# Programming Models for FPGA Application Accelerators

Muiris Woulfe, Eoin Creedon, Ross Brennan, Michael Doyle and Michael Manzke
Graphics, Vision and Visualisation Group (GV2)
Trinity College Dublin
Ireland
{woulfem, eoin.creedon, ross.brennan, mjdoyle, michael.manzke}@cs.tcd.ie

## Abstract

*Algorithms can be accelerated by offloading compute-intensive operations to application accelerators comprising reconfigurable hardware devices known as Field Programmable Gate Arrays (FPGAs). We examine three types of accelerator programming model – master-worker, message passing and shared memory – and a typical FPGA system configuration that utilises each model. We assess their impact on the partitioning of any given algorithm between the CPU and the accelerators. The ray tracing algorithm is subsequently used to review the advantages and disadvantages of each programming model. We conclude by comparing their attributes and outlining a set of recommendations for determining the most appropriate model for different algorithm types.*

## 1. Introduction

Application accelerators utilising Field Programmable Gate Arrays (FPGAs) have been shown to improve the performance of a wide variety of algorithms including collision detection for graphics simulations [27, 17], mathematical computation [18], ray tracing [26] and scientific simulation [11]. Improvements garnered by these accelerators are achieved through parallel implementations of the algorithms in hardware. The degree of parallelism can be scaled by increasing the size of the FPGAs or through the use of multiple interconnected FPGAs and a parallel programming model. The approach taken to increase performance is to split the algorithm across a CPU and one or more FPGAs. The algorithm split is determined by the accelerator configuration; different configurations have different programming model requirements that can be quantified in terms of scalability, latency, bandwidth and programmability.

In this paper, we investigate FPGA application accelerator programming models as they relate to systems comprising both single and multiple interconnected FPGAs. We examine three programming models – master-worker, message passing and shared memory – defining the characteristics of each approach and outlining their relation to the implementation of a sample algorithm – ray tracing. Based on our experiences of using these approaches, we formulate a set of recommendations outlining the most appropriate programming model for a range of algorithm types.

## 2. Background and related work

FPGA application accelerators have traditionally been attached to the CPUs of host PCs using the peripheral bus. To overcome the limitations imposed by these systems, researchers have focused on increasing their throughput [3] and on configuring the CPU-FPGA application interface to hide communication overheads [24]. With the introduction of FPGA accelerators that use direct Front Side Bus (FSB) [21] or HyperTransport [13] interfaces, the need for an appropriate programming model has intensified in order to ensure the best possible performance is achieved. Gelado et al [6] have designed an efficient communication paradigm for these systems and Underwood et al [24] have investigated ways to best achieve the performance potential of the acceleration logic. They show that a correctly implemented Application Programming Interface (API) can have substantial benefits for algorithm acceleration. Tripp et al [23] demonstrate how such an API can be used to partition an algorithm across software and hardware efficiently. The work focuses on traditional CPU-FPGA configurations but can be readily applied to standalone FPGA acceleration clusters.

These standalone clusters consist of standard CPU-based compute nodes and interconnected FPGA nodes. A parallel programming model – message passing or shared memory – is required for FPGA intercommunication. Message passing is used by many research groups. TMD-MPI [19] demonstrates the practicality of this programming model, while the Reconfigurable Compute Cluster (RCC) [20] and Baxter et al [1] investigate its suitability for point-to-point

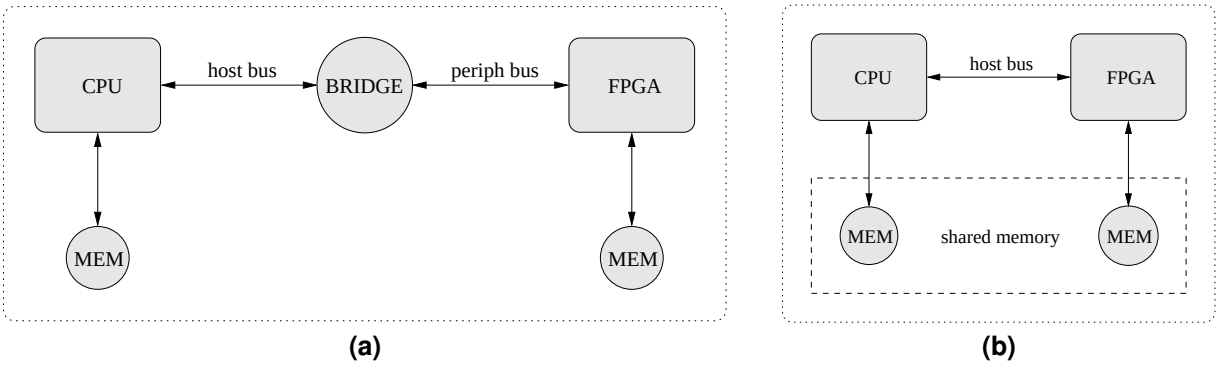


**Figure 1. Schematics of two generic master-worker accelerators. (a) PCI or PCIe. (b) HyperTransport.**

intercommunication. Creedon and Manzke [4] and Pedraza et al [16] present application and communication results for FPGA message passing clusters.

A number of shared memory implementations also exist. The Research Accelerator for Multiple Processors (RAMP) [22] examines the application of a global address space programming model using Unified Parallel C (UPC) as a programming language for massively parallel, many-core systems. Brennan et al [2] use a Distributed Shared-Memory (DSM) FPGA compute cluster with Scalable Coherent Interface (SCI) as the communication fabric. The shared memory abstraction is implemented directly in the FPGAs and combines the scalability of a network-based architecture with the convenience of the shared memory programming model.

## 3. Overview

Programming models for FPGA application accelerators may be classified into three broad types based on their interaction with each other and the host system. We classify the programming approaches as master-worker, message passing and shared memory. Although any programming model can theoretically be used with any accelerator configuration, each programming model is typically used with a specific configuration to maximise performance. This section describes the characteristics of typical configurations used with each programming model.

### 3.1. Master-worker

Accelerators using the master-worker programming model comprise one or more CPUs interoperating with one or more FPGAs inside a single PC. These can be separated by means of a bridge, as with Peripheral Component Interconnect (PCI) [14] and PCI Express (PCIe) [15], or directly connected, as with FSB and HyperTransport [8], as illustrated in Figure 1. They are very constrained in the number of FPGAs supported and their defining characteristic is that all FPGA intercommunication must be performed via the CPUs.

The interface between the CPUs and the FPGAs is provided by the FPGA vendor's board support package, which comprises appropriate software libraries and hardware modules. After deploying the algorithm, a typical sequence of operations is as follows. A CPU transfers data to an FPGA's local memory or to a shared memory location. The CPU subsequently writes to the FPGA's registers indicating the number of items transferred and instructing the FPGA to proceed with execution. The CPU either waits for an interrupt or polls an FPGA register to determine when execution is complete. The CPU then reads from another register to establish the quantity of data stored, before reading the results. All operations are performed consecutively with the various read and write operations functioning as implicit barriers.

### 3.2. Message passing

In addition to these master-worker accelerators, it is possible to create clusters of networked FPGAs. These FPGA clusters require the ability to exchange and synchronise data to perform all algorithm computations. They consist of one or more commodity PCs and one or more FPGAs interconnected across a network, as illustrated in Figure 2. In these systems, the FPGAs are viewed as peers while the commodity PCs are used to initialise and provide data to the FPGAs.

Message passing has been investigated in the context of distributed memory FPGA clusters. Algorithm data is local to each node and is exchanged between nodes using explicit communications, which are controlled and requested by the programmer.

Like the master-worker configuration, a message passing

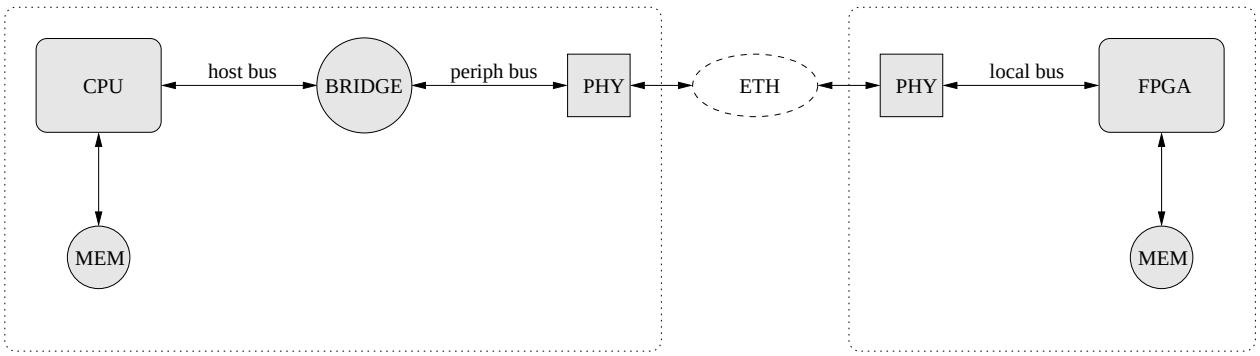


**Figure 2. Schematic of a generic message passing accelerator.**

accelerator receives data from a CPU across the network and performs computations on that data. However, unlike the master-worker configuration, the networked FPGAs are peers and can be programmed to exchange algorithm data directly, allowing all computations to occur in parallel.

From the programmer's perspective, a single API is provided which enables the necessary send and receive operations, both on the central PC and FPGAs. All network operations are performed transparently by the API so that the programmer's only concern is with parallelising the algorithm. To use a message passing configuration correctly, the programmer requires the ability to explicitly perform communications, to access and use local memory and to be able to control the operations of the hardware. At the core of the algorithm will be a computational processing element, similar to one that could also be used by master-worker systems. This is made possible by the API providing the necessary operational controls independent of the implementation used by different message passing accelerators.

### 3.3. Shared memory

Shared memory accelerators comprise one or more CPUs interoperating with one or more FPGAs, which may be housed in multiple nodes, communicating across a tightly-coupled interconnect such as SCI [9], as illustrated in Figure 3. The nodes share a global memory address space and communication is accomplished through shared variables or messages deposited in shared-memory buffers. There is no requirement for the programmer to explicitly manage the movement of data, unlike in message passing implementations.

The FPGAs are typically placed on separate nodes along with some local memory and additional hardware components that support the network interconnect. The choice of interconnect affects the way in which the coprocessor nodes are connected to the commodity PCs. Memory references made by one of these nodes into its own address space are automatically translated into a shared memory transaction and transported to the correct remote node. The remote node translates this transaction back into a local memory access, providing a hardware DSM implementation.

In shared memory models, any processing element can theoretically make direct hardware memory references into the global address space without requiring knowledge of whether the memory location that it is reading from or writing to is situated locally or remotely. In practice, it is still important for the processing element to know if it is accessing local or remote memory due to the increased latencies involved in accessing remote memory. Consequently, it is important to outline a set of basic shared memory communication primitives that implement certain standard parallel computation features such as process locks and barriers. A method of hiding latencies from the processes must also be developed. This may include commonly used techniques such as transfer scheduling, block transfers and prefetching.

Both hardware and software-based DSM systems can be used to implement a shared memory abstraction across multi-processor architectures, combining the scalability of network-based architectures with the convenience of shared-memory programming by providing a single global virtual address space. Nitzberg and Lo [12] provide an overview of several different hardware- and software-based DSM systems.

Software Infrastructure for SCI (SISCI) [5] is the API that covers different aspects of how SCI interconnects can be accessed from host systems. It specifies the general functions, operations and data types made available as part of the SCI standard. It also takes care of mapping local address segments into the shared memory address space and checking whether errors have occurred during data transfer. Low level communication among nodes is accommodated by a set of SCI transactions and protocols that include support for reading and writing data, cache coherence, synchronisation and message passing primitives. SCI supports

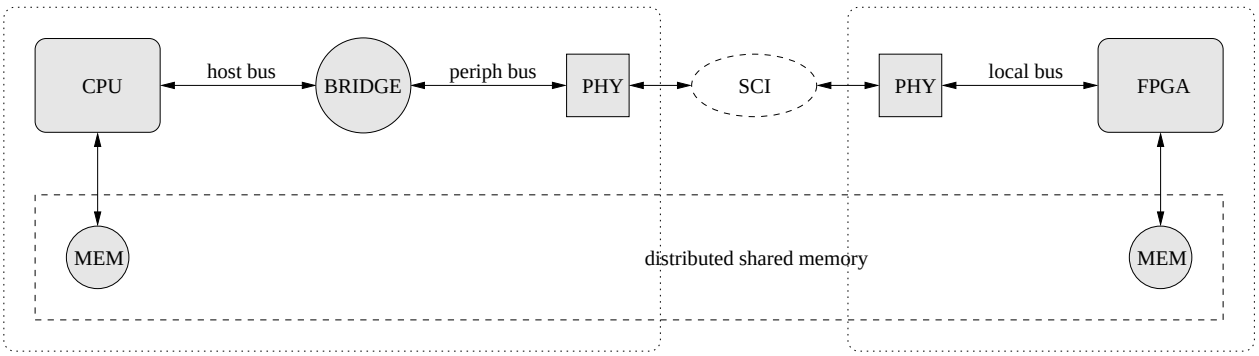


**Figure 3. Schematic of a generic shared memory accelerator.**

multiprocessing with cache coherence for the very general DSM model. The use of cache coherence is optional and applications may choose to maintain cache coherence under software control instead of using SCI's automatic coherence mechanism.

## 4. Algorithm example

An illustrative example of an algorithm that would benefit from the use of such accelerators is the ray tracing algorithm used in graphics rendering. Although historically ray tracing has only been practical for offline rendering, interactive and realtime frame rates have recently been achieved [25]. The algorithm renders images by casting rays from the camera viewpoint into a scene, intersecting these rays with scene geometry and finally shading these intersection points to determine the pixel colour. Ray tracing has very high computational requirements and may also have high bandwidth requirements when dealing with complex geometry. We therefore chose this algorithm as a means to compare the three application accelerators in the context of a realistic algorithm.

Ray tracing has been applied to master-worker accelerators [26]. The strategy employed has typically been to offload parts of the algorithm, such as ray/object intersections, and to reserve a large portion of the algorithm for computation on the CPU. The primary advantage of these accelerators is their high bandwidth. Moreover, since parts of the algorithm remain on the host machine, complex programming tasks, such as the building of acceleration data structures, can be implemented in software. This approach is also amenable to commodity hardware in desktop PCs. However, master-worker systems lack the scalability of other approaches and are likely to possess limited local memory, precluding the rendering of large scenes.

Ray tracing has also been applied with some success to message passing environments [7]. The large memory space of such machines is of use in rendering large scenes, as the scene geometry can be split evenly among the accelerators. Each accelerator can then calculate a subset of the scene's ray operations. Increasing the number of active nodes is simpler than for master-worker accelerators and the large number of nodes could potentially allow a greater portion of the algorithm to be implemented on the FPGAs, possibly splitting different phases of the algorithm over multiple accelerators. In addition, as the number of processors increase, the overall memory capacity of the system similarly increases. Therefore, the benefits of message passing accelerators for ray tracing are twofold. The disadvantages of this approach are the overhead associated with explicit communication between nodes, the bandwidth requirements associated with large data transfers and the potential for load imbalance across large numbers of nodes.

Shared memory accelerators are also suited to parallelising the ray tracing algorithm [10]. Mapping the algorithm to these accelerators typically follows a similar approach to message passing implementations. They possess similar advantages, including scalability in both the number of processing nodes and the memory capacity of the system. Such accelerators are also easier to program as communication need not be performed explicitly. Furthermore, the largely unpredictable memory access patterns of the ray tracing algorithm means that each node will likely require a substantial amount of data from the other nodes in the system, perhaps making shared memory accelerators preferable to message passing accelerators. Shared memory accelerators share the need for high bandwidth communication and the potential for load imbalance but do not suffer the overhead of explicit communication. However, the complexity involved in building a shared memory accelerator is greater and their cost is often prohibitive.

## 5. Comparison

Our analysis of the ray tracing application example highlights some factors that influence which programming

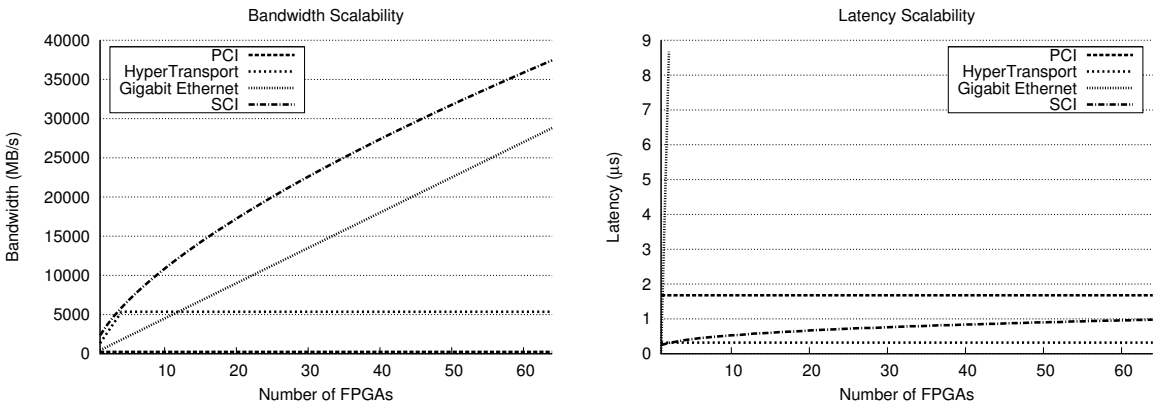


**Figure 4. Graphs illustrating the scalability of the bandwidth and latency for four interconnect types.**

model is optimal for a given algorithm. By generalising these findings, it is evident that the factors of interest to programmers are scalability, communication speed, cost-effectiveness and programmability. In this section, we consider the different programming models under these categories to reach a conclusion on the optimal accelerator for different algorithm types.

Scalability comprises algorithm, interconnect and system scalability. Algorithm scalability is determined by the quantity of logic resources available and hence by the number of FPGAs in the system. Interconnect scalability is determined by the bandwidth available for communication in addition to the maximum number of devices permitted by the interconnect technology. System scalability comprises both algorithm and interconnect scalability while also considering the algorithm implementation in use. For example, a suboptimal implementation executing on a large number of FPGAs may not achieve the same performance as a superior implementation executing on fewer FPGAs. Ignoring implementation details, it can be seen that the master-worker model is particularly constrained in terms of scalability as the number of vacant interface slots in a single PC typically limits this model to five or fewer FPGAs. This limit may increase in the future if manufacturers provide additional interface slots although such increases will always be limited by physical space constraints. Even with such increases, intercommunicating via the CPUs would remain a bottleneck but the impact of this will lessen as CPU and interconnect technology improves. In contrast, message passing and shared memory are limited only by interconnect scalability.

Communication speed involves two distinct elements – bandwidth and latency. Bandwidth is the quantity of data that can be transferred between FPGAs over a given time, while latency is the overhead associated with initialising a transfer. Master-worker accelerators can achieve excellent bandwidth and latency, as in a HyperTransport implementation, although traditionally speeds have been lower, as in a PCI implementation. In contrast, message passing is traditionally interconnect agnostic and its communication speeds are determined by its implementation. For example, an Ethernet implementation has low bandwidth and high latency. Shared memory typically achieves good bandwidths and latencies, which are vital in supporting the continuous remote memory requests. This contrasts with message passing accelerators, where the bandwidth is typically better but the latency is typically worse. The scalability of these bandwidths and latencies is plotted in Figure 4.

Cost-effectiveness is related to the cost of purchasing a particular system. Master-worker systems are relatively expensive to purchase, but typically only a single FPGA is required so that the overall cost is reduced. Message passing and shared memory are typically purchased as multiple FPGAs, leading to increased costs. Of these two approaches, message passing has a lower cost as it is able to work with inexpensive equipment. Shared memory requires specialised equipment and will cost more per-FPGA.

In terms of programmability, master-worker systems are the simplest since the CPUs remain in control of the entire system, removing the need to consider asynchronous communication between FPGAs. Message passing is more complicated since FPGAs can intercommunicate asynchronously. Moreover, the programmer must consider the location of any required data before issuing a message to access that data. Shared memory systems can also communicate asynchronously between FPGAs but they are simpler to program than message passing systems since all memory is globally accessible and the programmer does not need to consider the location of the data when issuing a memory request. To achieve maximum performance, it is

still necessary to consider whether a memory access is local or remote but this does not add significant complexity.

Based on these differences, we have created a series of recommendations that outline the most amenable accelerator for different algorithm types. These recommendations are graphed in Figure 5.

**Master-worker accelerators** are suitable for algorithms amenable to acceleration using approximately five or fewer FPGAs. This limitation is determined by a combination of scalability and cost, but may increase with future architectural developments.

**Message passing accelerators** are appropriate for algorithms that can be parallelised across large numbers of FPGAs, where high latency is not an overriding concern but where keeping costs minimal is an important factor.

**Shared memory accelerators** are recommended for algorithms that can be parallelised across large numbers of FPGAs, where latency sensitivity impacts algorithm performance and efficient execution must be achieved irrespective of cost.

We do not consider programmability to be a deciding factor in determining the optimal application accelerator for a given application type, since this remains relatively constant and is mostly unaffected by the algorithm under consideration. However, if programmability did vary significantly, we feel that it would not be a major factor as it should be possible to overcome all of the programming challenges that could arise without significant difficulty.

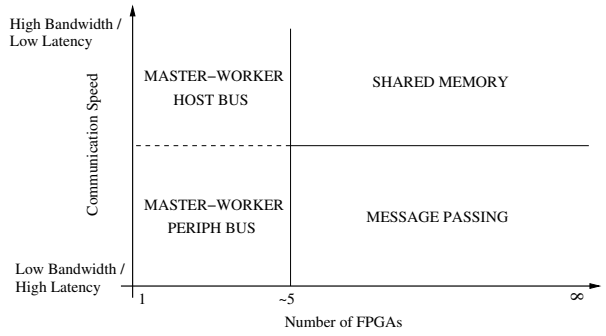


**Figure 5. Graph illustrating the appropriate accelerator for an algorithm based on system scalability and communication speed.**

## 6. Conclusions

This paper has examined three programming models for FPGA application accelerators and considered the suitability of each for performing parallel ray tracing. Based on this examination, it is evident that the choice of programming model significantly affects scalability, communication speed, cost-effectiveness and programmability. Using these four categories, we have formulated a set of recommendations for selecting the model best suited to different algorithm types. We believe that these recommendations should prove invaluable to programmers when attempting to select the most appropriate model and accelerator combination for a particular algorithm.

## References


[1] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. Maxwell – a 64 FPGA supercomputer. Technical report, FHPCA, 2007.

[2] R. Brennan, M. Manzke, K. O'Conor, J. Dingliana, and C. O'Sullivan. A scalable and reconfigurable shared-memory graphics cluster architecture. In *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2007)*, pages 284–290. CSREA Press, June 2007.

[3] R. Chamberlain, B. Shands, and J. White. Achieving real data throughput for an FPGA co-processor on commodity server platforms. In *Proceedings of the 1st Workshop on Building Block Engine Architectures for Computers and Networks (BEACON 2004)*, pages 1–6. ACM, Oct. 2004.

[4] E. Creedon and M. Manzke. Scalable high performance computing on FPGA clusters using message passing. In *Proceedings of the 18th International Conference on Field-Programmable Logic and Applications (FPL 2008)*, pages 443–446. IEEE, Sept. 2008.

[5] Dolphin Interconnect Solutions. SISCI API user guide. Technical report, May 2001.

[6] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W. W. Hwu. CUBA: An architecture for efficient CPU/co-processor data communication. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS 2008)*, pages 299–308. ACM, June 2008.

[7] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics Applications*, 9(6):12–26, Nov. 1989.

[8] HyperTransport Consortium. HyperTransport I/O link specification revision 3.10, 2008.

[9] IEEE. IEEE standard for Scalable Coherent Interface (SCI), 1992.

[10] M. J. Keates and R. J. Hubbold. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum*, 14(4):189–202, Dec. 1995.

[11] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, pages 182–199. IEEE Computer Society, Apr. 2002.

[12] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, Aug. 1991.

[13] M. Nüssle, H. Fröning, A. Giese, H. Litz, D. Slogsnat, and U. Brüning. A Hypertransport based low-latency reconfigurable testbed for message-passing developments. In *Proceedings of the 2nd Workshop Kommunikation in Clusterrechnern und Clusterverbundsystemen (KiCC 2007)*, pages 1–6. Technische Universität Chemnitz, Feb. 2007.

[14] PCI-SIG. Conventional PCI 3.0, 2004.

[15] PCI-SIG. PCI Express base specification 2.1, 2007.

[16] C. Pedraza, E. Castillo, J. Castillo, C. Camarero, J. L. Bosque, J. I. Martínez, and R. Menendez. Cluster architecture based on low cost reconfigurable hardware. In *Proceedings of the 18th International Conference on Field-Programmable Logic and Applications (FPL 2008)*, pages 595–598. IEEE, Sept. 2008.

[17] A. Raabe, S. Hochgürtel, J. Anlauf, and G. Zachmann. Space-efficient FPGA-accelerated collision detection for virtual prototyping. In *Proceedings of the 2006 Design, Automation and Test in Europe Conference and Exhibition (DATE 2006)*, pages 206–211. European Design and Automation Association, Mar. 2006.

[18] S. Rousseaux, D. Hubaux, P. Guisset, and J.-D. Legat. A high performance FPGA-based accelerator for BLAS library implementation. In *Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute (RSSI 2007)*, pages 1–10. National Center for Supercomputing Applications, July 2007.

[19] M. Saldaña and P. Chow. TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs. In *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications (FPL 2006)*, pages 1–6. IEEE Computer Society, Aug. 2006.

[20] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, P. Beeraka, K. Datta, D. Andrews, R. S. Miller, and D. Stanzione Jr. Reconfigurable Computing Cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing. In *Proceedings of the 15th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 127–238. IEEE Computer Society, Apr. 2007.

[21] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell Jr., L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi. High-performance Ethernet-based communications for future multi-core processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC 2007)*, pages 1–12. ACM, Nov. 2007.

[22] A. Schultz. RAMP Blue: Design and implementation of a message passing multi-processor system on the BEE2. Master's thesis, University of California at Berkeley, 2006.

[23] J. L. Tripp, A. Å. Hanson, and M. Gokhale. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC 2005)*, pages 1–12. IEEE Computer Society, Nov. 2005.

[24] K. D. Underwood, K. S. Hemmert, and C. Ulmer. Architectures and APIs: Assessing requirements for delivering FPGA performance to applications. In *Proceedings of the ACM/IEEE 2006 Supercomputing Conference (SC 2006)*, pages 1–12. IEEE Computer Society, Nov. 2006.

[25] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG 2003)*, pages 1–9. IEEE Computer Society, Oct. 2003.

[26] S. Woop, J. Schmittler, and P. Slusallek. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics – Proceedings of the 32nd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2005)*, 24(3):434–444, July–Aug. 2005.

[27] M. Woulfe, J. Dingliana, and M. Manzke. Hardware accelerated broad phase collision detection for realtime simulations. In *Proceedings of the 4th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS 2007)*, pages 79–88. Eurographics Association, Nov. 2007.