Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 652620, 16 pages
doi:10.1155/2010/652620

Research Article

Traversal Caches: A Framework for FPGA Acceleration of Pointer Data Structures

James Coole and Greg Stitt

Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611-6550, USA

Correspondence should be addressed to Greg Stitt, gstitt@ece.ufl.edu

Received 9 March 2010; Revised 10 October 2010; Accepted 13 December 2010

Academic Editor: Viktor K. Prasanna

Field-programmable gate arrays (FPGAs) and other reconfigurable computing (RC) devices have been widely shown to have numerous advantages including order of magnitude performance and power improvements compared to microprocessors for some applications. Unfortunately, FPGA usage has largely been limited to applications exhibiting sequential memory access patterns, thereby prohibiting acceleration of important applications with irregular patterns (e.g., pointer-based data structures). In this paper, we present a design pattern for RC application development that serializes irregular data structure traversals online into a traversal cache, which allows the corresponding data to be efficiently streamed to the FPGA. The paper presents a generalized framework that benefits applications with repeated traversals, which we show can achieve between 7x and 29x speedup over pointer-based software. For applications without strictly repeated traversals, we present application-specialized extensions that benefit applications with highly similar traversals by exploiting similarity to improve memory bandwidth and execute multiple traversals in parallel. We show that these extensions can achieve a speedup between 11x and 70x on a Virtex4 LX100 for Barnes-Hut n-body simulation.

1. Introduction

Numerous studies have shown that field-programmable gate arrays (FPGAs) and other reconfigurable computing (RC) devices can achieve order of magnitude or larger performance improvements over microprocessors [1, 2] for application domains including embedded systems, digital signal processing, and scientific computing. In addition to providing superior performance, FPGAs have also been shown to improve power consumption and energy efficiency compared to microprocessors and alternative accelerator technologies such as graphics processing units (GPUs) [3].

The advantages of FPGAs result from the ability to implement custom circuits that exploit tremendous amounts of parallelism, often using deep pipelines with additional parallelism ranging from the bit level up to the task level. As a consequence of enabling large amounts of parallelism, however, FPGA circuits require high memory bandwidth to avoid frequent pipeline stalls that can prevent potential speedups from being realized [4].

One significant limitation of FPGAs is that due to the requirement for high memory bandwidth, FPGAs are typically unable to accelerate applications with irregular access patterns [5]. In this paper, we define an irregular access pattern as a sequence of memory accesses that are not sequential in memory, or patterns that cannot be buffered based on compile time analysis [6]. Although irregular access patterns can result in many different ways, in this paper, we focus on the common example of pointer-based data structure traversals, such as lists and trees. Traversals of pointer-based structures reduce effective memory bandwidth through indirection, which requires multiple memory accesses to fetch a single item of data. In addition, consecutively accessed nodes of pointer-based structures are rarely stored at consecutive memory locations, resulting in poor cache performance and requiring frequent expensive row address strobes (RAS) in SDRAM memories [7]. Nonsequential accesses also prevent the use of specialized burst memory access modes [7] that provide maximal memory performance.

To enable usage of FPGAs on more pointer-based applications, we present a framework for improving memory bandwidth of traversals of pointer-based data structures. As illustrated in Figure 1, the presented framework dynamically serializes data accessed during traversal of

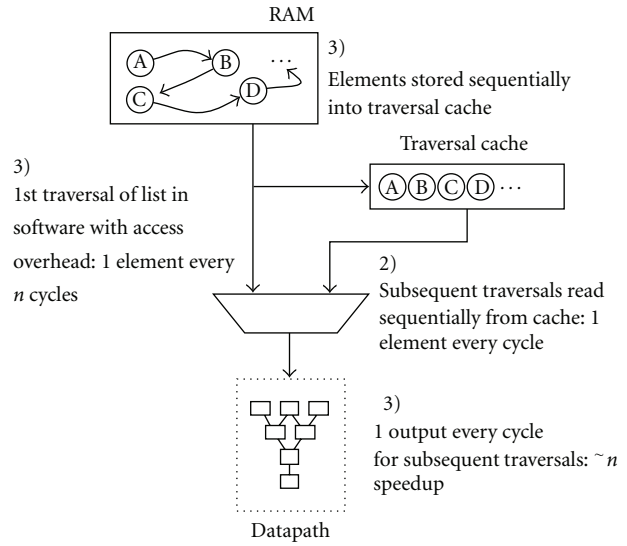

Figure 1: Conceptual idea of *traversal caches*, where repeated traversals of pointer-based structures are stored sequentially to improve memory bandwidth to custom circuits.

large pointer-based data structures and stores the serialized representation in a local memory, referred to as a traversal cache, where it can be more efficiently accessed by the FPGA. By serializing the elements accessed in a traversal, traversal caches eliminate the bandwidth reduction caused by indirection and nonsequential data storage while also enabling memory burst modes.

The simplest use of traversal caches involves storing repeated traversals of a pointer data structure in the cache memory. When the cache is larger than the typical traversal, multiple traversals may be stored at different locations in the cache to reduce the miss rate, with software managing placement and bookkeeping. This usage model is most analogous to traditional caches, with traditional cache lines corresponding to entire traversals.

In this repetition-based model, the concept of a cache hit corresponds to an FPGA repeating an identical traversal that is already stored in the cache. Traversal cache misses result from two situations. First, if an application requires a traversal not currently stored in the cache, the framework must first load the corresponding traversal (i.e., a conflict or compulsory miss). Second, if any part of the application modifies the data involved with a traversal stored in the cache, the cache representation must be marked invalid, resulting in a compulsory miss on the next access. Therefore, the overall performance improvement resulting from the use of a traversal cache is highly dependent on the invalidation rate. For applications with high invalidation rates (or otherwise low repetition), the framework provides the least benefit, behaving similarly to a system with no traversal cache with additional overhead due to frequent data structure serialization. In Section 5, we show that these techniques can achieve large speedups for applications with different rates of repeated traversals and are capable of matching sequential-access (array-based) software in some situations.

Though this model is simple and generic, it achieves limited speedup for applications that never or very rarely have identically repeated traversals resulting in a high invalidation rate and constant thrashing. However, even applications that rarely repeat a traversal often exhibit a large similarity between traversals. This can be the case for applications, where a large data structure remains static over a (relatively) long period of time and traversals occur in the same order through the data structure (e.g., preorder for a tree). In these situations, we show that additional data can be included in the traversal cache to describe the data structure itself, allowing hardware to generate the multiple traversals from the cache without additional software intervention.

This similarity-exploiting use model has the added benefit of allowing hardware to generate and process multiple traversals in parallel, which can enable a large amount of data reuse when the similarity between traversals is high, resulting in additional speedup. This approach to traversal caches also handles repeated traversals as a special case, automatically handling the repeated traversals in parallel, enabling for example greater loop unrolling by improving access to memory. In this paper, we evaluate these extensions exploiting similarity between traversals in the Barnes-Hut n-body simulation algorithm [8] and discuss how the framework could be used for other applications. The experimental results show that the memory bandwidth bottleneck is almost completely eliminated for highly similar traversals, resulting in kernel speedup that increases approximately linearly with area. In Section 5, we show speedups from 11x to 70x for instances of the Barnes-Hut n-body simulation algorithm, which essentially never repeats traversals.

Besides requiring a high level of similarity between traversals, current implementations require manual creation of the traversal cache, including application-specific logic in the case of the similarity-based framework and modification of existing application codes to perform cache management. These steps are mostly data structure specific, enabling the creation of traversal cache compatible libraries that can be used by multiple applications; however, future work will introduce techniques for performing these tasks as part of high-level synthesis. The frameworks also generally assume that at least one entire traversal fits in the traversal cache. Since the cache is typically implemented using off-chip SDRAM (commonly available on commercial FPGA accelerator boards or shared with the host processor), this assumption is not unreasonable. Otherwise, ensuring this condition is generally a matter of partitioning the application to run within the platform's resource constraints, as would also be required by multinode computation typical for such large problems. Other considerations are discussed where appropriate later in this document.

The paper is formatted as follows. Section 2 discusses previous work. Section 3 describes the traversal cache framework supporting repeated traversals. Section 4 presents a generalization of the framework to support parallel processing of similar traversals through a case study on Barnes-Hut n-body simulation. Section 5 presents experimental results. Conclusions and future work are discussed in Section 6.

2. Previous Work

Previous work has investigated hardware synthesis techniques for code utilizing pointers. In [9], Semeria integrated alias analysis techniques into a high-level synthesis tool flow to help resolve aliases at compile time, thus enabling further optimization and utilization of multiple memories. These techniques were later extended in [10] to support dynamic memory allocation by integrating a memory manager into the synthesized circuit. The traversal cache framework is a complementary approach that targets hardware/software codesign by not restricting the hardware to using a separate memory management unit—a situation that may not be practical or efficient for all FPGA accelerators.

Diniz and Park [5] utilized FPGAs to create smart memory engines capable of reorganizing data from pointer-based data structures to improve data locality and cache performance. The traversal cache framework has a similar goal, but does not reorder data in main memory, and thus does not have the alias restrictions of [5]. Furthermore, the traversal cache framework can be applied to any FPGA accelerator, including those that access memory via DMA.

Impulse [11] introduced a memory controller that remaps physical addresses to improve cache performance and memory bandwidth. Impulse remapped data using specialized languages and operating system support. The traversal cache framework does not have these restrictions, and only requires use of a specific library.

Specialized cache architectures and memory allocation techniques have also been introduced to better handle pointer operations. Collins et al. [12] introduced the pointer cache to efficiently handle chained pointer traversals by prefetching data based on pointer transitions. Weinberg [13] eliminates pointer-based memory accesses at runtime by caching previous evaluation results. Hu et al. [14] predicts memory behavior and utilizes a time-based victim cache to improve hit rate. Chilimbi et al. [15] discuss manual programming practices that can improve data locality of pointer structures, in addition to automatic data layout optimizations that are integrated into garbage collection. Calder et al. [16] considers cache-conscious data placement for heap and stack objects. The traversal cache framework provides similar optimizations for FPGAs, which commonly have direct access to memory and, therefore, do not benefit from traditional cache optimization.

Smart buffers [6] are a data-caching scheme for FPGAs that prevent reused data from being read multiple times from memory. Smart buffers greatly improve memory bandwidth and FPGA performance but do not support pointer-based data structures.

Numerous compiler optimizations [7, 17, 18] modify data layout at compile time based on memory access patterns. Traversal caches improve on these previous approaches by supporting pointer-based data structures. Baradaran and Diniz [19] and some parallelizing compilers [20] also consider mapping accesses to multiple FPGA-internal memory resources, improving parallelism. The similarity-based traversal cache also optimizes for parallelism but seeks it through access coalescing, which enables efficient use of large external memories such as SDRAMs that are optimized for sequential accesses.

Traversal caches were originally introduced in [21], which presented a general framework that exploits identically repeated traversals. In [22], the approach was extended to exploit similarity. This paper integrates the contributions of previous work and evaluates the framework for additional examples.

3. Traversal Cache Framework

In this section, we present the traversal cache framework for caching and reusing repeated traversals. Section 4 discusses extensions for exploiting nonidentical traversals with high similarity. The following subsections discuss the functionality required to enable the traversal cache framework, which includes the system architecture, hardware/software communication, and traversal cache software library.

3.1. System Architecture. Figure 2 illustrates the system architecture used by the traversal cache framework, which consists of two main components: the microprocessor and the FPGA accelerator. The microprocessor executes all software regions of the code and assists in fetching traversals whenever the traversal cache is empty or invalidated. The FPGA accelerator implements custom circuits to speed up computationally intensive kernels of the targeted application, typically implemented as deeply pipelined datapaths, which read data from the traversal cache to efficiently handle pointer-based data structures. Future work will look at automatically partitioning traversal fetch and serialization logic as part of high level synthesis.

In addition to the processor's main memory, the framework assumes two separate memories local to the FPGA that are used to simultaneously stream data in and out of the accelerator—a common architecture in commercially available FPGA accelerator cards and other accelerators. The microprocessor also has read/write access to the FPGA's memories, either through an external controller or a controller implemented by the FPGA. Communication details are discussed in the next section.

Details of the FPGA accelerator are shown in the expanded box in Figure 2. The controller interprets messages from the microprocessor and enables the address generators when the microprocessor activates the accelerator. The address generators control the input and output memories to read a stored traversal or write datapath results. Since elements within a traversal are stored in order in the traversal cache, the address generators typically need only to support linear access patterns, possibly with a fixed stride. In other situations, more complicated address generators with buffering [6] can be substituted. The datapath is a pipeline implementing an application kernel, which can usually be isolated from the specifics of the framework allowing reuse from previous design efforts or the use of existing HLS techniques.

The input memory hierarchy used by the accelerator consists of three possible data sources. The accelerator uses

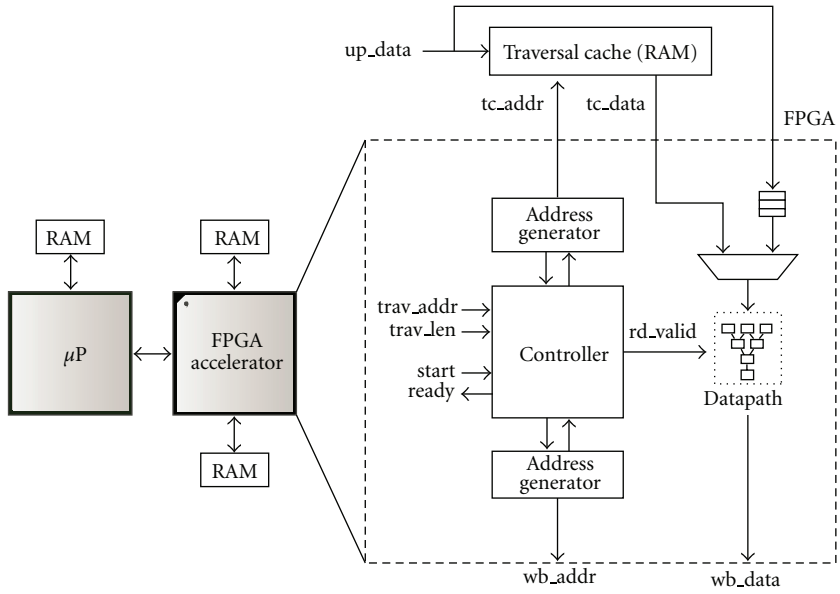

Figure 2: System architecture of the generalized traversal cache framework. On the first of a repeated traversal of a pointer-based data structure, software sends the elements to the FPGA on *up_data*, which is simultaneously written to the cache memory and consumed by the accelerator datapath. On subsequent traversals, the accelerator reuses the sequence of data in the cache, which can be read sequentially using simple constant stride address generators.

one of the two external RAMs shown in Figure 2, leaving the other for buffering results. In our implementation, processor writes to the FPGA's traversal cache (RAM) are handled by logic on the FPGA itself, allowing incoming data to be used directly while a traversal is being transferred to the traversal cache, which can help mitigate the effect of cache misses.

There is obviously flexibility in this architecture for systems with different needs. Our implementation uses SDRAMs for the traversal cache (read memory) and results buffer (write memory); however, systems with more modest memory requirements might be able to use external SRAM or distributed memory on the FPGA itself. The high latency of nonsequential SDRAM accesses impacts the framework's performance for similar traversals (discussed in Section 6) so that the use of lower latency memories could also result in performance improvements. Although not a focus of this paper, potential optimizations in this area are discussed in more detail in Section 4.

Systems with tighter integration between the FPGA and CPU could also read or write directly from main memory, with the traversal cache occupying a dedicated section of main memory. In such tightly coupled systems, the performance overhead involved with copying traversals to the FPGA's memories would be reduced, possibly improving speedup for applications with shorter or less frequently repeated traversals.

3.2. Hardware/Software Communication. Communication between the microprocessor and FPGA occurs through the signals shown on the left side of the controller in Figure 2. For simplicity, we do not show signals or control logic for moving data between the microprocessor and local FPGA memories (which could also be external to the FPGA). Other application-specific signals used to set up the datapath, for example, selecting which calculation to perform during this traversal, are also not shown.

When the microprocessor reaches a kernel requiring traversal of a pointer-based data structure, software makes a determination about whether an up-to-date serialized version of the traversal already exists in the cache. This determination is left to software, since it may be arbitrarily complex and is frequently the result of modifications to the data structure, which is also likely to be a complicated operation and, therefore, best handled in software.

If the cache does not contain the current traversal, representing a cache miss, the microprocessor transfers a serialized version of the traversal to the cache (mediating controller not shown), placed at an address in the cache chosen by software given by *trav_addr*. In the process of loading the cache, the accelerator's controller is notified to expect the valid traversal elements to begin appearing on the buffered processor write bus *μp_data*, and the controller is primed by asserting *start*. Once the first elements appear on *μp_data*, the controller manages the address generators and datapath, stalling the datapaths as necessary for new elements to arrive from the microprocessor while writing elements into the cache. In the unlikely event that the transfer rate exceeds the datapath's rate of consumption, the *μp_data* FIFO will eventually overflow. Upon overflow, the *μp_data* path is disabled after flushing the FIFO, and the controller begins sourcing data from the traversal cache memory instead, beginning with the element that overflowed in the FIFO (address known to the controller by observing the FIFO's overflow signal).

In the case of a cache hit, software tells the controller to begin processing the traversal from address *trav_addr* and asserts *start*. Elements of the traversal are simply read out of the cache in order, starting at address *trav_addr* and ending at address *trav_addr* + *trav_len*.

Once all the elements in the traversal have been processed through the datapath, the accelerator is complete and notifies the controller by asserting *ready*, after which the processor can safely issue new work for the accelerator. Our implementation also provides a signal to the microprocessor with the results memory's current fill state (not shown), which can allow software to overlap additional processing of the results with lengthy calculations on the accelerator in some applications.

The framework currently implements all control and synchronization signals using memory mapped registers inside the FPGA. Communication currently occurs over a PCI-X interface. Note that the framework is independent of the communication architecture, potentially supporting any underlying architecture that can implement the discussed control signals.

3.3. Software Library. To utilize the traversal cache, the accelerator requires assistance from a software library running on the microprocessor. The library is responsible for specifying when a traversal occurs, detecting invalidations of a traversal, and passing data to the accelerator when the traversal cache is empty or needs different or updated traversals. We currently implement this functionality using a library of wrapper functions around standard data structures. This process could also potentially be automated as part of a high-level synthesis and hardware/software partitioning tool, which could theoretically allow a user of the framework to utilize any library.

To specify a traversal to be processed by the accelerator, the wrapper functions first check if a valid serialization already exists in the traversal cache, only refetching and serializing traversal elements if the traversal is not already in the cache. The library uses a different wrapper function for each type of traversal, such as an in-order tree traversal, a depth-first search of a graph, etc. The most challenging task required by the software is to detect traversal invalidations. Currently, the framework invalidates traversals when elements in active traversals (traversals whose serializations are currently in the traversal cache) are changed. This often requires that the data structures be augmented with additional data identifying membership in currently active traversals. Detecting changes to the data structure is a challenging problem in general due to aliasing issues that may exist in the code. To avoid these issues, the framework requires that any changes made to the data structure be made through the use of the wrapper functions. This requirement guarantees that the traversal cache will be invalidated for any modification to the data structure. Furthermore, this requirement does not restrict the use of aliases outside the wrappers because those aliases cannot modify the structure.

Since multiple traversals might be kept active (in the traversal cache) at the same time, in the hope that they will be reused later, the libraries also do bookkeeping to track which traversals are in the cache and where they are located. This data is also required for placing traversals when being added to the cache and for carefully evicting individual traversals as necessary. This cache-management functionality is needed by all libraries and can be implemented in a way that allows sharing between libraries for different data structures. This is currently implemented as a mapping from library-defined keys (usually incorporating order and other data that uniquely generates the corresponding traversal) to the base address and extent of that traversal in the cache.

The penalty for evicting the wrong traversal (equal to the time required to generate and add a traversal) is high compared to traditional caches, and the total number of active traversals tends to be low, since accelerators are more effective on longer traversals. This suggests that otherwise expensive eviction strategies might be useful for traversal caches. Direct management of the cache by libraries also allows for the use of strategies based on an individual algorithm's dynamics. Traversals also generally vary in length and must be allocated in the cache contiguously, complicating placement and leading to fragmentation issues. Here too, the cost of a poor policy outweighs the relatively small cost associated with even complex strategies like heap compaction. In this paper, we evaluate different possibilities abstractly by using an invalidation rate (IR) which includes the effects of any such strategies. Future work will address these issues in more depth.

3.4. Limitations. The main limitation of the general traversal cache framework is that not all applications using pointer-based data structures are amenable to speedup. To achieve speedup typically requires that an application have frequently repeated traversals. However, as shown in Section 5, for some applications, speedups are possible even with relatively few repetitions. In fact, for some computation-intensive applications, speedup can be obtained even if the traversal cache is invalidated for every traversal due to the speedup provided by the accelerator's deeply pipelined datapaths. Furthermore, as discussed in Section 4, the framework can be extended for some applications to support traversal of similar but nonidentical traversals without suffering the penalty incurred by invalidations.

Another limitation is that the traversal cache must be manually created and a specialized software library must be used, or existing code modified for new applications. Ideally, a hardware/software partitioning tool could partition the application automatically, high-level synthesis could determine the appropriate size and amount of traversal caches, and also modify the software source code appropriately for use with any data structure library. These issues are outside the scope of this paper, but we plan to introduce synthesis techniques for traversal caches as part of future work.

In situations where the local FPGA memory is too small to store an entire traversal, software could load only the first part of the traversal, paging in later parts as needed by the accelerator. Though this would impact performance, depending on how fast the accelerator consumes data out of

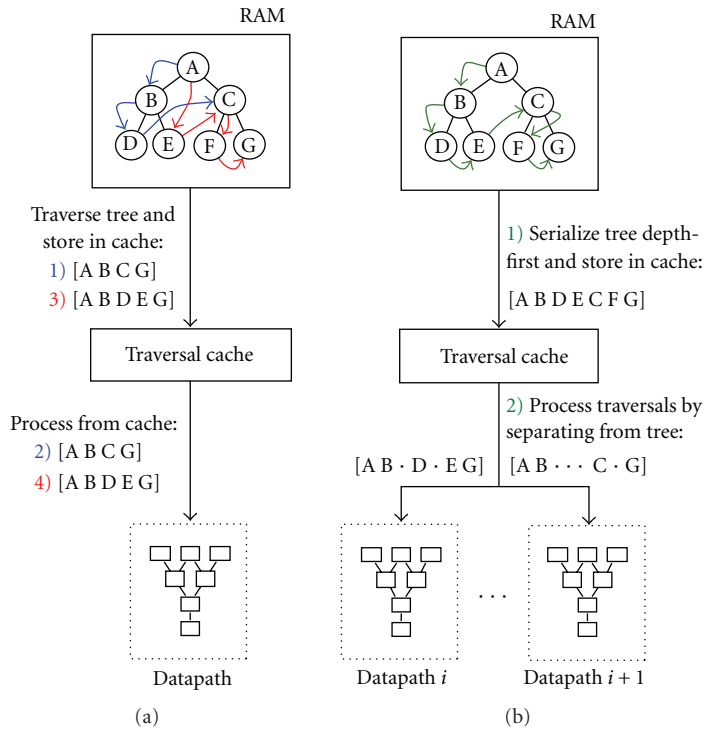

FIGURE 3: (a) Despite large amounts of similarity, nonrepeated traversals result in a 100% miss rate using the general framework, transferring a new traversal each time. (b) Extensions to the framework exploit the similarity between traversals by storing the entire tree in the cache and separating out individual traversals while streaming, also enabling datapath replication. (Elements excluded from separated traversals are shown as $(\cdot)$.)

the cache, at least some of the time required for paging could be hidden by consuming data in stream, as is done when the cache is populated with a new traversal. For example, in Section 5.1, we present a speedup of 15.5x for a linear search benchmark (*search,* IR = 20). Reducing the size of the traversal cache so that only one half of the traversal fits at a time and paging in the second half reduces the speedup to 8x without any effort to overlap computation paging and computation.

4. Exploiting Traversal Similarity: A Case Study on Barnes-Hut

The previous section discussed a generalized traversal caching framework that improves memory bandwidth for applications with repeated traversals. Unfortunately, many applications do not have this characteristic, limiting the achievable speedup. However, some important applications that lack sufficient repeated traversals have or can be made to have large amounts of similarity between traversals over time, including many applications that involve searching trees. In this section, we discuss modifications to the traversal cache framework to exploit this similarity in order to improve memory bandwidth for a broader range of applications.

Unlike the general framework in the previous section, the extended version of the traversal cache framework relies on the cooperation of hardware in generating traversals for processing in the accelerator's datapaths. This tighter integration of hardware and software provides new opportunities for optimization, but it has the effect of requiring application-specific control logic in the accelerator, which increases the complexity of accelerator design and hardware/software partitioning.

We evaluate these extensions for a type of Barnes-Hut n-body simulation application, summarized in Section 4.1. Barnes-Hut is a motivational example for the traversal cache framework because for n-body simulations using FPGAs, Barnes-Hut is typically avoided due to irregular memory access patterns resulting from its use of a quad- or octtree. FPGA implementations instead usually use a straightforward $O(n^2)$ implementation that does not require a tree data structure. Although this straightforward implementation enables highly parallel FPGA execution, the reduced $O(n \log n)$ complexity of Barnes-Hut often negates any performance advantage of the FPGA accelerated implementation. One key result of the traversal cache framework is that it can enable highly parallel FPGA implementations of the $O(n \log n)$ versions of Barnes-Hut as well as other tree-based algorithms.

4.1. Overview. Figure 3 compares the general traversal cache framework with the extensions for exploiting similarity. In the general framework shown in Figure 3(a), two different (but highly similar) traversals of the pointer-based data structure would have required two invalidations, causing

each new traversal to be serialized in software, stored in the traversal cache, and then streamed to the datapath in the FPGA. The extended framework exploits the similarity between the traversals by combining the multiple traversals into a data structure (in this case the entire tree serialized in depth-first order) that is stored in the traversal cache as shown in Figure 3(b). The FPGA can then stream the data from multiple traversals into datapaths by using additional logic, discussed in more detail in Section 4.4, to separate the elements needed by each traversal into the corresponding datapath. Since multiple traversals can be separated out from the same stream, the figure shows how these extensions further improve performance by enabling the two traversals to be processed in parallel in replicated datapaths.

In general, the improvement in memory bandwidth and the corresponding increase in useful datapath replication depend on the similarity between traversals. For circuits with n replicated datapaths performing n parallel traversals, the memory bandwidth provided by the framework becomes close to n times higher than the physical memory as the similarity between the traversals approaches 100%. Therefore, the framework can enable near-linear performance increases for linear increases in area when applications have highly similar traversals—a situation that overcomes the common memory bandwidth bottleneck of many FPGA applications [4, 23].

4.2. Barnes-Hut N-Body Simulation. N-body simulation is a common scientific computing problem that simulates the movement of n bodies under the influence of physical forces (e.g., gravitational or electrostatic forces). N-body simulation is used in a key step for a number of scientific applications including molecular dynamics, cosmological simulations, and problems in statistical learning.

A straightforward algorithm for an n-body gravity simulation consists of computing the forces exerted on each body by all the other bodies, which has a time complexity of $O(n^2)$. The Barnes-Hut algorithm [8] can reduce this complexity by treating groups of distant bodies as a single body centered at the corresponding center of mass. The algorithm creates this approximated solution by recursively subdividing space into a quadtree or octtree, with internal nodes representing an average of its leaf bodies (e.g., center of mass or electric multipoles). For concentrated collections of distant objects, set by a threshold ratio *theta*, the force is computed due to these average nodes instead of the individual bodies. Depending on the value chosen for *theta*, Barnes-Hut has a complexity ranging from $O(n \log n)$ to $O(n^2)$, with $O(n \log n)$ versions for *theta* < 0.5 being common in practice [8].

Pseudocode for the Barnes-Hut n-body algorithm is shown in Algorithm 1. The input is the number of time steps to simulate in addition to the set of bodies, where each body is typically represented by a mass, position, and velocity. At the beginning of each time step, Barnes-Hut creates a new quad/octtree based on the current position of the bodies. Then, for each body, the algorithm traverses the tree to calculate the force on that body, using the threshold *theta* to determine when to treat distant body clusters as a single body. The algorithm then updates the state (position, velocity, etc.) of the body based on this force. The algorithm then returns to the first loop, builds a new tree for the new body positions, and repeats.

```
def BarnesHut (TimeSteps, Bodies, theta):
    for # of TimeSteps:
        Tree = BuildTree (Bodies)
        for each body b_i in Bodies:
            force = CalculateForce (b_i, Tree, theta)
            b_i = UpdateState (force, b_i)
        Bodies = (b_0, b_i,. . ., b_N)
    return Bodies
```

Algorithm 1: Pseudocode of software Barnes-Hut implementation.

```
def BarnesHut_TC_SW (TimeSteps, Bodies):
    Tree = BuildTree (Bodies)
    for # of TimeSteps:
        Tree_Serial = SerializeTree (Tree)
        Bodies_Serial = SerializeOrderedBodies (Tree)
        PopulateCache (Bodies_Serial, Tree_Serial)
        ClearTree (Tree)
        StartFPGA ()
        while FPGABusy ():
            if FPGAHasData ():
                PartialBuildTree (Tree, ReadFPGAResults())
    return GetBodies (Tree)
```

Algorithm 2: Pseudocode of the software portion of the (similarity-based) traversal cache implementation of Barnes-Hut.

4.3. Software Extensions. The software for the extended framework has several responsibilities in addition to those required by the general framework. Pseudocode is shown in Algorithm 2. The basic structure of the code is similar to the Barnes-Hut code, with code that creates a new tree structure based on the body positions at each time step. This is done once initially for the initial positions of all bodies and is done during each time step in parallel with the accelerator's calculations by adding the updated bodies as they are made available by the accelerator.

After constructing the tree, the software serializes the entire tree in the order used by the Barnes-Hut traversal (i.e., preorder), while including address offset information necessary for the accelerator to execute skips that may occur due to grouping of distant bodies (i.e., *theta* threshold comparisons). These skips are processed by the FPGA, but because we cannot statically determine when the skips occur, the serialized tree representation must include all nodes of the tree and information at each node about how to skip over the node's subtree. After creating the serialized tree, the software transfers the serialized form to the traversal cache. This behavior is, in a sense, a generalization of the previous approach which provided the data for a single traversal at a time, in the order of just that traversal.

The software also provides the traversal cache with any data outside the data structure necessary to generate each traversal, which we refer to as *traversal inputs.* For Barnes-Hut, the traversal inputs are the individual bodies used by each datapath (the bodies subject to the net force being computed by that datapath). In general, the performance of the FPGA is maximized by the traversal cache when the traversal inputs are ordered to maximize the similarity between the traversals for adjacent traversal inputs. Our Barnes-Hut implementation determines this ordering (shown in the figure as *SerializedOrderedBodies*) by dividing the bodies into fixed-sized *clusters* of closely-located bodies, which are likely to traverse the tree in similar ways because of similar threshold comparisons at a distance. For Barnes-Hut, this clustering has little overhead because particles are already spatially grouped in the tree structure. Bodies within each cluster are then processed by the accelerator in parallel. Note that for some applications including Barnes-Hut, increasing the cluster size can still result in additional speedup even when further datapath replication is limited by area constraints, as discussed in the next section.

Once it has populated the traversal cache, software starts the accelerator, which processes the clusters one at a time, generating traversals and accumulating forces for the individual bodies. After completing each cluster, the accelerator adds results data to the results memory and asserts a *cluster_ready* signal, which is checked by software (shown in the figure as *FPGAHasData*). This allows software to construct the next version of the Barnes-Hut tree while the accelerator is still working. When all clusters have been processed by the accelerator, the *ready* signal is asserted and software increments the current time step, serializes the quadtree and clusters, and restarts the process.

4.4. FPGA Framework. The FPGA portion of the traversal cache framework is shown in Figure 4. The majority of the computation is handled by the datapaths, which are replicated p times up to the area of the target device, with each replication handling a separate traversal stream. Before work is begun on a cluster, the traversal inputs are read into registers (not shown) in the datapath logic, beginning from the address *clusters_addr*. The data structure is streamed out of the traversal cache, starting with the element at *ds_addr*, with elements appearing on the common data bus *tc_data*. The elements belonging to each individual traversal are identified from the common traversal cache stream by logic referred to as a *generator kernel*. The decisions made by these generator kernels are also used by a common controller called a *traversal generator* to steer further exploration of the data structure.

More precisely, the decision about whether a particular element from the cache is included in a given traversal (the traversal corresponding to a given traversal input) is determined by application-specific logic, referred to as a *generator kernel*, which is replicated once for each of the p parallel traversals. This logic is rarely the same between applications since the exploration of pointer data structures is almost always driven by computation unique to the application. For

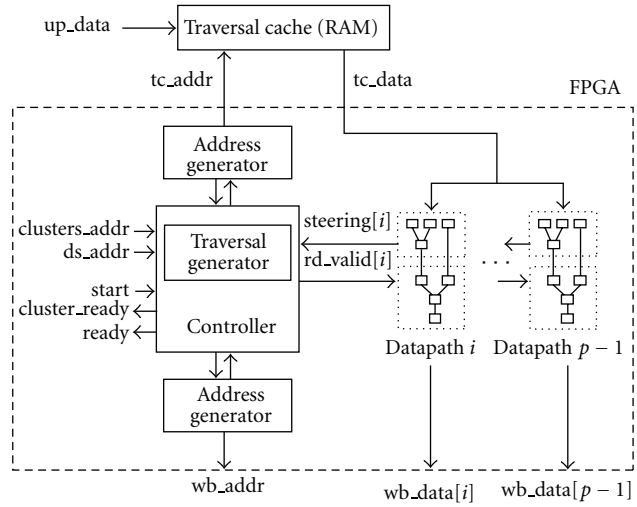

Figure 4: FPGA portion of the extended traversal cache framework with unrolled datapaths, each provided with separate traversals through selectively enabled *rd_valid[i]* signals. Element membership in each traversal is determined by application-specific logic, *generator kernels*, as part of each datapath (shown here as a dotted segment) and typically data structure-specific logic in the *traversal generator*, shown here as part of the controller.

example, although Barnes-Hut generally explores the tree in depth-first order, which elements are included in the depth-first traversal for a given body is determined by the *theta* threshold calculation for internal nodes. Thus, the Barnes-Hut generator kernel is this threshold calculation, which is dependent on the traversal's body (the corresponding *traversal input*) and the current element. In Barnes-Hut, as is often the case, the generator kernel logic produces many intermediate values needed for datapath computation and is itself rather computation intensive. This tight relationship between generator kernel and datapath is illustrated in Figure 4 by showing the generator kernel as an upstream section of each datapath.

Depending on the data structure being explored, a generator kernel may also decide *steering*, in addition to the element *membership* signal, describing how its traversal should proceed exploring the data structure. In Barnes-Hut, the only steering data is whether to skip over the current subtree, which is logically equivalent to the membership signal. However, membership and steering are collectively shown as a *steering* signal in Figure 4 for generality.

The *traversal generator* is application-specific control logic that explores the in-cache data structure (by controlling the read address generator) to stream all the elements required to process a cluster of traversals. The generator uses the steering data provided by the generator kernels to decide how to traverse the data structure minimally, skipping over any elements not required by any traversals in the cluster, while ensuring that even elements required by any single traversal are still read. In the case of Barnes-Hut (or any depth-first traversal) this condition is met by only skipping when all generator kernels agree to skip the current section of tree.

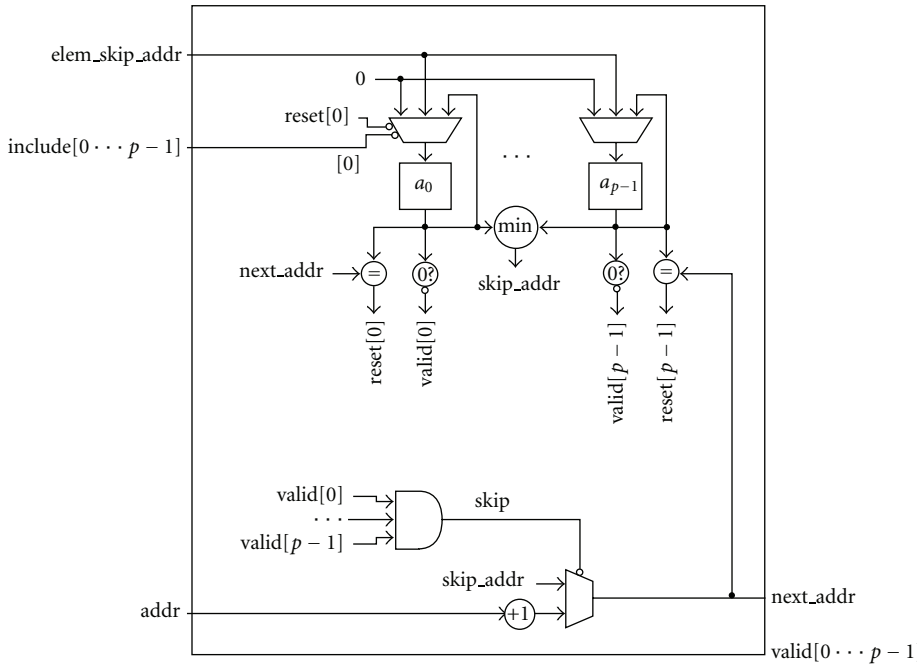

Figure 5: Simplified *traversal generator* logic for preorder tree traversal used to implement Barnes-Hut. The traversal generator guides exploration of the data structure for p simultaneous traversals, minimizing accesses by coordinating multiple datapaths by selectively asserting *valid[ ]* using membership data from each datapath's *generator kernel* on *include[ ]*.

Handling a skip is implemented by augmenting the in-cache representation of the data structure with additional data. For example, Barnes-Hut skips over subtrees when an internal node can be treated as a center of mass. Our implementation of Barnes-Hut implements these skips by including a child node count on each internal node in the octtree's serialized representation. Since the tree is serialized in preorder, skipping over the current subtree is handled by simply incrementing the current address by the current (head) node's child count multiplied by the size of a serialized node (a constant).

The traversal generator must also ensure that elements read for the benefit of only some traversals in the cluster are not delivered for traversals (generator kernels or datapaths) that have previously opted to steer around them. For example, in our Barnes-Hut implementation, if all but one datapath can skip a portion of the tree, the generator must still explore the section for the holdout while blocking that data from reaching the other datapaths.

Note that although the traversal generator is application-specific in general, a single generator can often be reused between multiple applications, since it only requires knowledge of the data structure and how to move through it (e.g., what types of skips are possible). This reuse is the primary reason for separating traversal generator logic from the generator kernels. Barnes-Hut, for example, performs a preorder traversal of an n-ary tree. However, besides the logic governing when to skip a subtree, which is implemented in the generator kernels, only the node size and n-ary size of the tree are specific to Barnes-Hut and can be made parameters of a generic preorder generator component.

The traversal generator implementation for Barnes-Hut is illustrated with simplifications in Figure 5. The generator takes as inputs *addr*, the address of the current element being streamed from the cache, *elem_skip_addr*, the address of the next element after the current subtree (part of the current element record as stored in the cache), and *include[ ]*, a vector of membership bits from the generator kernels shown in Figure 5. The generator outputs *next_addr*, the next address to read from the traversal cache, and *valid[ ]*, a vector of data valid bits for the datapaths used to separate the individual traversals.

Registers a_0 through a_{p-1} are used to track the progress of each traversal through the preorder serialized data structure. Each register starts at value 0, indicating that the corresponding traversal is not skipping the current node. When a generator kernel determines the current subtree should be skipped, the corresponding *include* signal is deasserted, storing *elem_skip_addr* (the address of the next potentially included element after the current subtree) in the register. Each register is *reset* to 0 on the cycle before its stored address (if any) is accessed in the cache. As long as the register is nonzero, the corresponding *valid* signal is deasserted, skipping over the current element for that traversal by preventing processing in the datapath. The skip signals for individual traversals are ANDed together to enable the generator to jump over sections of the data structure by conditionally setting *next_addr* to the minimum nonskipped element address stored in the registers. The tree of comparators and MUXs implementing this min operation is the primary source of overhead due to the traversal cache in our Barnes-Hut implementation, growing roughly

as $n \log n$ in area and $\log n$ in performance. In general, the overhead from a given traversal cache implementation is application dependent, depending heavily on the logic required by the traversal generator. However, since an efficient FPGA implementation is often not possible without caching for applications that would use the framework, including Barnes-Hut, this "overhead" is not really in conflict with accelerator speedup though it must still be minimized to optimize performance.

Sometimes generator kernels may be complex enough to require pipelining. In this case, the *include* signals in Figure 5 are delayed to the generator logic by as long as the generator kernel's latency. In order to avoid stalling excessively the generator must continue to fetch elements from the cache, predicting a skip will not occur. Since the generator kernels may continue to receive elements they will eventually steer around, they must either be stateless (i.e., no dependence on past data) or must be capable of reverting their state when it is determined that previously determined elements were invalid. This was not an issue for Barnes-Hut, since the *theta* threshold calculation that determines membership is a function of the current element only.

In the simplest case, the number of traversals handled by the framework in parallel is limited to p, the number of replicated datapath-generator kernel pairs, which is limited by the resources available on the FPGA. The number of parallel traversals can be increased without increasing datapath replication by creating clustered traversal inputs that can be divided into s-sized subclusters. For each subcluster, the traversal generator computes traversals in parallel, then sequentially swaps in a new subcluster and repeats until the entire cluster is processed for the current serialized tree element. The amount of datapath replication p and the number of subclusters s are architectural parameters of the framework that can be varied for different devices and inputs. The total number of traversals considered simultaneously is then given by $p \times s$, which is equivalent to the cluster size. This approach requires additional logic not shown in Figures 4 or 5 for clarity. The performance of this type of clustering is evaluated in Section 5 for our implementation of Barnes-Hut.

4.5. Extensions for Other Applications. The previous sections discuss how the traversal cache framework exploits similarity between traversals for an n-body simulation. The framework can potentially be extended to other applications using the following methodology.

First, the software must be able to serialize the data structure used by all traversals. Serialization is straightforward if the same basic traversal order is used for all traversals. For example, Barnes-Hut used a preorder traversal of the tree, while occasionally skipping nodes. Therefore, we serialized the tree using a preorder traversal. In addition, the traversal generators in the FPGA must be able to identify when incoming data is part of the traversal for a given input. For example, the traversals for Barnes-Hut differed for every input, but the traversal generator could detect when there was overlap by applying the threshold comparison used for skipping over nodes. If an application uses completely different traversal orderings, the framework could potentially still achieve speedup, but only if the traversal generator in the FPGA can detect when overlap occurs.

After serializing the tree, the software must also be able to determine an ordering of traversal inputs that provides for good similarity. For Barnes-Hut, this ordering was based on clustering close particles. If it is not clear what order would produce high similarity, or if determining that order would have a large overhead, then the framework may not benefit the corresponding application.

4.6. Limitations. The maximum speedup using the similarity-based traversal cache is achieved for algorithms with independent traversals in the same order, for example depth first, across a data structure. In the case that an algorithm's traversals are dependent on previous traversals, the framework cannot process traversals in parallel ($p = s = 1$). For algorithms whose traversals occur in different orders across a data structure resulting in no data reuse, the framework is equivalent to the general framework and requires software intervention between processing traversals.

The potential for speedup from the framework increases with the similarity between successive traversals. Because not all applications exhibit high similarity between traversals, the framework does not always improve performance. Low similarity between traversals results in disagreement among generator kernels about which regions of the data structure can be ignored. Since the generator must satisfy the needs of all the kernels to maintain correctness, it must include regions needed by any single kernel, stalling any other kernels (and datapaths) that do not need those elements and reducing parallelism. Since element accesses are grouped for traversals computed in parallel, it can also be shown that the total number of accesses is minimized when similarity is maximized. However, because efforts to maximize the similarity between traversals would also benefit pure software implementations due to caches on main memory, implementations using the framework might benefit from existing work along those lines.

In this paper, we assume the data structure fits in the traversal cache, which we currently store in a memory on the FPGA board. In situations where the size of memory prevents storing the entire data structure in the cache, software could load only part of the data structure, paging other parts into the cache as they are reached by the accelerator. However, the effect on performance would depend heavily on the algorithm being implemented due to the overhead of loading potentially many unused elements included in the data structure's cached representation. If the number of elements within each traversal is comparable to the size of the data structure itself (e.g., instances of Barnes-Hut with low *theta*), paging in and out the entire data structure might provide acceptable performance. Otherwise, application-specific approaches would probably be necessary for acceptable performance. One possibility is prefiltering the data structure to remove elements that it is known will not be required by the current window of traversals.

However, implementations not using a traversal cache would also be limited by the size of available memory, and possibly sooner since the ordering assumed by the framework allows the data structure's serialized form to often be smaller than the original form in memory. More detailed analysis of the effects of cache size on the performance of implementations using the framework is left as future work.

5. Experiments

5.1. General Framework

5.1.1. Experimental Setup. To evaluate the traversal caches, we implemented the framework on a system combining a 3.2 GHz Xeon with a Nallatech H101-PCIXM [24], which consists of a Xilinx Virtex 4 LX100 FPGA with 4 SRAM banks and 1 SDRAM. We mapped the framework onto this target architecture in the following way. All software uses the 3.2 GHz Xeon, and all custom circuits execute on the FPGA. All communication between the microprocessor and FPGA is sent over a PCI-X bus. Control and synchronization signals are read from and written to memory mapped registers inside of the LX100 using memory map nodes provided by Nallatech. For each example, the traversal cache is implemented in either an SRAM bank or on the SDRAM depending on the required size. To investigate effects of invalidation rate on performance, we do not base the invalidation rate on a specific input stream, and instead manually test different invalidation rates. This approach allows us to test different input possibilities ranging from the worst case to the best case. For each example, we test invalidation rates of 1 (invalidate every traversal), 5 (invalidate every 5 traversals), 10, 20, 40, and 80.

We evaluated the framework using the following benchmarks, which we developed. For each benchmark, we describe the pointer-based data structure and justify tested invalidation rates. For each software baseline, GCC's prefetch builtin was used as soon as the next node's address was available to begin loading the next node's data as early as possible.

Search scans a linked list of 1 million 16-bit integers and determines the number of occurrences of a specified value. The FPGA implementation performs 16, 16-bit comparisons and 15 additions every cycle. The invalidation rate for *search* is likely to be low for any application that searches a data structure multiple times without changing, such as a database application.

Audio performs convolution of an input signal consisting of 16-bit audio samples with a 64 sample impulse response. The data structure used by audio is a linked list of audio streams, which may likely occur in a digital audio workstation or a video game. Repeated traversals are likely since the actual audio stored in these applications does not change frequently. The circuit implementation performs 64 multiplications and 63 additions every cycle.

Graphics performs 3-dimensional vertex transformations by multiplying 4×4 transformation matrices with 4×1 vertex matrices. *Graphics* uses a list of vertex arrays, where each node of the list represents an object to be rendered. The implementation performs 16 multiplications and 28 additions every cycle. All operations are floating point.

Hardware/software communication and bookkeeping common to all the benchmarks was implemented in standalone C and VHDL libraries. These libraries required under a week to develop, test, and refine, using a number of components provided by Nallatech including a PCI-X interface, SRAM/SDRAM controllers, and memory map interfaces. The software implementation of each benchmark was completed in one day on average. The application-specific data structure serialization, invalidation bookkeeping, and VHDL datapaths collectively took on average between 1 and 2 additional days for each case study to develop and optimize.

We executed each example at the maximum possible clock frequency obtained after placement and routing using Xilinx ISE 9.2, which ranged from 115 MHz for *graphics* to 135 MHz for *search*. For all experiments, we compare traversal cache performance to software running on a 3.2 GHz Xeon. We compiled each benchmark using GCC 3.4.6 with –O3 optimizations to ensure that the baseline in the comparisons was as fast as possible.

5.1.2. Speedup Compared to Pointer-Based Software. This section presents performance advantages of the general traversal cache framework (not handling similarity) compared to software running on a 3.2 GHz Xeon. Figure 6(a) shows speedup in execution time obtained by the traversal cache compared with the original pointer-based software. The speedup is given for a number of invalidation rates (abbreviated IR). Comparisons are not shown for an FPGA implementation without the traversal cache as in most cases the cache less implementation did not provide a speedup over software.

Each benchmark was also rewritten to use an analogous approach to the traversal cache in software by maintaining traversed elements in an array "cache", enabling sequential access. The speedup of this approach is shown in Figure 6(a) as IR (sw). In the interest of space, we only show these results for the lowest and highest invalidation rates. Finally, the benchmarks were rewritten to use arrays natively, shown as array (sw) in the figure.

For the *search* example, only the highest invalidation rate (IR1) was slower than the pointer-based software. All other invalidation rates achieved large speedup compared to pointer-based software, reaching as high as 29x for IR80. Array-based software performance was better than the traversal cache framework for all invalidation rates under 40. For the IR80 case, the traversal cache was 1.3x faster than the software array implementation.

For *audio*, all invalidation rates, including IR1, were faster than both the pointer-based software implementation and the array implementation. Speedup ranged from 6.2x to 8.2x. The reason for the increased speedup at higher invalidation rates was because *audio* performed more computation for each piece of data, minimizing the effects of transferring the data to the FPGA.

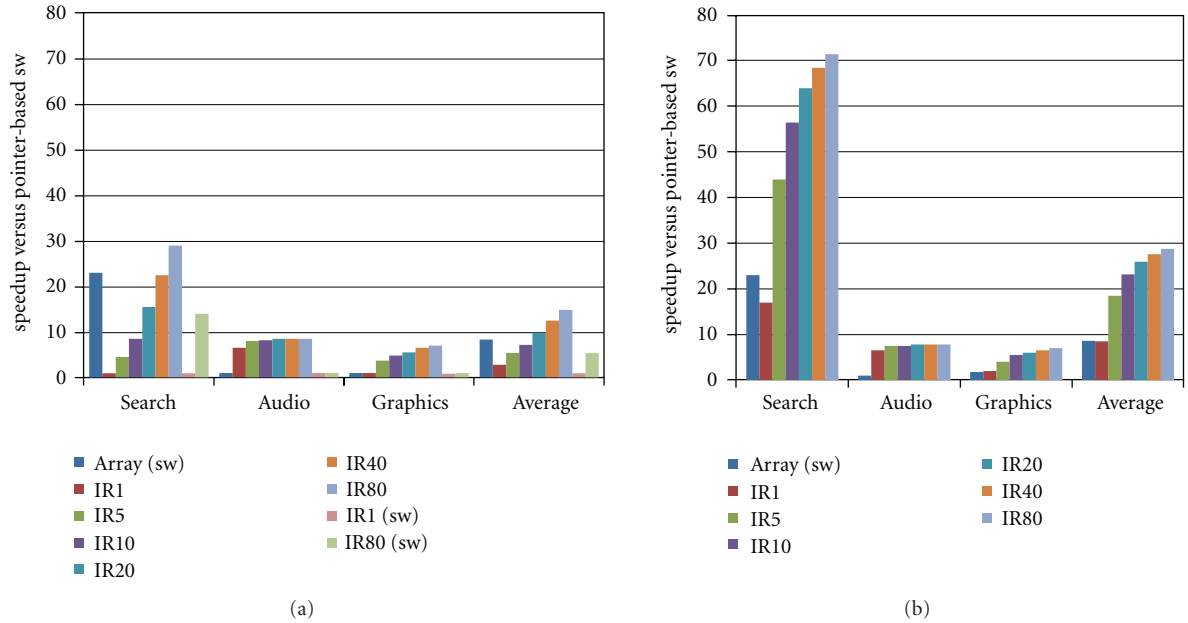

Figure 6: (a) Speedups versus the original pointer-based software implementation obtained for: array-based software (array) and an accelerator using the traversal cache framework for invalidation rates ranging from 1 to 80 (IR1-80). The results show that for some algorithms even high invalidation rates can achieve large speedup. (b) The same comparisons against pointer-based software after recoding the traversal cache implementations to use arrays. A similar speedup for *audio* and *graphics* shows that traversal caches can sometimes nearly completely hide the overhead of pointer-based structures.

Graphics achieved similar results, always outperforming the pointer and array software implementations with speedup ranging from 1.2x to 6.7x.

5.1.3. Speedup after Recoding. To determine how much improvement could be obtained by using arrays within the traversal cache framework instead of pointer-based data structures, in this section, we report the speedups assuming a designer were to recode the benchmarks to use arrays.

Figure 6(b) illustrates the speedup, again compared to pointer-based software, after recoding. For *search*, speedup more than doubled, reaching 70x for IR80. The increased speedup resulted from the significantly slower pointer-based software, which was more than 20x slower than the array-based software. We believe this performance difference is due to page faults caused by the large list size. The other examples achieved almost identical performances after being implemented with arrays. This surprising result implies that for certain examples, traversal caches make pointer-based code just as efficient on FPGAs as array-based code—a significant achievement considering the traditionally bad performance that has resulted from pointer-based structures.

5.2. Similarity Extensions: Barnes-Hut Case Study

5.2.1. Experimental Setup. We implemented Barnes-Hut for classical gravitational forces in two and three dimensions. The quad/octtree traversal logic was implemented as a generator supporting generic preorder traversal and search over an n-ary tree, as illustrated in Figure 5, with Barnes-Hut generator kernels skipping subtrees depending on a programmable *theta*. Similarity between traversals is increased by loosely ordering the bodies by spatial locality in the universe, which is available inexpensively as the order of the leaves in the fully constructed octtree. Our software implementations also use this ordering, which we found provides a speedup of up to 50% due to improved use of the processor's cache. Therefore, speedup estimates are slightly pessimistic compared to normal software execution. In the hardware implementation, we also construct the next time step's tree concurrently with running the accelerator by streaming outputs as they become available although our simple implementation of tree construction remains a factor limiting speedup in our full-application studies (Figures 7 and 8). Implementing parallel tree construction methods [25] would likely improve total application speedup.

To evaluate the similarity extensions for the traversal cache, we implemented fixed-point force calculation datapaths and preorder control logic, using generics for the number of traversals computed in parallel p and the number of p-sized subsets computed sequentially, s, in each pass. In the case that $s = 1$, the design is nearly identical to the block diagram of Figure 4. The framework was configured to use a single SDRAM input memory and an SRAM results buffer. Cycle counts were extracted from an HDL test bench that used a memory model of the SDRAM available on the Nallatech H101-PCIXM, accessed through Nallatech's SDRAM controller. Timing was measured and verified using a hardware implementation on the Nallatech H101-PCIXM FPGA accelerator card.

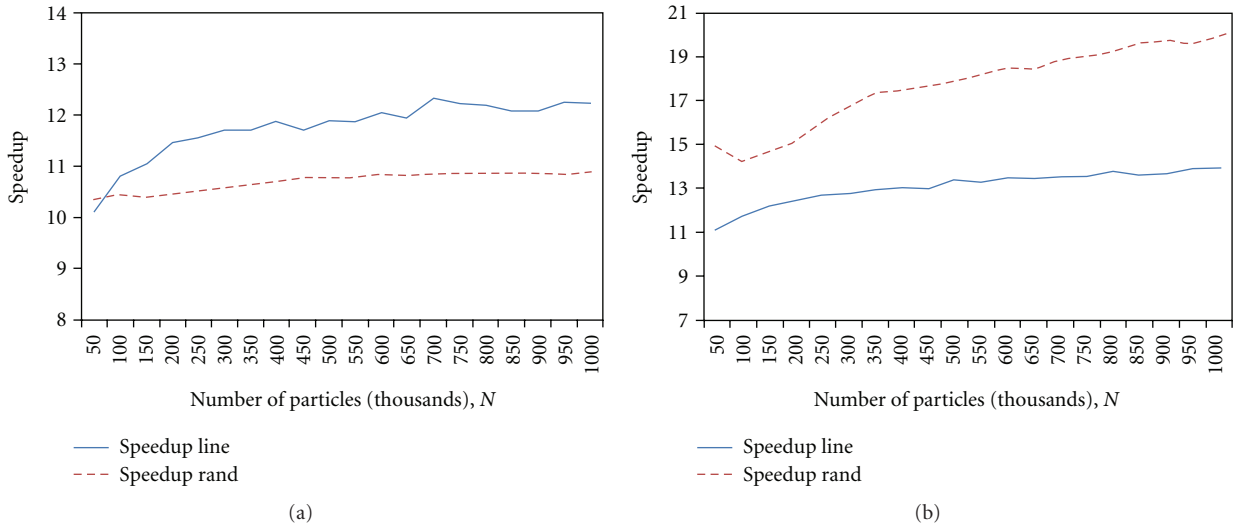

Figure 7: 2D (a) and 3D (b) Barnes-Hut application speedup achieved by the traversal cache framework on a Virtex 4 LX100 compared to a 3.2 GHz Xeon for various numbers of particles and representative distributions. The speedup is larger for the 3D algorithm, despite approximately equal similarity for each distribution, indicating that the increased computational intensity of the 3D algorithm takes better advantage of the FPGAs resources. *Theta* = 0.5.

The HDL simulation and timing data were combined to create a cycle-accurate C-based simulator of the hardware's behavior on arbitrary input data. Because synthesizing each test case would have been very time consuming, this C-based simulator enabled rapid exploration of the framework's design space, while also enabling consideration of configurations that would not fit on the Virtex 4 LX100 FPGA. The data provided for all configurations tested assumes the clock rate achieved for the largest design that would fit on the LX100 ($s = 1$, $p = 25$), which underestimates the performance of the smaller configurations tested.

Our software implementation of Barnes-Hut and the naïve n^2 algorithm took about a day to develop and test, with an additional day to implement optimizations. By comparison, the traversal cache implementation was developed and tested over about 2 weeks from start to finish, including all hardware and software components.

5.2.2. Performance Results. Our traversal cache Barnes-Hut implementation was compared to software running on a 3.2 GHz Xeon. The framework was tested with the maximum parallelism supported by the LX100 with no sequencing within a cluster ($s = 1$, $p = 25$). Sequencing is discussed in the next section. The time required to serialize the quad/octtree was included in the execution time for the hardware implementations and is reflected in the speedups reported. The serialization process required additional host memory equal to the size of the serialized data structure although this could obviously change significantly with other implementations (e.g., by generating and sending in chunks).

Because the traversal generator provides maximum bandwidth for high similarity traversals, and for Barnes-Hut the similarity between traversals is data dependent, we compared the system's performance on two data sets

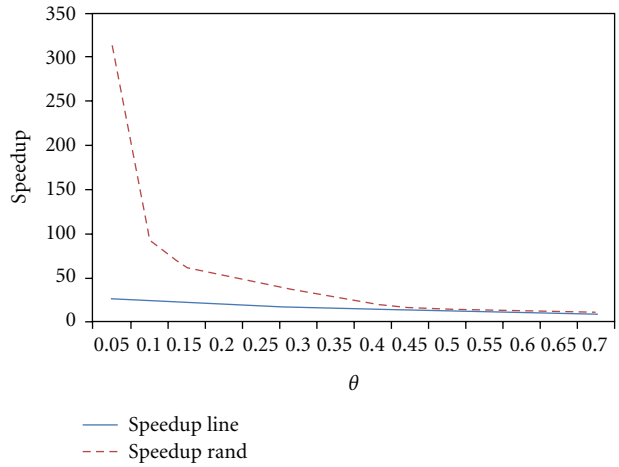

Figure 8: For lower values of theta (i.e., less approximation in Barnes-Hut), the traversal cache framework achieves larger application speedup. Higher values of *theta* than shown are not common in practice. $N = 100$ k.

representing extremes in similarity. The lowest similarity data set, *random*, has bodies distributed randomly in space, with on average 83% (for *theta* = 0.5) of elements in common between traversals for adjacent bodies. The highest similarity data set, *line*, has bodies evenly distributed in a line between opposite corners of space, with on average 90% of elements in common between subsequent traversals.

Figure 7 shows the application speedup of the framework relative to software for different problems sizes N and *theta* = 0.5, in two and three dimensions. Even though this value of *theta* is high for most practical n-body applications, it is used here to estimate worst-case performance, since higher values of *theta* result in shorter traversals. This is discussed in more

detail below in relation to the data in Figure 8, which shows speedup for different values of *theta*.

For the problem sizes tested, a speedup of between 10x and 12x was achieved for the 2D implementation, and between 11x and 21x for the 3D implementation, depending on the data set. The 3D simulations are more representative of practical n-body applications, which tend to be much more computation intensive than our gravitational force calculations and are usually physically three dimensional as well (e.g., molecular dynamics). The 2D simulations are still useful to illustrate the effectiveness of caching to enable speedup even for relatively low computation/memory ratios.

In both cases, speedup increased with problem size due to longer individual traversals resulting in more computational work being done before loading the next cluster, which better utilizes the FPGA's deep datapaths. The speedups for the 2D case were generally lower for the same reason, with less work being performed in the datapaths for each memory access. Execution times were universally lower for *line* due to better processor cache performance in software and improved parallelism in hardware due to better data reuse keeping more of the datapaths active.

Since real-world n-body problems tend to be more computationally complex than classical gravitational calculations, these results suggest even better speedups for real n-body applications. Although more involved calculations can entail larger pipelines and more limited unrolling due to area constraints, other techniques discussed in Section 3 can mitigate this effect.

However, the time required to construct the next tree structure confounds making some observations about the behavior of the traversal cache itself. The reduced effectiveness of increasing N is actually primarily due to the time required to construct the next tree, which quickly approaches the time spent doing actual computation. In the 3D case, application speedup is lower for the *line* data set despite a greater kernel speedup, because the lower total runtime in both hardware and software means tree construction figures more prominently in the total runtime. The cache's behavior is analyzed in more detail in Section 3 by focusing on kernel execution time.

The effect of the algorithm's precision parameter *theta* is shown in Figure 8 for a 3D simulation with 100 k bodies. The framework achieves higher application speedup for smaller values of *theta*, where the algorithm's complexity approaches $O(N^2)$. Traversals include more elements for these instances of the problem, resulting in a greater reduction in total memory accesses (improved bandwidth) due to the traversal cache. Including more elements per traversal also eventually results in fewer pipeline stalls due to fewer skips, contributing to a greater speedup. Note that comparing speedups of different *theta* values is different than comparing execution times. As *theta* increases, the complexity and execution time of the algorithm decreases, which contributes to the decreasing speedup due to the time required for the next tree construction. The value of *theta* chosen in practice depends on the particular application doing n-body, as it represents a tradeoff of simulation accuracy for execution time, but *theta*s less than 0.5 are common in real-world applications.

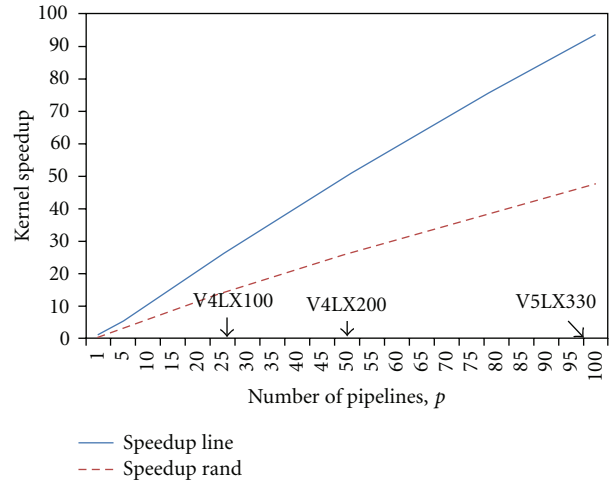

Figure 9: N-body kernel speedup obtained by the traversal cache framework for different amounts of datapath replication p. Note *that for high similarity, speedup increases almost linearly due to the framework exploiting similarity between traversals to eliminate the memory bandwidth bottleneck.* $N = 100$ k, *theta* = 0.5.

Previous FPGA implementations have focused primarily on n^2 implementations of n-body (*theta* = 0), which can usually only provide speedups for extremely precise simulations.

5.2.3. Effect of Framework Parameters. The traversal cache framework can be configured for an application through the parameters p and s, as discussed in Section 4.4. In this section, we explore the effect of these parameters for our traversal cache implementation of Barnes-Hut. Simulation data is provided in Figures 9 and 10 for a 3D system of 100 k bodies with *theta* = 0.5. In order to focus on the behavior of the framework itself, the data in these figures deals only with execution times and speedups for the force calculation and traversal kernels, referred to as the *kernel speedup*, which corresponds to the portion of Barnes-Hut implemented on the accelerator. The results for the Barnes-Hut application as a whole are discussed at the end of this section.

Figure 9 demonstrates that the kernel speedup provided by the framework increases with p by an amount determined by the similarity between traversals and grows nearly linearly for the high-similarity *line* case. The traversal cache is able to achieve this speedup *without increased demands on physical memory bandwidth by increasing effective memory bandwidth*, requiring only a single access for any single data structure element within a batch of traversals. Since p is limited by the size of the device, labels were added showing the amount of unrolling achievable on some common FPGAs.

Figure 10 shows the effect of s, the number of p-sized groups of traversals computed in sequence within a cluster, for Barnes-Hut force calculation implemented with 5 datapaths ($p = 5$). The simulation demonstrates that additional kernel speedup can be obtained after maximum unrolling by increasing s, up to a maximum, with reducing speedups after the maximum. This maximum speedup is also shown to be limited by the similarity between traversals,

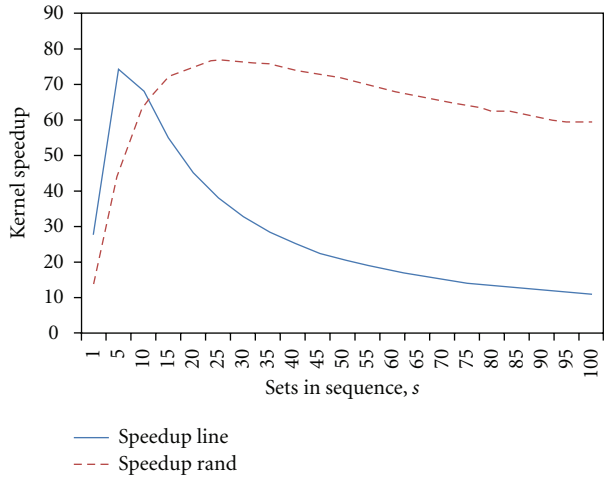

Figure 10: N-body kernel speedup obtained by the traversal cache framework for different amounts of datapath replication s. N = 100 k, *theta* = 0.5.

with higher similarities allowing a greater speedup for lower values of s. However, since the optimum amount of sequential processing s depends on the amount of similarity, which, as is the case for Barnes-Hut, is likely not constant or known *a priori*, implementations using sequencing would likely need to be adaptive, adjusting s according to an observed or predicted amount of similarity.

The speedup that can be achieved for a full application is limited by the execution time for the software parts of the application. For our implementation, application speedup was limited to 11x (*line*) to 21x (*random*) for N = 1 M and *theta* = 0.5. Using variable cluster sizes by optimizing s for the data set, our implementation could achieve a maximum application speedup of 14x (*line*) to 70x (*random*) for N = 100 k and *theta* = 0.5, being limited by tree construction. Methods for adaptively varying s are left as future work.

6. Conclusions

In this paper, we introduced a traversal cache framework that enables efficient FPGA execution of applications with irregular memory access patterns. The general framework dynamically serializes pointer-based data structure traversals and stores them in a memory local to the FPGA as a cache, enabling reuse of repeated traversals. Such serialization greatly improves effective memory bandwidth for repeated traversals. We also presented extensions that exploit similarity between nonidentical traversals that enable multiple traversals to be processed in parallel. For a Barnes-Hut n-body case study, the framework was shown to achieve speedups ranging from 11x to 21x compared to software on a 3.2 GHz Xeon processor using a Virtex4 LX100, with higher speedups of over 70x possible through runtime-adaptive techniques. More importantly, by exploiting similarity between traversals, the framework can nearly eliminate the common memory bandwidth bottleneck, leading to nearly linear increases in speedup with additional area.

Future work includes automating designer specified portions of the frameworks for integration with high-level synthesis and hardware/software partitioning tools. For the generalized traversal cache framework, additional work on cache eviction placement strategies should also be explored as a means to reduce invalidation rates and provide even larger speedups.

References

[1] A. DeHon, "Density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[2] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *Proceedings of the ACM/SIGDA 12th ACM International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, pp. 162–170, 2004.

[3] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh, "Computational density of fixed and reconfigurable multi-core devices for application acceleration," in *Proceedigns of Reconfigurable Systems Summer Institute 2008 (RSSI '08)*, 2008.

[4] 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 1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '07)*, pp. 1–10, ACM, New York, NY, USA, 2007.

[5] P. Diniz and J. Park, "Data search and reorganization using FPGAs: application to spatial pointer-based data structures," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 207–217, 2003.

[6] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*, pp. 249–256, 2004.

[7] P. Grun, N. Dutt, and A. Nicolau, "Access pattern based local memory customization for low power embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 778–784, IEEE Press, Piscataway, NJ, USA, 2001.

[8] A. Y. Grama, V. Kumar, and A. Sameh, "Scalable parallel formulations of the Barnes-Hut method for n-body simulations," in *Proceedings of the ACM/IEEE conference on Supercomputing*, pp. 439–448, ACM, New York, NY, USA, 1994.

[9] L. Semeria and G. De Micheli, "SpC: synthesis of pointers in C. Application of pointer analysis to the behavioral synthesis from C," in *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 340–346, 1998.

[10] L. Semeria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 743–756, 2001.

[11] L. Zhang, Z. Fang, M. Parker et al., "The impulse memory controller," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1117–1132, 2001.

[12] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proceedings of the Proceedings of the 35th Annual ACM/IEEE International Symposium on*

Microarchitecture, pp. 62–73, Computer Society Press, Los Alamitos, Calif, USA, 2002.

[13] N. Weinberg and D. Nagle, "Dynamic elimination of pointer expressions," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, p. 142, IEEE Computer Society, Washington, DC, USA, 1998.

[14] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behaviour," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 209–220, May 2002.

[15] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, no. 12, pp. 67–74, 2000.

[16] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, pp. 139–149, 1998.

[17] P. Grun, N. Dutt, and A. Nicolau, "Memory aware compilation through accurate timing extraction," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 316–321, ACM, New York, NY, USA, 2000.

[18] P. R. Panda, F. Catthoor, N. D. Dutt et al., "Data and memory optimization techniques for embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.

[19] N. Baradaran and P. C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 4, article no. 61, 2008.

[20] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, pp. 165–176, 2002.

[21] G. Stitt, G. Chaudhari, and J. Coole, "Traversal Caches: a first step towards FPGA acceleration of pointer-based data structures," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS '08)*, pp. 61–66, ACM, New York, NY, USA, October 2008.

[22] J. Coole, J. Wernsing, and G. Stitt, "A traversal cache framework for FPGA acceleration of pointer data structures: a case study on Barnes-Hut N-body simulation," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '09)*, pp. 143–148, December 2009.

[23] J. A. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors," in *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 145–154, 1999.

[24] Nallatech Inc. Nallatech PCIXM FPGA accelerator card, 2010, http://www.nallatech.com/~nallatech/index.php/PCI-Express-Cards/h101-pcixm.html.

[25] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes," *Computer Graphics Forum*, vol. 26, no. 3, pp. 395–404, 2007.