Implementation of a Clifford Algebra Co-Processor Design on a Field Programmable Gate Array

Christian Perwass
Christian Gebken
Gerald Sommer

ABSTRACT We present the design of a Clifford algebra co-processor and its implementation on a Field Programmable Gate Array (FPGA). To the best of our knowledge this is the first such design developed. The design is scalable in both the Clifford algebra dimension and the bit width of the numerical factors. Both aspects are only limited by the hardware resources. Furthermore, the signature of the underlying vector space can be changed without reconfiguring the FPGA. High calculation speeds are achieved through a pipeline architecture.
Keywords: Clifford co-processor, FPGA.

1 Introduction

Clifford algebra has been applied to many different fields of research, as for example quantum mechanics, theories of gravity, automated geometric reasoning, computer vision and robotics. Clifford algebra is a powerful mathematical tool for symbolic calculations. In order to perform numerical calculations with Clifford algebra, multivectors in $C\ell_n$ have in general to be treated as 2^n dimensional vectors. Therefore, to evaluate the geometric product of two multivectors, 2^{2n} product operations and $2^n(2^n-1)$ additions have to be performed with the multivector elements in the worst case.

Matrix multiplication has a similar computational complexity. Due to the need for very fast matrix multiplications, as for example in computer graphics, hardware implementations of the matrix product were developed. Since Clifford algebra is increasingly used in applied fields where computa-

tional speed is of importance, a hardware implementation of the geometric product and associated operations is of great interest.

We present the design of a Clifford algebra co-processor and its implementation on a Field Programmable Gate Array (FPGA). To the best of our knowledge this is the first such design developed. The design is scalable in both the Clifford algebra dimension and the bit width of the numerical factors. Both aspects are only limited by the hardware resources. Furthermore, the signature of the underlying vector space can be changed without reconfiguring the FPGA. High calculation speeds are achieved through a pipeline architecture.

The difference between the symbolic power of Clifford algebra (CA) and its high computational complexity has long been noted by researchers in this field. In order to numerically evaluate, or solve, symbolically powerful CA equations on a computer, one can often translate them into matrix equations which may then be solved or evaluated with standard matrix libraries. This has the advantage that one can use readily available matrix software packages and specialized matrix processing hardware. However, it may not always be obvious how to express a CA equation as a matrix equation. Furthermore, if one always has to translate CA equations into matrix equations, then the symbolic advantage we have gained by using CA is somewhat lost.

Therefore, many software packages have been developed recently, to evaluate and solve CA equations directly. There are packages for the symbolic computer algebra systems Maple [1, 2] and Mathematica [3], a package for the numerical mathematics program MatLab called GABLE [5], the C++ software libraries CLU [14], GluCat [9], the C++ software library generator Gaigen [6], a Java library [4] and stand alone programs CLUCalc, CLUit [14] and CLICAL [11], to name just a few. For researchers who are interested in the geometric aspects of CA, GABLE, CLUCalc and CLUit also visualize the geometric interpretation of multivectors in particular spaces.

When working with matrices one can take advantage of hardware accelerated matrix multiplications. Our goal was to see to what extend a hardware implementation of CA operations is indeed feasible and will speed up the evaluation process.

In this paper we can only give an overview of the design of the co-processor. Only some aspects are discussed in more detail. For a complete, detailed report (in German) see the Diploma thesis of Christian Gebken [15].

We will not give an introduction to Clifford algebra here. Introductory material can be found for example in [7, 10, 12, 16]. However, we give a short introduction to the geometric product, since this is the main Clifford algebra operation to be implemented by the co-processor. Let the basis of a universal Clifford algebra $C\ell_n$ be given by the set $\mathcal{B}_n = \{E_i\}$, which consists of 2^n basis blades. A multivector $A \in C\ell_n$ is then given by $A =$

$\sum_{i=1}^{2^n} \alpha^i E_i$, with $\alpha^i \in \mathbb{R}$, $\forall\, i \in \{1, \ldots, 2^n\}$. If we agree on a particular basis $\mathcal{B}_n$ of $C\ell_n$, we can therefore represent the multivector A by the vector $(\alpha^1, \alpha^2, \ldots, \alpha^{2^n})$.

The geometric product of two elements of $\mathcal{B}_n$ results again in an element of $\mathcal{B}_n$, up to a sign. The relationship can be expressed with a tensor as follows: $E_i E_j = \sum_{k=1}^{2^n} {g_{ij}}^k E_k$, where the entries of ${g_{ij}}^k$ can be 1, -1 or zero. If $\mathcal{B}_n$ is the basis of a universal Clifford algebra, which we assumed, then the geometric product of basis blades is invertible. Let the three multivectors $A = \sum_{i=1}^{2^n} \alpha^i E_i$, $B = \sum_{i=1}^{2^n} \beta^i E_i$ and $C = \sum_{i=1}^{2^n} \gamma^i E_i$ be related by the geometric product $AB = C$. Then the relationship between their scalar components is given by $\gamma^k = \sum_{i=1}^{2^n} \sum_{j=1}^{2^n} \alpha^i \beta^j {g_{ij}}^k$. This relationship can be used to solve multivector equations [13]. Many software packages use pre-calculated multiplication tables, i.e. the ${g_{ij}}^k$, to implement the geometric, inner and outer product. As will be discussed later, the co-processor represents multivectors as lists of basis blades with associated scalar factors. The geometric product of basis blades will be evaluated explicitly, without a multiplication table, which is more effective in this case.

An explicit evaluation of the geometric product of two blades is done as follows. Given an orthonormal basis $\{e_1, e_2, \ldots, e_n\}$ of a vector space $\mathbb{R}^n$, two blades of the Clifford algebra $C\ell_n$ over $\mathbb{R}^n$ may, for example, be given by $a = \alpha\, e_1 e_2 e_4$ and $b = \beta\, e_1 e_3 e_4$, with $\alpha, \beta \in \mathbb{R}$. Their geometric product may then be evaluated as follows. First of all, the scalar factors of the blades can be multiplied separately, i.e. $ab = (\alpha\beta)\,(e_1 e_2 e_4 e_1 e_3 e_4)$. The resultant blade component can be evaluated by applying the associativity of the geometric product and the rules $e_i e_i = 1$ and $e_i e_j = -e_j e_1$. Note that $e_i e_i = -1$ is also possible, depending on the signature of the vector space. The resultant blade component therefore is $-e_2 e_3$, and thus $ab = -(\alpha\beta)\, e_2 e_3$.

2 Hardware Designs

The goal is to numerically evaluate CA operations like the geometric product with a digital circuit design. Before we discuss the evaluation of a CA product itself, we have to decide on what the Clifford co-processor should be able to do. This, of course, influences the design of the whole chip. Since we want to build a co-processor on an FPGA which is external to the CPU, we do not want to constantly transfer data between the CPU and the FPGA. Therefore, the co-processor should have a list of operations and multivectors which it can evaluate independently from the CPU. The possible operations should allow us to evaluate most CA expressions. Therefore, we want the processor to have the following features. 1. Evaluation of the geometric, inner and outer product, addition and subtraction.

2. Operations between given multivector and previous result. 3. Operations between two previous results. 4. Choice of order of mutlivectors in operation.

This list of features makes certain demands on the arithmetic logic unit (ALU) of the co-processor. Before we discuss possible ALU designs, we will give a short overview of how a FPGA actually works.

The FPGA we used was a Xilinx XC4085XLA-0.9. This FPGA contains 3136 configurable logic blocks (CLBs) which are connected in a configurable array. Each CLB contains two delay flip-flops (DFF), two look up tables (LUT) and a fast carry logic for arithmetic operations. A DFF can be used as a register to store information and a LUT allows logic functions up to 4 bit to be implemented, like AND, OR, XOR, etc. and combinations of these. Using the CLBs and the configurable connection array, any digital circuit design can be implemented.

Note that this FPGA does not contain any predefined arithmetic units for multiplication or addition. These have to be implemented by a designer. A single 32×32 bit floating point multiplier would already use up most, if not all, available CLBs of this FPGA.

Although the internal clock of the FPGA can run at more than 100 MHz, the maximal achievable frequency for a particular design may be much lower. In our case there was an additional problem due to the FPGA's external RAM on the FPGA board, which only allowed simultaneous read/write access at 20 MHz.

We programmed the FPGA design using the C++ hardware description language (CHDL) [8]. CHDL is a C++ software library which allows the programmer to define a digital circuit using C++ objects and operators. This simplifies the design process compared to the standard hardware description language VHDL.

In general the clock frequencies of FPGAs cannot be as high as those of non-configurable ICs, due to the FPGA's configurable connections between CLBs.

Despite these problems, FPGAs are certainly a good choice to build and test a Clifford co-processor prototype. Furthermore, in certain applications we might want to use different digital circuit designs on the same FPGA consecutively, to solve different aspects of a problem. Here the Clifford co-processor may be a design which could follow some pre-processing operations.

We will now discuss two possible ALU designs.

2.1 Direct Computation

This design expects as input two complete mutlivectors. Each operation between basis blades is hardwired, so that all necessary operators exist on the chip simultaneously. This allows for pipelined, parallel processing, but it also needs a large amount of resources.

As an example take the geometric product. For each multiplication between basis blades there exists a separate multiplier which is hardwired to the appropriate multivector elements. Its output is connected to an adder which collects all blade multiplications which result in the same blade. In $C\ell_n$ we would need 2^{2n} multipliers. Since there are 2^n different basis blade combinations whose products result in the same basis blade, each of the 2^n resultant basis blades is the sum of 2^n values. These have to be added in a cascade of $\sum_{i=1}^{n-1} 2^i$ adders. Therefore, we have a total of $2^n \sum_{i=1}^{n-1} 2^i$ adders. For 3d-space this gives 64 multipliers and 48 adders. For 5d-space we already have 1024 multipliers and 960 adders. Even the most advanced FPGAs available today do not have enough capacity to deal with the 5d case. Furthermore, a change of signature or dimension would mean that we have to reconfigure the chip.

Although this would be the simplest and fastest implementation, we did not follow this approach due to its enormous need of resources and its inflexibility.

2.2 Basis Blade Pipeline Design

A much more flexible design is that of basis blade pipelines. Here we have a number of pipelines which each deal with an operation between two basis blades. The number of pipelines per operation depends on the amount of resources available. Of course, there has to be additional logic which distributes the different combinations of basis blades between the different pipelines and collects the results appropriately. This does in fact cause some non-trivial problems, which will be discussed later on. Note that at each clock tick (the moment when the clock changes from low to high) we can push a new basis blade combination into every pipeline. This means that if a basis blade pair needs n clock cycles to be processed by a pipeline, we initially have to wait n clock cycles for the first result to appear. However, after this time we obtain a new result after each clock cycle.

In this setup, the geometric product could be evaluated using two different methods: a multiplication table or an explicit calculation. For software packages a multiplication table is the easiest and also a very efficient solution. Here this is not the case, since a multiplication table would have to be stored in memory. On the FPGA we used the memory could only be accessed serially, which would not allow any parallel processing. Furthermore, other parts of the design need to access the memory at the same time, which would have slowed down the processing even further. Therefore, we decided on evaluating the geometric product explicitly. This is discussed in some detail later on.

For example, to evaluate the geometric product each pipeline only needs a single pipelined multiplier and a single adder to add the result to the appropriate result blade. Depending on the amount of resources available we can vary the number of pipelines working in parallel. In fact, we could

even have a number of ALUs working in parallel, although then we might run into additional trouble due to interdependencies of the operations.

Due to its flexibility and extensibility we chose this ALU design for the Clifford co-processor.

2.3 Co-processor Design

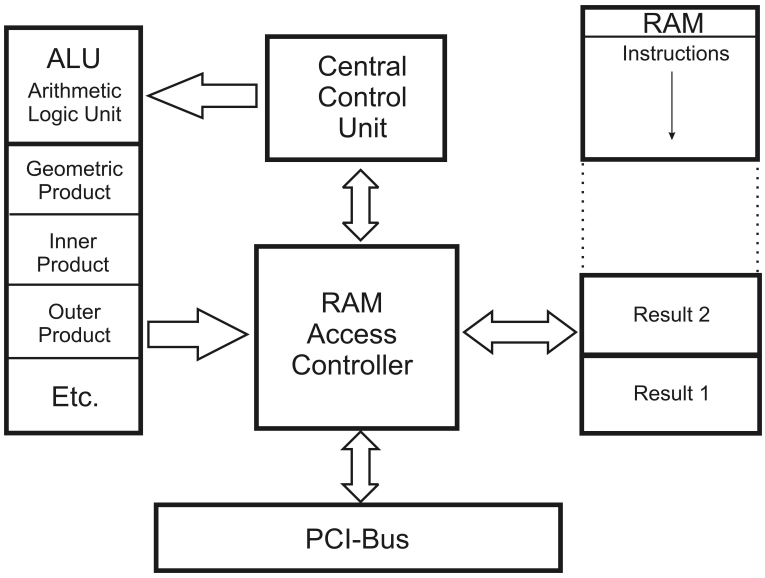

FIGURE 1. General data flow in CA co-processor.

Figure 1 shows the overall data flow of the co-processor. We assume here that the FPGA which implements the co-processor sits on a PCI card with its own memory and is accessible through the PCI bus. The RAM Access Controller (RAM-AC) has two main objectives. First of all it is used to transfer data from the main board memory to the FPGA memory and vice versa. Secondly, it allows the Central Control Unit (CCU) to access the instructions and previous results. Furthermore, it allows the ALU to write evaluation results to the appropriate addresses of the on-board RAM.

The CCU takes an instruction from the RAM, and feeds the appropriate evaluation pipelines of the ALU with basis blade pairs. The ALU accumulates the evaluation results in a result area in the RAM. More details are given in the next section.

3 Implementation

The FPGA available for the design implementation was rather small, which meant that we had to make some restrictions.

1. We could only implement a single basis blade pipeline,
2. scalar factors of basis blades are 24 bit integer numbers,
3. Clifford algebras of up to 8d-vector spaces can be used,
4. only the geometric product has been implemented.

3.1 Basis Blades and Instructions

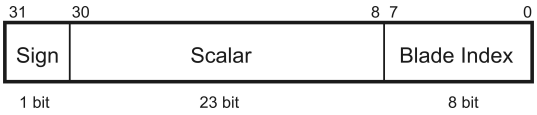

FIGURE 2. Structure of a basis blade.

Each basis blade consists of a scalar factor and an algebra component. This is shown in figure 2. The scalar component is a 23 bit integer value plus a sign bit. The blade index follows a well known method used to express basis blades in binary code. Each bit in the blade index stands for a basis vector. If a bit is high, the corresponding basis vector exists in the blade, otherwise it does not. In this way all basis blades of a Clifford algebra can be represented. For example, in $C\ell_8$ the basis blade $e_1e_3e_8$ is represented by the binary code 10000101 . Each blade has a unique blade index, so that the latter can be used as a memory offset address.

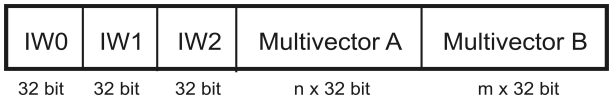

FIGURE 3. Structure of an instruction

Each operation instruction consists of three instruction words (32 bit each) and the corresponding multivectors, as show in figure 3. Each multivector consists of a collection of basis blades as shown in figure 2. The number of elements within the multivectors is contained in the instruction words IW1 and IW2, respectively. Furthermore, the operand multivectors do not have to follow the instruction words. The operands can also be result multivectors from previous operations. The structure of the instruction words is as follows.

IW0 Bit	Description
0-9	Offset to the start of the next instruction
11-14	ID of operation type
15-18	Flags
19-31	Reserved

IW1 Bit	Description
0-22	Address of A-operand (first multivector)
13-31	Number of basis blades in A-operand

IW2 Bit	Description
0-22	Address of B-operand (second multivector)
13-31	Number of basis blades in B-operand

The elements of IW0 are fairly self explanatory. The first 10 bit contain the offset to the next instruction. This can be variable, since the number of elements in the multivectors passed with the instructions is variable. There are four bit for the operation type ID, which allows for 16 different operations. Currently only the geometric product has been implemented. The next four bits contain flags which indicate whether this is a proper instruction or the end of the instruction list. Furthermore, there is a flag which tells the CCU whether each element of operand A is multiplied with all elements of operand B or vice versa. That is, the roles of A and B can be switched.

Note that the addresses of the A and B operands are not given as relative address offsets but as absolute values. This has been done to simplify the design somewhat. However, it also means that the multivectors passed with the instructions and the result multivectors have to be at previously known addresses. This can be achieved as follows.

When preparing an instruction list, we know how many result multivectors we will obtain. Since each result multivector can have at most $2^8 = 256$ basis blades, we know how much memory we have to reserve for each ($256 \cdot 4$ bytes $= 1024$ bytes). By placing the n^{th} result multivector memory block at the highest memory address minus $n \cdot 1024$ bytes, we have a simple way of knowing the address of a result multivector when preparing the instruction list. Note that the instruction list itself starts at the lowest memory address, as indicated in figure 1.

3.2 The Geometric Product Pipeline

Due to size restrictions of the FPGA we used, only one geometric product pipeline could be implemented. At this point we assume that the CCU has fed the pipeline with an appropriate pair of basis blades and the resultant basis blade will be added to the correct result multivector.

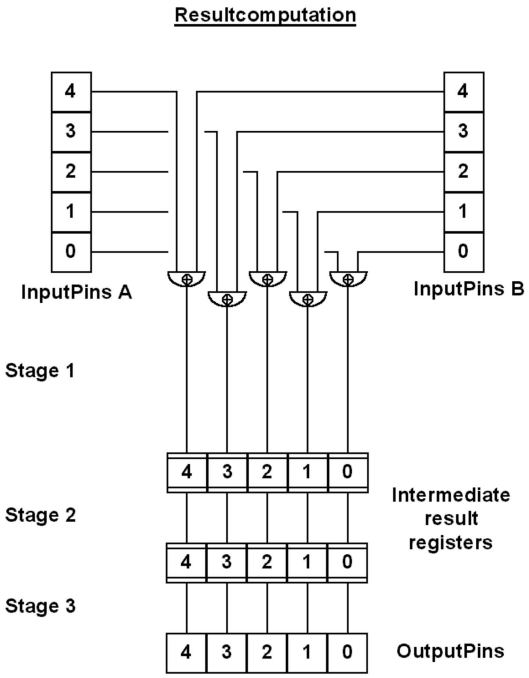

FIGURE 4. Geometric product without sign contributions.

In order to evaluate the geometric product of two basis blades, we have to perform two main operations: the scalar parts of the two basis blades have to be multiplied with a standard multiplier and the blade parts (the blade indices) have to be multiplied with the geometric product operation. If we disregard the sign for a moment, the geometric product of blade indices is simply a XOR operation: if both basis blades contain the same basis vector (e.g. e_1e_2 and e_1e_3 both contain e_1), then this basis vector (e_1 in this case) squares to unity, otherwise the basis vector remains (e_2 and e_3 in this case). This is shown in figure 4.

For clarity, we have only drawn the digital circuit for the lower 5 bit of a basis blade index. Empty half circles closed by a straight line symbolize AND gates and if they contain an encircled plus they represent XOR gates. The square boxes represent registers. Registers basically load the value at their input at a each clock tick and store it until the next. In this way one can realize a pipeline design.

Due to hardware restrictions, only a limited number of logic gates can be evaluated consecutively within a clock cycle. In order to have a synchronized, pipelined design, registers have to be inserted after a certain number of consecutive logic gates.

In figure 4 we have shown a design with three stages. That is, we need three clock cycles to process the data applied at the input pins. However,

at each clock tick a new set of data can be applied to the input pins since the intermediate results of the previous data sets are stored in the registers.

Evaluating the appropriate sign is what makes the geometric product somewhat more complicated. There are three contributions to the sign: the sign of the scalar factors, the sign due to the signature of the basis vectors and the sign due to the swapping of basis vectors. The sign due to the scalar factors is taken care of by the scalar multiplier, which we will not discuss.

Signature

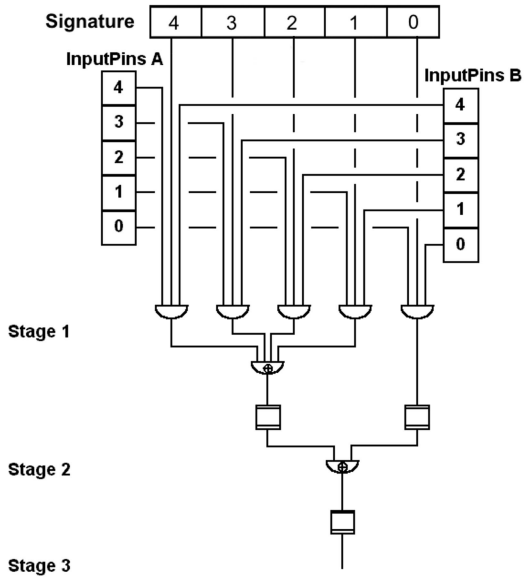

FIGURE 5. Evaluation of sign due to signature.

The sign due to the signature of two multivectors is evaluated as shown in figure 5. The co-processor has an 8 bit register which stores the signature of the basis vectors, and which can be set before an instruction set is executed. If a bit in the sign register is high, then the corresponding basis blade squares to minus one. Otherwise it squares to plus one. Hence, the circuit in figure 5. If both basis indices have a common basis vector, i.e. a common high bit, then the squaring of these basis vectors contributes a minus if the corresponding signature bit is high. This is achieved through the AND gates. The XOR gates combine all the separate minus signs. They return low (plus) if there is an even number of minus signs and otherwise high (minus).

Swapping

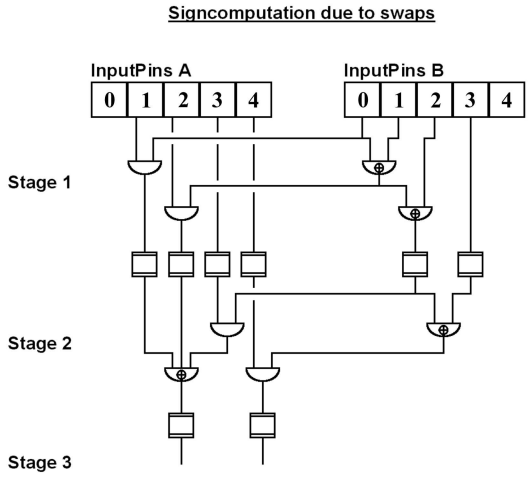

FIGURE 6. Sign evaluation due to basis vector swaps.

The digital circuit evaluating the sign due to the swapping of basis vectors is shown in figure 6. For example, if we want to evaluate $(e_1e_2)(e_1)$ we write $e_1e_2e_1 = -e_2e_1e_1 = -e_2$, where we have made one swap, exchanging e_1 and e_2. The number of swaps necessary can be evaluated by a XOR cascade. Input pin 0 stands for e_1, input pin 1 for e_2, and so on. If in both operands input pin 4 is high then both operands have an e_5 component. Before we can square these we potentially have to swap e_5 of operand A with e_1, e_2, e_3 and e_4 of operand B. If an odd number of these are actually present in operand B then a minus is introduced, otherwise not. This is achieved by the cascade of XORs. The combination of the different swap signs is done in stages 2 and 3.

Sign combination

The different sign contributions due to the geometric product are finally combined in stage 3 as shown in figure 7. This circuit shows the final stage of the circuits drawn in figures 5 and 6. Of course, this resultant sign has to be XORed again with the resultant sign of the scalar multiplication.

3.3 Inner and Outer Product

The inner and outer product can be evaluated in a very similar manner to the geometric product. In fact, they become the geometric product if certain conditions are satisfied. An implementation of these products would therefore only introduce an additional step, where these conditions are

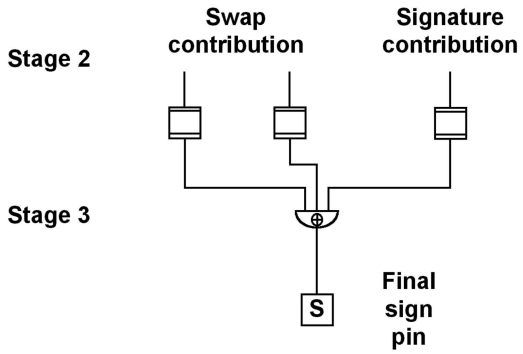

FIGURE 7. Combination of signs due to geometric product.

checked and then either zero is returned directly or the geometric product is evaluated.

The inner product of two basis blades is only non-zero if one basis blade is completely contained within the other. We can check this by evaluating operand A AND operand B, which has to be equal either to operand A or to operand B. If this is the case then the inner product of the two operands is simply the geometric product.

The outer product of two basis blades is only non-zero if they have no basis vector in common. We can check this with a simple AND operation. Again the outer product becomes the geometric product if the two operands have no element in common.

Currently, we have not implemented the inner and outer product.

3.4 Geometric Product Evaluation

In the following we discuss how the geometric product of two multivectors is evaluated by the co-processor. The three main steps are:

1. The CCU takes the current set of instruction words and initializes counters with the number of A and B operands (basis blades).

2. For each A operand the CCU loops through all B operands and passes each pair to the ALU. If an operand is zero the A and B operands are still passed to the ALU but they are marked as invalid, which means that they will not be added to the result multivector.

3. The blade indices of the resultant basis blades that exit the ALU pipeline are used to load the appropriate basis blade from the result multivector memory block. The two basis blades are then added and written back to memory.

Steps 2 and 3 are executed pipelined. That is, at each clock tick a new pair of basis blades is passed to the ALU. If a valid result is available at the

ALU output it is added to the result multivector memory block in parallel to the ALU execution.

There are a number of difficulties which mainly have to do with the pipeline synchronization, which we cannot discuss here in detail. However, there is one issue which should be mentioned. For a fixed A operand the geometric product with each B operand will create a different basis blade. Although loading and writing basis blades from and to the result multivector memory block takes a couple of clock cycles, this only introduces a constant delay. That is, we have to store the results of the ALU in a FIFO until the corresponding basis blades from the result multivector memory block arrive. They are then added and written back to memory.

However, if we are at the end of the B operand loop, the A operand changes. This means that, by chance, the new combination of A and B operands might result in the same basis blade as the last combination of operands from the previous B operand loop. Therefore, the CCU would load the same basis blade from the result multivector memory block a second time, before the first basis blade could have been added to it and written back. This is a so called *read after write* conflict: we read obsolete data before it has been updated.

The design we implemented recognizes such conflicts. Any blade pair that causes a conflict is stored in a FIFO and is executed at the end of the B operand loop.

4 Evaluation and Conclusions

In order to test the evaluation speed of the co-processor and the software packages CLU and Gaigen, we evaluated the geometric product of multivectors that contained all basis blades of the respective Clifford algebra. Of course, this does not test how the skipping of zero basis blades in multivectors is handled by the co-processor and the software packages. However, if we wanted to test this, the question would be what a realistic distribution of basis blades in multivectors is. The benchmark we used is nonetheless one indicator of the performance of the different packages.

Due to hardware restrictions we could only run the FPGA at 20MHz. CLU and Gaigen were tested on 1.5 GHz machines. In real terms, both software packages were much faster than our implementation of the co-processor. However, in order to see whether the hardware implementation does offer an advantage in principle, we have to compare the results at the same frequency. The results are shown in figure 8, where the x-axis gives the dimension of the vector space over which the Clifford algebra is formed and the y-axis gives the number of geometric operations per second (GOPS), i.e. the number of geometric products between multivectors. It can be seen that the optimized code generated by Gaigen is nearly as

fast as the co-processor. The co-processor is about 1.5 times faster than Gaigen. Since, so far, the co-processor does not make any use of parallel pipelines, and modern CPUs also use pipeline processing, this shows that programming a modern CPU can be about as efficient as a pipelined hardware implementation.

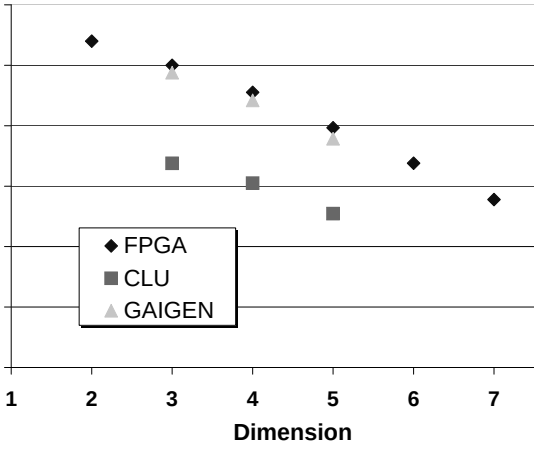

FIGURE 8. Evaluation speed benchmarks. GOPS stands for Geometric Operations Per Second.

The main advantages of a hardware implementation of Clifford operations are the following.

- If the co-processor is implemented on an application specific integrated circuit (ASIC) or FPGA, it can run in parallel to the main CPU. That is, the CPU can deal with other things, while the Clifford operations are evaluated.

- If enough resources are available, a number of evaluation pipelines can work in parallel. There could even be a number of parallel instruction pipelines. This could offset the low FPGA clock frequencies.

- A Clifford co-processor could be integrated into the main CPU, just as multimedia operations have been (e.g. MMX).

In conclusion we can say that a Clifford co-processor implementation on even the most modern FPGAs might only be about as fast as an optimized software implementation of Clifford operations. This is mainly due to the comparatively low clock frequencies and a lack of direct support of basic arithmetic operations like floating point multiplication and addition, which are essential for many applications.

Of course, it would be most desirable to have Clifford operations available as part of a standard CPU. However, this will only occur if Clifford operations offer a functionality that is needed by many (profitable) applications, which cannot easily be provided through other mathematical tools, like matrices.

References

[1] R. Abłamowicz, Clifford algebra computations with Maple, *Clifford (Geometric) Algebras,* Banff, Alberta Canada, 1995, Ed. W. E. Baylis, Birkhäuser, Boston, 1996, 463–501.

[2] R. Abłamowicz, B. Fauser, The CLIFFORD Home Page, `math.tntech.edu/rafal/cliff5/index.html`, last visited 15. Sept. 2003.

[3] J. Browne, The GrassmannAlgebra Book Home Page, `www.ses.swin.edu.au/homes/browne/grassmannalgebra/book/`, last visited 15. Sept. 2003.

[4] A. Differ, The Clados Home Page, `sourceforge.net/projects/clados/`, last visited 15. Sept. 2003.

[5] L. Dorst, The GABLE Home Page, `carol.wins.uva.nl/~leo/GABLE/`, last visited 15. Sept. 2003.

[6] D. Fontijne, The Gaigen Home Page, `carol.wins.uva.nl/~fontijne/gaigen/`, last visited 15. Sept. 2003.

[7] D. Hestenes, G. Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics.* Dordrecht, 1984.

[8] K. Kornmesser. The CHDL Home Page, `www-li5.ti.uni-mannheim.de/fpga/?chdl/`, last visited 15. Sept. 2003.

[9] P. Leopardi, The GluCat Home Page, `glucat.sourceforge.net/`, last visited 15. Sept. 2003.

[10] P. Lounesto. *Clifford Algebra and Spinors.* Cambridge University Press, 1997.

[11] P. Lounesto, The CLICAL Home Page, `www.helsinki.fi/~lounesto/CLICAL.htm`, last visited 15. Sept. 2003.

[12] C.B.U. Perwass, *Applications of Geometric Algebra in Computer Vision.* PhD thesis, Cambridge University, 2000.

[13] C.B.U. Perwass, G. Sommer, Numerical evaluation of Versors with Clifford Algebra, in *Applications of Geometric Algebra in Computer Science and Engineering,* Eds. Leo Dorst, Chris Doran, Joan Lasenby, Birkhäuser, 2002, pages 341–349.

[14] C.B.U. Perwass, The CLU Home Page, `www.perwass.de/cbup/clu.html`, last visited 15. Sept. 2003.

[15] C. Gebken, *Implementierung eines Koprozessors für geometrische Algebra auf einem FPGA*, Diploma thesis, Christian-Albrechts-University Kiel, 2003.

[16] G. Sommer, editor. *Geometric Computing with Clifford Algebra.* Springer Verlag, 2001.

5 Information about the Authors

Christian Perwass
Institut für Informatik
Christian-Albrechts-Unversität Kiel
24105 Kiel, Germany
E-mail: chp@ks.informatik.uni-kiel.de

Christian Gebken
Institut für Informatik
Christian-Albrechts-Unversität Kiel
24105 Kiel, Germany
E-mail: chg@ks.informatik.uni-kiel.de

Gerald Sommer
Institut für Informatik
Christian-Albrechts-Unversität Kiel
24105 Kiel, Germany
E-mail: gs@ks.informatik.uni-kiel.de

Submitted: August 31, 2002.

Index