Acceleration and Energy Efficiency of A Geometric Algebra Computation using Reconfigurable Computers and GPUs

Holger Lange
LOEWE Research Center AdRIA
Technische Universität Darmstadt
lange@esa.cs.tu-darmstadt.de

Florian Stock, Andreas Koch
Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt
{*stock*|*koch*}*@esa.cs.tu-darmstadt.de*

Dietmar Hildenbrand
Interactive Graphics Systems Group (GRIS)
Technische Universität Darmstadt
dhilden@gris.informatik.tu-darmstadt.de

Abstract

Geometric algebra (GA) is a mathematical framework that allows the compact description of geometric relationships and algorithms in many fields of science and engineering. The execution of these algorithms, however, requires significant computational power that made the use of GA impractical for many real-world applications. We describe how a GA-based formulation of the inverse kinematics problem from computer animation and robotics can be accelerated using reconfigurable FPGA-based computing and using a graphics processing unit (GPU). The practical evaluation covers not only the sheer compute performance, but also the energy efficiency.

1. Introduction

Geometric Algebra (GA) is a mathematical framework for the concise description of complex geometrical relationships. The execution of these algorithms is highly compute intensive, which is one of the reasons that GA has only seen limited use in real-world applications.

Reconfigurable Adaptive Computer Systems (ACS) and General Purpose Computing on Graphics Processing Units (GPGPUs) both offer compute performance beyond that of conventional processors (CPUs), but normally use very different models of computation. GPUs generally support a coarse grained SIMD approach [1], while ACSs additionally allow a much finer parallelism at the instruction or pipeline level [2]. Other differences include the clock frequencies and the power consumption.

The GA-based formulation of the inverse kinematic problem, which is a standard problem in the computer animation and robotics, is significantly shorter and much more accessible than the traditional one. We will then use it as a benchmark for acceleration both on an FPGA-based ACS as well as a current generation GPU. With the increased importance of power and thermal budgets in high-performance computing, especially in embedded environments, our experimental evaluation will not just cover the sheer compute performance, but also extend to energy efficiency.

2. Geometric Algebra

GA unifies many other mathematical concepts like imaginary numbers, quaternions or projective geometry. It is based on the work of Hermann Grassmann and William Clifford ([3], [4]). Pioneering work has been done by David Hestenes, who firstly applied Geometric Algebra to problems in mechanics and physics [5], [6].

GA is able to easily describe and manipulate high-level geometric objects like spheres, circles and planes as well as operations combining objects. By composing additional primitives such as points and planes, applications from diverse domains are easily formulated. Examples include GA Fourier transforms, or the classification and clustering of spatial patterns with GA [7], or the inverse kinematics application which will be discussed later.

3. Related Work

Despite the tremendous expressive power of the geometric algebra (GA), it has only seen very limited practical use. One of the reasons for this might be that the execution or evaluation of GA algorithms (actually transforming coordinates) requires significant computational effort. To resolve this quandary, it is promising to look at dedicated hardware architectures for the acceleration of this computation. Current integrated circuit technology offers a means to achieve this in the form of FPGAs.

Prior attempts to accelerate GA computations include composing GA specific hardware blocks using PROLOG ([8]), hardware acceleration of just the geometric product [9], and GA operations which were decomposed into microcode controlled primitive calculations ([10], [11], [12], [13]).

4. Acceleration Approaches

When studying the prior attempts, it is obvious that most of them lead to an application slowdown instead of the hoped-for acceleration. The major reason for this disappointing result is due to the architectural choices made.

Our first implementation (Sec. 5) is completely different architecturally from the accelerator approaches mentioned above and takes advantage of a high degree of parallelization and pipelining. Instead of coarse granular computation units capable of handling entire GA operators, we decomposed the GA description into the underlying scalar equations, which employ only basic arithmetic operators. The set of equations from the GA model was implemented one arithmetic operator at a time. For each of these arithmetic operators we carefully examined the range of values to be processed for the specific problem. With this data, and external requirements on computational precision, we determined for each operator the optimal numerical representation (e.g., values in the range of 0 to 100 mm with 1/16 mm of accuracy would be represented as 11 bit unsigned fixpoint numbers). The circuits of the operators were then optimally matched to their representation as well as to one of their operands being the constant.

The second implementation (Sec. 6), however, trades fine-grained parallelism and custom arithmetic for the brute-force SIMD parallelism achievable with a modern GPU.

The GA algorithm for the inverse kinematics [14] leads to a set of equations describing a function $f : \mathbf{R}^3 \mapsto \mathbf{R}^6$, a mapping of the 3D coordinates of the target point to the screw and curl angles of the arm's shoulder and elbow joints, expressed as quaternions. These equations were manually optimized for performance, exploiting algebraic equalities (e.g., the distributive law) and common subexpressions elimination (this saved up to 50% of some operators).

5. FPGA Implementation

For the FPGA implementation, the optimized GA equations were translated into a dataflow graph (DFG).

To reduce the required ressources on the FPGA, we use only fixed point calculations instead of floating point operations (which is possible but not as efficient as fixed point). As a side effect of calculating in dedicated arithmetic hardware, the cost of operations drastically changes: e.g., constant multiplication and addition need the same calculation time, whereas reciprocal value, division and square root are more costly in both execution time and FPGA resources.

For the optimization of numerical types on the FPGA implementation, we thus analyzed the function domain and ranges with regard to the required numerical precision. The animation model has a positional accuracy of 1/16 mm (which thus is the upper bound of the required precision of the result). For the fitting of word-lengths of individual operators to achieve this result precision, we employed two methods: an analytical approach (using general and domain specific knowledge) and an empirical (Monte-Carlo) approach for cases where the analytical approach does not yield satisfying results and for overall verification.

Note that some attempts at automatically performing this optimization exist (e.g., [15]), but they are often limited with regard to the operators supported.

Consider the high degree of parallelism in this fully spatial pipeline: When it is filled in steady-state, all 140 operators in all 365 pipeline stages compute in parallel. This far exceeds the capabilities even of modern superscalar processors which can handle about half a dozen parallel operations per clock cycle [16, Chapter 3.6].

6. GPGPU Implementation

FPGAs are no longer the only easily accessible way to exploit parallelism. In recent years the architecture of Graphics Processing Units (GPUs) has advanced from mostly fixed-function graphics pipelines to increasingly flexible processor arrays. The latter now allow *general purpose* computing on GPUs using dedicated languages such as Brook+, Rapidmind, or CUDA.

For our experiments, we used the Compute Unified Device Architecture (CUDA) from NVIDIA Inc. [1], which extends the C language and provides a library for standard tasks, e.g., determining GPU characteristics or performing memory transfers.

NVIDIA refers to the CUDA compute model as *SIMT*: Single Instruction Multiple Threads. It addresses a device as an array of processing elements operating in a SIMD manner, thus called multiprocessors.

A multiprocessor can schedule instructions from up to 512 different threads on its internal eight datapaths, aiming to hide long latencies (e.g., accesses to external memory). CUDA has certain limitations due to its architecture [1], but those are not relevant to our implementation, as we use a multi-threaded approach that executes independent, data flow only computations (for different target points) in parallel.

We implemented a GPU kernel which computes the inverse kinematics function f. As it contains no control flow the computation requires no thread synchronization. By passing the three input parameters and the six output parameters as separate arrays (struct of arrays instead of array of structs) the parallel threads access sequential memory addresses, that can be coalesced. The kernel uses 31 registers per kernel, so with the 16384 registers available per block (in the NVIDIA GTX 280 we employed), we can actually start threads up to the device limit of 512 threads per block. Neither local memory nor shared memory is accessed while computing.

The high-end NVIDIA GTX 280 card we used has 30 multiprocessors of eight scalar datapaths each. Each of the 30 multiprocessors handles one block of data, and can schedule instructions from one of 512 threads on its datapaths. Thus, the GTX 280 executes our application in 15360 independent threads, each processing 1/15360th of the total input data set.

Table 1: Mapping results on XILINX 5VLX155

Frequency	170 MHz	# LUTs	34912 (35%)
Throughput	1 eval / cycle	# FFs	49938 (51%)
Latency	365 cycles	# DSPs	74 (57%)

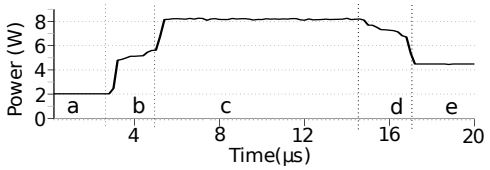

Figure 1: Power consumption of the FPGA implementation

Since the GPU executes single-precision floating point as quickly as integer operations, the fixed-point optimizations we performed for the FPGA are thus not necessary. We do, however, use the target-independent optimizations described at the end of Sec. 4.

7. Experimental Results

In this section, we evaluate the performance and power consumption not only of the FPGA and GPU implementations, but for completeness also cover software running on conventional processors.

The Verilog HDL description of the DFG was mapped to a Xilinx Virtex 5 LX155 FPGA [17] using Synplify Premier 9.4 and Xilinx ISE 10.1.03. Tab. 1 shows the mapping and performance results for the complete DFG.

The power consumption of the FPGA, shown in Fig. 1, was estimated with the Xilinx XPower Analyzer using a complete signal change dump from post-layout simulation. In this scenario the FPGA is first reset, shown as time interval *a* in the Figure. The clock is stopped since the clock manager is being held in reset, yielding a quiescent power of $\approx$ 2 W. In interval *b*, the pipeline gradually fills when random values are applied to its inputs. Hence, an increasing number of flip-flops toggle, manifesting in a rise of power consumption. Interval *c* shows the device in steady-state operation with the whole pipeline calculating in parallel. Here, the power consumption peaks at 8.3 W. In interval *d* the pipeline is drained, with inputs held constant. Most of the flip flops now gradually stop toggling, contributing to a declining power intake. Power consumption drops further to 4.5 W afterwards (in interval *e*), but remains higher than in the reset-state since the clock net is still active. The steep gradients in intervals *b* and *d* are attributable to the I/O pins starting/stopping to toggle and driving output loads.

Our GPU implementation performs the complete calculation for all target points as one kernel on the GPU to avoid the high overhead of a CPU-GPU function invocation ($\approx$ 40 μs per call).

The C implementation of the inverse kinematic computation is the same code as the GPU code (sans the GPU-specific attributes). To achieve maximum performance, we experimented with both icc and gcc compilers and multiple optimization options. In the end, gcc with the options -O3 -lm -ffast-math -fstrict-aliasing -fwhole-program -combine proved to be fastest targeting a 2.4 GHz Intel Core 2 Quad Q6600 running an otherwise idle Linux system. All four cores were used by executing the computation in four parallel threads.

Power measurements are more difficult in this setting, since we did not want to interfere with the GPU's PCIe connection by directly inserting a watt meter into the GPUs power supply lines. We thus used an indirect approach, measuring the power drawn between mains and the host PCs power supply. We first established a baseline by measuring the power for an idle system, and then measure the power drawn when the benchmark is actually running. This set-up was validated on two different host PCs, in both cases, the power difference between idle and computing states was identical. To reduce measurement errors, we processed a large data set and let the GPU warm-up with dummy computations to ensure that its fan was running on the otherwise idle system. Total energy for the GPU is then estimated as the product of peak power (which remained almost constant during the computation) and run-time.

8. Comparison

In all cases, we ensure that the required data is available in local memory (i.e. on-card on the GPU and FPGA, in the cache on the CPU). Furthermore, we made the assumption that the accelerators are placed in a host system and are not running stand-alone (as would be possible with the FPGA).

Table 2 shows the results for the three different platforms when computing 10^9 function evaluations (expressed as "million evaluations per second", MEPS).

On the GPU, which incurs the high call overhead as described above, short-running functions (such as the inverse kinematics for a single point) will be very expensive and should be avoided in practice.

When deviating from the ideal condition that all data is available in local memory, the performance advantage of the GPU dwindles: Today's GPUs are generally connected to the rest of the system (and thus main memory) by PCI Express (PCIe). Modern GPUs have 16 PCIe 2.0 lanes, so they have a maximum transfer rate of 8 GB/s. This becomes the bottle neck when the data is not resident in GPU on-board memory and has to be fetched over PCIe from main memory (shown in Table 2 as "GPU (*bus limited*)" where the data is copied parallel to the computation over the PCIe bus). Thus, with PCIe 2.0, the inverse kinematic could be computed at 333 MEPS, a 75% drop in throughput.

The GPU performance is also degraded by the high communications latency. Note that a single computation in

Table 2: Comparison of the different implementations for a data set with 10^9 points. All used the optimized equations. Power is the difference of active power - idle power. System energy includes the host system.

Implementation	Throughput 10^6 evals/s [MEPS]	Latency [μs]	Power [W]	Energy [Ws]	System Energy [Ws]	HW Cost [EUR]	Impl. Effort [Days]
CPU	24.5	0.163	32.0	1304.80	5830.82	150	< 1
GPU	1366.0	40.146	170.0	124.50	210.05	400	< 1
GPU (*bus limited*)	333.0	40.146	170.0	510.51	840.84	400	< 1
FPGA (*in host PC*)	170.0	2.147	8.3	48.82	695.88	2000	10
FPGA (*embedded*)	170.0	2.147	8.3	48.82	78.24	2000	10

itself just takes 146ns on the GPU (and is thus almost 15 times faster than on the FPGA). However, it takes 40μs to start the GPU kernel.

The power numbers for the CPU and GPU are shown as peak difference between idle and computing states, and simulated for the FPGA (we used the maximum steady-state power drawn of Fig. 1, time interval c).

Execution times and power consumption are combined in the energy required to perform the 10^9 computations. Here, we show both the active energy consumption for the computation itself (sans system idle power) as well as the total system energy consumption (standard PC, one hard disk, idle power drawn is 110 W). Note that the FPGA could execute stand-alone, while the GPU and CPU require host systems. We show both scenarios for the FPGA: Running in a PC host (110 W idle power) and running as a stand-alone embedded platform (requiring 5 W for memories and network interface).

9. Conclusion

Computing on non-standard processors such as GPUs or FPGAs allows the use of GA algorithms in practical applications, these platforms outperform a conventional processor by one order of magnitude. The best specific technology is highly dependent on the scenario: In a high-performance computing setting, the GPU provides a tremendous speed-up, especially when keeping the complete computation on the GPU and not interacting with the host. This capability, however, requires a significant power supply and associated cooling, which might make it less attractive for embedded applications. This is where the FPGA shines: It is 7x faster than the CPU, but requires only a fraction of the power.

References

[1] NVIDIA Corp., *NVIDIA CUDA Compute Unified Device Architecture – Programming Guide*, June 2007.

[2] A. D. Scott Hauck, Ed., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.

[3] W. K. Clifford, "Applications of grassmann's extensive algebra," in *Mathematical Papers*, R. Tucker, Ed. Macmillian, London, 1882, pp. 266–276.

[4] ——, "On the classification of geometric algebras," in *Mathematical Papers*, R. Tucker, Ed. Macmillian, London, 1882, pp. 397–401.

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

[6] D. Hestenes, *New Foundations for Classical Mechanics*. Dordrecht, 1986.

[7] M. Pham, K. Tachibana, E. Hitzer, T. Yoshikawa, and T. Furuhashi, "Classification and clustering of spatial patterns with geometric algebra," in *International Conference on Applications of Geometric Algebras in Computer Science and Engineering (AGACSE), Leipzig*, 2008.

[8] D. Crookes, K. Alotaibi, B. Bouridane, P. Donachy, and A. Benkrid, "An environment for generating FPGA architectures for image algebra-based algorithms," in *Proc. International Conference on Image Processing (ICIP)*, 1998.

[9] C. Perwass, C. Gebken, and G. Sommer, "Implementation of a Clifford algebra co-processor design on a field programmable gate array," in *CLIFFORD ALGEBRAS: Application to Mathematics, Physics, and Engineering*, ser. Progress in Mathematical Physics, R. Ablamowicz, Ed., 6th Int. Conf. on Clifford Algebras and Applications, Cookeville, TN. Birkhäuser, Boston, 2003, pp. 561–575.

[10] A. Gentile, S. Segreto, F. Sorbello, G. Vassallo, S. Vitabile, and V. Vullo, "Cliffosor, an innovative FPGA-based architecture for geometric algebra," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2005, pp. 211–217.

[11] S. Franchini, A. Gentile, M. Grimaudo, C. Hung, S. Impastato, F. Sorbello, G. Vassallo, and S. Vitabile, "A sliced coprocessor for native Clifford algebra operations," in *Euromico Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2007.

[12] B. Mishra and P. Wilson, "Color edge detection hardware based on geometric algebra," in *European Conference on Visual Media Production (CVMP)*, 2006.

[13] B. Mishra and P. R. Wilson, "VLSI implementation of a geometric algebra parallel processing core," Electronic Systems Design Group, University of Southampton, UK, Tech. Rep., 2006.

[14] D. Hildenbrand, H. Lange, F. Stock, and A. Koch, "Efficient inverse kinematics algorithm based on conformal geometric algebra using reconfigurable hardware," in *International Conference on Computer Graphics Theory and Applications (GRAPP), Madeira*, 2008.

[15] K. Han, "Automating transformations from floating-point to fixed-point for implementing digital signal processing algorithms," Ph.D. dissertation, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, 2006.

[16] J. L. Hennessy and D. A. Patterson, *Computer architecture*. Amsterdam [u.a.]: Kaufmann [u.a.], 2007.

[17] Xilinx, *Virtex 5 Family Overview*, Xilinx, 2008.