ACCELERATION AND ENERGY EFFICIENCY OF GEOMETRIC ALGEBRA COMPUTATIONS USING RECONFIGURABLE COMPUTERS AND GPUS

Holger Lange

LOEWE Research Center AdRIA
Technische Universität Darmstadt
email: lange@esa.informatik.tu-darmstadt.de

Florian Stock, Dietmar Hildenbrand, Andreas Koch

Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt
email: {stock|koch}@esa.cs.tu-darmstadt.de
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 robotics can be accelerated using reconfigurable FPGA-based computing and on a graphics processing unit (GPU). The practical evaluation covers not only the sheer compute performance, but also the energy efficiency of the various solutions.

1. INTRODUCTION

Geometric Algebra (GA) is a mathematical framework for the concise description of complex geometrical relationships. It can be used as the base for compact algorithms in many fields of science and engineering. However, 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. Thus, a number of attempts (discussed in Sec. 3) have been made in the past to increase the execution speed of GAs.

Reconfigurable adaptive computers (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 [16], while ACSs additionally allow a much finer parallelism at the instruction- or pipeline level [19]. Other differences include the clock frequencies, which easily exceed 1 GHz for the GPU but are limited to a few hundred MHz at best on the often Field Programmable Gate Array (FPGA)-based ACSs, and the power consumption.

This work will give a short introduction into geometric algebra and then discuss the classical robotics problem of inverse kinematics (compute arm swivel angles to allow the robot's hand to reach a specific target point). The GA-based formulation of that problem 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 even 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.

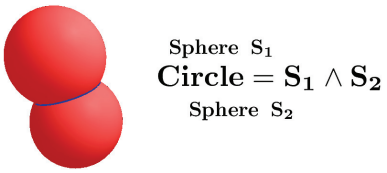

Fig. 1. Spheres and circles are among the basic entities of geometric algebra. Geometric operations such as the intersection of two spheres are easily expressible.

2. FUNDAMENTALS OF 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 [11] [10].

GA is able to easily describe and manipulate high-level geometric objects like spheres, circles and planes as well as operations combining objects.

The spheres of Fig. 1 for instance are concisely represented by the algebraic object

$$S = P - \frac{1}{2} r^2 e_\infty \tag{1}$$

with a center point P, radius r and the basis vector e_∞ (representing the point at infinity). Their intersecting circle is then computed with the help of the *outer product* operator as

$$Z = S_1 \wedge S_2 \tag{2}$$

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 [18], or the robotics application that will be discussed in Sec. 4.

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.

One of the first attempts to accelerate GA computations is [5]. It proposes to structurally compose hardware blocks for primitive GA operators using the PROLOG programming language. However, this work does not contain an experimental evaluation of a nontrivial application.

A more serious approach is described in [17], even though that accelerator only realises the geometric product. It is implemented on a 20 MHz FPGA connected via the PCI bus to the host computer. Due to the limited capacity of the FPGA employed, techniques such as wide parallel or pipelined processing, and the use of fast on-chip memories, were not exploited. Similarly, subspace coefficients consist only of 24 bit integers, other fixed or floating point formats are not supported. The architecture is able to process multivectors of up to eight dimensions, with smaller vectors being processed faster. While the resulting accelerator does achieve a speedup over a conventional software programmable processor when counting clock cycles, this advantage completely disappears when considering the actual execution *time* (wall clock time): While the software running on the processor requires many more clock cycles, each of these cycles is only 0.666 nanoseconds long (on the 1.5 GHz processor used for the evaluation). Each of the FPGA clock cycles, however, has a duration of 50 ns. This leads to a practical *slow-down* by a factor of 50x when using the FPGA-based solution over simple software running on a conventional computer.

A different approach was presented in [7]: This accelerator supports functions beyond the geometric product, namely, the outer product, contractions etc., each being implemented on a dedicated hardware unit. While the architecture is limited to multi-vectors of three to four dimensions, it is sufficient for many practical applications of GA in computer graphics. This design decision is reflected directly in the architecture of the unit: all operations are performed on six coefficients, with the per-coefficient computations being performed in parallel. As before, the coefficients are limited to integers, in this case 16 bit wide. The FPGA implementation of this accelerator achieves a frequency of 50 MHz and is able to compute the geometric product in 56 clock cycles. However, 49 of these are required for communicating with the host computer over the PCI bus. Despite this inefficiency of the FPGA-attachment, the computation of the geometric product requires fewer clock cycles than the 249 cycles taken by a conventional processor. But again, when taking the different clock frequencies into account to compute the real world execution times, this approach also leads to a slowdown by a factor of 9x.

An update of this work is given in [6]: the operation-specific hardware units have now been replaced by a variable number of so-called slices. Each slice is able to compute all operations of the four-dimensional GA. The coefficients have now been extended to 32 bit integers. In terms of hardware, a slice consists of a 32 bit wide arithmetic logic unit capable of addition, subtraction, multiplication, and logical computations. The GA operations are decomposed into these primitive calculations, with their execution being orchestrated step-by-step by on-chip software (microcode). Parallel or pipelined processing, which lies at the heart of all high-speed conventional processors, is not employed. The authors argue that their slice-based architecture achieves better scalability than the original one with dedicated hardware units. For high-performance, they suggest using multiple of the slice units in parallel. However, their experimental evaluation benchmarks just a single slice: the FPGA implementation achieves a clock frequency of 45 MHz and runs by a factor 3x to 4x faster than a software programmable processor when counting cycles. When actually considering the 2 GHz clock frequency of the reference processor, the actual execution time again slows down by a factor of 9x to 12x versus software.

The first coprocessor to lift the integer limitation on coefficients is introduced in [14], which allows multi-vectors with double precision floating-point coefficients. Each core consists of a floating point adder and multiplier, supported by smaller hardware units to compute the product of basis blades. While pipeline-parallel execution is employed within these compute units, actual GA operations (geometric product, rotor, etc) are again computed sequentially by decomposing them into primitive calculations controlled by microcode. The resulting accelerator was realised not as an FPGA, but as a custom-fabricated integrated circuit (ASIC) and thus achieved a higher clock frequency of 130 MHz. The experimental evaluation of the system in [15] shows a real wall-clock *speed-up* of 3x over a software programmable processor. However, the authors do not state which processor they used as a reference.

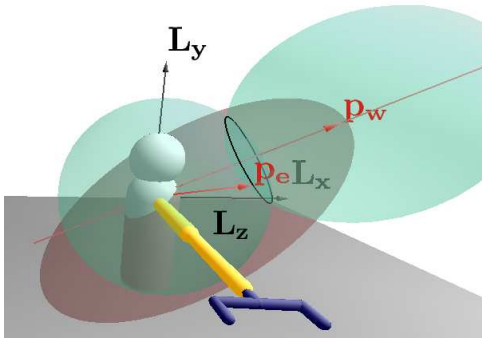

Fig. 2. GA model for inverse kinematic of a robot arm.

4. INVERSE KINEMATICS

One application for which GAs can be used is inverse kinematics. The task is to calculate for a kinematic chain (limbs connected via joints) the angles for the joints to reach a given end point p_w (see Fig. 2). Such an inverse kinematic is used for controlling robot arms or modelling human movements in computer animation, which makes it a realistic sample GA model for the evaluation of acceleration techniques. The details of the algorithm are discussed in [12]. In contrast, this work concentrates on the architecture aspects of FPGA and GPU-based accelerators.

5. 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. The discrepancy in achievable clock frequencies of conventional processors (which are now into multiple gigahertz), and that of FPGAs (which currently top out at 500-600 MHz), implies the need for massive parallelism in the FPGA to achieve comparable or even better performance. In all of the prior approaches, however, the degree of parallelism was highly limited. In many cases, software-like sequential processing (e.g., microcode) was employed instead of pipelining and parallel vector processing. Furthermore, the decision to treat the primitive operations of the GA as primitive operations for the hardware accelerator impedes any attempt at inter-operator parallelism and the parallel execution of intra-operator computations. Additionally, by treating all of the computation units as general-purpose black boxes, a very effective optimization, namely the folding of constants directly into the hardware structures, cannot be performed. As an example, the addition of the constant value 42 is both smaller and faster than the addition of a variable factor.

Our first implementation (Sec. 6) is completely different architecturally from the accelerator approaches described above. Instead of coarse granular computation units capable of handling entire GA operators, we decomposed the GA description into the underlying scalar equations. These equations, which employ only basic arithmetic operators, were optimized both manually as well as using automatic tools such as Maple. The resulting set of equations was then 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 (in this case, the positional accuracy of the robot's hand), we determined for each operator the optimal numerical representation (e.g., values in the range of 0 to 100 with 1/16mm 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.

Operation	Original	Optimized
Adds/Subs	97	48
Multiplication	168	71
Division	13	11
Square-Root	8	8
Mult/Div by power of 2	31	8

Table 1. Required operations before and after manual optimization.

The second implementation (Sec. 7), however, trades fine-grained parallelism and custom arithmetic for the brute-force SIMD parallelism achievable with a modern GPU.

Before describing the different implementations, we will discuss the target technology-independent optimizations we performed when translating the GA expressions to primitive arithmetic operators. The GA algorithm [12] 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. Table 1 shows the number of operations before and after the optimization.

6. FPGA IMPLEMENTATION

For the FPGA implementation, the optimized equations were then translated into a dataflow graph (DFG). Fig. 3 shows the sub-graph for the equation describing the X-coordinate p_{e_x} of the location of the elbow joint p_e

$$p_{e_x} = \frac{PP_k \cdot PP_2^- - PP_j \cdot PP_3^- + tmp_{sqrt} \cdot PP_2^-}{einf_{PP}} \quad (3)$$

6.1. Numerical Optimization

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 modeled robot 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:

Analytical approach: This propagates the precision and value ranges of the inputs (the target coordinate in our example) forward through the data flow graph and sizes the operators appropriately to avoid loss of precision for the result. This generally leads to overly wide operators (the intermediate results are much more precise than required in the end). To correct this, we then propagate the required precision and value ranges backward from the final result. Fig. 3 shows some of the required value ranges annotated to the operator nodes. Additionally, we exploit knowledge about the problem domain to further narrow the ranges. Given a length d_1 of the upper part of the robot arm, we know that each of the X, Y, and Z coordinates can at most be d_1 away from the origin. The ranges computed during the propagation can thus be narrowed down further (shown in Fig. 3 for the X-coordinate p_{e_x} of the elbow joint.

Empirical (Monte-Carlo) approach: In some cases the analytical approach does not yield satisfying results. The division operator, for example, can lead to a very wide range for the result. While domain knowledge can be exploited to narrow it down again (see Fig. 3), this is not always possible.

To handle these cases where no special domain knowledge is available and to verify the analytical approach and to obtain results it fails, the DFG was implemented in *MATLAB* using double precision arithmetic as well as the analytically determined fixed-point formats. A stream of random, but valid target positions was then fed into the equations and the results of both implementations compared and checked for sufficient precision. In all cases, a total word length of 32 bits (but with varying position of the binary point!) proved sufficient to perform the calculation.

Note that some attempts at automatically performing this optimization exist (e.g., [8]), but they are often limited with regard to the operators supported.

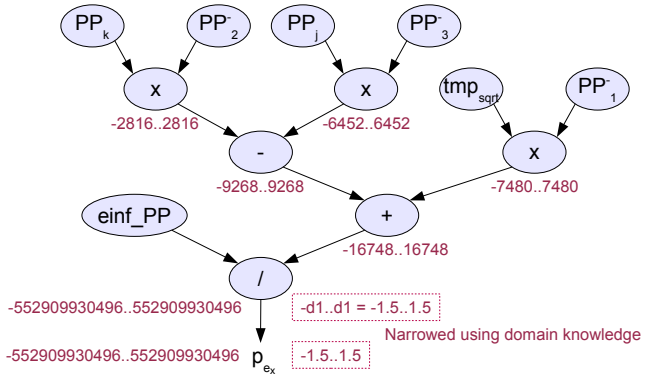

Fig. 3. Dataflow graph for p_{ex}

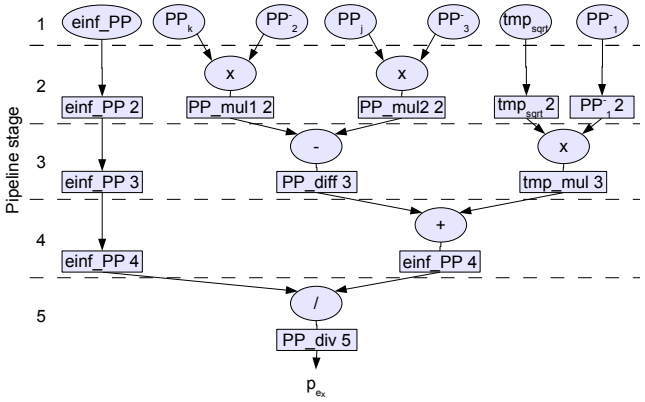

Fig. 4. Pipeline schedule for p_{ex} (cf. figure 3)

6.2. Hardware Realization

The DFG was implemented as a fully *spatial pipeline* in the hardware description language Verilog. We chose to optimize the pipeline for maximum throughput to demonstrate the potential of accelerating the algorithm in hardware. Nevertheless other optimization goals (minimum use of hardware resources/area, low latency, low power consumption) are possible as well, but not discussed here. The exact pipeline timing (also known as *schedule*) for p_{ex} (cf. 3) is shown in Figure 4. The pipeline was carefully balanced (equal path lengths from a given inputs/intermediate result to all uses of that value) by inserting registers (shown as rectangles) for einf_PP, tmp_{sqrt} and PP_1^-.

When synthesizing the Verilog description using Synplify Premier 9.4, the registers are automatically mapped to flip-flops or shift-register primitives as appropriate for the delay depth.

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 super-scalar proces-

sors which can handle about half a dozen parallel operations per clock cycle [9, Chapter 3.6].

7. GPGPU IMPLEMENTATION

FPGAs, being available as off-the shelf PCI/PCIe expansion cards for standard PCs, are no longer the only easily accessible way to exploit parallelism. In recent years, not only has the compute power of GPUs significantly improved, but their architecture has advanced from mostly fixed-function graphics pipelines to increasingly flexible processor arrays. The latter now allows *general purpose* computing on GPUs using dedicated languages (which in some cases resemble traditional high-level languages) such as Brook+ [2], Rapidmind [13], CUDA [16] and Stream [1]. The first two are portable, the latter two are specific to GPU vendors.

For our experiments, we used the Compute Unified Device Architecture (CUDA) from NVIDIA Inc., which extends the C language by constructs to annotate, e.g., function callability from the host CPU or the GPU, or which memory resource should hold a data structure. Furthermore it 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 (30 on the card we used) operating in a SIMD manner, thus called multiprocessors. Each multiprocessor has eight eight scalar datapaths (all executing the same instruction) that share access to a fast but small (just 16 KB on our GPU) on-chip memory.

A multiprocessor can schedule instructions from up to 512 different threads on its internal datapaths, aiming to hide long latencies (e.g., accesses to external memory). Between multiprocessors, all communication (data transfers, synchronization) occurs through the relatively slow global chip-external memory. Accesses to this memory should be performed in some regular patterns, otherwise each access will incur very high latencies. Other restrictions due to bank assignments and data alignment also exist, but these are not relevant for our implementation of the inverse kinematics application.

Since the SIMT model precludes the fine-grained spatial parallelization we performed for the FPGA, we use a multithreaded approach that executes independent computations (for different target points) in parallel.

We implemented a GPU kernel which computes the inverse kinematics function f. As it is pure data flow and contains no control flow the computation requires no thread synchronization (neither within nor between multiprocessors). 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. Thus coalesced, memory accesses can be efficiently performed in streaming mode. 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 global memory is only accessed at the beginning of the computation to read the parameters in, and at the end to write the result back.

```
__host__ __device__ struct single_computation_result
geometric_algebra_computation(
    float pwx, float pwy, float pwz, // input params
    float phi, float d1, float d2 // robot spec.
    )
{ ... }

/* Head of compute kernel */
__global__ void
compute_points_on_gpu(
    int number_points,
    float *d_x, float *d_y, float *d_z, // input params
    float *d_qe1, float *d_qe2, // output params
    float *d_qs1, float *d_qs2, // output params
    float *d_qs3, float *d_qs4, // output params
    float phi, float d1, float d2 // robot params
    )
{
  int startId = blockIdx.x * blockDim.x + threadIdx.x;
  struct single_computation_result result;
  int i;

  for (i = startId;
       i < number_points;
       i+= blockDim.x * gridDim.x)
    {
      result = geometric_algebra_computation_fast
                 (d_x[i], d_y[i], d_z[i], phi, d1, d2);
      d_qe1[i] = result.qe1;
      ...


/* invocation of the kernel */
compute_points_on_gpu<<<nr_blocks, 512>>>(....);
```

Listing 1. Excerpt from computation kernel and its invocation.

Listing 1 gives the an outline of the computation routine. It shows the interface of the routine `geometric_algebra_computation` which performs the computation of f, the attribute `__device__` marks the function as code which should be run on the GPU. The `__host__` enables the additional generation of code for the host platform. This is used for later benchmarking on the CPU (it uses the same C code).

The per-thread processing is described in its caller function `compute_points_on_gpu`. Each individual thread is uniquely identified by the special variables `blockIdx` and `threadIdx`. They are used in a `for`-loop to have each thread process only a sub-region of the input data, beginning at index `startId` and then processing every n-th point of the data.

The n is determined from the special variables `blockDim.x` and `gridDim.x` (our problem just uses a 1-D vector of input data). These are set when actually invoking the function in a thread-parallel fashion using the CUDA-specific `<<<`, `>>>` notation: The first parameter indicates into how many blocks the data input should be partitioned (this will become `gridDim.x`), the second one how many parallel threads should execute for each block (`blockDim.x`). This organization leads to high bandwidth streaming accesses to global device memory.

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.

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. 5.

8. 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.

8.1. FPGA

The Verilog HDL description of the DFG was mapped to a Xilinx Virtex 5 LX155 FPGA [22] using Synplify Premier 9.4 and the Xilinx ISE 10.1.03 tools. For synthesis, we generated dedicated dividers and square-root cores using the Xilinx Core Generator (part of the ISE tools). Multiplications were automatically mapped to DSP48E blocks, which are fast hardwired units on the FPGA that can be configured as 25x18 multipliers. At the time of this writing, no specialized CORDIC cores (needed for the square-root) were available for Virtex 5, hence we had to use a Virtex 4-version CORDIC on the Virtex 5. Table 2 shows the mapping and performance results for the complete DFG.

The power consumption of the FPGA, shown over time in Fig. 5, 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 ≈ 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.

Number of LUTs	34912 (35% of max.)
Number of FFs	49938 (51% of max.)
Number of DSP48Es	74 (57% of max.)
Clock frequency	170 MHz
Throughput	1 set of results / cycle
Latency	365 cycles

Table 2. Hardware mapping results on XILINX 5VLX155

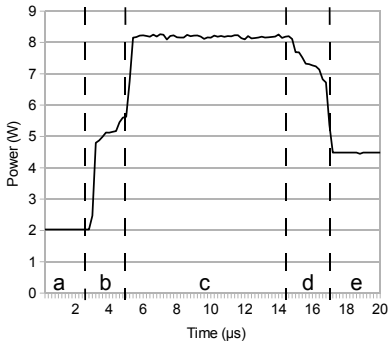

Fig. 5. Power consumption of the FPGA implementation

8.2. GPU

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 ($\approx 40\ \mu s$ per call).

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 (10^9 target points) 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.

8.3. CPU

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 and energy measurement was done in the same manner as for the GPU: The peak difference between idle and computing states of the system was used. In our case the system was an Intel QuadCore Q6600 clocked at 2.6 GHz.

8.4. 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).

The target-independent optimizations yield a speed-up of $\approx 2x$ over the initial equations on the CPU and GPU. Due to the increased design effort, only the optimized version was implemented on the FPGA. Note that, with the lack of dynamic scheduling and out-of-order execution on the GPU, static optimizations are even more important than for conventional CPUs (which often have these dynamic optimization features to compensate for sub-optimal code). As an example, if the GPU-compiled code would require more registers for a thread, fewer threads could be spawned and memory latencies no longer be fully hidden.

Table 3 shows the results for the three different platforms when computing 10^9 function evaluations of the inverse kinematic function. The table shows the number of million function evaluations per second (abbreviated MEPS) and the smallest achievable latency for a single computation. 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). With PCIe 2.0 having a maximum transfer rate of 500 MB/s per lane and modern GPUs generally having 16 lanes, this would yield a maximum 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 3 as "GPU (*bus limited*)" where the data is copied parallel to the computation over the PCIe bus). One computation requires 24 bytes of I/O bandwidth (three floats input, six floats output, performed full-duplex in parallel). Thus, with PCIe 2.0, the inverse kinematic could be computed at 333 MEPS, a 75% drop in throughput.

The software solution on the CPU is less affected when the data exceeds the cache size. The processor and chipset memory systems employ mechanisms such as intelligent prefetching and manage to match the data rates required by the slow (compared to the GPU) software computations running on the CPU. A PCIe-attached attached FPGA accelerator would fare similarly: It also computes sufficiently slowly to avoid data starvation, even when having to transfer data over the PCIe bus.

The GPU performance is also degraded by the high communications latency. Note that a single computation in itself just takes 146ns on the GPU (and is thus almost 15 times than on the FPGA). However, it takes 40μs to start the GPU and to exchange data and results with the host computer.

The power numbers shown reflect the power drawn during computation minus the system idle power. They were measured for the CPU and GPU as described above, and simulated for the FPGA (the maximum steady-state consumption of Fig. 5.d was used).

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).

Finally, we considered implementation time. Programming the GPU for such a straightforward application is as simple as normal software development for the CPU. Note that this does not always hold true, in some cases, the heterogeneous memory architecture of the GPU requires very careful design of algorithms and data structures. Even for a highly experienced designer, the FPGA implementation, with its steps of DFG building, fixpoint conversion, scheduling, Verilog coding and performance optimization, took significantly more effort. This might be alleviated to some extent by starting from more abstract descriptions than a HDL, such as Simulink or MATLAB, and compiling these into an FPGA [20] [21]. We experimentally evaluated a MATLAB to RTL flow for this application, but were unable to generate viable hardware in this fashion.

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	1304.80	5830.82	150	< 1
GPU	1366.0	40.146	170	124.50	210.05	400	< 1
GPU (*bus limited*)	333.0	40.146	170	510.51	840.84	400	< 1
FPGA (*in host PC*)	170.0	2.147	7	41.18	688.23	2000	10
FPGA (*embedded*)	170.0	2.147	7	41.18	70.59	2000	10

Table 3. 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.

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 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.

10. REFERENCES

[1] ATI. *AMD Stream Computing - Technical Overview*. ATI, 2008.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786, 2004.

[3] William K. Clifford. Applications of grassmann's extensive algebra. In R. Tucker, editor, *Mathematical Papers*, pages 266–276. Macmillian, London, 1882.

[4] William K. Clifford. On the classification of geometric algebras. In R. Tucker, editor, *Mathematical Papers*, pages 397–401. Macmillian, London, 1882.

[5] 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.

[6] 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 *Euromico Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, 2007.

[7] 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)*, pages 211–217, 2005.

[8] Kyungtae Han. *Automating Transformations from Floating-point to Fixed-point for Implementing Digital Signal Processing Algorithms*. PhD thesis, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, 2006.

[9] John L. Hennessy and David A. Patterson. *Computer architecture*. Kaufmann [u.a.], Amsterdam [u.a.], 2007.

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

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

[12] D. Hildenbrand, H. Lange, Florian Stock, and Andreas 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.

[13] Michael D. McCool. *Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform*. Rapidmind, 2006.

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

[15] Biswajit Mishra and Peter R. Wilson. Vlsi implementation of a geometric algebra parallel processing core. Technical report, Electronic Systems Design Group, University of Southampton, UK, 2006.

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

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

[18] 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.

[19] Andre DeHon Scott Hauck, editor. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.

[20] Xilinx. *MATLAB for Synthesis*. Xilinx, 2008.

[21] Xilinx. *System Generator for DSP*. Xilinx, 2008.

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