View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Crossref

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 760954, 11 pages
doi:10.1155/2011/760954

Research Article

FPGA Acceleration of Communication-Bound Streaming Applications: Architecture Modeling and a 3D Image Compositing Case Study

Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner

Paderborn Center for Parallel Computing, University of Paderborn, 33098 Paderborn, Germany

Correspondence should be addressed to Christian Plessl, christian.plessl@uni-paderborn.de

Received 23 February 2010; Revised 19 January 2011; Accepted 19 February 2011

Academic Editor: Viktor K. Prasanna

Copyright © 2011 Tobias Schumacher et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Reconfigurable computers usually provide a limited number of different memory resources, such as host memory, external memory, and on-chip memory with different capacities and communication characteristics. A key challenge for achieving high-performance with reconfigurable accelerators is the efficient utilization of the available memory resources. A detailed knowledge of the memories' parameters is key for generating an optimized communication layout. In this paper, we discuss a benchmarking environment for generating such a characterization. The environment is built on IMORC, our architectural template and on-chip network for creating reconfigurable accelerators. We provide a characterization of the memory resources available on the XtremeData XD1000 reconfigurable computer. Based on this data, we present as a case study the implementation of a 3D image compositing accelerator that is able to double the frame rate of a parallel renderer.

1. Introduction

Reconfigurable accelerators achieve performance gains over CPUs by turning application hot spots into customized hardware cores and providing customized memory architectures to deliver the required high data bandwidth. Typical reconfigurable platforms for high-performance computing come with a certain fixed memory architecture with no or limited possibility to change the size and organization of the external memory on an per-application basis. A specific challenge is to find new methods for reducing the design effort for accelerators which are capable of using the given memory layout in a flexible yet effective way.

For supporting reconfigurable accelerator design, we have created the IMORC: Infrastructure for Performance Monitoring and Optimization of Reconfigurable Computers [1, 2]. IMORC consists of an architectural template and an on-chip network. An application is split into an arbitrary number of cores that run at full speed in their own clock domains and communicate asynchronously via FIFO-buffered links. IMORC inserts bitwidth conversion modules into the links which speeds up the accelerator design process and facilitates the reuse of developed processing cores. The IMORC infrastructure also includes memory controllers and host interfaces which provide the cores with a unified and transparent way of accessing different kinds of memory, for example, on-chip memory, off-chip memory, or host memory.

Related work on architectural templates like IMORC includes SIMPPL [3], which also connects different cores in a field programmable gate array (FPGA) using asynchronous FIFOs. An example for a performance modeling and profiling approach can be found in [4], where a model for determining an application's suitability for FPGA acceleration is presented. Similarly, [5] introduces the reconfigurable computing amenability test (RAT) methodology, which is a high-level, analytical performance prediction model. In [6], a framework for performance analysis for high-performance reconfigurable computing is proposed, which gathers performance information using manually inserted load sensors. Further work in the area of performance prediction is presented by Smith and Peterson [7, 8] who

focus on synchronous iterative algorithms running on high-performance computers equipped with FPGAs. IMORC differs from these approaches in (i) its more flexible interconnect and (ii) extended infrastructure support. The IMORC interconnect employs a multibus architecture with slave-slide arbitration which allows for a multitude of topologies. Infrastructure cores such as bitwidth converters, farming cores, and load sensors [2] greatly ease accelerator design and performance analysis.

The contribution of this paper is the communication performance characterization for IMORC cores executing on the XtremeData XD1000 system and, based on that, the development of an accelerator for parallel rendering. This paper extends our work presented in our publication "Communication Performance Acceleration for Reconfigurable Accelerator Design on the XD1000" presented at ReConFig '09 [9]. In addition to our previous work, we present an introduction to the IMORC architecture template and its communication infrastructure. The architecture characterization was extended by an analysis of the CPU's memory interface and by a characterization of the achievable performance when the CPU directly accesses the FPGA's address space. Furthermore, we added a discussion of alternative approaches for 3D compositing using the SIMD (single instruction, multiple data) units of the CPU, or graphics processing units (GPU). Finally, we elaborate on the reasons why our FPGA-based hardware accelerator is able to accelerate streaming applications, such as 3D compositing, even if the computational kernels are comparatively simple and provides only limited inherent parallelism.

The remainder of this paper is organized as follows: in Section 2, we give an introduction to the IMORC architecture template that is used as a basis for the architecture characterization and the case study. In Section 3, we discuss the memory layout on the XD1000 reconfigurable computer and the IMORC infrastructure for accessing memory on that machine. We then experimentally characterize the maximally achievable communication performance. In Section 4, we use these data to develop an IMORC accelerator for a z-buffer compositing kernel, a hot spot in our parallel rendering application. The compositing kernel is a data-centric streaming kernel with almost no computation and thus ideal for evaluating the efficiency of the memory architecture. Measurements presented in Section 5 show that the IMORC accelerator can double the frame rate of the parallel rendering application. In Section 6, we discuss the applicability of the presented approach to other applications. Finally, Section 7 concludes the paper.

2. Overview of the IMORC Architecture Template

When implementing FPGA-based accelerators, the final design often consists of several cores that need to communicate with each other or with some kind of memory. For simplifying the design of such accelerators, we have created the IMORC: Infrastructure for Performance Monitoring and Optimization of Reconfigurable Computers.

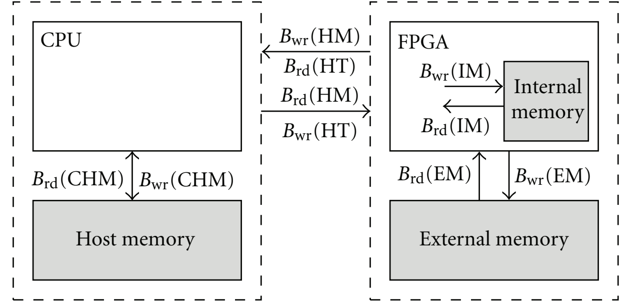


Figure 1: Memory architecture of the XD1000 architecture.

IMORC consists of an infrastructure that enables such communication between cores. Cores provide an arbitrary number of master and slave ports, which enable them to communicate with each other using links. IMORC links use asynchronous FIFOs for buffering data and requests and enable each core to run at full speed in its own clock domain. Additionally, the links insert bitwidth conversion modules enabling each core to operate at its native data width without any knowledge of the other cores' native data width. 1 : 1 connections can directly be achieved by connecting a master and a slave port using an IMORC link, while for n : 1 connections (multiple masters access one slave), slave side arbiters are inserted.

IMORC also provides load sensor cores for collecting performance information at runtime, for example, statistics on how frequently FIFOs have run full or drained empty. Additionally, user-specified load sensors are supported. The data gathered by these load sensor cores allow the designer to optimize the performance of the accelerators during a redesign, for example, by increasing the width of buses that have been identified as bottlenecks. Further available infrastructure cores include the IMORC-to-Register converters for setting and reading control registers of cores, farming cores for dynamic load balancing, and, depending on the target platform, memory controller and host interface cores. An in-depth overview of the IMORC infrastructure is presented in [1].

3. Communication Performance Characterization

In this section, we quantitatively characterize the achievable performance for a reconfigurable core accessing memory on the XtremeData XD1000 reconfigurable computer.

3.1. The XtremeData XD1000 Architecture. The XD1000 features a 2.2 GHz AMD Opteron CPU with 4 GB of host memory, and a module equipped with an Altera Stratix II EP2S180-3 FPGA and 4 GB of external memory installed in a second Opteron socket. The FPGA connects to the Opteron processor via a 16-bit-wide HyperTransport link running at 800 MT/s. Figure 1 displays the XD1000 architecture with the three types of memory a reconfigurable core can access:

(i) *Host Memory*. For accessing the host memory, we implement an IMORC interface to HyperTransport which is based on the HT cave described in [10]. The cave maps three distinct address regions into the address space of the CPU and forwards incoming HyperTransport packets to our interface. The interface decodes the requests and converts packets that hit one of the first two address regions into equivalent IMORC packets which are posted on two separate IMORC links. Packets targeting the third address region directly access an embedded block of memory used as a page mapping table. The first two links are used for accessing control registers and for large data transfers, respectively. The page mapping table is responsible for a mapping in the other direction: host memory is mapped into the address space of a third IMORC link. For this purpose, the page mapping table needs to be configured with physical page addresses of the target host memory by the user application. When cores send requests over this link, the upper bits form an index into the page mapping table and the lower bits are the offset into the page. Using this address mapping, an IMORC packet is converted into an equivalent HyperTransport packet and posted to the HyperTransport cave.

(ii) *External Memory*. For off-chip memory access, IMORC wraps the Altera DDR SDRAM controller core which can access memory in blocks of configurable burst sizes. DDR SDRAM writes always cover a complete burst, that is, 2, 4, or 8 clock cycles, depending on the controller configuration. The XD1000 system provides a 128-bit-wide memory. Hence, depending on the burst size, 256 bit, 512 bit, or 1024 bit are written to the memory during a transfer. Since the XD1000 does not provide data mask pins for the memory to mask out bytes not to be written during a burst, IMORC implements a read-modify-write cycle for writes that do not match one of the burst sizes. The read-modify-write cycle incurs an overhead but at the same time increases the flexibility and potential for core reuse, as the core designer is not necessarily bound to predetermined burst sizes and link widths.

(iii) *Internal Memory*. IMORC also provides a versatile interface for accessing on-chip memory which is functionally equivalent to the interface for accessing off-chip memory. Typical FPGA on-chip memories provide byte-enables, which makes the implementation of the interface for internal memory less complex than for external memory. Moreover, on-chip memory can be easily adapted to an application's needs since it is customizable to a high degree regarding parameters such as width and depth. While the achievable bandwidth for accessing internal memory can be huge, the capacity of internal memory is rather limited.

Table 1 summarizes the capacities and theoretical maximum bandwidths for all memories in the XD1000 system, as reported in specifications and data sheets. Note that these bandwidth figures must be considered as upper bounds which will not be attained in any concrete implementation, such as IMORC, due to controller and protocol overheads. Typically, such overheads make the achievable bandwidth dependent on the request size. Furthermore, the actual bandwidth available to an accelerator implementation can be further reduced by contention when several cores compete for accessing the same memory.

Table 1: Capacities and theoretical bandwidths for the XD1000.

Memory	Capacity	Parameter	Bandwidth
Host	4 GB	$B_{\mathrm{rd}}(\mathrm{HM})$	1.6 GB/s
		$B_{\mathrm{wr}}(\mathrm{HM})$	1.6 GB/s
External	4 GB	$B_{\mathrm{rd}}(\mathrm{EM})$	5.4 GB/s
		$B_{\mathrm{wr}}(\mathrm{EM})$	5.4 GB/s
Internal	1 MB	$B_{\mathrm{rd}}(\mathrm{IM})$	$\gg$
		$B_{\mathrm{wr}}(\mathrm{IM})$	$\gg$

In order to obtain a more detailed bandwidth characterization for different request sizes, we have implemented a micro benchmark. The micro benchmark essentially is an IMORC core consisting of a request generator, a data source, and a data sink. The core provides one IMORC link per memory to be tested and can be configured in the test type (read/write), the overall size of data to be transferred, and the request size per transfer. The link to the HyperTransport interface is 64-bit wide, the link to the memory controller for external memory 256 bit. The benchmarking core gathers the number of clock cycles required for completing a request and sends this data to the host application.

3.2. CPU ↔ Host Memory Performance Characterization. For measuring the throughput of the host memory when accessed by the host CPU, we used the RAMspeed benchmark [11]. Just like the popular STREAM benchmark [12], RAMspeed performs several simple computations on large arrays of data, hence it accesses memory with consecutive addresses. Contrary to STREAM, RAMspeed does not only operate on scalar values but can also perform benchmarks using the SIMD units (MMX and SSE) integrated in modern CPUs for measuring the throughput. The basic operations performed in RAMspeed are Copy ($A = B$), Scale ($A = m \cdot B$), Add ($A = B + C$) and Triad ($A = m \cdot B + C$). Figure 2 shows the results of the RAMspeed benchmark operating on 64 bit integer values, 64 bit floating point values, when using optimized SIMD instructions with optional prefetching operations. These prefetching variants of the code use explicit commands to preload the cache with the data needed in future iterations of the benchmarking loop in order to reduce cache misses.

The results show an average bandwidth from 2.34 GiB/s to 2.46 GiB/s for integer and floating point operations as well as for the MMX and SSE operations without explicit prefetching. The MMX and SSE operations with explicit prefetching achieve a much higher bandwidth of about 4.29 GiB/s. This shows that the CPU's built-in automatic prefetching units are not able to generate optimal prefetching commands even for these basic streaming operations.

3.3. CPU ↔ FPGA Communication Performance Characterization. Figure 3 presents the achievable bandwidth when the CPU reads from or writes to the FPGA. The write bandwidth nearly reaches the maximum theoretical bandwidth of the

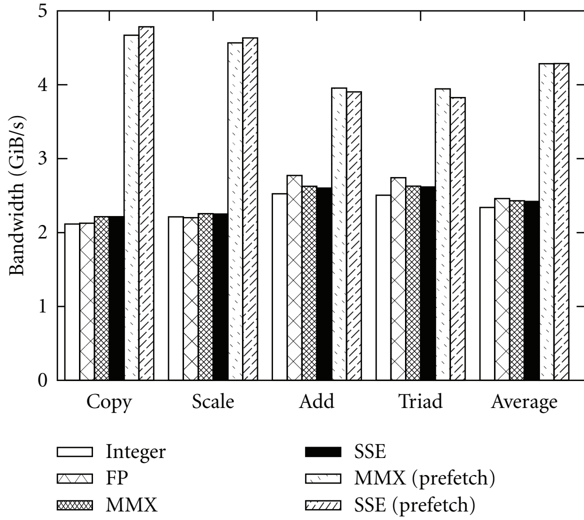


Figure 2: Results of the RAMspeed benchmark.

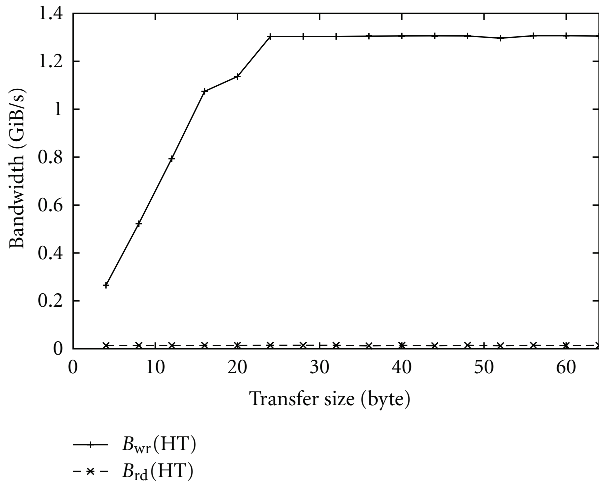


Figure 3: Measured bandwidth when the CPU accesses the FPGA.

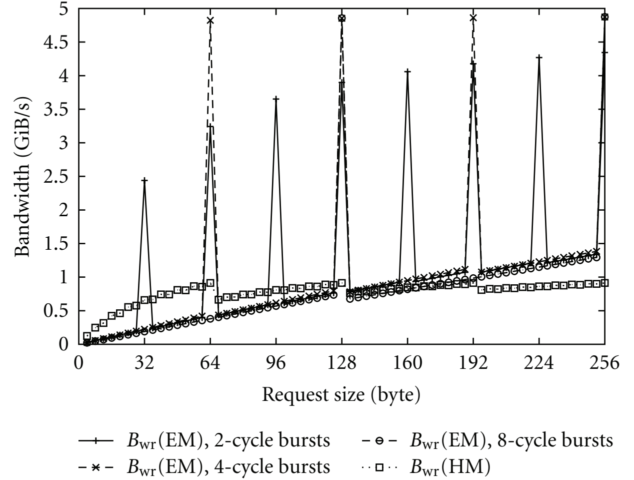


Figure 4: Measured write bandwidth (B_{wr}(EM), B_{wr}(HM)) for an IMORC core on the XD1000.

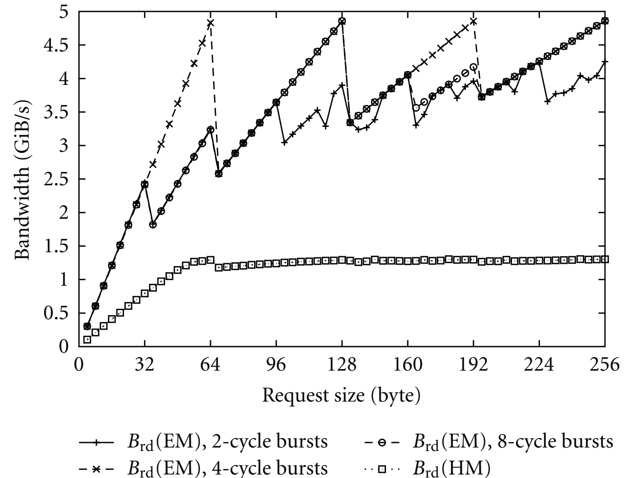


Figure 5: Measured read bandwidth (B_{rd}(EM), B_{rd}(HM)) for an IMORC core on the XD1000.

HyperTransport link if sufficiently large amounts of data are transferred. Reading from the FPGA results in a very poor performance of about 14 MiB/s. The reason for this behavior is that the CPU does not send a sequence of read requests to HyperTransport link that connects the CPU to the FPGA, which could be responded with a continuous stream of data, but sends only a single request for 64 bit of data at a time. The next request is not sent before a response has been received. Thus, directly reading data from the FPGA should be avoided. Instead, data transfers from FPGA to host memory should be initiated by the FPGA that is able to better exploit the available bandwidth of the HT link as shown in the following paragraph.

3.4. FPGA ↔ DDR SDRAM and FPGA ↔ Host Memory Performance Characterization. Figures 4 and 5 present the measured performance for accessing the host and the external memory from the FPGA with varying request sizes. External memory on the FPGA module is tested using different controller configurations with 2-, 4-, and 8-cycle bursts. As mentioned in the overview of the XD1000 memory architecture, writing data that does not match the burst size of the memory or does not start at a burst boundary address requires that a read-modify-write cycle is inserted. This procedure consists of reading a complete burst from DDR SDRAM, decoding the error correcting code (ECC) and performing an error correction, multiplexing the data to be written on the resulting data stream, encoding the ECC and writing the data stream back to the memory. As expected, for writes to the external memory, Figure 4 clearly shows that the read-modify-write cycle produces a significant overhead. Memory-bandwidth bound applications should thus prefer write block sizes matching the controller's burst size. For request sizes that do not match the controller's burst size, all configurations achieve roughly the same bandwidth.

For full-burst writes, the controller configurations differ strongly. Configurations with 4- and 8-cycle bursts achieve a write bandwidth of about 4.8 GiB/s, while the 2-cycle burst configuration is inferior. The 2-cycle burst configuration delivers a write bandwidth of only 2.5 GiB/s for small request sizes that comprise one burst, and improves to about 4.3 GiB/s for requests of 256 bytes.

For write accesses to host memory over the HyperTransport interface, the bandwidth increases with the block size to about 1 GiB/s for requests of 64 bytes, which is the maximum packet size of HyperTransport. For larger request sizes, the bandwidth drops to about 0.6 GiB/s since the data now has to be split into two packets of which only one packet exploits the maximum packet size, and again increases linearly to peak at about 1 GB/s for multiples of HyperTransport's maximum packet size.

Figure 5 presents the measurements for read requests. The bandwidth for reads from external memory increases linearly with the request size, reaches a maximum when a multiple of the configured burst size is reached, and drops again when exceeding this size. Interestingly, for some non-matching request sizes 4-cycle bursts are superior to 8-cycle bursts which tend to match the lower 2-cycle performance. In these cases, the Altera DDR SDRAM controller performs sub-optimally and terminates the burst after receiving data for 2 cycles before initiating another 2-cycle burst.

The achievable bandwidth for read requests to the host memory increases linearly for request sizes up to 64 bytes, resulting in a bandwidth of about 1.3 GB/s. Increasing the request sizes beyond this threshold does not result in any further significant deviations.

4. An Compositing Accelerator for a Parallel Rendering Framework

4.1. Parallel Rendering. Modern *3D computer-aided design* applications such as production planning and optimization and mechanical component design require substantial computation for rendering 3D scenes. Using highly detailed object models originating from computer-aided design (CAD) tools or 3D scanners, such simulation and visualization applications generate complex 3D scenes with a huge numbers of polygons to be rendered. Parallel rendering approaches are required to meet the stringent performance requirements, especially for interactive modes of operation. Molnar et al. [13] introduce three approaches for parallel rendering: sort-first, sort-middle, and sort-last.

Our case study focuses on an in-house parallel renderer using the sort-last approach. A master node divides a scene (frame) into $N - 1$ subscenes and distributes the workload to $N - 1$ rendering nodes. The subscenes have roughly the same number of geometric primitives (polygons) but the assignment of polygons to rendering nodes is arbitrary. Each rendering node runs a rendering pipeline that computes a set of display primitives (pixels) from the received geometric primitives. A rendering node computes two buffers for its subscene, the frame buffer containing the color information and the z-buffer containing the depth information for each pixel. The resulting $2 \cdot (N - 1)$ buffers are transferred back to the master node for compositing. Compositing performs the sorting step by comparing the distances of the $N - 1$ candidates for each pixel to the view plane. Only the closest display primitive is visible.

There are different hardware solutions to accelerate the compositing of the different frames [14–16]. These solutions usually comprise some kind of application-specific integrated circuit (ASIC) or FPGA attached to the graphics card and can further accelerate different tasks commonly used in computer graphics. Lightning2 [14] for example is a standalone hardware that is connected to the graphics cards in the rendering cluster using DVI (digital visual interface) interface. The rendering processes running on the different nodes are modified to copy the depth buffer into the pixel buffer right after the pixel information and sends both buffers to the DVI port. The Lightning2 hardware receives these buffers, performs the image composition, and sends the resulting image to a separate DVI port. Other approaches such as the Sepia architecture [17, 18] consist of PCI attached hardware that is added to every node of the rendering cluster. The hardware reads the pixel and depth buffer of each frame from the graphics card and additionally receives corresponding buffers from other nodes using a separate special-purpose network. The frames are composed and again sent to the special-purpose network for further compositing steps or to the graphics card's pixel buffer for being displayed. The main issues with these solutions are their inflexibility and the limited number of supported applications [19].

While such special-purpose hardware solutions may be appropriate for dedicated rendering clusters, equipping general-purpose clusters for which parallel rendering is just one of many applications is often not affordable. Instead, advances in computer system architecture allow for achieving the same objective with off-the-shelf components. Fast PCI express interfaces allow for attaching hardware accelerators with a standardized interface, and high-speed networks allow for sending images fast enough over the network, which enables flexible software solutions, for example [19, 20]. However, while the rendering process can be distributed easily, composing images with the CPU still takes a significant amount of time. Compositing has been identified as a bottleneck for achieving sufficiently high frame rates in our in-house parallel renderer, in particular for high-resolution images. Hence, parallel renderers could benefit from acceleration with general-purpose FPGA accelerators, which are becoming increasingly popular in the area of high-performance computing. As FPGA-based accelerators can be used for accelerating a wide range of applications, equipping a general-purpose cluster with commodity FPGA accelerators is more justifiable than installing specific hardware for rendering.

4.2. Compositing Accelerator Performance Estimation. The parallel renderer we are studying in this work is part of a visualization application that can be run in batch-mode or interactively. The master node stores or displays the

composited image and distributes the next subscenes to the renderer nodes. The application is implemented with the MPI (message passing interface) library using double buffering for frames to overlap computation and network communication. To analyze potential bottlenecks in the parallel rendering application, it suffices to look at the following parameters:

(i) H and W are the height and the width of a subscene (frame) in pixels. Each rendering node processes a frame buffer and a z-buffer for each subscene, resulting in $P = W \times H \times 8$ bytes of data (32 bit frame buffer, 32 bit z-buffer).

(ii) T_R (s/frame) is the time required for one rendering node to compute its subscene. T_R depends on the size of the subscene, that is, on P and the number of polygons. Since the workload is evenly distributed, we can work with an average value over all rendering nodes.

(iii) T_C (s/frame) measures the computation time for the master node and comprises the times for composing the images, displaying or storing the resulting image, and redistributing the next workload. The compositing time is dominating and depends on P and the number of rendering nodes, $N - 1$.

(iv) B_{net} (byte/s) is the bandwidth of the link which connects the master node to the computer network.

The aggregated data bandwidth generated by the rendering nodes is $(N - 1) \times P/T_R$ (byte/s). Depending on P, the complexity of scenes, and parameters of the compute cluster, a reasonably designed and configured system will try to set the number of rendering nodes such that the network or the master node is not saturated. Bottlenecks occur if the aggregated renderer bandwidth exceeds B_{net} or if the master node's computation time T_C limits the throughput. The latter will be more likely in practice which makes compositing an interesting target for acceleration.

Listing 1 shows the pseudocode for the compositing function. The function is computationally very simple, only comprising a regular loop with comparisons and assignments which can be parallelized in a straight-forward way. However, the main challenge for accelerating this code is to establish a continuous stream of data through the computing core. The main design decisions for the FPGA accelerator are (i) where to store the images and (ii) how many parallel comparisons to implement. Referring to the communication characterization for the XD1000 system (Section 3), we conclude that internal memory is not available in sufficient capacity for storing realistically large images.

To support the streaming nature of the application, both the CPU and FPGA implementation of the compositing function need to store the image (frame and z buffers) from the first rendering node in memory. Then, the stored image is compared to each newly arriving picture and updated if necessary.

Overall, the accelerator has to transfer $(N - 1) \times P$ bytes of data from the host to the FPGA module. Since the used OpenMPI implementation [21] is unable to store the

```
/* compose function
  Parameters:
    pic_a: base address of first framebuffer
    pic_b: base address of second framebuffer
    size: frame size in pixels
  Note:
    –pic_a is also the destination of the composed
      frame
    –framebuffers are directly followed
      by z-buffers, that is, z_a = pic_a + size
*/
void compose (int * pic_a, int * pic_b, int size) {
    //calculate base addresses of z-buffers
    int *z_a = pic_a + size;
    int *z_b = pic_b + size;
    for (int i = 0; i < size; i++) {
        if (z_b[i] < z_a[i]) {
            pic_a[i] = pic_b[i];
            z_a[i] = z_b[i];
        }
    }
}
```

Listing 1: Code for the compositing.

received data directly into the FPGA's address space, we have to explicitly transfer the frames to the FPGA.

Figure 6 visualizes the execution of the compositing accelerator during the different phases and the data channels to be used. We transfer the first frame to the external memory on the accelerator module. Since $B_{\text{rd}}(\text{HM})$ is lower than $B_{\text{rd}}(\text{EM})$, the time needed for this first phase of the accelerator is determined by the HyperTransport performance. The following $N - 2$ frames of size P stream from the host to the FPGA accelerator, and, at the same time, the stored frame streams from external memory to the FPGA, and the resulting frame streams back to external memory. The time required for this second phase of computation is dominated by memory accesses and given as $P/B_{\text{rd}}(\text{EM}) + P/B_{\text{wr}}(\text{EM})$ for the external memory and $P/B_{\text{rd}}(\text{HM})$ for the host memory read over the HyperTransport link. Consulting the bandwidth measurements of Section 3, that is, Figures 4 and 5, we conclude that for reasonably chosen request sizes the host memory access will limit the execution time. This holds true only if the external memory is accessed with request sizes that are multiples of the controller's burst size. The actually chosen burst size of the memory controller influences the access time for the external memory, but has no effect on the overall compositing application.

For the last frame, we read P bytes from each host memory and external memory but write only $P/2$ bytes back to host memory since the resulting z-buffer is not needed for displaying the image. Despite the fact that the write bandwidth to the host memory is much lower than the read bandwidth, the execution time of this last accelerator phase will be determined by reading the host memory since writing involves only half the data size. Using this performance estimation, we compare the execution times

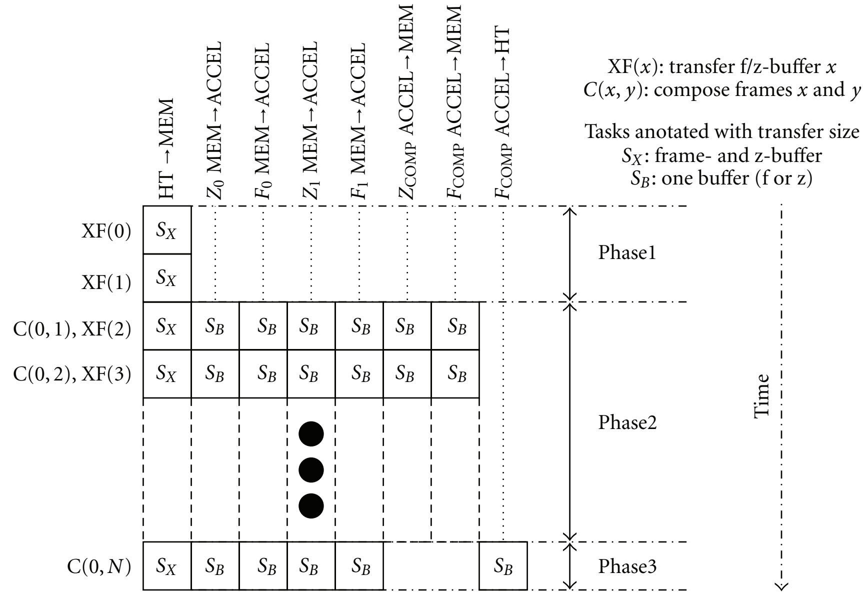


Figure 6: Diagram of the communication performed in each phase.

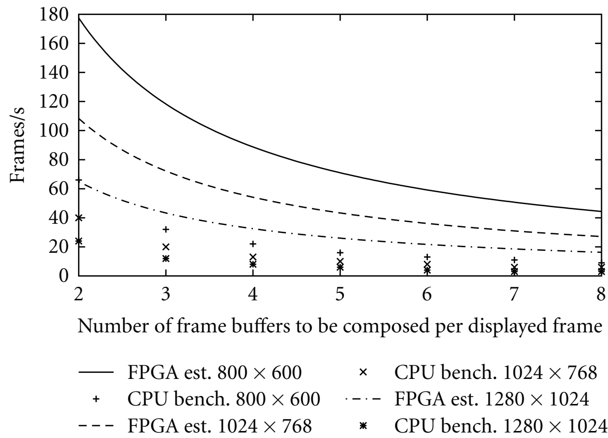


Figure 7: Comparison of CPU performance and estimated FPGA performance for the compositing function.

for the FPGA compositing accelerator with measured CPU execution times for the same task. Figure 7 shows the results of this performance comparison for frame resolutions of 800×600, 1024×768, and 1280×1024 pixels. The FPGA accelerator achieves higher frame rates than the CPU. When composing the results from four rendering nodes operating in parallel, the estimated speedup ranges from 4× to 4.5×. For eight parallel rendering nodes, the speedup is between 5× and 5.7×. Naturally, the achievable frame rate decreases with an increasing number of subimages that need to be composed for generating the final image. However, Figure 7 presents only the compositing time. The overall performance of the parallel rendering application will scale with the number of rendering nodes, up to the point where we hit a bottleneck.

5. Implementation and Measurements

In this section we discuss the implementation of the compositing accelerator based on the statements given in the previous section. We then evaluate the performance of the accelerator by integrating the XD1000 into a test setup and rendering a test scene on a different number of CPUs. We compare the compositing performance of the accelerator to the performance achieved by the XD1000 CPU and a CUDA-based implementation executed on an NVidia graphics card.

5.1. Accelerator Design. Based on the analysis of the compositing function and the bandwidth estimations, we implement a compositing accelerator using our IMORC infrastructure. The accelerator architecture is shown in Figure 8 and comprises a host interface core encapsulating the HyperTransport interface, a memory controller for accessing the external memory, and a number of cores implementing the actual compositing function.

As shown in Listing 1, for each iteration we need to read two values of the z-buffer and two values of the frame buffer, and write one z-buffer and one frame buffer value. We always write back one z-buffer and one frame buffer value in order to keep regularity in the data flow. Figure 8 shows that we employ a separate stream buffer core for each of these six memory accesses. The memory controller is configured to 8-cycle bursts and the request size is set to 128 bytes. A seventh stream buffer handles frame buffer writes, when the resulting image is streamed back to the host.

The composer core of the accelerator implements the actual comparison of z-buffer values and runs at 200 MHz synchronously to the HyperTransport core. To fully utilize the HyperTransport link to the host memory, which is the limiting factor in the design, we unroll the loop in Listing 1 and compute two iterations in parallel. Hence, all links from

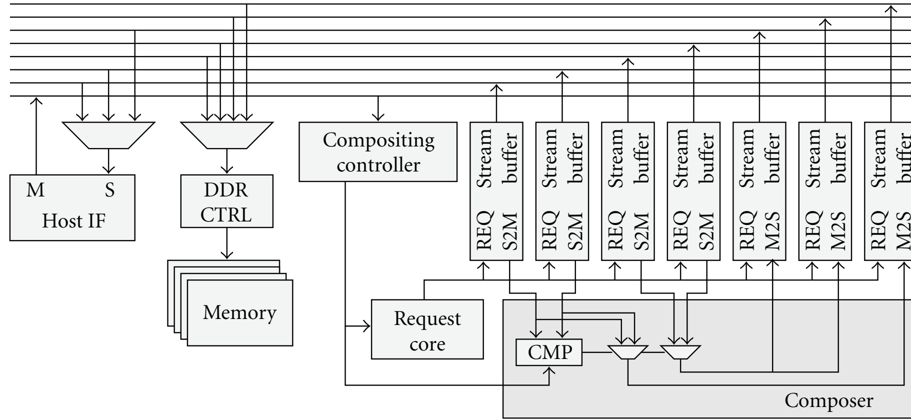


Figure 8: Architecture of the compositing accelerator.

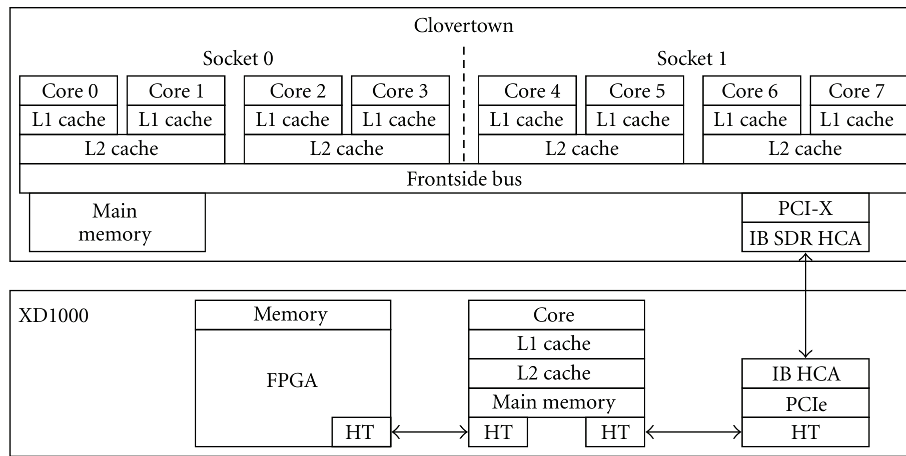


Figure 9: Architecture diagram of the test setup comprising an Intel Clovertown system with two 4-core processors connected to the XD1000 using Infiniband.

composer cores to the HyperTransport interface core are 64-bit wide.

The remaining two cores are the request core and the compositing controller. The request core is responsible for issuing read/write requests to the stream buffer cores. The compositing controller includes control registers for parameters such as the width and height of an image and the number of rendering nodes, as well as logic needed for exchanging state information between the host and the FPGA accelerator.

5.2. Performance Evaluation. In order to evaluate the overall performance of the parallel renderer, we implement a test system comprising an Intel Clovertown machine connected via Infiniband to the XtremeData XD1000 as pictured in Figure 9. The Intel Clovertown features two quad core processors running at 2.66 GHz and 8 GB of main memory. We use this machine to implements 1 to 8 rendering nodes. The XD1000 implements the master node including the compositing function. Theoretically, the Infiniband interconnect provides a peak bandwidth of 10 GBit/s. Measurements with the Intel IMB benchmark show that our test setup reaches a sustained Infiniband bandwidth of 700 MB/s.

For image compositing, we compared the performance of off-the-shelf components. We attached an NVidia GeForce 8800GTS graphics adapter with an NVidia G80 GPU to the PCIe 16× slot of the XtremeData XD1000 platform. Using the `bandwidthTest` application of the NVIDIA CUDA software development kit (SDK), we measured a write bandwidth of about 1457 MB/s from the host memory to the memory of the graphics card, which is comparable to the achievable bandwidth of the CPU/FPGA interface of the XD1000. We have tested different approaches for implementing such a CUDA-based compositing accelerator. The first implementation variant starts an asynchronous receive operation for the frame- and z-buffers of all rendering nodes. As soon as the first two frames have been received, these images are composed on the GPU. As soon as the next frame has been received, the next compose operation is started on the GPU, and so forth. Benchmarking has shown that the performance of these procedures is suboptimal, hence we have implemented a second variant of a GPU compositing application which waits until all frames have been received. Then, all frames are copied to the GPU and a different compositing kernel that processes all frames is invoked on the GPUs. After compositing, the resulting image

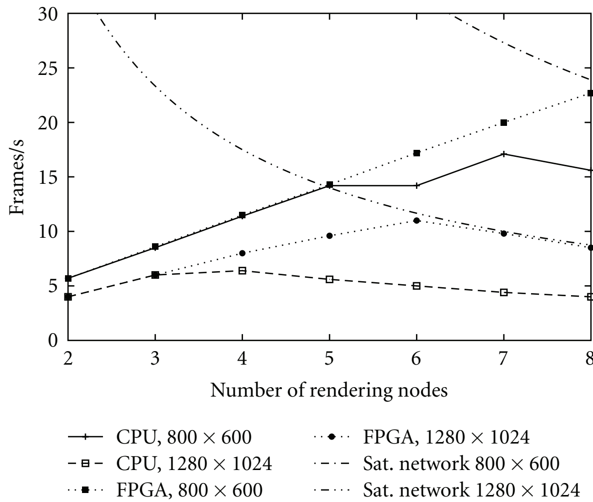


Figure 10: Performance values of the complete rendering application.

is read back from the GPU memory. On the host CPU, we are using a double buffering approach. Hence, while the GPU is busy with compositing, the host CPU is already receiving the next frames from the rendering nodes in the meantime.

However, although this implementation is much faster than the first variant, composing the frames on the 2.2 GHz Opteron CPU of the XD1000 is still faster than composing on the GPU. In our test setup, we achieved a frame rate of about 11.4 fps for a resolution of 800 × 600 with four rendering nodes when using the CPU for compositing. The maximum frame rate for this resolution was achieved with 7 rendering nodes and was around 17.1 fps. The CUDA-based compositing kernel only achieved frame rates of about 9.7 fps and 15.5 fps with these parameters, respectively. A similar behavior could be observed with higher resolutions like 1280 × 1024, although in this case the values are more close. At this resolution, the CPU-based implementation achieves a maximum frame rate of 6.4 fps, which is achieved when using four rendering nodes. The corresponding CUDA implementation in this case only reaches a rate of 5.8 fps. The main reasons for these results is that the CUDA implementation needs to transfer all images to the GPU first, before the compositing operation can be started and the final image can be read back to the CPU. In contrast, a pure CPU implementation can access the frame data much faster than the copy operations to and from the GPU memory. Since only a few simple operations are performed for each pixel of the frames—basically a conditional copy operation—the application is essentially bound by the memory bandwidth and the GPU is not able to demonstrate its advantage in computational density. Hence, for accelerating the compositing task in parallel, rendering it is key to achieve a high throughput through the accelerator.

In order to provide a fair comparison among different compositing implementations, we have made efforts to optimize our baseline implementation for the CPU. To this end, we have explored the use of SSE2 SIMD instructions which look like a good match for compositing, because four comparison and assignment operations can be performed in parallel. While this reduces the iterations of the compositing loop by a factor of four, the SIMD model requires to perform the same sequence of operations on each of the four pixels. This excludes the possibility to skip the copy operations for depth and color on a per-pixel basis, when the z-value comparison reveals that no copying is required. As a result, the use of SSE2 SIMD instructions did not provide a speedup over a sequential implementation. Hence, we will restrict our performance comparison in the following to the use of the FPGA and the use of the CPU without SSE2 instructions.

Figure 10 compares the overall performance, measured in frames/s, of the purely CPU-based parallel renderer with the FPGA-accelerated parallel renderer. Additionally, the figure shows the frame rates that could be achieved if the Infiniband interconnect was fully utilized. Figure 10 covers all three system states of the parallel renderer application. In the following, we discuss these states for a resolution of 1280 × 1024 pixel, For a small number of rendering nodes (up to three), the performance increases linearly, and since neither the network nor the master node is saturated, there is no benefit from using FPGA acceleration. Increasing the number of rendering nodes further (four to five nodes) makes the master a bottleneck. The CPU-based system achieves its maximum frame rate for four rendering nodes. In this system state, FPGA acceleration is highly useful as the improved compositing performance allows us to achieve a higher peak frame rate. For example, for four rendering nodes, the performance gain is 1.25×. At a certain point (six nodes and more), the aggregate bandwidth from the rendering nodes saturates the network. Figure 10 clearly shows that the performance of the FPGA-accelerated system is limited by the Infiniband bandwidth. Obviously, also in this state FPGA acceleration is beneficial and delivers an improvement in the frame rate of 2.1× for eight rendering nodes.

6. Applicability of Our Approach

While we have concentrated on one specific communication-bound application that performs only a few operations on each data element in this paper, the presented approach can be equally applied to other communication-bound streaming applications, such as digital filtering or image processing. For example, in [22, 23] we have applied the same methodology for implementing and optimizing an accelerator for the Cube Cut problem, which is an unsolved problem in geometry. The computational kernel of Cube Cut is very simple and requires to perform comparison operations on wide bit vectors which can be performed very cheaply in FPGAs. The key for achieving high performance is to find a decomposition of the functionality into possibly replicated cores and to determine memory access patterns that allow for exploiting the maximum bandwidth. The availability of accurate bandwidth characterization data is key to drive this systematic optimization process.

The IMORC infrastructure as such is however much more general and is not restricted to communication-bound streaming applications in any way. For example, in [2] we have presented the design and implementation of an accelerator for the kth nearest neighbor (KNN) thinning problem. KNN thinning is a computation-bound application with a more irregular, data-dependent data processing pattern. The KNN thinning accelerator also leverages the IMORC infrastructure and uses several cores which operate on a large data set. For maximizing the utilization of data processing cores in the presence of data-dependent processing times, the KNN accelerator uses dedicated hardware load balancing cores for scheduling processing jobs to cores. Even if this application performs much more complex operations on the data to be processed, the overall performance greatly depends on the times needed for data access, making a detailed architecture characterization necessary also for this application.

7. Conclusion

One specific challenge in mapping applications to high-performance reconfigurable computing platforms is to optimally utilize the available memory architecture and layout. Leveraging our IMORC infrastructure and architectural template for creating reconfigurable accelerators, we experimentally characterize the communication performance for cores mapped to the XtremeData XD1000 reconfigurable computer. The achievable bandwidth is strongly varying with request sizes and controller configurations. An optimal design point cannot be chosen locally, but requires us to consider the complete accelerator architecture and the overall application.

In the case study, we have elaborated on accelerating the 3D image compositing function of a parallel rendering application. It may come as a surprise to the reader that this operation can be successfully accelerated, given that compositing is a computationally simple operation without significant inherent parallelism. Hence, any performance improvements of an FPGA-based hardware accelerator will have to result from an optimized memory architecture that supports the streaming nature of the application. Although the DDR SDRAM attached to the Opteron processor provides a higher theoretical peak performance than the HyperTransport interface connecting the CPU and the FPGA and also a higher theoretical peak performance than the DDR SDRAM attached to the FPGA, our microbenchmarks have shown that the Opteron processor is not able to fully exploit this performance when using integer arithmetics. Although the SIMD unit of the Opteron processor should be able to efficiently implement such a streaming, the performance when using SSE even drops due to the increased number of operations to be performed. Manual cache prefetching as suggested by the microbenchmarks did not further increase the performance.

Using an FPGA compositing accelerator that leverage the IMORC infrastructure we are able to fully support the streaming nature of the application, utilizing the different kinds of memory at their maximum performance. Additionally, separate memories are used for storing data: host memory for the original frame- and z-buffers as received by the rendering nodes and off-chip memory for storing the intermediate buffers. Both memories are accessed in parallel, reducing the contention occurring on each of these memories.

Experiments with the overall parallel rendering application show that the FPGA accelerator is useful when the aggregate bandwidth of the distributed rendering nodes drives the compositing node into saturation. In this situation, the accelerator is able to double the achievable frame rate of the overall parallel renderer. This is significant since one FPGA module does not only boost the performance of a single CPU node, but also increases the usability and thus the practical value of a visualization compute cluster which constitutes a considerable investment.

Arguably, FPGA accelerators are not yet standard components in most clusters, while GPUs are standard components in many computer systems. To compare our accelerator with a GPU-based solution, we have ported our compositing application to the GPU. Our benchmarks have shown that offloading compositing to the GPU results in a lower performance than a pure CPU implementation due to the overheads caused by copying data between the CPU and the GPU and vice versa.

In this paper, we report on measurements conducted on a setup with a rather small number of rendering nodes and limited Infiniband bandwidth. A practical compute cluster will employ a higher number of rendering nodes connected by a faster network. This will further increase the pressure on the master node and allow the FPGA accelerator to achieve even higher speedups (see the network bandwidth limitation for the accelerator's performance in Figure 10). An FPGA module with one or several network interfaces would eliminate the HyperTransport bottleneck and allow us to further increase the compositing performance.

Obviously, alternative approaches to parallel rendering such as sort-first and sort-middle or parallel compositing will show other bottlenecks and, perhaps, remove the need for FPGA acceleration. A comprehensive study of different parallel rendering approaches and their bottlenecks is, however, beyond the scope of this work.

Acknowledgments

This work has been partially supported by the German Science Foundation (DFG) project DA155/31-1, ME872/11-1, FI1491/1-1 "Synchronisierte Analyse und 3D-Visualisierung paralleler Ablaufsimulationen in interaktiv erstellten Ausprägungen" (AVIPASIA) and the XtremeData-Sun-Altera-AMD University program by donating an XtremeData XD1000 Development system.

References

[1] T. Schumacher, C. Plessl, and M. Platzner, "IMORC: application mapping, monitoring and optimization for highperformance reconfigurable computing," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '09)*, pp. 275–278, IEEE Computer Society, 2009.

[2] T. Schumacher, C. Plessl, and M. Platzner, “An accelerator for k-th nearest neighbor thinning based on the IMORC infrastructure,” in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL ’09)*, pp. 338–344, IEEE, September 2009.

[3] L. Shannon and P. Chow, “Simplifying the integration of processing elements in computing systems using a programmable controller,” in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’05)*, vol. 2005, pp. 63–72, IEEE, 2005.

[4] C. Steffen, “Parametrization of algorithms and FPGA accelerators to predict performance,” in *Proceedings of the Reconfigurable System Summer Institute (RSSI ’07)*, pp. 17–20, 2007.

[5] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. D. George, “RAT: a methodology for predicting performance in application design migration to FPGAs,” in *Proceedings of the High-Performance Reconfigurable Computing Technologies and Applications Workshop (HPRTCA ’07)*, 2007.

[6] S. Koehler, J. Curreri, and A. D. George, “Performance analysis challenges and framework for high-performance reconfigurable computing,” *Parallel Computing*, vol. 34, no. 4-5, pp. 217–230, 2008.

[7] M. C. Smith and G. D. Peterson, “Analytical modeling for high performance reconfigurable computers,” in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS ’02)*, July 2002.

[8] M. C. Smith and G. D. Peterson, “Parallel application performance on shared high performance reconfigurable computing resources,” *Performance Evaluation*, vol. 60, no. 1–4, pp. 107–125, 2005.

[9] T. Schumacher, T. Süß, C. Plessl, and M. Platzner, “Communication performance characterization for reconfigurable accelerator design on the XD1000,” in *Proceedings of the International Conference on Reconfigurable computing and FPGAs (ReConFig ’09)*, pp. 119–124, IEEE Computer Society, Los Alamitos, Calif, USA, 2009.

[10] D. Slogsnat, A. Giese, and U. Brüning, “A versatile, low latency Hypertransport core,” in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’07)*, pp. 45–52, ACM, February 2007.

[11] “RAMspeed”, http://www.alasir.com/software/ramspeed/.

[12] “STREAM Benchmark”, http://www.cs.virginia.edu/stream/.

[13] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, “A sorting classification of parallel rendering,” Tech. Rep. TR94-023, 8, 1994.

[14] G. Stoll, M. Eldridge, D. Patterson et al., “Lightning-2: a high-performance display subsystem for PC clusters,” in *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’01)*, pp. 141–148, August 2001.

[15] S. Dominick and R. Yang, “Anywhere pixel router,” in *Proceedings of the ACM/IEEE 5th International Workshop on Projector Camera Systems (PROCAMS ’08)*, pp. 1–2, ACM, August 2008.

[16] S. Muraki, M. Ogata, K.-L. Ma et al., “Next-generation visual supercomputing using pc clusters with volume graphics hardware devices,” in *Proceedings of the Conference on Supercomputing*, p. 51, ACM, New York, NY, USA, 2001.

[17] L. Moll, A. Heirich, and M. Shand, “Sepia: scalable 3D compositing using PCI pamette,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM ’99)*, pp. 146–157, 1999.

[18] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich, “Scalable interactive volume rendering using off-the-shelf components,” in *Proceedings of the Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 115–121, IEEE, Piscataway, NJ, USA, 2001.

[19] S. Eilemann and R. Pajarola, “Direct send compositing for parallel sort-last rendering,” in *Proceedings of the ACM SIGGRAPH ASIA Courses*, December 2008.

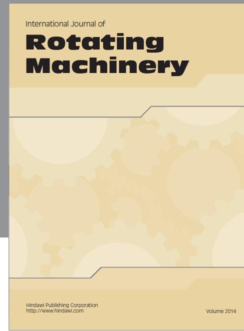
[20] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, “Parallel volume rendering using binary-swap compositing,” *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 59–68, 1994.

[21] “OpenMPI homepage”, http://www.open-mpi.org/.

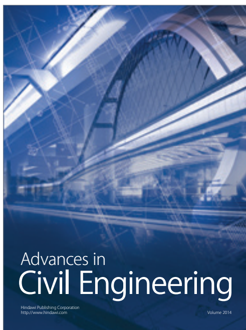
[22] T. Schumacher, E. Lübbers, P. Kaufmann, and M. Platzner, “Accelerating the cube cut problem with an FPGA-augmented compute cluster,” in *Proceedings of the ParaFPGA Symposium International Conference on Parallel Computing (ParCo ’07)*, vol. 38, pp. 749–756, John von Neumann Institute for Computing, Jülich, Germany, 2007.

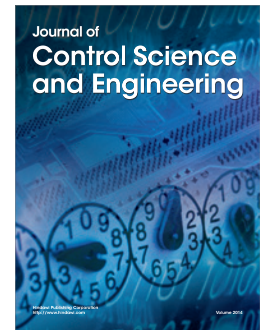
[23] T. Schumacher, *Performance modeling and analysis in highperformance reconfigurable computing*, Ph.D. Thesis, University of Paderborn, 2011.

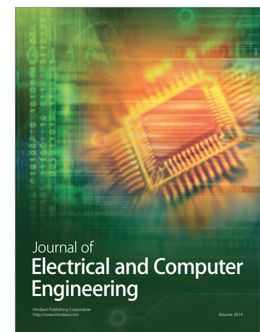
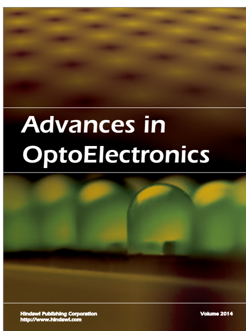
Journal of
Engineering

The Scientific
World Journal

International Journal of
Rotating
Machinery

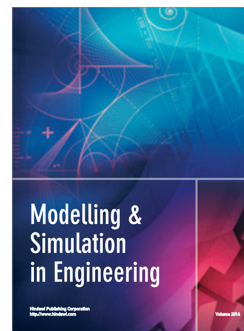
Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

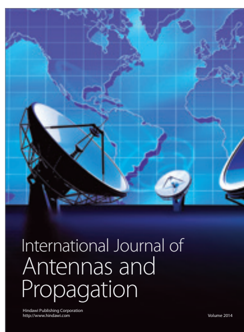
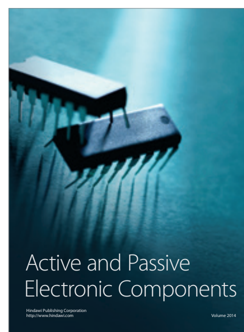
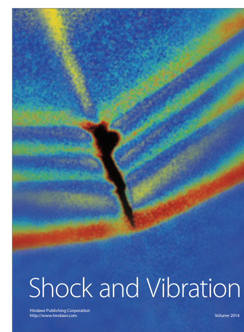
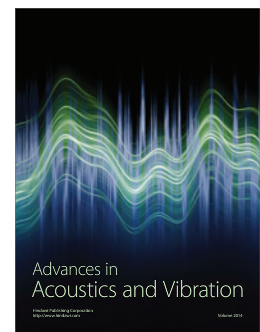
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration