebra. This data structure has the form of a matrix with $2^n \times 2^n$ entries. Note that an arbitrary geometric algebra is non-euclidean, i.e. its metric is not diagonal. So, in general, a single entry of the lookup matrix consists not only of a simple product of basis vectors but rather of a sum of expressions. In the 5D conformal algebra, for example, $e_0 e_\infty = -1 + e_0 \wedge e_\infty$. The maximum number of summands in this algebra is 2, namely the scalar -1 and the 2-blade $e_0 \wedge e_\infty$.

Each entry in the lookup data structure can be viewed as a list of references to basis blades named E_1 to E_N with $N = 2^n$. For a three-dimensional algebra, for example, the basis blades are named as shown in the following table.

E_1	1
E_2	e_1
E_3	e_2
E_4	e_3
E_5	$e_1 \wedge e_2$
E_6	$e_1 \wedge e_3$
E_7	$e_2 \wedge e_3$
E_8	$e_1 \wedge e_2 \wedge e_3$

Table 1: Naming of basis blades.

The internal representation can be made very compact using a bitwise encoding for "active" basis vectors in a blade, e.g. 101 for $E_6 = e1 \wedge e3$ as used in Gaigen [4], for instance. We assume a canonical ordering of blades, starting with scalars, 1-blades, 2-blades and so on, labeled with $E_1, E_2, ...E_N$.

Exploiting geometric algebra properties such as anti-commutativity ($e_i \wedge e_j = -e_j \wedge e_i$) and the knowledge of the inner product of basis vectors from the first step, it is possible to simplify the entries in the lookup matrix. The goal is to have only very few references to signed (+/-) basic blades. This applies to euclidean algebras, for instance, where the metric is diagonal and each entry consists of a single blade. The following table gives an example for the optimized lookup table of a 2-dimensional euclidean algebra. More details about the table-based approach of implementing and optimizing geometric algebra operations like the geometric product can be found in [5].

	b	E_1	E_2	E_3	E_4
a		1	e_1	e_2	e_{12}
E_1	1	E_1	E_2	E_3	E_4
E_2	e_1	E_2	E_1	E_4	E_3
E_3	e_2	E_3	-E_4	E_1	-E_2
E_4	e_{12}	E_4	-E_3	E_2	-E_1

Table 2: Example of a lookup table.

We calculate the lookup table using the freely available reference implementation of Gaigen [3], which is implemented in Java and provides basic functionality without performance considerations but serves our purposes. After the calculation of the lookup table, we get a list of basis blades for each entry of the table containing only non-zero references to the basis blades E_i. Each entry represents a part of the expression that contributes to the coefficient of the result multivector. For each reference to basis blades E_i in the entry, the input coefficients associated with the row and column of the position in the lookup table contribute to the final coefficient of blade E_i in the result multivector.

2.3 Multivector Components

From the information stored in the lookup table, expressions for each component of the result multivector can be determined. Each multivector consists of a combination of 2^n basis blades and associated coefficients, where the i-th coefficient is multiplied by E_i. Since only coefficients of the input multivectors are of interest, the references to basis blades found in the lookup table have to be translated. For each multivector component i, the references to the E_i are looked up in the table.

Let a, b be the input multivectors with coefficients $a_1, a_2, \ldots a_N$ and $b_1, b_2, \ldots b_N$. Then for an entry E_i in

the lookup table the coefficients are selected according to the column and row where the entry has been found. So, for the first occurrence of E_1 in the example from Table 2, which is placed in the first column and first row, indices a_1 and b_1 have to be selected. We will associate the row index with multivector a and the column index with multivector b, as indicated by Table 2. The sign to be used corresponds directly to the sign of the reference to the basis blade found in the table.

The final expressions for the coefficients of the result multivector c from the above example are shown below.

$$c_1 = a_1b_1 + a_2b_2 + a_3b_3 - a_4b_4$$

$$c_2 = a_1b_2 + a_2b_1 - a_3b_4 + a_4b_3$$

$$c_3 = a_1b_3 + a_2b_4 + a_3b_1 - a_4b_2$$

$$c_4 = a_1b_4 + a_2b_3 - a_3b_2 + a_4b_1.$$

Note that these expressions only consist of a sum of products of coefficients. The GPU which will finally calculate these coefficients and therefore does not require any knowledge about geometric algebra operations like outer or inner product, for example.

2.4 Code generation

The last step in the code generation process is creating the CUDA output. We parallelize the calculation of each multivector component, so each component is calculated in a separate thread. This is possible since each thread only depends on the input coefficients of multivectors a and b. For an n-dimensional algebra there will be 2^n threads executed in parallel. In practice, the actual number of parallel threads depends on the amount of processing units on the target platform.

In fact, the parallelization scheme is application-dependent. For a single application calculating a single geometric product at a time, the above principle is sufficient. But for applications which might have a large number of geometric products to be calculated in parallel at the same time, this principle might be inappropriate. In this case, other concepts like streaming, working queues or warp-level parallelization should be more useful. As a first attempt of a GPU-based implementation in this paper, we stick to the first method for parallelization, calculating one coefficient per thread.

Expressions for different multivector components differ as well in the coefficients to be multiplied as in the structure. Depending on the underlying algebra, some multivector components could be always zero while others consist of a large expression with multiple additions and subtractions. In order to distribute parallel threads calculating different components, it is therefore necessary to find a generic representation of such an expression, since parallel threads have to execute the same code[1]. To cope with differences in the length and structure of these expressions, we define a meta data structure which will be used in the CUDA code produced by the code generator. From this data structure, parallel threads can look up which elements of the input multivectors have to be multiplied and added or subtracted to the current result.

The meta data structure consists of two parts. The first one represents a single *summand* in the expression for a multivector component, e.g. a_1b_3. Since these *summands* can also be counted negatively, there is an additional sign field. This structure is modeled as a C struct *Summand* keeping track of the indices of the input multivectors and the sign. To minimize the memory footprint, the code generator generates summand structures only for summands that actually will be used in the calculations. All the summands are finally stored sequentially in an array. A thread calculating the i-th multivector component must know which summands to select from this array. Therefore, the second part of the meta data structure keeps track of which summands correspond to which expression. This is modeled in another C struct *Info* which stores the offset to the relevant summands. This requires the code generator to store summands in the correct order. For each multivector component there is exactly one info object, so each thread selects the info object according to its thread index. The number of summand objects to be involved in the calculation is determined by the current offset and the offset of the next component or the maximum number of summand objects in the case of the last index.

Calculation of the meta data structure can be done before the actual calculation of the geometric product since the related information is static and not dependent on concrete coefficients of the input multivectors. Finally, each thread needs references to

- input multivectors a, b,
- result multivector c,
- array of summands and
- the array of info objects.

From this information each thread selects the info object of the multivector component to be calculated, from which it gets offset and length of the summand objects to involve. Then for each summand object the coefficients of a and b are multiplied according to the index in the summand info and signed accordingly. This temporary result is aggregated until all relevant summand objects have been processed. The final result is written to the result multivector c. The calculation of a multivector coefficient c_i is illustrated in Figure 1.

[1] This is in fact a shortcoming of CUDA-based applications: It is not possible to access the GPU's parallel processing units to be used for different tasks at the same time.

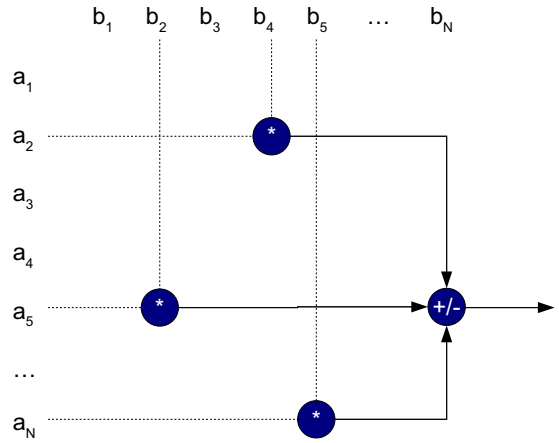

Figure 1: Calculation scheme for coefficient c_i

2.5 Implementation details

When generating CUDA code, it is important to consider the structure of the underlying hardware. The main aspect to be concerned is the memory hierarchy. Depending on the amount of data and the way data is accessed, it is very important to specify which data should be stored in which parts of the device memory. For details on the memory hierarchy, please refer to the CUDA Programming Guide [9].

Listing 1 shows an exemplary CUDA code which is executed by each thread. Variables a and b are vectors containing the input coefficients, c is used to store the results. The offset to the array of summand objects is obtained by the current thread index. According to the index and current offset, the number of summands to be used is determined. Then, for each relevant summand, the indices to the input multivectors are retrieved, and the corresponding coefficients are multiplied and signed. The final result is written into c.

```
__global__ void Calculate(float* a,
                          float* b,
                          float* c) {
  int i = blockDim.x*blockIdx.x+threadIdx.x;
  float res = 0.0;
  int offset = offsets[i];
  int length;
  if (i == N-1) {
    length = num_summands-offset;
  } else {
    length = offsets[i+1]-offset;
  }
  for (int j = 0; j < length; j++)
  {
    Summand s = summands[offset+j];
    if (s.sign) {
      res += a[s.index_a]*b[s.index_b];
    } else {
      res -= a[s.index_a]*b[s.index_b];
    }
  }
  c[i] = res;
}
```

Listing 1: Kernel code executed by each thread

This way it is possible that each kernel executes the same code although the expressions for different multivector components are variable.

The formal parameters to the kernel function, a, b and c, reside in the shared device memory automatically. Since they have to be passed from host side each time a geometric product should be calculated, this is already the place as close to the multiprocessor as possible.

Since the meta data structure consisting of the summand array and the offset objects has to be used in each thread, it is necessary to keep this data in the global device memory. Here, it is important to know that the meta data becomes very large, dependent on the dimension of the algebra. So, for algebras with dimensions larger than 6, other portions of the device memory like shared memory or constant memory are too small. Passing the meta data each time on a kernel call would place the data in the shared memory, which is restricted in size on the one hand and available only for a set of threads from one block on the other hand. Constant memory is cached, so there is in general a performance benefit over global device memory. But constant memory is restricted to 64 KB, which is far too less for large algebras. Details on memory and time consumption will be shown in section 3.

Another problem related to the meta data is the setup procedure, which has to be done before the actual computation can begin. This step is required only once, afterwards an arbitrary number of geometric products can be calculated without the overhead of setting up the data structure again. For each multivector component, there is a variable number of summands to be included in the calculation. Each summand object, which consists of the indices to the input multivectors and the corresponding sign, has to be added to the array of summands. Since in the worst case there are $2^n * 2^n$ summands in total, 2^{2n} objects have to be created and added to the array. This is a considerable amount of code to be produced, because an exponential number of lines of code has to be processed. If the dimension were too high, the produced code would be so large that most compilers would have problems compiling this code. It can therefore be necessary to swap this information out of the program source code, i.e. in a separate file which has to be read at runtime. Doing this leads to a constant size of the actual source code, whereas the size of the input file to be read during the setup increases exponentially with the number of dimensions.

3 EVALUATION

The approach presented in this paper is to be seen as a very first attempt to implement geometric algebra on a multiprocessor platform like GPUs. As a proof of concept, no optimizations have been applied at all. So, allowing to implement the geometric product of an *arbitrary* algebra requires to assume worst case scenarios

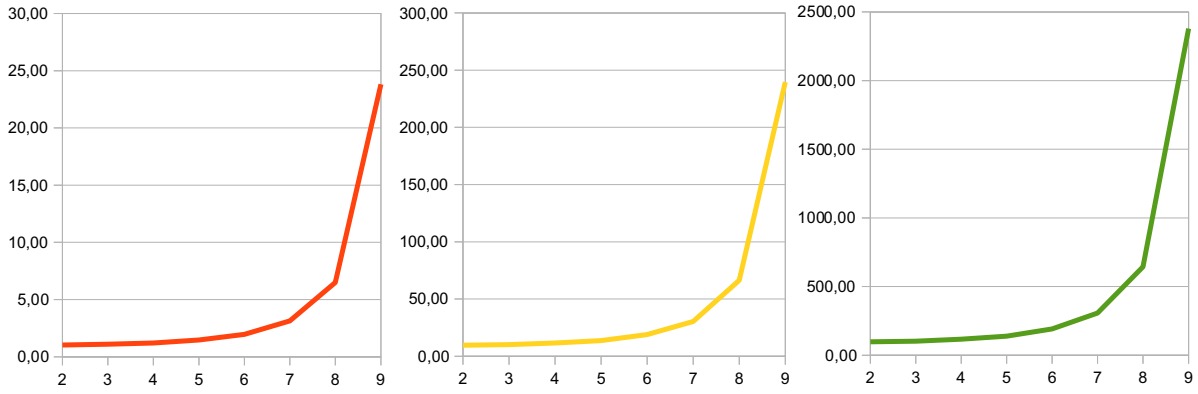

Figure 2: Comparison of time consumption (y-axis) for 10 (left), 100 (middle) and 1000 (right) geometric products. Times are measured in milliseconds. Values on the x-axis specify the dimension of the algebra.

where multivectors all have non-zero coefficients and lookup tables are fully occupied with no entries being zero. Consequently, a euclidean geometric algebra of dimension n is expected to have a lookup table with exactly $2^n * 2^n = 2^{2n}$ elements, each one representing one summand in the different sum-of-product expressions for every coefficient of the result multivector component. Due to this exponential growth and the assumption that multivectors have non-zero coefficients, the meta data required to represent these expressions becomes very large, even for small dimensions.

Performance considerations

Memory

Restricting the maximum dimension of the algebra to 15 allows to encode a summand of some coefficient into a single integer value in the following way[2]:

- 1 bit for sign (e.g. 0 for positive, 1 for negative),
- 15 bits for the index to the first multivector,
- 15 bits for the index to the second multivector and
- 1 unused bit for padding.

Consequently, for an algebra of dimension 15, there would be 2^{15} multivector elements, requiring 2^{15} indices. A summand object, consisting of two indices and the sign would therefore occupy 31 bits of a 32-bit integer variable. As argued above, this would mean to create $2^{2*15} = 2^{30}$ summand objects, each one requiring the space of a single integer, i.e. 4 bytes. The total amount of memory would therefore be $2^{30} * 2^2 = 2^{32}$ bytes = 4 Gbyte.

Obviously, the memory footprint is the limiting factor in this approach. Without optimizations, high dimensions are not feasible at all, because more time would be spent copying the data back and forth from and to the device memory while losing the advantages of parallel processing.

[2] Note that otherwise the meta data would be about three times larger, making the application infeasible due to the memory overhead.

Time

Figure 2 shows three graphs plotting the time to calculate 10, 100 and 1000 geometric products over the dimension of the algebra. For dimensions ≤ 7 with moderate memory footprint, time rises linearly with the number of dimensions. Since memory consumption rises exponentially, higher dimensions have a lot more influence on the computation time, as indicated in the plots. Comparing the plots with respect to the number of geometric products shows that increasing the number of products by factor 10 results in an increase in time consumption by approximately factor 10, too.

These results give a hint that the time required to calculate a number of products is influenced by the memory needed to store the structure of expressions for the different multivector components, which itself depends on the dimension.

Comparison to CPU-based Calculation

The main disadvantage of a GPU-based approach is the necessity to copy data back and forth to and from the device, because the GPU has its own set of memory, registers and caches. Besides copying the meta data required to represent the instructions required to calculate a result coefficient, which has to be done only once per application lifetime, it is necessary to copy the vectors representing the input coefficients onto the device and output coefficients from the device. As opposed to the static meta data, this information changes in each call to the kernel function, so copying has to be performed on each calculation. The effort for copying input and output data is directly related to the dimension of the algebra, due to the fact that each vector of input and output coefficients has 2^n elements, i.e. 2^n times the

size of the data type, e.g. $2^n * 4$ bytes in the case of floats. As an example, calculating the product in an 8-dimensional algebra, $3 * 2^8$*sizeof(float)$=$ 3 KB of input and output data have to be copied for each product. This is a significant amount which has to be considered when measuring the performance of the implementation. In our benchmarks, the time spent for I/O operations between host and device took up to 25% of the overall computation time.

Gaigen [4] is a CPU-based implementation generator for geometric algebra. Gaigen uses advanced optimization techniques and uses specializations that tell the tool where further optimizations can be made, according to the type of multivector. As a CPU-only application, Gaigen does not suffer from memory overhead which is inevitable on parallel platforms as GPUs, for instance. This is why Gaigen is about factor 10 faster than our approach without any optimizations. In turn, Gaigen suffers from a similar problem as outlined in section 2.5, what makes Gaigen unusable for dimensions larger than 6 because the generated code becomes too large so compilers have problems compiling it.

Without considering the memory overhead, our approach is slightly faster than Gaigen, even without applying further optimizations on the structure of multivectors. With a certain knowledge about the types of multivectors which are about to be multiplied, a lot of computation time can be saved by removing parts from the meta data that are actually zero. For example, let the second half of both multivectors be always zero. Then 75% of the lookup table are zero and meta data shrinks to 25% of the worst case, which finally leads to better overall performance. For details, please refer to [5].

Moreover, our approach targets applications using high-dimensional geometric algebras. Only in cases where the dimension is high enough, memory overhead can be compensated by exploiting the parallel architecture, i.e. keeping the device fully loaded.

4 CONCLUSION

We have shown a very first approach of implementing geometric algebra on the GPGPU platform CUDA. We implemented a code generator producing an implementation of the geometric product on the CUDA platform. Applications running CUDA-enabled hardware are therefore able to use this implementation to make use of the computational power of modern GPUs. Without optimizations it is currently theoretically possible to calculate geometric products in algebras of dimension up to 15 without exceeding device memory restrictions[3]. In practice, this is not usable, since there would be too much memory overhead for this number of dimensions.

[3] For dimensions > 10 it is necessary to put input and output multivectors in global memory since shared memory where kernel parameters reside is limited to 16 KB.

Supporting arbitrary algebras while always considering the worst case of full multivectors with non-zero coefficients is too limiting. For most applications, e.g. using the 5D conformal geometric algebra, more than half of the multivector coefficients are usually zero. This is important to know in advance, because multiplication tables as mentioned in section 2.2 can be reduced to a large extent. This is a nice property which has to be exploited as much as possible. Otherwise, there is far too much "infrastructure" required to manage the calculation for different multivector components, as outlined in the previous sections. Since time consumption depends directly on the number of dimensions and the related amount of meta data required for calculation, reducing the lookup data and related memory consumption on the device to the minimum possible value is the most important step to achieve reasonable performance while supporting high-dimensional algebras. Of course, this requires the knowledge of multivectors to be multiplied. One solution is to tell the code generator which specializations of multivectors will be used. This is done in Gaigen, for instance. But creating specializations requires the user to know the structure of multivectors in advance. In complex algorithms, there might be situations where it is hard to decide which parts of multivectors might always be zero. It is therefore desirable to have a tool that optimizes parts of algorithms automatically like Gaalop [6]. This is why the integration of a code generator for parallel platforms into Gaalop is planned for future releases.

5 FUTURE WORK

Our next step will be to investigate the potential for minimization of memory needed to store and process multiplication tables. This can be done automatically by using the algorithm optimizer tool Gaalop, which optimizes algorithms written in the interactive visualization and calculation tool CLUCalc [10] and produces different output formats like C++, for instance. With the help of Gaalop, it will be possible to detect the structure of multivectors and to optimize multiplication tables according to the automatically detected specializations as described in [5].

Gaalop [6] makes it possible to optimize algorithms in GA rather than single calculations. Therefore, the code generator for parallel architectures will be integrated into the Gaalop optimizer software.

Furthermore, we will study existing libraries for geometric computing like described in [1] to find new methods how to generate optimized code for arbitrary algebras and dimensions.

A new standard for parallel computing, OpenCL [7], has recently been released. Using OpenCL makes it possible to address multiple computing devices like CPUs, GPUs or cell processors to be used for general-purpose computing. This standard generalizes vendor-

specific GPGPU approaches like CUDA or ATI Stream [2], for example. We see this as an important step for future developments of general-purpose computing. Since as from now OpenCL drivers are officially released by NVIDIA and OpenCL is supported in the latest version 10.6 of the Mac OS operating system, we will extend our code generator to produce OpenCL code in order to support different hardware platforms automatically.

REFERENCES

[1] John Browne. The GrassmannAlgebra Book Home Page. Available at `http://grassmannalgebra.info/grassmannalgebra/book/`, 2002.

[2] AMD Corp. The ATI Stream Technology home page. `http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx`, 2009.

[3] Leo Dorst and Daniel Fontijne. Geometric Algebra for Computer Science. `http://www.geometricalgebra.net/reference_impl.html`, 2009.

[4] Daniel Fontijne. Gaigen 2: A Geometric Algebra Implementation Generator. In *GPCE'06*. ACM, 2006.

[5] Dietmar Hildenbrand. Geometric Algebra Computers. *submitted to the proceedings of the GraVisMa workshop, Plzen*, 2009.

[6] Dietmar Hildenbrand and Joachim Pitt. The Gaalop home page. `http://www.gaalop.de`, 2008.

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

[8] H. Lange, F. Stock, D. Hildenbrand, and A. Koch. Acceleration and Energy Efficiency of a Geometric Algebra Computation using Reconfigurable Computers and GPUs. *FCCM*, 2009.

[9] NVIDIA. The CUDA home page. `http://www.nvidia.com/object/cuda\_home.html`, 2009.

[10] Christian Perwass. The CLU home page. HTML document http://www.clucalc.info, 2008.

[11] M. T. Pham, K. Tachibana, E. M. S. Hitzer, T. Yoshikawa, and T. Furuhashi. Classification and Clustering of Spatial Patterns with Geometric Algebra. *AGACSE*, 2008.