GAPPCO: an easy to configure Geometric Algebra coprocessor based on GAPP programs

D. Hildenbrand, S. Franchini, A. Gentile, G. Vassallo and S. Vitabile

Abstract. Because of the high numeric complexity of Geometric Algebra, its use in engineering applications relies heavily on tools and devices for efficient implementations. In this article, we present a novel hardware design for a Geometric Algebra coprocessor, called GAPPCO, which is based on Geometric Algebra Parallelism Programs (GAPP). GAPPCO is a design for a coprocessor combining both the advantages of optimizing software with a configurable hardware able to implement arbitrary Geometric Algebra algorithms. The idea is to have a fixed hardware easily and fast to be configured for different algorithms. We describe the new hardware design together with the complete tool chain for its configuration.

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

Keywords. Geometric Algebra, Geometric Algebra Computing, Gaalop, GAPP, GAPPCO.

1. Introduction

Especially since the introduction of CGA (Conformal Geometric Algebra) by David Hestenes et al. [9] [13] there is an increasing interest in using Geometric Algebra in engineering. Since for many engineering applications runtime performance is a big issue, many tools have been developed for efficient implementations of Geometric Algebra algorithms.

Most existing hardware architectures are based on vectorization and vector operations such that components of an operation can be performed independently in parallel and there are already solutions making use of this architecture [11]. Another approach to overcome the limitations of Geometric Algebra is to look for dedicated hardware architectures for the acceleration of

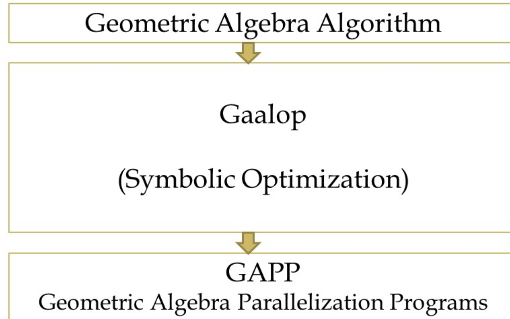

FIGURE 1. Gaalop generating optimized GAPP programs from Geometric Algebra algorithms

Geometric Algebra algorithms. Integrated circuit technology offers a means to achieve high performance with field-programmable gate arrays (FPGAs). The first serious approach was described by Perwass et al. [16] in 2003. A different approach was presented by Gentile et al. [8] in 2005. An update on this work was given by Franchini et al. in a series of papers such as [1], [2], [3], [4], [5] and [7]. The first custom-fabricated integrated circuit ASIC implementation was introduced by Mishra and Wilson [14] in 2006.

A good way of losing the high complexity of Geometric Algebra before going to the real computing device is to precompute / precompile Geometric Algebra algorithms based on the **G**eometric Algebra **al**gorithms **op**timizer Gaalop [12]. What remains after its optimization process are mainly parallel computations of multivector coefficients each consisting of sums of products, which are again efficiently to be parallelized. The GAPP (Geometric Algebra Parallelism Programs) language as the general structure of these optimized computations is generated by Gaalop (see Fig. 1). It is described in the book [10]. Please refer to Sect. 2 for an introduction to GAPP.

Gaalop already supports optimized hardware (HW) generation [18]. But, with every new algorithm the FPGA has to be completely reprogrammed. The solution that we present in this paper is more far-reaching. Here, we combine the GAPP technology with new developments in fixed HW solutions for Geometric Algebra, especially the *ConformalALU* of [6], implementing conformal transformations in hardware. We use this solution as an example for GAPPCO, a general coprocessor design based on GAPP. GAPPCO is a coprocessor design combining both the advantages of optimizing software with a configurable hardware able to implement arbitrary Geometric Algebra algorithms. The idea is to have a fixed hardware, easily and fast to be configured for different algorithms. Compared to standard hardware architectures it makes use of variable-size vectors, pipeling and fast register access.

Please refer to Sect. 3 for the general design and to Sect. 4 for the first version called GAPPCO I.

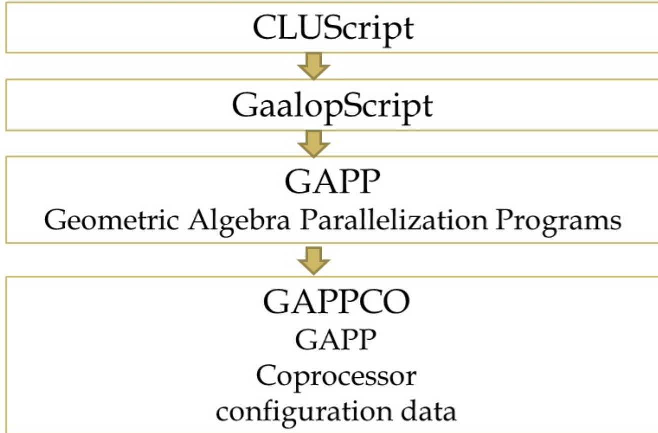

FIGURE 2. Configuration chain for GAPPCO.

2. CLUScript, GaalopScript and GAPP

We describe the configuration chain for GAPPCO based on an example: the reflection of a sphere in a plane. It is inspired by the reflector functionality of the ConformalALU [6], implementing this operation in hardware. The reflection oeration is the most basic transformation in Conformal Geometric Algebra, since all the conformal transformations can be composed based on it. Rotations and Translations, for instance, consist of two consecutive reflections related to two specific planes.

Figure 2 describes the chain of the main languages, needed for the configuration of GAPPCO, namely

- CLUScript for the interactive and visual development of the Geometric Algebra algorithm
- GaalopScript for the symbolic optimization of the algorithm
- GAPP for the generation of the specific configuration data for GAPPCO

resulting finally in the specific configuration data of GAPPCO.

CLUScript is the scripting language of CLUCalc [15], a tool for the visualization of Geometric Algebra algorithms. Figure 3 shows the reflection of a green sphere related to the red plane, which is resulting in the reflected yellow sphere as described in Listing 1.

LISTING 1. CLUScript for the computation of the reflection of an object a (point or sphere) related to a plane m

```
DefVarsN3();                 // use the conformal geometric algebra
_BGColor = Color(1,1,1);     // Background white
:IPNS;                       // use the IPNS representation

a1 = 1;
```

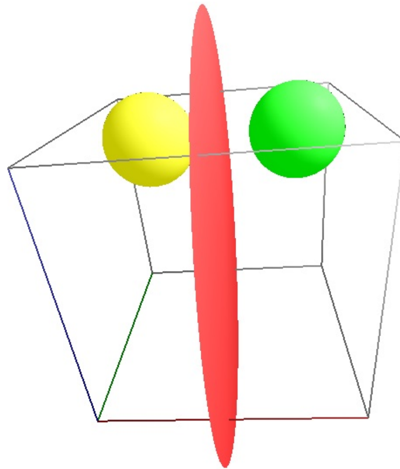

FIGURE 3. Reflection of a sphere.

```
a2 = 1;
a3 = 1;
a4 = 1.3;
m1 = 1;
m2 = 0;
m3 = 0;
m4 = 0;

a = a1*e1+a2*e2+a3*e3+a4*einf+ e0;
m = m1*e1+m2*e2+m3*e3+m4*einf;
?Dotproduct = a.m;
a_par = 2*(Dotproduct)*m;
?a_Refl = a - a_par;

:Green;
:a;
:Red;
:20*m;
:Yellow;
:a_Refl;
```

This CLUScript first defines the Geometric Algebra to be used (in this case the Conformal Geometric Algebra), the background colour for the visualization (white in this case) and the representation of the geometric objects (in this case the inner product null space representation). Then, the specific parameters of the objects a and m are defined. The object a is either a

point or a sphere to be reflected at m which is a plane with normal vector (m1,m2,m3) and m4 as the distance to the origin (please refer, for instance, to [10] for details about the representation of geometric objects in Conformal Geometric Algebra). The question mark indicates the multivectors to be shown in the output window of CLUCalc. At the end of the CLUScript, the original sphere is visualized in green, the plane in red and the reflected sphere in yellow.

Listing 2 describes the corresponding Geometric Algebra algorithm as a *GaalopScript*. In principle, GaalopScript is a subset of CLUScript, essentially describing the core Geometric Algebra algorithm. The big advantage is, that after the interactive and visual development of an algorithm, the core CLUScript algorithm can be cut and paste as input language for the symbolic optimization process of Gaalop.

LISTING 2. GaalopScript for the computation of the reflection of an object a at a plane m

```
a = a1*e1+a2*e2+a3*e3+a4*einf+ e0;
m = m1*e1+m2*e2+m3*e3+m4*einf;
?Dotproduct = a.m;
a_par = 2*(Dotproduct)*m;
?a_Refl = 0.5*(a - a_par);
```

The question mark in a GaalopScript indicates the multivectors to be computed explicitly (either as an intermediate or as a final result).

Note: the multiplication of the result by 0.5 is done, because the resulting computations become easier (and in CGA the multiplication with a scalar does not change the geometric object).

The *GAPP* (Geometric Algebra Parallelism Programs) language describes the general structure of the computations after the Gaalop optimization process (see Fig. 1). As described in the book [10], Geometric Algebra algorithms with all kind of products of multivectors always have the same principle structure. This is described in the GAPP instruction set of Table 4. The initial GAPPCO design focuses on this standard GAPP instruction set. If, for instance, divisions or square roots are needed, the instruction set has to be extended (see [17]).

The reflection algorithm of Listing 2 does not use any extended operations such as divisions and square roots. This is why the corresponding GAPP code (generated by Gaalop) uses the standard GAPP instruction set as presented in Listing 3.

LISTING 3. Resulting GAPP code of Listing 2.

```
assignInputsVector inputsVector = [a1,a2,a3,a4,m1,m2,m3,m4];

resetMv Dotproduct[32];
setVector ve0 = {inputsVector[-7,2,1,0]};
```

```
setVector ve1 = {1.0,inputsVector[6,5,4]};
dotVectors Dotproduct[0] = <ve0,ve1>;

//a_Refl[1] = (0.5 * inputsVector[0])
//           - (Dotproduct[0] * inputsVector[4])
resetMv a_Refl[32];
setVector ve2 = {0.5,Dotproduct[-0]};
setVector ve3 = {inputsVector[0,4]};
dotVectors a_Refl[1] = <ve2,ve3>;

//a_Refl[2] = (0.5 * inputsVector[1])
//           - (Dotproduct[0] * inputsVector[5])
setVector ve4 = {0.5,Dotproduct[-0]};
setVector ve5 = {inputsVector[1,5]};
dotVectors a_Refl[2] = <ve4,ve5>;

//a_Refl[3] = (0.5 * inputsVector[2])
//           - (Dotproduct[0] * inputsVector[6])
setVector ve6 = {0.5,Dotproduct[-0]};
setVector ve7 = {inputsVector[2,6]};
dotVectors a_Refl[3] = <ve6,ve7>;

//a_Refl[4] = (0.5 * inputsVector[3])
//           - (Dotproduct[0] * inputsVector[7])
setVector ve8 = {0.5,Dotproduct[-0]};
setVector ve9 = {inputsVector[3,7]};
dotVectors a_Refl[4] = <ve8,ve9>;

//a_Refl[5] = 0.5
assignMv a_Refl[5] = [0.5];
```

First of all, the 8 input values of the program (a1,a2,a3,a4,m1,m2,m3 and m4) are assigned to the inputsVector. Then, the coefficients of the multivectors Dotproduct and a_Refl are computed.

From the multivector *Dotproduct* only the index 0 is needed. It is the dot product of the two vectors ve0 and ve1 consisting of entries from the inputsVector as well as the value 1.0 as a constant.

Then the entries 1, 2, 3, 4 and 5 of the multivector *a_Refl* are computed. The first entry, for instance, is the dot product of the vectors ve2 and ve3 consisting of input values, constants and the result of the negated Dotproduct[0] (because of consistency reasons, -Dotproduct[0] is written Dotproduct[-0] in the GAPP language).

3. GAPPCO

GAPPCO is a design for a Geometric Algebra Computing coprocessor combining the advantages of optimizing software with a fixed hardware able to implement arbitrary Geometric Algebra algorithms. GAPPCO is based on Geometric Algebra Parallelism Programs (GAPP) that are already optimized in a sense that only the really needed computations are left. The idea is to have a fixed hardware easily and fast to be configured for different algorithms. While the GAPPCO design is a design for reconfigurable hardware, it is flexible in the sense that it can be used for ASICs and SOCs in the future.

GAPPCO consists of one or more GAPP units. Each GAPP unit is able to realize small GAPP programs (as, for instance, the GAPP program of Listing 3). The host interface is responsible for the data communication between

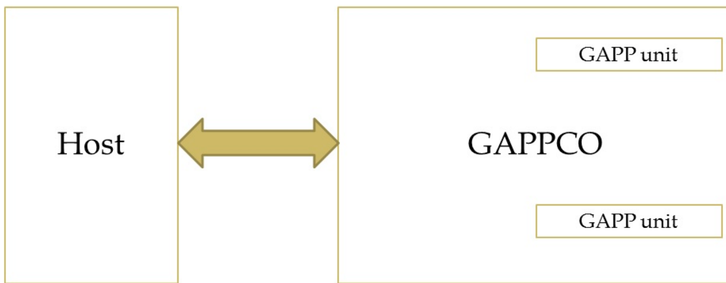

FIGURE 4. GAPPCO coprocessor with host interface.

host and GAPPCO and for the configuration of GAPPCO (see Fig. 4).

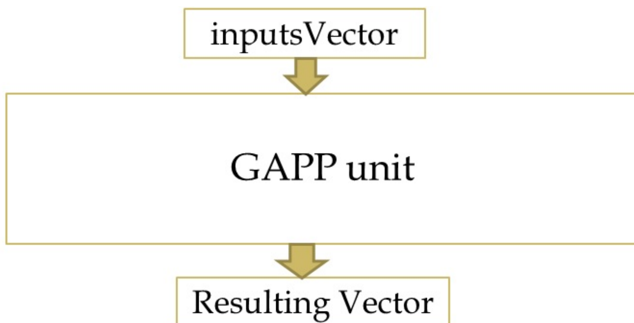

FIGURE 5. GAPP unit processing an inputsVector (vector with all the scalar input values) to resulting vectors.

As soon as a GAPP unit is configured, inputsVectors can be received from the host and resulting vectors can be sent to the host (see Fig. 5). For

runtime performance purposes, the architecture of GAPP units is pipelined as much as possible.

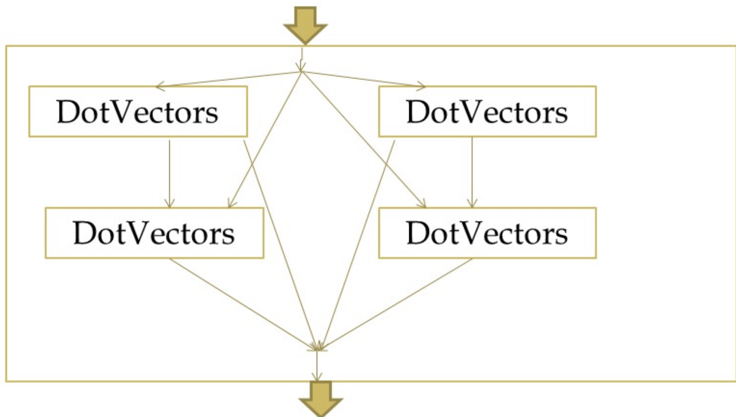

FIGURE 6. GAPP unit consisting of 2 levels of DotVectors units equipped with a configuration interface.

A GAPP unit consists of a network of one or more DotVectors units organized in one or more levels (see Fig. 6). The connections have to be configured before runtime based on the specific configuration data (see Sect. 4.2).

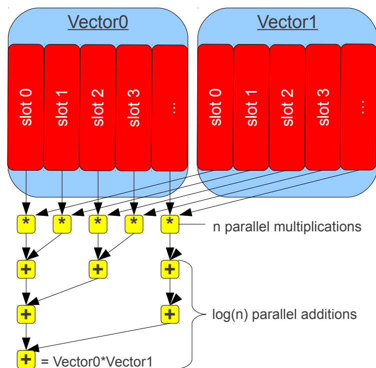

FIGURE 7. Parallel dot product of two n-dimensional vectors Vector0 and Vector1 (n parallel products followed by $\log(n)$ parallel addition steps).

Each DotVectors unit is responsible for the computation of one coefficient of one multivector. There is an implicit parallelism in this computation according to Fig. 7. The multiplications of each of the vector elements can be done in parallel as well as parts of the additions (see Fig. 7).

The GAPP example of Sect. 2 is restricted to the number of two vectors to be multiplied, while GAPP, generally, is supporting a higher number of vectors to be multiplied.

```
inputsVector:  a1  a2  a3  a4  m1  m2  m3  m4

Routing list:
0: inputsVector[0]
1: inputsVector[1]
2: inputsVector[2]
3: inputsVector[3]
4: inputsVector[4]
5: inputsVector[5]
6: inputsVector[6]
7: inputsVector[7]
8: 1.0
9: 0.5
10: Dotproduct[0]


DotVectors unit
input:  -7  2  1  0
input:  8  6  5  4
Level 1 result: Dotproduct[0]

DotVectors unit
input:  9  -10
input:  0  4
Level 0 result: a_Refl[1]

DotVectors unit
input:  9  -10
input:  1  5
Level 0 result: a_Refl[2]

DotVectors unit
input:  9  -10
input:  2  6
Level 0 result: a_Refl[3]

DotVectors unit
input:  9  -10
input:  3  7
Level 0 result: a_Refl[4]
```

FIGURE 8. Internal configuration data structure for GAPPCO based on the reflection example

Fig. 2 presents the configuration chain for GAPPCO as described in Sect. 2. For the last step, the generation of the GAPPCO configuration data, the particular GAPP listing has to be translated into a more suitable format as presented in Fig. 8. It mainly describes the DotVectors units for *Dotproduct* (1 dotVectors unit with 2 vectors of width 4) and for *a_Refl* (4 dotVectors units, each with 2 vectors of width 2) with their intermediate or final results. Very important for the GAPP unit configuration is the routing list assigning numbers to input values, constant values and intermediate values. These numbers are used for the input definitions of the DotVectors units. In our example, the needed constants are 0.5 and 1.0. The routing list is the base for the register file as one important component of the GAPPCO I design presented in Sect. 4.

4. GAPPCO I

Here, we describe the first GAPPCO design (GAPPCO I) together with its configuration data in some more detail.

4.1. GAPPCO I architecture

GAPPCO I is a configurable coprocessor consisting of N DotVectors units (Fig. 9).

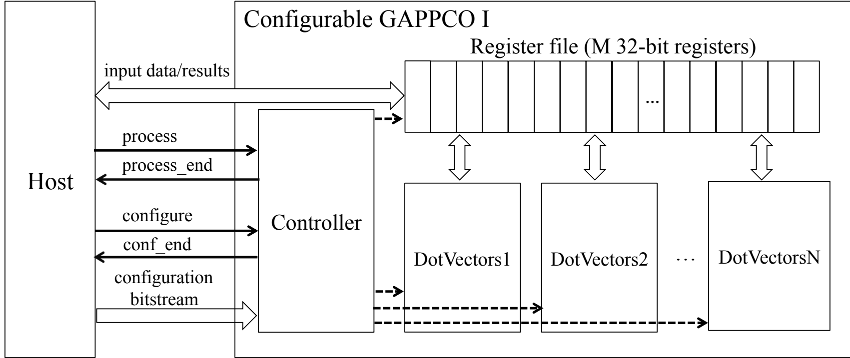

Figure 9. Configurable GAPPCO I block diagram

Each basic DotVectors unit can calculate the sum of 4 products (4-width DotVectors unit). The block diagram of each DotVectors unit is depicted in Fig. 10. It is composed of 4 multiplier units (MULT11, MULT12, MULT13, and MULT14) and 3 adder units (ADD11, ADD12, and ADD13). Each DotVectors unit can be configured as one 4-width DotVectors unit or two 2-width DotVectors units. To make the unit configurable, considering the first level of adders, the output of each adder unit can be either provided as result or used as input of a further adder unit. A demultiplexer unit is used for this purpose. The enable inputs of each demultiplexer are part of the configuration data provided by the GAPP configurator (GAPPConf) on the host side.

This first GAPPCO design uses basic 4-width DotVectors units since this width is sufficient to support reflection operations and therefore the fundamental conformal geometric operations. As described in [6], each conformal geometric operation can be obtained as multiple consecutive reflection operations. The final GAPPCO design will be based on larger DotVectors units so as to support more complex GA algorithms.

The number N of DotVectors units depends on the resource availability on the target FPGA device. As shown in Fig. 9, GAPPCO I also includes a controller unit and a register file consisting of M 32-bit registers to store input data (inputsVector and constants) and output results (both intermediate and final) of the DotVectors units.

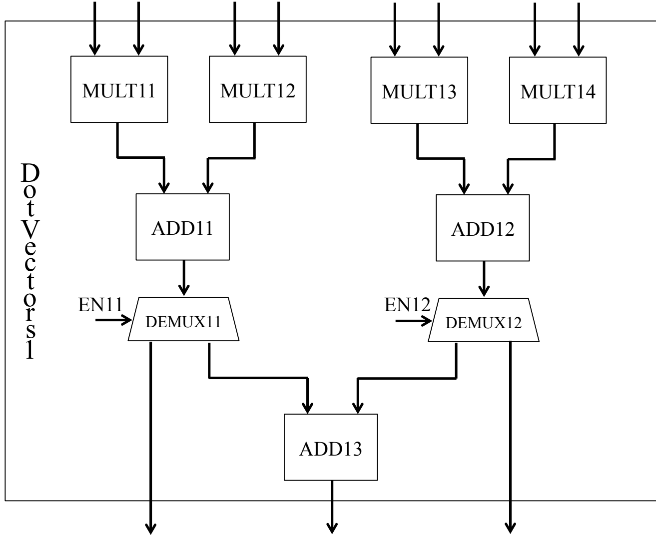

FIGURE 10. DotVectors1 block diagram

4.2. Configuration phase (before runtime)

Before runtime, the GAPP configurator (GAPPConf) on the host side provides the configuration bitstream needed to configure DotVectors units.

When the controller receives the "configure" command from GAPPConf, it configures GAPPCO I according to the configuration bits and sends the "conf_end" status signal to the host (see Fig. 9).

As shown in Fig. 10, each basic DotVectors unit consists of 4 multiplier units, 3 adder units, and 2 demultiplexer units. The first index of each multiplier, adder or demultiplexer unit indicates the number of the DotVectors unit, while the second index specifies the number of the multiplier, adder or demux unit. For each 4-width DotVectors unit, the configuration bitstream is composed of:

- enable inputs of demultiplexer units
- addresses of input operands of each multiplier unit
- signs of input operands of each multiplier unit
- address (addresses) of output result (results)
- result/results type (intermediate or final)

As an example, the configuration bits for the DotVectors1 unit in Fig. 10 are reported in Table 1.

The enable bit values of the demultiplexer units (EN11 and EN12) specify if the DotVectors unit has to be configured as one 4-width DotVectors unit (EN11=EN12=0) or two 2-width DotVectors units (EN11=EN12=1). Furthermore, for each multiplier unit, the configuration bitstream specifies the signs of the input operands (e.g. MULT11_sign1 and MULT11_sign2) as well as the addresses of the registers (within the register file) where the input

operands have to be read (e.g. MULT11_addr1 and MULT11_addr2). If the operand sign is negative (sign bit = 1), a sign changing operation will be performed before the operand is sent to the multiplier unit to be processed. Regarding the register addresses, a register file composed of 32 32-bit registers has been considered for this first design and therefore each address field in the configuration bitstream is composed of 5 bits. Finally, for each configured DotVectors unit, the configuration data specifies the address of the register where the output result has to be written back (e.g. RESULT11_addr) as well as the result type (intermediate calculation or final result). If the DotVectors unit is configured as two 2-width DotVectors units, two results will be provided (RESULT11 and RESULT12). The configuration data are read by the controller unit that configures accordingly the configurable DotVectors units.

TABLE 1. Configuration bits for DotVectors1 of Fig. 10 (A register file composed of 32 32-bit registers has been considered for this first design and therefore 5-bit addresses are needed)

Field	**N. of bits**
EN11	1
EN12	1
MULT11_addr1	5
MULT11_sign1	1
MULT11_addr2	5
MULT11_sign2	1
MULT12_addr1	5
MULT12_sign1	1
MULT12_addr2	5
MULT12_sign2	1
MULT13_addr1	5
MULT13_sign1	1
MULT13_addr2	5
MULT13_sign2	1
MULT14_addr1	5
MULT14_sign1	1
MULT14_addr2	5
MULT14_sign2	1
RESULT11_addr	5
RESULT11_type (intermediate or final)	1
RESULT12_addr (optional)	5
RESULT12_type (optional)	1

This first GAPPCO design is restricted to two types of results, namely, intermediate and final, as needed in the examples chosen for this first proof-of-concept. Therefore, one configuration bit is used to specify the result type

(0 for final results and 1 for intermediate results). However, more complex GA applications need different levels of intermediate results. Final computations need intermediate results of level 1, level 1 computations have to wait for intermediate results of level 2, level 2 computations have to wait for intermediate results of level 3, and so on. In the final GAPPCO design, further configuration bits will be used to specify intermediate results of different levels. As an example, two configuration bits will allow us to define four levels of results: 00 for final results, 01 for intermediate results of level 1, 10 for intermediate results of level 2, and 11 for intermediate results of level 3.

GAPP units of different size each containing a different number of variable-width DotVectors units can be configured within the GAPPCO coprocessor.

The entire configuration bitstream of GAPPCO I will be composed of the fields reported in Table 2. The first 4-bit field of the bitstream specifies the number of GAPP units to be configured within the coprocessor (e.g. r). Furthermore, for each GAPP unit, the configuration bitstream specifies the number of basic 4-width DotVectors units that are needed to obtain the required GAPP unit as well as the configuration bits needed to configure each basic DotVectors unit. The configuration data for each basic DotVectors unit are structured as reported in Table 1. Using the configuration bits listed in Table 2, we can configure a GAPPCO I coprocessor composed of up to 16 GAPP units each consisting of up to 16 4-width DotVectors units.

Table 2. Configuration bitstream of GAPPCO I

Field	**N. of bits**
N. of GAPP units	4
N. of 4-width DotVectors units for GAPP unit 1	4
Configuration bits for DotVectors1 of GAPP unit 1	as listed in Table 1
Configuration bits for DotVectors2 of GAPP unit 1	”
...	...
Configuration bits for DotVectors$n$ of GAPP unit 1	”
N. of 4-width DotVectors units for GAPP unit 2	4
Configuration bits for DotVectors1 of GAPP unit 2	as listed in Table 1
Configuration bits for DotVectors2 of GAPP unit 2	”
...	...
Configuration bits for DotVectors$m$ of GAPP unit 2	”
...	...
N. of 4-width DotVectors units for GAPP unit r	4
Configuration bits for DotVectors1 of GAPP unit r	as listed in Table 1
Configuration bits for DotVectors2 of GAPP unit r	”
...	...
Configuration bits for DotVectors$p$ of GAPP unit r	”

4.3. System operation phases (at runtime)

As shown in Fig. 9, at runtime, when the host sets the "process" control signal, GAPPCO I starts input data processing, and, after completion, sets the "process_end" status signal.

GAPPCO I is based on a pipelined architecture.

System operation phases at runtime are as follows:

1. The host writes input data (inputsVector and constantsVector) to the register file
2. The coprocessor reads input data from the register file
3. The coprocessor executes operations
4. The coprocessor writes results (both intermediate and final) back to the register file
5. The host reads results from the register file

The controller unit supervises the DotVectors units operation. Each DotVectors unit reads its input data from the proper registers of the register file (as set during the configuration phase) and writes the result (results) to the proper register (registers) of the register file. If this result is an intermediate result, it will be further processed by other DotVectors units.

4.4. Example (Reflector)

Fig. 11 shows the GAPP unit configured to execute reflection operations. As required by the reflection operation, three basic DotVectors units are used: the first one is configured as one 4-width DotVectors unit, while the second one and the third one are both configured as two 2-width DotVectors units.

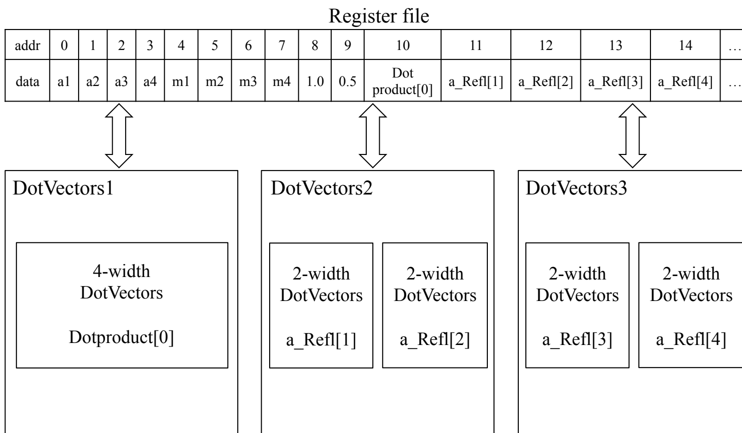

Figure 11. GAPP unit configured to execute reflection operations

The configuration bitstream is reported in Table 3. It is based on the configuration data format presented in Fig. 8.

5. Conclusion

This article presents a new quality of Geometric Algebra hardware solutions. GAPPCO is a hardware design for a coprocessor combining both the advantages of optimizing software with a fixed hardware able to implement arbitrary Geometric Algebra algorithms. The idea is to have a fixed hardware easily and fast to be configured for different algorithms. This article describes the new hardware design together with the complete tool chain for its configuration. While the GAPPCO design of this article is a design for reconfigurable hardware, it is flexible in the sense that it can be used for ASICs and SOCs in the future.

References

[1] 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 *Proceedings of the 10th IEEE Euromicro Conference on Digital System Design - Architectures, Methods and Tools (DSD 2007)*, pages 436–439, Aug 2007.

[2] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. An embedded, fpga-based computer graphics coprocessor with native geometric algebra support. *Integration, The VLSI Journal*, 42(3):346–355, June 2009.

[3] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. Fixed-size quadruples for a new, hardware-oriented representation of the 4d clifford algebra. *Advances in Applied Clifford Algebras*, 21(2):315–340, June 2011.

[4] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. Design space exploration of parallel embedded architectures for native clifford algebra operations. *IEEE Design and Test of Computers*, 29(3):60–69, June 2012.

[5] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. Design and implementation of an embedded coprocessor with native support for 5d, quadruple-based clifford algebra. *IEEE Transactions on Computers*, 62(12):2366–2381, Dec 2013.

[6] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. Conformalalu: A conformal geometric algebra coprocessor for medical image processing. *IEEE Transactions on Computers*, 64(4):955–970, April 2015.

[7] S. Franchini, A. Gentile, F. Sorbello, G. Vassallo, and S. Vitabile. Embedded coprocessors for native execution of geometric algebra operations. *Advances in Applied Clifford Algebras*, 2016.

[8] 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.

[9] 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.

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

[11] Dietmar Hildenbrand, Justin Albert, Patrick Charrier, and Christian Steinmetz. Geometric algebra computing for heterogeneous systems. *Advances in Applied Clifford Algebras Journal*, 2016.

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

[13] 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.

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

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

[16] 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.

[17] Christian Steinmetz. Optimizing a geometric algebra compiler for parallel architectures using a table-based approach. In *Bachelor thesis TU Darmstadt*, 2011.

[18] 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.

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

S. Franchini
Innovative Computer Architectures Lab,
DIID Department,
University of Palermo,
Italy
e-mail: `silvia.franchini@unipa.it`

A. Gentile
Innovative Computer Architectures Lab,
DIID Department,
University of Palermo,
Italy
e-mail: `antonio.gentile@unipa.it`

G. Vassallo
Innovative Computer Architectures Lab,
DIID Department,
University of Palermo,
Italy
e-mail: giorgio.vassallo@unipa.it

S. Vitabile
Department of Biopathology and Medical Biotechnologies,
University of Palermo,
Italy
e-mail: salvatore.vitabile@unipa.it

Table 3. Configuration bitstream for GAPP unit realizing reflector functionality

Fields	**Values**
N. of GAPP units	1
N. of 4-width DotVectors units for GAPP unit 1	3
Configuration bits for DotVectors1	
EN11/EN12	0/0
MULT11_addr1/MULT11_sign1 (0 for +, 1 for -)	7/1
MULT11_addr2/MULT11_sign2	8/0
MULT12_addr1/MULT12_sign1	2/0
MULT12_addr2/MULT12_sign2	6/0
MULT13_addr1/MULT13_sign1	1/0
MULT13_addr2/MULT13_sign2	5/0
MULT14_addr1/MULT14_sign1	0/0
MULT14_addr2/MULT14_sign2	4/0
RESULT11_addr/RESULT11_type (0 for final, 1 for intermediate)	10/1
RESULT12_addr (optional)/RESULT12_type (optional)	-/-
Configuration bits for DotVectors2	
EN21/EN22	1/1
MULT21_addr1/MULT21_sign1 (0 for +, 1 for -)	9/0
MULT21_addr2/MULT21_sign2	0/0
MULT22_addr1/MULT22_sign1	10/1
MULT22_addr2/MULT22_sign2	4/0
MULT23_addr1/MULT23_sign1	9/0
MULT23_addr2/MULT23_sign2	1/0
MULT24_addr1/MULT24_sign1	10/1
MULT24_addr2/MULT24_sign2	5/0
RESULT21_addr/RESULT21_type (0 for final, 1 for intermediate)	11/0
RESULT22_addr (optional)/RESULT22_type (optional)	12/0
Configuration bits for DotVectors3	
EN31/EN32	1/1
MULT31_addr1/MULT31_sign1 (0 for +, 1 for -)	9/0
MULT31_addr2/MULT31_sign2	2/0
MULT32_addr1/MULT32_sign1	10/1
MULT32_addr2/MULT32_sign2	6/0
MULT33_addr1/MULT33_sign1	9/0
MULT33_addr2/MULT33_sign2	3/0
MULT34_addr1/MULT34_sign1	10/1
MULT34_addr2/MULT34_sign2	7/0
RESULT31_addr/RESULT31_type (0 for final, 1 for intermediate)	13/0
RESULT32_addr (optional)/RESULT32_type (optional)	14/0

Table 4. The main commands of the Geometric Algebra Parallelism Programs (GAPP) language; a more detailed list can be found in [17]

Command:	**assignInputsVector**
Syntax:	assignInputsVector inputsVector = $[var_1, var_2, \ldots, var_n]$;
Description:	Assigns scalar inputs variables var_1, var_2, …, var_n to the vector of the inputs.
Arguments:	var_i: the i-th scalar input variable.
Example:	assignInputsVector inputsVector = [x1,x2,x3,y1,y2,y3];
Command:	**resetMv**
Syntax:	resetMv *multivector*;
Description:	Creates a multivector and initializes it with zeros.
Arguments:	*multivector*: the name of the multivector which should be created.
Example:	resetMv v1;
Command:	**setMv**
Syntax:	setMv $dest[sel_{dest_1}, sel_{dest_2}, \ldots, sel_{dest_n}]$ $= src[sel_{src_1}, sel_{src_2}, \ldots, sel_{src_n}]$;
Description:	Copies certain blades from a source to a destination multivector. The order of the selector lists is important.
Arguments:	*dest*: the destination multivector. sel_{dest_i}: the i-th component of the destination multivector. *src*: the source multivector/vector. sel_{src_i}: the i-th component of the source multivector.
Example:	setMv v1[1,2] = inputsVector[0,3];
Command:	**setVector**
Syntax:	setVector $dest = \{arg_1, arg_2, \ldots, arg_n\}$;
Description:	Creates a vector from a list of arguments
Arguments:	*dest*: the destination vector. arg_i: the i-th argument to be assigned in the destination vector, which can be a constant (e.g. 2.0) or an extract from a multivector/vector (e.g. $mv[1,2]$).
Example:	setVector ve0 = {0.5,v1[1,2],v3[1],v1[4,5]};
Command:	**dotVectors**
Syntax:	dotVectors $dest[sel] = \langle vector_1, vector_2, \ldots, vector_n \rangle$;
Description:	Calculates the dot product of the given vectors and stores it in a component of a multivector.
Arguments:	*dest*: the destination multivector. *sel*: the component index of the destination multivector.
Example:	dotVectors p1[4] = ⟨ve0,ve1,ve2⟩;
Command:	**assignMv**
Syntax:	assignMv $dest[sel_{dest_1}, sel_{dest_2}, \ldots, sel_{dest_n}] = [val_1, val_2, \ldots, val_n]$;
Description:	Assigns values to components of a multivector.
Arguments:	*dest*: the destination multivector. sel_{dest_i}: The i-th component of the destination multivector. val_i: the i-th constant value.
Example:	assignMv p1[5,6] = [1.0,3.0];